AN ANSWER SET PROGRAMMING BASED APPROACH TO REPRESENTING AND

QUERYING TEXTUAL KNOWLEDGE

by

Dhruva Rajan Pendharkar

APPROVED BY SUPERVISORY COMMITTEE:

_____

Dr. Gopal Gupta, Chair

_____

Dr. Vibhav Gogate

_____

Dr. Vincent Ng

Dedicated to my loving parents

AN ANSWER SET PROGRAMMING BASED APPROACH TO REPRESENTING AND

QUERYING TEXTUAL KNOWLEDGE

by

DHRUVA RAJAN PENDHARKAR, BE

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2018

ACKNOWLEDGMENTS

AN ANSWER SET PROGRAMMING BASED APPROACH TO REPRESENTING AND

QUERYING TEXTUAL KNOWLEDGE

Dhruva Rajan Pendharkar, M.S.
The University of Texas at Dallas, 2018

Supervising Professor: Dr. Gopal Gupta

Knowledge Representation and Reasoning (KR&R) is a field of Artificial Intelligence that deals with converting information into knowledge in a form that the computer can process. It applies concepts from the field of psychology, about how humans make rational decisions, to build formal rules that model the human cognitive processes. Using the generated knowledge bases, the computer is then able to solve complex tasks like question answering, summarization, automated reasoning, medical diagnosis and many more. Many of these complex tasks, mentioned above, require an understanding of natural language text. A vast amount of knowledge that we have today comes from books and is in the form of natural language text. Such knowledge is in an unstructured form and is not easily interpretable by computers.

An approach based on answer set programming (ASP) is proposed in this thesis for representing knowledge generated from natural language text. This knowledge is then used to perform reasoning with the help of advanced implementations of ASP such as s(ASP). ASP representation of techniques such as default reasoning, hierarchical knowledge organization, negation as failure, etc., are used to model common-sense reasoning methods required to accomplish this task. Automation of the question answering task has been used in this thesis to demonstrate the

effectiveness of our ASP-based KR&R techniques. The automated Q & A system developed as part of this thesis parses and converts natural language text to an ASP knowledge base. Users can pose questions in a natural language that are parsed and converted into ASP queries automatically. These queries are next solved against the knowledge base obtained from the natural language text augmented with related, auxiliary knowledge obtained from other resources such as WordNet. In contrast to approaches based on machine learning, our system answers questions based on *actually understanding the text*. This approach has been tested on the SQuAD dataset and the results are promising.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

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. For acting rationally, a machine must be able to obtain information and understand it. Automated reasoning and representation of information are important fields that lie at the intersection of computer science, formal logic, and philosophy.

Classical logic based approaches have traditionally been used to build automated reasoning systems. While there have been many successes, these approaches have not lead to systems that can be called truly intelligent. The main reason is that such systems do not deliver desired results, we believe, is due to the fact that classical logic is monotonic, suffers from incompleteness (*a la* Gödel), and its decision procedures are undecidable, in general. Classical logic thus is unable to effectively model human-like, common-sense reasoning. One could argue that developing a system that can perform common-sense reasoning and thus can emulate human thinking is the ultimate goal of AI. Arguably, humans do not use classical logic in their day to day reasoning tasks. Humans considerably simplify their burden of reasoning by using techniques such as defaults, exceptions, and preference patterns. Default rules are generic rules that can be applied to concepts. Such rules can sometimes have exceptions. As an example, we can have a default rule

that says, "*All plants are green*". This is true in most cases as '*Chlorophyll*' is the prominent green pigment in most plants that makes them green. But there are some plants, that contain high quantities of red pigments or absence of chlorophyll, that are not green. Such instances of plants form exceptions to the default rule. Other important features of human reasoning are non-monotonicity, which states that humans can revise their conclusions in light of newer information, and the ability to deal with incomplete information. Humans can easily make decisions or come to conclusions in the absence of data. All these properties of human reasoning need to be modelled if we want to automate common-sense reasoning. It has been shown that common-sense reasoning can be realized via *negation as failure* [1] [2]. Answer set programming is one of the popular formalisms that exhibits non-monotonicity. Apart from non-monotonic reasoning, the ASP paradigm also includes both *classical negation* as well as *negation as failure* [1]. ASP is a well-developed paradigm and has been applied to problems of planning, constraint satisfaction and optimizations. There are comprehensive, well known implementations like CLASP [3] and DLV [4] that have been carried out for ASP. Thus, ASP is well suited for modelling common-sense reasoning patterns.

Many of the fields of knowledge lack structured resources. Most of the resources available today are in the form of unstructured data, either in the form of written documents, or information present in the form of articles and paragraphs on websites like Wikipedia. Many tasks that are being automated would benefit from the availability of this information in the form of structured data. With the help of formalisms such as the event calculus [5] or the situation calculus [8], ASP could prove to be a helpful paradigm to represent textual knowledge and reason with it.

In this thesis we propose a system to automatically convert textual knowledge into ASP programs, and use it answer questions coded as ASP queries. The main idea is to automatically translate text into knowledge represented as an answer set program. Questions that we wish to ask of this knowledge can be converted to ASP queries as well and run against the answer set program obtained in the previous step. 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. Humans bring to bear the vast amount of common sense knowledge they have learned in the past while answering questions. Thus, if a human reads a story about Nikola Tesla's scientific career, and we ask the human if Tesla has a mother, then even if the passage does not talk about Tesla's family, every human will be able to answer that question in the affirmative. As stated earlier, when we humans read a passage, we automatically draw upon a large amount of common sense knowledge that we possess in understanding the passage, in answering questions related to the passage, or making new inferences from it. An automated common-sense reasoning system ought to do the same. In our work, we resort to resources such as WordNet that encapsulate some of the common-sense knowledge that humans use. Given a passage that our system has translated to an ASP program, our system will also extract all the concepts found in the passage, access their WordNet entries and convert all the information contained there into the ASP formalism as well. This additional knowledge will be then used in answering questions. Thus, our system will be able to answer correctly that Tesla has a mother, even though this information is not discussed in the passage.

Our system runs on the s(ASP) answer set programming system. s(ASP) [6] is a query-driven system that is crucial to the scalability of answer set programming. While the answer set

programming, paradigm has been around for nearly 20 years and its underlying formalism of stable model semantics [7] for more than 30 years, a major road block in the success of answer set programming systems is that they are not scalable. This is because traditional implementations of ASP are based on grounding answer set programs and using a SAT solver to find models of the resulting theory in order to compute answers. The requirement of grounding results in not only the programmer having to spend more effort in writing the program, it limits the size and type of the programs that can be executed, as grounding can result in an exponential blowup in the size of the resulting grounded program. The s(ASP) system allows answer set programs to be executed directly in a Prolog-like, query-driven manner and without being grounded first. The s(ASP) system also supports predicates. Thus an arbitrary answer set program can be executed in s(ASP). The s(ASP) system can also provide a justification for the query being processed, so it is possible, for example, to give the reasoning behind an answer to a question in an automated question-answering system. The availability of the s(ASP) system is crucial to our efforts in this project.

1.2 Related Work

Cyc is one of the oldest artificial intelligence project, that tries to collect information about basic concepts and about how the world works. This knowledge is presented in the form of a vast knowledge base or ontology that consists of implicit knowledge and rules about the world that we as humans call common sense knowledge. Such a project aimed at enabling AI applications to reason about their surroundings like a human and be able to handle failures gracefully. To build a project that could express common sense knowledge in the form of machine understandable code there needed to be a highly expressive representation language, developing a knowledge base using the language that would be used to infer from, developing a fast inference engine that would be

4

able to reason like humans and would be able to handle human like complexity. The ontology of Cyc was about 100,000 terms till 1994 but then as of 2017 contains around 1,500,000 terms. This includes around 500,000 collections, 50,000+ predicates and around a million well known entities. Cyc's language CycL made it efficient to represent common sense knowledge in the project and decided how this knowledge is represented in the project. To solve the problem of efficiently inferring from hundreds of millions of arguments Cyc used a community-of-agents architecture where it employed various types of reasoning agents call heuristic modules to solve the inference problem. Currently, Cyc uses more than 1000 heuristic models for inference. Much of the current work on Cyc includes knowledge engineering that includes representing facts about the world by hand and implementing inference techniques on that knowledge. Recently, Cycorp has planned to use Cyc's natural language processing to parse unstructured data like that presented by the documents on the internet to extract structured data.

The medical domain has always been one of the major focuses of expert knowledge-based systems. One of such applications based on chronic pulmonary disease or heart failure was modelled and built by Dr. Zhuo Chen under the guidance of Dr. Gopal Gupta. Its goal is to provide management of chronic illnesses like diabetes, cardiac and pulmonary diseases. One of the ways in which such diseases can be managed is by following a set of rules or guidelines laid out by experts in the domain. As most of these guidelines are highly complex and rely on many factors, physicians sometimes ignore them or fail to abide by them. This can adversely affect the patients' health. Most of the problems come from the fact that these guidelines apart from being highly complex are also large in number making the task of recommendation prone to errors. This problem was tackled in the work with the help of answer set programming. The approach included creation of

a set of knowledge pattern or rules that act as reasoning templates, which were used to model the guidelines provided by experts.

1.3 Contributions

The main contribution of this thesis is an effective and efficient method for converting textual data into knowledge represented as an answer set program. This includes developing a generic calculus that helps to represent knowledge (Chapter 6 dives deeper into how text can be converted into a knowledge). Another novelty is the use of knowledge sources such as WordNet to acquire common sense knowledge about terms found in the text. These resources are used to create a custom ontology for the problem at hand. The custom ontology created is dynamic in nature and only deals with the concepts that occur in the input paragraph. This helps in ensuring that our system shows good execution performance as this custom ontology can be either generated every time if space is the issue or else it could be built and augmented iteratively as more paragraphs are encountered in the text being processed. Yet another novelty is in showing how word sense disambiguation can be modeled using answer set programming rules (Chapter 6 also discusses how to apply a default preferential pattern for finding out the best sense of a given concept).

The thesis 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 by broadening the query by dropping constraints, thus increasing the possibility that the question will indeed be answered. As constraints are dropped from the query, the answer generated loses accuracy, and our system will give an estimate of the accuracy of the answer (one of certain, likely, possible, guess). This approach to handling question answering is yet another novelty of our system.

1.4 Structure of the Thesis

In this section, a layout of the remaining chapters is provided with a summary for each of them.

Chapter 2 mainly deals with Answer Set Programming. It describes the syntax of answer set programs and goes over the various ASP semantics used in the thesis. This chapter gives the background on answer set programming which is crucial for understanding the rest of the thesis

Chapter 3 introduces the architecture of the proposed system. It goes over the components of the system viz. Knowledge Generation, Query Generation, and the Common Resource Framework. It further discusses on how these components interact with each other to produce answer sets.

Chapter 4 discusses the Test-Driven Development approach that was used to build the system and create rules for knowledge generation. It goes through the various steps of a development cycle in TDD and explains its importance.

Chapter 5 provides information about the various Natural Language Resources used to support the system. This chapter covers the concepts required for understanding what these tools produce and how it can be used in later chapters.

Chapter 6 covers the various predicates that have been defined for the knowledge generation module. It explains their structure and what they represent. Apart from predicates generated from the passage it also talks about how information is extracted from WordNet by building an ontology.

Chapter 7 provides a method of automatic Query Generation from a natural language question. This chapter elaborates on how various queries can be generated automatically from a question and applies a confidence metric with each query that relates to the accuracy of the answer obtained from that query.

Chapter 8 describes how the above-mentioned system performs when applied to the task of Question Answering. Here, we introduce the SQuAD dataset by Stanford and how it is structured. This chapter states the results obtained from question answering and analyzes them.

Chapter 9 elaborates on a set of features or enhancements that can be made to improve the current system. It touches upon modelling temporal reasoning, cause effect reasoning and certain other common-sense reasoning patterns.

Finally, we draw some conclusions in Chapter 10. We summarize some of the salient points and review a few contributions of this thesis. This thesis also includes an Appendix at the end including all the references.

CHAPTER 2

ANSWER-SET PROGRAMMING

2.1 Overview

The system that has been designed uses an ASP-based approach to represent knowledge from natural language text. So, a basic understanding of answer-set programming is required to understand the remainder of the thesis. This chapter introduces the answer-set programming paradigm and further elaborates on some of the important definitions, concepts and patterns used in answer-set programming.

2.2 What is Answer-Set Programming (ASP)

Answer-Set Programming is a declarative problem-solving paradigm that uses both non-monotonic reasoning and logic programming. It is widely used in automatically solving problems relating to representation and reasoning tasks such as modeling reasoning agents, non-monotonic inferences, common-sense reasoning, modeling preferences and priorities and many more. An answer set program is a collection of statements that describe the objects of a domain and model relations between them. The semantics of an ASP Program defines a set of possible beliefs that an agent has associated with the program. This set of beliefs is called as an answer-set. The basic constituents of an ASP program are the rules, facts and constraints that describe the problem. Such a program is then passed onto an answer-set solver, which generates answer-sets to the program, that are used to obtain solutions to the problem.

2.3 Syntax

In this subsection, we introduce the syntax of an ASP program.

2.3.1 Atom

The most basic constituent of the ASP program is an atom. An atomic statement or an atom, is an expression of the form *p (t₁,…, tn)* where *p* is a predicate symbol of arity *n* and $t_1...t_n$ are n terms belonging to the predicate *p*. Here *n >= 0* and the terms t_i can be integers or strings of letters, numbers, or underscore that either begin with an underscore or a lower-case letter. If in an atomic statement *n = 0*, then the brackets are omitted. As an example, '*parent (mary, alice)*' and '*alice*' are both atoms, whereas '*parent (mary, girl(alice))*' is not an atom.

2.3.2 Literal

A literal is an atom of the form *p (t₁,…, tn)* or its negation *-p (t₁,…, tn)*. Here, *-p (t₁,…, tn)* is referred to as a negative literal. It means that *p (t₁,…, tn)* is false. An atom is called as a ground literal if every term t_i in the atom is ground. For example, '*parent (X, Y)*' is a literal whereas '*parent (mary, alice)*' is called as a ground literal.

2.3.3 Rule or Clause

An ASP Program consists of a collection of rules of the form

a. l_0

b. $l_i \leftarrow l_{i+1}, …, l_m$, not l_{m+1},…, not l_n

Here, the symbol 'not' is a logical connective and is called as a default negation or negation as failure. Its semantic is discussed later in the chapter. An ASP rule is divided into two parts viz. head and a body. A head is a literal on the left side of the rule and a body is a set of literals on the

right side of the rule. The head or the body in a rule can be empty. A rule with an empty head is called as a constraint whereas a rule with an empty body is called as a fact.

2.4 Semantics

Using the earlier mentioned syntax, we create an ASP program as a collection of rules, facts, and constraints. In this section, we shall discuss about the meaning of these rules and how they are interpreted while reasoning using these rules. The following are a few semantic patterns commonly used in answer-set programs.

2.4.1 Modelling Implication

As we saw earlier, every rule (excluding facts) in ASP has two parts separated by the consequence operator ":-". In such a rule the head of the rule is said to succeed only if every literal in the body of the rule succeeds. As an example, consider the rule

p :- q, r.

we can read this kind of rule as "if *q* and *r* succeed then *p* succeeds". Such kind of a pattern is commonly used in ASP programs to show implications.

2.4.2 Classical Negation

Classical Negation is a pattern in which we use negative literals, to show the fact that the literal under consideration has been proved to be false. As an example, consider the following rule.

-p(a) :- q(a)

The above rule states that if *q(a)* is shown to succeed then *p(a)* is false or *-p(a)* is true. Classical negation is one of the ways to represent negations in ASP programs.

2.4.3 Epistemic Disjunction

We model epistemic disjunctions in ASP when we need to model the semantics for the statement, "Either *p(a)* succeeds or *q(a)* succeeds". Epistemic disjunction is different from exclusive or, where both *p(a)* and *q(a)* might succeed at the same time. Thus, to model epistemic disjunction we can make use of even loops in the following manner.

p(a) :- not q(a).

q(a) :- not p(a).

If we solve the above ASP program using an answer-set solver we will get two answer sets {*p(a)*} and {*q(a)*}, i.e. either *p(a)* succeeds or *q(a)* succeeds.

2.4.4 Constraints

Constraints are applied in places where we know that certain rules are always false and should not be part of the answer-set. As an example, if we know that it is impossible for *p(a)* to succeed then we can model this constraint as follows:

:- p(a).

The above rule states that *p(a)* is always false. Here we see that a constraint limits the sets of beliefs that an agent has but does not help to derive new information.

2.4.5 Default Negation or Negation as Failure (NAF)

Default Negation, also called as Negation as Failure is used to make conclusions based on the absence of information. This type of negation is used to conclude about default rules and assume defaults to be true in case of absence of enough information. As an example, consider the following example where we state that if we are not able to prove that *q(a)* succeeds then *p(a)* succeeds.

p(a) :- not q(a).

So, in the above rule we assumed that *p(a)* has succeeded based on the absence of information about *q(a)*. NAF is an important tool to model defaults in ASP programs. Negation as Failure assumes closed-world assumption (CWA), in which we assume, what is not currently known to be true, as false.

2.5 Default Reasoning

Default Reasoning or Representing Defaults is one of the advantages of using ASP. The concept of closed-world assumption discussed earlier is an example of default reasoning where we default the value of the literal to fail in the absence of the literal in the answer set. Default reasoning is very useful in modelling human reasoning as we can draw conclusions even in the absence of information by defaulting to the default rule. Default reasoning thus plays an important role in common-sense reasoning and understanding. In case of ASP, a default d stated as "Normally elements of class C have property P" is represented as the following rule:

p(X) :- c(X),
    not ab(d(X)),
    not -p(X).

Here, *ab(d(X))* can be read as "X is abnormal with respect to the default assumption d" and *not-p(X)* can be read as "We cannot successfully prove that *p(X)* is false" or "*p(X)* may be true". Default reasoning uses two kinds of exceptions viz strong exceptions and weak exceptions. Weak exception makes the default inapplicable and stop the agent from making a default conclusion. For example, in the above-mentioned default rule we can apply a weak exception *e(X)* by adding the following rule to the program

ab(d(X)) :- not -e(X).

The exception states that X may not be applicable to d if *e(X)* may be true. Similarly, Strong Exceptions refute the defaults conclusion by allowing the agent to derive the opposite to be true. This can be demonstrated by adding the following rule to the program

-p(X) :- e(X).

The above rule states that *p(X)* is false if *e(X)* succeeds, which allows us to defeat d's conclusion that normally class C elements have the property P.

CHAPTER 3

SYSTEM ARCHITECTURE

3.1 Overview

This chapter mainly focuses on the various parts of the system and how they interact with each other. It also describes the various sub-components and modules used in generating the knowledge base and goes through the various steps required to answer queries with the help of the generated ASP program.

3.2 System Architecture

The system is composed of two main components or sub systems viz. the Knowledge Generation System and the Query Generation System. Both these systems function independent of each other. The architecture comprises of a common resource framework that is shared by both these systems. This chapter will describe all these components in detail in the rest of the chapter.

3.2.1 Components of the System

As illustrated in Figure 3.1, the Knowledge Generation System, the Query Generation System, and the Common Resource Framework are the three components of the architecture. The Common Resource Framework consists of Natural Language Processing tools such as Stanford Core NLP Tools, WordNet API as well as modules for preprocessing 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, Stanford Dependency Parser, and the Stanford NER Tagger to gain more information about the input text.
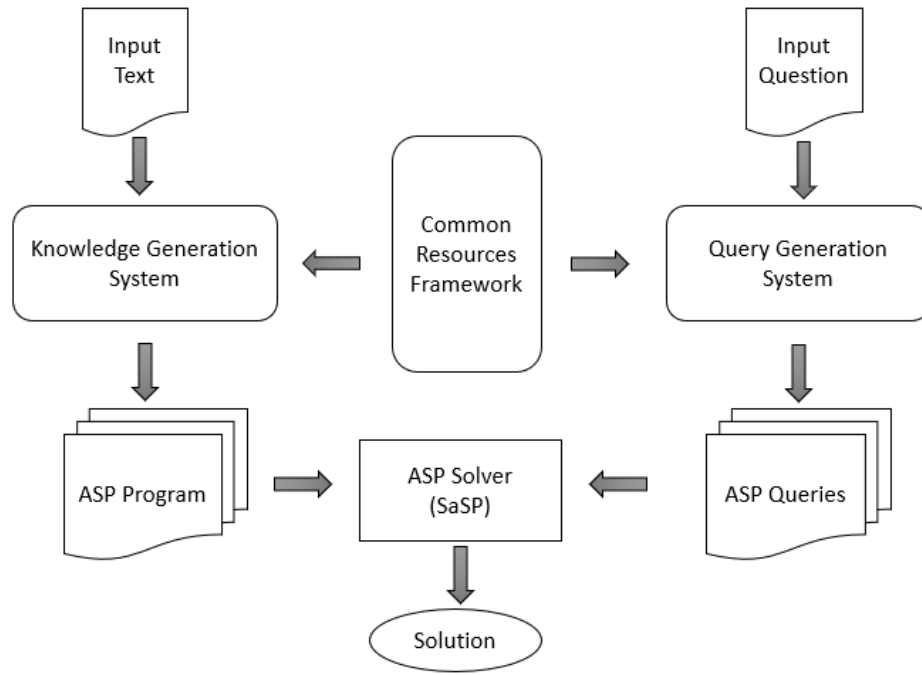
Figure 3.1 System Architecture

Apart from these resources it also taps into the vast information that is provided by WordNet and tries to extract information from the same. As currently there are very few digital resources about verbs in the NLP domain, this component provides a flexible way to add custom information about verbs that would be reusable in many scenarios. Thus, the Knowledge Generation System takes in the natural language passage as input and produces rules in the form of three chunks of information, which can be aggregated together to form an ASP program representing all the extractable knowledge from the source text.

To help answer questions posed in Natural Language, the Query Generation System is used to automatically generate a set of queries that can be used to find solutions from the answer-sets generated by the ASP program. To answer queries from the ASP program we need to provide both the queries as well as the ASP program to an Answer-Set Solver like s(ASP) or CLASP. The Query

Generation System generates multiple queries for a question and arranges them in the order of significance, keeping the more constraint queries before the less constraint ones. Hence, the kind of query that would lead to an answer is also a rough metric as to the quality of the answer. Now let's dive deep into the various components in the architecture and talk about its sub modules and their interactions.

3.3 Common Resources Framework

The Common Resources Framework consists of the following modules as illustrated in Figure 3.2.
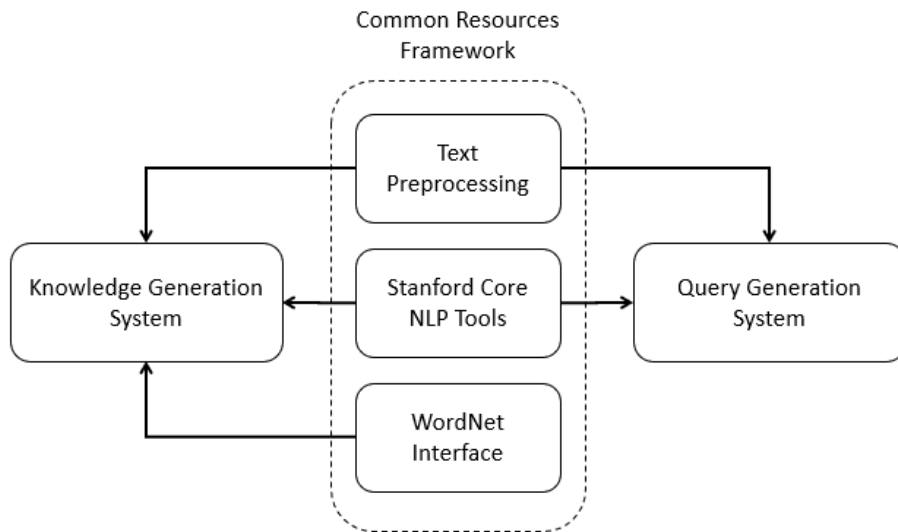
Figure 3.2 Common Resources Framework

3.3.1 Text Preprocessing Module

The style of writing in natural language texts changes based on the domain, author, title of the text and many other factors. Automating text processing, becomes a very hard task when we must consider all these different writing styles. Thus, for this system we assume certain properties about the incoming natural language text. The text pre-processing module is the first module that the input text passes through and it makes sure that the input text conforms to these assumptions. Some

of the assumptions that we make about the incoming text include concatenation of compound nouns and resolution of coreferences. National Aeronautics and Space Administration or NASA is an excellent example of a compound noun. In this example, we assume that the system detects and treats NASA to be a single concept (National_Aeronautics_and_Space_Administration) as opposed to separate words. Coreference resolution is the task of finding all expressions that refer to the same entity in a text. It plays an important role in higher level NLP tasks and so we assume that the coreferences in the incoming text have already been resolved. Many a times due to informal writing styles, humans miss certain words or assume certain words while reading and writing texts. Working with such informal style of English is hard, so we assume that the incoming text is written in formal English. As this system depends on many NLP tools for semantic resources, it is susceptible to any flaws in these tools. The preprocessing module tries to correct any mis-tagged entries in the text, due to ambiguous wording, with the help of relations given by higher level semantic tools.

3.3.2 Stanford NLP Core Tools

Stanford Core NLP Tools is a set of linguistic tools that help in analyzing and understanding natural language text. It consists of a lot of different sub tools that can be pipelined one after the other to analyze a piece of text. It provides solutions to NLP tasks like POS Tagging, Parsing, NER Tagging, Coreference resolution and many more that play a vital role in higher level NLP tasks like text understanding. This system uses the Stanford-Core-NLP version 3.9.1 on the Java Platform and makes extensive use of its POS Tagger, NER Tagger, Stanford Dependency Parsing, and some other tools on the framework to process incoming text.

3.3.3 WordNet Interface

One of the important things in text understanding is being able to extract more information about concepts in the passage. This helps the system gain a deeper understanding into a concept. WordNet is one such digital resource that helps in gaining more knowledge about a concept. WordNet is a large lexical database of English. It consists of a large number of concepts grouped into sets of words that are synonyms i.e. synsets. WordNet has thus created a huge network of concepts by linking these synsets based on lexical relations and conceptual-semantics. WordNet's structure makes it a useful tool for computational linguistics and natural language processing. Java WordNet Interface or JWI is a Java library for interfacing with Wordnet created at MIT. With the help of JWI this system interfaces with WordNet and extracts semantic relations like hypernyms, hyponyms, meronyms etc. to gain more information on the passage.

3.4 Knowledge Generation System

The Knowledge Generation System deals with the generation of rules from text and extracting information from other sources like WordNet. This system is made up of 3 modules, as shown in the Figure 3.3, which are described as follows

3.4.1 Knowledge Extraction from Text

This module is responsible for generating rules and facts from the passage itself. It uses the various rules and patterns mentioned in Chapter 6 to generate part of the ASP program. The ASP rules generated by this module should contain all the information present in the input passage. The input file that is supplied to this module is assumed to be preprocessed according to the earlier mentioned assumptions.
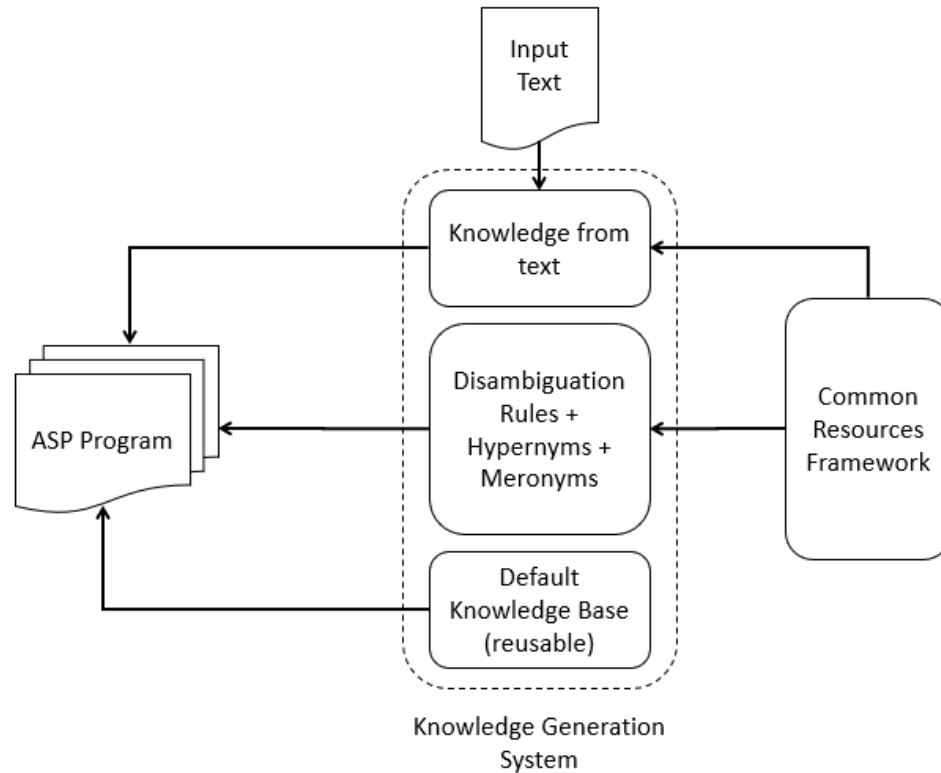
Figure 3.3 Knowledge Generation System

3.4.2 WordNet Ontology Generation

To further understand the concepts mentioned in the input passage, the ontology generation module generates rules regarding disambiguation, hypernyms, and meronyms. Word sense disambiguation forms an integral part of NLP and is also tackled by this module using default reasoning. The Hypernym relation and the Meronym relation along with other relations like Antonyms and Synonyms help capture more semantic information into the ASP program. These topics would be touched upon in detail in the forthcoming chapters.

3.4.3 Default Knowledge Base

As mentioned earlier, there is very little digitalized information about the semantics of verbs in the NLP domain. Hence, to gain complete understanding of verbs and their usage, it is required to

create rules, describing their complete meaning, manually. The default knowledge base makes it feasible to add knowledge about verbs and nouns by hand. Care must be taken to make sure that the knowledge being added is generic in nature and is reusable for other similar scenarios. With the help of such an increasing knowledge base the system can become more efficient and accurate.

3.5 Query Generation System

The Query Generation System is responsible for understanding the question asked in natural language text and converting it into a set of ranked queries that could be understood by the ASP Solver to answer the question. It is comprised of the following 2 modules as shown in Figure 3.4.
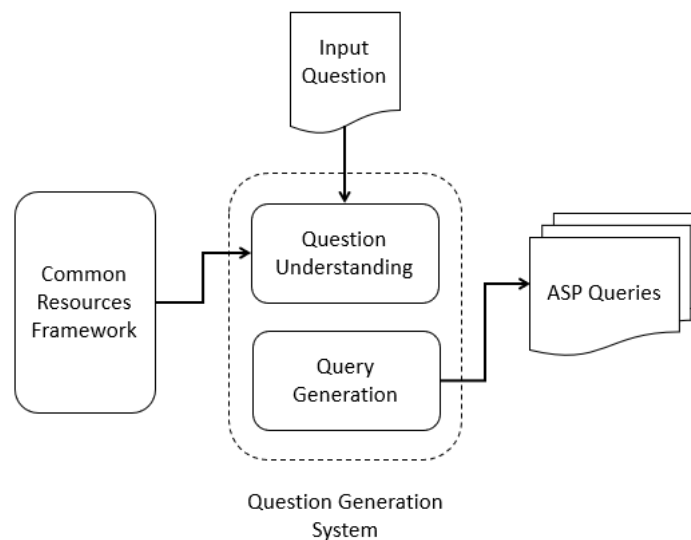
Figure 3.4 Query Generation System

3.5.1 Query Understanding

Questions asked in natural language can be classified into multiple types based on various theories. To classify a question into a specific type requires complete understanding of the question along with the type of answer expected by the question. This module is tasked at finding the various

components of the question including the kind of question, based on the 'Wh' word and the lexical type and kind of answer expected.

3.5.2 Query Generation

Using the information provided by the query understanding module, the query generation module first creates the most constraint query applicable for the question under consideration. This module then starts relaxing certain constraints in the query giving rise to lower quality queries or queries with lower confidence. In the later chapters, we discuss this approach in detail.

CHAPTER 4

SOFTWARE DEVELOPMENT APPROACH

4.1 Overview

This chapter describes the software development approach taken to build the system. I found out that Test Driven Development is one of the best software development approaches for building a rule-based system. This chapter will elaborate on the various stages involved in test driven development and how its principles help in building a stable rule-based system.

4.2 Test Driven Development

Using this approach, a paragraph was divided into multiple tests having a sentence each which was used to develop code for the Knowledge Generation System. Each sentence in the paragraph contributed to my understanding of the grammar rules and patterns in the input text. Test Driven Development, abbreviated as TDD, is a software development process that relies on the repetition of a very short development cycle: first the developer writes an automated test case that defines a desired improvement or new function, then produces a minimum amount of code to pass that test and finally refactors the new code to acceptable standards.

Test Driven Development is known to encourage simple designs that inspires confidence in the code developed under the technique. It is related to the test-first programming concepts of extreme programming. Programmers also apply it to improving and debugging legacy code developed with older techniques. A graphical representation of the typical development cycle can be shown in Figure 4.1.
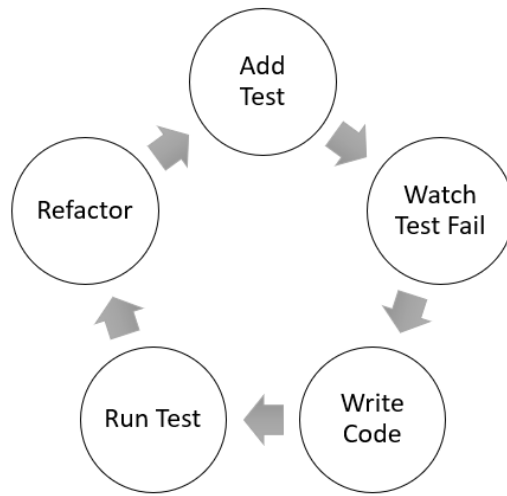
Figure 4.1 Test Driven Development Cycle

4.3 Steps involved in TDD

As shown in the above development cycle, there are 5 major stages in a single development cycle of the approach. A cycle in TDD follows the below mentioned steps in sequence.

4.3.1 Add a test

In test driven development, each new feature begins with writing a test. This test is supposed to fail as it is written before the feature has been implemented. If the test does not fail, then either the proposed feature is already implemented, or the test is defective. To write such a test the developer must completely understand the specifications and the requirements of the new feature. A developer can accomplish this through use cases and user stories to cover the requirements and exception conditions and can write the test in whatever testing framework is appropriate to the software environment. It could also be a modification of an already existing test case. This is a differentiating feature of test driven development which makes the user focus on the requirements before writing any code.

4.3.2 Run all tests and see if the new test fails

This is an important step in TDD, which makes sure that the test harness is working correctly, and the new test does not accidentally pass without requiring writing new code. This step also rules out the possibility that the new test always passes thus making the new test useless. Another important factor in this step is that the test should fail for the right reasons. This makes sure that the test is testing the intended condition and only passes when those conditions are met.

4.3.3 Write some code

The next step in this process is to write some code to pass this newly added test. The code written in this step may not be efficient and may pass the test inelegantly. This is acceptable, here as we are going to improve and hone the design in later stages. It is important to note here that the newly added code should only be designed to pass the current test, and no further functionality should be assumed or predicted.

4.3.4 Run the automated tests all see them succeed

Once all the automated tests pass with the inclusion of the newly added test and its corresponding code then the developer can be confident that the code meets all the current testing requirements. This is a good point from which we can begin the final step of the development cycle.

4.3.5 Refactor code

The entire code can now be refactored to accommodate any new updates and changes required in the design triggered by the addition of new code. By running the entire testing suite, the developer can guarantee that the refactoring has not hampered any functionality in the code. Removing duplicate and dead code is important in software development. This step gives the developer a chance to remove duplicates and improve the code design without affecting code.

4.3.6 Repeat

Now, the developer can start with another test case and repeat the cycle to improve the functionality of the system. It is recommended that the size of the edit should always be small, with as few as 1 to 10 edits between each test run. If the new code does not rapidly satisfy a new test or other tests start failing unexpectedly then it is recommended to undo the previous code change as compared to excessive debugging. In such cases, continuous integration helps by providing revertible checkpoints.

4.4 Development Principles

There are various principles that help while using test-driven development. These principles include "Keep it simple stupid" (KISS), "You are not going to need it" (YAGNI) and many more. Some of these principles along with their advantages has been discussed below.

By focusing on only writing code for passing a single test, the designs can be cleaner than other approaches. To achieve complex design patterns, tests can be written to generate the design pattern. This helps in keeping the changes short and simple to understand, which allows the developer to focus on what is important. Writing tests first before coding up the functionality has been claimed to have many benefits. It helps the developers think about testing from the outset instead of worrying about it later. Also, writing tests first creates a deeper understanding about the concept or feature in the developers mind which in turn helps in writing better code. Failing the test case first, before implementing its required feature, ensures that the test really works and can catch bugs. Test driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage in the development cycle reinforces the developers mental model of the code and boosts confidence in the code. Keeping

the changes to be smaller has proved to have multiple benefits including reduced debugging effort and better understanding of code. These principles if followed correctly enable developers to build large scalable systems without adding to the complexity of debugging and maintaining the system.

CHAPTER 5

NATURAL LANGUAGE RESOURCES

5.1 Overview

This chapter gives an overview on the various natural language resources that are used to build the system. The Stanford Core NLP Toolset consists of many tools including the POS Tagger, Parser, Co-Reference Resolution, and Dependency Parser. This chapter would mainly deal with the details about the dependency parser, POS tagger, NER Tagger and the concepts relating to WordNet and its relations. We will make use of the concepts discussed in this chapter in the next chapter while discussing the different knowledge extraction techniques used.

5.2 Resource Tools

5.2.1 Dependency Parser

A dependency parser analyses the grammatical structure of a sentence and returns a set of relations between different words of the sentence. In general, one of these words is the independent word or the head word and the other is the dependent word in the relation. The dependent word modifies the independent word in the sentence using the relation. Consider the sentence "John gave Mary the book.". The Figure 5.1 marks the various dependencies in the sentence. The dependencies can be given as follows
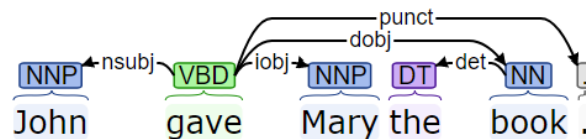
Figure 5.1 Universal Dependencies for the sentence "*John gave Mary the book*"

root(ROOT-0, gave-2), nsubj(gave-2, John-1), dobj(gave-2, book-5), iobj(gave-2, Mary-3), det(book-5, the-4), punct(gave-4, .-6).

We will discuss each of these dependencies and their meanings in detail later in the chapter. In the above-mentioned dependencies, the first word is the independent word, the second word is the dependent word and the predicate of the dependency describes the type of relation between the words. Consider the nominal subject dependency relation from the sentence

nsubj(gave-2, John-1)

Here, "gave" is the independent word, "John" is the dependent word and the relationship between these words is of the "nominal subject" or "nsubj". Sometimes these relations have specifics mentioned along with them e.g the relation "nmod:poss" states that the relation is of a nominal modifier which shows a possessive relation.

5.2.2 Part of Speech Tagger

A Parts of Speech Tagger is responsible for assigning parts of speech to words in a sentence. The English language has eight parts of speech: noun, verb, pronoun, preposition, adverb, conjunction, particle, and article. Apart from these parts of speech categories tags are also applied to punctuations in a sentence. A tagging module uses certain predefined tag sets to tag various words. A tag set defines the various tags and their meanings that the POS Tagger outputs. The English tagger in the Stanford POS Tagger uses the Penn Treebank tag set. Let us take the previously mentioned example as a sentence and tags its parts of speech. You can understand the meaning of each tag from the Table 5.1.
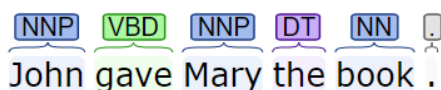
NNP VBD NNP DT NN .
John gave Mary the book .

Table 5.1 Penn Treebank Parts of Speech Tag Set

Tag	Description	Tag	Description
CC	Coordinating conjunction	PRP$	Possessive pronoun
CD	Cardinal number	RB	Adverb
DT	Determiner	RBR	Adverb, comparative
EX	Existential *there*	RBS	Adverb, superlative
FW	Foreign word	RP	Particle
IN	Preposition or subordinating conjunction	SYM	Symbol
JJ	Adjective	TO	to
JJR	Adjective, comparative	UH	Interjection
JJS	Adjective, superlative	VB	Verb, base form
LS	List item marker	VBD	Verb, past tense
MD	Modal	VBG	Verb, gerund or present participle
NN	Noun, singular or mass	VBN	Verb, past participle
NNS	Noun, plural	VBP	Verb, non-3rd person singular present
NNP	Proper noun, singular	VBZ	Verb, 3rd person singular present
NNPS	Proper noun, plural	WDT	Wh-determiner
PDT	Predeterminer	WP	Wh-pronoun
POS	Possessive ending	WP$	Possessive wh-pronoun
PRP	Personal pronoun	WRB	Wh-adverb

In general, many of the most common words used in English belong to more than one category of part of speech. As an example, the "book" can both be a verb and a noun. Thus, the task of parts of speech tagging is also required to disambiguate between the various possible tags that can be applied to a word. Following are some of the major problems that most taggers face, which affect tasks like information extraction that use these taggers as a source of information.

a. *Confusion between NN/NNP/JJ*

Proper Nouns, Nouns and Adjectives are predominantly hard to distinguish between as all of them form parts of the Nominal Phrase and can be reordered in multiple ways in English.

b. *Confusion between RP/RB/IN*

All the above parts of speech can occur immediately after the verb. It is especially hard to distinguish between particles and prepositions as both classes share certain words.

c. *Confusion between VBD/VBN/JJ*

Boundaries of noun phrases are determined using the above parts of speech and hence differentiating between them plays a crucial role in parsing.

5.2.3 Named Entity Recognizer

A Named Entity Recognizer is a module used to label a sequence of words in a sentence with predefined tags of Named Entities. Named Entities are names of things, such as person, company, organization, locations, cities and many more. Named Entity Taggers can be built for custom texts and passages with a rich predefined tag set. The Stanford Named Entity Tagger is trained on various models for the English Language. The various models are given as follows:

a. 3 class: *LOCATION*, *PERSON*, *ORGANIZATION*

b. 4 class: *LOCATION*, *PERSON*, *ORGANIZATION, MISC*

c. 7 class: *LOCATION*, *PERSON*, *ORGANIZATION, MONEY, PERCENT, DATE, TIME*

Let us take the sentence "John, who works at UTD, lives in Dallas." as an example for the NER

The Named Entity Recognizer is one of the important sources of information for the information extraction task in NLP.

5.3 Stanford Universal Dependencies

The Universal Dependency (UD) Relation taxonomy divides the relations into structural categories as Nominals, Clauses, Modifier Words and Function Words. Similarly, the relations can be divided into functional categories with relation to the head as Core Arguments, Non-Core Dependents, and Nominal Dependents. We will study the meaning of some important dependency relations based on the functional categorization.

5.3.1 Core Arguments

Core Arguments are mostly relations that talk about the various participants that are involved in the event. These relations play an important role in understanding the sentence. Some of these core argument relations are given as follows:

5.3.1.1 *nsubj: nominal subject*

A nominal subject is a nominal that acts as the subject or agent of a clause. This nominal contains the do-er of the action. The head of such a nominal could be a noun, pronoun, or other things such as adjectives. Similarly, if the verb involved in the *nsubj* relation is a copula then the governor of the relation may not be a verb but now is a noun or an adjective related with the copula. In the case of passive sentences the relation *nsubj:pass* is used to indicate the subject instead.

John is liked by teachers .
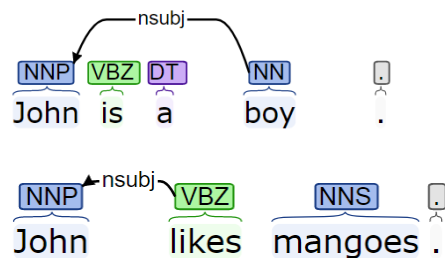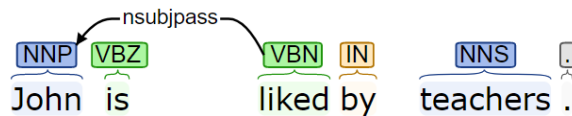
5.3.1.2 *dobj: direct object*

A direct object is a noun phrase or a nominal that is the accusative object of the verb. The verb, here is the head of the verb phrase that governs this relation.

John likes mangoes .

5.3.1.3 *iobj: indirect object*

A nominal phrase that is neither the subject nor the direct object of the verb but is a core argument of the verb is called the indirect object of the verb. In general, if there is only one object then its considered to be a direct object, but in the case of more than one objects, one of them is direct whereas all other are indirect objects of the verb.

John gave me an apple .
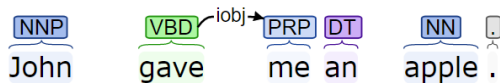
5.3.1.4 *csubj: clausal subject*

A clausal subject is basically a clause that acts as the subject of the sentence. In general, the governor of this relation is a verb but, in the case, where the main verb is a copular verb the governor can be a non-verb. The dependent word for a *csubj* relation is the head verb of the subject clause. The *csubj* relation is also used for the passive verb or a verb group. We make use of the specific *csubj:pass* in passive transformations.

What John says is always true
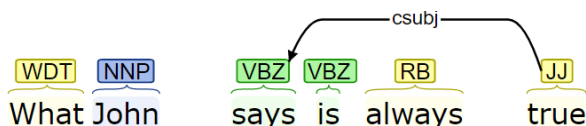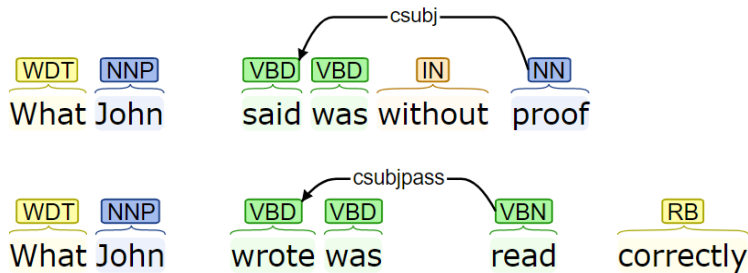
5.3.1.5 *ccomp: clausal complement*

The dependent clause which is the core argument of the verb is called as the clausal complement. Thus, such a clause behaves like a subordinate clause to the main verb i.e. the governor of this relation.
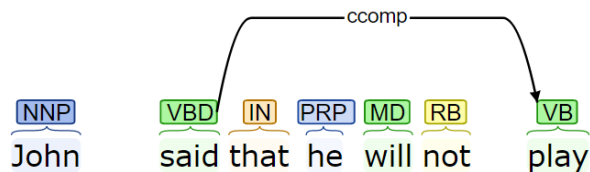
5.3.1.6 *xcomp: open clausal complement*

A clausal complement that does not contain its own subject is called as the open clausal complement. The subject is usually determined by a higher clause in the sentence.
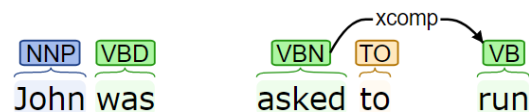
5.3.2 Non-Core Dependents

5.3.2.1 *advcl: adverbial clause*

An adverbial clause is a clause that modifies a verb. An adverbial clause can include a temporal clause, condition, consequence, effect, purpose etc. Here the dependent entity must be a clause otherwise the relation becomes *advmod*. The dependent entity represents the head verb of the clause.
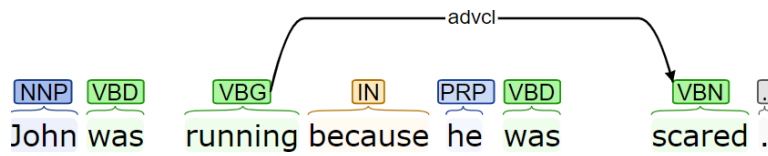
5.3.2.2 *advmod: adverbial modifier*

An adverbial modifier is an adverb or an adverbial phrase that modifies a predicate. Here, the adverbial phrase should not be a clause, otherwise it is marked as *advcl*. Here in general the dependent is an adverb whereas the governor is the head verb of the predicate that the adverb is modifying.
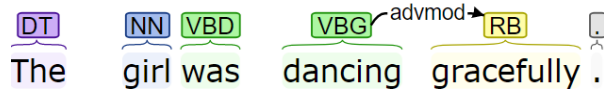
5.3.2.3 *aux: auxiliary*

An auxiliary is a function word in a clause that expresses categories like mood, tense, aspect, voice and evidentiality, Auxiliaries are often verbs, that may or may not have non-auxiliary uses, but many languages also have non-verbal auxiliaries. In general, auxiliaries used to create passive voice are also marked with the *aux* relation.
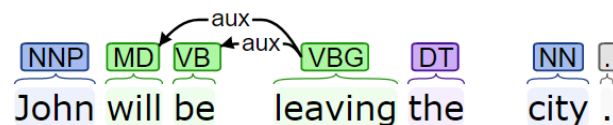
5.3.2.4 *cop: copula*

A copula relation is used to connect a subject to a non-verbal predicate. Although, copulas are verbs in general, there exist non-verbal copulas in certain languages. The copula "be" is not considered as the head of the clause but is a non-verbal predicate. A copula relation is not applied when the non-verbal predicate is used in the form of a clause.

cop

NNP	VBZ	DT		NN
John	is	a		boy

ccomp

DT	NN		VBZ	IN	DT	VBZ	**cop**	JJ
The	issue		is	that	this	is		illegal
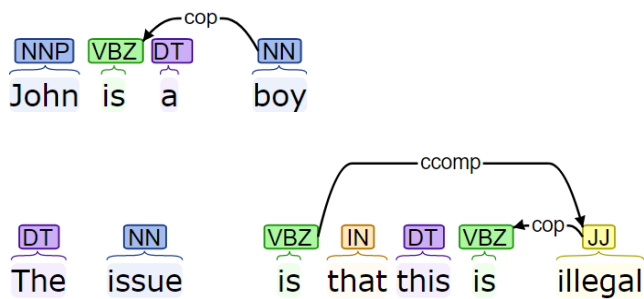
5.3.2.5 *mark: marker*

The marker is a word that connects the subordinate clause to another clause. For a clausal complement a marker is a word like that or whether in English. Whereas for an adverbial clause the marker is a subordinating conjunction. The mark relation has the governor as the subordinate clause head and the dependent is the marker itself.

mark

NNP		VBD	IN	PRP	MD		VBG
John		said	that	he	will		sleeping
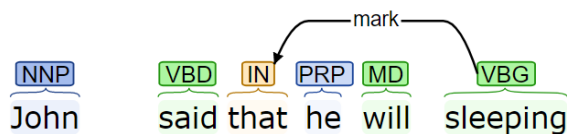
5.3.3 Nominal Dependents

5.3.3.1 *nmod: nominal modifier*

The nominal modifier relations describe different attributes or properties of verbs, adverbs, nouns. The dependents in such a relation are nouns or noun phrases and the governors can be verbs, nouns or other parts of speech. Nominal modifiers can have specifics like *nmod:poss*, describing possessive relations or *nmod:tmod*, describing temporal relations.

nmod

NNP		VBZ	TO	DT		NN
John		goes	to	the		mall

nmod:poss

DT	VBZ	NNP		POS	NN
This	is	John		's	book
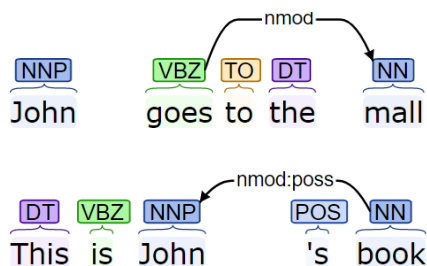
5.3.3.2 *appos: appositional modifier*

Appositional Modifiers are nominals that directly follow the noun that defines, modifies, or describes the noun. Appositional modifiers also include parenthesized example as well as abbreviations.
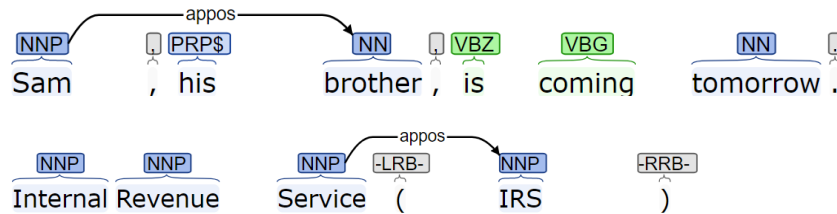
5.3.3.3 *nummod: numerical modifier*

Any number or a number phrase that describes more about a noun or a noun phrase in the form of a quantity is part of the numerical modifier relation. Here, indefinite quantifiers such as few, many, a lot etc. are marked as *det* and not *nummod*.
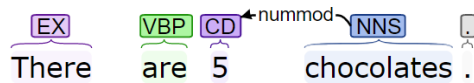
5.3.3.4 *acl: adjective clause*
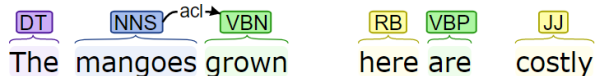
Clauses that modify a nominal are marked as adjective clauses or *acl*. These are different from *advcl*, which modifies a predicate. The head of the relation is the nominal that is modified and the dependent of the relation is the head of the clause that modifies the relation.

5.3.3.5 *amod: adjective modifier*

An *amod* or adjective modifier is an adjective or an adjectival phrase that describes more about the noun or gives meaning to the noun. Here the head of the relation is the modified noun whereas the dependent of the relation is the modifier.

JJ ←amod— NNS   VBP   JJ
Green-leafy vegetables are   healthy
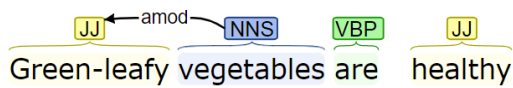
5.3.3.6 *det: determiner*

The determiner relation holds between a nominal and its determiner. Generally, the words having the POS tag as DET belong to this relation. The head is the nominal and the dependent is the determiner in this relation.

DT VBZ DT VBP   NN
This is   a   brave   man
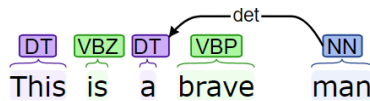
5.3.3.7 *case: case marker*

The case relation is used to mark any element treated as a separate syntactic word like a preposition, a possessive altercation etc. Here the governor of the relation is the head of the nominal phrase whereas the dependent is the syntactic word under consideration.

NNP   VBD   IN ←case— NNP   TO ←case— NNP
John   traveled from   Europe to   Asia

NNP —case→ POS NN VBZ   VBG
Mary   's   bag is   missing
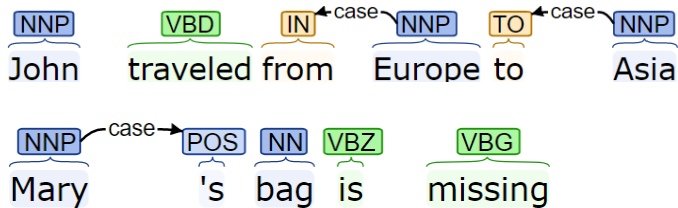
5.3.4 Other Dependency Relations

5.3.4.1 *conj: conjunct*

The conjunct relation is marked between words or elements the are connected via a coordinating conjunction like *and*, *or* etc. The conjunct relation is asymmetrical, here the head of the relation is first conjunction and the dependents are the other conjunctions that depend on the first conjunction through the *conj* relation. Such a relation is also marked if the conjunctions are omitted or replaced by commas or any other punctuation symbols.

PRP He | VBZ likes | NNS apples | , | NNS mangoes | CC and | NNS oranges

NNP John | VBD understood | CC and | VBD solved | DT the | NN problem
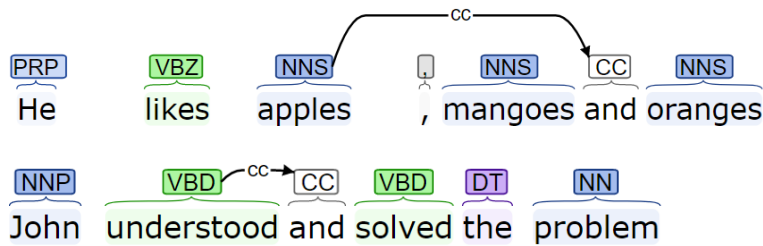
5.3.4.2 *cc: coordinating conjunction*

The relation *cc* is marked between the conjunct and a preceding coordinating conjunction. The *cc* relation accompanies the *conj* relation mentioned before.

PRP He | VBZ is | DT a | JJ strong | CC and | JJ healthy | NN child
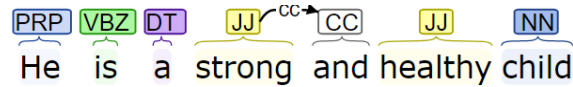
5.3.4.3 *compound: compound*

The compound relation is used for multiword expressions like joint ventures. These can also be applied to proper nouns like American Airlines and verbs that have particles like stand up. For particles the compound relation contains a specific and is marked as *compound:prt*.

DT The | NNP American | NNPS Airlines | NN flight | VBD took | RP off
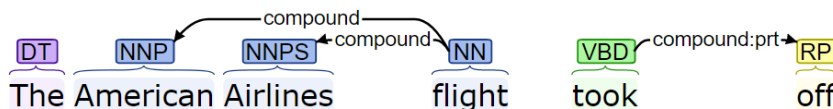
5.3.4.4 *root: root*

The root relation marks the root of the sentence. A fake node ROOT-0 is created which governs the relation and the dependent is the head of the sentence, which in most cases is the main verb. The ROOT node starts at 0 as the rest of the sentence is indexed from 1.

ROOT | PRP I | VBP like | NNS fries

39
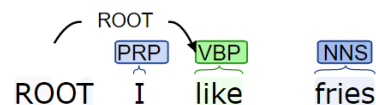
5.4 WordNet

WordNet is one of the most commonly used resources in English. It is a lexical database consisting of sense relations between English words. WordNet consists of separate databases one each for nouns, verbs and a database for adjectives and adverbs. Databases are only created for open word classes in WordNet. WordNet contains a set of near synonyms called as synsets and marks relations between these synsets. We will discuss some of the important relations, present in WordNet, later in the chapter. In each of the databases, there consists a set of lemmas, each one annotated with a set of senses. WordNet can be accessed on the Web or downloaded and accessed locally. A typical entry for the noun "*lion*" in WordNet yields the different senses shown in Figure 5.2.

```
The noun lion has 4 senses (first 1 from tagged texts)

1. (2) lion, king of beasts, Panthera leo -- (large gregarious predatory feline of Africa and
   India having a tawny coat with a shaggy mane in the male)
2. lion, social lion -- (a celebrity who is lionized (much sought after))
3. Leo, Lion -- ((astrology) a person who is born while the sun is in Leo)
4. Leo, Leo the Lion, Lion -- (the fifth sign of the zodiac; the sun is in this sign from about
   July 23 to August 22)
```

Figure 5.2 Entry for the concept of "lion" in WordNet

Due to the presence of multiple senses for each word in WordNet, disambiguation is of the utmost importance when using WordNet as a source for any application.

5.4.1 WordNet Relations

WordNet contains various relations depending upon the word type. Some of the noun relations include Hypernym, Hyponym, Instance Hypernym, Instance Hyponym, Meronyms, and many more. Some of the verb relations include Hypernyms, Troponyms, Antonyms etc. We now discuss some of the above-mentioned category relations.

5.4.1.1 *Hypernyms*

One of the most commonly used WordNet relations is the Hypernym relation. It connects specific entities to their more general entities. This relation is also called as the IS_A relation. This relation states that the category *vehicle* includes the *motor vehicle* which in turn includes the *car*. In this way all noun hierarchies eventually end in the root node, *entity*. The Hypernym relation is transitive in nature i.e. if a *car* is a *motor vehicle* and the *motor vehicle* is a *vehicle*, then we can conclude that the *car* is also a *vehicle*. WordNet can also distinguish between types of nouns versus instances of nouns. As an example, a car is a type of a vehicle, whereas Kenya is an instance of a country.

5.4.1.2 *Meronyms*

Meronyms indicate the part-whole relationship between any two concepts in WordNet. For example, A *horn*, *air bags*, and an *engine* are parts of a *car*. This relationship can be inherited from the super ordinates but not from the subordinates as some of the relations may be characteristic to certain concepts. As an example, all cars will have horns, but not all vehicles (submarines) will necessarily have a horn.

5.4.1.3 *Synonyms*

Synonyms are two different concepts having similar or nearly identical senses. Two words can also be said to be synonymous if they are substitutable for each other in the sentence. The synonym relation can be found in nouns, adjectives, adverbs as well as verb categories. For example, *car/automobile*, *eat/consume/take in*, *pretty/beautiful*, *quickly/rapidly* are all synonyms of each other.

5.4.1.4 *Antonyms*

Antonyms on the contrary to Synonyms are concepts or words with opposite meaning. Antonyms have various definitions, which make it hard to define antonyms. Antonyms are concepts that may be on the opposite end of a scale or a measurement i.e. *long/short*, *fast/slow*, or *positive/negative* which are all concepts that lie on the opposite sides of a scale. Another definition of antonyms describes some change in the direction or movement in opposite direction which can be given by *up/down, left/right*, and so on. Antonyms are somewhat like synonyms, as antonyms almost have similar meanings in all aspects except for the fact that they belong to the opposite sides on a scale. Thus, due to the cryptic definitions of antonyms it is often difficult to decide between synonyms and antonyms.

5.4.2 WordNet Senses

WordNet has defined 45 lexical categories for synsets during its development. Synsets were organized into these categories based on the syntactic category and logical groupings of the synsets. The syntactic categories based on which the synsets are divided are NOUN, VERB, ADJECTIVE and ADVERB. Let us consider some of the sense categories for nouns as generated by WordNet. With the help of these sense categories we will later discuss how we can generate ASP code to represent hypernym relations efficiently without blowing up the space requirements for the system. The Table 5.2 shows the lexical categories for nouns along with their meaning.

Table 5.2 Lexical Categories for WordNet Senses

Name	Content
noun.Tops	unique beginner for nouns
noun.act	nouns denoting acts or actions
noun.animal	nouns denoting animals
noun.artifact	nouns denoting man-made objects
noun.attribute	nouns denoting attributes of people and objects
noun.body	nouns denoting body parts
noun.cognition	nouns denoting cognitive processes and contents
noun.communication	nouns denoting communicative processes and contents
noun.event	nouns denoting natural events
noun.feeling	nouns denoting feelings and emotions
noun.food	nouns denoting foods and drinks
noun.group	nouns denoting groupings of people or objects
noun.location	nouns denoting spatial position
noun.motive	nouns denoting goals
noun.object	nouns denoting natural objects (not man-made)
noun.person	nouns denoting people
noun.phenomenon	nouns denoting natural phenomena
noun.plant	nouns denoting plants
noun.possession	nouns denoting possession and transfer of possession
noun.process	nouns denoting natural processes
noun.quantity	nouns denoting quantities and units of measure
noun.relation	nouns denoting relations between people or things or ideas
noun.shape	nouns denoting two and three-dimensional shapes
noun.state	nouns denoting stable states of affairs
noun.substance	nouns denoting substances
noun.time	nouns denoting time and temporal relations

CHAPTER 6

KNOWLEDGE REPRESENTATION

6.1 Overview

This chapter describes how using all the concepts mentioned in the previous chapter, the system generates facts and rules from text. This process involves creation of a semantic graph which acts like an event tree that binds various events in the text together. Using this semantic graph, we start generating facts in the form of some predefined predicates. Section 6.3 defines these predicates with examples. Apart from the knowledge that the system extracts from the text, it also makes use of WordNet and creates certain rules from it. Section 6.4 dives deep into how we can make use of the various WordNet synsets to understand text and how we can use ASPs preference patterns for sense disambiguation while using these synsets.

6.2 Semantic and Event Graph

One of the first steps after parsing the sentences in the text is to generate a semantic graph from the sentence. We create such a graph with the help of the Stanford Typed Dependencies representation mentioned in the earlier chapter. It gives a simple description of the grammatical relationships in a sentence that can easily be understood and effectively used to extract textual relations. Consider the following sentence as an example

Example: "NASA carried out the Apollo program."

For this sentence some of the Stanford Dependency (SD) representations can be given as:

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)

ROOT
|
root
↓
carried
— *nsubj*
prt
|
dobj
↓
out   program   NASA

compound   *det*

Apollo   the
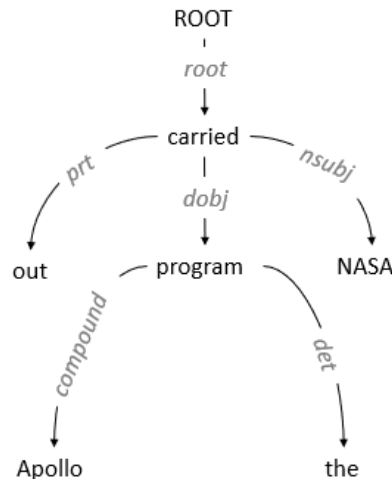
Figure 6.1 DAG representing dependency relations

These dependencies map straightforwardly onto a directed graph representation, shown in the Figure 6.1, in which words in the sentence are nodes in the graph and grammatical relations are edge labels. The Figure gives the graph representation for the example sentence above. The meanings of these type dependencies are explained in the previous chapter in detail. This semantic graph thus created has been used to extract the different facts from the sentence. 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. In the above sentence there is only one event having the head verb "carried

45

out". We can demonstrate the use of the event graph by taking a more complicated example as follows.

Example: "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."
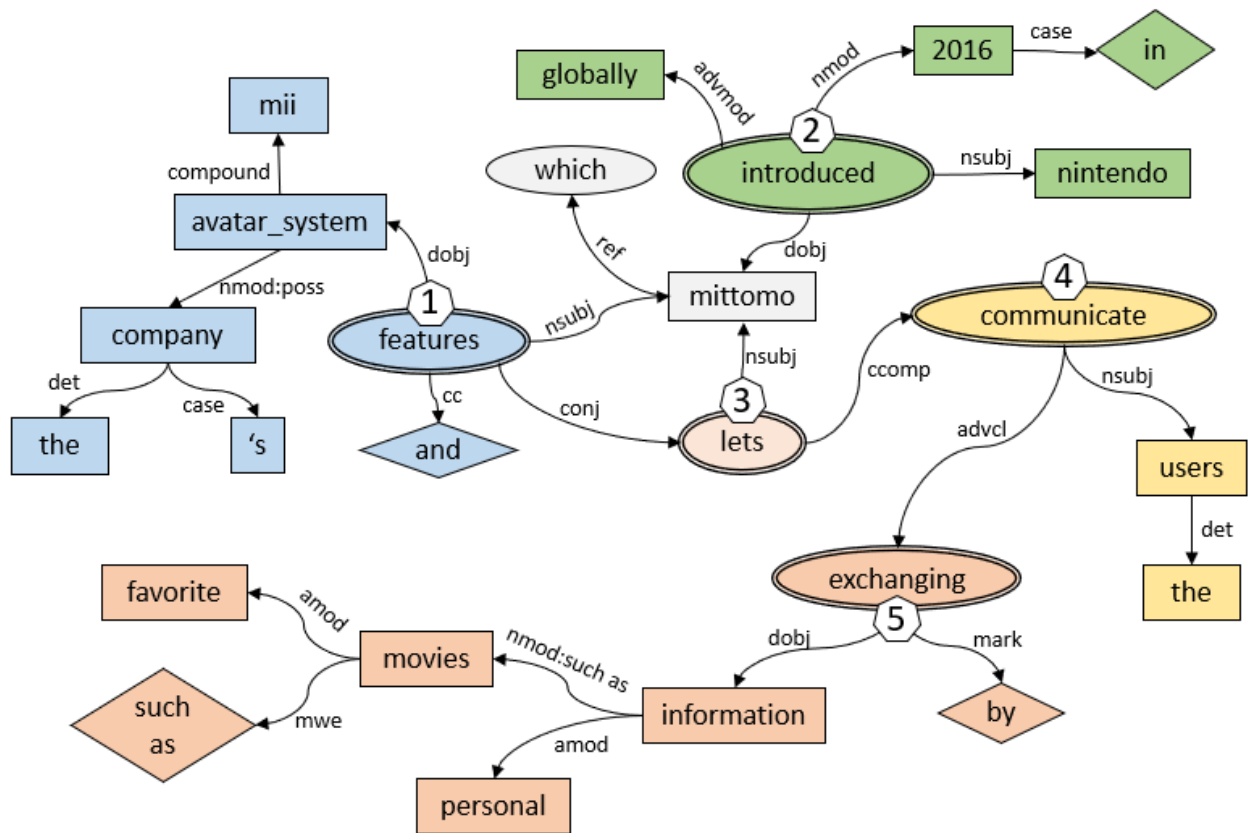
Figure 6.2 Semantic graph representing event regions

The Figure 6.2 shows how various words are connected to each other in the sentence. This example is a complex sentence consisting of multiple clauses. The main verbs of the sentence being "feature" and "let" connected by a coordinating conjunction. In the above diagram the verbs have been marked using double-bordered ovals, the prepositions have been shown using diamonds

whereas all other words like adjectives, nouns and adverbs have be shown using rectangular boxes. The verbs in the diagram 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. Many of the concepts in the sentence also take part in multiple events, for example the word, Mittomo. It is with the help of these event regions that we can infer missing or implied information from the sentence. We will go over the details of how the semantic graph with these event regions are used to create facts and rules in the rest of this section.

6.3 Predicate Generation

In this section, we would go over the various predicates that have been defined to represent information. Some of the predicates are special cases like abbreviation, start_time etc. whereas others are more generic like mod, event and so on. The more generic predicates we generate the easier it is to query about information from the ASP program. The generic predicates that have been generated try to convey information that is more explicitly present in the text. Whereas the special kinds of predicates can be used to model implied information in text. Implied reasoning forms an integral part of the human or common-sense reasoning, thus we can add more special predicates for better performance. Following are some of the important predicates that have been used by the system.

6.3.1 Event Predicate

The event predicate defines an event that happens in the sentence. The verb marks the head of the event predicate. Any event in general 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*)

The signature of the event predicate consists of 4 terms. The *event_id* is an integer that uniquely identifies that event in the sentence or paragraph. It is using this *event_id* that we can differentiate various properties and relations in the text. The *event_id* is tightly coupled with the events *trigger_verb*. The *trigger_verb* denoted in the event predicate is the lemma of the actual word used in the sentence. Using lemmas of trigger verbs help in easily querying for an event. Event predicates are created for all kinds of events i.e. both active versus passive events. Active events are those that are trigged by action verbs whereas events triggered by passive verbs like copulas or modal verbs are more passive in nature. 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*. Here *nsubj:xsubj* is a special controlling subject used to find relations between the main subject and the subordinate clause. Just as actors can be obtained from the subject of the sentence, the participants can be determined from the direct object dependency (*dobj*). Consider the following sentence from a 'Super Bowl 50' passage as an example along with the event predicates that are generated from it.

Example: "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)

From the above predicates, we can understand that there are two main events occurring in the sentence, i.e. the first relating to a defeat and the other relating to earning of something. The system

has marked the event "defeat" with ID 1, whereas the event of earning has been marked with the ID 2. This is how we can capture the information that 'denver_broncos defeated carolina_panthers' and 'afc earned the title'. Here we can get richer information about the event if we know more about the actors and the participants in the event. Thus, duplicate event predicates are also created for each *modifier* for the actor as well as the participants involved in the event. To generate such event predicates, we can use the *amod* or *nummod* dependencies for the actors and the participants and create compound atoms from the modifiers and their governors. Such supplementary event predicates, although are duplicates but make querying easier. As an example, consider the same sentence as before and this duplicate event predicate that is generated shedding light upon the kind of title won.

event (2, earn, afc, third_super_bowl_title)

The default value for the actor and the participant field is *null*. A *null* value indicates that either the term is absent for the event or the system was not able to determine it.

6.3.2 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. Here, the *modifier* is different from the modifiers that are encountered as adjectives or adverbs that are marked as *amod* and *advmod* respectively. A property predicate is coupled with an event and describes the modification only for that event. The signature for the property predicate can be given as follows

_property (event_id, modified_entity, preposition, modifier)

This signature contains 4 terms that together define the modification. The *event_id* of this predicate should belong to one of the events in the context. It is with the help of this term that we can

constraint our query to an event and answer queries accurately with respect to the event under consideration. The *event_id* used here is propagated to the *modified_word* while creating the semantic graph. The *modified_word* 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. The *preposition* in this relation helps in identifying the kind of *modifier* used in the relation. The following example gives example of both the *modified_word* being a noun as well as a verb.

Example: "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*. To give context let us also consider the event with ID 2 along with this example.

event (2, play, null, game)

From the above facts we understand that the game is/was/will be played '*on february_7_2016*', '*at levis_stadium*', '*in san_francisco_bay_area*', and '*at santa_clara*'. Here we can also conclude that santa_clara is '*in california*'. Thus, with the help of the property predicates we can understand more about the event regarding its properties like time, place and many more. While generating property rules we exclude the *nmod* relations having the specifics like *poss*, and *such_as* as these are handled as special cases.

6.3.3 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. The signature of the *mod* predicate can be given as follows.

_mod (modified_word, modifier_word)

Here, 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. Similarly, if the *modified_word* is a verb then the *modifier_word* is an adverb given by the *advmod* relation. Sometimes, nouns are also modified using the adjectives of quantity, which are marked using the numerical modifier relation or the *nummod* dependency. An example for the same can be shown as follows

Example: "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)

The above sentence generates the modifier predicate having both the noun as well as the verb as the *modified_word*. Here the "*forest*" is modified by the adjectives "*broadleafed*" and "*moist*", and the verb "*know*" is modified by the adverb "*also*". Using this predicate, we can infer about the qualities of the nouns and the verbs in the sentence.

6.3.4 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. The possessive predicate given by _possess 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*. As defined previously appositional modifiers defines or describes a noun. An example that shows genitive case in English and the extension of the possessive case using appositional modifiers can be given as follows

Example: "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*). According to the first two facts, the "*american_football_conference*" and the "*national_football_conference*" each possess a team. The names for these team can be inferred using the *appos* relation given by the dependency relations "*appos (team, denver_broncos)*" and "*appos (team, carolina_panthers)*". From the above

two observations we can conclude the two facts that "the *american_football_conference* possesses *denver_broncos*" and "the *national_football_conference* possesses *carolina_panthers*"

6.3.5 Instance Predicate

Instance predicates are used to model the concept associated with being an instance of a concept. As an example, red is an instance of a color, so we need to model the fact that "red is a color" as follows *color (red)*. This fact states that red is an instance of the concept of color. Alternatively, we can also generate the predicate *_is (red, color)*, which lets us query both the concept or the instance. This 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*. As mentioned while describing the *nsubj* relation, if the verb in the sentence is a copula then the governor of the relation is the noun associated with the copula. In such a case we can make use of this relation with the copula to generate this predicate. An example of such a generation can be seen in the following sentence.

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

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

The above example, generates the following dependencies, *nsubj (inventor, nikola_tesla)* and *cop (inventor, was)*. From these dependencies we can generate the above given instance predicate. Similarly, 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 the following other facts.

_is (nikola_tesla, electrical_engineer).
_is (nikola_tesla, engineer).
_is (nikola_tesla, futurist).
_is (nikola_tesla, mechanical_engineer).
_is (nikola_tesla, physicist).

Adding these facts makes the knowledge base stronger and is now able to infer many other things about the passage. Such other facts can be generated with the help of the *conj* dependency that connects concepts like *engineer, physicist*, *and futurist* to the first concept of an *inventor*. 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. Consider the following sentence as an example for the same.

"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.",

_is (movie, personal_information)
_is (favorite_movie, personal_information)

In the above example we use the expression "*such as*" to compare two concepts to be equivalent in the sense that one concept is a likely equivalent of the other. In such cases the generated instance predicate is shown above. Such predicates can be generated using the *nmod*, *mwe*, and the *case* dependencies.

6.3.6 Relation Predicate

The relation predicate is used to connect two concepts in events. This predicate is generated to model mainly two relations i.e. dependent clauses and conjunctions. The signature of the relation predicate can be given as follows.

_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* is always 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*. All these relations are discussed below with examples.

As mentioned above the *_clause* relation is generated for adjective and adverb clauses. Examples for both these clauses is given as follows.

Example: "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"

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

In the above given sentence, there are two clauses, one headed by the verb "*defeat*" and the other by the verb "*earn*". Here, the clause headed by "*defeat*" is the main clause whereas the adverb

clause headed by "*earn*" is the dependent clause. Such a relation can be found out using the dependency *advmod*. Similarly, consider the adjective clause in the following sentence

"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)

In this sentence, the noun "*American_Broadcasting_Company*" is modified by the dependent clause headed by the verb "*stylize*". Such a predicate 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. An example of such a relation is given as follows.

"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.

"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)

In the above example we use the *_conj* relation type to model the conjunction relation between the verbs "heat" and "transform". As seen above the *_conj* relation type uses ID's for verb conjunctions whereas actual nouns for noun conjunctions.

6.3.7 Named Entity Predicates

As mentioned in the Chapter 5, the system uses the Named Entity Tagger 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. We will discuss about how the predicates generated using the Named Entity Tagger look like and examples where they may be useful.

6.3.7.1 *Time Predicate*

The time predicate deals with recognizing a date, timestamp, time of the day, month, or year in the sentence. Thus, a time predicate is used to understand that the term inside the predicate represents temporal knowledge. The signature of the time predicate can be given as

time (term)

Most of the models in the Stanford Named Entity Tagger mark time entities using the tag 'DATE'.

John was born on 5 January 1990 `[Date]`

This predicate proves useful in constraining queries regarding time. Questions that look for a date, month or a year can apply this predicate as an additional constraint to improve accuracy. Some of the examples using this predicate include '*time (1745)*', '*time ('7_january_1943')*'.

6.3.7.2 *Location Predicate*

The location predicate is used for identifying specific locations. These may include cities, states, regions, countries and many more. The Named Entity Tagger marks locations with the tag LOCATION.

John lived in the urban part of San Francisco [Location]

The signature for this predicate can be given as

location (instance)

The *instance* represents the name of the location. Locations can be used to constraint Where queries. Furthermore, with the help of knowledge from WordNet, we could infer more about the event under consideration.

6.3.7.3 *Organization Predicate*

The organization predicate is used to recognize organizations like groups, companies, products etc. With the help of this predicate we can better answer questions regarding entities like companies, their ventures, subsidiaries etc. The Named Entity Tagger marks organizations with the tag ORG.
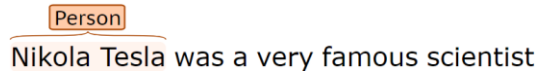
Skype [Org] was bought by Microsoft [Org]

The signature of the Organization predicate can be given as

organization (instance)

6.3.7.4 *Person Predicate*

The person predicate marks all the people in the given text. It is used to identify persons in the text so that we could infer more about those people with the help of WordNet. The NER would annotate the person with the tag PERSON as shown below.

Person
Nikola Tesla was a very famous scientist

The signature of the Person predicate can be given as

person (instance)

Using this predicate, we can apply generalization rules about persons on the instance. For example, if we have a generic rule that "All persons can run", then now if we can prove that 'Nikola Tesla' is a person then we can automatically infer that "Nikola Tesla can run".

6.3.8 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 out specially. 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. We will discuss some of them in this chapter.

6.3.8.1 *Time Span Predicates*

Time spans are sometimes expressed in text using the bracketed notion. The meaning of these time spans depends upon the noun that they follow. If the noun is a person, then it may mean that the time span indicates the birth and the death date of the person. On the contrary if the noun is an organization or a product or a project then the time span indicates the start and end date for the organization or the project. A timespan is generally of the format "(DATE - DATE)". The first

date is considered as the start date and the second is considered as the end date in the format. The signature for the start and the end date can be given as

_start_date (entity, date)
_end_date (entity, date)

Here the entity refers to the noun that precedes this timestamp form and the date would indicate either the start or the end date extracted from the format. Some examples for these are given in the following sentences.

Example: "Project_Mercury was followed by the two-man Project_Gemini (1962 – 1966)"

_start_date (project_gemini, 1962)
_end_date (project_gemini, 1966)

Example: "Martin_Luther (10 November 1483 – 18 February 1546) was a German_professor of theology, composer, priest, former_monk and a seminal_figure in the Protestant_Reformation."

_start_date(martin_luther, '10_november_1483')
_end_date (martin_luther, '18_february_1546')

In the first example the timespan represents the start and the end time for project gemini and in the second example the same denotes the life span of Martin Luther.

6.3.8.2 *Date Part Predicates*

The time predicates that have been detected from the Named Entity Tagger, can be utilized to get the day, month, and year parts from the time. This information about the various parts of time is understood implicitly by humans. We can use simple information extraction techniques to find out the year, month, and day predicates from the time predicate. The signature of these predicates can be given as follows

day (time, day)

month (time, month)

year (time, year)

Here, the *time* refers to the entity tagged as TIME by the NER and the second term refers to the extracted day, month and year represented in the time. This is useful information for answering specific queries relating to time. Consider the time predicate and its parts mentioned below.

time ('10_november_1483')

day ('10_november_1483', 10)

month ('10_november_1483', november)

year ('10_november_1483', 1483)

From the previous section we know that the time mentioned is the birth date of Martin Luther, so using these date parts we can now query the day, month, or year that Martin Luther was born.

6.3.8.3 *Concept Predicates for Appositional Modifiers*

Appositional modifiers directly follow the noun they describe. We could use this information to model the fact that most appositional modifiers follow the *instance_of* pattern. This is used to generate concept predicates from the appositional modifiers. Consider the following example for creating a concept predicate.

<u>*Example:*</u> "Jim's brother, Sam, is coming to town today."

brother(Sam)

<u>*Example:*</u> "Jim met Sam, Jim's brother, today."

brother(Sam)

In the above sentences, we have a relation of an appositional modifier between the words "*brother*" and "*Sam*". This tells us that these words may share an *instance_of* relation between each other,

which leads us to generate the above facts from the appositional modifier. While generating the predicate it is important to look at who is acting as the governor and the dependent in the *appos* relation. Adding more concept predicates to the knowledge base opens the possibility of being able to infer better.

6.3.8.4 *Abbreviation Predicate*

Sometimes, texts contain abbreviation of words. It becomes very important for a system to understand the meaning of these abbreviations for querying. This is because the queries asked can sometimes contain short forms for organizations and projects rather than their long forms. Abbreviations can be detected by either doing pattern matching i.e. looking for the pattern "CONCEPT (ABBREVIATION)" or by using the *appos* dependency relation. The signature for the abbreviation predicate is as follows

_abbreviation (short_form, long_form)

In the above definition, the *short_form* refers to the abbreviation whereas the *long_form* refers to the abbreviated concept. This can be shown using the following example

Example: "The Apollo program was the third United_States human spaceflight program carried out by the National_Aeronautics_and_Space_Administration (NASA)"

_abbreviation (nasa, national_aeronautics_and_space_administration)

Capturing the abbreviations of various concepts in the text makes the inferring system more robust.

6.3.8.5 *Number Predicate*

The number predicate is used to identify numbers. Identifying a word to be a number helps in constraining the domain for certain queries. Other than that, for questions like "how many" and

"how much" it is more accurate to give a number as the answer. The signature for the number predicate can be given as

number (value)

Here the value is the actual number occurring in text. Finding if a word is a number can be either done using regex matching or by relying on the POS tagger and using the Cardinal Number (CD) POS Tag. The following sentence can be used as an example

Example: "Kenya covers 581,309 km2 and had a population of approximately 45 million people in July 2014."

number ('581,309').
number (45).
number (million).

6.4 Supplementary Knowledge Generation

The knowledge that we extract using the above sections comes directly from the input text. Apart from this, we can make use of various knowledge sources like WordNet to gain supplementary information about the concepts in the passage. Such supplementary knowledge plays an important role in common-sense reasoning. We as humans use this knowledge, often referred to as "Common World Knowledge", subconsciously while reasoning. This system uses the Hypernym relation from WordNet to build its ontology. The following sections dive deep into how these relations are used for reasoning.

6.4.1 Ontology Representation

The ontology section of the generated ASP program contains rules generated automatically from the Hypernym relations and rules used for word sense disambiguation. To generate these rules, we

use set patterns from answer-set programming like the preference pattern and the default reasoning pattern. Once rules and facts are generated from the sentences, we extract all the nouns from the passage for further processing. These nouns are used for building the ontology for the passage. Before building the ontology all these nouns are converted into concepts. In general the concept is represented using the following signature and we will continue using this signature to represent concepts all through the rest of this chapter.

concept (concept_instance, instance_sense)

In the above signature the *concept* represents the noun under consideration, *concept_instance* is the instance of the concept also a noun from the passage, and *instance_sense* is one of the lexical category mentioned in Chapter 5 Table 5.2. An example of a concept can be given as follows

lion (lion1, noun_animal)

Here, *lion1* is an instance of a lion, with the sense *noun_animal*. The following subsections will go through the procedure of generating the ontology rules from the concepts that are retrieved from the passage.

6.4.2 Hypernym Processing

Hypernyms can be used to infer about various properties and functions about concepts. Hypernyms make use of the generalization principle to transfer properties from more general concepts to their specific concepts. To do this, we first need to identify the concepts or entities from the passage and generate a hypernym graph from those concepts. These individual graphs can then be aggregated to join common base concepts. This aggregated graph is now ready to be converted into rules that transfer properties from one concept to the other. Let us consider the concept "*lion*" to demonstrate this process. From the figure shown in Chapter 5 Figure 5.2, we know that there

are 4 different senses of the word lion viz. animal, person, person, location. Let us explore the *noun.animal* sense of the concept *lion* and see how the hypernym rules can be generated. From the Figure 6.3, we can understand that if we are able to prove that some concept *lion1*, which is an instance of a lion, having the sense *noun_animal* is true i.e. the fact *lion (lion1, noun_animal)* holds then we can infer that "*lion1 is an animal*" [*animal(lion1, noun_tops)*] or "*lion1 is a living thing*" [*living_thing(lion1, noun_tops)*]. The Figure 6.3 just shows rules 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 6.3 Rules representing the animal branch of "lion"

Let us now discuss in detail how these rules can be generated for all the senses of a concept in the following sub-sections.

6.4.2.1 *Generate Hypernym Concept Graph*

The first step to generate 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 the different senses of the word as well. As an example, let us consider the concept of a "*car*". The word "*car*" has 5 distinct senses. A car can mean the following things

C.1 A motor vehicle with four wheels; usually propelled by an internal combustion engine

C.2 A wheeled vehicle adapted to the rails of railroad (rail car)

C.3 A conveyance for passengers or freight on a cable railway (cable car)

C.4 A compartment that is suspended from an airship and that carries personnel and the cargo and the power plant (gondola)

C.5 A compartment where passengers ride up and down (elevator car)

Here, sense 1 used for a motor vehicle is the most frequently used sense for the word "*car*". Now, as "*vehicle*" is a hypernym for sense *C.1* 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, mentioned below, vehicle also has other 3 meanings.

V.1 A conveyance that transports people or objects

V.2 A medium for the expression or achievement of something (A vehicle for political views)

V.3 A substance that facilitates the use of a drug or pigment or other material that is mixed with it

V.4 Any inanimate object that can transmit infectious agents from one person to another

From all the above meanings of vehicle only properties of sense *V.1* can be applied to that of car in sense *C.1*, thus also considering the sense of the entity while extracting knowledge from WordNet is 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.

The *GetHypernyms* procedure used in the Figure 6.4 is a recursive method that keeps on retrieving

Hypernyms for the concept till it reaches '*entity*', which is the root of all nouns in the WordNet

Ontology.

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

Figure 6.4 Pseudocode for Generating Hypernym Ontology Rules

In this procedure mentioned in Figure 6.5, hypernyms are selected from a set of all hypernyms that

either belong to the current sense or are hypernyms belonging to the "noun.Tops" group. The

current concept is then connected with its '*parentConcept'* and added to the concept bag. This is

continued until WordNet reaches entity or no more hypernyms are found. Using such a

conservative approach helps to not blow up the information obtained exponentially and helps

increase the speed of the reasoning systems by pruning the information at the source.

The algorithm below can be explained better with an example of the concept of "*lion*". As we

mentioned earlier, the word "*lion*" has 4 senses, if we track the 4 senses of the word to 'entity' we

get the following hypernym relations

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

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

```
procedure GetHypernyms(sense, parentConcept){
        // This function gets all the hypernyms from WordNet
        hypernyms = getHypernymRelation(sense)
        if (hypernyms contain an hypernym with 'sense')
                hypernym = hypernyms.get(sense);
        else if (hypernyms contain an hypernym with sense 'noun.Tops')
                hypernym = hypernyms.get(noun.Tops);
        end if

        Concept concept = conceptBag.get(hypernym);
        parentConcept.SetHypernym(sense, concept);
        conceptBag.add(hypernymConcept);

        GetHypernyms(sense, concept);
end procedure
```

Figure 6.5 Pseudocode for Extracting Hypernyms from WordNet

The next step is the aggregate all the duplicate entries in these hypernym links and create a common graph that represents the knowledge extracted from WordNet.

6.4.2.2 *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 the edges of the graph indicate

different senses. The edges in this graph are directed edges going from a more specific concept to a more general concept or the hypernym for a sense of the word. The graph in Figure 6.6 is formed after merging the 4 senses mentioned above.
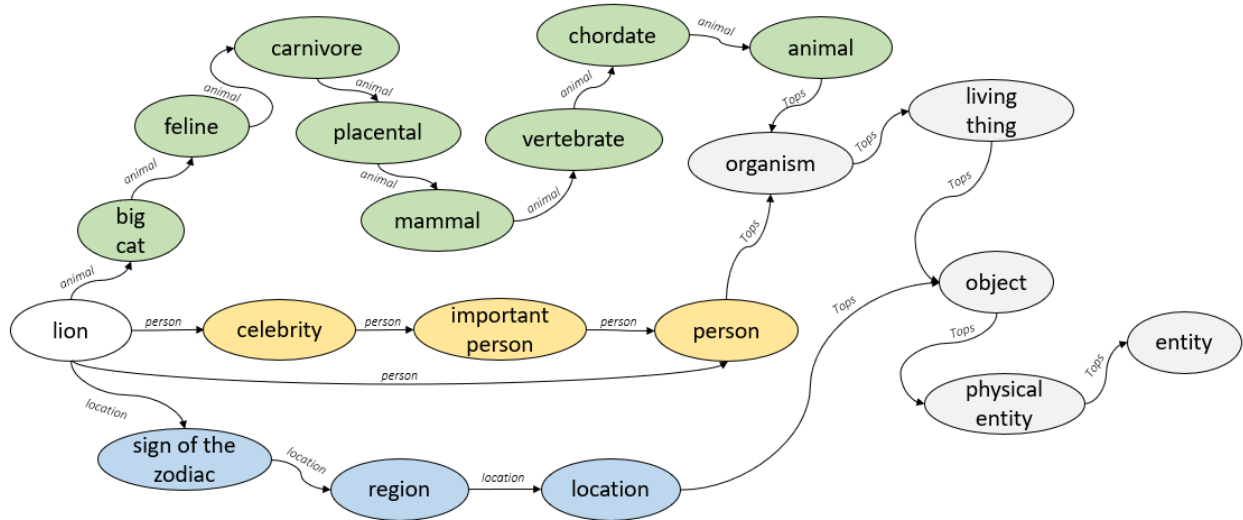
Figure 6.6 Concept Graph representing the Hypernym relation

From the Figure 6.6, we can see how the concepts are connected to each other under the hypernymy relation. In this graph we can see the closer the senses are to each other, the earlier they get merged into a higher-level concept. As an example, the senses of "*animal* and *person*" are closer to each other as compared to the senses "*person* and *location*" or "*animal* and *location*". Using this graph, we can generate rules directly based on certain pre-defined patterns.

6.4.2.3 *Generate Hypernym Rules*

There are two different ways to represent the hypernymy relationship using answer set programming. We will discuss both patterns along with their use cases. One of the patterns uses the '*class*' whereas the other one directly uses the '*instance*' of the class in the relationship. Let us consider the pattern using the *class* first.

The *is_a* relationship between the concepts of *lion* and *big_cat* and other concepts in its chain can

be represented as facts as follows

is_superclass (big_cat, lion),
is_superclass (feline, big_cat),
is_superclass (carnivore, feline), ... ,
is_superclass (entity, physical_entity)

We can then infer from these facts using the *superclass* predicate defined as follows

superclass (X, Y) :- is_superclass (X, Y).
superclass (X, Y) :- is_superclass (Z, Y), superclass (X, Z).

These kinds of modelling for the inheritance relationship is typically used in ASP. Although, this

model helps us to understand that the classes *lion* and *animal* are related with the relationship "*A*

lion is an animal", it is not able to infer anything about the instances of a lion in general. This kind

of representation helps when the domain for classes is infinite. In the current application it would

be better to represent the hypernymy relations with the help of the instances of the class instead as

the number of classes used are finite. We have seen an example of such a representation in Section

6.4.1. Let us consider a part of its simplified representation here as an example

big_cat (X) :- lion (X).
feline (X) :- big_cat (X).
...
animal (X) :- vertebrate (X).

Thus, given the sentence "*Sam is a lion*", if we can prove that the predicate "*lion (sam)*" holds then

using the above rules we can prove that "*animal (sam)*". Doing this kind of inference helps us

know more about "*sam*" and can help in complex tasks like question answering. When extracting

relationships from WordNet we need to deal with multiple senses. This introduced the concept of *sense* in the hypernym relations as well. Thus, the above relations become

big_cat (X, noun.animal) :- lion (X, noun.animal).
feline (X, noun.animal) :- big_cat (X, noun.animal).
...
animal (X, noun.tops) :- vertebrate (X, noun.animal).

Adding the sense term to the predicate makes the hypernym rule representation more accurate, as now only if we can prove that the instance of the *lion* belongs to the animal lexical category, then only we would be able to prove that the instance also belongs to the animal class. We will discuss how we implement the word sense disambiguation using these senses in the next section.

6.4.3 Word Sense Disambiguation

Simply put, word sense disambiguation is a task of selecting the best sense out of a collection of senses applicable to a concept. word sense disambiguation or WSD is one of the important sub tasks in most natural language tasks. When queried from WordNet we get a list of senses for a specific concept ordered from the most used to the least used. The following sections will discuss more on how ASP patterns are used to select the most correct sense for every concept using default and preference patterns.

6.4.3.1 *Representation of Word Senses*

Word senses are represented using two different logic patterns. The first of those will be discussed in this sub section and the later i.e. the preference pattern will be introduced in the next section. Using both these patterns in union, senses are selected for the various concepts in the text which activate their hypernym relations. The pattern discussed here, tries to prove a sense for a concept. Its template can be given as follows

$$c\ (X,\ s_i) :-\ c\ (X),$$
$$properties\_s_i\ (X),$$
$$not\ \text{-}c\ (X,\ s_i).$$

In the above template, we are trying to prove that "*X is an instance_of concept c with 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 use sense to the least used sense. Let us consider each term of the rule body and understand its importance. The first term given by '*c (X)*' is responsible to short circuit and fail the rule it 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.

These kinds of rules help the system prove that a certain sense applies if such information exists. In general, these kinds of rules are created so that users of the system are able to prove certain sense using external manually coded rules and properties. As an example, let us take the example of a *tree* concept. The WordNet gives 3 different senses for the *tree* concept as given in the Figure 6.7.

```
The noun tree has 3 senses (first 1 from tagged texts)

1. (107) tree -- (a tall perennial woody plant having a main trunk and branches forming a distinct
   elevated crown; includes both gymnosperms and angiosperms)
2. tree, tree diagram -- (a figure that branches from a single root; "genealogical tree")
3. Tree, Sir Herbert Beerbohm Tree -- (English actor and theatrical producer noted for his lavish
   productions of Shakespeare (1853-1917))
```

Figure 6.7 Entry for the concept of "tree" in WordNet

Here, *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).*

6.4.3.2 *Preference Patterns for senses*

From the above section we know that every concept has one or more senses. One of the principles that we as humans follow when performing word sense disambiguation in our day to day life is use a preference pattern over these senses and choose the most preferential pattern. This section describes, how to system models this preferential pattern to decide a default sense to every concept, if a sense is not provided or it can't be proved. The preferential pattern can be given as follows

Consider a concept c having three senses s_1, s_2 and s_3 ordered according to the frequency of their use from the most used to the least used. Then, we first assume the sense to be s_1, unless we know that s_1 is not the sense from some other source. Then we move on to the next sense s_2 unless we know that both s_1 and s_2 can't be the senses. And then finally we choose s_3. This process of elimination of senses and choosing senses according to preferences can be modeled in the following template.

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

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 pth sense of the concept c such that $1 > p > n$. Here n is the total number of senses of concept c. If $p = 1$ then the template omits the classical negation terms as follows

> $c\ (X,\ s_1) :- c\ (X),$
> $not\ -c\ (X,\ s_1),$
> $not\ c\ (X,\ s_2),\ not\ c\ (X,\ s_3),\ \ldots,\ not\ c\ (X,\ s_n).$

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

> $c\ (X,\ s_n) :- c\ (X),$
> $not\ -c\ (X,\ s_n),$
> $-c\ (X,\ s_1),\ -c\ (X,\ s_2),\ \ldots\ -c\ (X,\ s_{n-1}),$

In the above templates the combination of classical negation and negation as a failure play an important role in modelling the preferential pattern. Let us discuss the 4 terms in the general

template of this rule. The first term '*c (X)*' does the function of a guard clause as mentioned before. The second term given by '*not -c (X, s_p)*' act like a strong exception against the sense s_p. The third set of terms representing classical negation make sure that senses are applied according to preference i.e. the pth sense is only considered if all other senses before p have been proved to be false. The fourth set of terms representing negation as a failure, make sure that no other sense falling in the lower order can be proved to be valid before assuming s_p to be the correct sense. We can see this pattern in action with the *tree* concept as an example.

> *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 on sense for every concept in the text. The preferential pattern along with the property pattern mentioned in the above section help to solve the task of word sense disambiguation with the help of assumptions based on common world reasoning.

6.4.3.3 Sources for Word Sense Disambiguation

Word sense disambiguation is a very common problem and there are many statistical and dictionary-based solutions to it. This framework also makes it possible for such methods to be used as inputs to the program. Dictionary based algorithms like the Lesk Algorithm, that uses the context of the ambiguous word, or statistical classifiers like the Naïve Bayes classifier, that uses statistical methods, are effective methods that can provide an accurate sense of a concept to the program.

We can override the preferential pattern mentioned above using these methods by generating facts of the type '*concept (c, s)*' and placing them before the ontology patterns in the ASP execution. In the above fact, we know that *c* is the instance of the *concept* with the sense *s*. Due to the use of negation as failure terms in the pattern, if we are able to prove any one sense with the help of facts as above, then it can be applied thus avoiding the default pattern altogether.

CHAPTER 7

QUERY GENERATION

7.1 Overview

One of the parts in the system architecture is query generation from a natural language question. This chapter will deal with the idea behind and the steps required to query generation. First, we will go over the various schools of thought regarding classification of questions. Out of the many ways that questions can be classified, the system classifies the questions into categories based on the question word or the 'Wh-word' in the question. Once the question is posed to the system in natural language the system goes through some steps to convert it into a set of queries, that are ranked according to confidence. This chapter dives deep into the steps of conversion, discusses the various confidence classes and relaxation strategies to make the system more robust.

7.2 Question Classification

In general question classification can be based on many different of schools of thought. In this section we will learn about four techniques for question classification and will describe in detail three of them. The fourth question classification technique for questions based on the question word has been used by the system for query generation and will be discussed in detail in the next section. The different techniques for question classification can be broadly classified into the following

    A. Based on Bloom's Taxonomy

    B. Based on reading comprehensions

    C. Based on the purpose of the question

D. Based on the question word

We will discuss the first three methods in some detail in this section.

A. <u>Based on Blooms Taxonomy</u>

Blooms taxonomy provides an important framework to focus on the higher order of thinking. This taxonomy is divided into 6 different levels. These levels consist of keywords that define a critical level of thinking. These 6 levels are shown in Figure 7.1.
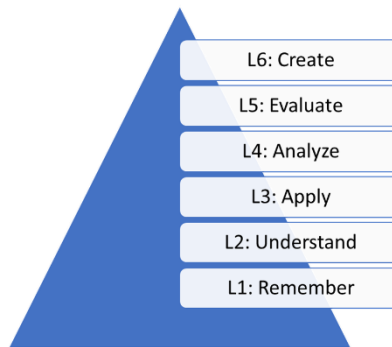
Figure 7.1 Stages in Blooms Taxonomy

These levels go from 1 to 6 in increasing order of difficulty. Level 1 deals with the task of remembering and tests the user's ability to memorize, recall terms, facts, and details without necessarily understanding the concept. Level 2 is based on comprehension and deals with understanding the concepts and being able to describe them in their own words or being able to summarize these concepts. Level 3 wants the users to be able to apply or transfer the knowledge gained in their own experiences or to a different context. Level 4 deals with analyzing concepts like finding out relations between multiple concepts, being able to differentiate and distinguish between two entities. Level 5 deals with evaluation or judging that asks users to form their own

opinions and make decisions. The final level i.e. Level 6 is based on creating or being able to create something completely new based on the ideas from various sources.

B. Based on reading comprehensions

According the reading comprehensions the questions are classified into 4 types. Literal questions, Cause and Effect questions, Inferential questions, and Vocabulary questions. Literal questions are fact-based question and the answers to those questions are present in the passage. Cause and Effect questions are why-based questions, that ask about the effects of an action or the cause of the action. Inferential questions require non-trivial inferential techniques to answer them. Vocabulary questions deal with grammar and require the user to understand the meaning of a word in the context to answer such questions.

C. Based on the purpose of the question

These types of questions are classified based on the purpose of the question. The various classes in this type of categorization as Factual, Convergent, Divergent, Evaluation and Combinations of all the classes. Factual questions are reasonably simple, straightforward questions usually based on obvious facts or awareness. These require the lowest form of cognition. Convergent questions can be answered with a finite level of accuracy. These may be at different levels of cognition but the answers to these questions lie in the passage in an indirect form. Convergent questions include questions on reasoning and contemplation. Divergent questions ask users to create different variations, alternate answers, and scenarios. Correctness of these kinds of questions are usually based upon context. Contrapositives and counterintuitive questions are examples of divergent questions. Evaluation questions require a highly sophisticated level of cognitive thinking and

emotional intelligence. Answers to such questions are analyzed at multiple levels before reaching to a conclusion.

7.3 Classification Based on Question Word

Questions that are classified according to their question word are of three types: Questions starting with a Wh-word, starting with how or have, and starting with "Is". Let us go over each of these types in detail

7.3.1 Starting with 'Wh-word'

These kinds of questions contain Wh-word like what, why, which, where, when and who. Each of these question types are looking for a different type of answer

7.3.1.1 *What Questions*

These kinds of questions ask for the determination of cause, judgement, or properties. Questions like what caused something to happen (antecedent) or what caused something to happen (consequence) belong to the 'what' questions. Other than this, 'what' questions also ask for the kind of properties of an object or concept. What question as generally asked when the answer domain set is unknown or is infinite and hence, are harder to answer as they can result in a lot of different type of answers.

e.g. What color is the bus? What caused the accident?

7.3.1.2 *Why Questions*

These kinds of questions are looking for a cause-effect relationship and ask about goals or explanations from the user. These types of questions may require a higher level of cognitions and can't be always answered using the paragraph.

e.g. Why did he leave? Why are you shivering?

7.3.1.3 *Which Questions*

Which questions ask for an identification of a person, place, object, or event. These kinds of questions may be answered with instances of concept that are asked along with the question. 'Which' questions indicate that the number of answer are finite, and the answer domain is completely known. If this is not the case, then the question changes to a 'What'.

e.g. Which box is damaged? Which boy is the tallest?

7.3.1.4 *Where Questions*

Where questions are one of the easier questions to answer. These questions are mainly looking for places or locations as answers.

e.g. Where do you live? Where should we go to eat?

7.3.1.5 *When Questions*

When questions are asked to know about the time of an event or a process.

e.g. When is your next meeting? When did you reach here?

7.3.1.6 *Who Questions*

Who questions ask for an instance of a person or a group of people.

e.g. Who called you here? Who owns this company?

7.3.2 Starting with 'How' or 'Have'

7.3.2.1 *How Questions*

How questions are posed to usually ask for a procedure or a quantity. How questions can be further extended to questions like "how much", "how many", "how far" and so on.

e.g. How much is price for cotton? How many people were injured?

7.3.2.2 *Have Questions*

Have questions ask for a yes or no response. These kinds of questions the modal verb as the main question word.

e.g. Have you finished the work? Have you turned off the lights?

7.3.3 Starting with 'Is'

Questions that start with Is usually ask for verification or permission from someone.

e.g. Is this machine working? Is this seat taken?

7.4 Steps in Query Generation

The query generation module proposes a method for converting natural language questions to answer set queries. As a question is also a sentence, it is processed in the same way that any other sentence would as mentioned before in the thesis. 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. In this section, we will go through the various rules that this module uses for query generation. The question understanding phase tries to analyze the question and uses this analysis for generating query predicates in the next phase. Currently this module is only built to deal with simple interrogative sentences and can be extended to deal with more complex questions.

7.4.1 Understanding Question

For analyzing the question, we try to get four kinds of information from the question viz. 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 case there are more than one Wh-words in the question then we select the one that is closest to the root of the sentence. As an

example, in the sentence, "*Who was climbing the tree?*" there is only one Wh-word, '*Who*', that is acting as the subject to the root of the sentence i.e. *climbing*. But in the sentence "*Who was shouting when I walked into the class?*" there are two Wh-words one in the main clause with the head *shouting* and one in the subordinate clause headed by *walked*. In this case the question word is the Wh-word in the main clause that is the closest to the root of the sentence given by *shouting*. 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 may have the copulas as their question word or modals their question words.

Once the question word is found then, we can set the question type with the help of the 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. Most of the Wh-word questions map to a question type but in case of other questions we need to look at the relations of the question word to infer more. As an example, in the question "*How much do these apples cost?*" after determining the question word to be '*How*', we need to look at the *advmod* relation to determine that the question type is *HOW_MUCH*. These categories can be extended to add more question types to improve the system.

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*. As an example, questions having the question type *WHAT* and *WHICH* are the ones that

require an answer type. For these questions, the answer word can be found by looking at the *det* relationship where the governor is the answer type and the Wh-word, or the question word is the dependent. As an example, in the question, "*What city is the Eifel tower located in?*", the expected answer is some city or a location. Similarly, the question "*Which flavor of ice cream do you like?*" expects a flavor name as the answer.

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*. The Table 7.1 talks about some expected answer types depending on question types.

Table 7.1 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

In the case of *WHAT* and *WHICH* question types, the answer type depends on the answer word that we extracted earlier. In the sentence, "*On what months is school closed?*", the base word for

84

the answer word is "*month*" which makes the answer type *MONTH*. Thus, for *WHAT* and *WHICH* questions different answer types like *YEAR, MONTH, DAY, NUMBER, TIME* etc. can be used. The answer types *SUBJECT* and *OBJECT* are also used with these question types. The *SUBJECT* answer type tells the system that the answer that it is looking is in the subject position of the main clause, whereas the *OBJECT* answer type directs the system to look for the object in the event. As an example, in the sentence "*What company owns YouTube?*" the answer word *company* lies in the subject relations from the main verb *owns*, thus indicating that the plausible answer lies in the subject of an event. Similarly, in the case of the sentence "*Whom are you calling?*" the *dobj* relation between the governor *calling* and the dependent Wh-word "*whom*" indicate that the answer lies in the modifier of some event. With the help of this basic analysis, we start generating the predicates for the query generation.

7.4.2 Generating Query Predicates

Using the information that we gathered above, we start generating predicate rules for all the words in the question. We will use predicates mentioned in Section 6.3 as a reference to generate predicates from the question. In general, verbs in the question participate in event predicates and property predicates whereas the nouns participate in property, possessive, and modifier predicates. All the predicates that are generated are combined to apply better constraints on the query. Let us study the *event* and *property* predicate generation separately. The rest of the predicates like the *possess*, *mod*, and named entity predicates are generated in a similar manner as mentioned in Section 6.3.

7.4.2.1 *Event Predicate*

The event predicate in general can be given as follows

event (*event_id*, *trigger_verb*, *actor*, *participant*)

Here, we do not know the *event_id* of the event in the question so we replace that using 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. The first is with the help of the *nsubj* relation, the second one is using the *nmod:agent* relation modeled using the property predicate with the *_by* preposition and the third one is finding the main clause subject for the subordinate clause trigger word. There may be other methods to search for the subject and can be easily added as patterns. Thus, for any question, we have these three variations mentioned above that help model the *event* predicate.

The three patterns discussed above for the event predicate triggered by the *verb* are given as follows

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. It is this index that helps in constraining the predicates while reasoning about different predicates clubbed together. 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'.

Here, we are using subject and object tags in the event predicates instead of using explicit atoms that are obtained from the question. This is done to make the querying process robust. We apply the subject and object constraints mentioned in the question on the generated *event* predicates with the help of the *_similar* predicate. With every subject and object tag generated we may have a *_similar* tag associated that would apply the respective constraint on the tag from the question. We will discuss how the *similar* predicate works in a later section in this chapter.

Consider the question "*what company owns the walt_disney?*" as an example for the *event* predicate generation. The question suggests that we have an object constraint '*walt_disney*' on the verb '*owns'* and the answer type is *SUBJECT,* so the following different query parts are generated.

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

7.4.2.2 *Property Predicate*

Property predicates are generated from verbs and nouns that are modified by nominal phrases in the sentence. A general property constraint has the following template

_property (event_id, modified_entity, preposition, modifier)

Here, like the event predicate we are unaware of the *event_id* and hence it will be 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). Consider the sentence "*On what streets is the ABC's headquarter located?*", that shows the *_property* predicate as a constraint.

_property (located, on, X2).

7.4.2.3 *Similar Predicate*

The similarity predicate tries to model 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 common-sense reasoning where we as humans make use of similar entities while reasoning. We sometimes use the last name of people to refer them instead of their entire name e.g. Usage of Einstein instead of Albert Einstein. This is also true for abbreviations, we use the short or the long forms of organizations interchangeably e.g. Usage of NASA instead of National Aeronautics and Space Administration or USA instead of United States of America. The *similar* predicate covers the following definitions of similarity.

a. Everything is similar to itself

_similar (X, X).

b. Abbreviations of concepts are similar to each other e.g. NYPD and New York Police Dept

_similar (X, Y) :- _abbreviation (X, Y).
_similar(X, Y) :- _abbreviation (Y, X).

c. Instances of concepts are similar to each other e.g. Mercedes is similar to a car

_similar(X, Y) :- _is (X, Y).

d. Similarity follows transitivity e.g. New York, NY, New York City and NYC are all interchangeable concepts that similar to each other.

_similar(X, Y) :- _similar(X, Z), _similar(Z, Y).

7.4.2.4 *Special Conditions and Predicates*

We described some special predicates in the Section 6.3.8 which included predicates like timespans, abbreviations etc. We can detect special question patterns and use these predicates to get better answers. Some of the question patterns that can be detected are as follows

"*What time was the Project Apollo started?*"

"*When did Nikola tesla die?*"

"*What is the short form of United States of America?*"

"*What is the full form of HTML?*"

"*What is the long form of WYSIWYG?*"

The time span predicate models time intervals for concepts. So, by detecting the patterns in the first two questions we can directly model the query using the *_start_date* and the *_end_date* predicate. Similarly, the rest of the questions show patterns ask for the abbreviations. We can detect the patterns in these questions and use the *abbreviation* predicate appropriately to answer them.

7.4.3 Applying Base Constraints

Base constraints are generated at the end after all the constraints from the question have been applied. These constraints refer to the constraint that depend on the answer type of the question. Answer types have already been described in Section 7.4.1, so here we will discuss the generation of some of the base constraints. Let us consider generation of the base constraints one by one.

For cases where the answer type is TIME, DAY, YEAR or MONTH 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)*

For Where-questions the answer type is PLACE, which can be constraint by applying the following constraint on the query

PLACE → *location (X_k) or location (X_k, noun_location)*

For questions asking for people or Who-questions the answer type is PERSON

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 as used in What and Which questions. Thus, in such cases the base constraint comes from the answer word itself. It is represented by

UNKNOWN → *concept (X_k)* e.g. *company (X_k)* or *city (X_k)*

7.4.4 Combining constraints

After generation of the constraints from Section 7.4.2 and the base constraints from Section 7.4.3 we combine all the constraints to create query. This can be best explained with examples, so let us take a few questions and go over how the query has been generated from the different parts of the question. Consider the following examples for query generation

1. In what borough of New York City is ABC headquartered?

_property (E2, borough, of, new_york_city), _property (E2, headquarter, _by, S2), _property (E2, headquarter, in, X2), _similar (abc, S2), event (E2, headquarter, _, O2), organization (abc), borough (X2, _).

_property (E2, borough, of, new_york_city), _property (E2, headquarter, in, X2), _relation (S2, E2, _clause), _similar (abc, S2), event (E2, headquarter, _, _), organization(abc), borough (X2, _).

_property (E2, borough, of, new_york_city), _property (E2, headquarter, in, X2), _similar (abc, S2), event (E2, headquarter, S2, O2), organization(abc), borough (X2, _).

In the above question we have three types of subject event constraints, property constraints enforcing the constraint "*borough of New York City*", named entity constraint forcing "*abc*" to be an organization and a base constraint forcing "*abc to be headquartered in a borough*".

2. Since what year did ABC stylize its logo, as 'ABC'?

_possess (abc, logo), _property (E2, stylize, _by, S2), _property (E2, stylize, since, X2), _similar (abc, S2), _similar (logo, O2), event (E2, stylize, _, O2), organization(abc), time(T2), year (T2, X2).

_possess (abc, logo), _property (E2, stylize, since, X2), _relation (S2, E2, _clause), _similar (abc, S2), event (E2, stylize, _, _), organization (abc), time (T2), year (T2, X2).

_possess (abc, logo), _property (E2, stylize, since, X2), _similar (abc, S2), _similar (logo, O2), event (E2, stylize, S2, O2), organization (abc), time (T2), year (T2, X2).

In the above question we have the similar constraints forcing the constraint that some entity like ABC has stylized an entity like logo. Apart from that, we have subject constraints, named entity constraint forcing "*abc*" to be an organization and a base constraint forcing "*styling to be since a year*".

3. When was Nikola Tesla born?

event (E2, bear, S2, O2), _similar (nikola_tesla, S2), property (E2, bear, on, X2), time(X2).

event (E2, bear, _, O2), _property (E2, bear, _by, S2), _similar (nikola_tesla, S2), property (E2, bear, on, X2), time(X2).

event (E2, bear, _, _), _relation (S2, E2, _clause), similar (nikola_tesla, S2), property (E2, bear, on, X2), time(X2).

_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'. All the above are examples of how queries can be generated from natural language questions.

7.5 Query Confidence Classes and Relaxing Constraints

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 better to make any question answering system robust enough, so that it fails gracefully. Once such attempt has been made in the query generation module by introducing the concept of confidence classes on the generated queries and by relaxing constraints on the queries to make the query more flexible. The queries that were generated in the previous section represent the most constraint queries. If an answer is found to one of those queries, then the system has the most confidence in the answer. In the absence of an answer the system starts removing constraints on the query and relaxing the query with a belief of obtaining some answer. Currently, queries have been divided into 4 confidence classes that range from most confident Class I to the least confident Class IV. Let us discuss how these classes are applied to queries

1. *Class I*: These kinds of queries have all the constraints mentioned in the natural language question. These may contain fact predicates, subordinate constraints, answer predicates and base constraints. This class has the highest confidence level.

2. *Class II*: These kinds of queries do not contain any fact predicates. Fact predicates are all the predicates in the queries that do not contain any variables. Such predicates come from the question or the named entity tagger and act as constraints.

3. *Class III*: These kinds of queries only contain the answer predicates and base constraints. This means that they don't contain fact predicates and any subordinate constraints that do not contain the answer tag (X_k).

4. *Class IV*: These queries only contain the base constraints. Answers from these queries are not very reliable as this class has the lowest confidence.

The current hierarchy of classes demonstrates one way of relaxing constraints, but there are many other ways in which constraints can be relaxed to get better answers. Let us demonstrate confidence classes and relaxed queries using the following example

Question: "*In what borough of New York City is ABC headquartered?*"

Class I Queries

_property (E2, borough, of, new_york_city), _property (E2, headquarter, _by, S2), _property (E2, headquarter, in, X2), _similar (abc, S2), borough (X2, _), event (E2, headquarter, _, O2), organization (abc).

_property (E2, borough, of, new_york_city), _property (E2, headquarter, in, X2), _relation (S2, E2, _clause), _similar (abc, S2), event (E2, headquarter, _, _), organization (abc), borough (X2, _).

_property (E2, borough, of, new_york_city), _property (E2, headquarter, in, X2), _similar (abc, S2), event (E2, headquarter, S2, O2), organization (abc), borough (X2, _).

93

Class II Queries

_property (E2, borough, of, new_york_city), _property (E2, headquarter, _by, S2), _property (E2, headquarter, in, X2), _similar (abc, S2), event (E2, headquarter, _, O2), borough (X2, _).
_property (E2, borough, of, new_york_city), _property (E2, headquarter, in, X2), _relation (S2, E2, _clause), _similar (abc, S2), event (E2, headquarter, _, _), borough (X2, _).
_property (E2, borough, of, new_york_city), _property (E2, headquarter, in, X2), _similar (abc, S2), event (E2, headquarter, S2, O2), borough (X2, _).

Class III Queries

_property (E2, headquarter, in, X2), borough (X2, _).

Class IV Queries

borough (X2, _).

In the above example, we see that Class I queries are generated using the rules defined in Section 7.4. From these queries Class II queries can be generated by dropping the fact predicate "*organization (abc)*". Class III queries drop fact predicates as well as subordinate constraints, the predicates that don't contain the answer tag, e.g. "*_property (E2, borough, of, new_york_city)*". The final confidence class drops all the predicates except the base constraints i.e. here we also drop certain answer predicates that aren't base constraints like "*_property (E2, headquarter, in, X2)*".

CHAPTER 8

APPLICATIONS IN QUESTION ANSWERING

8.1 Overview

Knowledge Representation is a task in Artificial Intelligence that proves useful in solving other tasks in AI like summarization, question answering, automatic reasoning and justification and many more. Here, we use question answering as a task to show the efficiency of using answer set programming for representing knowledge. This chapter deals with how the Stanford Question Answering Dataset or SQuAD was used in developing the system and any results that followed it.

8.2 SQuAD Dataset

The SQuAD Dataset contains more than 100,000 reading comprehensions along with question and answers on those reading passages. SQuAD dataset uses the top 500+ articles from English Wikipedia. These articles are then divided into paragraphs. The Dev Set v1.1 for the SQuAD Dataset was used 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. The structure of one of the articles in the dataset can be given as follows

```
data:
        ##article list start
        "title": <article_title>,
        "paragraphs":
                ##paragraph list start
                "context": <paragraph_content>,
                "qas":
                        ##questions list start
                        "answers":
                                ##answers list start
                                "answer_start": <word index in paragraph>,
                                "text": <answer text from paragraph>
```

```
                        …
                ##answers list end
            "question": <question text>,
            "id": <question_id>
            …
            ##questions list end
        …
        ##paragraph list end
    …
    ##article list end>
```
Thus, from the above structure we know that for each article in the article list there are a set of paragraphs each having its own set of questions and answers. One of the examples of a paragraph on Nikola Tesla and a few questions related to it are as follows

Paragraph: "Nikola Tesla (10 July 1856 - 7 January 1943) was a Serbian American inventor, electrical engineer, mechanical engineer, physicist, and futurist best known for his contributions to the design of the modern alternating current (AC) electricity supply system."

Questions and Answers

1. In what year was Nikola Tesla born?

A. [1856, 1856, 1856]

2. What was Nikola Tesla's ethnicity?

A. [Serbian, Serbian, Serbian]

A set of crowd workers generated the questions and the set of answers. To make the evaluation from the dataset more robust, each question was also provided with at least two additional answers. For finding the secondary answers the users were asked to provide an answer that was closest to the correct answer from the paragraph.

8.3 Categorization of Articles

Out of the 48 different articles in the SQuAD dev set, I choose 20 articles from different domains to help build the system. I tried to take articles such that a lot of different domains were covered using the articles. We can roughly categorize these articles into 5 different categories: People articles, Scientific articles, Project/Event articles, Region articles and Misc. articles. People articles are those that generally talk about either a single person or a group of people like communities. These are articles contain knowledge about the birth dates/ death dates of people, various places that the people lived in or worked from. As, in general these articles talk about famous people or celebrities they sometimes talk about their inventions, discoveries, or their special traits. Some of the people articles include ones on Nikola Tesla, Martin Luther, The Normans etc. Scientific articles talk about scientific concepts like naturally occurring elements, scientific processes, and different streams of science. Such articles are filled with scientific terms and their names. Sometimes it includes the abbreviations of processes or biological names or scientific symbols of these concepts that can be modelled using the *similar* predicate. Some of the scientific articles used included Oxygen, Chloroplasts, Geology, Steam Engines etc. Project or Event articles describe about the various projects undertaken or events held by organizations. Such articles may contain information about other related projects or events by the organization or may talk about results of the project or its mission. Articles about Super Bowl 50 and Project Apollo were of this type. Region articles describe various regions all over the world. These articles contain measurements for the region, the different kinds of flora and fauna found in the region and its geographical specifications. Thus, such articles help the system model and learn about the different

measurement scales and their equivalence. All the article categories and types mentioned above teach something new to the system and help the system model different kinds of information.

8.4 Results

Using the 20 different articles mentioned above the ASP program was generated on one paragraph from each article. Then, queries were generated manually 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. This result can be viewed in terms of the Table 8.1. The result shows that almost 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 queries generated manually for the questions are very similar to the original question and convey the same meaning.

Some examples for the same can be given as follows.

1. What day was the game played on?

> *event (E1, play, S1, O1), _similar (game, O1), _property (E1, play, on, T), day (T, X1), time (T).* OR
> *event (E1, play, S1, O1), _property (E1, play, on, X1), time (X1).*

The above question can be converted into two queries as mentioned above, 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 query is semantically very close to the question asked.

2. What city did Super Bowl 50 take place in?

> *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.

3. What team was the NFC's champion?

   _possess (O, X), _similar (nfc, O), team (X).

In the above query we are trying to find "*something that NFC possesses that is a team*".

Table 8.1 Results for Question answering

No	Article	Result		%
		Correct	Question Count	
1	ABC	5	5	100.00
2	Amazon Rainforest	12	14	85.71
3	Apollo	4	5	80.00
4	Chloroplasts	4	5	80.00
5	Computational Complexity	3	3	100.00
6	Ctenophora	9	12	75.00
7	European Union Law	13	13	100.00
8	Genghis Khan	3	5	60.00
9	Geology	4	5	80.00
10	Immune System	13	15	86.67
11	Kenya	5	5	100.00
12	Martin Luther	2	5	40.00
13	Nikola Tesla	6	7	85.71
14	Normans	4	5	80.00
15	Oxygen	8	15	53.33
16	Rhine	5	8	62.50
17	Southern California	3	5	60.00
18	Steam Engine	4	5	80.00
19	Super Bowl 50	25	29	86.21
20	Warsaw	3	5	60.00
Total		135	171	78.95%
Average Result		77.76%		

Thus, from the above examples we can see that automatic generation of query from questions is not too far out of reach. A method to do so has also been proposed in Chapter 7 of this thesis. One of the main pain points which 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 world knowledge, but the machine fails to do so due to absence of enough digital semantic resources.

CHAPTER 9

FUTURE WORK

9.1 Overview

In this thesis we described a system that automatically converts text to knowledge represented as an answer set program and augments this knowledge with common sense knowledge imported from ontology resources such as WordNet. We showed how this resulting knowledge can be used for developing an automated question-answering system. The system has shown good performance, however, there are a lot of avenues for further developing this system, improving its accuracy, and supporting other applications. This chapter will go over some of the enhancements that can be done in the future and will elaborate on how they can be achieved.

9.2 Future Work

The following points will touch upon a few features that can be added into the system as a whole. These would include improvements in both the knowledge generation modules as well as the query generation module.

9.2.1 Support for Temporal reasoning

Temporal reasoning is one of the important aspects of human reasoning. It is using temporal reasoning or reasoning with time that humans take decisions. One of the important examples of temporal reasoning is one's daily schedule which humans use to navigate and plan through the day. Every event that happens or is part of one's life is connected to other events with a temporal relation, i.e., either two events happen together or separately. If events happen separately then one

event happens before the other or vice versa. Thus, we can also add another predicate indicating tying events to a global time. This could look something like

happensAt (*event_id*, *time_id*)

Using such predicates would give more insight into how the events happens and will be able to model an important feature that of time. We could use this kind of modelling technique to also define the time at which an event no longer applies. This would make reasoning more flexible and non-monotonic.

9.2.2 Understanding cause effect relationship

One of the important relationships between events is the causal relationship. This relationship defines which event was caused by which other event or which event was the effect of which event. Cause-Effect relations between events play an important role in justification of an event or reasoning about predictions. Currently, the knowledge generation system does not model cause and effects due to unavailability of good resources to detect such relations. But in the future, cause and effect relations can be modelled using *event_id* as follows

cause (event_id$_1$, event_id$_2$).
effect (X, Y) :- cause (Y, X)

This also gives rise to a transitivity relationship in causes and effect, which can be modelled as

cause (X, Y) :- cause (X, Z), cause (Z, Y).

Modelling the cause and effect relations also allow the query generation system to generate queries for the Why-question.

9.2.3 Modelling more knowledge patterns

With the help of the 20 paragraphs from the 20 different articles, we were able to learn about some special patterns that appear in text like time spans, appositional modifiers, etc. With the help of diverse articles and writing styles, more natural language patterns can be captured by the system that humans implicitly use. This would allow the system to understand text in a better fashion.

9.2.4 Modelling semantic similarity between parts of speech

Currently the _similar predicate tries to capture if any two concepts are similar to each other. However, this can be explicitly obtained from WordNet using the synonymy relation between concepts. Synonyms can be between nouns, verbs, adjectives etc. There are some other similarity measures like the path-length based similarity, information content similarity, and Resnik similarity which help in modelling this predicate. Thus, with the help of WordNet or any other trusted source, relations like synonymy and antonymy can be modelled which will make the system more robust.

9.2.5 Adding other information sources

Currently, we are using WordNet and the input textual passage as our main information sources. But there are many other sources of information that we can make use of to answer questions. Ontological databases like YAGO, a joint project of the Max Planck Institute for Informatics and the Telecom ParisTech University, for example, can help in automatically generating knowledge about common concepts, which will indirectly make reasoning more effective and accurate.

9.2.6 Marking facts with a confidence metric

In an environment with multiple sources of information, we need to introduce the concept of confidence on the facts that are generated from these sources. This will help the system to choose

the facts with the most confidence in the event of a conflict. Consider that we are reasoning about a medical diagnosis depending upon certain symptoms. Two sources back our knowledge, a medical journal and Wikipedia. In this case, we should give more weightage to the facts generated in the medical journal as compared to those generated from Wikipedia. This can be modelled by adding a confidence measure to facts relying on the source and the context.

9.2.7 Modelling other common-sense reasoning principles

In this thesis, we have seen how default senses allocation, generalization of properties, and similarity of concepts is modelled. Similar to these common-sense principles, there are other weaker principles that can be modelled from common-sense reasoning. Some of them include modelling the hyponym relation between concepts, which states that if we know that some concept v is a vehicle then v may be similar to a car. The hyponym can be modeled within the ASP framework using a combination of classical negation and negation as failure.

9.2.8 Relaxing of Query Constraints

While doing query generation, we used the concept of relaxing constraints to make the system robust. For this we used the concepts of confidence classes and removed certain kinds of predicates from the query depending on the confidence class. Removing predicates from the query is one way of relaxing constraints, but we can also relax constraints by ignoring certain terms in the predicate instead of directly removing to constraint. By doing this, we still obtain a query that has a higher confidence value than the query obtained by completely removing the constraint.

9.2.9 Answering complex questions

With the help of various information sources and the text, we were able to answer questions that were more factual in nature. These kinds of questions belong in the first level of blooms taxonomy.

In the future, we can attempt to go higher in the taxonomy by adding knowledge about more complex tasks like summarization, critiquing, application etc.

CHAPTER 10

CONCLUSION

10.1 Overview

In this thesis we have proposed a two-part system, one to represent and the other to query knowledge automatically from natural language text. The knowledge generation system proposed in Chapter 6, defines a generic calculus to represent knowledge from passages. It defines the structure and the meaning of various predicates that are generic enough to represent almost any sentence structure in English. Using the rules generated from the passage we are efficiently able to represent most of the knowledge from text. Knowledge present in text is both implicit as well as explicit. Explicit knowledge can be extracted with the help of grammatical structure, but implicit knowledge is based on context and needs pre-defined patterns to extract. This system models some of these knowledge patterns to model implicit knowledge like time spans, appositional modifiers and so on.

Apart from the passage, we also made use of WordNet as a source of information. With the help of WordNet, we were able to generate an ontology that helps gather more information about the entities in the passage. For using the synsets from WordNet we need to use some approach for word sense disambiguation. In this thesis, we have used the preferential pattern to disambiguate between senses and apply the sense with the most frequent use. The preferential pattern defined in Chapter 6 is flexible enough to take inputs from external word sense disambiguation (WSD) modules, if present. The hypernym relation, the preferential pattern for WSD, etc., are all patterns that humans use for common-sense reasoning. The system we developed helps model a few of these patterns for reasoning.

Once knowledge has been represented using our custom calculus, we need to make sure that queries posed to the system adhere to this calculus for accurate results. Thus, Chapter 7 describes a proposed method to convert questions to a set of ordered ASP queries. Using this proposed method, we can answer simple questions from the SQuAD dataset. Not only can the system give answers to the queries but can also provide alternate solutions to the question if a solution was rejected. The question generation system is developed to be robust in nature. Once a question is converted to the most constrained query, some of the predicates in the query are relaxed using certain techniques mentioned in Chapter 7. This gives the system a chance to find solutions, even if it is just a guess. One of the important advantages of using ASP and the s(ASP) system as a system for knowledge representation is that we can get an interpretable justification for any answer provided. With the help of justification trees provided by the s(ASP) system we can give a detailed explanation for the computed answer. The availability of the s(ASP) query-driven answer set programming system is crucial to the success of our question answering system. The s(ASP) system's query driven nature results in three major advantages: (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.

We discussed about the various features and benefits about the system, now let us discuss about a few hurdles faced while building the system. In Chapter 5 we mentioned about the various natural language resources used to support the system. Although these tools give accurate results in most cases there are some sentences that they fail to parse accurately. This leads to erroneous results. One way to solve this problem is to use multiple tools at the same time. As an example, the system

uses both a POS Tagger as well as the Stanford Dependency Parser. Sometimes the POS tagger wrongly tags some words which leads to wrong assumptions and wrong interpretations of the sentence. But sometimes these tags can be overridden using the findings of the dependency parser. Humans use the similarity relation a lot when reasoning. As mentioned earlier, we sometimes use last names of people interchangeably with their full name while referring to them. All these implicit connections that we make as humans adversely affects the question answering accuracy of the system. To solve this problem, we need to have noun or verb similarity metric that tells us about all the concepts that we can use interchangeably.

Verbs form one of the important parts of a sentence. They are generally the head words of sentences or clauses. To gain a complete understanding of a sentence we need to understand the exact meaning of a verb in the context of the sentence. Such information about verbs is currently not present in digital format. This makes it difficult to model human thinking completely since we, as humans, have a clear understanding of what verbs mean in different contexts of a sentence. This problem can be overcome by adding custom rules and knowledge about verbs manually. It was seen that after adding generic and reusable rules for describing verbs and their meanings, the system was able to answer questions better. Although currently adding all such information about verbs is a monumental task, even incremental steps in adding such information makes a significant impact on the system. One other problem faced by the system is that it is not able to answer questions that require significant cognitive abilities like differentiating between concepts, applying knowledge to other tasks, etc. All these problems can be solved by adding a set of rules that give the system the ability to perform these tasks with the information it has extracted from the passage.

BIBLIOGRAPHY

1. C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press. 2003

2. Knowledge Representation, Reasoning, and the Design of Intelligent Agents (The Answer-Set Programming Approach), Micheal Gelfond, Yulia Kahl

3. GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. 2008. A User's Guide to gringo, clasp, clingo, and iclingo.

4. Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, Giorgio Terracina: The Disjunctive Datalog System DLV. Datalog 2010: 282-301

5. R. Kowalski and M. Sergot (1986) "A Logic-Based Calculus of Events," New Generation Computing 4: 67–95.

6. Kyle Marple, Elmer Salazar, Gopal Gupta, Computing Stable Models of Normal Logic Programs Without Grounding

7. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes), pages 1070–1080. MIT Press, 1988.

8. F. Lin. Situation Calculus. Chapter 16 in Handbook of Knowledge Representation

9. Edited by F. van Harmelen, V. Lifschitz and B. Porter. 2008. pp. 649-671.

10. Cyc, Wikipedia Page, https://en.wikipedia.org/wiki/Cyc

11. BREWKA, G., EITER, T., AND TRUSZCZY´N SKI, M. 2011. Answer Set Programming at a Glance. Communications of the ACM 54, 12, 92–103.

12. Zhuo Chen, Kyle Marple, Elmer Salazar, Gopal Gupta, Lakshman Tamil: A Physician Advisory System for Chronic Heart Failure Management Based on Knowledge Patterns

13. Dissertation – Goal Directed Answer Set Programming, Kyle Marple

14. MARPLE, K., SALAZAR, E., CHEN, Z., AND GUPTA, G. 2017a. The s(ASP) Predicate Answer Set Programming System. The Association for Logic Programming Newsletter. April 2017.

15. Answer-Set Programming – A Primer, Thomas Eiter, Giovambattista Ianni,and Thomas Krennwallner

16. Stanford CoreNLP – Natural language software, https://stanfordnlp.github.io/CoreNLP/

17. Manning, Christopher D., Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55-60.

18. Kristina Toutanova and Christopher D. Manning. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp. 63-70.

19. Kristina Toutanova, Dan Klein, Christopher Manning, and Yoram Singer. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In Proceedings of HLT-NAACL 2003, pp. 252-259.

20. Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005), pp. 363-370. http://nlp.stanford.edu/~manning/papers/gibbscrf3.pdf

21. Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser Using Neural Networks. In Proceedings of EMNLP 2014.

22. George A. Miller (1995). WordNet: A Lexical Database for English. Communications of the ACM Vol. 38, No. 11: 39-41.

23. Christiane Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.

24. Finlayson, Mark Alan (2014) Java Libraries for Accessing the Princeton Wordnet: Comparison and Evaluation. In H. Orav, C. Fellbaum, & P. Vossen (Eds.), Proceedings of the 7th International Global WordNet Conference (GWC 2014) (pp. 78-85). Tartu, Estonia.

25. Test Driven Development, https://martinfowler.com/bliki/TestDrivenDevelopment.html

26. Test Driven Development: By Example: Kent Beck

27. Penn Treebank English POS tag set: 1993Computational Linguistics article in PDF, AMALGAM page, Aoife Cahill's list.

28. Alphabetical list of part-of-speech tags used in the Penn Treebank Project, https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

29. A Maximum Entropy Model for Part-Of-Speech Tagging, Adwait Ratnaparkhi

30. Universal Dependencies, http://universaldependencies.org/u/dep/

31. WordNet A Lexical Database for English, https://wordnet.princeton.edu/

32. List of WordNet lexicographer file names and numbers
https://wordnet.princeton.edu/documentation/lexnames5wn

33. Different Types of Questions based on Bloom's Taxonomy,
http://faculty.academyart.edu/faculty/teaching-topics/teaching-curriculum/enhancing-teacher-student-interaction/different-types-questions-blooms-taxonomy.html

34. Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, Percy Liang. SQuAD: 100,000+ Questions for Machine Comprehension of Text

BIOGRAPHICAL SKETCH

Dhruva Pendharkar was born in Pune, Maharashtra, India. After completing his school education in Pune, he joined the Bachelor of Engineering program at the University of Pune. He completed his B.E. Computer Engineering degree in 2013 with distinction. After completing his bachelor's degree, Dhruva was employed at Hyderabad at the Microsoft India Development Centre. Here he worked on the Azure Application Platform Team and later the Networking Team. After working for 3 years at Microsoft, Dhruva decided that he should pursue a master's degree to enrich his knowledge about Natural Language Processing and Automated Reasoning. In the Fall of 2016, he joined the Master of Science in Computer Science program at the University of Texas at Dallas.

<div align="center">

CURRICULUM VITAE

DHRUVA RAJAN PENDHARKAR

</div>

LinkedIn Profile: www.linkedin.com/in/dhruva-pendharkar
E-Mail: dhruva.pendharkar@gmail.com

EDUCATION

Master of Science in Computer Science (GPA: 4.0)   *Expected May 2018*
Thesis: An Answer Set Programming Approach to Representing and Querying Textual Knowledge
The University of Texas at Dallas (UTDallas), Richardson, TX
Bachelor of Engineering, Computer Engineering, *May 2013 (Distinction)*
University of Pune, Pune, Maharashtra, India

PROFESSIONAL EXPERIENCE

Software Development Intern Uber Technologies
Team: Driver Growth – Viral Channels Team, *May 2017 – Aug 2017*
Product: *Signups Incentives Platform*

Software Development Engineer at Microsoft India Development Center
Team: Azure Application Platform Team, *Jul 2013 – Sep 2014*
Product**:** *Microsoft Workflow Services*
(*https://msdn.microsoft.com/en-us/library/dd456788(v=vs.110).aspx*)
Product: *Windows Azure BizTalk Services*
(*https://azure.microsoft.com/en-in/services/biztalk-services/*)
Product: *Azure App Service*
(https://azure.microsoft.com/en-us/services/app-service/)

Software Development Engineer at Microsoft India Development Center
Team: Networking, *Sep 2014 – Jul 2016*
Product: *Network Performance Monitor Service*
*(*https://blogs.technet.microsoft.com/msoms/2016/07/27/introducing-oms-network-performance-monitor/)

RESEARCH INTERESTS

My research interests mainly include Artificial Intelligence, Natural Language Processing, and Answer Set Programming. Apart from that I also like to read about Algorithm Design, and Programming Languages.

TECHNICAL SKILLS

Programming Languages: C, C++, C#, Java, Python, GO
Web: HTML, JSP, CSS, JavaScript, Knockout Framework
Databases: SQL, Oracle, MongoDB, MS Access, Schemaless

LANGUAGUES

I am fluent in my mother tongue which is Marathi. Apart from that I am well versed with both English and Hindi. I also picked up German as a hobby and am quite comfortable in having a conversation in it.