6-22-2020

Translating Counting Problems into Computable Language Expressions

Zach Prescott
University of Missouri-St. Louis, zjpnr4@umsystem.edu

Translating Counting Problems into Computable Language Expressions

Zachary J. Prescott

A Thesis

Submitted to The Graduate School of the

University of Missouri-St. Louis

in partial fulfillment of the requirements for the degree

Master of Science in Computer Science

August 2020

Advisory Committee

He, Wenje, Ph.D.

Chairperson

Badri, Adhikari, Ph.D.

Chakraborty, Uday, Ph.D.

Translating Counting Problems into Computable Language Expressions

Section 1: Introduction

The field of Natural Language Processing (NLP) is an increasingly popular area of research in the Artificial Intelligence (AI) field. There are many significantly difficult problems in this area, such as natural language understanding, universal language modelling, etc., (see [6]). In this thesis, we target a subproblem of NLP: Mathematical Language Processing (MLP). This subfield is important because it is essential for developing a powerful AI mathematics problem solver. Additionally, many complex issues that most NLP projects require, such as understanding idioms, politics, culture background, etc., can be avoided in MLP. Although the context used in MLP is greatly reduced compared to generalized NLP, it is still a challenging problem. Since there are so many types of mathematics problems in a large number of areas, we will not target the general mathematics problems at this point. Thus, in this thesis, we restrict our focus to a special class of mathematics problems – *Counting Problems*.

There are several research inquiries in this area. Some immediate examples that come to mind are the ANALOGY program written by Evans [7] and ARIS, which solves arithmetic word problems with verb categorization [4]. These papers mainly focus on a special type of mathematics problem, *Word Problems*, and most of them deal with relatively simple word problems. One of the best programs in this area that is currently available is Wolfram Alpha, an application that provides a large array of mathematical functionalities including the ability to solve word problems. Unfortunately, despite the advanced mathematical capabilities that Wolfram Alpha possesses, it is somewhat lacking in the domain of NLP, and when faced with a college-level problem, fails to

extrapolate vital components of the problem. For example, it cannot solve the first counting problem in our database. These approaches do two things together: 1) Understand the meaning of those mathematics problems; 2) Solve those problems based on that understanding. There is a big drawback with this approach. Currently, the AI problem solver is weak and non-robust, as it cannot handle relatively hard mathematics problems. If an AI program that intends to understand the meaning of a mathematics problem must worry about the problem-solving part, then a large variety of mathematics problems cannot be included in the MLP program. For example, the famous *Goldbach Conjecture* can address this point.

Goldbach Conjecture: Every even integer greater than two could be written as the sum of two primes.

This problem demonstrates a problem for an AI to work with: it is extremely hard to solve, but it is very easy to understand the meaning. A better approach for MLP is *decoupling* the tasks of understanding the meaning of a mathematics problem and solving the problem, so that the MLP program can focus on understanding without worrying about the problem-solving part. Based on this, we will separate a typical AI mathematics problem solver into two independent problems: 1) Translate a mathematics problem from a human language to a computer-friendly language; 2) Solve the mathematics problem automatically by an AI. In this thesis, we will work on the first problem: Developing an AI program that can translate a mathematics counting problem into some computer-friendly language.

Section 2: Framework of Mathematical Language Translation

Part I.

We will introduce our model for problem translation into our mathematical language. Before discussing the specifics of the translated mathematical language, we would like to first cover our goals in developing the language. Consider the following problem:

> *How many integers from 1 through 999,999 contains each of the digits 1, 2, and 3 at least once?*

There are four qualities that we would like to discuss in our translation.

First, the translation is intended to be computation friendly. This project primarily aims to take a math problem in an unstructured, natural language and reconvey it in a format that a computer could more easily understand. As an example, the simple phrase, "How many integers," implies to the AI that it must return some quantity specifying a number of integers. This simple notion can be extrapolated as follows:

[#enum#] := {#int#}

(This representation will be expanded on in later sections.) By representing the object to be counted in this way – an idea we call the *enumerator* – the AI will be able to draw upon any functions, mathematical rules, and/or properties, which apply specifically to integers.

Second, the translation must be accurate. By restricting ourselves to counting problems, we may take a set-oriented approach to solving them. This means that virtually every problem we work with can be approached by defining a set, filling that set with every instance of the object to be counted, and then returning the cardinality of that set. As a student might similarly brute force a counting problem, likewise the result that the AI returns should be the solution. For this to work, the AI must be able to precisely translate the problem into our new language.

For example, when the AI attempts to solve the listed problem, it will first define a set. Next, it will fill that set with every integer from 1 through 999,999 that contains each of the digits 1, 2, and 3, at least once. Finally, it will return the

cardinality of that set. How the AI fills the set is beyond the scope of this thesis; all that matters is making the AI translate the problem into a format that it can work with.

Third, the translation must also be unambiguous. In natural languages, there are often phrases that require context to be understood. For example, look at the phrase, “Go down the road a ways and turn right.” The true meaning of this instruction depends entirely on the interpretation of, “a ways,” which does not specify any distance or landmarks to watch for and may lead the reader to the incorrect destination. However, the message itself is not necessarily incorrect. Whereas the quality of accuracy was related to ensuring the correct meaning was specified, the quality of being unambiguous entails there being no other possible meaning for our translation, or no other way that the components of the translation can be understood.

Lastly, the translation should be concise. This means that any information which is not critical to the solution will not appear in the translated work. In the problem introduced at the beginning of this section, there is little text data that is noncritical or without use. Consider a different problem:

> James and a friend played 14 games of tic-tac-toe one night. In how many ways can James end the night with eight wins, two ties, and four losses?

One might consider 8 Ws, 2 Ts, and 4 Ls, then count all the different ways that these 14 symbols can be arranged in order to reach the final answer. However, phrases such as “James and a friend” or “one night” do not provide the AI with any special constraints to which to adhere, nor any extra information regarding how the individual elements of the set should be organized. The translated text will not include this extraneous information. Conversely, the number 14 is

significant, as are the quantities of wins, losses, and ties. The translated text must parse this information from the input.

Part II

In this part, we would like to go more in depth into how the process of problem translation takes place.

Natural Language Processing

Because the user input is in the form of textual data, we will be using natural language processing techniques to determine our translation. Natural language processing is broadly defined as the automatic manipulation of natural language, like speech and text, by software. Although this area of study draws on only half of a century's worth of research, there is a broad process to which we will attempt to conform.

There are said to be two basic approaches to NLP. One is through rule-based processing, in which every rule and step in translation was written by a human being. The other approach is machine-learning processing; this draws on a large corpus of text data in order to solve the given problem. Our project takes on a rule-based approach, as our focus on counting problems and using a set-oriented perspective should simplify the problem enough that we will not have to draw too heavily on the work of linguistics, as that would cause us to stray from our original problem. Furthermore, although it is a common pitfall of rule-based approaches, we do not expect our system to grow overly complicated.

There is a third way that one might approach NLP called the hybrid approach, which uses both methods. Though it is true that we intend to use some amount of machine learning to help our knowledge base grow, this will be largely overshadowed by the usage of rules and prewritten patterns to recognize key phrases and words similar to the model in Wang et al. [8]. These will help us determine the important information.

In terms of the processing, the first step in building a translation is tokenization, or the splitting of the input text into individual words or phrases. Unfortunately for our purposes, this is not as straightforward as breaking a sentence into words, removing punctuation, and then placing the words into a list.

"How many integers from 1 through 999,999 contains each of the digits 1, 2, and 3 at least once?"

…

[*"How", "many", "integers", "from", "1", "through", "999999", "contains", "each", "of", "the", "digits", "1", "2", "and", "3", "at", "least", "once"*]

There are several things wrong with this approach. For one, the phrase "how many" is significant to understanding the problem because the noun immediately following the phrase gives us the context into what we are trying to count, such as what processes are acceptable on it. It would be far more efficient and logical for those two words to be processed as a single unit. Another instance is how the range in a problem might be specified; although a basic tokenizer splits this into four separate words, the phrase "from 1 through 999,999" conveys a single piece of information. In fact, the phrase "from [some number] through [some number]" is a commonly reproduced pattern; we intend to create a database of patterns that the AI can compare the original text with in order to formulate the text in logical, processable tokens.

In short, an off-the-shelf tokenizer will not suit our needs. This observation is magnified by our intention to use a database to interface with our AI (expanded on later). Below, the problem is reorganized into more logical tokens:

"How many integers from 1 through 999,999 contains each of the digits 1, 2, and 3 at least once?"

…

[*"How many", "integers", "from 1 through 999999", "contains", "each of", "digits", "1, 2, and 3", "at least once"*]

We next consider stemming. Stemming involves simplifying a word to its root. Manning et al. [5] describes the goal of this process as "to reduce inflectional forms and sometimes derivationally related forms of words to a common base form."

> *"How many four digit odd numbers can be created using only the digits 0, 1, 2, 3, 4, and 5 if repetition of digits is not allowed?"*
>
> …
>
> *"How many four digit odd number can be create use only the digit 0 1 2 3 4 and 5 if repeat of digit be not allow?"*

This example serves a dual purpose; it demonstrates the process of stemming by showing the sort of sentence that a stemmer might create. A similar process is used by Google's search engine in order to understand different word forms and find related articles. However, it also conveys how the original meaning has been somewhat lost. Notice how when "is" is changed back into its root form, the connection between "repeat of digit" and "not allow" is far less clear from a high-level perspective? The parser may have a more difficult time drawing a connection between these two tokens. This is an obstacle we believe can be overcome with proper processing of tokens. Eliminating ambiguity is a worthwhile tradeoff.

Stemmers often come in NLP software packages and should be sufficient for our purposes. By using one, it will be simpler for our AI to interface with our database of previous problems.

There is one final NLP-related aspect of the translation left to be mentioned. Earlier, we indicated the usefulness of tokenization, and the necessity of constructing tokens that did not consist of words by themselves, but rather connected words that conveyed related ideas. We need a reliable way to identify significant terms into phrases. This is a two-fold process. Keywords (such as the

aforementioned “how many” or “from” and “through”) can be used to indicate related words and phrases. Alternatively, tokens may be generated with the aid of part-of-speech tagging, or POS tagging.

POS tagging uses a lexicon to identify words and determine to which category they belong. Referencing our usual example:

“How many integers from 1 through 999,999 contains each of the digits 1, 2, and 3 at least once?”

…

[(‘How’, ‘QW’), (‘many’, ‘ADV’), (‘integers’, ‘NN’), (‘from’, ‘IN’), (‘1’, ‘NUM’), (‘through’, ‘IN’), (‘999,999’, ‘NUM’), (‘contains’, ‘VB’), (‘each’, ‘DET’), (‘of’, ‘ADP’), (‘the’, ‘DET’), (‘digits’, ‘NN’), (‘1’, ‘NUM’), (‘2’, ‘NUM’), (‘and’, ‘CC’), (‘3’, ‘NUM’), (‘at’, ‘ADP’), (‘least’, ‘ADV’), (‘once’, ‘NUM’)]

For this example, we used the Universal Part-Of-Speech Tagset to classify the words, as in chapter five of Bird et al. [1]. Based on our understanding of tokenization, virtually every token should have either a noun (marked NN) or a numeral (marked NUM). If there are others, they will have to be detected by keywords. Furthermore, identifying which words belong in any given token will be based on the way they are structured. As an example, a determine or article (marked DET) usually indicates the beginning or end of a token.

With this groundwork, we can discuss the first step in translation: breaking the input text into tokens. First, we stem the input text, breaking every word into its original form. Next, we use POS tagging to obtain extra information about the structure of the text. Finally, using pattern matching and keywords, we divide the text into tokens using a tokenizer that we will develop ourselves.

"How many integers from 1 through 999,999 contains each of the digits 1, 2, and 3 at least once?"

…

"How many integer from 1 through 999999 contain each of the digit 1 2 and 3 at least once?"

…

[('How', 'QW'), ('many', 'ADV'), ('integer', 'NN'), ('from', 'IN'), ('1', 'NUM'), ('through', 'IN'), ('999999', 'NUM'), ('contain', 'VB'), ('each', 'DET'), ('of', 'ADP'), ('the', 'DET'), ('digit', 'NN'), ('1', 'NUM'), ('2', 'NUM'), ('and', 'CC'), ('3', 'NUM'), ('at', 'ADP'), ('least', 'ADV'), ('once', 'NUM')]

…

[*"How many", "integers", "from 1 through 999999", "contains", "each of", "digits", "1, 2, and 3", "at least once"*]

Once the text has been broken into tokens, the next step is to translate the tokens into our new format. For some parts of the problem, this is relatively straightforward; a single token can be represented as one or two lines in our translation. Other pieces of information require more ingenuity to extrapolate, as they require us to recognize the connections between multiple tokens. In either case, however, our translation AI uses keywords and pattern matching to properly derive meaning.

Patterns and keywords are intrinsically related in this project. Patterns are stored in our database. Each one corresponds to at least one line in our translation. Some of them require the classification that the POS tagger provided us, while others incorporate the actual words. In either case, patterns consist of keywords. For this project, keywords are simply said to be the building blocks of patterns. As the translator parses each word, it compares the word with a database of keywords which allow the AI to correctly select the patterns needed.

Patterns for obtaining ranges:

1. from [NUM] through [NUM]
2. from [NUM] to [NUM]
3. between [NUM] and [NUM]
4. [NUM] to [NUM]

Words that are written out are meant to be understood as they are. However, words that are in brackets specifically refer to the tags that the POS tagger assigns. In this case, NUM refers to a numeral. If one of these patterns is matched, then the translator will generate a range. A range merely consists of a minimum number and a maximum number. Referencing our usual example:

[(‘How’, ‘QW’), (‘many’, ‘ADV’), (‘integer’, ‘NN’), (‘from’, ‘IN’), (‘1’, ‘NUM’), (‘through’, ‘IN’), (‘999999’, ‘NUM’), (‘contain’, ‘VB’), (‘each’, ‘DET’), (‘of’, ‘ADP’), (‘the’, ‘DET’), (‘digit’, ‘NN’), (‘1’, ‘NUM’), (‘2’, ‘NUM’), (‘and’, ‘CC’), (‘3’, ‘NUM’), (‘at’, ‘ADP’), (‘least’, ‘ADV’), (‘once’, ‘NUM’)]

…

[*“How many”, “integers”, “from 1 through 999999”, “contains”, “each of”, “digits”, “1, 2, and 3”, “at least once”*]

Once the AI begins processing the third token, “from 1 through 999999,” it will create a range from 1 through 999,999, inclusive. Keep in mind that, at this point, we do not know to what this range refers; until we process the other tokens, we will not know if this is a range to select integers from or perhaps a range to NOT select integers from. For a smaller range, like “1 to 5”, it could be a set to select digits from, or something else entirely. However, the four keywords in this token are proof enough that the AI needs to indicate that this range exists.

Another important pattern is the phrase “how many.” Not only are these words given priority by the tokenizer, “how many” maps to a specific pattern in our database that is used to indicate the enumeration in the counting problem.

How to process “how many” token:

1. Locate the “how many” token (there should be one in every problem we work with).

2. Obtain the token immediately after the “how many” token.

3. Extract the last noun in this token.

The “how many” token is first processed by immediately taking the next token as input. The tokenizer will have prepared this token according to a rule that looks like:

{[NN] | [ADJ]}+ [NN]

The pattern above uses several symbols that we intend to use when describing patterns in our database. These will be expanded on in later sections, but to briefly summarize: The braces {} are used to indicate a group of keywords, the brackets [] are used to indicate a linguistic category such as nouns and verbs, and the cross + is used to indicate zero, or more, iterations of whatever is immediately to the left of it. The pipe | is used to indicate multiple possibilities; this means that “[NN] | hello” will match either a noun or the word “hello.”

Let us explain the example above in more detail. The pattern will match some repeating number of nouns and adjectives (or none at all) followed by a noun. This is only the structure. In order to process this token, the AI must obtain the last noun in this token. A few examples should illustrate this:

1. [*“How many”, “integers”, “from 1 through 999999”, “contains”, “each of”, “digits”, “1, 2, and 3”, “at least once”*]

2. [*“How many”, “four digit odd numbers”, “can be created”, “using only the digits”, “0, 1, 2, 3, 4, and 5”, “if repetition of digits is not allowed”*]

3. [*“How many”, “3 digit positive integers”, “can be formed using”, “odd digits less than six”*]

The first example is our usual example. It is the simplest of them all; following the “how many” token is the token “integers,” which can only consist of the noun, “integers.” Thus, integers are what the translator will interpret as the item to be counted. The second and third examples start out the same; the translator goes to translate the “how many” token, which requires that it look to the next token. However, these new tokens both have four words in them. Furthermore, the word “digit” should be an adjective in this case but may be erroneously classified as a noun by the POS-tagger. Therefore, the AI translator searches for the last noun in the token.

Experience-based learning

Several times, we have referenced a database with which the AI will interface. Here, we will expand on its role.

Our translator AI will store several pieces of information, but the most important will be patterns. Patterns may be used by either the tokenizer or the translator; we have not decided if the two will be separated or if it is possible to reuse patterns in both areas. This would allow patterns to conform to a specific, defined usage; one type of pattern is used to generate tokens, the other to generate lines of the translation. In practice, however, this may create unnecessary redundancy.

The benefits of using a database itself, however, are clear. We do not intend to treat the entirety of the translator AI as one collection of code as this may lead to unnecessary confusion on the task of its original purpose; each step that the translator takes is meant to be clearly defined. More importantly, this will aid us in making the translator more extendable. In the future, adding more refined functionality to the AI would simply be a matter of adding more patterns to the database.

Pattern Structure

Patterns will be used to generate either a token or a line in the translation of the problem. Every pattern consists of individual components called keywords. A keyword can refer to either an actual word or a classification of a word. Furthermore, we use operators to enhance the expressiveness of the words. Operators slightly change the meaning of any keywords in the pattern. Keywords are case-insensitive; "integers" and "Integers" will both map to the same pattern.

This approach to expressing patterns is intentionally made similar to regular expressions in order to make them more universally understood. However, they are not exactly the same. The notation of regular expressions is designed to work with specific characters and character classes. Our pattern-notation will map to words and word classes.

The brackets, indicated [], will contain a word class. The text inside of brackets is used to express a range of possible words.

[NN]: *This pattern will map to any noun.*

[ADJ] integers: *This pattern will map to the word "integers" with an adjective attached to it.*

A pipe, indicated |, can be used to make a pattern match different things.

Integer | digit: *This pattern will map to either the word "integer" or the word "digit."*

[NN] | [ADV]: *This pattern matches either a noun or an adverb.*

Braces can be used to group things together, and they can be used to apply notation to a larger group of keywords.

[ADJ] integer | integer: *This pattern is miswritten; it is guaranteed to require an adjective, followed by either the word "integer" or the word "integer." The next example shows it written in a more useful way.*

{[ADJ] integer} | integer: *This pattern will match either the word "integer" or an adjective, followed by the word "integer."*

The asterisk, indicated *, is used to describe that a keyword or set of keywords may appear multiple times, one time, or none at all.

How many [ADJ]* [NN]: *This pattern will match any of the following phrases (and more): "How many integers", "How many odd integers", or "How many 3-digit odd integers".*

The escape character, indicated \, can be used to deviate from the original meaning of any of the symbols that we have presented here. For word classes, only a single escape character is needed to deviate from its original meaning.

[NN] | \[NN]: *This pattern will match either a noun or the keyword "[NN]".*

\[{[NUM], }*[NUM]\]": *This pattern will match a set of numbers delineated with brackets. For example, "[1, 2, 3, 4, 5]"*

Pattern Usage

Patterns are used in two key areas of the AI: the tokenizer and the translator. The tokenizer uses them to parse the language into definable sections which can then be compared and independently processed. The translator uses them to better determine what lines to generate in our translation.

The patterns mentioned so far have been based purely on our knowledge of the structure of counting problems. We did not consult any guides on linguistics,

but the patterns we have fit the sample problems that we have prepared, which were randomly sampled from several books on discrete mathematics and algebra.

User Data

One last feature of the database is to collect user data and attempt to derive patterns from them. All problems submitted by users will be stored. The AI will parse the problems it receives in intervals and attempt to determine new patterns. This involves keeping statistics on the number of occurrences of any particular set of keywords. New entries will be added as the components arrive to the database, and entries that are made particularly common may be incorporated later as new patterns.

Section 3: Design of a Mathematical Machine Language

In this section we will look at the design of a Mathematical Machine Language. The typical case of general natural language translation is imprecise, thus making it unfit for computation in its raw form. In order to make a future AI problem solver understand the translation, we need a highly accurate computable mathematical language which can be processed by the AI problem solver. This language will conform to the guidelines discussed at the beginning of the paper. We need to use natural language pre-processing to understand the meaning of a given mathematics problem so that we can translate it into a formal and rigorous format, which we refer to as our mathematical machine language. In this section, we discuss the details of the design of this language.

Language Constructs

The understanding of a mathematics problem will be determined by the structure of the problem. Our translation must describe the components of the structure with formal expressions. Since we are targeting counting problems, our language design should accurately reflect the properties of the counting problems.

1. **Enumerator**

   Every counting problem needs to count a certain mathematical object, which we call the *enumerator*. For example, we might count the number of integers that satisfy certain conditions; or we could count the number of triangles following the given conditions. In our formal mathematical machine language, we denote it by [#enum#].

How many integers from 1 through 999,999 contains each of the digits 1, 2, and 3 at least once?

The enumerator representation is: [#enum#]:={#int#}

2. **Unit**

   An enumerator is typically comprised of basic components; for example, an integer is formed by digits. We define such a component as the *unit* of the enumerator and use [#unit#] to represent it. The unit of the previous example would be denoted as:
   [#unit#]:={#digit#}

3. **Subunit**

   A unit may also be formed by smaller components. For example, when we count the number of triangles, the enumerator is a triangle; the unit is a vertex; and a vertex is formed by coordinates. We call its *x*-coordinate or *y*-coordinate as the subunits. The subunits could have multiple levels. When we have a single subunit, we use [#subunit#] to represent it. When we have multiple subunits, we denote them by [#subunit:A#], [#subunit:B#], and so on. We may also have multiple-levels of subunits. To represent subunits in that situation, we use [#subunit:1#], [#subunit:2#], etc. If there are several subunits in one level, we can combine both formats with [#subunit:1A#], [#subunit:1B#], etc. See the example below for the representation:

How many triangles with positive area have all their vertices at points (i, j) in the coordinate plane, where i and j are integers between 1 and 5 inclusive?

[#enum#]:={#triangle#}

[#unit#]:={#vertex#}

[#subunit#]:={#coordinate#}

4. Constraint

In a typical counting problem, there are many constraints, that are applied on the enumerator, the unit, or the subunits. In order to specify to which object a constraint is applied, we use level numbers to refer to those components. For example, the enumerator corresponds to level 0; the unit corresponds to level 1; and the subunit corresponds to level 2, and so on. The notation we use for a constraint is: [#constr#]. The standard format we use for a constraint is: *Category:Rule*

We use categories to organize different types of rules, such as *Range, List, Form*. Some of the rules do not belong to any category, and we omit the category part for these kinds of constraints. We treat this case as the *general* category. We use the previous example to illustrate the way we use to represent the constraints.

How many integers from 1 through 999,999 contains each of the digits 1, 2, and 3 at least once?

[#enum#]:={#int#}

[#unit#]:={#digit#}

[#constr:0#]:={@Range:[1 → 999999]@}

[#constr:1#]:={@List:{1,2,3}@}

[#constr:1#]:={@Repet:Num(Each())>=1@}

We enclose a constraint inside a pair of special tags: {@ … @}, so that our MLP-system can parse the constraints easily. A range rule is enclosed inside a pair of brackets, such as [1 → 999999]. A list rule is enclosed inside a pair of curly braces, such as {1,2,3}. When we need to refer to a rule later, we use a symbol to denote the list, such as {@List:=L={1,2,3}@}. The notation [#constr:0#] means that the constraint is applied on the enumerator; and [#constr:1#] means that the constraint is applied on the unit. The third constraint rule above uses two functions: *Num()* and *Each()*, which we will discuss next.

5. Function

In order to represent a large number of mathematical properties, we need to define many functions in the form of *FunctionName(ParameterList)*, although sometimes the parameter list might be empty. In such a case, we use some default meaning, which we will explain below. In the example above, we used two functions: *Num()* and *Each().* The expression *Each(L)* refers to any element in the list *L*. The function Num() represents the number of occurrences of the object as the parameter. For example, *Num(Each(L))* means that the number of occurrences of any element in the list {1,2,3}. Actually we can combine *Num()* and *Each()* as one function *NumEach().* There are a large number of functions that need to be defined by human experts, because this part belongs to the system design, and we do not expect that our MLP-system can do this part automatically. Let us go back to another example above to see more functions.

How many triangles with positive area have all their vertices at points (i, j) in the coordinate plane, where i and j are integers between 1 and 5 inclusive?

[#enum#]:={#triangle#}

[#unit#]:={#vertex#}

[#subunit#]:={#coordinate#}

[#constr:0#]:={@Area()>0@}

[#constr:1#]:={@Form:{#2-tuple#}@}

[#constr:2#]:={@List:[frm]{:int:}@}

[#constr:2#]:={@Range:[1 -> 5]@}

In our first constraint [#constr:0#], which is applied on the enumerator, we use a function *Area()*. Since the parameter list is empty, we assume that the default parameter is used, and in this situation, we define the default parameter as a triangle. Thus the expression *Area()>0* means that the area of any triangle is positive. Our second constraint is applied on a vertex, and it belongs to the *Form* category. We use a 2-tuple, that is (*i, j*), in the form of {#2-tuple#} to represent the form rule. Our third constraint is applied on the coordinate with a list rule which specifies that each coordinate is an integer. In the expression, we used an operator [frm], which means *selected from* operation. This constraint means that the list of the individual coordinates is not the whole integer set, only part of the set. Our fourth rule is also applied on the coordinate with a range rule that provides a specific range for each coordinate.

Translation Principles and Rules

With the basic constructs defined above, in order to complete mathematics problem translation, we need a set of principles and rules to guide our MLP-system. In order to present these principles and rules, we would like to use a set of examples to derive them in a natural way.

"How many four-digit odd numbers can be created using only the digits 0, 1, 2, 3, 4, and 5 if repetition of digits is not allowed?"

We have looked at this example before. Now we may use our formal mathematics machine language to represent its translation. Based on the described patterns, we can extract a constraint on the enumerator *integer* as a "four-digit odd" requirement. We extract an explicit list of digits as {0,1,2,3,4,5}, and there is a repetition rule given by "no digit repetition". Thus, we have the following translation.

[#enum#]:={#odd#}
[#unit#]:={#digit#}
[#constr:0#]:={@Form:NumElements()=4@}
[#constr:1#]:={@List:{0,1,2,3,4,5}@}
[#constr:1#]:={@Repet:NumEach()<=1@}

In this example, we train our MLP-system with the following rules or patterns.

Translation Rule R1: Merge Rule

When certain constraint on the enumerator has a very close relationship, we can apply the ***merge rule*** *to get one single expression.*

In this example, the constraint "odd" and the enumerator "integer" are *mergeable* as a single expression {#odd#}. How does the system know if a constraint and an enumerator are mergeable? This decision should be made by a human expert, and the system uses the experience data to make its decision.

Translation Rule R2: Form Rule

After we extract the enumerator and the unit, we need to extract certain constraint that describes the way to form an enumerator from the units.

In this example, we have a constraint that specifies the number of units in each enumerator; then we use the function *NumElements()* to describe the form

rule. We can extend the form rule **R2** to the unit-subunit pair and subunit:1-subunit:2 pair easily.

Translation Rule R3: List Rule

After we determine the unit, we need to look for a list of values that the unit will take.

In this example, we extract an explicit list of values for the unit: {0,1,2,3,4,5}, which follows the pattern: [NUM]*, and [NUM].

Translation Rule R4: Repetition Rule

After we get a list of values for the unit, we need to extract the constraint that describes the repetition rule.

For the repetition rule, we typically use the function *NumEach()* to specify the number of occurrences of each element in the list.

We use a few more examples to train our MLP-system with more rules and patterns.

How many 3 digit positive integers can be formed using odd digits less than six?

[#enum#]:={#posint#}

[#unit#]:={#digit#}

[#constr:0#]:={@Form:NumElements()=3@}

[#constr:1#]:={@List:[frm]{:odd:}@}

[#constr:1#]:={@Value()<6@}

In this translation, we applied the merge rule **R1** which combines the "positive" constraint and the enumerator "integer" as a single expression: {#posint#}. For the second constraint, we do not use the expression

{@List:{1,3,5}@}, which is equivalent to the requirement "odd digits less than six". Doing so would involve data processing, although it is simple processing. We set the following principle for our MLP-system.

Translation Principle P1: Minimum Processing Principle

When we translate the mathematics problems, refrain from excessive data processing. Preserve the meaning of the original problem as much as possible.

The reasoning behind this principle is simple: we wish to set a clear boundary between our translator and the future problem solver. The data processing belongs to the problem-solving part. Our future problem solver will be used for educational purpose, and it will need to explain all meaningful data processing steps to the students. If our translator does excessive data processing, then the problem solver would not have the chance to explain each part to the students. As a caveat, sometimes we may need to do some necessary processing. We will see that below.

James and a friend played 14 games of tic-tac-toe one night. In how many ways can James end the night with eight wins, two ties, and four losses?

[#enum#]:={#way#}

[#unit#]:={#state#}

[#constr:0#]:={@Form:NumElements()=14@}

[#constr:1#]:={@List:{"W","T","L"}@}

[#constr:0#]:={@NumElements("W")=8@}

[#constr:0#]:={@NumElements("T")=2@}

[#constr:0#]:={@NumElements("L")=4@}

In this translation, we omit a lot of information in the problem that we feel is not essential for a solution. We summarize this type of data processing as the following principle.

Translation Principle P2: Necessary Abstraction Principle

When we do translation on mathematics problems, some detailed information that is not essential for the problem-solving should be omitted. But the change must not alter the nature of the original problem.

In this example, the name of the person "James" is not important. The game type is also not important. We only want to know the results of the game: win, tie, or loss. This type of abstraction is very common in mathematics problem-solving, which is quite different from the general natural language processing.

Darina has 5 sticks measuring 5cm, 5cm, 8cm, 14cm, 14cm. Using exactly 3 sticks as the size of the triangle, how many non-congruent triangles are possible if the sticks are joined only at their end points?

[#enum#]:={#triangle#}

[#unit#]:={#side#}

[#constr:0#]:={@CountBy:Congru()=No@}

[#constr:1#]:={@List:{5,5,8,14,14}@}

In this example, we use the principle **P2** to skip the information like Darina, cm, and make abstraction of a stick as a side of a triangle. For the first constraint, we add a category called "*CountBy*," which specifies the counting rule: what criterion we use to identify different enumerator. The rule we use here "Congru()=No" means that if two triangles are congruent, then we do not treat them as different enumerators. Here there is a hidden rule we do not specify here,

that is, we use 3 sides to form a triangle. The reason is that we do not need to represent knowledge that is supposed to be known by everyone.

Translation Principle P3: No Known-Fact Principle

For any property that is treated as a known fact, we should not translate it in our output. The future problem solver will easily recover it when solving the problem.

This principle tells the system to only do appropriate translation, neither doing more than necessary nor less than necessary.

How many distinct sums can be obtained by adding 3 different numbers from the set {-2, -1, 1, 2, 3, 4, 5}

[#enum#]:={#sum#}

[#unit#]:={#number#}

[#constr:0#]:={@CountBy:Value()=Distinct@}

[#constr:1#]:={@Set:{-2,-1,1,2,3,4,5}@}

[#constr:0#]:={@Form:NumElements()=3@}

In the first constraint, we use the "Value()" function to specify the counting criterion. When two sums have the same value, we cannot count them as two different sums. In the second constraint, we use the "Set" category instead of the "List" category. The main difference is that for the "Set" category, the elements selected must be different, which is not a requirement for the "List" category.

Translation Principle P4: Context-Based Explanation Principle

The context of the problem is important when we understand the meanings of certain keywords. Therefore, we need to determine the context using certain keywords in the problem.

How many different eight-card hands are there with no more than three black cards?

[#enum#]:={#hand#}

[#unit#]:={#card#}

[#constr:0#]:={@Form:NumElements()=8@}

[#constr:0#]:={@CountBy:OrderElements()=No@}

[#constr:0#]:={@Form:NumElements("black")<=3@}

In this example, the meaning of "hand" relies on the context. This question uses the card-game context, otherwise we cannot eliminate the ambiguity in our understanding. Based on this observation, we have the principle P4. In the above example, we use the function "OrderElements()" to define the counting criterion: when two hands have the same set of cards, but the cards are arranged in different orders, we treat them as the same hand. Or in other words, rearranging the order of the cards in a hand would not result in a new hand.

Translation Rule R5: Multi-Component Unit Rule

When we detect that a unit is composed of multiple components, we can use an n-tuple to represent the unit. Then we need to define the constraints on individual components.

> *A local pizza store offers a choice of seven toppings and three sizes (small, medium and large). How many three-topping pizzas do they offer?*

[#enum#]:={#pizza#}

[#unit#]:={#2-tuple#}

[#subunit:A#]:={#topping#}

[#subunit:B#]:={#size#}

[#constr:2A#]:={@Form:NumElements()=7@}

[#constr:2B#]:={@Form:NumElements()=3@}

[#constr:0#]:={@Form:NumElements(A)=3@}

The main property of this problem is that the unit has two components, topping and size. So we use a 2-tuple to represent this unit. Then we define the

constraints on individual components. With this arrangement, we can represent the structure clearly.

Section 4: Conclusion

This AI translator has the potential to provide a new approach to automated problem solving and understanding natural language as a whole. Each token represents an idea, and each pattern gives us a way to not only recognize, but also express that idea. By focusing our efforts on translation, we are able to draw focus away from a more difficult part of the process, namely answering the question, and instead come up with a computation-friendly way of expressing statements.

References

1. Bird, S., Klein, E., & Loper, E. (2009, June). Natural Language Processing with Python. Retrieved March 17, 2020, from https://www.nltk.org/book/ch05.html

2. Brownlee, J. (2019, August 7). What Is Natural Language Processing? Retrieved March 17, 2020, from https://machinelearningmastery.com/natural-language-processing/

3. Dorash, M. (2017, December 26). Machine Learning vs. Rule Based Systems in NLP. Retrieved March 17, 2020, from https://medium.com/friendly-data/machine-learning-vs-rule-based-systems-in-nlp-5476de53c3b8

4. Hosseini, M. J., Hajishirzi, H., Etzioni, O., Kushman, N. (2014). Learning to Solve Arithmetic Word Problems with Verb Categorization. Retrieved April 1, 2020, from https://www.aclweb.org/anthology/D14-1058.pdf

5. Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Retrieved March 17, 2020, from https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

6. Ruder, S. (2020, February 2). The 4 Biggest Open Problems in NLP. Retrieved from https://ruder.io/4-biggest-open-problems-in-nlp/

7. Thomas G. Evans. (1964). A heuristic program to solve geometric-analogy problems. In *Proceedings of the April 21-23, 1964, spring joint computer conference* (*AFIPS '64 (Spring)*). Association for Computing Machinery, New York, NY, USA, 327–338. DOI:https://doi.org/10.1145/1464122.1464156

8. Wang, Y., Liu, X., & Shi, S. (2017). Deep Neural Solver for Math Word Problems. Retrieved April 1, 2020, from https://www.aclweb.org/anthology/D17-1088/