A System Description of P^4: Possible Punctuation Points Parser

Thomas Boehnlein and Jennifer Seitzer

Department of Computer Science
University of Dayton, 300 College Park, Dayton, OH 45469

Abstract

We present a Natural Language Understanding (NLU) implementation that automatically inserts punctuation marks into a sequence of words to create a group of one or more syntactically correct sentences. The software, Possible Punctuation Points Parser (P^4) provides the ability for the user to input a string of words to process, performs the punctuation possibilities, and then provides several visualizations to illustrate how the software arrived at its final solution. P^4 uses a chart parsing algorithm combined with a search algorithm that creates data visualization structures. A potential application of this software is to serve as a formidable starting point for automatic punctuation mark insertion during voice-to-text conversion found on many mobile platforms.

Introduction

Natural language understanding (NLU) manipulates and internally represents natural language to perform algorithmic linguistic operations for the purpose of understanding. This NLU system is a work in progress and a result of the first author's Master's software project that automatically punctuates a string of words into syntactically correct sentences. Semantic analysis and representation are not addressed here. Nor have any probabilistic methods based on usage history been utilized in the system. Instead, a syntactic approach is used by organizing and identifying the words and their syntactic relationships to one another by using a chart parsing algorithm [1]. Much work has been done in chart parsing [1][2]. Work has also been done in prediction of punctuation in NLU [3][4]. This is a proof of concept of applying the chart parsing algorithm to automatic punctuation, now to be applied to voice-to-text systems.

The current interfaces typically provided by voice-to-text systems awkwardly require the user to explicitly state individual punctuation marks during the process of voice-to-text conversion. This method is used most notably by Apple's iOS [5]. Explicitly stating punctuation marks is unnatural because people generally do not interact with each other verbally in that way. Instead, they rely on conversational, visual, and intuitive cues. Speech patterns indicate when a sentence ends or if a question is being asked without the need of specifically stating any punctuation marks. However, punctuation marks are necessary for efficient processing of the text as these visual and auditory cues that occur in conversations are completely removed for the reader of a mobile voice-to-text transcription. The lack of automatic punctuation insertion in mobile systems presents a usability gap for either the speaker being forced to speak each punctuation mark or the reader having to do without the punctuation marks. The hope for this work is to contribute to the improvement of mobile voice-to-text systems by processing the data after the transcription is performed then inserting new punctuation information. The system presented here, P^4, is an operable small-scale implementation that is a step toward achieving this goal.

Overall Architecture of P^4

The main facets of this work include:

- the definition and implementation of an efficient parsing system
- the implementation of a novel search algorithm that allows for efficient processing of a large search space of possible punctuation mark insertions
- the implementation of an interface to help visualize the process of both parsing and searching while inserting possible punctuation marks

Overview of Parsing System

The parser is a chart parser that employs a type of dynamic programming. Chart parsers have the benefit of reusing sub-parses to avoid backtracking [1]. This is accomplished by doing both a top-down and a bottom-up parse in parallel. Both time and space complexity is polynomial which is important given the multitude of parses performed while navigating the punctuation mark search space. The parsing output is then analyzed to find the best parses that are complete (since partial parses are part of the output of the chart parser). A selection method is then used to rank the parses and choose the best. The grammar used by the parser is read into the system from a text file and converted into appropriate data structures for later use. The user is able to see all of the rules for guidance of valid input into P^4 and can modify the rules as needed.

Overview of Searching System

The number of permutations of possible punctuation schemes grows at an exponential rate of **O(*m^n*)** where ***m*** is the number of punctuation points available (plus whitespace) and ***n*** is the number of words in the list. For the word list "*good morning mom how are you*" using four (plus one) possible punctuation marks 15625 permutations of punctuation possibilities exist. If the list grows to fifteen words, over 30 billion punctuation permutations exist.

In general conversation, one typically uses a much larger vocabulary than six words and requires more than four punctuation marks. Thus, trying to exhaustively generate all possible combinations quickly becomes prohibitive, and an efficient search algorithm that prunes much of the search space is necessary. Fundamentally, the search algorithm does this by ignoring the parts of the search graph that do not form valid sentences which is ascertainable by using simple graph accessibility techniques described later.

Overview of Visualization System

Visualization methods are included in P^4 to gain insight into system operation, to assess the efficiency of the algorithms, and to diagnose failure points. NodeXL, created by the Social Media Research Foundation, was used for all of the visualization controls [6]. The library is highly optimized for visualizing complex graphs that may need to be organized using a variety of different node layout algorithms. This is important since the punctuation mark search space is extremely large. In this system, plots visualize the navigation of the search space and how the final result was parsed.

Overview of Main Operating Loop

The interaction of the three sub-systems described above is shown in Figure 1. After the user enters the ordered collection of words to be punctuated, the Search Engine and Parse System cyclically interact to create and validate numerous punctuation-possibilities. When the search system ultimately chooses the best one ("the result"), both systems send data to the visualization system for the output of the result and the meta-information of how that result was achieved.

Implementation of P^4

In this section, implementation details of P^4 are explained by describing the constituent classes, data structures, and algorithms of the three sub-systems: the parser, the search engine, and the visualization tool.

Parsing System Implementation

It is the responsibility of the parsing system to analyze the individual strings (punctuation possibilities) that are generated by the search engine of P^4. It is a chart parser [1] made up of the classes ParseEdge, ParseGrammar and ParseParser. A chart parser differs from a traditional left-to-right recursive descent parser by avoiding backtracking [2]. That is, it expands all possible production paths initially and stores them in the data structure called a chart along with meta-information of how the chart was created.

A chart can be considered a subgraph of the parse space made up of instances of the data structure called an *edge*. That is, a chart is a collection of edges. In particular, the chart is made up of a vertex before the first word, in-between each word and after the last word. This results in ***n+1*** vertices for every collection of ***n*** words. These are used to define the beginning and ending points of each edge. Intuitively, edges are used to define how a subsection of the sentence was parsed. An example of a chart for the sentence *"The boy ate candy."* is shown in Figure 2.

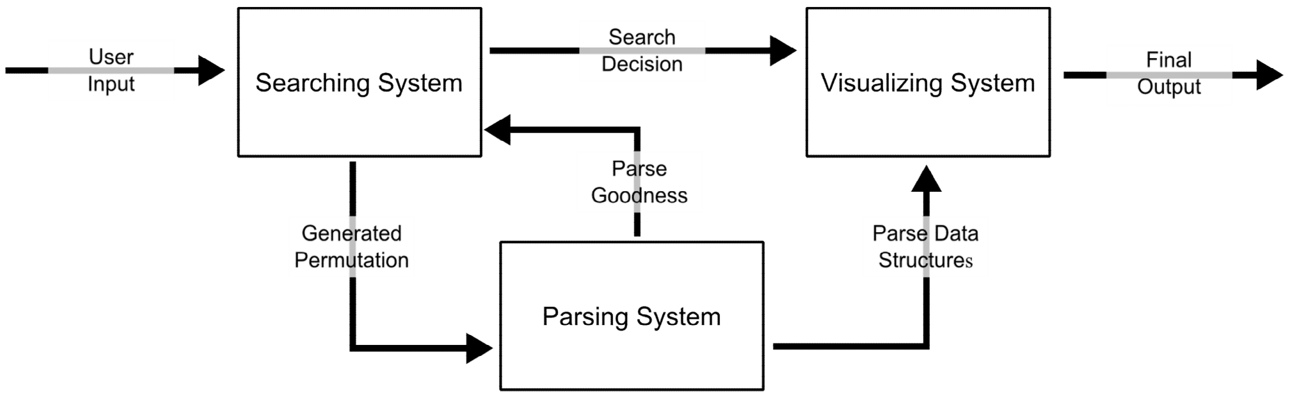


Figure 1 - Data Flow Diagram of Possible Punctuation Points Parser

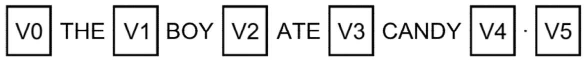


Figure 2 Chart Vertices and Edges

"*The boy*" would be a valid edge in the sentence which begins at **V0** and ends at **V2**.

In addition to storing the starting point, ending point and the edge goal, the edge also stores two lists: the Haves and the Needs. The Haves list represents what the parser has identified so far. Going back to the "*The boy*" edge, the goal of the edge is a sentence. The Haves list has a ***noun phrase***. But in order to be a sentence, which is the goal, the edge also needs a ***verb phrase***. So the Needs list includes a ***verb phrase***. If the edge cannot find a ***verb phrase*** in the remaining sentence, it is called an incomplete edge. Otherwise, the edge is called a complete edge. Complete edges always have an empty Needs list. The edges are implemented by the class ParseEdge.

The ParseParser class operates on a list of edges for each vertex. Edges that end at a particular vertex are stored in the same list. The ParseParser class organizes the edges like this to facilitate simultaneous top-down and bottom-up parsing using its three main functions: Predictor, Scanner, and Extender. These functions create the edges and new edges as the parse proceeds.

The first function, Predictor, uses a top-down approach. It creates new edges from the needs of the current edge. "*The boy*" edge needs a verb phrase. Thus, Predictor creates a new edge that starts at the vertex right after the word "boy" in the sentence and has a goal of a verb phrase. Each new edge contains the right hand side (RHS) of one of the available rules in the grammar that has verb phrase on the left hand side (LHS) of a rule. An example of the new edge creation process using the predictor function is shown in Figure 3.

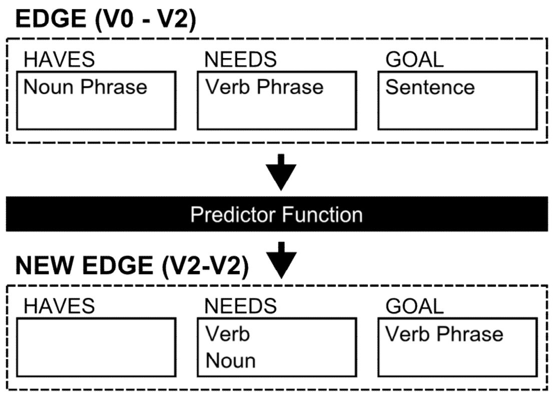


Figure 3 – Execution of Prediction Function

The next function, Scanner, uses a bottom up approach. Scanner looks at each edge in a chart to see if the remaining words can be used to move the first object from the Needs list to the Haves list. If it can be moved, then a new edge is created with the new word(s) and added to the appropriate edge list. An example of the new edge creation process using the scanner function is shown in Figure 4.

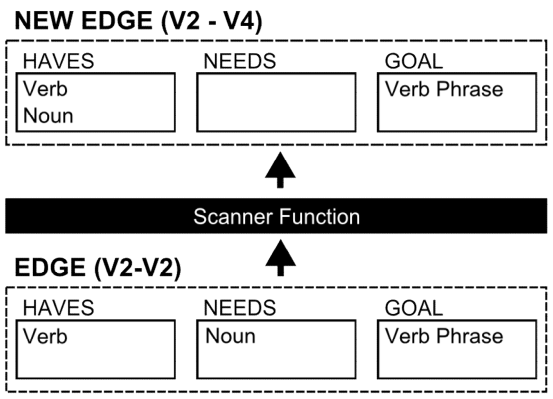


Figure 4 - Execution of Scanner Function

The final function, Extender, also operates in a bottom-up manner. It takes an edge *X* and looks at all other edges in an edge list that ends where *X* begins. If the goal of *X* matches with the first item in the Needs list of one of the searched edges, a new edge is created by combining the two edges. The new edge starts at the searched edge and ends at the end of edge *X*. Lastly, the matching object from the Needs list is moved to the end of the Haves list. An example of the new edge creation process using the extender function is shown in Figure 5.

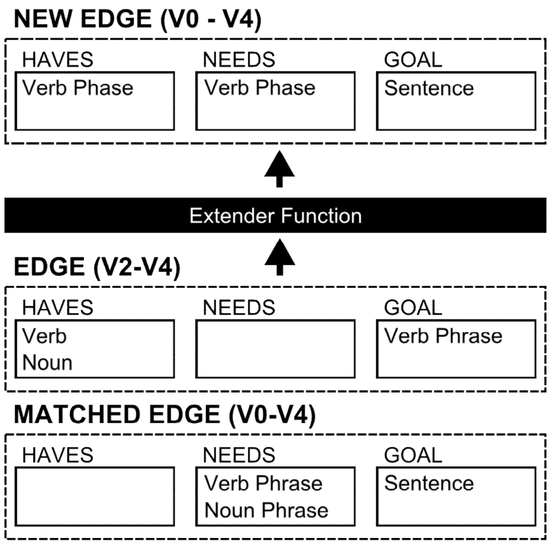


Figure 5 - Execution of Extender Function

Searching System Implementation

The Searching System creates all feasible punctuation possibilities. It uses five punctuation marks: period, comma, semicolon, and question mark, plus whitespace when no punctuation mark is chosen. The rules that govern the insertion of punctuation marks are found in the class ParseGrammar.

Because of the combinatorial complexity of considering all possible punctuation schemes, ParseSearcher only generates the sections of the search space that have the possibility of containing a valid parse. The decision to stop looking for more punctuation insertion points is made by ascertaining the nonexistence of an edge in the chart that begins at the first vertex and ends at the last vertex and has a sentence, sentence phrase, partial sentence, connected sentence, or sentence with an ending punctuation, as the last item in the Haves list.

During each new iteration, P^4 takes advantage of the fact that a chart parser can reuse past partial parses. Thus, as it goes down another level in the search space, it passes along the previous parse, copying all of the edges from the previous parse into the current parse. Thus, during this new parse, the parser only has to create the new edges added by the new word or punctuation mark. This dramatically increases the efficiency of the search system.

The final task of the search system is deciding which collection of one or more valid sentences that has been generated is the best one. In some cases, it is impossible to pick the best parse without knowing the context of what was said. In such cases, one would need the timing information to decide if a pause meant a comma or a period. For example, *"good morning, mom. how are you?"* is as equally valid as *"good morning. mom, how are you?"* P^4 makes no attempt to resolve this ambiguity. To choose the final result, P^4 selects the parse that created the most sentences with the least amount of punctuation while still remaining valid as a best parse. This is done by rewarding the parse with a large increase in rating per sentence found and subtracting a small amount for each punctuation point found.

As an example, consider the following ordered sequence of words: "*good morning mom how are you I am okay are you okay I got a telescope for my birthday the telescope is good and I like it*". This word sequence generates 7.4E18 possible punctuation permutations if created using a brute force approach. P^4 required only 200 parses while searching by quickly eliminating branches that would not result in a solution, and found a correct solution of "*good morning, mom. how are you? I am okay. are you okay? I got a telescope for my birthday. the telescope is good and I like it.*" Moreover, in testing, P^4 only took 0.135 seconds to find the final solution on a modern notebook computer.

Visualization System

The final sub-system of P^4, the visualization system, shows the results of the previously discussed parse and search sub-systems. It is responsible for creating the visualizations and tabular results that are shown in the user interface of P^4. This includes the Edge Table, Search Plot, and Parse Plot. The actual drawing of the charts is handled by NodeXL.

The Edge Table lists all edges generated by the parser's parse chart during an execution. The top of the table contains a list of the parsed words and the vertices that the edges used. Then each edge is written to the table. An example of edges as presented in the Edge Table are shown in Figure 6. Each edge entry shows the span of indices that the edge covers, the goal of the edge, the edge's Haves objects, the edge's Needs objects, and the words that the edge covers. The edge's goal appears first and is separated from the rest of the parts by "-->". The Haves and Needs objects are separated by "@". Finally, the words covered by the edge are separated by "-->" if the edge is complete. These edges show how the grammar was explored to figure out how to parse the collection of words and punctuation marks. In addition to showing how the parse search space was explored, they are also critical to creating the final parsing chart which will be discussed last.

```
(0, 5)    S_END --> S Pun_End  -->  good
morning , mom .
(5, 6)    Question --> Interrogative @ VP NP
```

Figure 6 - Example of a Complete and Incomplete

The Search Plot depicts the paths traveled while searching for possible punctuation points. It is created by first inserting a root node into the tree which is represented as a cyan square at the top of the tree. A new node is added for each decision point that is made while searching for the punctuation point placements. Each decision node is colored black. All possible decision outcomes are given a node: comma (red), period (purple), question mark (blue), semicolon (green) and no punctuation mark (white) as shown in Figure 7. This creates an n-ary tree where there are ***n*** possible search decision choices. The black circle represents this decision point in the tree and allows the shape of the tree to remain visually pleasing.

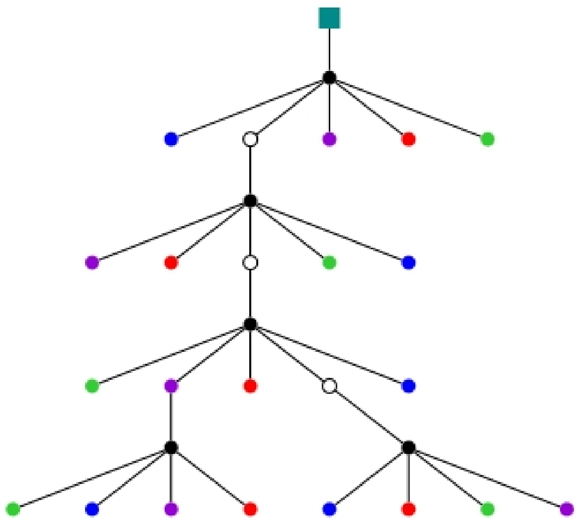
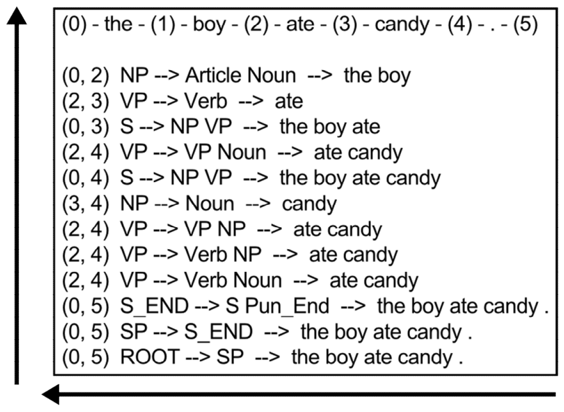


Figure 7 - Example Search Plot

The Parse Plot is built from the complete edges found in the parser's parse chart. Not all complete edges in the parser are part of the final chosen parse. In order to determine which edges belong and which edges do not, the table of complete edges must be recursively searched and matched using a bottom up approach relative to the Edge Table as illustrated in Figure 8.



```
(0) - the - (1) - boy - (2) - ate - (3) - candy - (4) - . - (5)

(0, 2)  NP --> Article Noun  -->  the boy
(2, 3)  VP --> Verb  -->  ate
(0, 3)  S --> NP VP  -->  the boy ate
(2, 4)  VP --> VP Noun  -->  ate candy
(0, 4)  S --> NP VP  -->  the boy ate candy
(3, 4)  NP --> Noun  -->  candy
(2, 4)  VP --> VP NP  -->  ate candy
(2, 4)  VP --> Verb NP  -->  ate candy
(2, 4)  VP --> Verb Noun  -->  ate candy
(0, 5)  S_END --> S Pun_End  -->  the boy ate candy .
(0, 5)  SP --> S_END  -->  the boy ate candy .
(0, 5)  ROOT --> SP  -->  the boy ate candy .
```

Figure 8 - Creating Parse Plot from Edge Table

The algorithm finds the correct ROOT edge first by starting at the bottom of the table then recursively expands up the table. The objects in the Haves list are matched in a right-to-left manner to the goal of subsequent edges in the table. As edge goals are matched to objects in the Haves list, the goal is flagged as used and cannot be used by other Haves objects as a potential match. In addition, the Haves object is also flagged once it has a match so it can no longer be expanded. If a match cannot be found, it must be a leaf node of the parse tree. The words and punctuation marks found in the final solution are placed in the leaf node in a right-to-left manner. Once all Have objects derived from the ROOT edge have been completely matched, the parse tree is complete and can be displayed to the user interface. An example of the Parse Plot is shown in Figure 9.

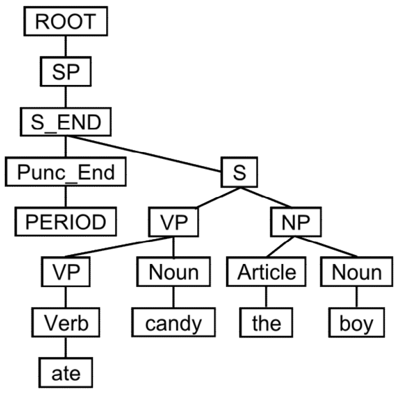
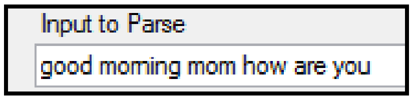


Figure 9 - Example of Parse Plot

Sample Runs and Results

In this section, a sample run of P^4 is described. The Input to Parse text box is located at the top of the window. It is where the user will input the series of words that P^4 will use. The sample run text is shown in the Input to Parse text box in Figure 10.

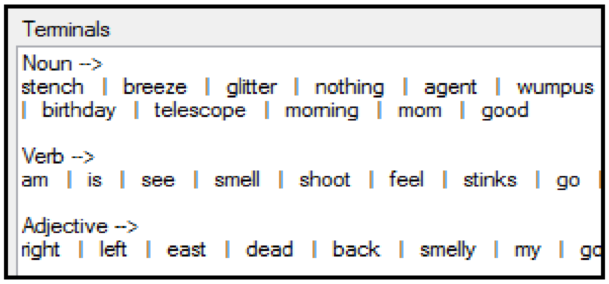


Input to Parse

good morning mom how are you

Figure 10 - Input to Parse Text Box

Since the current grammar is limited in scope, the user must know the rules of the grammar in order to create a series of words that can be processed by P^4. To find out the terminal and non-terminal rules of the grammar, the user can press the Show Grammar button located above Input to Parse. Sections of the grammar window are shown in Figures 11 and 12. These rules can be modified by the user since they are stored in a simple text file.

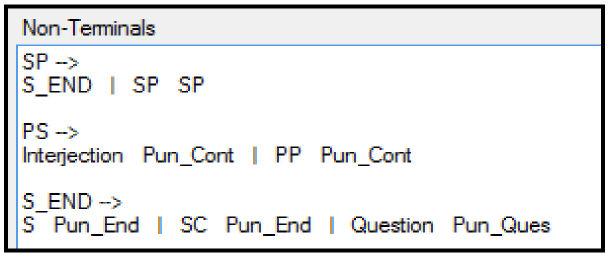

Terminals

Noun -->
stench | breeze | glitter | nothing | agent | wumpus
| birthday | telescope | morning | mom | good

Verb -->
am | is | see | smell | shoot | feel | stinks | go |

Adjective -->
right | left | east | dead | back | smelly | my | go

Figure 11 – Part of Terminals from Grammar Window


Non-Terminals

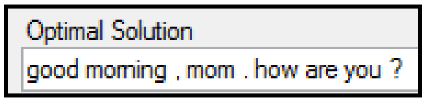
SP -->
S_END | SP SP

PS -->
Interjection Pun_Cont | PP Pun_Cont

S_END -->
S Pun_End | SC Pun_End | Question Pun_Ques

Figure 12 – Part of Non-Terminals from Grammar Window

Below the Input to Parse text box is the Optimal Solution text box. The best possible fully punctuated parse that P^4 could find will be written here after the search is complete as shown in Figure 13.


Optimal Solution

good morning , mom . how are you ?

Figure 13 - Optimal Solution Text Box

Running the sample input with P^4 generates the three main visualization outputs: Edge Table, Search Plot and Parse Plot. The Edge Table control features a table of edges and two checkboxes. The top of the table displays the final solution with numbered nodes. In this case, there are ten nodes and nine edges to represent the parse chart after punctuation marks were inserted. Figure 14 shows the top of the Edge Table for the sample input.

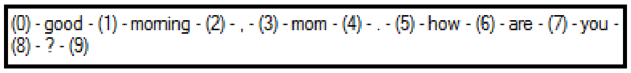

(0) - good - (1) - morning - (2) - , - (3) - mom - (4) - . - (5) - how - (6) - are - (7) - you -
(8) - ? - (9)

Figure 14 - Top of Edge Table from Sample Run

The checkboxes determine whether incomplete or complete edges are shown. Figure 15 displays a portion of the Edge Table from running the sample input.

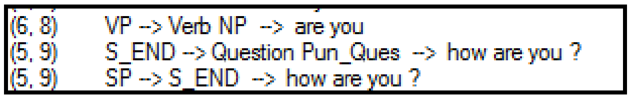

(6, 8) VP --> Verb NP --> are you
(5, 9) S_END --> Question Pun_Ques --> how are you ?
(5, 9) SP --> S_END --> how are you ?

Figure 15 - Portion of Edge Table from Sample Run

In the highlighted example, the middle edge has a goal of S_END. It has completed that goal with Question and Pun_Ques. The phrase "*how are you*?" is located between the indices of 5 and 9 as seen at the top of the Edge Table. Next, the Search Plot displays how P^4 efficiently navigated the search space of possible punctuation permutations. The plot is shown in Figure 16.

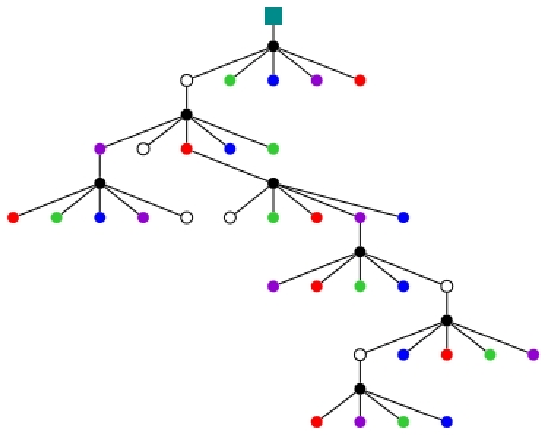
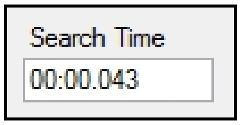


Figure 16 - Search Plot from Sample Run

To the right of the Search Plot is a text box showing the amount of time it took for the search to conclude. It allows the user to track the increase of time as the parses become more complex when adding additional words. The time it took to search and parse for *"good morning mom how are you"* is shown in Figure 17.


Search Time

00:00.043

Figure 17 - Search Time from Sample Run

The final output is the Parse Plot. This plot, shown in Figure 18, displays how the sample input was parsed after it was punctuated by P^4. It is made up of a parse tree where each node is the LHS of a grammar rule except for the leaf nodes which are terminals. Punctuation marks are represented by the word in all capital letters since the actual punctuation marks may be hard to see due to their small size. The organization of the graph does not take into account the order of the words in the sentences. It only ensures that they are on the right ply. Currently, the software cannot specify the left-to-right order in which the graphs are drawn. The tree starts at the root and then is split into sentence phrases which are made up of one or more sentences. The sentence phrases are then defined as sentences with ending punctuation. Those are then defined as their respective type of sentence and punctuation mark. Phrases such as verb phrase are defined next which is then followed by parts of speech for each of the words. Finally, the words and punctuation marks are inserted to complete the parse tree.

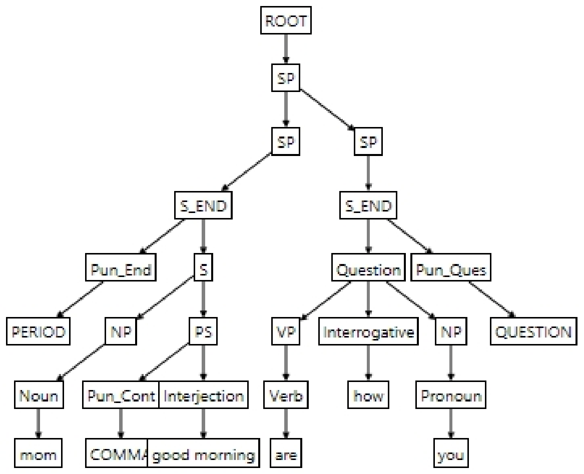


Figure 18 - Parse Plot from Sample Run

As shown in the results from the sample run, P^4 provides a relatively straightforward interface with a goal of creating syntactically correct sentence(s) by adding punctuation marks to a sequence of words supplied by the user.

Conclusions

This work presented a system, Possible Punctuation Points Parser (P^4) that efficiently inserts punctuation marks into a string of words to form a syntactically correct sequence of one or more sentences. It does this by using a chart parser that simultaneously performs a bottom-up/top-down parse with an optimized search algorithm that maximizes pruning of the search space. In order to make the results as easy as possible to be analyzed by the user, P^4 provides several data visualizations to show the output from the chart parser, how the punctuation space was searched, and finally, how the final solution was generated by the grammar. The main contribution of this work is that it provides a proof of concept that shows that punctuation marks can be automatically inserted into voice-to-text transcriptions produced by mobile devices rather than requiring the user to explicitly state the necessary punctuation marks.

Future Work

Currently, the most pressing limitation is the grammar's small size which limits input, thus rendering the system's ability to handle only toy problems. The grammar needs to be expanded to allow for a wider range of use cases. In addition, as more rules and terminals are added, this will impact the parsing time as chart parsers create exhaustive solutions by examining every rule. This may lead to considering combining probabilistic methods with the chart parser.

Additionally, the ranking algorithm in selection of the final result can potentially be improved by incorporating syntax usage patterns found in the American English language. The final improvement of the parsing itself would be to add error handling to the parsing system by either exiting gracefully with a partial parse or looking at other methods to predict the missing information when a correct parse is not possible with the supplied grammar. Concerning the interface, the plots could be made more robust in terms of organization and presentation of the nodes found in the trees generated by P^4. This includes the ability to add additional information about the parses that occurred at each node through changes in visual characteristics such as color, size and location. Finally, individual partial parses need to be examined in depth. This could be accomplished with additional pop-up windows made available by simply selecting one of the nodes in the Search Plot.

References

[1] Arnold, D. "Chart Parsing", Dept. of Language & Linguistics, University of Essex.
[2] Russell & Norwig. AI A Modern Approach. 2013.
[3] Huang, J., and Zweig, G. "Maximum entropy model for punctuation annotation from speech". In *Proc. of ICSLP*, pp. 917-920, 2002.
[4] Kim, J., and Woodland, P. "The use of prosody in a combined system for punctuation generation and speech recognition". In *Proc. of Eurospeech*, 2001.
[5] "iPhone User Guide For iOS 6.1 Software", p. 25, 2013.
[6] NodeXL: Network Overview Discovery and Exploration for Excel 2007/2010. Social Media Research Foundation. Retrieved March 13, 2013 from http://www.smrfoundation.org/nodexl.