A Robust *k*-Best Shift-Reduce GLR Parser

Danny Nemer
Washington University in St. Louis
danny.nemer@wustl.edu

Senior Project Proposal

Abstract

This paper presents a robust natural language parser that outputs the *k*-best suggestions for an input query. Provided the input, the parser uses a weighted context free grammar (WCFG) to output the *k*-best parse trees, accommodating ill-formed input. *n*-gram based language models identify entities and terms in the input string, and lexical analysis is conducted on the input to account for semantics and morphology. The input query is then matched to terminal rules in the grammar, which are sent to the parser.

The parser is a bottom-up, online, GLR(0), shift-reduce parser, using a state transition table constructed from the grammar. Each tree is returned in a memory-efficient graph-structured stack. The parser employs *k*-shortest path routing to search for the *k*-best, legal parse trees. To be legal, a tree must reach the start symbol defined in the grammar and consist of consecutive, non-overlapping symbols that span the entire input. The cost parameterization utilized in the search algorithm is a function of: (1) the rule costs defined in the WCFG, (2) the costs of matching input segments with terminal rules based on lexical analysis, and (3) editing costs for insertion, deletion, substitution, and transposition of the input to form each suggestion.

1 Introduction

Searching a vast graph of structured data, such as a relational database, is challenging. Consider a database where each entry represents a node, and relations between the entries represent edges. Each entry also has structured information, which is the node's properties. The working example for this paper will be LinkedIn's database of professionals. Entries (nodes) are users, companies, industries, cities, etcetera. Each entry can have relations (edges) to others, such as users that are employees of a company, or a company that operates in a specific industry and city. Each entry has properties such as a user's name, dates of employment, or degrees held.

A query interface for searching this complex graph needs to understand the user's intent precisely. One possible interface is keyword-based search; however, the ability to understand the user's intent is limited. For example, "`people St. Louis`" can mean "`people who live in St. Louis`", "`people who work in St. Louis`", or "`people who have worked or lived in St. Louis`". Another possible interface is a filter-based form of checkboxes and drop-down selections; however, the interface would become too complicated and inefficient when implemented to support hundreds of different filters for nodes and edges. The simplest and most efficient interface for querying a complex graph of structured data is a natural language interface. For example:

```
Computer scientists I am connected to who work with people who graduated
from my college last year and do not live in St. Louis.
```

Such a system needs to be absolutely robust. This entails handling ill-formed queries (including traditional keyword searches not in the form of a proper query) and queries that have different linguistic forms but are semantically identical.

The proposed interface parses the user's input and outputs the *k*-best suggestions in natural language:

```
input: People who work nearby
output:
   People who work nearby
   People who work nearby from St. Louis
   Connections of people who work nearby
   People who work nearby and work at Washington University in St. Louis
```

Each suggestion expresses the system's interpretation of the user's exact intention. By outputting the *k*-best suggestions, the user knows whether their query has been correctly understood before he or she executes the search. The user also knows the parser will correctly interpret any suggestion the user chooses, even if it does not match the input, as with ill-formed queries:

```
input: People who nearby
output:
   People who live nearby
   People who work nearby
   People who work near Washington University in St. Louis
   People who work nearby and live in St. Louis
```

This solves a common problem of natural language interfaces: the user does not know if an interaction (input) is successful until after execution, which can consume more time than the manual alternative if the parser fails. The output also includes suggestions for completing the search as it is being input, providing a real-time experience:

```
input: People wh
output:
   People who I am connected to
   People who attend Washington University in St. Louis
   People who work at Washington University in St. Louis
   People who live in St. Louis
```

This demonstrates what queries are possible and allows the user to accelerate his or her input.

While this system is introduced in the frame of searching complex graphs, additional natural language interfaces can be built with it. For example, the system can be applied to productivity software: "`schedule a meeting with John every Tuesday at 4:00 in the afternoon`", and "`remind me to take out the trash at sundown`". Searching complex graphs, however, is the most important and exciting problem this project solves.

2 Grammar

A weighted context free grammar (WCFG) defines the queries that the parser can understand. The grammar is composed of non-terminal rules, which form a parse tree's syntactic structure, and terminal rules, which are the actual items the parser will expect to see when parsing input. Figure 1 contains an example grammar simplified for demonstration purposes. This includes the reduction of production rules necessary for the exemplified queries and the omission of rule

```
“nonterminal_rules”: {
    "<start>": [
        [ "<users-start>" ]
    ],
    "<users-start>": [
        [ "<people>", "<user-filter>" ]
    ],
    "<user-filter>": [
        [ "<who>", "<employer-filter>" ],
        [ "<who>", "<residence-filter>" ]
    ],
    "<employer-filter>": [
        [ "<work-current>", "<employer-location>" ],
        [ "<work-current>", "<employer-subject>" ]
    ],
    "<employer-location>": [
        [ "<in-work>", "<location>" ]
    ],
    "<employer-subject>": [
        [ "<at-work>", "<employer>" ]
    ],
    "<residence-filter>": [
        [ "<live-in>", "<location>" ]
    ],
    "<location>": [
        [ "<city>" ]
    ]
},
“terminal_rules": {
    "<people>": [ "people" ],
    "<who>": [ "who" ],
    "<work-current>": [ "work" ],
    "<in-work>": [ "in" ],
    "<at-work>": [ "at", "for" ],
    "<live-in>": [ "live in" ],
    "<city>": [ "{city}" ],
    "<employer>": [ "{employer}" ]
},
“start_symbol": “<start>"
```

Figure 1: A simplified natural language CFG.

costs. In production, each rule has a predefined cost associated with it, which together constitute the grammar’s weights. For example, an obscure phrase, e.g., “`resident of`”, will have a higher cost than a more common phrase, e.g., “`live in`”.

3 Language Models

The terminal rules of the grammar consist of entities as well as natural language terms. Entities are organized into entity categories, such as `{employer}`, `{college}`, and `{city}`, each of which is a search index for a database of its entities. The natural language terms, such as “`who`”, “`and`”, and “`work at`”, are also stored in a single search index for the query language. To save having to search each index for matches to the input query, *n*-gram based language models are used to identify which indices each query segment has a high probability of belonging to.

The language models are comprised of conditional probabilities of observing the *i*-th word in the context of the preceding *i*-1 words, estimating the likelihood a sequence of words is produced by a source. Smoothing parameters are also applied to the models. *n*-gram statistics constitute each model, either for a specific entity category or for the query language. For example:

- The bigram `new+york` has a high probability in the `{city}` language model and a low probability in the `{employer}` language model.

- The trigram `work+at+{employer}` has a high probability in the query language model.

The system can use these language models to calculate the probability of any term (i.e., symbol) in an input query being stored in any term index.

4 Lexical Analysis

The grammar for this system is designed to be robust, allowing the user to query for any set of results in various ways. This is attained through lexical analysis of the input before matching it to terminal rules.

Using Princeton University's WordNet, a list of synonyms of terms in the query language is obtained that can be substituted in the input. This allows the parser to handle, for example, both "`universities close by`" and "`colleges nearby`", and output the same results. WordNet is further leveraged to find related forms of terms in the grammar, allowing "`people who study at {college}`" to also be represented as "`students at {college}`".

In order to extract the meaningful portions of a query for the parser, particular words are identified as optional in particular contexts:

- `people who have lived in St. Louis => people who lived in St. Louis`

- `the companies that I have worked for => companies I worked for`

In the first example, "`the`" is disregarded when preceding the head noun in "`the companies`". The term should not be disregarded in other contexts, however, such as "`people who work in the`", which can be completed to "`people who work in theater`".

Morphological analysis accounts for the inflected forms of the words in the grammar's terminal rules, which are also provided by WordNet. Some inflections do not change meaning, e.g., "`employer`" and "`employers`", while others do, e.g., "`people who attend {college}`" opposed to "`people who attended {college}`".

By conducting lexical analysis, the parser only outputs suggestions that are grammatically correct. Each edit made via these operations, such as changing a term's inflection to the appropriate tense, yields a cost for the respective terminal rule when matched. Ultimately, this allows the parser to produce plausible suggestions for any reasonable user input.

5 Parser

Using the output of the language models and lexical analysis, the input query is executed against the appropriate search indices which retrieve and rank database entries using the matched input text. The entries serve as terminal rules in the grammar and form the actual display text of the query suggestions. To achieve quick lookup of terms and entities, all search indices are served entirely from RAM. The matched terminal rules, along with their matching costs and starting and ending positions in the input query, are sent to the parser.

The parsing algorithm is the focus of this project. Dozens of papers have been consulted and numerous algorithms have been tested. The parser encompasses 95% of this project's work and 100% of its challenge. For this reason, this section will be significantly expanded when complete, relative to the other sections of the paper. This includes thorough explanations of the design of the algorithm(s), data structures, and heuristics used.

```
0:
    <start> => 1
    <users-start> => 2
    <people> => 3
1:
    accept
2:
    [<start> -> <users-start>]
3:
    <user-filter> => 4
    <who> => 5
4:
    [<users-start> -> <people> <user-filter>]
5:
    <employer-filter> => 6
    <residence-filter> => 7
    <work-current> => 8
    <live-in> => 9
6:
    [<user-filter> -> <who> <employer-filter>]
7:
    [<user-filter> -> <who> <residence-filter>]
8:
    <employer-location> => 10
    <employer-subject> => 11
    <in-work> => 12
    <at-work> => 13
9:
    <location> => 14
    <city> => 15
10:
    [<employer-filter> -> <work-current> <employer-location>]
11:
    [<employer-filter> -> <work-current> <employer-subject>]
12:
    <location> => 16
    <city> => 15
13:
    <employer> => 17
14:
    [<residence-filter> -> <live-in> <location>]
15:
    [<location> -> <city>]
16:
    [<employer-location> -> <in-work> <location>]
17:
    [<employer-subject> -> <at-work> <employer>]
```

Figure 2: A shift-reduce parse table generated from the grammar in Figure 1.

The parser is a GLR parser, where G stands for “generalized”, L stands for “left-to-right”, and R stands for “rightmost derivation”. It efficiently handles nondeterministic, cyclic, and ambiguous context free grammars, as is the case for natural language grammars, and has a worst-case time complexity of $O(n^3)$, where n is the number of input tokens (i.e., symbols). Unlike traditional LR parses, GLR uses breadth-first search to handle all possible interpretations of a given input. The parser introduced here extends the algorithm to output the k-best parse trees instead of a single tree.

The parser first constructs a shift-reduce parse table, a type of state transition table, from the context free grammar. See Figure 2 for an example of a parse table. This initialization is separate from the parsing, during which the parser processes input through a combination of shift steps and reduce steps, under the direction of the parse table:

- A shift step advances in the input query by one terminal symbol, which itself becomes a new single-node parse tree.

- A reduce step applies a completed production rule to previously constructed parse trees, merging them into one tree with a new root symbol.

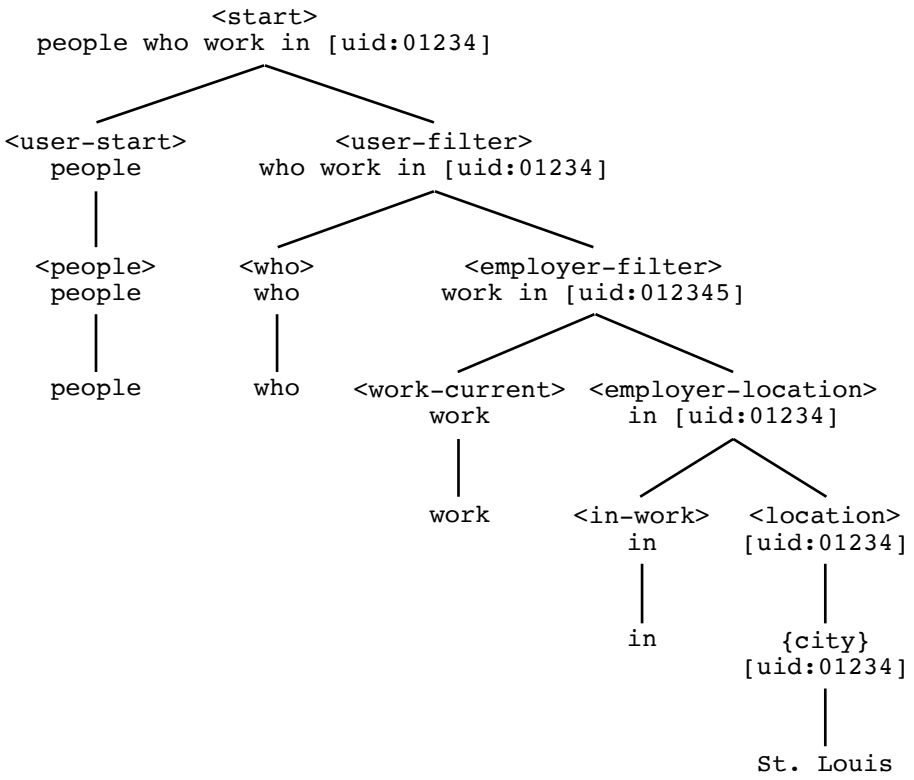

Figure 3: A parse tree constructed from the parse table in Figure 2.

Parse trees are built incrementally, bottom-up (i.e., from terminal rules to the `<start>` symbol) and left-to-right, without looking ahead (i.e., LR(0)) or backtracking (i.e., online). Each shift step is followed by a reduce step, after which the parser has accumulated a list of subtrees that have been already parsed. The subtrees are not combined until the parser reaches the right end of the production rule.

Given a state and input symbol, this parse table allows for multiple state transitions where traditional parse tables allow for only one transition. This allows for shift/reduce and reduce/reduce conflicts. When the parser encounters a conflicting transition, it forks the parse stack into two or more stacks. The previous node becomes a packed-node and the state corresponding to each possible transition becomes a sub-node and resides at the top of each these stacks. The next shift step determines the next transition(s) for each of the sub-nodes, where further forking can occur. If the input symbol and a given sub-node's state do not have at least one possible transition (in the parse table), then the stack cannot produce a valid tree and is discarded.

The algorithm also is significantly optimized by sharing common nodes in the parse stacks. When forking a stack, each packed-node shares identical parent and child nodes instead of duplicating the whole stack. This improvement makes the search graph a directed acyclic graph-structured stack rather than a tree. These complex structures constrain the search space and memory usage required to parse the input.

After each reduce step, the subtrees that were constructed constitute a parsing state, which the parser then pushes to a heap to implement k-shortest path routing . As the parser advances, it calculates the costs of each parser state it pushes to the heap, which sort according to the costs, using the following parameterization:

1. Rule costs defined in the WCFG.

2. Matching costs of query segments with terminal rules based on lexical analysis. This includes costs for matching synonyms, varying inflections, misspellings, etcetera.

3. Editing costs for insertion, deletion, substitution, and transposition of each query segment necessary to build each parse tree (i.e., suggestion) from the input query.

The parser continues with the shift-reduce steps until it has parsed *k* legal trees. To be legal, a tree must reach the `<start>` symbol and consist of consecutive, non-overlapping terminal symbols that comprise the entire input. See Figure 3 for an example of a parse tree. In production, the parse tree would be considerably larger for an identical query as the actual grammar will be much larger to accommodate numerous possible inputs.

Each production rule is associated with a semantic function. In the parse tree in Figure 3, the nonterminal rule `<employer-location>` yields `employer-location(01234)`. As a result, each query's parse tree is also its semantic tree. Ultimately, "`people who work in St. Louis`" outputs the following semantic which can be executed against a database search index:

```
"object_type": "user",
"filter": {
   “employee_current": true,
   "employer_location": 01234
}
```

6 Project

As stated, the parser is the focus and inventive portion of the project; the natural language grammar, language models, and lexical analysis are applications of existing, well-documented techniques that only add application to the parser. Everything proposed in this paper is subject to, and likely will, change. Arriving at this design was a long, arduous process of trial-and-error, research, and required starting over from scratch several times. I would be surprised if I do not run into more issues in the future.

Currently, I have a prototype parsing table generator and GLR parser. The parser has not yet been extended to output the *k*-best parse trees, does not address several cases and necessary constraints, and is far from the speed necessary to be real-time (i.e., output results as input by the user). Despite GLR parsing and parse tables being existing techniques in computer science, their designs will merely be adapted while the final parser will be largely of my own invention. This project is substantive, original, and certainly can occupy the two semesters.