Handling Complex Queries Using Query Trees

Vemuru Srihari, Eric John, Shrisha Rao
International Institute of Information Technology Bangalore,
Bangalore 560100, India

Abstract

Humans can easily parse and find answers to complex queries such as "What was the capital of the country of the discoverer of the element which has atomic number 1?" by breaking them up into small pieces, querying these appropriately, and assembling a final answer. However, contemporary search engines lack such capability and fail to handle even slightly complex queries. Search engines process queries by identifying keywords and searching against them in knowledge bases or indexed web pages. The results are, therefore, dependent on the keywords and how well the search engine handles them. In our work, we propose a three-step approach called parsing, tree generation, and querying (PTGQ) for effective searching of larger and more expressive queries of potentially unbounded complexity. PTGQ parses a complex query and constructs a query tree where each node represents a simple query. It then processes the complex query by recursively querying a back-end search engine, going over the corresponding query tree in postorder. Using PTGQ makes sure that the search engine always handles a simpler query containing very few keywords. Results demonstrate that PTGQ can handle queries of much higher complexity than standalone search engines.

1 Introduction

Search engines aim to provide a relevant set of high-quality search results that fulfill the user's query as immediately as possible. Studies [1, 2, 3] show three out of every four queries submitted to the web search engines either contain entities or aim to find information about entities. Search engines primarily use knowledge graphs to answer such queries, which specialize in finding concrete or abstract objects (people, organizations, dates, etc.) as opposed to the documents or blogs. Users are willing to express such information more elaborately rather than with a few keywords. Furthermore,

advances in automatic speech recognition (ASR) simplifies stating of the search queries [4, 5].

Commercial search engines use a combination of knowledge graphs with web crawling, indexing, and page ranking to achieve good results for queries based on entities. The first component knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge [6]. We can think of an ontology as a giant encyclopedia containing as much as a billion facts. The ontology or knowledge base represents a knowledge domain characterized by high semantic expressiveness and constructed under domain experts. A knowledge graph reasoner is an AI-enabled system that infers facts from an underlying knowledge base using a set of established rules or axioms. The reasoner derives new knowledge from the ontology through deduction or induction. A search engine with a knowledge graph does not search for an input string. It identifies the specified keywords and searches in the knowledge base for schemas that include them. All relevant data items associated with schemas that match get included in the result ordered on priority [7]. However, the possibility of information overload makes it difficult to build the optimal knowledge base and update the connections to accommodate the exponential increase in data [8].

Search engines are used by millions of users every day to find information, and the method of searching continues to be the same as when the first search engines appeared years ago, relying mainly on keyword search [9]. It has resulted in unsatisfactory search results, as a simple keyword cannot always convey the complex search semantics a user wishes to express, returning irrelevant information and eventually disappointing users. A highly expressive query, on the contrary, is long and requires more keywords to express it [10]. The knowledge graph reasoner may end up choosing the wrong set of relevant keywords from such long queries. The search fails to produce correct results if the reasoner operates with the wrong set of relevant keywords. By breaking down a search query into smaller pieces and giving the individual components to the knowledge graph reasoner, one can make sure that the number of keywords the reasoner works with are always small. Moreover, the chance of a search failing due to looking up irrelevant keywords in the knowledge base becomes lower. Query splitting techniques are therefore relevant in searching [11].

Several works [12, 13, 14] in the past have discussed methods to convert long and complex sentences into blocks of smaller ones to make them easier to process [15]. Narayan et al. [12] proposed a sentence simplification task: *split and rephrase* that splits a complex sentence into a sequence of shorter

sentences that preserves meaning. Vu et al. [13] used an architecture with augmented memory capabilities called neural semantic encoders for sentence simplification. Wang et al. [14] introduced a separator network capable of selecting semantic components from the source sentence. The separator network, along with a seq2seq model [16], avoids duplication and detects clues while splitting sentences. Zhang et al. [17] proposed a multistage encoder-based seq2seq model for sentence simplification. Guo et al. [18] proposed a seq2seq model that uses fact-aware sentence encoding that enables the model to learn facts from long sentences to improve the precision of split and permutation invariant training to order the sentences.

Cui et al. [19] considered the utilization of the dependency relationship amongst words for question answering. They used fuzzy relation matching over term-density ranking to improve the answering capabilities of their model. Tur et al. [20] presented a parsing-based sentence simplification method and demonstrated its use in intent determination and slot filling tasks in spoken language understanding (SLU) systems. Das et al. [21] proposed an algorithm called SSG for sentence simplification, which forms smaller sentences by analyzing the dependency tree [22] of the input sentence.

These works use deep-learning techniques (like seq2seq models, decoder-encoder models, etc.) or dependency parsing coupled with other methods (like term-density ranking) to construct simple sentences. But these methods omit relations that exist between the words present in the simple sentences and are not practical for automated searching.

We use dependency parsing to split a complex search query into smaller queries. However, unlike other methods, we use syntax analyzers and deterministically construct the simple queries. We then construct a query tree corresponding to the simple queries, which holds the relations of the words present in the complex search query.

The method is simple, fast, and extendable to queries of arbitrary complexity, unlike other methods. We formally introduce query complexity as a measure of the complexity of a search query and use it to rank queries of different complexities. Moreover, we have a set of 1000 queries that we make available for further work (see Section 3.1).

Our work, *parsing, tree generation, and querying* (PTGQ), is a three-step approach for processing complex queries. PTGQ breaks the search query into smaller pieces at relevant positions, orders these into its corresponding query tree, and processes it recursively with a search engine. This ensures that the reasoner always works with a few keywords.

The three steps involved in PTGQ are:

- **Dependency Parsing:** This generates a dependency tree from the search query that describes the structure of sentences and the relationships among the words present as a tree. (See Section 2.1).
- **Query Tree Construction:** This iterates over the dependency tree to identify keywords using syntax analyzers, generates corresponding simple queries, and constructs a query tree with them. Each node in the query tree (Section 2.2.3) corresponds to a simple query. The query tree obtained at the end of this step is answerable in postorder. (See Section 2.2).
- **Progressive Querying:** This recurses using a search engine over the query tree in postorder to answer each node present in the tree. Each node in the query tree waits for answers from its children, adds them to its corresponding simple query, and searches this newly formed query in a search engine. The result of the root node is the answer to the search query. (See Section 2.3).

A search query like “*Who is the discoverer of the element which has the atomic number 1?*” is not answerable by current search engines. PTGQ converts the query into a tree with the root as “*Who is the discoverer of _*” and its child as “*the element which has atomic number 1*”. PTGQ answers the tree in postorder to get “*Henry Cavendish*” as the answer. Our method works with complex queries written in natural languages.

PTGQ operates with a conventional search engine at the backend. It massively augments the capability of a search engine. To prove this, we compile a detailed set of test cases, distributed as shown in Table 3, which represents search queries that our implementation of PTGQ handles. We explain the formulation of test cases in detail and show how these capture several complex scenarios. Moreover, we segregate the test cases based on a new measure which we call as query complexity (see Section 2.2.3). We integrate PTGQ with a search engine and compare its performance with two stand-alone search engines. Results show that PTGQ can significantly improve the performance of a search engine (see Section 3). The stand-alone search engines show a drop in the fraction of passed test cases with the increase in query complexity and struggle to pass test cases with query complexities higher than three. As PTGQ processes a query using a sequence of searches defined by the query tree, it continues to perform well for even higher query complexities, highlighting the relevance of the approach to query parsing and processing.

2 Parsing, Tree Generation, and Querying (PTGQ)

This section explains the architecture of *parsing, tree generation, and querying* (PTGQ). PTGQ takes as input a search query and outputs its search result by first converting the search query into its corresponding dependency tree, then converting the dependency tree into its corresponding query tree, and finally recursively solving the query tree. Figure 1 shows the major components in the construction of PTGQ.

Figure 1: Architecture of PTGQ

2.1 Dependency Parsing

Dependency parsing [22] refers to the process of analyzing the grammatical structure of a language to identify the relationships amongst the words of a sentence. A popular method of representation is the dependency tree; dependency parsing mentioned here, constructs a dependency tree from the input query.

Figure 2 shows the dependency tree with the nouns, verbs and wh-words highlighted for the query "What was the capital of the country of the discoverer of the element which has the atomic number 1?" Each node contains a word in the query along with its part-of-speech tagging as per the Penn Treebank Project [23]. Additionally, each edge in the dependency tree represents a dependency between two nodes it connects.

2.2 Query Tree Generation

Query tree generation constructs a query tree from the dependency tree of a search query. Figure 3 shows the components of the query tree generation. *Keyword identification* identifies the keyword pairs that correspond to a

Figure 2: Dependency tree for the query "What was the capital of the country of the discoverer of the element which has the atomic number 1?"

simple search query. *Simple query generation* as the name suggests generates simple queries from the keyword pairs identified. *Query tree construction* constructs a query tree corresponding to these simple queries.

Section 2.2.1 deals with keyword identification, Section 2.2.2 deals with simple query generation, and Section 2.2.3 deals with query tree construction in detail.

Before going over the components in detail, we introduce a few terms that are used throughout this paper.

Elementary Query: We call a search query containing a single noun as an *elementary query*.

The query can be a single noun or proper noun: "*Alan Turing*", noun modifiers followed by a noun: "*oldest living person*", or a simple sentence with the subject as a noun or proper noun: "*Which is the tallest mountain?*"

Figure 3: Components of Query Tree Generation

Paired Query: We call a search query constructed from two elementary queries that are present within a search query a *paired query* if:

- the noun in one of the elementary queries is an ancestor of the other in the corresponding dependency tree of the search query; and
- the path between them in the dependency tree does not contain any other nouns.

Consider the dependency tree in Figure 2. "capital" is the ancestor of "country" and so "the capital of the country" is the corresponding paired query for the elementary queries "the capital" and "the country". Similarly, "the country of the discoverer", "the discoverer of the element", and "the element which has the atomic number 1" are paired queries from the search query.

Simple Query: We call a search query a *simple query* if it contains at most one paired query.

Simple queries containing a paired query can be identified using the two nouns that constitute them. In the example above (*capital, country*) identifies the simple query "the capital of the country". We use the term *keyword pair* to refer to the identifying nouns of a simple query.

Complex Query: We call a search query a *complex query* if it contains more than one paired query.

2.2.1 Keyword Identification

This process identifies the keyword pairs that correspond to simple queries within a query. It takes as input the rooted n -ary dependency tree, recurses over the tree, identifies the keyword pairs for each simple query present using *syntax analyzers*, and outputs the keyword pairs corresponding to the simple queries. Each analyzer identifies paired queries connected by a specified set of words, and stores the corresponding keyword pairs. For example, a syntax analyzer for prepositions identifies paired queries whose elementary queries are joined using prepositions.

Algorithm 1 for keyword identification works as follows. It iterates over the dependency tree in preorder, inorder, and postorder. The *left_children* (line 19) of a node corresponds to the words that are dependent on the node and appear before it in the search query. The *right_children* (line 22) of a node corresponds to the words that are dependent on the node and appear after it in the search query. For each node, `keywordIdentifier()` (line 16) visits, it calls `preorder()` (line 1) if it is visiting the node for the first time, `inorder()` (line 6) if it has completed visiting all of the left children, and `postorder()` (line 11) if it has completed visiting all of the left and right children.

The functions `preorder()`, `inorder()`, and `postorder()` iterate over the syntax analyzers and update the state of the analyzers (lines 3, 8, and 13). The state updates help each analyzer to identify the keyword pair corresponding to a simple query. For the example in Figure 2, keyword identification identifies the pairs *(element, number)*, *(discoverer, element)*, *(country, discoverer)*, *(capital, country)*. Moreover, special analyzers for wh-questions identify *(was, capital)* and *(has, number)* in addition. These analyzers identify interrogative sentences and are relevant in processing queries containing them.

Our implementation of a syntax analyzer for prepositions does the following. In `postorder()` it checks for a node with a noun tag having a preposition as its parent node. If it finds one such node it marks the node. If the analyzer already has a node marked, then it tries to find the nearest ancestor node that is a noun in further `postorder()` calls. If the analyzer finds one such node, it stores the same along with the marked node as a keyword pair for prepositions.

The `keywordIdentifier()` function visits each node thrice—preorder, postorder, and inorder—and calls all the syntax analyzers on the node. Each syntax analyzer checks if the node conforms to a condition and updates its state. Finally when the state of the analyzer meets a favorable condition

Algorithm 1: Keyword Identification

Input: root node of the dependency tree

Output: keyword pairs for simple queries

```

1 preorder(node, parent):
2   for analyzer in syntax analyzers do
3     analyzer.preorder_state_update(node, parent) // update
        states of each analyzer when nodes of a keyword
        pair are found
4     if valid pair identified then
5       analyzer.store_pair() // analyzer has identified a
        keyword pair corresponding to a simple query
6 inorder(node, parent):
7   for analyzer in syntax analyzers do
8     analyzer.inorder_state_update(node, parent)
9     if valid pair identified then
10      analyzer.store_pair()
11 postorder(node, parent):
12   for analyzer in syntax analyzers do
13     analyzer.postorder_state_update(node, parent)
14     if valid pair identified then
15      analyzer.store_pair()
16 keywordIdentifier(node, parent = None):
17   if node.children > 0 then
18     // has reached an intermediate node
19     preorder (node, parent)
20     for child in node.left_children do
21       // visit each dependent child appearing before
22       the current node
23       keywordIdentifier(child, node)
24     inorder (node, parent)
25     for child in node.right_children do
26       // visit each dependent child appearing after
27       the current node
28       keywordIdentifier(child, node)
29     postorder (node, parent)
30   else
31     // has reached a leaf node
32     preorder (node, parent)
33     inorder (node, parent)
34     postorder (node, parent)

```

the keyword pairs are stored. The complexity of keyword identification, therefore, is $O(n.z.k)$, where n is the number of nodes, z the number of analyzers, and k the cost of string operations.

2.2.2 Simple Query Generation

This process constructs simple queries from the pairs that keyword identification outputs for a dependency tree. It takes the keyword pairs along with the dependency tree as input, recurses over the tree, and compares each node with the stored values of keyword pairs. If a node in the dependency tree visited during preorder happens to be the first element of a keyword pair, then the analyzer corresponding to the keyword pair starts constructing a string. The string stores all the words that the following inorder traversal visits until reaching the second element. This string is the simple query corresponding to the keyword pair. Algorithm 2 for simple query generation constructs the queries as follows.

The `queryGenerator()` (line 11) function iterates over the dependency tree in preorder and inorder. For each node `queryGenerator()` visits, it calls `generateQueryPreorder()` (line 1) if it is visiting the node for the first time, and `generateQueryInorder()` (line 6) if it has completed visiting all of the left children. The `generateQueryPreorder()` function internally calls all syntax analyzers to check if the node corresponds to the first element of any of the keyword pairs that they contain. If so, then that particular syntax analyzer marks the beginning of a new string.

The `generateQueryInorder()` function similarly iterates over all the syntax analyzers. If an analyzer had previously marked a beginning, then during `generateQueryInorder()` the analyzer adds to the string the nodes that it visits until the second element of the keyword pair. At that particular point, the analyzer commits the string. Each analyzer can ultimately store multiple simple queries. Moreover, `queryGenerator()` makes sure that the analyzers construct the simple queries in the same order as they are present in the search query.

Consider the keyword pair (*capital*, *country*) routine that the preposition syntax analyzer has. When `generateQueryPreorder()` visits the node "*capital*" in preorder, it calls the analyzer for prepositions. The analyzer identifies that the node "*capital*" is part of a keyword pair. When `generateQueryInorder()` calls the analyzer for preposition during further iterations, the analyzer stores the nodes visited in a string until it finds the node "*country*" that is the other element in the keyword pair. The string formed is a simple query for the keyword pair (*capital*, *country*).

Algorithm 2: Simple Query Generation

Input: root node of the dependency tree, keyword pairs

Output: strings of simple queries

```

1 generateQueryPreorder(node):
2   for analyzer in syntax analyzers do
3     analyzer.preorder_state.update(node) // update the state
        on finding starting keyword
4     if addable node then
5       | add node to analyzer.simple_query[]

6 generateQueryInorder(node):
7   for analyzer in syntax analyzers do
8     analyzer.inorder_state.update(node) // update the state
        on finding ending keyword
9     if addable node then
10      | add node to analyzer.simple_query[]

11 queryGenerator(node):
12   if node.children > 0 then
13     // has reached an intermediate node
14     generateQueryPreorder(node) // preorder
15     for child in node.left_children do
16       | queryGenerator(child)
17     generateQueryInorder(node) // inorder
18     for child in node.right_children do
19       | queryGenerator(child)
20   else
21     // has reached a leaf node
22     generateQueryPreorder(node) // preorder
23     generateQueryInorder(node) // inorder

```

The `queryGenerator()` function visits each node exactly twice—preorder and inorder—and calls all the syntax analyzers on the node. The syntax analyzers on each call check if the node is present in the keyword pairs that they contain, and update the states of the analyzers. If so, all following nodes are stored in a string until the analyzer visits the other node in the keyword pair. The complexity of simple query generation is $O(n.z.k)$, where n is the number of nodes, z the number of analyzers, and k the cost of string operations.

2.2.3 Query Tree Construction

This process constructs a query tree using the simple queries identified by syntax analyzers and the original search query. Before going over the steps in query tree construction, we first formally introduce a query tree.

Query Tree for a Search Query: A tree $G_z \equiv (V_z, E_z)$, where V_z is the set of vertices, and $E_z \subseteq V_z \times V_z$ the set of edges, is called a *query tree* for a search query z if:

- every vertex $v_z \in V_z$ represents a simple query; and
- every edge $e_z \in E_z$ signifies a relationship between any two of the identifying keywords present in the simple queries that it connects.

Query Complexity of a Query Tree: The query complexity of a query tree $G_z \equiv (V_z, E_z)$ for a search query z is the count of the number of simple queries present in the query tree. The query complexity c_z of the query tree G_z is

$$c_z = |V_z|$$

Consider the previous example: *"What was the capital of the country of the discoverer of the element which has the atomic number 1?"* Its query complexity is 4. Its equivalent query tree is shown in Figure 4. On close observation, we see that each intermediate node is holding only one keyword from the keyword pair, while the leaf node contains both keywords. The leaf node is queriable as is, but the intermediate nodes require a few more words to represent a simple query. The additional words required are the search results of all the subtrees of the intermediate node. (See Section 2.3).

During simple query generation, each analyzer works disjointly. The simple queries identified by two separate analyzers, therefore, can contain

Figure 4: Example of a query tree

the same words. Moreover, the queries are stored based on the order of their appearance in the search query.

Algorithm 3 for query tree construction works as follows. The `treeConstructor()` function iterates over each word in the search query (line 3). The syntax analyzers take the first word in their ordered list of simple queries and check if the two words match. We call an analyzer that passes the condition above as *active* (line 4).

If none of the syntax analyzers are active, then `treeConstructor()` appends the word to the current node and continues execution (line 6).

If at least one active analyzer exists, then the `treeConstructor()` function chooses the analyzers with the maximum priority (line 5). Assume that `treeConstructor()` chose an analyzer with priority p during the iteration of the previous word and another analyzer with priority p' in the current iteration.

Case 1: When $p' \geq p$ and the word encountered happens to be the starting word in the current simple query of one of the active analyzers with maximum priority. The `treeConstructor()` function has encountered an active analyzer with the word as the start of the analyzer's current simple query. The `treeConstructor()` function adds a new node to the current node in the query tree and appends the word to this node (line 9).

Case 2: When $p' = p$ and the word encountered is not the starting word in the current simple query of any of the active analyzers with maximum priority. The word corresponds to the continuation of the simple query already under construction. Therefore, the `treeConstructor()` function

Algorithm 3: Query Tree Construction

Input: search query, strings of simple queries

Output: query tree for search query

```

1 treeConstructor(search_query, simple_queries[]):
2   prev_priority = 0 // initialize priority
3   for word in search_query do
4     // find all active analyzers
5     active_analyzers = {analyzers} such that word =
        simple_queries[analyzers].curr_query.curr_word // find
        analyzers having the word in its simple query
6     max_active_analyzers, curr_priority = max(priority of
        active_analyzers) // find all active analyzers with
        max priority
7     if active_analyzers =  $\emptyset$  then
8       // no active analyzers
9       query_tree.add_to_curr_node(word)
10      continue
11     else if curr_priority  $\geq$  prev_priority and start of simple
        query in one of the max_active_analyzers then
12       query_tree.add_new_node(word, curr_priority)
13     else if curr_priority = prev_priority and not start of simple
        query in any of the max_active_analyzers then
14       query_tree.add_to_curr_node(word, curr_priority)
15     else if curr_priority < prev_priority and not start of simple
        query in any of the max_active_analyzers then
16       query_tree.add_to_ancestor(word, curr_priority)
17     else if curr_priority < prev_priority and start of simple
        query in one of the max_active_analyzers then
18       query_tree.add_new_node(word, curr_priority)
19   prev_priority = curr_priority // update priority
20   remove simple_queries[active_analyzers].curr_query.curr_word
21   // remove the word from all active analyzers

```

appends the word to the current node (line 11).

The case, $p' > p$ but the word encountered is not the starting word cannot arise because an increase in priority necessarily means a switch from one query to another. When $p' > p$, the `treeConstructor()` function always switches queries (line 9).

Case 3: When $p' < p$ with the word encountered not being a starting word in the current simple query of any of the active analyzers with maximum priority. The `treeConstructor()` function, similar to Case 2, has encountered the continuation of a simple query already under construction. However, the `treeConstructor()` function appends the word to an ancestor node in the query tree (line 13). While Case 2 handles the addition of a word from the same simple query, this particular case handles the addition of the remaining words of a simple query visited earlier. This case can arise if a simple query exists within a query identified by a special analyzer.

Case 4: When $p' < p$ and the word encountered is a starting word in the current simple query of one of the active analyzers with maximum priority. Similar to Case 1, the `treeConstructor()` function adds a new node to the query tree and appends the word to the node (line 15). Here the highest priority analyzer in the previous step has completed adding its simple query, and now in the current iteration, `treeConstructor()` has encountered the start of a new simple query in another analyzer. After each iteration, each analyzer removes the word encountered from its current simple query. (line 18).

Adding to a node, however, depends on a few more considerations. After simple query generation completes execution on the example in Figure 2, all analyzers contain the following simple queries. The analyzer for wh-questions has the simple queries *"What was the capital of the element"* and *"which has atomic number 1"*. This special analyzer identifies the sentences which contain one noun and ask an interrogative question. The keyword pair for this analyzer identifies *(verb, noun)* corresponding to the sentence. Additionally, it continues to store the nodes it visits during simple query generation until it reaches the beginning of a new wh-question. The analyzer for relational pronouns has the simple query *"the element which has the atomic number 1"*. The analyzer for prepositions has the simple queries: *"the capital of the country"*, *"the country of the discoverer"*, and *"the discoverer of the element"*.

We answer search queries with prepositions and relative pronouns by answering the simple queries *right-to-left*. That is, a complex query like *"the capital of the country of Henry Cavendish"* is answered by first answering the simple query *"the country of Henry Cavendish"* and then *"the capital of*

" appended with the answer. However, for search queries with possessive endings, we have to answer them *left-to-right*. For example, consider the complex query *"Henry Cavendish's country's capital"*. We answer *"Henry Cavendish's country"* first followed by *"'s capital"*. Note that, as each analyzer stores the simple queries in the same order as they are found in the search query, the answering order of each analyzer can either be right-to-left or left-to-right.

Sometimes we have to combine two simple queries that share a noun. For example, in the query tree, constructed from the dependency tree in Figure 2, we construct *"What was the capital of _"* as one node. Although `treeConstructor()` identifies this part as two separate queries: *"What was _"* with the child node as *"the capital of _"*, we combine them because the second search is redundant. But the query tree for the search query *"When was the president of the USA born"* has to be two separate simple queries *"When was _ born"* with the child node as *"the president of the USA"*. The distinction arises because the verbs *"was"* and *"born"* identified by the analyzer for wh-questions during simple query generation have different relevance in actual searching.

The `treeConstructor()` function in query tree construction creates a query tree, say Q , answerable in postorder. For prepositions and relative pronouns that are answerable right-to-left, `treeConstructor()` adds the simple queries that appear later as a descendant of the current node in the query tree Q . Recursing in postorder makes sure that the simple queries that appear later are answered first. However, possessive endings cannot be added in the same way, since they are answerable left-to-right. To overcome this issue, whenever we reach a simple query on possessive endings, we start the construction of a separate query tree, say Q' . Subsequent simple queries on possessive endings get added as a parent node instead of as a child to Q' . When a simple query of any other analyzer starts constructing, this additional tree Q' constructed for possessive endings is added as a child to Q . Recursing over the query tree Q in postorder will therefore answer the search query in the correct order.

Figure 5 shows the major steps during the formation of a query tree for another example: *"What was the USA's 37th president's age at the death of Jawaharlal Nehru's daughter"*. Steps 2 and 7 show the creation of a separate query tree in the case of encountering prepositions. Step 3 shows the addition of another preposition as a parent node to the new query tree. Steps 4 and 8 show the result of the two query trees after combining them. Step 4 additionally removes a redundant search by combining two nodes.

The `treeConstructor()` function visits each word in the search query

Figure 5: Steps of query tree construction for the query: What was the USA's 37th president's age at the death of Jawaharlal Nehru's daughter

Figure 6: Components of Progressive Querying

exactly once and finds the analyzer with the highest priority that has the word in the simple queries it contains. Each addition of a node to the tree can be done in constant time by maintaining additional pointers to the root node, current node and incomplete ancestor nodes. For possessive endings, the analyzer creates a new query tree or adds the node to an already existing query tree. All other syntax analyzers either add one word to the current node or create a new child node and add the word to it. The complexity of query tree construction therefore is $O(n.z.k)$, where n is the number of words, z the number of analyzers, and k the cost of string operations.

2.3 Progressive Querying

This process takes the query tree as input and outputs the result of the input query. As mentioned in Section 2.2.3, query tree generation constructs a query tree answerable in postorder and progressive querying recursively solves the tree. Figure 6 show the components of progressive querying.

Algorithm 4 for progressive querying executes as follows. The `progressiveSearcher()` function iterates over the query tree in postorder (line 1). During recursion, on reaching an intermediate node, `progressiveSearcher()` adds the results of the child nodes to the simple query stored there (lines 4 and 6). After addition, it calls the search engine with the simple query in the current node as the search query (line 6). Finally, it returns the result from the search engine to the parent of the current node. The search engine answers the leaf nodes as-is, as they do not require any processing (line 8).

At the end of the recursion, the search engine searches the simple query formed in the root node. The output of the search engine for the root node is the result of the query tree. Figure 4 shows the above process for an

Algorithm 4: Progressive Querying

Input: root node of the query tree

Output: search result of the query

```

1 progressiveSearcher(query_node):
2   simple_query = "" // initialize simple query
3   if query_node.children > 0 then
4       // has reached an intermediate node
5       for child in query_node.children do
6           // add results of children to simple query
7           concat (simple_query, progressiveSearcher(child))
8           // add contents of node to simple query
9           concat (simple_query, query_node.string) return
10          search_engine(simple_query)
11   else
12       // has reached a leaf node
13       return search_engine(query_node.string)

```

example.

As the process iterates in postorder, the `progressiveSearcher()` function first encounters the simple query *"the element which has the atomic number 1"*. The search engine returns the result as *"Hydrogen"* after searching. Now the `progressiveSearcher()` function reaches the parent node and finds that the node does not have any other children. Therefore, it constructs its simple query by concatenating the result of its child node. The node now contains *"the discoverer of Hydrogen"* as its corresponding simple query and the process continues. Finally, `progressiveSearcher()` searches the simple query *"What was the capital of United Kingdom"* and receives *"London"* as the result. *"London"* is the answer for the search query *"What was the capital of the country of the discoverer of the element which has the atomic number 1?"*

Let us consider two search queries such that the answer to the query tree for the second query is present as a keyword in the first query. Let us construct a larger search query by connecting the two queries, such that, the second query replaces that keyword in the first query. The query complexity of the new search query is at most the sum of the query complexities of the two queries. For example, consider the two queries *"Who was the discoverer of the element which has the atomic number 1"* and *"What was the capital*

Figure 7: Constructing a larger query tree with smaller query trees

of the country of Henry Cavendish”. Each of them has a query complexity of 2. We construct new query by connecting the two queries above on the keyword ”Henry Cavendish”. The query ”What was the capital of the country of the discoverer of the element which has the atomic number 1” has a query complexity of 4 as shown in Figure 7. We can note that the query complexity of the larger query is the sum of the query complexities of the smaller ones.

Claim: If a new query is constructed by connecting two search queries of query complexity n and m respectively, then the query complexity of the new query is at most $n + m$.

Let z be a search query composed of z_1 and z_2 with G_z , G_{z_1} , and G_{z_2} as their query trees and c_z , c_{z_1} , and c_{z_2} as their query complexities respectively. Assume we construct the new query z by replacing a keyword in the search query z_1 with the search query z_2 . The number of keywords present in the new search query z is less than the sum of the number of words present in the individual search queries. Moreover, the number of simple queries present in z is no more than the sum of the number of simple queries present in z_1 and z_2 . So, no additional nodes gets created.

During query tree construction, the query tree G_{z_2} is added as a subtree to the node corresponding to the keyword in G_{z_1} if the syntax analyzer identifying this connection has high priority. Otherwise, the root node of z_2 gets concatenated to the node directly. In the first case, we have $c_z = c_{z_1} + c_{z_2}$

as the query complexity. In the second case, this number decreases by one to $c_z = c_{z_1} + c_{z_2} - 1$. The new query tree G_z , therefore, has a query complexity c_z of at most $c_{z_1} + c_{z_2}$.

Progressive querying visits each node once during recursion. It constructs the simple query on each intermediate node (not required in leaf nodes) and searches for the simple query in a search engine. Therefore, the complexity of progressive querying is $O(n.y.k)$, where n is the number of nodes, y the roundabout time for the search engine, and k the cost of string operations while constructing the simple query.

3 Results

Our implementation uses the Wolfram Alpha search API as the backbone search engine in progressive querying. Although PTGQ only searches simple queries using Wolfram Alpha, it may sometimes fail to provide the desired result. For example, if the result’s format does not match the assumed structure, or the search engine returns no result, then PTGQ fails to produce an answer for the search query.

For dependency parsing, we use a pre-trained model [24, 25] from an open-source library for natural language processing (NLP) called spaCy¹[26]. The parser model typically visits $2N$ states for a sentence of length N but may end up visiting more in case it backtracks with non-monotonic transitions [27].

For evaluation, we first search all the test cases directly in Wolfram Alpha. Next, we run the test cases with PTGQ as the front-end, which internally invokes the Wolfram Alpha search API for searching. Finally, we search the test cases directly in Google Search. We note the number of test cases that pass. Moreover, we measure the time taken by each component during the execution of each test and add them up to calculate the total time taken.

Section 3.1 describes the test cases. Section 3.2 compares the performance of a search engine (Wolfram Alpha) with PTGQ against the search engine as a stand-alone. Section 3.3 compares the performance of a search engine with PTGQ against another search engine that is strictly stronger (Google) than the first search engine. Section 3.4 discusses the additional time taken by the PTGQ application and its feasibility.

¹spaCy is available at: <https://github.com/explosion/spaCy> and <https://spacy.io/>

Table 1: Distribution of queries constructed

Complexity	Queries	Avg. Words
1	332	4.79
2	324	7.91
3	173	9.92
4	102	14.24
5	43	18.88
6	14	25.35

3.1 Test Cases

We created a set of 1000 factually correct queries that are within the syntax of our implementation. These test cases are classified based on the query complexity that we note during the generation of the test cases. Table 1 illustrates the test case distribution and the average number of words per each query complexity level. We constructed the base queries, having query complexity 1, based on search topics (people, geography, history, society, dates, etc.) available in Wolfram Alpha. We generated queries of higher query complexity from the base queries by nesting them with connective words. A few of them are shown in Table 2.

PTGQ generates the correct query tree for 894 of the 1000 queries in our set. (The answers to these ought to be correct if the search engine at the back end can handle all the simple queries correctly, but in a small number of cases the results are incorrect because of the search engine.) However, it fails to produce a tree for 9 of the 1000 search queries and generates incorrect query trees for the remaining 97 search queries. These search queries fail to generate the expected query trees due to the following reasons: they contain a connective that PTGQ does not handle, some of the simple queries identified need to be broken down further, or some of the simple queries identified need to be combined.

3.2 Comparison of Performance with a Stand-alone Search Engine

To compare the performance gain, we attach PTGQ to the Wolfram Alpha search API. Then we iterate over all the test queries and mark the queries that pass. Table 3 shows the percentage of queries answered by Wolfram Alpha and PTGQ with Wolfram Alpha search API. PTGQ significantly

Table 2: A few samples from the test cases

Query	Complexity
What is the leg count of a spider	1
Who was the monarch of the United Kingdom when french revolution had started	2
Who was the king of France 500 years after the start date of Hundred Year’s War	3
prime minister of Canada 114 years after the start date of the first prime minister of Canada	4
cube root of the cube of the cube of the square root of the square of the smallest prime number	5
What is the population of capital of largest country in the continent of the most populous country in the continent of South Africa	6
Who is the president of the largest country of the continent of the country of the president of the largest country in the continent of China	7

augments the ability of Wolfram Alpha. PTGQ manages to generate the correct query tree for almost all the queries. Wolfram Alpha’s performance gradually fades as the query complexity of the search query increases. Its performance is bumpy and irregular as the number of words in the search query increases. This indicates that the complexity measure of a search query does not directly depend on the number of words it has. On the other hand, PTGQ always produces the correct answer as long as the simple queries that compose the search query are correctly answerable by the search engine.

Both PTGQ and Wolfram Alpha answers 92.8% queries of query complexity 1. Wolfram Alpha and PTGQ give correct answers for 46.3% and 81.8% of all test cases with query complexity 2, respectively. Although PTGQ correctly generates the query tree for 90.7% of the test cases, it answers only 81.8% of the test cases. PTGQ fails to provide the correct result for the remaining queries because the search engine fails to give a correct answer for at least one of the simple queries in the query tree, parsed by PTGQ. From query complexity 3, the results produced by Wolfram Alpha start to dwindle, giving no accurate answer while PTGQ continues to perform considerably well. Figure 8 plots the performance of PTGQ against Wolfram Alpha as the query complexity increases.

Table 3: Comparison of performance of PTGQ attached to a search engine with stand-alone search engines

Complexity	Wolfram	Google	Wolfram + PTGQ	Wolfram + PTGQ (QT)
1	92.8 %	94.3 %	92.8 %	92.8 %
2	46.3 %	53.4 %	81.8 %	90.7 %
3	0.0 %	9.2 %	76.3 %	85.0 %
4	0.0 %	0.0 %	86.3 %	90.2 %
5	0.0 %	0.0 %	90.7 %	90.7 %
6	0.0 %	0.0 %	85.7 %	100.0 %

Figure 8: Ratio of queries solved v/s Query complexity

3.3 Comparison of Performance with a Stronger Search Engine

Additionally, we compare the accuracy of PTGQ with a stronger search engine. We once again attach PTGQ to the Wolfram Alpha search API. Wolfram Alpha, therefore, acts as the base search engine. We choose Google Search as the other search engine as it answers all queries that Wolfram Alpha answers and also produces correct answers for some complex queries that Wolfram Alpha fails to answer. We confirm this by testing it with complex queries that fall within and outside the query syntax that PTGQ handles. We manually search the test cases in Google Search and consider a query to pass if Google Search displays the result in an infobox on the first page. We show the results in Table 3.

Google Search answers 94.3% of the queries of query complexity 1. It gives correct answers for around 53.4% of the queries of query complexity 2. However, the performance starts to drop from query complexity 3, where it answers only 9.2% of the queries. After that, it fails to give an accurate answer for any of the queries of higher query complexity, while PTGQ continues to generate correct answers for most of them. The main reason for this is the fact that PTGQ always searches with a few keywords. Figure 8 illustrates the same.

3.4 Time Analysis

Query tree generation consists of keyword identification, simple query generation, and query tree construction. The time complexity for query tree generation is, therefore, the sum of their time complexity: $O(n.z.k)$, where n corresponds to the number of words in the search query, z corresponds to the number of syntax analyzers and k corresponds to the cost of string operations. The time complexity for query tree generation is therefore proportional to n .

Figure 9 plots the time taken to generate the query tree following dependency parsing against the number of words of a search query. Query tree generation time shows a nearly linear increase as the number of words in the search query increases. This behavior is expected as the time complexity of query tree generation is proportional to the number of words. Similarly, as illustrated in Figure 10, query tree generation time increases linearly as the query complexity of the search query increases. This is once again expected,

²The average query tree generation time shows spikes at higher word counts due to lower number of test cases.

Figure 9: Query tree generation time v/s Word count ²

as the query complexity of a query tree depends on the number of keywords present.

4 CONCLUSION

This paper introduces a three-step approach named PTGQ for processing complex queries. The steps—dependency parsing, query tree generation, and progressive querying—answer a search query by constructing its corresponding query tree and recursively solving the tree. PTGQ is modular, providing the flexibility to use different implementations for each of its components. Although commercial search engines start to perform poorly from a query complexity of around 3, results show that PTGQ continues to perform well for even higher query complexities.

The answering capabilities of PTGQ depend on the syntax analyzers it contains. By adding more syntax analyzers to PTGQ, we can answer more kinds of queries. PTGQ can be made even more robust by making progressive querying use multiple knowledge bases to fetch results for a query tree. Additionally, syntax analyzers in keyword identification and query tree generation can run as parallel instances, further improving the performance.

Figure 10: Query tree generation time v/s Query complexity

References

- [1] Jiafeng Guo, Gu Xu, Xueqi Cheng, and Hang Li. Named entity recognition in query. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, page 267–274, New York, NY, USA, 2009. Association for Computing Machinery.
- [2] Jeffrey Pound, Peter Mika, and Hugo Zaragoza. Ad-hoc object retrieval in the web of data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 771–780, New York, NY, USA, 2010. Association for Computing Machinery.
- [3] Thomas Lin, Patrick Pantel, Michael Gamon, Anitha Kannan, and Ariel Fuxman. Active objects: Actions for entity-centric search. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, page 589–598, New York, NY, USA, 2012. Association for Computing Machinery.
- [4] Ana Berdasco, Gustavo López, Ignacio Diaz, Luis Quesada, and Luis A. Guerrero. User experience comparison of intelligent personal assistants: Alexa, google assistant, siri and cortana. *Proceedings*, 31:51, 11 2019.
- [5] Kallirroi Georgila, Anton Leuski, Volodymyr Yanov, and David Traum. Evaluation of off-the-shelf speech recognizers across diverse dialogue domains. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 6469–6476, Marseille, France, May 2020. European Language Resources Association.
- [6] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. In Michael Martin, Martí Cuquet, and Erwin Folmer, editors, *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems - SEMANTiCS2016 and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16) co-located with the 12th International Conference on Semantic Systems (SEMANTiCS 2016), Leipzig, Germany, September 12-15, 2016*, volume 1695 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.
- [7] A. Hogan, E. Blomqvist, Michael Cochez, C. d'Amato, Gerard de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, R. Navigli, A. N. Ngomo, S. M. Rashid, A. Rula, Lukas Schmelzeisen,

-
- Juan Sequeda, Steffen Staab, and A. Zimmermann. Knowledge graphs. *Communications of the ACM*, 64:96 – 104, 2021.
- [8] Christopher N. Carlson. Information overload, retrieval strategies and internet user empowerment. pages 169–173. Media Lab UIAH, November 2003.
- [9] Bzar Hussan. Comparative study of semantic and keyword based search engines. *Advances in Science, Technology and Engineering Systems Journal*, 5(1):106–111, 01 2020.
- [10] Emanuele Di Buccio, Massimo Melucci, and Federica Moro. Detecting verbose queries and improving information retrieval. *Information Processing & Management*, 50, 01 2013.
- [11] Qian Liu, Bei Chen, Haoyan Liu, Jian-Guang Lou, Lei Fang, Bin Zhou, and Dongmei Zhang. A split-and-recombine approach for follow-up query analysis. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5316–5326, Hong Kong, China, November 2019. Association for Computational Linguistics.
- [12] Shashi Narayan, Claire Gardent, Shay B. Cohen, and Anastasia Shimorina. Split and rephrase. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 606–616, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.
- [13] Tu Vu, Baotian Hu, Tsendsuren Munkhdalai, and Hong Yu. Sentence simplification with memory-augmented neural networks. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 79–85, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.
- [14] Mengru Wang, Hiroaki Ozaki, Yuta Koreeda, and Kohsuke Yanai. Split first and then rephrase: Hierarchical generation for sentence simplification. In Le-Minh Nguyen, Xuan-Hieu Phan, Kôiti Hasida, and Satoshi Tojo, editors, *Computational Linguistics - 16th International Conference of the Pacific Association for Computational Linguistics*,

PACLING 2019, Hanoi, Vietnam, October 11-13, 2019, Revised Selected Papers, volume 1215 of *Communications in Computer and Information Science*, pages 15–27. Springer, 2019.

- [15] R. Chandrasekar, Christine Doran, and B. Srinivas. Motivations and methods for text simplification. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 2, COLING '96*, page 1041–1044, USA, 1996. Association for Computational Linguistics.
- [16] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14*, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.
- [17] Lemin Zhang and Huifang Deng. Sentence simplification based on multi-stage encoder model. *IEEE Access*, 7:174248–174256, 2019.
- [18] Yinuo Guo, Tao Ge, and Furu Wei. Fact-aware sentence split and rephrase with permutation invariant training. volume 34, pages 7855–7862, 04 2020.
- [19] Hang Cui, Renxu Sun, Keya Li, Min-Yen Kan, and Tat-Seng Chua. Question answering passage retrieval using dependency relations. volume 39, pages 400–407, 08 2005.
- [20] Gokhan Tur, Dilek Hakkani-Tur, Larry Heck, and Sarangarajan Parthasarathy. Sentence simplification for spoken language understanding. pages 5628 – 5631, 06 2011.
- [21] Bidyut Das, Mukta Majumder, and Santanu Phadikar. A novel system for generating simple sentences from complex and compound sentences. volume 10, pages 57–64, 01 2018.
- [22] Sandra Kübler, Ryan T. McDonald, and Joakim Nivre. *Dependency Parsing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2009.
- [23] Ann Taylor, Mitchell Marcus, and Beatrice Santorini. *The Penn Treebank: An Overview*, pages 5–22. Springer Netherlands, Dordrecht, 2003.
- [24] Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. pages 959–976, Mumbai, India, 12 2012. The COLING 2012 Organizing Committee.

- [25] Eliyahu Kiperwasser and Yoav Goldberg. Simple and accurate dependency parsing using bidirectional LSTM feature representations. volume 4, pages 313–327, 2016.
- [26] X. Schmitt, S. Kubler, J. Robert, M. Papadakis, and Y. LeTraon. A replicable comparison study of ner software: Stanfordnlp, nltk, opennlp, spacy, gate. In *2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pages 338–343, 2019.
- [27] Matthew Honnibal and Mark Johnson. An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1378, Lisbon, Portugal, September 2015. Association for Computational Linguistics.