Syntactic Search by Example

Micah Shlain[1,2] **Hillel Taub-Tabib**[1] **Shoval Sadde**[1] **Yoav Goldberg**[1,2]
[1] Allen Institute for AI, Tel Aviv, Israel
[2] Bar Ilan University, Ramat-Gan, Israel
{micahs,hillelt,shovals,yoavg}@allenai.org yogo@cs.biu.ac.il

Abstract

We present a system that allows a user to search a large linguistically annotated corpus using syntactic patterns over dependency graphs. In contrast to previous attempts to this effect, we introduce a light-weight query language that does not require the user to know the details of the underlying syntactic representations, and instead to query the corpus by providing an example sentence coupled with simple markup. Search is performed at an interactive speed due to efficient linguistic graph-indexing and retrieval engine. This allows for rapid exploration, development and refinement of syntax-based queries. We demonstrate the system using queries over two corpora: the English wikipedia, and a collection of English pubmed abstracts. A demo of the wikipedia system is avilable at https://bit.ly/36IyT2I

1 Introduction

The introduction of neural-network based models into NLP brought with it a substantial increase in syntactic parsing accuracy. We can now produce accurate syntactically annotated corpora at scale. However, the produced representations themselves remain opaque to most users, and require substantial linguistic expertise to use. Patterns over syntactic dependency graphs[1] can be very effective for interacting with linguistically-annotated corpora, either for linguistic retrieval or for information and relation extraction (Fader et al., 2011; Akbik et al., 2014; Valenzuela-Escárcega et al., 2015, 2018). However, their use in mainstream NLP as represented in ACL and affiliated venues remain limited. We argue that this is due to the high barrier of entry associated with the application of such patterns. Our aim is to lower this barrier and allow also linguistically-naïve users to effectively experiment with and develop syntactic patterns. Our proposal rests on two components:

(1) A light-weight query language that does not require in-depth familiarity with the underlying syntactic representation scheme, and instead lets the user specify their intent via a natural language example and lightweight markup.

(2) A fast, near-real-time response time due to efficient indexing, allowing for rapid experimentation.

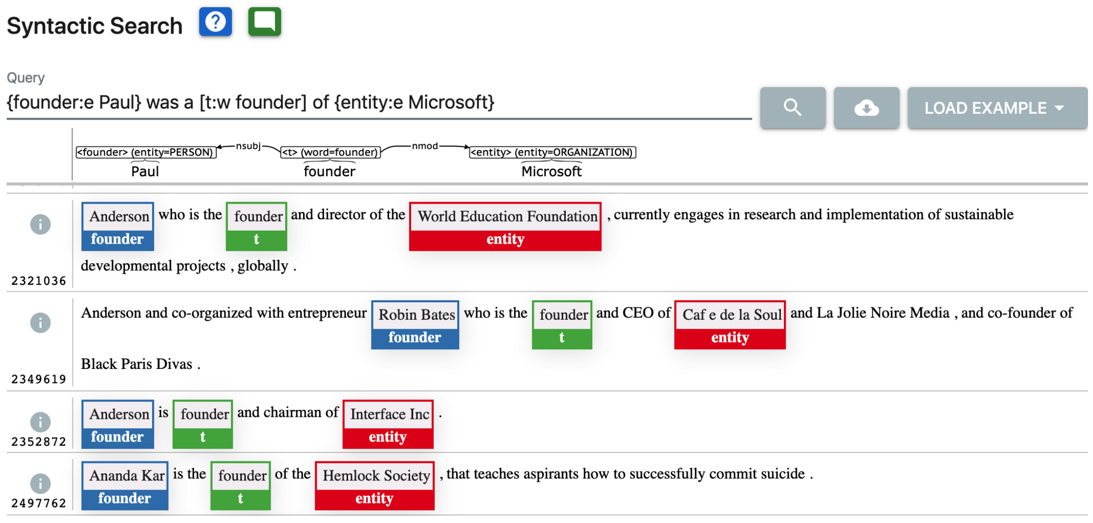
Figure 1 (next page) shows the interface of our web-based system. The user issued the query:

{founder:e Paul} was a [t:w founder] of {entity:e Microsoft}.

The query specifies a sentence (Paul was a founder of Microsoft) and three captures: *founder*, *t* and *entity*. The *founder* and *entity* captures should have the same entity-type as the corresponding sentence words (PERSON for Paul and ORGANIZATION for Microsoft), and the *t* capture should have the same word form as the one in the sentence (founder) . The syntactic relation between the captures should be the same as the one in the sentence.

The query was translated into a graph-based syntactic representation, which is shown in the figure below the query, where each graph-node is associated with the query word that triggered it. The system also returned a list of matched sentences. The matched tokens for each capture group (*founder*, *t* and *entity*) are highlighted. The user can then issue another query, browse the results list, or download all the results as a tab-separated values file.

[1] In this paper, we very loosely use the term "syntactic" to refer to a linguistically motivated graph-based annotation over a piece of text, where the graph is directed and there is a path between any two nodes. While this usually implies syntactic dependency trees or graphs (and indeed, our system currently indexes Enhanced English Universal Dependency graphs (Nivre et al., 2016; Schuster and Manning, 2016)) the system can work also with more semantic annotation schemes e.g, (Oepen et al., 2015), given the availability of an accurate enough parser for them.



{founder:e Paul} was a [t:w founder] of {entity:e Microsoft}

2321036 Anderson (founder) who is the founder (t) and director of the World Education Foundation (entity) , currently engages in research and implementation of sustainable developmental projects , globally .

2349619 Anderson and co-organized with entrepreneur Robin Bates (founder) who is the founder (t) and CEO of Caf e de la Soul (entity) and La Jolie Noire Media , and co-founder of Black Paris Divas .

2352872 Anderson (founder) is founder (t) and chairman of Interface Inc (entity) .

2497762 Ananda Kar (founder) is the founder (t) of the Hemlock Society (entity) , that teaches aspirants how to successfully commit suicide .

Figure 1: Syntactic Search System

2 Existing syntactic-query languages

While several rich query languages over linguistic tree and graph structure exist, they require a substantial amount of expertise to use. The user needs to be familiar not only with the syntax of the query language itself, but to also be intimately familiar with the specific syntactic scheme used in the underlying linguistic annotations. For example, in Odin (Valenzuela-Escárcega et al., 2015), a dedicated language for pattern-based information extraction, the same rule as above is expressed as:

```
- label: Person
  type: token
  pattern: |
   [entity="PERSON"]+
- label: Organization
  type: token
  pattern: |
   [entity="ORGANIZATION"]+
- label: founded
  type: dependency
  pattern: |
    trigger = [word=founded]
    founder:Person = >nsubj
    entity:Organization = >nmod
```

The Spacy NLP toolkit[2] also includes pattern matcher over dependency trees,using JSON based syntax:

```
[{"PATTERN": {"ORTH": "founder"},
  "SPEC": {"NODE_NAME": "t"}},
 {"PATTERN": {"ENT_TYPE": "PERSON"}},
  "SPEC": {"NODE_NAME": "founder",
           "NBOR_RELOP": ">nsubj",
           "NBOR_NAME": "t"}},
 {"PATTERN": {"ENT_TYPE": "ORGANIZATION"},
  "SPEC": {"NODE_NAME": "entity",
           "NBOR_RELOP": ">nmod",
           "NBOR_NAME": "t"}}]
```

[2]https://spacy.io/

Stanford's Core-NLP package (Manning et al., 2014) includes a dependency matcher called SEMGREX,[3] which uses a more concise syntax:

```
{ner:PERSON}=founder
  <nsubj ({word:founder}=t
           >nmod {ner:ORG}=entity)
```

The dep_search system[4] from Turku university (Luotolahti et al., 2017) is designed to provide a rich and expressive syntactic search over large parsebanks. They use a lightweight syntax and support working against pre-indexed data, though they do not support named captures of specific nodes.

```
  PERSON <nsubj founder >nmod ORG
```

While the different systems vary in the verboseness and complexity of their own syntax (indeed, the Turku system's syntax is rather minimal), they all require the user to explicitly specify the dependency relations between the tokens, making it challenging and error-prone to write, read or edit. The challenge grows substantially as the complexity of the pattern increases beyond the very simple example we show here.

Closest in spirit to our proposal, the PROPMINER system of Akbik et al. (2013) which lets the user enter a natural language sentence, mark spans as *subject*, *predicate* and *object*, and have a rule being generated automatically. However, the system is restricted to ternary subject-predicate-object patterns. Furthermore, the generated pattern is an over-restricted rule written in a path-expression SQL variant (SerQL, (Broekstra and Kampman,

[3]https://github.com/explosion/spaCy/blob/master/spacy/matcher/dependencymatcher.pyx
[4]http://bionlp-www.utu.fi/dep_search/

2003)), which the user then needs to manually edit. For example, to the best of our understanding, our query above will be translated to:

```
SELECT subject, predicate, object
FROM predicate.3 nsubj subject,
     predicate.3 nmod object,
WHERE subject POS NNP
AND predicate.3 POS NN
AND object POS NNP
AND subject TEXT PAUL
AND predicate.3 TEXT founder
AND object TEXT Microsoft
AND subject FULL_ENTITY
AND object FULL_ENTITY
```

All these systems require the user to closely interact with linguistic concepts and explicitly specify graph-structures, posing a high barrier of entry for non-expert users. They also slow down expert users: formulating a complex query may require a few minutes. Furthermore, many of these query languages are designed to match against a provided sentence, and are not indexable. This requires iterating over all sentences in the corpus attempting to match each one, requiring substantial time to obtain matches from large corpora.

Augustinus et al. (2012) describe a system for syntactic search by example, which retrieves tree fragments and which is completely UI based. Our system takes a similar approach, but replaces the UI-only interface with an expressive textual query language, allowing for richer queries. We also return node matches rather than tree fragments.

3 Syntactic Search by Example

We propose a substantially simplified language, that has the minimal syntax and that does not require the user to know the underlying syntactic schema upfront (though it does not completely hide it from the user, allowing for exposure over time, and allowing control for expert users who understand the underlying syntactic annotation scheme).

The query language is designed to be linguistically expressive, simple to use and amenable to efficient indexing and query. The simplicity and indexing requirements do come at a cost, though: we purposefully do not support some of the features available in existing languages. We expect these features to correlate with expertise.[5] At the same time, we also seamlessly support expressing arbitrary sub-graphs, a task which is either challenging or impossible with many of the other systems.

[5]Example of a query feature we do not support is quantification, i.e., "nodes a and b should be connected via a path that includes *one or more* 'conj' edges".

The language is based on the following principles:

(1) The core of the query is a natural language sentence.

(2) A user can specify the tokens of interest and constraints on them via lightweight markup over the sentence.

(3) While expert users can specify complex token constraints, effective constraints can be specified by pulling values from the query words.

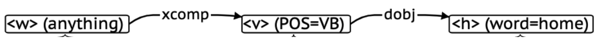
The required syntactic knowledge from the user, both in terms of the syntax of the query language itself and in terms of the underlying linguistic formalism, remain minimal.

4 Graph Query Formalism

The language is structured around between-token relations and within-token constraints, where tokens can be *captured*.

Formally, our query $G = (V, E)$ is a labeled directed graph, where each node $v_i \in V$ corresponds to a token, and a labeled edge $e = (v_i, v_j, \ell) \in E$ between the nodes corresponds to a between-token syntactic constraint. This query graph is then matched against parsed target sentences, looking for a correspondence between query nodes and target-sentence nodes that adhere to the token and edge constraints.

For example, the following graph specifies three tokens, where the first and second are connected via an 'xcomp' relation, and the second and third via a 'dobj' relation. The first token is unconstrained, while the second token must have the POS-tag of VB, and the third token must be the word home.



Sentences whose syntactic graph has a subgraph that aligns to the query graph and adheres to the constraints will be considered as matches. Example of such matching sentences are:

- *John* $\underline{\textit{wanted}}_w$ *to* $\underline{\textit{go}}_v$ $\underline{\textit{home}}_h$ *after lunch.*

- *It was a place she* $\underline{\textit{decided}}_w$ *to* $\underline{\textit{call}}_v$ *her* $\underline{\textit{home}}_h$.

The <w>, <v> and <h> marks on the nodes denote *named captures*. When matching a sentence, the sentence tokens corresponding to the graph-nodes will be bound to variables named 'w', 'v' and 'h', in our case {`w=wanted, v=go, h=home`} for the first sentence and {`w=decided, v=call, h=home`} for the second. Graph nodes can also be unnamed, in which case they must match sentence tokens but will not bind to any variable. The graph structure is not meant to be specified by hand,[6] but rather to be inferred from the example based query language described in the next section (an example query resulting in this graph is "They [w wanted] to [v:tag go] [h:word home]").

Between-token constraints correspond to labeled directed edges in the sentence's syntactic graph.

Within-token constraints correspond to properties of individual sentence tokens.[7] For each property we specify a list of possible values (a disjunction) and if lists for several properties are provided, we require all of them to hold (a conjunction). For example, in the constraint tag=VBD|VBZ&lemma=buy we look for tokens with POS-tag of either VBD or VBZ, and the lemma *buy*. The list of possible values for a property can be specified as a pipe-separated list (tag=VBD|VBZ|VBN) or as a regular expression (tag=/VB[DZN]/).

5 Example-based User-friendly Query Language

The graph language described above is expressive enough to support many interesting queries, but is also very tedious to specify query graphs G, especially for non-expert users. We propose a simple syntax that allows to easily specify a graph query G (nodes with names and constraints, connected by labeled edges) using a textual query q that takes the form of an example sentence and lightweight markup. The user writes queries q and can observe the resulting graphs G.

Let $s = w_1, ..., w_n$ be a proper English sentence. Let D be its dependency graph, with nodes w_i and labeled edges (w_i, w_j, ℓ). A corresponding textual query q takes the form $q = q_1, ..., q_n$, where each q_i is either a word $q_i = w_i$, or a *marked* word $q_i = m(w_i)$. A marking of a word takes the form: [name:constraints word] or [name word], for constrained and unconstrained nodes, respectively. Consider the query:

John [w wanted] to [v:tag=VB go] [h:word=home home]

corresponding to the above graph query. The marked words are:

q_2 =[w wanted] (unconstrained, name:w)
q_4 =[v:tag=VB go] (cnstr:tag=VB, name:v)
q_5 =[h:word=home home] (cnstr:word=home, name:h)

Each of these corresponds to a node v_{qi} in the query graph above.

Let m be the set of marked query words, and m^+ be a minimal connected subgraph of D that includes all the words in m. When translating q to G, each marked word $w_i \in m$ is translated to a named query graph node v_{q_i} with the appropriate restriction. The additional words $w_j \in m^+ \setminus m$ are translated to unrestricted, unnamed nodes v_{q_j}. We add a query graph edge (v_{q_i}, v_{q_j}, ℓ) for each pair in V for which $(w_i, w_j, \ell) \in D$.

Further query simplifications Consider the marked word [h:word=home home]. The constraint is redundant with the word. In such cases we allow the user to drop the value, which is then taken from the corresponding property of the query word. This allows us to replace the query:

John [w wanted] to [v:tag=VB go] [h:word=home home]

with:

John [w wanted] to [v:tag go] [h:word home]

This further drives the "by example" agenda, as the user does not need to know what the lemma, entity-type or POS-tag of a word are in order to specify them as a constraint.

Finally, full property names can be replaced with their shorthands w,l,t,e:

John [w wanted] to [v:t go] [h:w home]

Unnamed marks In some cases we want to add a node to the graph, without an explicit name. In such cases we can use the special name $. These are interpreted as having a default constraint of :word, which can be overriden by providing an alternative constraint ([$:e John]), or an empty one ([$:* John]).

Expansions When matching a query against a sentence, the named matches bind to sentence words. Sometimes, we may want the match to be expanded to a larger span of the sentence. For example, when matching a word which is part of a entity, we often wish to capture the entire entity rather than the word. This is achieved by replacing the capture brackets from [ ] to "expanding brackets" { }. The default behavior is to expand the match from the current word to the named entity boundary or NP-chunk that surrounds it, if it exists. We are currently investigating the option of providing additional expansion strategies.

[6] Indeed, we currently do not even expose a textual representation of the graph, though we may consider adding it in future versions as an "expert" feature.

[7] Currently supported properties are word-form (word), lemma (lemma), pos-tag (tag) or entity type (entity). Additional types can be easily added, provided that we have suitable linguistic annotators for them.

Summary To summarize the query language from the point of view of the user: the user starts with a sentence $w_1, ..., w_n$, and marks some of the words for inclusion in the query graph. For each marked word, the user specifies a name, and an optional constraints. The user query is then translated to a graph query as described above. The results list highlights the words corresponding to the marked query words. The user can choose for the results to highlight entire entities rather than single words.

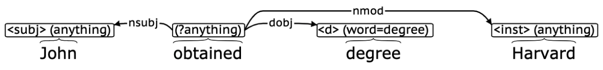
6 Interactive Pattern Authoring

An important aspect of the system is its interactivity. Users enter queries by writing a sentence and adding markup on some words, and can then refine them following feedback from the environment, as we demonstrate with a walk-through example.

A user interested in people who obtained degrees from higher education institutions may issue the following query:

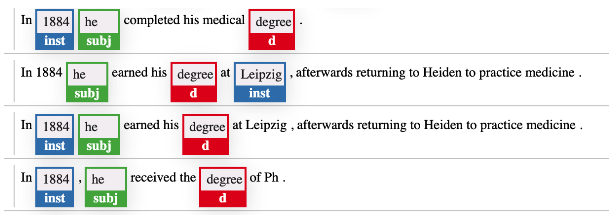
[subj John] obtained his [d:w degree] from [inst Harvard]

Here, the person in the "subj" capture and the institution in the "inst" capture are placeholders for items to be captured, so the user uses generic names and leaves them unconstrained. The "degree" ("d") capture should match exactly, as the user specified the "w" constraint (exact word match). When pressing `Enter`, the user is then shown the resulting query-graph and a result list. The user can then refine their queries based on either the query graph, the result list, or both. For the above query, the graph is:



Note that the query graph associates each graph node with the query word that triggered it. The word "obtained" resulted in a graph node even though it was not marked by the user as a capture. The user makes note to themselves to go back to this word later. The user also notices that the word "from" is not part of the query.

Looking at the result list, things look weird:



In 1884 he completed his medical degree .

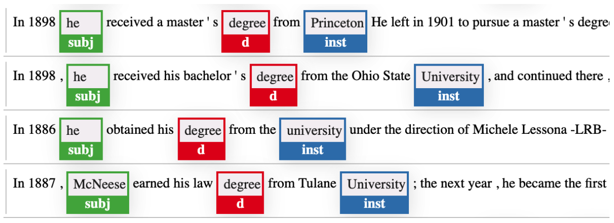
In 1884 he earned his degree at Leipzig , afterwards returning to Heiden to practice medicine .

In 1884 he earned his degree at Leipzig , afterwards returning to Heiden to practice medicine .

In 1884 , he received the degree of Ph .

Maybe this is because the word *from* is not in the graph? Indeed, adding a non-capturing exact-word constraint on "from" solves this issue:

[subj John] obtained his [d:w degree] [$ from] [inst Harvard]



In 1898 he received a master ' s degree from Princeton He left in 1901 to pursue a master ' s degre

In 1898 , he received his bachelor ' s degree from the Ohio State University , and continued there ,

In 1886 he obtained his degree from the university under the direction of Michele Lessona -LRB-

In 1887 , McNeese earned his law degree from Tulane University ; the next year , he became the first

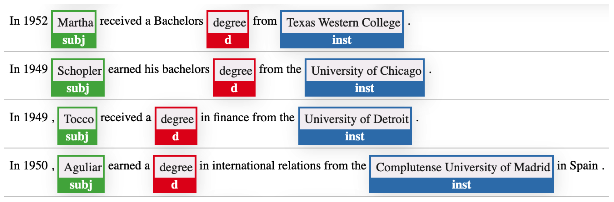
However, the resulting list contains many non-names in the *subj* capture. Trying to resolve this, the user adds an "entity-type" constraint to the *subj* capture:

[subj:e John] obtained his [d:w degree] [$ from] [inst Harvard]

Note that the user didn't specify an exact type, yet the query graph correctly resolved PERSON.

The user is interested in the full name of the person and organization, so they change from single-word capture to expanded capture, with the default expansion level (changing the brackets from [ ] to { }). This results in:

{subj:e John} obtained his [d:w degree] [$ from] {inst Harvard}



In 1952 Martha received a Bachelors degree from Texas Western College .

In 1949 Schopler earned his bachelors degree from the University of Chicago .

In 1949 , Tocco received a degree in finance from the University of Detroit .

In 1950 , Aguliar earned a degree in international relations from the Complutense University of Madrid in Spain .

These are the kind of results the user expected, but now they are curious about degrees obtained by females, and their representation in the Wikipedia corpus. Adding the pronoun to the query, the user then issues the following two queries, saving the result-sets from each one as a CSV for further comparative analysis.

{subj:e John} obtained [$ his] [d:w degree] [$ from] {inst Harvard}

{subj:e John} obtained [$ her] [d:w degree] [$ from] {inst Harvard}

Our user now worries that they may be missing some results by focusing on the word *degree*. Maybe other things can be obtained from a university? The user then sets an exact-word constraint on the word "Harvard", adds a lemma constraint to "obtain" and clears the constraint from "degree":

{subj:e John} [o:l obtained] his [d degree] [$ from] {inst:w Harvard}

Browsing the results, the *d* capture includes words such as “BA, PhD, MBA, certificate”. But the result list is rather short, suggesting that either *Harvard* or *obtain* are too restrictive. The user seeks to expand the “obtain” node’s vocabulary, adding back the exact word constraint on “degree” while removing the one from “obtain”:

{subj:e John} [o obtained] his [d:w degree] [$ from] {inst:w Harvard}

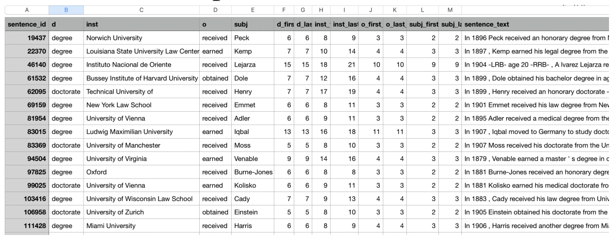
Looking at the result list in the *o* capture, the user chooses the lemmas “receive, complete, earn, obtain, get”, adds them to the *o* constraint, and removes the degree constraint.

{subj:e John} [o:l=receive|complete|earn|obtain|get obtained] his [d degree] [$ from] {inst:w Harvard}

The returned result-set is now much longer, and we select additional terms for the degree slot and remove the institution word constraint, resulting in the final query:

{subj:e John} [o:l=receive|complete|earn|obtain|get obtained] his [d:w=degree|MA|BA|MBA|doctorate|masters|PhD degree] [$ from] {inst:w Harvard}

The result is a list of person names earning degrees from institution, and the entire list can be downloaded as a tab-separated file which includes the named captures as well as the source sentences (over Wikipedia, this list has 6197 rows).[8]



sentence_id	d	inst	o	subj	d_firs	d_las	inst_	inst_last	o_first_	o_last_	subj_first	subj_la	sentence_text
19437	degree	Norwich University	received	Peck	6	6	8	9	3	3	2	2	In 1896 Peck received an honorary degree from I
22370	degree	Louisiana State University Law Center	earned	Kemp	7	7	10	14	4	4	3	3	In 1897 , Kemp earned his legal degree from the
46140	degree	Instituto Nacional de Oriente	received	Lejarza	15	15	18	21	10	10	9	9	In 1904 -LRB- age 20 -RRB- , A lvarez Lejarza re
61532	degree	Bussey Institute of Harvard University	obtained	Dole	7	7	12	16	4	4	3	3	In 1899 , Dole obtained his bachelor degree in ag
62095	doctorate	Technical University of	received	Henry	7	7	17	19	4	4	3	3	In 1899 , Henry received an honorary doctorate -
69159	degree	New York Law School	received	Emmet	6	6	8	11	3	3	2	2	In 1901 Emmet received his law degree from Nev
81954	degree	University of Vienna	received	Adler	6	6	9	11	3	3	2	2	In 1895 Adler received a medical degree from the
83015	degree	Ludwig Maximilian University	earned	Iqbal	13	13	16	18	11	11	3	3	In 1907 , Iqbal moved to Germany to study docto
83369	doctorate	University of Manchester	received	Moss	5	5	8	10	3	3	2	2	In 1907 Moss received his doctorate from the Un
94504	degree	University of Virginia	earned	Venable	9	9	14	16	4	4	3	3	In 1879 , Venable earned a master ' s degree in c
97825	degree	Oxford	received	Burne-Jones	6	6	8	8	3	3	2	2	In 1881 Burne-Jones received an honorary degre
99025	doctorate	University of Vienna	earned	Kolisko	6	6	9	11	3	3	2	2	In 1881 Kolisko earned his medical doctorate fro
103416	degree	University of Wisconsin Law School	received	Cady	7	7	9	13	4	4	3	3	In 1883 , Cady received his law degree from Univ
106958	doctorate	University of Zurich	obtained	Einstein	5	5	8	10	3	3	2	2	In 1905 Einstein obtained his doctorate from the
111428	degree	Miami University	received	Harris	6	6	8	9	4	4	3	3	In 1906 , Harris received another degree from Mi

The query can also be further refined to capture *which* degree was obtained, e.g.:

{subj:e John} [o:l=... obtained] his [kind law] [d:w=... degree] [$ from] {inst Harvard}

capturing under *kind* words like *law*, *chemistry*, *engineering* and *DLitt* but also *bachelors*, *masters* and *graduate*.

This concludes our walk-through.

[8]The list can be even more comprehensive had we selected additional degree words and obtain words, and considered also additional re-phrasings.

7 Additional Query Examples

To whet the reader’s appetite, here are a sample of additional queries, showing different potential use-cases. Over wikipedia:

- {p:e Sam} [$:l=win|receive won] an [$ Oscar].
- {p:e Sam} [$:l=win|receive won] an [$ Oscar] [$ for] {thing something}
- [$ fish] [$ such] [$ as] {fish salmon}
- {hero:t Spiderman} [$ is] a [$ superhero]
- I like [kind coconut] [$ oil]
- [kind coconut] [$ oil] is [$ used] for [purpose eating]

Over a pubmed corpus, annotated with the SciSpacy (Neumann et al., 2019) pipeline:

- {x:e aspirin} inhibits {y thing}
- a [$ combination] of {d1:e aspirin} and {d2:e alcohol} [$:l causes] {t thing}
- {patients:t rats} were [$ injected] [$ with] {what drugs}

8 Implementation Details

The indexing is handled by Lucene.[9] We currently use Odinson, an open-source Lucene-based query engine developed at Lum[10] as a successor of Odin (Valenzuela-Escárcega et al., 2015), that allows to index syntactic graphs and issue efficient path queries on them. We translate our queries into an Odinson path query that corresponds to a longest path in our query graph. We then iterate over the returned Odinson matches and verify the constraints that were not on the path. Conceptually, Odinson works by first using Lucene’s reverse-index for retrieving sentences for which there is a token matching each of the specified token-constraints, and then verifying the syntactic between-token constraints. To improve the Lucene-query selectivity, tokens are indexed also with information about incoming and outgoing syntactic edge labels, an information which is incorporated as additional token-constraints to the Lucene engine. The system easily supports millions of sentences.

9 Conclusions

We introduce a simple query language that allow to pose complex syntax-based queries, and obtain results in an interactive speed.

A search interface over Wikipedia sentences is available at `https://bit.ly/36IyT2I`. We intend to release the code as open source, as well as providing hosted open access to a PubMed-based corpus.

[9]`https://lucene.apache.org`
[10]`http://lum.ai`

References

Alan Akbik, Oresti Konomi, and Michail Melnikov. 2013. Propminer: A workflow for interactive information extraction and exploration using dependency trees. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 157–162, Sofia, Bulgaria. Association for Computational Linguistics.

Alan Akbik, Thilo Michael, and Christoph Boden. 2014. Exploratory relation extraction in large text corpora. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 2087–2096.

Liesbeth Augustinus, Vincent Vandeghinste, and Frank Van Eynde. 2012. Example-based treebank querying. In *LREC*.

Jeen Broekstra and Arjohn Kampman. 2003. Serql: A second generation rdf query language. In *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, pages 13–14.

Anthony Fader, Stephen Soderland, and Oren Etzioni. 2011. Identifying relations for open information extraction. In *Proceedings of the conference on empirical methods in natural language processing*, pages 1535–1545. Association for Computational Linguistics.

Juhani Luotolahti, Jenna Kanerva, and Filip Ginter. 2017. Dep_search: Efficient search tool for large dependency parsebanks. In *Proceedings of the 21st Nordic Conference on Computational Linguistics*, pages 255–258, Gothenburg, Sweden. Association for Computational Linguistics.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60.

Mark Neumann, Daniel King, Iz Beltagy, and Waleed Ammar. 2019. Scispacy: Fast and robust models for biomedical natural language processing.

Joakim Nivre, Marie-Catherine De Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. 2016. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 1659–1666.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinková, Dan Flickinger, Jan Hajič, and Zdeňka Urešová. 2015. SemEval 2015 task 18: Broad-coverage semantic dependency parsing. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 915–926, Denver, Colorado. Association for Computational Linguistics.

Sebastian Schuster and Christopher D Manning. 2016. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 2371–2378.

Marco A Valenzuela-Escárcega, Özgün Babur, Gus Hahn-Powell, Dane Bell, Thomas Hicks, Enrique Noriega-Atala, Xia Wang, Mihai Surdeanu, Emek Demir, and Clayton T Morrison. 2018. Large-scale automated machine reading discovers new cancer-driving mechanisms. *Database*, 2018.

Marco A Valenzuela-Escárcega, Gus Hahn-Powell, Mihai Surdeanu, and Thomas Hicks. 2015. A domain-independent rule-based framework for event extraction. In *Proceedings of ACL-IJCNLP 2015 System Demonstrations*, pages 127–132.