Targeted Data Acquisition from Open Data Repositories

By

Shay Pripstein

A Research Project
Submitted to the Department of Computer Science
École Polytechnique

Supervised by
Prof. Ioana Manolescu, Ms. Nelly Barret, Ms. Madhulika Mohanty

December 2023

Introduction

With the increasing amount of data usage and extraction in the last few years, most notably the rise of Large Language Models, there is a great need for finding more information about given entities, that we may not know a lot about. Usually, we may only know the name of the entity, and perhaps a few details about it. But the linkage between the name of the entity and the entity itself is not trivial at all, although it might appear so in the first glance. Moreover, the entity is seldom equivalent to its digital representation, as detailed as it may be.
Nevertheless, the sheer amount of information on the web, and the ability to access it from almost any corner of the world, makes the web our default option concerning finding more information about almost any entity. On top of that, the web provides the users the option to discover new entities, that they weren't looking for in the first place.
One of the ways to do so is through text processing. Consider the following sentence: "Alexander the Great took his horse, Bucephalus, for a walk in Central Park". It is not hard to notice that 3 Named Entities (NE) appear in this sentence – Alexander the Great, Bucephalus the Horse, and Central Park. Or, more precisely, to recognize that 3 names appear in the sentence, that can be traced to some entities. We may have used our general knowledge and comprehension of the English grammar to do that. But that is not a simple task at all. A notorious sentence in that regard is "*Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo".* [1] The task of extracting named entities from text is called Named Entity Recognition (NER).
A tool of particular interest here is ConnectionLens (CL) [2], developed by the CEDAR team in INRIA. CL is a system that ingests heterogenous datasets that can be structured, semi-structured and unstructured into a unique data graph. One of the unstructured datasets it handles is text, and it does so by using NER tools.
NER tools may vary in few parameters, in particular their competence and computational demand, while a tradeoff exists between the two. CL utilizes two tools for handling NER tasks in English and French language texts. They both find named entities in text, and assign them a single type among {Person, Location, Organization}. One is flair [3], a competent but slower tool, and the other is SNER [4], a faster but less competent tool.
A convenient way of accessing CL is by using ConnectionStudio (CS). [5] CS is a Web interface for CL where users can interact with and query any data graph produced by CL. Users are generally interested in Named Entities and what to know more about them.
A limitation of the tools for NER presented above, especially if a faster but less competent tool is used, is mistyping. NER tool may recognize that a NE appears in the texts but fail to type it correctly. In case the NE was correctly typed, the user may still wish to retrieve more information about it, perhaps more than what was provided by those tools. In Order to achieve that, the user needs to match the entity in another Knowledge Base (KB), with just the limited amount of data it has on the NE, provided by tools above.

The goal of this research is to tackle those issues. More precisely, we will focus on retrieving more information on a given NE in some KB, whereas the information provided on it is just **its name and type**. We'd like to identify under which circumstances a NE is likely to be found in some selected KB, when a mistyping is likely, and what was the computational cost of the process. For NE that were successfully retrieved, we'd like to know if the retrieved entity the same that was searched for, and is there a systematic way to get the most important attributes of it.

Background

Knowledge Bases (KB)

Knowledge-based systems have a computational model of some domain of interest in which symbols serve as surrogates for real world domain artefacts, such as physical objects, events, relationships, etc.
The domain of interest can cover any part of the real world or any hypothetical system about which one desires to represent knowledge for computational purposes. A knowledge-based system maintains a **knowledge base** which stores the symbols of the computational model in form of statements about the domain. [6]
At the heart of many KB lies the RDF language, including the KB used in this project - Wikidata [7] and Yago [8].

RDF Language and Triplets

At the heart of many knowledge bases and graphs lies the Resource Description Languguage (RDF).
The core structure of the RDF language is a triplet of subject-predicate-object, that represent the abstraction of two nodes – the subject and the object, and a directed arc - the predicate, that describes the relationship between them.

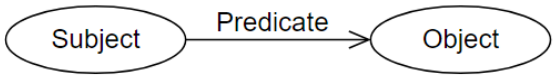


Figure 1: The structure of a triplet

RDF graphs are sets of those triples, where the elements may be IRIs, blank nodes, or datatyped literals. They are used to express descriptions of resources. [9]

SPARQL

RDF is a directed, labeled graph data format for representing information in the Web. This specification defines the syntax and semantics of the SPARQL query language for RDF. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware.
The example below shows a SPARQL query to find the title of a book from the given data graph. The query consists of two parts: the SELECT clause identifies the variables to appear in the query results, and the WHERE clause provides the basic graph pattern to match against the data graph. The basic graph pattern in this example consists of a single triple pattern with a single variable (?title) in the object position. An example of a SPARQL query is provided in the appendix, Figure 7.

Methodology

In this section, a break-up of the main tusk to subtasks will be provided first, and then abstraction of the algorithms created to solve them will be offered. It will be culminated by providing relevant information about the KB that were investigated, especially the reasoning behind adjusting the main algorithm to their unique challenges and advantages.
Repository: https://gitlab.inria.fr/cedar/Targeted-Data-Acquisition

Overall Methodology

Per the introduction, we will focus on entities that we have two pieces of information about – name and type, and we may have a few other details concerning it. To get more information about the entity, there are a few tasks that should be dealt with, after choosing a knowledge base:

1. Match an entity to the name and type. On this step, if we got no matches, we shall choose another knowledge base. It should be noted that the map $(name, type) \xrightarrow{yields} (entity)$ isn't injective. for example, searching for (Jerusalem, Location) on Wikidata, a knowledge base, gives us 14 results.
2. If we have more information about the searched entity than just name and type, we must check if the entities we found in the previous step indeed match our information. If none of the entities match the information, we shall choose another knowledge base.
3. We may now choose one of the entities that made it through the previous steps, as it is what we were looking for, for the best of our knowledge. We shall ask the knowledge base for the information that we are looking for, or just all information about it, depending on our needs.

The focus of the research is on task number 1. As the presumed received information about the entity is just a name and its type, creating a tool for task number 2 isn't in the scope of this project, but an evaluation of the tools created for task number 1 in that regard will take place instead. A tool for task number 3 was created, but a measurement of its success depends on the user. Therefore, an algorithm for dealing with this task will be described, under the section further investigation, along with some possible approaches to generalize it.

Abstract Algorithm

RDF provides a unified approach to find entities in the Data Knowledge Bases (DKB) that are based on it, in the form of the SPARQL language. As each DKB poses unique challenges and advantages the specifics often differ. Nevertheless, there are a few main issues that must be addressed in any implementation, as they are caused by the input data itself, and are independent of the DKB.

1. Querying may be, and often is, case sensitive. In case the original search doesn't yield any result, the user should request to run the query again with a different casing for the name of the desired entity. Proper

casing may depend on the language or local customs. When querying for results in English, the title function ("abc def" ⇒ "Abc Def") is useful, but it is not necessarily the case. For example, in French, calling the title function on “porte d'orléans” will change it into “Porte D'orléans”, instead of the correct form, “Porte d'Orléans”.

2. The entity’s name may be wrongly typed. This issue is especially common if the typing was done automatically by a Machine Learning model. An example of that is Betty Crocker, an American food brand, which is an organization named after a fictional character. A ML model might classify it wrongly as a Person.
3. We may get more results than we need. The number might even be larger than what we’d be willing to check for compatibility in tasks 2 and 3. In this implementation, sending a maximal number of results was the chosen approach, but it is by no means the only possible one.
4. Some inputs might crash the SPARQL endpoint. For example, names containing quotation marks. It will be assumed that the data was cleaned beforehand, for simplicity’s sake.

To address the first three issues, a common interface for dealing with task 1 is offered, in the form:

crawl(name, type, n, casing_function, interchengable_types**):**

name: *the name of the searched entity.*
type: *the type of the searched entity.*
n: *the number of matching entities the user wishes to receive.*
casing_function: *function that changes the case to a suitable form. Necessary to address issue 1.*
interchangeable_types: *a list of other possible types for the name. Necessary to address issue 2.*
return: *a list of up to n entities that match the name and the type (or any of the compatible types).*

Algorithm 1: Abstract Crawl

For example, assume one has as an input the pair (jerusalem, Person). They’d like to get up to 3 results, and it’s known to be retrieved from an English text, so a suitable casing function is the title function. Moreover, it is suspected that the typing is incorrect, so one decides to check for Location as well. A fitting call for the function is:
crawl (“jerusalem”, “Person”, 3, title_func, [“Location”]**).**
Thus, that the expected behavior of the function is going to be:

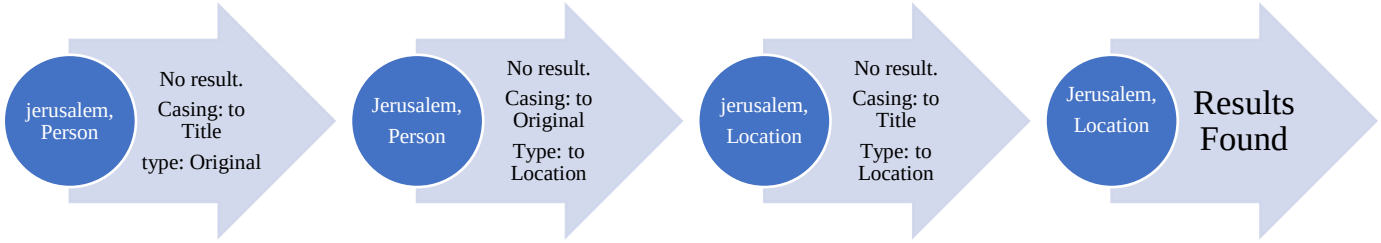


Figure 2:Schematic view of the expected crawler behavior

The implementation of the functions that constitute the project was done in Python, and the functions that were designed to solve the task 1 adhere to the signature provided above. The pseudo-code for the function used in Wikidata and YAGO are provided as an appendix to the project. The project’s repository is:
https://gitlab.inria.fr/cedar/Targeted-Data-Acquisition

Wikidata

url: https://www.wikidata.org

Wikidata is one of the prominent open-source knowledge bases. It is free site that can be read and edited by machines and humans alike. Wikidata acts as the central storage for the structured data of its Wikimedia sister projects, including Wikipedia. [7]
The SPARQL endpoint of Wikidata allows the user to query a name with the desired type, thereby allows a naïve algorithm to implemented. On the other hand, the type of hierarchy in Wikidata poses the user a challenge – if a user

searches for **x** of type **A**, while on Wikidata **x** is of type **B**, and **B** is a subtype of **A**, the query won't return the desired result. This means that the user must verify the type for a potential entity **on the server**. The most straight-forward way to do that with a single query is to use a recursive search, but it comes at the cost of runtime.
For example, executing the following non recursive query:

```
SELECT ?item ?itemLabel
    WHERE{
      ?item wdt:P31 wd:486972.          .
      ?item rdfs:label "Be'eri"@en.
      SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }}
```

Figure 3: SPARQL naïve query used to search for Be'eri of type human settlement in Wikidata.

Which searches for a "human settlement" by the name of "Be'eri", yields no results. By making the following change, $(wdt: P31) \Rightarrow (wdt: P31/wdt: P279*)$, a recursive search will run for "Be'eri", with all the subtypes of "human settlement". In that case, one will yield the Q-code Q2467005, the Q-code of Be'eri, a Kibbutz in southern Israel. A Pseudo code for crawling Wikidata is provided in the appendix, Algorithm 2.

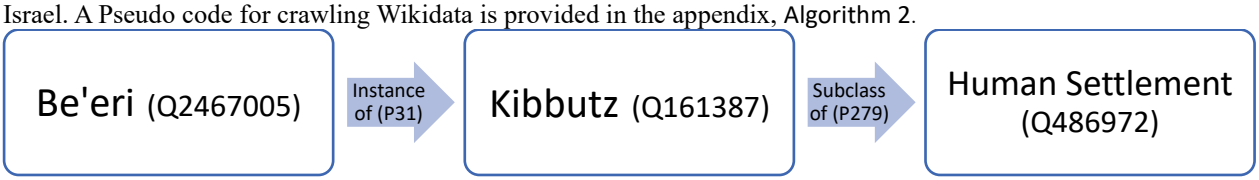


Figure 4: Classification of Be'eri in Wikidata

YAGO

url: https://yago-knowledge.org/

YAGO is a knowledge base based on Wikidata and schema.org. It was developed by the Max Planck Institute of Informatics. [8] YAGO querying has the following property: If **x** is of type **A**, and **A** is a subtype of **B**, then both (**x, A**) and (**x, B**) will be returned. This property allows the user to perform the quick **typeless query**, that searches for all entities the given name, whatever their type may be. Then, the user can filter the results for the correct type (or one of the compatible types) on **their local machine**. This causes a faster runtime, as a maximum of 2 calls to the server need to be performed, one for the original name and perhaps another one for the name with a different casing. A Pseudo code for crawling YAGO is provided in the appendix, Algorithm 3

```
SELECT ?entity ?type
   WHERE {
   ?entity rdf:type/rdfs:subClassOf* ?type .
   ?entity rdfs:label "Be'eri"@en .}
```

	entity	type
1	yago:Be'eri	yago:Kibbutz
2	yago:Be'eri	yago:Intentional_community
3	yago:Be'eri	yago:Engineering_Project
4	yago:Be'eri	yago:Technical_Project
5	yago:Be'eri	yago:Project
6	yago:Be'eri	yago:Activity
7	yago:Be'eri	schema:Event



Figure 5: Properties of Typing in YAGO. Top – Typeless query for Be'eri. Left – The results of the query. Right – Typing hierarchy of Be'eri

Experimental Evaluation

Settings

For evaluation 2 datasets were used. Both contain pairs of (name, type), where type is one of Person, Location and Organization.

- **CAC40** – The CAC40 dataset contains 64 Persons, 90 Locations, 341 Organizations, summing up to a total of 495 entities. It contains some of the most well-known entities in France. It was extracted manually from the CAC40 Wikipedia page.
- **HATVP** – The HATVP dataset contains 8894 Persons, 5210 Locations and 38894 Organizations, summing up to a total of 52998 entities. Due to its significantly larger size, many of its entities are less known. As the dataset is large, a random sample of 250 entities in each category was chosen to represent it. It was extracted from an XML file, provided by the French Government.

Analysis

As mentioned in the introduction, an evaluation regarding the performance of the algorithms provided in dealing with tasks 1 and 2 is measured. The first two measures are used to evaluate task 1, while the last is used to evaluate task 2.

1. Quantity Analysis – The success rate of getting a result based on the name and type. The results were obtained by running the **crawl** function in YAGO and Wikidata on the CAC40 dataset, and checking whether any matching result was found. It is split into YAGO, Wikidata, and the union of the two. For each (name, type), the query was run initially with the original type, and if a result **wasn't** retrieved, the name was queried again with a different type. And if a result **wasn't** retrieved this time as well, it was queried again with the remaining type.
2. Runtime Analysis – Break the runtime of the search by category, and check whether it's affected by successfully retrieving a result or not. The analysis was performed by running the **crawl** function 5 times on each entry, calculating the mean and variance for each. Then, aggregation was performed based on the parameters described above. The analysis was done on a random sample of the HAVTP dataset. It was performed on YAGO and Wikidata separately.
3. Quality Analysis – Checking whether the desired result among the found results (e.g., when searching for "Paris, Location" we expect to get "Paris, France" and not "Paris, Texas"). The analysis was done by manually evaluating if the retrieved entity matched the one expected in the context. The analysis was performed on samples of 10 random entities of each type that were found in each of the datasets (except for the sample of HATVP Organizations, where only 7 organizations were found among the 250 checked). The Evaluation took place on the results of Wikidata.

Results

Quantity Analysis

Guide: **P –** Person, **L –** location, **O –** Organization.

The values in the table below describe the proportion of entities retrieved with the queried type, out of the total number of entities of this type in the dataset. For example, the first three columns should be read the following way: Let us denote by A_0 the set of entities classified as Person in the input, and A'_0 the set of entities found as Person out of them, then $[P] = \frac{|A'_0|}{|A_0|}$. By denoting $A_1 = A_0 \backslash A'_0,\ A_2 = A_1 \backslash A'_1,$ where A'_i is defined as the set of entities that were found by using the successive type (**O** and then **L**) on A_i, we obtain $[P\ as\ O] = \frac{|A'_1|}{|A_0|}$, $[P\ as\ L] = \frac{|A'_2|}{|A_0|}$.

	P	P as O	P as L	L	L as P	L as O	O	O as P	O as L
WD	0.62	0.16	0.06	0.44	0.01	0.16	0.49	0.01	0.03
YAGO	0.45	0.09	0.05	0.6	0.01	0.05	0.37	0.11	0.02
WD+YAGO	0.62	0.16	0.12	0.61	0.01	0.05	0.49	0.01	0.03

Table 1: Breakdown of found results by type, type changes and knowledge base.

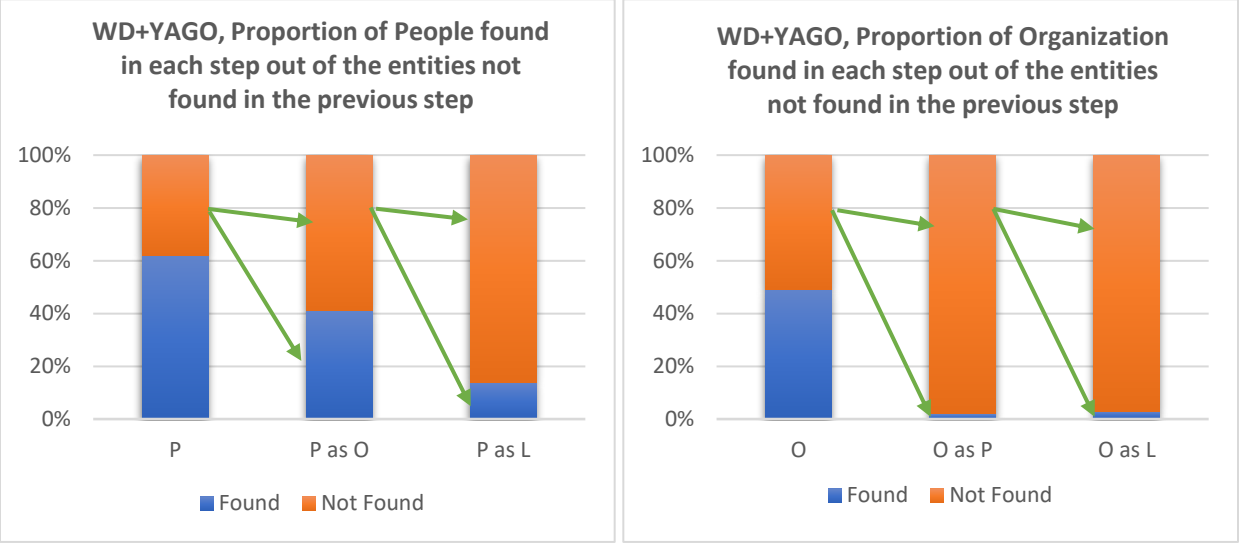


Figure 6: People and Organization, queried as various types.

Conclusions:

- Some types are unlikely to be mismatched as other types. For example, if an entity was matched as O, it is unlikely that it was mismatched. On the contrary, if an entity was classified as a P, it is rather likely to be O. Therefore, it won't be wise to recheck some entities with unlikely type, for example check an entity that was classified as location as a person, since it will increase the runtime for a very meagre gain.
- This table may be used to review the extractor typing policies, as some type mismatches are rather likely.
- It should be noted that in the samples from HATVP database the success rate was significantly lower – 0.46 for Person, 0.26 for Location, and 0.02 for Organization in Wikidata. In YAGO, it was 0.12 for Person, 0.28 for Location and 0.02 for Organization. This conforms to previous findings that knowledgebases are suitable for finding famous entities.

Runtime Analysis

Wikidata

Category	**#Found**	**Found** time mean (s)	**Found** time variance (s)	**#Not Found**	**Not Found** time mean (s)	**Not Found** time variance (s)
Organization	6	4.667	1.459	239	0.744	0.129
Person	116	0.368	0.014	134	0.548	0.028
Location	65	0.835	0.075	185	0.809	0.044

Table 2: Runtime analysis of queries in Wikidata

Yago

Category	**#Found**	**Found** time mean (s)	**Found** time variance (s)	**#Not Found**	**Not Found** time mean (s)	**Not Found** time variance (s)
Organization	4	0.489	0.008	241	0.409	0.019
Person	20	0.385	0.005	164	0.342	0.004
Location	34	0.324	0.009	79	0.354	0.003

Table 3:Runtime analysis of queries in YAGO

Conclusions:

- All queries, except the found ones in Person, take longer in Wikidata, and the difference is especially significant for organizations.
- In Wikidata, people are classified directly as a Person, without any type hierarchy that requires a more expensive search. The complexity of its typing system for Organization is visible here.
- For Person and Location, Wikidata offers a significant increase in the number of matches, for a reasonable increase in runtime. For organizations, neither database performed well on the HATVP dataset, but when results were found, it took a long time. Thereby, usage of computationally intensive tools for data acquisition with higher accuracy rates might be recommended.

Quality Analysis

Dataset	**Person**	**Location**	**Organization**
HATVP	0.8	1.	0.71 (n=7)
CAC40	0.8	0.8	1.
TOTAL	0.8	0.9	0.88

Table 4: Proportion of correctly matched entities in Wikidata

The mismatched organization in the HATVP file are *FFF* and *IDI*. This corresponds to previous research, where it was found that acronyms are hard to query. [10]
Conclusion:

- The entities that were found are likely to be the ones searched for, thus by creating a tool for task 1 of the methodology, its output is likely to satisfy task 2 as well.

Further Investigation

- A tool for dealing with task 3 was created for Wikidata. For any entity, Wikidata offers a quantitative value for each field, called *references.* It states the number of resources from which the relevant field was retrieved. High number of *references* is likely to be a sign for the importance of the field. There are two ways to generalize this approach:
    - In knowledge bases where one can develop some ranking function from the entity page itself, one can rank by this metric.
    - In knowledge bases where there is no natural way to rank the fields, one can either hardcode some most important fields to specific types, or link the resource to the same entity in other knowledge base, where a ranking function exist.

- Wikidata and Yago were chosen for this project due to their convenient API. One can try to generalize the results here on other significant knowledge bases, for example Google Knowledge Base.
- On the CAC40 dataset, the usage of the title function increased the overall percentage of results found from 0.45 to 0.50. The increase was especially prominent in organizations, where there was almost 25% increase in the number of organizations found. The efficiency of different casing functions, across different languages in cultures, should be investigated.

References

[1] "Wikipedia - Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo," [Online]. Available: https://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo.

[2] "ConnectionLens," [Online]. Available: https://team.inria.fr/cedar/connectionlens/.

[3] "flairNLP," [Online]. Available: https://github.com/flairNLP/flair.

[4] "SNER," [Online]. Available: https://nlp.stanford.edu/software/CRF-NER.shtml.

[5] "ConnectionStudio," [Online]. Available: https://connectionstudio.inria.fr/.

[6] P. H. A. A. Stephan Grimm, "Knowledge Representation and Ontologies," [Online]. Available: https://www.emse.fr/~beaune/websem/kr-onto-07.pdf.

[7] "Wikidata Main Page," [Online]. Available: https://www.wikidata.org/wiki/Wikidata:Main_Page.

[8] "YAGO Getting Started," [Online]. Available: https://yago-knowledge.org/getting-started.

[9] "W3 - RDF Concepts," [Online]. Available: https://www.w3.org/TR/rdf12-concepts/.

[10] J. C. H. G. D. S. João L. M. Pereira, "AcX: System, Techniques, and Experiments for Acronym," [Online]. Available: https://www.vldb.org/pvldb/vol15/p2530-pereira.pdf.

[11] "W3 - SPARQL query," [Online]. Available: https://www.w3.org/TR/sparql11-query/.

Appendix

Figure 7: Example SPARQL query

Data:

```
<http://example.org/book/book1>
<http://purl.org/dc/elements/1.1/title> "SPARQL Tutorial" .
```

Query:

```
SELECT ?title

WHERE

{

  <http://example.org/book/book1>
<http://purl.org/dc/elements/1.1/title> ?title .

}
```

This query, on the data above, has one solution:
Query Result:

title
"SPARQL Tutorial"

Algorithm 2: Naive Crawling, used for Wikidata.

```
function naive_crawling(name, type, n, casing_func, compatible_types):
    set results = []
    results = run_recursive_query(name, type)
    if results is empty:
        cased_name = casing_func(name)
        results = run_recursive_query(cased_name, type)
    if results is not empty:
        return results[ : n]
        else if compatible_types is not empty:
    for other_type in compatible_types:
        results =  naive_crawling( name, type, n, casing_func, [] )
        if results were found:
            return results[ : n]
    return []
```

Algorithm 3:Crawling then Matching, used for YAGO.

```
function crawling_then_matching(name, type, n, casing_func, compatible_types):
    matching_results = []
    results = run_typeless_query(name)
    cased_name = casing_func(name)
    cased_results = run_typeless_query(cased_name)
    results = results + cased_results
    foreach result in results:
        if result.type == type:
            matching_results.append(result)
        if len(matching_results) == n:
            break
    for result in results:
        if result.type in compatible_types:
            matching_results.append(result)
        if len(matching_results) == n:
            break
    return matching_results
```