Knowledge Graph
Web Search Engine - Final Project Report

Group 16

December 15, 2014

1 Motivation & Related Works

We have gone through all fundamental infrastructures of web search engine during the whole semester so far, now we are trying to get the useful information from the raw data and well organized them to be queried and utilized by the users. We want to structurally learn knowledge from the web context. As search engines become more advanced, they now tend to present a relevant and *specific* piece of information to the users without users having to click on certain links. While this is achieved through knowledge graphs, it is not the only use of knowledge graph. Knowledge graphs may be used to find related documents on the basis of underlying themes instead of simply the frequencies, and even to explore relationships which no one believes exist.

In this exploratory project, we have attempted to create knowledge graph using the (semi-)structured information present in Wikipedia. As of the writing of this report, we had extracted 827K entities and 2M relationships from 70K articles. With millions of tables, lists and info-boxes, Wikipedia is the largest organized "database" of information. We have attempted to explore a small part of this corpus and extract relationships and properties from it. Moreover, our focus has been on extracting most information without human supervision, especially in the form of predefined ontologies and relationships. This of course leads to noise and many errors, but in the long run offers very few restrictions.

Our primary motivator for a free text extractor without predefined ontologies and relationships was TextRunner, though this is certainly not the only knowledge extraction system available. KnowItAll, a forerunner to TextRunner, adopted a pure text based extraction approach but was limited in the type of information it could extract. What interested us the most was TextRunner's storage mechanism - it does not use any organized tables to store the extracted entities and relationships; instead it has a simple indexing system, quite unlike a database of carefully related tables, that stores entity-relationship-entity triples. The relationships are raw text, which means that an actual single relationship may take several forms in the index.

While focusing on pure text for information extraction, TextRunner and similar extraction do not exploit the wealth of information packed in semi-structured forms such as tables and lists. For example, to extract the awards that someone has won is much easier to do if one parses a list under such a heading, rather than look for instances in a text corpus. TextRunner overcomes this by parsing billions of documents, but perhaps such an approach is not necessary. This is what we will try to answer in this project report.

2 Design & Architecture

Our project evolved through various stages before its current form. Some functionality was considered way too advanced for this project, although in some cases it would have greatly boosted the performance of the system. The biggest and most important decision we made was to skip plain text parsing completely - even though this was the *only* form of parsing we considered when we first conceived the idea of this project. At some point we decided that infoboxes, tables and lists are probably better yet rather ignored sources of information. Full text parsing was eventually dropped, mostly done due to time limitation, since this would have a completely separate approach from table/list parsing. The departure from text parsing then led to a different paradigm, one that focuses on extracting and storing entities and their *properties*, which in turn could be other entities. We therefore arrived at a model where we index (entity-property-entity) triples.

Superlative adjective recognition

One feature that we would have liked to have in this project is the ability for the software to learn the meaning of adjectives. For example, quite often queries are made in the following form: "longest bridges in the word". For our system to return the relevant results, it would have to know that "longest" should translate to "order by length". We intended to do this by scanning list articles in Wikipedia that had titles of the form "<Superlative Adjective> <entities> in <some constraint>". We were able to successfully match columns with the adjective in the title for most such articles, by looking at the order of elements in each column. However, even though this had a very high success rate for each article, overall the data was noisy to attach meaning to adjectives, and this feature was eventually excluded.

Crawling Wikipedia

Initially, we downloaded the entire Wikipedia corpus, which unpacked to a single XML file. We used some utilities to parse the file and extract documents in Wikipedia Markup Language (Wikimarkup). However, the extracted content was not of very high quality, in that several new line characters and sometimes even parts of text were missing. Since such characters are critical to correctly parse infoboxes, tables and lists, we instead retrieved documents in Wikimarkup using the Mediawiki API. To extract information in these structures, we wrote the parsers ourselves.

Language processing

We have made significant use of the Stanford Core NLP libraries in this project. We performed POS tagging in the article titles to determine how it should be processed, and perform lemmatization on entities and properties to enhance extraction and search.

Architecture

The system consists of the following components, which are run in phases in a sequence:
Crawler (based on some seed "lists of lists", this crawls articles within a certain distance of the seed), *Parser and Extractor* (parse the infobox and tables, if any, of individual Wikipedia articles and most of special lists. All parsers get several of structured "relations" to be indexed), *Indexer* (extract important and precise information from the relation as the index, which will be used when serving), *Query Processor* (Process the query [more work if it is unstructured] and go through our index base to construct response), and *Server* (mediates between various components and the user).

3 Implementation

Details of the each component of the system follow:

1. The crawler retrieves Wikipedia documents in Wikimarkup. seeds are lists of lists, for example "lists of countries", which may in turn have other meaningful lists, such as "list of countries by GDP". etc.

2. Parsing Wikimarkup, specially for (semi-)structured data, requires very careful analysis of the Wikimarkup grammar. The same output can be achieved with several slight variations of the markup,

hence there is a variety in the markup code of all articles. The existence of templates and nesting renders the use of regular expressions insufficient, and stack-based parsing has to be employed.

Reverse relations give us precious information, so we have to utilize them carefully. For parsing infoboxes, we assign the infobox type as the property of every reverse relation. For example, the infobox type of Barack Obama is *officeholder* and he has a relation: Barack Obama (entity) → officeholder (type) → spouse (property) → Michelle Obama (entity). Then the reverse relation here is Michelle Obama (entity) → spouse (type) → officeholder (property) → Barack Obama (entity). Basically, type and property is interchangeable in our model and even in Wikipedia. They don't use verb or adjective to describe a relation. For a list, such as the list of volcanoes in Spain, the idea is the same but we get type from entity's infobox. For example, Fuerteventura (entity) → volcano (type) → country (property) → Spain (entity), in which country is extracted from Spain's infobox.

3. The inverted index follows the same idea as that of a search engine, except that each entry in the posting is a relationship-entity pair, thereby making each posting list a set of triples (in a way similar to that of TextRunner). The algorithm performs a full scan of the posting list when searching for properties of an entity.

Enhancements

- To allow better user experience, we applied loose matching when we are handling the query. That is, a query "population" will get not only "population" but also "population density" as the result, which lower the precision. Also we will have to do is applying strict matching. But in this case, we have to make sure we are doing good stemming, synonym handling and other query/index normalizing stuff. This problem needs to be solved with a machine learning approach, where we learn the usage of properties given the context.
- The indexing structure can be significantly improved. If a user wants to find property P of some entity, then a complete scan of the posting list is unnecessary and slow. Instead, a tree like structure should be adopted that makes it easier to reach a property. A tree-like structure as opposed to simply clustering the properties has the advantage that properties can be made into a hierarchy.

4 Usage

1. Compile all .java files
2. Execute wseproject.server.KGServer
3. Open a browser of your choice and then access the system via `http://localhost:25816/knowledge`
4. Input the query for fields (either just the entity or the entity and property/collection).
5. If you only do the entity query, you will be able to drop down the content of a certain property by clicking on the property and hide it back by clicking again.
6. Our engine supports union and intersection. You can achieve those by putting "or" and "and" in the query. For example, "films of Matt Damon and Brad Pitt" will give you a list of movie stared by both of them

5 Evaluation

5.1 Manaul Evaluation

Accuracy

We were interested in the accuracy of our knowledge graph. To evaluate the accuracy, we must compare it to some ground truth. Since there is no computer-processable ground truth of suitable extent, we had to rely on manual evaluation. We issued hundreds of queries and find the ground truth manually. Then compare with our results to compute the accuracy. We define five categories, person, movie, country, university, river, with several properties attached to each category. For each category, we generate a sample queries and do the experiment. Table 1 shows the accuracy.

In Table 2, we compare the size of our index to those created by other systems. Our index is several orders of magnitude more than the others. We believe that this is because these systems had wither hand-curated relationships or were very restrictive (like KnowItAll). Our indexer, on the other hand, extracts all sorts of properties, but contains a lot of noise.

6 Conclusion

Our results show a fair degree of accuracy. However, we cannot just yet claim that the approach of simply using semi-structured data from a very small corpus such as ours is advantageous. Ee have determined some approaches that can be used to parse relationships more accurately, and may take this on as a project later.

7 Contribution

Syed Ali Ahmed: List and Table Parsing, Crawling, Indexing, Server Interface, Query Processing, Project Report
Yen-Tin Liu: Infobox Parsing, Crawling, Project Report
Hao Xu: Evaluation, Consulting, Classification, Project Report

Category	**Property**	**Sample size**	**Acc.**
Country	{capital, currency, president, population, demonym}	100	76.00%
Scientist	{birth, field, nationality}	210	70.48%
Movie	{director, producer, release date}	150	58.67%
University	{established, president, motto}	94	48.97%
River	{length}	93	11.83%

Table 1: Accuracy of Knowledge Graph

Name	**Entities**	**Properties/Relations**
Group 16 KB	827,000	2,000,000
KnowItAll	N/A	29,835
SUMO	20,000	60,000
WordNet	117,597	207,016
OpenCyc	47,000	306,000

Table 2: Table 2 shows the number of entities and properties learnt in our knowledge graph. For comparison purpose, we also list the number of the most popular knowledge graph.