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* peice 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 databse of carefully related tables, that stores enitity-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.

1. Preview: Check the features and formats of Wikipedia corpus to see where we can take advantage with and how to get access to all Wikipedia pages. 2. Crawling: Based on some seed lists, parse articles within a certain distance. 3. Parsing: Parse the infobox and tables, if any, of individual Wikipedia articles and most of special list. For example the List of longest bridges in the world. All parsers get tons of structured "relations" to be indexed. 4. Indexing: Extract important and precise information from the relation as the index, which will be used when serving. 5. Serving: Process the query (more work if it is unstructured) and go through our index base to constitute response.

" Architecture 1. Crawler: Crawler.java and some supporting classes 2. Parser: Two main parts, List parser and Infobox parser and some supporting object. 3. Indexer: Indexer.java and some other help classes. 4. Nlp: Used when dealing with real sentence like list title and handling unstructured query. 5. Query: make instance of the query and handle its language structure. 6. Server: serve the user 7. Library and API: Stanford-corenlp (http://nlp.stanford.edu/software/corenlp.shtml), jsoup (http://jsoup.org/), MediaWiki (https://www.mediawiki.org/wiki/MediaWiki)

" Implementation 1. For the parsing part, since most of our targets are formatted in xml/html, an open and a close tag/parenthesis are required. Based on this feature, we used stack to check if we have got an entity and if we reach the end of the content. 2. We tried to learn the pattern from the title of a list say "List of longest bridges" to know which column (in this case, length) we should extract. We assume the column, which is in order, is out target because of the superlative adjective here. Unfortunately, because of the relatively small size of our samples, we couldn't be so sure about the effectiveness of this approach. But that's how we implement. 3. Reverse relation gives us some precious information, so we have to utilize them carefully. For parsing infobox, 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. 4. Indexer analyzes above relations and store information a map, which maps entity to their relations. When handling the query, we just check that posting-list-like record and build our response.

" Usage 1. Compile all .java files 2. Execute wseproject.server.KGServer 3. Structured query: http://localhost:25816?entity=karachi&property=population 4. To some degree, unstructured query: http://localhost:25816?query=capital+of+venezuela 5. Our engine supports union and intersection. You can achieve those by putting "or" and "and" in the query. For example, "movies of Matt Damon and Brad Pitt" will give you a list of movie stared by both of them.

" Evaluation " Improvement 1. " Group Member and Contribution Syed Ali Ahmed: List Parsing, Indexing, and Server Interface Yen-Tin Liu: Infobox Parsing, Project Report Hao Xu: Evaluation, Consulting