SEEKER: KNOWLEDGE-BASED Q & A SYSTEM FOR WIKI FARMS

by Jinglin Tao, M.S.
Washington State University
April 2019

ABSTRACT

In this paper, we built a knowledge-based Q & A system for small wiki farms, aiming at enhancing the searching capabilities of their search engines and make full use of the information on their servers.

As its name SEEKER, we will go deep into the semi-structured data and discover the relationship between articles and categories. We will realize natural language processing, graph discovering and data visualization tasks. The project is written in Python 3.

Here's the link of a two-minute demo video on Youtube, which can help you better understand of our project. The project code is available on Github.

ACKNOWLEDGMENTS

I would like to thank my academic advisor Doctor Yinghui Wu for advice on the outline, methods and implementation of this project and helping me with this final exam report. I also thank Tolkien Gateway for the original data and SerGawen for introducing the technology of current wiki farms.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION AND MOTIVATION

1.1 Introduction and Motivation

With the widespread collaborative spirit, there is an increasing willingness to share the knowledge and skills on the Internet. One of the popular knowledge-sharing projects is called setting up wiki farms. Wiki farms, also known as Wiki Hosting Service, are servers installed core wiki code, which offer editors simpler tools to create and develop individual independent wikis[1]. After the successful establishment of Wikipedia (January 15, 2001)[2], many other wiki hosting services were built in the past few years, like Wikia also known as Fandom (October 18, 2004)[3], Wikidot (August 1, 2006)[4], Huiji (March 2015)[5]. The prosperity of wiki farms also brought a great transformation to the ways of information retrieval.

Different from Wikipedia, well-known multi-language on-line encyclopedia, many small-scale, non-commercial and non-profit wikis can go deeper into their specialty with more detailed information which can help the researchers find better supporting materials and open their minds with something they didn't know before. However, these wiki farms usually don't have enough money and developers (volunteers) to go further into the optimization.

Weak search engines of those wikis will affect their high-quality data collections not used to its full potential, especially for some literature and art wikis, which include a bunch of terminologies. For example, the search engine used by Huiji uses Elasticsearch[6], based on the Lucene library[7]. Elasticsearch is an excellent search engine, and Lucene is a wonderful library as well. Though the answers exactly appear in some pages, somehow it still can't return the results we want, mainly caused by the terminology. This situation becomes even worse when the size of wiki database grows sharply.

Facing the problem caused by an inefficient search engine and increasing data volume on those wiki farms, building a more powerful and efficient search engine for those wiki farms is an urgent matter. We decide to design another way for semi-structured data searching,

[1] https://en.wikipedia.org/wiki/Wiki_hosting_service

[2] https://en.wikipedia.org/wiki/Wikipedia

[3] https://www.fandom.com/

[4] http://www.wikidot.com/

[5] https://www.huijiwiki.com/

[6] https://www.elastic.co/

[7] http://lucene.apache.org/

which can help those small wiki farms easily build an efficient search engine and make better use of their editors' contributions.Thus, we build SEEKER, a knowledge-based Q&A system for small wiki farms, which combines the tasks of Natural Language Processing, Graph Discovering, Python programming and Data Visualization.

1.2 Challenges

Among all the challenges we faced during the development, there are 5 important points:

- **Modular programming**. This project can be easily embedded in small wiki farms. The only variable need to be changed is the domain of wiki farm. The data for building graphs can be updated every week or month. It is like a plug-in that won't influence the original data and functions.
- **Data cleaning**. Though crawling original data from wiki farm isn't a hard work while using API, there are still other problems we need to concern. Every editor has his or her own writing style. It is difficult to extract specific data from a large paragraph of text which could be used to build other data structures.
- **Keyword searching**. It is essential to those literary wikis which contains fictional languages, neologism and other self-created words appeared decades ago but not having a wide use in the real world. We have to make SEEKER terminology friendly.
- **Natural language processing**. This is the most difficult part in the project. We need to design a module which can understand the meaning of natural language queries.
- **Data visualization**. Plotting clean, tidy, readable, dynamic graphs is always a big issue in illustrating results to users. We need to find a suitable tool to make outputs attractive and interactive.

1.3 Related Work

In this paper, we build a graph based on the theory of Zesch's paper [1]. It is called Wikipedia Category Graph (WCG) which has a semantical network of articles building from related terms and a taxonomy-like structure of categories. The paper concludes that WCG is a scale-free, small-world graph like WordNet [2] which can be used for NLP tasks through a series of graph-theoretic analyses according to semantic relatedness (SR) measures. Thus, if G meets the standard of those semantic networks, the NLP algorithms can be applied on this graph as it is described in the paper.

The other part is based on a project called Seq2SQL [3]. It is a deep neural network based on a Long Short-Term Memory network [4] which can translate natural language questions to corresponding SQL queries. It is trained by the WikiSQL, a dataset of hand-annotated examples of questions with their corresponding SQL, which gives rewards to learn a better policy to generate the query. Also, Seq2SQL can offer an advantage of the SQL structure and simplify the original problem.

1.4 Structure of the Paper

This paper contains 6 chapters. Chapter 1 talks about the motivation of setting up this program, the introduction of SEEKER and the problems need to be solved. Chapter 1.3 shows the process of how to collect data, how to extract information and how to construct the Knowledge-graph from Text-corpus. Chapter 3 discusses a method of teaching SEEKER understand natural language questions by changing the natural language questions to SQL queries. Chapter 4 combines the second and third chapters, which describes the frame of SEEKER with flow charts and the tools used for data visualization. Chapter 5 gives four scenarios to explain the process of SEEKER intuitively and what kind of results we can get from the Q & A system. The last chapter concludes the entire work and looks forward into the future development.

CHAPTER 2. CONSTRUCT THE KNOWLEDGE-GRAPH FROM TEXT-CORPUS

In this part, we will get data from the public API of a wiki farm, extract information from the raw data and design the graph structure.

2.1 Problem Statement

From this section, we will state our problem definition. The data structure from Tolkien Gateway is like the one from Wikipedia. Raw data shared by GNU Free Documentation License[8] which we can get directly from request messages by using web crawlers through TG APIs and save the data into local text files for the next steps. Then, based on the paper of WCG [1] (mentioned in section 1.3), an article graph and a category graph can be set up from the raw data. Category graph consists of categories and their affiliations on Wikipedia, which are organized in a taxonomy-like structure. Article graph consists of articles on Wikipedia, taking each article as an entity and inner links between articles as edges. According to the paper [1], the category graph is a scale-free, small world graph and article graph is a scale-free, heavily linked graph.

As this search engine needs to meet the requirements of Graph discovery and natural language process, an undirected graph with labels and attributes is necessary to the experiment, which can also fit the structure of wiki pages (articles) with categories.

So, we define the input raw data as $D := (P, C)$, and the final output graph as $G := (V, L, E)$. D represents raw data combined with P as wiki article/content pages and C as category pages. G refers to the undirected graph we need to build from D to store these relationships, including V as vertices, a collection of nodes (article Ids); A as attributes, a small set of data linked with its attribute name described each article vertex $v \in V$ in detail;L as labels, a collection of nodes (category Ids), separated with V by different shapes and colors; E as edges, a group of relationships between nodes, including four different kinds: V and V belongs to E_V, L and L belongs to E_L, V and L belongs to E_{LV}, V and A belongs to E_A [5]. We also take $G_L := (L, E_L)$ as label graph (like category graph mentioned in paper of WCG [1]), like Figure 2.1a, $G_V := (V, E_V)$ as article graph, $G_{LV} := (V, L, E_L, E_{LV})$

[8] http://www.gnu.org/licenses/fdl-1.3.en.html

Notation	Definition and Description
P	wiki article/content pages
C	category pages
$D := (P, C)$	input raw data
V	vertices, a collection of nodes (article Ids); $v \in V$
L	labels, a collection of nodes (category Ids); $l \in L$
A	attributes, a collection of nodes describing vertices; $a \in A$
E_V	a collection of edges between vertex and vertex
E_L	a collection of edges between label and label
E_{LV}	a collection of edges between label and vertex
E_A	a collection of edges between attribute and vertex
$G_L := (L, E_L)$	label graph
$G_V := (V, E_V)$	article graph
$G_{LV} := (V, L, E_L, E_{LV})$	article with category graph
$G_{VL} := (V, L, E_L, E_{LV}, E_V)$	category-article graph
$G := (V, L, E)$	final output graph data
T	relational table of attribute; $t \in T$

Table 2.1: The Definition of Notations

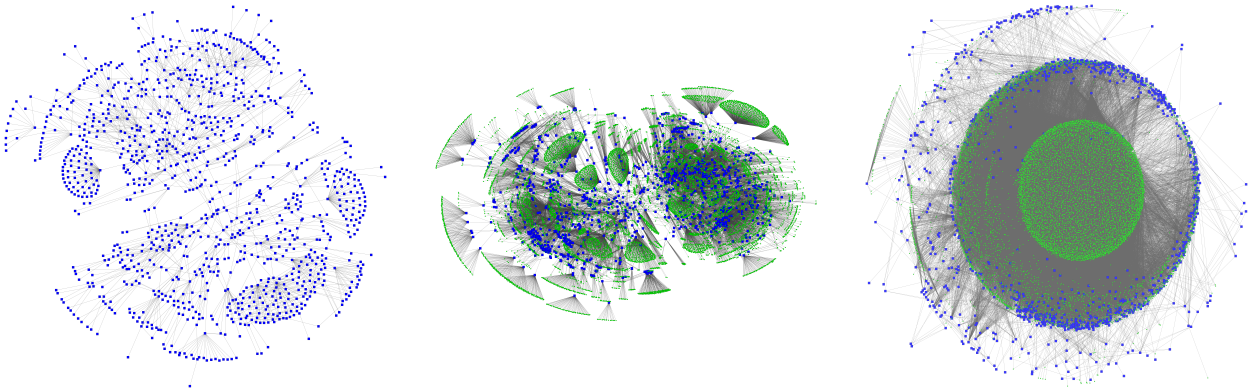
as article with category graph, $G_{VL} := (V, L, E_L, E_{LV}, E_V)$ as category-article graph, like Figure 2.1b. We also build a relational table T with A to store those tuples for next chapter.

2.2 Summary of Technique

For the original raw data, we choose the semi-structure data from Tolkien Gateway Website[9] also known as TG. This is a not-for-profit collaborative wiki which aims to collect and organize the works of J.R.R. Tolkien for fans and researchers, like a Tolkien encyclopedia, while Tolkien Estate holds the copyright over the literary texts[10]. What's more, Tolkien Gateway has become the largest and most well-built Tolkien-related encyclopedia on the

[9]http://www.tolkiengateway.net

[10]https://en.wikipedia.org/wiki/Tolkien_Estate



(a) Category Graph **(b)** Article with Category Graph **(c)** Category-article Graph

Figure 2.1: Process of Standardizing a Graph from Raw Data

World Wide Web since 2010 based on research by Mith[11], which gives [12]. It offers APIs[13] with documents in detail for users to craw data from its website. The APIs are a little different from the ones of Wikipedia, so we will discuss them in the following subsections. To get the raw data from a wiki farm, clean the data and change the semi-structure files into a graph, we use these techniques listed below.

Scrapy[14] is an open-source web-crawling framework which works with Python program using BSD License[15]. It can be used as a data extraction with APIs or as a general-purpose web crawler[16]. This framework can help us to get raw data from a wiki farm. The raw data can be set in many structures. Considering the convenience for next steps, the data is organized in Extensible Markup Language (XML). Scrapy also offers a demo frame which can help users build their own crawler. Next, we need to extract data from the XML sources. BeautifulSoup is a popular web scraping python library though a bit slow[17] and lxml is an XML parsing library with ElementTree API[18] though not in Python standard library. Thus,

[11]Noticing that LOTRO-Wiki on the top of the list with 86,649 articles is a game wiki exclusively "Lord of the Rings Online"-related. Thus, it isn't a good choice to do the research work based on a Massively Multiplayer Online Role-Playing Game set in Tolkien's Middle-Earth.

[12]LOTRO-Wiki, "Statistics" (retrieved 15 March 2019)

[13]Tolkien Gateway, "API" (retrieved 15 March 2019)

[14]https://en.wikipedia.org/wiki/Scrapy

[15]https://en.wikipedia.org/wiki/BSD_licenses

[16]https://scrapy.org/

[17]https://www.crummy.com/software/BeautifulSoup/

[18]http://lxml.de/

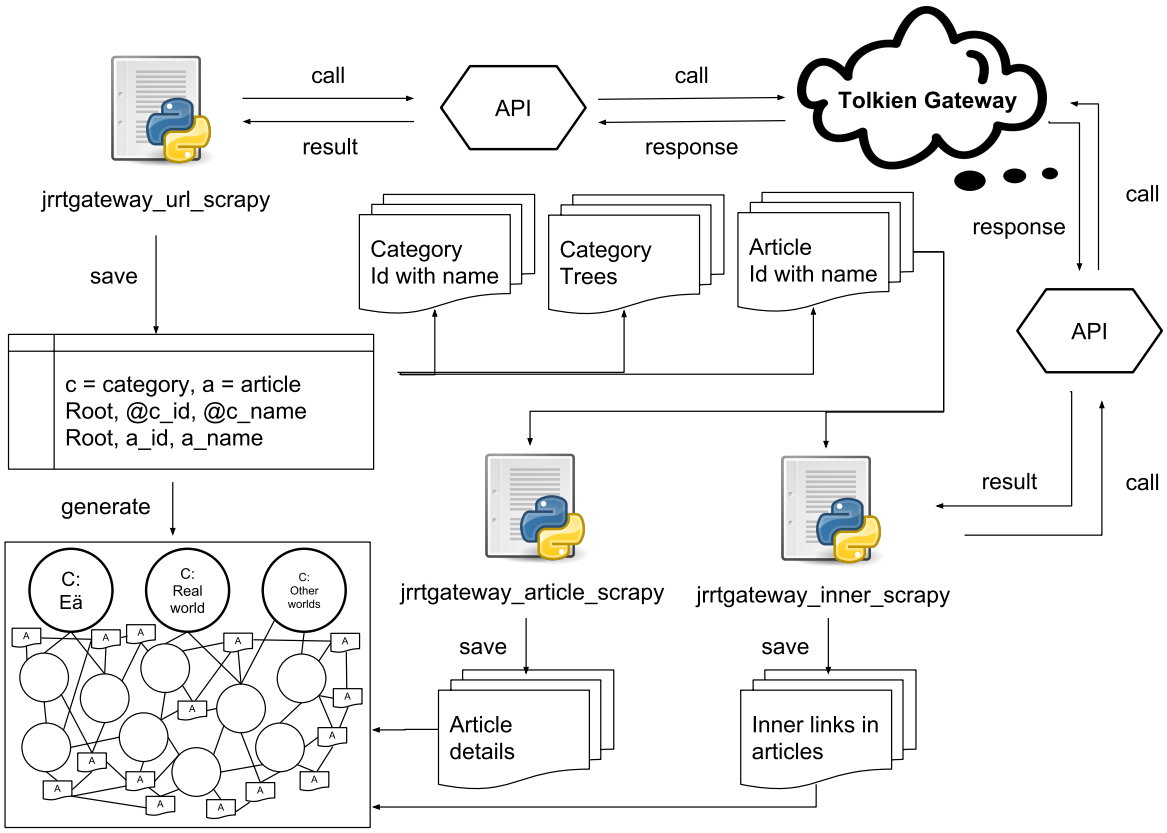


Figure 2.2: Process of Standardizing Graph Data

we choose to use Scrapy Selectors for data extraction. Categories (id, name), category tree and articles (id, name, detail, inner link) can be stored after this step.

Finally, we need a library to build an undirected graph with labels and attributes. igraph is an open source package with network analysis tools, well-known for its high efficiency and portability, supporting R, Python, Mathematica and C/C++.

2.3 Implementation

We use demo frame of Scrapy to build three major crawlers, and get the categories, articles and the inner links between categories and articles. The process illustrated in Figure 2.2 will be discussed in the following sections.

URL crawler is labeled as jrrtgateway_url_scrapy in Figure 2.2.

1. Focus three category roots ("E", "Real-world" and "Other fictional worlds") in this wiki. Take "E" as example. Set API parameters as action: "query", list: "catego-

Parameter	TG graph G_L (our work)	TG graph G (our work)	WCG [1]	WordNet [2]	Explanation
$\|V\|$	1120	13332	27,865	122,005	number of vertices
D	13	8	17	27	maximum eccentricity of any vertex
$\bar{k}$	2.85	62.53	3.54	4.0	avg number of edges connected with a vertex
γ	2.66	1.91	2.12	3.11	a particularity of scale-free network
$\bar{L}$	7.05	2.70	7.18	10.56	average shortest path lengths
$\bar{L}_{random}$	∼6.70	∼2.30	∼8.10	10.61	average shortest path length for a random graph
C	0.021	0.25	0.012	0.027	clustering coefficient
$\bar{C}_{random}$	0.0087	17.75	0.0008	0.0001	clustering coefficient for a random graph

Table 2.2: Parameter Values of G_L, G, WCG [1] and WordNet [2]

rymembers", cmlimit: 1000, format: "xml", cmtitle: "Category:E". Put "E", "Real-world" and "Other fictional worlds" in stack S and dictionary L.

2. Pop the last item i in the stack, change "cmtitle" to "Category:i" and run the crawler.
3. Extract data from XML files returned by API request and get children's ids and names (including categories, articles, files and other special pages). If a child is still a category, push the category name into the stack S, add category id to the key of L and set category name as value. If a child is an article, article id will be added to the key of V and set article name as value. Then, relationships are appended between l and l or l and v to list E.
4. Repeat step 2 and 3, until stack S is empty.

Inner Links Crawler is labeled as jrrtgateway_inner_scrapy in Figure 2.2.

1. Get article Ids K from V keys.

2. Set API parameters as action: “query”, generator: “links”, gpllimit: “max”, format: “xml”, pageids: “k” and run the crawler.
3. Extract inner link nodes and add the relationships between vertices to edges E.
4. Repeat Step 2 and 3, until get all article inner links.

Article Crawler is labeled as jrrtgateway_article_scrapy in Figure 2.2.

1. Get article Ids K from V keys.
2. Set API parameters as action: “query”, generator: “links”, gpllimit: “max”, format: “xml”, pageids: “k” and run the crawler.
3. Extract inner link nodes and add the relationships between vertices to edges E.
4. Repeat Step 2 and 3, until get all article inner links.

Finally, we merge the clean data extracted from the results of three crawlers shown in Figure 2.1. We also calculate the graph parameters of G_L and G listed in Table 2.1. The parameters from the paper of WCG [1] are number of vertices $|V|$, diameter D, average degree $\bar{k}$, power law exponent γ [6] [7], average shortest path length $\bar{L}$, average shortest path length for a random graph $\bar{L}_{random}$ [8], clustering coefficient C [9] and clustering coefficient for a random graph $\bar{C}_{random}$ [10].

CHAPTER 3. INTERPRET NATURAL LANGUAGE TO STRUCTURED QUERY

In this chapter, we begin to set up the NLP module required in the Q&A system.

3.1 Problem Statement

Now we have a cleaned dataset which includes vertices V and their attributes A. Giving a better way to connect with the MySQL database of the wiki farm. Thus, we decide to change the natural language question to a SQL query. The relationships between V and A are changed into a relational database T. each attribute has its own table $t \in T$.

The graph with labels and attributes is still kept for the basic searching, giving the user an intuitive grasp of what they want. To enhance the comprehension of search engine, we annotate manually 100 questions with its corresponding SQL queries like WikiSQL [11].

3.2 Summary of Technique

SQLite can build lightweight disk-based SQL databases not requiring installing separate server process. However, SQLite is a C library which needs interface to interact with other programming language. Thus, we choose sqlite3 to finish this part of work. The sqlite3 module is a DB-API 2.0 interface for SQLite databases written by Gerhard Hring supporting Python language. NLP2SQL is a problem which has many ways to solve it. One of them is using a deep neural network [11]. In another paper of Seq2SQL [3], the author gives a model of translating natural language questions to corresponding SQL queries. However, small wikis doesn't have enough original data to support the reinforcement learning. Thus, we choose a simpler implementation called Ln2SQL based on Fr2SQL [12]. Although Ln2SQL is not quite state-of-the-art and has some problems of finding the stems, it has speed optimizations which can save a lot of time.

3.3 Implementation

Firstly, build the relational table with tuples (node id, attribute name and attribute id). The attribute ids are node ids, which can be used for graph data visualization in next chapter. Then, we fork an implementation called Ln2SQL from Github. We update our new features and teach this program to reorganize the terminology from our database. Finally, when it can correctly reply some simple questions, we pack this NLP2SQL module and move to the next step.

CHAPTER 4. SYSTEM IMPLEMENTATION

In this chapter, we will combine the work of Chapter 2 and Chapter 3. A data visualization task also includes in this chapter.

4.1 Architecture of System

The flow chart Figure 4.1 illustrates the frame of whole project. The Q & A system is separated in three parts:

1. Data Processing, including data crawling, data cleaning, data storing and data standardizing, referring to the work in Chapter 2.
2. Search Engine, including subgraph generating, natural language processing and APIs with 3 databases, referring to the work in Chapter 3.
3. Data Visualization, including user interface, Q & A system module, APIs with search engine and enhancing user search results.

4.2 Visualization Technology

We use Plotly[19] and Dash[20] for the data visualization task. They are modern analytics applications which can build beautiful web-based interfaces in Python.

At first, the project is built with Kivy and the plot function of igraph and the interface only loads the static figures. However, the interaction between these two packages can't realize what we desire. Different purposes with different builders influence their compatibility. It's hard to build a dynamic graph which can interact with users.

Then, we decide to search for other tools to solve the data visualization tasks and choose the plotly instead. It can plot powerful, tidy, clean, dynamic graphs which meets our requirements, especially Scatter3d component. It is used for graph data plotting and user interaction. HTML components are used for listing text answers. Dropdown and input components are also used for user interaction.

4.3 Implementation of Data Visualization

We set an input box on the main page and it has two modes. One is called Keyword Search, the other is called Fuzzy Search.

[19]https://plot.ly/

[20]https://plot.ly/products/dash/

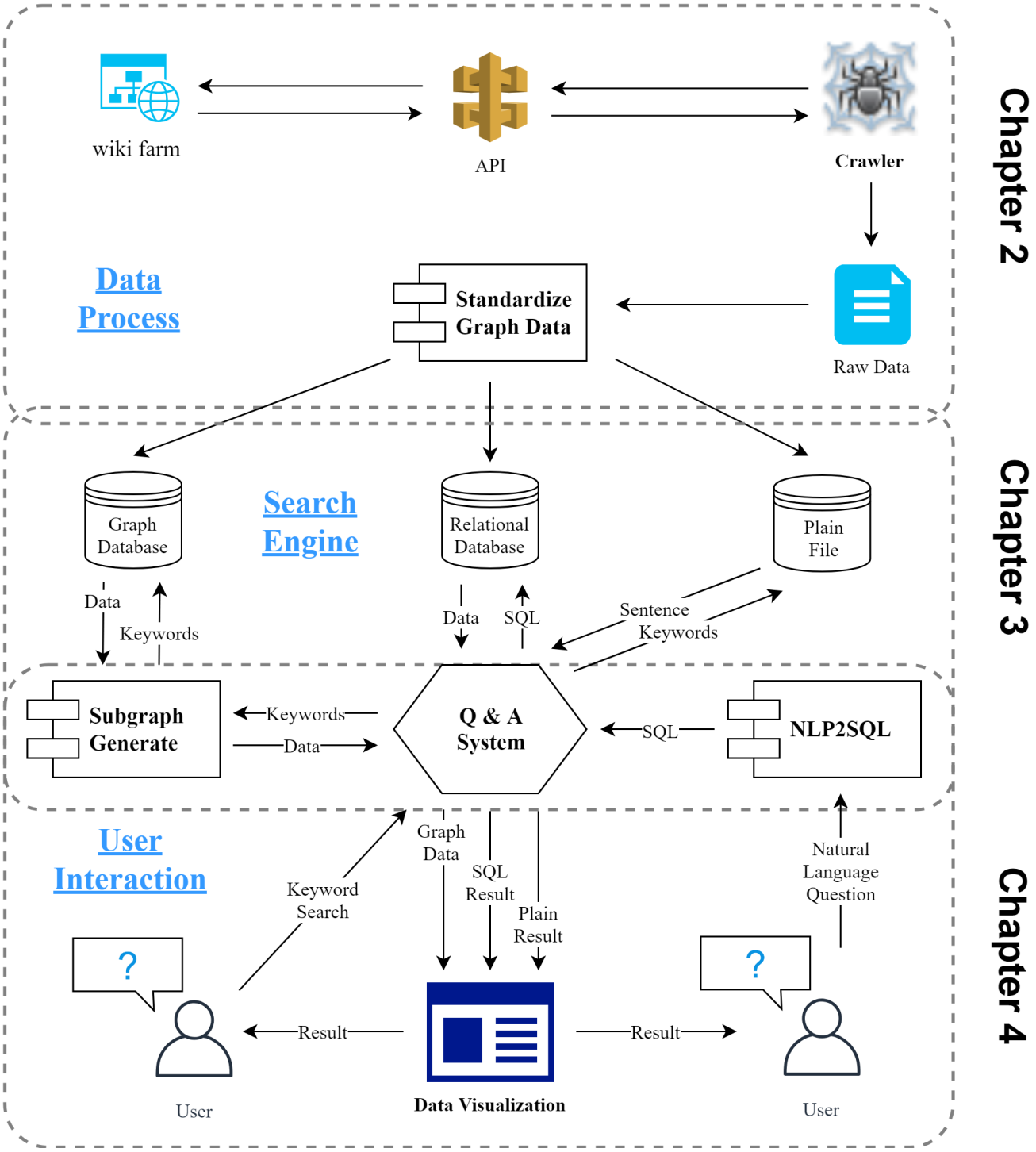


Figure 4.1: The Structure of System Implementation

In Keyword Search mode, the background program will find the articles and categories with similar characters to what user inputs. SEEKER will also return the plain file of the last keyword in the input box, and the text will print on the right side of the screen.

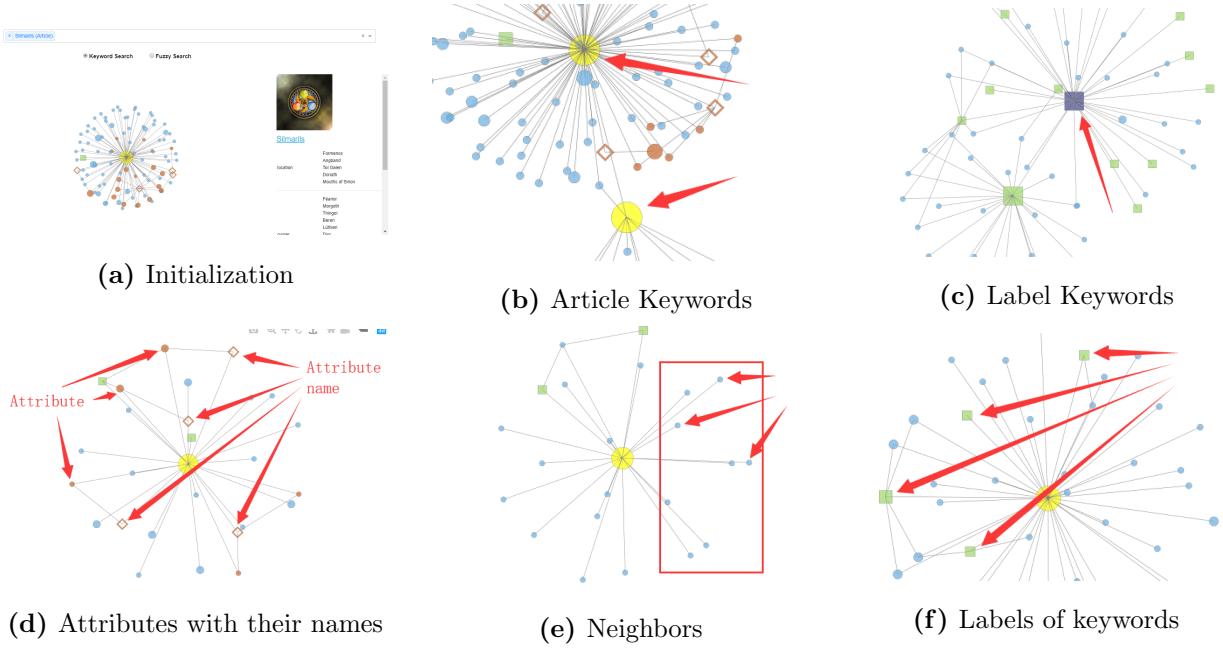
In Fuzzy Search, the question that user types into the box will be sent to NLP2SQL module and translated into SQL query. After that, SEEKER will return the result and print it on the right side of the screen.

In both cases, if there are keywords in the questions, the Subgraph Generate module will create a subgraph with keywords as its pivots, the relationships between keywords, categories and attributes of article keywords and their neighbors. Once the keywords change, the graph will regenerate immediately.

CHAPTER 5. EXPERIMENT AND DEMO

In this chapter, we will give four scenarios to describe how SEEKER works. Here is the link of a two-minute demo video on Youtube.

5.1 Scenario 1: Keyword Matching



(a) Initialization **(b)** Article Keywords **(c)** Label Keywords

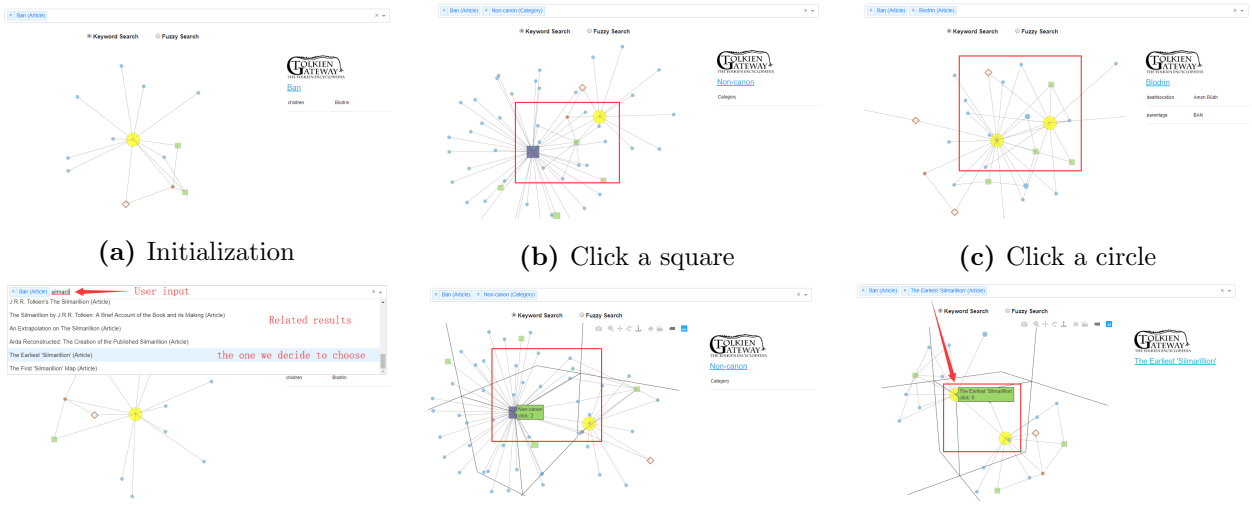
(d) Attributes with their names **(e)** Neighbors **(f)** Labels of keywords

Figure 5.1: Scenario 1: Keyword Matching

In Figure 5.1, we can see the initialization of SEEKER (Figure 5.1a). The yellow circles represent article keywords (Figure 5.1b), while purple squares represent label keywords (Figure 5.1c). Red circles are attributes, which connected to their red square-frame attribute names (Figure 5.1d) and blue circles are neighbors of keywords (Figure 5.1e). Green squares illustrate the labels of keywords (Figure 5.1f). The size of those shapes shows their importance to the keywords.

5.2 Scenario 2: Finding Relationships

In scenario 2, we will introduce two ways to find the relationships between keywords. The first way is clicking a linked green square (label) (Figure 5.2b) or a blue circle (article) (Figure 5.2c). The second way is using the input box. Users can input what they want to find and the program will help them find the similar keyword (Figure 5.2d). We can see the



(a) Initialization **(b)** Click a square **(c)** Click a circle

(d) Input a word **(e)** Choose a label keyword **(f)** Select an article keyword

Figure 5.2: Scenario 2: Find Relationships

edges and nodes inside the red frame are the relationships between keywords in the input box (Figure 5.2e and 5.2f).

5.3 Scenario 3: User Feedback Enhancement

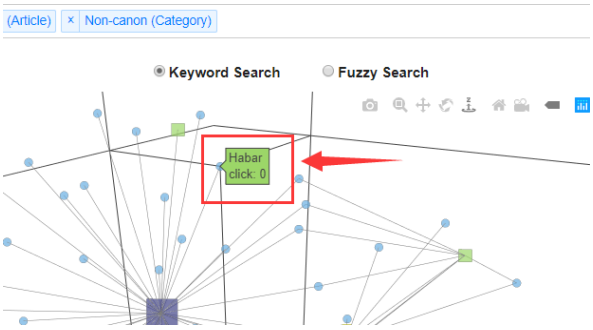
The user feed back enhancement works when SEEKER plots the graph and users click the circles and squares.

There is a file which records the action of user click events. Every time the user clicks on the node, the click times of that node will plus 1. What's more, the click times is added to the size of node. If the user clicks one node for several times, the size will increase until it reaches the upper bound.

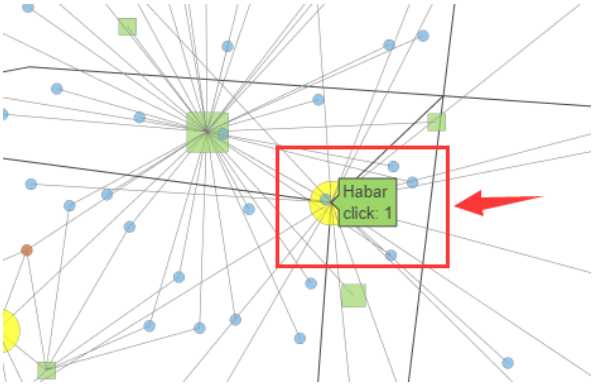
In Figure 5.3a, we can see the click times of "Habar" is 0. After the user clicks that blue node, it becomes a keyword (larger yellow circle) and the click times increases to 1 (Figure 5.3b). In Figure 5.3c, the click times of "Non-conon" is 2. After clicking, it becomes a keyword (larger purple square) and the click times raises to 3 (Figure 5.3d).

5.4 Scenario 4: Answering Natural Language Questions

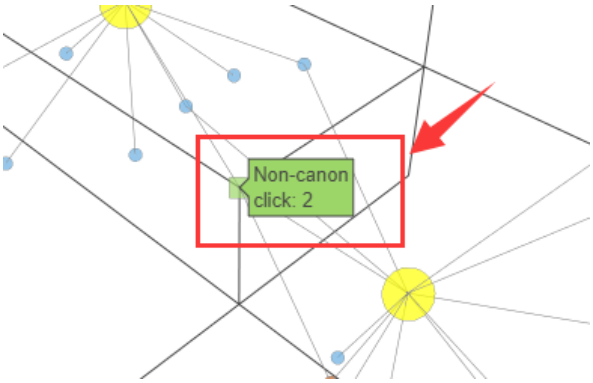
In this scenario, we will test the natural language processing part. The user asks a question "How many owner(s) there are with the object name like Silmarils?". Then, SEEKER can give the answer of this question as "11" on right side of the screen (Figure 5.4a). Thus, users can get the information directly and don't need to count the names one by one.



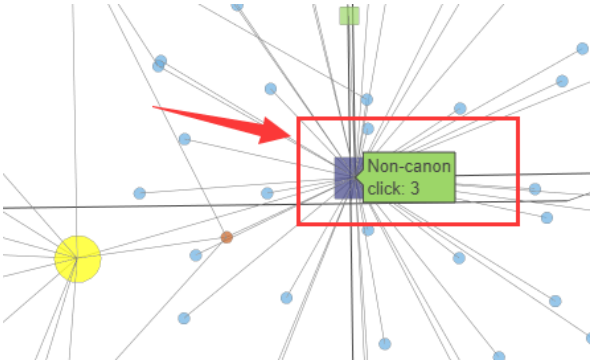
(a) Before click an article node



(b) After click an article node

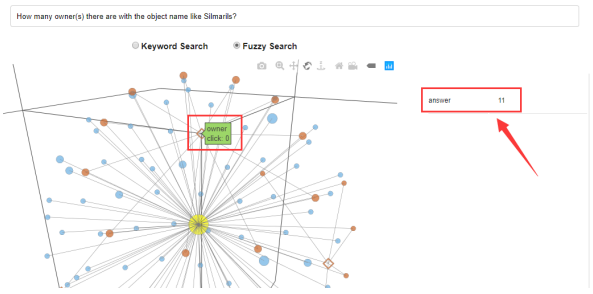


(c) Before click a label

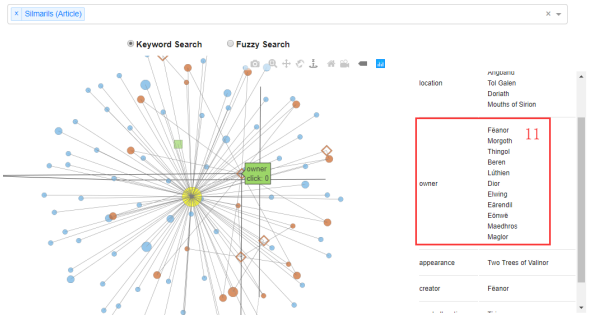


(d) After click a label

Figure 5.3: Scenario 3: User Feedback Enhancement



(a) Fuzzy Search



(b) Using keyword search to verify results

Figure 5.4: Scenario 4: Answering Natural Language Questions

Finally, we use the keyword search to verify the result. In Figure 5.4b, we can see that the 11 owners of Silmarils are Fanor, Morgoth, Thingol, Beren, Lthien, Dior, Elwing, Erendil, Enw, Maedhros and Maglor. Thus, the answer of Fuzzy Search is correct.

CHAPTER 6. CONCLUSION AND FUTURE WORK

In this paper, we develop a Q & A system for a small wiki farm which supports both keyword searching and natural language querying. Users can interact with the dynamic graph and more detailed semi-structured explanation on its side. SEEKER is more powerful than the original search engine.

Then, we want to move a bit forward from NLP2SQL method. For now, we only use the attributes of each article for natural language processing and doesn't make full use of the knowledge graph. What's more, there are restrictions of using SQL queries to answer more complicated questions. In the next step, we want to teach SEEKER understanding the plain text from the content of each article and expand our knowledge graph. Then, using subgraph isomorphism over the graph to answer the more detailed natural language questions from the users [13].

Bibliography

[1] T. Zesch and I. Gurevych, "Analysis of the wikipedia category graph for nlp applications," in *Proceedings of the Second Workshop on TextGraphs: Graph-Based Algorithms for Natural Language Processing*, 2007, pp. 1–8.

[2] G. Miller, *WordNet: An electronic lexical database*. MIT press, 1998.

[3] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *CoRR*, vol. abs/1709.00103, 2017.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[5] M. E. Newman, "The structure and function of complex networks," *SIAM review*, vol. 45, no. 2, pp. 167–256, 2003.

[6] M. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary Physics*, vol. 46, no. 5, pp. 323–351, 2005.

[7] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[8] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[9] S. Wasserman and K. Faust, *Social network analysis: Methods and applications*. Cambridge university press, 1994, vol. 8.

[10] V. Zlatić, M. Božičević, H. Štefančić, and M. Domazet, "Wikipedias: Collaborative web-based encyclopedias as complex networks," *Physical Review E*, vol. 74, no. 1, p. 016115, 2006.

[11] P. Yin, Z. Lu, H. Li, and B. Kao, "Neural enquirer: Learning to query tables with natural language," *arXiv preprint arXiv:1512.00965*, 2015.

[12] B. Couderc and J. Ferrero, "fr2sql: Interrogation de bases de données en français," in *22ème Traitement Automatique des Langues Naturelles*, 2015.

[13] S. Hu, L. Zou, J. X. Yu, H. Wang, and D. Zhao, "Answering natural language questions by subgraph matching over knowledge graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 5, pp. 824–837, 2018.