

SHANGHAI JIAO TONG UNIVERSITY

**学士学位论文**

BACHELOR’S THESIS



论文题目：Mining Cultural Difference from Multi-Source Big Data

学生姓名: 陶宇超

学生学号: 5130309765

专 业: 电子信息科学类（计算机）

指导教师: 朱其立

学院(系): 电子信息与电气工程学院

**Contents**

[Chapter One Introduction 4](#_Toc484005741)

[1.1 Previous Work 4](#_Toc484005742)

[1.2 Similar Problems 6](#_Toc484005743)

[1.3 Challenges 7](#_Toc484005744)

[1.3.1 Linear Reasoning 7](#_Toc484005745)

[1.3.2 Word Representation 10](#_Toc484005746)

[1.3.3 Quality of the graph 14](#_Toc484005747)

[1.4 Conclusion 17](#_Toc484005748)

[Chapter Two Approaches 17](#_Toc484005749)

[2.1 Fetch graph data 18](#_Toc484005750)

[2.1.1 Extraction from plain text 18](#_Toc484005751)

[2.1.2 Use extracted data set 22](#_Toc484005752)

[2.2 Establish a query engine 30](#_Toc484005753)

[2.3 Build the Graph 33](#_Toc484005754)

[2.4 Compare the graph 39](#_Toc484005755)

[2.4.1 Linear regression 39](#_Toc484005756)

[2.4.2 The concept of layer level 44](#_Toc484005757)

[2.4.3 The concept of abstract entity 48](#_Toc484005758)

[2.4.4 Compare the elements in phonetic and spelling feature 50](#_Toc484005759)

[2.4.5 Similar words 51](#_Toc484005760)

[2.5 Visualization 51](#_Toc484005761)

[2.6 Conclusion 55](#_Toc484005762)

[Chapter Three Code Design 55](#_Toc484005763)

[3.1 Structure of the codes 55](#_Toc484005764)

[3.2 Element 57](#_Toc484005765)

[3.3 Cache 58](#_Toc484005766)

[3.4 Util 59](#_Toc484005767)

[3.5 W2VServer 59](#_Toc484005768)

[3.6 NEResolver 59](#_Toc484005769)

[3.7 EN2CNDict 61](#_Toc484005770)

[3.8 GraphBuilder 61](#_Toc484005771)

[3.9 GraphMatcher 63](#_Toc484005772)

[3.10 GraphStatistics 63](#_Toc484005773)

[3.11 GraphVisualizer 64](#_Toc484005774)

[3.12 Conclusion 64](#_Toc484005775)

[Chapter Four Result 64](#_Toc484005776)

[4.1 The result of GraphMatcher 65](#_Toc484005777)

[4.1.1 The first version of result 65](#_Toc484005778)

[4.1.2 The second version of result 65](#_Toc484005779)

[4.2 The result of Visualizer 66](#_Toc484005780)

[4.3 Conclusion 67](#_Toc484005781)

[Chapter Five Conclusion 68](#_Toc484005782)

[Chapter Six Acknowledgement 68](#_Toc484005783)

Chapter One Introduction

The 21st century is an information explosion era. The word ‘Big Data’ is presented to describe the phenomena that numerous amount of data is being generated in the current daily life. Information is exchanged in a rapid speed through Internet. Hotspot no longer springs just one month a time, it springs every day. Moreover, with the wide spreading of smart phone, it has become much more easily for people to fetch the latest information. People currently are used to exploring interesting staff or other information on online social platforms/community, like Sina Weibo, Wechat Group, Zhihu. New terms are created and spread widely since some event that attracts a lot of people happens. These kinds of new terms have a strong relation to the entity they refer, like ‘平西王’ refers to ‘薄熙来’ due to the similarity between these two entities. We call this kind of new term as ‘Morph’, while the one that it refers is called ‘Target Entity’. But due to the aging effect, people who are not familiar with the background of the creation of the morph may feel confused about the origin of the new term. Only when the relevant event is still the hot spot can people easily find the reason of the origination, because numerous We-Media write blogs to analyze and explain the event, which will introduce the meaning of the morph. So when the period of the hot spot passes, it is hard to understand the originating relation between the morph and the target entity. Moreover, due to the blockade of the Internet, some web documents are not accessible to most of people, which aggravate the blindness to understanding the originating relation. Thus it is meaningful to find a way, which can explain such kind of latent relation. This is the purpose of my project.

* 1. Previous Work

Before I presented this project, I’ve read three papers about morph, which inspired me to do this project. Following the temporal order, they are ‘Resolving Entity Morphs in Censored Data’([1], Huang et al., 2013), ‘Be Appropriate and Funny: Automatic Entity Morph Encoding’ ([2], Zhang et al. 2014), ‘Context-aware Entity Morph Decoding’([3], Zhang et al. 2015). The first and third paper is to find out morphs from corpus, namely to detect the entity and further detect whether it is a morph or not. The second paper is to generate morphs from some entities. The first paper uses a graph to compute the similarity between a target morph and the base entity, as the Figure 1 shows. Figure 1 is the ‘Figure 4: Example of Morph-Related Heterogeneous Information Network’ in ([1], Huang et al., 2013)’s paper.

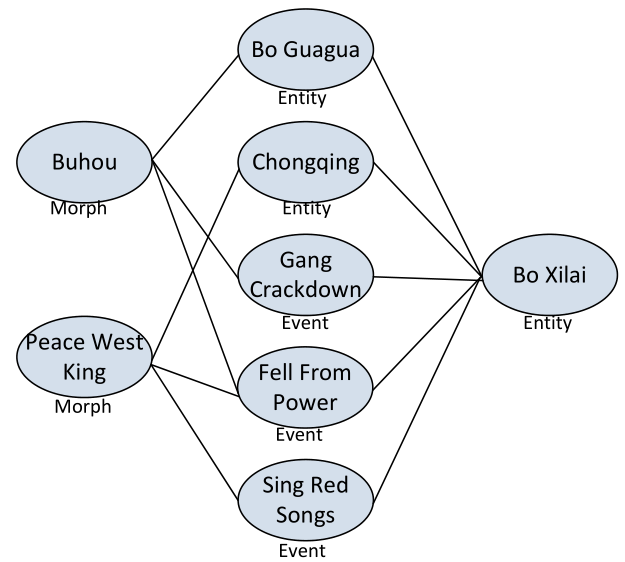


Figure Morph-Related Heterogeneous Information Network

It calculates the similarity based on the graph, along with other prior knowledge, such as the social features. This graph gives me a lot of inspiration. If machine can learn the strong relation between a morph and its target entity, human beings can also learn from it. But this relation network in Huang’s paper is used to candidate the best morph. It’s a comparison between the numbers. So the absolute number of the score may not be able to reflect the originating relation between the morph and its target entity. The third paper pays more attention on the morph verification and morph candidate ranking. The second paper introduces 7 ways to generate a man-made morph, from phonetic view, spelling view, nickname view, translation and transliteration view, semantic view, historical figure view, and characteristics view. These views show clearly how a morph can relate to an entity.

* 1. Similar Problems

The problem, explaining the originating relation between a morph and its target entity, is very similar to reason the relation between two entities from the Knowledge Graph. This word is presented by Google, whose purpose is to enhance its search engine. This graph is useful in semantic searching. The query like ‘Who is Thomas Jefferson?’ is processed with the help of Knowledge Graph. The nodes within this graph represents the real entities that exist in the real world, the edges within this graph represents the relations between the entities. You may get the result like ‘Born: April 13, 1743’ from this query, inside which ‘Born’ is the relation edge and ‘April 13, 1743’ is the entity node. As to my project, the query may become ‘Why 平西王 refers to 薄熙来?’. It may tell you that they are both related to ‘重庆’, ‘政府’. So it is very similar. But the difference is, to explain the originating relation between the morph and its target entity is a subset of the general Knowledge Graph query, also it needs a more complicated explanation other than just tell you a simple list of relation-entity tuples.

You may hear another word ‘Linked Data’. This term is created by Tim Berners-Lee (the director of W3C), which is a publishing method, aiming to link data to help semantic query. This concept is very similar to Knowledge Graph, but this term becomes a standard to create this kinds of graph. It is based on HTTP, RDF and URI. RDF is used to describe tuples, URI is used to locate resources, HTTP is used as the transport protocol. Following this publishing standard, there are some datasets existed: DBpedia, FOAF, GeoNames, UMBEL, Wikidata. DBpedia is extracted from Wikipedia, has almost 10 thousand tuples in 11 languages. FOAF mainly describe the relation between people. GeoNames has >= 7,500,000 geometric information.

Another good data set is called YAGO. Yet Another Great Ontology. It is extracted from Wikipedia (e.g. categories, redirects, infoboxes), Wordnet (e.g., synsets, hyponymy), and GeoNames. It has need linked to DBpedia ontology and to the SUMO ontology. It provides dumps in Turtle or TSV forms. It also provides query endpoint (Browser or SPARQL).

Ontology is a term which is similar to the namespace in C++. Each field has its own ontology. To separate the knowledge into different fields, we use ontology to describe the aggregation of different knowledge.

Entity Cube is a good relation searching engine build by Microsoft. It is embedded in Bing. When I search ‘薄熙来’ (In Chinese), I can get terms separated into three categories: People, Locations and Organizations. From my perspective, these outs are highly related to ‘Bo Xilai’. But the problem is they are just been put into a big category, which lose the more exact relationship to the target entity. For example, ‘Bo Guagua’ is in the category of People, but it doesn’t show the ‘son’ relation. This is the case when I search in Chinese. The result is better if I search in English. For example, input ‘Bo Xilai’ (In English), ‘Bo Guagua’ is marked the ‘son’ of ‘Bo Xilai’. But the final problem is, it only supports the relations in English corpus. So if I want to leverage this Entity Cube, I have to do a lot of translation work. Which is another hard part of NLP.

* 1. Challenges
     1. Linear Reasoning

Most of the graph reasoning is based on the linear relation chains between two entities, as the Figure 2 shows. It illustrates the linear logical path between the entity ‘Steve Jobs’ and the entity ‘Apple Inc.’. They both have one common entity ‘California’, which is the birth/death place of ‘Steve Jobs’ and the location city of ‘Apple Inc.’. People can tell the common part between ‘Steve Jobs’ and ‘Apple Inc.’ easily from this graph. People can say, these two entities are similar due to the common parts they share.

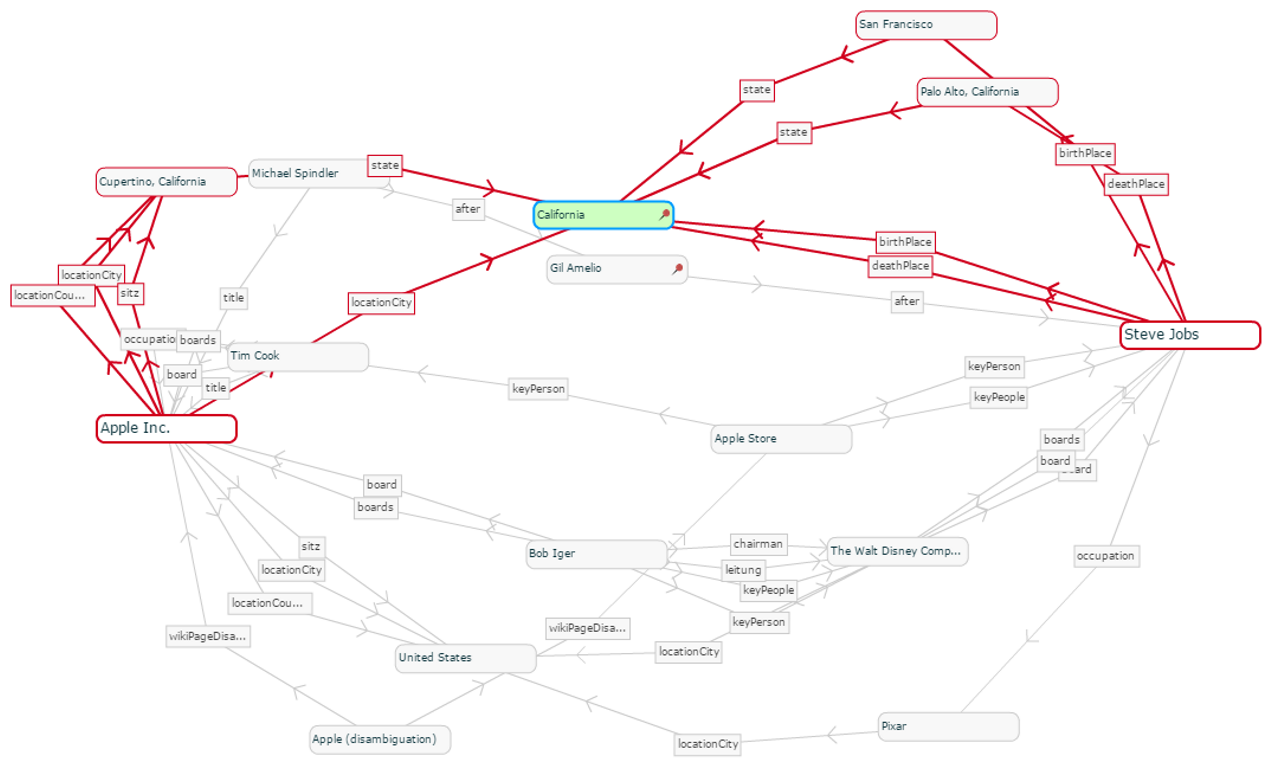


Figure Linear relation chains

‘Steve Jobs’ has four paths to ‘California’, while ‘Apple Inc.’ has four paths to ‘California’ too. You may wonder, which path pair is the most important one, or, which path can tell the deepest relation between them. From this aspect, we need to compute a score between different path pairs.

One approach is to compute the similarity. This is useful when two entities are similar in prior, like ‘平西王’ and ‘薄熙来’. The sub paths of each relation graph may share some similarity, for example, the relation between ‘平西王’ and ‘重庆’ and the relation between ‘薄熙来’ and ‘重庆’ may be similar. We can even make the constraints looser, such as allowing the end points of two paths are not the same. We can make graph bigger, adding phonetic and spelling relations to the entity.

Besides the similarity between paths, we can also compute the similarity between the sub graph. For graph A = (E1, (R11, E11), (R12, E12)) and graph B = (E2, (R21, E21), (R22, E22)), where E represents the entity node, R represents the relation node. If R11 is similar to R21, R12 is similar to R22, the relation between E11 and E12 is similar to the relation between E21 and E22, then we say these two graphs are similar. This kind of similarity definition can extend the expression ability of the relation graph.

The similarity can help explaining the originating relation between the morph and its target entity, besides the traditional linear logical chains. It can be easily adopted by human beings because we learn new knowledge from comparison. It two items show their similarity, then we can better understand why they are related.

But to define how two relations are similar is very difficult. First, concept has many ways of expression, like the word ‘胜利’, we have a lot of synonyms, like ‘取胜’, ‘打倒’,’摆平’,’获胜’,’成功’,’成就’. They have high similarities, but they are different in characters. Also, they are different in word senses, some are verbs, some are nouns. What’s more, even they are similar, we want to know to which extent are they similar. So we need to score the similarity, not just give a Boolean value to tell whether it is similar or not. We can compute the editing distance, also we can compute the similarity between the context they occur. Second, the noise of the data may interfere the accuracy of the similarity analysis. The data of the relation graph is extracted from web documents like Wikipedia. The content from it are all man-made, so some of them may not follow the format. Also the extraction framework also affects the quality of the information retrieved. Bad format of data may result in the failure of the detection of similarity or the failure of filtering out nonsense similarity. The previous one is called ‘False Negative’, the second one is called ‘False Positive’. For example, instead of ‘胜利’, it is ‘胜利了’, then the additional nonsense word ‘了’ may reduce the similarity between it and ‘打倒’. Another example is, ‘图像大小’. This may refer to the image width of the figure of some entity, but too many of entities have this relation. So at the end this relation has a very high score, but it is very common, whose score should be lowered down.

The similarity between entities should also be considered. But most of entities are more abstract than relations. For example, ‘薄熙来’ 🡪 ‘母校’ 🡪 ‘北京大学’, the relation ‘母校’ has a fairly clear meaning, which means the school that one has attended. But ‘薄熙来’ and ‘北京大学’ are abstract, many concepts are related to these two entities. You may say, this project is aimed to explain the similarity between the entities, to compute the similarity between entities should be considered as a sub task. This project aims to use the relation-entity graph to explain the similarity between two entities. The entities inside this relation-entity graph have different levels of abstraction. Some are very obvious, like ‘研究生’. For those not so complex, we may consider explain by its sub relation-entity graph. This means, this a recursive problem. This may result in the endless depth of the problem.

* + 1. Word Representation

Word vector may deal with this problem in a simple but efficient way. Word vector is a vector used to represent a word. The word representation has a closely 30-year history. From 1986 Hinton proposed the concept ‘distributed representation’ to currently Google created a highly-efficient word vector computation algorithm ‘word2vec’, the study to word representation has been continuing for a long time. At the first beginning, people use one-hot representation to represent a word. Within a given corpus, the scale of words is determined. We call the set of words as a dictionary. Inside a dictionary, we give each word an index. The word vector is thus created whose dimension is the size of the dictionary and order is the order of word index. Each word can be represented by a long sparse vector with all zeros but one number one at the position of its index. This kind of representation is useful to turn word into numbers, and use it to do some tasks. For example, to judge the theme of a document. Since each word can be represented as a sparse vector, a sentence can be represented as the sum of these vectors. It is similar to process a document. This is called BOW, bag of words. Due the sparseness of the vector, different document has different distribution of the vector elements. Then we can apply K-means to cluster the documents, thus find different themes. One more step, we can use this model to anticipate the theme of an unknown document. LDA is a more complicated algorithm to deal with this problem.

However, this kind of representation cannot tell the relation between words. Synonyms and antonyms cannot directly be got from this representation. That’s why Hinton proposed ‘distributed representation’. This word currently is called ‘word embedding’, or people can this model as vector space model. This model is famous in information retrieval from corpus whose documents are all in the form of a vector. This vector, which is the sum of word vectors, means the numerical importance of every single word within one document. Thus, two documents’ similarity can be calculated by some similarity measure, such as the cosine method. Word2vec is a fully connected neural network with one hidden layer. It can train a language model for each word as a vector. Similar word will have similar vector, which is computed in cosine distance. Google uses two methods to train word2vec. One is CBOW, continuous bag of words. Another is Skip-Gram Model. In CBOW, each word is surrounded by a context, which consists of some words. CBOW considers this context in the training of each word. Unlike traditional BOW, CBOW scroll a window through each words, instead of computing a single sum of vectors to a sentence or a corpus. This method helps focusing on the context of each word. Skip-Gram Model is similar to CBOW, but actually its strategy is the opposite of CBOW. Skip-gram model makes the target word as the input to the neural network, while the output is repeated to concatenate to be as a context, which should be close the original context of the target word. With the usage of CBOW and Skip-Gram, the neural network trains the words iteratively, and finally make similar words as similar as possible, opposed words as opposed.

Back to this problem. If we want to compute the similarity between entities, we can use word2vec as a solution. Abstract words are hard to be found with other similar words, but word2vec can help do this. Though abstract words are hard to describe its meaning, the context of similar abstract words may share a lot of commons. Thus word2vec will train the vectors for these similar abstract words as similar as possible. For example, within the corpus of Sina Weibo in 2012, the most similar word to ‘总理’ is ‘总书记’, whose cosine similarity is around 0.742168. This model even can detect that ‘平西王’ and ‘薄熙来’ is very close. From this point of view, we can say word2vec is powerful in detecting the similarity between abstract words, but it still cannot explain why they are similar. But it can help us to narrow down the searching area within the relation-entity graph, because the dissimilar entities may indicate that the sub graph of them also cannot reflect the relations between the morph and the target entity.

The usage of word2vec has another problem. If the comparison pair are not single words; say, they are phrases, then how should we compare? The phrase is composed by word. Though most words can find their word embedding, this is not often the same for phrases. To solve this, one straightforward idea is to compare all possible combinations of the words in phrases, then pick the highest score. But this may result ‘False Positive’. For example, ‘termStart’ and ‘termEnd’. Both of them have ‘term’, so it may consider them as a high score pair. But actually, ‘start’ and ‘end’ means totally different meanings. So this score should be given a discount. But this discount should not be too big; because though ‘start’ and ‘end’ are opposite words, their concepts all have relations to the concept of ‘time’. This kind of common part is very latent. From figure 3 we can see that opposite words ‘光明’ and ‘黑暗’ have a very high similarity even more than the synonym ‘得意’ and ‘骄傲’. Actually the word vector generated by word2vec is not suitable to describe the semantic relation. It can only describe the context similarity. People often mention ‘光明’ and ‘黑暗’ at the same time, or use them separately in the similar scenario. Thus why they share a high similarity. But if two words are totally irrelevant, like ‘宝贝’ and ‘革命’, then they a have strong negative similarity score, as the figure 3 shows. So we can use word2vec to compute the correlation between two words, not the similarity.

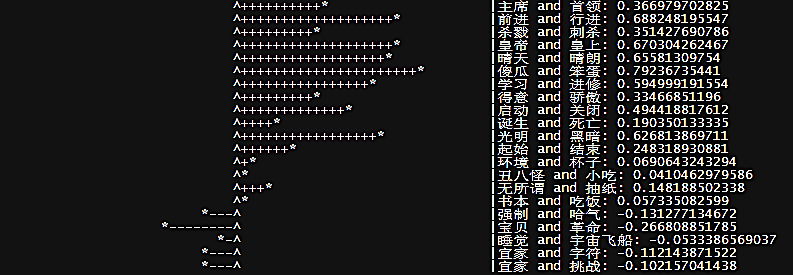


Figure word similarity

Besides word2vec, another choice is TransE. From this paper ‘Translating Embeddings for Modeling Multi-relational Data’ ([4], Bordes, Antoine, et al., 2013), TransE is a method to learn vector embeddings for both entities and relations. It significantly improves the performance of link prediction between two knowledge graphs. Though the idea of TransE, but is solves the comparison problem of relation. In the model of TransE, the difference between two word embeddings is the vector of a relation. So the comparison between relations can be more abstract and complex, since it doesn’t need to interpret each part of the relation. Also, it is useful to find latent relations, since relations are calculated by the difference of entities, not recoded from information extraction. But to apply TransE, we need a big and accurate graph. In English world, we have big knowledge graph with high quality like ‘Freebase’, but in Chinese language world things are not as simple as in the previous language world. Maybe we can translate the English version of ‘Freebase’ into Chinese, but the word sense disambiguation is not an easy task. So in this project, TransE is not the first choice. I would prefer to use word2vec to be as an experiment. If it works, then it’s find; if it doesn’t, I may try to focus on the information extraction from most Chinese corpus.

* + 1. Quality of the graph

However good the algorithm of similarity between graphs is, the quality of the graph determines the quality of the result. If the nodes inside the graph has a lot of noise, or it is to say that the graph has a low signal-noise-ratio, then the result will be mixed with irrelevant nodes. One kind of the noise is caused by the extraction framework. For example, the linked data dumped from Wikipedia by DBpedia. The Chinese version of this dumped data is mixed of a lot of noise, also it lacks of entity linking, so the connectivity of the graph is very low. For example, one tuple is (‘薄熙来’, ‘儿女’,’薄望知（1977年由李丹宁所生）\n薄瓜瓜（1997年由谷开来所生）’, the third element is very noisy, the better relation tuples should be (‘薄熙来’, ‘儿女’, ’薄望知’), (‘薄望知’,‘母亲’, ‘李丹宁’), (‘薄望知’, ‘生日’, ‘1977’), (‘薄熙来’, ‘儿女’, ’薄瓜瓜’), (‘薄瓜瓜’,‘母亲’, ‘谷开来’), (‘薄瓜瓜’, ‘生日’, ‘1997’). With this extension, the tuples become more clear now, and each third element of the tuple can be represented as an entity, otherwise they are just literals. We cannot compare the similarity between the literals, since they are too long to represent a clear concept. Besides the lack of entity linking, this extraction framework will include many English relations inside the graph. I think it is due to the template mechanism of Wikipedia. The content of Wikipedia follows some format, a structured text format. This template is written in English. Tough you may see Chinese relations in some info-boxes in Wikipedia, actually they are recorded in English in the database. Wikipedia uses a mapping table to map the English relation into Chinese relation. For example, (‘薄熙来’, ‘继任’, ‘张德江’), the relation node of this tuple is ‘继任’, but in Wikipedia database it is recorded as successor[x]. The x is used to distinguish different successor, but the DBpedia extraction framework simply ignores this x. As to this example, I can still use a dictionary to translate it. This is also a sub task of entity linking. But this word, ‘successor’, has no context. It doesn’t appear among a sentence. It just appears as a single word. Many meanings of this word exists, but I don’t know which one should I choose. That’s the problem of the data. It loses some important information. Moreover, relation like ‘termStart’ is more complicated. It is a combination of two words, which works as a marker as the start of a term. The first problem is, how to translate it into Chinese. The second problem is that the concept of it is complex. It works as a verb, meaning when the term starts. This consists of two concepts, one is the noun ‘term’, another is the verb ‘start’. So to compare this compound relation item to other relation item is hard; how can I compare a compound concept to a simple concept?

Even the graph is consisted of a set of extremely clear data, that all meanings of relation is simple and clear, that all non-relation nodes are true entities, the integrity of the graph still affects the quality of the similarity comparison algorithm. For example, the linked data extracted from Info-Box of Wikipedia in Chinese by DBpedia have 8,895,892 lines of tuples, and in total there are 784,840 entities in Chinese Wikipedia. So in average, each entity has 8,895,892/784,840 = 11.3 relation-entity/literal tuples. If we image a relation-entity/literal tuple as attributes of an entity, then 11.3 attributes are too small to describe an entity. Many attributes are very simple, such as the one describing where one was born, who was one’s father, things like these. More meaningful verb-noun pairs are hoped to be added in this graph. We call these verb-noun pairs as events. Extracting events from documents is not an easy work. Even with the help of NLP tools so that we can determine which one is subject, which one is predicate and which one is object, we still cannot extract subject-predicate-object, SPO, from a complex sentence easily. The order of SPO is maybe different from sentence to sentence, also the number of subject, predicate, object is maybe different too. It is hard to make sure which combination of SPO is correct, also the importance of SPO is not easy to be made as a ranking score. We may use dependency analysis a powerful weapon to conquer the challenges of event extraction. With this tool, the relation between many subject, predicate and object items within one sentence is clear. But the running efficiency of this tool is low. With Stanford Core NLP module, applying this algorithm to parse all Chinese Wikipedia documents may take one month to finish. So it is not applicable; it is useful though. Another simpler algorithm is to extract all possible SPO and count the occurrence, then the ones with low occurrence are eliminated. This is workable, but not accurate. It will bring noise to the data. But this can ensure the integrity of the data. So it’s a tradeoff.

Another possible way to extend the graph is to use alias. In Wikipedia articles, editors often use aliases to refer the corresponding entities. For example, ‘薄熙来’ is an entity, but it is also an alias referring to the entity ‘薄熙来事件’. This gives an import extensibility to the graph, because the relation between alias and its source may hard to use a word or phrase to describe, which is the reason why its source doesn’t appear in the Info-Box. But still there some problems with these aliases. First is, how to describe the relation between alias and its source. If it is marked as a universal mark ‘alias’, then it will lose a lot of complex information. For example, in Chinese Wikipedia, ‘北大’ is the alias of ‘北京大学’, but the relation between them is just ‘the same as’. For example, in Chinese Wikipedia, ‘中国’ is the alias of both ‘中华人民共和国’ and ‘中华民国’. The relation between same can be said as ‘the same as’, but should be under different historical background. For another example, in Chinese Wikipedia, ‘苹果’ is the alias of both '苹果 (电影)' and '苹果公司'. In this example, the relation is much more complicated. In daily life, ‘苹果’ refers to a common breed of fruit. But its meaning may differ in other context. So it is hard to use a simple ‘same as’ to be the relation between these kinds of alias and its sources. If we have to use this mark as the relation, then we have to do disambiguation – often we call it as entity linking – to eliminate irrelevant sources.

* 1. Conclusion

This project is a fresh project; Many challenges are waited to be conquered.

Chapter Two Approaches

The aim of this project is to explain the originating relation between the morph and its target entity. A morph is something like a metaphor or an alias to a concrete entity. For example, ‘平西王’ is the morph of ‘薄熙来’. People use this to refer to ‘薄熙来’ for some reasons. Maybe it is due to the censorship, or it is just to make fun. The later one is called as sense of humor. The task to detect whether an entity is a morph is called Morph Detection. Take this sentence as an example, ‘庭审的主角无疑是薄熙来,江湖上戏称“平西王”。’The entity ‘平西王’ here is a morph, refers to ‘薄熙来’. But for this sentence, ‘平西王,中国古代封建王朝的一种封号,“王”的地位一般仅次于皇帝。’, the entity ‘平西王’ is not a morph, it just refers to a title. Morph detection is a bit different from entity linking, because it doesn’t need to link actually link the entity, it just need to justify whether is a morph candidate or not. The task to find which entity is the morph that refers to is called Morph Resolution. This step is the further process of Morph Detection. Among all morph candidates, the correlation between them and the target entity is ranked from high to low. If the right morph appears as high as possible, then this model succeeds.

Besides these two tasks, I present a new morph task: explaining the originating relation between the morph and its target entity. Previous tasks can only tell you which are a morph and target entity pair, but they cannot tell you why they are related. So it is meaningful to use some kind of illustration methods to demonstrate how the morph is created for the target entity.

From paper ‘Resolving Entity Morphs in Censored Data’([1], Huang et al., 2013), Huang used an heterogeneous information network to track the relation between the morph and target entity. At the first beginning, I thought that I can use this network to explain the originating relation between them. Also as I said in the ‘Challenges’ section in chapter one, using linear logical chains inside the knowledge graph has already been studied. So I want to focus more on how to compare the similarity between two graphs. It is well known that people tend to learn new stuff from modeling something similar. From this point of view, it is meaningful to study the similarity between two graphs.

Many sub tasks have to be dealt with for this project. As the ‘Challenges’ section said, the prerequisite for solving this problem is the integrity of the graph, along with its high signal-noise-ratio and high connectivity. Besides it, a good comparison algorithm is also needed. Neither of each part is easy to be solved.

2.1 Fetch graph data

2.1.1 Extraction from plain text

At the very beginning of this project, is to fetch the data. As the ‘Challenges’ section said, it is not an easy problem. One straightforward to do this is to extract the relation tuple from raw text from Wikipedia and Sina Weibo in year 2012. The text in Wikipedia are more formal, and thus with less noise. Also the information in Wikipedia is tighter; it has a lower redundancy than Weibo corpus. But the shortage of it is its advantage; it is so formal that many interesting sentences are not included in it. Also it may lack the ability to include the hot spot in the cyber space.

Considering the timing feature of morph and fruitfully formal concepts that the target entity has, I chose to search the relation tuple in Sina Weibo 2012 for the morph and in Wikipedia for the target entity. For each morph, I plan to search every tweets in Weibo that consist it, and find all terms before and after it and the verbs between them among the same clause to form a (morph, verb, term) or (term, verb, morph) tuple. Here the word ‘term’ refers to a potential entity – a noun or a noun phrase or a verb-object phrase. Then for each term, do this search again, so as to find the tuple whose center is this term. Thus the graph can be expanded. Replicate this iteration, and finally stop before the depth of search reached. For each target entity, I plan to search in Wikipedia and retrieve the tuples similar in what I plan to do in Weibo. Additionally, the text in Wikipedia has a better structure. Many words in Wikipedia text has a hyperlink, some are aliases, some are outer links. This information can be used to expand the graph.

Before I begin to parse the corpus, I have to fetch them. For Weibo corpus, I can download them from a CUHK website. For Wikipedia corpus, at the first beginning, I built a web crawler to fetch the data. Though later I knew that the organization of Wikimedia has published the dumped file of Wikipedia, I found that the dumped one is a bit different from what it shows the wiki page on the website. The biggest difference is from the Info-Box. Also website can be used as a search engine, while the dumped text is hard to be used as a database; this problem has been solved by a SPARQL engine though. Back to the web crawler. I designed a web crawler which can do what a basic web crawler does: fetch one page, extract all links inside it, and finally add this link to the pool. But I have to distinguish the inner links and the outer links. I also designed a thread pool to accelerate the crawling process.

I used several regular expressions to locate the region of content, the links, root link, sub link, tag link:

regionRex = re.compile(r'(?s)<div id="content" class="mw-body" role="main">.\*?</div>(?=\s\*<div id="mw-navigation">)');

linkRex=re.compile( r'<a[^>]\*?href="(?P<href>[^"]\*)"(?:(?!/>)[^>])\*?>\s\*(?!<)(?P<entity\_name>.\*?)</a>');

rootLinkRex = re.compile(r'(?P<rootLink>.\*?://[^/]\*)');

subLinkRex = re.compile(r'^/(?!/).\*');

tagLinkRex = re.compile(r'^#.\*')

Also I designed some cascaded structures to link the wiki pages and entities:

class CascadePage(): def \_\_init\_\_(self, page, cascadeLink): self.page = page; self.cascadeLink = cascadeLink

class CascadeLink(): def \_\_init\_\_(self, link, cascadeEntityName): self.link = link; self.cascadeEntityName = deepcopy(cascadeEntityName);

class CascadeEntityName(): def \_\_init\_\_(self, names): self.names = list(names); def \_\_getitem\_\_(self, index): return self.names[index]; def \_\_len\_\_(self): return len(self.names)

Page and cascaded link are combined together as a cascaded page. Link and the cascaded entity names are combined as a cascaded Link. The list of names is considered as a cascaded entity name. The order of these names means the cascaded relation between them. Say, if A mentions B, C, D, then three CascadeEntityName are created for B, C, D as CascadeEntityName(B) = [CascadeEntityName(A), B], CascadeEntityName(C) = [CascadeEntityName(A), C] and CascadeEntityName (D) = [CascadeEntityName(A), D]. This kind of design can populate the cascaded relation deeper into the search. From this point of view, I used these cascaded structure to build the link extraction framework:

Define a function called as extract\_links with input of a cascaded page. Search the content region according to the regionRex, and build an iterator of a linkRex finder. Iterate all results found by this regular expression, extract the group ‘href’ as the link, parse this link to get the full path of this link, assign this full path link to the link, extract the group ‘entity\_name’ from the found results, cascade this entity\_name to the inputted cascaded page’s cascaded link. Following this way, the newly found link will be cascaded to its parent. This function will expand the graph by one level.

These codes will extract the links according to the regular expression of the link (To locate the links within a certain website, a regular expression is the best choice; It is better than to use a xml parser to parse the whole web documents and then retrieve all hyperlink text.), and record them in a cascaded manner.

The codes have a cache feature. The result of downloaded path is restored in a cascaded folder structure. The following is a sample of the list (74/401) of the names of all links extracted from the wiki page ‘薄熙来’ (each one’s web page is stored in the disk):

三峡大坝，上山下乡运动，世界史，东北，东北地区，中共中央，中共中央书记处，双开，受贿，受贿罪，司法机关，吕福源，吴仪，吴文康，周永康，国务院副总理，尼尔·伍德，尼尔·海伍德，尼爾·海伍德，山东省，山东省济南市中级人民法院，山东省高级人民法院，山西省，市委书记，市政府，市長，布鲁金斯学会，广州，开除党籍、公职，张国光，张德江，张文岳，张晓军，徐明，徐鸣，德维尔，李永金，李铁映，李雪峰，欧元，死亡案，母校，每日电讯报，毛泽东思想，江泽民，汪洋，沈阳市，海伍德死亡案，温家宝，温家宝内阁，滥用职权，滥用职权罪，王奉友，王旭光，王正刚，王立军，王立军事件，王立軍事件，王鸿举，薄一波，薄小莹，薄昌福，薄望知，薄洁莹，薄熙來，薄熙宁，薄熙成，薄熙来事件，薄熙来受贿、贪污、滥用职权一案开庭审理，薄熙来案一审判决书，薄熙来案二审裁定书，薄熙来涉嫌受贿、贪污、滥用职权案提起公诉，薄熙来简历，薄熙来获开除党籍、开除公职处分

From this list we can see a lot of related entities. But these codes haven’t extracted the relations. It is only used to expand the graph. In my design, the expansion of the graph is the first step: fetch the data. The next step is to parse the relation-entity tuples from it. But before I further move on this, I realized that there are already existing some out-of-box data sets that describe the relation-entity network. What’s more, the extraction of relation-entity tuple itself can be a big project. So it makes sense to make this part easier by using something that already existed. But just like what I mentioned in ‘Challenges’ section, the ready-made one may lose the integrity and connectivity of the graph.

2.1.2 Use extracted data set

Beyond the ‘Entity Cube’, ‘YAGO’ and ‘DBpedia’ that I mentioned before, a group from Fudan University built a CN-DBpedia, whose definition is the chinses version of DBpedia. Table 2.1 is a sample of the searching result of ‘薄熙来’ from CN-DBpedia. The introduction of CN-DBpedia says it is a large scale, universal field applicable, structured encyclopedia. Its data is extracted from Chinese cyclopedia sites, such as Hudong Baike, Baidu Baike and Chinese Wikipedia. It has commercial cooperation with many companies to facilitate different fields of intelligent applications. It has 16,532,759 entities, 213,477,603 of relation-entity tuples. So the average tuples for one entity is 213,477,603/16,532,759 = 12.9. This is even higher than the Chinese data set extracted by DBpedia, which is 11.3. What’s more, the quality of CN-DBpedia is better. Table 2.2 is also a sample of the searching result of ‘薄熙来’ from the Chinese data set of DBpedia. It has 55 tuples, far more than the one from CN-DBpedia. But it has more noise as the same time, like (‘图像大小’, ‘200’), (‘姓名’, ‘薄熙来’). This means, the actual high quality tuples from DBpedia is much less than ones from CN-DBpedia. But the advantage of DBpedia is, from this example, it can extract more information for entities which are under the censorship in China. Since a certain amount of morphs are created due to the censorship, the advantage of DBpedia makes me to use it instead of CN-DBpedia.

Table 2.1 relation-entity tuples of ‘薄熙来’ from CN-DBpedia

|  |  |
| --- | --- |
| Relation | Entity |
| 中文名 | 薄熙来 |
| 主要成就 | 曾任中央政治局委员 |
| 主要成就 | 重庆市委书记 |
| 入党时间 | 1980年10月 |
| 出生地 | 山西定襄 |
| 出生日期 | 1949年07月 |
| 参加工作 | 1968年1月 |
| 国籍 | 中国 |
| 毕业院校 | 中国社会科学院 |
| 民族 | 汉族 |

Table 2.2 relation-entity tuples of ‘薄熙来’ from DBpedia Chinese data

|  |  |
| --- | --- |
| Relation | Entity |
| termStart | 2001 |
| termStart | 2004 |
| termStart | 2007 |
| termStart | 1992 |
| 母校 | 北京市第四中学 |
| 母校 | 中国社会科学院研究生院 |
| 母校 | 北京大学 |
| 图像大小 | 200 |
| office | 遼寧省人民政府省长 |
| office | 大連市人民政府第2任市長 |
| office | 中华人民共和国商务部部长 |
| office | 重庆市委员会书记 |
| office | 中央政治局委员 |
| 专业 | 世界歷史 |
| 专业 | 新闻学 |
| 出生日期 | 1949-07-03 |
| 政黨 | (（1980-2012）) |
| secretary | 曹伯纯于学祥 |
| secretary | 闻世震 |
| 性別 | 男 |
| 父母 | 母亲：胡明 |
| 父母 | 父亲：薄一波 |
| 图像 | VOA-Bo Xilai.jpg |
| 居處 | 秦城监狱(（2012年至今）) |
| 親屬 | 薄熙永(（兄長）) |
| term | 中国共产党第十七届中央委员会 |
| deputy | 王鸿举黄奇帆（市长） |
| 子女 | 薄瓜瓜(（1987年由谷开来所生）) |
| 子女 | 薄望知(（1977年由李丹宇所生）) |
| 出生地點 | （解放区） |
| 籍貫 | 山西省定襄县 |
| premier | 温家宝 |
| termEnd | 2000 |
| termEnd | 2012 |
| termEnd | 2004 |
| termEnd | 2007 |
| 姓名 | 薄熙来 |
| 網站 | http://news.xinhuanet.com/ziliao/2002-02/21/content\_285068.htm |
| generalSecretary | 胡锦涛 |
| 配偶 | 谷开来(（1987年至今）) |
| 配偶 | 李丹宇(（1976年至1984年，离婚）) |
| successor | 张文岳 |
| successor | 陈德铭 |
| successor | 李永金 |
| successor | 张德江 |
| 宗教信仰 | 无 |
| predecessor | 吕福源 |
| predecessor | 张国光 |
| predecessor | 汪洋 |
| predecessor | 魏富海 |
| job | 政治人物 |
| 學歷 | 研究生 |
| 職業 | 政治人物 |
| title | 曾任职务 |
| period | 21 |

The dataset from DBpedia is of the form of Turtle. Turtle is a file format, meaning terse RDF Triple Language. RDF refers to the Resource Description Framework (Resource Description Framework). It is a framework for describing resources on the Web. It provides models and syntax for data so that independent groups can exchange and use it. It is designed to be readable and understood by the computer. It is designed not to show people. It is written in XML. It is an integral part of W3C semantic web activity. It is a W3C recommendation. It can describe the properties of a shopping item, such as price and availability, the timetable for Web events, information about the page, such as the content, the author, and the date that was created and modified, the content and level of the web image, the content for the search engine, and electronic library. Figure 4 is an example of RDF. This format of data is very redundant, though it has a good cascaded structure. If it is used to record all the relation-entity items, it will make the file very large.

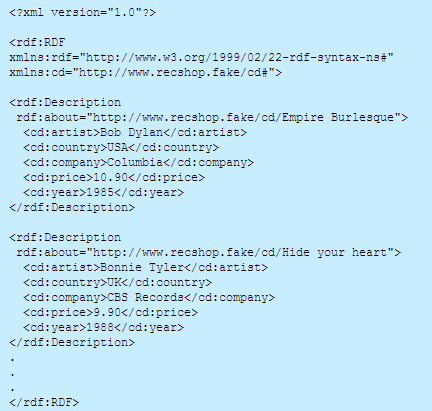


Figure a sample of RDF

Turtle is a terse version of RDF. It uses tuple instead of cascaded structure to describe the relation between entities. Here is an example of one record of Turtle file from the Chinese data set of DBpedia :

<http://zh.dbpedia.org/resource/薄熙来> <http://zh.dbpedia.org/property/籍貫> "山西省定襄县"@zh .

There all 4 terms in this record, one is ‘<http://zh.dbpedia.org/resource/薄熙来>', one is ‘<http://zh.dbpedia.org/property/籍貫>' , one is ‘"山西省定襄县"@zh’, the last one is ‘.’. The last one is used as a record separator. From this example we can find some basic elements in this record. The first is the IRI description. The first term in written in the form of IRI. IRI is the unique ID of an entity. The last part of IRI is the name of the entity, the part before it is the name scope and transport protocol. For example, as to ‘<http://zh.dbpedia.org/resource/薄熙来>', the name of the entity is ‘薄熙来’, the name scope is ‘zh.dbpedia.org/resource’, the transport protocol is ‘http’. The second term is the relation. It follows the similar form of the first term. The name scope of it is ‘zh.dbpedia.org/property’, the name of it is ‘籍貫’. The third term, ‘"山西省定襄县"@zh’, is a literal. Literal is not an entity; it is treated as a plain text. Here it is treated as a string, in Chinese language. Other types of literal includes Boolean, Numbers, etc.

This form is much terser, though it has to repeat the name scope yet. It doesn’t need to repeat the end marker for each row. Also it loosens the limitation of data; You only need to append a tuple, not a whole cascaded structure of an entity. Thus it has a high extensibility. New relation-entity tuples can be added anywhere, in any temporal sequence. So the searching engine of Turtle is also flexible.

But as you can see from this example, the third part of the tuple is not an entity. It is a literal. Obviously it can be linked to an entity, but the extraction framework of DBpedia can’t resolve it with Chinese data set. So I decide to fork the repository of DBpedia extraction framework, and tried to rewrite the extraction module of Chinese dataset. But it is too complicated. The following figure 5 is the class structure of DBpedia extraction framework. What I want to focus is the ‘Infobox Extractor’. DBpedia extraction framework can extract a lot of different categories of data from Wikipedia. Here is a sample list of the category: ‘External Links’, ‘Infobox Properties Mapped’, ‘Infobox Property Definitions’, ‘Long Abstracts’, ‘Out Degree’, ‘Page Length’, ‘Page Links’, ‘Redirects’, ‘Transitive Redirects’, ‘Short Abstracts’, ‘Wikipedia Links’, ‘Wikipedia XML source dump file’. The ‘Infobox Extractor’ can extract ‘Infobox Properties Mapped’, which is something like the record I’ve given above. Figure 6 shows the source code files under the ‘mappings’ folder. ‘Infobox Extractor.scala’ is among these files. It has around 300 lines. To learn how this file works, I paid one week to learn how the language rules and features of Scala. This language is so powerful and flexible that it can embed functional programming and object-oriented programming together. The sentences written in Scala is very terse, so 300 lines of Scala have a lot of information. I paid extra days to read the codes, but still found it was too hard to understand.

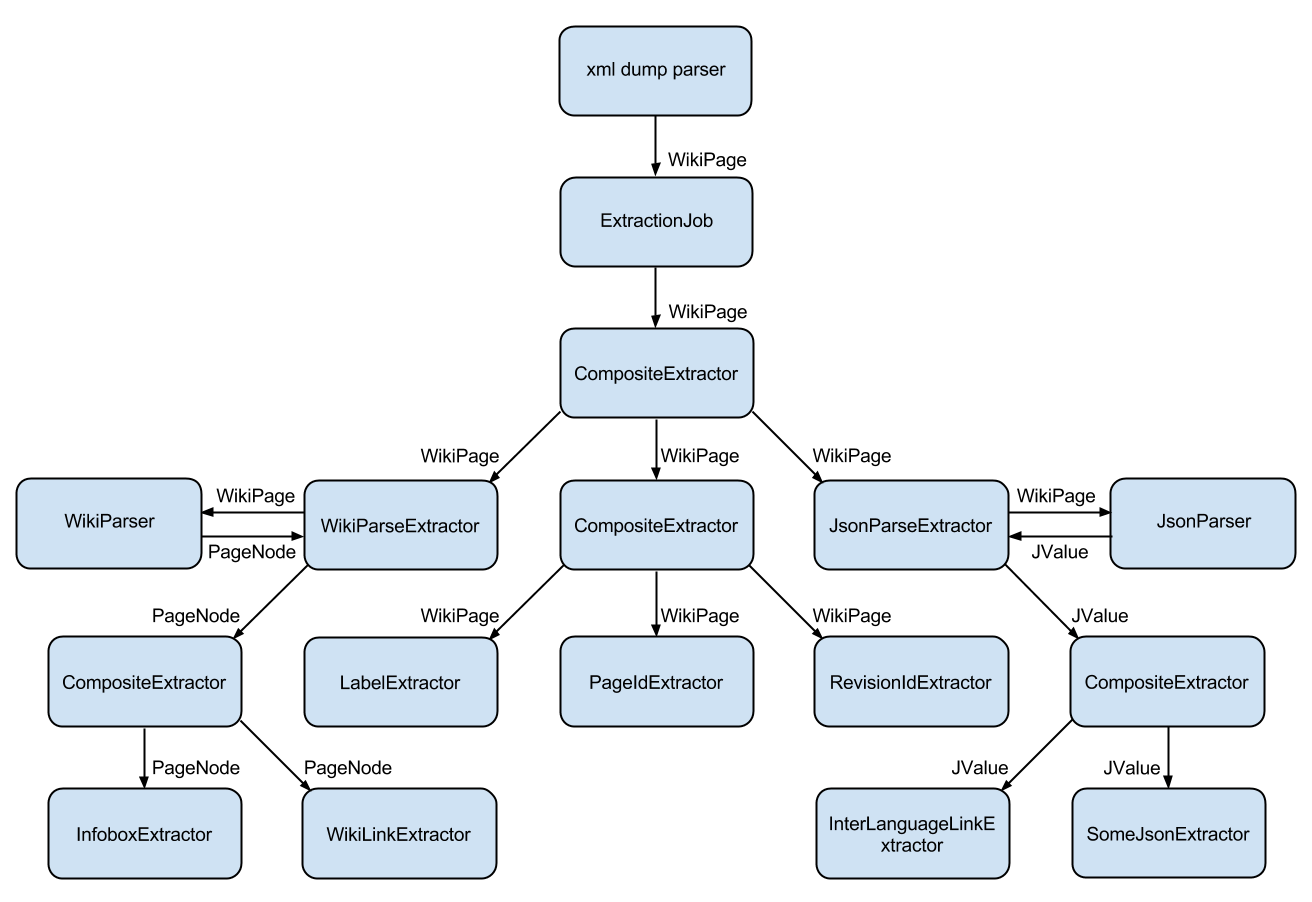


Figure The class structure of DBpedia extraction framework

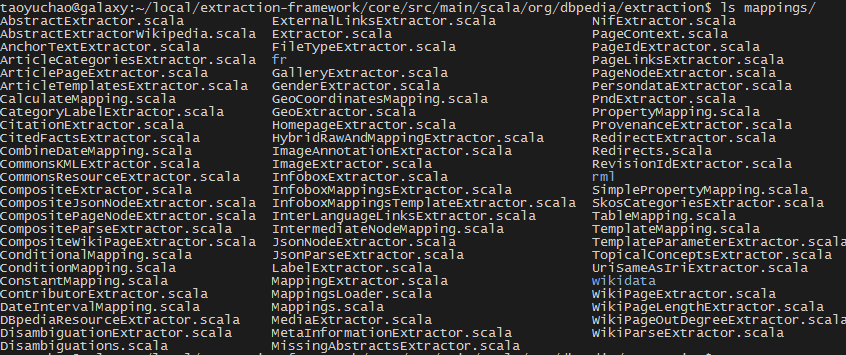


Figure Files of mappings

To run the DBpedia extraction framework, I have to download the source dump file of Wikipedia. Following is the list I gathered about the file links about dump files:

Table 2.3 File links about Wikipedia dump files

|  |  |
| --- | --- |
| File | Link |
| Data Dump | <https://meta.wikimedia.org/wiki/Data_dumps> |
| Dump Index | <https://dumps.wikimedia.org/zhwiki/latest> |
| Embedded Templates in wiki | <https://www.mediawiki.org/wiki/Help:Templates> |
| Wikipedia Extractor | <http://medialab.di.unipi.it/wiki/Wikipedia_Extractor> |

2.2 Establish a query engine

If I use plan 1 to fetch the data, which is to crawl the web page from Wikipedia with self-designed cascaded structure and cache engine, and parse the text to get the relation-entity tuples, then it has already built a query engine for me. I can search for any entity through http protocol; Each entity has its URI, so I can easily visit it through web server. I can also write this into my python script. Also, I can use my cascaded cache structure to visit the pages I’ve downloaded (But I didn’t deal with the loop problem). But finally I use plan 2 to fetch the data. How can I make query among a Turtle file? The Turtle file is around 5.7G, If I build a dictionary for this Turtle file then it would occupy around 1G memory (This is just a number I guessed’). Also a dictionary is not powerful to do the query. At least the speed of simple query may be not fast. This will affect a lot if later a lot of queries. So I decided to use some professional Turtle query engine. SPARQL comes.

RDF is a targeted graphical data format used to represent information in the Web. This specification defines the syntax and semantics of RDAR's SPARQL query language. You can use SPARQL to represent queries between different data sources, whether the data is stored as RDF or through RDF as a middleware. SPARQL contains the query and optional graphics modes and their connection and separation functions. SPARQL also supports aggregation, subquery, deny, create values through expressions, extensible value tests, and query by source RDF graphs. The result of the SPARQL query can be a result set or an RDF graph.

Following is some examples to show the grammar rules of SPARQL:

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.

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"

The result of a query is a solution sequence, corresponding to the ways in which the query's graph pattern matches the data. There may be zero, one or multiple solutions to a query.

Data: @prefix foaf: <http://xmlns.com/foaf/0.1/> .\_:a foaf:name "Johnny Lee Outlaw" . \_:a foaf:mbox <mailto:jlow@example.com> . \_:b foaf:name "Peter Goodguy" . \_:b foaf:mbox <mailto:peter@example.org> . \_:c foaf:mbox <mailto:carol@example.org> .

Query: PREFIX foaf: <http://xmlns.com/foaf/0.1/> SELECT ?name ?mbox

WHERE { ?x foaf:name ?name . ?x foaf:mbox ?mbox }

Query Result: name , mbox, "Johnny Lee Outlaw", <mailto:jlow@example.com> , "Peter Goodguy" <mailto:peter@example.org>

From these examples we can find that SPARQL’s query sentence is similar to SQL, but it focuses more on the formulation of the path structure of the query object. The WHERE clause describes the meta format of a tuple. It is very useful in searching for the relation-entity pairs related to one entity.

SPARQL is just a query language, so an instance of search engine is still needed. After searching around the Internet, I found an open source project called Apache Jena. Apache Jena is a free and open source Java framework for building Semantic Web and Linked Data applications. It provides RDF API and ARQ (an instance of SPARQL). It provides Triple Store – TDB and Fuseki (this sub project is a SPARQL endpoint). It provides OWL – Ontology API and Inference API. ARQ and Fuseki are both the instance of SPARQL, but the previous one is an embedded SPARQL engine, which is used in Apache Jena. Fuseki is a sub project, called as Apache-Jena-Fuseki. It is much more independent than ARQ. It can be started as server, so you can submit your SPARQL query through URL. I prefer the later one, since it is much more flexible. It can load Turtle file too. For 103M Turtle file ‘Redirects.ttl’, it took around 1 minutes to load. For 1G Turtle file ‘infobox.ttl’, it took around 10 minutes to load. These time are taken under the condition that the memory limit is 1G.

Apache-Jena-Fuseki also provides some entry point to access the server of SPARQL. One of them is called ‘s-query’. Here is an example of it:

s-query --service=http://202.120.38.146:9600/data/sparql 'select ?y ?z where {<http://zh.dbpedia.org/resource/薄熙来> ?y ?z} LIMIT 100'

The service parameter is the address where the Apachae-Jena-Fuseki listens. Here I opened a 9600 port for this server. The string at the last part of this command is the query sentence. The sentence used here means ‘I want to find all relation-entity tuples related to ‘薄熙来’. Please list no more than 100 results.’.

The following is a snapshot of the result of this query:

{"head": {"vars": [ "y" , "z" ]} ,"results": {"bindings": [{"y": { "type": "uri" , "value": "http://zh.dbpedia.org/property/termStart" } ,"z": { "type": "literal" , "datatype": "http://www.w3.org/2001/XMLSchema#integer" , "value": "2001" }} ,{"y": { "type": "uri" , "value": "http://zh.dbpedia.org/property/termStart" } ,"z": { "type": "literal" , "datatype": "http://www.w3.org/2001/XMLSchema#integer" , "value": "2004" }} ,

We can see that fuseki server returns a json object. Later we need to parse this json objects to get the tuples.

SPARQL can also provide very complicated query input. Here is an example:

Query = SELECT \* WHERE { <http://zh.dbpedia.org/resource/张德江> ?pf1 ?of1 . ?of1 ?pf2 ?of2. ?of2 ?pf3 <http://zh.dbpedia.org/resource/薄熙来> FILTER ((!isLiteral(?of1)) && (?of1 != <http://zh.dbpedia.org/resource/薄熙来> ) && (?of1 != <http://zh.dbpedia.org/resource/张德江> ) && (?of1 != ?of2 ) && (!isLiteral(?of2)) && (?of2 != <http://zh.dbpedia.org/resource/薄熙来> ) && (?of2 != <http://zh.dbpedia.org/resource/张德江> ) && (?of2 != ?of1 ) ). } LIMIT 10

This query is used to find all middle relations between ‘薄熙来’ and ‘张德江’. This is what the DBpedia’s project ‘RelFinder’ did during searching. This project I will introduced later in the visualization part.

2.3 Build the Graph

After we get the approaches to fetch and search the relation-entity network data, the next step is to design how to make them as a graph, and along with several iterating methods.

First is to build an interface with SPARQL. I defined a rawQuery(resource) function, which returns a (status,data), to open a subprocess within the python script to visit the s-query command. This will return the json object about the relation-entity tuples. The query is designed as the one I mentioned above. I defined a function query(name), which returns (status, empty, data). The state is the http status after visiting the SPARQL server. The empty is a Boolean value, indicating whether the data is empty or not. The data is processed into a list of relation entity tuples (in python object format). I defined a entitiesOf(literal) function to deal with the entity linking. This function serves to turn the literal into entity, which can be searched by SPARQL. The key point is, this is can as simple as a word segmentation, or as complicated as accurate entity linking. At the first beginning, I use punctuations and word segmentation to split the literal into a list of words. These words are used to try whether each one is an entity or not. The way or try is to use query(name) function to judge whether the empty (return parameter) is true or false. If it is false, then the term split from literal is regarded as an entity. I defined a fullQuery(name) function, which is similar to query(name), but the feature of entitesOf(literal) function is used in this function. It is used to help expand the literal point. If the end point of a graph is a literal, then it can be expanded by this function.

Next is to design the structure of the graph. Inspired by the philosophy of Scala, I prefer to use recursive lambda function to build the graph. Scala is powerful to express the recursive lambda function. Here is an example how it expresses a quick sort function:

def quicksort (xs : List[Int] ): List[Int] = { if (xs.isEmpty) xs else quickSort(xs.filter(x => x<xs.head ) ):::xs.head::quickSort(xs.filter(x=>x>xs.head))}

It very terse, isn’t it? Also in python we can achieve the same level terse expression as in Scalar. Here is an example:

flat = lambda L: sum(list(map(flat,L)),[]) if isinstance(L,list) else [L]

This line of code defines a function flat, which can make a cascaded list into a flat list. This function is common in PYSPARK, but in native python APIs there is no place for flat function. This code is very beautiful. Function are treated as an object, that’s why this code can be so terse.

Back to the design of the graph. I chose to use python’s object: tuple and list to describe the graph. The following line of code is the expansion function which can expand the graph bigger with one hop:

expand = lambda t: (t[0] ,list(map(expand, t[1]))) if isinstance(t[1], list) else (t[0], expand(t[1])) if isinstance(t[1], tuple) else (t[0], (t[1], self.\_fullQuery(t[1])))

It is a bit complex, but it only needs one line. Looks so cool. This line of code takes a tuple as an input, and return a new tuple. It will copy the original tuple, and only expand the last leaf. You can think this code in a deep-first search way. For each node, it’s children will be mapped into this expansion function again. It will keep recur until the children is not a string object.

The simplest graph is looked as this: (X1, [(R11, Y11), (R12, Y12) … (R1N, Y1N)]. Each X,R,Y here is a string object. The non-leaf nodes are absolutely entities and relations, but leaf nodes are not necessarily to be the entity. Each Y here can be expanded in the form same as this. So it is embedded one layer to another layer.

Also I want to design some facilitating functions to help the further graph comparison. The basic one is the iterator of the graph. There are many ways to iterate the graph. In layer-wise way, in deep-first way, in wide-first way. Also, the problem here is a bit special. To facilitate the graph comparison, I want to record which sub graphs are similar. Sub graphs can be decomposed into a set of chains. So to record the sub graph, I can record the chain instead. There are also many ways to record the chains. One obvious and easy method is to iterate all possible chains of each graph, and then enumerate all possible combination of chains from two graphs. To iterate all possible chains, one possible method is to deep-firstly iterate all the nodes. Each node can represent a chain from the root to it. Thus all possible chains of the graph can be enumerated. But the problem of this enumeration is that, it will waste the computation resource. Different chains may have a lot of common parts. If the similarity between the same common parts of the chain pair haven’t been recorded, then many of them will be repeatedly computed. Thus there should be a cache dictionary to record which chain pair has been recorded. Since all chains are starting from the root, so only the end node is needed to be recorded into this dictionary. But if we want to extend this comparison mechanism into a more relaxing condition, then we need to record the start node and the end node together as the representative of the chain. But to record two nodes is a waste too, since we can use the end node and its height to record the chain. This way of recording is better than the previous one, because it can not only provide the whole details of the chain, but also save the time to compute how long this chain is.

If the condition is limited to the layer-wise comparison, I think I can also use a cluster to cluster all similar elements in one layer. To better control the graph, I also need to use a new class to represent a node, instead of list and tuple. List and tuple can make the expression of a graph as tight as possible, but it is not easy to visit its sub graph. One has to walk through the graph to fetch the one it wants. Also this expression cannot support loop. Each node is represented by a string, which also lacks the ability to contain more information. To improve the extensibility, I design a new class called Element to substitute the original graph. I wrote a function called ‘tuple2Graph(tuple\_map)’ to map the original list-tuple format of graph into an element-linked format of graph. This format is very similar to the design of linked list in C++. Each element has parent, children. Also I rewrote the \_\_str\_\_ function of the class, so the print() function can print it in a beautiful and information-fruitful way instead of just a class information. The function ‘tuple2Graph(tuple\_map)’ returns a root-element. From this root, I can visit all the nodes inside the graph. Here I make the relation also as an element. Otherwise, the list of the children is not a simple list of elements, it would be a list of tuple where each tuple consists one relation and one element object. To make the design as simple as possible, I decided to treat the relation also as an element. That’s why I use the word ‘element’ instead of ‘entity’. To distinguish the difference between the ‘entity’ and ‘relation’, I add one more attribute to the class Element, which is called as the ElementType. This is a enumeration type, defined as follows:

ElementType = enum(entity='entity', relation='relation')

The enum function is not a built-in function. It is:

def enum(\*\*enums): return type('Enum', (), enums)

It’s a trick of python grammar. The type function takes three arguments, the name of the class, the parent class of the class, and the dictionary of its attributes. Normally this function is used to get the class of a given instance, but when it is given three parameters, actually it can generate a new class. So the code here is to generate a new class, while the attributes of this class can be determined by the input of this function, which is the \*\*enums. When we use this function, we need to pass a dictionary like parameters into the parameter list of this function, and then this function will return a new class whose attributes are the same as what we defined in the dictionary like parameters. So it’s a totally trick of python grammar. Python itself doesn’t have built-in enumerator. So we use this kinds of trick to make a handy tool of enumerator.

After the Element class is defined, the next is to define the cluster. In my design, all similar elements in one layer should be gathered into a cluster. Here one layer means the set of all elements that have the same heights to the root. After all elements are categorized into clusters, then each layer can be viewed as the set of clusters. Then the comparison between the same layer can be viewed as the comparison of the set of clusters, instead of the comparison of the elements. This can save a lot time, from O(Log(N)\*N^2) to O(Log(N)\*N). This is due the reason below. The comparison between the cluster can be element-wise, or be cluster-wise. If it is element-wise, then all the elements in each cluster will be enumerated to be compared. Before the comparison of the cluster, a check of the similarity of the cluster will be done. So the totally-irrelevant clusters will be filtered out from the layer-wise comparison. That’s why the time complexity of the algorithm can be reduced to O(Log(N)\*N). This condition guarantees that the clusters that are being compared are quite similar. So as to the elements to be compared with these two similar clusters, they are no longer needed to be compared for the features that their clusters have already shared; they only need to be compared something that differ a little. Otherwise the comparison will take much more time if it also wants to perform as good as it.

In my first design, the similarity is a Boolean value. There are only two words to describe whether a chain of a graph is similar to another: ‘similar’ and ‘dissimilar’. So after the comparison of one layer, only the element pair which are similar are allowed to make their children compared. But since I perform the comparison on the cluster level, I have to turn these children into clusters too. So I have to use a list to record all the children of the element that has been marked as the similar one to another element. The problem is, it is hard to do the true layer-wise comparison. I cannot simply to jump to the next layer, because the condition that all similar chains have to start from the same root. So I have to filter out the ones that cannot follow this condition. But since the comparison is between two elements, I cannot flatten all legal children into one layer. So I have to do multiple layer comparison in next layer; Each element pair at the parent layer will result in a new layer comparison.

Another solution is simpler. It is to first find all similar element pairs between two graphs, only if they are at the same layer. Then for each element that recorded in all similar element pairs, check whether their parents can form a similar element pair. If yes, then take this new similar element pair as the base, check whether their parent element pair can form a similar element pair. Do it until they meet the root. During the process, the element pair that have been removed from the checklist. So the similar parent element pair will no longer needed to be checked several times.

Following this simple solution, we can even find more chains from different roots. During the finding of similar element pair, we can relax the condition that they have to be at the same layer. They can be anywhere. Then we check all possible chains they can form by checking whether their parents are similar. At last, we can rank these chains by the height. If the height is longer, then it should be ranked higher. Here the height is regarded as the absolute depth of the chain. Since the roots of the chain pair can be different, the starting points of each chain may not be at the same layer. The height is computed from the graph root to the farthest node in the chain, The height used to rank is taken from the longest height from the chain pair. But due to the noise of the data, non-sense common elements ranks very high. So the data quality affects the result very much.

2.4 Compare the graph

2.4.1 Linear regression

As I mentioned in ‘2.3 Build the graph’, the comparison between the graph can be reduced to the comparison between the chains. Thus the comparison can be reduced to a single pair of elements. But later I realized that some similar chains may have different lengths. Thus I want to find a method to compare the chain with different lengths.

The element-wise comparison cannot solve this problem. Even with a little change to this algorithm, for example, allowing some gaps, it still cannot solve this problem. Allowing gaps is a too much strong relaxation of the condition. Allowing gaps is to say, one or two elements of a chain can be ignored during the comparing. Of course the correspondence of element type has to be followed; that is to say, the entity must correspond to entity, relation must correspond to relation. This may result in the bad side effects that the chain share the common elements will gain a very high score. For example, X = A🡪B🡪C🡪D, Y = A🡪M🡪B🡪C🡪D, then these two chains can have a very high similarity even Y is different from X due to the existence of M. B is maybe a very common attribute, thus they have so many parts in common. This chain pair is even ranked higher than the chain p air like A🡪M🡪N and A🡪M🡪, where N and is the similar element, because the pair of X and Y is longer than this. So I was thinking about, how to design a length-irrelevant chain comparison algorithm.

During that time, I was touching with the Manifold Learning. Since each element can be represented as a vector, the chain of elements can be represented as a list of line segments. According to the Manifold Learning, I can reduce the dimension of the list of line segments and find a curve to approximate these ordered vectors. If this is applicable, then the comparison between two chains with different lengths is possible. Moreover, this method makes the comparison more flexible; the comparison is no longer element-wise, it’s chain-wise. The deviation of one element pair can be compensated by other element pairs in a chain. This is a way of relaxing the condition. For example, let chain A = 吴三桂🡪生于🡪清朝🡪首都🡪北京, let chain B = 薄熙来🡪母校🡪北京大学🡪城市🡪北京。Man can tell that this chain pair is very similar, since ‘生于’ and ‘母校’ both have the meaning of ‘from’, ‘清朝’ and ‘北京大学’ are all the concept that can contains people, ‘首都’ and ‘城市’ are both that indicate the place, ‘北京’ and ‘北京’ are the same entity. Though ‘吴三桂’ and ‘薄熙来’ cannot be regarded as the similar element pair, the chain after them has a strong similarity. If both two chains are modeled as a curve, then the Euclidian of these two curves must be very small, even when some of them deviate.

This method can also be used to train the vector of the elements. If the similar chains of two graphs are already known, then we can train the vectors of each elements by letting the reduced curve of the chains as similar as possible. This is a supervised learning, so this will require a lot of human effort to do. So up to now, this is just a proposal.

The key point of this method is how the curve is formed. The regression of a sparse set of data in a high dimension is very inaccurate. Suppose there are only five points in the list of segments while the dimension is 30, then the reduced curve is almost as linear as a straight line. Think about it. Suppose here are five points in 2 dimensional space, which are (0,0), (1,1), (0,2), (1,3), (0,4), (1,5). Following the order of these nodes, a list of segments is made. Then what is the reduced curve for it? I think it is a vertical line at x=0.5. But what I really want is a ‘S’ shape curve that starting from (0,0) to (1,5), following the order of the list of segments. So how can I solve it?

One possible method is to use interpolation method. For each segments, if we insert sufficient interpolation points, then the liner regression can better fit the trend of the segment order. You may wonder, how to use a function to express a ‘S’ shape curve? In 2-dimension space, we can only write a y=f(x) function to represent a curve. For each x value, there is only on y value. Thus the ‘S’ shape can never be achieved by this formula. But we can turn this function into a more general formula, which is g(x,y) = 0. For example, when g(x,y) = x^2 + y^2 – r^2, then this formula can represent a circle in a two-dimension space. This formula can view as the mapping from a three-dimension space to a two-dimension space. Let z=g(x,y), these this formula turns to be a function z=g(x,y) where z = 0. Another interpretation is to view the g(x,y) as X\*W\*YT. Here X and Y are the higher space vector of x, like <x, x^2, x^3 …>, <y, y^2, y^3>. W here is a weight matrix. For example, as to g(x,y) = x^2 + y^2 –r^2, we can write the decomposition in this way:

(1)

The weight matrix is a bit complex to compute, so I just leave the symbols in the matrix. This interpretation indicates that this is the dimension expansion of x and y. With higher dimension vector, they can express more. What’s more, we can also use a kernel function to substitute this matrix computation, since matrix decomposition is not always easy to perform. Besides this linear expression, we can use some non-linear expression. To perform this, a three-layer neural network is a good choice. It has three layers, one for input, one for hidden layer, one for output. With a tanh and reLu output of the hidden layer, the expressing power of the neural network can be very powerful: from the proof of mathematicians, it can express all possible curves.

From this point of view, I’d prefer to use a neural network to do this regression task. But the problem, how to compare the curves generated by the neural network. Actually, neural network is a predictive model, it can only tell whether the input is on the target curve or not. If I want to retrieve the curve, I have to try a lot of points to judge whether they on the curve or not (A threshold value should be set to judge, such as < 0.1). Then I will get two sets of points from two curves. Then I can compute the average nearest neighbor length for each points from one curve to another curve. This would take a lot time. It’s a time-consuming job. There are so many chains for me to compare, if I have to do neural network training, curve retrieving and curve comparing, then the total comparison time would be very large. To make the task simple, I can use the weight matrix of the neural network to compare the similarity between the curve. If two curve are similar, then their parameters of the curve must be similar. I can compute the L1 norm or L2 norm of the difference of the matrix. But I’m not sure whether it is appropriate or not. For example,

(2)

(3)

(4)

(6)

As you see from these four functions, they share the same difference of weight matrix, but their actual distance of curve is far from the same. The matrix difference of g1 and g2 is 1, the matrix difference of g2 is also 1. But the actual curve difference between g1 and g2 is:

(7)

The actual curve difference between g3 and g4 is:

(8)

This is 1000 times of difference! So to use the difference of matrix is not an efficient method to describe the difference of the curve. Maybe we should use normalized matrix first, then use the difference methods. There are a lot of normalization methods, which one could be the best choice? I want to leave this to the future.

Though powerful and interesting this method is, the practicability of it is remained being questioned to. First, the vector of an element is based on word vector. The word vector that I use is generated by word2vec with the Weibo corpus as input. The design purpose of word2vec is to make the words with similar context to be have similar vector. Under this condition, the vector itself cannot tell any information about a word. Thus, the distribution of element vectors of a given chain may be very sparse. The elements that on the same curve may have strong different meanings. For example, (-1,1), (0,0) and (1,0) are on the curve of y=x^2, also (-10, 100), (0,0), (10, 100) can be reduced to this curve. But the cosine similarity between (-1, 1) and (-10, 100) is not close to zero. The problem caused by this condition is that this method may result in ‘False Positive’. All elements on two close curves may be very different from all others. Then it is hard to say that these two chains are similar. The key point is, given two word vectors which can represent two points, the points between these two points may have no relation to these two points if they are very far away from each other. Second, how to control the dimensionality of the reduced curve? If we perform dimension regression to the reduced curve, the workload of computation of the comparison between the curves will be reduced a lot. But we have to make sure two reduced curves are under the same dimensionality space. This makes the reduction process harder.

2.4.2 The concept of layer level

Based on the challenges I proposed in 2.4.1 section, the method of linear regression is voted down. It is too time consuming, also its practical meaning is not that strong. So I want to focus more on how use the traditional graph searching methods but still can deal with the different chain length problem.

After thinking carefully about the content of Info-Box of the Wikipedia, I made an conclusion; Only the relation like ‘wikiPageRedirects’ should we make an gap on the chain comparison. The relation in Wikipedia Info-Box is very regular and universal, thus it is not likely that one relation1-entity1-relation2-entity2 can be reduced to relation-entity. If there is, then the relation1 here is ‘wikiPageRedirects’. So I’m thinking about adding a layer level concept to each element, so the element with higher physical depth inside the graph can have a shallower depth value. This value is its layer level.

There is another reason for this design. Beside the relation ‘wikiPageRedirects’, I also want to extend more relations at the same layer. For example, other similar entities but not recorded in Info-Box. Take ‘薄熙来’ as an instance, ‘薄熙来事件’ is another entity that related to ‘薄熙来’ closely. In Wikipedia, there some places that ‘薄熙来’ is used to refer the event ‘薄熙来事件’. This is called ‘alias’. Alias is an important supplement to the entity. Alias is not like metaphor, alias is used to be as an abbreviation in most cases. Here, ‘薄熙来’ is an abbreviation of ‘薄熙来事件’. The relations of ‘薄熙来事件’ has ‘被告人员’ 🡪 ‘薄熙来’, which shows the strong relation to ‘薄熙来’. It makes sense to let the alias have the layer level as the original entities have. But some alias have some ambiguities. For example, ‘2001’ is the alias of 573 terms. Here is a sample of these terms: '2001年亞洲青少年女子排球錦標賽','2001年世界青年冰球錦標賽','2001年度全国花样滑冰锦标赛','第73回選拔高等學校野球大會','第83回全國高等學校野球選手權大會','2001年中華職棒明星賽','第13回日本女子足球聯盟','2001年U17世界盃足球賽','2001年亞洲輕艇競速錦標賽','2001年美国拉丁裔媒体艺术奖','2001年欧洲歌唱大赛','2001年歐洲花式滑冰錦標賽'. ‘2001’ can refer to a lot of concepts, including the starting year of the term of ‘薄熙来’. From the inverse view, the alias of one entity may have no strong directivity to the entity it refers. For example, here is the list of all aliases related to ‘阿富汗戰爭 (2001年)’: {'2001': 1,'2001年入侵': 1,'2001年入侵阿富汗': 2,'2001年對阿富汗的入侵': 1,'2001年美国对阿富汗攻击': 1,'2001年阿富汗战争': 6,'2001年阿富汗戰爭': 128,'入侵阿富汗': 1,'军事攻击': 1,'出兵阿富汗': 1,'參與維和行動': 1,'叛亂份子': 2,'叛亂部隊': 1,'塔利班政權垮台': 2,'巡邏': 1,'戰爭': 1,'攻擊行動': 1,'炸彈爆炸': 3,'炸彈襲擊': 1,'發動襲擊': 1,'發生戰鬥': 1,'空襲阿富汗': 1,'第二次阿富汗戰爭': 1,'美国入侵阿富汗': 1,'美國介入': 1,'美國攻擊阿富汗': 1,'美軍攻打阿富汗': 1,'英軍自2001年對阿富汗展開軍事行動': 1,'襲擊': 1,'造成': 1,'阿富汗': 11,'阿富汗作战': 1,'阿富汗作戰': 1,'阿富汗入侵作戰': 1,'阿富汗反恐戰爭': 2,'阿富汗战争': 20,'阿富汗戰事': 1,'阿富汗戰場': 1,'阿富汗戰爭': 106,'阿富汗戰爭（2001年－現在）': 1,'阿富汗的軍事行動': 1,'阿富汗行動': 1}. The number after the alias is the repeating time of this alias. For example, '2001年阿富汗戰爭': 128, this means that the alias '2001年阿富汗戰爭' has appeared for 128 times in Wikipedia. From this list, we may find some aliases are not that strong related to ‘阿富汗戰爭 (2001年)’, for instance, '造成': 1, '發生戰鬥': 1. These words are too common. Also, the alias like '2001': 1 has strong disambiguates. It can refer to a lot of other events that happened at year 2001. So the alias data has to be cleaned before it is used. Common aliases should be filtered out. But how to define an alias to be common or not is not easy. If we use the count number as a threshold, then many unimportant and important aliases share the same count number. For example, as to ‘阿富汗戰爭 (2001年)’, the alias '阿富汗作战': 1 and '造成': 1 have both 1 count number of appearance, but the importance of them are much different. Maybe a better way to resolve the relevance between the alias to the entity is to count the tf-idf of the alias inside the page of this entity. Or we can even train the model to determine whether an alias should be remained to be along with the entity.

Another problem is the linking problem. One entity may have many aliases, while one alias may have many entities. Like I mentioned before, the alias ‘2001’ have 573 related entities. If we want to find the related entities of ‘2001’ from the clause ‘the starting term of ‘薄熙来’ is 2001’, then among 573 related entities, which ones should we choose? Most of 573 entities are events happening on 2001, so which events are most close to ‘薄熙来’? It requires more information about ‘薄熙来’, otherwise without information it is hard to determine.

So the usage of aliases to expand the graph is not easy. Though it cannot guarantee the accuracy of the graph, it can help improve the integrity of the graph. In my design, for each entity in the graph, I split them into some basic morphemes and their neighborhood-way combinations. Then for each split item, I treat it as an alias and try find its related entities to it. The entities are treated as the same layer to it.

During the splitting, if the item split out from the element is an entity, then I make a new relation-entity to the element as a new member of its children. The relation for it is ‘subEntity’. For example, the sub entities of ‘母亲：胡明’ is ‘母亲’ and ‘胡明’. This design is used to expand the literal node whose component is complex to recognize. The focus point of ‘母亲：胡明’ is ‘胡明’, but we need to tell machine to recognize it too. The simplest way to do so is to do word segmentation, so the quality of word segmentation would affect the quality of the extraction of the entity. For example, ‘上一任市长’ should be segmented into ‘上一任’, ‘市长’, but it can also possibly be segmented into ‘上一’, ‘任市长’. The best solution is to try all possible segmentations, and then filter out the one that has least entities. But this may bring some noise.

In conclusion, I designed two ways to expand the graph. One is to find the entities among the element, another is to find the alias to the entity. An element with the element\_type attribute equaling to ElementType.entity is not sufficient to be an entity if it is at the leaf layer of the graph. The elements at this layer are the raw query results from SPARQL, so the value is either entity or literal. To expand the graph, we have to turn literal into entity. Case one is that the entity is embedded into the literal, this can take 母亲：胡明’ as the example. Case two is that the entity is from the related entities from the split literal, this can take ‘2001年’ as the example. The entities found from these two ways are marked as the same level, so as the entity linked by ‘wikiPageRedirects’. The entities at the same level will be compared with.

2.4.3 The concept of abstract entity

Some entities are hard to compare; For example, two persons. Person is not the entity with concrete concepts. In this project, most morph and target entity pair is the name of a person. If person entity appears in the sub graph, how can we deal with it? Can we treat them as a new morph and target entity pair? Then where is the end of this comparison? The depth of this comparison could be endless. Also, the abstract concept is not limited to person. Organizations, geographical positions, terminologies and so on are all abstract concepts. Maybe we can reduce our project to the general abstract concept linking, which is much harder. To make the abstract concept linked, there are must be some concrete concepts inside the graph. What is the basic concrete concept? How to describe? If we can deal with it, then we can find the knowledge structure of human beings, the way that we observe the world. Maybe it is starting from basic geometrical relationships, numerical systems, the physical rules of things, and so on. From another point of view, it is the analysis of language. Our human beings exchange ideas by using language. The words, tune, grammar in the language itself describe the rule we learn the world. We link the concepts in the world to the language. So there must be some basic elements in the language that can strongly related to the real world. If we can dig out these basic elements inside the language, and use them to describe the abstract concepts, then abstract concepts can be compared. For example, the word ‘宰相’ and ‘总理’ are both abstract words, but they both have relation to the ‘high-order position of the job’ and ‘managing the country’. Two abstract concepts can be linked together due to their common sub concepts. This kind of relation network is hard to be found from Info-Box of Wikipedia, since the duty of Info-Box is not to explain what the entity is but to tell other entities that have relations to it. These describing information are more likely to be found in the main content of the wiki page, from which it is also harder to extract relation-entity tuples.

Besides the concept network, the word vector may also can describe the similarity between the abstract concepts. But for some similar entities but under different cultural background, the word vector may fail to do so. For example, with the word2vec model generated from Sina Weibo 2012 corpus, the most similar words to ‘宰相’ is ‘首府， 明武宗， 严嵩， 重丞 … 相国， 言官， 刺史’, the most similar words to ‘总理’ is ‘总书记, 总统，外长，领导人，温总理，…，朱镕基’. From these lists we can find that each entity has a strong cultural background. If we compare them directly, the cosine similarity value is 0.479970956699. The similarity between ‘宰相’ and ‘首府’ is 0.76245748, the similarity between ‘总理’ and ‘总书记’ is 0.742168. So 0.479970956699 is not a high value to indicate that ‘宰相’ and ‘首府’ are similar.

Another way to describe the similarity of abstract concept is to use the connectivity. Suppose we have two concepts P1 and P2, from each one’s sub graph we can find some similar concepts (Here concept represent an element inside the graph). The chain between the concept P1 or P2 to the concept of similar concept pairs in each sub graph may have weak similarities. I designed a connectivity to relaxing the similarity requirements to such kind of chain. If the similar concepts have a strong connectivity to P1 or P2, the similarity value of P1 and P2 should be higher. For example, (P1, R11, C11, R12, C) and (P2, R21, C21, R22, C) both have a concept C at the end of each chain. R11 is not similar to R21, C11 is not similar C21, R12 is not similar to R12. But R11, R12, R21, R22 all indicates very strong relation, such as ‘儿子’, ‘继任者’,’担任’. Then P1 and P2 have a similarity. Actually in Wikipedia’s Info-Box, all relations are very strong. So this formula may be reduced to, if from one concept A, we can find some sub concepts related to it and similar to the other from other concept B, then A and B have a similarity. If A and B have more sub concepts that have connectivity, then A and B have more similarity.

This problem can also be a supervised problem. If we can label all element pair with similar or non-similar, then we can use the relation-entity network to train how to judge whether an element pair have similarity pair or not, not limited to the abstract element pair. But this may take a lot of efforts.

2.4.4 Compare the elements in phonetic and spelling feature

One important relation between the morph ant its target entity is the phonetic and spelling feature. For example, ‘西红市’ is phonically similar to ‘西红柿’. This morph is used to refer to ‘重庆’, because both of them have some relation to the entity ‘红’. Though ‘西红市’ itself can have relation to ‘西红柿’, the later one are actually a concrete concept that can be better understood by human beings. The phonetic feature can greatly help to find more relations between the morph and its target entity. For some morph and entity pair, it’s vital to explain their originating relation. Similarly, the spelling feature also has the same importance. The example of ‘西红市’ and ‘西红柿’ is also an example of similarity in spelling feature. This spelling feature is much common in Chinese language, since our language is hieroglyph.

Whether to let these features to appear as new entities that have the same level as the center entity or to appear as the attributes of the center entity is remained being discussed. Some are suitable to extend as extra entity with same level, but others are not. It is hard to judge whether a literal with the similar phonetic feature to the center entity is an entity. Few of these variants are entities. Thus to find the phonetically similar entities will aggravate the computation work load. It is kept as an attribute, then only when comparison are two entities needed to compare whether they are similar or not. The disadvantage of it is the lack of finding out interesting entities in a positive way. These interesting entities may help human beings understanding the originating relation between the morph and its target entity.

2.4.5 Similar words

Since we can use word vectors to represent the elements, we can also use word vectors to find the similar words. Thus we can expand the graph by adding these similar words. The relation between them is ‘similar’ or ‘closely related to’. Also, they are at the same level.

Up to now, the relation ‘wikiPageRedirects’, ‘subEntity’ and ‘similar’ are all the relations that at the same level of the center entity. The entities mapped by alias is included in ‘subEntity’ relation.

As I mentioned before, the most similar words to ‘总理’ is ‘总书记, 总统，外长，领导人，温总理，…，朱镕基’ from the word2vec model generated by Sina Weibo 2012. Among this list, there are terminologies, person names and so on. These are reasonable associations to the entity ‘总理’. But they are associated in Sina Weibo corpus. The relation-entity graph I used is from Wikipedia. This will result in a cross-genre problem.

2.5 Visualization

To better show the graph, the visualization can help a lot. Before I start this project, I found a powerful visualization tool for SPARQL. This project is a sub project of DBpedia. Its name is ‘RelFinder’. Here is its web url: <http://www.visualdataweb.org/relfinder/relfinder.php>. This project is to find all middle relations between two entities and link them as a graph. You can make different settings on this website. If you have your own SPARQL server, then you also utilize this webpage to search for the data in your database. The website itself provides the DBpedia database. It’s all English. So the Chinese Wikipedia data cannot be found from the default setting. Following Figure 7 is an example of the visualization. As you can see, this follows the linear logical reasoning principle, which is not what we want to do. Besides, it also cannot show the entire sub graph of one entity under a certain depth limit. Though the power of it is limited, it still can provide very useful visualization to explore the data set.

Since the default setting of it is from English Wikipedia, after I built a SPARQL query end point with Chinese Wikipedia as the data set, I created a new setting for this visualization tool. Figure 8 is an example between ‘薄熙来’ and ‘张德江’. But due the entity linking of the extracted Chinese Wikipedia data set hasn’t been updated to the data set itself (this task is done during building the graph to save the time), so the searching result with this data set may not cover all. In the future, the entity linking would be done before other works begin. Currently, no relations can be found between ‘薄熙来’ and ‘平西王’.

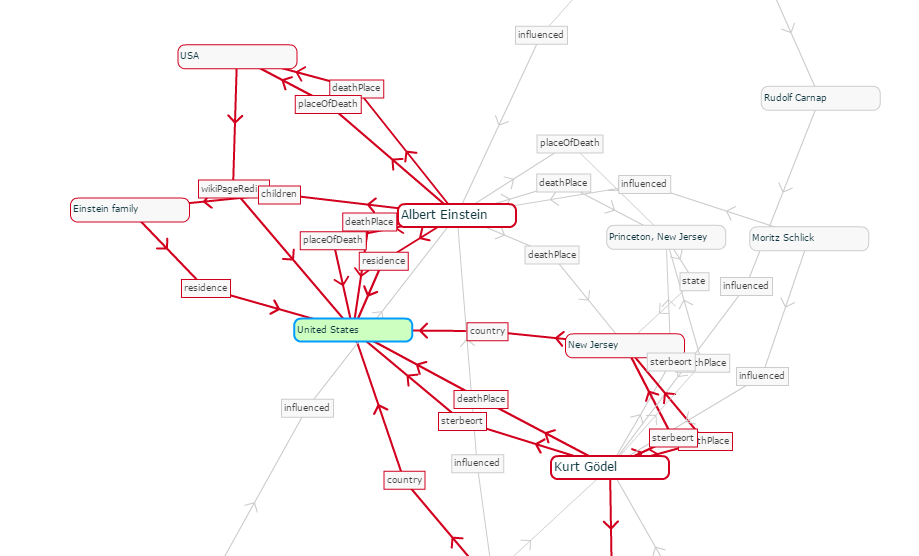


Figure Relation map between Einstein and USA

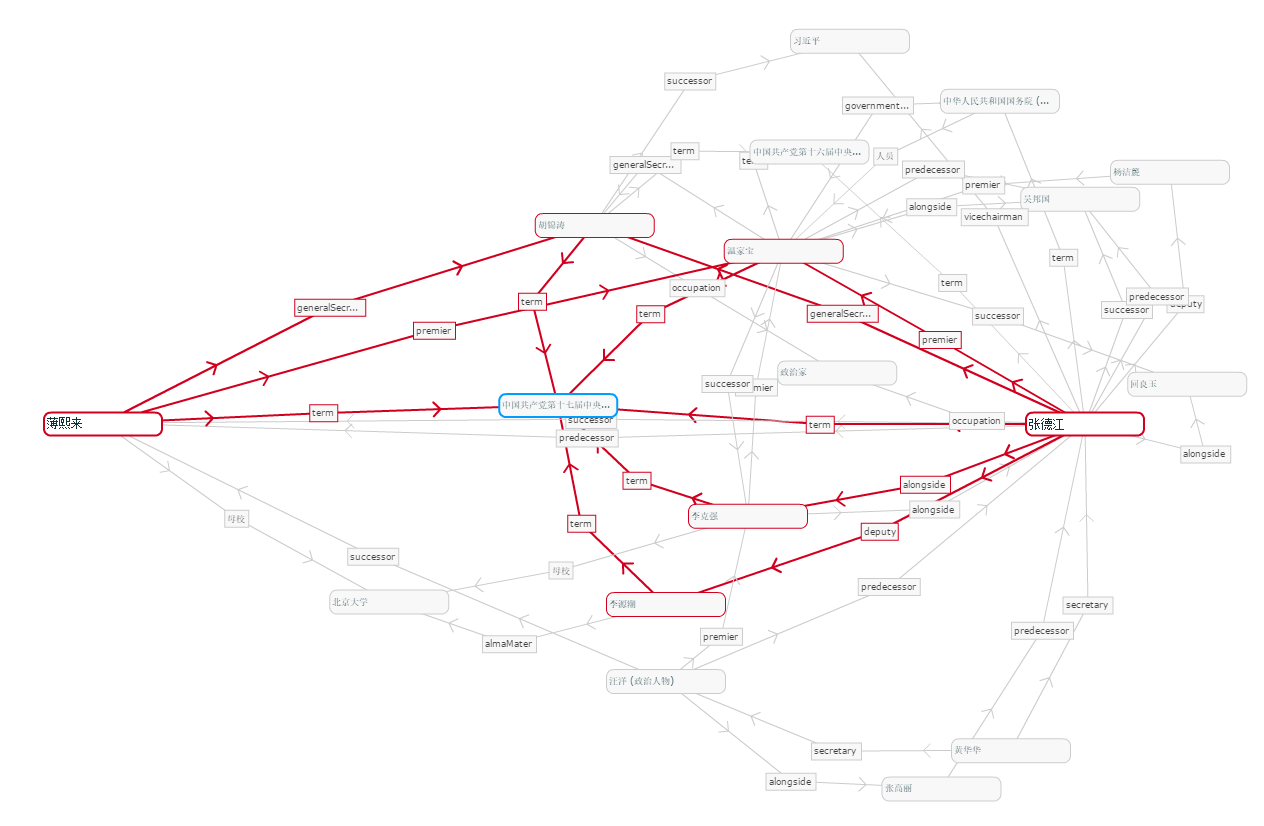


Figure Relation map between 薄熙来 and 张德江

Another visualization tool is built by graph-tool. Figure M and Figure N shows the example of ‘薄熙来’. Graph-tool is a powerful network visualization tool, and it use python as its coding language, so I chose it as my visualization tool. I can control the display effects and the graph elements, so it makes the visualization more flexible.

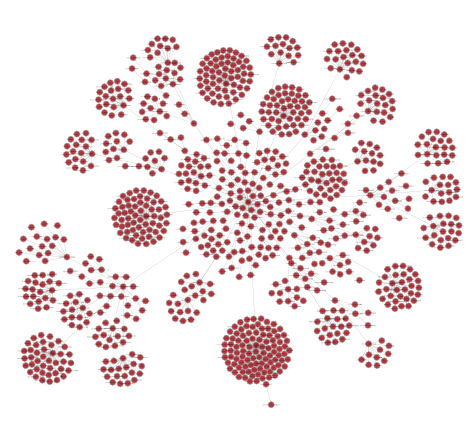


Figure Entire graph of 薄熙来

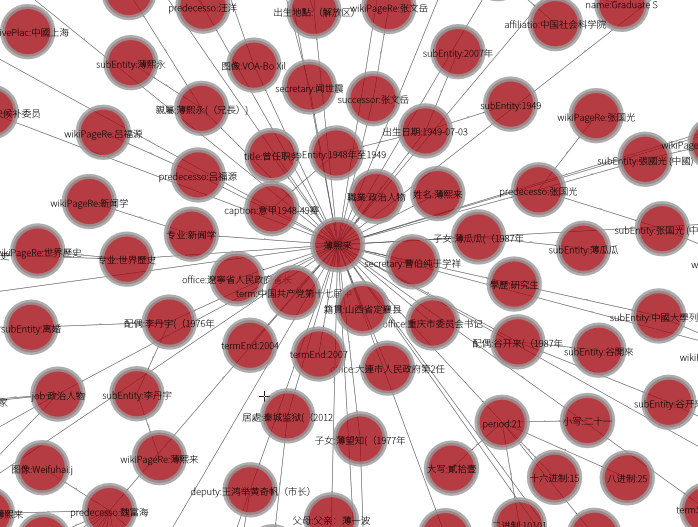


Figure Regional graph of 薄熙来

2.6 Conclusion

Half of my designs are aborted finally. Many trials have been made after one design was raised. Also it is hard to find similar works.

Chapter Three Code Design

3.1 Structure of the codes

As the Figure 11 shows, in my project repository, I have a codes folder. I wrote this project as a module, so you can see \_\_init\_\_.py. There are four main classes in my project, they are: GraphBuilder, GraphMatcher, GraphStatistics, GraphVisualizer. There is one important basic class: Element. This class serves to be the basic element of the graph. There are two facilitating module: Cache and util. The first provide the cache feature to my other functions. The second one provide some auxiliary functions, like the python decorator of displaying the computing the time. There is one module: NEResolver to retrieve all named entity from Chinese Wikipedia. There are two external modules: jieba and zhtools. The first one is used to do word segmentation, the second one is to convert the Chinese word from simple version to traditional version, or vice versa. W2VServer is used to launch a web server that can calculate the similarity between two Chinese words. EN2CNDict is a module to build an English to Chinese dictionary. The remaining ones are legacy codes during my exploring period. The scripter folder contains some scripts to start the fuseki server or to do SPARQL query.

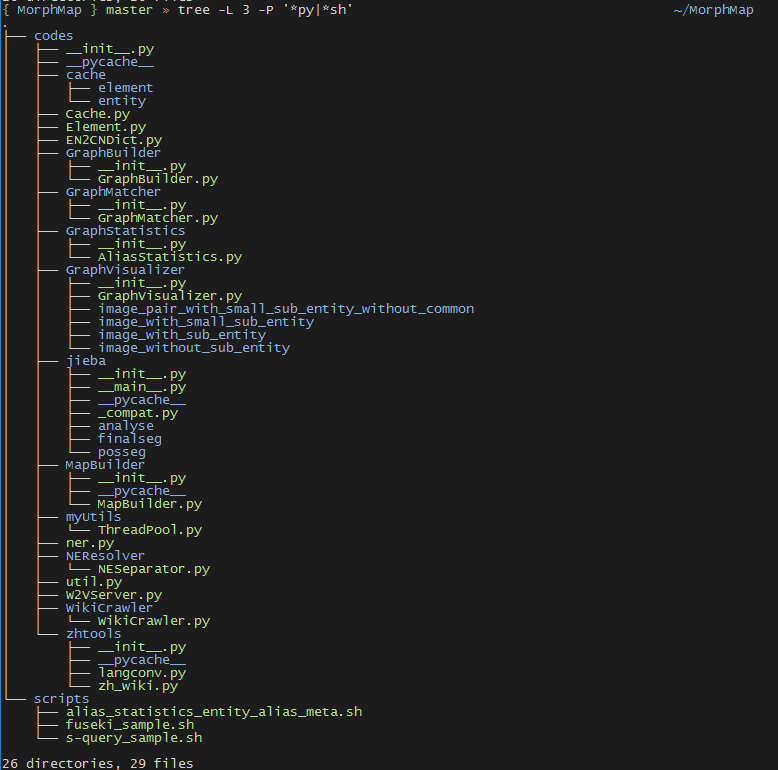


Figure Code structure

3.2 Element

In the module of Element, I defined a class Element. Its attributes are name, children, parent, level and element\_type. The default value of children is an empty list, but it has to assign a new empty list to the children, otherwise the initialization mechanism of python will let all instance of this class share the same list. During the initialization, I also add the new instance to a ‘entity\_dict’ dictionary so that the entity that has been created could be recorded. This design is to deal with the loop problem. Also I rewrite the \_\_str\_\_ function of this class, so that when print function is called to this class, this can print in a pretty way. Here is its printing example Figure 12:



Figure sample of printing an element

It will recursively print its children and make them in a line. If the graph is not big, this is also a good to print the graph. Also I wrote a function called getHistoryText, which can print all the parent chain from this element. Also I wrote a function called getTrueParent to get the parent has one bigger level than itself. Due to the level design of the graph, the parent of one element may be still at the same level. Also I wrote a classmethod called concat(cls, element1, element2) to concatenate two elements as the parent and its child. Because to concatenate two elements, I have to append element2 to element1’s children, also I have to set the parent of element2 as element1. The level of element2 also should be set to one plus the level of element1. In this module, I also wrote a class ElementList. This class is originally designed to serve as a cache to record all elements that have been created. But it seems having some problem with the code structure, so I finally put it away.

3.3 Cache

The cache module is designed to provide general cache feature. The cache function is designed as cache(filename, func, \*args, \*\*keywords). If the file with filename exists, then cache function will use pickle module to load the file and parse it as a python object and finally return. It this file doesn’t exist, then cache function will run func(\*args, \*\*keywords) to get the data, and then dump it by pickle to the file with given filename.

3.4 Util

This module currently contains one function showComputingTime(f). This serves as an python function decorator, which can print the running time after the function is finished.

3.5 W2VServer

This module is used to launch a web server that can take two Chinese words as inputs and return the similarity. This is built upon web.py. The word2vec model is loaded by gensim module. This server can save a lot time of loading the model for the scripts that uses this model. Now script can compute the similarity from just one simple URL query. For example, <http://202.120.38.146:9602/?w1=%E5%AE%B0%E7%9B%B8&w2=%E6%80%BB%E7%90%86>. %E5%AE%B0%E7%9B%B8 means 宰相, %E6%80%BB%E7%90%86 means 总理. The result is 0.479970956699

3.6 NEResolver

This is module is used to find all named entities among Chinese Wikipedia. A sample of the list of NE is: ‘GPE 釜山,ORG 基督教神學院,GPE 釜山,GPE 釜山,MISC 忠清南道天安,PERSON 大衛·當臣’. Each named entity is clustered into 5 categories: ['GPE', 'ORG', 'MISC', 'PERSON', 'LOC']. The named entity is extracted by Stanford Core NLP. To make things easier, I launched this tool as a server, so it can run as daemon process. To start it is very time-consuming every time, so it’s better to separate it out from the codes to be as a single server. To let it support Chinese words, I’ve taken a lot of time in studying on to let it load the Chinese settings. The setting is very complex:

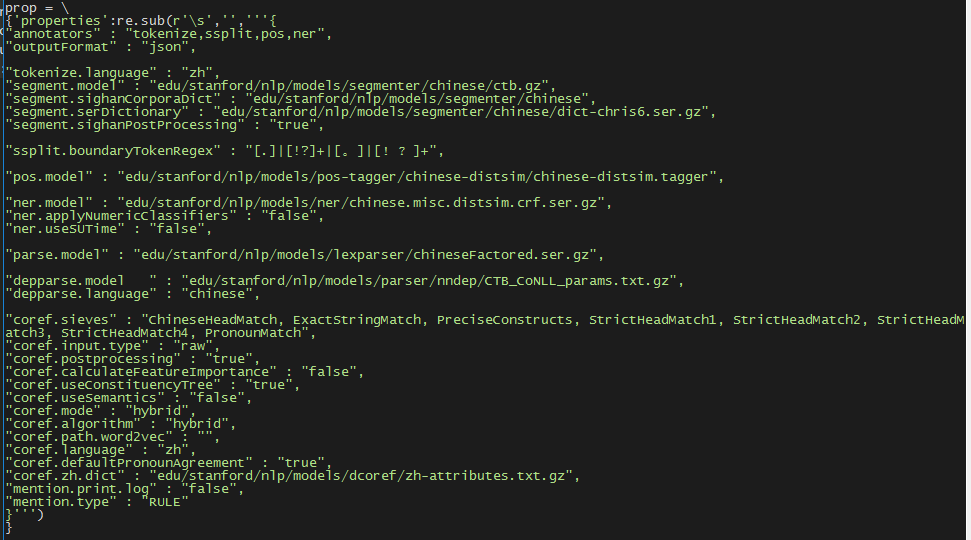


Figure Chinese property for Stanford Core NLP

Also I wrote a very interesting shell command to parse the result. This command is

awk -F{ '{for(i=1;i<=NF;i++)print $i}' | sed -n '/ner/p' | sed '/"ner":"O"/s/.\*//'| sed 's/.\*"word":"\(.\*\)","originalText.\*"ner":"\(.\*\)"},/\\1|\\2|/;' | awk BEGIN{RS=EOF}'{gsub(/\\n/," ");print}' | sed ':a;s/ //;t a' |awk -F '|' '{printf("%s ",$2);for(i=1;i<NF;i+=2)printf("%s",$i);printf("\\n");}'

This command can make the words with adjacent named entity tag be combined together and marked with the same tag. For example, ‘双 MISC 十一 MISC 是 O …’ The words ‘双’ and ‘十一’ have the same tag ‘MISC’ and they are adjacent.

These named entities are used to detect whether the entity inside the relation-network is a named entity. If so, then mark it as an abstract entity, which is needed to be resolved by other entities.

3.7 EN2CNDict

This module is to load a ‘English to Chinese’ dictionary file to be a python dictionary. I wrapped it into a class called EN2CNDict and rewrote the \_\_getitem\_\_ and \_\_iter\_\_ function so that this class can be used as a dictionary. For each English word, there are several Chinese meanings. I append all Chinese meanings to each English word.

3.8 GraphBuilder

The GraphBuilder module serves to build a graph. Inside it there is a GraphBuilder class. When it is initialized, a root Element is initialized. Inside this class, I designed a rawQuery function, which can make an shell command ‘s-query’ to get the output of SPARQL query. The SPARQL server is set as the fuseki server launched at the lab node. I also design a function query(self, name). The name will be converted into simple Chinese and traditional Chinese, then each one will be passed into \_query(name) function. This process is designed to guarantee more search result. The Chinese Wikipedia data is mixed with simple Chinese and traditional Chinese. So for each query, all words will be converted into two versions. This is done by zhtools module. As to the \_query(name) function, it uses ‘http://zh.dbpedia.org/resource/’ to concatenate with the name, and then pass it to rawQuery function. If the result of rawQuery is not empty, then the result will be converted from json text into a python object and assign it to data. I use a map function to make the data cleaner. Finally, the data is filtered by filterQuery(self, query\_result) function. This function is used to filter out tangential relations, such as ‘图片大小’, ‘align’. I also designed a getOneHop(self, ex) function to get one hop for one entity. The input entity/literal is first exploded into many sub literals. This done by a entitiesof(self, literal) function. This function serves to link the entity of the given literal, though currently it just can split the literal into some meaningful segments. Firstly, the literal is split by punctuations, then each part is word segmented by jieba module. After the input entity/literal is exploded, then getOneHop function will append other related entities according the alias2entity dictionary. The sub entity list contains the original input, the exploded literals and related entities. Then for each entity in this list it is queried for its other relations by using query(self, name) function. I also designed another function called tup2graph(self, tup, init\_level, init\_el = None). This function can convert the data generated by getOneHop(self, ex) into a linked list with Element as its nodes. This function works an adapter. The mission of it is to set the level, element type and their parent-children relations. Also, it has to deal with the memory leakage carefully. In the design of Element class, a duplicate entity will be created but its children are linked to the original entity’s children. I also designed a doElementOneHop(self, element\_ex) function to wrap the getOneHop(self, ex) function. This function can take an element as input and do one hop for it. If one of the relations among the first hop is ‘wikiPageRedirect’, then it will do one hop more, until no ‘wikiPageRedirect’ is met. I also designed a expandGraph function(self, deep\_level = None) function to expand the graph with one hop. Each leaf element will be got one hop out until the deep\_level limitation is met. I also designed a getGraph(self, deep\_level = 2) to wrap the expandGraph function. It will automatically repeatedly expand the graph until the deep\_level is met.

3.9 GraphMatcher

This module is served to compute the similar parts between two graph. I designed a layerSearch(self, node) to iterate all nodes at the same layer. This is a generator function. Also I designed a getSameLevel(self, node) function to retrieve all nodes that at the same level. This function is used in layerSearch. I also designed a function called computeBiGraphScore(self, graph1, graph2). This function will iterate layers from two graphs and compute all possible combinations between two nodes and then record their similarity score into a dictionary. The score of the parent nodes will be multiplied with a factor alpha and then be added to the score of current node pair. This score is the final score of current node pair. So the computation can have some history feature. The similarity computation is done by a getScore(self, element1, element2) function. This function will try to compute the word vector similarity between two elements if they are not the same. Then the score will be passed into a math function f = lambda x:-1 + 2\*((e(x)-e(-1))/(e(1)-e(-1))). The word vector similarity is computed by function getW2VSimilarity(self, name1, name2). This function will visit the web url = 'http://202.120.38.146:9602/?{0}'.format(data) where data = urllib.parse.urlencode({'w1':name1, 'w2':name2}). The getScore(self, element1, element2) also uses getComparableList(self, literal) to expand the element so that they can be easily compared by word vector. Finally, each chain will be computed with a score, the chain will be sorted according to the score.

3.10 GraphStatistics

This module is to count everything about the graph, alias data and so on. The dictionary from alias to entities are also generated from here.

3.11 GraphVisualizer

This module is to visualize the graph. First is has to convert the graph built with Element node to be the nodes and edges defined by graph\_tool. Also according to the rule of graph\_tool, the property map is required to be created for nodes and edges. Finally, with one command the image of the graph can be computed: gt.graph\_draw(vg, bg\_color=[1.,1.,1.,1], vertex\_text\_position=-0.5, vertex\_text=vg.vp.name,vertex\_text\_color=[0.,0.,0.,1], edge\_text=vg.ep.name, vertex\_font\_size=15, edge\_font\_size=1, edge\_pen\_width=1, edge\_text\_distance=0.9, vertex\_size=80, vertex\_font\_family='Noto Sans CJK SC Thin', edge\_font\_family='Noto Sans CJK SC Thin', output=output\_image, output\_size=(6000,6000)). The font family has to be set, otherwise the Chinese character won’t be correctly displayed.

3.12 Conclusion

There are near 40,000 characters of codes. Most of them are written in a terse way. You can visit it from <https://github.com/122689305/MorphMap>.

Chapter Four Result

4.1 The result of GraphMatcher

4.1.1 The first version of result

--> 薄熙来 --母校--> 北京市第四中学 --district-->

北京市 --民族-->

--> 李自成 --首都--> 北京市 --民国-->

北京市 --民族-->

From this we can find the ‘薄熙来’ and ‘李自成’ both has similarity to the ‘北京市’.

4.1.2 The second version of result

8 李永金 吳世璠 -0.7376

-->薄熙来0--successor继承者1-->李永金2--subEntity2-->李永金\_(大连市长)2--姓名3-->李永金4

-->平西王0--wikiPageRedirects0-->吴三桂0--successor继承者1-->吳世璠2--姓名3-->吳世璠4

From this we can find ‘薄熙来’ and ‘平西王’ has similarities due to ‘李永金’ and ‘吴世璠’. It is ranked as 8.

92 1992 1908 -1.7616000000000005

-->薄熙来0--term学期Start动身1-->19922--subEntity2-->1992年冬季奥林匹克运动会2--opening开Ceremony典礼3-->19924

-->平西王0--wikiPageRedirects0-->吴三桂0--birth出生Date日期1-->1612-06-082--subEntity2-->1908年夏季奥林匹克运动会2--opening开Ceremony典礼3-->19084

The result of ranking 92 has lower relation.

4.2 The result of Visualizer

The following figures are visualization of ‘杨幂\_函数’. It is better to use some tools to open the image. The image is recorded in svg format. Each one has 6000\*6000 points.

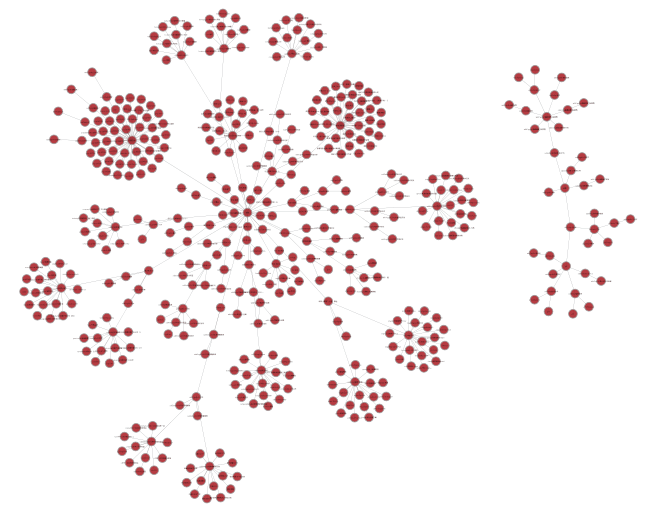


Figure Entire graph of 杨幂 and 函数

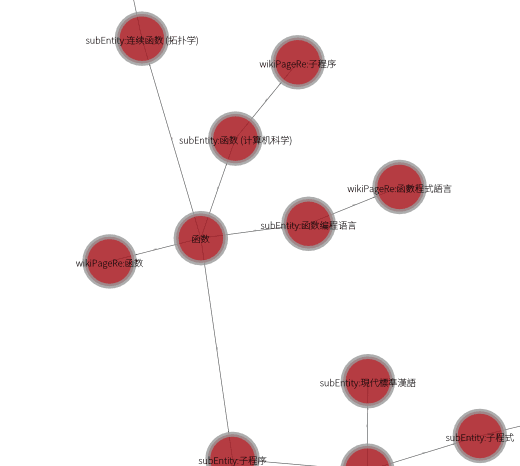
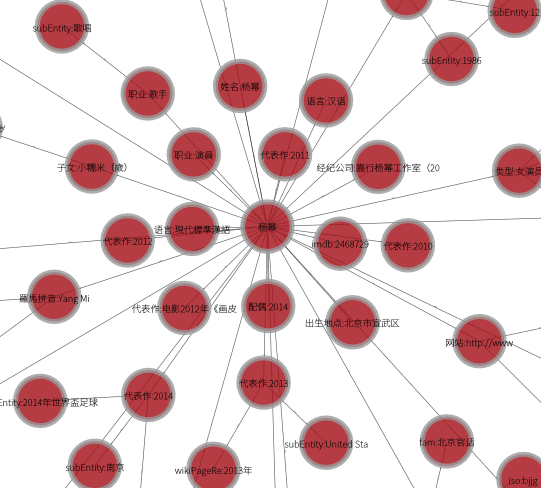


Figure Snapshot of 杨幂 graph Figure 16 Snapshot of 函数 graph

4.3 Conclusion

The visualization makes the result easier to be understood. The result of comparison makes some sense, but more accuracy is needed to perform a better explanation.

There are in total 1,288,121 entities, 1,983,129 aliases, 5,552,007 alias counts. There are in total 1,299,830 GPE, 211,401 LOC, 2,484,339 MISC, 391,918 ORG, 1,035,438 PERSON counts.

Chapter Five Conclusion

This project involves entity linking, knowledge graph, network visualization, information extraction, word sense disambiguation, graph algorithm and so on. To explain the originating relation between the morph and its target entity is a tough task. The integrity and the accuracy of the graph determines a lot to the final result. Also the way that how we understand the entity also affects the result a lot. To turn this understanding into the way that machine can understand is another big task. Here I have to admit that the final result is not as expected as I thought. Maybe if I just use a linear path to describe the relation between the morph and its target entity is much easier and easier to be understood. All I need to do is to clean the data, make the entity linking as accurate as possible. Though to do this I have to deal with the entire graph – 5.7G, which will take a lot of time, the result may be very impressive. The word vector that I used in this project also has its advantages and disadvantages. It is not powerful as I thought. To core part of this project is to compute the similar sub graph between two graphs. In the future, more effort should be paid in the data cleaning and the comparison algorithm.

Acknowledgement

Thanks a lot to Kenny, Frank and Bill. They provided a lot of ideas and encouraged me to proceed this project. Thanks to other lab members that have provided help to me.

Reference

1. Huang, Hongzhao, et al. "Resolving Entity Morphs in Censored Data." *ACL (1)* (2013): 1083-1093.
2. Zhang, Boliang, et al. "Be Appropriate and Funny: Automatic Entity Morph Encoding." *ACL (2)* (2014): 706-711.
3. Zhang, Boliang, et al. "Context-aware Entity Morph Decoding." *ACL (1)* (2015): 586-595.
4. Bordes, Antoine, et al. "Translating embeddings for modeling multi-relational data." *Advances in neural information processing systems*. 2013.