PyScholarGraph: A graph-based framework for indexing, searching and visualising relationships between academic papers

NICHOLAS CROUCH¹ AND DAVID M.W. POWERS²

Abstract

Although they have been studied for over 25 years, Graph Databases – which model data as a set of nodes and relationships – have been unfashionable for some time. This has led to a large development gap between Graph Database systems and more traditional Relational Database systems, such as SQL Server, Oracle and MySQL. However, the recent rise of social networking sites such as Facebook and Twitter, where data easily maps to a set of nodes with relationships between them, has led to resurgent interest and development in these systems and the unique visualisations and algorithms that they enable. This paper presents PyScholarGraph, a software package that builds upon existing Graph Database tools to provide an easily modified framework for indexing, searching and visualising academic papers and the relationships between them. The paper also presents two example ranking algorithms, which demonstrate how a traditional textual search can be augmented by graph-based algorithms, and the visualisation system, which provides an exceptionally intuitive way of finding interesting papers within the result set. Finally, the paper includes a short discussion of possible future expansions of PyScholarGraph, and concludes that while representing academic papers in the graph format shows excellent promise as a concept, much work remains in order to realise that promise.

¹ School of Computer Science, Engineering and Mathematics, Flinders University, South Australia.
² School of Computer Science, Engineering and Mathematics, Flinders University, South Australia.
Introduction

In the 25 years since research on Graph Databases began, there have been multiple phases of development on these systems (Wood, 2012). While relational databases remain the largest market for enterprise data storage (Emison, 2014), it is also obvious that graph data structures map well to both traditional and emerging data and problem domains.

In the current database landscape, particularly in enterprise data applications, relational database technologies remain the dominant market leaders (Emison, 2014). However, the emergence of large structured and unstructured datasets with non-relational characteristics has forced large organisations and researchers to find new and performant ways of handling that data. In particular, NoSQL (Not only SQL) (Tudorica & Bucur, 2011) databases have become relatively prevalent. In describing the hierarchy of NoSQL databases, Tudorica and Bucur (2011) edit the taxonomy suggested by the NoSQL Wikipedia article (‘NoSQL’, 2014) only slightly, suggesting that such databases can be divided into three categories:

- Document Stores;
- Graph Stores; and
- Key-Value Stores.

Of these categories, probably the least explored is Graph Stores (Angles & Gutierrez, 2008). Although researched extensively (Wood, 2012), Graph Stores have until recently been of little practical interest. However, they lend themselves to any data where the structure of relationships between data points is of as much – or more – interest than the data points themselves. The almost clichéd example of this is the social network, in which people are linked to each other by friendships. There are, however, other examples of data which have both obvious and non-obvious relationships of significance.

This paper presents a system, PyScholarGraph, that first retrieves data from the public-access CiteSeerX repository of academic paper metadata (‘CiteSeerX’, n.d.), indexes it to a Neo4j (‘Neo4j, The World’s Leading Graph Database’, n.d.) Graph Database, and presents a small number of both traditional text search and graph-based search algorithms to retrieve and rank that data. It is the intention that this forms a framework for further research and experimentation in both graph-based retrieval and visualisation mechanisms, as well as (eventually) being useful for user adoption. In that vein, a number of future development and research directions are presented, which would answer interesting questions and/or raise the utility of the system itself.
Program architecture

The system is written in the Python programming language (van Rossum & de Boer, 1991), and is divided broadly into three sections. The first section focuses on retrieving, storing and indexing the metadata, which also requires scraping the CiteSeerX web interface for information not provided through the Open Archive Initiative 2 protocol. The second section focuses on providing appropriate search algorithms with an appropriate mix of established good information retrieval practice and graph-based strategies. The final section is focused on visualising the data in a way that provides both structural understanding, and the ability to navigate through subgraphs intuitively.

Python language

The selection of the Python language for use in this project was largely based on personal preference. However, it is important to recognise the way that this selection influences the design of the project. For instance, there are a number of libraries (including those used to access the initial metadata, and parse the scraped data) that are readily available for Python that make certain tasks easier. Conversely, the Neo4j system is built in Java, and this necessitates the use of an intermediary library rather than the library that the Neo4j developers provide, or that much of the community is focused on.

Bulbs, or BulbFlow, is a library for the Python language developed mostly by James Thornton that aims to provide vendor-independent mapping to the graph domain (Thornton, n.d.), and serves to bridge this gap. In theory, the library allows connection to any Graph Database that implements the Blueprints framework, ‘a collection of interfaces, implementations, ouplementations [sic], and test suites for the property graph data model’ (‘tinkerpop/blueprints’, n.d.). In practice, the implementation can be somewhat difficult for various Graph Databases, and in fact the simplest method of using Bulbs can be to use it with version 1.9.4 of Neo4j (the current version is 2.0.3). There are numerous reasons for this, most of which revolve around breaking changes made to the graph model in version 2.0 of the software (and incompatibilities with other software).

Indexing

There are two separate conceptual parts of the indexing performed by PyScholarGraph: seed retrieval and citation following. In the actual implementation, there are several other indexing steps. These include the addition of titles to full-text indices and the generation of a term frequency list.
All such indexing steps should be included in the node-creation process but are omitted here as they represent little of interest past the vagaries of the software creation process.

Seed retrieval – OAI2

The seed retrieval stage consists of retrieving records from the Open Archive Initiative Protocol for Metadata Harvesting 2 (Lagoze et al., 2002) (OAI2) compatible database maintained at the CiteSeerX project (‘CiteSeerX Data | CiteSeerX’, n.d.). In Python, the simplest way to achieve this is to use the Sickle library (Loesch, 2013). Using Sickle and OAI2, it is simple to iterate over the collection sequentially. It is also, therefore, quite simple to build a small (or large) database of records in that library. This constitutes the seed for the larger index to follow.

It would normally be said that this process is embarrassingly parallel (Foster, 1995, Section 1.4.4). However, the iterative process implemented by the OAI2 protocol uses a resumptionToken, which cannot be relied upon to refer to a specific location in the record set (Lagoze et al., 2002). As such, the only way to divide this part of the task would be via date, which was determined to be inelegant and out of scope for this project. For this reason, it is quickest to use a small seed selection, and then expand it using the parallelised citation following stage. However, this presents a risk of under-seeding the index, as citation following will not seek out additional papers past the graph built from the seed. Conversely, over-seeding the index will be inefficient, and depending on the application may also result in a dataset too large to be managed.

Citation following

The second stage of the indexing process is to build the citation graph. Given the initial seed set of papers, $S$, the final set of papers, $V$, and citations, $E$, can be obtained via the following algorithm:3

3 Note that similar code samples in this text are not valid Python. They been simplified for ease of publication and for reading by those not familiar with the language.
In PyScholarGraph, this method depends on the ‘cited by’ page provided by CiteSeerX. As this page is presumably generated by a templating system, the data is particularly easy to obtain using the BeautifulSoup HTML parsing system.

Unlike the first stage of the indexing process, this is embarrassingly parallel, as it does not rely on the OAI2 access method except for singleton paper retrieval. As such, the process can be parallelised, using either the Python multiprocessing or threading library. In either case, it is essential if using the Bulbs and/or Sickle library to ensure that new Graph handles or Sickle instances are created for each thread or process, as they will block execution.

**Retrieval**

PyScholarGraph has three levels of retrieval mechanism. The first, similar to the seeding stage of the indexing process, serves to seed the user’s searching process. The second is similar in mechanism to the first, but seeks to expand upon the original search rather than a new process. The final, and most interesting, are the graph-based stacking search mechanisms, which take advantage of the unique features of the Graph Database.

**Initial text search**

The first searching mechanism is based on a free text search query provided by the user. This query, $Q$, is considered a set of words. Each of these words is then compared to a term frequency list, and ranked according to their inverse frequency over the entire corpus. Each word in the list is then scored on a normalised scale, again according to inverse term frequency. Words that are extremely popular throughout the corpus (‘the’, ‘or’, etc.) are removed from the list by specifying a cut-off on the normalised scale. Each paper containing any of the words in the query is then retrieved using the Lucene full-text index of titles provided by Neo4j. Each of these papers is then scored by adding up the scores of words in the initial query, and the sorted list is displayed to the user. This process can otherwise be expressed as follows:

---

4 For the purpose of this project, document titles were used as they were the most reliably available property in the dataset.
The secondary search, the *comparison search*, is similar to the initial search. However, instead of using a user supplied query string, the comparison search uses the title of the node currently being viewed by the user.

**Advanced graph algorithms**

The final search method uses the graph structure of the data to add several methods of ranking to the search results. Each of these ranking algorithms first requires the secondary search to be run using a seed paper. From there, they will change the ranking of each of the original results depending on the algorithm used. There are several rules that must be followed in order to ensure that compatibility is maintained when ranking algorithms are used together (or ‘stacked’):

1. Each ranking algorithm must accept and return a list of papers $P$, with a corresponding ranking score $P_s$. This ranking score is altered by some value $\Delta P$ to reflect the relevance of that paper to the seed paper $Q$.
2. Papers must not be added to $P$ by a ranking algorithm. However, they may be removed.
3. Each paper $P_p$ must be compared independently of other papers in $P$. That is, $\Delta P$ must be constant given constant $P_p$ and $Q$.
4. $\Delta P$ must be between zero and one, where a higher score indicates higher relevance.
These rules ensure the order of rankings algorithms is unimportant, and that each of the ranking algorithms produces comparable results. There are two graph algorithms that are currently implemented by PyScholarGraph.

The Advanced Citation Score iterates through each paper in the result set and attempts to find the topic of each paper using the papers citing or cited by that paper. The detected topic is then compared to the detected topic of the seed paper, and the comparison paper is ranked accordingly. Topic detection on a single paper is performed by first retrieving the normalised inverse proportional word count for each word in the title, similar to the method used in the simple search. However, these inverse proportional word counts are then aggregated and again normalised, and the most common words in the aggregation will be scored highly. These words are thought to represent the most common topic matter around the paper, and hopefully therefore the paper itself. The same process is performed on the seed paper, and thus by comparing the two aggregations commonality in topic matter can be determined. A simple sample run of this process is visualised in Figure 1.

The Tree Support Score similarly iterates through each paper in the result set, but instead assigns a score based on how well cited a paper is compared to how many papers it cites. In theory, a paper that cites few papers but is cited by many papers represents a seminal paper – one which introduces new ideas and spawns a new field of research. In contrast, a paper that cites many papers and is also cited by many papers might represent a survey paper. Both of these classes will be ranked reasonably highly by this algorithm, which looks not only at each paper in the result set, but also a configurable number of ‘levels’ away to determine the weight of both the tree and the support. A third desirable class – papers that are new, but also relevant – will be penalised by this algorithm until they reach their appropriate number of incoming citations. This could be compensated for by weighting the date of the paper.
The final interesting aspect of graph structures is their intuitive and rich visualisations. PyScholarGraph provides a simple, but quite interesting, visualisation of search results. Each of the first 10 results, and up to three layers of citation from those results, are displayed as a graph using colour and spatial orientation to separate them.

Figure 1. A visualisation of the Advanced Citation Score ranking algorithm using sample numbers.
Source: Authors’ figure.

Visualisation
Retrieval

The primary issue with this visualisation is defining a suitable subgraph such that the visualisation is legible and can be processed, while also maintaining enough information to be interesting. This is particularly difficult given that some papers will have many citations that 'clog' the graph, and some will have very few, leaving the graph too sparse. In fact, some result sets can exhibit both sets of behaviour. After limited experimentation, the following method seems to deliver an aesthetically pleasing subgraph most of the time:

1. List the first 10 results of the query. These are the root papers.
2. For each root paper, add the papers that are cited or cite that paper to the list of rendered nodes. Add the relationships to the list of vertices.
3. For each of those results, add the papers that are cited or cite the result to the lists of nodes and vertices.
4. Repeat twice more. At any time if the number of nodes added as the result of a single root paper is 1,000 or more, finish the current node and move on to the next root paper.

This generates a list of nodes and vertices, which can then be passed to the display part of the software.

Display

For our purposes, the display of the visualisation is handled in the web browser, using the JavaScript library Sigma.js (Jacomy, n.d.). This library, with some small amount of work, is able to take a JSON dictionary containing both a list of nodes and a list of vertices, and create the display. This directs to the client machine the task of determining the optimal position of each node, and rendering the nodes in this position. PyScholarGraph currently makes use of the ForceAtlas2 placement algorithm supplied by Sigma.js to do this, leaving the majority of processing code to the Sigma.js tools. Sigma.js requires only that the server specify the original position and properties of each node in the graph.

PyScholarGraph makes use of some directed randomisation to present a reasonable set of original positions. First, each root paper is randomly located on the X and Y axes where $0 < x_i < 2000$ and $0 < y_i < 2000$. Then, while each other paper is being added, it is placed on the axes such that $-500 < x_n - x_i < 500$ and $-500 < y_n - y_i < 500$. As this uses Python's `random.randint()` function, which is uniformly distributed, this creates a square pattern when nodes are dense. Sigma.js' ForceAtlas2 will generally then form a circle of nodes around densely cited papers, with notable deformities where two root nodes share citations.
Future work

This paper essentially presents an indexing system, two graph-based ranking algorithms, a method of subgraph determination for visualisation and the framework to tie each of them together. Given that this is a first foray into the field, a large number of questions remain unanswered. This section outlines some expansions, refinements and experiments that we believe would be valuable for any interested researcher to pursue.

Indexing and term weight improvements

Currently, only metadata provided by CiteSeerX’s OAI2 interface is indexed, and that in a full-text Lucene index. Further, each algorithm currently relies on the title of each paper to determine the topic and first relevancy. The most obvious way to improve this would be to index full-text versions of papers, or at least abstracts. Indeed, even the current term weightings could be improved by calculating n-tuples rather than 1-tuples. However, it is clear that there are far more interesting ways to represent academic papers in keeping with the graph theme of the PyScholarGraph system.

Blanco & Lioma (2007, 2012) suggest representing papers themselves as graphs, and present a reasonably comprehensive summary of methods to do so in Blanco & Lioma (2012). They also present a number of ways to calculate term weighting, based on these graphs. Each of these would make for an interesting drop-in replacement for the current method of calculating term weighting (inverse proportional term weighting).

Further, there must be other relationships between papers that are not currently used in the search algorithms. One such relationship currently being indexed, but not explored, is that of the author–paper relationship. Other relationships may have to be indexed or discovered based on existing metadata – could one, for example, find interesting patterns in keyword usage? Improving the amount of information indexed in the graph could be especially powerful if combined with improvements in the visualisation technique.

Visualisation improvements

There are a number of deficiencies in the currently implemented visualisation; and a large amount of experimentation left to do. It is outside of the scope of this paper to provide a full literature review of the topic of effective visualisation, but some specific improvements are evident. The most obvious revolve around the legibility of the visuals – nodes should be bordered, and experimentation is required around the best method of distributing nodes to maximise legibility. The second improvement is to increase the interactivity of the visualisation.
Currently, basic zooming and panning, as well as hovering to view the title of a node, are available. However, in our testing, it was often frustrating that the graph could not be expanded, contracted or traversed from the visualisation itself. Improving this functionality such that it was useful for spotting interesting relationships and topic clusters would be of particular use. Finally, more methods of choosing a subset should be evaluated, particularly ones which show the different relationships obtained through indexing improvements. This includes showing how other users have traversed areas of the graph, and using other types of visualisation (such as heat maps) to aggregate large portions of the database.

**Personalisation**

The original principle behind PyScholarGraph was the idea that the user could ‘explore the graph’ – using one paper they knew to be relevant in order to find other papers about the same topic. It follows that, much like Google customises search results based on previous searches and user-chosen search results, it would be valuable for the user to provide feedback on which papers returned are truly relevant to their search. This could then be used to provide more relevant results to users that follow them, and also prioritise different papers for the user themselves.

This could be implemented in a number of different ways. However, the following seems particularly in keeping with the graph methodology. First, it seems necessary for the user to provide feedback in some way. In a simple form, this would be a yes/no answer to ‘Was this question relevant?’ From there, storing this data could take the form of properties of the existing relations. However, it is important that the data be stored by session – that is, not only should it be stored by user, but also that user should be able to have multiple ‘lines of investigation’. This would indicate that a more advanced data structure, possibly one that consists of multiple separate graphs, would be appropriate. Given enough user input, the exploration of storing, indexing and visualising these lines of investigation would be promising.

**Conclusion**

This paper presents the early version of PyScholarGraph, a Python-based software package that provides a framework for interesting research in to information retrieval and visualisation from graph data structures. This paper has also presented two algorithms which make use of PyScholarGraph to provide alternate relevance measures, and a system that performs visualisation. However, PyScholarGraph is a work in progress, and while it clears the way for some interesting research questions to be answered, it needs further work before
it could be considered a ‘release’ version. With this in mind, PyScholarGraph will be open-sourced at github.com/fphhotchips/PyScholarGraph, under an Apache 2.0 licence. This will be done so that any researcher with the interest may make changes and improve the PyScholarGraph tool.

Bibliography


‘CiteSeerX data | CiteSeerX’. (n.d.). Retrieved 2014-06-17, from csxstatic.ist.psu.edu/about/data.


This text is taken from The ANU Undergraduate Research Journal, Volume Six, 2014, edited by Jonathon Zapasnik and Alexandra Hogan, published 2015 by ANU eView, The Australian National University, Canberra, Australia.