Evolving Graphs and Similarity-based Graphs with Applications

Weijian, Zhang

2018

MIMS EPrint: **2018.34**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

EVOLVING GRAPHS AND SIMILARITY-BASED GRAPHS WITH APPLICATIONS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2018

Weijian Zhang
School of Mathematics

Contents

Word count 51,170

List of Tables

List of Figures

The University of Manchester

Weijian Zhang
Doctor of Philosophy
Evolving Graphs and Similarity-based Graphs with Applications
October 17, 2018

Abstract

A graph is a mathematical structure for modelling the pairwise relations between objects. This thesis studies two types of graphs, namely, similarity-based graphs and evolving graphs.

We look at ways to traverse an evolving graph. In particular, we examine the influence of temporal information on node centrality. In the process, we develop EvolvingGraphs.jl, a software package for analyzing time-dependent networks.

We develop Etymo, a search system for discovering interesting research papers. Etymo utilizes both similarity-based graphs and evolving graphs to build a knowledge graph of research articles in order to help users to track the development of ideas. We construct content similarity-based graphs using the full text of research papers. And we extract key concepts from research papers and exploit the temporal information in research papers to construct a concepts evolving graph.

Thesis Supervisor: Professor Nicholas J. Higham

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University's Policy on Presentation of Theses.

Publications

1. The material in Part I is based on the papers:

    - Weijian Zhang, Dynamic Network Analysis in Julia. MIMS EPrint, 2015.83, (2015)
    - Jiahao Chen and Weijian Zhang, The Right Way to Search Evolving Graphs. Proceedings of IPDPS 2016.
    - Jiahao Chen and Weijian Zhang, A Closer Look at Time-Preserving Paths on Evolving Graphs. To include more experiments on social networks and to be submitted to KDD 2019.

2. The material in Part II is based on the papers:

    - Weijian Zhang, Jonathan Deakin, Nicholas J. Higham, and Shuaiqiang Wang, Etymo: A New Discovery Engine for AI Research. The 2018 Web Conference Companion, pp. 227-230, April, 2018, Lyon, France. ACM, New York, NY, USA.
    - Weijian Zhang, Jonathan Deakin, Steven Elsworth, Xiaoyi Li, Nicholas J. Higham, and Shuaiqiang Wang, Evolving Knowledge Graphs for Idea Tracking in Research Literature. To be submitted to PeerJ Comput. Sci.

3. The material in Part III is based on the paper:

    - Weijian Zhang and Nicholas J. Higham, Matrix Depot: An Extensible Test Matrix Collection for Julia. PeerJ Comput. Sci., 2:e58 (2016).

Acknowledgements

I am extremely grateful to my supervisor, Prof. Nicholas Higham for his valuable insight, unwavering support, and trust. Without his efforts this thesis would not have been possible.

I thank Prof. Alan Edelman and Dr. Jiahao Chen for arranging for me a fruitful visit to MIT CSAIL, where I had the priviledge to work and learn from the core Julia team.

I am grateful to work with Jonathan Deakin, Xiaoyi Li, and Steven Elsworth on the Etymo project. I also thank Dr. Shuaiqiang Wang and Tak Lo for their insights and support for the project.

I am grateful to Prof. Françoise Tisseur for getting me interested in PageRank and network science.

Finally, I am grateful for Jinan and my parents for their love and support. Without them I would never have made it this far.

Chapter 1

Introduction

The information age is changing all aspects of our lives. First, everything is online. The internet has made information accessible everywhere at any time. Second, social networks have made continuous connectivity widely available. We can connect with people almost anywhere in the world. Third, cloud computing has given everyone practically infinite computing power and storage [71]. In September 2013, the Turing prize winning computer scientist John Hopcroft presented an interesting talk at the Heidelberg Laureates Form on the future of computer science. In the talk, he noted that in the past 30 years computer science was concerned with making computers useful. The future is about how computers are being used. He listed some topics for the future [1]. This thesis is concerned with one particular topic from his list: tracking the flow of ideas in scientific literature.

What is an idea? As we are primarily concerned with scientific literature, we decide to use keywords and a group of related keywords to represent ideas. We focus on two things to study: idea relationship and idea representation.

1. We model the connectivity between ideas as a sequence of time-dependent networks, also known as evolving graphs. We then use evolving graph traversal to study the flow of ideas and evolving graph centrality to study trends. We develop our own software package EvolvingGraphs.jl for analysing time-dependent networks.

2. We use a vector space model to represent ideas as vectors of real numbers. These

[1] See the SIAM news post on the topic list: https://sinews.siam.org/Details-Page/the-future-of-computer-science

vector representations are not static but evolve in time. We build a research paper discovery system, called Etymo [2] that realises our idea.

Link analysis refers to the technique of exploiting the additional information inherent in the Web's hyperlink structure. This simple idea changed the information retrieval scene in 1998 [55]. Search engines began exploiting the information inherent in the hyperlink structure of the Web to improve the quality of search results and web search improved dramatically. Nowadays, link analysis is the norm in internet search: nearly all major search engines now combine link analysis scores, like the one used by Google, with other methods in their system.

The rise of deep learning is transforming all types of business activities and is becoming the foundation of many important applications, including web search, speech recognition, product recommendation. Traditional statistical natural language processing such as Hidden Markov Model (HMM) [72], N-gram, and parsing is replaced by deep learning based methods such as word embedding [16, Chap. 6.1], convolutional neural network (CNN) [16, Chap. 6.2] sequence embedding or recurrent neural network (RNN) [16, Chap. 6.4] attention mechanism. With vector space models such as Word2Vec [62] and Doc2Vec [57] we can represent words and documents as vectors and represent the relationship between these word embeddings as graphs.

We can now more effectively extract the entity relationship and content similarity just from a big text corpus, things like "King" is a "Male" and "Good" and "Fine" are similar words. Linking these words, phrases and documents can give us additional information to assist information retrieval. Recent years have seen a big increase of large-scale knowledge bases, including Wikipedia, Freebase, YAGO, Knowledge vault, Microsoft's Satori, Google's Knowledge Graph. We propose an automatic method for constructing knowledge bases from research articles, focusing primarily on recency and trends. Another novelty is that we use an evolving graph model to model the knowledge base, which enables us to track the development of ideas over time.

In scientific research, a research article contains an author's understanding of a subject. Even though many concepts and definitions are universally agreed, different authors have different interpretations of how concepts are related. In particular, the

[2] At the time of writing this thesis, we've renamed the search engine as Etymo Scholar. See Etymo Scholar https://scholar.etymo.io

keywords in an article tell us what an author thinks is important in a given subject. If we extract keywords from an article and link them in a tree structure, for example, from the more general concepts, things like "data science" or "linear algebra" to more specific concepts like "recursive neural network (RNN)" or "polar decomposition", it becomes a graph that represents an author's understanding of a subject. We could then combine these small personal knowledge graphs to make a collective knowledge graph.

Extracted keywords have two important applications. First, we could give each unique keyword a Uniform Resource Identifier (URI). These keyword URIs allow users to get the exact information that is related to the keyword. In our case, a keyword URI points to a web page that contains all the related papers. This looks a bit like Wikipedia, but we do it automatically and we constantly update keywords and their connections. Suppose each paper contains a list of related keywords. A user can navigate to a new web page of related papers by clicking a keyword. In this way, a researcher can traverse a collection of related articles by ideas. This is much more powerful than current scholar engines such as Google Scholar where a user usually needs to navigate around related articles by following citations. Second, we could use time-dependent network models to study trends in research. One simple approach is to use keywords as nodes, their connections as edges, and published dates as time stamps. We will discuss the construction of Etymo evolving knowledge graph in later chapters.

This thesis is written in the journal format, which means the main content is a collection of published and publishable work. We provide background information in Chapter 2, which covers the key concepts in the vector space model and evolving graph models. We use Julia [7], an open source high performance high level programming language, to illustrate the concepts with simple examples. As Julia's syntax is quite similar to mathematical notation, using Julia increases readability for our readers. The rest of the thesis is structured in two main parts.

Part I is about evolving graph traversal and centrality [38]. We model time-dependent networks as an ordered sequence of networks, each node having a time stamp label. We develop EvolvingGraphs.jl, a Julia software package for studying time-dependent networks. We investigate graph traversal and graph centrality for

evolving graphs.

Part II is about the applications of evolving graphs. We link similar papers to form a similarity-based network. We develop a discovery engine called Etymo (https://etymo.io) for researchers to find interesting research papers and ideas. Etymo constructs and maintains an adaptive similarity-based network of research papers as an all-purpose knowledge graph for ranking, recommendation, and visualisation

A graph can be represented by an adjacency matrix. Let $G = (V, E)$ be a graph with N nodes, where V is a set of nodes and E is a set of edges. Then the (i, j) entry of the $N \times N$ adjacency matrix A is equal to 1 if there is an edge from node v_i to node v_j and is zero otherwise. Many interesting graph algorithms can be expressed using the language of numerical linear algebra. For example, matrix-vector multiplication is equivalent to a step of breadth first search [51].

Part III is about Matrix Depot, a test matrix collection written in Julia. Matrix Depot was developed as a test bed for exploring (the then very immature) Julia prior to using it for network analysis. Test matrices are important for exploring the behaviour of linear algebra algorithms and for measuring their performance with respect to accuracy, stability, convergence rate, speed, or robustness. Matrix Depot takes advantage of many nice Julia features, such as using multiple dispatch to help provide a simple user interface and to allow matrices to be generated in any of the numeric data types supported by the language.

For all the papers presented in this thesis, I played a major role in conceiving of the presented ideas, drafting the papers, carrying out the experiments, and revising drafts of the papers. Here are the specific contributions each author made:

1. "A New Discovery Engine for AI Research". I conceived and designed the system, performed the experiments, analyzed the data, wrote the main part of the paper, prepared figures and/or tables, performed the computation work, reviewed drafts of the paper.

   Jonathan Deakin conceived the designed the system, analyzed the data, wrote parts of the paper, reviewed drafts of the paper.

   Nicholas J. Higham and Shuaiqiang Wang wrote parts of the paper and reviewed drafts of the paper.

2. "Evolving Knowledge Graphs for Idea Tracking in Research Literature". I conceived and designed the system, performed the experiments, analyzed the data, wrote the main part of the paper, prepared figures and/or tables, performed the computation work, reviewed drafts of the paper.

   Jonathan Deakin, Steven Elsworth, and Xiaoyi Li conceived the designed the system, analyzed the data, wrote parts of the paper, and reviewed drafts of the paper.

   Nicholas J. Higham and Shuaiqiang Wang wrote parts of the paper and reviewed drafts of the paper.

3. "Dynamic Network Analysis in Julia". I conceived and designed the software, performed the experiments, analyzed the data, wrote the paper, prepared figures and/or tables, performed the computation work, and reviewed drafts of the paper.

4. "The Right Way to Search Evolving Graphs". I conceived and designed the experiments, performed the experiments, analyzed the data, wrote the main part of the paper, prepared figures and/or tables, performed the computation work, reviewed drafts of the paper.

   Jiahao Chen conceived and designed the experiments, analyzed the data, wrote parts of the paper, performed the computation work, and reviewed drafts of the paper.

5. "A Closer Look at Time-Preserving Paths on Evolving Graphs". I conceived and designed the experiments, performed the experiments, analyzed the data, wrote the main part of the paper, prepared figures and/or tables, performed the computation work, and reviewed drafts of the paper.

   Jiahao Chen conceived the experiments and reviewed drafts of the paper.

6. "Matrix Depot: An Extensible Test Matrix Collection for Julia". I conceived and designed the experiments, performed the experiments, analyzed the data, wrote the main part of the paper, prepared figures and/or tables, performed the computation work, and reviewed drafts of the paper.

Nicholas J. Higham conceived the experiments, analyzed the data, wrote parts of the paper, and reviewed drafts of the paper.

We make the following specific contributions:

1. We develop EvolvingGraphs.jl, the first dynamic network analysis framework in Julia. To the best of our knowledge, it is one of the earliest attempts to build a full-feature package for such models and the first such package in the programming language Julia.

2. We show that naive unfoldings of adjacency matrices miscount the number of temporal paths between two temporal nodes in an evolving graph. By mapping an evolving graph to an adjacency matrix of an equivalent static graph, we prove that our generalisation of the breadth first search algorithm correctly accounts for paths that traverse both space and time.

3. We study the impact of different time-preserving paths on node centrality by varying the edge weights of static and causal edges. We also compare evolving graph centrality with the aggregated static graph case to gain insight about the advantage of an evolving graph model.

4. We develop Etymo (https://etymo.io), a discovery engine to facilitate artificial intelligence research and development. Etymo uses a novel form of search that can quickly identify relevant and important new papers and displays related papers in a graphical interface.

5. We introduce a new search interface, which combines traditional item list and data visualisation, for tracking the development of ideas in scientific literature.

6. We develop a new test matrix collection in Julia called Matrix Depot. Matrix Depot is extensible by the users to include their own test problems. It amalgamates in a single framework two different types of existing matrix collections, comprising parametrized test matrices and real-life sparse matrix data.

Chapter 2

Background

We consider representing words and documents as vectors in a vector space and analysing the relationships between these vectors using time-dependent networks. Mapping from text in a vocabulary to vectors of real numbers in a vector space is known as *word embedding*. We discuss two methods for word embdding: Term Frequency and Inverse Document Frequency (TF-IDF) and Word2Vec. One particular network model we are considering is *evolving graphs*, which represent the relationship between entities as an ordered sequence of networks, each with a time stamp label. This chapter introduces the basic building blocks that we will use in later chapters.

2.1 Vector Space Models

The vector space model transforms documents of text into numeric vectors of real values. The technique was developed by Gerard Salton in the early 1960s [69] and is widely used in information retrieval (IR) problems. Latent Semantic Indexing (LSI) [25], for example, can uncover the hidden semantic structure in a document collection, which is a very powerful tool. However, the vector space model is computationally expense. For IR tasks at query time, we need to compute a similarity score between the query vector with every document vector in the database. LSI requires singular value decomposition (SVD) of a large matrix. The computational cost scales quadratically with the number of documents, which makes it less attractive for a corpus with millions of words or documents.

Here we consider techniques to build a vector representation of word and documents.

2.1.1 Word Embeddings

A word vector is a vector of real numbers. The simplest word vector, probably, is the *one-hot vector*, which is a $|V|$ dimensional vector with all the 0s and one 1 at the index of that word in the vocabulary. Here $|V|$ represents the size of the vocabulary V. For example, suppose our vocabulary has only four words: King, Queen, Man, Woman. Then the word vectors of the four words are

$$w(King) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad w(Queen) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad w(Man) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad w(Woman) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Such a representation, however, does not directly yield any notion of similarity. In particular, the dot product between any two word vectors above is zero. For example,

$$w(King)^T w(Man) = 0.$$

A better word vector representation is *featured representation*. Let our features be female, food, royal. Then we could represent our four words as

$$\begin{array}{c} \\ female \\ food \\ royal \end{array} \begin{array}{c} \begin{array}{cccc} King & Queen & Man & Woman \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Then we can see, for example, King is more similar to Queen than Woman.

$$w(King)^T w(Queen) = 1 > 0 = w(King)^T w(Woman)$$

Recall that for the one-hot vector representation the dimension of a vector is equal to the size of the vocabulary. In featured representation the dimension of a vector is equal to the number of features.

How do we choose features? We could use high frequency words as features such that a corpus of documents can be represented by a matrix with one row per document and one column per feature word (term). We only consider the word occurrences while ignore the relative position information of the words in the document. This term-document matrix is very sparse in general. This specific strategy is called the *bag of words* representation. A different strategy is to use one feature to represent more than one meaning, i.e., distribute the meanings in a document among a list of features. This distributed representation is usually dense and has lower dimension than the bag of words representation. We will consider both of the these strategies below.

2.1.2 Term Frequency and Inverse Document Frequency

Frequency is an important measure of relevance. In a search engine, a document that mentions a query term more often is usually more related to that query. However, the most common words in a language are usually function words, such as "the", "is", "at", "which", and "on". Simply considering frequency can be misleading.

The term frequency of a term t in document d, denoted by tf(t,d), is the number of occurrences of term t in document d. Using Julia with StatsBase.jl [1], we can compute the term frequency of all the terms in a document as

```
>julia using StatsBase
>julia tf(document) = countmap(split(lowercase(document)))
```

For example, the term frequency of each term in a short document $d1$ "The sky is blue and the sun is bright" is

```
>julia tf(d1)
Dict{SubString{String},Int64} with 7 entries:
  "bright" => 1
  "the"    => 2
  "is"     => 2
  "sky"    => 1
  "blue"   => 1
  "sun"    => 1
  "and"    => 1
```

[1] https://github.com/JuliaStats/StatsBase.jl

If a term appears in a lot of documents then it is likely to be a function word. The document frequency of a term t, denoted by df(t), is the number of documents in the collection that contains term t. Let N be the total number of documents in a collection. We define the inverse document frequency idf(t) of a term t, to be

$$\text{idf(t)} = \log \frac{N}{\text{df(t)}}.$$

Let $d2$ be "The birds are singing and it is beautiful day". Then the inverse term frequency of the word "the" in both $d1$ and $d2$ is equal to 0 since the document frequency of "the" is 2 and there are 2 documents in the collection.

In Julia we can write the document frequency as

```
function df(word, doclist)
    word = lowercase(word)
    doc_counter = 0
    for doc in doclist
        if haskey(tf(doc), word)
            doc_counter +=1
        end
    end
    return doc_counter
end
```

Then the inverse term frequency idf(t) can be written as

```
function idf(word, doclist)
  num_doc = length(doclist)
  df_w = df(word, doclist)
  return log(num_doc/ df_w)
end
```

The tf-idf weighting is a combination of term frequency and inverse document frequency. A term with high tf-idf score is a term with high term frequency in a given document and a low document frequency in the whole collection of documents. Hence tf-idf adjusts for the fact that some common words appear very frequently in all document. The tf-idf weighting of a term t in document d, denoted by tf-idf(t,d) is calculated by

$$\text{tf-idf(t,d)} = \text{tf(t,d)} \times \text{idf(t)}.$$

We can use high frequency words as features and represent each document in the collection by the tf-idf weighting of the feature words.

2.1.3 Word2Vec

We consider representing words or documents by low-dimensional real vectors, also known as distributed representations of words or documents. A distributed vector representations of words is not a new idea. In fact, we can trace it back to the 1986 paper by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams on learning representations by back-propagating errors [67]. But a recent approach by Mikolov et al. [62] shows how to reduce the computational complexity of learning such representations, which makes it practical to learn such vectors on a large dataset.

The key insight is that we could represent a word by means of its neighbours [28]. One model Mikolov et al. proposed is called the continuous bag of words model (CBOW). In this model, we predict the centre word (focus word) from its surrounding words. Say, for example, we have a piece of text "I like deep neural networks." The model will be able to predict the word "deep" from the surrounding words "I", "like", "neural", "networks". The training objective is to maximise the conditional probability of observing the actual centre word from given input surrounding words.

In more detail, we first create two matrices $W_{in} \in \mathbb{R}^{n \times |V|}$ and $W_{out} \in \mathbb{R}^{|V| \times n}$, where n is the size of the embedded vectors and $|V|$ is the size of the vocabulary V. We call W_{in} the *input word matrix* and W_{out} the *output word matrix*. We also define v_i and u_i to be the input and output vector representations of the word i respectively.

For the centre word "deep", we first generate the one-hot vectors x_i, x_{like}, x_{neural}, $x_{network}$ for the surrounding words "I", "like", "neural", and "network".

By multiplying the input word matrix with the one-hot vectors, we get the word embeddings of these surrounding words. For example, the word embedding of "neural" is

$$v_{neural} = W_{in} x_{neural}.$$

We then take an average of all these input word embeddings

$$\hat{v} = \frac{v_i + v_{like} + v_{neural} + v_{network}}{4}$$

An output score vector can be generated by multiplying the output word matrix with the average $\hat{v}$

$$u_{deep} = W_{out}\hat{v}.$$

We can convert scores into probabilities using the softmax function

$$\hat{y}_{deep} = \text{softmax}(u_{deep}).$$

The softmax function takes a vector of real-valued scores and converts it to a vector of values between zero and one that sum to one. Here each element of $\hat{y}_{deep}$ represents the probability of a word in the vocabulary to be the centre word. In particular, the element representing the probability of word i is

$$\hat{y}_{deep}(i) = \frac{\exp(u_i^T\hat{v})}{\sum_{j\in vocabulary}\exp(u_j^T\hat{v})},$$

We desire the probabilities $\hat{y}_{deep}$ to match the one hot vector of the word "deep", denoted by y_{deep}. The cross entropy $H(\hat{y}_{deep}, y_{deep})$ between two probability vectors $\hat{y}_{deep}$ and y_{deep} is defined as

$$H(\hat{y}_{deep}, y_{deep}) = -\sum_{j\in vocabulary} y_{deep}(j)\log(\hat{y}_{deep}(j)).$$

Since $y_{deep}(i) = 1$ if and only if $y_{deep}(i)$ is the element represents word "deep" and 0 otherwise, we have

$$H(\hat{y}_{deep}, y_{deep}) = -\log(\hat{y}_{deep}(deep)).$$

Our objective function is to minimize the loss function for all words in the training dataset, given by

$$J = \frac{1}{|T|}\sum_{w\in T} H(\hat{y}_w, y_w) = -\frac{1}{|T|}\sum_{w\in T}\log(\hat{y}_w(w)), \tag{2.1}$$

where

$$\hat{y}_w(w) = \frac{\exp(u_w^T\hat{v})}{\sum_{j\in vocabulary}\exp(u_j^T\hat{v})}$$

and T represents the training data set. To solve the optimisation problem, we use gradient descent to update all word vectors in equation (2.1).

Another approach is to create a model to predict surrounding words I", "like", "neural", and "network" from the given centre word "deep". This is known as a Skip-Gram model.

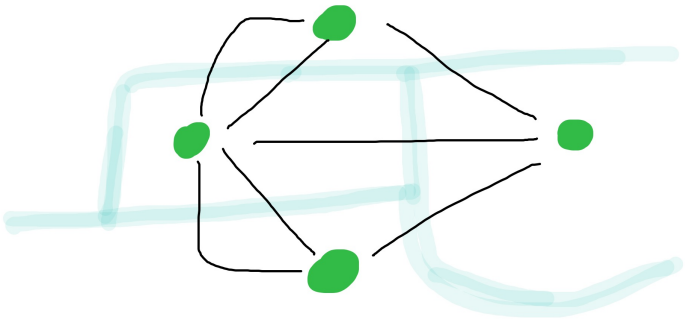


Figure 2.1: A graphical representation of the Königsberg bridge problem.

2.2 Euler and Graphs

The story of graphs starts with Leonhard Euler and steven bridges of Königsberg problem. The city of Königsberg was connected by seven bridges. The problem was to devise a walk through the city that can cross each of the bridges once and only once. Figure 2.2 shows a graphical representation of the bridges. Euler's 1736 paper proved that the problem has no solution. The insight is to reformulate the network of bridges in Königsberg as dots connected by lines, since only the connection information is relevant to the problem: it does not matter whether the lines are straight or curve or the dots are big or small. In modern language, we call such model a *graph*, where the dots are called *nodes* and the lines connecting them are called *edges*.

Euler realised that the possibility of a walk on a graph depends on the *degrees* of the nodes. The degree of a node is equal to the number of edges connecting it. Euler showed that a necessary condition for the walk stated in the problem is that the graph has exactly zero or two nodes of odd degree.

Graphs turn out to have many more applications beyond solving the steven bridges of Königsberg problem. For example, we could represent the relationship between Machine Learning and three branches of machine learning: Supervised Learning, Unsupervised Learning, and Reinforcement Learning as a directed graph shown in Figure 3.5.

Formally, a graph [11] is tuple $G = (V, E)$, where V is the set of nodes in G and

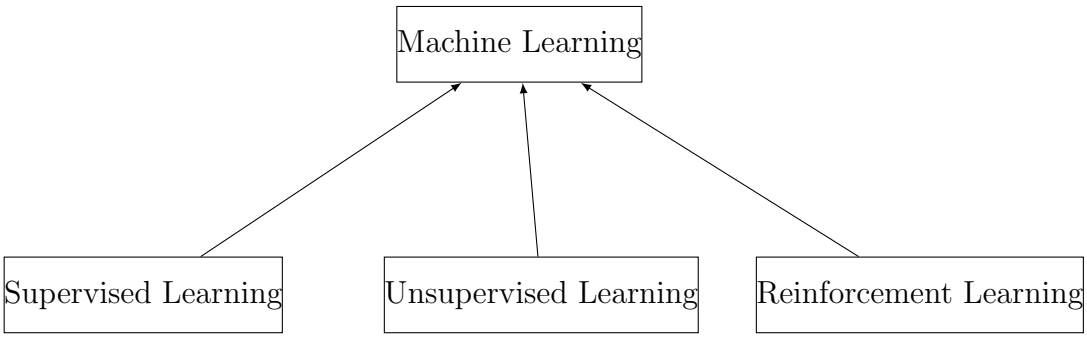


Figure 2.2: A simple graph with 4 boxes representing nodes, and 3 arrows representing edges.

$E \subseteq V \times V$ is the set of edges. Each edge consists of a pair of nodes. We say two nodes are *adjacent* to each other when there is an edge between them. If the edges have a direction associated with them, we call this graph a *directed graph*. In Figure 3.5, the node set V is { "Machine Learning", "Supervised Learning", "Unsupervised Learning", "Reinforcement Learning" } and the edge set E is {("Supervised Learning", "Machine Learning"), ("Unsupervised Learning", "Machine Learning"), ("Reinforcement Learning", "Machine Learning")}. "Machine Learning" and "Supervised Learning" are adjacent.

2.2.1 Evolving Graphs

Much real-world relationship data stores time stamps with the interactions. Let's consider a group of users interacting through messaging. We could represent each message sent from user v_i to user v_j at time stamp t by an temporal edge from node v_i to node v_j at time stamp t, denoted by (v_i, v_j, t). We could therefore represent the user interaction network as an ordered sequence of (static) networks, each with a time stamp label. For example, Figure 5.4 represents an evolving graph with three time stamps t_1, t_2, and t_3.

In Julia using our package EvolvingGraphs.jl [2], we could generate the evolving graph in Figure 5.4 as

```
julia> using EvolvingGraphs

julia> g = EvolvingGraph{Node{String}, Int}()
Directed EvolvingGraph 0 nodes, 0 static edges, 0 timestamps
```

[2] https://github.com/EtymoIO/EvolvingGraphs.jl

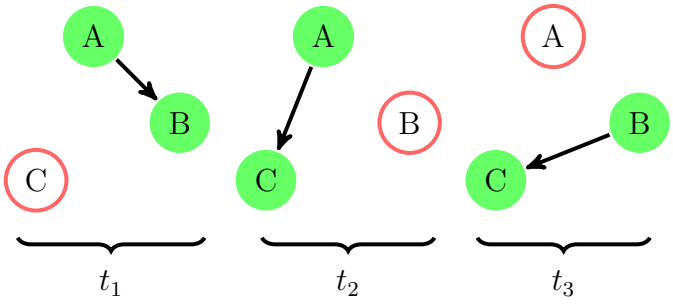


Figure 2.3: A simple evolving directed graph with 3 time stamps t_1, t_2 and t_3. At each time stamp, the evolving graph is represented as a graph. The green filled circles represent active nodes while the red circles represent inactive nodes. Directed edges in each time stamp are shown as black arrows.

```
julia> add_edge!(g, "A", "B", 1)
Node(A)−1.0−>Node(B) at time 1

julia> add_edge!(g, "A", "C", 2)
Node(A)−1.0−>Node(C) at time 2

julia> add_edge!(g, "B", "C", 3)
Node(B)−1.0−>Node(C) at time 3
```

Here we store an evolving graph as a sequence of temporal edges. We can access the temporal edges using the edges function

```
julia> edges(g)
3−element Array{EvolvingGraphs.WeightedTimeEdge{...}:
 Node(A)−1.0−>Node(B) at time 1
 Node(A)−1.0−>Node(C) at time 2
 Node(B)−1.0−>Node(C) at time 3
```

2.2.2 Evolving Graph Centrality

The degree of a node is the number of edges connected to it. If the graph is directed, we have two kinds of degrees: in-degree, which counts the number of incoming edges and out-degree, which counts the number of outgoing edges. This simple measure of node importance is called *degree centrality*. In essence, a node is important if it has many neighbours. In general, centrality algorithms answers the question "what

characterises an important node?".

Traditional graph centralities can be generalised for evolving graphs. Indeed, one approach is to use the fact that the elements of the matrix product of adjacency matrices at different time stamps can correctly count the number of temporal paths [38]. A different approach is to first generalise paths for evolving graphs and then replace the definition of paths in graph centrality with time-respecting paths [64].

Let $G_3 = \langle G^{[1]}, G^{[2]}, G^{[3]} \rangle$ be a directed evolving graph such as that in Figure 5.4. Let $A^{[i]}$ be the adjacency matrix representation of graph $G^{[i]}$. The Katz centrality on evolving graphs can be derived by considering a sequence of matrix products

$$Q = (I - \alpha A^{[1]})^{-1}(I - \alpha A^{[2]})^{-1}(I - \alpha A^{[3]})^{-1}. \tag{2.2}$$

The ith row and column sums

$$C_i^{broadcast} = \sum_{k=1}^{n} Q_{ik}, \quad C_i^{receive} = \sum_{k=1}^{n} Q_{ki}$$

are the broadcast centrality and receive centrality of node i. Here is the implementation of Katz centrality in EvolvingGraphs.jl.

```
function katz(g::AbstractEvolvingGraph, alpha::Real = 0.3)
    n = num_nodes(g)
    ns = nodes(g)
    ts = timestamps(g)
    v = ones(Float64, n)
    A = spzeros(Float64, n, n)
    spI = speye(Float64, n)
    for t in ts
        A = sparse_adjacency_matrix(g,t)
        v = (spI - alpha*A)\v
        v = v/norm(v)
    end
    return [(node, v[node.index]) for node in ns]
end
```

Betweenness centrality is a measure of centrality based on shortest paths. The betweenness centrality for each node is the number of shortest paths that pass through the node. We could extend betweenness centrality on evolving graphs by replacing

shortest paths with temporal shortest paths as follows:

$$C_i^{betweenness} = \sum_{j \in V} \sum_{k \in V, k \neq j} \frac{\alpha_{jk}(i)}{\alpha_{jk}}, \tag{2.3}$$

where α_{jk} is the number of temporal shortest paths from node j to node k and $\alpha_{jk}(i)$ is the number of temporal shortest paths from node j to node k that pass through the node i.

Part I

Evolving Graph Traversal and Centrality

Chapter 3

Dynamic Network Analysis in Julia

3.1 Introduction

We describe EvolvingGraphs [1], a Julia software package for analyzing dynamic networks. A dynamic network is network in which the interactions among a set of elements change over time. Examples of dynamic networks include a network of mobile phone users interacting through messaging and the spread of diseases in communities. It is natural to model a dynamic network by an evolving graph G, defined as a sequence of static graphs $\{G_1, G_2, \ldots, G_n\}$, where $G_i = (V(i), E(i))$ is a snapshot of the evolving graph G at timestamp i. For example, Figure 4.1 shows an evolving graph with 3 timestamps t_1, t_2, t_3, where a green shaded circle denotes an active node (see Section 3.2) and a pink circle denotes an inactive node.

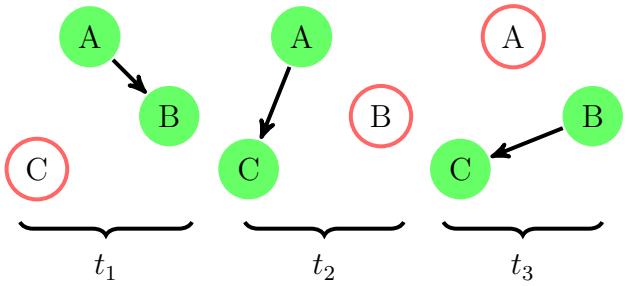


Figure 3.1: An evolving graph with 3 timestamps.

[1] This paper describes EvolvingGraphs.jl v0.1.0 and was tested on Julia 0.4. https://github.com/EtymoIO/EvolvingGraphs.jl/tree/v0.1.0. The latest version is v0.2.0. See the documentation at: https://etymoio.github.io/EvolvingGraphs.jl/latest/

Julia [7] is a high-level, high-performance dynamic programming language for technical computing. It takes advantage of LLVM-based [56] just-in-time (JIT) compilation to approach the performance of statically-complied languages like C, yet allows programmers to write clear, high-level code that closely resembles mathematical formulas. EvolvingGraphs makes particular use of Julia's multiple dispatch combined with its type system. Since user defined types are first class in Julia, it makes sense to implement new graph types to design user-friendly interfaces.

EvolvingGraphs is designed to have a similar interface to standard static graph packages such as Python's NetworkX[2] or Julia's Graphs [3]. To get a taste of EvolvingGraphs, we may consider representing the evolving graph of Figure 4.1:

```
> g = evolving_graph(Char, String)
> add_edge!(g, 'A', 'B', "t1")
> add_edge!(g, 'A', 'C', "t2")
> add_edge!(g, 'B', 'C', "t3")
> nodes(g)
3-element Array{Char,1}:
 'A'
 'B'
 'C'
> edges(g)
3-element Array{TimeEdge{V,T},1}:
 TimeEdge(A->B) at time t1
 TimeEdge(A->C) at time t2
 TimeEdge(B->C) at time t3
```

where the arguments of evolving_graph indicate the node type and timestamp type respectively.

This paper is organized as follows. In Section 3.2, we introduce a node-active model for evolving graphs, including the definition of temporal path and temporal distance. In Section 3.3, we consider the type system of EvolvingGraphs and discuss ways to represent evolving graphs. The concept of connected components in evolving graphs is introduced in Section 3.4. A breadth first search (BFS) based implementation for finding weakly connected components is also discussed. We consider a generalization

[2] https://networkx.github.io/
[3] http://graphsjl-docs.readthedocs.org/

of the Katz centrality in Section 3.5 and provide examples of usage in Section 3.6.

3.2 Node-Active Model

We assume $G = \{G_1, G_2, \ldots, G_n\}$ is a directed evolving graph, where each edge e is of the form (v_i, v_j, t_k), indicating a link from node v_i to node v_j at timestamp t_k. Unlike in most existing models [35] [38] [64] [76], in which the node set is assumed to be fixed throughout time, in our model (a) nodes are time-dependent and are changing over time; (b) nodes are present only if they are connected by edges. We say a node v at timestamp t, denoted by (v, t), is *active* if v is connected to at least one other node at timestamp t. For example, in Figure 4.1 the following nodes are active: $(A, t_1), (B, t_1), (A, t_2), (C, t_2), (B, t_3), (C, t_3)$. We disregard the inactive nodes when we study and analyze evolving graphs.

Definition 3.2.1 (Temporal path)**.** *We define a* temporal path $p((v_i, t_1), (v_j, t_n))$ *between active nodes* (v_i, t_1) *and* (v_j, t_n) *to be an ordered sequence of active nodes (without repetition)* $\{(v_i, t_1), (v_{i+1}, t_2), \ldots, (v_j, t_n)\}$ *such that* $t_1 \leq t_2 \leq \cdots \leq t_n$ *and* $((v_h, t_k), (v_{h+1}, t_{k+1}))$ *is an edge at timestamp* t_k *if* $t_k = t_{k+1}$ *otherwise we have* $v_h = v_{h+1}$*. A* shortest temporal path *is a temporal path with the least number of unique nodes.*

For example, there are two temporal paths from (A, t_1) to (C, t_3) in Figure 4.1:

1. $(A, t_1) \to (A, t_2) \to (C, t_2) \to (C, t_3)$
2. $(A, t_1) \to (B, t_1) \to (B, t_3) \to (C, t_3)$

The first one is the shortest temporal path since it passed 2 nodes A and C while the second one passed 3 nodes.

Definition 3.2.2 (Temporal distance)**.** *We define the* spatial length *of a temporal path* $p((v_i, t_1), (v_j, t_n))$ *to be the number of unique nodes in* p*. The* shortest temporal distance $d((v_i, t_1), (v_j, t_n))$ *between* (v_i, t_1) *and* (v_j, t_n) *is the spatial length of the shortest temporal path.*

For example, the shortest temporal distance between (A, t_1) and (C, t_3) in Figure 4.1 is 2.

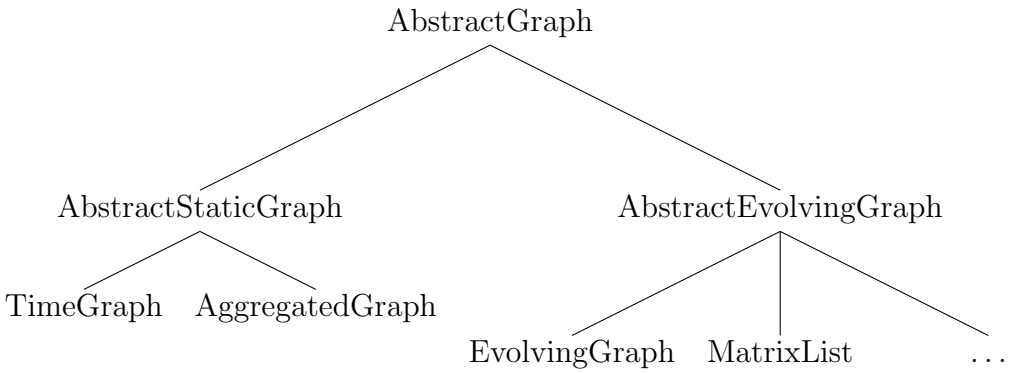


Figure 3.2: Graph type hierarchy.

Definition 3.2.3 (Temporal connectedness)**.** *If there exists a temporal path from node* (v_i, t_m) *to node* (v_j, t_n)*, we say* (v_i, t_m) *and* (v_j, t_n) *are* temporally connected*. We say node* v_i *and* v_j *are temporally connected if* (v_i, t_m) *and* (v_j, t_n) *are temporally connected for some timestamps* t_m, t_n*.*

3.3 Representing Evolving Graphs

The graph type hierarchy in EvolvingGraphs is shown in Figure 3.2.

The root of all graph types is AbstractGraph. It has two children: AbstractStaticGraph (the abstract type of all static graphs) and AbstractEvolvingGraph (the abstract type of all evolving graphs). For both static and evolving graphs, we focus on directed graphs and model undirected graphs using directed graphs with twice as many edges. There are two kinds of static graphs in EvolvingGraphs: TimeGraph represents an evolving graph at a specified timestamp; AggregatedGraph represents a static graph constructed by aggregating an evolving graph, i.e., all the edges between each pair of nodes are flattened in a single edge. For example, the aggregated graph of Figure 4.1 is shown in Figure 3.3.

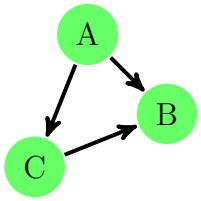


Figure 3.3: The aggregated graph of Figure 4.1.

All static graphs in EvolvignGraphs are represented by adjacency lists. For evolving

graphs, we consider three types: EvolvingGraph, MatrixList and IntEvolvingGraph, which are represented by (a) edge lists, (b) adjacency matrices (c) a mixture of adjacency lists and edge lists respectively. Other evolving graph types are variants of one of the three types.

The edge lists representation of G is specified by

1. the number of nodes n;
2. the list of edges in G, given as a sequence of ordered tuples (v_i, v_j, t_n), which represents an edge from v_i to v_j at timestamp t_n.

The evolving graph of Figure 4.1 can be represented as follows.

1. $n = 3$;
2. $(A, B, t_1), (A, C, t_2), (B, C, t_3)$.

We can also represent an evolving graph by a list of adjacency matrices $\{A_1, A_2, \ldots, A_m\}$. Let $V = \cup_i V(i)$ be the union of all the node sets $V(i)$. Then each A_k is a $|V| \times |V|$ matrix where the (i, j) entry is equal to 1 if and only if there exists an edge from the ith element of V to the jth element of V at timestamp k, and 0 otherwise. For the evolving graph of Figure 4.1, we have $V = \{A, B, C\}$ and

$$A_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, A_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

Finally, we can represent an evolving graph G as a mixture of adjacency lists and edge lists:

1. the number of edges e;
2. the list of active nodes, given as a sequence of ordered pairs (v_i, t_n), which represents an active node v_i at timestamp t_n;
3. the list of timestamps;
4. m lists $E_1, E_2, \ldots, E_m$, where E_i contains all the edges at timestamp i;
5. the lists of out-neighbours of each active node.

Recall that in a static graph the out-neighbours of a node v are all the nodes pointed by node v. For evolving graphs, we define the out-neighbours of a node v at time t to be the set of all active nodes pointed by node v at time t and the active node v itself at time t_i, where $t_i > t$. For example, the out-neighbours of (A, t_1) in Figure 4.1 are (B, t_1), (A, t_2). The evolving graph of Figure 4.1 may be represented as follows:

1. $e = 3$;
2. $(A, t_1), (B, t_1), (A, t_2), (C, t_2), (B, t_3), (C, t_3)$;
3. t_1, t_2, t_3;
4. $E_1 : (A, B, t_1); E_2 : (A, C, t_2); E_3 : (B, C, t_3)$;
5. $(A, t_1) : (B, t_1), (A, t_2)$;
   $(B, t_1) : (B, t_3)$;
   $(A, t_2) : (C, t_2)$;
   $(C, t_2) : (C, t_3)$;
   $(B, t_3) : (C, t_3)$;
   $(C, t_3) : \varnothing$.

Basic graph functions, like checking directedness (is_directed) or finding out-neighours (out_neighbors) are defined for both static and evolving graphs. The implementations of these functions are dispatched based on the type of graph.

3.4 Components

In this section, we discuss an algorithm for computing weakly connected components in EvolvingGraphs. We start by introducting the notion of weak connectedness.

Definition 3.4.1 (Weak connectedness)**.** *We say two nodes (v_i, t_m) and (v_j, t_n) are* weakly connected *if information can flow from (v_i, t_m) to (v_j, t_n), i.e., (v_i, t_m) and (v_j, t_n) are temporally connected.*

Note that our notion of connectedness is reflexive and transitive but not symmetric. The order of time breaks the symmetry. In fact, (v_i, t_m) and (v_j, t_n) are weakly connected only if $t_m \leq t_n$. We can think of an edge from v_i to v_j at timestamp t_n

as information flows from v_i to v_j at timestamp t_n. In modern technology communication, a message can be received immediately after it is sent. Thus we assume that the information flow duration is 0, which is different from models like that in [49]. Given an active node (v, t), we form the information passing tree by collecting all the temporal paths that start at (v, t).

Definition 3.4.2 (Information source)**.** *An* information source *is a root (an active node) of the information passing tree. We say an information source* (v, t) *is a* global information source *if* (v, t) *is an information source and no node is temporally connected to* (v, t)*.*

For example, (A, t_1) is a global information source in Figure 4.1. Using the notion of weak connectedness and information source, we define the *weakly connected components* of an evolving graph G as follows.

Definition 3.4.3 (Weakly connected components)**.** *A* weakly connected component *associated with a node* (v_j, t_n) *is the set of nodes in* G *which are weakly connected by the same information source as* (v_j, t_n)*.*

A node v_i in an evolving graph can belong to multiple components but the partition is unique. For example, suppose at timestamp t_1, A passed a message to B and C passed a message to D. At timestamp t_2, B passed a message to D (see Figure 3.4). Then there are two information sources in this evolving graph: (A, t_1) and (C, t_1) and the weakly connected components are:

- $(A, t_1), (B, t_1), (B, t_2), (D, t_2)$;
- $(C, t_1), (D, t_1), (D, t_2)$.

Note (D, t_2) belongs to both components.

To explore an evolving graph, we need to pay attention to the order of time. In particular, we can not go to a node at a previous timestamp. Recall that the out-neighbours of a node (v, t) are the active nodes pointed by node v at timestamp t and the active node v itself at timestamp t_i, where $t_i \leq t$. Using this notion of out-neighbours, we can extend the BFS algorithm for the case of evolving graphs. Here is the BFS algorithm in EvolvingGraphs:

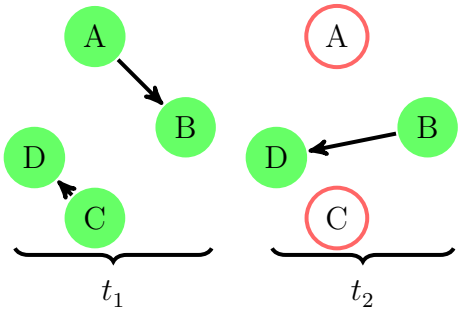


Figure 3.4: An evolving graph with 2 time windows.

```
function breath_first_visit(
    g::AbstractEvolvingGraph, s::Tuple)
    level = Dict(s => 0)
    i = 1
    fronter = [s]
    reachable = [s]
    while length(fronter) > 0
        next = Tuple[]
        for u in fronter
            for v in out_neighbors(g, u)
                if !(v in keys(level))
                    level[v] = i
                    push!(reachable, v)
                    push!(next, v)
                end
            end
        end
        fronter = next
        i += 1
    end
    reachable
end
```

This algorithm finds all the reachable nodes from a given starting node (s, t_1). Let t_1 be the earliest timestamp of G and let $\tilde{V}$ be the set of all the active nodes of G and $E = \cup_t E(t)$ be the set of all edges of G. Since the algorithm explores the

out-neighbours of each (reachable) active node, the computational cost is

$$O\Big(\sum_{(v,t)\in\tilde{V}} Adj[(v,t)]\Big) = O(|E|).$$

To determine the weakly connected components of G, we need to find all the global information sources and then use BFS to find all the reachable nodes from these global information sources. We may use the function weakly_connected_components(g) in EvolvingGraphs to discover the weakly connected components of an evolving graph g. Let $\bar{V}$ be the set of global information sources. The computational cost for finding the weakly connected components is $O(|\bar{V}||E|+|\bar{V}|^2/2)$, where calling BFS has complexity $O(|\bar{V}||E|)$ and checking connected components has complexity $O(|\bar{V}|^2/2)$.

3.5 Katz Centrality

Let A be an adjacency matrix of a static graph G. The Katz centrality rating [50] of a node i is the ith row sum of $(I-\alpha A)^{-1}$, where $\alpha < 1/\rho(A)$, the spectral radius of A. The Katz centrality vector r can be computed by solving

$$(I-\alpha A)r = \mathbf{1},$$

where $\mathbf{1}$ is a vector of ones. It follows from the analysis on static networks that the centrality matrix C_n of an evolving network [38] can be formulated as

$$C_n = (I-\alpha A_1)^{-1}(I-\alpha A_2)^{-1}\cdots(I-\alpha A_n)^{-1}, \tag{3.1}$$

where $\{A_1, A_2, \ldots, A_n\}$ are the corresponding adjacency matrix representations of the evolving graph $G = \{G_1, G_2, \ldots, G_n\}$ and $\alpha < 1/\max_k \rho(A_k)$. The (i,j) entry of the matrix C_n gives a weighted count of the number of dynamic walks from node i to node j. The broadcast and receive communicability vectors are

$$C_n\mathbf{1} \quad \text{and} \quad C_n^T\mathbf{1},$$

respectively. We can compute the broadcast vector using the following algorithm in Julia:

```
function katz_centrality(
g::AbstractEvolvingGraph, alpha::Real)
```

```
    n = num_nodes(g)
    ts = timestamps(g)
    v = ones(Float64, n)
    A = spzeros(Float64, n, n)
    spI = speye(Float64, n)
    for t in ts
        A = spmatrix(g,t)
        v = (spI - alpha*A)\v
        v = v/norm(v)
    end
    return v
end
```

A short list of graph functions implemented in EvolvingGraphs is shown in Table 3.1. By considering walks that started recently as more important than walks that started a long time age [37], Grindrod and Higham introduce a time-dependent factor $e^{-\beta\Delta t_n}$, $\Delta t_n = t_n - t_0$. A variant of (3.1) is given as

$$S_n = (I + e^{-\beta\Delta t_n} S_{n-1})(I - \alpha A_n)^{-1} - I, \quad n = 1, 2, \ldots, \tag{3.2}$$

where $S_0 = 0$, the zero matrix. To see how these two formulae are related, we write $(I - \alpha A_n)^{-1}$ as $(I + \alpha A_n + \alpha^2 A_n^2 + \cdots)$ and expand the right-hand side of (3.2). The function katz_centrality in EvolvingGraphs has more input options than the above implementation. In particular, we can specify the following parameters:

- α: a real-valued scalar which controls the influence of long distance walks;
- β: a real-valued scalar which controls the influence of old walks;
- – mode = :broadcast (by default) generates the broadcast centrality vector;
  – mode = :receive generates the receiving centrality vector;
  – mode = :matrix generates the communicability matrix.

3.6 Examples of Use Cases

Suppose we model a network of online social users interacting through messaging by the evolving graph of Figure 3.5. Each node i represents a user in the network and an

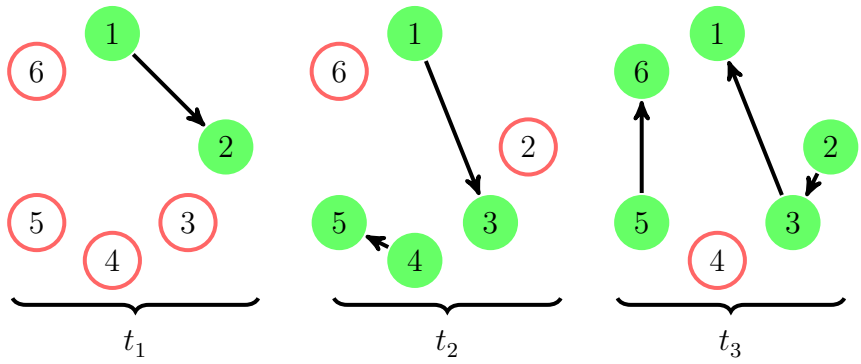


Figure 3.5: An evolving graph with 3 time windows.

edge from node i to node j at timestamp t represents user i sent a message to user j during timestamp t and $t+1$. Let us first generate this evolving graph:

```
> g = evolving_graph(Int, String);
> add_edge!(g, 1, 2, "t1");
> add_edge!(g, 1, 3, "t2");
> add_edge!(g, 4, 5, "t2");
> add_edge!(g, 2, 3, "t3");
> add_edge!(g, 3, 1, "t3");
> add_edge!(g, 5, 6, "t3");
> g
Directed EvolvingGraph
(6 nodes, 6 edges, 3 timestamps)
```

Now g is an evolving graph with 6 nodes, 6 edges and 3 timestamps. We can use the functions nodes, edges and timestamps to have a quick check to see if we have generated the evolving graph correctly:

```
> nodes(g)
6-element Array{Int64,1}:
 1
 4
 2
 3
 5
 6
> edges(g)
6-element Array{TimeEdge{V,T},1}:
 TimeEdge(1->2) at time t1
```

```
 TimeEdge(1->3) at time t2
 TimeEdge(4->5) at time t2
 TimeEdge(2->3) at time t3
 TimeEdge(3->1) at time t3
 TimeEdge(5->6) at time t3
> timestamps(g)
3-element Array{String,1}:
 "t1"
 "t2"
 "t3"
```

We use the function weak_connected to find out if two users "talked" to each other between timestamp t_1 and t_3.

```
> weak_connected(g, 1, 3)
true
> weak_connected(g, 1, 5)
false
```

So user 1 talked to user 3 but not to user 5. We can use the function weak_connected_components to detect small communities in the network.

```
> weak_connected_components(g)
2-element Array{Array{Tuple,1},1}:
 Tuple[(1,"t1"),(2,"t1"),(1,"t2"),
 (1,"t3"),(2,"t3"),(3,"t2"),(3,"t3")]

 Tuple[(4,"t2"),(5,"t2"),(5,"t3"),
 (6,"t3")]
```

We see users $1, 2, 3$ form a small community and users $4, 5$ form another small community. At each timestamp, g may be represented as an adjacency matrix:

```
> int(matrix(g, "t2"))
6x6 Array{Int64,2}:
 0  0  0  1  0  0
 0  0  0  0  1  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
```

We can use katz_centrality to find out the "important" users in a network. Here users 1 and 4 are the most important users in terms of broadcasting information and users 2 and 3 are the most important users in terms of acting as information receivers.

```
> katz_centrality(g, 0.2, 0.3,
mode =:broadcast)
 (6,0.0)
 (2,0.27241247410068437)
 (3,0.27241247410068437)
 (5,0.27241247410068437)
 (4,0.32591575357149416)
 (1,1.5259157535714944)
> katz_centrality(g, 0.2, 0.3,
mode =:receive)
 (4,0.0)
 (5,0.2715964613095785)
 (1,0.32673176636260004)
 (6,0.32673176636260004)
 (3,0.7440089354102629)
 (2,1.0)
```

3.7 Conclusion

The software package EvolvingGraphs is written in Julia, a new dynamic programming language for technical computing. We discussed a node-active model for evolving graph and showed a few algorithms implemented in EvolvingGraphs. EvolvingGraphs currently performs all computations serially. In the future, we will design and implement parallel graph algorithms in EvolvingGraphs with the aim of handling extremely large scale dynamic network problems. EvolvingGraphs.jl is used for modelling evolving graphs in later chapters. At the time of writing this thesis, EvolvingGraphs.jl has 29 stars on GitHub.

Function name	Description
is_directed(g)	returns true if g is a directed graph and false otherwise.
undirected!(g)	converts a directed graph to an undirected graph.
num_nodes(g)	returns the number of nodes in g.
nodes(g)	returns a list of nodes of g.
edges(g)	returns a list of edges of g.
timestamps(g)	returns a list of timestamps of g, in ascending order.
add_edge!(g, v1, v2, t)	adds an edge from v1 to v2 at timestamp t to g.
add_graph!(g, tg)	adds a time graph tg to g.
out_neighors(g, (v,t))	returns the out-neighbours of node (v,t) in g.
aggregated_graph(g)	converts g to an aggregated graph.
issorted(g)	returns true if the timestamps of g are sorted and false otherwise.
sorttime!(g)	sorts g so that the timestamps of g are in ascending order.
slice!(g, t_min, t_max)	slices g between the timestamp t_min and t_max.
matrix(g, t)	generates an adjacency matrix representation of g at timestamp t.
spmatrix(g, t)	generates a sparse adjacency matrix representation of g at timestamp t.
matrix_list(g)	converts g to a list of adjacency matrices represented by MatrixList.
shortest_temporal_path(g, (v1, t1), (v2, t2))	finds the shortest temporal path from (v1, t1) to (v2, t2).
temporal_connected(g, v1, v2)	returns true if v1 are v2 are temporally connected and false otherwise.
weak_connected_components(g)	returns the weakly connected components of g.

Table 3.1: Examples of graph functions implemented in EvolvingGraphs, where g is an evolving graph.

Chapter 4

The Right Way to Search Evolving Graphs

4.1 Introduction

Let's imagine a game played by three people, numbered 1, 2, and 3, each of whom has a message, labeled a, b, and c respectively. At each turn, one particular player is allowed to talk to one other player, who must in turn convey all the messages in his or her possession. The goal of the game is to collect all the messages. Suppose 1 talks to 2 first, and 2 in turn talks to 3. Then, 3 can collect all the messages even without talking to 1 directly. However, if 2 talks to 3 before 1 talks to 2, then 3 can never get a.

We can analyze the spread of information between the players using graph theory. In this process, the time ordering of events matters, and hence its graph representation $G(t) = (V(t), E(t))$ must be time dependent. Such a graph is called an "evolving graph" [29, 3], "evolving network" [13] or "temporal graph" [76].

Treatments of evolving graphs vary in their generality and focus. Kivelä *et al.* [52] treat time dependence as a special case of families of graphs with multiple interrelationships. Others like Flajolet *et al.* [29] use time to index the family of related graphs, but are not concerned with explicit time-dependent processes. Yet others focus on incremental updates to large graphs [3]. Here, we describe evolving graphs as a time-ordered sequence of graphs, similar to the study of metrized graphs by Tang and coworkers [64, 76, 78, 77] and of community dynamics by Grindrod, Higham and

coworkers [38, 37].

The game described above can be encoded in an evolving graph. The spread of information to the winner can be described in terms of traversing this graph using discrete paths that step in both space and time. Traversals of an ordinary (static) graph may be computed using well known methods such as the breadth-first search (BFS). An informal description of BFS generalized to evolving graphs can be found in [76]. However, it turns out that naïve extensions can lead to incorrect descriptions of the resulting graph traversals by accounting for traversals of edges in space, but not necessarily in time. A proper treatment requires the notions of *node activeness* to describe the set of paths that can only traverse time or edges, which we call *temporal paths*, as well as *causal edges* which connect active nodes with the same identity across different times. As a result, our treatment can be applied to any evolving graph, even those that are highly dynamic with arbitrary changes to the nodes and edges.

It is well known that sparse matrix-vector product is equivalent to a one-step BFS on the corresponding (static) graph [51, Sec. 1.1]. In this paper, we demonstrate a correct corresponding result for an evolving graph by constructing a block triangular matrix representation of the graph that takes into account both static and causal connections between active nodes. In Section 4.2, we explain how the BFS algorithm can be applied to an evolving graph to enumerate paths that traverse edges across time and space. Section 4.2.1 provides an example showing that considering only products of the time-dependent adjacency matrices fails to enumerate certain temporal paths. We present and demonstrate the BFS algorithm over evolving graphs in Section 4.2.2, showing its formal equivalence to BFS over a particular static graph generated by adding causal edges that connect active nodes. This static graph generates an algebraic representation of the BFS as power iteration of its adjacency matrix to a starting search node, as shown in Section 4.3.4. The algebraic formulation also demonstrates interesting connections between properties of the BFS algorithm and the adjacency matrix. We describe in Section 4.4 an implementation of the algorithm in Julia. Finally in Section 4.5, we explain how BFS on evolving graphs may be applied to study dynamical processes over citation networks.

4.2 Breadth-First Search over Evolving Graphs

4.2.1 Temporal Paths over Active Nodes

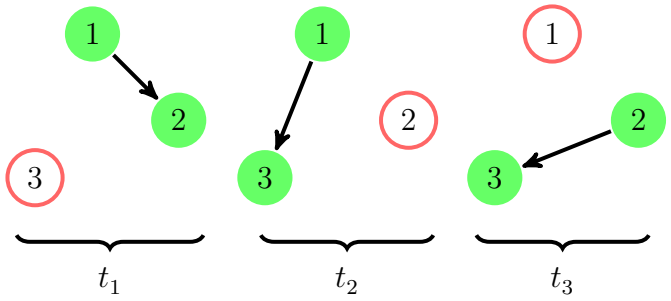


Figure 4.1: An evolving directed graph with 3 time stamps t_1, t_2 and t_3. At each time stamp, the evolving graph is represented as a graph. The green filled circles represent active nodes while the red circles represent inactive nodes. Directed edges are shown as black arrows.

The key new idea in generalizing BFS to evolving graphs is to be able to compute paths that evolve forward in time and can only traverse the node space along existing edges. We call these paths *temporal paths*.

Figure 4.1 shows a small example of an evolving directed graph, $G_3 = \langle G^{[1]}, G^{[2]}, G^{[3]} \rangle$, consisting of a sequence of three graphs $G^{[i]}$, each bearing a time stamp t_i. There are directed edges $1 \to 2$ at time t_1, $1 \to 3$ at time t_2, and $2 \to 3$ at time t_3. Each edge exists only at a particular discrete time and the nodes connected by edges are considered *active* at that time.

Temporal paths connect only active nodes in ways that respect time ordering. Thus the sequences $\langle (1, t_1), (1, t_2), (3, t_2), (3, t_3) \rangle$ and $\langle (1, t_1), (2, t_1), (2, t_3), (3, t_3) \rangle$ are both examples of *temporal paths* from $(1, t_1)$ to $(3, t_3)$, which are drawn as dotted lines with arrowheads in Figure 4.2. However, $\langle (1, t_1), (1, t_2), (2, t_2), (3, t_2), (3, t_3) \rangle$ is not a temporal path because node 2 is inactive at time t_2.

The restriction that temporal paths may only traverse active nodes reflects underlying causal structure in many real world applications, such as analyzing the influence of nodes over social networks. We will also show later in Section 4.3.1 that the resulting structure of allowable temporal paths leads to nontrivial subtleties in the generalization of algorithms and concepts from ordinary (static) graphs.

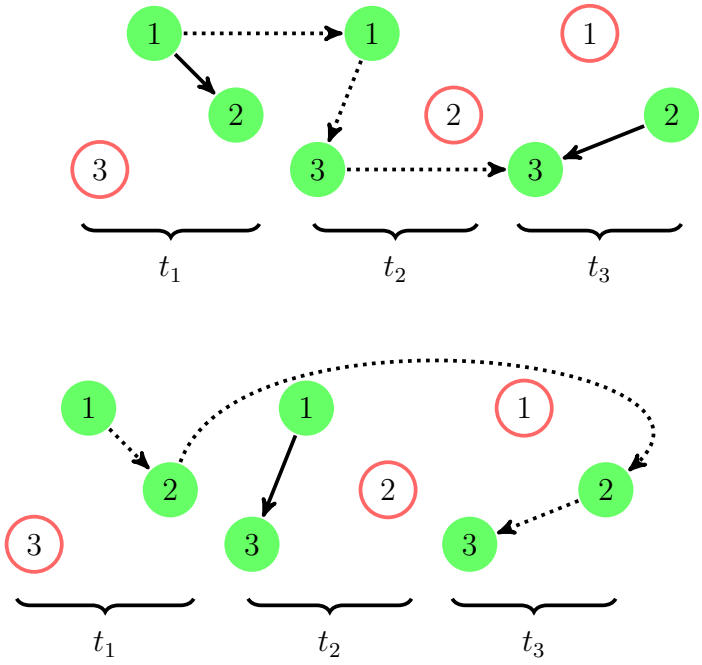


Figure 4.2: The two temporal paths of length 4 from $(1, t_1)$ to $(3, t_3)$ on the evolving graph shown in Figure 4.1, shown in black dashed lines. The paths traverse only active nodes along edges, and are allowed to advance between the same node if it is active at different times.

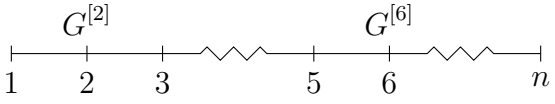
4.2.2 Breadth-First Traversal Over Temporal Paths

The example presented above in Section 4.2.1 demonstrates how active nodes restrict the set of temporal paths that need to be considered when traversing an evolving graph.

We now give a general description of the BFS algorithm over evolving graphs, both directed and undirected, which correctly takes into account the structure of temporal paths. Our notation generalizes that for static graphs presented in [27, 51].

Definition 4.2.1. *An **evolving graph** G_n is a sequence of (static) graphs $G_n = \langle G^{[1]}, G^{[2]}, \ldots, G^{[n]} \rangle$ with associated time labels $t_1, t_2, \ldots, t_n$ respectively. Each $G^{[t]} = (V^{[t]}, E^{[t]})$ represents a (static) graph labeled by a time t.*

Intuitively, an evolving graph is some discretization of the continuous-time family $G(t)$:



We assume no particular relation between the node and edge sets for each static graph $G^{[t]} = (V^{[t]}, E^{[t]})$. In particular, we allow the node sets to change over time, so that each $V^{[t]}$ may be different. Changing node sets happen naturally in citation networks, where nodes may appear or disappear from the citation network over time. The addition, removal, or relabeling of nodes can be expressed in terms of a map $\Pi^{[t,t']} : V^{[t]} \to V^{[t']}$ that expresses the appropriate permutations and/or projections.

Definition 4.2.2. *A **temporal node** is a pair (v, t), where $v \in V^{[t]}$ is a node at a time t.*

Definition 4.2.3. *A temporal node (v, t) is an **active node** if there exists at least one edge $e \in E^{[t]}$ that connects $v \in V^{[t]}$ to another node $w \in V^{[t]}$, $w \neq v$.*

*An **inactive node** is a temporal node that is not an active node.*

In Figure 4.1, the temporal nodes $(1, t_1)$ and $(2, t_2)$ are active nodes, whereas the temporal node $(3, t_1)$ is an inactive node.

Definition 4.2.4. *A **temporal path** of length m on an evolving graph G_n from temporal node (v_1, t_1) to temporal node (v_m, t_m) is a time-ordered sequence of active nodes, $\langle (v_1, t_1), (v_2, t_2), \ldots, (v_m, t_m) \rangle$. Here, time ordering means that $t_1 \leq t_2 \leq \cdots \leq t_m$ and $v_i = v_j$ iff $t_i \neq t_j$.*

This definition of a temporal path differs from that of the dynamic walk in [38, 37] in that **causal edges**, i.e. sequences of the form $\langle (v, t), (v, t\prime) \rangle$ are included explicitly in temporal paths but are only implicitly included in dynamic walks and are not counted toward the length of dynamic walks. Our definition implies that if either or both end points of a temporal path are inactive, then the entire temporal path must be the empty sequence $\langle\rangle$. Keeping track explicitly of the time labels of each temporal node allows greater generality to cases where the node sets change over time. Furthermore, we shall show later in Sec. 4.3.1 that the explicit bookkeeping of the time labels is essential for correctly generalizing the BFS to evolving graphs.

The following definition of **forward neighbors** generalizes the notion of neighbors and reachability in static graphs.

Definition 4.2.5. *The k-**forward neighbors** of a temporal node (v, t) are the temporal nodes that are the $(k+1)$st temporal node in some temporal path of length $k+1$*

starting from (v, t). *The* ***forward neighbors*** *of a temporal node* (v, t) *are its 1-forward neighbors.*

In Figure 4.1, the forward neighbors of $(1, t_1)$ are $(2, t_1)$ and $(1, t_2)$ and the only forward neighbor of $(2, t_1)$ is $(2, t_3)$. The 2-forward neighbors of $(1, t_1)$ are $(2, t_1), (1, t_2), (2, t_2)$ and $(3, t_2)$. By construction, time stamp of every forward neighbor of an active node (v, t) must be no earlier than t.

Definition 4.2.6. *The* ***distance*** *from a temporal node* (v, t) *to a temporal node* (w, s) *is the* k *for which* (w, s) *is a* k*-forward neighbor of* (v, t).

Our definition of distance, again, differs from the definition of distance in the formulation of [38, 37] in that we explicitly count causal edges toward the distance. It also differs from the notion of temporal distance in the work of Tang and coworkers [77], which is the number of time steps between t and s (inclusive). In this respect, our formulation of the BFS on evolving graphs differs from these other works by minimizing a different notion of distance over an evolving graph.

Note that this notion of distance is not a metric, since the distance from (v, t) to (w, s) will in general differ from the distance of (v, t) from (w, s) owing to time ordering.

Definition 4.2.7. *A temporal node* (w, s) *is* ***reachable*** *from a temporal node* (v, t) *if there exists some finite integer* k *for which* (w, s) *is a* k*-forward neighbor of* (v, t).

4.2.3 Description of the BFS algorithm

The BFS on evolving graphs is described in Algorithm 1. Algorithm 1 is identical to the BFS on static graphs except for line 8, where we visit the forward neighbors of a given temporal node in both space and time. Given an evolving graph G_n and a root (v_1, t_1), Algorithm 1 returns all temporal notes reachable from the root and their distances from the root. *reached* is a dictionary from temporal nodes to integers whose key set represents all visited temporal nodes and whose value set are the corresponding distances from the root.

The BFS constructs a tree inductively by discovering all k-forward neighbors of the root before proceeding to all $(k + 1)$-forward neighbors of the root. Within the

outermost loop, the algorithm iterates over $frontier$, a list of all temporal nodes of distance k from the root. The $nextfrontier$ list is populated with all temporal nodes that are forward neighbors of any temporal node in the $frontier$ list which have not yet been reached by the algorithm.

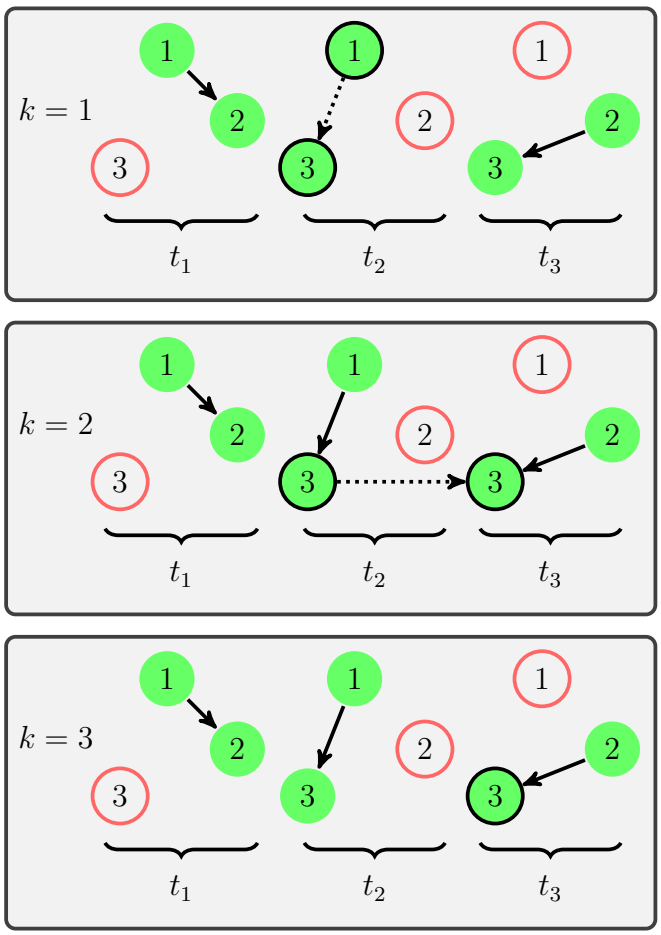


Figure 4.3: Breadth-first search (BFS) on the evolving graph shown in Figure 4.1 starting from the root $(1, t_2)$ at iteration $k = 1, 2, 3$. Note that the time t_1 does not participate in the BFS. Black circles indicate the active nodes forming the $frontier$ and $nextfrontier$ sets in Algorithm 1, connected by the dotted black lines.

As a simple example, consider the BFS on the example graph in Figure 4.1 starting from the root $(1, t_2)$. The procedure is shown in Figure 4.3. The $frontier$ list is first initialized to $\{(1, t_2)\}$. Since the only forward neighbor of $(1, t_2)$ is $(3, t_2)$, iteration $k = 1$ produces $reached[(3, t_2)] = 1$ and $nextfrontier = \{(3, t_2)\}$. In the next iteration $k = 2$, the only forward neighbor of $(3, t_2)$ is $(3, t_3)$, so $reached[(3, t_3)] = 2$ and $nextfrontier = \{(3, t_3)\}$. The algorithm terminates after $k = 3$ after verifying that $(3, t_3)$ has no forward neighbors.

The preceding example illustrates the fact that $G^{[1]}$ plays no part in the BFS

traversal of G_n starting from $(1, t_2)$. In general, all $G^{[t]}$ with time stamps $t < t'$ for a starting node (v, t') are irrelevant to the BFS traversal. Hence without loss of generality we may assume that BFS is always computed with a root at time t_1, the earliest time stamp in G_n.

```
function BFS(G_n, (v_1, t_1))
    reached[(v_1, t_1)] = 0
    frontier = {(v_1, t_1)}
    k = 1
    while frontier ≠ ∅
        nextfrontier = ∅
        for (v, t) ∈ frontier
            for (v', t') ∈forwardneighbors((v, t))
                if (v', t') ∉ reached
                    reached[(v', t')] = k
                    nextfrontier = nextfrontier ∪ {(v', t')}
        frontier = nextfrontier
        k = k + 1
    return reached
```

Algorithm 1: Breadth-first search (BFS) on an evolving graph G_n starting from a root (v_1, t_1). The return value, *reached*, is a dictionary mapping all reachable temporal notes from the root to their distances from the root. At the end of each iteration k, the $frontier$ set contains all temporal nodes of distance k from the root.

Theorem 4.2.1 (Correctness of the evolving graph BFS)**.** *Let G_n be an evolving graph and (v_1, t_1) be an active node of G_n. Then Algorithm 1 discovers every active node that is reachable from the root (v_1, t_1), and reached$[(v, t)]$ is the distance from (v_1, t_1) to (v, t).*

Proof. Let's first consider the case when G_n is directed. Define the set of temporal nodes $\tilde{V}_L^{[t]} = \{(v_1, t) | (v_1, v_2) \in E^{[t]}\}$, which consists of the active nodes at time t which participate on the left side of an edge. Similarly, $\tilde{V}_R^{[t]} = \{(v_2, t) | (v_1, v_2) \in E^{[t]}\}$ contains the corresponding active nodes on the right side of an edge. Then $\tilde{V}^{[t]} = \tilde{V}_L^{[t]} \cup \tilde{V}_R^{[t]}$ is the set of active nodes at time t, and $V = \bigcup_t \tilde{V}^{[t]}$ is the set of all active nodes in G_n.

Similarly, define the set of **causal edges** $E' = \{(u_s, v_t) | u_s = (u, s) \in V, v_t = (v, t) \in V, v = u, s < t\}$, which consists of temporal nodes that connect active nodes sharing the same node at different times. Each edge in E' is then in 1-1 correspondence with a temporal path of length 2, $\langle (v, s), (v, t) \rangle$. Define also the set of **static edges at time** t, $\tilde{E}^{[t]} = \{(e, t) | e \in E^{[t]}\}$, which are simply the edge sets in G_n with time labels,

and the set of **static edges** $\tilde{E}$, being simply the union over all times, $\bigcup_t \tilde{E}^{[t]}$. Then $E = \tilde{E} \cup E'$ is the set of all edges representing all allowed temporal paths of length 2.

The node set V and edge set E now define a static directed graph $G = (V, E)$ that is in 1-1 correspondence with the evolving graph G_n. The node set V of G is in 1-1 correspondence with active nodes of G_n while the edge set E is in 1-1 correspondence with all temporal paths of length 2 on G_n.

We now establish a similar 1-1 correspondence of forward neighbors of an active node with a subset of G. By induction, all new nodes populated into the key set of *reached* at iteration k are of distance k from the root. By definition, the forward neighbors of some active node $(v, t) \in G_n$ are active nodes of either the form (v, t') for some $t' > t$ or (u, t) for some $u \neq v$. In other words, they are connected either by a causal edge or a static edge. Clearly, the former are elements of $E' \subseteq E$ while the latter are elements of $\tilde{E} \subseteq E$. Thus each forward neighbor of an active node $(v, t) \in G_n$ is in 1-1 correspondence with a node in V that is a neighbor of $v_t \in V$.

When G_n is undirected, every edge in $\tilde{E}^{[t]}$ can be represented by two edges in G: from an active node in $\tilde{V}_L^{[t]}$ to an active node in $\tilde{V}_R^{[t]}$ and the reverse. Every edge in E' is in 1-1 correspondence with an edge in G by causality. Therefore, the forward neighbors of an active node is in 1-1 correspondence with a subset of G and the analysis above follows.

The correctness of BFS on the evolving graph G_n now follows from the correctness of BFS on the static graph G, since we have also established a 1-1 correspondence for every intermediate quantity in Algorithm 1. □

As presented, the BFS over evolving graphs makes no assumptions about how the evolving graph G_n is represented. Suppose it is represented by a collection of adjacency lists, one for each active node in G_n. Then we have that the asymptotic complexity of BFS on G_n is the same as that for BFS on G, using the 1-1 construction of G from G_n.

Theorem 4.2.2 (Computational complexity of the evolving graph BFS)**.** *Let G_n be an evolving graph represented using adjacency lists, (v_1, t_1) be an active node of G_n, and $G = (V, E)$ be the static graph constructed from G_n using the 1-1 correspondences defined in the proof of Theorem 4.2.1. Then the asymptotic computational complexity of Algorithm 1 is $O(|E| + |V|)$.*

Proof. Any edge in any edge set of G_n can be accessed in constant time in random access memory. By construction, BFS on G_n is in 1-1 correspondence with BFS on the static graph G. The number of operations of BFS on G is $O(|E| + |V|)$, and so the result follows. □

Note that in the theorems in this section we construct an equivalent static graph G corresponding to the evolving graph G_n. However, G contains more edges than the union of all the static parts of G_n, as we also add causal edges E'. To our knowledge, our formulation of the BFS represents the first attempt to include these edges explicitly in the treatment of evolving graphs.

4.3 Formulating the Evolving Graph BFS with Linear Algebra

4.3.1 The importance of causal edges

For each static graph $G^{[t]} = (V^{[t]}, E^{[t]})$ that constitutes the evolving graph G_n, define its corresponding $\left|V^{[t]}\right| \times \left|V^{[t]}\right|$ one-sided adjacency matrix with elements

$$A_{ij}^{[t]} = \begin{cases} 1 & \text{if } (i,j) \in E^{[t]}, \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

We can then represent G_n using a sequence of adjacency matrices $A_n = \langle A^{[1]}, A^{[2]}, \ldots, A^{n]} \rangle$. The example in Figure 4.1 can be represented as

$$\left\langle \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \right\rangle.$$

For a static graph G with adjacency matrix A, $(A^k)_{ij}$ counts the number of paths of length k between node i and node j. Naïvely, one might want to generalize this result to evolving graphs by postulating that the (i,j)th entry of the discrete path

sum

$$S^{[t_n]} = A^{[t_1]}A^{[t_n]} + \sum_{t_1 \le t \le t_n} A^{[t_1]}A^{[t]}A^{[t_n]} + \cdots + \sum_{t_1 \le t \le t' \le \cdots \le t_n} A^{[t_1]}A^{[t]}A^{[t']} \cdots A^{[t_n]} \tag{4.2}$$

counts the number of temporal paths from (i, t_1) to (j, t_n). However, this postulate is incorrect. In the example of Figure 4.1,

$$(S^{[t_3]})_{13} = \left(A^{[t_1]}A^{[t_2]}A^{[t_3]} + A^{[t_1]}A^{[t_3]}\right)_{13} = 1$$

even though there are clearly two temporal paths from $(1, t_1)$ to $(3, t_3)$ as shown in Figure 4.2.

The first term in the sum $S^{[t_3]}$ vanishes since $A^{[t_1]}A^{[t_2]} = 0$. Furthermore, the vanishing of $S^{[t_2]} = A^{[t_1]}A^{[t_2]}$ itself reflects the absence of any temporal path from t_1 to t_2 that goes through at least one edge at t_1. However,

$$\langle (1, t_1), (1, t_2), (3, t_2) \rangle \tag{4.3}$$

is a clearly a valid temporal path as shown in Figure 4.2 which cannot be expressed by a product of adjacency matrices.

Sums $S^{[t]}$ of the form (4.2) produce an incorrect count of temporal paths because they do not capture temporal paths with causal edges, i.e. subpaths of the form $\langle (v, s), (v, t) \rangle$, $s < t$. One might attempt to amend the sums $S^{[t]}$ in (4.2) by redefining the adjacency matrices to include ones along the diagonal, hence allowing paths containing the sequence $\langle (i, t_1), (i, t_2) \rangle$. However, the resulting sum is still incorrect, as it counts paths with subsequences $\langle (3, t_1), (3, t_2) \rangle$ and are hence not temporal paths. Instead, the temporal path (4.3) is counted by the matrix product $M^{[t_1,t_2]}A^{[t_2]}$, where

$$M^{[t_1,t_2]} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{4.4}$$

$M^{[t_1,t_2]}$ describes the forward time propagation of temporal nodes that are active at both times t_1 and t_2, i.e. it counts temporal paths that contain subsequences $\langle (i, t_1), (i, t_2) \rangle$, and both (i, t_1) and (i, t_2) are active nodes.

The simple example of Figure 4.1 demonstrates why sums over products of adjacency matrices of the form (4.2) do not count temporal paths correctly: they neglect the combinatorics associated with the causal edge set E'. In the next section, we show how to account for these causal edges by introducing a new matrix–vector product $\odot$.

4.3.2 Defining forward neighbors algebraically

The algebraic representation of evolving graphs presented in Section 4.3.1 allows us to exploit a graphical interpretation of matrix–vector products involving the adjacency matrix [51]. If A is the adjacency matrix of a (static) graph G and e_k is the kth elementary unit vector, then the nonzero entries of $A^T e_k$ have indices that are neighbors of k. The algebraic formulation of BFS on evolving graphs follows similarly, but requires a new kind of matrix–vector product, $\odot$, defined by

$$A^T \odot b = \begin{cases} b & \text{if } A^T b \neq 0 \text{ or } Ab \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

The condition $A^T b \neq 0$ ensures that the product is nonzero in components involving left active nodes $\cup_t \tilde{V}_L^{[t]}$, and the condition $Ab \neq 0$ is the analogue for right active nodes $\cup_t \tilde{V}_R^{[t]}$. The forward neighbors of a temporal node (k, t_1) in A_n can then be determined from the indices and time stamps of the nonzero elements in the sequence

$$\left\langle (A^{[1]})^T e_k,\ (A^{[2]})^T \odot e_k,\ \ldots\ (A^{[n]})^T \odot e_k \right\rangle. \tag{4.5}$$

The nonzero entries of the first vector represent forward neighbors that are on the same time stamp t_1, whereas nonzero entries of the other vectors represent forward neighbors that are advanced in time but remain on the same node k. The quantity (4.5) therefore encodes a BFS tree of depth 2, as its nonzero entries are labeled by all temporal nodes of distance 1 from (k, t_1).

Referring back to the example of Figure 4.1, the forward neighbors of node $(1, t_1)$

can be computed by

$$\left\langle \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right\rangle$$
$$= \left\langle \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\rangle$$

From this computation, we can deduce that $(2, t_1)$ and $(1, t_2)$ are the forward neighbors of $(1, t_1)$.

4.3.3 Evolving graphs as a blocked adjacency matrix

The proof of Theorem 4.2.1 provides a construction for representing an evolving graph G_n by a static graph G with nodes corresponding to active nodes of G_n. It turns out that the block structure of G is useful for understanding the nature of the $\odot$ operation.

Consider the second iteration of BFS on G_n with root (k, t_1), which requires computing the sequences

$$\left\langle (A^{[1]})^T c_1, (A^{[2]})^T \odot c_1, \ldots, (A^{[n]})^T \odot c_1 \right\rangle \tag{4.6a}$$
$$\left\langle (A^{[2]})^T c_2, \ldots, (A^{[n]})^T \odot c_2 \right\rangle \tag{4.6b}$$
$$\ldots \tag{4.6c}$$
$$\left\langle (A^{[n]})^T c_n \right\rangle \tag{4.6d}$$

where $c_1 = (A^{[1]})^T e_k$ and $c_i = (A^{[i]})^T \odot e_k$ for $i > 1$. Summing resultant vectors that share the same time stamp, we obtain vectors whose nonzero elements have indexes labeled by the forward neighbors of the nodes computed at step 1.

Compare this with the matrix

$$\boldsymbol{M}_n = \begin{bmatrix} A^{[t_1]} & M^{[t_1,t_2]} & \ldots & M^{[t_1,t_n]} \\ 0 & A^{[t_2]} & \ldots & M^{[t_2,t_n]} \\ & \ldots & & \\ 0 & 0 & \ldots & A^{[t_n]} \end{bmatrix}$$

where $M^{[t_i,t_j]}$ is the matrix whose rows are labeled by $V^{[t_i]}$ and columns are labeled by $V^{[t_j]}$, and whose entries are

$$M_{uv}^{[t_i,t_j]} = \begin{cases} 1 & \text{if } (u,v) \in E', \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix blocks $A^{[t]}$ encode the static edge set $\tilde{E}$, whereas the off-diagonal blocks $M^{[t_i,t_j]}$ together encode the causal edge set E', which capture temporal paths with subsequences of the form $\langle (v,t_i),(v,t_j)\rangle$. Then $\boldsymbol{M}_n$ is the adjacency matrix of the graph $(\cup_t V^{[t]}, E)$, which is the graph G together with all the inactive nodes. From the definition, $\boldsymbol{M}_n$ has nonzero entries only in rows and columns that correspond to active nodes V, and so retaining only these rows and columns corresponding to V produces the adjacency matrix $\boldsymbol{A}_n$ of $G = (V, E)$.

The off-diagonal blocks $M^{[t_i,t_j]}$ provide an explicit matrix representation for the $\odot$ product in that $(M^{[t_i,t_j]})^T b = (A^{[t_i]})^T \odot b$. An example of such an off-diagonal block was already provided in (4.4). These off-diagonal blocks represent traversal between active nodes with the same node space labels but are still separated by time, and are essential for the correct enumeration of temporal paths. The upper triangular structure of $\boldsymbol{M}_n$ (and hence $\boldsymbol{A}_n$) reflects the causal nature of temporal paths in that they cannot go backward in time.

The BFS algorithm presented above can therefore be interpreted as computing the sequence of matrix–vector products $\boldsymbol{b}$, $\boldsymbol{A}_n^T\boldsymbol{b}$, $(\boldsymbol{A}_n^T)^2\boldsymbol{b}$, ..., formed by applying successive monomials of $\boldsymbol{A}_n^T$ to the block vector $\boldsymbol{b}^T = [b^T, 0, \cdots, 0]$ where b^T encodes the root in the space of active nodes $\tilde{V}^{[t_1]}$.

For the example of Figure 4.1, we have

$$\begin{aligned} V &= \{(1,t_1),(2,t_1),(1,t_2),(3,t_2),(2,t_3),(3,t_3)\}, \\ \tilde{E} &= \{((1,t_1),(2,t_1)),((1,t_2),(3,t_2)),((2,t_3),(3,t_3))\}, \\ E' &= \{((1,t_1),(1,t_2)),((2,t_2),(2,t_3)),((3,t_2),(3,t_3))\}. \end{aligned}$$

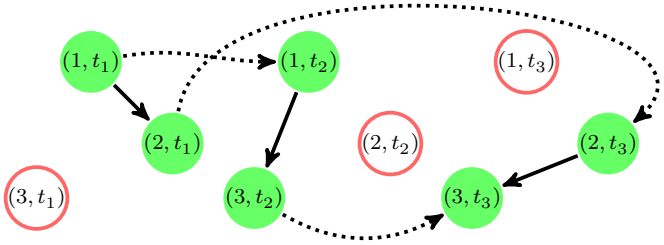


Figure 4.4: Static graphs corresponding to the evolving graph example of Figure 4.1. The green nodes are active nodes while the red nodes are inactive nodes. The black lines are edges in the static edge set $\tilde{E}$ and are encoded algebraically in the diagonal blocks $A^{[t]}$ of the adjacency matrix $\boldsymbol{A}_3$ or $\boldsymbol{M}_3$. The dotted lines are edges in the causal edge set E' and are encoded algebraically in the off-diagonal blocks $M^{[t_i,t_j]}$. The static graph G constructed in the proof of Theorem 4.2.1 is formed by retaining all the edges shown and only the active nodes, and has the adjacency matrix $\boldsymbol{A}_3$. The graph containing all the edges and temporal nodes shown has adjacency matrix $\boldsymbol{M}_3$.

In the order specified for V, the adjacency matrix of G is then

$$\boldsymbol{A}_3 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Starting from the vector $\boldsymbol{b} = e_1$, the sequence of iterates is then

$$\left\langle \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \ldots \right\rangle.$$

We see that $(\boldsymbol{A}_n^T)^2$ encodes all the products in (4.6a)-(4.6d), including both the ordinary matrix–vector product and the $\odot$ product. Furthermore, $(\boldsymbol{A}_3^T)^3\boldsymbol{b}$ correctly counts the two allowed temporal paths from $(1, t_1)$ to $(3, t_3)$, and that the off-diagonal structure encoded in E' and the $M^{[t,t']}$ blocks are critical to obtaining the correct count.

Finally, we note that some results regarding the BFS on evolving graphs can be derived easily using properties of the block adjacency matrix $\boldsymbol{A}_n$. For example, we can prove a simple lemma that the block adjacency matrix $\boldsymbol{A}_n$ is nilpotent whenever all the subgraphs $G^{[t]}$ of G_n are acyclic.

Lemma 4.3.1 (Acyclicity implies nilpotence)**.** *Let $G_n = \langle G^{[t_i]} \rangle_{i=1}^{n}$ be an evolving directed graph and let all the directed graphs $G^{[t]}$ be acyclic. Then $\boldsymbol{A}_n$ is nilpotent.*

Proof. Recall from the definition of the matrix $\boldsymbol{A}_n$ that it is block upper triangular, reflecting causality. Since each directed graph $G^{[t]}$ is acyclic, we can apply a topological ordering on the graph such that its corresponding adjacency matrix $A^{[t]}$ is strictly upper triangular. As a result, $\boldsymbol{A}_n$ must be upper triangular.

Furthermore, none of the graphs $G^{[t]}$ can have any self-edges, i.e. edges of the form (u, u), and so all diagonal entries of $A^{[t]}$ must be zero. Therefore all the diagonal entries of $\boldsymbol{A}_n$ by construction must be zero also.

We have now proven that $\boldsymbol{A}_n$ is an upper triangular matrix whose diagonal entries are all zero. Therefore, $\boldsymbol{A}_n$ is nilpotent. □

Lemma 4.3.1 also holds for acyclic undirected graphs so long as the corresponding adjacency matrix representation is encoded in an asymmetric fashion akin to (4.1).

The blocked matrix structure of the adjacency matrices presented here provide interesting relationships between their matrix properties and the algorithmic properties of BFS, made possible because of the reformulation of BFS as repeated power iterations of the adjacency matrix in Algorithm 2. Note, however, that these matrices need never be instantiated for practical computations. Rather, since Algorithm 2 only requires the matrix–vector product involving the adjacency matrix, the formulation of Algorithm 1 provides an efficient way to exploit the block structure of $\boldsymbol{A}_n$. The $\odot$ operation provides an efficient way to compute the action of the off-diagonal products. Representing the diagonal blocks $A^{[t]}$ as sparse matrices further reduces the cost of BFS by exploiting latent sparsity in graphs that show up in practical applications.

4.3.4 The algebraic formulation of BFS on evolving graphs

The blocked matrix–vector products introduced in the previous section allows us to write down an elegant algebraic formulation of BFS on evolving graphs, as presented

in Algorithm 2.

function *ABFS*$(A_n, (v_1, t_1))$
  Form $\boldsymbol{A}_n^T$ from A_n.
  $\boldsymbol{b}_{v_1} = 1$
  $k = 1$
  $reached[(v_1, t_1)] = 0$
  while nonzeros($\boldsymbol{b}$) $\neq \varnothing$
    $\boldsymbol{b} = \boldsymbol{A}_n^T \boldsymbol{b}$
    for $k \in$ nonzeros($\boldsymbol{b}$)
      if activeNodes(k) $\in$ *reached*
        $\boldsymbol{b}_k = 0$
    for *node* $\in$activeNodes($\boldsymbol{b}$)
      $reached[node] = k$
    $k = k + 1$
  return *reached*

Algorithm 2: An algebraic formulation of BFS on evolving graphs. Given A_n, the adjacency matrix representation of G_n and (v_1, t_1), a node of G_n, returns *reached* as defined in Algorithm 1. The function nonzeros(v) returns the nonzero indices of the vector v, and the function map(b) maps a block vector's indices to their corresponding active nodes.

Theorem 4.3.1. *Algorithm 2 terminates.*

Proof. First, we prove that the BFS terminates in the case of acyclic evolving graphs. Recall from Lemma 4.3.1 that $\boldsymbol{A}_n^T$ is nilpotent, i.e. there exists some positive integer k for which $(\boldsymbol{A}_n^T)^k = 0$. Hence, after iteration k, $\boldsymbol{b}$ is assigned the value $(\boldsymbol{A}_n^T)^k \boldsymbol{b} = 0$. Therefore, Algorithm 2 must terminate after iteration k.

For evolving graphs with cycles, lines 9-11 of Algorithm 2 enforce that the BFS visits each active node at most once. Since the kth block of $\boldsymbol{b}$ is zeroed out if an active node has already been visited, the subgraph traversed in the BFS cannot have cycles. Thus all that is required is the previous result that the BFS on an acyclic graph terminates. □

Theorem 4.3.2. *Algorithm 1 and Algorithm 2 are equivalent.*

Proof. The initialization steps are trivially equivalent. At the beginning of iteration k, the block vector $\boldsymbol{b}$ represents the frontier nodes encoding the $frontier$ set of Algorithm 1. The matrix–vector product $\boldsymbol{A}_n^T \boldsymbol{b}$ encodes the forward neighbors of all the frontier nodes. Subsequently, active nodes that have already been visited in previous iterations are zeroed out of the new $\boldsymbol{b}$. □

4.3.5 Computational complexity analysis of the algebraic BFS

The complexity of the algebraic BFS of Algorithm 2 is significantly more complicated that that of Algorithm 1. While the latter uses the usual adjacency list representation for graphs, the computational cost of the former depends critically on the actual representation of the matrices. Furthermore, the average case analysis is complicated by the expected fill-in of the vector $\boldsymbol{b}$, which influences the cost of the matrix-vector product on line 7 and the expected number of iterations of the **while** loop beginning on line 6.

While a full complexity analysis is beyond the scope of this paper, it is straightforward to present worst-case results for dense and compressed sparse column (CSC) matrices.

Lemma 4.3.2 (Number of iterations)**.** *In the worst case, the number of iterations in the* ***while*** *loop of Algorithm 2 is* $k = O(|E|)$.

Proof. In the worst case, the BFS must traverse every active node, and only one new active node is discovered in each iteration. The number of active nodes is bounded above by the cardinality of the full edge set, $|E|$. □

The average case analysis for the number of iterations is considerably more complicated and is beyond the scope of this paper.

Theorem 4.3.3 (Dense matrices)**.** *Suppose* $\boldsymbol{A}_n$ *is represented as a dense matrix. Then the computational complexity of Algorithm 2 is* $O(k|V|^2)$*, which in the worst case is* $O(|E||V|^2)$.

Proof. Since $\boldsymbol{A}_n$ is a $|V| \times |V|$ matrix, the matrix-vector product $\boldsymbol{A}_n\boldsymbol{b}$ takes $O(|V|^2)$ operations to compute. Thus the cost of Algorithm 2 is $O(k|V|^2) = O(|E||V|^2)$ in the worst case. □

It is clear that practical implementations of the BFS should never construct the full matrix $\boldsymbol{A}_n$ in memory. What happens if we use a sparse blocked representation?

Theorem 4.3.4 (Block diagonal sparse matrices)**.** *Suppose* $\boldsymbol{A}_n$ *is represented by a collection of compressed sparse column matrices for each diagonal block* $A^{[t]}$*. Then the computational complexity of Algorithm 2 is* $O(k(|\tilde{E}| + |V|))$*, which in the worst case is* $O(|E|(|\tilde{E}| + |V|))$.

Proof. The gaxpy operation for CSC matrices costs $2n_{nz}$ flops, where n_{nz} is the number of stored values in the matrix. The cost of each diagonal subblock calculation $A^{[t]}b_t$ is therefore $O(|E^{[t]}|)$, since $A^{[t]}$ by construction has nonzero entries only when a static edge exists.

The off-diagonal products $M^{[t,t\prime]}b_{t\prime}$ can be computed in $O(|V^{[t]}| + |E^{[t]}|)$ time for all $t\prime \geq t$ since it can be implemented by the $\cdot$ operation, which constructs either a zero vector or keeps the same vector. The cost of checking the condition $(A^{[t]})^T b_{t'} \neq 0$ is $O(|E^t|)$ in the worst case since all that is required is to check whether or not each column of A is empty. Similarly, checking the condition $A^{[t]}b_{t'} \neq 0$ reduces to checking if each row of A is empty, and thus is of cost $O(|V^t|)$.

Thus, the cost of multiplying one block row of $\boldsymbol{A}^T$ (for some time t) with $\boldsymbol{b}$ is $O(|V^{[t]}| + |E^{[t]}|)$. Summing over all times yields the desired result. □

We can see that even implementing the BFS algebraically using CSC matrices is insufficient to reduce the running time to linear, which can be achieved for the adjacency list representation in Algorithm 1. This result strongly suggests that additional work is needed to produce true algorithmic equivalence at the computational level.

4.4 Implementation in Julia

To study evolving graphs and experiment with various graph types, we have developed EvolvingGraphs.jl [81], a software package for the creation, manipulation, and study of evolving graphs written in Julia [7]. It is freely available online at

https://github.com/weijianzhang/EvolvingGraphs.jl

and available with the MIT "Expat" license. The package contains an implementation of the evolving graph BFS of Algorithm 1. IntEvolvingGraph, a data type in EvolvingGraphs.jl, represents an evolving graph as adjacency lists.

We now present some simple timing data to show that our implementation of Algorithm 1 is indeed linear scaling in computational cost.

We generate a sequence of random (directed) IntEvolvingGraphs with 10^5 active nodes and 10 time stamps. The first IntEvolvingGraph in the sequence has about 10^8 static edges. We consecutively add new random static edges to this IntEvolvingGraph.

For example, the second random IntEvolvingGraph in the sequence has about 1.5×10^8 static edges and the third has 1.8×10^8 static edges. Note that in this experiment, we do not have direct control over the full edge set E, only the static edge set $\tilde{E}$. When we add new static edges, new causal edges may be added as well (if the corresponding temporal nodes were not active before). However, the number of newly introduced causal edges for each active node is bounded by the number of time stamps, so it suffices to demonstrate linear scaling in $|\tilde{E}|$. Figure 4.5 shows the plots of number of edges against the computation time for running Algorithm 1 in Julia. All experiments are conducted on a single core of a Linux system with 1TB of RAM and 80 cores of Intel(R) Xeon(R) E7-8850s running at 2.00 GHz clock speed. The results show Algorithm 1 can be computed in linear time, which agrees with the result of Theorem 4.2.2.

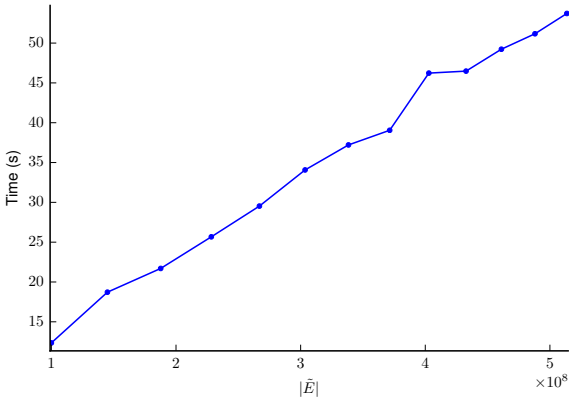


Figure 4.5: Experimental run time of Algorithm 1 on a collection of random evolving graphs with 10^5 active nodes and 10 time stamps, showing linear scaling in $|\tilde{E}|$. The horizontal axis shows the number of edges ($|\tilde{E}|$, including only the static edges) of each evolving graph, while the vertical axis shows the corresponding computation time.

4.5 Application to Citation Networks

Evolving graphs have found many applications to analyzing networks that change over time [38, 37]. In this section, we focus specifically on citation networks, and show that evolving graph formalism presented above can be used to capture the dynamical structure of citation networks. Consider the evolving graph $G_n = \langle G^{[t]} \rangle_t$ such that $G^{[t]}$ has node set corresponding to authors active at time t and directed edge set $E^{[t]} \ni (i, j)$

representing a citation of author j by author i in a publication at time t.

Then given an author a at time t_1, the evolving graph BFS described above can compute $T(a, t_1)$, the set of all the authors that have been influenced by a's work at time t_1. Define also a *community* to be a group of researchers that have been influenced by the same authors. For example, given a paper published by a at time t, we can determine a's community by searching backward in time to find $T^{-1}(a, t)$, the authors that influenced a at time t, and then searching forward to find $T(l_1, t_1) \cup T(l_2, t_2) \cup \cdots \cup T(l_k, t_k)$, where $(l_1, t_1), (l_2, t_2), \ldots, (l_k, t_k)$ are the leaves of $T^{-1}(a, t)$. The backward search in time follows straightforwardly from the forward time traversal presented above simply by reversing the time labels, e.g. by the transformation $t \to -t$.

We are currently investigating the use of our evolving graph BFS on citation networks.

4.6 Conclusion

The correct generalization of BFS to evolving graphs necessitates a careful enumeration of temporal paths. The structure associated with causal edges E' turns out to be of vital importance, and cannot be capture simply by products of successive adjacency matrices, which by construction can only capture the topologies of the static edges $\tilde{E}$. Only by considering both causal edges and static edges can we show that BFS over any evolving graph G_n computes the correct result for our notion of distance. The new concepts of activeness, temporal paths, and causal edges make possible a correct implementation of BFS to evolving graphs and we expect that these ideas will continue to provide powerful new insights into how similar graphical algorithms may be generalized correctly.

Furthermore, we show that BFS on evolving graphs admits an algebraic formulation that easily provides nontrivial results, such as termination of the algorithm. However, our current understanding tells us that the BFS over evolving graphs is most efficiently computed in the adjacency list representation, thus never forming explicit matrix-vector products. Further work is needed to elucidate more efficient formulations of the algebraic BFS for evolving graphs.

Chapter 5

A Closer Look at Time-Preserving Paths on Evolving Graphs

5.1 Introduction

In the 1985 science-fiction film "Back to the Future", Marty McFly (played by Michael J. Fox) travelled back in time and accidentally changed history. However, we cannot travel back in time or even send a message to the past. To traverse an evolving graph (a time-dependent graph), one has to consider the order of time stamps. In particular, a walk on an evolving graph needs to be time-preserving, meaning we can not pass a message (through an edge) to a node at a earlier time. Time-preversing walks and paths are very important concepts for network flow on evolving graphs.

We represent an evolving graph as a time-ordered sequences of graphs, similar to the work of Tang and coworkers [64, 76, 77, 78] and Grindrod, Higham and coworkers [37, 38]. The idea is to divide the system's time span into temporal slices and regard it as a sequence of static graphs $G^{[t]}$, one for each layer. Such a representation arises naturally in many real-world applications. For example, consider networks of users interacting through messaging. Each interaction between two users A and B is stored with a time stamp t. If we represent user A as a node then user A at time stamp t is represented by a pair (A, t) and a message sent from user A to user B at time stamp t can be represented as $\langle (A, t), (B, t) \rangle$.

In general, a node v on $G^{[t]}$ is represented by a pair (v, t), where t is the time stamp of the graph $G^{[t]}$. A sequence of interactions between users form a walk. Suppose A

sent a message to B at time stamp t_1 and then B sent this message to C at time stamp t_2. We can represent this activity as $\langle (A, t_1), (B, t_1), (B, t_2), (C, t_2) \rangle$, where each pair of successive nodes has meanings: $\langle (A, t_1), (B, t_1) \rangle$ and $\langle (B, t_2), (C, t_2) \rangle$ represent interactions between users A, B, and C and $\langle (B, t_1), (B, t_2) \rangle$ represents the fact that user B holds the message at time stamps t_1 and t_2.

A time-respecting walk is a sequence of nodes $\langle (v_1, t_1), (v_2, t_2), \ldots, (v_m, t_m) \rangle$, where $t_1 \leq t_2 \leq \cdots \leq t_n$ are the corresponding time stamps. If $t_i = t_j$, then $\langle (v_i, t_i), (v_j, t_j) \rangle$ is an edge where both nodes are on the same temporal slice, otherwise it is an edge linking nodes from different temporal slices. We call edges of the first kind *static edges* and of the second kind *causal edges*. Recall that in static graph, a walk of length m is defined as a sequence of nodes $v_1, v_2, \ldots, v_{m+1}$ such that for each $i = 1, \ldots, m$ there is an edge from v_i to v_{i+1}. How can we determine the length of a time-respecting walk? One could define the length of a time-respecting walk as the total number of temporal slices (see the shortest temporal distance in [76]) or the number of static edges (see the dynamic walk in [38]). This paper studies the impact of these two different definitions. In particular, we study a general definition of temporal distance that takes account of both temporal slices and static edges.

Researchers have generalised many static graph centrality algorithms for evolving graphs. One approach is to use the fact that the elements of the matrix product of adjacency matrices at different time stamps can correctly count the number of temporal paths. However, not every centrality algorithm can be generalised in this way. Another approach is to define evolving graph centrality algorithms using time-respecting paths. This approach is applicable to algorithms that are defined using shortest paths.

Borgatti [12] noted that the manner of traffic flow entails a new way to think about centrality. In particular, the importance of a node in a network cannot be determined without reference to how traffic flows through the network. Inspired by this idea, we simulate network flows in evolving graphs to measure the centrality scores. The aims of this paper are as follows. First, to derive important evolving graph centrality from first principles via network flow. Second, to study the impact of temporal distance between nodes on centrality. Third, to generalise PageRank on evolving graphs. We carry out the experiment using the Julia dynamic network analysis package EvolvingGraphs.jl

(https://github.com/EtymoIO/EvolvingGraphs.jl). We use the research paper data gathered from Etymo (https://etymo.io), a visual search engine for data scientists.

5.2 Motivation: A Closer Look at Katz Centrality on Evolving Graphs

We let $G_n = \langle G^{[1]}, G^{[2]}, \ldots, G^{[n]} \rangle$ be an evolving graph and $A^{[1]}, A^{[2]}, \ldots A^{[n]}$ be the corresponding adjacency matrices. Then the dynamic communicability matrix is defined as

$$Q^{[k]} = (I - \alpha A^{[1]})^{-1}(I - \alpha A^{[2]})^{-1} \cdots (I - \alpha A^{[n]})^{-1}, \tag{5.1}$$

where the parameter α satisfies $\alpha < 1/\max_k \rho(A^{[k]})$, the spectral radius of $A^{[k]}$. The ith row sum of $Q^{[k]}$ is the broadcast centrality.

Suppose A, B, and C are three co-workers in a large internet company. Figure 5.1 shows how they exchange ideas during three days at work. The directions of the arrows indicate the direction of the flow of ideas. For example, A shares a new idea with B on day 1 and B shares a new idea with C on dat 3. As a result, A's idea can influence C. Here, and in subsequent figures, green filled circles represent active node and red circles represent inactive nodes.

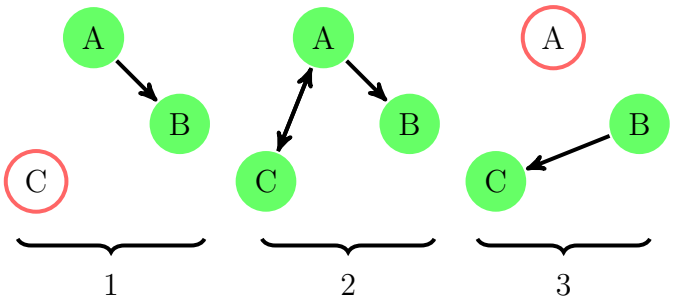


Figure 5.1: An evolving directed graph with 3 time stamps 1, 2 and 3. At each time stamp, the evolving graph is represented as a graph. The green filled circles represent active nodes while the red circles represent inactive nodes. Directed edges in each time stamp are shown as black arrows.

We could use EvolvingGraphs.jl to generate the above evolving graph and compute the Katz centrality.

```
using EvolvingGraphs
```

```
using EvolvingGraphs.Centrality

g = EvolvingGraph{Node{String}, Int}()
add_bunch_of_edges!(g, [("A", "B", 1), ("A", "C", 2),
("A", "B", 2),("C", "A", 2),("B", "C", 3)])
```

The centrality values for each node are

```
katz(g)
3-element Array{Tuple{EvolvingGraphs.Node{String},Float64},1}:
 (Node(A), 0.742301)
 (Node(B), 0.42943)
 (Node(C), 0.514373)
```

As expected A is the most important node in the network. C is the second most important node in the network as it influenced A at time stamp 2. If we reverse the time of communication between day 1 and 3, i.e., B shares a new idea with C on day 1 and A shares a new idea with B on day 3. Then in this case the idea A had on day 3 cannot pass to C. This is illustrated in Figure 5.2. In this case, we have

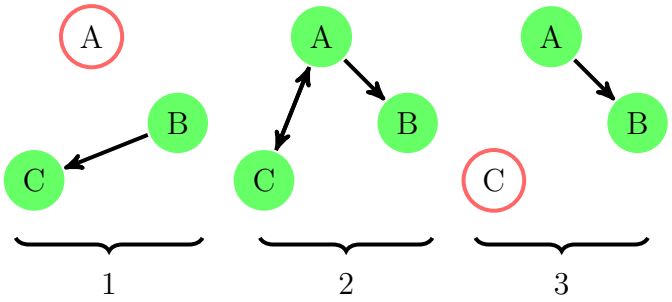


Figure 5.2: An evolving directed graph with 3 time stamps 1, 2 and 3.

```
g2 = EvolvingGraph{Node{String}, Int}()
add_bunch_of_edges!(g2, [("A", "B", 3), ("A", "C", 2),
("A", "B", 2),("C", "A", 2),("B", "C", 1)])
```

The centrality ratings of the three nodes are

```
katz(g2)
3-element Array{Tuple{EvolvingGraphs.Node{String},Float64},1}:
 (Node(A), 0.687679)
 (Node(B), 0.490062)
 (Node(C), 0.535666)
```

Notice the rating of A decreased as its influence is not as important as in Figure 5.1. If the communication between A, B, and C are not on three consecutive days but on day 1, day 10, and day 100, as shown in Figure 5.3, we would expect the centrality scores to change. However, according to (5.1), the scores are unchanged. We notice that the generalised Katz centrality on evolving graphs only takes account of the edges at each time stamp but not the communication time between edges at different time stamps.

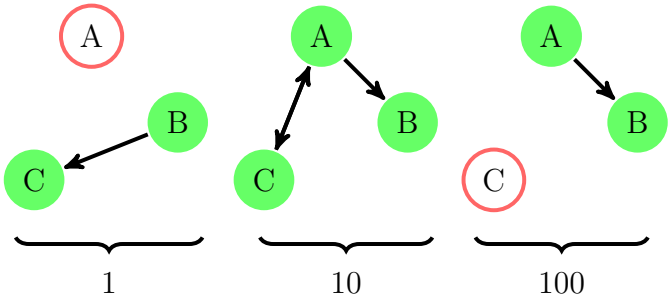


Figure 5.3: An evolving directed graph with 3 time stamps 1, 10 and 100. At each time stamp, the evolving graph is represented as a graph. The green filled circles represent active nodes while the red circles represent inactive nodes. Directed edges in each time stamp are shown as black arrows.

5.3 Time-Preserving Walks and Paths

A walk of length ℓ is a sequence of nodes $v_1, v_2, \ldots, v_l, v_{l+1}$ such that for each $i = 1, 2, \ldots, \ell$, there is an edge from v_i to v_{i+1}. A path of length ℓ is a walk of length ℓ such that all the nodes are different. Similarly, we define a temporal walk to be a time-ordered sequence of temporal nodes. A temporal path is a temporal walk such that all the temporal nodes are different. The definition of centrality depends on the manner in which traffic flows through a network [12]. Time-preserving walks induce temporal Katz centrality and temporal closeness centrality while time-preserving paths induce betweenness centrality and closeness centrality.

We now describe evolving graph network flow in more details. We follow the definition of *evolving graph* in [15].

Definition 5.3.1. *An* ***evolving graph*** G_n *is a sequence of (static) graphs* $G_n = \langle G^{[1]}, G^{[2]}, \ldots, G^{[n]} \rangle$ *with associated time stamps* $t_1 \leq t_2 \leq \ldots \leq t_n$. *Each* $G^{[t]} =$

$(V^{[t]}, E^{[t]})$ represents a (static) graph labelled by a time t.

Note that the node sets $V^{[t]}$ can change over time, i.e., nodes may appear or disappear at a particular time stamp. For example, in Figure 5.4, at time stamp 1998, $V^{[1998]} = \langle 1, 2\rangle$ and $E^{[1998]} = \langle (1, 2)\rangle$. Each graph $G^{[t]}$ can be represented by its adjacency matrix $A^{[t]}$. We can represent G_n by a list of adjacency matrices $A_n = \langle A^{[1]}, A^{[2]}, \ldots, A^{[n]}\rangle$.

If we only measure the temporal difference in a temporal walk, we have a new definition of Katz centrality. In the matrix form, the nonzero entries of $A^{[t_i]}A^{[t_{i+1}]}$ counts all the temporal walks of length 1, and $A^{[t_i]} \cdots A^{[t_j]}$, where $i < j$ counts all the temporal walks of length $j - i$. For example, $A^{[1]}A^{[4]}$, $A^{[1]}A^{[1]}A^{[4]}$ and $A^{[1]}A^{[2]}A^{[3]}A^{[4]}$ count all walks of length 3. In the original Katz centrality, the 'attenuation' factor α is imposed on the number of paths, i.e., longer paths has smaller effect to the overall rating. Hence the final result is the summation of all products of the form

$$\alpha^k A^{[i]} \cdots A^{[i+k]}$$

Note this is different from (5.1) as discussed in Section 5.4.1.

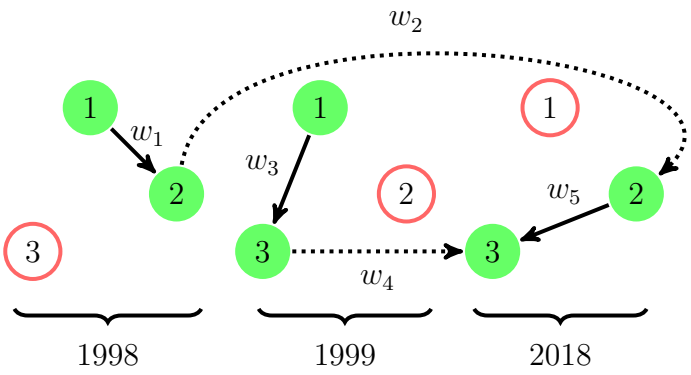


Figure 5.4: An evolving directed graph with 3 time stamps 1998, 1999 and 2018. At each time stamp, the evolving graph is represented as a graph. Directed edges in each time stamp are shown as black arrows and directed edges between graphs are shown as dotted arrows, where $w_1, w_2 \ldots w_5$ are edge weights.

Definition 5.3.2. *A **temporal node** is a pair (v, t), where $v \in V^{[t]}$ is a node at a time t.*

Definition 5.3.3. *A temporal node (v, t) is an **active node** if there exists at least*

one edge $e \in E^{[t]}$ that connects $v \in V^{[t]}$ to another node $w \in V^{[t]}, w \neq v$. Otherwise it is called an inactive node.

In Figure 5.4, the filled green circles are active nodes. For the adjacency matrix representation, if a node i is active at time stamp t then the ith row or the ith column of $A^{[t]}$ has at least one none-zero entry.

Definition 5.3.4. *A **static edge** $\langle (v_i, t), (v_j, t) \rangle$ is a pair of elements of $V^{[t]}$, i.e., $v_i \in V^{[t]}$ and $v_j \in V^{[t]}$. A **causal edge** $\langle (v, t_i), (v, t_j) \rangle$ is a pair of the same node at different time stamps.*

For example, in Figure 5.4 $\langle (1, 1998), (2, 1998) \rangle$ is a static edge at time stamp 1998 and $\langle (3, 1999), (3, 2018) \rangle$ is a causal edge between time stamp 1999 and time stamp 2018. The difference between static edges and causal edges was first explicitly considered in [15].

We can denote a *weighted static edge* as $\langle (v_i, t), (v_j, t), w_s \rangle$ and a *weighted causal edge* as $\langle (v, t_i), (v, t_j), w_t \rangle$. Here w_s represents the spatial distance between the two nodes v_i and v_j; w_t represents the temporal distance of v at different time stamps. In Figure 5.4, the black arrows represent static edges and the dotted arrows represent causal edges. Suppose we assign all the static edge weights to 1 and let the causal edge weight be the number of time stamps between the pair of nodes. Then in Figure 5.4 the edge weight between $(1, 1998)$ and $(2, 1998)$ is 1 and the edge weight between $(2, 1998)$ and $(2, 2018)$ is 10. In general, a weighted time-preserving walk can be defined as follows.

Definition 5.3.5. *A **weighted time-preserving walk** of length m on an evolving graph G_n is an alternating time-ordered sequence of active nodes and edges, $\langle (v_1, t_1), e_1, (v_2, t_2), e_2, (v_3, t_3) \ldots, e_m, (v_m, t_m) \rangle$, where $t_1 \leq t_2 \leq \cdots \leq t_m$ and $v_i = v_j$ if and only if $t_i \neq t_j$, and $e_j = \langle (v_j, t_j), (v_{j+1}, t_{j+1}) \rangle$ can be either a weighted static edge or weighted causal edge.*

It is natural to define the length of a walk on a static graph as the number of nodes travelled through the walk. For the weighted graph case, the length of a walk is the total weight of all the edges in the walk. We observe that static edges link nodes in space while causal edges link nodes in time. Hence we would like to distinguish

temporal distance from spatial distance in an evolving graph. For two linked temporal nodes, we set the length of a static edge between them to be one and the length of a causal edge to be the temporal distance between them. The following Julia function measures the temporal distance between two active nodes in an evolving graph.

```
function temporal_distance(v1::TimeNode,
                           v2::TimeNode, beta::Real = 1.)
  beta* abs(node_timestamp(v1) - node_timestamp(v2))
end
```

This function defines the temporal distance to be the difference between their time stamps scaled by a hyper-parameter beta. In the simplest case, the time-respecting walk $\langle(1, 1998), (2, 1998), (2, 2018), (3, 2018)\rangle$ in Figure 5.4 has length 22 in total because two static edges $\langle(1, 1998)(2, 1998)\rangle$ and $\langle(2, 2018), (3, 2018)\rangle$ each have length one, and $\langle(2, 1998)(2, 2018)\rangle$ traverse 20 time stamps and thus has length 20. A time-preserving walk (and path) can traverse both causal edges and static edges.

5.4 Centrality Measures

Centrality measures the importance of nodes within a graph. To generalize centrality measures for evolving graphs there are two common approaches. First, we could exploit the fact that matrix product of adjacency matrices at different time stamps counts the number of dynamic walks in an evolving graph. Using this idea, we can generalize walk-based centrality measures such as the Katz centrality [26] and communicability betweenness [1]. Second, we could generalize the definition of shortest paths as shortest temporal paths and replace shortest paths in centrality measures with shortest temporal paths where possible. This way we can define temporal betweenness centrality and temporal closeness centrality [64].

We observe both approaches ignore the communication time between edges at different time stamps, i.e., the time gap between different (static) graph slices. We take account of this communication time by deriving centrality measures from the original theses of the ideas and show communication time has important impact on the final node ratings (and rankings). We note that our goal is not to provide efficient centrality algorithms, which is out of the scope of this paper.

5.4.1 Temporal Katz Centrality

Katz centrality accounts the influence of nearest-neighbours to a give node and the influence of other nodes separated at a certain distance from it. The kth power of the adjacency matrix A of a graph G accounts for walk of length k. The expansion

$$A^0 + \alpha A + \alpha^2 A^2 + \cdots \tag{5.2}$$

converges to $(I - \alpha A)^{-1}$ when $\alpha < 1/\rho(A)$. The (i, j) entry of (5.2) counts all walks from i and j with the influence of walks of length k scaled by a factor of α^k. The Katz centrality of node i is the ith row sum of $(I - \alpha A)^{-1}$, which is the sum of influence of all the nodes that node i can reach via a walk. In other words, we start with node i and each out-neighbour of node i has influence α on node i and the out-neighbour of out-neighbour of node i has influence α^2 on node i, and so on.

For evolving graphs, we replace out-neighbours with forward neighbours which preserve the direction of time. Then the Katz score of a node i at time stamp t, denoted by (i, t), is the sum of influence of all the active nodes that (i, t) can reach via a time-preserving walk. Unlike [38], we also consider the temporal distance between two nodes. For example, the temporal distance between $(i, 2001)$ and $(i, 2018)$ is 17. With the definition of temporal_distance introduced in Section 5.3. We define the temporal Katz score of node (i, t) by the algorithm given in Listing 5.1.

```
function temporal_katz(g::AbstractEvolvingGraph,
  start::TimeNode; alpha = 0.2, max_level = 10)
    score = 0.
    v = start
    fronter = [v]
    level = 0
    while level < max_level
        next = []
        for u in fronter
            for v in forward_neighbors(g, u)
                push!(next, v)
                if node_key(v) != node_key(u)
                  td = temporal_distance(start, u)
                  d = td + level
                  score += alpha^d
                end
```

```
            end
        end
        fronter = next
        level += 1
    end
    return score / num_edges(g)
end
```

Listing 5.1: Temporal Katz Centrality of Single Node

The algorithm temporal_katz performs a breadth first search (BFS) on an evolving graph g that starts at TimeNode v, which represents a node at a specific time stamp. For each TimeNode v, we accumulate the influence of linked nodes according to their spatial and temporal distance from v. The spatial distance is recorded in variable level and the temporal distance is calculated by temporal_distance. The algorithm stops searching when level is larger or equal to max_level. We set alpha to be 0.2 and max_level to be 10. Then for Figure 5.2, we find the rating of each active node as shown below.

```
TimeNode(A, 2) => 0.466667
TimeNode(A, 3) => 0.2
TimeNode(B, 2) => 0.0
TimeNode(B, 3) => 0.0
TimeNode(B, 1) => 0.202347
TimeNode(C, 1) => 0.0117333
TimeNode(C, 2) => 0.293333
```

Here, for example, TimeNode(A,2) represents active node $(A, 2)$.

The overall score of a node is the sum of scores of the node at different time stamps. We use the overall scores to measure the overall importance of nodes in an evolving graph. Here are the overall scores of the three nodes.

```
"A" => 0.666667
"B" => 0.202347
"C" => 0.305067
```

Notice that A is the most important node in the evolving graph and C is the second most important node. Thus the ranking agrees with the Katz centrality rankings in Section 5.2.

For Figure 5.3, we have

```
TimeNode(A, 10)  => 0.458333
TimeNode(A, 100) => 0.2
TimeNode(B, 1)   => 0.2
TimeNode(B, 10)  => 0.0
TimeNode(B, 100) => 0.0
TimeNode(C, 1)   => 2.98666e-8
TimeNode(C, 10)  => 0.291667
```

The sum of node scores at each time stamp is

```
"A" => 0.658333
"B" => 0.2
"C" => 0.291667
```

We see the scores of all three nodes are decreased when it takes longer to communicate between different time stamps. In this case, the overall ranking is not affected but in general the communication time between time stamps can change the overall ranking.

Temporal Resolvent Betweenness Centrality

Betweenness centrality measures the importance of a node as the ability to facilitate the communication among other nodes in the network. Commonly it is defined based on the shortest paths. However, key messages do not necessarily follow the shortest paths and thus it makes sense to consider walks instead of shortest paths. Recall that the (i, j) entry of the resolvent matrix $(I - \alpha A)^{-1}$ provides information about the communication from i to j. By considering the difference in rating with and without edges linking to node v, we could define the resolvent betweenness for node v as

$$\sum_i \sum_j \frac{(I - \alpha A)^{-1}_{i,j} - (I - \alpha(A - E_v))^{-1}_{i,j}}{(I - \alpha A)^{-1}_{i,j}}, \quad i \neq j, i \neq v, j \neq v. \tag{5.3}$$

where E_v has non-zeros only in row and column v, and in the vth row and column has 1 where A has 1. The matrix $A - E_i$ represents a graph with all edges involving the node v removed. For $(I - \alpha A)^{-1}_{i,j}$ we could simply modify Listing 5.1 so that it only accumulates scores when the end node is j. For the $(I - \alpha(A - E_v))^{-1}_{i,j}$ part, we can apply the same algorithm on a graph with node v and related edges removed. Therefore the general temporal resolvent betweenness centrality can be defined by the algorithm in Listing 5.2.

```
function temporal_resolvent_betweenness(g1, g2,
  v::TimeNode; alpha = 0.2, k = 10)
    r = 0.
    ns = active_nodes(g1)
    for i in ns
        for j in ns
            if i != j && i != v && j != v
                score1 = temporal_katz(g1, i, j,
                    alpha = alpha, k = k)
                score2 = temporal_katz(g2, i, j,
                    alpha = alpha, k = k)
                r += score1 == 0.0 ? 0 :
                    (score1 - score2)/score1
            end
        end
    end
    return r
end
```

Listing 5.2: Temporal Resolvent Betweenness of Single Node

Here temporal_katz only takes account of walks from node i to j. Evolving graph g1 is the original evolving graph and evolving graph g2 is g1 with all the edges related to v removed.

For example, by removing node $(A, 2)$, i.e., node A at time stamp 2, from Figure 5.2 we find the temporal resolvent betweenness score of node A is 5.0. While the temporal resolvent betweenness score of node $(B, 3)$ is 4.0. This shows that $(A, 2)$ is more important than $(B, 3)$. We note that temporal communicability betweenness centrality can be analysed similarly.

5.4.2 Temporal Closeness Centrality

Let d_{ij} be the length of the shortest path from i and j in a static graph. The closeness centrality of a node i is a measure of importance calculated by considering all the shortest paths between node i and all other nodes in the graph, that is

$$C_i^{closeness} = \frac{N-1}{\sum_j d_{ij}},$$

where $N-1$ is the total number of reachable nodes. The more important a node is, the closer it is to all other nodes. For evolving graphs, we need to consider the shortest temporal path from (i, t_i) to (j, t_j), for any time stamp $t_i \leq t_j$. Depending on how we define the length of shortest temporal path, the closeness centrality can counts static edges, causal edges, or both. The key difference between Katz centrality and closeness centrality is the fact here we only consider shortest temporal path, not every temporal walk. We could apply BFS to find the shortest temporal path from (v, t) to any other nodes in the evolving graphs. The algorithm is shown as the algorithm given in Listing 5.3.

```
function temporal_closeness(g::AbstractEvolvingGraph,
  start::TimeNode)
  v = start
  level = Dict(v => 0)
  i = 1
  fronter = [v]
  while length(fronter) > 0
      next = []
      for u in fronter
          for v in forward_neighbors(g, u)
              if !(v in keys(level))
                  if node_key(u) != node_key(v)
                    td = temporal_distance(start, u)
                    level[v] = i + td
                  else
                    level[v] = i - 1
                  end
                  push!(next, v)
              end
          end
      end
      fronter = next
      i += 1
  end

  total_scores = sum(values(level))
  return total_scores > 0.0 ?
      (length(level) - 1)/total_scores : 0.0
```

```
end
```

Listing 5.3: Temporal Closeness of Single Node

The algorithm starts at node start and explores the forward neighbours and store all the reached nodes so far in level. If node_key(u) == node_key(v), we traverse the same node at different time stamps. In this case, we consider that we've already reached it at the previous time stamp.

For Figure 5.2 the scores are

```
TimeNode(A, 2) => 0.666667
TimeNode(A, 3) => 1.0
TimeNode(B, 1) => 0.346154
TimeNode(B, 2) => 0.0
TimeNode(B, 3) => 0.0
TimeNode(C, 1) => 0.315789
TimeNode(C, 2) => 0.5
```

and the sum of scores at different time stamps gives

```
"A" => 1.66667
"B" => 0.346154
"C" => 0.815789
```

Note that the ranking is the same as the temporal Katz centrality case. For Figure 5.3, the scores are

```
TimeNode(A, 10)  => 0.0707071
TimeNode(A, 100) => 1.0
TimeNode(B, 1)   => 0.0597015
TimeNode(B, 10)  => 0.0
TimeNode(B, 100) => 0.0
TimeNode(C, 1)   => 0.0390625
TimeNode(C, 10)  => 0.0412371
```

and the sums are

```
"A" => 1.07071
"B" => 0.0597015
"C" => 0.0802996
```

Similar to the temporal Katz centrality case, we see a decrease of centrality scores for all the nodes in the evolving graph. We note that the analysis for temporal betweenness centrality is similar to above.

5.4.3 Temporal PageRank

We see from our analysis for temporal Katz centrality and temporal betweenness centrality that the temporal distance between nodes in an evolving graph can impact the final centrality scores. We would like to design a temporal PageRank algorithm that takes account of the temporal distance and the spatial distance. The generalisation is based on two ideas: reversed temporal walks and block adjacency matrix.

Reversed Temporal Walks

We can not travel back in time. But reversing the direction of temporal flows can help identify the "hub" nodes or the source of the flows. Recall in the HITS algorithm [53], the hub score describes the quality of a web page as a link collection of important related pages. In reverse PageRank we compute PageRank on the graph with reversed direction, i.e., reverse the direction of each edge (i, j) to (j, i). Fogaras [30, 33] shows that Reversed Page Rank scores express hub quality.

Recall a time-preserving walk is a time-ordered sequence of active nodes. We define a *reversed temporal walk* to be a reversed time-ordered sequence of active nodes $\langle (v_1, t_1), (v_2, t_2), \ldots, (v_m, t_m) \rangle$, where $t_1 \geq t_2 \geq \cdots \geq t_m$.

Block Adjacency Matrices

Aggregating edges at each time stamp to form a simple static graph can provide misleading information about the structure of the network. In [15], we derive a block adjacency matrix representation of an evolving graph and show it is possible to represent an evolvig graph as a block adjacency matrix. Let E' be the set of causal edges and $\hat{E}$ be the set of static edges. Then an evolving graph can be represented as

$$\boldsymbol{M}_n = \begin{pmatrix} A^{[t_1]} & M^{[t_1,t_2]} & \ldots & M^{[t_1,t_n]} \\ 0 & A^{[t_2]} & \ldots & M^{[t_2,t_n]} \\ & \ldots & & \\ 0 & 0 & \ldots & A^{[t_n]} \end{pmatrix},$$

where $M^{[t_i,t_j]}$ is the matrix whose rows are labeled by $V^{[t_i]}$ and columns are labeled by $V^{[t_j]}$, and whose entries are

$$M_{uv}^{[t_i,t_j]} = \begin{cases} 1 & \text{if}(u,v) \in E' \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix blocks $A^{[t]}$ encode the static edge set $\hat{E}$, whereas the off-diagonal blocks $M^{[t_i,t_j]}$ encode the causal edge set E'. To take account of the temporal distance between nodes, we could let

$$M_{uv}^{[t_i,t_j]} = \begin{cases} \frac{1}{|t_j - t_i|} & \text{if}(u,v) \in E' \\ 0 & \text{otherwise.} \end{cases}$$

For Figure 5.4, we have

$$V = \{(1,1998),(2,1998),(1,1999),(3,1999),(2,2018),(3,2018)\},$$
$$\tilde{E} = \{((1,1998),(2,1998)),((1,1999),(3,1999)),((2,2018),(3,2018))\},$$
$$E' = \{((1,1998),(1,1999)),((2,1999),(2,2018)),((3,1999),(3,2018))\}$$

and corresponding block adjacency matrix is

$$\boldsymbol{M}_3 = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/9 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/9 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The static graph representation of Figure 5.4 is shown in Figure 5.5.

Notice that the lower triangular blocks of matrix $\boldsymbol{M}_n$ are zero. This is because all the causal edges are time preserving. We consider the transpose of matrix $\boldsymbol{M}_n$, which reverse the direction of static edges and causal edges. Let $b \in [0,1]$ be a balance scalar. Then the block matrix

$$b\boldsymbol{M}_n + (1-b)\boldsymbol{M}_n^T$$

takes account of both the temporal walks and reversed temporal walks. We now derive the block Google matrix as

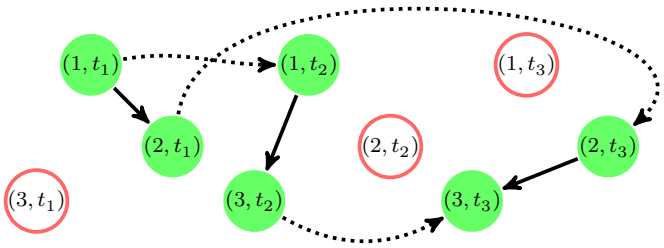


Figure 5.5: A static graph corresponding to the evolving graph example of Figure 5.4. The black lines are edges in the static edge set $\tilde{E}$ and are encoded algebraically in the diagonal blocks $A^{[t]}$ of the adjacency matrix $\boldsymbol{M}_3$. The dotted lines are edges in the causal edge set E' and are encoded algebraically in the off-diagonal blocks $M^{[t_i,t_j]}$. The graph containing all the edges and temporal nodes has adjacency matrix $\boldsymbol{M}_3$.

```
function block_google_matrix(g; alpha = 0.85, balance = 0.75)
  M = full(block_adjacency_matrix(g))
  M = balance * M + (1-balance) * M'
  N, N = size(M)
  p = ones(N)/N
  dangling_weights = p
  dangling_nodes = find(x->x==0, sum(M,2))
  for node in dangling_nodes
      M[node,:] = dangling_weights
  end
  M ./= sum(M,2)
  return alpha * M .+ (1- alpha) * p
end
```

Listing 5.4: Construct a block Google matrix from an evolving graph g.

The function block_adjacency_matrix converts an evolving graph g to a block adjacency matrix $\boldsymbol{M}$. Every row sum of the block Google matrix is equal to one. Each element of the dominant eigenvector of the block Google matrix represents the PageRank rating of a node. The rank of each node can be computed iteratively from the block Google matrix using the power method. For Figure 5.2, we let *alpha* be 0.85 and *balance* be 1.0. Then the PageRank scores are

```
TimeNode(A, 2) => 0.289123
TimeNode(A, 3) => 0.128054
TimeNode(B, 1) => 0.182477
```

```
TimeNode(B, 2) => 0.354335
TimeNode(B, 3) => 0.480941
TimeNode(C, 1) => 0.128054
TimeNode(C, 2) => 0.250931
```

and the sum of node scores at all three time stamps are

```
"A" => 0.417177
"B" => 1.01775
"C" => 0.378986
```

We see node B is the most important node in an evolving graph. On the other extreme, if we set *balance* to be 0.0, we get the overall scores

```
"A" => 0.678502
"B" => 0.647353
"C" => 0.527927
```

Thus A is the most important node in this case.

One potential problem with temporal PageRank is that the computational cost is expensive for large evolving graphs with many time stamps. The dimension of the block matrix is $|V| \times |T|$, where $|V|$ represents the number of nodes and $|T|$ represents the number of time stamps. In practice, we do not need to form or store the dense Google matrix. The power method can be applied to the extremely sparse block adjacency matrix $\boldsymbol{M}$ [54].

5.5 Experiments

We conduct the following experiments using graph type IntAdjacencyList from EvolvingGraphs.jl, which represents an evolving graph g with integer nodes and integer time stamps as adjacency lists. In particular, each active node in g stores its forward neighbours in a list. We count both static edges and causal edges. The following experiments study the influence of temporal distance on centrality scores. All experiments are conducted on a MacOS system with 8 GB of RAM running at 2.3 GHz clock speed.

5.5.1 Random Evolving Graphs

We generate a random evolving graph g with 500 nodes and 499183 static edges and 10 time stamps. We focus on studying the changes of rating of the following nodes. The second element in each tuple is the Katz Centrality rating computed using (5.1).

```
(459, 0.126296)
(147, 0.116541)
(326, 0.113667)
(183, 0.108705)
(469, 0.0936847)
(296, 0.0925748)
(424, 0.0921004)
(377, 0.091973)
(313, 0.0910352)
(127, 0.089123)
```

For each node in the above list, we compute the node score at each time stamp and the overall score. Here we set $alpha = 0.5$ and $max\_level = 3$. The results are shown in Table 5.1 and Table 5.2. Table 5.1 shows the ratings where the causal edges all have edge weight zero. Table 5.2 shows the ratings where the weights of causal edges are their temporal differences. For each table we notice that ratings (and rankings) at the first time stamp t_1 are very different from the overall ratings (and rankings). Comparing the two tables, we observe that the Katz ratings in general are smaller when we take account of the temporal information. The rankings, however, are stable. For rankings at time stamp 1, only the top two nodes 459 and 469 change positions. The overall rankings stays unchanged.

For the same random evolving graph g, we also compute temporal closeness centrality scores. The results are shown in Table 5.3 and Table 5.4. As before, for both tables the ratings (and rankings) at the first time stamp t_1 are very different from the overall ratings (and rankings). However, when we compare the two tables Table 5.3 and 5.4, we notice that for temporal closeness centrality most of rankings are changed when we take account of the temporal information.

Ranking at t_1	Rating at t_1	Overall ranking	Overall rating
469	0.7193	183	5.975
459	0.7180	469	5.932
424	0.7108	326	5.804
296	0.7042	313	5.797
326	0.6853	459	5.788
183	0.6647	147	5.782
377	0.6622	377	5.691
127	0.6316	296	5.681
147	0.6303	424	5.636
313	0.5715	127	5.613

Table 5.1: Temporal Katz centrality ranking and rating on evolving graph g. We set the weight of all causal edges to zero.

Ranking at t_1	Rating at t_1	Overall ranking	Overall rating
459	0.6290	183	5.576
469	0.6281	469	5.533
424	0.6236	326	5.414
296	0.6167	313	5.409
326	0.5983	459	5.401
183	0.5778	147	5.396
377	0.5775	377	5.311
127	0.5492	296	5.299
147	0.5471	424	5.258
313	0.4916	127	5.237

Table 5.2: Temporal Katz centrality ranking and rating on evolving graph g. We set the weight of a causal edge between two active nodes to be their temporal difference. We use bold font to highlight the differences in ranking from Table 5.1.

5.6 Conclusion

When we consider evolving graph centrality, it is very useful to consider the influence of temporal information. A time-preserving path contains both static edges and causal edges. We study the influence of causal edges on graph centrality by varying the edge weights. We derive the centrality algorithms from the view of traffic flow.

We observe that temporal Katz centrality is stable when we vary the causal edge weights. However the rankings of nodes computed by temporal closeness centrality changes a lot in our experiment. Further experiments especially on larger real world datasets are needed to confirm our observation. We also derive a temporal PageRank

Ranking at t_1	Rating at t_1	Overall ranking	Overall rating
296	0.4938	469	5.040
469	0.4933	183	5.030
459	0.4915	313	5.016
424	0.4888	326	5.009
326	0.4879	147	5.008
377	0.4872	459	5.003
183	0.4858	296	4.002
127	0.4831	377	4.996
147	0.4790	127	4.989
313	0.4773	424	4.986

Table 5.3: Temporal closeness centrality ranking and rating of evolving graph g. We set the weight of all causal edges to zero.

Ranking at t_1	Rating at t_1	Overall ranking	Overall rating
296	0.2540	469	3.562
469	0.2487	**147**	3.557
377	0.2478	**183**	3.555
459	0.2475	**313**	3.548
424	0.2471	**377**	3.545
326	0.2470	**296**	3.541
127	0.2445	**459**	3.540
183	0.2422	**127**	3.538
313	0.2389	**424**	3.535
147	0.2373	**326**	3.533

Table 5.4: Temporal closeness centrality ranking and rating of evolving graph g. We set the weight of a causal edge between two active nodes to be their temporal difference. We use bold font to highlight the differences in ranking from Table 5.3.

centrality algorithm by considering reversed temporal walks and block adjacency matrix. It would be interesting to compare the ratings (and rankings) computed by this temporal PageRank algorithm on an evolving graph with the original PageRank on the corresponding aggregated static graph.

Part II

Project Etymo

Chapter 6

Etymo: A New Discovery Engine for AI Research

6.1 Introduction

The rapid growth of global scientific output creates new challenges for information retrieval. The problem is particularly acute in AI (artificial intelligence) research. ArXiv (https://arxiv.og), for example, gains around 500 new AI-related papers every week and the number is growing. As a result, it is difficult for researchers to keep up-to-date with the latest developments in AI research. We have built a new discovery engine for scholarly research called Etymo that addresses this challenge.

Citations of scientific papers are generally considered an important indicator of a paper's impact [48]. Google Scholar's ranking algorithm is not publicly known, but research [5] has shown that citation counts have the highest weighting in its ranking algorithm. However, recent publications have few or no citations so it is difficult to use citations to judge the importance of very recent papers; thus, recent insightful publications are difficult to rank. Our idea is to build a similarity-based network and use this information for information retrieval tasks.

How can we obtain links in a non-hypertext setting? Our approach is to infer links from the distributed vector representation of the full-text papers, i.e., if the cosine similarity between the vector representations is large, we link these two papers. It turns out that inferred links are similar to the citation network because papers talking about the same subject tend to cite one another. However, the analogy between

hyperlinks and generated links is not perfect. In particular, auto-generated links are a noisier source of information and much more prone to spam.

He et al. [44] proposed a meta-approach called HICODE (HIdden COmmunity DEtection) for discovering the hidden community structure in a network. By removing certain edges from the network (weakening the strength of certain structures), one can uncover other structures in the network. Similarly, we can strengthen and weaken the connectivity of the network structure so as to improve our ranking algorithm and filter out unwanted papers. We do this by exploiting papers' social media activities (such as the number of retweets on Twitter) and certain information from user feedback. The resulting graph is used for ranking, recommendation, and visualisation. Note also that inferred links can be generated faster than citations (papers may take 1 year to be cited, but inferred links can be generated almost instantly).

We use a combination of PageRank and Reverse PageRank for ranking papers, which we find gives better search results than pure PageRank or HITS [53]. In Reverse PageRank, we compute PageRank on the graph with reversed direction, i.e., reverse the direction of each edge (i, j) to (j, i). Fogaras [30] shows that Reversed Page Rank scores express hub quality. We have also designed and implemented a new search interface where we display search results as a combination of a list and relationship visualisation. This new interface allows readers to quickly locate relevant and related papers.

Our Etymo discovery engine provides a way to evaluate new research papers by exploiting the full-text of research papers. It challenges the traditional list-based search interface by combining an item list with item relationship visualisation. Etymo updates its database on a daily basis and is free to use for all (demo available at https://etymo.io). Registered users can also write notes and receive recommendations.

6.2 Related Works

Recently, researchers have realised that the full text of scientific papers is an important resource for search and other applications. Indeed, Salatino et al. [68] use the semantic enhanced topic network (where nodes are topics and edges are their co-occurrences in a sample of publications) to identify the appearance of new topics. Sateli et al. [70]

analyze full-text research articles to generate semantic user profiles. Semantic Scholar (https://www.semanticscholar.org) [1] is a search engine for scholarly research that analyzes and links key information from the full text of research papers for improving search results. Similarly, Etymo makes use of the full text of papers to generate a similarity-based network, which is then used for information retrieval tasks.

6.3 Architecture Overview

The dependency graph of Etymo is shown in Figure 6.1. Etymo has several crawlers for downloading research papers from different journal websites. For each paper in our database, we store both the PDF version of the paper and the metadata, including author name, journal name, paper abstract, and the date of publication.

In the Analysis stage, we convert all the PDFs to text using pdftotext [2]. We then apply Doc2Vec [57] and TF-IDF [60, Chap 6] to represent a document d as numeric vectors $v_{Doc2Vec}(d)$ and $v_{tf-idf}(d)$ respectively. Both algorithms represent a paper by a numeric vector such that papers with similar content are close to each other in the vector space. This content similarity information is then used for building a similarity-based network of all the papers in the database. We generate two networks using the two alorithms and find in practice a combination of the two networks can product better results than just using one of the two. We use t-SNE to find the paper locations and network centrality algorithms for the paper ranking. We also generate a lexicon from the TF-IDF's global term weights, which is later used in search. The main components of Etymo consist of a search engine and a feed engine. Results from both engines are displayed as a list and graph visualisation.

6.4 System Features

Etymo has two important features that help it produce useful search results. First, it uses a document vector representation of the full-text papers to build a similarity-based network, where papers are nodes and similar papers are linked. This network

[1] https://techcrunch.com/2016/11/11/scientists-gain-a-versatile-modern-search-engine-with-the-ai-powered-semantic-scholar/

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

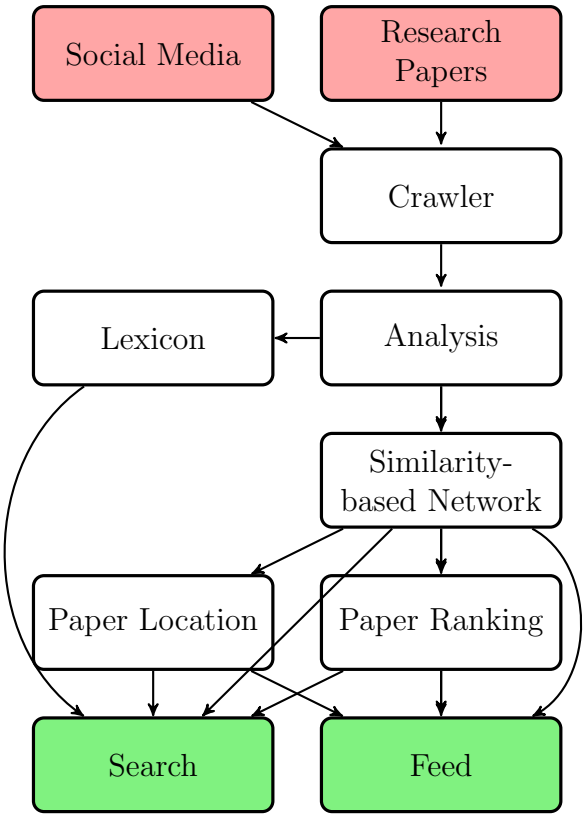


Figure 6.1: The dependency graph of Etymo. The direction of an arrow indicates the dependence of the two components connected by it. The red rectangles represent our data source; the green rectangles are the two main components of Etymo.

is adaptive because of a user feedback mechanism: users' stars, clicks and Twitter mentions are used to reinforce the 'correct' connections and weaken the 'unimportant' ones. The resulting network is then used for ranking and recommendation. Second, we have designed and implemented a novel search interface, with results presented both in a traditional item list and with a visualisation showing paper relationships in order to help users quickly find related papers and have a general idea of a research area.

6.4.1 Similarity-based Network

To construct the similarity-based network, one first needs to represent documents using numeric vectors. We use a distributed representation of the documents called Doc2Vec [57] and a bag of words model called TF-IDF. We then construct the similarity-based network using the cosine similarity measure. One potential problem with this similarity-based network approach is that it does not distinguish a high quality paper from a bad one. We argue that we can use user feedback to adjust the network structure in order to give important papers higher weights.

Adaptive Network and Ranking

Each paper is a node in the network and similar papers are linked together. We compute the cosine similarity of every pair of paper vectors in the database. If the cosine similarity score of two papers' vector representations is larger than a given threshold, we link these two papers. In other words, if $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^n$ (where $n = 1000$ in practice) are the vector representation of two papers p_u and p_v, we define

$$\text{cosine\_similarity}(u, v) = \frac{\sum_{i=1}^{n} u_i v_i}{\|u\|\|v\|},$$

where $\|u\| = \left(\sum_i u_i^2\right)^{1/2}$. If $\text{cosine\_similarity}(u, v) = w > \alpha$, where α is a threshold, we link paper p_u and p_v by an edge with weight w.

Calculating the similarity scores of two papers has a time complexity $O(n)$. Adding a new paper when there are already m papers in the network therefore costs $O(mn)$, which is clearly prohibitive for large m. One potential solution is to calculate a new paper's similarity on a representative subset of the existing papers, i.e., find top k high quality papers, where $k \ll n$. Since our graph centrality ranking provides a measure of

paper quality, we use the top k high ranking papers as a representative subset. Then for a new paper, we only calculate its similarity with these k papers.

Similarity-based networks are vulnerable to spam. For example, if a paper contains a large number of important key words in AI research, it may have a high connectivity on the network hence a high score in the ranking. We use authors' votes to adjust the structure of the network in three main ways:

1. We use user stars to increase the edge weights to a node, this increases the number of edges to that node. In other words, a paper with many user stars connects to more papers than a paper with few.

2. We use user libraries to infer connectivities: increasing edge weights between the papers in a user's library.

3. We weaken the connectivities of a highly ranking paper if it has poor click rates.

Finally, we turn the undirected similarity-based network into a directed network using the temporal information from the paper published date. As a result, a new paper on this network 'recommends' a old paper if they have similar content or user data suggests that they are related. Intuitively we attempt to predict the citation network structure of new research papers when their citations are not available.

6.4.2 Go Beyond List: Relationship Visualisation

A few recent scholar engines incorporate some form of visualisation in displaying their search results. AMiner (https://aminer.org/)[75] shows similar authors and the ego network, which consists of a centre node ("ego") and the nodes to whom the ego node is directly connected to. AceMap [74] displays the citation relationships between academic papers on a map, in a similar way to Google Maps. What we do differently is that we provide a combination of the traditional list of results with a content similarity-based relationship visualisation. Figure 6.2 shows the web interface of Etymo.

Why do we need a new interface? The most important reason is that it saves our time in finding interesting research papers. The information we usually need is the top ten papers from the search results and the papers related to them, but there is no easy way to access all of that information at once using the traditional list interface.

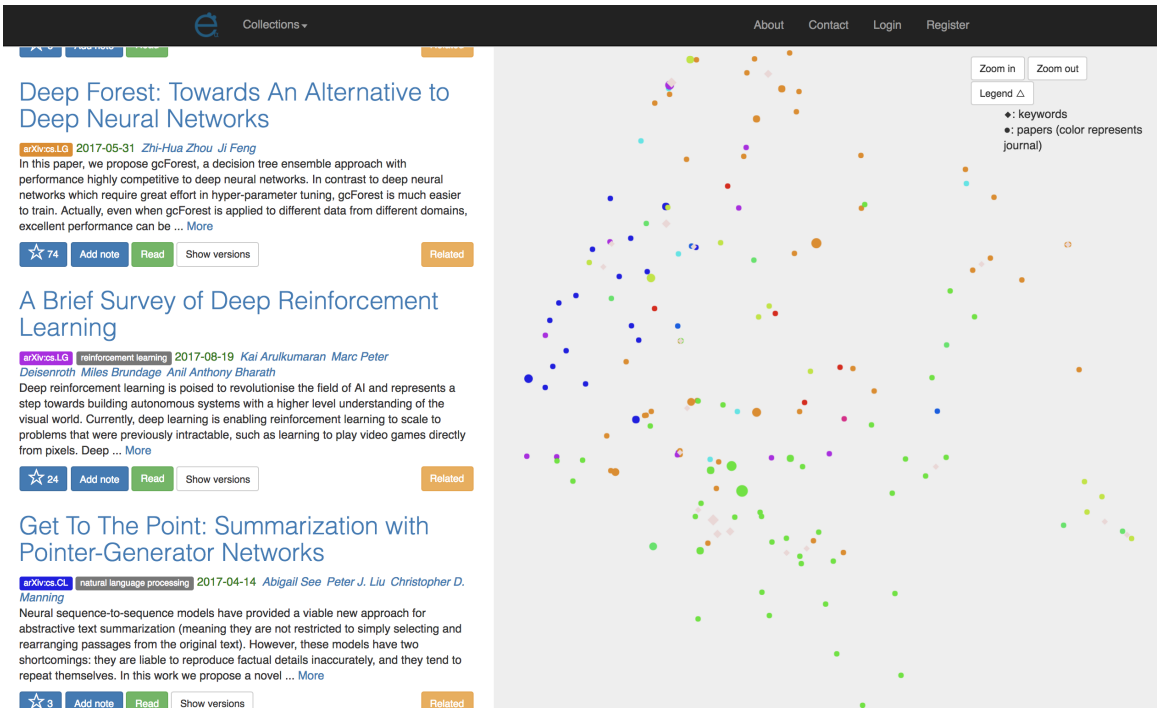


Figure 6.2: The web interface of Etymo. The left part is a list of search results ordered by importance, while the right part is the corresponding visualisation of each paper in the list. Each paper is represented by a node and related papers are close to each other. The size of the node represents the importance and colour represents journal.

For example, to see related papers of the current search result in Google Scholar, a user needs to click on multiple 'related article' links. In Etymo, a user can check the top ten rated papers on the list and locate all the related papers on the graph at the same time.

6.5 Experiments

Etymo has over 36000 papers in the database and we typically add 500 new papers every week. The Analysis uses an instance of Amazon Elastic Compute Cloud (Amazon EC2) m4.xlarge, which has 16 vCPUs and 64GB memory.

We update our database on a daily basis. During each update, we need to first find two sets of vector representations for all the newly added papers using Doc2Vec and TF-IDF. Training of both models are done on a weekly basis. We then use t-SNE to find the x,y location of all the papers, i.e., reduce these 1000 dimensional vectors to 2 dimensional vectors. The computation of T-SNE can be done in $O(N \log N)$ and requires $O(N)$ memory [80], which makes it possible to learn embedding of data sets

Table 6.1: Top 5 search results of the search query "t-sne". The results include a combination of PageRank and Reverse PageRank ratings.

Authors	Year	Title
Laurens van der Maaten, Geoffrey Hinton	2008	Visualizing Data using t-SNE
Laurens van der Maaten	2014	Accelerating t-SNE using Tree-Based Algorithms
Yanshuai Cao, Luyu Wang	2017	Automatic Selection of t-SNE Perplexity
George C. Linderman, Manas Rachh, Jeremy G. Hoskins, Stefan Steinerberger, Yuval Kluger	2017	Efficient Algorithms for t-distributed Stochastic Neighborhood Embedding
Yukun Chen, Jianbo Ye, Jia Li	2017	Aggregated Wasserstein Metric and State Registration for Hidden Markov Models

with millions of objects. The number of nodes in our similarity-based network is equal to the number of papers in our database. We apply PageRank and Reverse PageRank on this network. The predominant method for computing the PageRank is the power method. At each iteration, we do a sparse matrix vector multiplication, which has complexity $O(pN)$, where p is the average number of non-zero elements on every row of the matrix and N is the dimension of the matrix. Usually 10 iterations can produce a good approximate ranking [55, Chap 8.2].

In general, we found that network-based ratings can improve search results by highlighting historically important papers. For the query "t-sne" (a popular machine learning algorithm for dimensionality reduction) we show the top 5 search results in the tables below. Table 6.1 shows the results which include the PageRank and Reverse PageRank ratings on the similarity-based network, while Table 6.2 does not. We noticed that the one with network-based ratings gives more weight to important papers. Note that Maaten and Hinton's "Visualizing Data using t-SNE" is the original t-SNE paper. Comparing with Google Scholar's search results in Table 6.3, Etymo's top search results include more recent publications.

Table 6.2: Top 5 search results of the search query "t-sne". The results do not include any network-based ratings.

Authors	Year	Title
Yanshuai Cao, Luyu Wang	2017	Automatic Selection of t-SNE Perplexity
Laurens van der Maaten	2014	Accelerating t-SNE using Tree-Based Algorithms
Maaten, Laurens van der, Hinton, Geoffrey	2008	Visualizing Data using t-SNE
Richard R. Yang, Mike Borowczak	2017	Assessing Retail Employee Risk Through Unsupervised Learning Techniques
Martin Renqiang Min, Hongyu Guo, Dinghan Shen	2017	Parametric t-Distributed Stochastic Exemplar-centered Embedding

Table 6.3: Top 5 search results of the search query "t-sne" in Google Scholar.

Authors	Year	Title
Maaten, Laurens van der, Hinton, Geoffrey	2008	Visualizing Data using t-SNE
Laurens van der Maaten	2014	Accelerating t-SNE using Tree-Based Algorithms
AR Jamieson ML Giger, K Drukker, H Li	2010	Exploring nonlinear feature space dimension reduction and data representation in breast CADx with Laplacian eigenmaps and t-SNE
K Bunte, S Hasse, M Biehl, T Villmann	2012	Stochastic neighbor embedding (SNE) for dimension reduction and visualization using arbitrary divergences
Maaten, Laurens van der, Hinton, Geoffrey	2008	[PDF] Visualizing Data using t-SNE

6.6 Conclusion

It is hard to quantify new things. In research, the value of a newly published work is usually unknown until citations become available. The value of this work is to provide a new approach to improve search results on new papers by exploiting the paper's full text content and social media data. We illustrate how we build a similarity-based network of research papers and how to adjust its structure using social data. We can foresee similar ideas can be applied in other areas, such as news and fashion. Our new user interface combines the item list with item relationship visualisation, which reveals relationships between papers that saves researchers time in finding interesting research papers.

Chapter 7

Evolving Knowledge Graphs for Idea Tracking in Research Literature

7.1 Introduction

With the proliferation of scientific output, we are now far beyond our ability to find and process all the relevant research literature that is important to us. To solve this problem we need a digital assistant that can not only extract useful ideas and relationships from research articles but can also present the findings in a comprehensible way.

7.1.1 Motivation

New ideas are connected to the knowledge we already possess. As Marvin Minsky, one of the pioneers of intelligence-based robotics, put it [63]:

> "It seems to me that what we call 'creativity' is not simply an ability to generate completely novel conceptions; for a new idea to be useful to us, we must be able to combine it with the knowledge and skills we already possess – so it must not be too different from ideas with which we're already familiar."

To conduct research, we must understand existing knowledge. This process can typically be summarised in three stages:

1. Look for interesting and relevant research papers.
2. Understand ideas from research papers.
3. Track the development of ideas from research papers.

Our key objective is to design a system that can reduce the time needed in all three stages: we automatically extract concepts from the research literature, link these concepts in a knowledge graph, and present the results using a combination of item lists and data visualisation.

7.1.2 Knowledge Graph

Recent years have witnessed a rapid increase of large-scale knowledge bases, both from academia and industry, including DBpedia [2], Wikipedia, Freebase [10], YAGO [73], NELL [14], Knowledge Vault [22], DeepDive [65], Microsoft's Satori, and Google's Knowledge Graph. Structured knowledge of entities has become a critical component of many applications. For example, in internet search, a knowledge graph can help users to research a topic faster and in greater depth. In product recommendation, a knowledge graph can answer questions about products and related products. The knowledge graphs mentioned above focus mostly on static information. Recently Gottschalk and Demidova proposed a temporal knowledge graph model called EventKG [34]. EventKG aggregates event-centric information and temporal relations for historical and contemporary events. Temporal relations are relations valid over a certain time period.

We propose a concept-centric knowledge graph model, called an *evolving knowledge graph*, where a concept at a given time stamp is treated as a unique entity. The key insight is that concepts are time dependent and it is important to record the change of relationships throughout history. Our evolving knowledge graph is a combination of a concepts evolving graph and a content similarity-based graph. Concepts evolving graphs show us how concepts are connected and content similarity-based graphs show us how papers are related. Our evolving knowledge graph also tells us how concepts and papers are connected.

A concepts evolving graph is an ordered sequence of static graphs. Within a given time interval, a concepts evolving graph records the connections among concepts. We extract concepts and their relationship from research articles. Each paper can be viewed as a concept graph of the understanding of the author of a research field. We use Rapid Automatic Keyword Extraction (RAKE) to extract key concepts from research articles and use Word2Vec to filter out very similar words, such as "evolving graph" and "evolving graphs". We also use inverse document frequency (IDF) to assign a hierarchy score for each concept. A concept with higher IDF score is a more general concept. We link two concepts if they are similar according to the Word2Vec cosine similarity or if the two concepts are separated by very few words, i.e., appear in the same sentence or paragraph. We then aggregate these individual paper concept graphs if they are published during the same time interval. A concepts evolving graph is an ordered sequence of these concept graphs at different time stamps.

A content similarity-based graph is a graph of research papers, where each research paper is represented by a vector of real numbers. We use Doc2Vec to compute the vector representation of papers and cosine similarity to determine if we should link two research papers. The direction of a link between two papers is determined by the published date of the two papers: old papers point to new papers as a recommendation. The content similarity-based graph is also related to how we rate (and rank) research articles.

New research papers have no citations. Since graph-based ranking algorithm relies on inward links (as recommendations), we can not use graph-based ranking algorithm on citation networks to rank new papers. We use a combination of PageRank and Reverse PageRank on the content similarity-based graph for ranking papers, as we find that it gives better search results than pure PageRank or HITS [53]. Fogaras [30] shows that Reversed Page Rank scores express hub quality. Note that reverse PageRank computes PageRank on the similarity-based graph with reversed direction. Exploiting online users' votes (such as Etymo stars, Twitter mentions and retweets and Facebook posts), one can adjust the structure of the similarity-based graph (change edge weights) so that rankings computed from similarity-based graph agree with users' votes. We also use the paper vector representations to find different versions of the same paper. If the cosine similarity between two papers is very close to one (> 0.99)

we consider the two papers to be different versions of the same paper. In practice, this works very well and is more robust than rule-based methods.

In order to demonstrate the power of our knowledge graph, we have developed a new search system, called Etymo (https://scholar.etymo.io). Etymo has a novel search interface, where we display search results as a combination of an item list and a relationship visualization. This new interface allows fast retrieval of relevant papers. We compute the relationship visualization at run time using T-SNE on the pre-computed paper content vectors.

7.1.3 Contributions

We introduce here a new concept-centric knowledge graph model, called an evolving knowledge graph. We take inspiration from the evolving graph model to model the change of connectivity of concepts as an evolving graph. Furthermore, we have developed a full-feature search system to demonstrate the use case of an evolving knowledge graph. The search system is particularly good at finding interesting new papers and can enable users to track the development of ideas by following concepts. This is much more powerful than traditional search system for research papers, where users can only track ideas by following citations. We've made part of our database publicly available for research purposes. Researchers can find paper metadata and graph data at Etymo OpenData (https://github.com/EtymoIO/OpenData).

7.2 Related Work

There have been many recent developments in search systems for research articles. Semantic Scholar (https://www.semanticscholar.org) [1] is a search engine for scholarly research that analyzes and links key information from the full text of research papers to improve search results. For a given search query, Semantic Scholar shows a list of related concepts below the search box. ArXiv-Sanity (https://github.com/karpathy/arxiv-sanity-preserver) is a search tool for finding AI related papers on ArXiv that makes use of the full text of research papers for recommending papers and suggesting

[1] https://techcrunch.com/2016/11/11/scientists-gain-a-versatile-modern-search-engine-with-the-ai-powered-semantic-scholar/

related papers. A few recent scholar engines incorporate some form of visualisation in displaying their search results. Web of Science (http://wok.mimas.ac.uk/) provides a citation map to allow researchers to visualise citation connections. AMiner (https://aminer.org/)[75] shows the connections of an author in a social network. For example https://aminer.org/profile/geoffrey-hinton show the social graph of Geoffrey Hinton. ArXivTimes (https://arxivtimes.herokuapp.com/) uses histograms to display trends in AI research. AceMap [74] displays the citation relationships between academic papers on a map, in a similar way to Google Maps.

7.3 Concepts Evolving Graph

A research article contains an author's understanding of a subject. Even though many concepts and definitions are universally agreed, different authors have different interpretations of how concepts are connected. In particular, the concepts in an article tell us what an author thinks is important in a given research topic. This is related to the observation by Licklider and Taylor [58] that

> "Each member of the project brings to such a meeting a somewhat different mental model of the common undertaking – its purposes, its goals, its plans, its progress, and its status. They are strongly influenced by insight, subjective feelings, and educated guesses."

If we extract concepts from an article and link them in a graph structure, for example, from the more general concepts, things like "data science" or "linear algebra" to more specific concepts like "recursive neural network (RNN)" or "polar decomposition" and link related concepts, such as link "neural network" and "deep learning", we obtain a graph that represents an author's understanding of a subject. We could then combine these small personal knowledge graph to make a collective knowledge graph.

How can we extract and link concepts from a research article? The whole process can be divided into five steps. Here are the details:

1. IDF scores. We convert a collection of raw documents to a term-frequency inverse-document-frequency (TFIDF) matrix. The number of words (and terms)

in our vocabulary (also known as feature size) is 10000. Each word in our vocabulary is assigned a IDF score. The term frequency (TF) of a term t in document d, denoted by $tf_{t,d}$, is the number of occurrences of term t in document d. And the inverse document frequency (IDF) of a term t, denoted by idf_t is defined as

$$idf_t = \log \frac{N}{df_t},$$

where N is the number of documents in a collection and df_t is the number of documents that contain a term t. The TFIDF score of a term t in document d is given by

$$\text{tf\_idf}_{t,f} = tf_{t,d} \times idf_t.$$

2. Word2Vec vectors. We run Word2vec on all the paper articles to get vector representations of words (and phrases). Word2Vec goes through each word of the whole corpus and predicts surrounding words of each word.

3. RAKE algorithm. We apply the RAKE algorithm to find all possible concept words. In order to use the RAKE algorithm, we need to construct a stop word data set. Stop words are used to break down sentences into possible concepts. We build our stop word data set using a combination of generic English stop words such as “a” and “the” and TFIDF stop words, which are terms that occurred in too many or too few documents in Step 1. Choosing stop words is an art: too many stops words will break almost all concepts into words, while too few stops words will include sentences or half of a sentence in the concepts. We decide the size of stop word set using a heuristic approach.

4. Similar concept filter. The vector representation of a concept is the sum of its word Word2Vec representations. We filter out very similar concepts using the cosine similarity score between a pair of concept vectors. Very similar words such as “evolving graph” and “evolving graphs” have very high cosine similarity scores.

5. Concept Links. We link similar concepts in two stages. First, we link similar concepts according to their vector cosine similarity. Second, we link concepts

that appear in the same sentence or short paragraph. The position of concepts in text is an important indicator of whether two concepts are related.

We use the method above to find the concept graph of a paper. We then aggregate all the paper concept graphs published during the same period. In a fast growing field such as AI research, we aggregate graphs on a monthly basis. Hence at each time stamp t (a givne month), we have a concept graph $G^{[t]}$. The concepts evolving graph is an ordered sequence of all the concept graphs $G^{[t]}$ at different time stamps.

7.4 Content Similarity-Based Graph

To construct a similarity-based graph, one first needs to represent papers using numeric vectors. We use a distributed representation of papers called Doc2Vec [57]. Dai et al. found that Doc2Vec performs better than other methods for research paper data sets [18]. We link two papers according to the cosine similarity score of the corresponding vector representations. One potential problem with this similarity-based graph approach is that it does not distinguish a high quality paper from a bad one. We argue that we can adaptively change the graph structure by using user feedback to adjust the connectivity of cetrain nodes. A similarity-based graph has a number of applications, including ranking papers, finding related papers, and supporting run-time visualization of paper relationships.

We consider each paper as a node in the similarity-based graph and similar papers are linked together by edges. We calculate the cosine similarity of every pair of papers in the database. If the cosine similarity score of two papers' vector representations is larger than a given threshold, we link these two papers. In other words, if we let $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^n$ be the vector representation of two papers p_u and p_v, we have

$$\text{cosine\_similarity}(u, v) = \frac{\sum_{i=1}^{n} u_i v_i}{\|u\|\|v\|},$$

where $\|u\| = \left(\sum_i u_i^2\right)^{1/2}$. If $\text{cosine\_similarity}(u, v) = w > \alpha$, where α is a threshold, we link paper p_u and p_v by an edge with weight w. We turn an undirected similarity-based graph into a directed graph using the temporal information from the paper published date: an old paper points to a new paper. Intuitively, it means that an old paper "recommends" a new paper if they have similar content.

In practice, we update our database on a daily basis as new papers are processed every day. We need to finish the computation within a few hours (we currently use AWS EC2 m4.xlarge). Calculating the similarity scores of two papers has a time complexity $O(n)$. Adding a new paper when there are already m papers in the graph therefore costs $O(mn)$, which is clearly prohibitive for large m. We calculate a new paper's similarity on a representative subset of all the existing papers by finding k important papers in the similarity-based graph using graph centrality, where $k \ll n$. Then for a new paper, we only calculate its similarity with these k papers.

Similarity-based graphs are vulnerable to spam. For example, if a paper contains a large number of important key words in AI research, it may have a high connectivity on the graph and hence a high score in the ranking. We use users' votes to adjust the structure of the graph so that high ranked papers agree with users' votes. Here is how we do it:

1. Strengthen connectivity. Recall in practice we only calculate a paper's similarity on a representative subset of all the existing papers. For papers with many votes from users, we recompute its cosine similarity scores with more papers other than in the subset and hence increase the connectivity of the paper node in the graph. This increases the PageRank (in-link) and Reverse PageRank (out-link) scores of highly voted papers.

2. Weaken connectivity. For high ranked papers with poor click rates we randomly removing links connecting it. Similarly, this decreases the connectivity of the paper node in the graph and hence decreases the PageRank and Reverse PageRank scores of papers with poor click rates.

7.5 Evolving Knowledge Graph

Knowledge can be represented as a quadruple: (subject, predicate, object, timestamp). For example, consider the fact that the PageRank algorithm was mentioned by a paper called "The PageRank Citation Ranking" in 1999. This can be represented as a quadruple (PageRank, MentionedInPaper, "The PageRank Citation Ranking", 1999). Here we see the relationship between a concept and a paper. Formally, we can represent an evolving knowledge graph as a time-dependent multi-graph $G_n\{G^{[1]}, G^{[2]}, \ldots, G^{[n]}\}$,

where each $G^{[t]} = (V^{[t]}, E^{[t]})$ is a multi-graph. Here $V^{[t]}$ represents a set of nodes at time stamp t and $E^{[t]} \subseteq V^{[t]} \times V^{[t]}$ represents a set of edges at time stamp t. Note that there may be multiple labelled edges between the same pair of nodes. The fact that the PageRank algorithm was mentioned by a paper called "The PageRank Citation Ranking" in 1999 can be represented as an edge from a node representing the concept "PageRank" to a node representing the paper "The PageRank Citation Ranking" at time stamp 1999. Note that each paper in an evolving knowledge graph is unique but the same concept can appear in multiple time stamps as concepts can have different meanings at different time stamps. A evolving knowledge graph takes information from both the concepts evolving graph and the content similarity-based graph. Currently our evolving knowledge graph stores the following relationships:

1. the connection between two concepts at the same or different time stamps,
2. the connection between two distinct papers,
3. the connection between a concept and a paper.

This is illustrated in Figure 7.1.

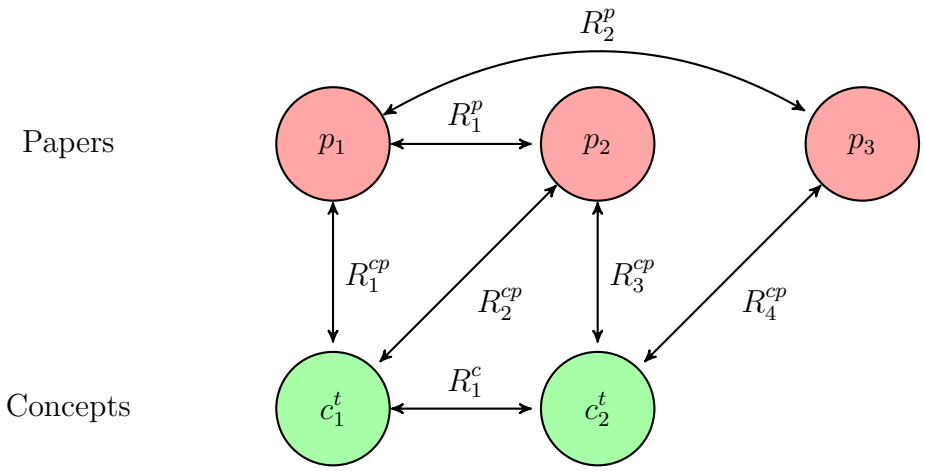


Figure 7.1: Each circle represents a node on the knowledge graph. The lines labelled R_i^p, R_i^c, R_i^{cp} are edges, which represent the connection between two papers, two concepts, and a concept and a paper respectively.

7.5.1 A Hierarchy of Representations

We've described several kinds of structures that we could use to represent various types of knowledge. Now we put everything together. The rudimentary representations are

numerical vectors. We represent papers and words as vectors of the same dimension. In Section 7.6, we show how we can combine word vectors with paper vectors to get better visualization.

The concepts evolving graph and the content similarity graph take information from word vectors and paper vectors to create knowledge about how concepts are connected at different time stamps and how papers are connected. Finally, we combine concepts evolving graphs and content similarity graph to create knowledge about how a concept and a paper are related. This is illustrated in Figure 7.2.

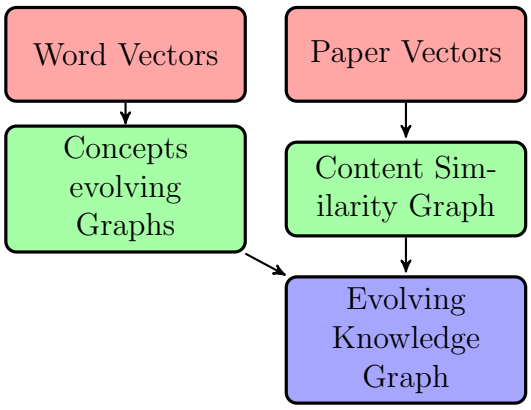


Figure 7.2: A hierarchy of knowledge representations. Word vectors and paper vectors are rudimentary representations. Concepts Evolving Graphs and Content Similarity Graph are higher level representations. Evolving Knowledge Graph is the highest level representation of knowledge.

7.6 Data Visualization

In this section, we discuss how we use run-time data visualization to help researchers to navigate a large collection of research papers. We improve the traditional search interface (item lists) to display search results as a combination of item list and relationship visualization.

Why do we need a new interface? Most importantly because we need to reduce the time it takes a user to find relevant research papers and obtain a basic understanding of a paper before reading it. Usually, the papers we need are the top ten ranked ones and all their related papers. Current available search interfaces do not provide an easy way to access this information. In Google Scholar, for example, a user needs to click 'related articles' to see related papers of a given paper. To gain a picture of the top

papers and all their related papers, a user needs to click ‘related articles‘ several times and write down notes simultaneously.

In Etymo, a user can look at the top ten ranked papers on the item list and locate all the related papers on the knowledge map at the same time. See Figure 7.3 for a quick look at Etymo's web interface.

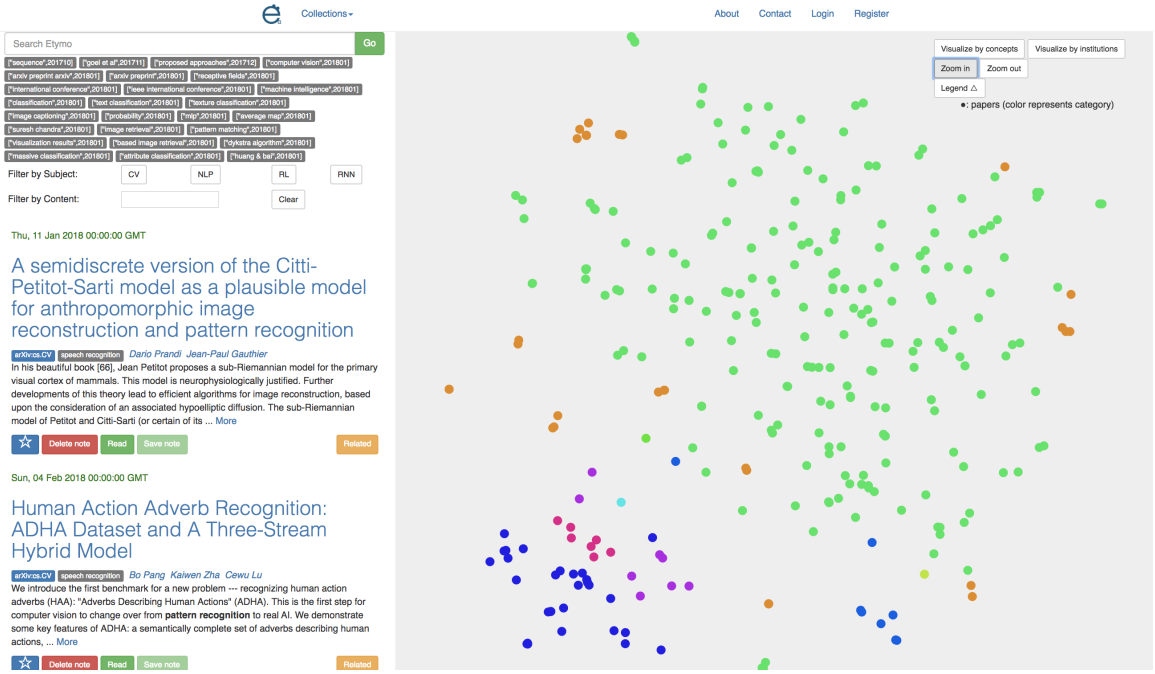


Figure 7.3: Etymo's web interface for the search query “pattern recognition”. The right-hand side of the interface shows the visualization of related papers that are clustered by concepts. Each paper is represented by a node and related papers are close to each other. The left-hand side is the list view of the search results. The top part of the list shows related concepts at different time stamps.

Another advantage of visualization is that it can help reveal interesting structures within a research field, allowing us to see patterns that would otherwise be invisible. Figure 7.4 shows the visualization of papers related to the search query “deep learning”. Each color represents a subject. From the figure, we can get a sense of roughly how many recent and interesting papers there are in each subject. For example, we see the majority of papers are about “transfer learning”, which are represented by blue circles. The locations of different paper clusters are also interesting. For example, the three bottom right clusters (dark blue, orange, and yellow) are computer vision, audio detection, speech recognition respectively. These are the three main applications of deep learning. Patterns like this are particularly useful for researchers who are new to the filed.

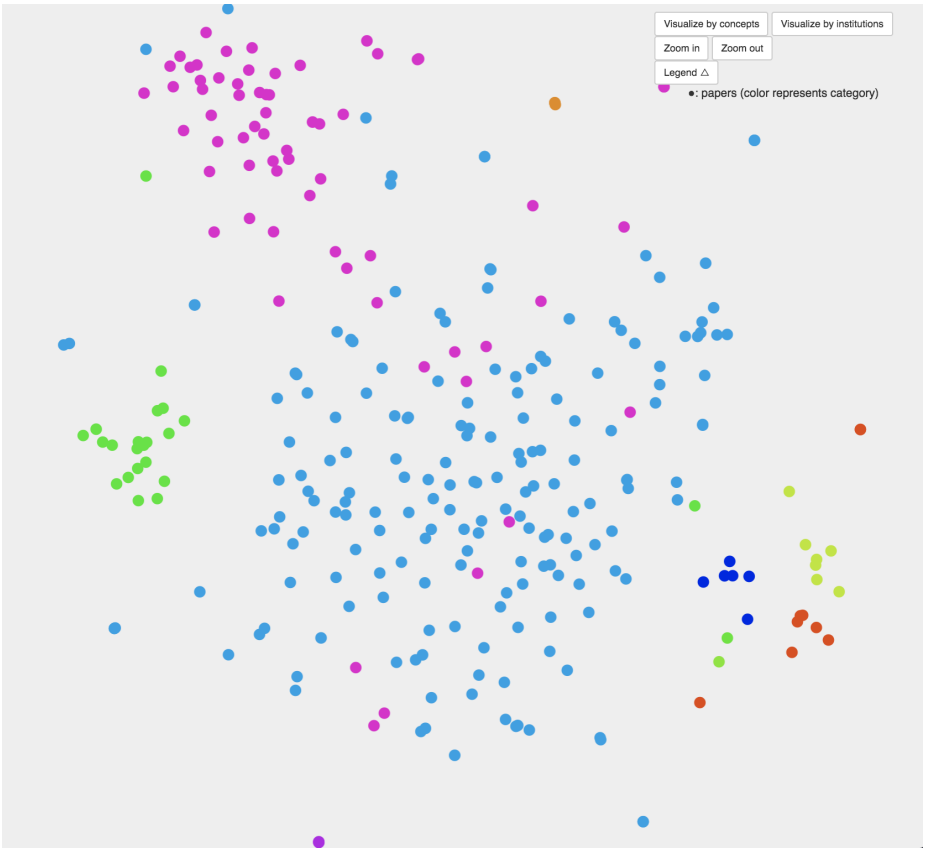


Figure 7.4: Etymo's web interface for the search query "deep learning". Each circle represents a paper and different colors represent different subject. For example, the circles in pink color are about reinforcement learning; the circles in blue color are about transfer learning; the circles in green color are about convolutional neural network.

We pre-compute the content vectors for all the papers (or global content vectors). At run time, when a user searches for a query or selects a paper collection, we run t-Distributed Stochastic Neighbor Embedding (T-SNE) [59] on the content vectors of selected papers (or local T-SNE), which projects the high dimensional vectors onto a two dimensional plane. We found in practice this generates better visualization than pre-computing T-SNE for all the paper content vectors (or global T-SNE). Since at run time we are only interested in a small subset of all the papers, the distance between vectors is not properly scaled (can be too wide or too narrow) for the global T-SNE case. In Figure 7.5, we highlight the most relevant search results to the search query "convolution" among 3500 papers. This illustrates the fact that the projected representations of our queried papers can come from different clusters in global T-SNE. Thus global T-SNE will not be able to provide the best visualisation for a subset of papers. Also we noticed that global content vector followed by local T-SNE and local

content vectors followed by local T-SNE gives very similar visualization. However, the former is cheaper to run at run time.

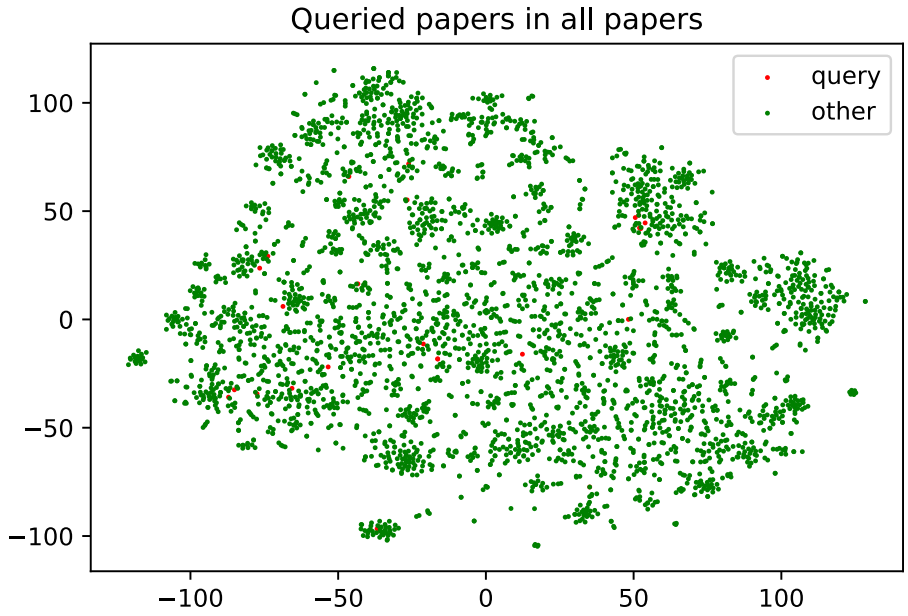


Figure 7.5: The T-SNE projection of 3500 paper content vectors. Each circle represents a paper and similar papers are close to each other. We hightlight the top 40 ranked papers that are related to search query "convolution" in red. The rest of papers are shown in green.

We compare the three cases: global content vectors followed by global T-SNE, global content vectors followed by local T-SNE, and local content vectors followed by local T-SNE. In our experiment, we use TF-IDF to compute the content vectors for 3500 papers. The results are shown in figure 7.6.

We use the vector representation of key concepts to adjust (or influence) the two dimensional projection so that papers with similar concepts are close to each other. If we let v_{paper} be the paper vector and $v_{concept}$ be the concept vector, then the adjusted vector is a convex linear combination of these two vectors

$$v_{adjusted} = (1 - \alpha)v_{paper} + \alpha v_{concept},$$

where α is a scalar between zero and one representing the amount of influence. For each concept, the vector representation is the sum of Word2Vec vector representations of all the words in the concept. In Figure 7.7, we show that the adjusted paper visualization forms better clusters than the one without adjustment.

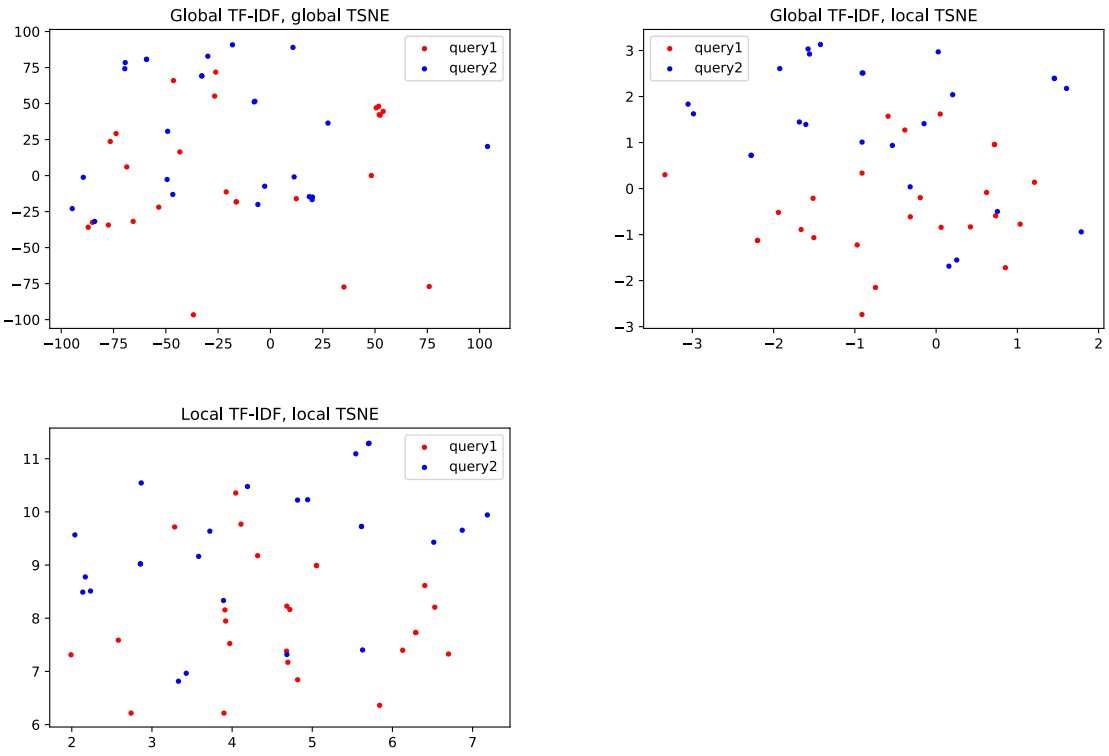


Figure 7.6: Comparing three cases of paper visualization: global content vectors followed by global T-SNE, global content vectors followed by local T-SNE, and local content vectors followed by local T-SNE. In the first case, the distances between circles can be too wide or too narrow. The second and third cases show similar results. But the third case is cheaper to compute at run time.

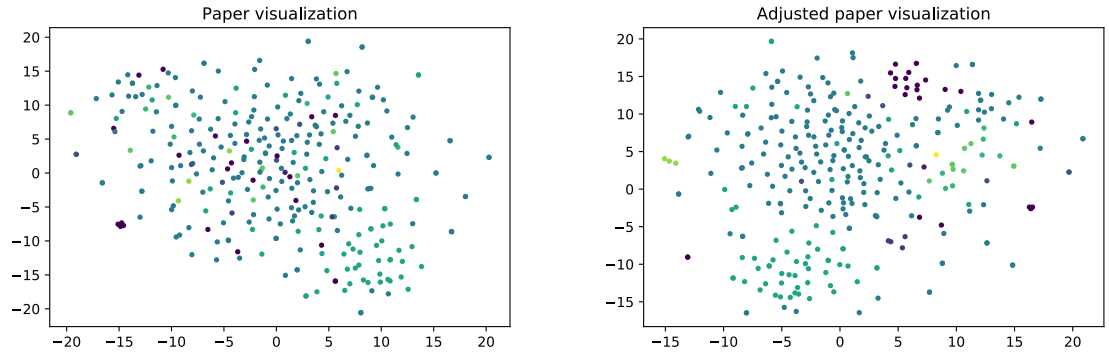


Figure 7.7: Comparing paper visualization with adjusted paper visualization using the concept vector. Here the influence scalar α is set to be 0.95.

7.7 Etymo Architecture Overview

In this section, we give a high level overview of how the whole Etymo system works. There are five main components: data crawler, database, data analysis, application programming interface (API), and web interface, as illustrated in Figure 7.8. Most of

Etymo is implemented in Python with Numpy and Scipy. Our web app is implemented in JavaScript and part of our data analysis code is implemented in Julia.

Our crawlers periodically check journal websites and company research pages. If a paper URL is not in our database, we fetch the paper metedata and paper PDFs (if permitted) and then send them to the store server. At this stage, we also convert paper PDFs to text and store the paper full texts as well. Each paper in our database has a unique paper_id. Data analysis interacts with the database and updates the following information: paper rating scores, related papers for each paper, paper collections, paper vector representations. The structure of the data analysis part is illustrated in Figure 7.9. Users can visit the Etymo web interface on the web, mobile, or the desktop, which connects the database via our API. The web interface uses Angular [19], a JavaScript framework which allows Etymo to dynamically generate HTML pages. The graph visualisations are produced by Sigma.js [2].

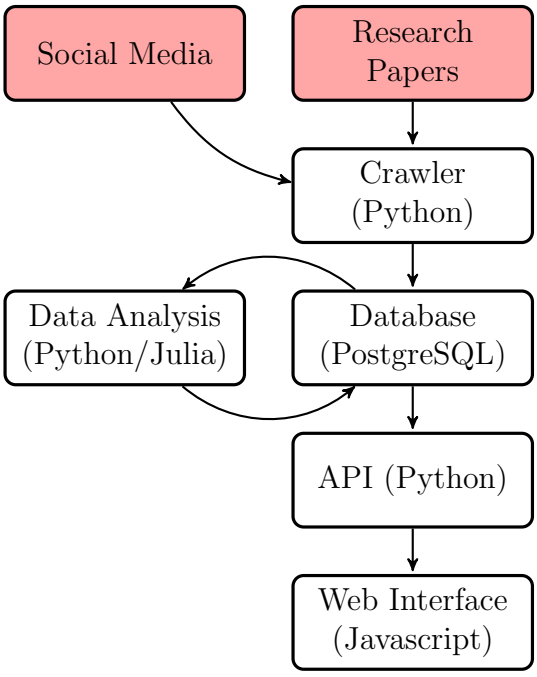


Figure 7.8: A high level overview of Etymo's key components. The direction of an arrow indicates the flow of data between components in Etymo. The red rectangles represent our data source.

In the data analysis stage, we apply Doc2Vec [57] and TF-IDF to represent a document d as numeric vectors $v_{Doc2Vec}(d)$ and $v_{tfidf}(d)$ respectively. This content similarity information is then used for building a similarity-based graph of all the

[2]https://github.com/jacomyal/sigma.js

papers in the database. We also extract concepts from research papers and build an evolving concept graph. The Evolving Knowledge Graph is a combination of similarity-based graph and evolving concept graph. We use t-SNE, a dimensionality reduction algorithm, to find the paper locations and network centrality algorithms for the paper ranking. We also generate a lexicon from the TF-IDF's global term weights, which is later used in search. Both our search engine and paper collection engine display results as a list and/or graph visualization.

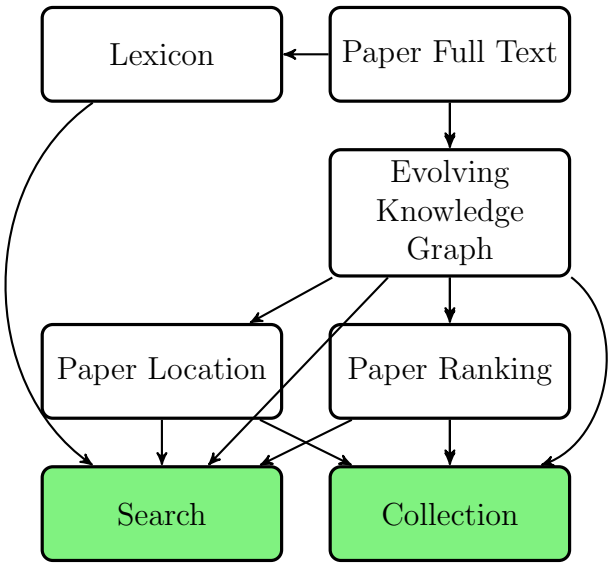


Figure 7.9: The workflow of data analysis in Etymo. The direction of an arrow indicates the flow of data. The green rectangles are Etymo APIs. Our evolving knowledge graph is a combination of a similarity-based graph and an evolving concept graph.

7.8 Experiments

The most important feature of an evolving knowledge graph is temporal concepts. We explore how this new feature will affect the way we search research articles on the internet. While a complete user evaluation is beyond the scope of this paper, our experiments with Etymo show that the our knowledge graph provide very useful information to help better track ideas.

When a user enters a search query q in the search box, we first check if q is part of our knowledge graph. If so, we display all instances of q at different timestamps. For example, Figure 7.10 shows the search results of query "matrix". We display all

instances of "matrix" at timestamp 201806, 201709 and 201802 just below the search box. Here 201806, for example, means June 2018. When a user clicks an instance of "matrix" (a node in our knowledge graph), we display all the related concepts at different time stamps. (This is equivalent to entering a search query in the form of query + time stamps, such as "matrix + 201806".) For example, Figure 7.11 shows the search results of query "deep learning + 201802". The temporal information tells us when a concept has appeared. Using this information, a user can not only find similar concepts but also track the development of ideas by following concepts over time. We note that the interface is still immature and further development is needed to make it more user friendly.

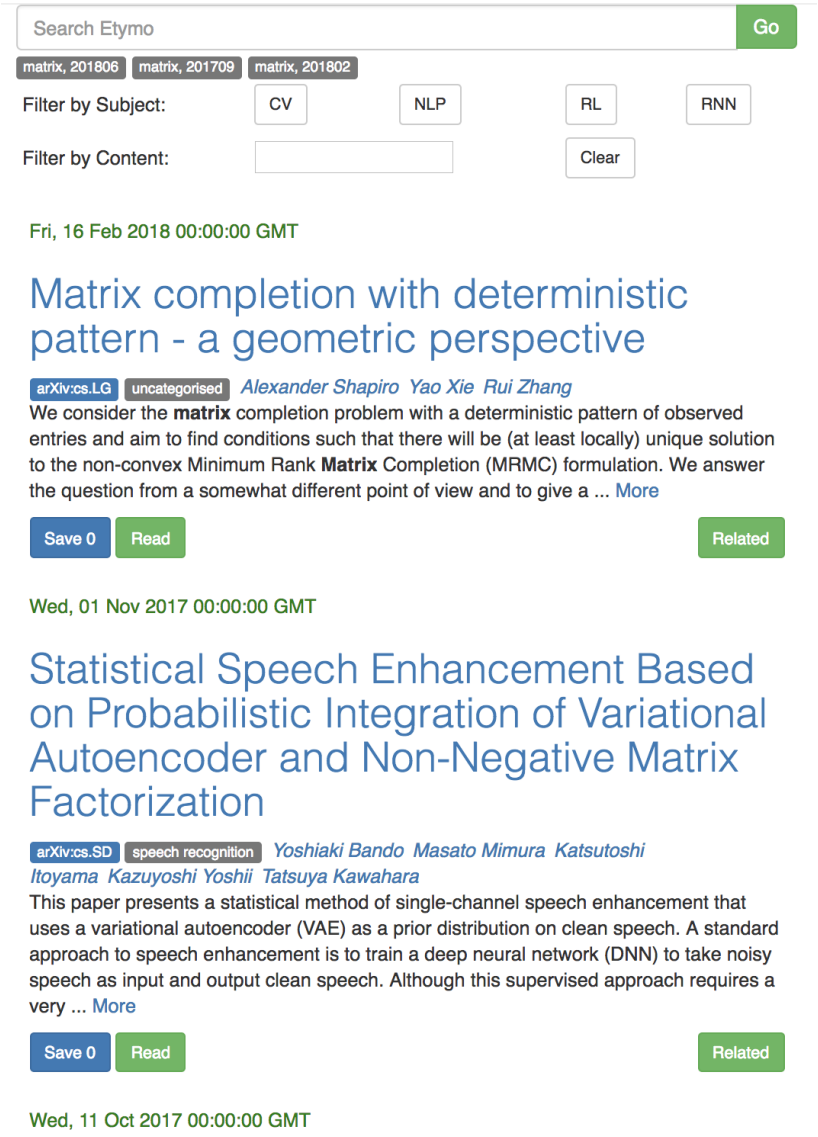


Figure 7.10: Search results of query "matrix". All instances of "matrix" at different time stamps are shown just below the search box.

Figure 7.11: Search results of query "deep learning + 201802". All related concepts at time stamp 201802 are shown just below the search box.

To get an idea of how concepts are linked, we construct concept graphs of small subsets of papers. Figure 7.12 shows the concept graph of a recent KDD (Knowledge Discovery and Data Mining) paper "Recommendations with Negative Feedback via Pairwise Deep Reinforcement Learning". From the figure, we can see that the extracted concepts are meaningful and similar concepts and nearby concepts are connected.

By aggregating concept graphs from the research papers in the same research area, we can get a picture of how concepts are related in general. Figure 7.13 illustrates the aggregated concept graph of five research papers on deep learning. We limit ourselves to five papers because it is hard to understand the visualization with too many nodes

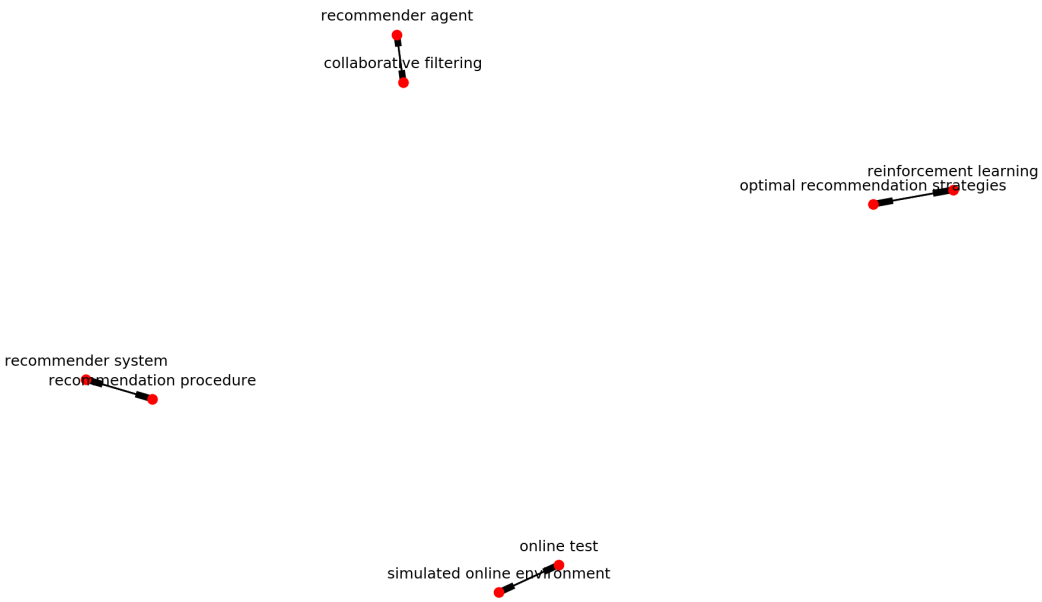


Figure 7.12: The connected concepts from research paper "Recommendations with Negative Feedback via Pairwise Deep Reinforcement Learning". Each red circle represents a concept. Connected concepts are linked are lines.

Concept name	backward neighbours	forward neighbours
(deep neural networks, 201802)	(low rank, 201802), (network, 201802), (deep learning, 201802), (machine learning, 201802), (neural information processing systems, 201802), (networks, 201802), (convergence rate, 201802), (generalization error, 201802), (machine learning models, 201802), (speech, 201802)	(networks, 201802), (microsoft azure machine learning, 201802), (adversarially trained networks, 201802), (representation systems, 201802), (generative model, 201802), (deep learning, 201802), (deep reinforcement learning, 201802)

Table 7.1: The forward and backward neighbours (ignoring corrupted texts) of concept "deep neural networks" at time stamp 201802.

and connections.

In Table 7.1, we show the forward neighbours and backward neighbours of concept ("deep neural networks", 201802) at time stamp 201802. These are the related concepts shown in Figure 7.11. We notice that some concepts are both backward neighbours and forward neighbours of ("deep neural networks", 201802), such as ("networks", 201802) and ("deep learning", 201802). This is because "networks" and "deep learning" are similar concepts of "deep neural networks", and they are connected by undirected edges.

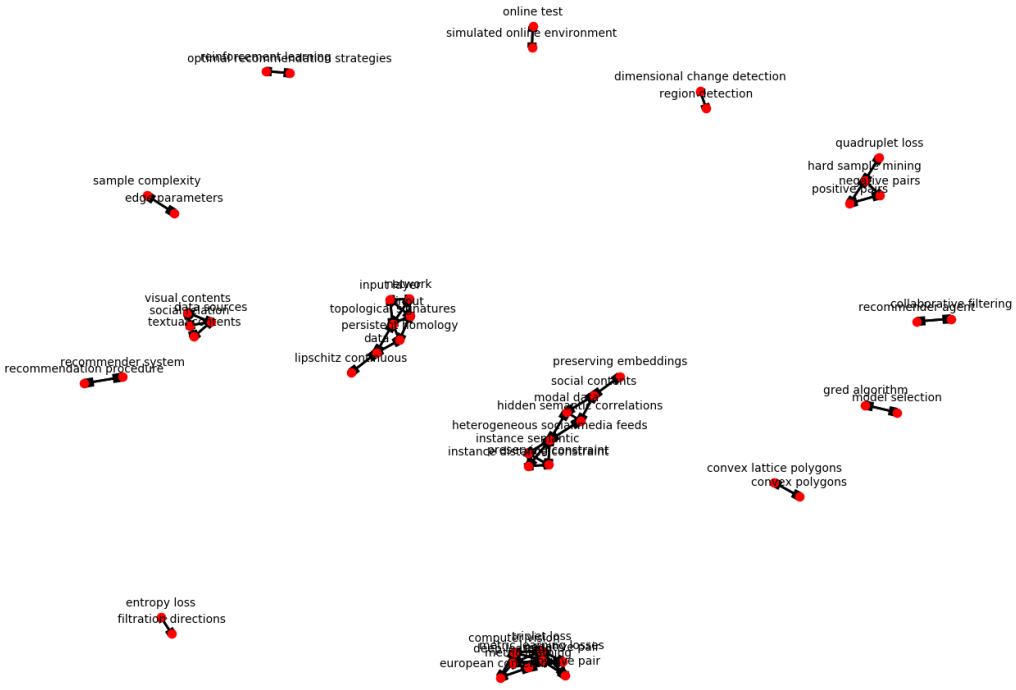


Figure 7.13: The connected concepts from five research papers on deep learning.

7.9 Other Potential Applications

We note that the same approach can be applied in other time-dependent natural language applications. For example, news articles share a number of similarities with research papers: a) mainly text based, b) time-dependent, c) have a central topic. Recent news in general is more important than old news. Users are interested in tracing the source and development of news articles and find out how different news articles are connected.

7.10 Conclusion

Knowledge graphs are active area of research. In this paper, we present Etymo's evolving knowledge graph, a time-dependent concept-centric knowledge graph model. We construct an evolving knowledge graph using a combination of concepts evolving graphs and content similarity-based graphs.

Using an evolving knowledge graph, we are able to enrich users' search experience to help them track the development of concepts. In particular, when a user searches

for a concept, we display a list of similar concepts in different time stamps. One can then click one of them to get all the related papers and other related concepts. In experiments, this turns out to provide richer and useful content for researchers. The intuition behind an evolving knowledge graph is that concepts are time dependent. For example, the concept "artificial intelligence" in 2000 is different from "deep learning" today.

We are still at the early stage of building a large-scale evolving knowledge graph. The number of concepts in our evolving knowledge graphs are still small (around 3000). In the future, we will include more concepts and incorporate knowledge graph data from other sources (such as Wikipeda) to provide more information about each concept. We will also improve the user interface so that it is easy for users to navigate the website.

Part III

Testing Numerical Linear Algebra Algorithms

Chapter 8

Matrix Depot: A Test Collection

8.1 Introduction

In 1969 Gregory and Karney published a book of test matrices [36]. They stated that "In order to test the accuracy of computer programs for solving numerical problems, one needs numerical examples with known solutions. The aim of this monograph is to provide the reader with suitable examples for testing algorithms for finding the inverses, eigenvalues, and eigenvectors of matrix." At that time it was common for journal papers to be devoted to introducing and analyzing a particular test matrix or class of matrices, examples being the papers of Clement [17] (in the first issue of SIAM Review), Pei [66] (occupying just a quarter of a page), and Gear [32].

Today, test matrices remain of great interest, but not for the same reasons as fifty years ago. Testing accuracy using problems with known solutions is less common because a reference solution correct to machine precision can usually be computed at higher precision without difficulty. The main uses of test matrices nowadays are for exploring the behavior of mathematical quantities (such as eigenvalue bounds) and for measuring the performance of one or more algorithms with respect to accuracy, stability, convergence rate, speed, or robustness.

Various collections of matrices have been made available in software. As well as giving easy access to matrices these collections have the advantage of facilitating reproducibility of experiments [23], whether by the same researcher months later or by different researchers.

An early collection of parametrizable matrices was given by Higham [45] and made

available in MATLAB form. The collection was later extended and distributed as a MATLAB toolbox [46]. Many of the matrices in the toolbox were subsequently incorporated into the MATLAB gallery function. Marques, Vömel, Demmel, and Parlett [61] present test matrices for tridiagonal eigenvalue problems (already recognized as important by Gregory and Karney, who devoted the last chapter of their book to such matrices). The Harwell–Boeing collection of sparse matrices [24] has been widely used, and is incorporated in the University of Florida Sparse Matrix Collection [20], [21], which contains over 2700 matrices from practical applications, including standard and generalized eigenvalue problems from [4]. Among other MATLAB toolboxes we mention the CONTEST toolbox [79], which produces adjacency matrices describing random networks, and the NLEVP collection of nonlinear eigenvalue problems [6].

The purpose of this work is to provide a test matrix collection for Julia [7], a new dynamic programming language for technical computing. The collection, called Matrix Depot [1], exploits Julia's multiple dispatch features to enable all matrices to be accessed by one simple interface. Moreover, Matrix Depot is extensible. Users can add matrices from the University of Florida Sparse Matrix Collection [2] and Matrix Market; they can code new matrix generators and incorporate them into Matrix Depot; and they can define new groups of matrices that give easy access to subsets of matrices. The parametrized matrices can be generated in any appropriate numeric data type, such as

- floating-point types Float16 (half precision: 16 bits), Float32 (single precision: 32 bits), and Float64 (double precision: 64 bits);
- integer types Int32 (signed 32-bit integers), UInt32 (unsigned 32-bit integers), Int64 (signed 64-bit integers), and UInt64 (unsigned 32-bit integers);
- Complex, where the real and imaginary parts are of any Real type (the same for both);
- Rational (ratio of integers); and

[1] This paper describes Matrix Depot v0.5.5 and it was tested on Julia 0.4. https://github.com/JuliaMatrices/MatrixDepot.jl/tree/v0.5.5 The latest version is Matrix Depot v0.7.0.

[2] The University of Florida Sparse Matrix Collection has been renamed as The SuiteSparse Matrix Collection.

- arbitrary precision type BigFloat (with default precision 256 bits), which uses the GNU MPFR Library [31].

This paper is organized as follows. We start by giving a brief demonstration of Matrix Depot in Section 8.2. Then we explain the design and implementation of Matrix Depot in Section 8.3, giving details on how multiple dispatch is exploited, how the collection is stored, accessed, and documented, and how it can be extended. In Section 8.4 we describe the two classes of matrices in Matrix Depot: parametrized test matrices and real-life sparse matrix data. Concluding remarks are given in Section 8.5.

8.2 A Taste of Matrix Depot

To download Matrix Depot, in a Julia REPL (read-eval-print loop) run the command

```
> Pkg.add("MatrixDepot")
```

Then import Matrix Depot into the local scope.

```
> using MatrixDepot
```

Now the package is ready to be used. First, we find out what matrices are in Matrix Depot.

```
> matrixdepot()
```

```
Matrices:
 1) baart          2) binomial       3) blur          4) cauchy
 5) chebspec       6) chow           7) circul        8) clement
 9) companion     10) deriv2        11) dingdong     12) fiedler
13) forsythe      14) foxgood       15) frank        16) golub
17) gravity       18) grcar         19) hadamard     20) hankel
21) heat          22) hilb          23) invhilb      24) invol
25) kahan         26) kms           27) lehmer       28) lotkin
29) magic         30) minij         31) moler        32) neumann
33) oscillate     34) parallax      35) parter       36) pascal
37) pei           38) phillips      39) poisson      40) prolate
41) randcorr      42) rando         43) randsvd      44) rohess
45) rosser        46) sampling      47) shaw         48) spikes
49) toeplitz      50) tridiag       51) triw         52) ursell
53) vand          54) wathen        55) wilkinson    56) wing
```

```
Groups:
  all           data          eigen         ill-cond
  inverse       pos-def       random        regprob
  sparse        symmetric
```

All the matrices and groups in the collection are shown. It is also possible to obtain just the list of matrix names.

```
> matrixdepot("all")
56-element Array{ASCIIString,1}:
 "baart"
 "binomial"
 "blur"
 "cauchy"
 "chebspec"
 "chow"
 "circul"
 "clement"
 "companion"
 "deriv2"
   ...
 "spikes"
 "toeplitz"
 "tridiag"
 "triw"
 "ursell"
 "vand"
 "wathen"
 "wilkinson"
 "wing"
```

Here, "..." denotes that we have omitted some of the output in order to save space. Next, we check the input options of the Hilbert matrix hilb.

```
> matrixdepot("hilb")
     Hilbert matrix
    ================

  The Hilbert matrix has (i,j) element 1/(i+j-1). It is notorious
  for being ill conditioned. It is symmetric positive definite and
```

```
totally positive.

Input options:

  * [type,] dim: the dimension of the matrix.

  * [type,] row_dim, col_dim: the row and column dimensions.

Groups: ["inverse", "ill-cond", "symmetric", "pos-def"]

References:

M. D. Choi, Tricks or treats with the Hilbert matrix, Amer. Math.
Monthly, 90 (1983), pp. 301-312.

N. J. Higham, Accuracy and Stability of Numerical Algorithms,
second edition, Society for Industrial and Applied Mathematics,
Philadelphia, PA, USA, 2002; sec. 28.1.
```

Note that an optional first argument type can be given; it defaults to Float64. The string of equals signs on the third line in the output above is Markdown notation for a header. Julia interprets Markdown within documentation, though as we are using typewriter font for code examples here, we display the uninterpreted source. We generate a 4×6 Hilbert matrix with elements in the default double precision type and then in Rational type.

```
> matrixdepot("hilb", 4, 6)
4x6 Array{Float64,2}:
 1.0       0.5       0.333333  0.25      0.2       0.166667
 0.5       0.333333  0.25      0.2       0.166667  0.142857
 0.333333  0.25      0.2       0.166667  0.142857  0.125
 0.25      0.2       0.166667  0.142857  0.125     0.111111

> matrixdepot("hilb", Rational, 4, 6)
4x6 Array{Rational{T<:Integer},2}:
 1//1  1//2  1//3  1//4  1//5  1//6
 1//2  1//3  1//4  1//5  1//6  1//7
 1//3  1//4  1//5  1//6  1//7  1//8
 1//4  1//5  1//6  1//7  1//8  1//9
```

A list of all the symmetric matrices in the collection is readily obtained.

```
> matrixdepot("symmetric")
21-element Array{ASCIIString,1}:
 "cauchy"
 "circul"
 "clement"
 "dingdong"
 "fiedler"
 "hankel"
 "hilb"
 "invhilb"
 "kms"
 "lehmer"
 "minij"
 "moler"
 "oscillate"
 "pascal"
 "pei"
 "poisson"
 "prolate"
 "randcorr"
 "tridiag"
 "wathen"
 "wilkinson"
```

Here, symmetric is one of several predefined groups, and multiple groups can be intersected. For example, the for loop below prints the smallest and largest eigenvalues of all the 4×4 matrices in Matrix Depot that are symmetric positive definite and (potentially) ill conditioned.

```
> for name in matrixdepot("symmetric", "pos-def", "ill-cond")
      A = full(matrixdepot(name, 4))
      @printf "%9s: smallest eigval = %0.3e, largest eigval = %0.3e\n"
      name eigmin(A) eigmax(A)
  end

   cauchy: smallest eigval = 2.131e-05, largest eigval = 9.776e-01
     hilb: smallest eigval = 9.670e-05, largest eigval = 1.500e+00
  invhilb: smallest eigval = 6.666e-01, largest eigval = 1.034e+04
```

```
      kms: smallest eigval = 3.750e-01, largest eigval = 2.086e+00
    moler: smallest eigval = 3.336e-02, largest eigval = 5.122e+00
oscillate: smallest eigval = 1.490e-08, largest eigval = 1.000e+00
   pascal: smallest eigval = 3.802e-02, largest eigval = 2.630e+01
      pei: smallest eigval = 1.000e+00, largest eigval = 5.000e+00
  tridiag: smallest eigval = 3.820e-01, largest eigval = 3.618e+00
```

Matrices can also be accessed by number within the alphabetical list of matrix names.

```
> matrixdepot(2)
"binomial"

> matrixdepot(2:5)
4-element Array{AbstractString,1}:
 "binomial"
 "blur"
 "cauchy"
 "chebspec"

> matrixdepot(15:20, 5, 6, 1:3)
11-element Array{AbstractString,1}:
 "frank"
 "golub"
 "gravity"
 "grcar"
 "hadamard"
 "hankel"
 "chebspec"
 "chow"
 "baart"
 "binomial"
 "blur"
```

Access by number provides a convenient way to run a test on subsets of matrices in the collection. However, the number assigned to a matrix may change if we include new matrices in the collection. In order to run tests in a way that is repeatable in the future it is best to group matrices into subsets using the macro @addgroup, which stores them by name. For example, the following command will group test matrices

frank, golub, gravity, grcar, hadamard, hankel, chebspec, chow, baart, binomial, and blur into test1.

```
> @addgroup test1 = matrixdepot(15:20, 5, 6, 1:3)
```

After reloading the package, we can run tests on these matrices using group test1. Here we compute the 2-norms. Since blur (an image deblurring test problem) generates a sparse matrix and the matrix 2-norm is currently not implemented for sparse matrices in Julia, we use full to convert the matrix to dense format.

```
> for name in matrixdepot("test1")
      A = full(matrixdepot(name , 4))
      @printf "%9s has 2-norm %0.3e \n" name norm(A)
  end

    baart has 2-norm 3.192e+00
 binomial has 2-norm 4.576e+00
     blur has 2-norm 8.298e-01
 chebspec has 2-norm 6.474e+00
     chow has 2-norm 3.414e+00
    frank has 2-norm 7.624e+00
    golub has 2-norm 2.050e+02
  gravity has 2-norm 6.656e+00
    grcar has 2-norm 2.562e+00
 hadamard has 2-norm 2.000e+00
   hankel has 2-norm 1.160e+01
```

To download the test matrix SNAP/web-Google from the University of Florida sparse matrix collection (see Section 8.4.2 for more details), we first download the data with

```
> matrixdepot("SNAP/web-Google", :get)
```

and then generate the matrix with

```
> matrixdepot("SNAP/web-Google", :r)
916428x916428 sparse matrix with 5105039 Float64 entries:
        [11343 ,      1]  =  1.0
        [11928 ,      1]  =  1.0
        [15902 ,      1]  =  1.0
        [29547 ,      1]  =  1.0
        [30282 ,      1]  =  1.0
```

```
[31301  ,      1]  =  1.0
[38717  ,      1]  =  1.0
...
[720325, 916427]  =  1.0
[772226, 916427]  =  1.0
[785097, 916427]  =  1.0
[788476, 916427]  =  1.0
[822938, 916427]  =  1.0
[833616, 916427]  =  1.0
[417498, 916428]  =  1.0
[843845, 916428]  =  1.0
```

Note that the omission marked "..." was in this case automatically done by Julia based on the height of the terminal window. Matrices loaded in this way are inserted into the list of available matrices, and assigned a number. After downloading further matrices HB/1138˙bus, HB/494˙bus, and Bova/rma10 the list of matrices is as follows.

```
julia> matrixdepot()

Matrices:
 1) baart          2) binomial       3) blur           4) cauchy
 5) chebspec       6) chow           7) circul         8) clement
 9) companion     10) deriv2        11) dingdong      12) fiedler
13) forsythe      14) foxgood       15) frank         16) golub
17) gravity       18) grcar         19) hadamard      20) hankel
21) heat          22) hilb          23) invhilb       24) invol
25) kahan         26) kms           27) lehmer        28) lotkin
29) magic         30) minij         31) moler         32) neumann
33) oscillate     34) parallax      35) parter        36) pascal
37) pei           38) phillips      39) poisson       40) prolate
41) randcorr      42) rando         43) randsvd       44) rohess
45) rosser        46) sampling      47) shaw          48) spikes
49) toeplitz      50) tridiag       51) triw          52) ursell
53) vand          54) wathen        55) wilkinson     56) wing
57) Bova/rma10    58) HB/1138_bus   59) HB/494_bus    60) SNAP/web-Google

Groups:
  all             data            eigen           ill-cond
  inverse         pos-def         random          regprob
```

```
sparse          symmetric       test1
```

8.3 Package Design and Implementation

In this section we describe the design and implementation of Matrix Depot, focusing particularly on the novel aspects of exploitation of multiple dispatch, extensibility of the collection, and user-definable grouping of matrices.

8.3.1 Exploiting Multiple Dispatch

Matrix Depot makes use of multiple dispatch in Julia, an object-oriented paradigm in which the selection of a function implementation is based on the types of each argument of the function. The generic function matrixdepot has eight different methods, where each method itself is a function that handles a specific case. This is neater and more convenient than writing eight "case" statements, as is necessary in many other languages.

```
> methods(matrixdepot)
# 8 methods for generic function "matrixdepot":
matrixdepot() ...
matrixdepot(name::AbstractString) ...
matrixdepot(name::AbstractString, method::Symbol) ...
matrixdepot(props::AbstractString...) ...
matrixdepot(name::AbstractString, args...) ...
matrixdepot(num::Integer) ...
matrixdepot(ur::UnitRange{T<:Real}) ...
matrixdepot(vs::Union{Integer,UnitRange{T<:Real}}...) ...
```

For example, the following two functions are used for accessing matrices by number and range respectively, where matrix_name_list() returns a list of matrix names. The second function calls the first function in the inner loop.

```
function matrixdepot(num::Integer)
    matrixstrings = matrix_name_list()
    n = length(matrixstrings)
    if num > n
        error("There are $(n) parameterized matrices,
               but you asked for the $(num)-th.")
```

```
    end
    return matrixstrings[num]
end

function matrixdepot(ur::UnitRange)
    matrixnamelist = AbstractString[]
    for i in ur
        push!(matrixnamelist, matrixdepot(i))
    end
    return matrixnamelist
end
```

As a result, matrixdepot is a versatile function that can be used for a variety of purposes, including returning matrix information and generating matrices from various input parameters.

In the following example we see how multiple dispatch handles different numbers and types of arguments for the Cauchy matrix.

```
> matrixdepot("cauchy")
    Cauchy matrix
    =============

  Given two vectors x and y, the (i,j) entry of the Cauchy matrix is
  1/(x[i]+y[j]).

  Input options:

    *  [type,] x, y: two vectors.

    *  [type,] x: a vector. y defaults to x.

    *  [type,] dim: the dimension of the matrix. x and y default
     to [1:dim;].

  Groups: ["inverse", "ill-cond", "symmetric", "pos-def"]

  References:

  N. J. Higham, Accuracy and Stability of Numerical Algorithms,
```

```
  second edition, Society for Industrial and Applied Mathematics,
  Philadelphia, PA, USA, 2002; sec. 28.1

> matrixdepot("cauchy", [1, 2, 3], [4, 5, 6])
3x3 Array{Float64,2}:
 0.2       0.166667  0.142857
 0.166667  0.142857  0.125
 0.142857  0.125     0.111111

> matrixdepot("cauchy", [0.2, 0.3, 0.4])
3x3 Array{Float64,2}:
 2.5       2.0       1.66667
 2.0       1.66667   1.42857
 1.66667   1.42857   1.25

> matrixdepot("cauchy", 3)
3x3 Array{Float64,2}:
 0.5       0.333333  0.25
 0.333333  0.25      0.2
 0.25      0.2       0.166667

> matrixdepot("cauchy", Float32, 3)
3x3 Array{Float32,2}:
 0.5       0.333333  0.25
 0.333333  0.25      0.2
 0.25      0.2       0.166667
```

Multiple dispatch is also exploited in programming the matrices. For example, the Hilbert matrix is implemented as

```
function hilb–T″(::Type–T″, m::Integer, n::Integer)
    H = zeros(T, m, n)
    for j = 1:n, i = 1:m
        @inbounds H[i,j] = one(T)/ (i + j - one(T))
    end
    return H
end
hilb–T″(::Type–T″, n::Integer) = hilb(T, n, n)
```

hilb(args...) = hilb(Float64, args...)

The function hilb has three methods, which enable one to request, for example, hilb(4,2) for a 4×2 Hilbert matrix of type Float64, or simply (thanks to the final two lines) hilb(4) for a 4×4 Hilbert matrix of type Float64. The keyword @inbounds tells Julia to turn off bounds checking in the following expression, in order to speed up execution. Note that in Julia it is not necessary to vectorize code to achieve good performance [7].

All the matrices in Matrix Depot can be generated using the function call

matrixdepot("matrix˙name", p1, p2, ...),

where matrix˙name is the name of the test matrix, and p1, p2, . . . , are input arguments depending on matrix˙name. The help comments for each matrix can be viewed by calling function matrixdepot("matrix˙name"). We can access the list of matrix names by number, range, or a mixture of numbers and range.

1. matrixdepot(i) returns the name of the ith matrix;
2. matrixdepot(i:j) returns the names of the ith to jth matrices, where $i < j$;
3. matrixdepot(i:j, k, m) returns the names of the ith, $(i+1)$st, . . ., jth, kth, and mth matrices.

8.3.2 Matrix Representation

Matrix names in Matrix Depot are represented by Julia strings. For example, the Cauchy matrix is represented by "cauchy". Matrix names and matrix groups are stored as hash tables (Dict). In particular, there is a hash table matrixdict that maps each matrix name to its underlying function and a hash table matrixclass that maps each group to its members.

The majority of parametrized matrices are dense matrices of type Array–T,2″, where T is the element type of the matrix. Variables of the Array type are stored in column-major order. A few matrices are stored as sparse matrices (see also matrixdepot("sparse"
in the Compressed Sparse Column (CSC) format; these include neumann (a singular matrix from the discrete Neumann problem) and poisson (a block tridiagonal matrix

Table 8.1: Predefined groups.

Group	Description
all	All the matrices in the collection.
data	The matrix has been downloaded from the University of Florida Sparse Collection or the Matrix Market Collection.
eigen	Part of the eigensystem of the matrix is explicitly known.
ill-cond	The matrix is ill-conditioned for some parameter values.
inverse	The inverse of the matrix is known explicitly.
pos-def	The matrix is positive definite for some parameter values.
random	The matrix has random entries.
regprob	The output is a test problem for regularization methods.
sparse	The matrix is sparse.
symmetric	The matrix is symmetric for some parameter values.

from Poisson's equation). Tridiagonal matrices are stored in the built-in Julia type Tridiagonal which is defined as follows.

```
immutable Tridiagonal{T} <: AbstractMatrix{T}
    dl::Vector{T}     # sub-diagonal
    d::Vector{T}      # diagonal
    du::Vector{T}     # sup-diagonal
    du2::Vector{T}    # supsup-diagonal for pivoting
end
```

8.3.3 Matrix Groups

A group is a subset of matrices in Matrix Depot. There are ten predefined groups, described in Table 8.1, most of which identify matrices with particular properties. Each group is represented by a string. For example, the group of random matrices is represented by "random". Matrices can be accessed by group names, as was illustrated in Section 8.1.

The macro @addgroup is used to add a new group of matrices to Matrix Depot and the macro @rmgroup removes an added group. All the predefined matrix groups are stored in the hash table matrixclass. The macro @addgroup essentially adds a new key-value combination to the hash table usermatrixclass. Using a separate hash table prevents the user from contaminating the predefined matrix groups.

Being able to create groups is a useful feature for reproducible research [23]. For

example, if we have implemented algorithm alg01 and we used circul, minij, and grcar as test matrices for alg01, we could type

```
> @addgroup alg01_group = ["circul", "minij", "grcar"]
```

This adds a new group to Matrix Depot (we need to reload the package to see the changes).

```
julia> matrixdepot()
```

Matrices:

1) baart	2) binomial	3) blur	4) cauchy
5) chebspec	6) chow	7) circul	8) clement
9) companion	10) deriv2	11) dingdong	12) fiedler
13) forsythe	14) foxgood	15) frank	16) golub
17) gravity	18) grcar	19) hadamard	20) hankel
21) heat	22) hilb	23) invhilb	24) invol
25) kahan	26) kms	27) lehmer	28) lotkin
29) magic	30) minij	31) moler	32) neumann
33) oscillate	34) parallax	35) parter	36) pascal
37) pei	38) phillips	39) poisson	40) prolate
41) randcorr	42) rando	43) randsvd	44) rohess
45) rosser	46) sampling	47) shaw	48) spikes
49) toeplitz	50) tridiag	51) triw	52) ursell
53) vand	54) wathen	55) wilkinson	56) wing

Groups:

all	data	eigen	ill-cond
inverse	pos-def	random	regprob
sparse	symmetric	alg01_group	

We can then run alg01 on the test matrices by

```
> for name in matrixdepot(alg01_group)
    A = matrixdepot(name, n)  # n is the dimension of the matrix.
    @printf "Test result for %9s is %0.3e" name alg01(A)
  end
```

8.3.4 Adding New Matrix Generators

Generators are Julia functions that generate test matrices. When Matrix Depot is first loaded, a directory myMatrixDepot is created. It contains two files, group.jl and generator.jl, where group.jl is used for storing all the user defined groups (see Section 8.3.3) and generator.jl is used for storing generator declarations.

Julia packages are simply Git repositories[3]. The directory myMatrixDepot is untracked by Git, so any local changes to files in myMatrixDepot do not make the MatrixDepot package "dirty". In particular, all the newly defined groups or matrix generators will not be affected when we upgrade to a new version of Matrix Depot. Matrix Depot automatically loads all Julia files in myMatrixDepot. This feature allows a user to simply drop generator files into myMatrixDepot without worrying about how to link them to Matrix Depot.

A new generator is declared using the syntax include_generator(FunctionName, "fname", f). This adds the new mapping "fname" $\rightarrow$ f to the hash table matrixdict, which we recall maps each matrix name to its underlying function. Matrix Depot will refer to function f using string "fname" so that we can call function f by matrixdepot("fname"...) The user is free to define new data types and return values of those types. Moreover, as with any Julia function, multiple values can be returned by listing them after the return statement.

For example, suppose we have the following Julia file rand.jl, which contains two generators randsym and randorth and we want to use them from Matrix Depot. The triple quotes in the file delimit the documentation for the functions.

```
"""
random symmetric matrix
=======================

Input options:

* n: the dimension of the matrix
"""
function randsym(n)
  A = zeros(n, n)
```

[3]Git is a free and open source distributed version control system.

```
    for j = 1:n
        for i = 1:j
            A[i,j] = randn()
            if i != j; A[j,i] = A[i,j] end
        end
    end
    return A
end

"""
random orthogonal matrix
========================

Input options:

* n: the dimension of the matrix
"""
randorth(n) = qr(randn(n,n))[1]
```

We can copy the file rand.jl to the directory myMatrixDepot and add the following two lines to generator.jl.

```
include_generator(FunctionName, "randsym", randsym)
include_generator(FunctionName, "randorth", randorth)
```

This includes the functions randsym and randorth in Matrix Depot, as we can see by looking at the matrix list (the new entries are numbered 43 and 45).

```
julia> matrixdepot()

Matrices:
   1) baart          2) binomial       3) blur           4) cauchy
   5) chebspec       6) chow           7) circul         8) clement
   9) companion     10) deriv2        11) dingdong      12) fiedler
  13) forsythe      14) foxgood       15) frank         16) golub
  17) gravity       18) grcar         19) hadamard      20) hankel
  21) heat          22) hilb          23) invhilb       24) invol
  25) kahan         26) kms           27) lehmer        28) lotkin
  29) magic         30) minij         31) moler         32) neumann
  33) oscillate     34) parallax      35) parter        36) pascal
  37) pei           38) phillips      39) poisson       40) prolate
```

```
  41) randcorr      42) rando         43) randorth      44) randsvd
  45) randsym       46) rohess        47) rosser        48) sampling
  49) shaw          50) spikes        51) toeplitz      52) tridiag
  53) triw          54) ursell        55) vand          56) wathen
  57) wilkinson     58) wing
Groups:
  all               data              eigen             ill-cond
  inverse           pos-def           random            regprob
  sparse            symmetric
```

The new generators can be used just like the built-in ones.

```
> matrixdepot("randsym")
      random symmetric matrix
      =======================

     Input options:

      * n: the dimension of the matrix

> matrixdepot("randsym", 4)
4x4 Array{Float64,2}:
 -0.00992523   0.174531   -1.73322    -0.765096
  0.174531     1.69308     0.269062    0.594058
 -1.73322      0.269062   -0.824277   -0.541458
 -0.765096     0.594058   -0.541458   -0.480428

> matrixdepot("randorth")
      random orthogonal matrix
      ========================

     Input options:

      * n: the dimension of the matrix

> A = matrixdepot("randorth", 4)
4x4 Array{Float64,2}:
 -0.233943    0.179893    0.563926   -0.771295
 -0.769649   -0.141938   -0.5807     -0.224235
  0.247165    0.832118   -0.449941   -0.20986
```

```
 -0.540204    0.505046    0.377263    0.557477

> A'*A - eye(4,4)
4x4 Array{Float64,2}:
 -2.22045e-16   1.66533e-16  -2.77556e-17  -1.66533e-16
  1.66533e-16  -1.11022e-16  -3.05311e-16   1.66533e-16
 -2.77556e-17  -3.05311e-16  -1.11022e-16   1.94289e-16
 -1.66533e-16   1.66533e-16   1.94289e-16   0.0
```

We can also add group information with the function include_generator. The following lines are put in generator.jl.

```
include_generator(Group, "random", randsym)
include_generator(Group, "random", randorth)
```

This adds the functions randsym and randorth to the group random, as we can see with the following query (after reloading the package).

```
> matrixdepot("random")
10-element Array{ASCIIString,1}:
 "golub"
 "oscillate"
 "randcorr"
 "rando"
 "randorth"
 "randsvd"
 "randsym"
 "rohess"
 "rosser"
 "wathen"
```

8.3.5 Documentation

The Matrix Depot documentation is created using the documentation generator Sphinx[4] and is hosted at Read the Docs[5]. Its primary goals are to provide examples of usage of Matrix Depot and to give a brief summary of each matrix in the collection. Matrices are listed alphabetically with hyperlinks to the documentation for each matrix. Most parametrized matrices are presented with heat map plots, which are produced using the

[4] http://sphinx-doc.org/
[5] http://matrixdepotjl.readthedocs.org

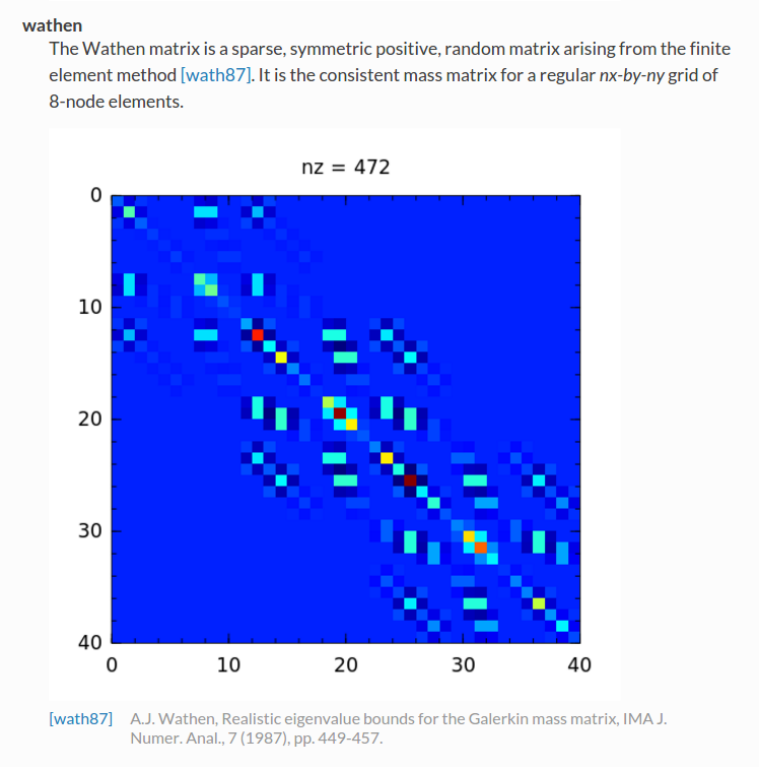


Figure 8.1: Documentation for the Wathen matrix

Winston package[6], with the color range determined by the smallest and largest entries of the matrix. For example, Figure 8.1 shows how the Wathen matrix is documented in Matrix Depot.

8.4 The Matrices

We now describe the matrices that are provided with, or can be downloaded into, Matrix Depot.

8.4.1 Parametrized Matrices

In Matrix Depot v0.5.5, there are 58 parametrized matrices (including the regularization problems described in the next section), most of which originate from the Test Matrix Toolbox [46]. All these matrices can be generated as matrixdepot("matrix˙name", n), where n is the dimension of the matrix.

Many matrices can have more than one input parameter, and multiple dispatch

[6] https://github.com/nolta/Winston.jl

provides a convenient mechanism for taking different actions for different argument types. For example, the tridiag function generates a tridiagonal matrix from vector arguments giving the subdiagonal, diagonal, and superdiagonal vectors, but a tridiagonal Toeplitz matrix can be obtained by supplying scalar arguments that specify the dimension of the matrix, the subdiagonal, the diagonal, and the superdiagonal. If a single, scalar argument n is supplied then an n-by-n tridiagonal Toeplitz matrix with subdiagonal and superdiagonal -1 and diagonal 2 is constructed. This matrix arises in applying central differences to a second derivative operator, and the inverse and the condition number are known explicitly [47, sec. 28.5].

Here is an example of the different usages of tridiag.

```
> matrixdepot("tridiag")
    Tridiagonal Matrix
   ====================

  Construct a tridiagonal matrix of type Tridiagonal.

  Input options:

    * [type,] v1, v2, v3: v1 and v3 are vectors of subdiagonal and
      superdiagonal elements, respectively, and v2 is a vector of
      diagonal elements.

    * [type,] dim, x, y, z: dim is the dimension of the matrix,
      x, y, z are scalars. x and z are the subdiagonal and
      superdiagonal elements, respectively, and y is the diagonal
      elements.

    * [type,] dim: x = -1, y = 2, z = -1. This matrix is also known
      as the second difference matrix.

  Groups: ["inverse", "ill-cond", "pos-def", "eigen"]

  References:

  J. Todd, Basic Numerical Mathematics, Vol. 2: Numerical Algebra,
  Birkhauser, Basel, and Academic Press, New York, 1977, p. 155.
```

```
> matrixdepot("tridiag", [2,5,6;], ones(4), [3,4,1;])
4x4 Tridiagonal{Float64}:
 1.0  3.0  0.0  0.0
 2.0  1.0  4.0  0.0
 0.0  5.0  1.0  1.0
 0.0  0.0  6.0  1.0

> matrixdepot("tridiag", 4, 5, 3, 1)
4x4 Tridiagonal{Float64}:
 3.0  1.0  0.0  0.0
 5.0  3.0  1.0  0.0
 0.0  5.0  3.0  1.0
 0.0  0.0  5.0  3.0

> matrixdepot("tridiag", Int, 4)
4x4 Tridiagonal{Int64}:
  2  -1   0   0
 -1   2  -1   0
  0  -1   2  -1
  0   0  -1   2
```

Test Problems for Regularization Methods

A mathematical problem is ill-posed if the solution is not unique or if an arbitrarily small perturbation of the data can cause an arbitrarily large change in the solution. Regularization methods are an important class of methods for dealing with such problems [40], [43]. One means of generating test problems for regularization methods is to discretize a given ill-posed problem.

Matrix Depot contains a group of regularization test problems derived from Hansen's MATLAB Regularization Tools [39], [41], [42] that are mostly discretizations of Fredholm integral equations of the first kind:

$$\int_0^1 K(s,t)f(t)\,dt = g(s), \qquad 0 \le s \le 1.$$

The regularization test problems form the group regprob.

```
> matrixdepot("regprob")
12-element Array{ASCIIString,1}:
 "baart"
 "blur"
 "deriv2"
 "foxgood"
 "gravity"
 "heat"
 "parallax"
 "phillips"
 "shaw"
 "spikes"
 "ursell"
 "wing"
```

Each problem is a linear system $Ax = b$ where the matrix A and vectors x and b are obtained by discretization (using quadrature or the Galerkin method) of K, f, and g. By default, we generate only A, which is an ill-conditioned matrix. The whole test problem will be generated if the parameter matrixonly is set to false, and in this case the output has type RegProb, which is defined as

```
immutable RegProb–T″
  A::AbstractMatrix–T″  # matrix of interest
  b::AbstractVector–T″  # right-hand side
  x::AbstractVector–T″  # the solution to Ax = b
end
```

If r is a generated test problem, then r.A, r.b, and r.x are the matrix A and vector x and b respectively. If the solution is not provided by the problem, the output is stored as type RegProbNoSolution, which is defined as

```
immutable RegProbNoSolution–T″
    A::AbstractMatrix–T″  # matrix of interest
    b::AbstractVector–T″  # right-hand side
end
```

For example, the test problem wing can be generated as follows.

```
> matrixdepot("wing")
      A Problem with a Discontinuous Solution
      =======================================

   Input options:

     * [type,] dim, t1, t2, [matrixonly]: the dimension of matrix
       is dim. t1 and t2 are two real scalars such that
       0 < t1 < t2 < 1.
       If matrixonly = false, the matrix A and vectors b and x in
       the linear system Ax = b will be generated(matrixonly = true
       by default).

     * [type,] n, [matrixonly]: t1 = 1/3 and t2 = 2/3.

   Groups: ["regprob"]

   References:

  G. M. Wing, A Primer on Integral Equations of the First Kind,
  Society for Industrial and Applied Mathematics, 1991, p. 109.


> A = matrixdepot("wing", 4)
4x4 Array{Float64,2}:
 0.031189   0.0921165  0.148804  0.198786
 0.0310674  0.0889342  0.134959  0.164156
 0.0309463  0.085862   0.122403  0.13556
 0.0308257  0.0828958  0.111014  0.111945

> r = matrixdepot("wing", 4, false)
Test problems for Regularization Methods
A:
4x4 Array{Float64,2}:
 0.031189   0.0921165  0.148804  0.198786
 0.0310674  0.0889342  0.134959  0.164156
 0.0309463  0.085862   0.122403  0.13556
 0.0308257  0.0828958  0.111014  0.111945
b:
```

```
4-element Array{Float64,1}:
 0.0804953
 0.0751385
 0.0701787
 0.0655842
x:
4-element Array{Float64,1}:
 0.0
 0.5
 0.5
 0.0

> r.x
4-element Array{Float64,1}:
 0.0
 0.5
 0.5
 0.0
```

8.4.2 Matrix Data from External Sources

Matrix Depot provides access to matrices from Matrix Market [8] and the University of Florida Sparse Matrix Collection [20], [21], both of which contain many matrices taken from applications. In particular, these sources contain many large, sparse matrices.

Matrix Market and the University of Florida Sparse Matrix Collection both categorize matrices by application domain and the problem source and both provide matrices in Matrix Market Format [9]. These similarities allow us to design a generic interface for both collections. The symbol :get (or :g) is used for downloading matrices from both collections and the symbol :read (or :r) is used for reading in matrices already downloaded. Downloaded matrix data is stored on disk in the Matrix Market format and when read into Julia is stored in the type SparseMatrixCSC.

MatrixDepot.update() downloads the matrix name data files from the two web servers.

```
> MatrixDepot.update()
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
```

```
100 1887k    0 1887k    0     0  97337      0 --:--:--  0:00:19 --:--:--
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
100 41552    0 41552    0     0   4421      0 --:--:--  0:00:09 --:--:--
```

The University of Florida Sparse Matrix Collection is divided into matrix groups and the group of a matrix forms part of the full name of the matrix [21]. For example, the full name of the matrix 1138_bus in the Harwell-Boeing Collection is HB/1138_bus.

```
> matrixdepot("HB/1138_bus", :get)
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
100 19829  100 19829    0     0   2320      0  0:00:08  0:00:08 --:--:--

> matrixdepot("HB/1138_bus", :read)
1138x1138 Symmetric{Float64,SparseMatrixCSC{Float64,Int64}}:
1474.78      0.0       0.0    ...    0.0       0.0          0.0     0.0
   0.0       9.13665   0.0           0.0       0.0          0.0     0.0
   0.0       0.0      69.6147        0.0       0.0          0.0     0.0
   0.0       0.0       0.0           0.0       0.0          0.0     0.0
  -9.01713   0.0       0.0           0.0       0.0          0.0     0.0
   0.0       0.0       0.0    ...    0.0       0.0          0.0     0.0
   0.0       0.0       0.0           0.0       0.0          0.0     0.0
   0.0       0.0       0.0           0.0       0.0          0.0     0.0
   0.0       0.0       0.0           0.0       0.0          0.0     0.0
   0.0      -3.40599   0.0           0.0       0.0          0.0     0.0
   ...                                                              ...
   0.0       0.0       0.0           0.0       0.0          0.0     0.0
   0.0       0.0       0.0    ...    0.0     -24.3902       0.0     0.0
   0.0       0.0       0.0           0.0       0.0          0.0     0.0
   0.0       0.0       0.0           0.0       0.0          0.0     0.0
   0.0       0.0       0.0           0.0       0.0          0.0     0.0
   0.0       0.0       0.0          26.5639    0.0          0.0     0.0
   0.0       0.0       0.0    ...    0.0      46.1767       0.0     0.0
   0.0       0.0       0.0           0.0       0.0      10000.0     0.0
   0.0       0.0       0.0           0.0       0.0          0.0   117.647
```

Matrices from the University of Florida Sparse Matrix Collection are stored in MatrixDepot/data and they are stored by group (to avoid duplicate names), i.e., one directory per group. Similarly, matrices from Matrix Market are stored in MatrixDepot/data/mm. Both

directories are untracked by Git. Many matrices in the University of Florida Sparse Matrix Collection contain problem-specific metadata, all of which is downloaded. The metadata is accessed by setting the keyword argument meta to true. Then instead of returning the matrix, Matrix Depot will return the metadata (including the matrix) as a dictionary. For example, the IMDB movie database Pajek/IMDB has metadata related to actors and movies. The following command stores all the metadata of Pajek/IMDB in a variable r, where r["IMDB"] is the test matrix.

```
> r = matrixdepot("Pajek/IMDB", :r, meta = true)
Dict{AbstractString,Any} with 8 entries:
"IMDB_colname"     => "'La Tata' Castro, Maria Tereza\n'La Veneno'...
"IMDB_MovieBacon" => 428440x1 Array{Float64,2}
"IMDB_code"        => "Drama\nShort\nDocumentary\nComedy
\nWestern\nFamily...
"IMDB_KevinBacon" => 1x1 Array{Float64,2}
"IMDB_ActorBacon" => 896308x1 Array{Float64,2}
"IMDB_category"    => 428440x1 Array{Float64,2}
"IMDB"             => 428440x896308 sparse matrix with 3782463 Float64...
"IMDB_year"        => 428440x1 Array{Float64,2}
```

We can download a whole group of matrices from the University of Florida sparse matrix collection using the command matrixdepot("group name/*", :get). The next example downloads all 67 matrices in the Gset group of matrices from random graphs (contributed by Y. Ye) then displays all the matrices in Matrix Depot, including the newly downloaded matrices.

```
> matrixdepot("Gset/*", :get)

Downloading all matrices in group Gset...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
100 48083  100 48083    0     0  95388      0 --:--:-- --:--:-- --:--:--
download:/home/weijian/.julia/v0.4/MatrixDepot/src/../data/uf/Gset/
G1.tar.gz G1/G1.mtx
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
100 55180  100 55180    0     0  75318      0 --:--:-- --:--:-- --:--:--
download:/home/weijian/.julia/v0.4/MatrixDepot/src/../data/uf/Gset/
G10.tar.gz G10/G10.mtx
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
100  5926  100  5926    0     0  23126      0 --:--:-- --:--:-- --:--:--
download:/home/weijian/.julia/v0.4/MatrixDepot/src/../data/uf/Gset/
G11.tar.gz G11/G11.mtx
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
100  6349  100  6349    0     0  24223      0 --:--:-- --:--:-- --:--:--
...

> matrixdepot()

Matrices:
   1) baart          2) binomial       3) blur           4) cauchy
   5) chebspec       6) chow           7) circul         8) clement
   9) companion     10) deriv2        11) dingdong      12) fiedler
  13) forsythe      14) foxgood       15) frank         16) golub
  17) gravity       18) grcar         19) hadamard      20) hankel
  21) heat          22) hilb          23) invhilb       24) invol
  25) kahan         26) kms           27) lehmer        28) lotkin
  29) magic         30) minij         31) moler         32) neumann
  33) oscillate     34) parallax      35) parter        36) pascal
  37) pei           38) phillips      39) poisson       40) prolate
  41) randcorr      42) rando         43) randsvd       44) rohess
  45) rosser        46) sampling      47) shaw          48) spikes
  49) toeplitz      50) tridiag       51) triw          52) ursell
  53) vand          54) wathen        55) wilkinson     56) wing
  57) Gset/G1       58) Gset/G10      59) Gset/G11      60) Gset/G12
  61) Gset/G13      62) Gset/G14      63) Gset/G15      64) Gset/G16
  65) Gset/G17      66) Gset/G18      67) Gset/G19      68) Gset/G2
  69) Gset/G20      70) Gset/G21      71) Gset/G22      72) Gset/G23
  73) Gset/G24      74) Gset/G25      75) Gset/G26      76) Gset/G27
  77) Gset/G28      78) Gset/G29      79) Gset/G3       80) Gset/G30
  81) Gset/G31      82) Gset/G32      83) Gset/G33      84) Gset/G34
  85) Gset/G35      86) Gset/G36      87) Gset/G37      88) Gset/G38
  89) Gset/G39      90) Gset/G4       91) Gset/G40      92) Gset/G41
  93) Gset/G42      94) Gset/G43      95) Gset/G44      96) Gset/G45
  97) Gset/G46      98) Gset/G47      99) Gset/G48     100) Gset/G49
 101) Gset/G5      102) Gset/G50     103) Gset/G51     104) Gset/G52
```

```
 105) Gset/G53     106) Gset/G54     107) Gset/G55     108) Gset/G56
 109) Gset/G57     110) Gset/G58     111) Gset/G59     112) Gset/G6
 113) Gset/G60     114) Gset/G61     115) Gset/G62     116) Gset/G63
 117) Gset/G64     118) Gset/G65     119) Gset/G66     120) Gset/G67
 121) Gset/G7      122) Gset/G8      123) Gset/G9
Groups:
  all             data            eigen           ill-cond
  inverse         pos-def         random          regprob
  sparse          symmetric
```

The full name of a matrix in Matrix Market comprises three parts: the collection name, the set name, and the matrix name. For example, the full name of the matrix BCSSTK14 in the set BCSSTRUC2 from the Harwell-Boeing Collection is Harwell-Boeing/bcsstruc2/bcsstk14. Note that both set name and matrix name are in lower case.

```
> matrixdepot("Harwell-Boeing/bcsstruc2/bcsstk14", :get)
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
100  292k  100  292k    0     0  22635      0  0:00:13  0:00:13 --:--:--
download:/home/weijian/.julia/v0.4/MatrixDepot/data/mm/Harwell-Boeing/
bcsstruc2/bcsstk14.mtx.gz

> matrixdepot("Harwell-Boeing/bcsstruc2/bcsstk14", :read)
1806x1806 Symmetric{Float64,SparseMatrixCSC{Float64,Int64}}:
      1.93161e6   0.0      -1.02166e5  ...      0.0          0.0
      0.0         1.0       0.0                 0.0          0.0
     -1.02166e5   0.0       1.93147e6           0.0          0.0
 -35568.9         0.0       1.65787e5           0.0          0.0
     -1.06959e5   0.0      -1.06959e5           0.0          0.0
     -1.65835e5   0.0   35568.9         ...      0.0          0.0
   -717.845       0.0       0.0                 0.0          0.0
      0.0         0.0   88998.5                 0.0          0.0
      0.0         0.0      -1.82865e6           0.0          0.0
      0.0         0.0       1.24988e5           0.0          0.0
      ...                                                    ...
      0.0         0.0       0.0                 0.0          0.0
      0.0         0.0       0.0                 1.06103e7   -5.25151e5
      0.0         0.0       0.0                -5.25151e5   -53434.0
```

```
0.0    0.0    0.0    ...    1.06959e5     -1.65835e5
0.0    0.0    0.0           0.0            0.0
0.0    0.0    0.0           1.06959e5      35568.9
0.0    0.0    0.0        -816518.0         1.21311e7
0.0    0.0    0.0           4.55624e7      8.15266e5
0.0    0.0    0.0    ...    8.15266e5      5.27942e8
```

We recommend downloading matrices from the University of Florida sparse matrix collection when there is a choice, because almost every matrix from Matrix Market is included in it.

8.5 Concluding Remarks

Matrix Depot follows in the footsteps of earlier collections of matrices. Its novelty is threefold. First, it is extensible by the user, and so can be adapted to the user's needs. In doing so it facilitates experimentation, and in particular makes it easier to do reproducible research. Second, it combines several existing test matrix collections, namely Higham's Test Matrix Toolbox, Hansen's regularization problems, and the University of Florida sparse matrix collection, in order to provide both parametrized test matrices and real-life sparse matrix data in a single framework. Third, it fully exploits the Julia language. It uses multiple dispatch to help provide a simple interface and, in particular, to allow matrices to be generated in any of the numeric data types supported by the language. Matrix Depot therefore anticipates the development of intrinsic support in Julia for computations with BigFloat and other data types.

Matrix Depot has been in development since 2014. It is an open source project[7] hosted on GitHub and is available under the MIT License. A first release was announced in December 2014. Matrix Depot v0.5.5 is the latest official release and consists of around $3,000$ lines of source code, with test coverage of 98.91% according to Codecov[8]. From GitHub traffic analytics, we learn that Matrix Depot has 40 to 70 unique downloads (unique cloners) every month. Matrix Depot also benefits the development of other Julia packages. LightGraphs[9], an optimized graph package for Julia, for example, has embedded Matrix Depot as its database.

[7] https://github.com/weijianzhang/MatrixDepot.jl
[8] https://codecov.io/
[9] https://github.com/JuliaGraphs/LightGraphs.jl

We built Matrix Depot to facilitate the development and testing of matrix (and other) algorithms in Julia. and we will continue to develop Matrix Depot by introducing new test matrices and integrating other test collections. Contributors from around the world have helped us maintain the package. Notable contributors include Joshua Adelman [10], Andreas Noack [11], and Klaus Crusius [12]. At the time of writing this thesis, Matrix Depot has 30 stars and 12 pull requests (11 merged requests) on GitHub.

[10] https://github.com/synapticarbors
[11] https://github.com/andreasnoack
[12] https://github.com/KlausC

Chapter 9

Conclusion

In this chapter, we provide a summary of previous chapters and identify some questions for future work (more detailed conclusions can be found at the end of each chapter).

We present a software package, called EvolvingGraphs.jl, in Chapter 3. Using EvolvingGraphs.jl, one can easily create an evolving graph object and use the algorithms and functions in the package for graph analysis. All evolving graph algorithms discussed in the thesis are implemented in EvolvingGraphs.jl. It performs all the computation works serially. In the future, we will develop parallel evolving graph algorithms for large scale network problems.

In Chapter 4, we generalize the breadth first search (BFS) algorithm for evolving graphs. We observe that the structure associated with causal edges cannot be captured by products of successive adjacency matrices. For our notion of temporal distance, we show that BFS over an evolving graph computes the correct result only if we consider both causal edges and static edges. We note that future work is needed to derive formulations of the algebraic BFS for evolving graphs.

By considering both causal edges and static edges in a time-preserving path, we study the influence of causal edges on graph centrality by varying the causal edge weights in Chapter 5. We derive new centrality algorithms from the view of traffic flow and observe that the rankings of nodes computed by temporal Katz centrality are stable when we change causal edge weights. Further experiments are needed to verify this observation on large real world datasets.

We describe a new discovery engine Etymo for finding interesting research literature in Chapter 6. We present a new approach to improve search results on new papers

by building a similarity-based network using the papers' full text content and social media data. In experiments, we find Etymo provides higher quality search results to users. We also describe Etymo's user interface, which combines the item list with item relationship. This reveals relationships between papers.

In Chapter 7, we further improve the search experience using an evolving knowledge graph, which is constructed using a combination of concepts evolving graphs and content similarity-based graphs. We use an evolving graph to help users track the development of ideas. Since concepts are time-dependent in an evolving graph, we can also examine the change of a concept over time. We note that the number of concepts in our current evolving knowledge graph are still small and future work is needed to include more concepts and other knowledge graph data.

Finally, in Chapter 8 we present a new test matrix collection in Julia called Matrix Depot. Matrix Depot follows in the footsteps of earlier collections of matrices and has several novel features. First, it is extensible by the user, and users can add new matrices and new matrix groups easily. Second, it combines many existing test matrix collections, including Higham's Test Matrix Toolbox, Hansen's regularization problems, and the University of Florida sparse matrix collection. Third, it uses multiple dispatch to help provide a simple interface and to allow matrices to be generated in any numeric data types supported by Julia. We will continue to develop Matrix Depot by introducing new test matrices and integrating other test collections.

Bibliography

[1] Ahmad Alsayed and Desmond J. Higham. Betweenness in time dependent networks. *Chaos, Solitons & Fractals*, 72:35–48, 2015.

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[3] Bahman Bahmani, Ravi Kumar, Mohammad Mahdian, and Eli Upfal. PageRank on an evolving graph. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 24–32. ACM, 2012.

[4] Zhaojun Bai, David Day, James Demmel, and Jack Dongarra. A test matrix collection for non-Hermitian eigenvalue problems (release 1.0). Technical Report CS-97-355, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, March 1997. LAPACK Working Note 123.

[5] Jöran Beel and Bela Gipp. Google Scholar's ranking algorithm: An introductory overview. In *Proceedings of the 12th International Conference on Scientometrics and Informetrics (ISSI'09)*, volume 1, pages 230–241, Rio de Janeiro (Brazil), 2009. ISSI.

[6] Timo Betcke, Nicholas J. Higham, Volker Mehrmann, Christian Schröder, and Françoise Tisseur. NLEVP: A collection of nonlinear eigenvalue problems. *ACM Trans. Math. Software*, 39(2):7:1–7:28, February 2013.

[7] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[8] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix Market: A Web resource for test matrix collections. In Ronald F. Boisvert, editor, *Quality of Numerical Software: Assessment and Enhancement*, pages 125–136. Chapman and Hall, London, 1997.

[9] Ronald F. Boisvert, Roldan Pozo, and Karin A. Remington. The Matrix Market exchange formats: Initial design. Technical Report NISTIR 5935, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA, 1996.

[10] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. AcM, 2008.

[11] Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 2013.

[12] Stephen P. Borgatti. Centrality and network flow. *Social networks*, 27(1):55–71, 2005.

[13] Pierre Borgnat, Eric Fleury, Jean-Loup Guillaume, Clémence Magnien, Céline Robardet, and Antoine Scherrer. Evolving networks. *NATO ASI on Mining Massive Data Sets for Security, NATO Science for Peace and Security Series D: Information and Communication Security*, pages 198–204, 2008.

[14] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr, and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3. Atlanta, 2010.

[15] Jiahao Chen and Weijian Zhang. The right way to search evolving graphs. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 867–876, New York City, 2016. IEEE.

[16] Francois Chollet. *Deep learning with Python*. Manning Publications Co., 2017.

[17] Paul A. Clement. A class of triple-diagonal matrices for test purposes. *SIAM Rev.*, 1(1):50–52, 1959.

[18] Andrew M. Dai, Christopher Olah, and Quoc V. Le. Document embedding with paragraph vectors. *arXiv preprint arXiv:1507.07998*, 2015.

[19] Peter Bacon Darwin and Pawel Kozlowski. *AngularJS web application development*. Packt Publ., 2013.

[20] Timothy A. Davis. University of Florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices.

[21] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Software*, 38(1):1:1–1:25, 2011.

[22] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–610. ACM, 2014.

[23] David L. Donoho and Victoria Stodden. Reproducible research in the mathematical sciences. In Nicholas J. Higham, Mark R. Dennis, Paul Glendinning, Paul A. Martin, Fadil Santosa, and Jared Tanner, editors, *The Princeton Companion to Applied Mathematics*, pages 916–925. Princeton University Press, Princeton, NJ, USA, 2015.

[24] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15(1):1–14, 1989.

[25] Susan T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, 1991.

[26] Ernesto Estrada, Desmond J. Higham, and Naomichi Hatano. Communicability betweenness in complex networks. *Physica A: Statistical Mechanics and its Applications*, 388(5):764–774, 2009.

[27] Shimon Even and Guy Even. *Graph algorithms*. Cambridge University Press, Cambridge, UK, 2nd edition, 2012.

[28] John R. Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, 1957.

[29] Philippe Flajolet, Donald E. Knuth, and Boris Pittel. The first cycles in an evolving graph. *Annals of Discrete Mathematics*, 43:167–215, 1989.

[30] Dániel Fogaras. Where to start browsing the web? In *International Workshop on Innovative Internet Community Systems*, pages 65–79. Springer, 2003.

[31] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Software*, 33(2):13:1–13:15, 2007.

[32] C. W. Gear. A simple set of test matrices for eigenvalue programs. *Math. Comp.*, 23(105):119–125, 1969.

[33] David F. Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.

[34] Simon Gottschalk and Elena Demidova. Eventkg: A multilingual event-centric temporal knowledge graph. In *European Semantic Web Conference*, pages 272–287. Springer, 2018.

[35] Danica Vukadinović Greetham, Zhivko Stoyanov, and Peter Grindrod. On the radius of centrality in evolving communication networks. *Journal of Combinatorial Optimization*, 28(3):540–560, 2014.

[36] Robert T. Gregory and David L. Karney. *A Collection of Matrices for Testing Computational Algorithms*. Wiley, New York, 1969. Reprinted with corrections by Robert E. Krieger, Huntington, New York, 1978.

[37] Peter Grindrod and Desmond J. Higham. A matrix iteration for dynamic network summaries. *SIAM Review*, 55(1):118–128, 2013.

[38] Peter Grindrod, Mark C. Parsons, Desmond J. Higham, and Ernesto Estrada. Communicability across evolving networks. *Physical Review E*, 83(4):046120, 2011.

[39] Per Christian Hansen. Regularization Tools: A Matlab package for analysis and solution of discrete ill-posed problems. *Numer. Algorithms*, 6(1):1–35, 1994.

[40] Per Christian Hansen. *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[41] Per Christian Hansen. Regularization Tools version 4.0 for Matlab 7.3. *Numer. Algorithms*, 46(2):189–194, 2007.

[42] Per Christian Hansen. Regularization tools. A Matlab package for analysis and solution of discrete ill-posed problems. Version 4.1 for Matlab 7.3. Report, Information and Mathematics Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark, March 2008.

[43] Per Christian Hansen. *Discrete Inverse Problems: Insight and Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2010.

[44] Kun He, Yingru Li, Sucheta Soundarajan, and John E. Hopcroft. Hidden community detection in social networks, February 2017. ArXiv preprint arXiv:1702.07462.

[45] Nicholas J. Higham. Algorithm 694: A collection of test matrices in MATLAB. *ACM Trans. Math. Software*, 17(3):289–305, September 1991.

[46] Nicholas J. Higham. The Test Matrix Toolbox for MATLAB (version 3.0). Numerical Analysis Report No. 276, Manchester Centre for Computational Mathematics, Manchester, England, September 1995.

[47] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[48] Jorge E. Hirsch. An index to quantify an individual's scientific research output. *Proceedings of the National Academy of Sciences of the United States of America*, 102(46):16569, 2005.

[49] Silu Huang, Ada Wai-Chee Fu, and Ruifeng Liu. Minimum spanning trees in temporal graphs. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 419–430. ACM, 2015.

[50] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[51] Jeremy Kepner and John Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*. Software, Environments, Tools. SIAM, Philadelphia, PA, 2011.

[52] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P. Gleeson, Yamir Moreno, and Mason A. Porter. Multilayer networks. *Journal of Complex Networks*, 2(3):203–271, 2014.

[53] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. Assoc. Comput. Mach.*, 46(5):604–632, September 1999.

[54] Amy N Langville and Carl D Meyer. Deeper inside pagerank. *Internet Mathematics*, 1(3):335–380, 2004.

[55] Amy N. Langville and Carl D. Meyer. *Google's PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2011.

[56] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[57] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196, 2014.

[58] Joseph CR. Licklider and Robert W. Taylor. The computer as a communication device. *Science and technology*, 76(2):1–3, 1968.

[59] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[60] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, 2008.

[61] Osni A. Marques, Christof Vömel, James W. Demmel, and Beresford N. Parlett. Algorithm 880: A testing infrastructure for symmetric tridiagonal eigensolvers. *ACM Trans. Math. Software*, 35(1), 2008. Article 8, 13 pages.

[62] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *ICLR Workshop*, 2013.

[63] Marvin Minsky. *The emotion machine: Commonsense thinking, artificial intelligence, and the future of the human mind*. Simon and Schuster, 2006.

[64] Vincenzo Nicosia, John Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. Graph metrics for temporal networks. In *Temporal Networks*, pages 15–40. Springer, 2013.

[65] Feng Niu, Ce Zhang, Christopher Ré, and Jude W. Shavlik. Deepdive: Web-scale knowledge-base construction using statistical learning and inference. *VLDS*, 12:25–28, 2012.

[66] M. L. Pei. A test matrix for inversion procedures. *Comm. ACM*, 5(10):508, 1962.

[67] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[68] Angelo Antonio Salatino and Enrico Motta. Detection of embryonic research topics by analysing semantic topic networks. In *International Workshop on Semantic, Analytics, Visualization*, pages 131–146. Springer, 2016.

[69] Gerard Salton and Chris Buckley. *Introduction to modern information retrieval*. McGraw-Hill, New York, 1983.

[70] Bahar Sateli, Felicitas Löffler, Birgitta König-Ries, and René Witte. Scholarlens: Extracting competences from research publications for the automatic generation of semantic user profiles. *PeerJ Computer Science*, 3:e121, July 2017.

[71] Eric Schmidt and Jonathan Rosenberg. *How google works*. Hachette UK, 2014.

[72] R. L. Stratonovich. Conditional markov processes. *Theory of Probability & Its Applications*, 5(2):156–178, 1960.

[73] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.

[74] Zhaowei Tan, Changfeng Liu, Yuning Mao, Yunqi Guo, Jiaming Shen, and Xinbing Wang. Acemap: A novel approach towards displaying relationship among academic literatures. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 437–442, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[75] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 990–998. ACM, 2008.

[76] John Tang, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. Temporal distance metrics for social network analysis. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 31–36. ACM, 2009.

[77] John Tang, Mirco Musolesi, Cecilia Mascolo, Vito Latora, and Vincenzo Nicosia. Analysing information flows and key mediators through temporal centrality metrics. In *Proceedings of the 3rd Workshop on Social Network Systems*, page 3. ACM, 2010.

[78] John Tang, Salvatore Scellato, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. Small-world behavior in time-varying graphs. *Physical Review E*, 81(5):055101(R), 2010.

[79] Alan Taylor and Desmond J. Higham. CONTEST: A controllable test matrix toolbox for MATLAB. *ACM Trans. Math. Software*, 35(4):26:1–26:17, 2009.

[80] Laurens Van Der Maaten. Accelerating t-sne using tree-based algorithms. *Journal of machine learning research*, 15(1):3221–3245, 2014.

[81] Weijian Zhang. Dynamic network analysis in Julia. Technical Report 2015.83,

Manchester Institute for Mathematical Sciences, The University of Manchester, UK, September 2015.