University of Glasgow | School of Computing Science

Implementation of Novel Subgraph Query Processing methods within GraphX

Iva Stefanova Babukova

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 20, 2016

Abstract

The GraphX system has recently been developed at Berkeley, over the Spark massively-parallel data processing system, as a system for high performance analytics over graph data. It is currently an important tool for graph-analytic tasks, which are core to many data science endeavours. At the same time, graph datasets have become increasingly popular, used to model applications from numerous domains from social networks to biology and bioinformatics. The goal of this project is to design, implement, and test an algorithm for subgraph queries, on top of GraphX.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

The first page, abstract and table of contents are numbered using Roman numerals. From now on pages are numbered using Arabic numerals. Therefore, immediately after the first call to \chapter we need the call \pagenumbering{arabic} and this should be called once only in the document.

The first Chapter should then be on page 1. You are allowed 50 pages for a 30 credit project and 35 pages for a 20 credit report. This includes everything up to but excluding the appendices and bibliograph, i.e. this is a limit on the body of the report.

You are not allowed to alter text size (it is currently 11pt) neither are you allowed to alter the margins.

Note that in this example, and some of the others, you need to execute the following commands the first time you process the files. Multiple calls to pdflatex are required to resolve references to labels and citations. The file bib.bib is the bibliography file.

```
> pdflatex example0
> bibtex example0
> pdflatex example0
> pdflatex example0
```

1.1 The Fox Jumps Over

The quick brown fox jumped over the lazy dog. The quick brown fox jumped over Uroborus (Figure 1.1).

The quick brown fox jumped over the [2] lazy dog. [5] The quick brown fox jumped over the lazy dog. [4] The quick brown fox jumped over [1] the lazy dog. The quick brown fox jumped over [3] the lazy dog.

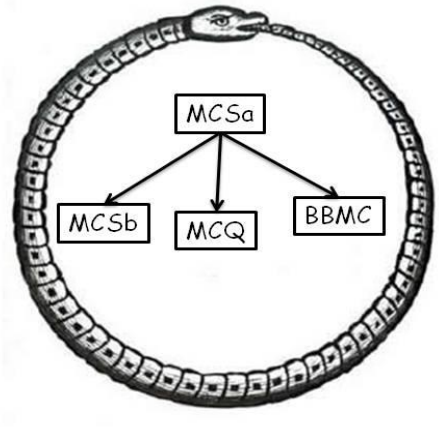


Figure 1.1: An alternative hierarchy of the algorithms.

Chapter 2

Introduction

2.1 Aims and motivation

- graphs are widely used nowadays to represent data
- the graph containment problem is widely addressed in many areas of science: genetics, chemistry, XML documents, images, fraud detection and prevention (there was an article in nature about this)
- graph indexing can help us reduce the waiting time
- we have a lot of data that often can't be processed in memory by one machine
- spark is an engine that allows us to perform cluster computing and load data in-memory and share it across many clusters
- graphx is like spark, but has special optimizations for graphs
- make graph indexing in java, no concurrency and see the trade off between various indexing approaches
- make graph indexing in spark, vary the number of worker threads, measure performance and decide what is best

In the core of many graph-related applications, lies a common and critical problem: *how to efficiently process graph queries and retrieve related graphs*. In some cases, the success of an application directly relies on the efficiency of the query processing system.

Applications:

- genome sequencing: find mutations responsible for rare diseases – nature vol 527 no 7576
- treating diseases like cancer: screen a patient's tumor for a set of biomarkers to choose the best treatment to fight the particular cancer – nature vol 527 no 7578

2.2 Background

2.3 Preliminaries

In this section, we introduce preliminary concepts and outline the main concepts and problems addressed in the document. In definition 1 we explain the format of all graphs used in the document.

2.3.1 Naming conventions

- The set of target graphs: T - Each graph in t: t_i, for i from 1 to the number of graphs in T - The set of pattern graphs: P - Each pattern graph in P: p_j, for i from 1 to the number of graphs in P - The candidate set of graphs: C. It is important to not that C is contained in T - There is a trade off between the size of C and the time it takes to be computed.

2.3.2 Definitions

Definition 1 (Graph Format)**.** A graph G = (V, E, L, λ) is defined as an undirected labeled graph where V is the set of vertices, E is the set of edges(unordered pair of vertices), L is the set of labels, and λ is a labeling function, $\lambda : V \cup E \rightarrow L$, that assigns labels to vertices and edges.

Definition 2 (Graph Isomorphism)**.**

Definition 3 (Subgraph Isomorphism)**.** Given a graph database T of target graphs t_0, t_1, $t_2 \ldots t_i$, where *i* is the number of target graphs in T, and a pattern graph P, find all targets in T that have P as a subgraph.

Definition 4 (Subgraph)**.** A graph whose vertices and edges are a subset of another graph.

Definition 5 (Graph Query Processing)**.** Given a graph database D = $\{ g_0, g_1, g_2 \ldots g_n \}$ and a pattern graph p, it returns the query answer set D_p = $\{ g_i | p \subseteq g_i, g_i \in D \}$

Definition 6 (In-memory computing)**.** The storage of information in the main random access memory (RAM) of dedicated servers rather than in relational databases operating on comparatively slow disk drives.In-memory computing gives ability to cache countless amounts of data constantly. This ensures extremely fast response times for searches.

Definition 7 (Fault-Tolerant Manner)**.** Property that enables a system to continue to operate properly in the event of a failure.

Definition 8 (Cluster Computing)**.** A form of computing in which a group of computers are linked together so that they can act like a single entity.

Definition 9 (Graph Index)**.**

2.4 Graph Indexing

In this section, we

filter-verification process — look at grapes

2.5 Data

Chapter 3

Graph Indexing

- smaller C, less time spent on subgraph isomorphism check, but more tie spent on indexing process

3.1 Motivation for usage

3.2 Common techniques

3.2.1 Graph-mining

3.2.2 Non-graph-mining

3.3 Methods with respect to choice of indexing unit

3.3.1 Path-based indexing approach

Follows the general idea: enumerate all the existing paths in a database up to *maxLen* length and index them, where a path is a vertex-edgeProperty-vetex sequence. *example* In order to create an index of a graph *g*, this approach breaks *g* into paths and in this way, the structural information of *g* could be lost. This leads to more false-positive answers returned after the Advantages:

1. Paths are easier to manipulate than trees and graphs.
2. The index space is predefined: all the paths up to *maxLen* length are selected.

Disadvantages:

1. Path is too simple: structural information is lost
2. There are too many paths: the set of paths in a graph database usually is huge.

3.3.2 Tree-based indexing approach

Chapter 4

Tools

4.1 Spark

4.1.1 Resilient Distributed Datasets

4.2 GraphX

4.2.1 The triplets

Chapter 5

Implementation

5.1 Graph input format

The input we are working with is a file with the following format:

5.2 CT-Index

This is an open-source implementation of the indexing algorithm from *this paper* which I haven't done myself. I use CT-Index to analyze its implementation details and use it to compare the performance of my indexing implementation and to verify the correctness of my results. CT-Index is used as a benchmark ...

5.2.1 The idea

5.2.2 Implementation

5.2.3 Performance, compared to other indexing techniques

Strengths and weaknesses of this approach ...

5.3 IB-Index algorithms

All algorithms described in this section generate the candidate set of graphs in the following steps:

1. Compute the index of all graphs in T;
2. Compute the index of all patterns in P;
3. Using the target and pattern indexes computed in the previous two steps, extract all graphs in T that contain all features in the pattern index;

4. All extracted graphs from step 3 form the candidate set C.

5.4 Choice of programming language

All implementations compared with CT-Index, and compared between themselves

5.5 IB-Index 1

This section gives an overview of the first indexing technique that we designed, implemented and compared with CT-Index. The following subsections describe the main idea of the algorithm, how it was implemented and its performance.

5.5.1 The idea

The algorithm follows the exhaustive path-enumeration approach, also used in other indexing techniques like GraphGrepSX.

IB-Index computes the index of a graph by enumerating exhaustively all paths it has up to a specified maximum path length $maxLen$. The index of every graph t in T is stored into a file, called target index. The file is used for computing the set C of all graphs in T that are to be checked for subgraph isomorphism to another set of pattern graphs P. We use the target and the pattern indexes to generate the candidate set C which contains all graphs candidates from T for subgraph isomorphism with the pattern in P. Using the target index, we check which target graphs contain all paths in the pattern index. All graphs from T that contain the paths stored in the pattern index form the candidate set C.

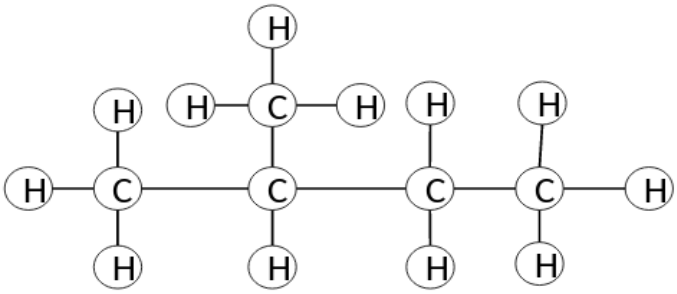


Figure 5.1: Target Graph

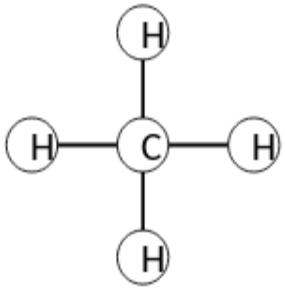
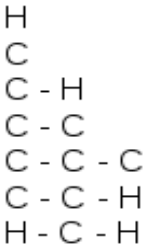


Figure 5.2: Pattern graph

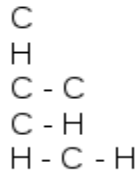
The procedure of extracting C from T with IB-Index is done as follows: Let the graph on figure 5.1 be a target graph in T, the graph on figure 5.2 be a pattern graph in P and $maxLen$ be equal to 3. In order to check whether 5.1 is a candidate for subgraph isomorphism with 5.2, we first compute the indexes of 5.1 and 5.2 for the specified maximum path length. The target index consists of the paths on figure 5.3a and the pattern index contains the paths on figure 5.3b.

After candidate extraction, IB-Index 1 returns target graph 5.1 as candidate for subgraph isomorphism with pattern graph 5.2, as all paths in 5.2, shown on figure 5.3b are contained in 5.1, shown on figure 5.3a.

We avoid storing the same paths in the index more than once in the following way. Before storing a path $path$ in the index, we reverse it and check whether $reversed - path$ is lexicographically smaller than $path$. We store the smallest variant of the path in the index file.



(a) Target graph path enumeration



(b) Pattern graph path enumeration

Figure 5.3: All paths up to maximum length 3

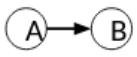
5.5.2 Implementation

Graph representation

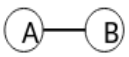
Each graph object has an integer id and is composed of a collection of node objects. Each node object has an id, a label and a list of edge objects, where the node is a source node for each edge contained in the list. The length of the list of the edge object is equal to the degree of the node. Therefore, each edge object does not need to keep a record of the source node: it only has a label and a destination node as fields.

The edge representation works for both directed and undirected graphs. If the graph is directed, only one edge object is created for each edge of the graph. If the graph is not directed, two edge objects will be created. The first one with one of the nodes as destination node and then added in the list of edges of the other node: the source node. The second edge object will have the source node of the first edge object as destination node and included in the list of edges of the destination node of the first edge object, which will be its source node. The label of both edge objects will be the same.

Figure 5.4 is an example of two nodes with labels A and B and an edge between them that is not labeled. To represent a directed edge, as in figure 5.4a, we create one Edge object with destination node B and with label an empty String. We add the created edge in node A's list of edges, as A is the source node. To represent an undirected edge, as in figure 5.4b, we create two edges. Edge 1 has B as destination node and it is included in node A's list of nodes, as A is the destination node. Edge 2 has A as destination node and B as source node and consequently, it is added to the list of edges of node B, the source node.



(a) Directed graph



(b) Undirected graph

Figure 5.4: An edge between the nodes A and B for directed an undirected graphs

Path enumeration

- we have stack: it is used for generating the path: an arraylist of nodes. It is also used for finding whether the current node has been visited in this iteration - explain the isVisited property each node has and why do we need it as well as the stack - the bug with visited nodes I had initially, how the stack was fixed

Let us consider a graph g with n number of nodes and m number of different labels, where $m < n$ and assume we want to extract all paths of maximum path length $maxLen$. Separate class includes all functionality needed to compute the index. Path extraction is done using a depth-first-search based algorithm and we use stack to keep the visited nodes as well as to generate the path string, stored in the index file. The graph is visited n times, with each node in g as start node. We push the start node on the stack (the stack is emptied before that) and we ....

Candidates extraction

I use data structures that are not efficient – for every node new Arraylist for edges, for every edge new object, for every graph hashmaps ...

I use path-based exhaustive enumeration – enumerate all paths up to a certain length. It is not good, because, as explained in section bla, path enumeration does not retain the graph structure

The index file is very big — compare sizes with CT-Index

5.5.3 Performance

As this approach is very naive, we expected it to show very poor performance. The actual results were surprisingly bad. IB-Index 1 does not only run extremely slow, but it doesn't manage to filter our any of the target graphs. The index file produced contains over 8 million paths and the running time of the program is ... compared to CT-Index with running time ... !

Limitations

5.6 IB-Index 2

isomers

5.6.1 The idea

- why do I think it is better - explain about containment, how it works - picture with worked example to show that it eliminates more graphs than the first technique

5.6.2 Implementation

- extend IB-Index 1 and put an option to extract paths using isomer labels. - describe the extension of the project to support both types of indexing

5.6.3 Performance

- the index file is bigger, but the candidate set is smaller

HC
CCHHH
CCCC
CCHHH
HC - CCHHH
HC - CCCHH
CCHHH - CCCC
CCCC - CCHHH
CCCHH - CCHHH
HC - CCHHH - HC
HC - CCHHH - CCCC
CCHHH - CCCC - CCHHH
CCHHH - CCCC - CCCHH
CCHHH - CCCC - HC
CCCC - CCCHH - HC
CCCC - CCCHH - CCHHH
CCCHH - CCHHH - HC

(a) Target graph path enumeration

HC
CCHHH
HC - CCHHH
CCHHH - CCHHH
HC - CCHHH - HC
CCHHH - CCHHH - HC

(b) Pattern graph path enumeration

Figure 5.5: All paths up to maximum length 3

5.7 IB-Index 3

isomers + different graph representation (array, bitset, hashing) maximal paths

Chapter 6

Experimental Evaluation

6.1 Experimental Data and Methodology

6.2 Pure Java implementation evaluation

6.3 Spark implementation evaluation

6.4 Pure Java vs Spark

6.5 Results Summary

Chapter 7

Conclusion and Future work

Appendices

Appendix A

Running the Programs

An example of running from the command line is as follows:

```
> java MaxClique BBMC1 brock200_1.clq 14400
```

This will apply $BBMC$ with $style = 1$ to the first brock200 DIMACS instance allowing 14400 seconds of cpu time.

Appendix B

Generating Random Graphs

We generate Erdós-Rënyi random graphs $G(n, p)$ where n is the number of vertices and each edge is included in the graph with probability p independent from every other edge. It produces a random graph in DIMACS format with vertices numbered 1 to n inclusive. It can be run from the command line as follows to produce a clq file

```
> java RandomGraph 100 0.9 > 100-90-00.clq
```

Bibliography

[1] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings IJCAI'91*, pages 331–337, 1991.

[2] P. Mutzel K. Klein, N. Kriege. Ct-index: Fingerprint-based graph indexing combining cycles and trees. *Data Engineering (ICDE), 2011 IEEE 27th International Conference, Hannover*, pages 1115 – 1126, 11-16 April 2011.

[3] Reynold Xin, Joseph Gonzalez, Daniel Crankshaw, Michael Franklin, Ankur Dave, and Ion Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. February 2014.

[4] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach, 2004.

[5] Matei Zaharia, Mosharaf Chowdhury, Ankur Dave, Michael Franklin, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI 2012*, 2012.