A Systematic Approach to Regular-Expression-Based Queries on Big Graphs

Le-Duc Tung

Doctor of Philosophy

Department of Informatics
School of Multidisciplinary Sciences
SOKENDAI (The Graduate University for Advanced Studies)

A Systematic Approach to Regular-Expression-Based Queries on Big Graphs

by

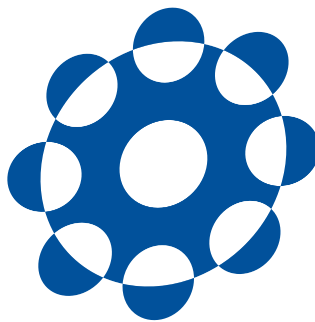
Le-Duc Tung

Dissertation

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy



SOKENDAI (The Graduate University for Advanced Studies)

March 2016

Abstract

Graphs have been increasingly important to represent data such as the World Wide Web, social networks, biological networks. With the explosion of information, big data leads to big graphs. These big graphs are often stored in a distributed system, leading to many difficulties in proposing efficient algorithms for processing big graphs.

While many distributed programming models for graphs have been proposed, MapReduce and Pregel have been shown to be scalable to deal with big data as well as big graphs. Nonetheless, to obtain scalability, these models offer restricted forms in which users specify their programs. Hence, it is non-trivial for users to write their complicated programs as well as to obtain efficient ones.

On the other hand, *regular expression* has been used as a powerful way to intuitively query data from graphs. Many useful applications of queries based on regular expressions have been discovered, such as, finding relationships in social networks, or finding chains of reactions in biological networks. However, evaluating regular-expression-based queries on distributed graphs is non-trivial. First, regular expressions imply a highly sequential evaluation. Second, distributed evaluations often produce intermediate graphs whose size is larger than the input graph, and require a large amount of communications.

This dissertation's objective is to bridge the gap between regular-expression-based queries and scalable distributed programming models. We study systematic approaches to build a general framework that automatically translates regular-expression-based queries into efficient distributed programs.

First, we focus on *select-where regular path (SWRP)* queries that return a graph constructed from subgraphs following paths whose labels spell a word in a regular expression. Queries can be nested and composed. We propose a structural-recursion

based approach to translating SWRP queries into efficient programs in Pregel. SWRP queries are first translated into structural recursive functions on graphs. Then structural recursive functions are compiled into efficient programs in Pregel. The approach ensures that the sizes of intermediate graphs generated during the evaluation are minimized and close to the size of the final result. To the best of our knowledge, this is the first time a Pregel algorithm for SWRP queries is proposed.

Second, we propose a functional-based approach to further improve the performance of SWRP queries. We observe that there is a computation during the evaluation of SWRP queries takes more time than the other computations. This demands further refinement of our framework. We start with a more fundamental query that is a *regular reachability (RR)* query. An RR query is to decide whether or not two given vertices are connected by a directed path the concatenation of whose edge/node labels spells a word in a given regular expression. We propose a functional-based approach to a distributed evaluation of RR queries, which uses functions to encode mappings between sets of states in the automaton of the given regular expression. This approach exploits parallelism by processing a long path in a distributed manner, and it also reduces the computation and communication costs during the evaluation by encoding state transitions. Then we show how to apply this approach to improve the performance of the evaluation of SWRP queries.

Finally, we extend SWRP queries to support shortest-path conditions. We show that this extension requires us to solve an additional problem that is a *shortest regular category-path (SRCP)* query. An SRCP query is a variant of a constrained shortest path query whose constraints are expressed by a *category-based regular expression.* By using a dynamic programming formulation, we show that SRCP queries can be answered efficiently by a series of single source shortest path searches. This is useful because we can utilize fast single source shortest path algorithms that are optimized for different graphs (road networks, social networks, biological networks) and environments (shared or distributed memory).

Acknowledgements

As a student, I have learned a lot from any person I met on my journey of research. I would like to send my thankful messages to all of them.

First of all, I would like to express my most sincere gratitude and appreciation to my supervisor, Professor Zhenjiang Hu for his valuable guidance, patience and encouragement throught my research during the past three years. Thanks to him, I have opened my eyes to the world of academic. He gave me lots of freedom in pursuing my researcher direction. At the same time, he was always open to help me during this long journey. To be his student is my great pleasure and luck, and I really appreciate everything he has done to me. Thank you, my respectable professor!

In addition, I would like to deeply thank my advisors Assistant Professor Soichiro Hidaka, Assistant Professor Hiroyuki Kato and the committee members, Professor Shin Nakajima, Professor Ken-ichi Kawarabayashi, and Professor Hideya Iwasaki for their encouragement, insightful comments and hard questions. My special thanks go to Assistant Professor Soichiro Hidaka for sharing with me his deep knowledge on structural recursion on graphs. His knowledge was very useful for me to complete my dissertation.

I would like to acknowledge The Hitachi Scholarship Foundation for not only providing me with financial supports to complete three years of my doctor course, but also helping me get familiar with Japanese culture as well as many high technologies from Hitachi company. I also thank National Institute of Informatics (NII) for providing a perfect working environment for my research.

I would like to deeply thank members of PoPP group for their invaluable comments on my research. I thank Professor Hideya Iwasaki for his timely encouragement, Associate Professor Kiminori Matsuzaki, Doctor Akimasa Morihata, Doctor Chong Li,

and Doctor Shigeyuki Sato for sharing their wide knowledge on parallel programming. I specially thank Associate Professor Kento Emoto for having a long discussion with me on the constrained shortest path problem. His professional research skills partially inspired me during my research.

I would like to send my thankful messages to all members in our IPL laboratory for their suggestions and comments in many works. I also thank my fellow labmates in our research room (R1611, NII) for all fun we have had in the last three years. Memories in our laboratory are going with me forever.

Moreover, I would like to thank all members of the group of Vietnamese and my colleagues at NII for sharing many useful research skills as well as life in Japan.

I would like to take this special occasion to thank my mother and my young brother for their continuous support and encouragement.

Finally, I thank with love to my wife Dinh Thi Nga for her tireless support, continuous encouragement, and unwaving love. Thank my beloved daughter Le Thu Thuy for being a source of unending joy and love during my research as well as my life.

To My Family.

Table of Contents

Abstract iii

Acknowledgements v

List of Figures xiii

List of Tables xvii

1 Introduction 1

1.1 Motivations and Objectives 1
1.2 Challenges 2
1.3 Problem Statement 4
1.4 Contributions 5
1.5 Dissertation Overview 7

2 Preliminaries 9

2.1 Graph Data Models 9
2.1.1 Definitions 9
2.1.2 The Meaning of ϵ-Edges 12
2.1.3 Graph Equivalence 13
2.2 Graph Constructors 14
2.3 Regular Expression 15
2.4 Distributed Programming Models 17
2.4.1 MapReduce 17
2.4.2 Pregel 17

3 Select-Where Regular Path Queries **23**

3.1 Definition of Select-Where Regular Path Queries 24

3.2 Specifications for SWRP Queries 27

3.2.1 Structural Recursion 27

3.2.2 Expressing SWRP Queries by Structural Recursion 28

3.3 On Obtaining Efficient Pregel Programs to Big Graphs 31

3.3.1 Specifications without Conditions 31

3.3.2 Specifications with Conditions 40

3.4 Implementation and Experiments 45

3.4.1 Implementation 45

3.4.2 Experiments 48

3.5 Related Work 57

3.6 Summary 58

4 Regular Reachability Queries **59**

4.1 Definition of Regular Reachability Queries 60

4.2 Simultaneous Finite Automaton 61

4.3 Functional-based Evaluation of Regular Reachability Queries 63

4.3.1 Evaluation Strategy 63

4.3.2 Partial Evaluation 64

4.3.3 Global Evaluation 66

4.3.4 Optimizations 69

4.4 Implementation and Experiments 70

4.4.1 Implementation 70

4.4.2 Experiments 70

4.5 Related Work 82

4.6 Discussion 84

5 SWRP Queries with Shortest Path Conditions **85**

5.1 Definition of SWRP Queries with Shortest Path Conditions 87

5.2 Practical Shortest Regular Category-Path Queries 88

5.2.1 SRCP Queries 88

5.2.2 P-SRCP: a Practical Class of SRCP Queries 89

5.2.3 Expressiveness of P-SRCP 91
5.3 Derivation of an Efficient Algorithm for P-SRCP Queries 94
5.3.1 SREs as Directed Acyclic Graphs 94
5.3.2 Dynamic Programming Formulation 95
5.3.3 Single Source Shortest Path Search 98
5.3.4 Optimizations 99
5.4 Implementation and Experiments 101
5.4.1 Implementation 101
5.4.2 Experiments 102
5.5 Related Work 107
5.6 Summary 109

6 Conclusion **111**
6.1 Summary of the Dissertation 111
6.2 Future Work 112
6.2.1 Supporting More Queries 112
6.2.2 Making the Framework More Efficient 113

Bibliography **115**

List of Published Works **125**

List of Figures

1.1 A rooted, directed edge-labeled graph of paper citation network. 2
1.2 Overview of a querying framework for graphs. 4

2.1 Examples of rooted edge-labeled graphs. 10
2.2 An example of a distributed vertex-labeled graph including 3 fragments. Here, we ignore edge labels (assuming that edges have no labels). . . . 12
2.3 Meaning of ϵ-edges. The left graph with ϵ-edges is value equivalent to the right graph without ϵ-edges. 13
2.4 Graph constructors. 16
2.5 MapReduce programming model. 18
2.6 Pregel programming model. 19
2.7 Maximum value example [15] in the Pregel model. Dotted lines are messages. Shaded circles are vertices voted to halt. 20

3.1 Example **a2d_xc**: relabels edges a to d and contracts edges c. 29
3.2 The syntax of structural recursion specifications for SWRP queries. . . 29
3.3 Bulk semantics to evaluate the specification **c_b2d**. 35
3.4 Graphs generated during the evaluation of the specification *c_b2d*. . . 38
3.5 The result of the 1st specification without conditions. 41
3.6 The result of *evalIfThenElse* algorithm. 42
3.7 The result of the 2nd specification without conditions. 43
3.8 A graph encoding a statement "**if** *isempty*$(f(\$g))$ **then** ...". 43
3.9 Overview of our framework. 46
3.10 Varying graph size (Citation dataset). 50

3.11 Varying the number of CPUs (Citation dataset). 50
3.12 Varying graph size (Youtube dataset). 51
3.13 Varying the number of CPUs (Youtube dataset). 51
3.14 Query Q1 on Yahoo! AltaVista Web Page Graphs. 54
3.15 Query Q2 on Yahoo! AltaVista Web Page Graphs. 55

4.1 A DFA for $(Books*)(Food*)(Books+)$. 60
4.2 An SFA for the DFA shown in Figure 4.1. 62
4.3 A distributed evaluation strategy for RR queries. 64
4.4 A dependency graph constructed on S_c. Note that this is not a distributed graph. We use dotted rectangles to denote that from which fragment vertices come. 68
4.5 Experiment results with various graph sizes (YouTube and DBLP). . . . 73
4.6 Experiment results with various graph sizes (MEME and Internet). . . 74
4.7 Experiment results with various query sizes (YouTube and DBLP datasets). 76
4.8 Experiment results with various query sizes (MEME and Internet datasets). 77
4.9 Experiment results with varying number of partitions (Youtube and DBLP). 78
4.10 Experiment results with varying number of partitions (MEME and Internet). 79
4.11 Experiment results for costs of different components (Youtube and DBLP). 80
4.12 Experiment results for costs of different components (MEME and Internet). 81
4.13 How functional-based algorithms are affected by graph characteristics. 83

5.1 Two equivalent graphs. 88
5.2 A P-SRCP query example $\langle s, t, \mathrm{O}\,(\mathrm{GY}\,|\,\mathrm{B})\,\mathrm{V}\rangle$. All edges have the same weight of 1. Two of the SRCPs are shaded in grey and pink. $d^R(s,t) = 9$. 91
5.3 Relationship between SRCP queries and existing queries. 93
5.4 A query graph of the query $\langle s, t, \mathrm{O}\,(\mathrm{GY}\,|\,\mathrm{B})\,\mathrm{V}\rangle$. Integers in nodes are OIDs. Superscripts of categories are equivalent row indices in the DP table. The subgraph from the node 1 to the node 5 is a DAG graph corresponding to the SRE $\mathrm{O}\,(\mathrm{GY}\,|\,\mathrm{B})\,\mathrm{V}$. 97

5.5 A table of optimal costs for the query in Figure 5.2. The first column contains names of categories. The first row contains indices of vertices in a category. Curved arrows on the left indicate computation steps and its orders. 98
5.6 A testbed framework for SRCP queries. 101
5.7 TPQ/GTSPP queries ($k = 5$). 104
5.8 TPQ/GTSPP queries ($g = 10,000$). 104
5.9 Queries with multiple options ($k = 8, g = 10,000$). 105
5.10 GSP queries ($g = 10,000$). 105

List of Tables

3.1 Real-life graphs. 48
3.2 Experiment graphs extracted from Yahoo! AltaVista Web Page Hyperlink Connectivity Graph. 52
3.3 Execution time (sec) on Amazon Product Dataset. 56

4.1 Mappings of states in the SFA shown in Figure 4.2. 63
4.2 An example of the output of the partial evaluation. 67
4.3 Real-life datasets. 71

5.1 The number of computation steps in each query. 106
5.2 Performance of optimizations. 107

1 Introduction

1.1 Motivations and Objectives

Graphs are flexible in modeling many kinds of data from the unstructured to the structured. To query graph data, a typical way is using regular expression to find paths in the graph [1]. Most of graph querying languages supports regular-expression-based queries, for example, Strudel [2], UnQL, [3], SPARQL [4, 5], GXPath [6]. There are many potential applications in various domains using regular-expression-based queries as a key component.

- Social networks: Checking whether there exists a common friend who lives in Tokyo among two American friends [7]. Finding an influence network of a person based on friendship, or location [8].
- Biological networks: Finding every gene whose expression is directly or indirectly affected by a given compound. Finding the shortest path between two substances that includes a third substance [9].

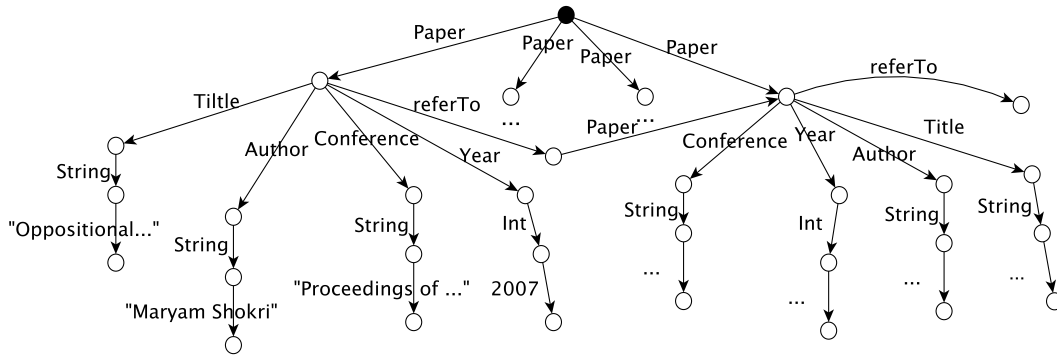


Figure 1.1: A rooted, directed edge-labeled graph of paper citation network.

- World-Wide-Web: Discovering patterns via path analysis patterns [10, 11]. Exploiting the structure and topology of the document networks [12].

With the explosion of data as well as complicated relationships between data entities, graphs are too big to be processed in the main memory of a single machine. This leads to distributed programming models targeting to a scalable processing of big data in general (MapReduce, [13, 14]) as well as big graphs in particular (Pregel, [15]).

Motivated by useful applications of regular-expression-based queries and the scalability of distributed graph processing models, the main objective of this dissertation is to propose automatic mechanisms to translate regular-expression-based queries into efficient programs in distributed graph processing models. These mechanism should be integrated together to form a general framework to query data from big graphs.

1.2 Challenges

Evaluating regular-expression-based queries in a distributed environment is a challenging problem. Let us consider a big graph representing a citation network as shown in Figure 1.1, where information is stored on edges. The following regular-expression-based queries (written in the UnQL+ language [16]),

$$\begin{aligned}&\textbf{select}\ \ \$p\\&\textbf{where}\ \{_^{*}\,.\,\texttt{Paper} : \$p\}\ \textbf{in}\ \$db,\\&\qquad\quad \texttt{Year}\,.\,\texttt{Int}\,.\,\texttt{2010}\ \textbf{in}\ \$p\end{aligned}$$

returns all papers published in 2010; it finds all subgraphs (bound by $*p*) reachable from the root (the black vertex in Figure 1.1) by a path whose edge labels form a word in the regular expression `_*.Paper`, while these subgraphs must satisfy the condition that they contain a path whose edge labels form a word in the regular expression `Year.Int.2010`.

To this query, the decision whether to include a paper bound by $*p* in the result or not needs additional computation to check the conditions over $*p*, which leads to two difficulties in evaluating the conditions efficiently. First, when we go deeper along directed edges to check the condition of $*p*, we will have to keep a record of where $*p* is bound, so as to trace back after checking. Second, the subgraph $*p* may refer to the whole big graph because of possible cycles and the condition may be involved and time-consuming, so we need to find a good way to parallelize the checking process and make it work efficiently.

The problem becomes even more difficult when there are subqueries and additional conditions. Consider that we would like to compute an influence graph of a paper based on citations.

```
select
  (select $a
   where
     {Author : $a} in $p,
     LivingIn.America in $a)
where
  {_*.Paper : $p} in $db,
  Year.Int.2010 in $p,
  _*.referTo.Paper.Title."Pregel" in $p
```

This query finds all authors living in America who have papers published in 2010 which directly/indirectly refer to the paper entitled "`Pregel`". To this query, the difficulty is not only evaluating conditions over graphs, but also checking whether a graph $*a* belongs to a graph $*p* or not. It is because these two graphs may be stored at two different machines.

In addition, although distributed processing models are designed to be scalable to

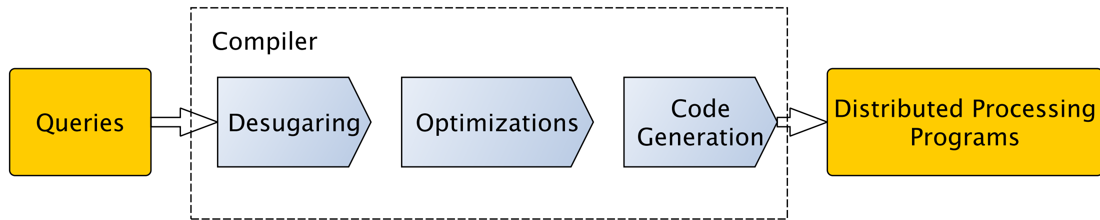


Figure 1.2: Overview of a querying framework for graphs.

big data, it is non-trivial to describe graph algorithms [17, 18, 14]. This is because, in order to obtain a good scalability, these models are often limited to a specific form of computation. For example, the MapReduce model requires user describing computations in functions "Map" and "Reduce", while the Pregel model requires writing programs in a common function that applies on every vertex of a graph. Some regular-expression-based queries have been proposed on top of MapReduce or Pregel. For example, Fan et al. [19] proposed a MapReduce algorithm to answer point-to-point regular reachability queries, which returns a boolean answer. Nolé et al. [20] proposed a Pregel algorithm to processing regular path queries (a subset of GXPath [6], not including backward navigation and branching), that returns a set of pairs of vertices connecting by a path satisfying the regular expression. Nonetheless, to the best of our knowledge, none of them returns a graph as a result, making queries not able to be nested or composed.

1.3 Problem Statement

A general framework for evaluating select-where regular path queries usually consists of the following components (as shown in Figure 1.2).

- **Desugaring**: This converts input queries into a calculational form which represents computations over graphs.

- **Optimization**: This component consists of high-level optimizations. Rules are applied to transform a naive program into an efficient one. Rules can be classified into two levels: general rules to minimizes operations of a program;

and model-targeted rules to make a program fit well with underlying distributed processing models (e.g., MapReduce, Pregel).

- **Code Generation**: This component accepts optimized programs and generates codes in the underlying distributed processing models.

With the above framework, the following research questions are raised.

- What are suitable calculational forms that not only allow expressing regular-expression-based queries on graphs and enjoy powerful optimization rules, but also potentially enable parallelism.
- Once we have found a suitable calculational form, how to transform a program written in that calculational form into one that consists of basic algorithms in the underlying distributed processing models. Once this can be done, we can utilize efficient algorithms that are well developed.
- How can the framework be extended to aggregation operators, e.g. shortest paths based on regular expressions.

1.4 Contributions

By answering those research questions, this dissertation makes the following four contributions.

First, we propose a structural-recursion-based approach to obtain Pregel programs automatically for a subset of select-where regular path queries. These queries are in the form of

$$Q_P : \textbf{select } Q_{P1}(\$g_1) \textbf{ where } \{RE : \$g_1\} \textbf{ in } \$g$$

This approach can help not only utilize many useful optimization rules developed for structural recursions, but also obtain an efficient Pregel program. In this approach, no matter how complex a query is, it is finally expressed in the form of *a structural recursive function* or a composition of structural recursive functions, by applying optimization rules. We then derive an efficient algorithm that minimizes intermediate data generated during its computation. Our idea is a *mark-and-generate* computation. In the phase *Mark*, we first mark each vertex with a set of states that are yielded by the

automaton of a regular expression for paths starting from the root vertex to each vertex. Then, in the *Generate* phase, we use a bulk computation to generate the final result by applying a common computation on every edge in parallel.

Second, we propose a novel approach to obtain Pregel programs for another subset of select-where regular path queries that consist of conditions over graphs. In general, this class has the form of

$$
\begin{aligned}
Q_C : \textbf{select } Q_{C1}(\$g_1) \ \ \textbf{where} \quad & \\
& \{RE : \$g_1\} \textbf{ in } \$g, \\
& RE \textbf{ in } \$g_1, \\
& \ldots
\end{aligned}
$$

where the condition "RE **in** $\$g_1$" says whether there exists a path in the graph $\$g_1$ whose edge labels form a word in RE. We may have multiple conditions over a graph. Our idea is a *generate-and-test* computation: speculatively generating results without considering the condition, and pruning (by testing) those that do not satisfy the condition. This idea is based on the observation that a single condition check could double the computation because of speculative computation compared to sequential computation, but as will be seen later, it can be fully compensated by a carefully designed full parallel computation with multiple processors. To this end, we rewrite these queries into two queries in the first class (Q_{P1}, Q_{P2}) and a specific iterative parallel algorithm. The first query Q_{P1} is to speculatively compute both result graphs (each constructed from the **select** part) and conditional graphs (each constructed from the **where** part) and group them into a single intermediate graph. Our proposed iterative parallel algorithm is to propagate conditional checking results around the intermediate graph. The second query Q_{P2} extracts the final result from the propagated graph. Interestingly, the query Q_{P2} is always the same and independent of the input query.

Third, we propose a functional-based approach to obtain a parallelism in answering a point-to-point regular reachability query. By lifting a deterministic finite automaton of a regular expression to a simultaneous finite automaton whose states are states-to-states mapping functions, we propose an efficient distributed algorithm in the MapReduce model. Practical results show that this approach can significantly speedup local computations and reduce communication overhead. We show how this functional-

based approach can be extended to apply to the phase *Mark* that takes much time during the evaluation of select-where regular path queries.

Fourth, we propose a shortest regular category-path query to return shortest paths whose vertex labels form a *category-based regular expression*. We show that a select-where regular path queries extended with shortest-path conditions can be solved by using a shortest regular category-path query. For example, a query,

$$\begin{aligned} Q_S:\ & \textbf{select} \\ & \quad \textbf{select}\ \$g_2 \\ & \quad \textbf{where}\ \{RE_2 : \$g_2\}\ \textbf{in}\ \$g_1 \\ & \quad\quad\quad \$g_2\ \textbf{closest to}\ \$g \\ & \textbf{where}\ \{RE_1 : \$g_1\}\ \textbf{in}\ \$g \end{aligned}$$

returns all graphs $\$g_2$ that are followed by a path—starting from the root of $\$g$—whose labels form a regular expression "$RE_1.RE_2$" and whose length is shortest (assuming that the length of a path is the number of edges on the path). This query requires to find all vertices (roots of $\$g_2$) connected from the root of $\$g$ by a path whose vertex labels form a category-based regular expression "$_ * .C_1._ * .C_2$", where C_1 and C_2 are sets (categories) of roots of $\$g_1$ and $\$g_2$, respectively. By using a dynamic programming formulation, we show that a shortest regular category-path query can be efficiently answered by a sequence of single source shortest path searches. This is useful because we can utilize fast single source shortest path algorithms that are optimized for different types of graphs (road networks, social networks, biological networks) and environments (shared or distributed memory).

1.5 Dissertation Overview

This dissertation is organized as follows.

In chapter 2, we introduce basic notions and models that are used throughout the dissertation.

In chapter 3, we give a formal definition of select-where regular path queries. Our contributions in translating select-where regular path queries into Pregel programs are discussed in detail. We design and implement a light-weight framework based

on our solutions to answer select-where regular path queries. Experimental results with real-life graph instances show that our framework has a good scalability. The intermediate graphs generated are small compared with the input graph.

In chapter 4, we introduce a functional-based approach to answer point-to-point regular reachability queries. This serves as a preliminary experiment to show the advantage of the functional-based approach in answering regular reachability queries. Our experiment with the MapReduce model shows that this approach gains a significant speedup compared with state-of-the-art approaches. We end this chapter by showing how this approach can be extended to speedup the evaluation of select-where regular path queries.

In chapter 5, we introduce a shortest regular category-path query and show its role in integrating shortest-path conditions into select-where regular path queries. We show how to reduce the query to a series of single source shortest path searches. Experimental results on road networks show that our solution can utilize existing fast single source shortest path algorithms.

In chapter 6, we give a summary of the dissertation and discuss future work.

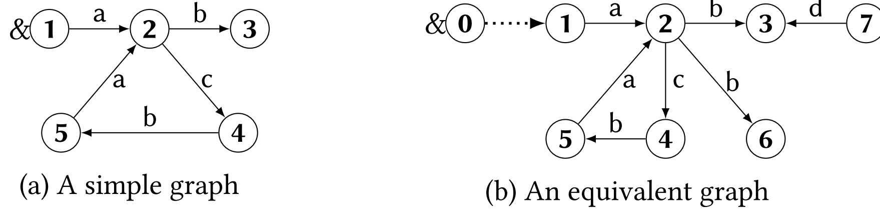
2
Preliminaries

In this chapter, we introduce basic notions that are used throughout the dissertation. We give a definition of graph data model. Graphs in this model are up to bisimilarity. Graph constructors are introduced to help build a graph from smaller graphs. Graph constructors are also used in queries to construct the final result. These notions are borrowed from UnCAL—an unstructured calculus for querying graph data. Then, we give a definition of regular expression we focus on in this dissertation. Finally, we briefly explain some well-known distributed processing models for big graphs.

2.1 Graph Data Models

2.1.1 Definitions

Following UnCAL (a unstructured calculus for querying graph data [3]), a graph is modeled as a directed edge-labeled graph extended with *markers* and ϵ*-edges*. It is shown that this graph model is powerful enough for representing various datasets



(a) A simple graph

(b) An equivalent graph

Figure 2.1: Examples of rooted edge-labeled graphs.

from the unstructured, the semistructured to the structured [21, 22]. In this model, edges contain data, while vertices are unique identity objects without labels. Markers (with a prefix &) are symbols to mark certain vertices as *input vertices* or *output vertices*. Edges labeled with a special symbol ϵ are called ϵ*-edges*. One could consider markers as initial/final states and ϵ-edges as “empty” transitions in automata.

Let $\mathcal{L}$ be a set of labels, $\mathcal{L}_\epsilon$ be $\mathcal{L} \cup \{\epsilon\}$, and $\mathcal{M}$ be a set of markers denoted by $\&x$, $\&y$, $\&z$, ... There is a distinguished marker $\& \in \mathcal{M}$ called a *default marker*.

Definition 2.1 (Directed Edge-Labeled Graph [3]) *A directed edge-labeled graph G is a quadruple $(\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O})$, where $\mathcal{V}$ is a set of vertices, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{L}_\epsilon \times \mathcal{V}$ is a set of edges, $\mathcal{I} \subseteq \mathcal{M} \times \mathcal{V}$ is an one-to-one mapping from a set of input markers to $\mathcal{V}$, and $\mathcal{O} \subseteq \mathcal{V} \times \mathcal{M}$ is a many-to-many mapping from $\mathcal{V}$ to a set of output markers.*

For $\&x, \&y \in \mathcal{M}$, let $v = \mathcal{I}(\&x)$ be the unique vertex such that $(\&x, v) \in \mathcal{I}$, we call v an *input vertex*. If there exists a $(v, \&y) \in \mathcal{O}$, we call v an *output vertex*. Note that there are no edges coming to input vertices or leaving from output vertices. Let $DB^{\mathcal{X}}_{\mathcal{Y}}$ denote data graphs with sets of input markers $\mathcal{X}$ and output markers $\mathcal{Y}$. When $\mathcal{X} = \{\&\}$, $DB^{\mathcal{X}}_{\mathcal{Y}}$ is abbreviated to $DB_{\mathcal{Y}}$, and $DB_{\varnothing}$ is abbreviated to DB. A graph that has only one input marker $\mathcal{X} = \{\&\}$ and has no output marker $\mathcal{Y} = \varnothing$ is called *a rooted graph*, in which the vertex $v = \mathcal{I}(\&)$ is called the root vertex of the graph. Graphs with multiple markers are internal data structures that are generated with graph constructors.

Figure 2.1(a) shows an example of a rooted directed edge-labeled graph in which $\mathcal{V} = \{1, 2, 3, 4, 5\}$, $\mathcal{E} = \{(1, \mathrm{a}, 2), (2, \mathrm{b}, 3), (2, \mathrm{c}, 4), (4, \mathrm{b}, 5), (5, \mathrm{a}, 2)\}$, $\mathcal{I} = \{(\&, 1)\}$, and $\mathcal{O} = \{\}$. The vertex with id 1 is the root of the graph, and is marked with &.

Definition 2.2 (Path) *A path ρ in a directed edge-labeled graph is a sequence of edges,*

denoted by

$$v_1 \xrightarrow{l_1} v_2 \xrightarrow{l_2} v_3 \xrightarrow{l_3} \cdots \xrightarrow{l_{k-1}} v_k,$$

where $(v_i, l_i, v_{i+1}) \in \mathcal{E}, 1 \le i \le k-1$.

We denote $(u, \epsilon^*, v) \Subset \mathcal{E}$ *and* $(u, \epsilon^*.a, v) \Subset \mathcal{E}$ *whenever there exists a path from* u *to* v *whose edge labels are* $\epsilon \ldots \epsilon$ *or* $\epsilon \ldots \epsilon a$*, respectively.*

Definition 2.3 (Weighted Graph) *A graph* $G^w = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O}, w)$ *is a weighted graph, if it is a directed edge-labeled graph, and the function* $w : \mathcal{L}_\epsilon \to \mathbb{R}_+$ *is to map each edge label to a positive, real-valued weight.*

Definition 2.4 (Vertex-Labeled Graph) *A vertex-labeled graph is a directed edge-labeled graphs extended with a labeling function over vertices. We denote a vertex-labeled graph by* $G^v = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O}, \sigma)$*, where* $\sigma : \mathcal{V} \to \mathcal{L}$ *is a function mapping each vertex in* G^v *to a label.*

We adopt the model for distributed vertex-labeled graphs presented in [23]. In a distributed graph, the vertices are partitioned into N fragments. Each fragment is maintained at a site, whereas each site may take charge of more than one fragment.

Definition 2.5 (Distributed Vertex-Labeled Graphs) *A distributed graph for a graph* $G^v = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O}, \sigma)$ *is a set of fragments, each of which is a 6-tuple. For example, the i-th fragment is* $DG_i = (\mathcal{V}_i, \mathcal{E}_i, \mathcal{I}_i, \mathcal{O}_i, \sigma_i, \mathcal{C}_i,)$. $\mathcal{V}_i$ *is a partition of* $\mathcal{V}$*, i.e.,* $\mathcal{V}_1 \uplus \cdots \uplus \mathcal{V}_n = \mathcal{V}$. $\mathcal{E}_i = \mathcal{E} \cap (\mathcal{V}_i \times \mathcal{L}_\epsilon \times \mathcal{V}_i)$ *denotes the corresponding edges.* σ_i *is the labeling function for* $\mathcal{V}_i$. $\mathcal{C}_i = \mathcal{E} \cap (\mathcal{V}_i \times \mathcal{L}_\epsilon \times (\mathcal{V} \setminus \mathcal{V}_i))$ *is the set of* cross edges*, each of which connects a vertex in* DG_i *to a vertex in another fragment.* $I_i = \{u \mid (v, u) \in (C_1 \cup \cdots \cup C_n), u \in V_i\}$ *denotes the set of input vertices.* $O_i = \{u \mid (v, u) \in C_i\}$ *denotes the set of output vertices. It is worth noting that output and input vertices are duplicated among fragments.*

Example 2.1 *Figure 2.2 shows an example of a distributed graph that is partitioned into three fragments,* DG_1*,* DG_2*, and* DG_3*. For example,*

$$DG_1 = (\mathcal{V}_1, \mathcal{E}_1, \mathcal{I}_1, \mathcal{O}_1, \sigma_1, \mathcal{C}_i)$$

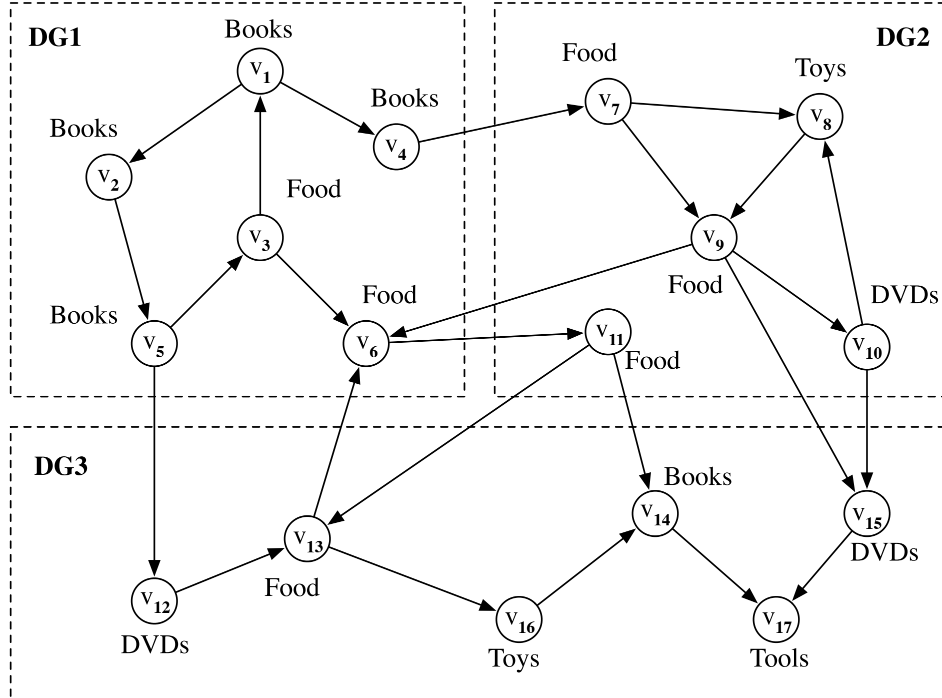


Figure 2.2: An example of a distributed vertex-labeled graph including 3 fragments. Here, we ignore edge labels (assuming that edges have no labels).

where

$$\mathcal{V}_1 = \{v_1, v_2, v_3, v_4, v_5, v_6\},$$
$$\mathcal{E}_1 = \{(v_1, v_2), (v_1, v_4), (v_2, v_5), (v_3, v_1), (v_3, v_6), (v_5, v_3)\},$$
$$\mathcal{I}_1 = \{(\&z_1, v_6)\},$$
$$\mathcal{O}_1 = \{(v_7, \&z_2), (v_{11}, \&z_3), (v_{12}, \&z_4)\}$$
$$\mathcal{C}_1 = \{(v_4, v_7), (v_6, v_{11}), (v_5, v_{12})\}.$$

Here, for brevity, we ignore edge labels when labels are empty. For example, (v_1, v_2) *denotes the edge with an empty label* $(v_1, \text{“ ”}, v_2)$.

2.1.2 The Meaning of ϵ-Edges

Theoretically, an ϵ-edge from a vertex v to v' means that all edges emanating from v' should be emanating from v [3]. Eliminating an ϵ-edge (v, ϵ, v') means removing this

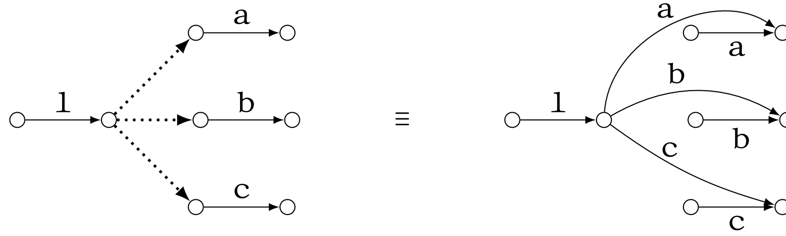


Figure 2.3: Meaning of ϵ-edges. The left graph with ϵ-edges is value equivalent to the right graph without ϵ-edges.

ϵ-edge and for each edge, (v', a, w), emanating from v', adding a new edge (v, a, w). Figure 2.3 shows an example of two equivalent graphs: one contains ϵ-edges and the other has no ϵ-edges. In this dissertation, we use dotted arrows to denote ϵ-edges.

2.1.3 Graph Equivalence

Two directed edge-labeled graphs G_1 and G_2 are *value equivalent*, in notation $G_1 \equiv G_2$, if there exists an extended bisimulation from G_1 to G_2.

Definition 2.6 ([3]) *Let $G_1 = (\mathcal{V}_1, \mathcal{E}_1, \mathcal{I}_1, \mathcal{O}_1), G_2 = (\mathcal{V}_2, \mathcal{E}_2, \mathcal{I}_2, \mathcal{O}_2)$ be two graphs, both with input markers $\mathcal{X}$ and output markers $\mathcal{Y}$. An extended simulation from G_1 to G_2 is a relation $\mathcal{S} \subset \mathcal{V}_1 \times \mathcal{V}_2$ such that:*

- *if $(u_1, u_2) \in \mathcal{S} \wedge (u_1, \epsilon^*.a, v_1) \Subset \mathcal{E}_1$ with $a \neq \epsilon$, then there exists a node v_2 s.t. $(u_2, \epsilon^*.a, v_2) \Subset \mathcal{E}_2$ and $(v_1, v_2) \in \mathcal{S}$,*
- *if $(u_1, u_2) \in \mathcal{S} \wedge (\&x, u_1) \in \mathcal{I}_1$ then $(\&x, u_2) \in \mathcal{I}_2$,*
- *if $(u_1, u_2) \in \mathcal{S} \wedge (u_1, \epsilon^*, v_1) \Subset \mathcal{E}_1 \wedge (v_1, \&y) \in \mathcal{O}_1)$, then there exists a node v_2 s.t. $(u_2, \epsilon^*, v_2) \Subset \mathcal{E}_2 \wedge (v_2, \&y) \in \mathcal{O}_2$, and*
- *$(\mathcal{I}_1(\&x), \mathcal{I}_2(\&x)) \in \mathcal{S}$, for every $\&x \in \mathcal{X}$.*

An extended bisimulation from G_1 to G_2 is an extended simulation $\mathcal{S}$ for which $\mathcal{S}^{-1}$ is also an extended simulation.

For example, the graph in Figure 2.1(a) is value equivalent to the graph in Figure 2.1(b). The new graph (Figure 2.1(b)) has an additional ϵ edge from the root and an

edge $(7, \mathrm{d}, 3)$ unreachable from the root. It adds a new edge labeled b, $(2, \mathrm{b}, 6)$, from the vertex 2.

2.2 Graph Constructors

Before looking at graph constructors in details, we need to define an additional operation "·" (Skolem function) to generate new markers. The operation · returns a different marker for every pair of $\&x$ and $\&y$. We assume · to be associative, $(\&x \cdot \&y) \cdot \&z = \&x \cdot (\&y \cdot \&z)$, and & to be its identity, $\& \cdot \&z = \&z \cdot \& = \&z$. Given two sets of markers $\mathcal{X}, \mathcal{Y}$, we denote $\mathcal{X} \cdot \mathcal{Y}$ the set $\{\&x \cdot \&y \mid \&x \in \mathcal{X}, \&y \in Y\}$.

There are nine graph constructors in UnCAL. From these constructors, we can build arbitrary directed edge-labeled graphs.

$$
\begin{array}{lcll}
G & ::= & \{\} & \{\text{empty graph (one vertex, one input marker)}\} \\
 & \mid & \{l : G\} & \{\text{singleton graph (an edge pointing to the root of a graph)}\} \\
 & \mid & G \cup G & \{\text{graph union}\} \\
 & \mid & \&x := G & \{\text{relabel the root vertex with a new input marker}\} \\
 & \mid & \&y & \{\text{graph with one output marker}\} \\
 & \mid & () & \{\text{empty data graph (no vertices, edges and markers)}\} \\
 & \mid & G \oplus G & \{\text{disjoint union}\} \\
 & \mid & G \mathbin{@} G & \{\text{append of two graphs}\} \\
 & \mid & \mathbf{cycle}(G) & \{\text{graph with cycles}\}
\end{array}
$$

Intuitively, definitions of the constructors are given in Figure 2.4. Informally, $\{\}$ constructs a graph of only one vertex labeled with default input marker &, $\{l : G\}$ constructs a new graph G' from the graph G by adding the edge labeled l pointing to the root of G. The source vertex of l becomes the root of G'. The operator $\cup$ unions two graphs of the same input markers with the aid of ϵ-edges. The next two constructors allow us to add input and output markers: $\&z := G$ takes a graph $G \in DB^{\mathcal{X}}_{\mathcal{Y}}$ and relabels input vertices with the input marker $\&z$, thus the result is in $DB^{\mathcal{Z} \cdot \mathcal{X}}_{\mathcal{Y}}$; $\&y$ returns a graph of a single vertex labeled with the default input marker & and the output marker $\&y$. $()$ constructs an empty graph without any markers and vertices. The disjoint union $G_1 \oplus G_2$ requires two graphs G_1 and G_2 have disjoint sets of input markers.

The operator $G_1 \,@\, G_2$ vertically constructs a graph by adding ϵ-edges from output vertices of G_1 to input vertices with the same markers of G_2. It requires $G_1 \in DB^{\mathcal{X}}_{\mathcal{Y}}$ and $G_2 \in DB^{\mathcal{Y}}_{\mathcal{Z}}$, thus $G_1 \,@\, G_2 \in DB^{\mathcal{X}}_{\mathcal{Z}}$. Finally, the last operator allows us to introduce cycles by adding ϵ-edges from an output marker to the input marker named after it.

Example 2.2 *The graph in Figure 2.1(a) can be constructed as follows (but not uniquely).*

$$
\begin{aligned}
\&z \,@\, \mathbf{cycle}((&\&z := \{a : \&z_1\}) \\
&\oplus (\&z_1 := \{b : \{\}\} \cup \{c : \{b : \&z_2\}\}) \\
&\oplus (\&z_2 := \{a : \&z_1\})) \qquad \square
\end{aligned}
$$

For brevity, we write $\{l_1 : G_1, \dots, l_n : G_n\}$ to denote $\{l_1 : G_1\} \cup \dots \cup \{l_n : G_n\}$, and $(G_1, \dots, G_n)$ to denote $G_1 \oplus \dots \oplus G_n$.

2.3 Regular Expression

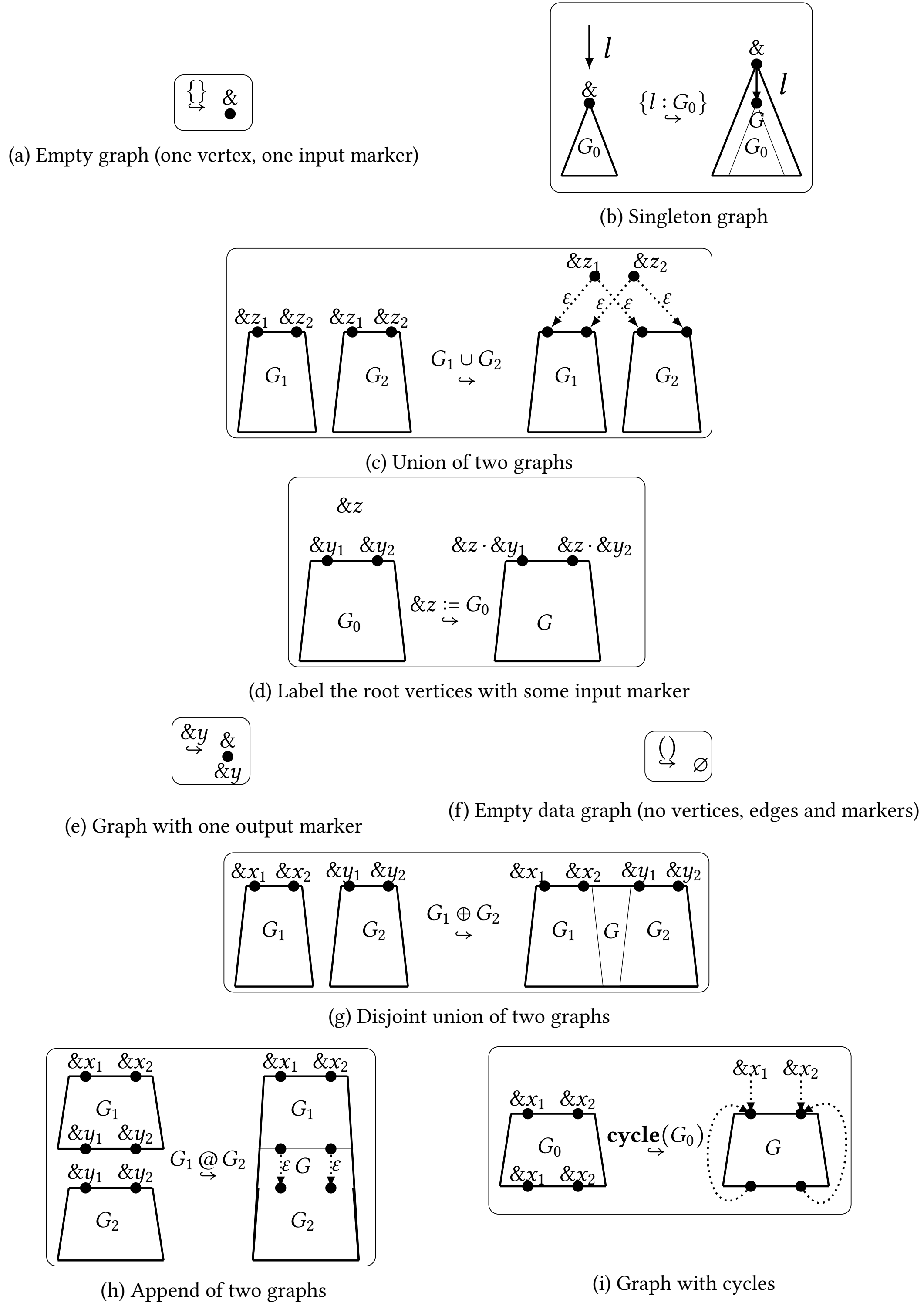
The syntax of a regular expression is:

$$R ::= a \;|\; _ \;|\; R\text{``}|\text{''}R \;|\; R\text{``}*\text{''} \;|\; R\text{``}.\text{''}R \;|\; (R)$$

where $a \in \mathcal{L}_\epsilon$ is a label, and _ denotes a *wildcard* that matches any label. Further, "|", "*" and "." denote alternation, Kleene closure, and concatenation, respectively. "." can be omitted. We may write $R+$ as a shorthand for $RR*$.

Definition 2.7 *(Path Satisfaction) A path $\rho = v_1 \xrightarrow{l_1} v_2 \xrightarrow{l_2} v_3 \xrightarrow{l_3} \cdots \xrightarrow{l_{k-1}} v_k$ is said to satisfy a regular expression R if the string "$l_1 l_2 \dots l_{k-1}$" spells out R.*

It is well known that any regular expression can be translated into a deterministic finite automaton (DFA) [24].



(a) Empty graph (one vertex, one input marker)

(b) Singleton graph

(c) Union of two graphs

(d) Label the root vertices with some input marker

(e) Graph with one output marker

(f) Empty data graph (no vertices, edges and markers)

(g) Disjoint union of two graphs

(h) Append of two graphs

(i) Graph with cycles

Figure 2.4: Graph constructors.

2.4 Distributed Programming Models

2.4.1 MapReduce

MapReduce [13] is a framework to process big data. A widely used open source implementation is Hadoop [25]. In MapReduce programming model, users only need to write two functions: Map and Reduce. Figure 2.5 shows the MapReduce programming model. Map functions accept a list of key-value pairs of (*key1*, *value1*) as its input and produce a list of pairs of (*key2*, *value2*). After that, Shuffle and Sorting phase will collect pairs with the same key and group them into pairs of (*key2*, *listOfValues*), these phases are automatically done by the underlying system. For each different key and a list of its values, the system will invoke a reduce function to process. Reduce functions will emit results that are pairs of (*key3*, *value3*). Data, which are used during computation of a MapReduce job, are usually stored in a high performance distributed file system.

2.4.2 Pregel

Pregel [15] is a model to process big graphs in a distributed way. It is widely used by Google and Facebook to analyze big graphs. Figure 2.6 shows the Pregel programming model. It is inspired by the *Bulk-Synchronous Parallel* (BSP) model [26] whose computation consists of a sequence of *supersteps*. It follows the vertex-centric approach where a common function, *compute*(), is applied to *every vertex*. A vertex can access its outgoing edges locally. During a superstep, a vertex receives messages from the other vertices, does its computations (updating its value, mutating outgoing edges, etc.), and sends messages to the other vertices. One vertex can decide not involving to the next superstep by voting to halt (to be inactive). A computation terminates when there is no message in transit or every vertex becomes inactive. Machines used to do vertex computations are called workers. A master is responsible for coordinating the activities of workers. A Pregel phase is a sequence of supersteps to do a computation unit logically.

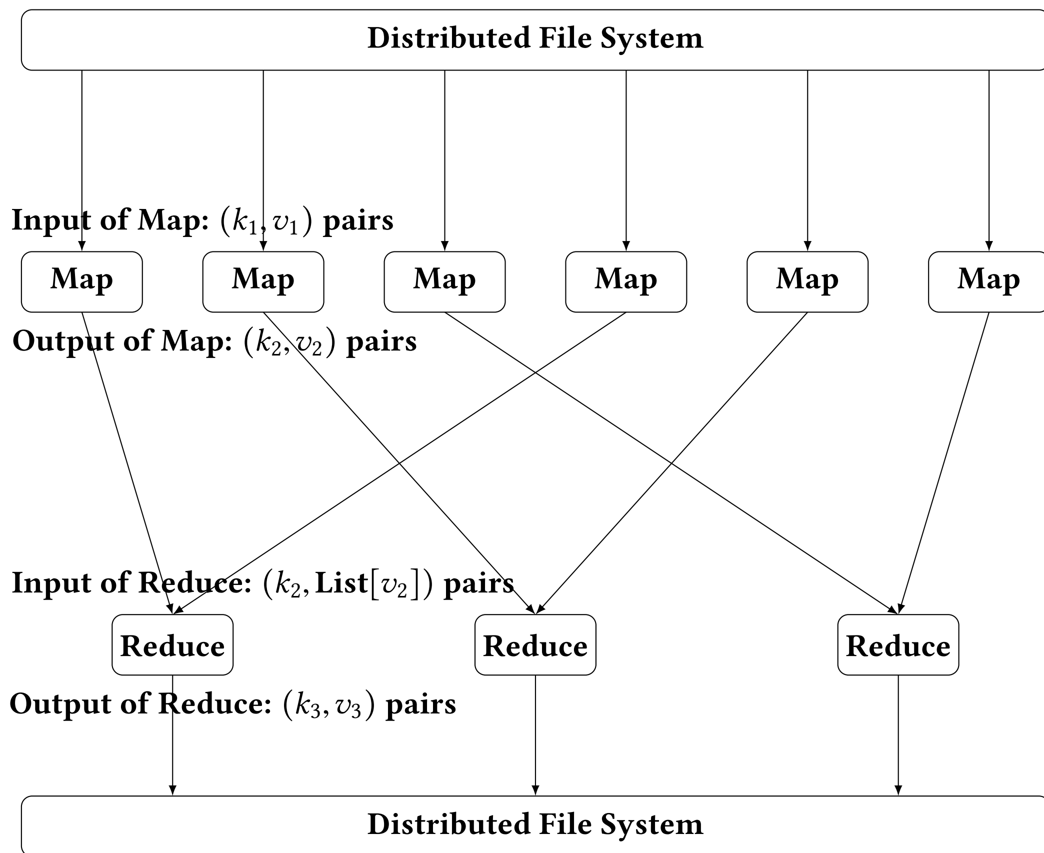


Figure 2.5: MapReduce programming model.

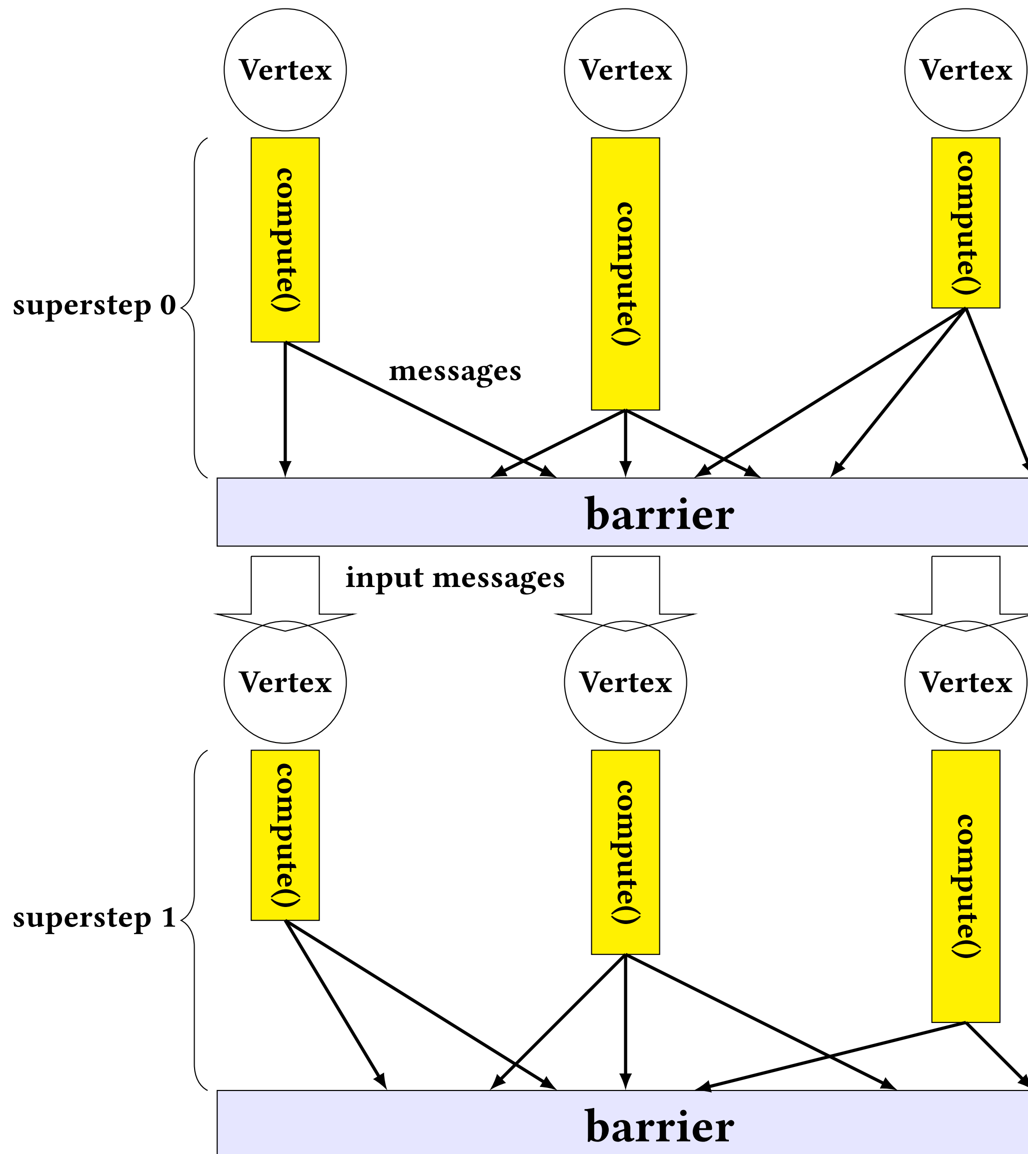


Figure 2.6: Pregel programming model.

Example 2.3 (Maximum Value Example [15]) *Figure 2.7 shows an example that propagates the maximum value of vertices' values to every vertex. During a superstep, a vertex learns the maximum value from coming messages sent by its neighbors. If its value is updated to a larger value, it sends the new value to its neighbors. Otherwise, it sends nothing, and becomes inactive (by voting to halt). The algorithm terminates when there are no messages in transit. Listing 2.1 shows the pseudocode for the algorithm.*

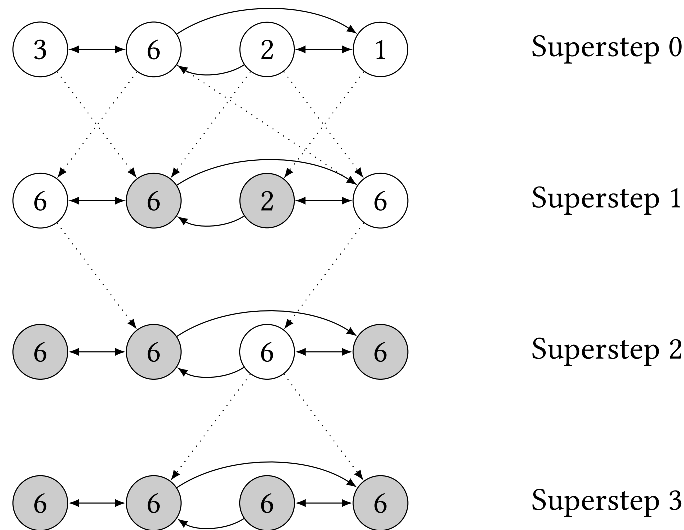


Figure 2.7: Maximum value example [15] in the Pregel model. Dotted lines are messages. Shaded circles are vertices voted to halt.

```
void compute(Vertex vertex, Iterable<MsgValue> msgs)
{
 if (getSuperstep() == 0) {
  // send messages to the neighbors
  for (Edge edge : vertex.getEdges()){
   VertexId targetId = edge.getTargetVertexId();
   sendMessage(targetId, new Message(vertex.getValue()));
  }
 } else {
  // get the maximum value from received messages
  Long MaxValue = vertex.getValue();
  for (MsgValue msg : msgs){
   MaxValue = (msg > MaxValue) ? msg : MaxValue;
  }

  if (MaxValue > vertex.getValue()){
   // update the vertex value
   vertex.setValue(MaxValue);
   // send messages to the neighbors
   for (Edge edge : vertex.getEdges()){
    VertexId targetId = edge.getTargetVertexId();
    sendMessage(targetId, new Message(MaxValue));
   }
  } else {
   voteToHalt();
  }
 }
}
```

Listing 2.1: Pseudocode for the function *Vertex*.**compute**() in the Pregel model to compute the maximum value.

3

Select-Where Regular Path Queries

In this chapter, we formally define select-where regular path queries. Our contributions are presented in detail, where we show how to obtain Pregel programs for select-where regular path queries. Structural recursion is introduced as a calculation form to express queries. Although it has been shown that structural recursion is a powerful tool to systematically develop parallel programs on lists, arrays and trees [27, 28, 29, 30], it is new to use it to derive scalable programs on graphs. In this chapter, we propose efficient computations for specifications written by structural recursion on graphs. Based on our solutions, we design and implement a light-weight framework on top of Pregel. We show experimental results in detail and conclude this chapter with a discussion.

3.1 Definition of Select-Where Regular Path Queries

A select-where regular path (SWRP) query $Q(\$g)$ on a rooted directed edge-labeled graph $\$g$ is defined as follows (the syntax is borrowed from the UnQL language [3]).

$$\begin{aligned} Q(\$g) ::= &\ \mathbf{select}\ E(\$g) \\ |&\ \mathbf{select}\ E(\$g_1)\ \mathbf{where}\ \{R : \$g_1\}\ \mathbf{in}\ \$g \\ |&\ \mathbf{select}\ E(\$g_1)\ \mathbf{where}\ \{R : \$g_1\}\ \mathbf{in}\ \$g, P(\$g_1) \\ |&\ \mathbf{select}\ E(\$l, \$g_1)\ \mathbf{where}\ \{R : \{\$l : \$g_1\}\}\ \mathbf{in}\ \$g, P(\$g_1) \end{aligned}$$

where R represents a regular expression described in Section 2.3. The **select** part is to return the final result by using an expression E over a graph variable or a pair of a label variable ($\$l$) and a graph variable ($\$g$). The **where** part is a generator using regular expressions, e.g., "$\{R : \$g_1\}$ **in** $\$g$" is to generate graphs following paths—starting from the root of $\$g$—whose labels form R. Each of these graphs is bound by $\$g_1$. Besides, the generator "$\{R : \{\$l : \$g_1\}\}$ **in** $\$g$" allows binding labels of edges following paths satisfied R to a label variable $\$l$ and use the label variable to construct the final result in an expression E. Inside the **where** part, we can define conditions P over a graph variable ($\$g_1$) as well as over label variables.

An expression E on a graph $\$g$ is to construct the final result. The expression $E(\$g)$ may consist of three of graph constructors, sub-queries ($Q(\$g)$), conditional statements, user-defined functions (*UDF*), and function application (*fname*).

$$\begin{aligned} E(\$g) ::= &\ \{\} \mid \{a : E(\$g)\} \mid E(\$g) \cup E(\$g) \mid Q(\$g) \\ |&\ \mathbf{if}\ P(\$g)\ \mathbf{then}\ E(\$g)\ \mathbf{else}\ E(\$g) \\ |&\ UDF(\$g), fname(\$g) \\ E(\$l, \$g) ::= &\ E(\$g) \mid \{\$l : E(\$g)\} \mid E(\$l, \$g) \cup E(\$l, \$g) \end{aligned}$$

Once there exists a label variable in an expression, we only allow it to be used in graph constructors.

A condition P over a graph $\$g$ is defined as follows.

$$\begin{aligned} P(\$g) ::= &\ \mathbf{isempty}(Q(\$g)) \mid R\ \mathbf{in}\ \$g\ {}^{1} \\ |&\ !P(\$g) \mid P(\$g)\ \&\&\ P(\$g) \mid P(\$g) \,\|\, P(\$g) \\ R ::= &\ a \mid _ \mid R\text{“}|\text{”}R \mid R\text{“}*\text{”} \mid R\text{“}.\text{”}R \mid (R) \end{aligned}$$

where the condition **isempty** is to check whether the result of a query is an empty graph or not; the condition "R **in** $\$g$" is to check whether there exists a path from the root of $\$g$ whose edge labels form R. Conditions can be composed by operators AND (&&), OR (||), or NOT (!).

One can define user-defined functions to do transformations, e.g., changing all edges labeled "`Publication Venue`" to "`Conference`", finding all graphs following paths satisfied a regular expression and removing them, etc. User-defined functions are defined by structural recursive functions as follows.

$$
\begin{aligned}
UDF(\$g) ::= \;& \textbf{let sfun } \mathit{fname}(\{lp_1 : gp_1\}) = E(lp_1, gp_1) \\
& \quad\; | \quad \mathit{fname}(\{lp_2 : gp_2\}) = E(lp_2, gp_2) \\
& \quad\; | \quad \ldots \\
& \quad\; | \quad \mathit{fname}(\{lp_n : gp_n\}) = E(lp_n, gp_n) \\
& \textbf{in } E(\$g) \\
| \;& \textbf{letrec sfun } \mathit{fname}_1(\{lp_{11} : gp_{11}\}) = E(lp_{11}, gp_{11}) \\
& \quad\quad | \quad \mathit{fname}_1(\{lp_{12} : gp_{12}\}) = E(lp_{12}, gp_{12}) \\
& \quad\quad | \quad \ldots \\
& \quad\quad | \quad \mathit{fname}_1(\{lp_{1n} : gp_{1n}\}) = E(lp_{1n}, gp_{1n}) \\
& \textbf{and} \\
& \quad\quad \ldots \\
& \textbf{and} \\
& \quad\quad \textbf{sfun } \mathit{fname}_m(\{lp_{m1} : gp_{m1}\}) = E(lp_{m1}, gp_{m1}) \\
& \quad\quad | \quad \mathit{fname}_m(\{lp_{m2} : gp_{m2}\}) = E(lp_{m2}, gp_{m2}) \\
& \quad\quad | \quad \ldots \\
& \quad\quad | \quad \mathit{fname}_m(\{lp_{mn} : gp_{mn}\}) = E(lp_{mn}, gp_{mn}) \\
& \textbf{in } E(\$g)
\end{aligned}
$$

where a single structural recursive function is defined by using the keywords "**let sfun**" and mutually structural recursive functions are defined by using "**letrec sfun**". Structural recursive functions are discussed in detail in Sections 3.2.1 and 3.2.2.

Let us look at some examples of SWRP query.

[1] our extension to UnQL+ queries

Example 3.1 *The simplest query is a regular path query [31]. Assume that we want to return all papers' titles from a citation graph. A such query is written as follows.*

$$\begin{array}{l} \textbf{\textit{select}}\ \ \$t \\ \textbf{\textit{where}}\ \{\texttt{Paper.Title.String}: \$t\}\ \textbf{\textit{in}}\ \$db \end{array}$$

where, the variable $t is bound to each graph that follows a path starting from a root *in which the concatenation of its edge labels satisfies the regular expression* `Paper.Title.String`. *The result graph is the union of graphs bound by $t.*

Example 3.2 *We can reorganize data returned by graph variables to construct a new graph.*

$$\begin{array}{l} \textbf{\textit{select}}\ \{\texttt{Article}: (\\ \quad \{\texttt{Year}: \textbf{\textit{select}}\ \$y\ \textbf{\textit{where}}\ \{\texttt{Year.Int}: \$y\}\ \textbf{\textit{in}}\ \$p\} \\ \quad \cup \\ \quad \textbf{\textit{select}}\ \$t\ \textbf{\textit{where}}\ \{\texttt{Title.String}: \$t\}\ \textbf{\textit{in}}\ \$p)\} \\ \textbf{\textit{where}}\ \{\texttt{Paper}: \$p\}\ \textbf{\textit{in}}\ \$db \end{array}$$

This query binds variables $t and $y to the title and the year of a paper $p, respectively. After that, it constructs, for each paper, a graph that has one edge labeled `Article` *pointing to the union of two graphs: one has one edge labeled* `Year` *pointing to the graph $y, and another is the graph $t.*

Example 3.3 *This is an example of using conditions in an SWRP query.*

$$\begin{array}{l} \textbf{\textit{select}}\ \ \$p \\ \textbf{\textit{where}}\ \{\texttt{Paper}: \$p\}\ \textbf{\textit{in}}\ \$db, \\ \qquad\quad \texttt{Year.Int.2010}\ \textbf{\textit{in}}\ \$p \end{array}$$

This query returns papers published in 2010. It combines regular path expressions and conditions over graphs. .

Example 3.4 *We can also perform transformations over graphs by using a user-defined function (UDF). For example, for each paper returned by the query in Example 3.3, we*

relabel edges `Conference` *to* `Venue`.

$$
\begin{array}{l}
\textbf{\textit{select}} \\
\quad \textbf{letrec sfun} \\
\qquad c2v(\{\texttt{Conference} : \$g\}) = \{\texttt{Venue} : c2v(\$g)\} \\
\qquad c2v(\{\$l : \$g\}) = \{\$l : c2v(\$g)\} \\
\quad \textbf{in}\ c2v(\$p) \\
\textbf{\textit{where}}\ \{\texttt{Paper} : \$p\}\ \textbf{\textit{in}}\ \$db, \\
\qquad \texttt{Year.Int.2010}\ \textbf{\textit{in}}\ \$p
\end{array}
$$

Here, we use a structural recursive function c2v to relabel edges `Conference`*. The function c2v is defined with matching patterns. This function does not change the structure of the input graph. It only modifies the label of edges—from* `Conference` *to* `Venue`*.*

3.2 Specifications for SWRP Queries

3.2.1 Structural Recursion

Structural recursion is a powerful mechanism to traverse and restructure data in functional programming. It is shown that structural recursion is useful to systematically construct parallel programs on lists, arrays and trees [30]. Structural recursion on graphs was developed from structural recursion on trees, in order to manipulate unstructured data [3]. One of the advantages of structural recursion is the ability of composing multiple structural recursive functions to describe many complex transformations over graphs. Although compositions usually lead to large intermediate graphs or multiple graph traversals, we can solve those problems systematically by rewriting multiple structural recursive functions into one structural recursive function using tupling/fusion rules or marker-directed optimizations [3, 32, 33]. Therefore, once we can transform SWRP queries to structural recursive functions and parallelize structural recursive functions, we can achieve the scalability for SWRP queries on big graphs and utilizes many useful rewriting rules over structural recursion.

Given a function $e :: \mathcal{L}_\epsilon \to DB^{\mathcal{Z}}_{\mathcal{Z}}$, where $\mathcal{Z} = \{\&z_1, \ldots, \&z_n\}$. A function,

$$f :: DB^{\mathcal{X}}_{\mathcal{Y}} \to DB^{\mathcal{X} \cdot \mathcal{Z}}_{\mathcal{Y} \cdot \mathcal{Z}},$$

is called a structural recursion if it is defined by the following equations

$$\begin{aligned} f(\{\}) &= (\&z_1 := \{\}, \ldots, \&z_n := \{\}) \\ f(\{\$l : \$g\}) &= e(\$l, \$g) \mathbin{@} f(\$g) \\ f(\$g_1 \cup \$g_2) &= f(\$g_1) \cup f(\$g_2) \\ f(\&y) &= (\&z_1 := \&y \cdot \&z_1, \ldots, \&z_n := \&y \cdot \&z_n) \end{aligned}$$

Intuitively, evaluation proceeds as follows: starts from the root of a graph and checks one of three cases. If the graph is empty, it applies the first line (return $\{\}$). If the graph is a singleton graph then it applies the second line: this leads to a recursive call of f on a subgraph. Finally, if the graph is not a singleton, then it decomposes the graph arbitrarily into two graphs $\$g_1 \cup \$g_2$, and it applies the function f recursively on each of them. The fourth line deals with graphs with only one output vertex, e.g, when computing $f(\{\$l : \&y\})$, we need to compute $f(\&y)$. It is obvious that the function terminates on trees, and interestingly it has been shown that the function terminates on graphs with cycles as well [3].

In this dissertation, the first, the third and the fourth equations are always in those forms, so we omit them in the sequel and encode f as $\mathbf{rec}_{\mathcal{Z}}(\lambda(\$l, \$g).e)$. For brevity, sometimes we just write $\mathbf{rec}(e)$.

Example 3.5 *The following structural recursion* **a2d_xc** *relabels edges* a *to* d *and contracts edges* c*. Applying this function to the graph in Figure 3.1(a) results in the graph in Figure 3.1(b).*

$$\begin{aligned} \mathbf{a2d_xc}(\$db) = \mathbf{rec}(\lambda(\$l, \$g).\ & \\ & \mathbf{if}\ \$l = a\ \mathbf{then}\ \{d : \&\} \\ & \mathbf{else\ if}\ \$l = c\ \mathbf{then}\ \{\epsilon : \&\} \\ & \mathbf{else}\ \{\$l : \&\})(\$db) \end{aligned}$$ □

3.2.2 Expressing SWRP Queries by Structural Recursion

The syntax of a specification for an SWRP query is shown in Figure 3.2. Functions between the keywords **main** and **where** are the starting point in a specification. By

(a) An Input Graph

(b) The Output Graph

Figure 3.1: Example **a2d_xc**: relabels edges a to d and contracts edges c.

prog	::=	**main** $f\ [\circ\ f]$ **where** *decl*···*decl*	{ specification }
decl	::=	$f(\{l : \$g\}) = t$	{ structural recursive function }
t	::=	$\{\} \mid \{l : t\} \mid t \cup t$	{ graph constructors }
	$\mid$	$f(\$g)$	{ function application }
	$\mid$	**if** *bcond* **then** t **else** t	{ if_then_else }
bcond	::=	**isempty**(t)	{ an expression returns an empty graph not}
	$\mid$	*bcond* && *bcond*	{ AND condition }
	$\mid$	*bcond* || *bcond*	{ OR condition }
	$\mid$	!*bcond*	{ NOT condition }
l	::=	$a \mid \$l$	{ label ($a \in String$) and label variables }

Figure 3.2: The syntax of structural recursion specifications for SWRP queries.

default, it applies on the input graph. Function composition is denoted by " $\circ$ ", and, from its definition, we have $(f_1 \circ f_2)\, x = f_2\, (f_1\, x)$. Structural recursive functions are defined in the form of pattern matching. The body of a function is an expression consisting of graph constructors, recursive function calls and conditional statements. We do not allow free graph variables inside an expression, instead we use a copy function, say $id(\$g)$, to obtain the value of the graph variable $\$g$. By using the *id* function, we can do optimizations, i.e. a tupling rule to obtain a structural recursive function from mutually structural recursive functions.

In this dissertation, specifications having no conditional statements are referred to as *specification without conditions*, and the ones having conditional statements are referred to as *specification with conditions*.

Example 3.6 *The following query finds all graphs following a path satisfying an expression _ * . c, and then transforms all edges labeled b in those graphs into edges*

labeled d,

$$
\begin{array}{l}
\textbf{\textit{select}} \\
\quad \textbf{letrec sfun} \\
\qquad b2d(\{b : \$g\}) = \{d : b2d(\$g)\} \\
\qquad b2d(\{\$l : \$g\}) = \{\$l : b2d(\$g)\} \\
\quad \textbf{in } \{c : b2d(\$r)\} \\
\textbf{\textit{where}} \ \{_ * .c : \$r\} \ \textbf{\textit{in}} \ \$db
\end{array}
$$

is translated to the following specification **c_b2d**.

$$
\begin{array}{l}
\textbf{main } f_1 \textbf{ where} \\
\quad f_1(\{c : \$g\}) = \{c : b2d(\$g)\} \cup f_1(\$g) \\
\quad f_1(\{\$l : \$g\}) = f_1(\$g) \\
\quad b2d(\{b : \$g\}) = \{d : b2d(\$g)\} \\
\quad b2d(\{\$l : \$g\}) = \{\$l : b2d(\$g)\}
\end{array}
$$

Example 3.7 *Nested queries, they are translated using a function composition to a query. As an example,*

$$
\begin{array}{l}
\textbf{\textit{select}} \ \$r \\
\textbf{\textit{where}} \ \{_ * .c : \$r\} \ \textbf{\textit{in}} \ (\textbf{\textit{select}} \ \$p \ \textbf{\textit{where}} \ \{_ * .d : \$p\} \ \textbf{\textit{in}} \ \$db)
\end{array}
$$

is translated to the following specification.

$$
\begin{array}{l}
\textbf{main } f_1 \circ f_2 \textbf{ where} \\
\quad f_1(\{d : \$g\}) = id(\$g) \cup f_1(\$g) \\
\quad f_1(\{\$l : \$g\}) = f_1(\$g) \\
\quad f_2(\{c : \$g\}) = id(\$g) \cup f_2(\$g) \\
\quad f_2(\{\$l : \$g\}) = f_2(\$g) \\
\quad id(\{\$l : \$g\}) = \{\$l : id(\$g)\}
\end{array}
$$

Example 3.8 *A query to find all papers which are published in 2010,*

$$\begin{array}{l} \textbf{\textit{select}}\ \$p \\ \textbf{\textit{where}}\ \{\texttt{Paper}: \$p\}\ \textbf{\textit{in}}\ \$db, \\ \quad\quad \texttt{Year.Int.2010}\ \textbf{\textit{in}}\ \$p \end{array}$$

is translated to the following specification.

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{main}\ f_1\ \textbf{where}} \\ f_1(\{\texttt{Paper}: \$g\}) & = & \textbf{if}\ !isempty(f_2(\$g)) \\ & & \textbf{then}\ id(\$g) \\ & & \textbf{else}\ \{\} \\ f_1(\{\$l: \$g\}) & = & \{\} \\ f_2(\{\texttt{Year}: \$g\}) & = & f_{21}(\$g) \\ f_2(\{\$l: \$g\}) & = & \{\} \\ f_{21}(\{\texttt{Int}: \$g\}) & = & f_{22}(\$g) \\ f_{21}(\{\$l: \$g\}) & = & \{\} \\ f_{22}(\{\texttt{2010}: \$g\}) & = & \{\texttt{2010}: \{\}\} \\ f_{22}(\{\$l: \$g\}) & = & \{\} \end{array}$$

Here, the result of the function f_1 depends on the result of the function f_2. If the function f_2 returns an empty graph, then the function f_1 returns an empty graph, otherwise, it calls the identity function id to obtain the final result.

3.3 On Obtaining Efficient Pregel Programs to Big Graphs

3.3.1 Specifications without Conditions

Parallelizable Structural Recursions

Given a function $e :: \mathcal{L}_\epsilon \to DB^{\mathcal{Z}}_{\mathcal{Z}}$, where $\mathcal{Z} = \{\&z_1, \ldots, \&z_n\}$. A function $h :: DB^{\mathcal{X}}_{\mathcal{Y}} \to DB^{\mathcal{X} \cdot \mathcal{Z}}_{\mathcal{Y} \cdot \mathcal{Z}}$ is called a parallelizable structural recursive function if the following equalities

for nine graph constructors hold [3, 34]:

$$h(\{\}) \equiv (\&z_1 := \{\}, \ldots, \&z_n := \{\}) \tag{3.1}$$
$$h(\{\$l : \$g\}) \equiv e(\$l) \mathbin{@} h(\$g) \tag{3.2}$$
$$h(\$g_1 \cup \$g_2) \equiv h(\$g_1) \cup h(\$g_2)$$
$$h(\&x := \$g) \equiv \&x \cdot h(\$g) \tag{3.3}$$
$$h(\&y) \equiv (\&z_1 := \&y \cdot \&z_1, \ldots, \&z_n := \&y \cdot \&z_n)$$
$$h() \equiv ()$$
$$h(\$g_1 \oplus \$g_2) \equiv h(\$g_1) \oplus h(\$g_2)$$
$$h(\$g_1 \mathbin{@} \$g_2) \equiv h(\$g_1) \mathbin{@} h(\$g_2)$$
$$h(\mathbf{cycle}(\$g)) \equiv \mathbf{cycle}(h(\$g))$$

In Eq. (3.3), $\&x \cdot (\&z_1 := \$g_1, \ldots, \&z_n := \$g_n)$ denotes $(\&x \cdot \&z_1 := \$g_1, \ldots, \&x \cdot \&z_n := \$g_n)$. For brevity, we denote the function h by $hom_{\mathcal{Z}}(e)$, and use Eq.(3.2) as its definition.

Structural recursive functions in a specification without conditions are $hom_{\mathcal{Z}}(e)$ functions whose function e is obtained by transforming pattern matchings into the construct **if** ... **then** ... **else** and substituting recursive calls by markers.

For example, the specification **a2d_xc** in Example 3.5 is equivalent to a $hom_{\{\&\}}(e)$, where

$$\begin{aligned} e(\$l) = \ & \mathbf{if}\ \$l = \mathrm{a}\ \mathbf{then}\ \{\mathrm{d} : \&\} \\ & \quad \mathbf{else\ if}\ \$l = \mathrm{c}\ \mathbf{then}\ \{\varepsilon : \&\}\ \mathbf{else}\ \{\$l : \&\} \end{aligned}$$

The specification **c_b2d** in Example 3.6 is equivalent to $\&f_1 \mathbin{@} hom_{\{\&f_1, \&b2d\}}(e)$, where the function e is obtained by tupling two mutually recursive functions f_1 and $b2d$ as follows.

$$\begin{aligned} e(\$l) = \ & (\&f_1 := \mathbf{if}\ \$l = \mathrm{c}\ \mathbf{then}\ (\{\mathrm{c} : \&b2d\} \cup \&f_1)\ \mathbf{else}\ \&f_1, \\ & \quad \&b2d := \mathbf{if}\ \$l = \mathrm{b}\ \mathbf{then}\ \{\mathrm{d} : \&b2d\}\ \mathbf{else}\ \{\$l : \&b2d\}) \end{aligned}$$

Lemma 3.1 (Fusion rule [3]) *Given two parallelizable structural recursions* $hom_{\mathcal{Z}_1}(e_1)$

and $hom_{\mathcal{Z}_2}(e_2)$, *the following equality holds.*

$$hom_{\mathcal{Z}_2}(e_2) \circ hom_{\mathcal{Z}_1}(e_1) = hom_{\mathcal{Z}_2}(hom_{\mathcal{Z}_2}(e_2) \circ e_1)$$

Theorem 3.2 *A specification without conditions can be transformed to an expression* $\&z_i \mathbin{@} hom_{\mathcal{Z}}(e)$, *where* $z_i \in \mathcal{Z}$.

Design of Pregel Programs

There are two ways to evaluate a structural recursion: recursive semantics and bulk semantics [3].

The idea of recursive semantics is to apply a $hom_{\mathcal{Z}}(e)$ function on each edge recursively from the root of an input graph. Memorization is used to avoid infinite loops in which results of each recursive call are stored at vertices. Whenever a vertex is visited, we will check if it is in a list of visited vertices we have seen so far. If it is not, we will create a new initial graph for it, make recursive calls and reflects results back to the graph. Otherwise, we look for a result from the list of visited vertices and return it. However, the disadvantage is that we have to do a heavy computation at each recursive iteration (applying $hom_{\mathcal{Z}}(e)$ on edge, creating new data, memorizing them and checking the termination condition, etc.), leading to a slow convergence. Furthermore, each iteration can only exploit parallelism between outgoing edges of a vertex, or between vertices at the same level of traversing.

On the other hand, bulk semantics is trying to delay computing the recursion by introducing ε-edges. General idea of bulk semantics is quite simple. First, it creates a *bulk graph* as follows. For each vertex v, $|\mathcal{Z}|$ disjoint copies of v are created, then the function e in the $hom_{\mathcal{Z}}(e)$ function is applied to every edge to create subgraphs with $|\mathcal{Z}|$ input markers and $|\mathcal{Z}|$ output markers. The bulk graph is created by connecting disjoint vertices and subgraphs via ε-edges. Finally, ε-edges are eliminated by computing their transitive closures. It is clear that bulk semantics is a parallel processing. The following equation captures the above computation.

$$hom_{\mathcal{Z}}(e) = eelim \circ bulk_{\mathcal{Z}}(e)$$

where the function $bulk_{\mathcal{Z}}(e)$ computes a bulk graph and *eelim* eliminates ε-edges.

Let us discuss in detail the size of the bulk graph generated during the bulk semantics. Assume that G is an input graph, and G_m is the bulk graph generated by the bulk semantics. It is clear that the number of vertices in G_m is $|\mathcal{Z}|$ times the number of vertices in G because for each vertex v in G, we create $|\mathcal{Z}|$ disjoint copies of v. For each edge (u, l, v) in G, the bulk semantics creates a subgraph with $|\mathcal{Z}|$ input vertices and $|\mathcal{Z}|$ output vertices, and ϵ-edges to/from input/output vertices. Therefore, assuming that the subgraph has no edges (if a user defines a function returning an empty graph), we need to create $2 \times |\mathcal{Z}|$ ϵ-edges in G_m.

Now, we consider the situation where we use the bulk semantics to evaluate our program, $\&z_i \mathbin{@} hom_{\mathcal{Z}}(e)$. Recall that the result of $hom_{\mathcal{Z}}(e)$ is a graph with $|\mathcal{Z}|$ input markers, and $\&z_i \in \mathcal{Z}$. Therefore, "$\&z_i \mathbin{@}$" is actually a reachability computation that returns a graph whose edges and vertices are reachable from the vertex v, where $v = I(\&z_i)$. Now, we have

$$\&z_i \mathbin{@} hom_{\mathcal{Z}}(e) = reach_{\{\&z_i\}} \circ eelim \circ bulk_{\mathcal{Z}}(e) \tag{3.4}$$

where the function $reach_{\{\&z_i\}}$ denotes the reachability computation for $\&z_i \mathbin{@}$.

Figure 3.3 shows an example of using bulk semantics to evaluate the expression $\&f_1 \mathbin{@} hom_{\{\&f_1, \&b2d\}}(e)$ for the specification **c_b2d**, where,

$$\begin{aligned} e(\$l) = \ (&\&f_1 := \textbf{if } \$l = \mathrm{c} \textbf{ then } (\{\mathrm{c} : \&b2d\} \cup \&f_1) \textbf{ else } \&f_1, \\ &\&b2d := \textbf{if } \$l = \mathrm{b} \textbf{ then } \{\mathrm{d} : \&b2d\} \textbf{ else } \{\$l : \&b2d\}) \end{aligned}$$

As for parallelizable structural recursion, since the function e only depends on edge labels, we can obtain a fully parallel computation. However, the function $bulk_{\mathcal{Z}}(e)$ generates a big intermediate graph whose size is at least $|\mathcal{Z}|$ times larger than the input graph, which causes a serious problem of memory when processing big graphs.

We now present how to derive an efficient algorithm in terms of intermediate data generated during its computation. To minimize the amount of ε-edges as well as redundant edges produced by $bulk_{\mathcal{Z}}(e)$, we propose a hybrid approach of recursive semantics and bulk semantics. This kind of optimization was informally mentioned in [3] via a practical example.

Our idea is to promote the function $reach_{\{\&z_i\}}$ to prune redundant data generated

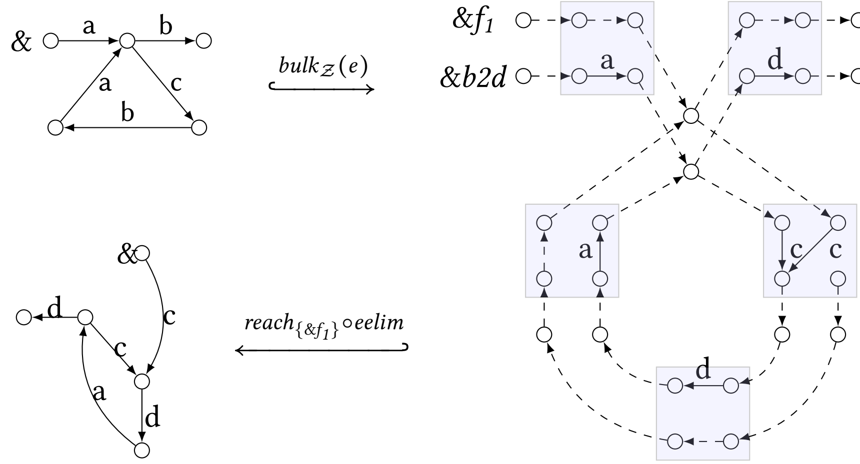


Figure 3.3: Bulk semantics to evaluate the specification **c_b2d**.

by the function $bulk_{\mathcal{Z}}(e)$. A recursive semantics is used to compute a *marker graph* whose each vertex u consists of a set of *reachable markers* $\mathcal{X}_u \subseteq \mathcal{Z}$. After that, in a bulk semantics, for each vertex u, we create exactly $|\mathcal{X}_u|$ disjoint vertices, and then the function e, instead of computing a graph of $|\mathcal{Z}|$ disjoint subgraphs, computes a graph of only $|\mathcal{X}_u|$ disjoint subgraphs. It is important to note that the intermediate graphs generated are very close to the final result.

In order to express both semantics, we extract two functions $e_{\rightarrow}$ and e_{π} from the function e of $hom_{\mathcal{Z}}(e)$

$$
\begin{aligned}
e_{\rightarrow} &:: (\mathcal{M}_t \times \mathcal{L}_{\epsilon}) \rightarrow \mathcal{M} \\
e_{\rightarrow}(\&z, \$l) &= \textbf{let } (vs, es, is, os) = \&z \,@\, e(\$l) \textbf{ in } map\, snd\, os \\
e_{\pi} &:: (\mathcal{M}_t \times \mathcal{L}_{\epsilon}) \rightarrow DB^{\mathcal{M}}_{\mathcal{Z}} \\
e_{\pi}(\&z, \$l) &= \&z \,@\, e(\$l)
\end{aligned}
$$

where, $\mathcal{M}_t$ is a type for markers; function $e_{\rightarrow}$ is like a transition function in automaton, where, for each label $\$l$ and input marker $\&z$ we compute a set of (output) markers reachable from $\&z$ in the graph generated by function e; function e_{π} is simply a

projection function. Note that these two functions are statically derived from a given function e.

Our program is now evaluated as follows.

$$\&z_i \,@\, hom_{\mathcal{Z}}(e) = eelim \circ bulk'_{\mathcal{Z}}(e_\pi) \circ mark_{\{\&z_i\}}(e_\rightarrow) \tag{3.5}$$

The function $mark_{\{\&z_i\}}(e_\rightarrow)$ computes a marker graph using recursive semantics. It starts from the root of an input graph with the input marker $\&z_i$, and recursively uses $e_\rightarrow$ to find all markers a vertex can have. The vertices having no markers will be removed after that. The function $bulk'_{\mathcal{Z}}(e_\pi)$ is similar to $bulk_{\mathcal{Z}}(e)$ but e_π is applied with respect to markers $\&z$ of source vertices. In a Pregel program, we refer to these three functions as three Pregel phases: *Mark*, *Bulk*, and *Eelim*.

Proof. (sketch) Now we prove the equation 3.5. We will show that, to evaluate "$\&z_i \,@\, hom_{\mathcal{Z}}(e)$", using Eq. 3.5 will produce the same graph as using Eq. 3.4 (the bulk semantics).

We first rewrite Eq. 3.4 by swapping two functions $eelim$ and $reach_{\{\&z_1\}}$. Eq. 3.4 becomes

$$\&z_i \,@\, hom_{\mathcal{Z}}(e) = eelim \circ reach_{\{\&z_i\}} \circ bulk_{\mathcal{Z}}(e)$$

This is because the function $eelim$ never makes an unreachable vertex from a root become a reachable vertex (according to the meaning of ϵ-edges in Section 2.1.2). It is worth noting that the result of "$eelim \circ reach_{\{\&z_i\}}$" may contain additional vertices that are unreachable from the root, while the result of "$reach_{\{\&z_i\}} \circ eelim$" does not. However, the two results are still equivalent up to bisimulation.

Assume that G_1 is a result graph of "$reach_{\{\&z_i\}} \circ bulk_{\mathcal{Z}}(e)$", and G_2 is a result graph of "$bulk'_{\mathcal{Z}}(e_\pi) \circ mark_{\{\&z_i\}}(e_\rightarrow)$". To prove Eq. 3.5, we need to show that $G_1 \equiv G_2$. Actually, G_1 and G_2 are isomorphic, and graph isomorphism implies value equivalence. Let us denote disjoint vertices created by $bulk_{\mathcal{Z}}(e)$ by S_1 vertices, input/output vertices of subgraphs by S_2 vertices. For each vertex in an input graph G, $bulk_{\mathcal{Z}}(e)$ creates $|\mathcal{Z}|$ vertices S_1 in which each vertex corresponds to one markers in $\mathcal{Z}$. After that, the function $reach_{\{\&z_i\}}$ keeps only disjoint vertices that reachable from input vertices with the markers $\&z_i$. These reachable disjoint vertices are exactly computed by the function $mark_{\&z_i}(e_\rightarrow)$. Similar to S_2 vertices, the function $bulk'_{\mathcal{Z}}(e_\pi)$ produces only vertices and edges that are reachable from input vertices with markers $\&z_i$. Hence, G_1

and G_2 are the same graph. □

We consider the example **c_b2d** to see in detail how to evaluate it with our approach. From its function e,

$$\begin{aligned} e(\$l) = \ & (\&f_1 := \textbf{if } \$l = \mathrm{c} \textbf{ then } (\{\mathrm{c} : \&b2d\} \cup \&f_1) \textbf{ else } \&f_1, \\ & \&b2d := \textbf{if } \$l = \mathrm{b} \textbf{ then } \{\mathrm{d} : \&b2d\} \textbf{ else } \{\$l : \&b2d\}) \end{aligned}$$

we have two functions $e_{\rightarrow}$ and e_{π} as follows.

$$\begin{aligned} e_{\rightarrow}(\&f, \$l): \ & \textbf{if } \&f = \&f_1 \textbf{ then} \\ & \quad \textbf{if } \$l = \mathrm{c} \textbf{ then } \{\&b2d, \&f_1\} \textbf{ else } \{\&f_1\} \\ & \textbf{else if } \&f = \&b2d \textbf{ then} \\ & \quad \textbf{if } \$l = \mathrm{b} \textbf{ then } \{\&b2d\} \textbf{ else } \{\&b2d\} \\ & \textbf{else } \{\} \end{aligned}$$ [2]

$$\begin{aligned} e_{\pi}(\&f, \$l): \ & \textbf{if } \&f = \&f_1 \textbf{ then} \\ & \quad \textbf{if } \$l = \mathrm{c} \textbf{ then } \&f_1 := \{\mathrm{c} : \&b2d\} \cup \&f_1 \\ & \qquad\qquad \textbf{else } \&f_1 := \&f_1 \\ & \textbf{else if } \&f = \&b2d \textbf{ then} \\ & \quad \textbf{if } \$l = \mathrm{b} \textbf{ then } \&b2d := \{\mathrm{d} : \&b2d\} \\ & \qquad\qquad \textbf{else } \&b2d := \{\$l : \&b2d\} \\ & \textbf{else } \{\} \end{aligned}$$

Figure 3.4 shows intermediate graphs generated during the evaluation. In the phase *Mark*, a marker graph is created by the function $mark_{\{\&z_i\}}(e_{\rightarrow})$. The marker graph is computed as follows. First, the root vertex is initialized with a singleton set $\{\&f_1\}$, where f_1 is the function we want to evaluate. We evaluate the first edge $u \xrightarrow{\mathrm{a}} v$ from the root. Its result, $e_{\rightarrow}(\&f_1, \mathrm{a}) = \{\&f_1\}$, is written to the vertex v. Next, we concurrently evaluate two edges $v \xrightarrow{\mathrm{b}} w_1$ and $v \xrightarrow{\mathrm{c}} w_2$ emanating from v, and results are written to respective targets w_1, w_2. This procedure is iterated and then terminated when it can not find any new markers to add to vertices. In the phase *Bulk*, a bulk graph is then computed by the function $bulk'_{\mathcal{Z}}(e_{\pi})$ as follows. For each vertex u, and

[2] Note that $\{\}$ in the function $e_{\rightarrow}$ denotes a set instead of a graph constructor

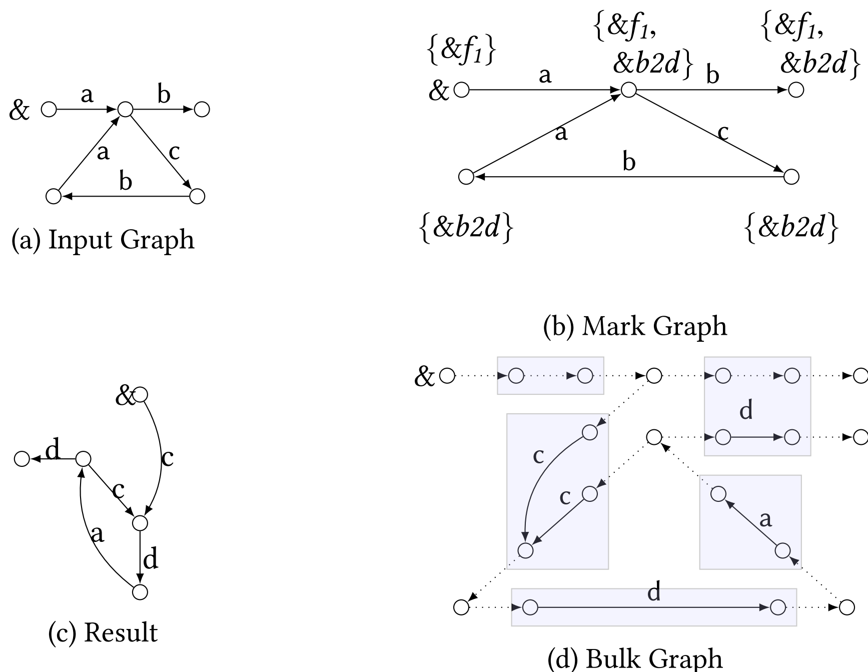


Figure 3.4: Graphs generated during the evaluation of the specification *c_b2d*.

its set of markers $\mathcal{X}_u$, we create $|\mathcal{X}_u|$ disjoint vertices. Next, we apply the function e_π on each edge $u \xrightarrow{L} v$ and each marker in $\mathcal{X}_u$, producing a subgraph of $|\mathcal{X}_u|$ input markers. In Figure 3.4, these subgraphs are surrounded by a shaded rectangle. After that, we use ϵ-edges to connect disjoint vertices and subgraphs. Finally, in the phase *Eelim*, the function *eelim* eliminates all ϵ-edges in the bulk graph to produce the final result.

Pregel Algorithm to Eliminate ϵ-Edges

In the remaining part of this section, we show a Pregel algorithm to eliminate ϵ-edges in a graph by using transitive closure of ϵ-edges. Theoretically, an ϵ-edge from a vertex v to v' means that all edges emanating from v' should be emanating from v [3]. Eliminating an ϵ-edge $v \xrightarrow{\epsilon} v'$ means removing this ϵ-edge and for each edge emanating from v', $v' \xrightarrow{a} w$, adding a new edge $v \xrightarrow{a} w$.

Our proposal is based on the algorithm of eliminating ϵ-transitions in a finite automata proposed by Hopcroft, Motwani and Ullman [24]. The algorithm consists of the following four steps: (1) compute the transitive closure of the ϵ-arcs only. As a

result, each state q has an ϵ-closure ECLOSE(q) which is a set of states, where if a state $p \in \text{ECLOSE}(q)$ then there exists a path whose arcs are all labeled ϵ from q to p; (2) for each $p \in \text{ECLOSE}(q)$, if there is a transition from p to r on input a (not ϵ), then add a transition from q to r on input a. (3) make state p an accepting state if p can reach some accepting state q by ϵ-arcs. (4) remove all ϵ-transitions.

However, the above algorithm is not suitable for the Pregel model. It is because it stores an ϵ-closure ECLOSE(q) at the vertex q, making the step 2 difficult to be implemented. In particular, in the step 2, assume that we have two transitions: $q \xrightarrow{\epsilon} p$, $p \xrightarrow{a} r$, we need to add a transition from q to r on input a, say, $q \xrightarrow{a} r$. In the current algorithm, the state q only knows the state p, but it does not know outgoing transitions from p, say $p \xrightarrow{a} r$. Hence, the state q cannot perform the addition operation. The addition operation must be performed by the state p. Unfortunately, the state p has no information of incoming states like q.

In order to solve the problem, we need to change the way we store ϵ-closure by which each state p will have a set of states $\overline{\text{ECLOSE}}(p)$ so that if $q \in \overline{\text{ECLOSE}}(p)$, then there exists a path whose arcs are all labeled ϵ from q to p. Our Pregel algorithm to eliminate ϵ-edges consists of the following three phases: (1) compute the transitive closure of the ϵ-arcs only. As a result, each state p has an ϵ-closure $\overline{\text{ECLOSE}}(p)$; (2) for each $q \in \overline{\text{ECLOSE}}(p)$, if there is a transition from p to r on input a (not ϵ), then add a transition from q to r on input a. (3) remove all ϵ-transitions.

After computing the transitive closure of ϵ-edges, there are many parts unreachable from the roots of a graph. These unreachable parts are removed by using the following proposition.

Definition 3.3 (Safe Vertex) *Given a graph G. A vertex v_i is called a safe vertex if all of the incoming edges to v_i are ϵ-edges.*

Proposition 3.4 *Given a graph G whose vertices and edges are reachable from the root vertex v_r. Let G_1 be a graph obtained by eliminating all ϵ-edges in G. Let G_2 be a graph obtained by removing all of the safe vertices of G from G_1, then a) G_2 and G_1 are equivalent; b) G_2 contains only vertices and edges which are reachable from the root v_r.*

Proof. (Proof by contradiction) Assume to the contrary that there exists one edge $(u, 1, v) \in E(G_2)$, such that u is unreachable from the root v_r. Because G_2 is obtained

from G_1 by removing all safe vertices, u cannot be a safe vertex. On the other hand, since G_1 is obtained from G by eliminating all ϵ-edges, this elimination makes u unreachable from the root v_r. In other words, all edges coming to u are removed, and they must be ϵ-edges. Hence, u is a safe vertex (contradicting our conclusion that u cannot be a safe vertex). □

The proposition says that after eliminating ϵ-edges from a graph G, we can compute the reachable part of a graph G_1 by just removing all of its safe vertices instead of doing a reachability computation from the root of G_1. It takes two Pregel supersteps to find safe vertices. Moreover, it is worth noting that removing safe vertices is simply done by a single superstep since vertices can mutate their outgoing edges directly.

3.3.2 Specifications with Conditions

We now turn to show how to evaluate specifications with conditions in (Figure 3.2). The difficulty in evaluating such specifications is relating to the computation for each edge. Recall that each declaration $f(\{l : \$g\})$ describes a computation for an edge labeled l. Once there exists a **if/then/else**, the function f certainly depends on the graph $\$g$, which is difficult to be implemented in Pregel, because each vertex only knows its outgoing edges instead of the whole graph $\$g$. Our idea is firstly evaluating all branches **if**, **then**, **else** at the same time by a specification without conditions, then using an iterative Pregel algorithm to check conditions in branches **if**, and finally using another specification without conditions to extract final results from branches **then** or **else**.

We sketch our idea via an evaluation of the query in Example 3.8.

The first specification without conditions is achieved by flattening **if/then/else** statements and representing them by graphs whose edge labels are keywords, e.g. `_if`, `_then`, `_else`, `_isempty`, etc. These edges are called *keyword edges*. Here, we use a special prefix "_" to distinguish keywords from users' data in a graph. One could view these graphs as Abstract Syntax Trees of **if/then/else** statements. The first

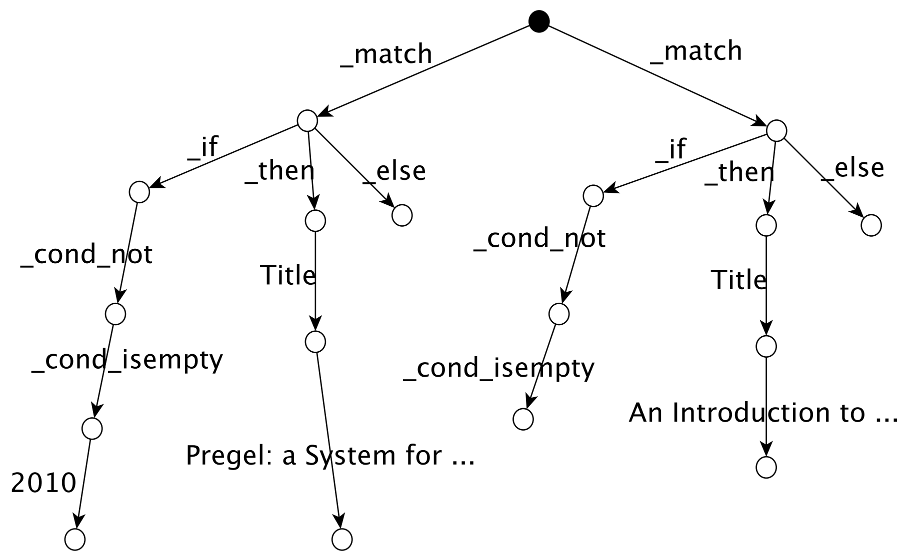


Figure 3.5: The result of the 1st specification without conditions.

specification without conditions for the query in Example 3.8 is the following:

$$
\begin{array}{l}
\textbf{main } f_1 \textbf{ where} \\
\quad f_1(\{\texttt{Paper}:\$g\}) = \{\texttt{_match}:(\\
\qquad\qquad \{\texttt{_if}: \\
\qquad\qquad\quad \{\texttt{_cond_not}: \\
\qquad\qquad\qquad \{\texttt{_isempty}:f_2(\$g)\}\}\} \\
\qquad\qquad \cup\{\texttt{_then}:f_3(\$g)\} \\
\qquad\qquad \cup\{\texttt{_else}:\{\}\})\} \\
\quad f_1(\{\$l:\$g\}) \qquad = \{\} \\
\quad\quad \ldots
\end{array}
$$

It is also worth noting that, for each **if/then/else** statement, we introduce an edge `_match` appending to the root of the **if/then/else** graph. These `_match` edges are important to derive another specification without conditions to extract the final result. Figure 3.5 shows an example of the result of the first specification without conditions.

The iterative Pregel algorithm, named *evalIfThenElse*, evaluates branches **if** in order to update edges `_match`. Basically, it starts from edges `_cond_isempty`, and checks the graphs followed by those edges are empty or not, then propagates results

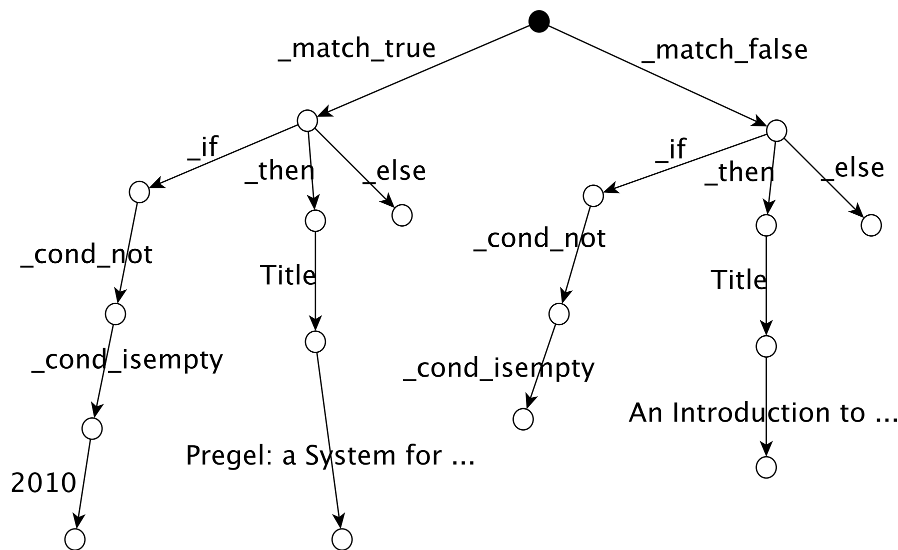


Figure 3.6: The result of *evalIfThenElse* algorithm.

back to the `_match` edges. Afterwards, `_match` will be updated to `_match_true` or `_match_false`. Figure 3.6 shows an example result when applying this algorithm to the graph in Figure 3.5.

Finally, we use another specification without conditions to extract the final result (Figure 3.7) from the graph in Figure 3.6. This specification finds all edges `_match_true` and `_match_false`, then extracts graphs followed by edges `_then` (or `_else`) once it meets edges `_match_true` (or `_match_false`). This specification is independent of input queries/specifications, and it is always written as follows.

$$
\begin{aligned}
&\textbf{main } f_1 \textbf{ where}\\
&\quad f_1(\{\texttt{_match_true}:\$g\}) = f_{then}(\$g)\\
&\quad f_1(\{\texttt{_match_false}:\$g\}) = f_{else}(\$g)\\
&\quad f_1(\{\$l:\$g\}) = \{\$l: f_1(\$g)\}\\
&\quad f_{then}(\{\texttt{_then}:\$g\}) = f_1(\$g)\\
&\quad f_{then}(\{\$l:\$g\}) = \{\}\\
&\quad f_{else}(\{\texttt{_else}:\$g\}) = f_1(\$g)\\
&\quad f_{else}(\{\$l:\$g\}) = \{\}
\end{aligned}
$$

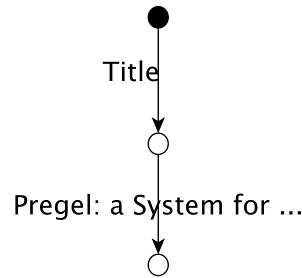


Figure 3.7: The result of the 2nd specification without conditions.

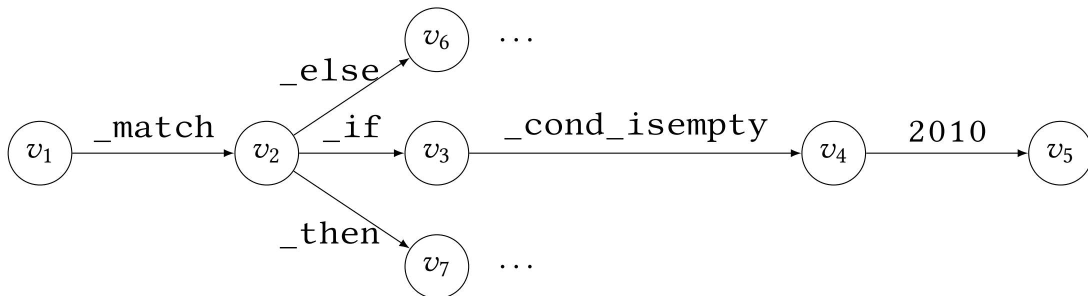


Figure 3.8: A graph encoding a statement "**if** *isempty*(f(\$g)) **then** ...".

We now explain the *evalIfThenElse* algorithm in detail. Let us firstly consider the case where there is no nested **if/then/else** statement *inside the branch* **if**, and then generalize it to arbitrary specifications. Assume that a statement "**if** *isempty*(f(\$g)) **then** ...", where f is a specification without conditions (not containing any **if/then/else** statement), is encoded by the graph in Figure 3.8, where the subgraph with the root v_4 is the result of the function f. Each vertex maintains a set of boolean values. The computation starts from vertices like v_4. If v_4 has some outgoing edges (the result of f is a non-empty graph), then we initialize the value of v_4 to a set $\{False\}$; Otherwise, a set $\{True\}$. It takes one superstep to mark vertices like v_4, and one superstep to initialize their values. Next, we do an iterative procedure to propagate these results back to the `_match` edge via message sendings. This procedure requires to reverse the graph. In a superstep, a message is a set of boolean values. On receiving messages, a vertex v simply takes the union of incoming messages and updates to its value. On sending a message to u along an edge $u \stackrel{L}{\leftarrow} v$, the message is constructed based on L and the vertex value $\mathcal{N}$ of v as follows.

- If L = _cond_isempty|_if|_match, the message is $\mathcal{N}$;
- If L = _cond_and|_cond_or, the message is *red*, where *red* is the reduction of elements in $\mathcal{N}$ with the logical operator AND or OR;
- If L = _cond_not, the message is $\bar{\mathcal{N}}$, where $\bar{\mathcal{N}}$ is a set of negative elements of $\mathcal{N}$;
- Otherwise, send nothing.

This procedure ends when there is no message in transit. We then reverse the graph back. Finally, for each edge $u \xrightarrow{\texttt{_match}} v$, the vertex u has the result of the branch **if** which is a singleton set, we change the label _match to either _match_true or _match_false, depending on whether the element in the set is *True* or *False*. One can do this update during the previous supersteps. However, we use another superstep, which makes our algorithm easy to generalize.

Now we consider nested **if/then/else** statements. Assume that we want to evaluate a statement "**if** *isempty*$(f(\$g))$ **then** ...", where f is also defined by a **if/then/else** statement, say, "**if** *isempty*$(f_1(\$g_1))$ **then** ...". In this case, the result of the outer branch **if** depends on the result of the whole inner **if/then/else** statement. In particular, in order to know the result of $f(\$g)$ is an empty graph or not, we need to know not only the result of *isempty*$(f_1(\$g_1))$ being *True* or *False*, but also the results of the branches **then** and **else** being an empty graph or not. Therefore, for each vertex, we maintain a triple of sets of boolean values $(bIf, bThen, bElse)$ to store the results of branches **if**, **then**, and **else**. We make some modifications as follows. For initial supersteps, we initialize a value for the vertices v of edges $u \xrightarrow{L} v$, where L is not only _cond_isempty, but also _then and _else. We do the same iterative procedure to the reverse graph as before, but messages are now a triple of sets of boolean values, and they are sent along edges _then and _else also. On receiving messages, a vertex v computes a triple $(bIf, bThen, bElse)$ by taking the union of corresponding elements in messages, e.g. bIf is the union of first sets in the incoming messages. The triple is then set to the value of v. On sending a message to u along an edge $u \xleftarrow{L} v$, the message is constructed based on L and the vertex value $(bIf, bThen, bElse)$ of v as follows.

- If L = _cond_isempty, the message is $(bIf, bThen, bElse)$;

- If L = `_if`, the message is $(bIf, \{\}, \{\})$;
- If L = `_then`, the message is $(\{\}, bThen, \{\})$;
- If L = `_else`, the message is $(\{\}, \{\}, bElse)$;
- If L = `_match`, the message is $(bIf, bThen, \{\})$ if the singleton set bIf has an element *True*; otherwise, $(bIf, \{\}, bElse)$;
- If L = `_cond_and`|`_cond_or`, the message is $(bIfred, \{\}, \{\})$, where $bIfred$ is the reduction of elements in bIf with the logical operator AND or OR;
- If L = `_cond_not`, the message is $(\overline{bIf}, \{\}, \{\})$, where $\overline{bIf}$ is a set of negative elements of bIf;
- Otherwise, send nothing.

This procedure ends when there is no message in transit. We then reverse the graph back. Finally, for each edge $u \xrightarrow{\texttt{_match}} v$, the label is updated based on the singleton set bIf of the vertex u, it becomes `_match_true` (`_match_false`) if the element in the set is *True* (*False*). This procedure can deal with nested **if/then/else** statements inside not only **if** but also **then**, or **else**.

3.4 Implementation and Experiments

3.4.1 Implementation

Thanks to structural recursion, we can propose a light-weight but powerful framework to query big graphs. There are many open source implementations of Pregel such as Giraph [35], GPS [36], Pregel+ [37], GraphX [38], Mizan [39]. However, we choose GraphX—a library written in Spark—to implement our framework due to two main reasons. First, GraphX supports a vertex-cut approach to distributed graph partitioning, which can reduce both communication and storage overheads. It is especially useful for our model where the root has a high out-degree. Second, in our algorithms, each Pregel phase may require a different data type for vertex values and messages. Since GraphX is a functional programming library, it is very convenient to work with types.

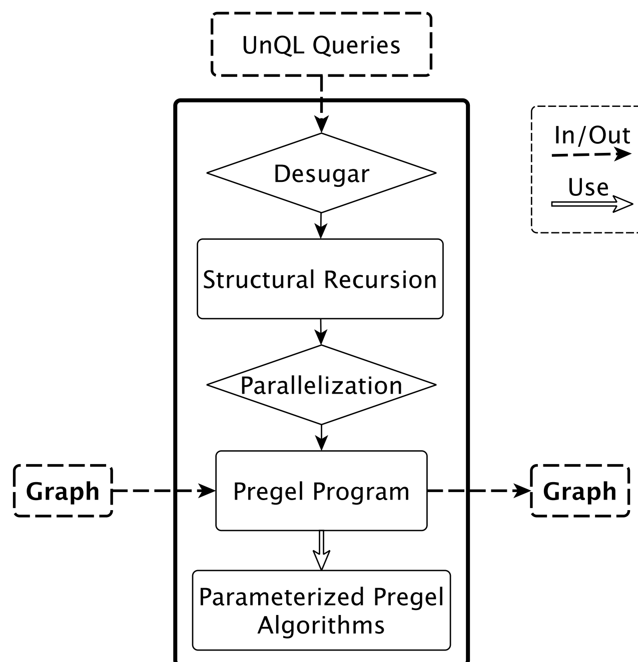


Figure 3.9: Overview of our framework.

In our experiments, we set up GraphX running with its default partitioning strategy that uses the initial partitioning of the edges. An input graph is stored in the form of a list of edges, using a single file. The file is partitioned by blocks, depending on the number of processors we run for experiments.

Figure 3.9 shows the major components of our framework. Details are described in the following.

- The **Desugar** accepts an input graph query, parses it and generates a specification in the form of structural recursion (Figure 3.2).

- The **Parallelization** generates Pregel programs from specifications written in the form of structural recursion. Depending on different specifications, it will generate different running strategies. After that, it compiles such strategies into a Pregel program that utilizes parameterized Pregel algorithms. Optimization rules such as Fusion, Tupling are implemented in this component.

- The **Parameterized Pregel Algorithms** consists of efficiently-made functions

that are used to implement strategies generated by the Parallelization. These functions are Pregel algorithms including *mark*, *bulk*, *eelim*, and *evalIfThenElse*.

- The **Pregel Program** is a main program that is generated by the Parallelization component. Users compile and run this program with an input distributed graph and the result is a new distributed graph.

The whole framework is about 2000 lines of code and available at `http://www.prg.nii.ac.jp/members/tungld/bigra.html` [3]

We have implemented some technical optimizations as follows.

Graph Mutations As for the algorithm to eliminate ϵ-edges we have proposed, we need to add new edges and remove safe vertices. However, GraphX originally does not support graph topology mutations. Hence, mutations are manually performed by creating a new graph. In a naive way, we mutate the vertex set of a graph (it is a RDD record in Spark), mutate the edge set, and create a new graph from these two sets. This approach is slow because we need to shuffle the whole data to create a distributed graph. Our solution is to mutate only the edge set, and use a filter to remove vertices. In particular, we do as follows. Firstly, we mutate the edge set. Then create a new graph with the new edge set and the *old* vertex set. Finally, we filter out vertices using an efficient API, *filter*, provided by GraphX. This optimization has significantly reduced the running time of the phase *Eelim*. Our approach is about 10 times faster than the naive way.

Local ϵ-edge elimination Because eliminating ϵ-edges in a distributed graph is very expensive, it is necessary to eliminate ϵ-edges as much as possible before creating a distributed graph. Hence, we perform a local ϵ-edge elimination for each subgraphs during the phase *Bulk*, leading to a distributed graph with a small amount of ϵ-edges. This makes the phase *Eelim* faster.

Tuning data structures In order to reduce the memory consumption in our framework when using Scala collection classes (e.g. HashMap, Set), we used collection classes from the library *fastutil*—a fastest implementation for many collection classes

[3]The current version is written in Spark version 1.4.0 (released Jun 11, 2015)

Table 3.1: Real-life graphs.

Graphs	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$
Citation	7.5M	8.8M
Youtube	13.5M	17M
Amazon Product	90M	103.6M
AltaVista Web Page	54M	620.8M

that are compatible with the Java standard library [40]. We tuned data structures used in the phase *Bulk*, making the phase two times faster than the one used Scala collective classes. We also tried to tune data structures for the other phases, but we could not gain a better performance.

3.4.2 Experiments

Datasets

We used four real-life datasets in our experiments, (1) Citation [41] that contains papers (Title, Authors, Conference, Year, References) and their citation relationships; (2) Youtube [42] that contains videos (Uploader, Category, Length, Related IDs, etc.) and their “related” relationships; (3) Yahoo! AltaVista Web Page Hyperlink Connectivity Graph provided by Yahoo! Labs Webscope [43] that contains URLs and hyperlinks for over 1.4 billion public web pages indexed by the Yahoo! AltaVista search engine in 2002; (4) Amazon Product Co-purchasing Network [44] that contains product metadata and review information of about $548,552$ different products from Amazon website.

These datasets need to be converted to rooted edge-labeled graphs. Table 3.1 summarized the sizes of the biggest graphs for each dataset that we used in our experiments. We also convert the Amazon Product dataset into tables in a relational database, which is used for comparing the performance of our framework with Spark SQL [45]—a relational data processing framework in Spark.

On Comparison with the Pure Bulk Evaluation

The environment of our experiments is the following: Intel Xeon CPU E5620@2.40GHz 16 cores, 48GB memory, GraphX in Spark 1.3.0, Hadoop 2.4.2. We set up GraphX working in a distributed mode.

We did experiments with queries whose specifications consist of four mutually recursive functions. First, we see how our framework performs when increasing the size of input data, while fixing the number of CPUs at 16. Figures 3.10 and 3.12 show the results for the citation and youtube graphs, respectively. It is clear that, for the citation graph, our framework outperforms the pure bulk semantics though the pure bulk semantics seems linear to the size of graph. However, for the youtube graph, the pure bulk semantics can not deal with a graph with 11 million edges. It can generate a bulk graph, but can not finish the ε-edge elimination due to an "out-of-memory" error. By contrast, our framework works smoothly even for the graph of 17 million edges.

Next, we changed the number of CPUs and fixed the graph size at 5 million edges for the citation graph (Figure 3.11) and at 17 million edges for the youtube graph (Figure 3.13). Both the pure bulk semantics and our framework follow the same shape. When we double the number of CPUs from 1 to 2, or 2 to 4, both achieve a speedup of 2. But, after that, for 8 and 16 CPUs, we do not have the same performance. This is because when we increase the number of CPUs, we will have more partitions. The local computation time for each partition is decreased but the communication time is increased. Also from these experiments, we can see that our framework is about 2-3 times faster than the pure bulk semantics. This is reasonable for the specifications with four mutually recursive functions, because the pure bulk semantics generates a bulk graph of about 4 times larger than the input, which makes its computation time slower than our framework where there is no duplication data generated.

Scalability

To see the scalability of our framework, we did experiments on Amazon EC2 machines. We configured 16 instances of type r3.2xlarge [46]. Each instance has 8 processors (Intel Xeon E5-2670) and 61 GB of RAM. This type of instance is optimized for memory-intensive applications.

We used two queries to evaluate the scalability of our framework with the graphs of Yahoo! AltaVista Web Page Hyperlink Connectivity Dataset (see Table 3.2). The first

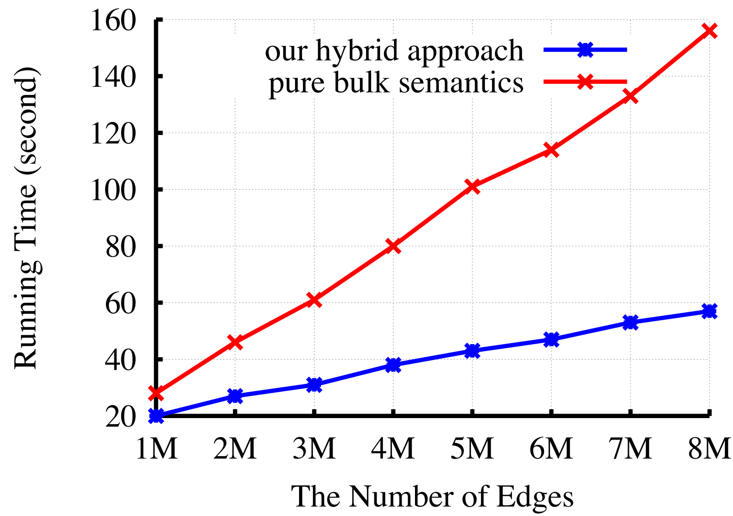


Figure 3.10: Varying graph size (Citation dataset).

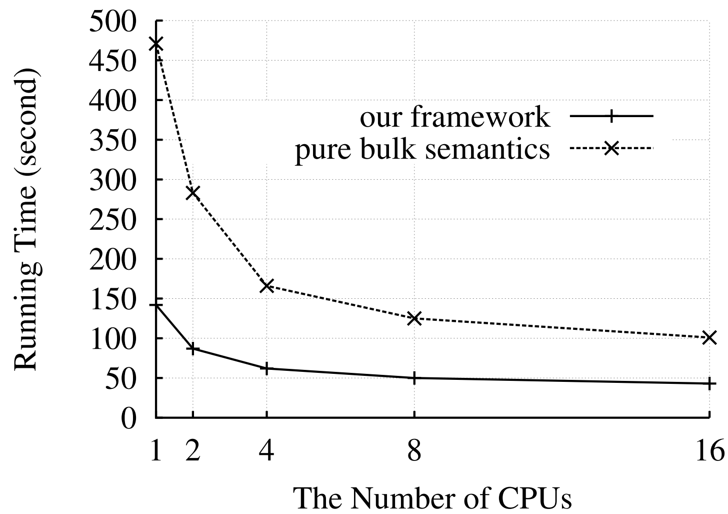


Figure 3.11: Varying the number of CPUs (Citation dataset).

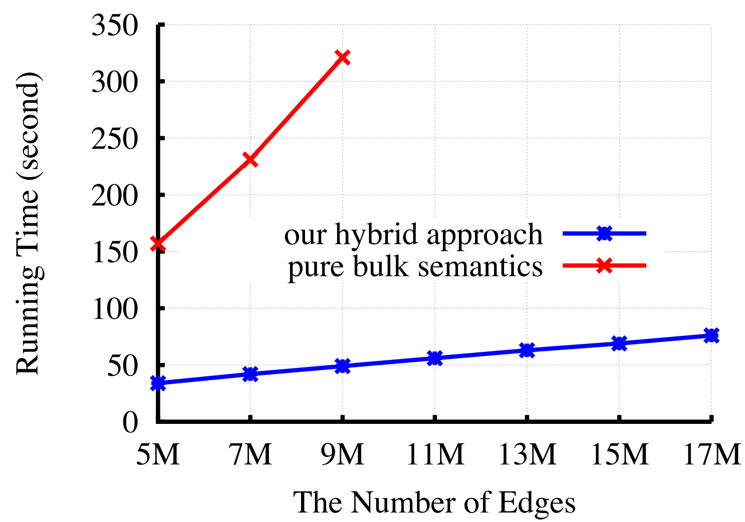


Figure 3.12: Varying graph size (Youtube dataset).

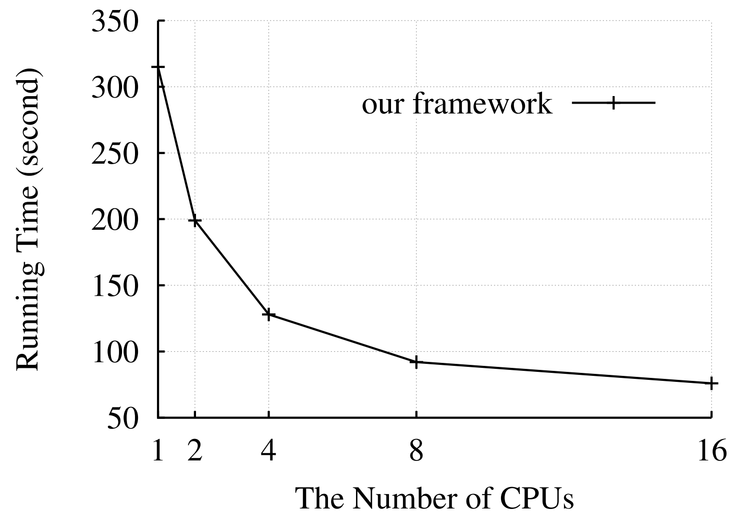


Figure 3.13: Varying the number of CPUs (Youtube dataset).

Table 3.2: Experiment graphs extracted from Yahoo! AltaVista Web Page Hyperlink Connectivity Graph.

Graph Name	G50	G100	G150	G200	G300	G400	G500	G600
# of Web Pages	191K	444K	687K	908K	1.5M	2M	2.4M	3M
# of Vertices	4.8M	10.8M	15.7M	19.3M	29M	38M	45M	54M
# of Edges	51.5M	103M	155M	206M	310M	413.8M	517M	620.8M

one is a regular path query that returns domains `com` and `net` of a URL:

$$
\begin{aligned}
Q1:\ &\textbf{select}\\
&\quad ((\textbf{select}\ \{\texttt{com} : \$c\}\\
&\quad\ \ \textbf{where}\ \{\texttt{com} : \$c\}\ \textbf{in}\ \$d)\\
&\quad \cup\\
&\quad (\textbf{select}\ \{\texttt{net} : \$n\}\\
&\quad\ \ \textbf{where}\ \{\texttt{net} : \$n\}\ \textbf{in}\ \$d))\\
&\textbf{where}\ \{\texttt{page.domain} : \$d\}\ \textbf{in}\ \$db
\end{aligned}
$$

The second one is a query with conditions, which retrieves all URLs hyperlinked by a web page whose top-level domain is `org`:

$$
\begin{aligned}
Q2:\ &\textbf{select}\\
&\quad (\textbf{select}\ \{\texttt{webpage} : \$u\}\\
&\quad\ \ \textbf{where}\ \{\texttt{links.page.url} : \$u\}\ \textbf{in}\ \$p)\\
&\textbf{where}\\
&\quad \{\texttt{page} : \$p\}\ \textbf{in}\ \$db,\\
&\quad \texttt{domain.org}\ \textbf{in}\ \$p
\end{aligned}
$$

For query Q1, we vary the size of input graph from about 100 million edges to 600 million edges by increasing each time about 100 million edges (corresponding to G100 to G600 in Table 3.2). For each graphs, we ran with 64, 96 and 128 processors, respectively. Figure 3.14(a) shows the execution times of Q1. It is clear that our framework has a very good scalability. When the size of the input graph increases, the execution time increases slightly. On small graphs, say G100, the query Q1 with 64 processors consumes the same execution time as with 96 or 128 processors, it is because

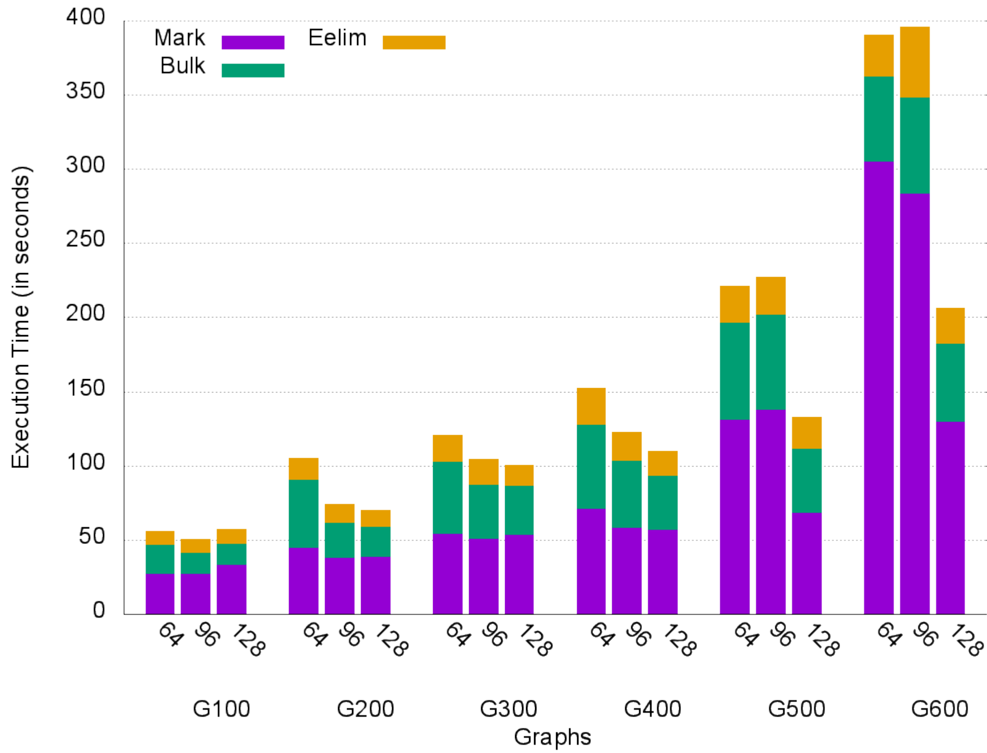
allocating more processors increases the cost of communication and synchronization between processors. On bigger graphs, we gain a good speedup when increasing the number of processors. In particular, when doubling the number of processors from 64 to 128, we gain the speedups of 1.5, 1.2, 1.3, 1.6, 1.8 on G200, G300, G400, G500 and G600, respectively. These speedups are quite good compared to Pregel-based frameworks where it is non-trivial to gain good speedup [47, 48]. However, results in Figure 3.14(a) show that our framework does not behave well when the number of processors is not the power of 2, say 96.

The execution of query Q2 is shown in Figure 3.15(a). This time we only use the numbers of processors that are the power of 2. Although evaluating Q2 is much more expensive than evaluating Q1, we still have a good scalability and speedup.

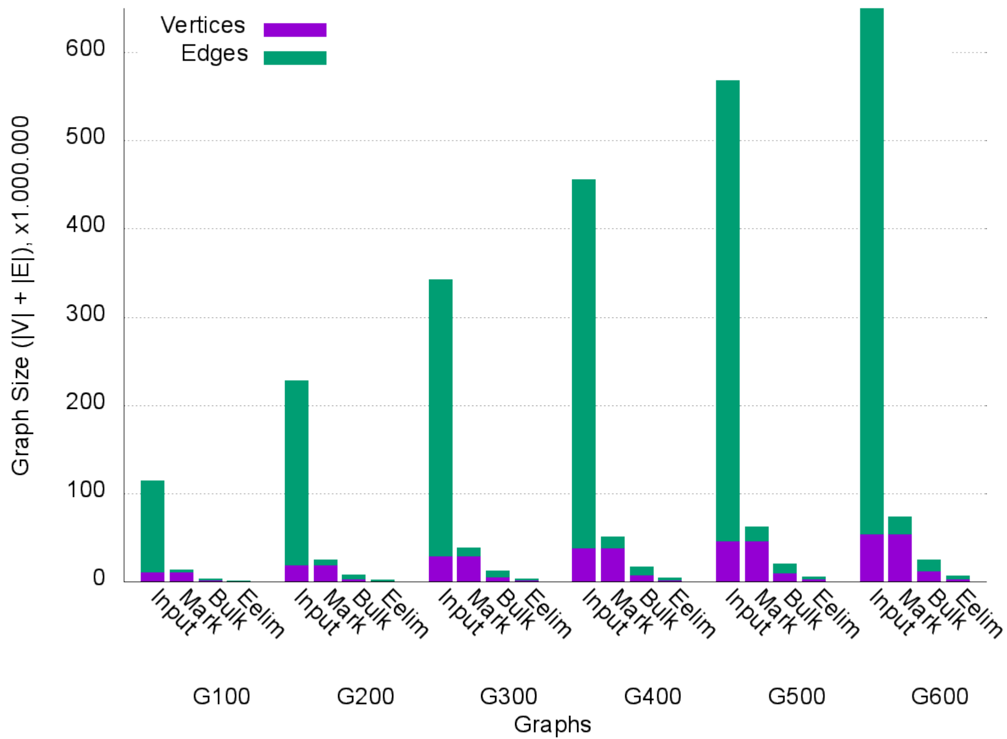
On Comparison with Spark SQL

The third query $Q3$ is used to compare the performance of our framework with a direct implementation of it in Spark SQL. This query is for the dataset of Amazon Product Co-purchasing Network. It returns all information of products that belongs to the group `Book`, in which we also return the field `asin` of *similar products of similar products of those products*:

$Q3$: **select** {`product` :
  ((**select** $\$a$ **where** {`asin` : $\$a$} **in** $\$p$) ∪
  (**select** $\$t$ **where** {`title` : $\$t$} **in** $\$p$) ∪
  (**select** $\$sr$ **where** {`salesrank` : $\$sr$} **in** $\$p$) ∪
  (**select** $\$c$ **where** {`category.Category.name` : $\$c$} **in** $\$p$) ∪
  (**select** $\$r$ **where** {`review.Review.customer` : $\$r$} **in** $\$p$) ∪
  (**select** $\$g$ **where** {`group` : $\$g$} **in** $\$p$) ∪
  (**select** $\$s$
   where {`similar.Product.similar.Product.asin` : $\$s$} **in** $\$p$)
)}
 where
  {`Product` : $\$p$} **in** $\$db$,
  `group.Group.name.Book` **in** $\$p$

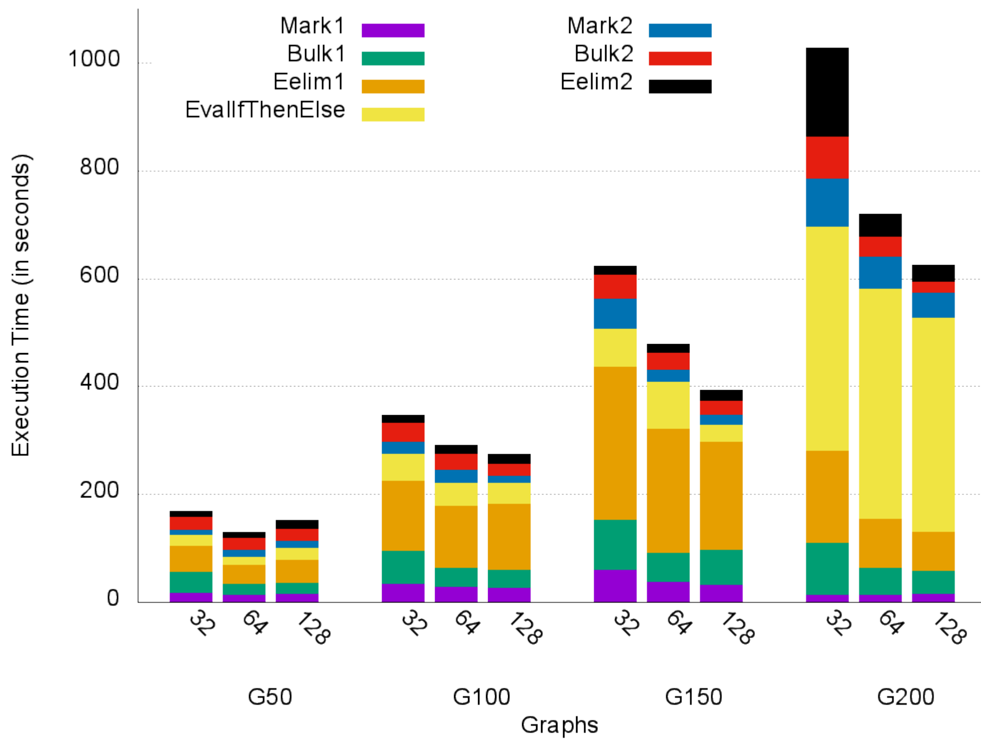


(a) Execution time of Q1

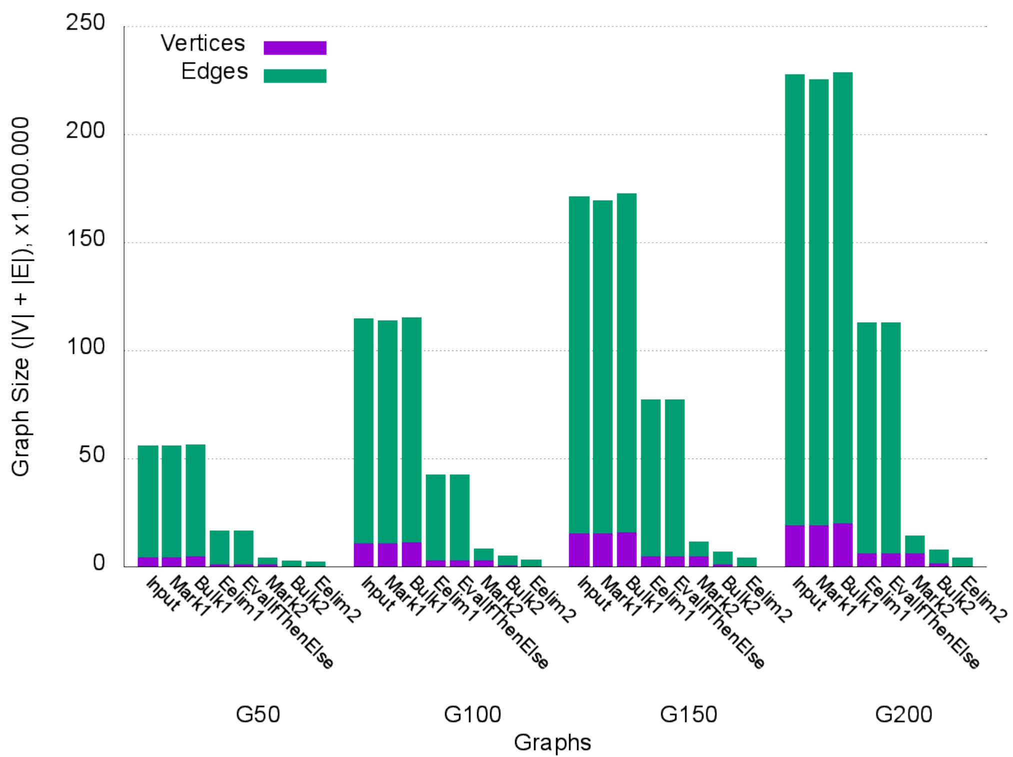


(b) Intermediate graphs generated during Q1

Figure 3.14: Query Q1 on Yahoo! AltaVista Web Page Graphs.



(a) Execution time of Q2



(b) Intermediate graphs generated during Q2

Figure 3.15: Query Q2 on Yahoo! AltaVista Web Page Graphs.

Table 3.3: Execution time (sec) on Amazon Product Dataset.

# of processors	16	32	64	128
Q3	421	217	199	254
Q3′	295	283	278	280

We write an SQL query in Spark SQL, named Q3′, which returns the same result as Q3, using *left outer join*. We will compare Q3 and Q3'.

$Q3'$: **select** $product.asin, product.title, product.salesrank, category.name,$
$customer.id, prodgroup.name, prod2.asin$
from $product$ **left outer join** $prodcat$ **on** $(product.id = prodcat.productId)$
left outer join $category$ **on** $(prodcat.categoryId = category.id)$
left outer join $review$ **on** $(review.productId = product.id)$
left outer join $customer$ **on** $(review.customer = customer.id)$
left outer join $prodsimilar$ **on** $(product.id = prodsimilar.productId)$
left outer join $product$ **as** $prod1$ **on** $(prodsimilar.productAsin = prod1.asin)$
left outer join $prodsimilar$ **as** $prodsim1$ **on** $(prod1.id = prodsim1.productId)$
left outer join $product$ **as** $prod2$ **on** $(prodsim1.productAsin = prod2.asin)$
left outer join $prodgroup$ **on** $(product.groupid = prodgroup.id)$
where $prodgroup.name$ = ”*Book*”

Table 3.3 compares the execution times of Q3 and Q3′ by varying the number of processors. The running time of Q3 is decreased faster than the one of Q3' when increasing the number of processors. In the beginning with 16 processors, Q3 is slower, but when using 32 processors, Q3 outperforms Q3′. Both Q3 and Q3' have the best performance with 64 processors. But, after that, their running times are increased.

Performance in Detail

Let us have a closer look at the performance of each phases in Pregel programs.

The algorithm to evaluate Q1 consists of three phases: *Mark*, *Bulk* and *Eelim*. From Figure 3.14(a), it is clear that the phase *Mark* takes most of the running time.

Figure 3.14(b) shows the size of intermediate graphs generated by each phase. We see that the phase *Mark* is rather effective when it filters out a huge amount of data from the input graphs, resulting in small intermediate graphs. This also helps reduce the running time of the latter phases (*Bulk* and *Eelim*). Therefore, it is worth doing further optimizations to the phase *Mark*.

The algorithm to evaluate Q2 consists of seven phases, where the first three phases are for the first specification without conditions, the last three phases are for the second specification without conditions, the fourth phase (middle phase) is for evaluating **if/then/else** statements. For this query, the first specification produces many ϵ-edges (Figure 3.15(b)). Hence the phase *Eelim1* takes much time (Figure 3.15(a)). The performance of the phase *EvalIfThenElse* is quite good except the one on the graph G200. In the future we need more analysis on the graph G200 to see where the issue comes from.

3.5 Related Work

Developing parallel algorithms that are scalable to big graphs has been studied intensively in recent years, particularly on development of easy-to-use distributed graph processing models and on design and implementation of DSL on these models.

Distributed graph processing models: MapReduce [13] is big data processing model, hence it can be used to process graphs. However, since the Map and Reduce computations are stateless, it does not naturally express iterative graph algorithms. GraphLab [49] adopts a vertex-centric model like the Pregel model, but it targets the asynchronous computation. Gonzalez et al. [50] propose PowerGraph that uses the gather-apply-scatter model to exploit parallelism in natural graphs. Nonetheless, to obtain scalability, these models offer specific forms in which users specify their programs. Hence, it is non-trivial for users to write their complicated programs as well as obtain efficient ones. Our framework offers a more intuitive way using a declarative language, to help users easily express large-graph computations.

Translating DSLs to graph processing models: Several researches proposed rules to translate the well-known languages for graphs into large-graph processing model. Nole et al. [20] translated regular path queries into the Pregel model, using Brzozowski's derivations of regular expressions. The result of the queries is a set of

vertex pairs. Krause et al. [51] proposed a high-level graph transformation framework on top of BSP model. In particular, they implemented the framework in Giraph, an open-source implementation of the Pregel model. The framework is based on graph grammars. However, these frameworks are not *compositional* in the sense that they do not support successive applications of queries/transformations. Another approach is done by Salihoglu et al. [52], in which they have found a set of high-level primitives that capture commonly appearing operators in large-graph computations. These primitives are also implemented in the GraphX library. Hong et al. [53] proposed translation rules to translate a subset of the Green-Marl language into the Pregel model, but this subset is not so *declarative* as ours because it still requires programmers to code explicitly the operations on each vertices and edges. SPARQL [4] is a popular query language for RDF data. Recently, there are many works trying to implement SPARQL in MapReduce or vertex-centric programming models like Pregel [54, 55, 56, 57]. Supporting regular expressions in SPARQL are intensively researched [58, 59, 60, 61, 62].

3.6 Summary

In this chapter, we have designed a light-weight framework on top of Pregel to answer select-where regular path queries. Its core is built based on a solid foundation of structural recursion on graphs and program transformations, making it a reliable and efficient framework. By using structural recursion to express queries, we can automatically derive Pregel programs. We have proposed an efficient evaluation for structural recursion, making it scalable to big graphs. To the best of our knowledge, this is the first Pregel-based framework to answer select-where regular path queries.

4
Regular Reachability Queries

As discussed in Chapter 3, the phase *Mark* in the evaluation of an SWRP query without condition is very effective to reduce the size of intermediate graphs, but it has a poor performance. Given a vertex v_s and a regular expression R, the phase *Mark* computes for each vertex v a set of states that are yielded by the automaton of R for all paths from v_s to v. The poor performance of the phase *Mark* is because of using a breadth-first-search like algorithm that only exploits the parallelism on branches of a vertex and is ineffective when dealing with long paths.

To exploit parallelism when using a regular expression to traverse a long path, Sin'ya et al. [63] has proposed a simultaneous finite automaton (SFA) for efficient parallel computation of the finite automaton of the regular expression. The idea of SFA is using functions to simulate all possible transitions in the finite automaton. Evaluating a regular expression over a long path is done by splitting the path into partitions and applying functions to the partitions in parallel. The final result is computed by doing function compositions (in parallel or sequential way). The good property of parallelism of an SFA allows easily developing a parallel implementation

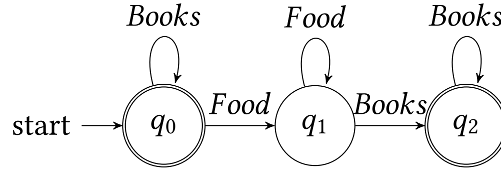


Figure 4.1: A DFA for $(Books*)(Food*)(Books+)$.

for a regular expression matcher.

In this chapter, we apply SFA to designing a distributed algorithm to answer a *regular reachability query* that checks whether there exists a path between two given vertices satisfying a given regular expression. Experimental results show that our algorithm outperforms a state-of-the-art algorithm [19] for the regular reachability query. To the best of our knowledge, this is the first work making an attempt at applying SFA to graphs. We believe that our algorithm has the potential to extend to speedup the phase *Mark* which is very close to the regular reachability query but needs tracking a set of states at each vertex. We discuss the extension at the end of this chapter.

4.1 Definition of Regular Reachability Queries

Given a vertex-labeled graph $G^v = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O}, \sigma)$, a regular reachability query (RRQ), $Q_R(v_s, v_t, R)$, checks whether there exists a path that starts from the vertex v_s, reaches the vertex v_t, and matches the regular expression R. Assume that edge labels are empty (ignoring edge labels). A path ρ is a sequence of vertices, denoted by $v_s = v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_w = v_t$, where $(v_i, \text{“”}, v_{i+1}) \in \mathcal{E}$ $(1 \le i \le k-1)$. We say that the path ρ matches R if and only if $\sigma(\rho) = \sigma(v_1)\sigma(v_2)\cdots\sigma(v_k)$ (excluding v_0 and v_w) matches R in the usual sense.

Figure 4.1 shows a DFA that corresponds to the regular expression

$$(Books*)(Food*)(Books+)$$

Here, q_0 is the initial state, and both q_0 and q_2 are the final states marked by concentric rings. Further, $q_i \xrightarrow{l} q_j$ denotes the transition from q_i to q_j according to the label l. We consider $\varnothing$ as a dead state, if transition $q_i \xrightarrow{l} \varnothing$ does not exist in the transition table.

4.2 Simultaneous Finite Automaton

Given a DFA $\mathcal{D}$, a *simultaneous finite automaton* (SFA, in short) [63], say $\mathcal{S}$, is an extended automata that involves speculative simulations from all states in $\mathcal{D}$. In particular, each state of $\mathcal{S}$ is a mapping from states to states in the original automata $\mathcal{D}$. In other words, let $L(\mathcal{A})$ denote the set of all the words accepted by $\mathcal{A}$, we have $L(\mathcal{D}) = L(\mathcal{S})$.

Definition 4.1 ([63]) *Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be an automaton. A simultaneous finite automaton (SFA) constructed from $\mathcal{A}$ is a quintuple $(Q_s, \Sigma, \delta_s, I_s, F_s)$:*

- *$Q_s \subseteq Q \to \mathcal{P}(Q)$ is a set of mappings from Q to the power set of Q;*
- *Σ is the same set of symbols as $\mathcal{A}$;*
- *δ_s is the additive extension of δ in $\mathcal{A}$ that is defined as below, where $f \in Q_s, \sigma \in \Sigma$,*

$$\delta_s(f, \sigma) := \bigcup_{q \in Q} (q \mapsto \bigcup_{q' \in f(q)} \delta(q', \sigma));$$

- *$I_s \subseteq Q_s$ is a singleton of identity mapping $\{f_I\}$ that satisfies $f_I(q) = \{q\}$ for any $q \in Q$;*
- *$F_s \subseteq Q_s$ is defined as $F_s = \{f \in Q_s \mid \exists q \in I,\ f(q) \cap F \neq \emptyset\}$.*

The SFA was originally invented for parallel regular expression matching. The following is a key property that enables us to perform a parallel computation [63]. Assume that the SFA yields f_1 for a word w_1 ($f_I \xrightarrow{w_1} f_1$) and f_2 for a word w_2 ($f_I \xrightarrow{w_2} f_2$); then, it yields $f_1 \bullet f_2$ for a word $w_1 w_2$, where $\bullet$ is the reverse function composition operator defined as follows.

$$(f_1 \bullet f_2)(q_1) = \bigcup_{q_2 \in f_1(q_1)} f_2(q_2)$$

In other words, if we split the input string and process each of substrings by the SFA in parallel; then, the above property enables us to merge the resulting states. It is worth noting that each substring is processed from the initial state f_I in the SFA.

An SFA of a regular expression R is constructed via the DFA of R as follows.

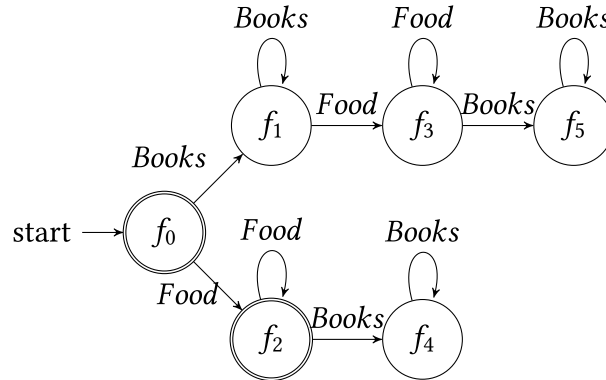


Figure 4.2: An SFA for the DFA shown in Figure 4.1.

Step 1 Construct a DFA $D,(Q,\Sigma,\delta,I,F)$, from R.

Step 2 Initialize a set $\mathcal{T}$ which contains only the mapping f_I, $f_I(q)=\{q\}, \forall q \in Q$.

Step 3 If $\mathcal{T}$ is empty then go to Step 6, otherwise go to Step 4

Step 4 Choose and remove a mapping f from $\mathcal{T}$. Add f to Q_s. For each label $a \in \Sigma$, do the following steps

(a) compute a new mapping $f_{next}(q), q \in Q$, as follows.

$$f_{next}(q) = \bigcup_{q' \in f(q)} \delta(q', a) \tag{4.1}$$

(b) $\delta_s(f,a) = f_{next}$

(c) If $f_{next} \notin Q_s$ then add f_{next} to $\mathcal{T}$

Step 5 Go to Step 3

Step 6 $I_s = \{f_I\}, F_s = \{f \in Q_s \mid \exists q \in I | f(q) \cap F \neq \emptyset\}$

Example 4.1 *Figure 4.2 shows an SFA for the DFA shown in Figure 4.1. Here, f_0 is the initial state. Table 4.1 summarizes states in the SFA.*

To distinguish states in SFA and states in DFA, we denote SFA states by f and DFA states by q.

Table 4.1: Mappings of states in the SFA shown in Figure 4.2.

f_0	f_1	f_2	f_3	f_4	f_5
$q_0 \mapsto \{q_0\}$ $q_1 \mapsto \{q_1\}$ $q_2 \mapsto \{q_2\}$	$q_0 \mapsto \{q_0\}$ $q_1 \mapsto \{q_2\}$ $q_2 \mapsto \{q_2\}$	$q_0 \mapsto \{q_1\}$ $q_1 \mapsto \{q_1\}$ $q_2 \mapsto \varnothing$	$q_0 \mapsto \{q_1\}$ $q_1 \mapsto \varnothing$ $q_2 \mapsto \varnothing$	$q_0 \mapsto \{q_2\}$ $q_1 \mapsto \{q_2\}$ $q_2 \mapsto \varnothing$	$q_0 \mapsto \{q_2\}$ $q_1 \mapsto \varnothing$ $q_2 \mapsto \varnothing$

4.3 Functional-based Evaluation of Regular Reachability Queries

In this section, we focus on how to use SFAs to obtain an efficient evaluation of RR queries on distributed graphs.

4.3.1 Evaluation Strategy

We utilize an evaluation strategy for RR queries on distributed graph in [19], and make it adapt to SFA. We consider a setting of distributed environment where there are a *coordinator* site S_c and m *worker* sites S_i. Without loss of generality, we assume that the number of sites is equal to the number of fragments of a distributed graph. The strategy is shown in Figure 4.3, and includes four communication steps and two computation steps.

1. The client sends a query $Q_R(v_s, v_t, R)$ to *coordinator* S_c.

2. S_c broadcasts the received Q_R to all *workers* $S_i (1 \leq i \leq m)$.

3. Each *worker* site S_i loads a fragment DG_i, and locally computes a partial result. This step is called *partial evaluation.*

4. The partial results will be sent back to S_c for further evaluation.

5. After collecting all the partial results, S_c constructs a dependency graph G_d. Then does a global computation on G_d to get the final result. This step is called *global evaluation.*

6. S_c sends the final result back to the client.

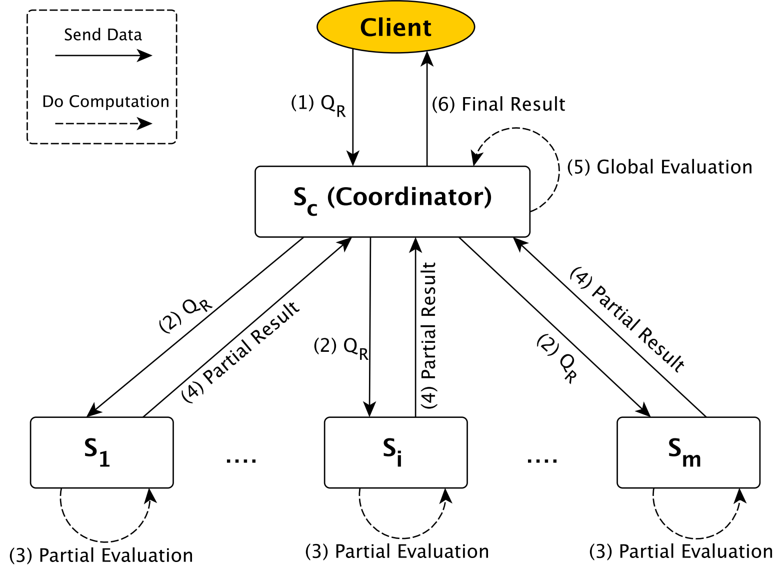


Figure 4.3: A distributed evaluation strategy for RR queries.

Our objectives are (1) minimizing the computational cost of partial evaluations to speedup the step 3; (2) minimizing the size of partial results to reduce the communication cost in the step 4, and to avoid the bottleneck at the coordinator when receiving partial results.

Our solution which is based on SFA will be introduced in the next sections, where we propose algorithms for the partial evaluation and global evaluation.

4.3.2 Partial Evaluation

Partial evaluation is performed in parallel by all *workers*. Each worker S_i invokes a function computePartialResult to compute a partial result by evaluating the received Q_R on a local fragment DG_i. The partial result is a set of mappings from an input vertex to a set of tuple of an output vertex and a state.

The pseudo-code of computePartialResult is shown in Algorithm 4.1. First, we compute an SFA, say $\mathcal{S} = (Q_s, \Sigma, \delta_s, I_s, F_s)$, for the regular expression R (line 1), using the method presented in Section 4.2. A partial result is stored in the variable *pSet* that is a set of mappings $v_{in} \mapsto \{(v_{out}, f)\}$, $v_{in} \in I_i, v_{out} \in O_i, f \in Q_s$, where a mapping implies that an input vertex v_{in} can reach output vertices v_{out} via paths for which the

SFA yields states f. Second, we initialize sets of input and output vertices, where v_s and v_t are also input and out vertices, respectively. For each vertex, we maintain two vectors, *visited* and *rset*. The vector *visited* is to mark a vertex whether it is visited with a given state or not, which avoids producing an infinite loop of computation. The second vector *rset* is to store a result for a given vertex and state once they are visited. Next, the evaluation starts from input vertices with *the initial state* in the SFA (line 8–9). Then it does a depth-first search (the function visit) to recursively visit other vertices (line 9). Finally, the results for each input vertex are speculated into the partial result (variable *pSet*, line 19).

The function visit accepts a vertex (not only input vertex) and a state, then computes a set of pairs of an output vertex and a state. It is a depth-first search. It terminates when reaching a visited vertex or an output vertex.

It is worth noting that for each input vertex, our algorithm always starts from the initial state (line 9, Algorithm 4.1). This is because we can construct the final result by utilizing the properties of SFA (Sec. 4.2). It is different from other approaches ([19],[64]) where one needs to consider all states for each input vertex. As a result, our approach reduces many computations at each vertex, making our partial evaluation more efficient than the one using DFA.

Procedure 4.1 Procedure computePartialResult

Input: Fragment $DG_i = (\mathcal{V}_i, \mathcal{E}_i, \mathcal{I}_i, O_i, \sigma_i, \mathcal{C}_i)$ and query $Q_R(v_s, v_t, R)$.
Output: A set of $(v_{in} \mapsto \{(v_{out}, f)\})$.

1: Generate an SFA, $S(Q_s, \Sigma, \delta_s, I_s, F_s)$, from R;
2: $pSet = \varnothing$; $iset = \mathcal{I}_i$; $oset = \mathcal{O}_i$;
3: **if** $v_s \in \mathcal{V}_i$ **then** $iset = iset \cup \{v_s\}$ **end if**;
4: **if** $v_t \in \mathcal{V}_i$ **then** $oset = oset \cup \{v_t\}$ **end if**;
5: **for each** $v \in V_i$ **do**
6:     **for each** $f \in Q_s$ **do** $v.visited[f] = false$ **end for**;
7: **end for**
8: **for each** $v \in iset$ **do**
9:     $v.rset[f_I] =$ visit(v, f_I); // $f_I \in I_s$ is the initial state
10:     $pSet = pSet \cup \{v \mapsto v.rset[f_I]\}$;
11: **end for**
12: **return** $pSet$;

Procedure 4.2 Procedure visit

Input: A vertex v, a state f.

Output: A set of (u, f').

1: **if** $v.visited[f]$ **then return** $v.rset[f]$ **end if**;
2: $v.visited[f] = true$;
3: **if** $v \in oset$ **then**
4: **if** v is v_t **then**
5: $v.rset[f] = v.rset[f] \cup \{(v, f)\}$;
6: **else**
7: $f_{next} = \delta_s(f, \sigma_i(v))$;
8: $v.rset[f] = v.rset[f] \cup \{(v, f_{next})\}$;
9: **end if**
10: **end if**
11: **for each** vertex $u \in \{u|(v,u) \in \mathcal{E}_i \cup \mathcal{C}_i\}$ **do**
12: $f_{next} = \delta_s(f, \sigma_i(u))$;
13: $v.rset[f] = v.rset[f] \cup$ visit(u, f_{next});
14: **end for**
15: **return** $v.rset[f]$;

Example 4.2 *Recall the graph in Figure 2.2 and query $Q_R(v_1, v_{17}, R)$, where*

$$R = (Books*)(Food*)(Books+)$$

We use DG_1 as an example. Considering the input vertex v_1 and the output vertex v_{11}, a path $v_1 \to v_2 \to v_5 \to v_3 \to v_6 \to v_{11}$. First, we visit v_1 with the initial state f_0. The next state $f_{next} = \delta_s(f_0, \sigma_1(v_2)) = f_1$, since $\sigma_1(v_2) = Books$. Next, we visit the vertex v_2 with the state f_1. This procedure continues until we reach the output vertex v_{11} with the state f_3. Therefore, we add a mapping $v_1 \mapsto \{(v_{11}, f_3)\}$ to the partial result. The whole partial result of the example are listed in Table 4.2.

4.3.3 Global Evaluation

Partial results computed in the partial evaluation are collected to the coordinator S_c, and a dependency graph is constructed from the partial results as follows. For each mapping $v \mapsto \{(u, f)\}$, create edges (i, f, j), where $\sigma(i) = v, \sigma(j) = u$, i and j are vertex identifiers generated arbitrarily. Note that there might have parallel edges of

Table 4.2: An example of the output of the partial evaluation.

DG_i	$pSet$
DG_1	$\{v_1 \mapsto \{(v_7, f_3), (v_{11}, f_3)\},$ $v_6 \mapsto \{(v_{11}, f_2)\}\}$
DG_2	$\{v_7 \mapsto \{(v_6, f_2)\},$ $v_{11} \mapsto \{(v_{13}, f_2), (v_{14}, f_1)\}\}$
DG_3	$\{v_{13} \mapsto \{(v_6, f_2)\}, v_{14} \mapsto \{(v_{17}, f_1)\}\}$

different labels between two vertices.

The problem of evaluating a query $Q_R(v_s, v_t, R)$ is now reduced to the one that checks if there exists a path $\rho = i \xrightarrow{f_1} j \xrightarrow{f_2} \ldots \xrightarrow{f_m} k$ satisfies (1) $\sigma(i) = v_s, \sigma(k) = v_t$; (2) $f_1 \bullet f_2 \bullet \ldots \bullet f_m$ yields an accept state f_ρ in the SFA of R; and (3) $\exists q \in f_\rho(q_0) \mid q$ is an accept state in the DFA of R, where q_0 is the initial state in the DFA of R. This check can be easily done by the algorithm computeFinalResult (Algorithm 4.3).

Procedure 4.3 Procedure computeFinalResult

Input: A dependency graph $G_d = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O}, \sigma, \mathcal{C})$ and query $Q_R(v_s, v_t, R)$
Output: A boolean value.

1: Generate an SFA, $S(Q_s, \Sigma, \delta_s, I_s, F_s)$, from R;
2: **for each** $v \in \mathcal{V}$ **do**
3: **for each** $f \in Q_s$ **do** $v.visited[f] = false$ **end for**;
4: $v.value = \varnothing$;
5: **end for**
6: $result$ = visitFunc(v_s, I_s)
7: **return** $result$;

Example 4.3 *Continuing with the previous example, G_d is built from the partial results as shown in Figure 4.4. The dependency graph holds several valid paths regarding the example query. We use the path $\rho = 1 \xrightarrow{f_3} 3 \xrightarrow{f_2} 2 \xrightarrow{f_2} 4 \xrightarrow{f_1} 6 \xrightarrow{f_1} 7$ as an example. The recursive calculation along the path would be $f_\rho = f_0 \bullet f_3 \bullet f_2 \bullet f_2 \bullet f_1 \bullet f_1 = f_5$. Since $f_5(q_0) = q_2$ which is an accept state, a valid path is detected. Thus, S_c returns True to the client.*

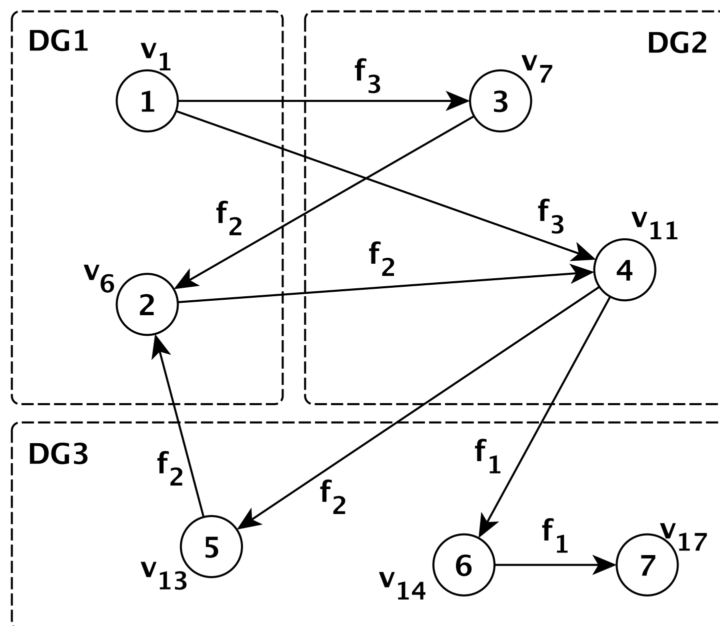


Figure 4.4: A dependency graph constructed on S_c. Note that this is not a distributed graph. We use dotted rectangles to denote that from which fragment vertices come.

Computational Cost In our approach, we visit each worker site only once after receiving a query from the coordinator site. The computation cost includes the local and global evaluation costs. Given a regular expression R, let $\mathcal{D}$ be the DFA of R, $\mathcal{S}$ the SFA of $\mathcal{D}$. Let $|\mathcal{S}|$ be the number of states in $\mathcal{S}$, $|\mathcal{D}|$ the number of states in $\mathcal{D}$, $|\mathcal{V}_i|$ the number of vertices in DG_i, and $|\mathcal{V}_c|$ the number of cross-edges. As for the algorithm computePartialResult, it takes $O(|\mathcal{S}|)$ time to construct an SFA. It visits each vertex at most $|\mathcal{S}|$, hence it takes at most $O(|\mathcal{S}||\mathcal{V}_i|)$ to compute all mappings in a partial result. The global evaluation takes at most $O(|\mathcal{S}||\mathcal{V}_c|^2|\mathcal{D}|^2)$ to compute the final result, because each function composition operator takes at most $|\mathcal{D}|^2$ time.

Communication cost Communication cost in our approach consists of broadcasting Q_R to all m workers and sending partial results from the worker sites to the coordinator site (We ignore communications between the system and the client). The broadcasting cost is $O(m|\mathcal{D}|)$. The size of a partial result is $O(|\mathcal{S}||\mathcal{I}_i||\mathcal{O}_i|)$. Hence, it takes $O(|\mathcal{S}||\mathcal{V}_c|^2)$ to send all partial results to the coordinator.

Procedure 4.4 Procedure visitFunc

Input: A vertex v, a set of state $\mathcal{F}$.
Output: A boolean value.

1: **for each** state $f \in \mathcal{F}$ **do**
2:   **if** $v.visited[f]$ **then continue end if**;
3:   $v.visited[f] = true$;
4:   **if** $L(v) = v_t$ **then**
5:     **if** $f \in F_s$ and $f(q_0)$ contains an accept state in the DFA **then**
6:       **return** *True*
7:     **end if**
8:   **else**
9:     **for each** edge $v \xrightarrow{f'} u$ **do**
10:       $f_{next} = f \bullet f'$;
11:       $result$ = computeFinalResult$(u, u.value \cup \{f_{next}\})$;
12:       **if** $result = True$ **then return** *True* **end if**;
13:     **end for**
14:   **end if**
15: **end for**
16: **return** *False*;

4.3.4 Optimizations

Global Evaluation In the global evaluation, recall that for each path from the source vertex i ($\sigma(i) = v_s$) to the target vertex j ($\sigma(j) = v_t$). We compute a mapping (state) f_ρ that is composed of mappings on the path, say $f_\rho = f_1 \bullet f_2 \bullet \ldots \bullet f_m$. Next, we compute $f_\rho(q_0)$ to get the final result. However, mapping composition is an expensive operator. Looking at its definition, $\forall q \in Q$,

$$(f_1 \bullet f_2)(q) = \bigcup_{q' \in f_1(q)} f_2(q')$$

it takes at most $O(|\mathcal{D}|^2)$ time to do a mapping composition. We can avoid doing mapping composition by computing the result of f_ρ for only q_0 [63]. It it done by sequentially applying the first mapping f_1 on q_0 to produce a set of states, then each of returned state will be the input of the next mapping in the sequence. This procedure is done by looking at the mapping table of states of the SFA (e.g. Table 4.1).

On-demand Function Generation Given that a complex query might result in a large amount of states in an SFA and that we need to generate an SFA for both partial and global evaluations, the construction of SFAs becomes slow. Therefore, we adopt a on-demand construction of SFAs [63].

4.4 Implementation and Experiments

4.4.1 Implementation

We implemented our algorithm on top of the Hadoop framework [25], which is one of widely used implementations of the MapReduce model [13], allowing applications to run in a distributed manner on a large cluster.

We assume that fragments of a distributed graph are stored in the hadoop distributed file system (HDFS) as files. An input query is also stored in a file in HDFS.

We use one MapReduce job to implement our evaluation strategy. First, the job loads the query and stores it in an environment variable so that it can be accessed by workers. Second, in the map phase, each mapper will load a fragment as its input, then access the query. Mappers invoke the function computePartialResult to compute a partial result *pSet* locally, and send a message $(1, pSet)$ to the reduce phase. By sharing the same key, all messages come to one reducer in the reduce phase. That reducer constructs a dependency graph and invokes the function computeFinalResult to compute the final result, and outputs the result to a file.

4.4.2 Experiments

Experimental Environment Experiments were carried out on a cluster of five Lenovo servers. Each server has two Intel Xeon E5645 processors (6 cores at 2.4GHz) and 32GB RAM. We deployed a Hadoop environment (version 2.5.0) on the cluster, where one machine served as the master node and the other four were the slave nodes. Each experiment was performed with 10 random RRQs. We ran each query three times and reported the average running time. The performance of an algorithm is the overall running time, including the time for generating and distributing an automaton table, the time for local and global evaluation, and the time for data transfer. The time

Table 4.3: Real-life datasets.

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{L}\rvert$
YouTube	730,119	8,048,796	14
DBLP	816,210	1,063,528	2978
MEME	1,073,512	4,653,081	7
Internet	76,521	836,527	256
Random	1,000,000	1.5M~50M	26

taken to load a graph from HDFS and the time taken to set up map/reduce jobs were disregarded in all the experiments.

Datasets We used real-life datasets of YouTube [42], DBLP [41], MEME [65], Internet [66], and a dataset of randomly generated graphs, Random. The size of graphs that generated from these datasets are summarized in Table 4.3, where $|\mathcal{L}|$ is the number of vertex labels in a graph. The **YouTube** graph contains metadata of videos and the relationship between videos according to the *recommendation*. Each node represents a video and each edge implies that one video is recommended to another. We chose video categories as node labels. **DBLP** is a citation network whose nodes are paper IDs, and the edges denote citations. The node labels are the publication venues of papers. **MEME** is a blog network, where each node is a web page and each edge denotes a hyperlink in the web page. The **Internet** graph was extracted from an Internet traffic dataset published by CAIDA. We take IP addresses as vertices, and each vertex is labeled with the first eight digits of its IP address. The edges between vertices describe Internet connection. **Random** is a synthetic graph generated by a graph-tool [67]. We generated random graphs with different numbers of edges (from 1.5 million to 50 million). The numbers of vertices of these graphs are constant and set to 1 million. A vertex label is a single character that was randomly selected from the alphabet of 26 letters.

Graph Partitioning We used a random partitioning strategy to split a graph into subgraphs. The strategy is vertex-based; it computes a hash value of the vertex id modulo the number of partitions.

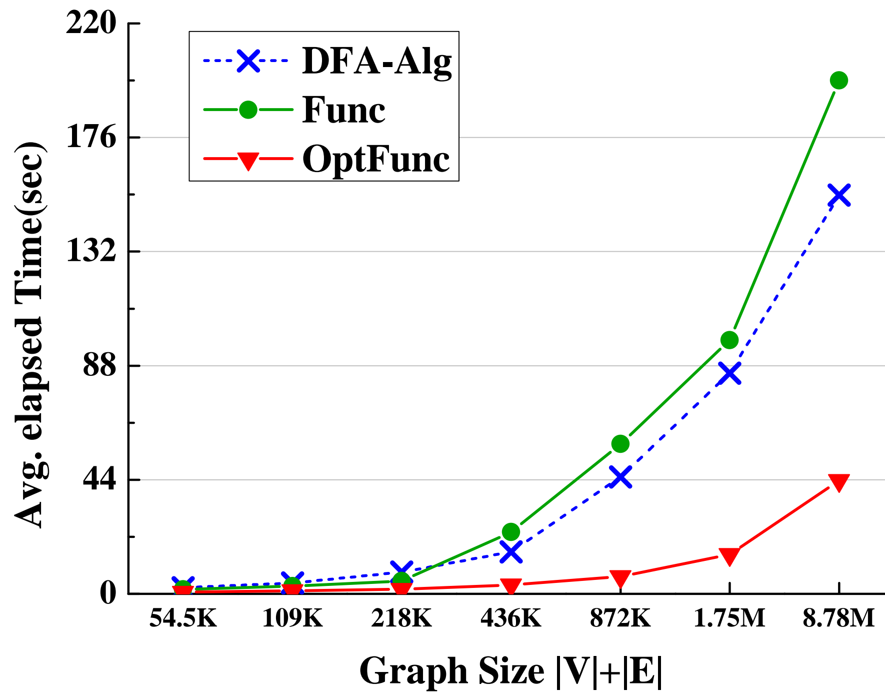
Random Query Generator Queries in our experiments were randomly generated in the following steps: (1) collecting all the distinct labels in the input graph to create a label collection $\mathcal{L}$; (2) randomly selecting labels from $\mathcal{L}$ and operators to combine a valid regular expression; (3) randomly selecting two vertices from the graph to be the source vertex and target vertex, v_s and v_t, respectively.

Benchmark To validate the performance of our algorithms, we compared them with a recent RRQ approach [19] that also used the partial evaluation. We refer this approach *DFA-Alg*. Our functional-based evaluation is denoted by *Func* and our functional-based evaluation with optimizations is denoted *OptFunc*.

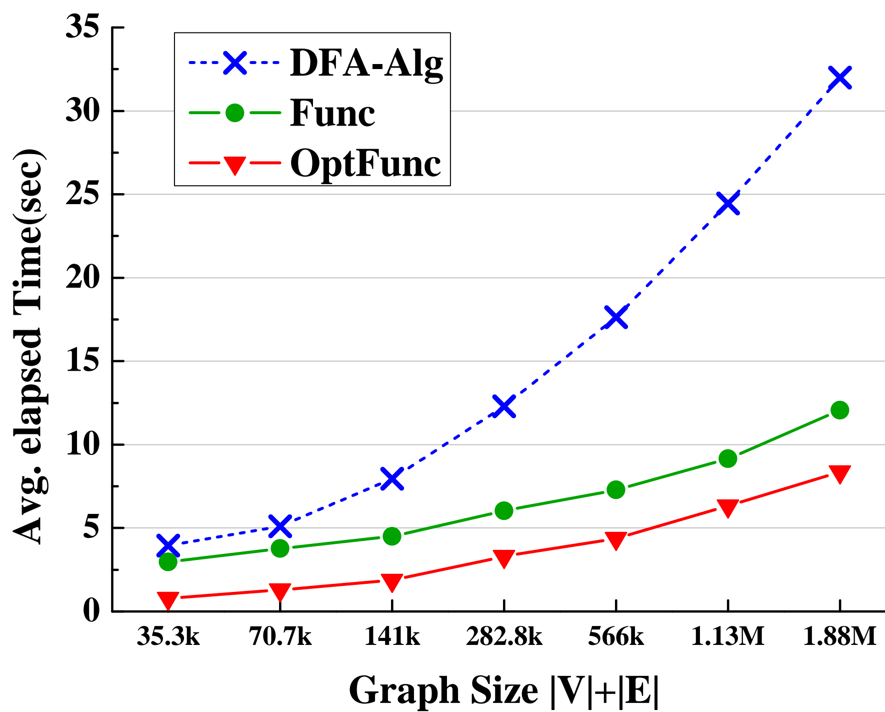
We now present and analyze our experimental results.

Effect of Dataset Size In the first set of experiments, we evaluated the scalability of the *Func* algorithm by using graphs of different sizes. We varied the graph size by traversing a graph with a constant number of vertices, and gradually doubling $|\mathcal{V}|$. We fixed the number of partitions to four and the query size to eight states. Figures 4.5 and 4.6 show the overall time consumed by different algorithms over four datasets (YouTube, DBLP, MEME and Internet). *OptFunc* is much faster than *DFA-Alg* for large graphs. With the largest graph, *OptFunc* performs four times faster than *DFA-Alg* on average. For a smaller graph, say Internet at $13.8K$, *OptFunc* is still around two times faster than *DFA-Alg*. The algorithm *Func* shows close performance to *DFA-Alg* for YouTube Figure 4.5(a)) and Internet (Figure 4.6(b)). This is because *Func* is sensitive to the characteristics of graph; the number of cycles in YouTube and Internet is much greater than that in other graphs, and hence *Func* became slower. We will study more on this issue at the end of this section.

Effect of Query Size The second set of experiments reports the performance of functional-based algorithms over various query sizes. We represented the query size by the number of DFA states. The query size was increased exponentially from 2 to 32. The number of partitions was fixed to four. As shown in Figures 4.7 and 4.8, *OptFunc* takes less time for the evaluation and is less sensitive to the query size. It is at least two times improvement compared with *DFA-Alg*, and its performance improves as the query size increases (four times faster with 32 states). This is because *DFA-Alg*

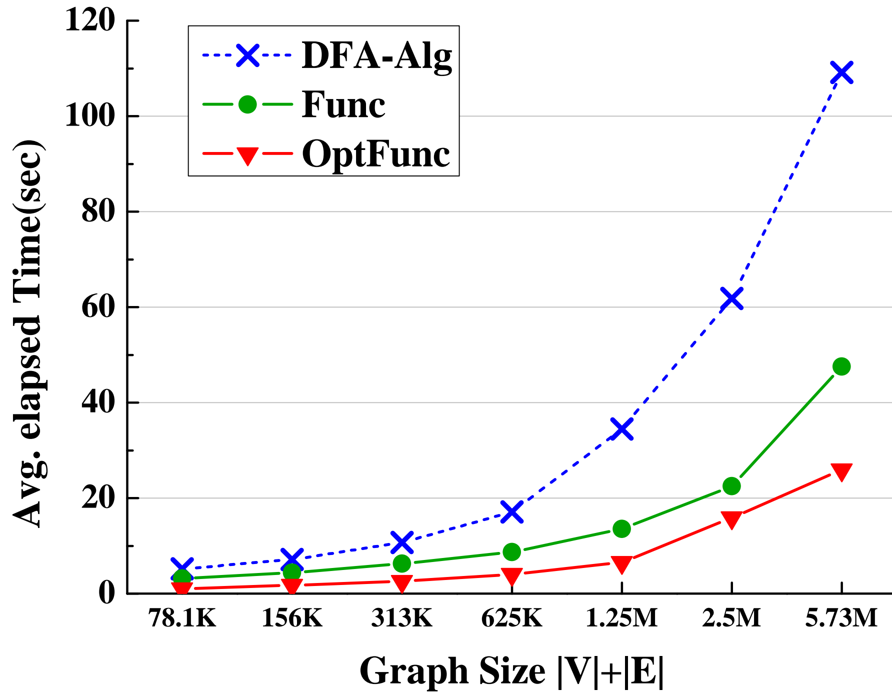


(a) YouTube

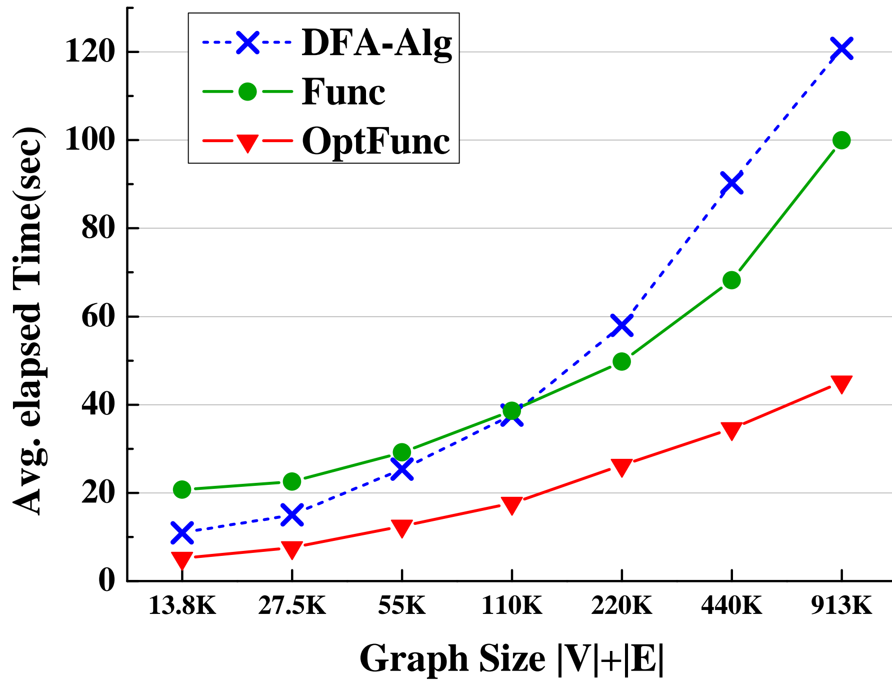


(b) DBLP

Figure 4.5: Experiment results with various graph sizes (YouTube and DBLP).



(a) MEME



(b) Internet

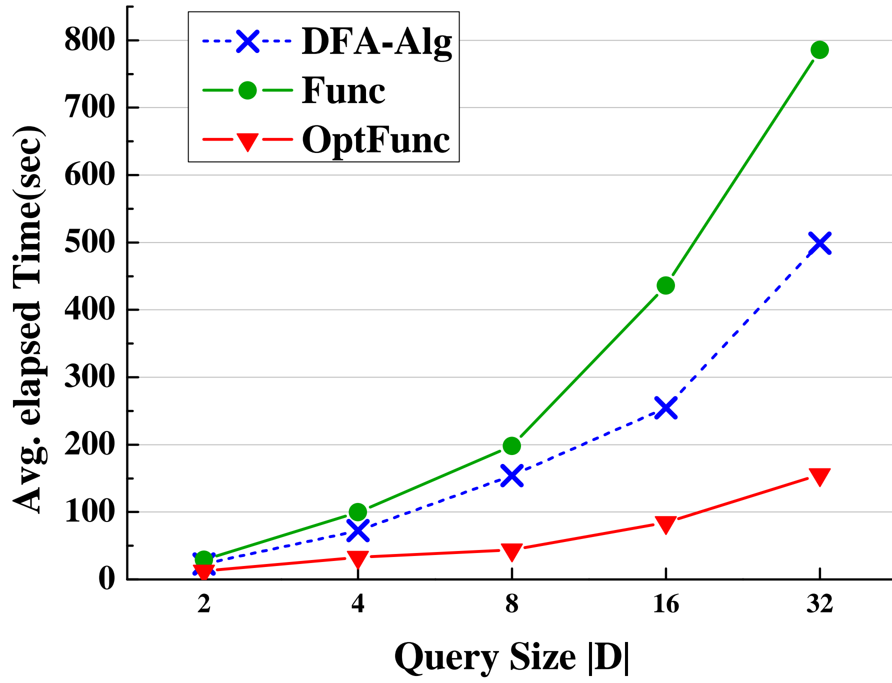
Figure 4.6: Experiment results with various graph sizes (MEME and Internet).

computes $|\mathcal{D}|$ times on each vertex, whereas *OptFunc* performs just one computation for all the states at the same time and traverses DG_i only once.

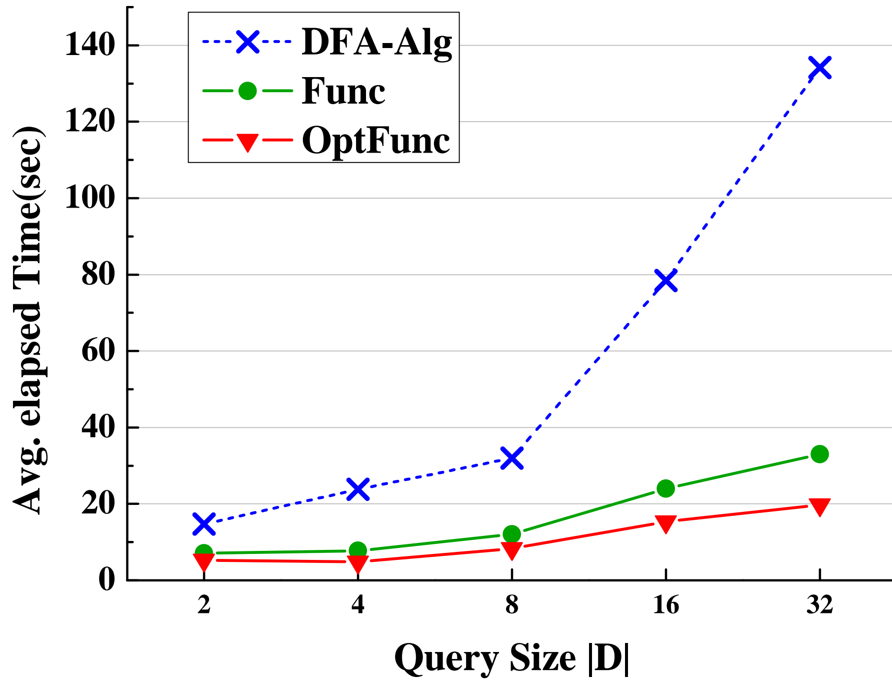
Effect of Number of Partitions This set of experiments reports the speedup of *Func* and *OptFunc* over the cluster size. We varied the number of partitions from 2 to 128, and set the number of mappers to be equal to the number of partitions. The query size was fixed to eight and we used the graphs with their largest size in all settings. From Figure 4.9(a), we can see a slight increase on 32 partitions for the algorithms *Func* and *OptFunc*. This is because the overhead of the global evaluation exceeds the gain due to more partitions. From Figures 4.9 and 4.10, it is worth noting that increasing the number of partitions does not have a significant effect on the overall performance of our approach. However, *OptFunc* still is around two times faster than *DFA-Alg* with more partitions (>64).

Mapper, Reducer and Network Cost The fourth set of experiments shows how functional-based evaluation exhibits better performance. We investigated the cost of each part of the algorithms, including **Mapper**, data transfer (**Trans**), **Reducer** and automaton generation (**Autogen**). Each experiment used four partitions and the largest graphs from each dataset. For all the runs, the size of the queries was set to eight. The results are shown in Figure 4.11(a). It is obvious that, for the *DFA-Alg* algorithm, the main cost is due to the map and data transfer stages, which consumed 90% of the total time (**Mapper**, 44.3%; **Trans**, 52.1%), On the other hand, our functional-based algorithms reduce the **Mapper** and **Trans** cost by more than 10 times, but correspondingly increase the **Reducer** cost. With the optimization of the sequential global evaluation, the cost of the **Reducer** of *OptFunc* is at the same level as that of *DFA-Alg*. Furthermore, the time consumed by **Autogen** is negligible for large graphs.

Properties of Functional-based Algorithms We also verified the characteristics of our functional-based algorithms by using synthetic datasets. It is clear that the functional-based algorithms only traverse a graph once from v_{in} to v_{out} if the graph is acyclic. When a cycle exists, functional-based algorithms will go through the cycle at most $|\mathcal{S}|$ times until a visited state is met. Compared with DFA, SFA transition table

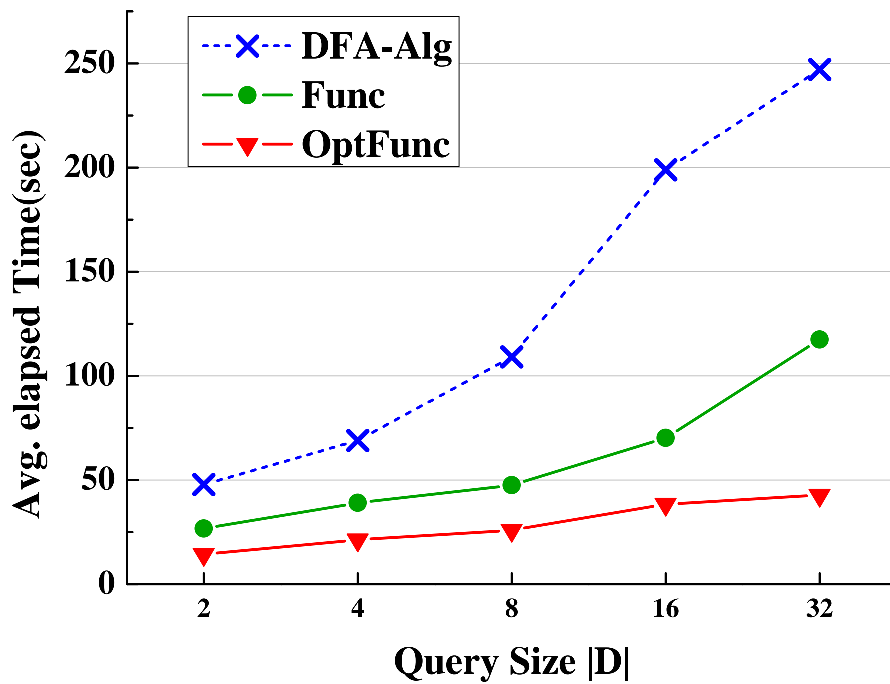


(a) YouTube

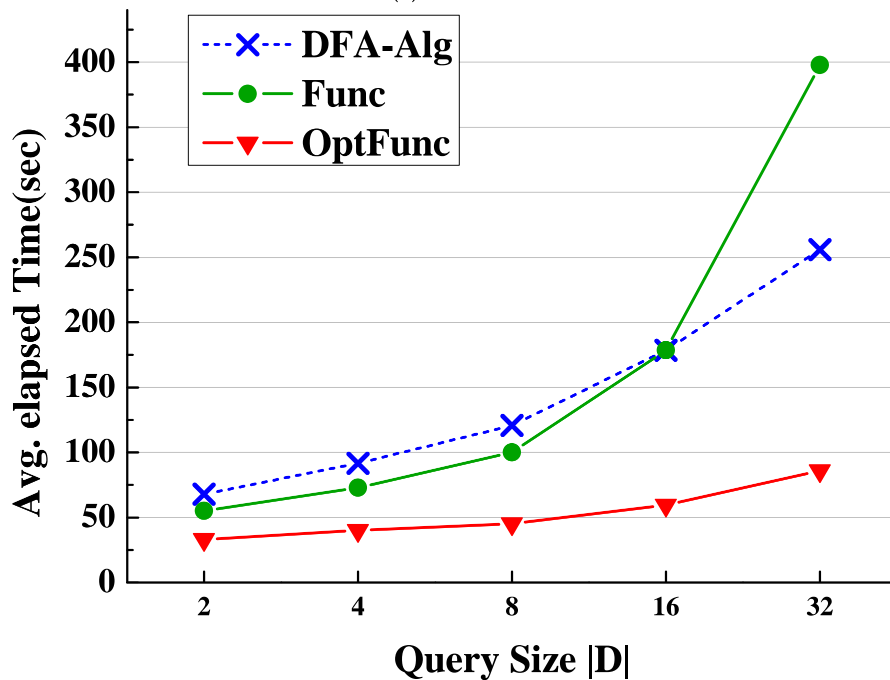


(b) DBLP

Figure 4.7: Experiment results with various query sizes (YouTube and DBLP datasets).

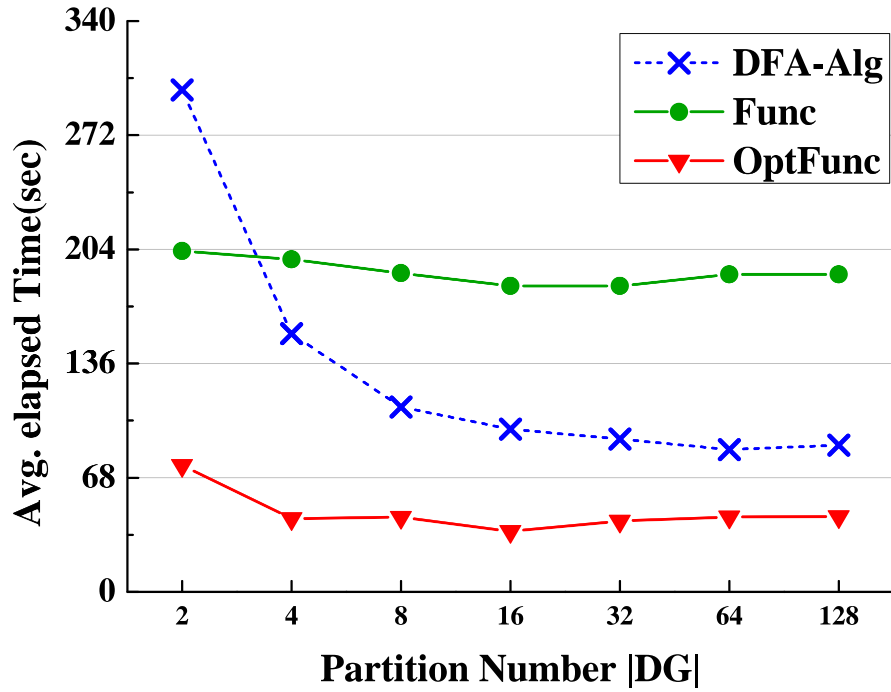


(a) MEME

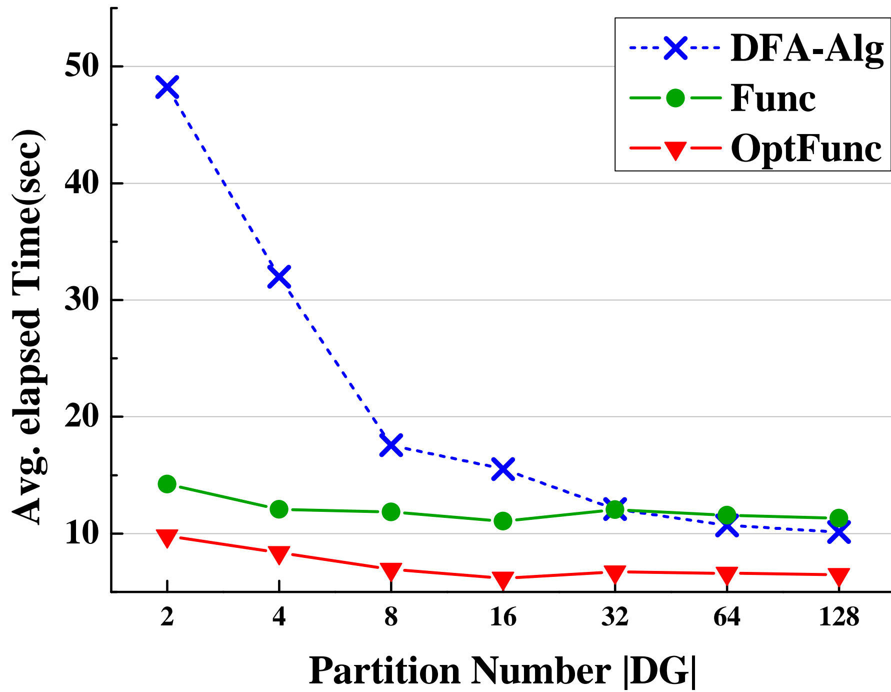


(b) Internet

Figure 4.8: Experiment results with various query sizes (MEME and Internet datasets).

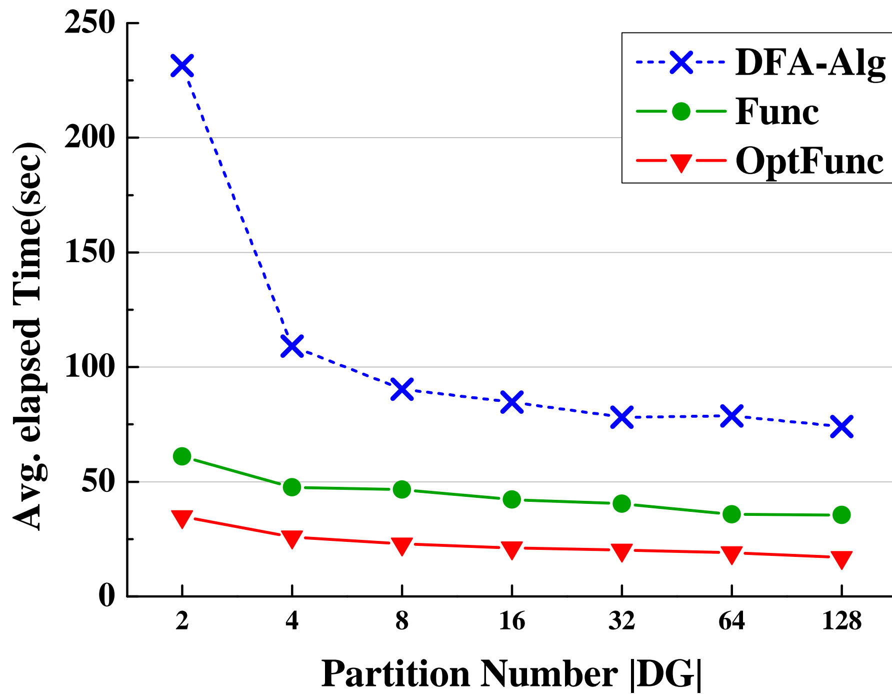


(a) YouTube

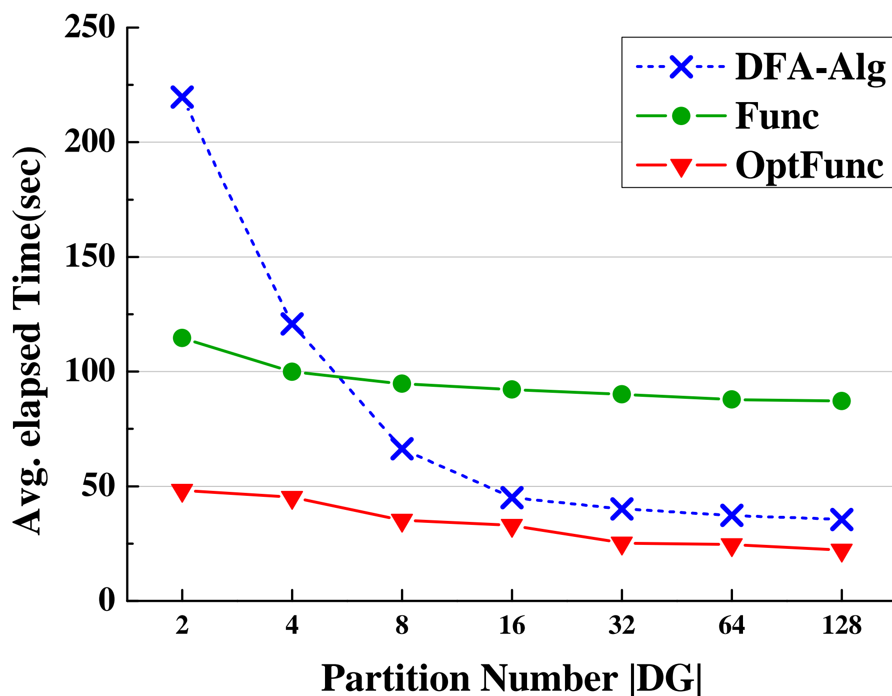


(b) DBLP

Figure 4.9: Experiment results with varying number of partitions (Youtube and DBLP).

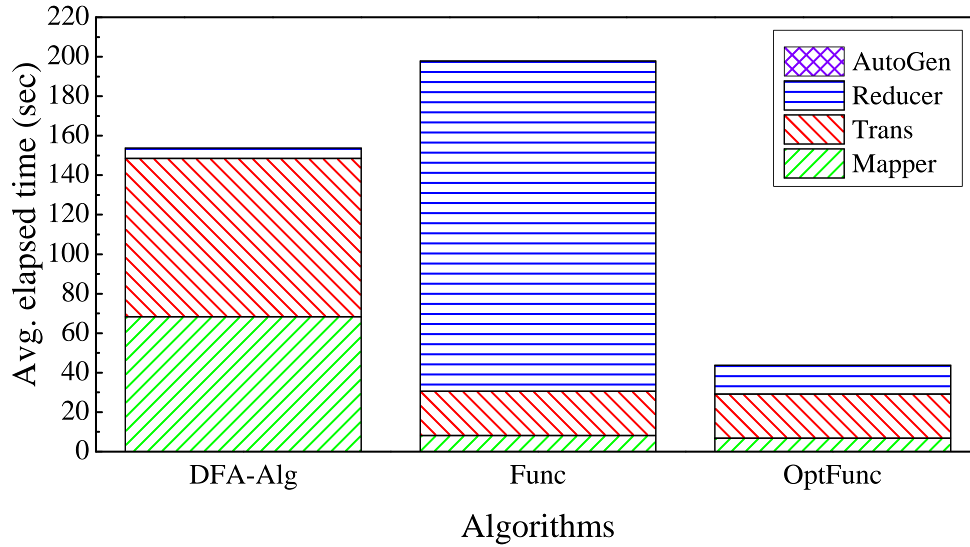


(a) MEME

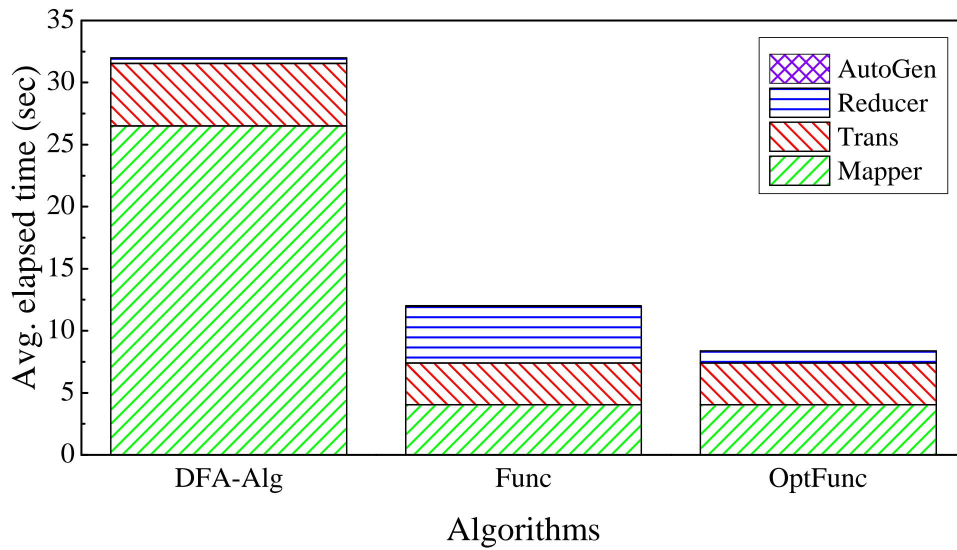


(b) Internet

Figure 4.10: Experiment results with varying number of partitions (MEME and Internet).

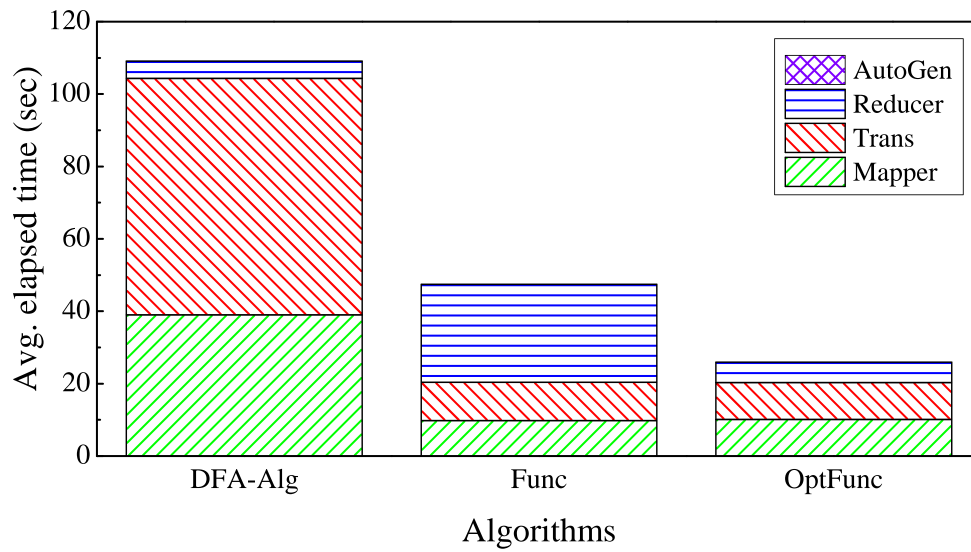


(a) YouTube

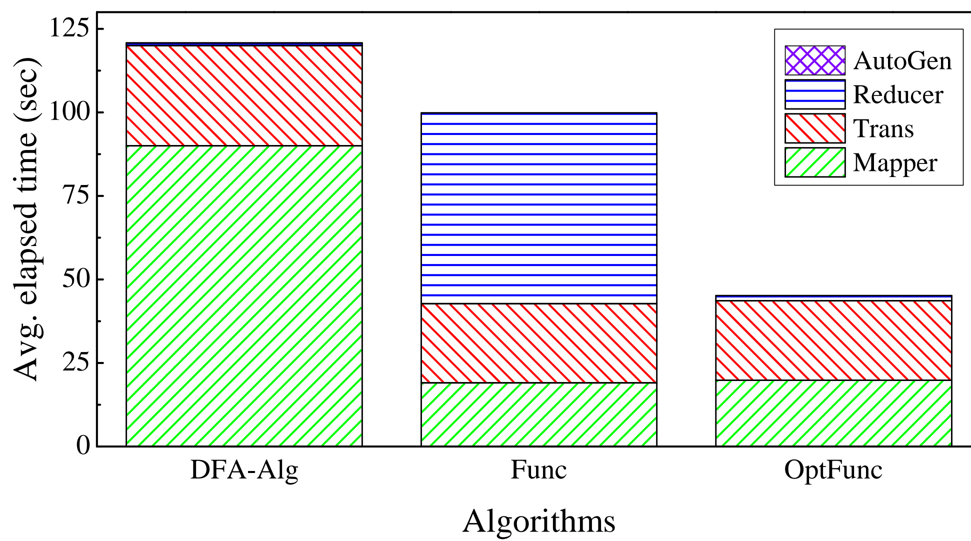


(b) DBLP

Figure 4.11: Experiment results for costs of different components (Youtube and DBLP).



(a) MEME



(b) Internet

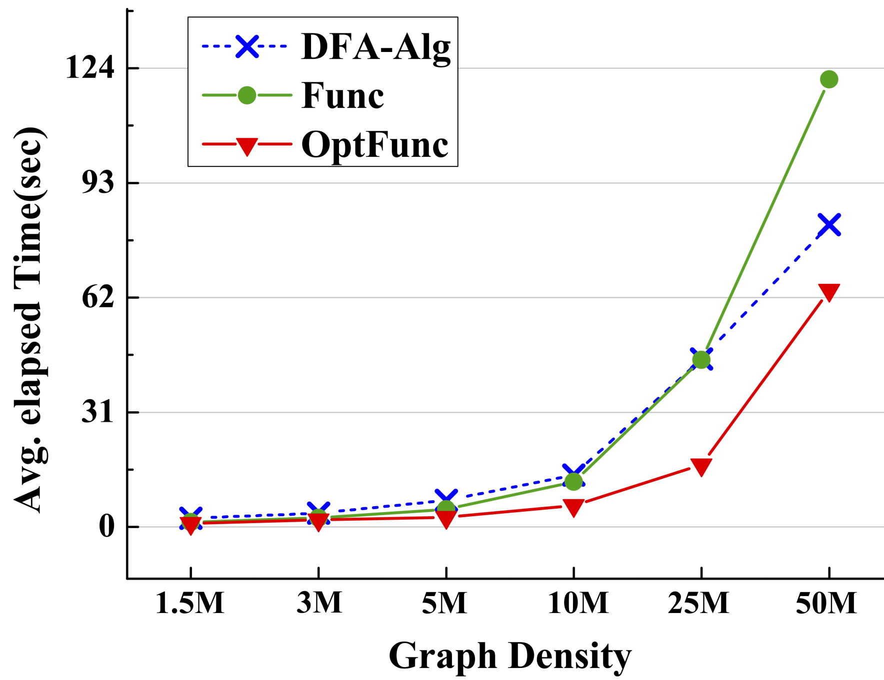
Figure 4.12: Experiment results for costs of different components (MEME and Internet).

could be very large. In theory, $|\mathcal{S}| = |\mathcal{D}|^{|\mathcal{D}|}$. Therefore, they might suffer from severe performance degradation when the number of cycles increases. In this experiment, we show the relationship between our approaches and the graph density $|\mathcal{E}|/|\mathcal{V}|^2$. Graphs were randomly generated with a fixed number of vertices. The number of edges ranged from 1.5 million to 50 million. The query size was fixed to eight and the number of partitions was four. The results are shown in Figure 4.13. Figure 4.13(a) clearly shows that the functional-based algorithms gradually lose advantage when graph density increases. Figure 4.13(b) shows the cost ratio of each part of *OptFunc*. Obviously, as the graph density increases, a bottleneck occurs in the reduce part, which consumes more than 50% of the total time.

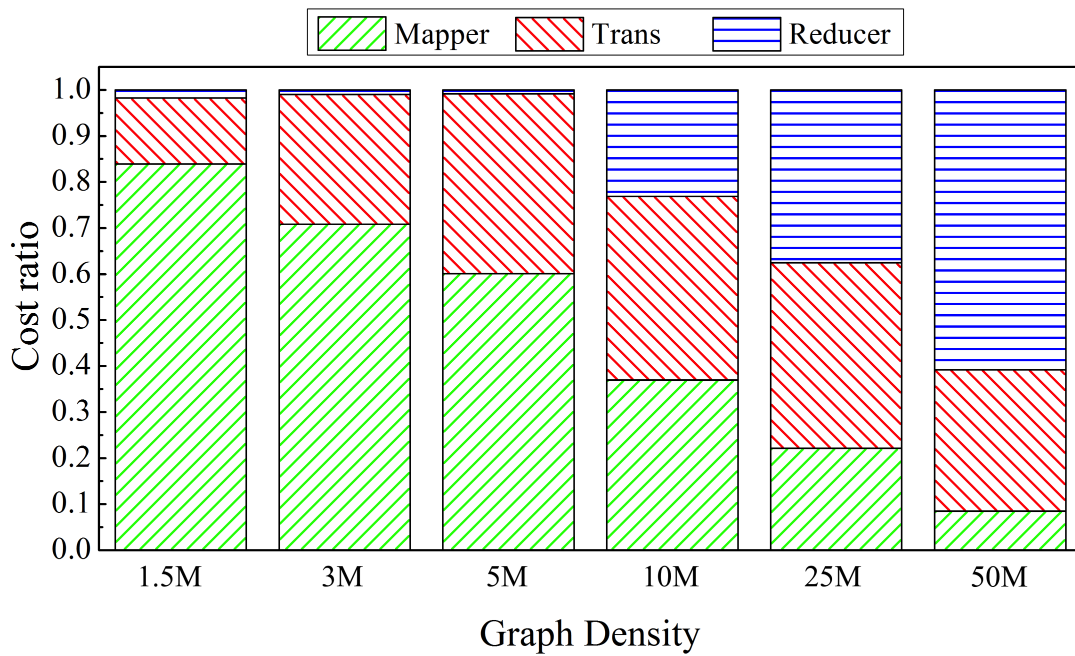
4.5 Related Work

Many approaches have been developed to evaluate RRQs on distributed graphs. They can be categorized into two types: message passing and partial evaluation. A regular path query (RPQ) that returns all the paths matching a regular path expression R can be used to answer RRQs. Abiteboul et al. [31] used message passing in the evaluation of RPQs on distributed graphs. Later, Stefanescu et al. [68] extended the algorithm proposed in [31] to weighted graphs. Shoaran et al. [69] accounted for process failures during the evaluation of RPQs, where algorithms can be resilient to any number of process failures. The use of partial evaluation to answer RRQs was proposed by Fan et al. [19]; they used Boolean formulas to describe reachability relationships between input and output nodes, and then these Boolean formulas were combined to build a dependency graph on the coordinator machine. Quyet et al. [70] proposed two algorithms to improve the algorithms proposed in [19]. They filtered and removed redundant nodes during the local evaluation and contracted the partial result to minimize the size of the data to be transferred. Our approach is also based on partial evaluation to improve the evaluation performance, but it is different from [70] by caching and reusing the same computation on each local site.

Our functional-based approach is motivated by the simultaneous finite automaton (SFA) proposed in [63] to solve the problem of regular expression matching in parallel. Holub et al. [71] introduced an algorithm for parallel execution of DFA, but its pre-computation of initial states makes it inefficient for general DFA. Memeti et



(a) Performance with various graph densities



(b) Standardized cost of *OptFunc* with various graph densities

Figure 4.13: How functional-based algorithms are affected by graph characteristics.

al. [72] improved the efficiency of the algorithm proposed in [71] by including an extra step to calculate all the states to estimate the initial states. The common idea of these approaches is that they split the input string into small chunks. By simulating automaton transitions, these chunks can be run in parallel. However, all these approaches are designed for regular expression matching on strings and lists. To the best of our knowledge, we are the first applying SFA to graph queries.

4.6 Discussion

We have shown that SFA is a good approach to deriving an efficient distributed algorithm for regular reachability queries on big graphs. We showed that the distributed algorithm based on SFA outperforms the state-of-the-art algorithm proposed in [19]. Therefore, now we discuss the potential to extend our algorithm to the phase *Mark*.

As mentioned at the beginning of this chapter, the phase *Mark* requires tracking a set of DFA states during the evaluation of an automaton at every reachable vertex. Our idea is that we first need to keep partial results for *every vertex* at each site during the partial computation, then once the global computation finished, send the global result back to each site instead of the client, and finally update partial results.

We sketch data structures for each step now. The partial result for each vertex should be a mapping from a set of input vertices to a set of SFA states, say $V_{in} \mapsto F_v$, denoting a set of SFA states yielded for paths from input vertices. Keeping a set of input vertices is necessary because it helps update the set of states. The global computation need some modifications. Instead of returning a boolean value, it returns a set of mappings $v \mapsto Q_v$, where v is an input or output vertex, Q_v is a set of DFA state.

Updating partial results at each vertex is convenient thanks to the properties of SFA. Assume that a vertex v has a partial result $V_{in} \mapsto F_v$, then we compute for v a set of DFA states r as follows.

$$r = \bigcup_{q \in Q,\ f \in F_v} f(q)$$

where,

$$Q = \bigcup_{v_{in} \in V_{in},\ v_{in} \mapsto Q_v} Q_v$$

5

SWRP Queries with Shortest Path Conditions

It is useful to extend the expressiveness of the select-where regular path queries to support shortest-path conditions. Let us consider a practical query on biological networks. Assume that we would like to find substances that are the closest to a substance typed "A" via a path including a third substance typed "B". With a little

extension, we can express this query in the form of SWRP queries as follows.

select
  select
    select $\$g_3$
    where $\{_*.\texttt{substance} : \$g_3\}$ **in** $\$g_2$
        $\$g_3$ **closest to** $\$g_1$ **via** $\$g_2$
  where $\{_*.\texttt{substance} : \$g_2\}$ **in** $\$g_1$
        $\texttt{Type.B}$ **in** $\$g_2$
where $\{_*.\texttt{substance} : \$g_1\}$ **in** $\$db$
        $\texttt{Type.A}$ **in** $\$g_1$

where, the extension condition "$\$g_3$ **closest to** $\$g_1$" says, for each graph $\$g_1$, let find a graph $\$g_3$ whose root connects to the root of $\$g_1$ via a shortest path. Combining with "**via** $\$g_2$", this shortest path must go through a root of graphs $\$g_2$. Without loss of generality, we assume that the shortest path is a path with the minimum number of edges, in other words, each edge has a weight of 1.

The above query requires computing two things. The first is to identify the roots of graphs $\$g_1$, $\$g_2$, $\$g_3$, which can be done by computing a marker graph, using the phase *Mark* of the normal SWRP query without the condition **closest to**. The second is to find the shortest path between the roots, which corresponds to find all vertices (roots of $\$g3$) connected from the root of $\$g_1$ by the shortest path whose vertex labels form a *category-based* regular expression "$C_1._*.C_2_*.C_3$", where C_1, C_2, C_3 are sets (categories) of roots of $\$g_1, \$g_2, \$g_3$, respectively. Queries using such category-based regular expressions are called *shortest regular category-path* queries.

In this chapter, we define shortest regular category-path queries and show that these queries can be efficiently answered by a sequence of single source shortest path searches. Hence, we can utilize different existing single source shortest path algorithms for different types of graphs to achieve the best performance.

5.1 Definition of SWRP Queries with Shortest Path Conditions

We formally define the extension of shortest path conditions for SWRP queries over a weighted graph (Definition 2.3). In particular, we add a new constructor to the condition part $P(\$g)$ in an SWRP query (defined in Chapter 3) as follows.

$$
\begin{aligned}
P(\$g) \ ::= \ & \textbf{isempty}(Q(\$g)) \mid R \ \textbf{in} \ \$g \\
\mid \ & !P(\$g) \mid P(\$g) \ \&\& \ P(\$g) \mid P(\$g) \parallel P(\$g) \\
\mid \ & \$g \ \textbf{closest to} \ \$g' \ [\ \textbf{via} \ C] \qquad \{\text{extension}\} \\
C \ ::= \ & \$g_1 \\
\mid \ & C, C \qquad \{\text{a sequence of graph variables}\} \\
\mid \ & C \mid C \qquad \{\text{OR condition}\}
\end{aligned}
$$

where $\$g'$ refers to a global graph variable in a SWRP query. The condition "$\$g$ **closest to** $\$g'$" says for each graph $\$g'$, let find a graph $\$g$ whose root connects to the root of $\$g'$ via a shortest path. The part "**via** $\$g_1$" says the shortest paths that satisfy the condition "$\$g$ **closest to** $\$g'$" must go though roots of graphs in C. If C is $\$g_1$, then the shortest paths must visit a root of $\$g_1$. "$C, C$" says the shortest paths must visit roots in the first C and then the ones in the second C. Finally, "$C \mid C$" says the shortest paths must visit roots in one of two Cs. Although we may allow multiple shortest path conditions in an SWRP query, to make our ideas clear, we only consider the case that there is only one shortest path condition in an SWRP query.

Moreover, we require that the roots of the graph $\$g$ and the graph $\$g'$ must be different. Otherwise, the query fails under the graph equivalence up to bisimulation. For example, two graphs in Figure 5.1 are equivalent up to bisimulation. The right graph is obtained by expanding the loop at the vertex 2 in the left graph. For every edge label l, the weight function $w(l)$ returns a value 1. Assume that the root of the graph $\$g'$ is the vertex 2, and that candidates for the root of the graph $\$g$ include 2 and 3. It is clear that, for the left graph, the vertex 2 must be the root of the graph $\$g$ because the shortest path $2 \xrightarrow{b} 2$ has the length of 1, shorter than the shortest path from 3 to 2 ($3 \xrightarrow{e} 4 \xrightarrow{d} 2$). While, for the right graph, the vertex 3 must be the root of the graph $\$g$ because the shortest path from 3 to 2, ($3 \xrightarrow{e} 4 \xrightarrow{d} 2$), is shorter than the one

Figure 5.1: Two equivalent graphs.

from 2 to 2, ($2 \xrightarrow{b} 5 \xrightarrow{c} 3 \xrightarrow{e} 4 \xrightarrow{d} 2$). Once the roots of the graph \$g and the graph \$g′ are disjoint, the expansion of loops does not affect the result of shortest path conditions.

5.2 Practical Shortest Regular Category-Path Queries

In this section, we formally define shortest regular category-path (SRCP) queries and its subclass P-SRCP queries, and demonstrate the expressiveness of P-SRCP queries via several typical examples.

5.2.1 SRCP Queries

SRCP query is a special case of the known shortest path query on weighted graphs (Definition 2.3).

Definition 5.1 *(Shortest Path) Let $G^w = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O}, w)$ be a weighted graph. Let $P_{s,t} = v_1 \xrightarrow{l_1} v_2 \xrightarrow{l_2} \dots \xrightarrow{l_{q-1}} v_q$ be any path in G^w from a vertex $s = v_1 \in \mathcal{V}$ to another vertex $t = v_q \in \mathcal{V}$, such that, for $1 \le i < q, (v_i, l_i, v_{i+1}) \in \mathcal{E}$. Let $cost(P_{s,t})$ be the cost of $P_{s,t}$ to represent $\sum_{1 \le i < q} w(l_i)$. A path $P'_{s,t}$ is called a* shortest path *from s to t if for all $P_{s,t}$ such that $P_{s,t} \in G$, we have $cost(P'_{s,t}) \le cost(P_{s,t})$. The* shortest path cost*, $cost(P'_{s,t})$, is denoted by $d(s,t)$.*

SRCP query is defined over vertex categories.

Definition 5.2 *(Category) Given a weighted graph G^w. A category C is a set of vertices in G, or $C \subseteq \mathcal{V}$. Two categories C_i and C_j are disjoint if $C_j \cap C_j = \varnothing$.*

Definition 5.3 *(Regular Expression on Category) The syntax for regular expression on Category is:*

$$R ::= \hat{C} \mid RR \mid R \text{ “}|\text{” } R \mid R\text{“}*\text{”}$$

Here $\hat{C}$ is to recognize a category C, i.e., a terminal symbol, RR denotes the concatenation, R “|” R denotes the alternation, and R“” denotes closure (Kleene star). As usual, we may write R^+ for $RR*$.*

Definition 5.4 *(Path Satisfaction) A path $P_{s,t}$ from s to t is said to satisfy a regular expression R over a set of categories if the concatenation of categories of the vertices in $P_{s,t}$ spells out R. Such a path is denoted by $P_{s,R,t}$.*

Definition 5.5 *(Shortest Regular Category-Path (SRCP) / SRCP Query) Given a weighted graph G^w, let $\{C_i \subseteq \mathcal{V} \mid 1 \leq i \leq k\}$ be a set of k disjoint categories of vertices in G^w, and R be a regular expression over C_is. An SRCP query is represented as a triple*

$$\langle s, t, R \rangle$$

where s and t denote the starting and ending vertices respectively.

A path $P^{min}_{s,R,t}$ is called an SRCP if it satisfies R, and for every path $P_{s,R,t}$ in G satisfying R, we have:

$$cost(P^{min}_{s,R,t}) \leq cost(P_{s,R,t}).$$

We refer $cost(P^{min}_{s,R,t})$ as $d^R(s,t)$.

Definition 5.6 *(SRCP problem) An SRCP problem is to find an SRCP for a given SRCP query.*

5.2.2 P-SRCP: a Practical Class of SRCP Queries

In general, SRCP queries are more difficult to solve than existing category-constrained shortest path queries. The difficulty comes from two constructors in the SRCP queries, one is the closure and the other is the alternation. In the case that each vertex belongs to only one category, SRCP queries correspond to regular simple path queries [73] whose evaluation are in general NP-hard. To provide an efficient and practical algorithm to

solve SRCP queries, we simplify them by restricting the use of the closure. We call such restricted queries *P-SRCP* queries.

Given an SRCP query

$$\langle s, t, R\rangle$$

where R is a regular expression, we show that we can simplify the SRCP problem by localizing the global closure. This is based on the following two observations. First, it is usually more practical to consider a path passing a vertex of category C (i.e., $_ * \hat{\text{C}}_*$, where _ denotes an arbitrary category) than a path *just* containing an *exactly single vertex* of category (i.e., $\hat{\text{C}}$). This would suggest us to regard $_ * \hat{\text{C}}_*$ as a primitive. Second, the concatenation of $_ * \hat{\text{C}}_*$ with a closure will cancel the effect of the closure. This means that the following two SRCP queries,

$$\langle s, t, _ * \hat{\text{C}}_ * R*\rangle$$
$$\langle s, t, R * _ * \hat{\text{C}}_*\rangle$$

will have the same effect as the query

$$\langle s, t, _ * \hat{\text{C}}_*\rangle$$

It is because the shortest path obtained from the last query should be the shortest path from the first two queries.

Given the above, we will simplify regular expressions to make the closure appear only in the form of $_ * \hat{\text{C}}_*$.

Definition 5.7 *(Simplified Regular Expression (SRE)) The syntax for SREs is:*

$$R ::= _*\hat{C}_* \mid RR \mid R \text{ ``}|\text{''} R$$

For simplicity, we use "C" as an abbreviation of "$_*\hat{C}_*$".

Definition 5.8 *(P-SRCP Query) A P-SRCP query is an SRCP query*

$$\langle s, t, R\rangle$$

where R is a simplified regular expression.

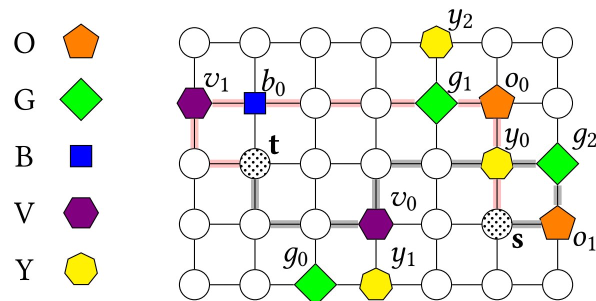


Figure 5.2: A P-SRCP query example $\langle s, t, \mathrm{O}\,(\mathrm{GY}\,|\,\mathrm{B})\,\mathrm{V}\rangle$. All edges have the same weight of 1. Two of the SRCPs are shaded in grey and pink. $d^R(s,t) = 9$.

In the rest of this chapter, we will focus on P-SRCP queries.

Figure 5.2 shows an example of P-SRCP query and paths satisfying the query $\langle s, t, \mathrm{O}\,(\mathrm{GY}\,|\,\mathrm{B})\,\mathrm{V}\rangle$. The input graph has five categories O, G, B, V, and Y. Vertices in the same category have the same shape and color. Because of alternations in the query, it is possible to have many optimal paths $P^{min}_{s,R,t}$ that have the same optimal cost $d^R(s,t)$. Also note that there is no requirement that two vertices in two different categories must be directly connected. For example, the optimal path $s \xrightarrow{1} o_1 \xrightarrow{1} g_2 \xrightarrow{1} y_0 \xrightarrow{1} \ldots \xrightarrow{1} v_0 \xrightarrow{1} \ldots \xrightarrow{1} t\rangle$ spells out the constraint "OGYV", however the vertex y_0 in Y connects to the vertex v_0 in V via two other vertices, although Y and V are contiguous in the constraint.

5.2.3 Expressiveness of P-SRCP

Although P-SRCP queries are restricted, they are powerful enough to express many interesting category-constrained shortest path queries including those with partial or total order constraints.

Generalized Shortest Path (GSP) Queries

This query is to find the shortest path from a starting point to a destination point, passing at least one point from each of a set of specified categories in a specified order [74]. A GSP is expressed in our P-SRCP query as follows.

$$\langle s, t, \mathrm{C}_1\mathrm{C}_2 \ldots \mathrm{C}_k\rangle$$

where s and t are the starting point and destination point, respectively.

Optimal Sequenced Route (OSR) Queries

An OSR query is to find the shortest path starting from a given point and passing through a number of categories in a particular order [75]. This query is different from the GSP query in the sense that the destination is not a point but a category. To express this query in our SRCP query, we create an artificial destination vertex t' in the input graph, and add edges with weight 0 from vertices in the last category of the order constraints to t'. The P-SRCP query is then as follows.

$$\langle s, t', \mathrm{C}_1\mathrm{C}_2 \ldots \mathrm{C}_k \rangle$$

Trip Planning Queries/Generalized Traveling Salesman Path Problem Queries (TPQ/GTSPP)

A trip planning query [76] or generalized traveling salesman path problem query [77] is to find the shortest path from a starting point to a destination point that passes through at least one point from each of a set of categories (there is no specific order specified in the query). A TPQ/GTSPP query with a set of k categories is written in our P-SRCP query as follows.

$$\langle s, t, R_1 \,|\, R_2 \,|\, \ldots \,|\, R_{k!} \rangle$$

where R_is ($i = 1 \ldots k!$) are permutations of the set $\{\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_k\}$. For example, R_1 is $\mathrm{C}_1\mathrm{C}_2 \ldots \mathrm{C}_k$, R_2 is $\mathrm{C}_2\mathrm{C}_1 \ldots \mathrm{C}_k$, and so on.

Optimal Route Queries (ORQ) with Arbitrary Order Constraints

This query is to find the shortest path that starts from a starting point and covers a user-specified set of categories (e.g. {gas station, museum, park, restaurant}) [78]. Moreover, users can specify partial order constraints between some specific categories of the set, e.g. a gas station must be visited before a restaurant, while other categories can be visited in an arbitrary order. Such order constraints are expressed in a *visit order graph*, in which each vertex is a category, an edge from a category C_i to a category C_j denotes that C_i must be visited before C_j.

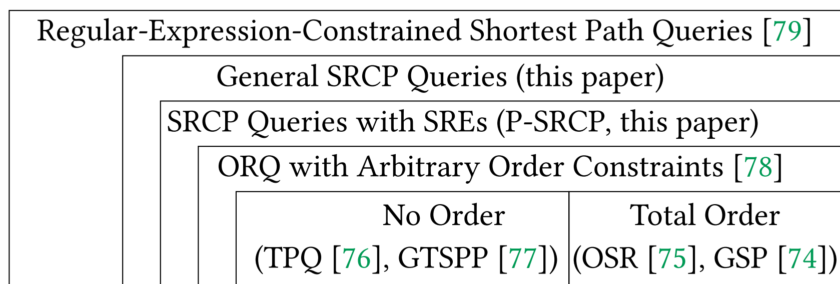


Figure 5.3: Relationship between SRCP queries and existing queries.

To express the optimal route queries with arbitrary order constraints in our P-SRCP query, there are two things needed to be done. First, we need to generate total order constraints from the visit order graph, then put them together in the form of P-SRCP queries by using alternation operators. A simple way to generate the total order constraints is first enumerating all permutations of the set of categories, and then filtering out permutations that do not satisfy constraints in the visit order graph. Second, we need to create an artificial destination vertex t' for the P-SRCP query, which is done by adding edges with weight 0 from vertices in all categories in the set of categories to t'. The P-SRCP query is finally as follows.

$$\langle s, t', R_1 \,|\, R_2 \,|\, \ldots \,|\, R_l \rangle$$

where R_is ($i = 1 \ldots l$) are total order constraints satisfying the visit order graph.

In summary, Figure 5.3 shows the relationships between our SRCP queries and other queries. The general SRCP query is a subset of the regular-expression-constrained shortest path query [79] in which its regular expression is defined over vertex labels. A practical subset of the general SRCP queries (P-SRCP), whose constraints are simplified regular expressions, is considered in this paper. Although it is limited, it covers all existing important problems such as queries with arbitrary order, total order, or no order constraints.

5.3 Derivation of an Efficient Algorithm for P-SRCP Queries

A naive approach for answering the P-SRCP query (s, t, R) is considering it as a combination of existing queries with total order constraints. To do that, we first generate all total order constraints R_i of the input SRE R. For example, consider an SRE "B(C | S)", we generate two equivalent constraints R_1 = "BC" and R_2 = "BS". Then we evaluate sub-queries $\langle s, t, R_i \rangle$, e.g. $\langle s, t, \mathrm{BS} \rangle$, $\langle s, t, \mathrm{BC} \rangle$, independently by efficient algorithms for total order constraints (e.g. algorithms for optimal sequenced route [75] or generalized shortest path [74]). Finally we take the minimum cost from results of each sub-queries. This approach is straightforward but inefficient due to many redundant computations between the evaluations of sub-queries, e.g. two sub-queries in the above example would share a computation for paths from s to the category B. Moreover, computations from the category B to the category C and S can be overlapped in terms of visited edges/vertices.

In this section, we propose a dynamic programming solution to solve the P-SRCP problem in an efficient way, in which we reduce the P-SRCP problem to a series of single source shortest path searches.

5.3.1 SREs as Directed Acyclic Graphs

It is well known that a regular expression can be expressed by a non-deterministic finite state automaton with ϵ-transitions (NFA-ϵ) [24]. However, the use of ϵ-transitions is not necessary due to the absence of closures in SREs of P-SRCP queries. Hence, we directly transform an SRE to an NFA without ϵ-transitions. This NFA is a directed acyclic graph (DAG) G_R that represents the structure of SREs in P-SRCP queries. Hereafter, we refer to vertices in a query graph as nodes, to distinguish them from vertices in the input graph. Nodes of G_R are object identifiers (OIDs) that are integers uniquely identifying a node, and edges of G_R are labeled by categories in R. Given an SRE R, the following function *gen_dag* generates a constraint graph G_R for R. Note

that, the generated graph is special in the sense that it has a source and a sink.

$$gen_dag\ R = \textbf{let}\ \langle sc, G_R, sk\rangle = rec\ R\ \textbf{in}\ dag$$

where

$$\begin{array}{rcl} rec\ C & = & \textbf{let}\ d = newEdge(C) \\ & & \textbf{in}\ \langle source(d), d, target(d)\rangle \\ rec\ (R_1R_2) & = & (rec\ R_1) \ominus (rec\ R_2) \\ rec\ (R_1 \,|\, R_2) & = & (rec\ R_1) \oplus (rec\ R_2) \\ \langle sc_1, G_{R_1}, sk_1\rangle \ominus \langle sc_2, G_{R_2}, sk_2\rangle & = & \langle sc_1, seq(G_{R_1}, G_{R_2}), sk_2\rangle \\ \langle sc_1, G_{R_1}, sk_1\rangle \oplus \langle sc_2, G_{R_2}, sk_2\rangle & = & \langle sc_1 \odot sc_2, merge(G_{R_1}, G_{R_2}), sk_1 \odot sk_2\rangle \end{array}$$

Given an SRE R, a recursive function *rec* computes a triple $\langle sc, G_R, sk\rangle$ where sc and sk are the source and sink node in the constraint graph G_R, respectively. For the terminal case C, we create a singleton graph G_C containing only one edge d labeled by the category C. The source node of G_C is the source node of d, and the sink node is the target node of d. The function *seq* is to construct a new constraint graph by first replacing sk_1 in G_{R_1} and sc_2 in G_{R_2} by a new OID $w = sk_1 \odot sc_2$, then unioning these two constraint graphs. The function *merge* is to construct a new constraint graph by first replacing sc_1 in G_{R_1} and sc_2 in G_{R_2} by a new oid $sc_{12} = sc_1 \odot sc_2$, and then replacing sk_1 in G_{R_2} and sk_2 in G_{R_2} by a new OID $sk_{12} = sk_1 \odot sk_2$, finally unioning these two constraint graphs.

To present the semantics of the whole query $Q = \langle s, t, R\rangle$, we introduce a *query graph* $G_Q = (V_Q, E_Q)$ that is constructed from the graph G_R by attaching an incoming edge labeled by $\{s\}$ to the source node of G_R, and an outgoing edge labeled by $\{t\}$ to the sink node of G_R. Figure 5.4 shows a query graph of the query $\langle s, t, \mathrm{O}\,(\mathrm{GY} \,|\, \mathrm{B})\,\mathrm{V}\rangle$, in which two sets $\{s\}$ and $\{t\}$ are called dummy categories (superscripts of edge labels will be explained later in the Sect. 5.3.2).

5.3.2 Dynamic Programming Formulation

Next, we formalize a dynamic programming strategy to answer P-SRCP queries by using their query graphs. Given a P-SRCP query $Q = \langle s, t, R\rangle$, we establish a DP table X in order to store values during computation. Each row in the table corresponds to a category on an edge of the graph G_Q. Therefore, the table X has $|E_Q|$ rows, and g

columns where g is the maximum size of categories in the query Q. "$X[i, j]$" is the value for the j-th vertex in the category at the row i. For simplicity, we add superscripts to categories in the query to denote their row indices in the table. For example, for this user-defined query $\langle s, t, \mathrm{O}\,(\mathrm{GY} \,|\, \mathrm{B})\,\mathrm{V}\rangle$, we have $\langle s^0, t^6, \mathrm{O}^1\,(\mathrm{G}^2\mathrm{Y}^3 \,|\, \mathrm{B}^4)\,\mathrm{V}^5\rangle$.

The DP table X is computed according to a topological sort of the query graph G_Q as follows. For each node u in the topological sort, we generate a *computation step* (CS), $in_u \to out_u$, where

$$\begin{aligned} in_u &= \{r \mid (v, l, u) \in E_Q, C^r \leftarrow l\} \\ out_u &= \{r \mid (u, l, w) \in E_Q, C^r \leftarrow l\}, \end{aligned}$$

computing values of the rows in the list out_u by using values in the rows in the list in_u. Computation steps form a dynamic programming formulation for the P-SRCP problem. Following is the computing formulation of the computation step $in_u \to out_u$.

$$\underset{i \in out_u}{X[i, j]} = \begin{cases} 0 & \text{If } i = 0 \\ \min\limits_{r \in in_u} \{ \min\limits_{0 \le l < |C^r|} \{X[r, l] + d(c_{r,l}, c_{i,j})\}\} & \text{If } i > 0 \end{cases}$$

where, C^r is the category corresponding to the r-th row in the DP table X, $c_{i,j}$ is the j-th vertex in the category C^i.

Lemma 5.9 *Value $X[i, j]$ in the DP table represents the optimal cost of the P-SRCP of the query $\langle s, c_{i,j}, R^i\rangle$ where R^i is the SRE corresponding to a subgraph of G_Q that includes all paths from the node just after the source node of G_Q to the source node of the edge labeled C^i.*

For example, consider the query $\langle s^0, t^6, O^1\,(G^2\,Y^3 \,|\, B^4)\,V^5\rangle$, its query graph is shown in Figure 5.4. The value $X[3, 1]$ will represent the optimal cost of the P-SRCP of the query $\langle s, y_1, R^3\rangle$ in which $R^3 = O^1 G^2$ corresponding to the subgraph having edges from the node 1 (the node just after the source node of G_Q) to the node 3 (the source node of the edge Y^3).

Proof. We prove this by induction on the sequence of computation steps $1 \le k \le N$, in which N is the number of computation steps (the number of nodes in G_Q).

For the base case, where $k = 1$, then this claim is trivially true, because $X[0, 0]$ is the optimal cost for the query $\langle s, c_{0,0}, R^0\rangle = \langle s, s, \{\}\rangle$ which has $d^R(s, s) = 0$.

For the induction step, $k > 1$. Let u be the node in G_Q that generates the k-th computation step. For a set of nodes V_u that are before the node u in the topological sort

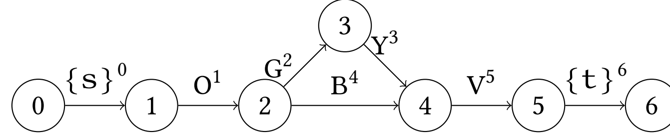


Figure 5.4: A query graph of the query $\langle s, t, \mathrm{O}\,(\mathrm{GY}\,|\,\mathrm{B})\,\mathrm{V}\rangle$. Integers in nodes are OIDs. Superscripts of categories are equivalent row indices in the DP table. The subgraph from the node 1 to the node 5 is a DAG graph corresponding to the SRE $\mathrm{O}\,(\mathrm{GY}\,|\,\mathrm{B})\,\mathrm{V}$.

of G_Q, let E_u be a set of edges related to nodes in V_u, and ps be a set of row indices of categories on the edges in E_u. Our induction hypothesis assumes that this claim holds true for all values $X[i, \bullet], i \in ps$. Let consider the $(k+1)$-th computation step generated by the node v in G_Q, that is $in_v \to out_v$, in which in_v and out_v is the set of row indices of all incoming and outgoing edges of the node v, respectively. It is clear that $in_v \subseteq ps$. Since each $c_{i,j}$ is a member of category C^i, $i \in out_v, j = 0 \ldots (|C^i| - 1)$, it suffices to find the shortest path cost from vertices $c_{r,l}$ in categories C^r ($r \in in_v, 0 \le l < |C^r|$) to $c_{i,j}$. It follows from our induction hypothesis that the value of $X[i, j]$ is computed by $\min_{r \in in_v} \{ \min_{0 \le l < |C^r|} \{X[r, l] + d(c_{r,l}, c_{i,j})\}\}$. □

Corollary 5.10 *Let m be the row index of the dummy category $\{t\}$. Value $X[m, 0]$ represents the cost of the P-SRCP of the query $\langle s, t, R\rangle$.*

Figure 5.5 shows a table containing costs during the computation of the query $\langle s^0, t^6, \mathrm{O}^1\,(\mathrm{G}^2\mathrm{Y}^3\,|\,\mathrm{B}^4)\,\mathrm{V}^5\rangle$. The order of computation steps is as follows.

1st step:	$[]$	$\to [0]$
2nd step:	$[0]$	$\to [1]$
3rd step:	$[1]$	$\to [2, 4]$
4th step:	$[2]$	$\to [3]$
5th step:	$[3, 4]$	$\to [5]$
6th step:	$[5]$	$\to [6]$
7th step:	$[6]$	$\to []$

The optimal cost $d^R(s, t) = 9$ of the query is stored at the last row (its row index is 6 that is the index of the dummy category $\{t\}$) of the table. Note that the first step is actually to initialize the value of $X[0, 0]$, and the last step does nothing.

	0	1	2
$\{s\}^0$	0		
O^1	1	2	
G^2	2	3	6
Y^3	3	7	4
B^4	6		
V^5	6	7	
$\{t\}^6$	9		

Figure 5.5: A table of optimal costs for the query in Figure 5.2. The first column contains names of categories. The first row contains indices of vertices in a category. Curved arrows on the left indicate computation steps and its orders.

5.3.3 Single Source Shortest Path Search

A computation step "$xs \rightarrow ys$" is to compute values in rows in the list ys by using rows in xs. Let U_{xs} be a union of categories corresponding to rows in xs and U_{ys} be a union of categories corresponding to rows in ys, then the step "$xs \rightarrow ys$" is equivalent to computing values for the vertices in category U_{ys} from values of the vertices in category U_{xs}. This can be done by using a many-to-many shortest path search from the vertices in U_{xs} to the vertices in U_{ys}. However, such a many-to-many search leads to many redundant computations due to repeatedly visiting the input graph. By creating a super-vertex s' and edges from s' to the vertices u in U_{xs} with weights being values of u in the DP table [74], we can efficiently compute values for the vertices in U_{ys} by using a single shortest path search from s' until all vertices in U_{ys} are settled (Assume that we use Dijkstra algorithm [80]).

Theorem 5.11 *Given a weighted graph $G^w = (\mathcal{V}, \mathcal{E})$ and a P-SRCP query $Q = \langle s, t, R \rangle$. Let $G_Q = (V_Q, E_Q)$ be the query graph of Q and T_{sssp} be the complexity of a single source shortest path search, the cost of our algorithm is $O(|V_Q| T_{sssp})$, and the space complexity of the algorithm is $O(|E_Q|)$.*

Corollary 5.12 *Given a weighted graph $G^w = (\mathcal{V}, \mathcal{E})$ and a P-SRCP query $Q = \langle s, t, R \rangle$.*

Let $G_Q = (V_Q, E_Q)$ be the query graph of Q. When a Dijkstra algorithm with a Fibonacci heap is used for SSSP searches [81], our algorithm answers Q in the time complexity of $O(|V_Q|(|\mathcal{E}| + |\mathcal{V}|log|\mathcal{V}|))$.

One advantage of this approach is that it is independent of the underlying SSSP search. Thus, we can use any fast SSSP algorithm to implement, such as Contraction Hierarchies technique [82, 74], Delta-Stepping [83], PHASE [84], etc. This is useful because we can apply different efficient SSSP algorithms for different kinds of graphs (road networks, social networks, biological networks, etc.)

5.3.4 Optimizations

Although the dynamic programming formulation can help solve the P-SRCP problem, there is a need in optimizing the P-SRCP query algorithm. First, the number of computation steps ($|V_Q|$) depends on user-defined queries. For example, two queries, $\langle s, t, (\mathrm{OGYV} \,|\, \mathrm{OBV})\rangle$ and $\langle s, t, \mathrm{O}\,(\mathrm{GY} \,|\, \mathrm{B})\,\mathrm{V}\rangle$, have the same meaning, but the former needs six computation steps and the latter needs five computation steps. Second, consider the Trip Planning Query that is to find the shortest path going through at least one point in each category of a given set of categories $\mathrm{C} = \{\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_k\}$. It can be presented in P-SRCP query as $\langle s, t, R_1 \,|\, R_2 \,|\, \ldots \,|\, R_{k!}\rangle$ where R_is are permutations of the set C, $i = 1 \ldots k!$. In this case, $|V_Q|$ will be $((k-1)k! + 2)$ which is not practical even for small k (e.g. $k = 5$. See Sect. 5.4 for more discussion). This section will discuss how to engineer an efficient algorithm for answering the P-SRCP query.

Time Complexity The problem of optimizing the time complexity is defined as finding a graph with the minimum number of nodes that generates exactly the same total order constraints as a given query graph. A query graph is a subclass of NFA [24] whose minimization is computationally hard, and cannot in general be solved in polynomial time. However, there exists a well-known algorithm for minimizing NFAs [24] that computes a minimal equivalent DFA with respect to the number of nodes, and consists of two steps: determinization and minimization. The determinization step is to compute a DFA from an NFA, then the DFA is minimized by the minimization step to get a minimal DFA. Ström [85] has proposed two simplified algorithms for the two steps determinization and minimization in the case of word graphs that represent a set

of hypotheses in speech recognition system. A word graph is a DAG with exactly one source node and one sink node. Here, we apply these algorithms to optimize query graphs which have a similar structure to word graphs.

Space Complexity For our approach, the number of rows in the DP table is equal to the number of edges in the query graph G_Q. Although the optimization of the number of nodes in G_Q also causes a decrease of the number of edges, this decrease is not remarkable. Moreover, because the number of elements in each row of the table is equal to the number of vertices in the equivalent category, the size of the DP table becomes large when the query contains a "long" total order constraints, leading to out-of-memory errors. Therefore we need to efficiently manage the DP table. One solution to managing the DP table is dynamically creating it. As discussed before, the DP table is constructed according to a topological sort of the query graph of a query. When considering a node in that topological order, incoming edges are used to compute values for rows corresponding to outgoing edges, and never used again. Thus, after each computation step, we do not need to store rows corresponding to incoming edges.

Query Structure Although two determinization and minimization optimizations result in a minimal query graph G_Q in terms of the number of nodes, for some user-defined queries, we can get a smaller graph G_Q by preprocessing the SREs in the user-defined queries. Consider P-SRCP queries in the following form,

$$\langle s, t, R_1 R_2 R_3 \,|\, R_2 \rangle,$$

where R_1, R_2, R_3 are arbitrary SREs. They have the same effect as the query

$$\langle s, t, R_2 \rangle.$$

Therefore, for this kind of queries, we can eliminate all SREs that contain another SREs. This can be generalized for P-SRCP queries having more alternatives.

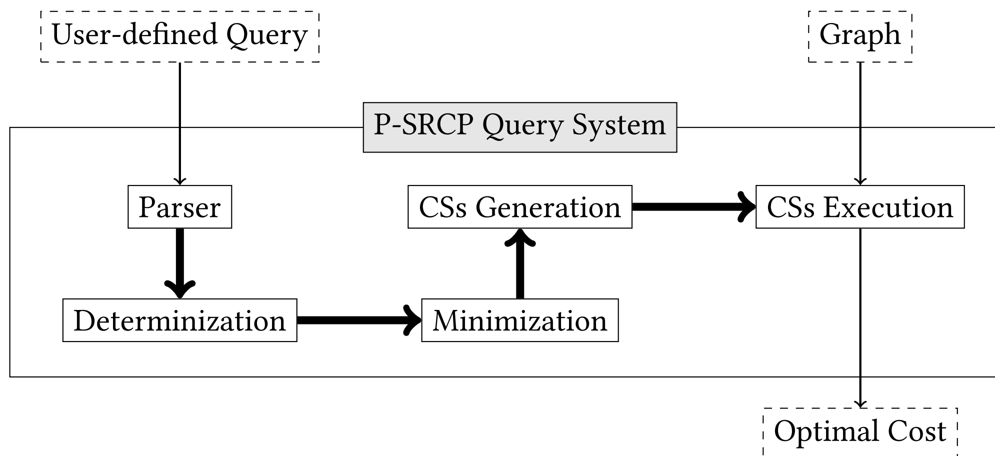


Figure 5.6: A testbed framework for SRCP queries.

5.4 Implementation and Experiments

5.4.1 Implementation

We implemented a testbed framework[1] to answer P-SRCP queries. An overview of our framework is showed in Figure 5.6. First, it takes a P-SRCP query as input, parses it to get a query graph, then optimizes the query graph by two steps "determinization" and "minimization". Next, it will generate a sequence of computation steps, in which each computation step is executed by a single source shortest path search. For simplicity, we just compute the optimal cost of the optimal paths satisfying the query. However, one can extract the optimal paths by tracing computation steps.

For an implementation of a single source shortest path, we used a fast algorithm proposed by Rice [74] that used Contraction Hierarchies. Contraction Hierarchies (CH) technique currently is one of the fastest speed-up technique for shortest path problem on road networks [86]. Its idea is preprocessing a graph by augmenting it with shortcuts so that shortest path costs are preserved. Shortcuts are then intensively used by a bidirectional Dijkstra algorithm to speed up the shortest path search on the augmented graph.

The following programs are implemented and used in our experiments to compare results of discussed algorithms.

[1] http://www.prg.nii.ac.jp/members/tungld/srcp-July2014.tar.gz

- **gsp**: the algorithm proposed by Rice [74] to answer the Generalized Shortest Path Query that uses total order constraints $\langle s, t, C_1 C_2 \ldots C_k \rangle$. We implemented the core of this algorithm (without some heuristic technique).
- **perm**: a naive algorithm, which was mentioned in the beginning of the Sect. 5.3, to answer the P-SRCP query by evaluating sub-queries for all total order constraints, then taking the minimal cost from the sub-queries. Sub-queries are evaluated by the **gsp** algorithm.
- **srcp-noopt**: our algorithm for the P-SRCP query without optimizations.
- **srcp-opt**: our algorithm for the P-SRCP query with optimizations. Two optimizations were implemented: the determinization and the minimization. To store a set of nodes, we used a standard class *std::set* which supports *equality comparison.* We used a class *std::unordered_map* as a hash table to store sets of nodes, which allows for fast access to the sets of nodes to determine their existing elements. The boost graph library [87] is used to implement the optimizations.

5.4.2 Experiments

All our experiments were run on a Macbook Pro machine that has a 2.6 GHz Intel Core i5, 8 GB 1600 MHz DDR3 memory, clang-503.0.40 (based on LLVM 3.4svn). Programs were compiled with optimization level 3. We used a library of contraction hierarchies written by Robert Geisberger [88] in C++ to create and access augmented graphs.

Experiments were performed with a graph of the Full USA road network, having $23,947,347$ vertices and $58,333,344$ edges. We borrowed the graph from the benchmarks of the 9th DIMACS implementation challenge [89]. It took about 25 minutes to create an augmented graph using the CH technique (this graph will be used as a common input graph for programs in our experiments).

Categories used in our queries are generated randomly and have the same number of vertices.

Influence of the Optimizations

We measure the performance of our algorithm for the trip planning queries which are expensive queries. The Trip Planning Query (TPQ) with k categories is written in the

P-SRCP queries as follows.

$$\langle s, t, R_1 \,|\, R_2 \,|\, \ldots \,|\, R_{k!} \rangle$$

where R_is are permutations of the set $\{C_1, C_2, \ldots, C_k\}$, $i = 1 \ldots k!$. For example, R_1 is $C_1C_2 \ldots C_k$, R_2 is $C_2C_1 \ldots C_k$, and so on.

First, we fix the number of categories being 5, and then change the size of categories. The naive solution that generates all permutations of categories requires 720 computation steps ($(5 + 1) * 5!$). Without optimizations, our algorithm generates 482 computation steps which is nearly half of that of the naive solution. By using optimizations, the number of computation steps reduces to 32 (2^5). Performance of algorithms is represented in Figure 5.7. It shows that the **srcp-opt** algorithm is quite scalable when the size of categories is increased.

Next, we will see the effect of the number of categories on the performance of the query. We fix the size of each category and change the number of categories in the query. As indicated in Figure 5.8, the running time of the **perm** algorithm is significantly increased, while the **srcp-opt** algorithm is quite stable. Although the **srcp-noopt** algorithm can reduce the number of computation steps twice, it still follows a factorial running time. This experiments show the importance of optimizations in our solution.

Influence of the Alternation Operators

To see the impact of alternation operators ($|$) for a given query on the performance of our optimal algorithm, we do experiments with queries which differ in the number of alternation operators, while the number of categories is the same. Starting with a query without alternatives, each time we insert one "$|$" operator to create a new query. In particular, we use the following queries.

$$Q1 = \langle s, t, C_1C_2C_3C_4C_5C_6C_7C_8 \rangle$$
$$Q2 = \langle s, t, (C_1 \,|\, C_2)C_3C_4C_5C_6C_7C_8 \rangle$$
$$Q3 = \langle s, t, (C_1 \,|\, C_2)(C_3 \,|\, C_4)C_5C_6C_7C_8 \rangle$$
$$Q4 = \langle s, t, (C_1 \,|\, C_2)(C_3 \,|\, C_4)(C_5 \,|\, C_6)C_7C_8 \rangle$$
$$Q5 = \langle s, t, (C_1 \,|\, C_2)(C_3 \,|\, C_4)(C_5 \,|\, C_6)(C_7 \,|\, C_8) \rangle$$

Figure 5.9 shows the result. It is interesting that when there are more options in the

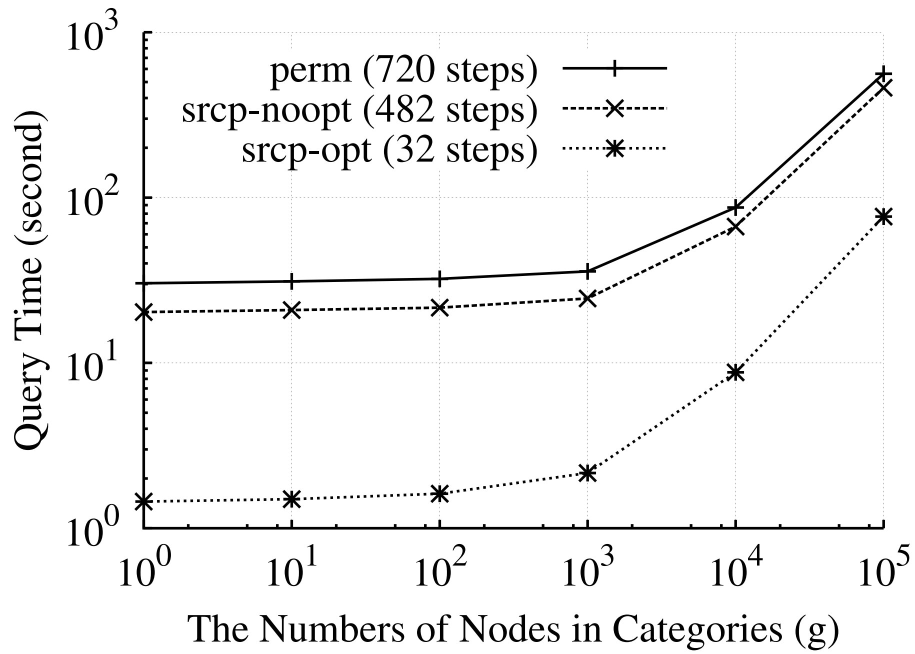


Figure 5.7: TPQ/GTSPP queries ($k = 5$).

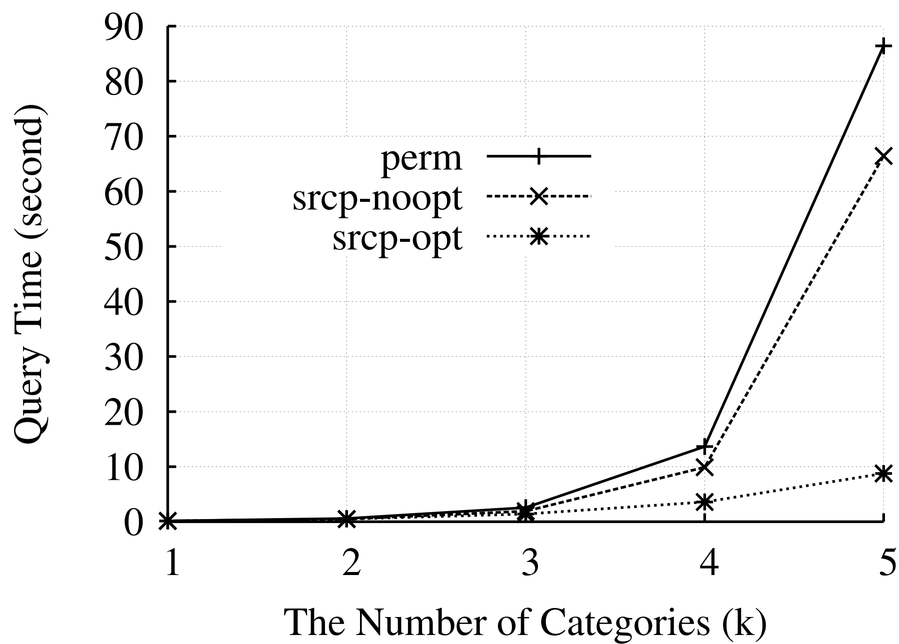


Figure 5.8: TPQ/GTSPP queries ($g = 10,000$).

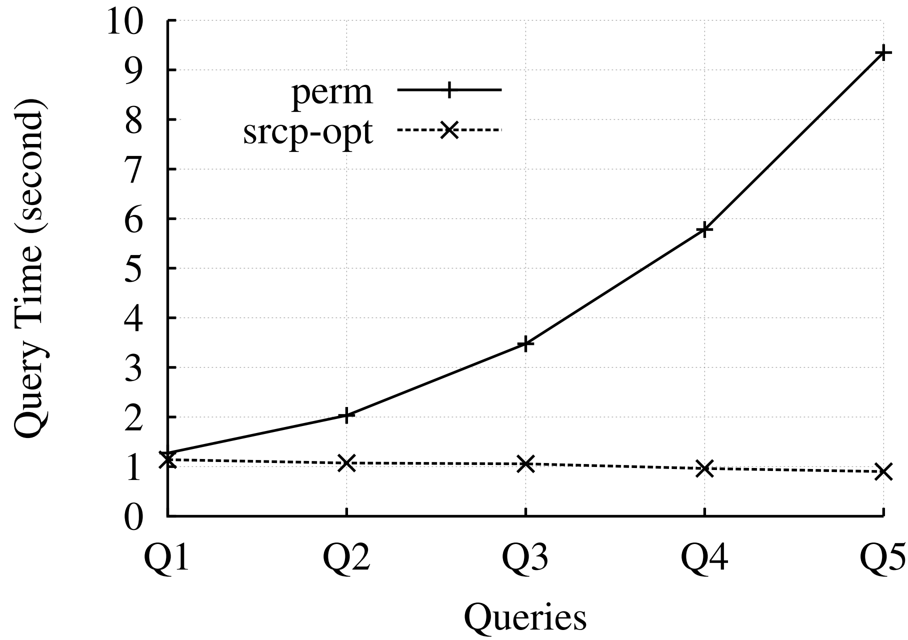


Figure 5.9: Queries with multiple options (k = 8, g = 10, 000).

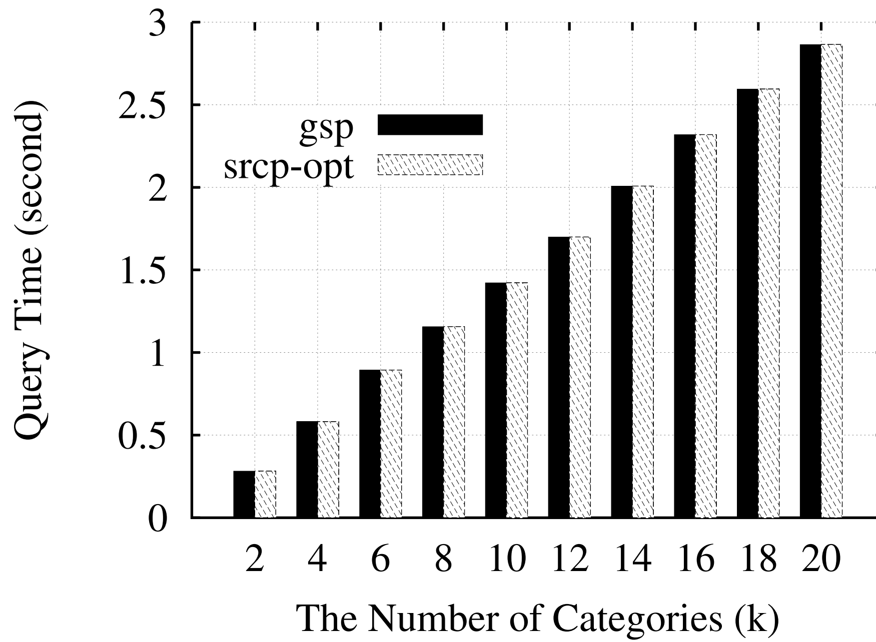


Figure 5.10: GSP queries (g = 10, 000).

Table 5.1: The number of computation steps in each query.

query	Q1	Q2	Q3	Q4	Q5
perm	9	16	28	48	80
srcp-opt	9	8	7	6	5

P-SRCP query, our algorithm becomes faster. Looking the Table 5.1 that shows the number of computation steps for each query, we see that the reason of such good performance is that the number of computation steps is reduced, leading to a decrease of the running time of our algorithm. Meanwhile, if we use the **perm** algorithm, then the number of computation steps will be dramatically increased because there are many total order constraints generated. This result also shows that our algorithm can reduce a large amount of redundant computation steps when alternation operators appear in the query.

Overhead of Optimizations

First, we measure the performance of our algorithm when answering the query GSP. We compare it to the algorithm proposed by Rice [74]. As can be seen in Figure 5.10, two algorithms have the same performance when the number of categories is increased. This is easy to understand because, for this query, our algorithm leads to the same dynamic programming table as that of **gsp** algorithm. Moreover, there is no improvement on the structure of the query when applying optimizations, thus the overhead is very small. However, for the trip planning queries, the overhead of optimizations, in particular, the overhead of the forward optimization, is expensive. Table 5.2 shows the running times of the determinization and minimization optimizations when the number of categories (k) is changed from 1 to 6. With small k $(1, 2, 3, 4)$, the minimization optimization takes less time than the determinization optimization. Nevertheless, the determinization optimization is more expensive with larger k $(5, 6)$. This is because, the determinization optimization has an exponential time complexity, while the minimization optimization takes a linearithmic time complexity. Furthermore, both optimizations highly depend on the way of storing sets of nodes during their computations. This effects the performance of determining whether a set already exists or not.

Table 5.2: Performance of optimizations.

k	1	2	3	4	5	6
determinization (ms)	0.026	0.033	0.069	0.290	2.566	53.697
minimization (ms)	0.018	0.034	0.084	0.302	1.580	10.872

5.5 Related Work

The category-constrained shortest path problem is a variant of regular-language-constrained shortest path queries in which constraints are on a set of vertices in a graph instead of individual vertices/edges. There are many solutions proposed to answer such queries. Each solution is for a specific kind of constraints over categories.

Trip Planning Queries [76] is the query that has no ordered constraints. The existence of multiple choices per category makes the problem difficult to solve. The complexity of the TPQ is NP-hard with respect to the number of categories. Several approximation algorithms are proposed. These algorithms are based on nearest neighbor searches. A feasible path is formed by iteratively visiting the nearest neighbor of the last vertices added to the path from all vertices in the categories that have not been visited yet. The second one is the minimum distance algorithm, a novel greedy algorithm, which results in a much better approximation bound. The algorithm chooses a set of vertices, one vertex per one category in the query. These vertices are chosen so that the total distance from the start vertex to it and from it to the end vertex is the minimum among vertices in the same categories. The algorithm then creates a path by following these vertices in a nearest neighbor order. Rice et al. [77] proposed an exact solution for the Generalized Traveling Salesman Path Problem Query (GTSPP) that is similar to TPQ. The algorithm is building a product graph of the original graph $G^w = (\mathcal{V}, \mathcal{E})$ and a covering graph built on the power set of the query's categories. Finding the answer of the GTSPP query is finding the shortest path in the product graph. The time complexity is $O(2^k(|\mathcal{E}| + |\mathcal{V}|k + |\mathcal{V}|log|\mathcal{V}|))$, in which k is the number of categories in the query. The algorithm is then improved by incorporating the graph preprocessing technique called Contraction Hierarchies (CH) [82], resulting in the time complexity of $O(2^k(m' + |\mathcal{V}|k))$, in which m' is the number of edges of the preprocessed graph. It is noted that the improved algorithm is slightly different from

the original algorithm, in which it executes a series of sweeping phases according to levels of an abstraction of the product graph, and highly depends on the CH technique.

In parallel to Li et al.'s work [76], Sharifzadeh et al [75] proposed the optimal sequenced route query (OSR) that is similar to TPQ but imposed a total order constraints over categories. In other words, OSR query is to find the shortest path starting from a given point and passing through a number of categories in a particular order. They proposed two algorithms to operate on the Euclidean distance. The first one is LORD, a light threshold-based iterative algorithm. First, LORD uses a greedy search to find an threshold (upper-bound) for the cost of the optimal path. The greedy search is a successive nearest neighbor search from the starting vertex to the last category. Then, the LORD finds the optimal path in the reverse order, from the last category to the starting vertex. During the finding, it updates the threshold value and uses it to prune vertices that cannot belong to the optimal path. The second algorithm is R-LORD, an extension of the LORD, that uses R-tree to efficiently examine the threshold values. However, both algorithms are impractical when applied to road networks where nearest neighbor searches are very expensive. Thus, another algorithm, progressive neighbor exploration (PNE), has been proposed in [75]. The idea of the PNE is to incrementally create the set of candidate paths. At each step it needs two nearest neighbor searches: one is to expand the current best candidate path, the other is to refine that path by replacing the last vertex in the path by a new vertex.

Sharifzadeh et al. [90] introduced a pre-processing approach for the OSR query by using additively weighted Voronoi diagrams. This approach is efficient and practical compared with R-LORD algorithm, however, one of the disadvantages is that it is not flexible when requiring fixed sequences among categories. Rice et al. [74] proposed another approach using Contraction Hierarchies technique and dynamic programming for the Generalized Shortest Path (GSP) query that was the same as the OSR query but having only one destination point. Its advantage is that it can be applied to any possible set of categories in a query. Our work is inspired by the idea of a dynamic programming formulation from Rice et al.'s work, and we extend their algorithm in two aspects. First, we allow multiple categories involved in a computation steps. Second, we introduce "jumping" computations in the DP table that compute an arbitrary row from an other arbitrary row in the table, allowing us to freely describe computation steps guided by a directed acyclic graph representation.

The one being close to our SRCP query is the optimal route queries with arbitrary order constraints proposed by Li et al. [78]. This query considers partial order constraints over categories, which are described by a *visit order graph*. Two algorithms have been proposed namely Backward search and Forward search. The backward search algorithm computes the optimal paths in reverse manner similar to R-LORD algorithm [75]. However, instead of loading vertices belonging to the last category, the backward search retrieves the set of candidate vertices that may be part of the optimal path, which belong to *any categories* contained in the visit order graph. The forward search is similar to a greedy algorithm. It also uses the backward search algorithm for backtracking process, eliminating some vertices that cannot be a part of the optimal path. Both algorithms have the time complexity of $O(N^2 \cdot 2^k)$, in which k is the number of categories in the visit order graph and N is the total number of vertices in the data set (road networks or spatial databases).

5.6 Summary

In this chapter, we have introduced a general SRCP query for finding optimal paths constrained by categories. The purpose is to help extend the SWRP queries to support shortest-path conditions as discussed at the beginning of the chapter. We have found a reasonable subset of the general SRCP queries that uses simplified regular expressions as constraints. Even though this subset is simplified, it covers all of the existing category-constrained shortest path queries and has efficient implementation. We have proposed a dynamic programming formulation to solve the subset of SRCP queries in which the queries are reduced to a sequence of single source shortest path searches on the input graph. This result is important because we can use any fast single source shortest path search even with preprocessing to speed up the query. By exploiting a directed acyclic graph representation of a query, we can easily derive an efficient algorithm for answering the query.

6
Conclusion

6.1 Summary of the Dissertation

Writing programs in distributed programming models is often non-trivial. In this dissertation, we have proposed a graph querying framework on top of Pregel to ease the burden of users in writing parallel programs to process big graphs, where users just write their queries in the form of "**select** . . . **where** . . ." and regular expressions. Our framework is a combination of a solid foundation of structural recursion on graphs and scalable graph processing models such as Pregel or MapReduce.

In the first part of this dissertation, we have presented our framework in detail. We have proposed an efficient evaluation for select-where regular path queries which do not include graph conditions. Our solution is a combination of recursive semantics and bulk semantics of structural recursion, which significantly reduces the size of intermediate graphs. As for queries with graph conditions, we have proposed a solution to reduce them to two queries without conditions, and one specific iterative parallel algorithm. Experimental results show that our solutions can produce efficient Pregel

programs that are scalable to big graphs.

In the second part, we have proposed a functional-based approach to exploit the parallelism of the phase *Mark* that takes much time during the evaluation of select-where regular path queries. This phase is close to the regular reachability queries. We have shown that the functional-based approach is very effective to speedup the regular reachability queries, where it reduces a large amount of computation during a local computation and minimize communication data. We have discussed how to extend the algorithm for regular reachability queries to the phase *Mark*.

In the remaining part, we have focused on extending the expressiveness of select-where regular path queries to support shortest-path conditions. We have shown that this extension requires to solve an additional query that is the shortest regular category-path query. By using a dynamic programming formulation, we have shown that a shortest regular category-path query can be efficiently answered by a sequence of single source shortest path searches. This is useful because we can utilize fast single source shortest path algorithms that are optimized for different graphs (road networks, social networks, biological networks) and environments (shared or distributed memory).

6.2 Future Work

There are two categories of work needed to be done in the future.

6.2.1 Supporting More Queries

Supporting *Cartesian product, GroupBy* and *Join* queries For such queries, their specifications in structural recursion need to be able to join graphs based on two edge variables that are parameters of two different structural recursive functions. It is not clear how to transform such specifications to existing specifications supported so far, or how to translate them to programs in distributed programming models.

Problems on graphs up to isomorphism The graph data model for select-where regular path queries in this dissertation assumes set semantics where the duplication of data is eliminated. For example, assume a query retrieves all family names of people in a community. If there are one million people, but only ten different family names,

then our query returns only ten elements. It is shown that we can extend structural recursion on graphs to bag semantics [91]. By that way, it has potential to solve problems on graphs up to isomorphism, say, counting triangles.

6.2.2 Making the Framework More Efficient

Graph partitioning strategies It is often the case that graph algorithms are affected by graph partitioning strategies. In our framework, each computation phase transforms a graph into a new graph. Hence, it might not obtain the best performance with a single partitioning strategy for the whole framework. It is necessary to do a comprehensive study on the affection of partitioning strategies on phases in the framework.

ϵ-edges elimination Another problem is related to an efficient algorithm to eliminate ϵ-edges. Afrati et al. [92] proposes an efficient distributed algorithm to compute transitive closure on clusters. It would be interesting to integrate that algorithm into our framework. Checking graph conditions without eliminating ϵ-edges may be quickly performed by random walks on graphs [93, 94, 7], it is however non-trivial to handle the general case where nested queries are allowed.

Bibliography

[1] Peter T. Wood. Query Languages for Graph Databases. *ACM SIGMOD Record*, 41(1):50, 2012.

[2] Mary Fernández, Daniela Florescu, Alon Levy, and Dan Suciu. Declarative Specification of Web Sites with Strudel. *VLDBJ*, 9(1):38–55, 2000.

[3] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: a Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDBJ*, 9(1):76, 2000.

[4] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language, March 2013. W3C Recommendation.

[5] Katja Losemann and Wim Martens. The Complexity of Evaluating Path Expressions in SPARQL. In *Proc. of PODS '12*, pages 101–112, 2012.

[6] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying Graph Databases with XPath. In *Proc. of ICDT'13*, pages 129–140, 2013.

[7] Royi Ronen and Oded Shmueli. SoQL: A Language for Querying and Creating Data in Social Networks. In *Proc. of ICDE'09*, pages 1595–1602, 2009.

[8] Anton Dries, Siegfried Nijssen, and Luc De Raedt. BiQL: a Query Language for Analyzing Information Networks. In *Proc. of Bisociative Knowledge Discovery*, pages 147–165. Springer, 2012.

[9] Ulf Leser. A Query Language for Biological Networks. *Bioinformatics*, 21(2):33–39, 2005.

[10] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. *SIGKDD Explor. Newsl.*, 1(2):12–23, January 2000.

[11] David Konopnicki and Oded Shmueli. W3QS: A Query System for the World-Wide Web. In *Proc. of VLDB'95*, pages 54–65, 1995.

[12] A.O. Mendelzon, G.A. Mihaila, and T. Milo. Querying the World Wide Web. In *Proc. of the Fourth International Conference on Parallel and Distributed Information Systems*, pages 80–91, Dec 1996.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[14] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. Scalable Big Graph Processing in MapReduce. In *Proc. of SIGMOD'14*, pages 827–838, 2014.

[15] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proc. of SIGMOD'10*, pages 135–146, 2010.

[16] UnQL+, 2015. http://www.biglab.org/demodoc/unqlplus/.

[17] Semih Salihoglu and Jennifer Widom. Optimizing Graph Algorithms on Pregel-like Systems. *Proc. VLDB Endow.*, 7(7):577–588, March 2014.

[18] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *Proc. VLDB Endow.*, 7(14):1821–1832, October 2014.

[19] Wenfei Fan, Xin Wang, and Yinghui Wu. Performance guarantees for distributed reachability queries. *Proc. VLDB Endow.*, 5(11):1304–1316, 2012.

[20] Maurizio Nolé and Carlo Sartiani. Processing Regular Path Queries on Giraph. In *Proc. of EDBT/ICDT Workshops*, pages 37–40, 2014.

[21] Peter Buneman. Semistructured Data. In *Proc. of PODS'97*, pages 117–121, 1997.

[22] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding Structure to Unstructured Data. In *Proc. of ICDT'97*, pages 336–350. 1997.

[23] Mehmed Kantardzic. *Data mining: concepts, models, methods, and algorithms.* John Wiley & Sons, 2011.

[24] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[25] Apache Hadoop, 2015. https://hadoop.apache.org/.

[26] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[27] Zhenjiang Hu, Hideya Iwasaki, and Masato Takechi. Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms. *ACM Trans. Program. Lang. Syst.*, 19(3):444–461, May 1997.

[28] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. A Compositional Framework for Developing Parallel Programs on Two-dimensional Arrays. *Int. J. Parallel Program.*, 35(6):615–658, December 2007.

[29] Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. Systematic derivation of tree contraction algorithms. *Parallel Processing Letters*, 15(03):321–336, 2005.

[30] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *Proc. of InfoScale'06*, 2006.

[31] Serge Abiteboul and Victor Vianu. Regular Path Queries with Constraints. In *Proc. of PODS'97*, pages 122–133, 1997.

[32] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. *ACM SIGMOD Record*, 25(2), June 1996.

[33] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, Keisuke Nakano, and Isao Sasano. Marker-directed optimization of uncal graph transformations. In *Proc. of the 21st International Conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'11, 2012.

[34] L. D. Tung. Pregel meets uncal: A systematic framework for transforming big graphs. In *Proc. of 31st IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 250–254, April 2015.

[35] Apache Giraph, 2015. http://giraph.apache.org/.

[36] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proc. of SSDBM'13*, pages 1–12, 2013.

[37] Pregel+, 2015. http://www.cse.cuhk.edu.hk/pregelplus/.

[38] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *Proc. of GRADES'13*, pages 1–6, 2013.

[39] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proc. of EuroSys'13*, pages 169–182, 2013.

[40] FastUtil Library, 2015. http://fastutil.di.unimi.it/.

[41] Citation Network Datasets, 2015. http://arnetminer.org/billboard/citation.

[42] Youtube Datasets, 2015. http://netsg.cs.sfu.ca/youtubedata/.

[43] Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, 2015. http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[44] Amazon Product Co-purchasing, 2015. http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[45] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei

Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proc. of SIGMOD'15*, pages 1383–1394, 2015.

[46] Amazon Instance Types, 2015. https://aws.amazon.com/ec2/instance-types/.

[47] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. *Proc. VLDB Endow.*, 7(12):1047–1058, 2014.

[48] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.*, 8(3):281–292, 2014.

[49] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[50] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proc. of OSDI'12*, pages 17–30, 2012.

[51] Christian Krause, Matthias Tichy, and Holger Giese. Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In *Proc. of Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 325–339. 2014.

[52] Semih Salihoglu and Jennifer Widom. HelP: High-level Primitives For Large-Scale Graph Processing. In *Proc. of GRADES'14*, pages 1–6, 2014.

[53] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *Proc. of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'14, pages 208–218, 2014.

[54] Eric L. Goodman and Dirk Grunwald. Using vertex-centric programming platforms to implement sparql queries on large graphs. In *Proc. of IA3'14*, pages 25–32, 2014.

[55] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In *Proc. of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC'10, pages 1–6, 2010.

[56] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. From SPARQL to MapReduce: The Journey Using a Nested Triplegroup Algebra. *Proc. VLDB Endow*, 4(12):1426–1429, 2011.

[57] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Georg Lausen. Rdfpath: Path query processing on large graphs with mapreduce. In *The Semantic Web: ESWC 2011 Workshops*, pages 50–64. Springer, 2012.

[58] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with Regular Expression Patterns (for Querying RDF). *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):57–73, 2009.

[59] Katja Losemann and Wim Martens. The Complexity of Evaluating Path Expressions in SPARQL. In *Proc. of PODS'12*, pages 101–112. ACM, 2012.

[60] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A Navigational Language for RDF. *Web Semant.*, 8(4):255–270, November 2010.

[61] Jinsoo Lee, Romans Kasperovics, Wook-Shin Han, Jeong-Hoon Lee, Min Soo Kim, and Hune Cho. An Efficient Algorithm for Updating Regular Expression Indexes in RDF Databases. *International Journal of Data Mining and Bioinformatics*, 11(2):205–222, 2015.

[62] Henrik Björklund, Wim Martens, and Thomas Timm. Efficient Incremental Evaluation of Succinct Regular Expressions. In *Proc. of CIKM'15*, pages 1541–1550, 2015.

[63] Ryoma Sin'ya, Kiminori Matsuzaki, and Masataka Sassa. Simultaneous Finite Automata: An Efficient Data-Parallel Model for Regular Expression Matching. In *Proc. of ICPP'13*, pages 220–229, 2013.

[64] Dan Suciu. Distributed Query Evaluation on Semistructured Data. *TODS*, 27(1):1–62, 2002.

[65] MEME Dataset, 2015. https://snap.stanford.edu/data/memetracker9.html.

[66] Internet dataset, 2015. http://data.caida.org/datasets/passive/passive-oc48/.

[67] Graph Tools, 2015. http://projects.skewed.de/graph-tool/.

[68] Dan Stefanescu and Alex Thomo. Enhanced Regular Path Queries on Semistructured Databases. In *Proc. of Current Trends in Database Technology–EDBT 2006*, pages 700–711. Springer, 2006.

[69] Maryam Shoaran and Alex Thomo. Fault-tolerant Computation of Distributed Regular Path Queries. *Theoretical Computer Science*, 1(410):62–77, 2009.

[70] Quyet Nguyen-Van, Le-Duc Tung, and Zhenjiang Hu. Minimizing Data Transfers for Regular Reachability Queries on Distributed Graphs. In *Proc. of SoICT'13*, pages 325–334. ACM, 2013.

[71] Jan Holub and Stanislav Štekr. On parallel implementations of deterministic finite automata. In *Proc. of CIAA'09*, pages 54–64, 2009.

[72] Suejb Memeti and Sabri Pllana. PaREM: A Novel Approach for Parallel Regular Expression Matching. In *Proc. of the IEEE 17th International Conference on Computational Science and Engineering (CSE)*, pages 690–697. IEEE, 2014.

[73] A. O. Mendelzon and P. T. Wood. Finding Regular Simple Paths in Graph Databases. In *Proc. of VLDB'89*, pages 185–193, 1989.

[74] Michael N. Rice and Vassilis J. Tsotras. Engineering Generalized Shortest Path Queries. In *Proc. of ICDE'13*, pages 949–960, 2013.

[75] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. The Optimal Sequenced Route Query. *VLDBJ*, 17(4):765–787, July 2008.

[76] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On Trip Planning Queries in Spatial Databases. In *Proc. of SSTD'05*, pages 273–290, 2005.

[77] Michael N. Rice and Vassilis J. Tsotras. Exact Graph Search Algorithms for Generalized Traveling Salesman Path Problems. In *Proc. of SEA'12*, pages 344–355, 2012.

[78] Jing Li, Yin Yang, and Nikos Mamoulis. Optimal Route Queries with Arbitrary Order Constraints. *TKDE'13*, 25(5):1097–1110, May 2013.

[79] Chris Barrett, Riko Jacob, and Madhav Marathe. Formal-Language-Constrained Path Problems. *SIAM J. Comput.*, 30(3):809–837, May 2000.

[80] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[81] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[82] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proc. of WEA'08*, pages 319–333, 2008.

[83] Ulrich Meyer and Peter Sanders. Delta-Stepping: A Parallel Single Source Shortest Path Algorithm. In *Proc. of ESA'98*, pages 393–404, 1998.

[84] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. Phast: Hardware-accelerated shortest path trees. In *Proc. of IPDPS'11*, pages 921–931, 2011.

[85] Nikko Ström. *Automatic Continuous Speech Recognition with Rapid Speaker Adaptation for Human/Machine Interaction.* PhD thesis, KTH, Stockholm, 1997.

[86] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Muller-Hannemann, Thomas Pajor, Peter Sanders, Dorathea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. Technical Report MSR-TR-2014-4, 2014.

[87] Boost Graph Library, 2015. http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/.

[88] Construction Hierarchies Library, 2015. http://algo2.iti.kit.edu/source/contraction-hierarchies-20090221.tgz.

[89] The 9th DIMACS Implementation Challenge, 2015. http://www.dis.uniroma1.it/challenge9/.

[90] Mehdi Sharifzadeh and Cyrus Shahabi. Processing Optimal Sequenced Route Queries Using Voronoi Diagrams. *Geoinformatica*, 12(4):411–433, December 2008.

[91] Kazuyuki Asada, Soichiro Hidaka, Hiroyuki Kato, Zhenjiang Hu, and Keisuke Nakano. A Parameterized Graph Transformation Calculus for Finite Graphs with Monadic Branches. In *Proc. of PPDP'13*, pages 73–84. ACM, 2013.

[92] Foto N. Afrati and Jeffrey D. Ullman. Transitive Closure and Recursive Datalog Implemented on Clusters. In *Proc. of EDBT'12*, 2012.

[93] Wolfgang Woess. *Random Walks on Infinite Graphs and Groups*, volume 138. Cambridge University Press, 2000.

[94] László Lovász. Random Walks on Graphs: A Survey. *Combinatorics, Paul Erdos is Eighty*, 2(1):1–46, 1993.

List of Published Works

International Journals

1. Le-Duc Tung, Zhenjiang Hu. *Towards Systematic Parallelization of Graph Transformations over Pregel*, International Journal of Parallel Programming (IJPP), pp. 1–20, Springer (First online: 28 March 2016).

Refereed Papers (International Conferences and Workshops)

1. Le-Duc Tung, Zhenjiang Hu, *Towards Systematic Parallelization of Graph Transformations over Pregel*, The 8th International Symposium on High-level Parallel Programming and Applications (HLPP 2015), 20 pages, Pisa, Italy, July 2-3, 2015.

2. Le-Duc Tung, Zhenjiang Hu, *Pregel meets UnCAL: a Systematic Framework for Transforming Big Graphs*, The 31st International Conference on Data Engineering (ICDE2015), Ph.D. Symposium, pp. 250-254, Seoul, Korea, April 13-17, 2015.

3. Le-Duc Tung, Nguyen-Van Quyet, Zhenjiang Hu, *Efficient Query Evaluation on Distributed Graphs with Hadoop Environment*, 4th International Symposium on Information and Communication Technology (SoICT2013), pp. 311—319, Da Nang, Vietnam, December 5-6, 2013.

4. Nguyen-Van Quyet, Le-Duc Tung, Zhenjiang Hu *Minimizing Data Transfers for Regular Reachability Queries on Distributed Graphs*, 4th International Symposium

on Information and Communication Technology (SoICT2013), pp. 325-334, Da Nang, Vietnam, December 5-6, 2013.

Domestic Conference and Workshop Papers

1. Le-Duc Tung, Chong Li, Xiaodong Meng, Zhenjiang Hu, *Processing UnQL Graph Queries with Pregel*, 日本ソフトウェア科学会第32回大会 (JSSST), 2015.
2. Le-Duc Tung, Kento Emoto, Zhenjiang Hu, *Shortest Regular Category-Path Queries*, Technical Report, GRACE center, National Institute of Informatics, no. GRACE-TR-2014-03, August, 2014.
3. Le-Duc Tung, Kento Emoto, Zhenjiang Hu, *Engineering Shortest Regular Category-Path Queries*, 日本ソフトウェア科学会第31回大会 (JSSST) , 2014.
4. Andres Pardo, Le-Duc Tung, Zhenjiang Hu, *Towards a MapReduce Implementation for Distributed Query Evaluation on Labeled Graphs*, 第15回プログラミングおよびプログラミング言語ワークショップ（PPL2013), Poster.
5. Le-Duc Tung, Nguyen-Van Quyet, Zhenjiang Hu, *Efficient Query Evaluation on Distributed Graphs with Hadoop Environment*, 日本ソフトウェア科学会第30回大会(JSSST), 2013