UNIVERSITY OF CALIFORNIA
RIVERSIDE

Reuse Techniques for Efficiently Evaluating a Sequence of Iterative Graph Queries

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xiaolin Jiang

September 2023

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Zizhong Chen
Dr. Zhijia Zhao

The Dissertation of Xiaolin Jiang is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Rajiv Gupta. I am truly fortunate to have had the opportunity to work under your mentorship, not only because of his dedicated attitude to research, but also because of his ability to bring the best out of everyone. He introduced me to the world of research, trained me into a skilled graph researcher.

I'm grateful to Prof. Zhijia Zhao and Prof. Nael Abu-Ghazaleh for the collaboration and guidance in refining my research. Further, I would like to thank my dissertation committee member Prof. Zizhong Chen for feedback and support.

I appreciate the insights and discussions from our research group members: Abbas Mazloumi, Arash Alavi, Chaitanya Ananda, Chengshuo Xu, Gurneet Kaur, Hongbo Li, Mahbod Afarin, Rui Yang, Xiaofan Sun, and Xizhe Yin.

This work was supported in part by National Science Foundation Grants CCF-2002554, CCF-2028714, CCF-1813173, and CCF-1524852 to the University of California Riverside.

Finally, I want to thank my cousin, the first person in my family to complete the PhD journey and always shows me care. I also want to thank my friend Yaqing Wang, for constantly encouragement to move forward, daily standups, and countless deep conversations. Your emotional support gave me strength to overcome challenges during pandemic times.

To my grandparents.

ABSTRACT OF THE DISSERTATION

Reuse Techniques for Efficiently Evaluating a Sequence of Iterative Graph Queries

by

Xiaolin Jiang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2023
Dr. Rajiv Gupta, Chairperson

Graph analytics is employed in many domains (e.g., social networks, web graphs) to uncover insights by analyzing high volumes of connected data. Real world graphs are often large and iterative graph analytics requires repeated passes over the graph till the algorithm converges to a stable solution. As a result, iterative graph queries are expensive to evaluate. Thus, there has been a great deal of interest in developing scalable and efficient graph analytics systems for various platforms (e.g., GPUs, multicore servers, clusters). While much of the research has focused on developing algorithms for efficiently evaluating a single global graph query, in practice we may be faced with a sequence of queries. This research develops two platform independent complimentary *reuse based* approaches, value reuse and graph structure reuse, that collect information from executing a small number of queries and then use this information to speedup the evaluation of all future queries.

The first approach is based on the observation that, due to their global nature, vertex specific graph queries present an opportunity for sharing work across queries. To take advantage of this opportunity, we have developed the VRGQ framework that accelerates the

evaluation of a stream of queries via coarse-grained *value reuse*. In particular, the results of queries for a small set of source vertices are reused to speedup the evaluation of all future queries. We present a two step algorithm that in its first step initializes the query result based upon value reuse and then in the second step iteratively evaluates the query to convergence. The reused results for a small number of queries are held in a *reuse table*. Our experiments with best reuse configurations on four power law graphs and thousands of graph queries of five kinds yielded average speedups of 143×, 13.2×, 6.89×, 1.43×, and 1.18×. We have also extended the above approach to evaluation of queries over streaming graphs and simultaneous evaluation of a batch of graph queries.

The second approach is based upon graph structure reuse. We have developed a new technique for identifying a proxy graph named the Core Graph (CG) that is not only small, it also produces highly precise results. A CG is a subgraph of the larger input graph that contains all vertices but on average contains only 10.7% of edges and yet produces precise results for 94.5–99.9% vertices in the graph. The identification of such an effective CG is based on our key new insight, namely, a small subset of *critical edges* are responsible for determining the converged results of nearly all the vertices across different queries. We develop algorithms for identifying a CG and demonstrate its benefits by improving the performance of three systems: Subway for GPU-based graph processing, GridGraph for out-of-core disk-based processing, and Ligra for in-memory graph processing. In our experiments with six kinds of graph queries and four large input graphs, we achieved speedups of up to 13.62× in GridGraph and 4.97× in Ligra.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph analytics is employed in many domains (e.g., social networks, web graphs) to uncover insights by analyzing high volumes of connected data. Real world graphs are often large (e.g., Twitter - TT has 2 billion edges and 52.6 million vertices) and iterative graph analytics requires repeated passes over the graph till the algorithm converges to a stable solution. As a result, in practice, iterative graph analytics workloads are highly data- and compute-intensive. Therefore, there has been a great deal of interest in developing scalable and efficient graph analytics systems such as Pregel [50], GraphLab [46], PowerGraph [20], Galois [57], GraphChi [41], Ligra [73], ASPIRE [85, 86] and others.

While the performance of graph analytics has improved greatly due to advances introduced in aforementioned systems, much of this research has focussed on developing highly parallel algorithms for solving a single iterative graph analytic query. For example, SSSP(s) query computes shortest paths from a single source s to all other vertices in the graph. However, in practice the query evaluation system may need to respond to multiple

queries for different source vertices. The queries may be generated by a single user or multiple users. In this work we develop two general and complimentary approaches, VRGQ and Core Graph, aimed at efficiently evaluating a sequence of *vertex specific queries* received from users for different source vertices of a large graph. For example, for SSSP algorithm, we may be faced with the following stream of queries: SSSP(s_1); SSSP(s_2); $\cdots\cdots$ SSSP(s_n).

1.1 Dissertation Overview

In this thesis, we investigated methods for caching graph queries values and graph structures to accelerate iterative graph queries. Initially, we reused queries values in a shared-memory architecture for static graphs to optimize the performance of subsequent queries, and adapted this technique in streaming graphs. Following this, we identified and labeled critical edges, consolidating them into a proxy graph to facilitate evaluation of future queries efficiently on multiple systems.

1.1.1 Value Reuse

While much of the research on graph analytics over large power-law graphs has focused on developing algorithms for evaluating a single global graph query, in practice there's a potential to encounter continuous query streams. In this work, our primary focus is to store important query value information and reuse their values. Also, this technique is applied to streaming graphs, further combined with batched query evaluation techniques.

Reuse for Static Graphs

Based on the observation that vertex specific graph queries present an opportunity for sharing work across queries. Different queries typically traverse the majority of the graph and thus present an opportunity for *reuse* across multiple queries. For example, the same shortest subpaths may contribute to solutions of many queries and hence *reusable* across them. To take advantage of this opportunity, the VRGQ framework has been developed to accelerate the evaluation of a stream of queries via coarse-grained *value reuse*.

Reuse for Streaming Graphs

In many real-world application scenarios, a stream of updates are continuously applied to the graph, often in batches for better efficiency, known as the streaming graph scenario. There exists a fundamental limitation in the existing design. Except pre-selected standing queries, an expensive full evaluation is required for arbitrary queries. So we propose a principled way to generalizing the incremental graph processing such that vertex-specific queries without their a priori knowledge may also benefit from incremental processing. Value Reuse can connect the evaluation of a vertex-specific query to the results of another query evaluation, providing solutions to the same type of query in static graphs.

Reuse for Batched Evaluation of Queries

Batching queries in graph processing systems enhances performance by enabling parallel processing and reducing overheads in query evaluation. Batched evaluation of queries benefits from Value Reuse due to the intrinsic data sharing that occurs between different queries within the batch. SimGQ is a shared memory system that optimizes

simultaneous evaluation of a group of vertex queries using the results of the shared queries in a shared table. MultiLyra is a distributed framework incorporated Reuse technique as an optimization where results from earlier batches of queries are used to accelerate the execution of later batches of queries.

1.1.2 Graph Structure Reuse

Real-world graphs are characterized by their irregular and large nature. Significant overheads are incurred due to movement of graph across the memory hierarchy and repeated propagation of values over the edges in the graph. These overheads are exacerbated due to the iterative nature of graph analytics. Thus, it remains a significant challenge for scalable graph analytics systems across various computing architectures such as GPUs, multicore servers, and clusters. We have developed a novel platform independent technique for identifying a proxy graph named the Core Graph (CG) that is not only small and also produces highly precise results.

Reuse for In-Memory Evaluation System

Graph structure reuse in shared-memory system eliminates the need for repetitive computations. Also, leveraging CoreGraph improved memory performance by storing edge-lists such that Core Graph edges first stored first and then the remaining edges. We further refined the Ligra system to employ two phase computation. In the initial phase, almost all vertices propagated on CoreGraph with precise values. Following this, the second phase usually completes the computation after just one iteration.

Reuse for Out-of-Core System

In out-of-core systems, data loading costs are predominant. We have extended GridGraph to load and iterate on Core Graph first which reduces the need to continually fetch and process irregular edges from disk. For GridGraph, the first phase of computation is performed in-memory after loading the CG from disk and then the second phase performs partition-based processing. This policy generally leads to fewer iterations. Also during an iteration blocks with no active edges may arise more frequently and hence their fetch from disk will be skipped due to the selective scheduling optimization in GridGraph.

Combining Value and Graph Structure Reuse

The second phase processing on full graph is expensive because we need to add all activated vertices into frontier again. We use value reuse idea to identify precise values and make the frontier smaller. Starting from all vertices whose values are impacted in the first phase, the second phase resumes propagation of values over the full graph to obtain precise results for all vertices. Since most results are computed precisely in the first phase efficiently using the small CG, the work performed during the second phase is greatly reduced giving significant speedups.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents VRGQ, the system for value reuse based evaluation of iterative graph queries. Chapter 3 describes how value reuse can be leveraged to quickly respond to arbitrary queries in the streaming

graph scenario and simultaneously evaluate a batch of queries. Chapter 4 presents the Core Graph, which forms the basis of graph structure reuse based query evaluation. Chapter 5 discusses how Core Graph can be exploited to speedup multiple existing graph processing systems including out-of-core query evaluation system and an in-memory query evaluation system. Chapter 6 concludes the thesis and discusses future work.

Chapter 2

VRGQ: Exploiting Value Reuse

While much of the research on graph analytics over large power-law graphs has focused on developing algorithms for evaluating a single global graph query, in practice we may be faced with a stream of queries. We observe that, due to their global nature, vertex specific graph queries present an opportunity for sharing work across queries. To take advantage of this opportunity, we have developed the VRGQ framework that accelerates the evaluation of a stream of queries via coarse-grained *value reuse*. In particular, the results of queries for a small set of source vertices are reused to speedup all future queries. We present a two step algorithm that in its first step initializes the query result based upon value reuse and then in the second step iteratively evaluates the query to convergence. The reused results for a small number of queries are held in a *reuse table*. Our experiments with best reuse configurations on four power law graphs and thousands of graph queries of five kinds yielded average speedups of 143×, 13.2×, 6.89×, 1.43×, and 1.18×.

2.1 Our Approach

We observe that different queries typically traverse the majority of the graph and thus present an opportunity for *reuse* across multiple queries. For example, the same shortest subpaths may contribute to solutions of many queries and hence *reusable* across them. Our approach for reuse is as follows. Given an input graph and type of vertex query, we *precompute* the results of queries for a small number of source vertices and save them in a table for coarse-grained value reuse to optimize the evaluation of all future queries. Note that an iterative algorithm updates vertex property values of active vertices in each iteration driving them towards their final stable solution. When all vertex values become stable, the algorithm terminates. Our proposed *reuse strategy* is designed to update property values of all vertices in a single reuse step such that a good number of vertex values arrive at their final *stable* solutions and thus the active vertex sets of subsequent iterations are greatly reduced.

In the development of VRGQ we consider the following factors. First, because the reuse step incurs significant cost as it updates property values of *all* vertices, to limit its cost reuse is performed only once during the evaluation of a query by both the presented algorithms. Second, to maximize the benefits of reuse, reuse should be performed as early as possible and therefore we develop an algorithm that performs reuse right at the start. In particular, we have developed the 2Step algorithm where the Step 1 of the algorithm safely initializes the values of all vertices with the benefit of the precomputed results of other source vertex queries and then Step 2 simply iterates till the algorithm converges. Experiments with four power law graphs show that 2Step delivers varying amounts of speedups across

queries of different kinds. In particular, with best reuse configurations, for thousands of graph queries of five kinds 2Step yielded average speedups of 143×, 13.2×, 6.89×, 1.43×, and 1.18×.

The remainder of the paper is organized as follows. Section 2 develops the 2Step reuse based query evaluation algorithm that is at the heart of VRGQ. Section 3 presents the evaluation of the 2Step algorithm. Section 4 describes our approach for selecting reuse source vertices whose queries are evaluated to populate the reuse table. Additional related work is discussed in Section 5 and concluding remarks are given in Section 6.

2.2 VRGQ: Two Step Value Reuse Algorithm

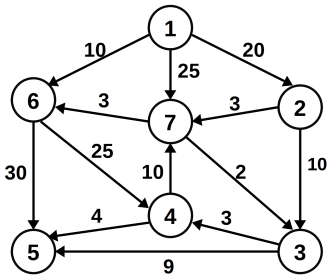
Let us assume that given a source vertex s, a query $Q(s)$ computes the desired property value val for every destination vertex d with respect to s, also denoted as $val(d \backslash s)$. A straightforward and intuitive approach for enabling reuse of the computed values is to explicitly transform the graph to express the computed property values so that they become available for reuse to future queries.

The above approach is illustrated by an example shown in Figure 2.1 using single-source shortest-path or SSSP queries. Figure 2.1(a) shows a small graph followed by the evaluation of query SSSP(2) using a push-style strictly synchronous algorithm. The shortest path values for all vertices are first initialized to 0 for source vertex 2 and ∞ for all other destination vertices. Following each iteration the updated shortest path values are shown along with list of active vertices, that is, vertices whose values have changed and thus must play a role in further propagation. Note that we do not add vertex 5 to the set of active

vertices because it has no outgoing edges. As we can see, in all it takes four iterations for the shortest path values to converge.

Now let us assume that SSSP(7) had been previously evaluated and we would like to reuse its result to accelerate the computation of query SSSP(2). In Figure 2.1(b) we observe that the results of SSSP(7) indicate that $val(4 \setminus 7)$ is 5 and $val(5 \setminus 7)$ is 9. As shown, this information can be incorporated explicitly into the graph by introducing the two additional edges, or *shortcuts*, one from 7 to 4 and the other from 7 to 5 with weights of 5 and 9 respectively. Next the results of computing SSSP(2) on the transformed are shown. We observe that the shortcuts cause faster propagation and hence convergence is achieved in one less iteration than before. While in the above example there is a reduction in number of total iterations, even if the number of iterations remains the same, reuse can lead to reduction in total amount of work performed due to reduction in number of active vertices.

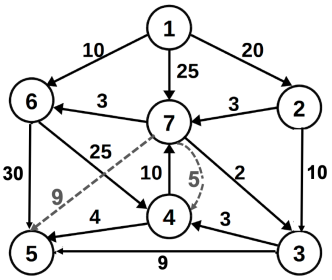
While the above approach is straightforward and applicable to other graph algorithms where query evaluation originates at a source vertex, it also has the following limitations. First, the approach based upon graph transformation may cause the shortcuts introduced to be processed multiple times increasing the overhead of reuse and thus cutting into its benefits. This limitation is illustrated in Figure 2.1(c) where evaluation of SSSP(1) on the transformed graph causes shortcuts to be processed twice as vertex 7 is activated twice. At first activation $val(7 \setminus 1)$ is 25 which is not stable and during second activation $val(7 \setminus 1)$ is 23 which is the final stable value. Second, introduction of shortcuts increases the number of edges in the graph. Hence there is an increase in the size of memory footprint



Iter#	1	2	3	4	5	6	7	Active Vertices
0	∞	0	∞	∞	∞	∞	∞	{2}
1	∞	0	10	∞	∞	∞	3	{3,7}
2	∞	0	5	13	19	6	3	{3,4,6}
3	∞	0	5	8	14	6	3	{4}
4	∞	0	5	8	12	6	3	{}

(a) Evaluation of SSSP(2) on the Original Graph.

Source	1	2	3	4	5	6	7
7	∞	∞	2	5	9	3	0



Iter#	1	2	3	4	5	6	7	Active Vertices
0	∞	0	∞	∞	∞	∞	∞	{2}
1	∞	0	10	∞	∞	∞	3	{3,7}
2	∞	0	5	8	12	6	3	{3,4,6}
3	∞	0	5	8	12	6	3	{}

(b) Evaluation of SSSP(2) on Graph Transformed Via Shortcuts to Reuse Results of SSSP(7).

Iter#	1	2	3	4	5	6	7	Active Vertices
0	0	∞	∞	∞	∞	∞	∞	{1}
1	0	20	∞	∞	∞	10	25	{2,6,**7**}
2	0	20	**27**	**30**	**34**	10	23	{3,4,**7**}
3	0	20	**25**	**28**	**32**	10	23	{3,4}
4	0	20	25	28	32	10	23	{}

(c) Evaluation of SSSP(1) on Graph Transformed Via Shortcuts.

Figure 2.1: Value Reuse via Graph Transformation.

as well as number of irregular memory accesses.

To overcome the above limitations we developed the 2Step algorithm for evaluating a query $Q(s)$. At the start of the computation, the Step 1 of the algorithm initializes the values of vertices using the results for another query, say $Q(r)$. By performing reuse exactly once right at the start, the benefits of reuse are maximized and its overhead is minimized. Then, in Step 2 the algorithm iterates applying conventional updates to all vertices till the algorithm converges. Instead of transforming the graph by adding shortcuts, in this approach we store results of evaluating a small number of queries in a *reuse table* and select a suitable $Q(r)$ for reuse. By using a reuse table, the footprint of the graph is not increased and its locality of graph accesses is not worsened. Moreover accesses of values from the reuse table exhibit good spatial locality.

The reuse table is designed to contain both the forward and backward results of query $Q(r)$. This ensures that when $Q(s)$ reuses results of query $Q(r)$, it makes use of stable value of $val(r \setminus s)$ which maximizes the stable values produced via reuse. When reuse table contains results of multiple queries for different source vertices, to limit the cost of reuse 2Step selectively reuses results of a small subset of most promising source vertices in the reuse table. Thus, 2Step, while controlling the cost of reuse, maximizes production of stable values.

Forward-Backward Reuse Table Our objective is to perform reuse first, i.e. apply reuse updates and then run the original iterative algorithm to completion. Moreover, we want to ensure maximal production of stable values. We observe that both these objectives can be met if we enhance the information contained in the reuse table such that it contains

both *forward* and *backward* information for a given reuse vertex r as described below:

- FwdTable [**r**] [d] ($\forall\ d \in$ AllVertices) represents the property values (e.g., shortest path) computed for query Q(**r**) on graph *Graph*.
- BwdTable [s] [**r**] ($\forall\ s \in$ AllVertices) represents the property values (e.g., shortest path) computed for query Q(**r**) on *edge-reversed* graph $Graph^R$ – which is obtained by reversing the direction of each edge in the original *Graph*.

Now let us see how the precomputed values can be used right away at the start of evaluation of query Q (**v**). Given a vertex d, its value can be initialized using the precomputed results of a reuse source vertex r in the table as follows:

$$\text{reuseFunc}\ (\ d, \text{BwdTable}[\mathbf{v}][r], \text{FwdTable}[r][d]\ )$$

The reuseFunc function for five algorithms is given in Table 2.1. For example, for SSSP,

$$\text{BwdTable}[\mathbf{v}][r] + \text{FwdTable}[r][d]$$

is the shortest path from **v** to d via r and thus it is the best estimate we can obtain for *d.value* by using the results for vertex r in the reuse table. If we reuse results for multiple r vertices in the reuse table then we find the shortest paths via each of these vertices and then take the minimum across all the computed shortest paths to initialize *d.value*. *Because the* BwdTable *contains stable values, reuse produces maximal number of stable values upon completion of the reuse step.*

Let us reconsider the computation of SSSP(1) whose evaluation was shown earlier in Figure 2.1(c). Now let us apply 2Step to this query; however, now the reuse table contains

both forward and backward information for vertex 7 as in Figure 2.2. The evaluation now involves reuse followed by iterations. Following reuse, only two iterations are required for termination. In contrast earlier it took an extra iteration.

Source		1	2	3	4	5	6	7
7	FWD	∞	∞	2	5	9	3	0
	BWD	23	3	13	10	∞	35	0

Iter#		1	2	3	4	5	6	7	Active
Init.		0	∞	∞	∞	∞	∞	∞	{1}
Reuse		$d \in \{3,4,5,6,7\}$ BwdTable[7][1] + FwdTable[7][d]							
		0	∞	25	28	32	26	23	{1}
P	1	0	20	25	28	32	10	23	{2,6}
	2	0	20	25	28	32	10	23	{}

Figure 2.2: Computing SSSP(1) via 2Step Algorithm by Reusing results of SSSP(7).

Push-style 2Step Reuse Algorithm In Algorithm 1 we summarize our 2Step algorithm. As we can see, in Step 1 (lines 4-9) of evaluating the query for a given source vertex s, we exploit the precomputed results for ReusableVertices whose queries were precomputed and used to populate the reuse table (BwdTable, FwdTable). The algorithm first carries out the reuse step and sets the values of all destination vertices by reusing n most promising reusable source vertices for which precomputed query results are stored in the reuse table. To find the n most promising vertices we use function Prioritize which looks at values in the BwdTable and prioritizes vertices in the increasing or decreasing order (depending upon algorithm characteristic) of BwdTable$[r][s]$ values where r is a reusable source vertex whose results were precomputed and stored in the reuse table and s is the source vertex

Algorithm 1 Backward-Forward Reuse Algorithm.

```
1: function 2Step ( Q ( s ), BwdTable, FwdTable, n )
2:     ▷ Initialize Active Vertex Set and Vertex Values
3:     Active ← Initialize ( Q ( s ) )
4:     ▷ Step 1: Reuse
5:     Q ← Prioritize( s )
6:     ▷ Select reuse vertices and perform Reuse
7:     for all r ∈ first n vertices in Q do
8:         Reuse ( s, r )
9:     end for
10:    ▷ Step 2: Iterate
11:    while Active ≠ φ do
12:        Active ← Process ( Active )
13:    end while
14: end function
15:
16: function Priortize ( s )
17:    ▷ Ordering Vertices Used to Populate Reuse Table
18:    Build Priority Queue Q by inserting all
19:    vertices r ∈ ReusableVertices such that
20:    they are sorted in increasing order of
21:    BwdTable[r][s] values.
22:    return Q
23: end function
```

Algorithm 2 Backward-Forward Reuse Algorithm.

```
1: function REUSE ( s, r )
2:   ▷ Only Reuse Valid Values
3:   if bwdTable[s][r] ≠ initialValue then
4:     for d ∈ AllVertices do
5:       ▷ Only Reuse Valid Values
6:       if fwdTable[r][d] ≠ initialValue then
7:         ▷ Perform Reuse Update of d
8:         reuseFunc ( d, bwdTable[s][r], fwdtable[r][d]) )
9:       end if
10:    end for
11:  end if
12: end function
13: function Process ( Active )
14:  newActive ← φ
15:  for all v ∈ Active do
16:    for each e ∈ Graph.outEdges(v) do ▷ Apply Conventional Update to e.dest
17:      changed ← edgeFunc (e)
18:      if changed then ▷ Update newActive Set
19:        newActive ← newActive ∪ {e.dest}
20:      end if
21:    end forall
22:  end forall
23:  return newActive
24: end function
```

for which the query $Q(s)$ is being evaluated. The details of function REUSE show how it uses REUSEFUNC (lines 32-35) to perform reuse updates of all the vertex values so that remainder of the iterative algorithm does not have to start from the initialization values of all vertices but rather better values computed by the reuse step. Once the reuse step has been completed, then the iterative computation is completed (see Step 2, lines 10-13) by applying conventional updates to all vertices using the Process function (lines 41-54).

Table 2.1: Functions for Reuse Updates for Four Algorithms.

$d.value \leftarrow$ VRFUNC (d, BWDTABLE$[s][r]$, FWDTABLE$[r][d]$)
SSWP(s): $d.value \leftarrow$ $max(d.value, min(\text{BWDTABLE}[s][r], \text{FWDTABLE}[r][d]))$
Viterbi(s): $d.value \leftarrow$ $max(d.value, \text{BWDTABLE}[s][r] * \text{FWDTABLE}[r][d])$
SSSP(s): $d.value \leftarrow$ $min(d.value, \text{BWDTABLE}[s][r] + \text{FWDTABLE}[r][d])$
BFS(s): $d.value \leftarrow$ $min(d.value, \text{BWDTABLE}[s][r] + \text{FWDTABLE}[r][d])$
SSNP(s): $d.value \leftarrow$ $min(d.value, max(\text{BWDTABLE}[s][r], \text{FWDTABLE}[r][d]))$

2.3 Experimental Evaluation of VRGQ

Next we evaluate VRGQ that is based upon the presented 2Step algorithm and report the speedups achieved, reduction in number of active edges processed, and extent to which stable values are produced.

We implemented our algorithms using Ligra [73] that provides a shared memory abstraction for vertex algorithms which is particularly good for graph traversal. Graph algorithms used include – Single Source Widest Path (SSWP), Viterbi [43], Single Source Shortest Path (SSSP), Breadth First Search (BFS), and Single Source Narrowest Path (SSNP). Ex-

periments were performed on a 64 core (8 sockets × 8 cores) machine with AMD Opteron 2.3 GHz processor 6376, 512 GB memory, and running CentOS Linux release 7.4.1708.

Table 2.2: Input graphs used in experiments.

Graphs	#Edges	#Vertices
Twitter (TT) [7]	2.0B	52.6M
Twitter (TTW) [40]	1.5B	41.7M
LiveJournal (LJ) [4]	69M	4.8M
PokeC (PK) [78]	31M	1.6M

We use four directed and edge-weighted power-law input graphs with relatively small diameter that are listed in Table 2.2 – TT, TTW, LJ, PK. We use the default weight generation tool provided by Ligra. Ligra generated weights range from 1 to the $log(n) + 1$ (where, $n = |vertices|$). For the four graphs we tested, $log(n)$ ranged from 20 to 25. We also varied the upper bound of the range to 64 and 128, but the results were similar.

For each input graph, we generated 20,000 queries. Table 2.3 characterizes the coverage of queries and reuse table that are used in our evaluation. The queries used were for source vertices that are H_{min} to H_{max} hops from the reuse source vertices in the reuse table. The reuse table populated with results of only 5 source vertices allows nearly all possible queries to take advantage of its contents for reuse (ALL ranging from 81.9% to 96.1%). The hops considered account for nearly all of these queries (REUSABLE ranging from 81.7% to 95.8%). For each input graph we used 20,000 queries spread across the four different hop values considered. For 80% of vertices, Hops ranged from 1 to 4 or 2 to 5 for four graphs. We randomly picked 5K corresponding to each hop. Thus, the selected queries

Table 2.3: Coverage Characteristics of Reuse Table with 5 ReusableVertices.

	H_{min}	H_{max}	ALL → REUSABLE	#Queries
TT	1	4	81.9% → 81.7%	20K (5K/hop)
TTW	1	4	96.1% → 95.8%	20K (5K/hop)
LJ	2	5	87.2% → 83.2%	20K (5K/hop)
PK	2	5	86.8% → 85.1%	20K (5K/hop)

Table 2.4: NoReuse: Average execution times in *Seconds* across 20,000 queries.

G	SSWP	Viterbi	SSSP	BFS	SSNP
TT	7.04s	9.28s	8.41s	0.42s	18.03s
TTW	3.10s	3.91s	3.62s	0.28s	11.02s
LJ	0.21s	0.29s	0.22s	0.04s	0.58s
PK	0.07s	0.14s	0.09s	0.02s	0.09s

maximize diversity in terms of number of hops between the source vertex and the reusable vertices.

Finally, we use multiple reuse table configurations of the form n *out of* m in these experiments, where m is the number of sources vertices whose full results are stored in the reuse table and n is the most promising subset number of these that are actually exploited during reuse. The configurations used are 5 *out of* 20; 2 *out of* 10; and 1 *out of* 5.

Speedups Table 2.5 presents the speedups achieved by 2Step for the three reuse table configurations for all the input graphs and algorithms considered. The baseline running times without reuse (NoReuse) that are used in computing speedups are given in Table 2.4. For power law graphs, the results for 20,000 queries are separated according to the 5,000 queries each for the four hop values considered. The average speedups obtained for power-law graphs and table configurations are substantial, 43× to 143× for SSWP, 6.05× to 6.89× for Viterbi, 1.33× to 1.43× for SSSP, 1.05× to 1.18× for BFS, and 12× to 13.2× for SSNP.

Table 2.5: 2Step: Speedups for three Reuse Table configurations: 5 out of 20; 2 out of 10; and 1 out of 5. Overall the 1 out of 5 configuration performs the best in 3 of 5 benchmarks as it minimizes the reuse overhead.

G	Hops	SSWP	Viterbi	SSSP
TT	1	66.5 : 155.9 : 249.0	9.51 : 10.2 : 10.5	1.97 : 1.85 : 1.72
	2	52.1 : 125.7 : 266.2	6.61 : 6.82 : 6.96	1.57 : 1.48 : 1.40
	3	48.2 : 106.5 : 205.1	3.81 : 3.94 : 3.90	1.49 : 1.42 : 1.35
	4	43.9 : 95.9 : 199.9	3.20 : 3.24 : 3.28	1.49 : 1.41 : 1.36
TTW	1	82.4 : 94.6 : 157.6	9.61 : 9.48 : 10.9	1.83 : 1.75 : 1.67
	2	71.0 : 69.3 : 125.6	6.83 : 7.44 : 7.34	1.39 : 1.33 : 1.30
	3	71.0 : 79.0 : 125.3	4.47 : 3.97 : 4.82	1.38 : 1.35 : 1.26
	4	47.7 : 51.7 : 111.6	3.01 : 3.76 : 3.86	1.32 : 1.27 : 1.21
LJ	2	26.2 : 72.7 : 122.2	8.50 : 10.9 : 10.4	1.14 : 1.13 : 1.13
	3	23.2 : 65.9 : 119.6	5.20 : 6.46 : 6.51	1.12 : 1.13 : 1.14
	4	25.2 : 57.3 : 105.2	4.53 : 5.71 : 4.80	1.12 : 1.14 : 1.14
	5	17.8 : 46.1 : 78.7	4.00 : 4.25 : 4.72	1.17 : 1.15 : 1.11
PK	2	38.0 : 89.3 : 138.5	11.5 : 15.6 : 14.5	1.47 : 1.51 : 1.43
	3	31.2 : 75.6 : 120.2	6.83 : 8.04 : 8.27	1.37 : 1.29 : 1.32
	4	23.0 : 57.1 : 88.0	5.19 : 4.89 : 5.29	1.50 : 1.48 : 1.37
	5	20.5 : 52.0 : 78.5	4.05 : 4.52 : 4.19	1.48 : 1.42 : 1.39
Average		43.0× : 80.9× : 143.2×	6.05× : 6.82× : 6.89×	1.43× : 1.38× : 1.33×

G	Hops	BFS	SSNP
TT	1	2.12 : 2.20 : 2.14	16.2 : 17.3 : 18.0
	2	1.15 : 1.22 : 1.24	16.1 : 15.3 : 16.1
	3	0.95 : 1.38 : 1.27	16.0 : 14.8 : 15.5
	4	0.95 : 0.95 : 1.01	11.8 : 12.1 : 12.4
TTW	1	1.69 : 2.01 : 2.10	14.4 : 15.3 : 15.7
	2	0.99 : 1.14 : 1.03	14.8 : 14.2 : 15.2
	3	0.87 : 1.00 : 0.94	12.4 : 15.3 : 14.5
	4	0.83 : 0.93 : 0.97	11.8 : 13.9 : 13.0
LJ	2	0.96 : 1.04 : 1.09	14.8 : 17.6 : 18.7
	3	0.88 : 0.90 : 0.97	11.9 : 14.1 : 15.9
	4	0.84 : 0.91 : 0.94	9.34 : 13.3 : 14.2
	5	0.82 : 0.90 : 0.95	6.75 : 8.62 : 9.62
PK	2	1.09 : 1.17 : 1.17	11.3 : 12.6 : 6.23
	3	0.95 : 1.02 : 1.04	4.91 : 5.47 : 5.57
	4	0.90 : 0.97 : 1.00	8.16 : 9.03 : 5.12
	5	0.88 : 0.93 : 0.97	7.40 : 8.03 : 4.28
Average		1.05× : 1.17× : 1.18×	12.0× : 13.2× : 12.8×

The larger power-law graphs (TT and TTW) experience higher speedups than smaller power-law graphs (LJ and PK). The number of hops from query source vertex to nearest reuse table source vertex also impacts performance. By and large, the smallest hop distance gives the best speedups while the largest hop distance gives the least speedup. For example, for SSWP on TTW with the smallest reuse table configuration, speedups decrease from 157.6× to 111.6× as hop distance increases from 1 to 4. For SSWP the smallest table configuration gives best speedups, for Viterbi and BFS the speedups for the two smaller table configurations are fairly close and significantly better than for the largest table configuration, and for SSSP mostly the largest table gives best speedups, though by a small margin. Larger reuse tables may enable more effective reuse but also incur higher reuse overhead. Thus, depending upon the algorithm characteristics, different sized reuse tables deliver the best performance for different benchmarks.

Reduction in Active Edges and Stable Values Produced Next we present additional data for the benchmarks to better understand the large degree of difference in speedups observed. Tables 2.6 and 2.7 provides reduction in active edges processed and extent to which reuse step produces stable values. The reason for varying degrees of speedups can be found in algorithms characteristics.

First, performance benefit of reuse is very high for SSWP because in this algorithm no new values are computed – the resulting property value for each vertex is essentially equal to the weight of a selected edge. Thus, reuse often produces stable values as high as 99.99% as shown in Table 2.7. This unique characteristic of SSWP resulted in high speedups.

Table 2.6: 2Step: % reduction in processed active edges due to reuse for table configuration of 2 *out of* 10.

G	Hops	SSWP	Viterbi	SSSP	BFS	SSNP
TT	1	99.99	95.28	73.14	29.94	99.99
	2	99.99	99.99	82.80	18.89	99.99
	3	99.99	89.30	66.84	2.21	99.99
	4	99.99	85.60	71.31	3.89	99.99
TTW	1	99.99	94.42	68.76	34.06	99.99
	2	99.99	99.99	80.48	38.09	99.99
	3	99.99	89.67	62.05	2.11	99.99
	4	99.99	91.44	70.93	1.67	99.99
LJ	2	99.99	96.66	58.53	5.86	99.99
	3	99.99	86.89	50.03	3.04	99.99
	4	99.99	94.86	46.84	0.41	99.99
	5	99.99	93.94	49.38	0.34	99.99
PK	2	99.99	98.03	74.59	6.78	99.99
	3	99.99	89.56	70.93	9.71	99.99
	4	99.99	93.21	69.54	1.10	99.99
	5	99.99	91.36	71.88	0.77	99.99
Average		99.99%	93.14%	66.75%	9.93%	99.99%

Table 2.7: 2Step: Percentage of values that become stable following reuse for table configuration of 2 *out of* 10.

G	Hops	SSWP	Viterbi	SSSP	BFS	SSNP
TT	1	99.99	90.57	41.37	93.04	99.99
	2	99.99	85.81	21.58	60.28	99.99
	3	99.99	74.70	22.64	42.91	99.99
	4	99.99	62.81	28.88	47.31	99.99
TTW	1	99.99	90.79	48.76	94.78	99.99
	2	99.99	89.38	31.81	72.83	99.99
	3	99.99	80.31	31.26	52.97	99.99
	4	99.99	79.10	30.86	47.83	99.99
LJ	2	99.99	91.72	21.08	46.24	99.99
	3	99.99	84.69	16.61	23.65	99.99
	4	99.99	84.72	11.94	16.52	99.99
	5	99.99	79.25	13.39	15.69	99.99
PK	2	99.99	92.99	46.34	66.88	99.99
	3	99.99	84.14	41.52	39.41	99.99
	4	99.99	71.65	33.97	32.76	99.99
	5	99.99	61.77	36.24	27.59	99.99
Average		99.99%	81.52%	29.89%	48.79%	99.99%

The nature of Viterbi and SSSP is quite similar as both compute new values, except that one involves real values and the other integer values. However, on average, Viterbi produces 81.52% stable values while SSSP produces only 29.89% stable values on average across all four power law graphs. Thus, average reduction in active edges is 93.14% for Viterbi which is significantly higher than 66.75% for SSSP across all power law graphs. This explains why Viterbi achieves higher speedups than SSSP via reuse even though the values were not stable following reuse.

Small benefits are expected for BFS as cost of computing a value for a vertex is similar to cost of generating the value via reuse. The competing factors of possibly increased work (when insufficient number of stable values are produced) and better memory behavior during reuse (because edge-lists are not accessed and vertices are visited in the order they are stored) result in small speedups or small slowdowns.

Speedups for Individual Queries So far we have presented the average execution times and speedups over 20,000 queries for the 2 *out of* 10 configuration. To demonstrate that the speedups are achieved across nearly *all* queries, we present plots in Figure 2.3 where for the TT graph, the execution times of 2Step and NoReuse for each query are plotted. The plots are given for the SSWP and BFS algorithms that give maximum and minimum average speedups across all benchmarks. The scatter plots in Figure 2.3 show that execution times for nearly all of the 20K queries are improved by **VRGQ** (only exception is BFS, Hops=4).

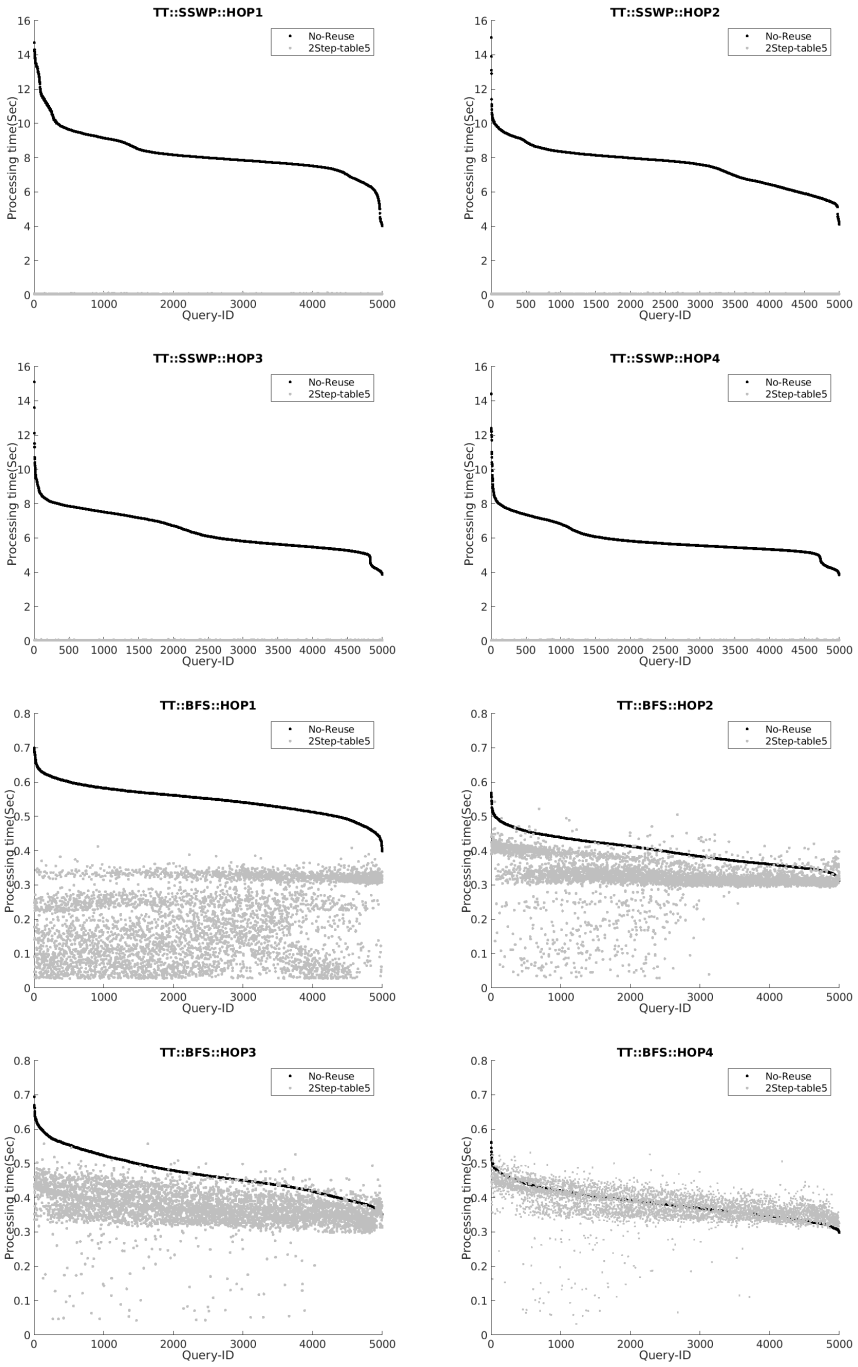


Figure 2.3: Performance of individual queries – each scatter plot depicts execution times of 2Step and NoReuse for 20,000 queries – black dots correspond to 2Step execution times and gray dots are for NoReuse execution times. The plots show that vast majority of queries benefit from 2Step algorithm.

Table 2.8: Populating Reuse Table: Overlap (OL) in 20 source vertices selected; Runtime Overhead (OH) of using different number of random queries to select 20 vertices; and Runtime overhead of populating the Reuse Table with full results for the 20 vertices.

G		Number of Random Queries					Popu. Table
		1000	10	20	40	60	
TT	OL	20	19	20	20	20	
	OH	6070s–3.8%	101s–3.1%	175s–5.1%	333s–6.2%	393s–4.7%	699s
TTW	OL	20	19	18	19	19	
	OH	3095s–5.7%	60s–7.9%	89s–1.5%	155s–6.4%	227s–6.8%	490s
LJ	OL	20	20	18	18	20	
	OH	164s–14.1%	3.1s–7.3%	5s–9.7%	7.4s–6.7%	13.6s–4.4%	43s
PK	OL	20	17	19	19	19	
	OH	81s–8.5%	1.3s–0.3%	2.5s–10.3%	4.2s–10.2%	5.6s–7.8%	16s

2.4 Populating the Reuse Table

To populate the reuse table we need to identify a small subset of vertices in the graph, say N, whose backward and forward query results will be precomputed and stored in the reuse table. Since an input graph typically contains millions of vertices, we need to develop a methodology for selecting the N vertex set. Our selection is aimed at achieving two goals:

- Maximize Reuse – Since all vertices are not equally effective in the degree of reuse they support, we will include vertices in N that have *high centrality*. That is, we will give preference to vertices that play an influential role in computation of results of large number of other queries.

- High Coverage – Ideally we would like to choose N such that, from all other vertices in the graph, at least some vertices in N can be reached in less than MaxHops. That is, all queries will encounter reusable vertices. Since this is not always possible, we attempt to achieve *high coverage*, i.e. for most queries reuse is possible.

Our three step methodology for selecting vertex set N such that it meets the above goals is presented below.

Step 1: Identify High Centrality Candidates using Random Queries – We evaluate randomly selected queries and during each query evaluation we maintain *impact counts* for all other vertices in the graph. Each time the value of some vertex v, causes an update of its out-neighbor, the impact count of v is incremented. Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between pairs of nodes. Vertices that have a high probability to occur on a randomly chosen shortest path between two randomly chosen vertices have a high betweenness. High *impact counts* are treated as an indicator for high centrality. Thus, extremely expensive centrality is avoided. For each query evaluated, the top N vertices, i.e. vertices with the highest impact counts, are identified. Since this process is repeated for multiple queries, say q_1, $q_2 \cdots q_n$, we obtain multiple candidate sets $\mathsf{N}(q_1)$, $\mathsf{N}(q_2) \cdots \mathsf{N}(q_n)$.

Step 2: Selecting top $\mid \mathbf{N} \mid$ High Centrality Candidates – Note that the same vertex may appear in multiple sets $\mathsf{N}(q_1)$, $\mathsf{N}(q_2) \cdots \mathsf{N}(q_n)$ obtained in the first step but not necessarily in all the sets. We select the final set N by both considering how frequently a vertex appears in different sets and its corresponding impact counts. This is achieved by summing up the impact counts of all occurrences of a vertex in the above sets and sorting the final sums to identify the top $\mid \mathbf{N} \mid$ vertices.

Step 3: Populating the Table – The top centrality candidates identified in the preceding step are processed one by one to populate the reuse table. By evaluating the query for each vertex v in both the forward direction on the original graph and backward

direction on the edge reversed graph, the reuse table is populated with FwdTable[v][*] and BwdTable[*][v].

Cost of populating strategy Finally we would like to describe the runtime cost of constructing the reuse table of size 20 – we present this cost when using different number of random queries – 1000, 10, 20, 40 and 60 – in Table 2.8. Considering the 20 vertices found using 1000 random queries as the best selection, we compared its results with sources vertices selected by using 10, 20, 40 and 60 random queries. The overlap among the selected vertices is very high – when using 60 random queries, the overlap is 20 out of 20 for TT and LJ and it is 19 out of 20 for TTW and PK. Thus, running 60 random queries is more than sufficient for identifying high centrality vertices. The runtime overheads for selecting 20 vertices using 60 random queries range for 5.6 to 393 seconds depending upon the size of the power law graph graph while populating the table with results takes 16 to 699 seconds. Note that while high centrality vertices are being selected, new queries can be processed in parallel. The cost of selecting high centrality vertices varies from 0.44s (for PK) to 18.47s (for TT) for power law graphs for 60 random queries in Table 2.8. Thus, the overhead of populating the reuse tables is modest as it has to be done only once.

In Table 2.9 we show results of our experiment justifying the selection of 5 *out of* 20 configuration. If in the reuse step, instead of reusing results of best 5 we reuse results of all 20, little to no reductions in vertices processed and increase in stable values produced are observed as shown in the table. We also increased the size of the reuse table from 20 to 40 source vertices. No improvements in vertices processed or stable values produced were

Table 2.9: Improvements if instead of using 5 *out of* 20 we reuse results of all 20 queries. The minimal improvements indicate that reusing results of a few queries is enough.

Improvements in	**SSWP**	**Viterbi**	**SSSP**	**BFS**
Vertices Processed	0%	0%	3%	5%
Stable Values	0%	0%	1%	7%

Table 2.10: Substantial overlap in high centrality vertices for different benchmarks and graphs for **20** Reusable- Vertices indicates that same reuse vertices can be used across different benchmarks.

	TT	**TTW**	**LJ**	**PK**
SSWP	20	20	20	20
Viterbi	20	19	18	19
SSSP	20	19	17	16
BFS	19	18	19	16

observed for the first three programs and 1% improvement was seen for BFS.

Heuristic effectiveness Our heuristic for finding high centrality vertices is very effective. To verify this we found high centrality vertices using the betweenness centrality algorithm available in Ligra and compared them with ones found by our heuristic. We found that top 5 vertices found by the original algorithm appear among the top 10 vertices found by our heuristic. Thus, there is great degree of overlap, though vertices appear in different order.

Finally, we identified the top 20 vertices in the four power law graphs by considering the centrality with respect to property values of all four algorithms and found that vast majority of vertices are the same for all algorithms. Table 2.10 shows how many of the 20 reuse sources found for SSWP also appear in the 20 reuse sources found for Viterbi, SSSP and BFS. The high overlap indicates that structure of the graph is the determinative factor much more so than the graph algorithm. Thus, for a given graph, it is possible to find reuse

Table 2.11: Higher Hub populating times in seconds and their comparison with VRGQ populating times.

Graph→	**TT**	**TTW**	**LJ**	**PK**
Hub=20	1102s 1.58×	732s 1.49×	43s 1.00×	16s 1.00×
Hub=100	5925s 8.50×	3390s 6.92×	211s 4.90×	77s 4.81×

Table 2.12: Evaluation of Hub and VRGQ applying reuse on 4K BFS queries with size of **20**. The results show that Hub-based reuse largely results in slowdowns.

Speedups		Reuse 1	Reuse 2	Reuse All
Hub=20	**PK**	0.89×	0.75×	0.35×
	TT	0.98×	0.96×	0.75×
Hub=100	**PK**	0.81×	0.74×	0.12 ×
	TT	1.01×	0.95×	0.31 ×
VRGQ	**PK**	1.06×		
	TT	1.39×		

vertices once and use them for all algorithms.

Hub Accelerator vs. VRGQ The Hub Accelerator [34] is used to speedup the evaluation of graph queries. At a high level it is similar to VRGQ as both rely upon precomputations and then use the results of precomputations to speedup subsequent query evaluations. However, VRGQ computes point-to-all (i.e., queries involving single source and all destinations) while the Hub Accelerator computes point-to-point queries (i.e., queries involving a single source and destination pair). We implemented the Hub Accelerator in Ligra and adapted it to compute point-to-all queries by reusing the Hub Accelerator results for all destinations. Next we show that VRGQ is far more effective than Hub Accelerator both because its precomputation is relatively inexpensive and the speedups obtained are higher.

The reason why the Hub Accelerator precomputation is more expensive than our

precomputation for populating the reuse table is as follows. For the Hub network to be effective it is typically chosen to have large number of vertices (e.g., in [34] the authors consider Hub sizes of 5K to 15K vertices) and then shortest paths among all these vertices are precomputed. In addition, the shortest path from each non-Hub vertex to the closest core-Hub vertex must also be precomputed. In contrast, the 2Step algorithm of VRGQ only precomputes results for a handful of high centrality vertices to populate the reuse table. Table 2.11 compares the precomputation costs of Hub Accelerator precomputation with reuse table precomputation. Table 2.11 first presents Hub Accelerator precomputation times in seconds and then the factor by which the cost of Hub Accelerator precomputation exceeds that of reuse table precomputation of VRGQ. We present this data for Hub sizes of 20 and 100 for all four graphs. We observe that Hub Accelerator precomputation is several times more expensive that VRGQ precomputation for Hub size of 100 and even for Hub size of 20 for the difference can be substantial for large graphs.

We also compare speedups for evaluating four thousand BFS queries by applying reuse with Hub Accelerator and VRGQ in Table 2.12. These four thousand queries were chosen from 20K queries used in earlier experiments, we randomly chose 1000 queries each from the 5000 queries for each of the four hop values. With Hub size of 20, the same as the number of reusable-vertices for VRGQ, the Hub Accelerator only experiences slowdowns as shown in Table 2.12. If we reuse all precomputed results, due to high reuse cost, we get significant slowdown. We therefore tried reuse of only 1 or 2 vertices from the Hub but this too did not produce any speedups although it reduced the slowdowns. When Hub size was increased to 100, for the TT graph, a slight speedup of one percent was

finally achieved though the precomputation cost is dramatically increased. In contrast VRGQ achieves average speedup of 1.39× for TT on BFS queries. Because reuse is very expensive, in VRGQ, we selected only one source vertex whose results are reused. For Hub Accelerator [34], each vertex can have multiple *corehubs*. The result in Table 2.12 shows that if reuse all corehubs we only achieve slowdowns. If we select a specific corehub vertex by distance, then VRGQ and Hub Accelerator will select the same vertex for reuse. Hence there is no advantage of using a selected corehub over VRGQ. Moreover, Hub Accelerator was designed for point-to-point queries and hence its reuse table contains partial results.

From the above results we conclude that the limitation of the Hub Accelerator approach is that it cannot deliver speedups for point-to-all queries. When small Hub sizes are used only slowdows are observed and when large Hub sizes are used the precomputation cost becomes very high. In contrast, VRGQ gives speedups with small number of reuseable-vertices while incurring low precomputation cost. Finally our 2Step algorithm requires minimal change to the original algorithms as it simply initializes the vertex values using reuse-table (not default initialization values) and then standard iterative algorithm is run.

2.5 Related Work on Evaluating Multiple Graph Queries

VRGQ reuses results of a few queries for achieving speedups in the evaluation of numerous other queries. We have adapted this concept and applied it in other scenarios as well. SimGQ [92] reuses the results of dynamically identified *sharing queries* to speed up the simultaneous evaluation of a batch of queries. Tripoline [33] uses this idea to create a generalized streaming graph model where continuously updated results of a small number

of standing queries are used to rapidly evaluate arbitrary user queries.

There are two recent works, Quegel [95] and PnP [93], that evaluate a stream of graph queries. However, both these works are aimed at evaluating point-to-point queries (e.g., shortest path from a single source to a single destination). Quegel derives improved throughput by evaluating queries in a pipelined fashion and taking advantage of the Hub [34] precomputation. PnP [93] is similar to other graph frameworks in that speedups are achieved by evaluating a single query faster using new dynamic optimizations.

There are other recent works [95, 79, 59, 94, 100, 48] that evaluate multiple queries simultaneously. Congra [59] handles each query independently using a separate process. Thus, it is unable to take advantage of value reuse opportunity across queries. CGraph [100] and Seraph [94] are out-of-core systems that evaluate multiple queries. However, their benefits are achieved via sharing of the graph as opposed to the result values. In [79] a specialized system that processes multiple BFS queries is presented. Finally, MultiLyra [48] and BEAD [49] simultaneously evaluate queries on distributed systems.

2.6 Summary

We have developed VRGQ that incorporates the 2Step iterative algorithm for evaluating a sequence of graph queries on individual vertices. The key feature of this framework is that during evaluation of any query, coarse-grained reuse of previously computed results for selected vertices is performed to accelerate query evaluation. The VRGQ system is different from related works in a key way. It takes advantages of results computed for a very small number of queries to optimize the execution of all future queries via value reuse.

Chapter 3

Exploiting Value Reuse in Other Scenarios

Value reuse is a platform independent technique that exhibits extensive applicability across various systems for vertex-specific iterative queries. It provides initial values for queries to reduce computation and memory access costs. Value reuse has been successfully implemented in multiple systems. Two prominent scenarios where value reuse manifests its effectiveness are arbitrary query evaluation for streaming graphs and batched evaluation of queries in both distributed and shared memory systems. In these two cases we leverage the shared values among computations, which not only decreases computation overhead but also enhances the overall system performance.

3.1 Streaming Graphs

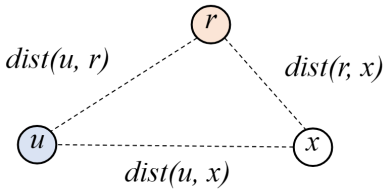
In many real-world application scenarios, a stream of updates are continuously applied to the graph, often in batches for better efficiency, known as the streaming graph scenario. Taking Twitter as an example, new followers are generated constantly, adding new edges to the existing graph. To get query results on the latest graph, a naive solution is simply evaluating the query on the latest graph from scratch. For iterative graph queries, this full query evaluation can be expensive. Instead, existing streaming graph processing frameworks choose to evaluate the queries continuously and incrementally for better efficiency. Several streaming graph systems have been proposed to support incremental evaluation, like Kineograph [9], Tornado [71], KickStarter[84] and others. Instead of reevaluating the query on the updated graph from scratch (i.e., view it as a completely new graph), existing streaming graph systems adopt an incremental graph query evaluation strategy and tend to terminate faster than a full reevaluation [9], [84].

Despite the promise of incremental graph processing, there exists a fundamental limitation in the existing design. Except pre-selected standing queries, an expensive full evaluation is required for arbitrary queries. In fact, vertex-specific graph queries hold significant relevance in real-world applications. Vertex-specific queries are concerned with the interests or capture the perspective of a specific vertex, which are common in online shopping and social networks, such as generating recommendations for individual customer[96] and finding the overlap of friends of two specific users. This limitation significantly compromises the generality of the existing incremental streaming graph systems. So we propose a principled way to generalizing the incremental graph processing such that vertex-specific

queries without their a priori knowledge may also benefit from incremental processing. In this chapter, we focus on adaption of Value Reuse to lift this requirement, such that a user query for an arbitrary source vertex can be incrementally evaluated.

3.1.1 Value Reuse in Streaming Graphs

Value Reuse can connect the evaluation of a vertex-specific query to the results of another query evaluation, providing solutions to the same type of query in static graphs. Similar principles exist in streaming graph problems and can be naturally derived. As a result, Value Reuse could be used to optimize streaming graph processing. Because of the nature of the incremental streaming graph evaluation model, the key to our solution is to connect an arbitrary user query with a standing query that satisfies Value Reuse principle. Next, we first use SSSP as an example to introduce the Value Reuse in streaming graphs.



for any u, r, and x $\quad dist(u, r) + dist(r, x) \geq dist(u, x)$

Figure 3.1: Value Reuse in SSSP

Value Reuse in SSSP as Example. Based on Value Reuse in static graphs, it is not hard to find similar distance between vertices in streaming graph. In static graphs, we selected high centrality vertices r, and store the distances from r to all vertices in graph. But in streaming graphs, we keep information for standing queries, and the connection between arbitrary source query u and any possible destination x has been illustrated in Figure 3.1.

In fact, the above Value Reuse in Figure 3.1 becomes obvious once one realizes that the shortest path(dashed lines represented) from u to r can be concatenated with the shortest path from r to x, and the resulted path is just one of the many paths from u to x, hence must be no shorter than the shortest path from u to x. As a result, we evaluate any source vertices from value $dist(u, r) + dist(r, x)$ instead of default, without a priori knowledge.

Rather than referring to the distance, we can also apply Value Reuse for a property between two vertices – $property(v1, v2)$, where $(v1, v2)$ is an ordered pair for directed graphs and an unordered pair for undirected graphs. And could be be formally defined by the following equation: $property(v1, v2) \bigoplus property(v2, v3) \succcurlyeq property(v1, v3)$, where $\bigoplus$ depicts an abstract addition and $\succcurlyeq$ represents an abstract greater than or equal operator. For SSWP, the widest path between two vertices is the path whose minimum edge weight is the largest. Based on this definition, it is not difficult to derive the Value Reuse application in SSWP for any u, r, and x: $min(wide(u, r), wide(r, x)) \leq wide(u, x)$. Here $\bigoplus$ is function MIN, and $\succcurlyeq$ represents $\leq$. We also turn to the reversed graph problem for directed graphs.

Value Reuse holds for SSWP because the widest paths from u to r and from r to x can be concatenated, and the width of the concatenated path must be no larger than the width of the widest path from u to x as it is just one of the paths from u to x. Similarly, we next briefly present the Value Reuse for other graph problems, including SSNP, Viterbi, SSNSP, and Radii [73].

3.1.2 Value Reuse based Incremental Evaluation Model

After establishing the relations between the user query and the standing query, we can design a more general user query evaluation framework, where the user query can reuse the results of the standing query, even when their source vertices are different. Similar to the static graph, Value Reuse with generalized distance and comparison operators may also be derived for vertex-specific queries in the streaming graph. Based on them, we can establish rigorous constraints between a user query (whose source vertex can be any vertex in the graph) and the pre-selected standing query, thus enabling reusing the results of the latter to accelerate the evaluation of the former. Figure 3.2 illustrates the basic idea of our solution.

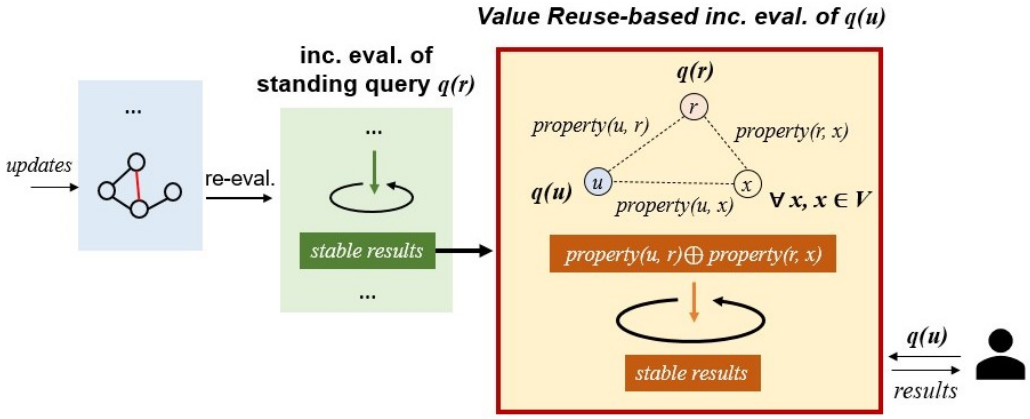


Figure 3.2: Value Reuse-based Incremental Evaluation

Each time, after the graph is updated, the standing query $q(r)$ (usually same high degree vertices) will be re-evaluated incrementally. Meanwhile, the system accepts user queries like $q(u)$ which is of the same type as $q(r)$, but its source vertex u could be any vertex in the graph. From the evaluation of $q(r)$, we can obtain the values of u to r, and r

to any other vertices in graphs. Next, instead of evaluating $q(u)$ from scratch (i.e., using the default initial values) on the current version of the graph, the system starts its evaluation directly from the intermediate value and runs until all the vertex values are converged. As these calculated initial values must be no worse than the default initial values, in fact, often much better, the evaluation usually takes less iterations and computations to terminate. From the high level, the Value Reuse module serves as an adaptor that connects the user query with the standing query.

3.2 Batched Evaluation of Queries

Batching queries in graph processing systems enhances performance by enabling parallel processing and reducing overheads in query evaluation. It leverages modern hardware's parallel execution capabilities and minimizes latency in distributed systems. Batching promotes efficient cache utilization, as the required data loaded once can be used multiple times, reducing cache misses. Furthermore, it opens avenues for query optimization, such as avoiding duplicate work across queries, and enables the system to anticipate data needs better, allowing for effective data prefetching. However, its applicability might be limited in real-time or low-latency scenarios where gathering enough queries for a batch could be impractical. Machine resources can be fully utilized delivering higher throughput by simultaneously evaluating multiple queries on a modern server with many cores and substantial memory resources.

Batch evaluation queries benefit from Value Reuse due to the intrinsic data sharing that occurs between different queries within the batch. In most cases, the various queries

traverse the majority of the graph. This enables the selection of only a few crucial queries, and their computation at a reduced cost. Then these important queries values can be stored in memory for single-machine systems like SimGQ [92], or across each machine in a distributed system like Multilyra [48], thereby enhancing efficiency.

3.2.1 Value Reuse in Batched Evaluation on a Shared-Memory System

SimGQ is a system that optimizes simultaneous evaluation of a group of vertex queries that originate at different source vertices (e.g., multiple shortest path queries originating at different source vertices) and delivers substantial speedups over a conventional framework that evaluates and responds to queries one by one. The performance benefits are achieved via batching and sharing. The SimGQ system contains three phases: identifying shared queries, accelerating batch queries using shared queries, and completing the evaluation of batch queries.

A Result Table maintains the results of all the queries for each vertex, and at termination the results of all queries can be found in it. We accelerate the convergence of the solution of the original batch of queries in Result Table using the results of the shared queries in a Shared Table. Since the cost for looping over all vertices and applying share updates is significant, we limit the number of shared vertices with which each query is used to speed up convergence of property values by choosing a small set size. The result of a shared query with source vertex benefit a batch of queries via reuse.

Batching fully utilizes system resources to evaluate a batch of queries and amortizes run-time overheads incurred due to fetching vertices and edge lists, synchronizing threads, and maintaining computation frontiers. Sharing dynamically identifies a shared query that

substantially represent subcomputations in the evaluation of different queries in a batch, evaluates the shared queries, and then uses their results to accelerate the evaluation of all queries in the batch. For SimGQ [92], First, it is easy to see that during batched evaluation, we can share the iteration overhead across the queries. This overhead includes the cost of iterating over the loop, synchronizing threads at the barrier, as well as fetching vertex values and edge lists of active vertices to update vertex values and the computation frontier. Second, synergy or overlap between computations performed by the queries can be exploited to reduce the overall computation performed. By evaluating the shared queries once, we can speedup the evaluation of the entire batch of queries. Note that the shared queries must be identified dynamically because they may vary from one batch to another.

3.2.2 Value Reuse in Batched Evaluation on a Distributed System

Various distributed graph processing frameworks have been developed to deliver scalable performance for evaluation of individual iterative graph queries. In practice though, we may need to evaluate many queries. MultiLyra [48] is a distributed framework that efficiently evaluates a batch of graph queries. To deliver high performance, this system is designed to amortize the communication and synchronization costs of distributed query evaluation across multiple queries. Reuse technique has been incorporated as an optimization where results from earlier batches of queries are used to accelerate the execution of later batches of queries. MultiLyra is a distributed framework that generalizes PowerLyra to handle a batch of queries efficiently. Powerlyra employs the GAS (Gather-Apply-Scatter) model to divide the distributed computation into three phases.

The Reuse optimization includes three steps. First, it identifies the top high-

centrality vertices while processing the first batch. Second, it selects the top five vertices and runs them as an extra batch, storing the results distributed across each machine for its local vertices. In the third step, these results from the initial five queries are reused throughout the following batches, significantly accelerating the convergence of each query. Consequently, the remaining batches of queries can exploit the notable speedup offered by the Reuse optimization. To accommodate this, a new Reuse-Batch phase has been introduced to the MultiLyra GAS model, placed between the Apply-Batch and Scatter-Batch phases.

The final results of high centrality vertices, serving as source vertices with destinations for all queries, are stored on every machine. Once a stored high centrality vertex becomes active, its stored results are used to update the intermediate results of query being evaluated during the Reuse-Batch phase. As the host machine holds the most recent values, it is responsible to send these values to other machines to ensure that all machines have the required data for the reuse process. After employing optimizations that improve scalability of expensive phases and perform reuse across the distributed computation, the improved maximum speedups range from 7.35× to 11.86×.

3.3 Summary

Value reuse exhibits extensive applicability across various systems. In streaming graph systems, Tripoline proposes to leverage the value reuse technique to solve a fundamental limitation of stream graph systems. This idea leads to a generalized incremental processing design for vertex-specific queries. Multilyra [48] performs reuse across distributed

computation, results from earlier batches of queries are used to accelerate the execution of later batches of queries. SimGQ [92] is a shared memory system dynamically identifies shared queries that substantially represent subcomputations in the evaluation of different queries in a batch, then evaluates and reuses their results to accelerate the evaluation of all queries in the batch.

Chapter 4

Core Graph: Exploiting Graph Structure Reuse

When evaluating an iterative graph query over a large graph, systems incur significant overheads due to repeated graph transfer across the memory hierarchy coupled with repeated (redundant) propagation of values over the edges in the graph. An approach for reducing these overheads combines the use of a small proxy graph and the large original graph in a two phase query evaluation. The first phase evaluates the query on the proxy graph incurring low overheads and producing mostly precise results. The second phase uses these mostly precise results to bootstrap query evaluation on the larger original graph producing fully precise results. The effectiveness of this approach depends upon the quality of the proxy graph. Prior methods find proxy graphs that are either large or produce highly imprecise results.

We present a new form of proxy graph named the Core Graph (CG) that is not only

small, it also produces highly precise results. A CG is a subgraph of the larger input graph that contains all vertices but on average contains only 10.7% of edges and yet produces precise results for 94.5–99.9% vertices in the graph for different queries. The finding of such an effective CG is based on our key new insight, namely, a small subset of non-zero centrality edges are responsible for determining the converged results of nearly all the vertices across different queries. We develop techniques to identify a CG that produces precise results for most vertices and optimizations to efficiently compute precise results of remaining vertices. Across six kinds of graph queries and four input graphs, CGs improved the performance of GPU-based Subway system by up to 4.48×, of out-of-core disk-based GridGraph system by up to 13.62×, and of Ligra in-memory graph processing system by up to 9.31×.

4.1 Our Approach

Graph analytics is employed in many domains (e.g., social networks, web graphs) to uncover insights from connected data. There has been much work resulting in scalable graph analytics systems for GPUs, multicore servers, and clusters [50, 46, 20, 57, 73, 86, 105, 41, 37, 36, 67, 89, 5, 27]. Real world graphs are irregular and large. Thus, significant overheads are incurred due to movement of graph across the memory hierarchy and repeated propagation of values over the edges in the graph. These overheads are exacerbated due to the iterative nature of graph analytics. Thus, in spite of the numerous advances, efficient processing of large and irregular graphs remains a challenge.

A general approach [39, 99] for dealing with the above challenge employs a *two-*

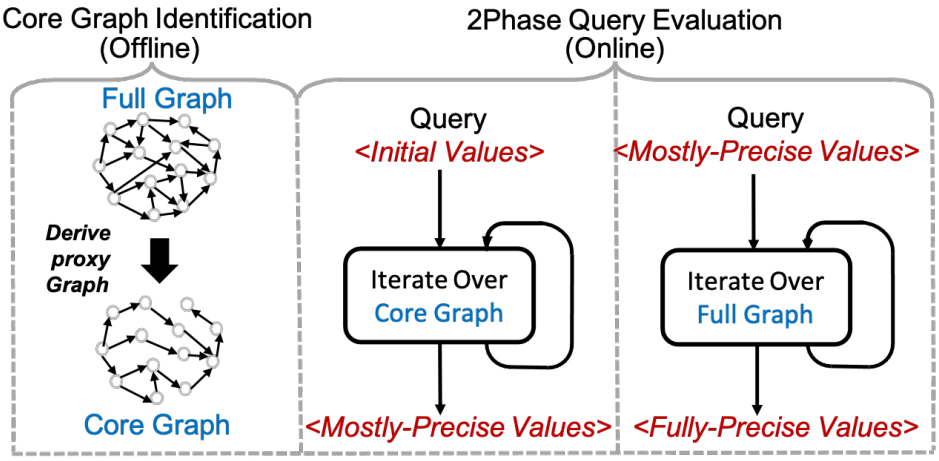


Figure 4.1: Proxy Graph based 2Phase Evaluation.

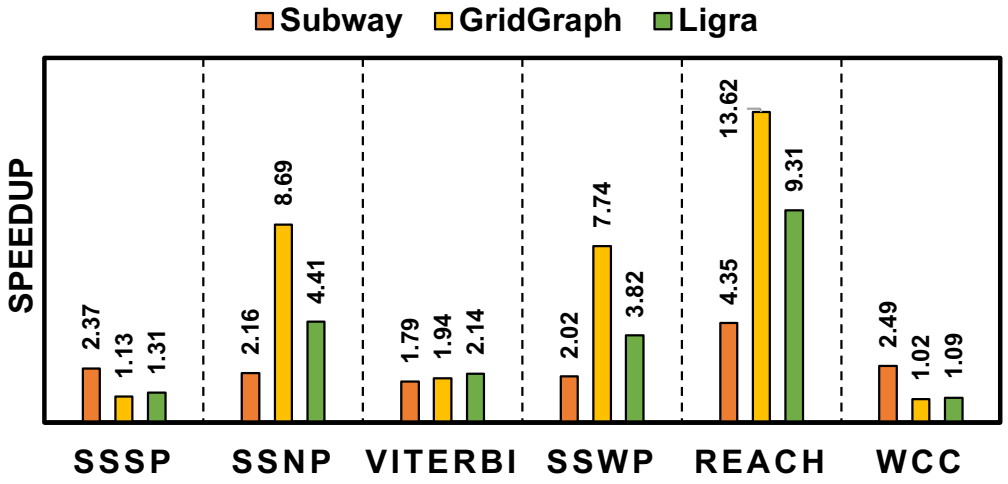


Figure 4.2: Speedups with CG over without CG for Friendster (FR) [19] input graph with 2.586 billion edges.

phase (2Phase) query evaluation as shown in Figure 4.1. Here a small proxy graph corresponding to the large original graph is identified once and then the combination of the proxy graph and original graph is used to evaluate all future queries. *Note that for graphs with a large number of vertices, there is an equally large number of vertex-specific queries*

(e.g., each vertex can serve as a source of a shortest path query). The first phase evaluates the query on the proxy graph incurring low overheads and producing mostly precise results (mostly green and some red values in Figure 4.1). Then, the second phase uses these mostly precise results to bootstrap query evaluation on the larger original graph producing fully precise results (all green values in Figure 4.1). Resuming query evaluation in the second phase from vertices whose property values are impacted in the first phase, guarantees that the 2Phase algorithm will produce correct results for 100% of vertices [39, 99]. Given its generality, this approach is applicable to different kinds of systems – GPU-based Subway, in-memory Ligra, out-of-core GridGraph – as shown in Figure 4.2. Moreover, the improvements are largely complimentary to other platform specific optimizations incorporated in different graph processing systems. However, for the above approach to be effective, the proxy graph must fulfill two key requirements:

- **RQ1**: The proxy graph should be **much smaller** than the original so that the first phase incurs substantially reduced graph transfer overhead and performs little redundant propagation of values over edges; and
- **RQ2**: Convergence over the proxy graph should produce query results that have **mostly precise**, i.e. most have converged to the same values that are obtained upon convergence over the original graph. Thus, the second phase requires little effort to reach full convergence (i.e., convergence for all vertices).

Prior methods [39, 99], *Reduced Graph* [39] and *Abstraction Graph* [99], are proxy graphs with significant limitations. The first method by Kusum et al. [39] collapses parts of the graph eliminating many vertices that cannot be queried and produces a proxy graph

that is too large (roughly 50% of original size [39]). Although the *Abstraction Graph* [99] overcomes the limitations of *Reduced Graph* and produces a small proxy graph, it yields query results that are imprecise – it produces imprecise results for over 53% of the vertices. Similar lack of precision is observed when graph sampling is used to to produce a small proxy graph [108, 106, 107]. Query-by-sketch [90] identifies a self-contained subgraph (meta-graph). However, it is limited to queries that find the shortest path between two vertices. Our approach handles other kind of queries beside shortest paths and it finds property values from a source vertex to all destination vertices.

In this paper we develop a new approach for identifying a proxy graph, called the **Core Graph** (CG), that satisfies both the aforementioned requirements. That is, Core Graph is both small and produces highly precise results. The CG includes all the vertices from the original graph so that any vertex-specific query can be evaluated and it contains only a subset of edges. Which edges to include in the CG, and how to identify these edges efficiently, is one of our key contribution.

For many kinds of *vertex-specific* queries (i.e., queries of the form $Q(s)$ where s is the source vertex), the final property value of a vertex v can be precisely computed from the property value of a single vertex u because edge $e(u, v)$ is critical to obtaining v's precise result. For example, for single-source shortest path query, the shortest path from s to v is determined by the vertex u if edge $e(u, v)$ belongs to the shortest path from s to v. *Thus, removing of all other incoming edges of v from the graph will not impact the precision of v's result. Removal of subsets of incoming edges of all vertices results in a small proxy graph that produces highly precise results.* Furthermore, critical edges are determined by

the graph structure and edge weights; thus, they are critical for computing precise results for many different queries of the same kind, i.e. they are *recurring*. Consequently, the above observations yield a Core Graph that satisfies both RQ1 and RQ2 requirements.

Consider finding the CG for use by single-source shortest-path (SSSP) queries. We note that the *edge betweenness centrality* (ebc), which is defined as the number of the shortest paths that contain an edge in a graph [**?**], can be used to identify edges that are important to SSSP queries. Each edge that has non-zero ebc value should be included in the CG as it plays a role in establishing a shortest path between at least a pair of vertices. If all non-zero centrality edges are included in the CG, the graph remains well connected via shortest paths, i.e. if there is a path between a pair of vertices in the original graph, then the shortest path is also present in the CG. However, identifying all edges with non-zero ebc values is extremely expensive – it requires computing shortest paths from every vertex to every other vertex.

Core graphs are useful for efficiently solving queries over large graphs on systems with limited memory including: a GPU where the full graph cannot be held in GPU memory and thus the overhead of repeated graph transfers between host and GPU memory is substantial; and a shared-memory out-of-core system where the full graph cannot be held in the memory and thus the overhead of repeated graph transfers from disk and memory is substantial. Recent research has led to systems with reduced graph transfer overheads for GPU-based [45, 23, 70, 38, 66] and Out-of-Core [41, 105, 65, 87, 99] systems. Nevertheless, Core Graphs can substantially improve performance of Subway [66] for GPUs and GridGraph [105] for out-of-core processing. Even for Ligra [73] where the entire graph

is held in memory, Core Graph can significantly reduce computation performed and yield performance improvements.

The key contributions of our work are as follows:

- **Recurring Critical Edges Phenomenon** (§2): Our study of multiple kinds of graph queries on irregular graphs shows that the solution of a query is determined by a small fraction of total edges, i.e. critical edges. Also, many edges recur frequently across critical edge sets of different queries.

- **Core Graph Identification and Exploitation**: We present algorithms for finding a core graph by solving a small set of queries to identify most non-zero centrality edges. We exploit CG and present a new optimization that improves the efficiency of the 2phase evaluation while producing 100% precise results.

- **Experimental Results** (§3): For the 2.586 billion FR graph, across six kinds of queries, our approach yielded CGs containing 5.42% to 10.45% edges and precise results for 97.1–99.9% vertices. Across six kinds of queries and four large input graphs our approach outperforms Subway [66] by up to 4.48×, GridGraph [105] by up to 13.62×, and Ligra [73] by up to 9.31× for computing precise results for all vertices.

4.2 Recurring Critical Edges

In this section we present the key concepts that, for a given large graph, enable identification of a subgraph that is both small in size and yet produces highly precise results for different kinds of queries. We will show that for a large class of vertex queries

the following observations hold:

- A small fraction of edges, named *critical edges*, are responsible for producing a query's results; and
- Across different vertex queries, same critical edges are found to be *recurring*.

In the remainder of this section we will define the notion of critical edges and carry out a study to demonstrate their characteristics.

4.2.1 Critical Edges and their Characteristics

Consider an input graph, $G(V, E)$, where edges in E are directed, and edges may or may not have weights. An edge from vertex u to vertex v is denoted as $e(u, v)$ and, when edges have weights, $w(u, v)$ denotes the edge weight. A directed path from u to v is denoted by $p(u, v)$.

The graph algorithms we consider solve different kinds of vertex-specific queries. A vertex query $Q(s)$ originates at the source vertex $s \in V$ and upon its evaluation has computed the property values $Q(s).Val(v)$ for all other vertices $v \in V - \{s\}$.

Given a source vertex s and a path $p(s, v)$, the property value of v along the path, denoted as $p(s, v).Val(v)$ is computed from $Val(s)$ and the weights of edges along path $p(s, v)$ using a **propagation** operator $\bigoplus$. For example, given the path $p(s, v) = s \rightarrow u \rightarrow v$, $p(s, v).Val(v)$ is given by $Val(s) \bigoplus w(s, u) \bigoplus w(u, v)$.

For the class of graph queries we consider, given multiple paths $p_1, p_2, \cdots p_n$ from s to v, the property value of v corresponding to query $Q(s)$, denoted as $Q(s).Val(v)$, is computed from $p_i(s, v).Val(v)$ values for all p_i's using a **selection** operator, viz., one of:

$MIN_i(p_i(s,v).Val(v))$ *or* $MAX_i(p_i(s,v).Val(v))$.

Once $Q(s)$ has been evaluated, $Val(v)$ for all $v \in V - \{s\}$ represents the query solution. For push-style propagation, the processing of a single edge $e(u,v)$ is expressed below; moreover, update of $Val(v)$ must be performed *atomically*:

$$atomic\,\{\, Val(v) \longleftarrow \bigodot\ (\, Val(v),\ Val(u) \bigoplus w(u,v)\ )\ \}$$

Many graph algorithms fall in this category including the six graph algorithms used in our evaluation. Note that 2Phase algorithm is also applicable to other graph algorithms where the two phase algorithm produces correct results (e.g., PageRank). However, for problems where update does not use selection, benefits are limited.

Definition 1 – The Critical Edge Set with respect to query Q(s) is denoted as $E_c \setminus Q(s)$. Given a graph $G(V,E)$ and vertex query $Q(s)$, where $s \in V$, the *Critical Edge Set* $E_c \setminus Q(s)$ is given by:

$$E_c \setminus Q(s)\ = \{\ e(u,v)\ st\ \ Val_s(u) \bigoplus w(u,v) == Val_s(v)\ \}$$

Each critical edge $e(u,v)$ is responsible for determining the final stable value of the destination vertex v, i.e. $\bigodot$ selects u to provide the final value $Val_s(v)$ at v. **Note that a vertex v can have multiple incoming critical edges if they all produce the same final value at v.** As an example, for SSSP, $\bigoplus$ is the + operator and $\bigodot$ is the selection operator MIN. Similarly, for SSWP, $\bigoplus$ is the MIN operator and $\bigodot$ is the selection operator MAX.

We further observe that a critical edge set $E_c \setminus Q(s)$ has the following two properties:

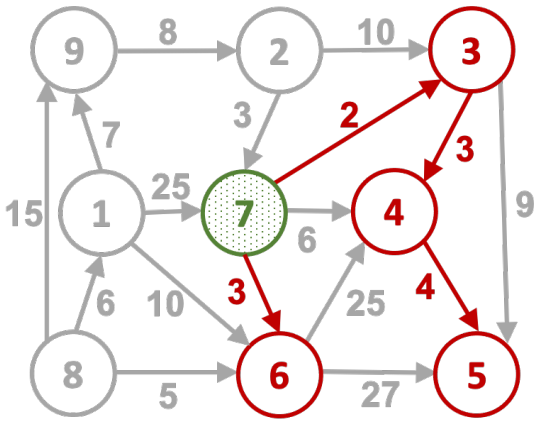


Figure 4.3: Critical Edge Set for query SSSP(7).

- *s-Reachability Preservation*: Any vertex that is reachable from s by following edges in E is also reachable from s via edges in $E_c \backslash Q(s)$.
- *s-Property Preservation*: The solution to $Q(s)$ computed using set of edges in E is identical to the solution computed using edges in $E_c \backslash Q(s)$.

The above discussion shows that after solving a query, its results can be efficiently *postprocessed* to identify critical edges and thus we can derive a **smaller graph which answers that query precisely**. While edges in this smaller graph are those in $E_c \backslash Q(s)$, the vertices include the following:

$$V_c = \{v \; st \; \exists \; e(*, v) \; or \; e(v, *) \in E_c\}$$

The edges in $E_c \backslash Q(s)$ are called ***critical edges***, the remaining edges in $E - E_c \backslash Q(s)$ are called ***non-critical edges***. Figure 4.3 illustrates the critical edge set corresponding to the shortest path query originating at source vertex 7. The red edges are critical edges while all remaining edges are non-critical with respect to query SSSP(7).

Definition 2 – Critical Edge Set with respect to a set of queries S is denoted

as $E_c \setminus S$. The critical edge set with respect to a ***set of queries*** for source vertices $S = \{s_0, s_1 \cdots s_n\}$ is computed as the union of the critical edge sets of queries in S as defined below:

$$\mathcal{E}_c \backslash S = \bigcup_{i=0..n} E_c \backslash Q(s_i).$$

The critical edge sets of multiple queries of same or different kind display the following characteristics:

Observation 1 – The Critical Edge Set corresponding to a given query Q contains a *small* fraction of edges in the graph;

Observation 2 – The Critical Edge Sets of a set of queries of the *same kind* greatly overlap, that is, many edges *recur frequently* in critical edge sets; and

Observation 3 – The Critical Edge Sets of a set of queries of *similar (but not same) kind* greatly overlap. In particular, SSSP, SSNP and Viterbi all are influenced by low weight edges and thus such edges *recur frequently* in their critical edge sets.

4.2.2 A Study of Critical Edge Sets

Next, we present empirical evidence that confirms the above *recurring critical edges* phenomenon, i.e. a small fraction of edges that constitute critical sets for a given query recurs frequently across critical edge sets of other queries, of the same or similar kind. We carry out a study based upon five different kinds of queries and four graphs described in Table 4.1.

In gathering this data we evaluated a set of queries that originate at highest degree vertices in the graph as such queries are expected to reach visit most of vertices and edges in

Table 4.1: Number of Edges, Vertices, and the Size of Unweighted–Weighted Graph Versions of Input Graphs: Friendster – FR; Twitter – TT; Twitter – TTW; PokeC – PK.

	| E |	| V |	Size (MB)
FR [19]	2,586,147,869	68,349,467	20,963
TT [7]	1,963,263,821	52,579,683	15,916
TTW [40]	1,468,365,182	41,652,231	11,914
PK [78]	30,622,564	1,632,804	252

Table 4.2: %age of all edges included in combined Critical Edge Set for 20 high-degree vertex queries. Union3 combines SSSP, SSNP and Viterbi critical edge sets.

G	SSSP	SSNP	Viterbi	SSWP	REACH	Union3
FR	6.7%	5.4%	5.3%	5.4%	4.1%	8.99%
TT	4.9%	5.2%	5.1%	5.3%	4.6%	8.11%
TTW	5.7%	5.2%	5.0%	5.2%	6.1%	8.74%
PK	13.3%	8.2%	7.7%	11.5%	8.4%	16.68%

the graph and hence produce largest critical edge sets. By comparing the edge sets obtained from different queries of the same kind as well as queries of different kinds, we study the degree of recurrence.

First, in Table 4.2 we show the critical edge set size for a set of high degree vertices in terms of the percentage of edges from the full graph that are included in it. The critical edge set size for the set of queries ranges from 4.1% to 6.7% for FR which is the largest graph. On the other hand, for the smallest graph PK, the size ranges from 7.7% to 13.3%. Thus this data confirms **Observation 1**, i.e. **the critical edge sets contain only a small fraction of all edges in the graph.**

Second, we show that same edges recur frequently across critical edge sets of different queries of the *same kind*. In Figure 4.5 we show the distribution of edges according

to the number of queries whose critical edge sets they appeared in. As we can see, for three kinds of queries (SSWP, Viterbi, SSNP) nearly all the edges appearing in the critical edge sets of different queries are the same. For SSSP we observe that most edges appear in multiple critical edge sets with sizeable number also appearing in all sets or one critical edge set. This also holds for REACH though the distribution varies across different input graphs. Overall, this data supports **Observation 2**, i.e. **critical edge sets of a set of queries of the same kind overlap substantially**.

Finally, we consider recurrence of critical edges present across queries of *similar* but not the same kind. We present the union of the critical edge sets for queries of three kinds whose results are influenced by lower weight edges (i.e., SSSP, SSNP, Viterbi). Table 4.2

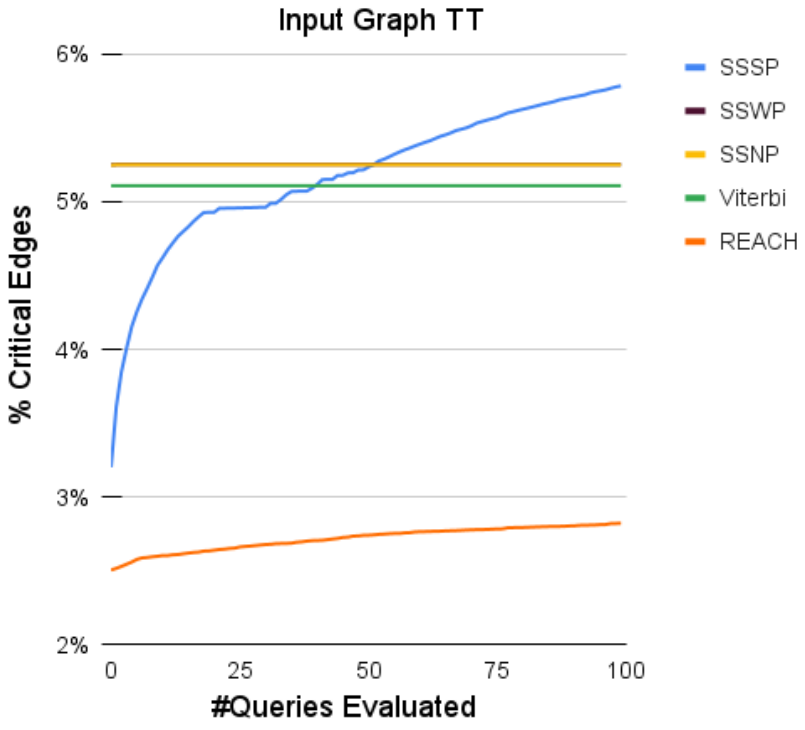


Figure 4.4: Cumulative %age of Critical Edges identified with increasing number of queries for TT input graph.

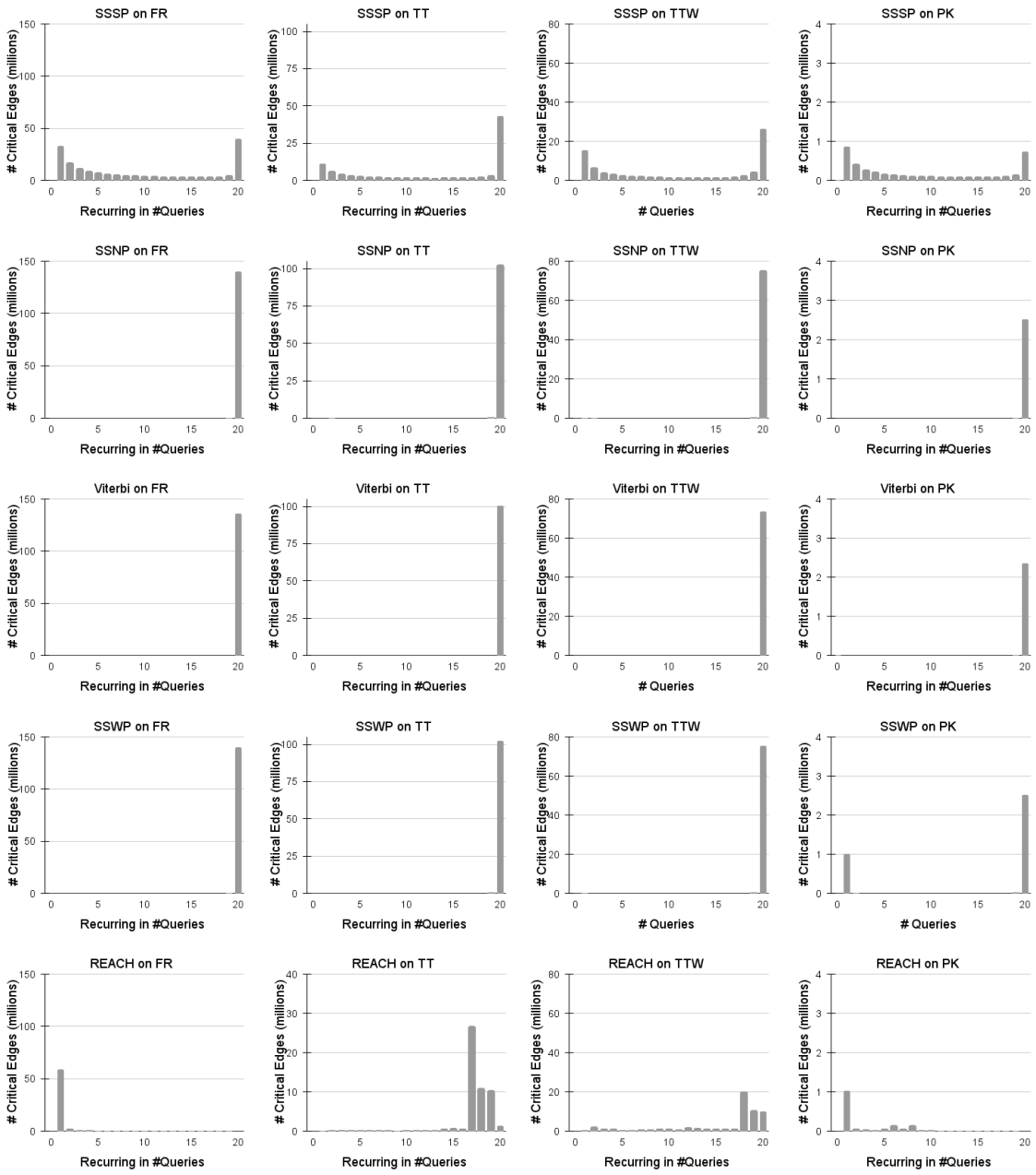


Figure 4.5: Number of critical edges recurring in exactly 1, 2, $\cdots$ 20 queries. The total number of these edges is a small fraction of all edges in the full graph as indicated by the percentages in Table 4.2.

shows that their combined size, **UNION3**, is only slightly bigger than the size of largest critical edge set among the three kinds of queries considered. In fact over 50% of edges in **UNION3** are critical edges that are common to the three kinds of queries. This data confirms **Observation 3**, i.e. **critical edge sets of a set of queries of similar kinds overlap substantially**.

Our key insight is that most edges with non-zero betweenness centrality can be identified by evaluating a small number of queries corresponding to the highest degree vertices in the graph. We can identify edges that lie along shortest paths found via these queries. That is, each edge $u \rightarrow v$ such that value of vertex v equals value of vertex u plus the weight of edge $u \rightarrow v$, lies along some shortest path. This simple approach is not only inexpensive, since it selects edges along shortest paths, it selects edges with non-zero centrality and edges that provide well connectedness. Figure 4.4 shows that as we solve increasing number of queries, inclusion of newly discovered edges found to fall on shortest paths causes the number of edges in the CG grow very slowly. That is, vast majority of edges included play an important role in the evaluation of many different queries. Moreover, the data for TT graph in Figure 4.4 shows that this observation is true across many graph algorithms to a very large degree. Table 4.3 shows that, when edges are identified using 20 forward queries on TT, most edges are frequently selected for inclusion by majority of the queries – similar behavior was also observed for other input graphs. Therefore, in rest of the paper we limit the number of vertices queried for identifying CG edges to 20.

We also considered larger number queries, up to 100, to observe the growth in critical edge set size with increasing number of queries. In Figure 4.4 we show how the size

Table 4.3: Average number of queries out of a total of 20 forward queries that select an edge added to CG.

G	SSSP	SSNP	Viterbi	SSWP	REACH
FR	12.12	19.99	20.00	19.86	1.06
TT	13.01	19.49	20.00	19.99	17.50
TTW	11.14	19.94	20.00	19.96	16.24
PK	8.95	19.99	19.99	14.57	2.65

grows with the query set size of high degree vertices evaluated to find critical edges. As we can see, for SSWP, Viterbi, and SSNP, the critical set size is mostly flat. The number of critical edges grows by a very small number with additional queries. Only in the case of REACH and SSSP, the critical edge set continues to grow though the growth continues to slow with increasing number of queries.

4.3 Identifying Core Graph

We have thus far considered critical edge sets that correspond to a single specific query or a set of specific queries. However, we would like to find a Core Graph that can be used to evaluate all possible queries of a given kind or even queries of multiple kinds. Ideally we would like to find a graph that is *complete*, that is, it is not only a subgraph of the full graph but it also preserves the properties for all vertex queries that are of interest as defined next.

4.3.1 Edge Centrality and Complete Core Graph.

As a consequence of evaluating a query $Q(s)$, it is possible to identify all edges belonging to solution paths (e.g., shortest paths). That is, we can identify every edge that belongs to some solution path $p(s, v)$ such that $p(s, v).Val(v) == Q(s).Val(v)$. This is

because for any edge $e(a, b)$ belonging to a solution path $p(s, v)$, property values of a and b are related as follows:

$$Q(s).Val(b) \;==\; Q(s).Val(a) \bigoplus w(a, b).$$

All edges that belong to a solution path have non-zero centrality values, i.e. they belong to at least one solution path. For a given graph $G(V, E)$, we introduce the notion of the **Complete Core Graph** $cCG(V_c, E_c)$ that is defined as follows.

Definition 3 – Complete Core Graph (cCG) is a subgraph of G which contains all vertices from G and all non-zero centrality edges in G, i.e.,

$$V_c = V; \qquad E_c = \{\, e(a, b) \mid \; Centrality\; of\; e(a, b) > 0 \,\}$$

Since the above definition states that any edge with non-zero centrality is included in cCG, it implies that any path $p(x, y)$ for which $p(x, y).Val(y)$ is equal to $Q(x).Val(y)$ in G, is also present in cCG. Thus, the evaluation of any query on cCG produces results that are identical to those produced by evaluating the query on G.

Note that finding the cCG does not require computing the exact centrality value of each edge $e(a, b)$ but rather it simply requires identifying edges with non-zero centrality. If upon solving some query $Q(s)$ we observe that

$$Q(s).Val(a) \bigoplus w(a, b) = Q(s).Val(b)$$

then edge $e(a, b)$ definitely has non-zero centrality. Nevertheless we observe that it is not practical to identify the Complete Core Graph. By solving a single query $Q(s)$, we can identify only the subset of non-zero centrality edges that play a role in computing the solution of query $Q(s)$. However, to identify all non-zero centrality edges, in general all queries must be evaluated. Therefore, next we present a heuristic for finding an incomplete core graph

that is nevertheless very effective in producing highly precise results.

However, we observe that it is not practical to compute the exact cCG because it would require solving all (tens of millions) possible queries to identify their critical edges. In addition, the resulting graph for some query types may also be large, i.e. it may contain majority of the edges from the full graph even ones that are rarely part of the solution. However, our goal in finding a smaller graph is to use it to solve queries with limited memory and with higher efficiency. Therefore, motivated by the phenomenon of recurring critical edges, we settle on building an *incomplete core graph* that is computed by solving a *small set of selected queries*. We refer to this graph as a Core Graph (CG) that we will use to evaluate queries at a reduced cost without a guarantee of 100% precision. That is, the CG produces prices results for most of the vertices while precise results for remainder of vertices are computed using FG.

Next we present two algorithms for identifying (incomplete but practical) core graphs. The first algorithm constructs a *specialized* core graph for evaluating all queries of a given kind. The second constructs a *general* core graph for queries of multiple kinds.

4.3.2 Specialized CG: For Queries of One Kind

A specialized CG includes *all vertices* from the full graph but only a *small subset of its edges*. For constructing a specialized (incomplete) CG, we develop a heuristic to identify the majority of frequently occurring critical edges. The set of critical edges identified for inclusion in CG preserve pairwise vertex ***reachability***, for those that are reachable within FG, with a *high likelihood* as defined below. Doing so ensures that queries involving an arbitrary vertex that succeeds in FG, will also succeed in the specialized CG.

If there is a path from u to v in the full graph $G(V, E)$, then there is a high likelihood that there is also a path from u to v in core graph $CG(V_c, E_c)$ which is composed of identified critical edges.

Note that since core graph is a subgraph of the full graph, if there is no path from u to v in the full graph, then there will not be any path from u to v in the core graph.

Since our goal is to accelerate the solving of queries in the first place, we settle on building an incomplete **Core Graph** that is computed by solving a small set of selected queries (we found 20 vertices are adequate) and use this graph to speedup the evaluation of all future queries. Henceforth we refer to this incomplete core graph simply as the Core Graph (CG) which contains a subset of all non-zero centrality edges. As shown in earlier in Figure 4.4, when we identify sets of non-zero centrality edges of different queries, there is very large overlap in these sets which implies that most of the non-zero centrality edges being found belong to many soultion paths. We will soon show that our approach for building the CG produces exact results for most vertices, further confirming the inclusion of most non-zero centrality edges in CG, while the exact results for remainder of vertices require further computation using the original full graph to account for the non-zero centrality edges that are missing from CG and belong to some solution path for the query being evaluated.

Forward and Backward Queries. Our work is aimed at large graphs with power law degree distribution. For such graphs, it is known that high degree vertices are good proxies for high centrality vertices [**?**]. Thus, a small number of highest degree vertices are used to identify edges with non-zero centrality. Given a chosen high-degree vertex h, we can find high centrality edges by solving query $Q(h)$ and then testing each

edge for non-zero centrality. Therefore, by choosing edges along paths passing through a small number of high-degree vertices we can, with *high likelihood*, preserve the reachability condition described above. As a result a CG obtained is expected to be useful for solving *all possible queries*. In a directed graph, we actually solve two queries corresponding to each chosen high-degree vertex h: $Q(h, forward)$ and $Q(h, backward)$. The forward query identifies non-zero centrality edges that lie along paths originating at h and leading to some other vertex. The backward query identifies non-zero centrality edges that lie along paths originating at some other vertex and leading to h. **By computing both forward and backward queries we are able to preserve pairwise reachability among vertices, via h, to a very large extent and thus producing a well connected CG.**

Additional Connectivity Edges. Once non-zero centrality edges corresponding to a small set of high degree vertices have been found, we ensure that every vertex with non-zero out-degree has at least one edge included in the core graph to make the graph well connected. **If no outgoing edge is included, we add one**. For SSSP (SSWP) lowest (highest) weight edges are chosen as they are more likely to belong to shortest (widest) paths. This approach ensures that each vertex is connected to the CG.

Algorithm 3 shows the above computation. The algorithm repeatedly uses different high degree vertices in H to find additional non-zero centrality edges. One by one queries of the specific kind are evaluated in both forward and backward directions originating at vertices in S. The critical edges are identified from the results of the queries and added to the critical set. Note that typically $CG(V_c, E_c) \backslash S$ includes all the vertices in V, i.e. $V_c = V$. Finally, it ensures that at least one out edge of each vertex with non-zero out

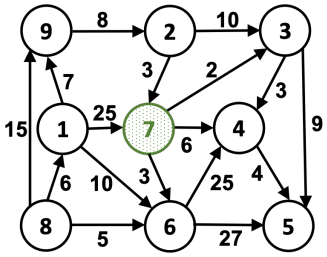
degree is added to E_c. And when the incomplete Core Graph as constructed above is used to solve a new query, it will produce exact results for some vertices but not for all vertices. Next we illustrate the above algorithm and observations.

Example. We illustrate the above algorithm by building a specialized core graph for the shortest path problem for the example graph in Figure 4.6(a). The red and blue edges in Figures 4.6(b) and (c) are found by solving queries SSSP(7,f) and SSSP(7,b). The core graph obtained by combining the two is shown in Figure 4.6(d). The two tables in Table 4.4 show results of all shortest path queries computed using the full graph (FG) and the core graph (CG).

Note that most of the results in the two tables are identical. Only four results shown in red do not match. First, the reachability for vertex pairs $(6, 4)$ and $(6, 5)$ is present in FG but not in CG causing the values computed using FG to be ∞. However, since no outgoing edge for vertex 6 is present, we will include the lowest weight outgoing edge from 6 to 4 to the critical set. This will cause the values for vertices 4 and 5 to change to 25 (precise) and 29 (imprecise). Second, although reachability for pairs $(8, 5)$ and $(8, 6)$ is satisfied by both graphs, in CG the lengths of the paths is longer than the shortest paths present in FG. Since CG contains only a subset of FG's edges, shortest path length for a vertex pair computed using FG can only be shorter than that computed via CG.

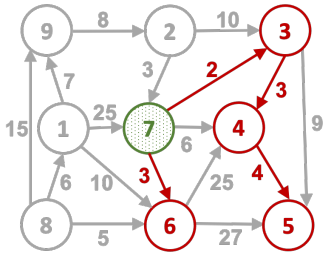
4.3.3 General CG: For Queries of Multiple Kinds

Next we present a heuristic for building one core graph for use in evaluating queries of multiple kinds. We observe that many of the common graph queries can be divided in



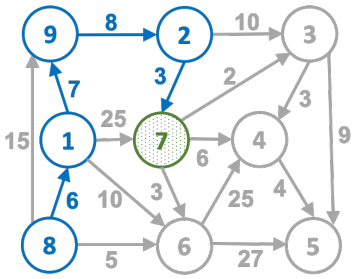
(a) **Full Graph** (FG).

$7 \rightarrow *$	1	2	3	4	5	6	7	8	9
7	∞	∞	2	5	9	3	0	∞	∞

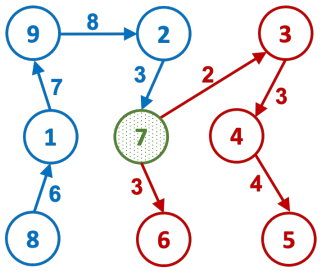


(b) **Forward Critical Edge Set** $E_c^f \backslash SSSP^f(7)$.

$* \rightarrow 7$	7
1	18
2	3
3	∞
4	∞
5	∞
6	∞
7	0
8	24
9	11



(c) **Backward Critical Edge Set** $E_c^b \backslash SSSP^b(7)$.



(d) **Core Graph** derived from $SSSP^{f+b}(7)$.

Figure 4.6: Illustration of Algorithm 1: Starting from Full Graph in (a) to derived Specialized Core Graph in (d).

Algorithm 3 Identifying Specialized Critical Edges.

Input: Full Graph $G(V, E)$ and Query Set S

Output: Critical Edges E_c

E_c^f = IDENTIFY ($G(V, E)$, DIRECTION f)

E_c^b = IDENTIFY ($G^R(V, E^R)$, DIRECTION b)

$E_c = E_c^f \cup E_c^b$

function IDENTIFY ($G(V, E)$, DIRECTION d)

  for $s \in S$ **do**

    Evaluate Query $Q(s, d)$ on $G(V, E)$

    for all $e(u, v) \in E$ **do**

      if $Val_s(u)$ is updated by $Q(s, d)$ **then**

        if ($Val_s(u) \bigoplus w(u, v) == Val_s(v)$) **then**

          if ($d == f$) **then**

            $E_c = E_c \cup \{ e(u, v) \}$

          else ▷ ($d == b$)

            $E_c = E_c \cup \{ e(v, u) \}$

          end if

        end if

      end if

    end for

  end for

end function

three categories: (a) queries that depend only on reachability (REACH, Weakly Connected Components–WCC); (b) queries that depend on reachability and lower weight edges (SSSP,

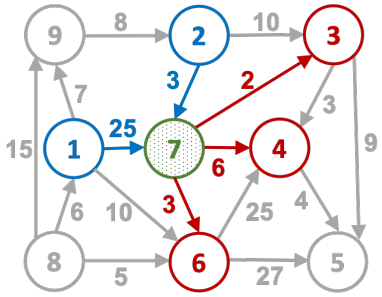
Table 4.4: Shortest Paths Computed: Using the Full Graph with 17 Edges (Top) vs. Core Graph with 8 Edges (Bottom).

FG	1	2	3	4	5	6	7	8	9
1	0	**15**	**20**	**23**	**27**	**21**	**18**	∞	**7**
2	∞	0	**5**	**8**	**12**	**6**	**3**	∞	∞
3	∞	∞	0	**3**	**7**	∞	∞	∞	∞
4	∞	∞	∞	0	**4**	∞	∞	∞	∞
5	∞	∞	∞	∞	0	∞	∞	∞	∞
6	∞	∞	∞	**25**	**27**	0	∞	∞	∞
7	∞	∞	**2**	**5**	**9**	**3**	0	∞	∞
8	**6**	**21**	**26**	**29**	**32**	**5**	**24**	0	**13**
9	∞	**8**	**13**	**16**	**20**	**14**	**11**	∞	0

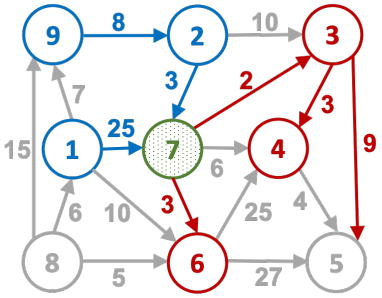
CG	1	2	3	4	5	6	7	8	9
1	0	**15**	**20**	**23**	**27**	**21**	**18**	∞	**7**
2	∞	0	**5**	**8**	**12**	**6**	**3**	∞	∞
3	∞	∞	0	**3**	**7**	∞	∞	∞	∞
4	∞	∞	∞	0	**4**	∞	∞	∞	∞
5	∞	∞	∞	∞	0	∞	∞	∞	∞
6	∞	∞	∞	∞	∞	0	∞	∞	∞
7	∞	∞	**2**	**5**	**9**	**3**	0	∞	∞
8	**6**	**21**	**26**	**29**	**33**	**27**	**24**	0	**13**
9	∞	**8**	**13**	**16**	**20**	**14**	**11**	∞	0

SSNP, Viterbi); and (c) queries that depend on reachability and higher weight edges (SSWP).

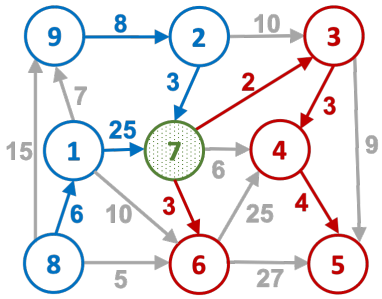
Since queries from all three categories rely on reachability characteristics, to capture these characteristics we can identify critical edges that belong to forward and backward breadth-first-trees corresponding to a set of high-degree vertices. Then, to consider edge weights, we can bias the selection of critical edges in favor of lower weight edges (for second category) or higher weight edges (for third category). Next, we first describe an algorithm for finding a small core graph that captures reachability characteristics of the graph without considering edge weights. After that we show how this algorithm can be adapted to consider edge weights.



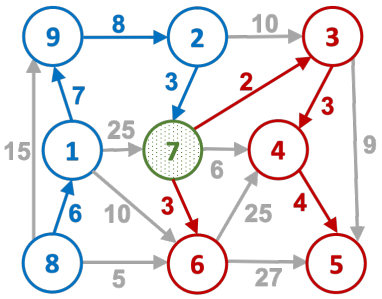
(a) Forward from 7 and Backward from 7.



(b) Forward from 3 and Backward from 2.



(c) Forward from 4 and Backward from 1.



(d) Forward from 6 and Backward from 9.

Figure 4.7: An example of General Core Graph. Each step shows forward and backward traversal from a node. Note forward (backward) traversal from 5 (8) does not traverse any edges as these vertices have no outgoing/incoming edges respectively.

Common Critical Edges. We note that when constructing a core graph that captures reachability characteristics via forward and backward BFS-traversals, same edges can be chosen by traversals originating at different high-degree vertices to the extent possible in order to produce smaller core graphs. Algorithm 4 takes advantage of this sharing in identifying critical edges. In this algorithm, QID(v) contains the id of the high-degree query vertex that is the first to add an incoming critical edge of v in the set of critical edges CS. When an edge $e(u, v)$ is encountered whose source and destination vertex QID's are

Algorithm 4 Critical Edges for Graphs With No Weights.

1: **Input**: Full Graph $G(V, E)$ and Query Set S

2: **Output**: Critical Edges E_c

3: QID[*] $\leftarrow 0$ for each vertex

4: $E_c^f \leftarrow E_c^b \leftarrow \phi$

5: **for** $s \in S$ **do**

6: $E_c^f = E_c^f \cup$ TRAVERSE (s, $G(V, E)$)

7: $E_c^b = E_c^b \cup$ TRAVERSE (s, $G^R(V, E^R)$)

8: **end for**

9: $E_c = E_c^f \cup$ REVERSE (E_c^b)

10: **function** TRAVERSE (s, $G(V, E)$)

11: $CE = \phi$; FIFO.PUSH(s);

12: **while** ! FIFO.EMPTY() **do**

13: $u \leftarrow$ FIFO.POP()

14: **for all** $e(u, v) \in Graph.Outedges(u)$ **do**

15: **if** QID(v) $\neq s$ **then** ▷ add $e(u, v)$ to CE

16: $CE = CE \cup \{ e(u, v) \}$

17: **if** QID(v)=0 **then**

18: FIFO.PUSH(v); QID(v) $\leftarrow s$

19: **end if**

20: **end if**

21: **end for**

22: **end while**

23: return (CE)

24: **end function**

different, then the edge is added to the graph but all critical edges emanating from v onward are reused by queries labeling vertices u and v.

Considering Edge Weights. To further consider the influence of weights, we can build BFS trees that are biased towards inclusion of edges with lower weights (e.g., for SSSP) or higher weights (e.g., for SSWP). To achieve this behavior we can modify the BFS algorithm so that when a vertex is visited the first time, the incoming edge e along which it is visited is included in the tree; however, during a future visit along another incoming edge e', if incoming edge e' has the lower weight (e.g., for SSSP) or higher weight (e.g., for SSWP) than e, then we replace e by e'. The levels in the BFS tree are used to ensure edge replacements do not disconnect the tree (i.e., a replacement must go from lower to higher level vertex).

Example. Figure 4.7 illustrates the identification of critical edges using forward-BFS (shown in red) and backward-BFS (shown in blue) from vertex 7. The edges with lower weights are given preference for inclusion in the critical edge set. Next, we illustrate the traversals in both directions.

When we carry out *forward traversal* from 7, edges (7,3), (7,4) and (7,6) are initially added to the critical set. Next traversal from vertex 3 adds lower weight edge (3,4) as a replacement for edge (7,4) and in addition edge (3,5) is added. Next traversal from vertex 4 adds lower weight edge (4,5) as a replacement of edge (3,5). Finally traversal from vertex 6 does not add or replace any edges as both outgoing edges of 6 have high weights. Vertex 5 has no outgoing edges. Thus, the critical edges are (7,3), (7,6), (3,4), and (4,5).

When we carry out *backward traversal* from 7, edges (2,7) and (1,7) are initially

added to the critical set. Next backward traversal from vertex 2 adds edge (9,2) and backward traversal from vertex 1 adds edge (8,1). Finally backward traversal from vertex 9 adds edge (1,9) as a replacement of edge (1,7). Vertex 8 has no incoming edges. Thus, critical edges identified are (2,7), (9,2), (8,1), and (1,9).

In carrying out replacement of edges we must ensure that the starting vertex (i.e., vertex 7 in the example) has at least one incoming and one outgoing edge as critical. Finally, we observe that the core graph shown in Figure 4.7(d) is identical to the one found earlier in Figure 4.6(d). However, in general this may not be the case.

4.4 Core Graph Sizes and Precision

Core Graph Sizes. In Table 4.5 we present the percentage of all edges that are present in specialized core graphs for SSSP, SSNP, Viterbi (where lower weight edges are more important), and SSWP where higher weight edges are more important. Finally in GENERAL weights play no role and this graph is essentially the General CG built using Algorithm 4. For the three large input graphs – FR, TT, TTW – the core graphs contain 7.27% to 13.77% of total edges. For the smaller PK graph this number is higher. **The average over all twenty core graphs generated is 10.7%**.

We also note that core graphs that are constructed by solving both forward and backward queries contain nearly twice the number edges than those constructed only from only forward queries (Table 4.5 vs. Table 4.2). This is the cost of ensuring that the core graph is useful for solving all queries. Across the four graphs, time for identifying the core graphs using Subway [66] for the most expensive Viterbi queries is 1 to 30 minutes and for

Table 4.5: (Top) % of Total Edges in the Specialized and General Core Graphs Computed from 20 High-degree Vertices. Overall average is 10.7%; (Bottom) MB of Memory Needed to Hold weighted CG.

CG (%)	SSSP	SSNP	Viterbi	SSWP	REACH
FR	10.45%	7.27%	7.33%	7.27%	5.42%
TT	9.36%	7.71%	7.73%	7.71%	7.02%
TTW	10.10%	13.77%	8.34%	13.58%	8.34%
PK	21.85%	18.05%	12.14%	18.18%	12.13%
RMAT1	2.78%	2.70%	12.61%	2.70%	3.95%
RMAT2	1.68%	1.65%	7.82%	1.65%	3.17%
RMAT3	3.05%	2.98%	21.29%	2.89%	5.17%

CG (MB)	SSSP	SSNP	Viterbi	SSWP	REACH
FR	2,436	1,777	1,789	1,777	834
TT	1,680	1,421	1,425	1,425	762
TTW	1,353	1,784	1,146	1,762	656
PK	60	51	36	51	21

the GENERAL graph it is 1 to 7 minutes.

In contrast, CGs provide highly precise results for the following reasons. First, since CGs are built from query results, they include paths formed by non-zero centrality edges giving us good connectivity. Second, as shown by a representative plot in Figure 4.8, the degree distributions of FG and CG are similarly power law. Third, as shown in Table 4.8, although the degrees of high degree vertices in CG are reduced, their relative degrees remain unchanged (e.g., the top 1000 vertices in CG and full graph are exactly the same).

Precision of Results from CGs. Though the core graph contains a very small fraction of edges from the full graph, it truly captures its essence. When we evaluated ten random queries for every combination of graph algorithm and input graph, **we found that on average for 94.5–99.9% of the vertices the core graph produces precise results**

Table 4.6: Average % of Vertices for which CG Produces Precise Results for 10 Random Queries.

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	97.1%	99.9%	99.9%	99.9%	99.9%	99.4%
TT	99.6%	99.9%	99.9%	99.9%	99.9%	99.9%
TTW	99.4%	99.9%	99.9%	99.9%	99.9%	98.7%
PK	94.5%	99.9%	99.9%	99.9%	99.9%	99.3%
RMAT1	96.5%	99.9%	95.3%	99.9%	99.9%	99.9%
RMAT2	97.8%	99.9%	91.4%	99.9%	99.9%	99.9%
RMAT3	95.2%	99.9%	99.4%	99.9%	99.9%	99.9%

Table 4.7: Average Times in Milliseconds for Generating Nearly Precise Results using CG for 10 Random Queries.

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	675.3	149.1	521.7	538.7	108.24	325.5
TT	314.9	85.0	379.5	308.1	58.67	115.6
TTW	292.1	97.3	316.4	287.9	46.18	91.8
PK	22.2	4.2	26.2	26.0	2.14	9.3

Table 4.8: Degree of Overlap in Sets of Topmost 100,000 Highest Degree Vertices of Full Graphs and Corresponding Core Graphs for SSSP.

G	Common High Degree Vertices		
	Top 1,000	Top 10,000	Top 100,000
FR	1,000	9,997	99,931
TT	1,000	10,000	99,997
TTW	1,000	10,000	99,988
PK	1,000	10,000	98,988

(i.e., same result as the one produced by the full graph). This data is given in Table 4.6. Note that the REACH (Reachability) and WCC (Weakly Connected Components) query evaluations were performed using the GENERAL CG.

Across four kinds of queries (SSNP, Viterbi, SSWP, REACH) CG generated imprecise results for only a tiny number of vertices – a maximum of 310, 40, 36, and 79 vertices

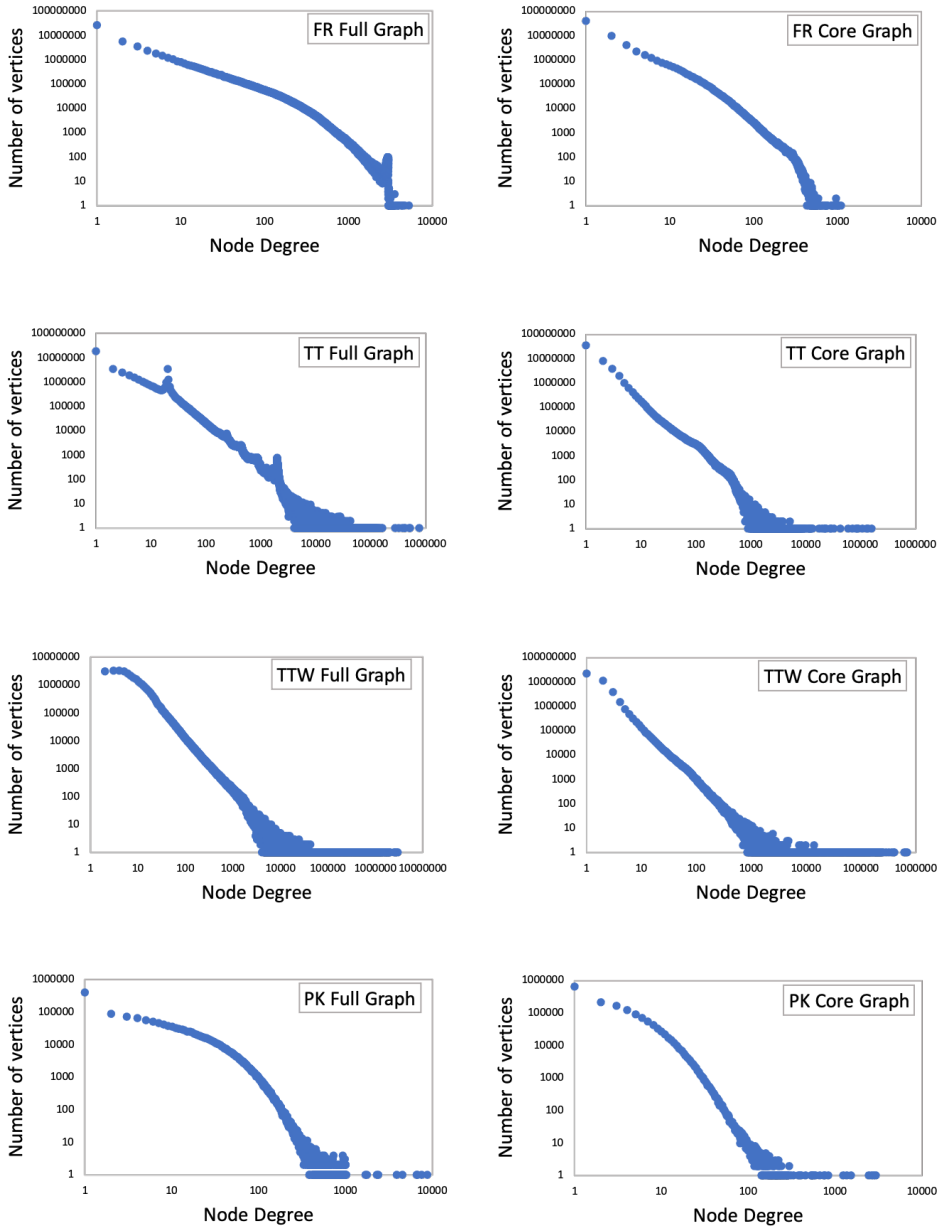


Figure 4.8: Power Law Degree Distributions of Full Graphs and Corresponding Core Graphs for SSSP.

for FR, TT, TTW, and PK respectively. For SSSP the fraction of vertices with imprecise results is the highest and the average percentage errors in the values for these imprecise

vertex values is 2.27%, 6.35%, 5.71%, and 3.79% for FR, TT, TTW, and PK. The times (in milliseconds) it takes to generate nearly precise results are given in Table 4.7.Finally, note that we also provide precision of WCC which is computed using the REACH's CG. The small CG sizes lead to low processing times (over an order of magnitude lower than the baseline for Subway), and should there be applications that can tolerate limited imprecision, they should be able to benefit from large improvements in performance by using the results produced by CG. We leave that exploration to future work.

The reason for high precision delivered by CGs is that in shrinking the size of the graph, some key characteristics of the graph are preserved. First, as shown in Figure 4.8, the degree distribution of both FGs and CGs follow similar power law distribution. Second, as shown in Table 4.8, although the degrees of high degree vertices in CG are reduced, their relative degrees remain unchanged. For example, for all four graphs, the top 1000 vertices in CG and full graph are exactly the same.

Precision of Abstraction Graphs Sampled Graphs vs. Core Graphs. We also studied the precision of Abstraction Graph (AG) for SSSP constructed according to the algorithm presented in [99]. This algorithm orders the edges according to increasing edge weights. First, pass over the edges adds those edges to the AG that connect two weakly connected components. Then, another pass includes additional edges till upper limit on number of allowed edges is reached – once again preference is given to lower weight edges. For fair comparison, we constructed AGs with equal number of edges as corresponding CGs and then compared their precision. The precision of AGs is given in Table 4.9. As we can see, the precision of AGs ranges from 6.1% to 69.9% while the precision of CGs ranges

Table 4.9: Precision of Abstraction Graph of size: AG - size equal to CG; 2AG - double the size of CG. Average % of Vertices with Precise Results for 10 Random Queries.

G		SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	AG	22.3%	52.4%	35.9%	52.7%	25.5%	9.4%
	2AG	36.2%	63.9%	62.7%	63.9%	44.6%	58.1%
TT	AG	34.4%	43.2%	27.9%	43.2%	26.9%	6.1%
	2AG	50.6%	61.6%	55.0%	63.4%	55.7%	6.2%
TTW	AG	29.0%	60.3%	23.7%	55.43%	43.5%	54.0%
	2AG	46.0%	77.9%	46.9%	77.67%	53.1%	67.8%
PK	AG	46.8%	69.9%	14.4%	71.8%	49.5%	59.6%
	2AG	73.5%	83.9%	44.7%	85.6%	62.3%	75.5%

Table 4.10: Precision of SGs of sizes: (SG) equal to CG; (2SG) double of CG. % Vertices with Precise Results for 10 Queries.

G		SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	SG-P	9.8%	15.2%	11.8%	12.7%	35.2%	38.1%
	2SG-P	11.2%	19.7%	15.1%	17.2%	39.7%	42.5%
TT	SG-P	8.7%	10.5%	15.5%	6.3%	41.8%	33.5%
	2SG-P	11.8%	14.1%	17.1%	10.1%	49.2%	35.3%
TTW	SG-P	10.4%	11.2%	18.8%	15.9%	34.8%	47.9%
	2SG-P	12.8%	14.8%	21.5%	18.1%	38.2%	51.7%
PK	SG-P	11.1%	14.9%	14.6%	17.8%	30.6%	48.5%
	2SG-P	15.0%	17.6%	19.5%	21.5%	32.7%	58.4%

from 94.5% to 99.9%. We also doubled the size of AGs (relative to CGs) to see study how the precision improves. However, as shown by the rows labeled 2AG in Table 4.9, though precision improves to 6.2% to 83.9%, it is still far lower than the precision of CGs.

Sampling techniques have been developed to scale down graph size [108, 106, 107]. We generated Sampled Graphs (SGs) using random walks [106] and used them in place of CGs for two phase evaluation of queries. The precision data of this approach is given in Table 4.10. We observe that overall the precision of SGs is even lower than AGs. This is because sampling does not guarantee creation of well-connected graphs.

Limitation. We would like to point out that the above observations hold for irregular graphs with power-law distribution. For other kinds of graphs, core graphs may have different forms and different degree of precision. Also, we have examined six graph properties, there may be other properties for which high precision may be difficult to attain. Finally, as mentioned earlier our work considers monotonic algorithms for vertex-specific queries.

4.5 Related Work on Proxy, Pruned, and Transient Graphs

A Proxy Graph used for evaluating all future queries. We provide additional comments on relevant works that derive smaller graph structure to speedup evaluation of all future queries over a large graph. As mentioned earlier, *input reduction* [39] employs property preserving graph transformations to reduce graph size and then uses 2Phase processing on reduce and then full graph. However, due to restrictive nature of safe transformations, graph size reductions are limited and so is the work that can be performed in the first phase. Best performance is reported when reduced graph has around 50% of the edges. Moreover, since parts of the graph are collapsed, the reduced graph can only be used to evaluate queries for subset of vertices in the full graph. Abstraction Graph [99] on the other hand delivers small proxy graph; however, it lacks precision as shown by results in Table 4.9. *In contrast, core graphs are much smaller (5.42–10.45% for FR) and yet produce accurate results for over 94% vertices, and leaving less work for the second phase.*

Query Specific Pruned Graph for Point-to-Point Queries. Our work focuses on evaluating vertex queries that originate at a source vertex and then compute property values

for all other vertices in the graph that are reachable from the source. There is another class of queries that compute a property value between a source and destination vertex pair. These are called point-to-point queries and as their first step perform graph pruning and then evaluate the query on the *pruned graph* [90, 93]. Unlike Core Graph, pruned graph must be recomputed for each new query. Due to limited scope of a point-to-point query, pruning parts of the graph that do not fall on paths from source to destination significantly reduces graph size. However, since our work is for point-to-all queries, pruning would only reduce the graph size minimally. Next we provide detailed comparison with two specific point-to-point query algorithms that employ pruning.

Query-by-Sketch (Qbs) [90] is a three-phase (offline labelling followed by query specific online sketching and searching) algorithm for finding of all shortest paths from one vertex to another (i.e., they solve a point-to-point query). Though core graph and Qbs speedup query evaluation over large graphs, Qbs has major limitations in comparison to core graph. First, a sketch is query specific and thus must be computed online for each query while the core graph is found once and used for all future queries. Second, core graph is *general* as it solves many kinds of queries as opposed Qbs that is for one kind of query – shortest path. Third, we evaluate demanding queries that compute property values from *one source vertex to all other reachable vertices* while Qbs evaluates a single point-to-point query. For the queries we evaluate, a sketch is expected to be very large fraction of the graph. Finally, not only is online sketching expensive, if the sketch produced is large in size, then computation of shortest path takes a long time. For TTW labelling and sketching takes 1,345 seconds and since sketch is large (0.76GB), the query evaluation time is also

high (164 seconds). *In summary, core graph construction is efficient, done only once and henceforth it gives fast query times for all queries – 9-16 seconds for TTW on a GPU.*

Pruning and Prediction (PnP) [93] is another method for evaluating point-to-point queries. This algorithm employs bidirectional BFS originating from source (forward) and destination (backward) to first prune the graph for a given query and then perform remainder of the query evaluation using the pruned graph. The pruning is query specific like the query sketch used by Qbs.

Query Specific Transient Graphs generated on-the-fly. Transient graphs are generated to minimize data transfer cost multiple times during the evaluation of a query. There are two contexts in which such work has been done: works considering graphs that do not fit in GPU memory; and out-of-core systems for graphs that do not fit in the memory of a single machine. Both these approaches can benefit from Core Graphs to reduce data transfers – between host and GPU-memory vs. disk and machine memory.

A number of approaches have been developed to reduce data transfer in context of a GPU [45, 23, 70, 38, 66]. Among them Subway is the most promising – it on-the-fly generates and loads *transient active subgraphs* covering the frontier from one iteration to next. Across iterations, the reachable graph is loaded at least once. *In contrast, a core graph is loaded in its entirety once and computes precise results for over 94% of vertices without requiring graph transfers at iteration boundaries.*

On shared-memory machines when graphs cannot fit in memory, out-of-core partition-based processing is employed [41, 105, 65, 87, 99]. Partitions are loaded from disk one at a time and processed. Typically the disk IO represents 70% of the runtime cost [87]. To

reduce disk IO, in [87] authors maximize the work performed on one partition before loading the next partition. Wonderland [99] organizes edges across partitions according to their weights so that fewer passes, and faster convergence, can be obtained. Nevertheless, the cost of disk IO is very high. *Out-of-core systems benefit from 2Phase since first phase loads the entire core graph and computes precise results for over 94% of vertices without additional IO.*

Eliminating redundant processing. As shown by our experiments, 2Phase processing on a CG eliminates redundancy. There are other works on eliminating redundancy via vertex scheduling (e.g. [75]). However, this work does not address the memory problem due to large graph size. Another approach for reducing redundant computation *across queries* uses results of a set of queries to accelerate future queries [32, 92, 33]. However, this approach increases memory requirements as, in addition to the entire graph, it must also keep results of a set of queries to be reused in memory. Finally, AsyncGraph [102] accelerates processing along preselected paths by increasing parallelism. It does not identify full graph structure or enable in-memory processing for computing results for vast majority of vertices. *Core graph eliminates substantial amount of redundancy by first using a small graph and thus provided speedups for in-memory system like Ligra.*

4.6 Summary

We identified the phenomenon of *recurring critical edges* and used it to identify a *core graph* with 10.7% of edges on average that rapidly yields precise results for 94.5–99.9% of vertices. In the next chapter we will present the details of the two pass algorithm where

the second pass over the full graph computes precise results for rest of the vertices leading to net reduction in execution times over baselines that only work using full graphs. Significant performance improvements for multiple existing systems will also be shown for six kinds of graph queries and for four different input graphs.

Chapter 5

Exploiting Core Graphs for Different Platforms

In this chapter we present a 2phase algorithm for precisely evaluating graph queries by first evaluating them on the core graph to obtain mostly precise results and then evaluating them on the full graph to quickly obtain fully precise results. We will then demonstrate the benefits of this approach for two platforms, out-of-core GridGraph system and in-memory Ligra system.

5.1 CG-based 2Phase Algorithm

Overview of using CG in the 2Phase approach is provided in Figure 4.1. The offline part finds the CG that is used to power a 2Phase algorithm (online part in Figure 4.1) that first evaluates the query on the small in-memory graph and then uses the results obtained to bootstrap the evaluation of the query on the full graph.

Table 5.1: Push Operations for Four Algorithms. Here, CASMIN(a; b) sets a = b if b < a atomically using compare-and-swap; CASMAX is similarly defined.

Alg	NEEDED $(e(u,v))$ EDGEFUNCTION $(e(u,v))$
SSWP	$Val(v) < min(Val(u), wt(u,v))$ $CASMAX(Val(v), min(Val(u), wt(u,v)))$
SSNP	$Val(v) > max(Val(u), wt(u,v))$ $CASMIN(Val(v), max(Val(u), wt(u,v)))$
SSSP	$Val(v) > Val(u) + wt(u,v)$ $CASMIN(Val(v), Val(u) + wt(u,v))$
Viterbi	$Val(v) < Val(u)/wt(u,v)$ $CASMAX(Val(v), Val(u)/wt(u,v))$
REACH	$Val(v) < Val(u)$ $CASMAX(Val(v), Val(u))$
WCC	$Val(v) > Val(u)$ $CASMIN(Val(v), Val(u))$

An arbitrary query, using both the core and full graphs, is evaluated using Algorithm 5 on a GPU as follows. In the *initialization* step (lines 3 to 8) of this algorithm, the host receives the query vertex and initializes the vertex values such that the values of the outgoing neighbors are computed using the source vertex value and these outgoing neighbors form the initial frontier. Next the host transfers the initial value array, the frontier, and the core graph to the GPU to begin query evaluation.

On the GPU, the query evaluation is carried out in two phases: Core Phase (lines 9 to 13) and Completion Phase (lines 14 to 18). During the first phase, the query evaluation is carried on the much smaller CG and when this phase stabilizes, the second phase begins. The second phase, starting from the source vertex, and this time using the FG, again evaluates the query till the values stabilize and the final result becomes available.

In the completion phase, all reachable vertices must be visited at least once to ensure that their values are push along outgoing edges in FG that were excluded from CG. This is achieved by ensuring that upon first visit to a vertex in this phase, it is always added to the frontier even if its property value is unchanged (FirstPhase2Visit(), line 23).

The first phase is fast and effective as it is an in-memory phase which produces

Algorithm 5 2Phase evaluation of a query for source vertex s on input graph $FG(V, E)$.

1: **Input**: s, $FG(V, E)$, and $CG(V_c = V, E_c)$

2: **Output**: Query Result – $Val_s(*)$

3: ▷ ***Initialization***: Initialize Val Array on Host

4: $Val_s(s) \leftarrow SourceInitVal$

5: $\forall v \in OutNeighbors(s)$, $Val_s(v) \leftarrow Val_s(s)$ $w(s, v)$

6: $\forall v \in V - OutNeighbors(s)$, $Val_s(v) \leftarrow InitVal$

7: Transfer from Host to GPU:

8: $Val_s(*)$, $OutNeighbors(s)$, & $CG(V_c, E_c)$

9: ▷ ***Core Phase***: Process Core Graph on GPU

10: Active $\leftarrow OutNeighbors(s)$

11: **while** Active $\neq \phi$ **do**

12: Active $\leftarrow$ Process (Active, $CG(V_c, E_c)$)

13: **end while**

14: ▷ ***Completion Phase***: Process Full Graph on GPU+Host

15: Active $\leftarrow$ { s }

16: **while** Active $\neq \phi$ **do**

17: Active $\leftarrow$ Process (Active, $FG(V, E)$)

18: **end while**

precise results for over 94% of the vertices. The second phase is efficient because most needed computations have already been completed in the first phase and the edge function that uses expensive atomic operation to propagate values is applied for incoming edges of only a small number of vertices.

Note that the 2Phase algorithm is general and can be used to enhance the performance of range of existing systems including out-of-core systems like GridGraph [105] and even in-memory systems like Ligra [73]. In context of in-memory system like Ligra, the Core Phase reduces overall computation performed while in the context of an out-of-core

Algorithm 6 Function Process for 2Phase evaluation of a query.

```
1: ▷ Push Val_s's of Vertices in ACTIVE Over outEdges
2: function PROCESS ( ACTIVE , Graph )
3:     newACT ← φ
4:     for all u ∈ ACTIVE do
5:         for all e(u, v) ∈ Graph.outEdges(u) do
6:             if Needed(u, v) then
7:                 change ← EDGEFUNCTION ( e(u, v) )
8:                 if change ∨ FIRSTPHASE2VISIT(v) then
9:                     newACT ← newACT ∪ { v }
10:                end if
11:            end if
12:        end for
13:    end for
14:    return newACT
15: end function
```

system like GridGraph an in-memory Core Phase reduces the cost of both computation and I/O performed during the second out-of-core phase. For simplicity, we maintain separate core graph and full graph representations and simply switch from core graph to full graph when we transition to the second phase.

We evaluate the benefits of core graphs in improving the performance of query evaluation for the following systems:

- **GridGraph** [105] disk-based out-of-core graph processing system with 8GB available memory exceeded by all graph sizes and 4×4 grid partitioning; and
- **Ligra** [73] in-memory graph processing system with push-based algorithms.

The evaluation employs Specialized CGs for SSSP, SSNP, Viterbi, and SSWP while General CG is used to evaluate reachability (REACH) and Weakly Connected Components (WCC). The CGs were derived from evaluation of queries for 20 highest degree vertices in each graph. The choice of 20 was made after it was observed that evaluation of additional queries contributed very few new edges to the CG. This behavior is shown in Figure 4.4. Four input graphs from Table 4.1 are used. The default weight generation tool from Ligra is used to generate weights ranging from 1 to the $log(n) + 1$ (where, $n = |V|$).

The baselines are the original GridGraph, and Ligra systems and same settings are used for CG-based 2Phase runs. We perform in-memory evaluation of the query on the core graph in the first phase. For GridGraph first phase is performed over unpartitioned graph. Data presented represents averages based upon execution of ten random queries for each graph and algorithm combination.

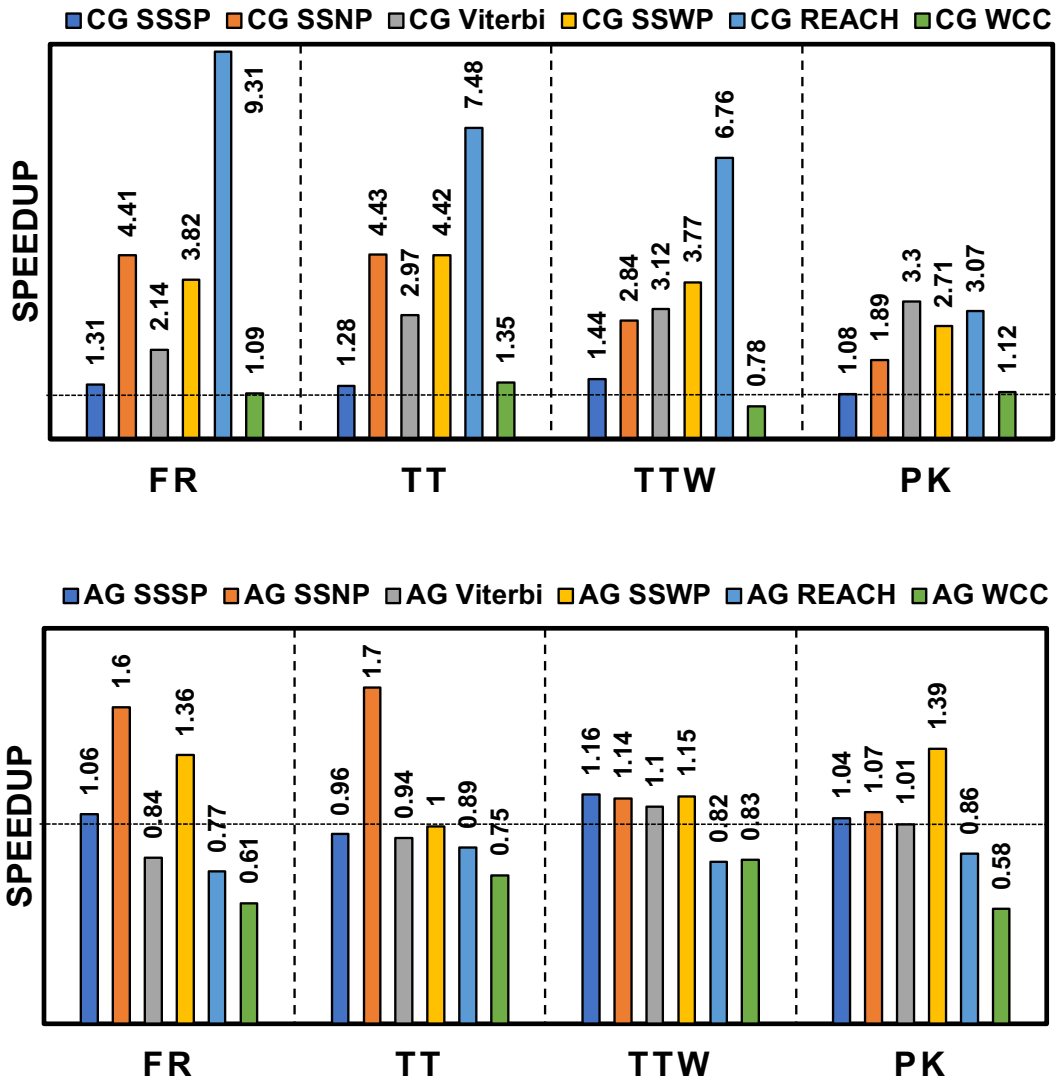


Figure 5.1: Speedups Over **Ligra** Due To Bootstrapping Initial Result from CG and AG.

5.2 In-Memory Graph Processing: Ligra Evaluation

Ligra [73] runs on a server where the graph fits in memory and thus gains from CGs can be expected from reduced computation, and enhanced locality in the caches due to the smaller CG. For weighted graphs, as shown in Figure 5.1, the 2Phase approach

Table 5.2: Average Execution Times in Seconds for Core Graph based 2Phase **Ligra** [73] across 10 Queries.

G	Specialized CGs				General CG	
	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	926.6s	358.1s	677.5s	405.1s	59.6s	377.7s
TT	137.7s	151.8s	186.4s	84.1s	28.3s	102.5s
TTW	235.4s	111.1s	130.5s	98.6s	25.2s	425.7s
PK	3.6s	1.8s	2.2s	2.4s	0.5s	1.2s

Table 5.3: Benefit of CG to **Ligra** [73]: Average % Reduction in Edges Processed (EDGES-RED).

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	10.2%	26.1%	56.0%	50.4%	94.8%	40.9%
TT	46.2%	29.6%	36.4%	19.0%	93.1%	42.5%
TTW	52.5%	35.2%	51.9%	39.7%	92.1%	41.0%
PK	52.7%	39.1%	75.0%	44.3%	88.2%	36.8%

delivers speedups of 4.42–2.71× for SSWP queries (with highest first phase precision) and for SSSP queries (with lowest first phase precision) speedups of 1.44–1.08× were observed. For REACH speedups are even higher. Average speedups across all queries are higher for larger graphs (e.g., FR) and least for the smallest graph (PK). Note that to save memory needed for CG and FG, while preserving efficiency, the edge lists can be organized to separate critical and non-critical edges so that latter can be easily skipped during the first phase. We observe that in comparison to CGs, AGs deliver significantly lower speedups over Ligra: highest speedup of 1.70× for AGs vs. 9.31× for CGs. As we can see, AGs frequently result in slowdowns over Ligra. Finally, Table 5.4 shows that significant reduction in computation (dynamic edges processed) causes the Ligra performance to improve due to CGs.

Table 5.4: Benefit of CG: Average % Reduction in Computation in Ligra [73].

G		SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	COMP-RED	10.2%	26.1%	56.0%	50.4%	94.84%	40.90%
	CG SPEEDUP	1.31×	4.41×	2.14×	3.82×	9.31×	1.09×
	AG SPEEDUP	1.06×	1.60×	0.84×	1.36×	0.77×	0.61×
TT	COMP-RED	46.2%	29.6%	36.4%	19.0%	93.09 %	42.51%
	CG SPEEDUP	1.28×	4.43×	2.97×	4.42×	7.48×	1.35×
	AG SPEEDUP	0.96×	1.70×	0.94×	1.00×	0.89×	0.75×
TTW	COMP-RED	52.5%	35.2%	51.9%	39.7%	92.10%	40.95%
	CG SPEEDUP	1.44×	2.84×	3.12×	3.77×	6.76×	0.78×
	AG SPEEDUP	1.16×	1.14×	1.10×	1.15×	0.82×	0.83×
PK	COMP-RED	52.7%	39.1%	75.0%	44.3%	88.24%	36.79%
	CG SPEEDUP	1.08×	1.89×	3.30×	2.71×	3.07×	1.12×
	AG SPEEDUP	1.04×	1.07×	1.01×	1.39×	0.86×	0.58×

5.3 Out-of-Core Graph Processing: GridGraph Evaluation

For GridGraph, the first phase of computation is performed in-memory after loading the CG from disk and then the second phase performs partition-based processing. The active frontier for second phase is set to all the vertices whose values have been changed by the first phase. This ensures that maximal amount of updates are performed in the first iteration. This policy generally leads to fewer iterations. During the second phase disk IO savings come due to two reasons: fewer iterations may be performed compared to baseline GridGraph; and during an iteration blocks with no active edges may arise more frequently and hence their fetch from disk will be skipped due to the selective scheduling optimization in GridGraph.

For GridGraph, we specified 4×4 grid for partitioning the graphs and 8 GB of available memory which is less than all graph sizes (except PK). Since same memory con-

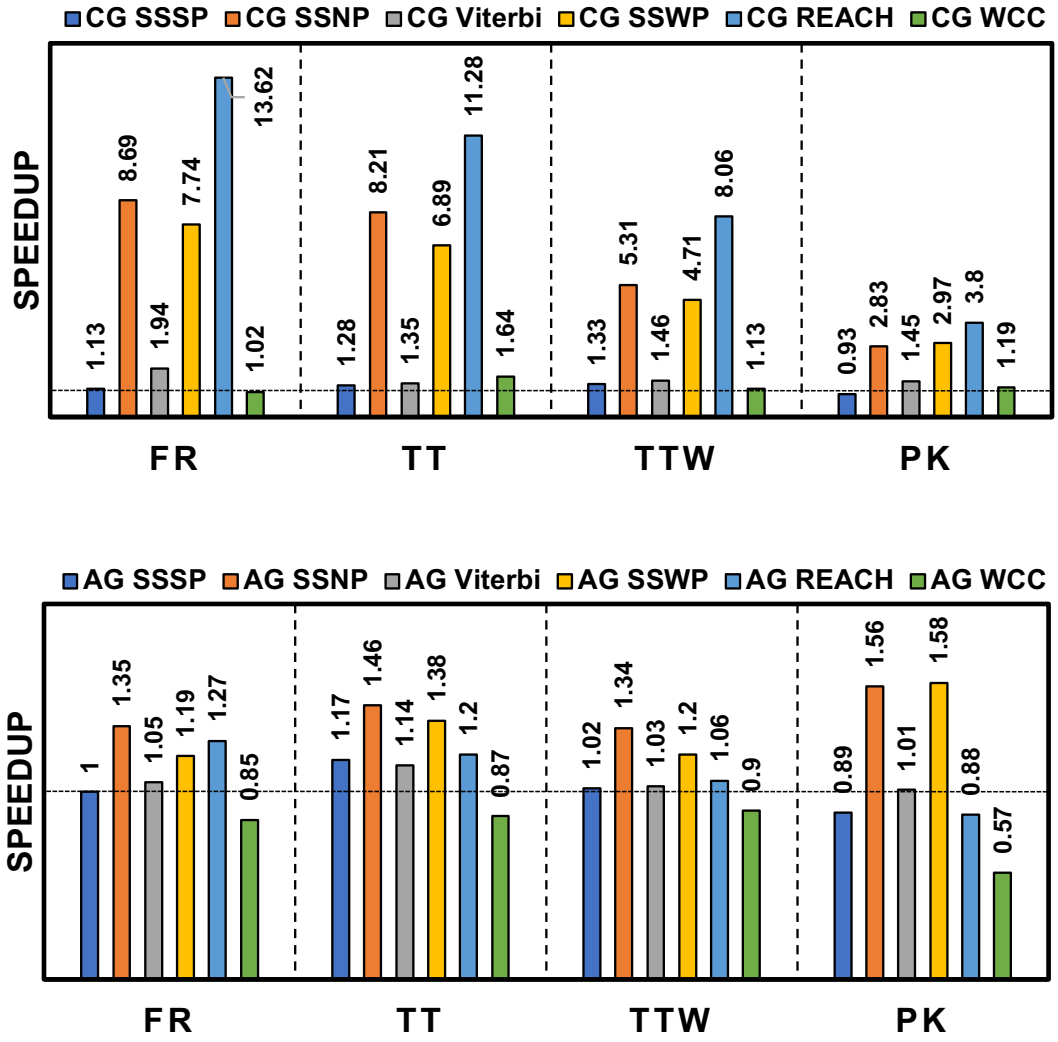


Figure 5.2: Speedups Over **GridGraph** Due To Bootstrapping Initial Result from CG and AG.

figuration was used for all graphs, larger graphs experienced more IO and hence greater benefits from use of CGs. Figure 5.2 shows speedups observed for larger graphs are greater than for smaller ones. Note, speedups for FR>TT>TTW>PK due to higher disk IO savings. Fewer iterations in second phase are shown in percentage reduction terms in Table 5.7. The speedups for queries with high precision (SSNP, Viterbi, SSWP, REACH) speedups range

Table 5.5: Average Execution Times in Seconds for Core Graph based 2Phase **GridGraph** [105] across 10 Queries.

G	Specialized CGs				General CG	
	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	274.4s	39.0s	200.5s	39.1s	11.0s	98.8s
TT	116.3s	24.8s	163.5s	23.2s	7.1s	24.2s
TTW	78.5s	27.4s	108.2s	29.8s	6.5s	34.4s
PK	3.2s	1.5s	3.0s	1.5s	0.3s	0.8s

Table 5.6: Benefit of CG to **GridGraph** [105]: Average % reduction in the # of iterations requiring disk IO.

G	SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	23.5%	96.4%	44.4%	97.1%	95.6%	0%
TT	29.3%	94.8%	33.3%	94.1%	93.1%	42.0%
TTW	36.7%	94.7%	36.1%	94.5%	93.8%	0%
PK	27.5%	96.5%	47.0%	96.8%	92.4%	28.6%

from 13.62× to 1.35×. The speedups for SSSP and WCC are modest because the number of iterations in second phase is closer to number of iterations performed by the baseline GridGraph. Note that for WCC on FR there is no change in the number of iterations yet there is no slow down because fetches of more blocks of edges can be skipped in the second phase. For SSSP on PK there is a slight slow down because cost of first phase offsets the benefits to second. Finally, we observe that speedups achieved when AGs are used are relatively small ranging from 1.58× to a slowdown of 0.57×. We also note that bootstrapping initial result from CG is superior to AG. Note that higher speedups for AGs in [99] are due to additional optimizations besides "bootstrapping an initial result" from AG.

Table 5.7: Average Speedups over GridGraph [105]; and Average % reduction in the # of iterations requiring disk IO.

G		SSSP	SSNP	Viterbi	SSWP	REACH	WCC
FR	ITER-RED	23.5%	96.4%	44.4%	97.1%	95.6%	0%
	CG SPEEDUP	1.13×	8.69×	1.94×	7.74×	13.62×	1.02×
	AG SPEEDUP	1.00×	1.35×	1.05×	1.19×	1.27×	0.85×
TT	ITER-RED	29.3%	94.8%	33.3%	94.1%	93.1%	42.0%
	CG SPEEDUP	1.28×	8.21×	1.35×	6.89×	11.28×	1.64×
	AG SPEEDUP	1.17×	1.46×	1.14×	1.38×	1.20×	0.87×
TTW	ITER-RED	36.7%	94.7%	36.1%	94.5%	93.8%	0%
	CG SPEEDUP	1.33×	5.31×	1.46×	4.71×	8.06×	1.13×
	AG SPEEDUP	1.02×	1.34×	1.03×	1.20×	1.06×	0.90×
PK	ITER-RED	27.5%	96.5%	47.0%	96.8%	92.4%	28.6%
	CG SPEEDUP	0.93×	2.83×	1.45×	2.97×	3.80×	1.19×
	AG SPEEDUP	0.89×	1.56×	1.01×	1.58×	0.88×	0.57×

5.4 Combining Benefits of Value and Structure Reuse

The exact evaluation of a query results requires a two phase approach where the first phase evaluates the query on the small in-memory core graph (CG) and then uses the results obtained to bootstrap the evaluation of the query on the full graph (G). Starting from all vertices whose values are impacted in the first phase, the second phase resumes propagation of values over the full graph to obtain precise results for all vertices. Since most results are computed precisely in the first phase efficiently using the small CG, the work performed during the second phase is greatly reduced giving significant speedups.

Next we present a new optimization over the above approach to improve the efficiency of the second phase. The key idea behind this optimization is as follows. After the first phase has completed, we introduce a step that is able to identify some (but not all) of the vertices whose values are already precise and hence will not change in the second

phase. For each such vertex v, the incoming edges of v are removed from the full graph G because any propagation via these edges will not change the value of vertex v and hence would be wasteful. Next we present a theorem that allows us to identify some (but not all) vertices with stable values following first phase. We first present the above results in context of the shortest path problem and later show that these results apply to many other graph algorithms.

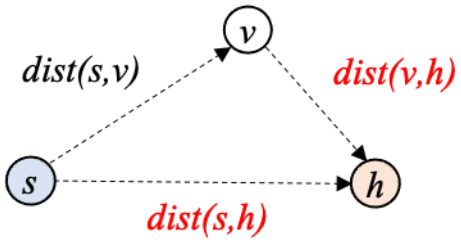
Given a full graph $G(V, E)$ and a high degree vertex h in V such that forward and backward shortest path queries $SSSP_f(h, G)$ and $SSSP_b(h, G)$ are evaluated to identify the corresponding core graph $CG(V, E_c)$. Now consider the first phase evaluation of user query $SSSP_f(s, CG)$ on $CG(V, E_c)$ that computes, for each vertex v reachable from s, the shortest path length $dist(s, v).CG$. The following theorem provides the condition under which $dist(s, v).CG$ is precise.

Theorem 1: The computed value $dist(s, v).CG$ is precise if one of the following conditions is true:

(a) $dist(s, v).CG == dist(s, h).G - dist(v, h).G$

(b) $dist(s, v).CG == dist(h, v).G - dist(h, s).G$

Proof: To prove the above, we rely on the triangle inequality over the shortest path property as given in [33].



According to the triangle quality for shortest path property: $dist(s,v).G + dist(v,h).G \geq dist(s,h).G$ *or* $dist(s,v).G \geq dist(s,h).G - dist(v,h).G$ (1)

Since CG is a subgraph of G:

$$dist(s,v).CG \geq dist(s,v).G \quad (2)$$

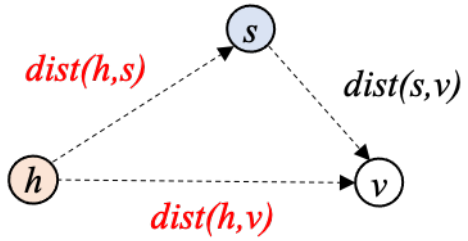
Therefore from (1) and (2) we conclude that:

$$dist(s,v).CG \geq dist(s,h).G - dist(v,h).G$$

Thus, **if** we observe that

$$dist(s,v).CG == dist(s,h).G - dist(v,h).G \quad \text{(a)}$$

then $dist(s,v).CG$ must be precise.



Similarly, we use another triangle inequality as follows:

$$dist(h,s).G + dist(s,v).G \geq dist(h,v).G$$

or $dist(s,v).G \geq dist(h,v).G - dist(h,s).G$ (1)

Since CG is a subgraph of G:

$$dist(s,v).CG \;\geq\; dist(s,v).G \qquad (2)$$

From (1) and (2) we conclude that if:

$$dist(s,v).CG \;\geq\; dist(h,v).G \;-\; dist(h,s).G$$

Therefore, **if** we observe that

$$dist(s,v).CG \;==\; dist(h,v).G \;-\; dist(h,s).G \qquad \text{(b)}$$

then $dist(s,v).CG$ must be precise.

As shown in [33], the graph triangle inequality abstraction given below applies to many different graph properties.

$$property(v_1,v_2) \oplus property(v_2,v_3) \succeq property(v_1,v_3)$$

Here $\oplus$ depicts an abstract addition and $\succeq$ represents an abstract greater than or equal operator. While operators vary across the different algorithms, the proposed optimization applies many graph algorithms such as widest path, narrowest path, breadth-first search, and others.

Algorithm 5 shows the 2Phase evaluation of a query on a GPU. In the initialization step (lines 3 to 8) of this algorithm, the host receives the query vertex and initializes the vertex values such that the values of the outgoing neighbors are computed using the source vertex value and these outgoing neighbors form the initial frontier. Next the host transfers the initial value array, the frontier, and the core graph to the GPU to begin query evaluation.

The data presented thus far did not make use of the triangle inequality optimization thus requiring no major changes to existing systems. We added the optimization to Ligra and reevaluated performance for the two largest input graphs, FR and TT. We consider the three algorithms identified in [33] for which the triangle inequality is the most effec-

Table 5.8: Impact of Triangle Inequality on **Ligra** Speedups.

G		SSNP	Viterbi	SSWP
FR	SPEEDUP	4.24×	4.40×	7.30×
	EDGES-RED	70.95%	78.71%	93.23%
TT	SPEEDUP	6.06×	4.52×	6.01×
	EDGES-RED	89.95%	80.48%	88.80%
TTW	SPEEDUP	2.86×	2.78×	3.20×
	EDGES-RED	84.29%	80.75%	83.40%
PK	SPEEDUP	1.79×	1.83×	1.87×
	EDGES-RED	85.67%	86.71%	83.72%

tive. The new speedups shown in Table 5.8 for SSWP, Viterbi, and SSNP are substantial improvements over prior speedups for the two largest graphs.

5.5 Summary

We identified core graphs with 10.7% of edges on average that rapidly yield precise results for 94.5–99.9% of vertices. An optimized second pass efficiently computes the precise results for rest of the vertices. The generality of the CG based approach allows it to be applied across existing systems without requiring any major modifications to them. We demonstrated this by enhancing three different systems. Significant performance improvements for these systems were observed – up to 13.62× in GridGraph [105], and up to 4.97× in Ligra [73]. Applying the triangle inequality optimization gave additional speedups.

Chapter 6

Contributions and Future Work

6.1 Contributions

In this thesis, we identify opportunities for improving query performance across various systems by capturing the essence of a graph both as values and structure. Results from solving a few queries are leveraged to benefit all subsequent queries. First, we investigate the feasibility of bidirectionally solving for values (results) of a limited set of high-centrality queries. We then extend the concept of value reuse to streaming and batching systems. Second, we developed an approach for the reuse of graph structures. We identify the core graph structure by marking critical edges. Ultimately, we combine graph structure reuse with value reuse to obtain best results across different systems.

This thesis has developed VRGQ that incorporates the 2Step iterative algorithm for evaluating a sequence of vertex queries. The key feature of this framework is that during evaluation of any query, coarse-grained reuse of previously computed results for selected vertices is performed to accelerate query evaluation. The VRGQ system is different from

related works in a key way. It takes advantages of results computed for a very small number of queries to optimize the execution of all future queries via value reuse.

Value reuse is applicable across various graph processing systems. In streaming graph systems, Tripoline leverages the value reuse technique to solve a fundamental limitation of stream graph systems. This idea leads to a generalized incremental processing design for vertex-specific queries, in which optimization is ensured via reuse based vertex functions. Multilyra reuses results from earlier batches of queries to accelerate the execution of later batches of queries. SimGQ is a shared memory system that dynamically identifies a shared query that substantially represent subcomputations in the evaluation of different queries in a batch, then evaluates and reuses its results to accelerate the evaluation of all queries in the batch.

Additionally, this thesis incorporated the concept of graph structure reuse. Firstly it identified the phenomenon of *recurring critical edges* and discovered a *core graph* with modest quantity of edges that yield precise results for virtually all vertices. A second pass over the full graph computes precise results for rest of the vertices leading to net reduction in execution times over baselines that only work using full graphs. Significant performance improvements for multiple existing systems are shown for six kinds of graph queries and for four different input graphs.

6.2 Future Work

6.2.1 Using CoreGraph for Other Benchmarks

CoreGraph was exploited in the evaluation of path-based graph queries to achieve significant speedups. There is further potential for leveraging critical edges to speed up convergence of both global and Personalized PageRank algorithms. Identifying certain paths and precomputing their results could lead to faster propagation and convergence of PageRank values. Rather than simply marking edges as critical or not critical, we could assign impact numbers to edges and use them to rank and select important paths. HubPPR [109] pre-computes and indexes auxiliary information for selected hub nodes that are often involved in PPR processing. Nevertheless, the challenging problem of identifying effective indexing schemes remains open for graph queries.

6.2.2 Reuse for Graph Partitioned Distributed Systems

For shared memory and out-of-core systems, we used CoreGraph to mainly reduce computation cost. In distributed graph processing systems, CoreGraph holds the potential to refine graph partitioning and reduce communication overhead. A straightforward approach is to store the compact CoreGraph on every machine, facilitating query evaluations locally. It is also interesting to store the CoreGraph along with its extensive neighbors on a master machine. Intermediate query values can then be scattered to other mirrored machines after evaluation on CoreGraph.

6.2.3 Streaming Graph and Incremental Evaluation

We highlighted how Tripoline generalizes, via value reuse, the use of incremental evaluation to random graph queries. It is also promising to identify critical edges across different snapshots to develop CoreGraph+, and further leverage graph structure reuse for queries over evolving graph systems. After computing the query on the CoreGraph+, we can add or delete edges and incrementally update the query results. Recent work on CommonGraph [110] converts all deletions to additions by finding a common graph that exists across all snapshots to evaluate an evolving graph query. CoreGraph+ may allow further generalization to achieve reuse across different queries.

Bibliography

[1] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient speculative parallelism for accelerators. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, pages 1247–1262, 2020.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. SIGARCH Comput. Archit. News, 43(3):105–117, June 2015.

[3] A. Ayupov, S. Yesil, M. M. Ozdal, T. Kim, S. Burns, and O. Ozturk. A template-based design methodology for graph-parallel hardware accelerators. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(2):420–430, Feb 2018.

[4] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), pages 44-54, 2006.

[5] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '17, pages 235–248, 2017.

[6] Markus Bläser. A new approximation algorithm for the asymmetric tsp with triangle inequality. ACM Transactions on Algorithms (TALG), 4(4):1–15, 2008.

[7] M. Cha, H. Haddadi, Fabrício Benevenuto, and K. Gummadi. Measuring user influence in twitter: The million follower fallacy. In ICWSM, 2010.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. ACM Transactions on Parallel Computing (TOPC), 5(3):1–39, 2019.

[9] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In Proceedings of the 7th ACM european conference on Computer Systems, pages 85–98, 2012.

[10] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. Proceedings of the VLDB Endowment, 8(12):1804–1815, 2015.

[11] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. ACM Trans. Math. Softw., 38(1):1:1–1:25, December 2011.

[12] Fethi Demim, Kahina Louadj, and Abdelkrim Nemra. Path planning for unmanned ground vehicle. In 2018 5th International Conference on Control, Decision and Information Technologies (CoDIT), pages 748–750. IEEE, 2018.

[13] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 918–934, 2019.

[14] David Ediger, Karl Jiang, Jason Riedy, and David A Bader. Massive streaming data analytics: A case study with clustering coefficients. In 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pages 1–8. IEEE, 2010.

[15] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In 2012 IEEE Conference on High Performance Extreme Computing, pages 1–5. IEEE, 2012.

[16] David Ediger, Jason Riedy, David A Bader, and Henning Meyerhenke. Tracking structure of streaming social networks. In 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pages 1691–1699. IEEE, 2011.

[17] Charles Elkan. Using the triangle inequality to accelerate k-means. In Proceedings of the 20th international conference on Machine Learning (ICML-03), pages 147–153, 2003.

[18] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. Grape: Parallelizing sequential graph computations. Proceedings of the VLDB Endowment, 10(12):1889–1892, 2017.

[19] Friendster data set. In http://konect.cc/networks/friendster/.

[20] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 17–30, 2012.

[21] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 599–613, 2014.

[22] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–13, Oct 2016.

[23] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a GPU. In Proceesdings of the 26th International Conference on Parallel Architectures and Compilation Techniques, PACT '17, pages 233–245, 2017.

[24] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In Proceedings of the Ninth European Conference on Computer Systems, pages 1–14, 2014.

[25] Thomas Little Heath et al. The thirteen books of Euclid's Elements. Courier Corporation, 1956.

[26] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In Proceedings of the forty-sixth annual ACM symposium on Theory of computing, pages 674–683, 2014.

[27] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P. Sadayappan. Multigraph: Efficient graph processing on gpus. In Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques, PACT '17, pages 27–40, 2017.

[28] Farzin Houshmand, Mohsen Lesani, and Keval Vora. Grafs: Graph analytics fusion and synthesis. In Proceedings of the International Conference on Functional Programming (ICFP), 2021.

[29] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, pages 1–6, 2016.

[30] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 228–241, Dec 2015.

[31] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. Data-centric execution of speculative parallel programs. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-49. IEEE Press, 2016.

[32] Xiaolin Jiang, Chengshuo Xu, and Rajiv Gupta. VRGQ: evaluating a stream of iterative graph queries via value reuse. ACM SIGOPS Oper. Syst. Rev., 55(1):11–20, 2021.

[33] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Tripoline: generalized incremental graph processing via graph triangle inequality. In EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021, pages 17–32. ACM, 2021.

[34] R. Jin, N. Ruan, B. You, and H. Wang. Hub-accelerator: Fast and exact shortest path computation in large social networks. CoRR, abs/1305.0507, 2013.

[35] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. Version traveler: Fast and memory-efficient version switching in graph processing systems. In Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '16, page 523–536, USA, 2016. USENIX Association.

[36] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In Proceedings of the International Conference on Parallel Architectures and Compilation, PACT '15, pages 39–50, 2015.

[37] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '14, pages 239–252. ACM, 2014.

[38] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD '16, page 447–461, 2016.

[39] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. Efficient processing of large graphs via input reduction. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016, pages 245–257. ACM, 2016.

[40] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In Proceedings of the 19th International Conference on World Wide Web, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[41] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012, pages 31–46. USENIX Association, 2012.

[42] Leslie Lamport. LaTeX: A Document Preparation System. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1994.

[43] Jüri Lember, Dario Gasbarra, Alexey Koloydenko, and Kristi Kuljus. Estimation of viterbi path in bayesian hidden markov models. METRON, 77(2):137–169, 2019.

[44] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics, 6(1):29–123, 2009.

[45] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, pages 49–63, 2019.

[46] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. arXiv preprint arXiv:1408.2041, 2014.

[47] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. In Proceedings of the VLDB Endowment, 5(8):716-727, 2012.

[48] A. Mazloumi, X. Jiang, and R. Gupta. MultiLyra: Scalable distributed evaluation of batches of iterative graph queries. In IEEE International Conference on Big Data (BigData), pages 349-358, 2019.

[49] A. Mazloumi, C. Xu, Z. Zhao, and R. Gupta. BEAD: Batched evaluation of iterative graph-queries with evolving analytics demands. In IEEE International Conference on Big Data (BigData), pages 461-468, 2020.

[50] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 135–146, 2010.

[51] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1–16, 2019.

[52] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. ACM Sigplan Notices, 47(8):117–128, 2012.

[53] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), page 1–14. IEEE, Oct 2018.

[54] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. PHI: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-52, page 1009–1022, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 439–455, 2013.

[56] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 457–468, Feb 2017.

[57] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP), page 456–471, 2013.

[58] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy efficient architecture for graph analytics accelerators. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pages 166–177, June 2016.

[59] P. Pan and C. Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In IEEE International Conference on Computer Design (ICCD), pages 217-224, 2017.

[60] S. Rahman, N. Abu-Ghazaleh, and R. Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 908–921, 2020.

[61] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. Proceedings of the VLDB Endowment, 4(11):726–737, 2011.

[62] Jason Riedy and Henning Meyerhenke. Scalable algorithms for analysis of massive, streaming graphs. SIAM Parallel Processing for Scientific Computing, 2012.

[63] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. SIAM Journal on Computing, 45(3):712–733, 2016.

[64] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In Proceedings of the 25th Symposium on Operating Systems Principles, pages 410–424, 2015.

[65] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 472–488, 2013.

[66] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: minimizing data transfer during out-of-gpu-memory graph processing. In Proceedings of the Fifteenth EuroSys Conference, EuroSys '20, pages 12:1–12:16, 2020.

[67] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. ACM SIGPLAN Notices, 53(2):622–636, 2018.

[68] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In Proceedings of the 25th International Conference on Scientific and Statistical Database Management, pages 1–12, 2013.

[69] Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. A sketch-based distance oracle for web-scale graphs. In Proceedings of the third ACM international conference on Web search and data mining, pages 401–410, 2010.

[70] Dipanjan Sengupta, Kapil Agarwal, Shuaiwen Leon Song, and Karsten Schwan. Graphreduce: Large-scale graph analytics on accelerator-based HPC systems. In IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPSW '15, pages 604–609, 2015.

[71] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In Proceedings of the 2016 International Conference on Management of Data, pages 417–430, 2016.

[72] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. Journal of the ACM (JACM), 28(1):1–4, 1981.

[73] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 135–146, 2013.

[74] M. D. Sinclair, J. Alsop, and S. V. Adve. Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), page 161–174, Jun 2017. Citation Key: RAts.

[75] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. Start late or finish early: A distributed graph processing system with redundancy reduction. Proc. VLDB Endow., 12(2):154–168, 2018.

[76] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1222–1230, 2012.

[77] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. Towards large-scale graph stream processing platform. In Proceedings of the 23rd International Conference on World Wide Web, pages 1321–1326, 2014.

[78] L. Takac and M. Zabovsky. Data analysis in public social networks. In International Scientific Conference and International Workshop Present Day Trends of Innovations, pages 1-6, 2012.

[79] M. Then, M. Kaufmann, F. Chirigati, T-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H.T. Vo. The more the merrier: Efficient multi-source graph traversal. In Proceedings of the VLDB Endowment, 8(4):449-460, 2015.

[80] Amanda L Traud, Peter J Mucha, and Mason A Porter. Social structure of Facebook networks. Phys. A, 391(16):4165–4180, Aug 2012.

[81] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In Proceedings of the 7th ACM international conference on Web search and data mining, pages 333–342, 2014.

[82] Leslie G Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103–111, 1990.

[83] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic analysis of evolving graphs. ACM Transactions on Architecture and Code Optimization (TACO), 13(4):1–27, 2016.

[84] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems, pages 237–251, 2017.

[85] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In Andrew P. Black and Todd D. Millstein, editors, Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, pages 861–878. ACM, 2014.

[86] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. Coral: Confined recovery in distributed asynchronous graph processing. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, pages 223–236, 2017.

[87] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In Ajay Gulati and Hakim Weatherspoon, editors, USENIX Annual Technical Conference (USENIX ATC) 2016, Denver, CO, USA, June 22-24, 2016, pages 507–522. USENIX Association, 2016.

[88] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single {PC}. In 2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15), pages 387–401, 2015.

[89] Yangzihao Wang, Yuechao Pan, Andrew A. Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. ACM Transactions on Parallel Computing, 4(1):3:1–3:49, 2017.

[90] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. Query-by-sketch: Scaling shortest path graph queries on very large networks. In SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pages 1946–1958. ACM, 2021.

[91] Zheng Wang and Jon Crowcroft. Quality-of-service routing for supporting multimedia applications. IEEE Journal on selected areas in communications, 14(7):1228–1234, 1996.

[92] Chengshuo Xu, Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. SimGQ: Simultaneously evaluating iterative graph queries. In 27th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), pages 1–10, December 2020.

[93] Chengshuo Xu, Keval Vora, and Rajiv Gupta. Pnp: Pruning and prediction for point-to-point iterative graph analytics. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 587–600. 2019.

[94] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai Seraph: an efficient, low-cost system for concurrent graph processing. In ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC), pages 227-238, 2014.

[95] D. Yan, J. Cheng, M.T. Ozsu, F. Yang, Y. Lu, J.C.S. Lui, Q. Zheng and W. Ng. A general-purpose query-centric framework for querying big graphs. In Proceedings of the VLDB Endowment, 9(7):564-575, 2016.

[96] Kaige Yang and Laura Toni. Graph-based recommendation system. In 2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP), pages 798–802. IEEE, 2018.

[97] Mindi Yuan, Kun-Lung Wu, Gabriela Jacques-Silva, and Yi Lu. Efficient processing of streaming graphs for evolution-aware clustering. In Proceedings of the 22nd ACM international conference on Information & Knowledge Management, pages 319–328, 2013.

[98] G. Zhang, W. Horn, and D. Sanchez. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), page 13–25, Dec. 2015.

[99] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 608–621. 2018.

[100] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu. Cgraph: a correlations-aware approach for efficient concurrent iterative graph processing. In USENIX Annual Technical Conference (USENIX ATC), pages 441-452, 2018.

[101] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. CoRR, abs/1710.05785, 2017.

[102] Yu Zhang, Xiaofei Liao, Lin Gu, Hai Jin, Kan Hu, Haikun Liu, and Bingsheng He. Asyngraph: Maximizing data parallelism for efficient iterative graph processing on gpus. ACM Trans. Archit. Code Optim., 17(4):29:1–29:21, 2020.

[103] S. Zhou, C. Chelmis, and V. K. Prasanna. High-throughput and energy-efficient graph processing on FPGA. In Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM '16, pages 103–110, May 2016.

[104] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. An FPGA framework for edge-centric graph processing. In Proceedings of the 15th ACM International Conference on Computing Frontiers, CF '18, pages 69–77, 2018.

[105] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In USENIX Annual Technical Conference (USENIX ATC), July 8-10, Santa Clara, CA, USA, pages 375–386. USENIX Association, 2015.

[106] Yang, Ke and Zhang, MingXing and Chen, Kang and Ma, Xiaosong and Bai, Yang and Jiang, Yong. KnightKing: A Fast Distributed Graph Random Walk Engine. In Proceedings of the 27th ACM Symposium on Operating Systems Principles., Association for Computing Machinery, New York, NY, USA, 524–537.

[107] S. Pandey, L. Li, A. Hoisie, X. S. Li and H. Liu. C-SAW: A Framework for Graph Sampling and Random Walk on GPUs. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 2020, pp. 1-15

[108] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '06), 2006. In Association for Computing Machinery, New York, NY, USA, 631–636.

[109] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. HubPPR: effective indexing for approximate personalized pagerank. In Proc. VLDB Endow. 10, 3 (November 2016), 205–216. https://doi.org/10.14778/3021924.3021936

[110] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. CommonGraph: Graph Analytics on Evolving Data. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)., Association for Computing Machinery, New York, NY, USA, 133–145. https://doi.org/10.1145/3575693.3575713