VU University Amsterdam

Master's Thesis

Extending the Lighthouse graph engine for shortest path queries

Author:
Peter Rutgers

Supervisor:
Prof. Dr. Peter Boncz

Second supervisor:
Dr. Spyros Voulgaris

Second reader:
Claudio Martella

August 2015

VU UNIVERSITY AMSTERDAM

Abstract

Faculty of Sciences
Department of Computer Science

Master of Sciences

Extending the Lighthouse graph engine for shortest path queries

by Peter Rutgers

Finding shortest paths based on edge weights has many applications in data analysis. It is desired to express such queries in a way that is easy to write and easy to detect by the query optimizer. However, many query languages for graphs do not currently provide this. We propose an extension to the Cypher query language and implement this as a new operator in Lighthouse, which is a scalable implementation of graph pattern matching based on the BSP computation model of Pregel. A number of algorithms for top-N shortest weighted paths are designed and evaluated in terms of performance and scalability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Graph databases are increasingly used to store and analyze datasets that contain a network-like structure. Modelling such a structure in a relational database can be inefficient, because analyzing paths or patterns in the network then requires a large number of join operations. For many applications, such as routing problems, social networks and user behavior analysis,[1] graph databases allow for very efficient querying. Another advantage is that the data model can be extended in a natural way: new types of vertices and relations can easily be added without modifying the existing data structure.

Considering that databases need to store an increasing amount of information, scalability is a very important aspect in designing such databases. A way to keep the system scalable is to distribute the graph over a number of processors. This has been implemented by the Lighthouse graph pattern matching engine [1]. Lighthouse is based on Apache Giraph [2], an open-source variant of the Pregel framework [3], which provides an API to specify computations within the Bulk Synchronous Parallel (BSP) model.

In the initial version of Lighthouse, a subset of the Cypher query language is accepted. This subset is able to express graph patterns consisting of vertices and edges between them. However, patterns with paths of variable length are not yet supported. Such path queries are expected to be very useful. They would allow searching by distance on a road map, evaluating reachability within a network, among other applications. In particular, we are interested in use cases like social networks, without depending on heuristics that use domain-specific metadata (such as coordinates on a road map). For

[1]Neo4j, Use Cases: http://neo4j.com/use-cases/

this reason, experiments are performed using the dataset provided by the LDBC social network benchmark [4].

1.2 Computation model

In the BSP computation model of Apache Giraph, vertices are evenly distributed over a number of workers. The computation is divided into *supersteps*, after each of which a global barrier is provided. The computation is vertex-centric: each vertex has a fixed ID and a value that can change during the computation, and communicates with other vertices through messages with a recipient vertex. Messages are not restricted to adjacent vertices, although it is usually the case that each vertex contacts its neighbours. All messages sent during superstep N are available at the recipient during superstep $N + 1$, in arbitrary order.

Like vertices, edges also have a value that can change during the computation. However, they do not have an associated computation method. Edges are always considered to be directed, although an undirected graph can often be modelled by having two directed edges in opposite directions. Each edge is located at the worker that contains their source vertex. Effectively, an outgoing edge can be considered local state information of a vertex. For this reason, incoming edges are not locally visible at the other vertex.

At the start of the computation, each vertex is active. A vertex can become inactive by voting to halt the computation. If a vertex receives a message, it becomes active until it again votes to halt. During each superstep, each active vertex calls its compute method. An algorithm can be implemented in Giraph by implementing this compute method. The final vertex values are often used as output, but it is also possible to collect output in different ways, such as writing to a file during the computation.

Aside from communication via messages, Giraph offers a few additional features. There is the concept of aggregators, where each vertex can aggregate a value and values are combined by a commutative and associative operation. The final value is available at all workers during the next superstep. Communication overhead is relatively low, since all values aggregated at a worker can be combined first, and only one value per worker needs to be sent over the network. This idea can also be applied to regular messages, by implementing a message combining operator that has commutative and associative properties.

Lighthouse

Lighthouse uses this model to perform pattern matching over a graph. Messages in Lighthouse are rows of a binding table, currently restricted to lists of vertex IDs. In this way, the binding table is partitioned over the vertices, based on the vertex in the last column of the table. A query plan is given as input, represented as a tree of operators such as Scan, StepJoin, and others. This tree is transformed by each worker into a set of linear execution paths, where each path contains a number of steps, and each first step in a path corresponds to a leaf node of the tree. A message represents a partial result at one of the steps.

The compute method of the Lighthouse core algorithm iterates over all received messages in some superstep, and applies for each message the operator corresponding to the next execution step. If it is a global operator, the message is sent to the corresponding vertex that is now at the last column of the table. After the last execution step of the last path, the message is written to an output file.

As the rows of the binding table often have to be transferred between execution steps, there is a relatively high volume of communication compared to the amount of computation. Therefore, the system is likely to be communication-bound for most realistic queries and graphs. This is confirmed by scalability experiments in [1].

1.3 Research questions

The main question to be solved in the project is how we can combine graph algorithms, specifically the shortest path algorithm, in the Lighthouse system. More concretely, this can be divided into the following research questions.

- How do different query languages currently handle queries for shortest paths? To what extent do they support general recursion mechanisms, that could be used for algorithms such as shortest paths? What are the benefits and disadvantages of different approaches?
- Could the Cypher-subset currently supported by Lighthouse be increased to include the required functionality? If not, is an extension to Cypher itself needed, or is it advisable to switch to a different query language entirely?
- How could a shortest path query best be represented at the level of query plan operators? What modifications would be needed in Lighthouse to support the

new operators required? What changes are needed to the semantics of the binding table, in order to support variable-length paths?

- What algorithms are available to solve the shortest path problem for large amounts of source-destination pairs in parallel? How do they scale on the BSP model? How can they be extended for our requirements, such as top N paths for each pair? [2]

- Are there different possible approaches in designing the algorithm? If so, what are the benefits and downsides to each approach? Can we find additional optimizations to reduce the risk of communication being a bottleneck?

These questions are addressed in the following chapters. In Chapter 2, we compare different approaches to shortest path algorithms that are related to our problem. Related work with regard to query languages is discussed in Chapter 3, followed by a proposal for an extension of the Cypher query language. In Chapter 4, required changes to the Lighthouse system are described that provide a new operator. In Chapter 5, a number of algorithms are extended and adapted for the BSP model. Different methods and heuristics for searching the graph are compared. Chapter 6 presents a number of optimizations that are generally independent of the method of searching, but intend to reduce communication and runtime in all cases. Some additional improvements are discussed in Chapter 7 that try to reduce the total amount of work done by sharing results between different searches. Finally, we conclude in Chapter 8 with an overview of findings and recommendations.

[2]When referring to top N shortest *paths*, we allow such paths to contain cycles, repeating vertices and edges, unless stated otherwise. Therefore, they are more precisely referred to as *walks*.

Chapter 2

Related work

The goal of our intended algorithm is to find the top-N shortest (variable-length) paths over some subset of the graph, in terms of some cost property. The subset could be specified by a condition that is evaluated on the vertices and edges. An example of a use case is to calculate three shortest routes over a network of roads, where the weight or cost of a path is its distance multiplied by its speed limit. The condition could be to only allow roads where cars are allowed.

In order to implement such a feature in Lighthouse, an algorithm should be chosen that is efficient in the BSP model of Pregel. This chapter presents an overview of such algorithms that are related to our computation model.

2.1 Implementation in Neo4J

The graph database Neo4J supports a number of path calculations: all paths, all simple paths, shortest path, the paths with fewest hops, or the paths with a fixed number of hops.[1] All of the algorithms need a PathExpander, to specify what types of edges are allowed. This is a method that takes a path as input and returns a set of edges that are incident to the last vertex of the path, allowing the user to filter on direction, edge type or edge properties. For the cheapest path calculation, two algorithms are available: A* and Dijkstra. In A*, an EstimateEvaluator must also be provided that estimates the remaining cost from a given vertex to the destination. The fewest hops calculation uses breadth-first search from the source and the destination, alternating between exploring a level forward from the source and a level backward from the destination.[2] These

[1]http://neo4j.com/docs/2.1.6/javadocs/org/neo4j/graphalgo/GraphAlgoFactory.html

[2]https://github.com/neo4j/neo4j/blob/master/community/graph-algo/src/main/java/org/neo4j/graphalgo/impl/path/ShortestPath.java

algorithms can be called from the Cypher query language or by the REST API that Neo4J provides. The A* implementation of shortest path can only be called from the Java API, after implementing an estimation heuristic.[3] In order to calculate shortest paths from many source vertices, Neo4J implemented the Floyd-Warshall algorithm for the all-pairs shortest paths problem. It is noted that this does not scale to larger graphs and is therefore no longer used.[4]

Returning the top-N shortest paths is possible, but only by using the all paths algorithm and filtering the paths that are found in this way.[5]

2.2 Algorithms designed for Pregel / Giraph

Shortest path problems are discussed as well in the paper presenting Pregel [3], which introduced the vertex-centric computational model used in Giraph. Three variants are mentioned: s-t shortest path (i.e. fixed destination), single-source, and all pairs. Because the last one requires O(V^2) of storage, this is not discussed. The first one, s-t, is considered a quite easy problem; experiments by Lumsdaine et al. [5] showed that only a small part of the vertices are visited before finding the shortest path. Only for the single-source variant an implementation was provided for Pregel, a parallel version of the Bellman-Ford algorithm [6]. They note that this parallel (BSP) implementation performs more comparisons than sequential versions (Dijkstra and Bellman-Ford), but is also much more scalable. More advanced parallel versions do exist: for this they refer to Thorup [7] and the Delta-stepping method by Meyers and Sanders [8].

The Delta-stepping method is an algorithm specifically designed for parallel computing, running in linear average time for most graphs. It uses buckets with a fixed width (Delta), containing the vertices to evaluate in each iteration. Edges with a small weight ($<$ Delta) are traversed in the same iteration, while edges with a large weight ($>$ Delta) are postponed to later iterations. By setting Delta very high, the algorithm behaves like Bellman-Ford; while by setting Delta very low, it behaves like Dijkstra. The best performance is obtained by setting it somewhere in between, although the exact value depends on the distribution of edge weights.

An experiment by Madduri et al. [9] applied this algorithm on billions of vertices and edges and found almost linear speedup on the Cray MTA-2 (of 31 on 40 processors).

[3]http://neo4j.com/docs/stable/tutorials-java-embedded-graph-algo.html
[4]https://github.com/neo4j/neo4j/blob/master/community/graph-algo/src/main/java/org/neo4j/graphalgo/impl/shortestpath/package-info.java
[5]http://stackoverflow.com/questions/12587263/cypher-order-by-path-cost/12588013

In a later experiment by Crobak et al. [10] Thorup's algorithm is compared to Delta-stepping. For individual runs, Delta-stepping turns out to be faster. For 40 simultaneous runs, Thorup is faster. This is because Thorup's algorithm first builds an large data structure (the Component Hierarchy) that can be used for multiple runs of the single-source algorithm. This structure then needs to be shared among all threads within the system. It is designed for integer weights, although an modification for floats is discussed by Thorup as well. Another disadvantage is that it requires an undirected graph.

In an experiment by Träff (1995) [11], two distributed versions of Dijkstra are compared. They use a setting with different processors and no shared memory, where communication is relatively expensive. The 20,000 vertices of the graph are divided over 16 processors. In one version, the Dijkstra algorithm is used in a "minimum-driven" way, only searching at vertices of which the shortest path is found. In the other version, called "update-driven", the search is continued at more vertices at a time, even before their minimum distance is found. In this way, labels may have to be corrected at a later stage of the computation. The update-driven version turns out to have better performance. This is consistent with the findings that very low Delta-values lead to a lower performance than intermediate values.

2.3 Multiple-pairs shortest path

Wang et al [12] note that usually single-source or all-pairs algorithms are used to solve the multiple-pairs shortest path problem. Each single-source run can be used to calculate all pairs with the same source or the same destination. To calculate the minimum number of such single-source runs, the minimum vertex cover can be calculated on a bipartite graph representing the pairs. They present a new algorithm that first applies a preprocessing step, determining an order to process the vertices based on the graph structure. Therefore, they consider their algorithm especially useful when the graph is fixed, but edge costs change between runs. In an experiment using grid graphs, the algorithm turns out to be comparable to other implementations of optimized versions of Dijkstra and others.

Klein [13] presents an efficient algorithm for MPSP specifically for planar graphs. The idea is to start from a fixed vertex and calculate its single-source shortest path tree. [6] The shortest path trees of neighbouring vertices can be efficiently calculated by determining what changes are necessary to the original tree. In this way, they can reuse previous results instead of running the single-source algorithm for each vertex. However, this optimization depends on the graph being planar. A general version of the algorithm

[6]http://courses.csail.mit.edu/6.889/fall11/lectures/L11.pdf

was presented by Cabello and Chambers [14] with time complexity O(g^2 n log n), where g is the genus of the graph. While this version in principle works for any graph, its performance very much depends on the genus. This is useful for almost-planar graphs, but does not seem to be very useful for a random graph such as a social network or the Web graph.

For Giraph, the Okapi project provides an multiple source shortest paths implementation. [7] It effectively runs a number of single-source computations in parallel, but combines communication from parallel runs into single, but larger, messages. In this way, the number of messages is reduced and performance is improved.

Like the example for Pregel, the Okapi implementation uses a label-correcting algorithm: each vertex stores the length of the shortest path found so far. When it is informed of a shorter path from the source vertex, it updates its stored value and informs its neighbors. Each vertex has a value of type Float and sends messages of the same type. Instead of running the SSSP algorithm for each vertex individually, multiple sources are selected and a SSSP instance is run in parallel. This is achieved by having each vertex store a map (type MapWritable) of ID's to distances. In the same way, messages sent between vertices contain a map of source ID's to distances. For example, vertex 1 might send the message {3: 10.0, 4: 15.0} to 2, meaning that it found a new shortest path from source vertex 3, via 1, to 2 of length 10; and also a new shortest path from source vertex 4. The receiving vertex 2 then compares each new length to the values stored in its own map to decide if the path is an improvement. If at a given iteration some improvement is found, all neighbors are informed by sending the new distance.

2.4 Distributed routing algorithms

Two important distributed algorithms for routing are Chandy-Misra and Merlin-Segall [15]. Both are used for a single-source shortest path computation, where each node in the network finds the length of the shortest path to the source and the first hop in the shortest path. However, Chandy-Misra has exponential message complexity in the worst case, while Merlin-Segall has a message complexity of $\Theta(N^2 \cdot E)$. For very large graphs, this will probably generate too many messages to perform well.

[7]https://github.com/grafos-ml/okapi/blob/master/src/main/java/ml/grafos/okapi/graphs/ MultipleSourceShortestPaths.java

2.5 Conclusion

For the multiple-source problem, there does not seem to be a general approach that avoids running the single-source algorithm multiple times. However, the number of messages can be reduced by performing them in parallel and combining messages sent in the same iteration. It would be interesting to try limiting the parallel search in a way comparable to the Delta-stepping method. As shown by Träff, in a parallel setting it is not recommended to limit searching in the "minimum-driven", ordered way like Dijkstra. However, postponing edges with large weights to a later phase may cause a significant decrease in the number of messages being sent around.

Chapter 3

Shortest paths in query languages

To extend the Lighthouse system for shortest path queries, an algorithm is needed that fits the computation model and the functional requirements. Aside from the algorithm, the query language used as input for Lighthouse must be extended as well.

On the one hand, it is desired to match the parameters and capabilities of the algorithm, so the query optimizer does not get excessively complex. If it were possible to express harder problems, e.g. the shortest path containing two adjacent vertices that satisfy some property, these problems could not be solved by the shortest path algorithm. Then, the query optimizer would have to analyse the query and decide if a specialized shortest path algorithm can be used, or an inefficient brute-force execution is needed. On the other hand, the extension should also be simple and easy to express for the user writing the query. In order to design the extension to the query language, some approaches in existing languages are explored.

In this chapter, a number of possible extensions for Cypher are presented, and one of them is argued to be preferable to the others. In the last section, a new operator is described to implement support in Lighthouse for the new type of shortest path queries.

3.1 Path queries in other languages

The existing Lighthouse project supports a subset of the Cypher query language, so we first evaluate existing approaches to shortest paths within Cypher.

The full Cypher language is implemented by the Neo4j graph database, an open source project written in Java. It includes support for variable length paths and calculating shortest paths.[1] Calculating weighted shortest paths is possible as well, but not in a

[1] Neo4j Cypher Refcard 2.1.6

very efficient way.[2] Variable length paths allow for queries such as calculating all cities within 200 km, but only by matching all paths towards cities and filtering them on total length. The language does not support general recursion or calculating fixed points.

Some other query languages do support recursion. A survey [16] has been done to analyse the capabilities of many web-related query languages. Most languages in this survey are used to query data stored in XML, RDF or Topic Maps. However, it is noted that all three data formats can be viewed as specific types of graphs. For example, XML is considered to be a "rooted, directed, non-ranked, ordered graph". Note that XML data can be considered a graph rather than a tree, taking into account the ID and IDREF attributes that create additional edges in its structure.

It is noted that some query languages specify "closure operators" to specify the transitive closure of certain types of edges (or relations). Others restrict closure to predefined edges, such as the subclass hierarchy in RDFS. A third group of languages support general recursion, a more expressive mechanism than the closure operation. The results of this classification are included as evaluation 2.1.13 in the survey. Based on those results, the languages are divided into classes as in the figure below (taken from [16], red marking added).

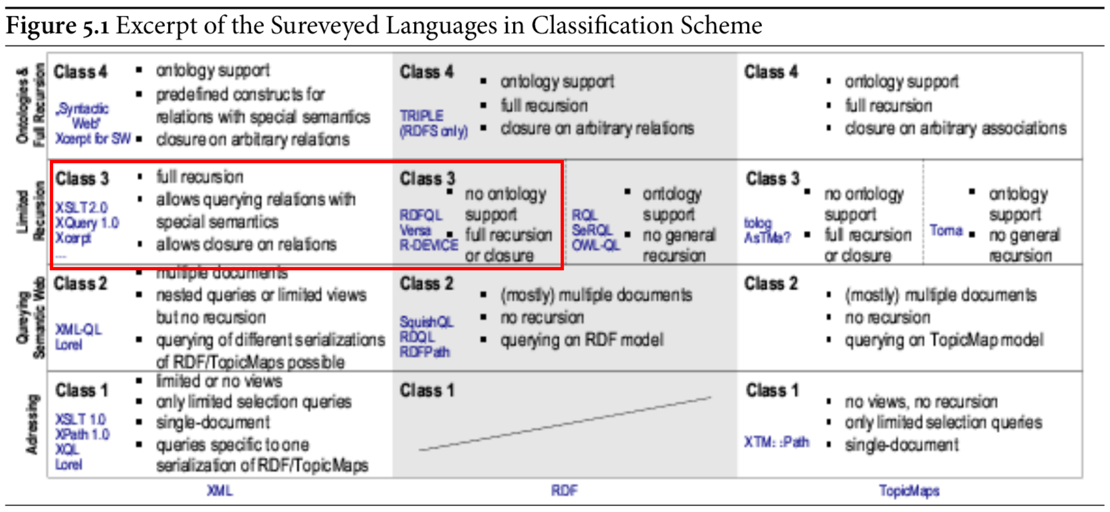


Figure 5.1 Excerpt of the Sureveyed Languages in Classification Scheme

	XML	RDF	TopicMaps
Ontologies & Full Recursion	**Class 4** „Syntactic Web" Xcerpt for SW ▪ ontology support ▪ predefined constructs for relations with special semantics ▪ closure on arbitrary relations	**Class 4** TRIPLE (RDFS only) ▪ ontology support ▪ full recursion ▪ closure on arbitrary relations	**Class 4** ▪ ontology support ▪ full recursion ▪ closure on arbitrary associations
Limited Recursion	**Class 3** XSLT2.0 XQuery 1.0 Xcerpt ... ▪ full recursion ▪ allows querying relations with special semantics ▪ allows closure on relations	**Class 3** RDFQL Versa R-DEVICE ▪ no ontology support ▪ full recursion or closure RQL SeRQL OWL-QL ▪ ontology support ▪ no general recursion	**Class 3** tolog AsTMa? ▪ no ontology support ▪ full recursion or closure Toma ▪ ontology support ▪ no general recursion
Querying Semantic Web	**Class 2** XML-QL Lorel ▪ multiple documents ▪ nested queries or limited views but no recursion ▪ querying of different serializations of RDF/TopicMaps possible	**Class 2** SquishQL RDQL RDFPath ▪ (mostly) multiple documents ▪ no recursion ▪ querying on RDF model	**Class 2** ▪ (mostly) multiple documents ▪ no recursion ▪ querying on TopicMap model
Adressing	**Class 1** XSLT 1.0 XPath 1.0 XQL Lorel ▪ limited or no views ▪ only limited selection queries ▪ single-document ▪ queries specific to one serialization of RDF/TopicMaps	**Class 1**	**Class 1** XTM: :Path ▪ no views, no recursion ▪ only limited selection queries ▪ single-document

FIGURE 3.1: Category of our intended language

Of those classes, most related to this project are the Class 3 languages supporting full recursion, which is referred to as Class 3(a) in the survey. Ontology support is not needed for the graph database extension we want to design.

XSLT is used for XML transformations, and is considered quite verbose for writing queries. This leaves two main examples in the XML category: XQuery and Xcerpt. The

[2]Example demonstrated on http://iansrobinson.com/2013/06/24/cypher-calculating-shortest-weighted-path/

recursion mechanism of XQuery has already been used to implement an inflationary fixed point operator [17]. The Xcerpt language has been designed by the REWERSE I4 group in an attempt to combine the best aspects of existing languages, having one language that can work with both XML and Semantic Web data. Like XQuery, it allows for recursion, but uses patterns to specify which data to return instead of paths such as XPath. It also allows for a concept called forward and backward 'rule chaining', which is considered an improved variant on the views on SQL [18].

In the RDF category, none of the three Class 3(a) languages are widely used. Instead, SPARQL has become the standard for querying RDF data. This is an extension of RDQL, a Class 2 language in the table. Some years after the survey was done, SPARQL 1.1 was published [19], allowing for property paths. Using property paths, it is possible to use closure of relations (a ZeroOrMorePath in the specification) or the non-existence of a path (NegatedPropertySet). Full recursion is still not supported, although a workaround has been published in the form of a query-executing function in SPARQL [20]. Before SPARQL was standardized, a review of RDF query languages was done with regard to recursion [21]. Two languages (Triple and N3) were rule-based, and allowed recursion by specifying rules for the base case and the inductive case. Another language, Versa, only allows for transitive closure.

SQL has also been extended to allow for recursive queries, in version SQL:99 [22]. Within a query, a recursive table can be calculated by specifying a base case and inductive case. This terminates when the least fixed point has been found. This is different from XQuery, where the termination condition needs to be made explicit in a recursive function.

3.2 Design of Cypher extension

We identified three related features that would be useful to support in the extended language for Lighthouse. They are listed in order of increasing expressiveness, where each feature is a more general version of the previous one.

- Paths of variable length, also known as reachability queries or relationship closure;
- Shortest path queries, with the following properties:
  - Conditions on the vertices and edges within the path
  - Supporting both the shortest path length and the path itself as result
  - Possibility to return multiple shortest paths (Top N)
  - Efficient calculation of multiple-source, multiple-destination paths

- Support for "recursive queries" based on an inflationary fixed point, like in SQL:99.

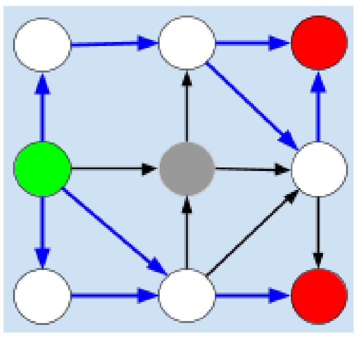


FIGURE 3.2: Example of top 2 shortest paths for two pairs, with one excluded vertex

Some of these features can be expressed in the full Cypher language, but are not yet supported by Lighthouse. For others, it is necessary to design an extension to the language itself. In this section, the current features of Cypher are presented that are useful in implementing the features above. Then, a number of possible extensions are discussed based on the advantages and disadvantages of each approach.

3.2.1 Current features in Cypher

Currently, Cypher provides two ways to specify a shortest path query. The first one is based on the number of hops, and does not support edge weights. It does not support the Top N feature either. In addition, the only way to apply conditions is in the WHERE clause, potentially removing the only result from the query. Filtering is applied after finding the shortest path, not before, so it is not possible to find the shortest path over a specified subset of the graph. For example, the query below will first find the shortest path from Start to Finish. If a node on that path then has a property called 'danger', the query returns an empty set.

```
MATCH p=shortestPath((a:Start)-[ROAD_TO*]->(b:Finish))
WHERE ALL (x in nodes(p) WHERE NOT x.danger)
RETURN p, length(p)
```

To work around the restrictions of the built-in shortestPath function, many users[3] apply a combination of the reduce function and the ORDER BY / LIMIT clauses. In this way, it is at least possible to express a weighted shortest path, and to apply a condition

[3]http://stackoverflow.com/questions/12587263/cypher-order-by-path-cost/12588013
http://iansrobinson.com/2013/06/24/cypher-calculating-shortest-weighted-path/
http://blog.bruggen.com/2013_08_01_archive.html

before choosing the path. However, we need a complex query to get the best path for each pair, and the resulting shortest path query is not feasible to detect by the query optimizer. Therefore, it is very hard to have the query optimizer switch to a specialized implementation for such queries. For instance, changing the keyword ALL into ANY or SINGLE would already be incompatible with a regular shortest path algorithm. If the system is unable to recognize when a specialized shortest path algorithm can be used, it would have to enumerate all paths and calculate all their lengths to decide which is shortest. This has exponential complexity and will not terminate in reasonable time for large graphs, as discussed in Section 5.1.2.

The query above can be adapted to a weighted shortest path using the following syntax, using the time spent traversing a road as its weight. By adding `LIMIT 5`, the query will return the top 5 shortest paths, instead of just one.

```
MATCH p=(a:Start)-[ROAD_TO*]->(b:Finish)
WHERE ALL (x in nodes(p) WHERE NOT x.danger)
RETURN p, reduce(time=0, r IN e | time +
   (r.distance/r.maxSpeed)) AS TotalTime
ORDER BY TotalTime ASC
LIMIT 5;
```

3.2.2 Discussion of possible extensions

For the three features described at the beginning of this section, a number of possible language extensions were explored and evaluated. For the first feature, variable length paths, the syntax is already provided in Cypher: `(a)-[r * 4..6]-(b)`. It allows for minimum and maximum length, both of which are optional. Implementing this feature in Lighthouse would be relatively simple. We now focus on the language extensions needed for the other two: shortest paths and recursion.

Shortest paths

For shortest paths, two alternatives are evaluated: one extending the existing shortestPath function with additional arguments for the weight function, conditions, and number of top N paths, and one alternative adding new aggregation functions. The first has the advantage that it does not require a new function, it just adds optional arguments that extend the function in a relatively natural way.

```
MATCH p=shortestPath((a:Start)-[ROAD_TO*]->(b:Finish),
     x in r | x.distance/x.maxSpeed, ALL y IN nodes(p) WHERE NOT y.danger)
RETURN p
```

If no cost function is given, the cost of each edge is considered to be equal, just like in the original function. For the cost function, we can use the same syntax as used by the reduce method of Cypher, so again little new syntax is required. The same holds for the third argument, which is a condition that would be moved from the WHERE clause to within the function. This provides a clear distinction between filtering before and after finding shortest paths.

There are also a number of downsides to this approach. First of all, it would be preferable if an existing Cypher feature could be used for the extension in Lighthouse. Second, the user is forced to decide whether to use a shortest path algorithm, instead of leaving this choice to the query optimizer. Finally, a natural way to add the number of pairs (Top N) is lacking, as is a way to easily refer to the distance of a path.

An alternative syntax extension is to implement shortest paths as an aggregation function. This is used in Cypher to find aggregated results over some result, such as minimum, sum, count, and more complex calculations such as arbitrary percentiles. In this variant, the 'minimum' aggregation function would be combined with the reduce function into a new shortest function that operates on paths:

```
RETURN startNode, endNode, shortest(path, x IN x.distance / x.maxSpeed,
    5), cost(path, x IN x.distance / x.maxSpeed)
```

While this syntax would be consistent with other aggregation functions, this variant would not completely solve the problem for the query optimizer: it is still possible to write complex conditions in the WHERE clause, making it impossible to run a shortest path algorithm. Then the query optimizer could decide to use the brute-force option of the reduce/minimum functions as a fallback method, but this decision would be complex to implement. It would have to evaluate if the path is subject to any conditions that can not be satisfied by excluding vertices or edges from the graph over which the algorithm is run.

Recursion

Inspired by the approach of SQL:99, a corresponding syntax is suggested for Cypher. This has a number of benefits: it is much more expressive than the other options, enabling variable length paths (closure) and shortest path queries. The change in syntax compared to the existing Cypher is relatively minor and one powerful mechanism could avoid the need for many different extensions to implement specific algorithms. However, it is both quite hard to optimize and hard to express shortest path queries with the desired functionality. Also, Cypher already has other syntax for most simple variable length paths, so it may be more consistent to extend the existing variable-length syntax.

The following syntax illustrates how such a recursive query could be used to traverse a hierarchy of employees and managers in an organisation. They are connected to their departments through `WORKS_AT` and `IS_HEAD_OF` edges, respectively.

```
MATCH (a)-[:IS_HEAD_OF]->(b)
WHERE NOT((a)-[:WORKS_AT]->())
RETURN a AS x, 1 AS n, NULL AS manager
UNION RECURSIVE
MATCH (x)-[:IS_HEAD_OF]->(:Department)<-[:WORKS_AT]-(y)
RETURN y AS x, n + 1 AS n, x AS manager
```

From a user perspective, it may be easier to write the recursive case in one clause, using the existing 'OPTIONAL MATCH' or a new 'RECURSIVE MATCH'. For example, compare the queries above and below. The disadvantage in this case is that the semantics are not very clear from the syntax, such as the distinction between variables for the base case and the recursive case. Additionally, it does not solve the other problems, such as the complications for the query optimizer. Therefore, the previous option is preferred over this one.

```
MATCH (a AS x)
OPTIONAL MATCH (x)-[:IS_HEAD_OF]->(:Department)<-[:WORKS_AT]-(b AS x)
WHERE NOT((a)-[:WORKS_AT]->())
RETURN a, x, b
```

3.2.3 Proposed language extension

The proposal in this section was presented at the Third Graph-TA workshop of March 18, 2015.[4] Compared to the earlier alternatives, it corresponds closest to the first extension for shortest paths. All of the new syntax was placed in the `MATCH` part of the query, like the existing shortestPath function. Because it is more powerful than a regular function,

[4]The proposal has also been discussed with Neo4J, who have stated they are also looking for a easier way to express cheapest paths. Neo4J is still considering multiple options, and suggested to use SUM OVER in the return statement - somewhat like the aggregation example in the previous section.

and requires more complex arguments, the syntax is different and separated from the shortestPath function.

```
MATCH path=()-[e* | r]->() CHEAPEST [n] SUM cost [AS len]
```

There are four arguments in this extension, marked in red. All of them are optional, except for the cost function.

- First, there is a selector on the edges in the path, which is applied before evaluating the WHERE condition. This is an important difference, as it enables filtering without reducing the number of paths found. Applying a filter afterwards, in the WHERE clause, may result in not having any result at all. This mechanism can also be used to apply a filter on the incident vertices, accessing them via the existing Cypher functions startNode(edge) and edgeNode(edge).
- The second argument is the number of alternative paths for each pair. This is different from a LIMIT clause, which is evaluated on the full set of results. If it is omitted, only the single best path is found for each pair. The number of paths should always be exactly this amount, unless not enough paths exist. In the case more than one path per pair is requested, we assume paths are allowed to contain cycles.
- After `CHEAPEST SUM`, a (monotonic) cost function is specified. This is called *cheapest* to distinguish from the *shortestPath* function, which only counts the number of hops. It is still possible to achieve the same result by specifying a cost function of 1, but much more complex calculations are possible as well. One restriction is that all variables are based on local information: either edge properties, or properties of the two incident vertices. For simplicity, the edge variable (e in the example above) in this context refers to each individual edge, whereas it would normally refer to the collection of edges.
- Finally, a method is desired to bind the distance found to a variable. Although it is possible in Cypher to find this distance again using the reduce function on the path, adding `AS x` is clearly preferable for usability.

Example use case

We have a graph where some vertices are named "Start", and some others are named "Finish". Additionally, the graph contains some vertices marked as "Danger" that should be avoided. For each pair of Start and End vertices, we want to find their top

3 fastest routes without crossing dangerous points. The time spent on a road is the distance divided by the speed limit. This problem can be expressed in the following query.

```
MATCH path= (a:Start)-[e* | not(endNode(e).danger)] ->(b:Finish)
CHEAPEST 3 SUM e.distance / e.maxSpeed AS length
RETURN a, b, path, length
```

Chapter 4

Implementation of the new operator

4.1 Existing operators in Lighthouse

As described by Filip [1], the Lighthouse system consists of three main components. First, there is a query parser that reads a Cypher query that is provided as input, resulting in a query graph. Then, there is a planner that estimates the cheapest query execution plan for the query. For this purpose, a statistics file is used containing information collected about the dataset. For example, if the query is to find friendships between a person in Amsterdam and one in New York, it would be faster to start looking for that pattern from the smaller set, the persons in Amsterdam. The third component is the execution engine, which takes a query plan and executes its steps on Giraph.

A query plan consists of a tree of operators, that operate on a binding table. The algebra is described in detail in [1] and contains six different operators. There are four local ones, which do not require starting a new superstep: Scan, Select, Project, and HashJoin. In contrast, the two global operators StepJoin and Move do need one additional superstep in order to send messages to neighbouring vertices.

This process is illustrated with a simple example. The query plan is `StepJoin(Scan(Person), Friend-of)`. In the first superstep, the Scan operation is executed: all vertices of type `Person` create a row for the binding table containing only their ID. Since it is a local operation, no communication is needed, and the rows are stored locally. In this way, a binding table is created that is stored distributively. The next operation, StepJoin, is performed in the same superstep. In this StepJoin operation, all vertices iterate over their local rows of the binding table. For each local row, they apply the join operation over each outgoing edge of the right type (`Friend-of`). This means a temporary copy is made of the row of the binding table, the target vertex ID is appended to this copy, and the copy is sent to the target vertex. After applying the join operation over each relevant edge, the old rows of length 1 are discarded and the new rows of length 2 have been sent to the target vertices. As communication in Giraph takes one superstep, the new rows are received at the second superstep. All the rows of length 2 together then make a new binding table that is still stored distributively.

A pipelined version of Lighthouse is currently in development in which the transfer of messages after each operator is limited to a fixed amount per superstep. Messages that would exceed the limit are postponed and stored locally in a queue until the next superstep. The corresponding execution step is added to each message to recognize older messages. The new operator described below is compatible with the new pipelined system, but does not support pipelining itself.

4.2 Operator for shortest paths

In order to efficiently find shortest paths, we need to extend the system with one or more new operators. As all the existing operators take either zero or one superstep, it is not currently possible to express a computation for which the number of supersteps is unlimited. To implement shortest paths, the most simple option is to add a corresponding new operator called ShortestPaths. Alternatively, we could add a more generic operator, potentially one enabling general recursion. In the end, this could make it easier to later extend the system for other graph algorithms. However, at this stage we chose to optimize the computation as a separate operator.

The new ShortestPath operator is a global one, with more complex behavior than the existing six. Unlike the other global operators, StepJoin and Move, the number of additional supersteps introduced by applying ShortestPath is variable. We will describe the behavior and implementation of this new operator.

In the process of parsing, a new parsing rule is introduced:

```
ShortestPath ( queryExpr , [ column_identifier ] , expr , expr, number )
```

The first argument contains the query plan that should be executed before applying the ShortestPath operator. Evaluating this sub-plan should result in a binding table containing the pairs for which shortest paths are to be found. Two columns of the table can together represent the set of source-destination pairs: one containing the identifiers of the source vertices and one for the identifiers of destinations. We assume that the sources are always in the last column, to be consistent with other operators like StepJoin.

For the column of destinations, there is no reason to assume it is located at any fixed position, so its location should be included in the query plan. It is represented by a column index in the second argument of the operator.

The third argument is used to specify a condition on the vertices that are allowed to be in shortest paths. This allows filtering both on labels and on property values of vertices. The fourth argument expresses how to calculate edge weights, as the name of a property or as expression. Finally, the last argument corresponds to the number of paths returned for each pair.

In the Lighthouse execution engine, global coordination is achieved by having each worker read the query plan from HDFS. Each worker then calculates what leaf of the query plan tree should be started at which superstep. Each message contains a counter representing which step of the query plan it corresponds to. When extending the system with a variable-length operator, it is not possible to use a fixed plan based on the number of supersteps, since the workers cannot predict how many supersteps the operator will take. Therefore we introduced a separate superstep counter for the Lighthouse core algorithm for which the value is still predictable. For this counter, a full run of the shortest path computation counts as one superstep. Each worker keeps a local version of this counter, without requiring any communication.

We use aggregation to inform all workers which ShortestPath operator is being executed, so each worker can lookup the relevant parameters and keep them available during the shortest path computation. In addition, the aggregator coordinating the phases of the shortest path computation is extended with an additional phase 0 at the beginning. This phase is used to wait until the pipeline is empty. Because the shortest path computation does not support pipelining, while the other operators in Lighthouse do, it is necessary to wait until all messages in the pipeline are collected in the source vertices. Only then should the system switch to the shortest path computation.

The new shortest path operator acts as a wrapper around the separate algorithm. When a part of the binding table is received, the source vertices are initialized and all received rows of the binding table are collected and stored in the vertex value. After running the algorithm, the operator iterates over the stored rows. For each row, the destination column is read again and the top N distances and paths are read from the results. For each entry in the top N, the row from the binding table is copied and the distance and path are added to the copy. The destination column is moved to the end, and the message is sent to the destination vertex. This is illustrated in Figure 4.1, in the communication marked in red. The next superstep, the Lighthouse core algorithm continues at the destination, in the same way as after applying another global operator.

The shortest path computation itself is also extended: at phase 1, the requested pairs are sent from sources to destinations. At an additional final phase, all paths and distances found are sent through the operator and all temporary state information is cleared. Some conversion functions are used to translate between classes used by Lighthouse and those used by the shortest path algorithm. They are used to read and write the vertex value, to calculate the relevant edge weights and to evaluate the vertex exclusion condition during the algorithm, all based on the parameters of the shortest path operator.

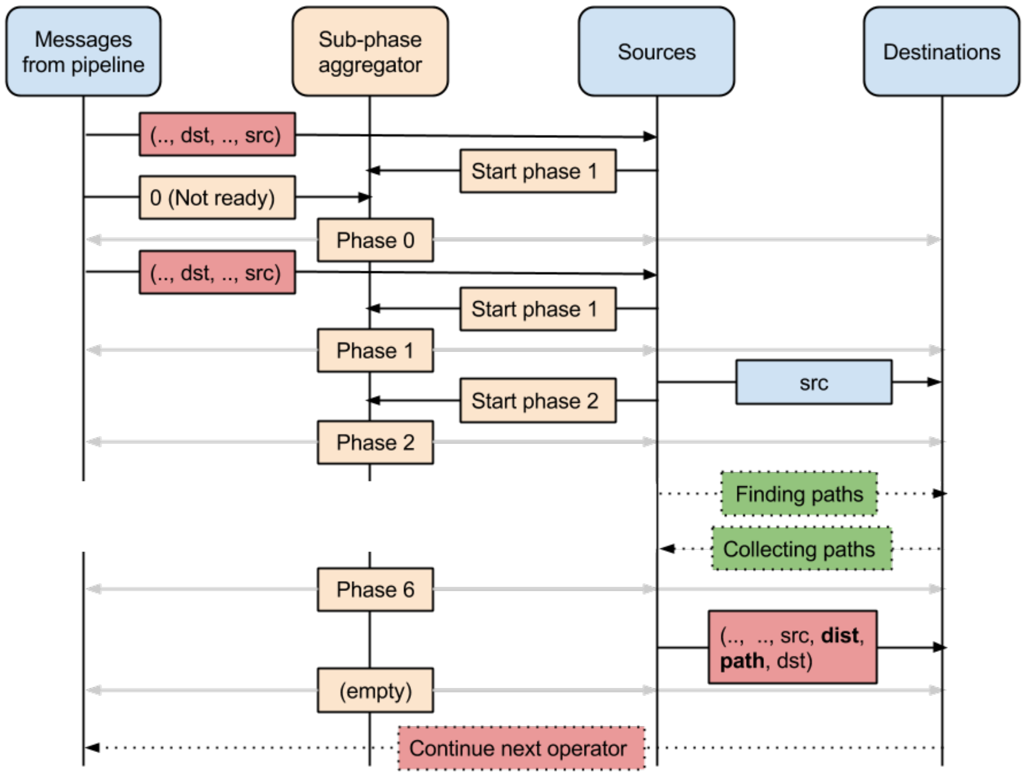


FIGURE 4.1: Illustration of communication around shortest path computation

Chapter 5

Extended Shortest Paths Algorithm

In this chapter, an algorithm for top N weighted shortest paths is presented, designed for the BSP computation model. The algorithm is evaluated on three social network datasets of different size. Different representations of paths are tested, and different methods of pruning are implemented. In the first variant using pruning, a boundary is used that is increased each superstep. In the second variant, a flexible boundary is used based on statistics collected during the previous superstep.

5.1 Basic algorithm

The algorithm from the Okapi project, described in section 2.3, has been extended for the requirements formulated in the design of the Cypher language extension:

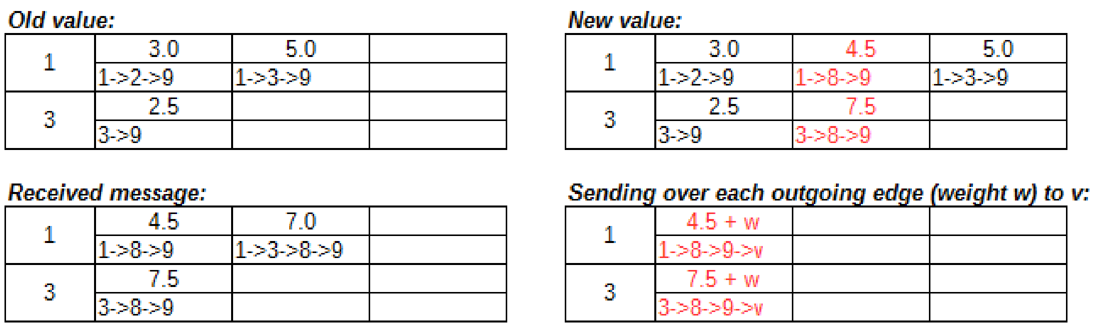
- possibility to return paths as well as the distance
- a feature to return the top N shortest paths between two vertices
- an option to run the algorithm over parts of the graph
- an option to only return results for certain destinations

The input to the algorithm is a list of sources, a list of destinations, a list of excluded vertices, the number of paths to return for each combination, and the graph itself. The output is a list of (source, destination, length, path) tuples.

This requires a more complex data structure than the map used in the original version. For each source, each vertex needs to store not just the minimum distance, but a sorted list of distances and a list of corresponding paths. The path can be stored as text that is later parsed again, or as a list of vertex IDs. We have chosen the latter option.

As in the original version, the messages between vertices have the same structure as the values stored by each vertex. They contain the new or improved paths found by the vertex sending the message.

In each iteration, a vertex creates a new map to store all improvements found and then processes each received message (Line 16 to 20). For each source described in the message, the (sorted) list of paths is merged with the list of the top N shortest paths found yet (Line 26 to 34). This results in a new list of shortest paths that is stored in the vertex value. Each occurrence of a received path in this new list is then an improvement to the old list. Therefore, it is added to the map of improvements (Line 37).



Old value:

1	3.0	5.0	
	1->2->9	1->3->9	
3	2.5		
	3->9		

New value:

1	3.0	4.5	5.0
	1->2->9	1->8->9	1->3->9
3	2.5	7.5	
	3->9	3->8->9	

Received message:

1	4.5	7.0	
	1->8->9	1->3->8->9	
3	7.5		
	3->8->9		

Sending over each outgoing edge (weight w) to v:

1	4.5 + w		
	1->8->9->v		
3	7.5 + w		
	3->8->9->v		

FIGURE 5.1: Example situation in the basic algorithm

At the end of the iteration, a modified version of the map is built for each outgoing edge (Line 42 to 51). For each outgoing edge e to some target vertex v, all distances in the map are increased by the weight of e, and all paths are updated by adding v at the end. Each modified version of the map is then sent to the target vertex v. If in a certain iteration a vertex did not find any improved path, the map of improvements is empty. In that case, nothing is sent to neighboring vertices. Instead, the vertex votes to halt the computation. If all vertices vote to halt and no messages were sent, all shortest paths have been found and are present in the vertex values. A custom output format is used to build the tuples containing the sources, destinations, path lengths and full paths.

5.1.1 Experiments

A shortest path implementation in Lighthouse was tested on the DAS-4 cluster. We used three datasets generated by the LDBC-SNB Data Generator. For edge weights

Algorithm 1 Pseudocode of the basic version of the algorithm

```
procedure INITIALIZE(vertex)
    vertex.paths = {}
    if vertex.isExcluded then
        vertex.voteToHalt(); return
    end if
    if vertex.isSource then
        vertex.paths.put(vertex.ID, [(0.0, [vertex.ID])])
        for edge : vertex.getEdges() do
            update = {(vertex.ID, 'distances'): [edge.weight], (vertex.ID, 'paths'), [[vertex.ID, edge.targetID]]}
        end for
        send update to edge.targetID
    end if
    vertex.voteToHalt()
end procedure
procedure COMPUTE(vertex, messages)
    changes = {}
    if vertex.isExcluded then
        vertex.voteToHalt(); return
    end if
    for msg : messages do
        for each key (src, 'distances') in msg.keys do
            load currentDistances and currentPaths from vertex.value
            load receivedDistances and receivedPaths from msg
            load changedDistances and changedPaths from changes
            initialize newDistances and newPaths
            curIndex, rcvIndex, chdIndex = 0
            for i = 0; i < TOPN; i++ do
                if currentDistances[curIndex] $\leq$ receivedDistances[rcvIndex] then
                    add a current path to new lists, updating indices
                else
                    add a received path to the new lists, updating indices
                    add the received path to the changed lists
                end if
            end for
            store newDistances and newPaths in vertex.value
            if changedDistances.size > 0 then
                store changedDistanceList and changedPathList in changes
            end if
        end for
    end for
    if changes.size() > 0 then
        for edge : vertex.getEdges() do
            distanceUpdates = {};
            for sourceID : changedMap.entrySet() do
                load newDistances and newPaths for sourceID
                increase each distance by edge.weight
                append edge.targetID to each path
                store newDistanceList and newPathList in distanceUpdates
                send distanceUpdates to edge.targetID
            end for
        end for
    end if
    vertex.voteToHalt()
end procedure
```

	LDBC Scale factor	Person vertices	Friendship edges
Dataset Rnd1K	N/A	1000	50000
Dataset SF1-time	1	10993	451522
Dataset SF3-time	3	26982	1366816
Dataset SF10-time	10	72949	4641430

TABLE 5.1: Sizes of different datasets used in the experiments

Dataset SF1-time

Number of Workers	1	3	5	8	16
Computation time (ms)	86149	26737	17924	17002	14345
Path reconstruction (ms)	2894	1803	1574	1773	4113
Memory usage (MBs)	669	638	711	1117	1970
Communication (MB)	806	808	808	806	808
Speedup	1	3,22	4,81	5,07	6,01

Dataset SF3-time

Number of Workers	1	2	4	8	16	32
Computation time (ms)	272974	129628	68053	44737	31985	30040
Path reconstruction (ms)	5718	3842	3106	2736	2837	3303
Memory usage (MBs)	1830	1413	1448	1809	2966	5250
Communication (MB)	2655	2661	2658	2656	2660	2662
Speedup	1,00	2,11	4,01	6,10	8,53	9,09

Dataset SF10-time

Number of Workers	1	2	4	8	16	32
Computation time (ms)	>1M	492215	222406	126173	89118	72001
Path reconstruction (ms)		9956	6976	5303	5588	5438
Memory usage (MBs)		4805	3873	4313	6195	10189
Communication (MB)		9324	9339	9335	9318	9306
Speedup (based on 2 workers) [1]		2,00	4,43	7,80	11,05	13,67

TABLE 5.2: Results for initial experiment on LDBC-SNB datasets

on friendship graphs, the time elapsed since the edge creation time was used. Some other, earlier experiments are based on a random graph containing 1000 vertices with 50 outgoing edges each, where edge weights are uniform between $[0, 1]$. The four datasets are compared in Table 5.1.

The results are shown in Table 5.2.

In the second experiment, the goal was to analyze the overhead of the algorithm extensions: keeping data structures for storing multiple paths and corresponding distances. Therefore, the algorithm was compared to the original MultipleSourceShortestPath computation from the Okapi project. This is tested on dataset Rnd1K.

The expected increase in communication would be about factor 5 (paths) * 6 (average path length). For example, in MSSP a result would be (23.0), while in the extension it would be something like: [1,2,5,6,7,8]: 23.0; [1,2,4,6,7,8]: 24.5; [1,2,6,7,8]: 25.5;

	Communication (Bytes)	Communication (# Messages)	Total time for supersteps (s)
MSSP: Top 1, No paths	5629521	208324	9.366
Extension: Top 5, Paths as lists	182204626	402628	35.92

TABLE 5.3: Result compared to original algorithm from the Okapi library

[1,2,5,7,8]: 27.0; [1,4,5,6,7,9,8]: 28.5 The observed increase is of factor 32.3, so the extended algorithm seems reasonably efficient compared to the original MSSP implementation.

5.1.2 Discussion

Comparison to regular database query

An alternative approach to finding shortest paths is to find all paths up to a fixed maximum length, sum their edge weights and sort by the distance obtained in this way. However, this brute-force approach can only work for very small paths. Assume there is a very small dataset of 1000 vertices, each of which has 50 outgoing edges. The total number of paths of length 8 that start from a given source, is therefore 50^8. Since there are 1000 vertices in the graph, a fixed source has on average $50^8/1000$ ways to reach a fixed destination in a path of 8 hops. To find the shortest path for just one destination would therefore require evaluating 39 billion paths.

Possible improvements

The results from our experiments suggest a number of improvements to the basic implementation of this algorithm:

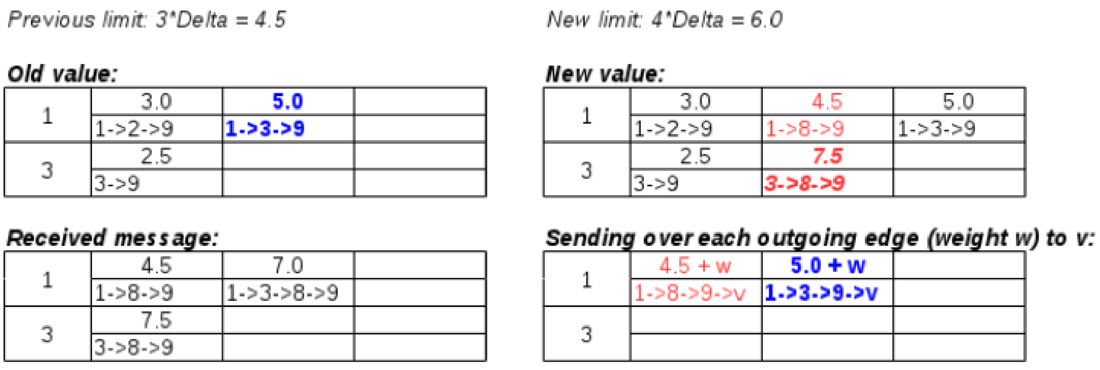
- From the experiment above, it seems like lists use a very inefficient representation when serialized to a message. By finding a smaller representation of this data structure, communication could be reduced. The reason for this was found to be that the number of vertices was quite small. Representing vertices using 64-bit identifiers takes more space than the string representation of them, as all identifiers below 10^7 are represented as seven bytes or less.
- Another possibility is to remove paths completely from the algorithm. Instead, each vertex could store the ID of their predecessor(s) for each shortest path. Afterwards, paths could be created from the predecessors listed in each vertex value.

- Postponing edges with large weights to later iterations, using buckets like Delta-stepping. A downside to this approach is that a good setting for Delta must be found for an efficient run, which can be based on statistics about the distribution of edge weights. However, this may be especially useful for Lighthouse, since the system already keeps statistics on many values present in the dataset.

- The algorithm could also use global or local coordination to find out which vertices know a relatively short path from the source. They would get to broadcast their path immediately, and relatively bad paths are postponed to a later iteration.

5.2 Postponing using fixed parameter

Inspired by the Delta-stepping algorithm, an improved version of the algorithm was implemented, to reduce the exponential growth of the exploration in the first few super-steps. The main idea is to have each vertex store its paths to the source in a number of "buckets", depending on the length of the path. For example, bucket 2 would contain all paths with length between Δ and $2 * \Delta$. All paths in bucket X are not broadcasted to neighbours until superstep X. In order to find a good setting for Δ, we can use the statistics kept by Lighthouse. Note that this is not equivalent to the Delta-stepping method, where the outgoing edges are divided into buckets based on their edge weights.


*Previous limit: 3*Delta = 4.5*

Old value:

1	3.0	**5.0**	
	1->2->9	**1->3->9**	
3	2.5		
	3->9		

Received message:

1	4.5	7.0	
	1->8->9	1->3->8->9	
3	7.5		
	3->8->9		

*New limit: 4*Delta = 6.0*

New value:

1	3.0	4.5	5.0
	1->2->9	1->8->9	1->3->9
3	2.5	**7.5**	
	3->9	**3->8->9**	

Sending over each outgoing edge (weight w) to v:

1	4.5 + w	**5.0 + w**	
	1->8->9->v	**1->3->9->v**	
3			

FIGURE 5.2: Example situation in the improved algorithm

The effect of this improvement has been tested using dataset Rnd1K. One vertex is selected as source, and the top 5 shortest paths are calculated from the source to each vertex. In the improved version, parameter Delta is set to 0.5. Computation is again divided over three workers.

Clearly, there is a significant difference in runtime and communication. Especially in early supersteps, a lot of unnecessary communication is prevented. Still, total communication is about 300 times larger than the output of 376735 bytes.

	Communication (Bytes)	Communication (# Messages)	Total time for supersteps (s)
Basic algorithm	182204626	402628	35.92
Improved version	92662001	337032	25.605
Difference	-49%	-16%	-29%

TABLE 5.4: Results for improvement using Delta-stepping

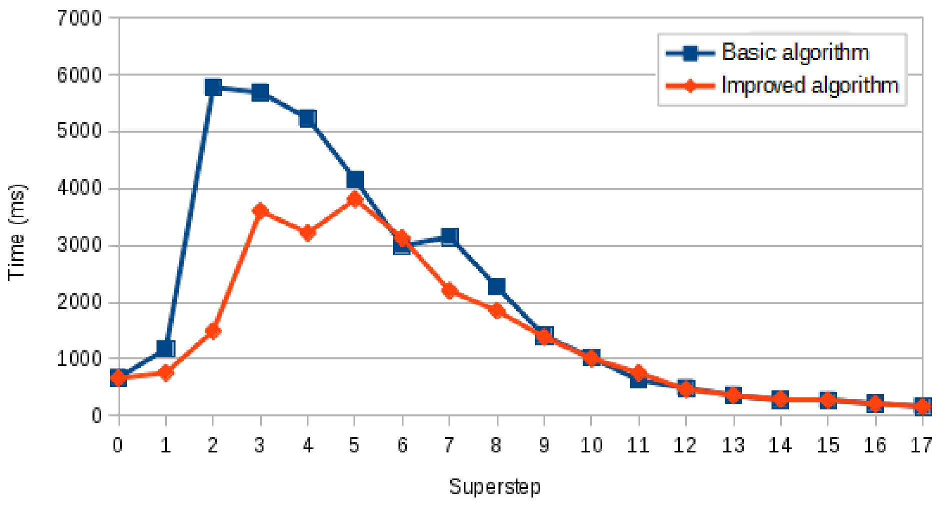


FIGURE 5.3: Comparison between runtime of basic and improved algorithm

5.3 Approximating Dijkstra using aggregations

One of the challenges in developing an efficient shortest path computation on the BSP model is to prune the search based on the information collected so far. In Dijkstra, this is done in a very effective way, by keeping a priority queue containing vertices. In each iteration, one vertex V with the smallest distance to the source is removed from the queue. All of its neighbours are then updated, potentially finding a smaller distance. A vertex can only be at the front of the priority queue if its smallest distance is fixed, avoiding unnecessary work.

In the BSP model, an exact implementation of this algorithm would require a synchronisation step after each iteration, to decide which vertices are in the queue and which one has the lowest distance. Since one vertex is processed in each iteration, a lower bound on the number of supersteps would be the number of vertices in the graph. (Without top N results, it would be exactly this number.) In each superstep, only the neighbours of one vertex would execute one operation, while all others are idle. Clearly, a way is needed to process more work in parallel.

Two alternatives have been described earlier. First, a modified parallel version of the Bellman-Ford algorithm was implemented, extended to return the paths themselves and find lists of top N results. This parallel version of Bellman-Ford is a common approach to shortest paths in the Pregel model, with the original Pregel paper even using it as one of the example applications of the framework. The main advantage is that a significant amount of work can be done between synchronisation steps, which very well fits the BSP model. A disadvantage is that the total amount of work is much higher than needed in a more sequential search. For example, in a graph `A-[1]->B-[1]->C`, `A-[10]->C`, the long path of 10 will be found and broadcasted one superstep earlier than the path of 2.

Therefore, a modified version was presented, inspired by the Delta-stepping algorithm. This prevents some unnecessary work by keeping a limit that is increased in each superstep. Paths with a large distance are not immediately broadcasted, but postponed to later iterations. In this way, shorter paths with a larger number of hops can be found in the meantime, and the reference to the long path may be replaced before it is broadcasted to neighbouring vertices. This modification was found to reduce runtime and communication significantly: 29% and 49% respectively, in experiments using a random graph. However, a disadvantage is that a value for Delta must be determined in advance, based on the distribution of edge weights.

Another parallel approach to finding shortest paths in the BSP model is described and evaluated by Goudreau et al [23]. In their algorithm, each worker maintains their own priority queue, running Dijkstra for a fixed period of time within their subgraph, while storing updates to vertices on other workers. After this period of time, all updates are sent to the other workers and the process is repeated in a new superstep. This does not fit well on the Pregel model, since the order of the vertices' computations is now fixed, and needs to be coordinated by the worker. Instead of the vertex-centric approach of Pregel, this requires a worker-centric approach.

5.3.1 Algorithm

As described above, it would not be feasible to maintain a global priority queue and use the Dijkstra algorithm, as the computation model does not allow communication between workers in the same superstep. However, it is still desired to be able to prune long paths during the search, even if they contain less hops than their shorter counterparts. While the Delta-based approach is able to achieve this, a more flexible alternative is preferable. Therefore, we explore a different way for vertices to decide when to postpone broadcasting certain distances. This decision could be based on statistics aggregated during the previous superstep, that provide information about the current state of the search.

When restricting statistics to the information available at the end of the previous superstep, the most useful information is likely to be the distribution of distances that have been sent in that superstep. Based on this distribution, a vertex could decide to only broadcast new distances that are in the top X% of all distances at the current frontier of the search. Instead of advancing the search at all vertices at the same time, this would make the search behave more like the Dijkstra algorithm.

For a multi-source shortest path algorithm, the situation would be slightly more complex. It could be the case that one source is very close to the majority of the vertices, while the other source is located very far away. In that case, the total distribution of distances is not very useful, because the two instances of Dijkstra are independent of each other. Since the two instances have a different frontier during the search, it seems better to keep separate information on the corresponding distributions.

The implementation of this idea can be found in the class PercentileEstimationAggregator. The aggregator keeps a map of source vertex IDs to lists of distances. Each list of distances is a random sample of paths that were discovered in the last superstep. To find an exact percentile, all the values would need to be kept and sorted, but the exact percentile is not required. After all, the only purpose is to determine if new distances are likely to belong to relatively short paths.

The option to postpone broadcasting certain new distances requires a way to track which distances still need to be broadcasted. Otherwise, some paths could be lost or sent multiple times. There are a number of ways this can be implemented. Firstly, each vertex could keep a boolean value for each distance to indicate which have been broadcasted. Alternatively, they could keep a list of all distances that have been postponed, together with the source vertex they belong to. Depending on the average size of this list, either the first or the second option would be more efficient. A third option is not storing this information at all for individual distances, but only the broadcast limit during the previous superstep. All distances between the previous broadcast limit and the current limit are then sent to the neighbours. This reduces the amount of information to store, but introduces a new restriction that the limit can only increase in subsequent supersteps; otherwise some messages would be sent repeatedly. Since this restriction does not seem to be a problem, we initially implement the third option.

5.3.2 Results

This algorithm was compared to the basic algorithm, and evaluated with regard to a number of metrics: the amount of paths received by vertices, to measure the degree of pruning; the amount of communication required; the number of supersteps, and their

	Paths exchanged	Communication (B)	Supersteps	Time (s)
Basic algorithm	14594143	171941655	18	52.712
Pruning at 11/50	6798145	137209745	42	92.237
Pruning at 6/50	6535650	138685540	62	90.836
Pruning at 3/50	5347701	139103432	139	118.580
Pruning at 2/50	$\sim$ 4960682	$>$ 126922608	$>$ 250	DNF

Table 5.5: Results for improvement using Dijkstra approximation

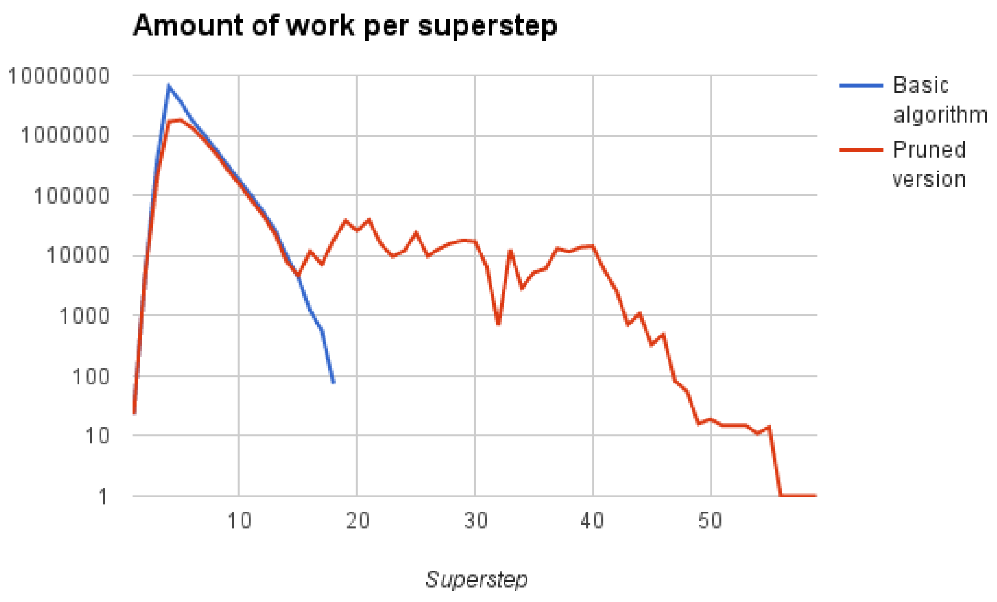
total runtime. Since pruning in the new algorithm is based on random subsets, all experiments below were repeated three times to reduce the effects of random variations. The dataset used was SF1-eucl+rnd. All paths starting from an arbitrary source[2] to all vertices were calculated, using top 5 results for each path. Different percentiles are used for pruning: percentile x/y means that samples of size y are kept and the x^{th} smallest out of them is used as a limit.

In order to evaluate the number of paths exchanged during each run, a sequential version of the algorithm was implemented in Python. This version has many similarities with the Dijkstra algorithm, with the main difference that not one shortest distance, but five shortest distances are eventually found for each vertex. Additionally, a counter is kept to simulate the number of paths exchanged during the run. Each time a new distance is processed at some vertex, the counter is increased by its outdegree, simulating that the distance is broadcasted to its neighbours. As expected, the final value of this counter is found to be 2257610: the total number of edges multiplied by 5 (for top 5 results). This confirms no suboptimal paths (i.e. paths outside the top 5) are being exchanged during the sequential version.

Using this information to evaluate the new parallel version, we find a large reduction in the number of suboptimal paths exchanged. For pruning at 3/50 of the current frontier, 75% of the suboptimal paths are first postponed and eventually overwritten by better paths. When pruning at lower values, this percentage will be even higher. However, runtime is much higher in the pruned version. To explore why, the runtime and work at each superstep was measured (Figure 5.4).

Based on the results, there are two main problems in the new parallel algorithm. First, there seems to be significant overhead in the process of aggregating and managing postponed paths. Second, the idea of postponing a fixed percentage does not work very well at the second half of the run. Pruning is not needed anymore at this stage, as the total amount of work remaining is low already. By postponing a fixed percentage each time, there are many supersteps with very little work at the end. As an alternative, the percentile could depend on the total amount of paths sent during the previous superstep.

[2]Vertex identifier of 16492675476531 in the dataset



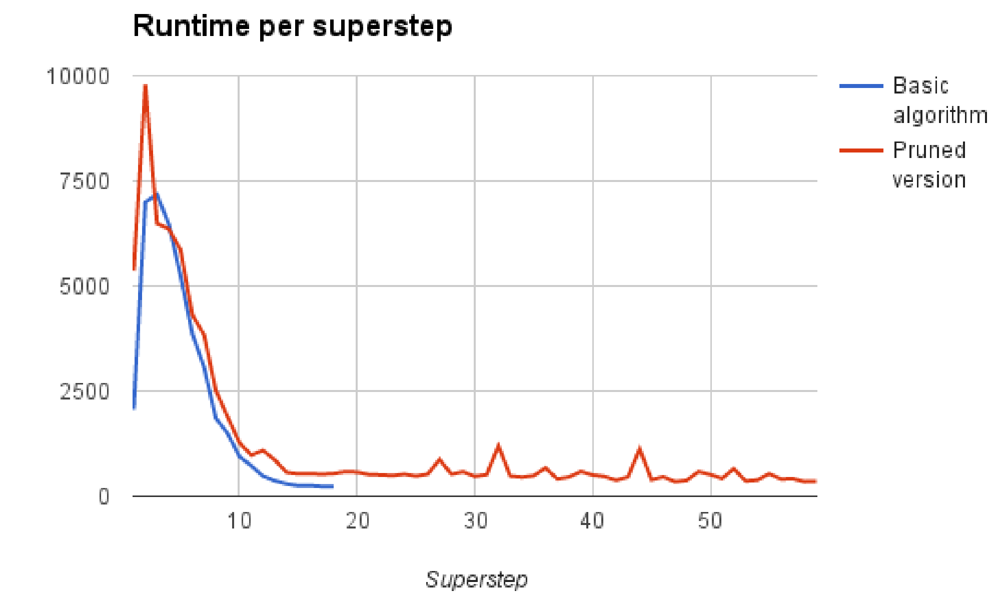


FIGURE 5.4: Comparison between basic algorithm and pruned version

5.3.3 Effect of edge weights

The main goal of the new algorithm is to avoid broadcasting paths that are unlikely to be optimal. This reduces the problem we observe in the basic algorithm, where long paths with few hops are traversed earlier than short paths with many hops. Clearly, this can only make a difference in a graph where the length of a path in terms of hops is different than the length in terms of distance. If each edge would have equal weight, such as in an unweighted graph, the original, breadth-first approach will probably perform better. The total amount of paths exchanged should be equal, but the new algorithm introduces some overhead in the process of comparing distances at the frontier.

Another interesting special case are road networks, where all vertices are assigned coordinates on a map, and the edge weights are (close to) the Euclidean distances between two vertices. Our hypothesis is that the optimization would be less useful in this situation, because of the triangle inequality that implies that a two-hop path can never be shorter than an one-hop path.

In the following experiment, we measure the influence of edge weights on the efficiency of the algorithm. Three algorithms are considered: a sequential Dijkstra algorithm, the original parallel algorithm, and the parallel approximation of Dijkstra described in this section. The first is not measured, as it has been confirmed before that the number is always $N \cdot E$, where N is the number of alternative paths in the top N setting and E is the number of edges.

In order to compare the influence of different factors, ten different datasets were generated based on the LDBC-SNB dataset SF1. Since the network of friends knowing each other is fixed, the differences are in the definition of edge weights, as listed in Table 5.6.

Since the starting vertex has a large influence on the amount of paths explored (a central vertex will sooner find optimal paths), we used a random subset of 25 vertices. The top 5 paths for each of the 25 sources to each vertex are calculated. The new algorithm is run three times and the average is reported. In the new algorithm, the percentile at 2/50 is used for pruning, and after 10 supersteps pruning is disabled. The number of workers is set to 8, but is not expected to make a difference on the amount of work.

Conclusion

The results in Table 5.7 confirm that the assignment of weights to edges greatly influences the effectiveness of both algorithms. It would be expected that the new version is especially useful in cases where the breadth-first variant does not perform well; however,

SF1-unit	Unweighted dataset (unit weights)
SF1-rnd	Random weights (uniform between zero and one)
SF1-time	Number of days since friends connected, at the last date in the dataset. The idea is that people still know each other in real life at the time they connect, while many old connections have not seen each other in years. On the other hand, it could be argued that distance should not increase but reduce, as people know each other for a longer time.
SF1-eucl	Euclidean distance, after assigning each person a random geographic coordinate.
SF1-eucl+rnd	Euclidean distance between places where persons are located (using DBpedia information), adding a random small weight between 0 and 6 kilometers.
SF1-eucl+pos	Same as SF1-eucl+rnd, but instead of adding random weights to the edges, each person is assigned a random location within 6 kilometers of his place.
SF1-soc-disc	Social distance as in query 14 of the LDBC benchmark: a comment of person A on a post of person B counts 1.0 towards their connection, while a comment on a comment counts for 0.5. Distances are then calculated as 1/(1+connection). This introduces relatively many equal weights of 1, 1/1.5 and 1/2.
SF1-soc-cont	Same as SF1-soc-disc, but using a continuous scale by also adding 0.2 when person A likes a post or comment of person B, and adding a random value between 0 and 0.1. This situation would better represent cases where some continuous value is taken in account when calculating social connections.
SF1-weak-soc-disc	Like SF1-soc-dist, but using social connection as distance metric, meaning that weak links are very close to each other.
SF1-weak-soc-cont	Like SF1-soc-cont, but again using social connection as distance metric.

Table 5.6: Variants on dataset SF1-time using different edge weights

this is contradicted by the results. Instead, we find that the most efficient datasets in a breadth-first setting, aside from the unweighted graph, have the largest relative improvement. Both datasets based on social connections are improved by around 70%. The smallest improvement is found for both random edge weights and time-based edge weights, suggesting that timestamps have been randomly generated in a uniform distribution in the LDBC social network benchmark dataset.

Dataset	Average hop count	Supersteps	Paths explored	Ratio of sub-optimal paths
SF1-unit	4.24	7	56440250	0%
SF1-rnd	11.13	27	358228615	535%
SF1-time	11.27	24	361185027	540%
SF1-eucl	6.72	19	186845641	231%
SF1-eucl+rnd	7.84	21	230070536	308%
SF1-eucl+pos	7.85	26	234126115	315%
SF1-soc-disc	6.23	15	173787662	208%
SF1-soc-cont	6.00	16	162583348	188%
SF1-soc-inv-disc	5.04	8	103990451	84%
SF1-soc-inv-cont	10.43	25	340394198	503%

	Supersteps (new version)	Paths explored (new version)	Ratio of sub-optimal paths	Reduction in sub-optimal paths
SF1-unit	8	56440250	0%	N/A
SF1-rnd	27	285204263	405%	24.2%
SF1-time	25	287641245	410%	24.1%
SF1-eucl	24	110374719	96%	58.6%
SF1-eucl+rnd	29	149433665	165%	46.4%
SF1-eucl+pos	30	168252890	198%	37.1%
SF1-soc-disc	16	94313119	67%	67.7%
SF1-soc-cont	17	87034822	54%	71.2%
SF1-soc-inv-disc	14	68680143	22%	74.3%
SF1-soc-inv-cont	26	262202944	365%	27.5%

Table 5.7: Influence of different edge weights

Chapter 6

Optimizations to the algorithm

A number of optimizations to the algorithm of the previous chapter are presented and evaluated. These optimizations are independent of the method chosen for pruning. Therefore, each optimization can be combined with any of the algorithm variants.

Section 6.1 and 6.2 describe two ways to reduce the amount of data stored for each intermediate path. First, the amount of data each vertex stores for a path is reduced to a single vertex ID. Then, it is reduced to nothing at all, except for the distances.

In Section 6.3, different methods to serialize maps are evaluated. Finally, Section 6.4 and 6.5 present ways to benefit from the features provided by the Pregel model, such as aggregation and message combiners. Using aggregators, the search can be pruned at distances higher than the current top N, and terminated when all non-explored paths are higher than the top N. This reduces the number of supersteps and avoids a potentially large number of small supersteps at the end.

6.1 Distributed storage of shortest paths

The original shortest path algorithms described in the Pregel paper and implemented in the Okapi project only return the total distance of the shortest paths, but not the path itself. We extended this algorithm to return the list of vertices contained in a path. The downside of this extension is a large increase in communication, because the lists that are passed around can be of arbitrary length. Instead of sending lists for each path, we want to represent each path only by the vertex ID of the predecessor. When the computation is finished and all calculated paths are fixed, the paths should be reconstructed from the references to predecessors. This is done by the 'CollectPathsComputation', described in this section.

6.1.1 Algorithm design

In order to reconstruct the paths, there are two main approaches possible: building paths starting at the source or in reverse direction by starting at the end. The latter approach has two advantages. Firstly, we can forward directly to the following vertex in the path, because the predecessors are known locally. Second, all the paths with an endpoint that is not selected as destination can be ignored immediately. There is also a disadvantage: when having one source and many destinations, all paths from the source will end up at a single vertex, potentially resulting in bad scalability. However, path reconstruction should in general be a very small part of the total work, as shown by the experiments below. Therefore, we have chosen to reconstruct paths starting from the destinations, back to the source vertex. The basic idea of the algorithm is to route the total distances corresponding to paths from the destination to the source, using the predecessor references as a routing table, and appending vertex ID's along the way. In order to support top N paths, it is not possible to use a regular routing table, since we need to restore the sub-optimal paths as well. Therefore a "routing table" is used that also contains up to N entries for each source. In this way, it is possible to reconstruct multiple different paths from a destination, even when the destination has no information to distinguish them.

This routing table is the result of the shortest path computation, and stored in the vertex value. It is a map of vertex IDs to top N lists of predecessor IDs. If vertex 6 has routing table $\{1 : [4, 4, 5, 5, 5]\}$, then it knows five paths from source 1 of which the best two are through 4 and the rest are through 5. The paths are forwarded by sending messages of type MapWritable, mapping partially reconstructed paths to their distances. A partially reconstructed path is stored as a list of vertex ID's: first the source, then the destination, followed by zero or more steps (in reverse order) from the destination.

At the initialisation step (Line 9), each destination creates a partially reconstructed path containing the source and the destination (Line 12). For each distance and predecessor belonging to the source-destination pair, the path and its distance is sent to the predecessor (Line 14). If the routing table contains <no predecessor>, i.e. when routing from X to X, the source of the path is reached. The partially reconstructed path is then converted to a normal path and saved (Line 3).

All following supersteps are used to process incoming messages. Each message contains a number of partial paths, with one or more distances for each partial path (Line 19). For each partial path, the source is read and the corresponding routing table is used to send the path to the right predecessors (Line 23). Before the partial path is sent, the

Algorithm 2 Pseudocode to reconstruct top N shortest paths

```
 1: procedure PROCESS(newPath, distance, pred)
 2:     if pred == NO_PRED then
 3:         save path newPath with distance
 4:     else
 5:         messagesFor[pred][newPath].add(distance)
 6:     end if
 7: end procedure
 8: procedure COMPUTE(vertex, messages)
 9:     if vertex.isDestination and initialization step then
10:         for each known source src in vertex.value do
11:             load distanceList and routingList for src
12:             newPath = [src, vertex.id]
13:             for i = 0; i < distanceList.size(); i++ do
14:                 process(newPath, distanceList[i], routingList[i])
15:             end for
16:         end for
17:     end if
18:     for each message msg do
19:         for each entry (path, distances) in msg do
20:             newPath = path + vertex.id
21:             load routingList for source path[0]
22:             for i = 0; i < distances.size(); i++ do
23:                 process(newPath, distances[i], routingList[i])
24:             end for
25:         end for
26:     end for
27:     for each (recipient, message) in messagesFor do
28:         send message to recipient
29:     end for
30: end procedure
```

current vertex adds its own vertex ID to the path (Line 20), so at the end all vertices along the path are in the list.

For example, a message {[1, 9, 8, 7]: [35.0, 42.0]} from vertex 7 to vertex 6 represents a request to find the two shortest paths to source 1. If the receiving vertex has two equal predecessors (e.g. ID 5) in the first two entries of its routing table to 1, it forwards the entire message after appending its own ID 6. If it has two different entries, the message is split into two separate messages. The shortest one {[1, 9, 8, 7, 6]: [35.0]} is sent to the vertex in the first entry and the longer one {[1, 9, 8, 7, 6]: [42.0]} to the vertex in the second entry.

	Communication (Bytes)	Communication (# Messages)	Supersteps	Total time supersteps
Basic, using lists	182204626	402628	18	35.92 s
Basic, without lists[1]	83884151	402628	18	27.134 s
Two-stage (5 dest)	83926097	402749	28 (18 + 10)	25.151 s
Two-stage (1000 dest)	86410529	408203	35 (18 + 17)	33.046 s

TABLE 6.1: Results for two-stage algorithm

	Communication (Bytes)	Communication (# Messages)	Supersteps	Total time supersteps
Basic (Δ=0.5)	92662001	337032	18	25.605 s
Two-stage (Δ=0.5)	49870410	342607	35 (18 + 17)	33.559 s
Two-stage (Δ=0.25)	37684986	339171	36 (19 + 17)	30.304 s
Two-stage (Δ=0.15)	32766532	395713	42 (25 + 17)	29.971 s
Two-stage (Δ=0.1)	34196429	502921	53 (36 + 17)	30.878 s

TABLE 6.2: Comparison of different Δ-parameter settings

6.1.2 Experiments

In order to find out the overhead of the path reconstruction, the algorithm above was combined with the modified shortest path computation, where each vertex only stores its predecessor. This new two-stage algorithm is then compared to the regular shortest path computation. Afterwards, the Delta-stepping improvement was applied to the two-stage algorithm, to find out if it still leads to a significant reduction in communication. For the results in this section, dataset Rnd1K was used.

The results in Table 6.1 show that the path reconstruction stage does not require very much communication, compared to the rest of the shortest path algorithm. Building the paths afterwards reduces the communication by more than 50%.

In the experiment of Table 6.2, the combination of path reconstruction and Delta-stepping was tested, and 1000 destinations were selected.

Clearly, the large reduction in communication holds even when using the algorithm using Delta-stepping. This was to be expected, since Delta-stepping reduces the amount of paths explored, while storing paths distributively reduces the amount of communication for each path. In this graph, where edge weights are distributed uniformly over [0,10], the optimal parameter setting is around 0.15.

6.1.3 Conclusion

The best algorithm found so far, in terms of communication, is the two-stage algorithm combined with the Delta-stepping approach. The decrease in communication is much larger than the decrease in runtime. It is hard to improve execution time considerably, since reducing communication usually requires additional logic that increases overhead again. A general rule for finding the optimal parameter is not yet known. In general, a relatively high setting is always safe, since Delta=Infinity corresponds to the basic algorithm. In addition, a method for global coordination of the shortest path search should be implemented and tested. However, regardless of the shortest path algorithm chosen, the CollectPathsComputation allows the algorithm to use the minimum amount of space and communication, by storing only one vertex of each path at each vertex.

6.2 Postponing storage of predecessors

In the two-stage algorithm described earlier, computation is divided in two parts: first, each vertex V builds a routing table containing its predecessor in the top N paths from each source to V. Then, the routing table is used to collect the full paths.

A possible optimization is to divide the first phase into two parts again, resulting in a three-phase algorithm. The goal of the first step is then to find the minimum amount of data from which the shortest paths can be reconstructed. Since the predecessors can be inferred from the shortest distances, they do not need to be tracked during the computation.

When the distances of all top N shortest paths are fixed, each vertex can find the predecessor corresponding to each distance by comparing the distances of its neighbours. Example: Vertex V has a list of distances [4, 6, 7] and two incoming edges from A and B. The edge weights are 1 and 2, respectively. Neighbour A has distances [3, 6, 8] and B has distances [4, 7, 9]. and its neighbours A and B have [6, 8, 10]. From this information we can infer that the routing table entry corresponding to [4, 6, 7] is [A, B, A] (3+1=4, 4+2=6, 6+1=7). Note that there may be more than one valid result, in the case that two paths have equal distance.

The reconstruction of predecessors takes two supersteps. In the first superstep, each vertex sends all of its top N distances to all neighbours. For each neighbour, the distances are increased with the edge weight of the edge between them (Line 7). The message is signed (10), so that the recipient knows where it came from, and is able to find the predecessor vertices.

In the second superstep, all vertices receive from their neighbours all top N distances to each source. Note that each vertex already has this information; the only new information is the sender. Each time distances are received for a source of which the vertex does not have any predecessor information, the predecessor table is initialized to null values

(Line 18). The source itself immediately fills the first predecessor value to ID_NONE, as it is a zero-length path to itself. Then, it iterates over the received distances (Line 25), trying to match known distances with received distances (Line 29). When a match is found, the predecessor value of null is replaced by the sender of the message (Line 30). Non-null values are not considered for later matches (Line 27), as their distances have already been matched with a predecessor. After the second superstep, all distances have been matched with their predecessor.

Algorithm 3 Pseudocode for reconstructing references to predecessors

```
 1: procedure COMPUTE(vertex, messages)
 2:     if not initialized then
 3:         for each outgoing edge do
 4:             msg = {}
 5:             for each entry (ID, dList) in vertex.value do
 6:                 for each distance in dList do
 7:                     msg[ID].append(distance + edge.weight)
 8:                 end for
 9:             end for
10:             msg['SENDER'] = vertex.ID
11:             send msg to edge.targetVertexID
12:         end for
13:     else
14:         for each message received do
15:             for each entry (ID, dList) in message do
16:                 if vertex.value[paths to ID] == null then
17:                     for each distance in currentMap[ID] do
18:                         vertex.value[paths to ID].append(null)
19:                     end for
20:                     if entry == vertex.ID then
21:                         vertex.value[paths to ID][0] = ID_NONE
22:                     end if
23:                 end if
24:                 receivedIndex = 0
25:                 for i = 0 to vertex.value[ID].size - 1 do
26:                     if vertex.value[paths to ID][i] != null then
27:                         continue
28:                     end if
29:                     if vertex.value[ID][i] == dList[receivedIndex] then
30:                         vertex.value[paths to ID][i]=message['SENDER']
31:                         receivedIndex += 1
32:                     end if
33:                 end for
34:             end for
35:         end for
36:     end if
37: end procedure
```

6.2.1 Experiments

The performance of the three-phase algorithm is compared to the two-phase version. Both use an optimized MapWritable (referred to as version 1 in the description above).

	Communication (Bytes)	Relative difference
Two-phase algorithm	251816581	
Three-phase algorithm	231826722	-7.9%

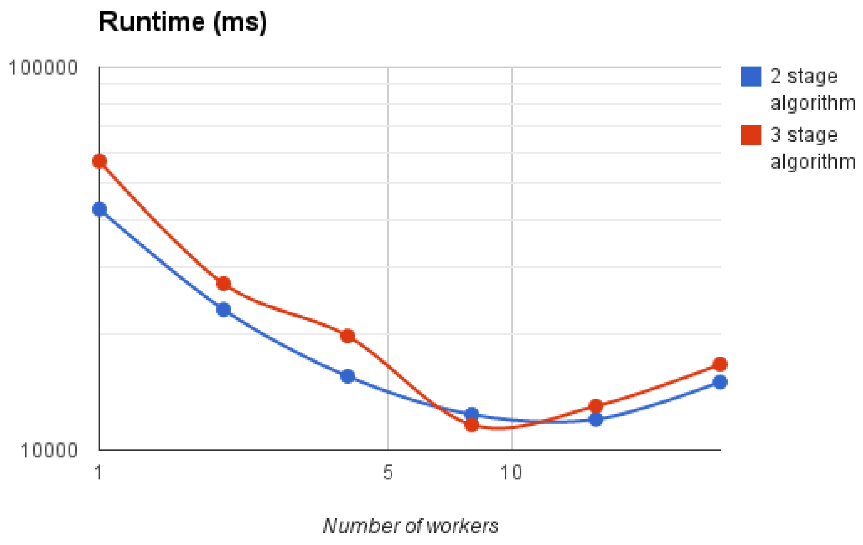


FIGURE 6.1: Evaluation of three-stage algorithm

Workers	1	2	4	8	16	32
2 stage	42552	23279	15591	12385	12032	15066
3 stage	56804	27203	19850	11646	13015	16750

In this case, the small reduction in communication does not lead to better performance. We have less communication, because the sender of the message is only transferred once (during the new superstep). However, the two additional supersteps take more time than the time saved by excluding the sender during the computation.

6.3 Serialisation of messages

In previous experiments, we found that the messages sent between vertices were much larger than expected. In this section, we describe the serialisation method used by the MapWritable class of Apache Hadoop. Next, two alternatives are described and evaluated.

Since a MapWritable instance can have keys and values of any Writable class, during serialisation it needs to track which class is being serialized. Therefore, it produces the following output:

- 4 bytes for the number of entries
- For each entry:
  - 1 byte if the key is of a known class, otherwise 30+ bytes
  - 1 byte if the value is of a known class, otherwise 30+ bytes
  - Serialized version of the key and value

In order to make serialisation efficient, it is not desired to write the class names in each message. Two possible approaches were considered and tested:

- The most simple approach is to extend MapWritable and add all classes used in the algorithm to the set of 'known' classes. In this way, all of them are assigned an one-byte identifier. This is expected to provide a significant reduction in communication, since the names of classes are no longer written to output. Still, typing information is exchanged that may also be fixed in a communication protocol.
- The other approach is to not refer to classes at all. Instead, all keys used in the algorithm could be converted to a long, using a LongMapWritable that only allows one type as key. Certain information on the vertex state, like whether it is a destination vertex, could be stored in a reserved value (such as 0xFFFFFFFE). The class of the corresponding value could be determined from the key. This does have the disadvantage that all keys are now 8 bytes long, including the keys that could be represented by a small string (like "dst": true to indicate that a vertex is a destination vertex). In the same way, we can use a PathMapWritable during the path reconstruction step, which maps paths (lists of vertex IDs) to lists of distances.

Performance and total amount of communication were compared for three versions of the algorithm. The three-phase algorithm was used as reference version, because it had the lowest amount of communication of the implementations so far. Then, the two approaches described above were implemented separately, referred to as "Version 1" and "Version 2" in Table 6.3.

As expected, communication is much more efficient when class names are not included in the serialisation. Removing the last 2 bytes leads to a quite small reduction of 3%.

	Communication (Bytes)	Relative difference
MapWritable	451980176	
Version 1	231826722	-48.7%
Version 2	224784762	-50.3% / -3.0%

Workers	1	2	4	8	16	32
MapWritable	83152	32200	23777	16758	16472	15913
Version 1	56804	27203	19850	11646	13015	16750
Version 2	36535	25297	16733	13711	11366	16412

Table 6.3: Effects of implementation of data structure for maps

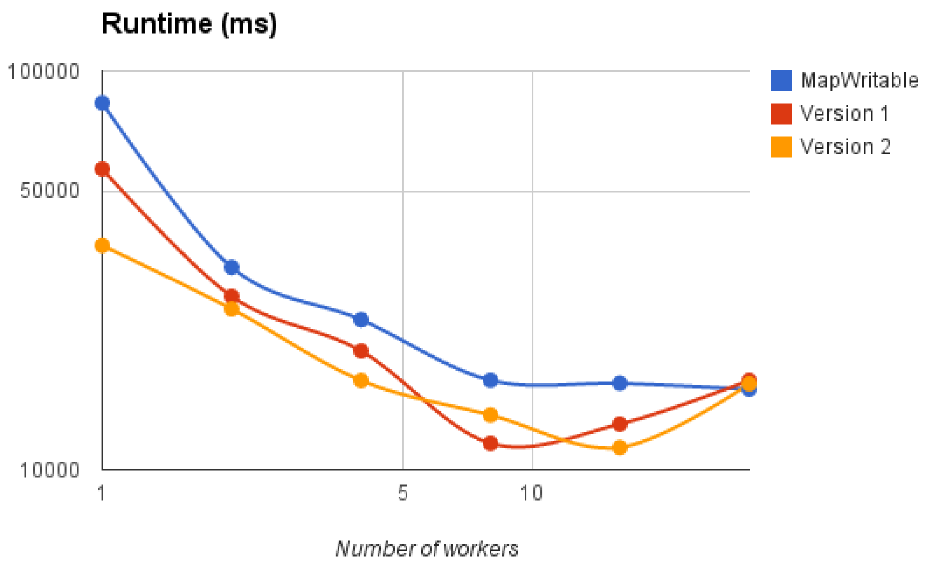


Figure 6.2: Comparison between implementations of MapWritable

More surprisingly, runtime is decreased significantly when the MapWritable does not keep track of the classes used in the keys and values it contains. We can conclude that the second version performs best in both runtime and communication.

6.3.1 A generic alternative for maps

Because of the large reduction in runtime observed in Version 2, it may be preferable to replace all uses of MapWritable with typed variants. The results are confirmed in a different version of the algorithm, as in the table below. However, version 2 described above is not satisfactory, as it requires maintaining a lot of reserved keys for state information, together with the datatype of each state attribute. Therefore, they have been replaced by new classes TypedMapWritable and VertexState. The generic, typed version of MapWritable has fixed key and value types. As a consequence, no typing information needs to be written when serializing, and no set of known classes needs to be maintained. To be able to keep other state information than just distance lists, the VertexState class is extended with a number of properties.

	Two-phase algorithm, normal MapWritable	Two-phase algorithm, refactored
Processing received paths	5.625 s	3.042 s
- with early skipping *	4.278 s	2.713 s
Building new paths	22.655 s	2.901 s
- of which initializing maps	19.905 s	0.083 s
Sending the new paths	1.761 s	2.714 s

* This improvement is not described in a separate section. By immediately ignoring the received (top N) list when the first received item is worse than the Nth known item, the merge operation was made slightly faster.

In conclusion, the refactored variant of version 2 should be used, as both runtime and communication are lower than in the other versions.

6.4 Finding individual paths

In many applications, it is desired to find individual shortest paths between two vertices, or sets of independent paths that do not have overlap in their (source, destination) pairs. Each source vertex is relevant for just one or a few destinations, and vice versa. Therefore, the exploration process started at some source vertex should be terminated when all requested paths have been found. For example, when requesting paths `A->B` and `A->C`, no vertices beyond distance $max($`|A->B|`,`|A->C|`$)$ need to take part in the computation. It may happen that some vertex D at higher distance is active anyway, being reached earlier than B or C, as a total ordering is not feasible in an efficient parallel search. However, when some intermediate top N distances are known for both B and C, the highest distance out of them can be used as an upper bound for the search from A.

In the Pregel model, this can be achieved using aggregators. We implement this as a new MapMaxAggregator, containing a map from vertex IDs to floating point numbers. Aggregation starts with an empty map. When two maps are combined, their entries are merged. If a vertex ID occurred as a key in both maps, and was mapped to two different numbers, then the vertex ID is mapped to the higher number of those two. This process is illustrated in Figure 6.3: the maps (in white) are aggregated by vertices (in blue) and merged during the process of aggregation into the final map.

Each vertex V keeps track of the queried pairs (U,V) for which it is the destination. At the end of each superstep, each vertex builds a map that specifies the Nth distance (U,V) known so far for each source U. If less than N paths have been found yet, Infinity is used. This map is sent to the aggregator to indicate what paths are considered interesting to explore during the next superstep.

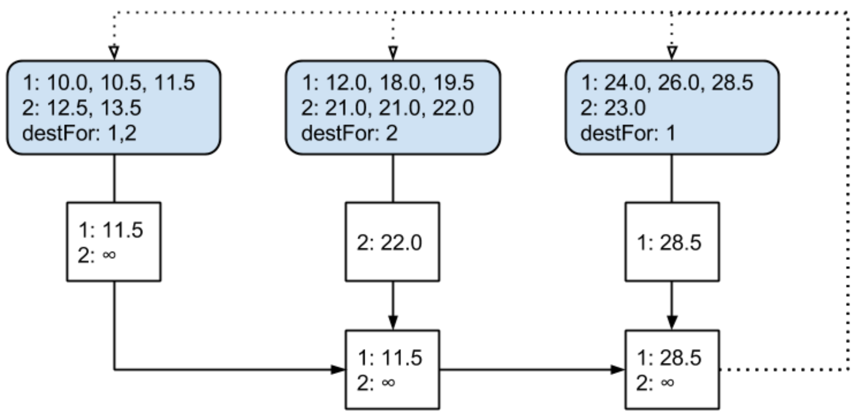


Figure 6.3: Example illustrating aggregation for termination

6.5 Message combiners

Another feature in the Pregel model is the concept of *combiners*. A combiner can be used if not all information sent to a vertex needs to be individually processed by that vertex, such as calculating the sum, minimum or maximum value. It is noted that they can only be used for commutative and associative operations, since the order and grouping of messages is unspecified. Messages at the same worker with the same recipient are combined into one, before they are sent to the recipient. In the single-source shortest path example, communication was found to be reduced by a factor of four [3].

In the multi-source version with top N paths, the same kind of combiner could be used. If we want to keep track of the paths, instead of just finding distances, it will be necessary to first duplicate some information. Without using combiners, a message can be 'signed' by the sender: by including its vertex ID once in a message, the recipient knows the predecessor for all distances within the message. This information about predecessors is eventually used to reconstruct the paths. With combiners, some messages are combined into one before reaching the recipient, removing the possibility for signed messages. Instead, each distance needs to be signed individually. A combined distance list could then contain distances belonging to different predecessors.

This is implemented in the class TopNBestPathsMessageCombiner, which combines maps from different messages into one. In case of two top N lists for the same source vertex and recipient, the two sorted lists are merged into one sorted list. The merge

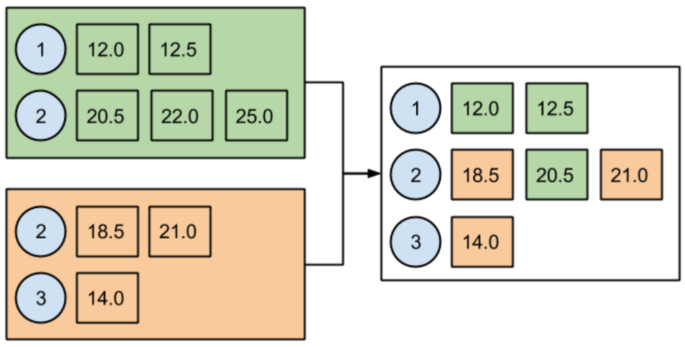


FIGURE 6.4: Example illustrating the use of a message combiner

	Not combined, messages signed	Not combined, distances signed	Combined, distances signed
8 workers	8.805 s	9.977 s	6.272 s
16 workers	7.861 s	8.599 s	6.617 s
32 workers	8.240 s	9.083 s	8.183 s
Bytes sent	843835646	1709797982	1595829065
Paths received	111196335	111196335	5280531

TABLE 6.4: Effect of using message combiners

operation makes a linear-time pass over both lists and combines them into a new list, keeping at most N items.

It is not clear if using a combiner will provide a net benefit in performance. On the one hand, the amount of communication will reduce, especially towards vertices with a large indegree. On the other hand, the original amount of data is now twice as large, for the reasons described. In order to find out which effect has the larger impact, we compare an implementation with combiners and one without. All results are averaged over three runs.

The results in Table 6.4 show that combiners in most cases improve the performance of the algorithm, even when the amount of data needs to be doubled in order to use them. However, when the ratio between vertices' indegree and the number of workers becomes low, there is little to combine within a worker. This causes the non-combined version to be more scalable than the combined version.

Chapter 7

Multiple-source algorithms

In the algorithms presented in Chapter 5, that were optimized in Chapter 6, the amount of work scales linearly with the number of sources. As it is likely that most pairs in the query will have different sources, it is desired that searches which started at different sources could in some way re-use each other's partial results. In this chapter, we explore a number of possibilities to identify shared work and improve performance in this way.

7.1 Pruning using landmarks

In some applications, it is enough to quickly find an estimate of the shortest path between two vertices, without calculating the exact distance. If the estimate is an upper bound, it can be also used in pruning the search for the exact shortest path. One method to find such upper bounds is to define a set of landmarks as a subset of the vertices in the graph. Instead of searching the shortest path for each pair, only paths from or to a landmark are searched. An upper bound for an arbitrary pair (A, B) can then be found by iterating over the landmarks L and finding the path `A-->L-->B` of smallest distance.

A recent all-pairs shortest paths (APSP) algorithm using landmarks was presented by Akiba et al [24]. We have explored the possibilities to adapt this algorithm for use in the BSP model of Giraph, while using a combination of extensions for top N results, directed edges, weighted paths and path queries (instead of only distances). The authors note that the algorithm would perform especially well in a BSP setting, since it is based on a breadth-first search where pruning is done locally. However, the extensions cause a number of complications, described below.

In the extension for weighted edges, it is suggested to replace breadth-first search by sequential applications of Dijkstra's algorithm. This introduces two problems in the parallel version of the algorithm: first, there is no way to parallelise Dijkstra without introducing additional work; and second, the pruning of each search should depend on the results of previous searches. In a parallel setting, we cannot afford to run N instances of Dijkstra sequentially, where N is the number of vertices. Instead, we can select a number of vertices to be landmarks, and run a shortest path algorithm for each landmark in parallel. In a second phase, we can prune all following searches based on the results found for the landmarks. Instead of using Dijkstra, we can use the Floyd-Warshall algorithm, which fits very well on the BSP model.

As an extension, it would even be possible to use multiple levels of landmarks. The minimum amount of work would be to calculate one vertex at a time, and use pruning based on all previous vertices, as in the original algorithm by [24]. While this is not feasible in a parallel setting, we could use levels of increasing size: first 4 landmarks, then 16, then 64, etc.

Another challenge in parallelising the algorithm is to efficiently prune the search, based on the distances found via landmarks. In the original algorithm, this is done using a "Query" function [24], returning the best path from all paths through landmarks. In the BSP setting using weighted paths, this means each (source, destination) pair needs to exchange their distances to each landmark. In order to do this efficiently, we use aggregation to send the best distances from each source to each landmark. Then, each destination can access the aggregated result, removing the need to send a copy of the same information to each destination vertex.

Additionally, pruning is more difficult when using top N results. Since paths are stored distributively, it is not possible to determine the top N paths from the top N paths to the landmarks. For example, if the top 3 paths `A->V` have lengths [10, 11, 12] and the top 3 paths `V->B` have lengths [1, 2, 3], the top 3 paths `A->V->B` could be [10+1, 11+1, 10+2]. However, a list constructed in this way may also contain duplicate instances of the same path, if a path passes through multiple landmarks or multiple times through the same landmark. Therefore, we use a more relaxed form of pruning, using three rules:

- If N paths `A->V` have length $\leq X$ and the best path `V->B` has length Y, then at least N paths `A->B` have length $\leq X + Y$.
- If the best path `A->V` has length X and N paths `V->B` have length $\leq Y$, then at least N paths `A->B` have length $\leq X + Y$.
- If there are N different distances $d \leq D$ where, for some V, `A->V->B` has distance d; then at least N paths `A->B` have length $\leq D$.

Finally, the vertices of the graph should be ordered by degree, since this has been shown to improve performance by an order of magnitude [24]. To order vertices distributively, we use an aggregator that keeps track of the top X vertices with highest degree. By breaking ties on vertex ID, we can establish a total ordering on vertices, so that exactly X vertices are within the top X. In this way, the vertices with most (outgoing) edges are assigned to be a landmark, as the best paths are most likely to pass through them.

7.1.1 Algorithm in Giraph

The resulting parallel algorithm consists of the following phases:

1. Initialization of sources, destinations, excluded paths; and aggregating vertex degrees
2. Running extended Floyd-Warshall for paths from landmarks
3. Reversing all edges in the graph
4. Running extended Floyd-Warshall for paths to landmarks
5. Reversing the edges back to normal
6. Exchanging paths from/to landmarks and calculating all upper bounds for pruning
7. Running extended Floyd-Warshall for all sources, pruning where possible
8. Reconstructing distributed paths from predecessors

The code for step 6 is shown in Algorithm 4. Step 6 takes two supersteps: aggregating relevant landmark information, and using the aggregated data to find pruning information. In the first superstep, all sources that are not landmarks broadcast the distances from themselves *to* each of the landmarks, via an aggregator (Line 3 to 8). In the second superstep, each vertex receives the distance of each source to each landmark, and combines this information with their own distances *from* each of the landmarks (Line 16). Using this information, the vertex iterates over all landmarks and checks the three rules above for each landmark, respectively on lines 18, 21 and 31. Each time a rule is evaluated, the upper bound on the distance from the source to the current vertex is updated.

Algorithm 4 Pseudocode to find upper bounds for pruning using landmarks

```
1: if first step then
2:     if not(vertex.isLandmark) and vertex.isSource then
3:         msg = {}
4:         for (landmark $\rightarrow$ distances_to) in vertex.value do
5:             msg.put(landmark, distances_to)
6:         end for
7:         containerMap = {vertex.id $\rightarrow$ msg}
8:         aggregate(REVERSE_PATHS, containerMap)
9:     end if
10: else
11:     reversedPaths = getAggregatedValue(REVERSE_PATHS)
12:     for (src $\rightarrow$ messageContent) in reversedPaths do
13:         results = []
14:         upperBound = Float.MAX_VALUE
15:         for (landmark $\rightarrow$ distsFirstPart) in messageContent do
16:             distsSecondPart = vertex.value[distances from landmark]
17:             if distsFirstPart.size == TOPN then
18:                 upperBound = min(upperBound, distsFirstPart[TOPN - 1] + distsSecondPart[0])
19:             end if
20:             if distsSecondPart.size == TOPN then
21:                 upperBound = min(upperBound, distsFirstPart[0] + distsSecondPart[TOPN - 1])
22:             end if
23:             for i = 0 to distsFirstPart.size - 1 do
24:                 for j = 0 to distsSecondPart.size - 1 do
25:                     results.add(distsFirstPart[i] + distSecondPart[j])
26:                 end for
27:             end for
28:             results.sort()
29:             result = find the TOPN$^{th}$ different distance
30:             if there are $\geq$ TOPN different distances then
31:                 upperBound = min(upperBound,result)
32:             end if
33:             if upperBound < Float.MAX_VALUE then
34:                 vertex.value[upperBound for src] = upperBound
35:             end if
36:         end for
37:     end for
38: end if
```

7.1.2 Results

The new algorithm was compared to the two-phase algorithm described earlier. We used the LDBC social network dataset of scale 1, selecting a random subset of 25 sources, and calculating the top 5 shortest path from the sources to each vertex in the graph. When using landmarks, the 2 vertices of highest degree were selected to be landmarks. The path reconstruction phase is excluded from this experiment, as the modifications do not affect this part.

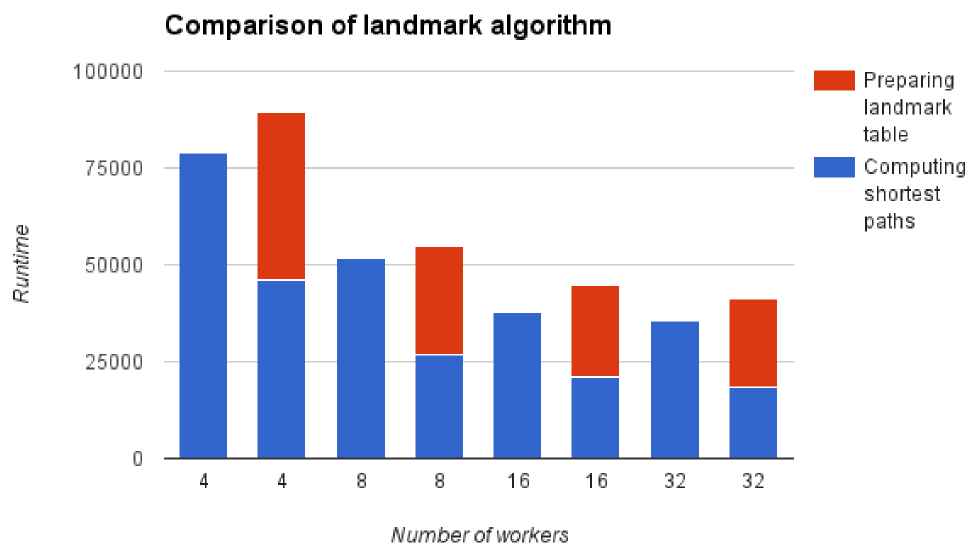


FIGURE 7.1: Evaluation of algorithm using landmarks

From the results, we can see that the actual computation is pruned significantly, reducing the search time by 40 to 50 percent. However, in the current setting, the total computation is not faster at all. In order to use landmarks as an effective optimisation, the number of sources should probably be higher than the 25 vertices selected. In addition, the runtime of constructing the landmark tables seems quite high, considering that this phase is using the same algorithm, only with a much lower number of sources. Runtime could probably be improved by calculating the two phases 'Finding paths to landmarks' and 'Finding paths from landmarks' in parallel.

Other directions for possible optimisations we wanted to look into are the following:

- Contacting landmarks using only local edges
- Using community detection to determine landmarks
- Analysing the path between landmarks, to see if the source or destination itself is on such a path

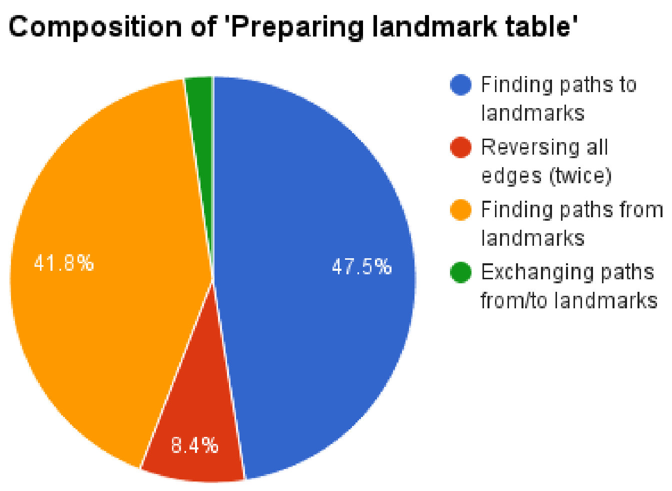


FIGURE 7.2: Composition of runtime during landmark-based algorithm

7.1.3 Optimal number of landmarks

When using the original landmark algorithm, dataset SF1-eucl+pos, 10993 destinations, and a random selection of 100 sources, up to 5 landmarks reduce the amount of work done. Selecting 10 landmarks fails at the moment, probably because too much data is communicated within one superstep. When this problem is solved, the optimal number could be explored further.

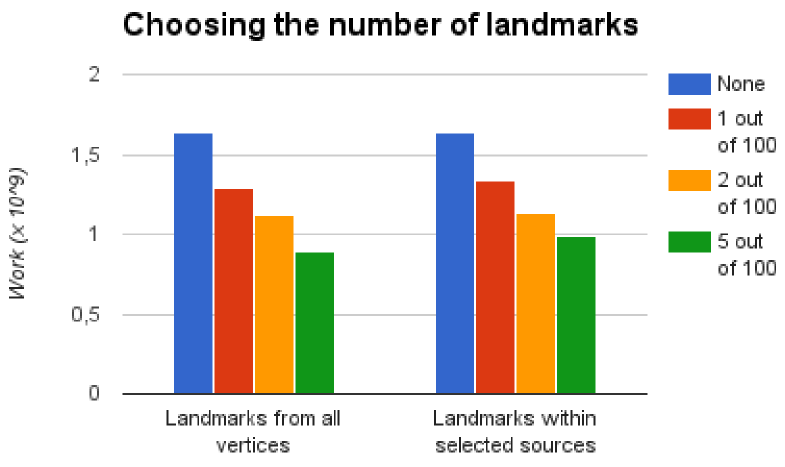


FIGURE 7.3: Different numbers of landmarks

7.2 Simulation of scalability

In previous experiments, we found that the algorithm did not scale very well to a high number of workers, especially on relatively small datasets. A possible explanation is an unequal distribution of work between vertices. Some workers could have more central vertices, or vertices with high degree, and therefore take much more time to finish than others. To find out if this is the case for our dataset, the algorithm was modified to keep a counter at each vertex. Each time a new path was received and compared to the known paths, the counter is incremented by 1. Afterwards, the counter values could be used to analyse the distribution of work. Using the LDBC friendship graph of scale 1, the work was distributed as in the following graph.

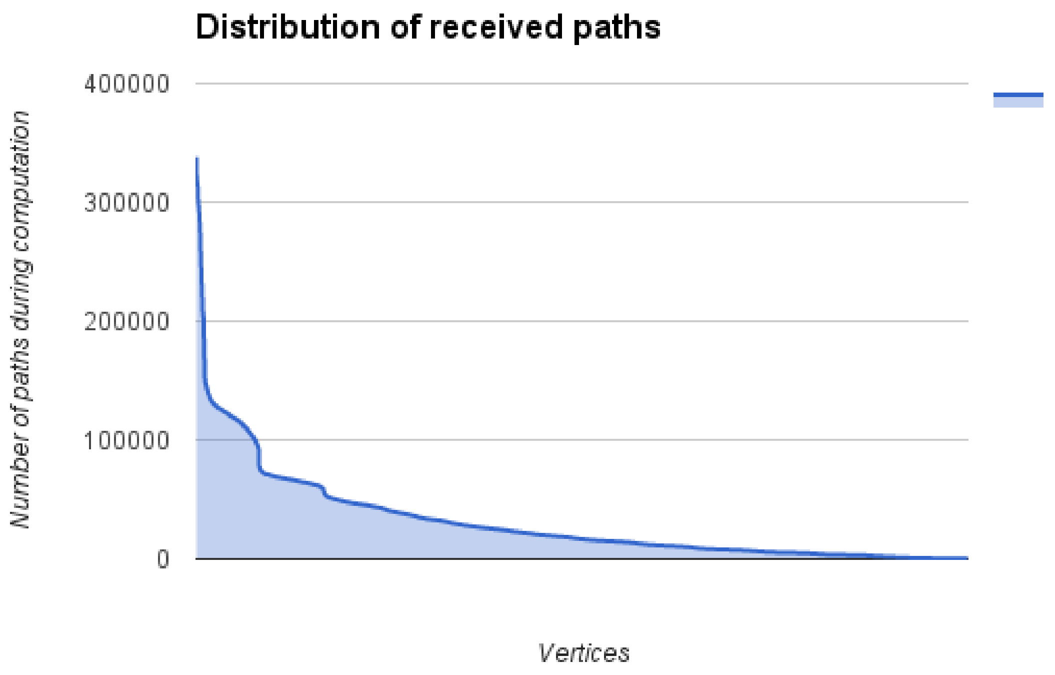


Figure 7.4: Distribution of work over vertices

This distribution was then used to simulate the division of work during the computation. In each simulation, vertices were randomly divided over workers, while keeping the amount of vertices equal for each worker. Then, the speedup was calculated as the total work divided by the highest amount of work assigned to one worker. This simulation was repeated 30 times for different numbers of workers, with the following results:

The results show that a small reduction in speed-up is expected, but smaller than the observed reduction in previous experiments. However, this simulation does not yet account for different supersteps. It may be the case that the distribution of work within each individual superstep is much more unequal then the total distribution of work.

Workers	1	2	4	8	16	32	64	128	256
Lowest	1.00	1.95	3.79	7.24	14.03	26.48	47.74	80.53	131.86
Average	1.00	1.98	3.89	7.59	14.71	28.06	51.80	90.00	155.81
Highest	1.00	2.00	3.96	7.83	15.23	29.34	55.90	99.33	171.01

TABLE 7.1: Upper bounds on scalability

For more accurate results, each vertex should keep track of its work separately at each superstep.

7.2.1 Improvement in simulation

Previously, an upper bound on scalability was estimated by finding the distribution of work across all vertices. This distribution was found to be quite unequal: the top 1% of vertices was receiving 9,3 times more data to process than the rest of the vertices; while the top 10% was receiving 6,2 times more data. Using this distribution, the achievable speedup was calculated by partitioning the vertices randomly over a number of workers. In order to provide a more realistic simulation, we repeated the experiment using more detailed statistics. For edge weights, this time the geometric distance is used between the places people are located in. First, the algorithm using landmarks was run using 2 landmarks, 25 sources and all 10993 destinations. During the algorithm, each vertex counted the number of distances received from its neighbours. One unit of work is then defined as the following: (1) receiving a distance D from vertex V regarding path source S, meaning that some path of distance D exists from S, through V, to the receiving vertex; (2) checking if the path is better than the Nth known path from S; (3) if so, storing the path and removing the old Nth path; (4) sending a new distance to each outgoing edge if an improvement was found.

Using the statistics collected in this way, the process of assigning vertices to workers was simulated again. For each number of workers, this simulation was run 30 times. The estimated speedup during a computation is calculated as follows:

- Time of worker W at superstep N = sum of (V,N) for all vertices V in W
- Total time at superstep N = maximum of (W, N) for all workers W
- Total sequential time at superstep N = sum of (V,N) for all vertices V
- Total time = sum of (Total time at superstep N) for all supersteps N
- Total sequential time = sum of (Total seq. time at superstep N) for all supersteps N

Workers	1	2	4	8	16	32	64	128	256
Lowest	1.00	1.94	3.82	7.34	13.98	26.51	46.96	85.53	142.95
Average	1.00	1.98	3.91	7.65	14.81	28.07	51.99	92.79	158.46
Highest	1.00	2.00	3.98	7.84	15.33	29.43	54.73	101.38	178.63

TABLE 7.2: Improved upper bounds on scalability

- Estimated speedup: (Total time) / (Total sequential time)

In the following graph, we compare the number of workers, the two simulations and the observed runtime.

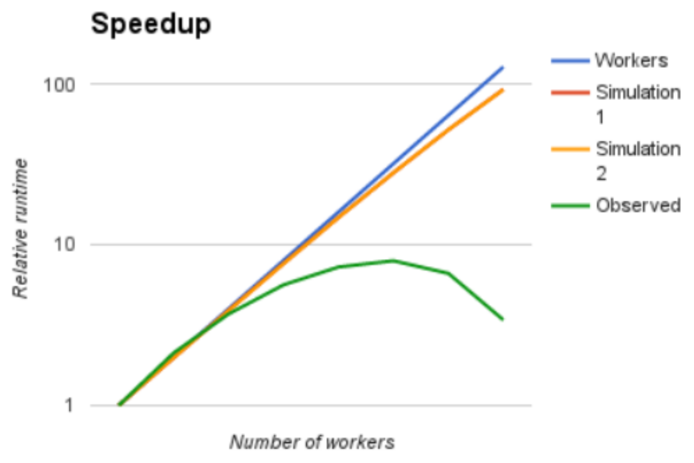


FIGURE 7.5: Comparison of actual and simulated speed-up

A first observation is that the results of the two simulations are very close to each other. This small difference implies that the distribution within a superstep is proportionate to the total distribution of work: if a vertex is receiving lots of data at one superstep, this will be the case in most other supersteps as well. In other words: across supersteps, the same vertices are receiving relatively much data. Both simulations, however, do not explain the observed speedup. Therefore, it does not seem that the distribution of vertices over workers is the main cause of the reduction in runtime. It is possible that the current definition of a 'unit of work' does not capture the actual runtime well. Some received paths need to be stored and broadcasted, while others can be ignored after checking they are longer than the top N paths known. An alternative definition would account differently for received improvements (that are to be stored and forwarded) and received non-improvements (that are to be ignored). Another possible cause is the amount of communication between workers.

Finally, we observe that runtime within a superstep does not have a linear relation to the amount of work processed within that superstep (either total or at the busiest worker). For two runs of the basic Floyd-Warshall algorithm, using top 5 paths and 2 sources, we measured the runtime per superstep and the measured amount of work. Each marker in the following graph corresponds to one superstep.

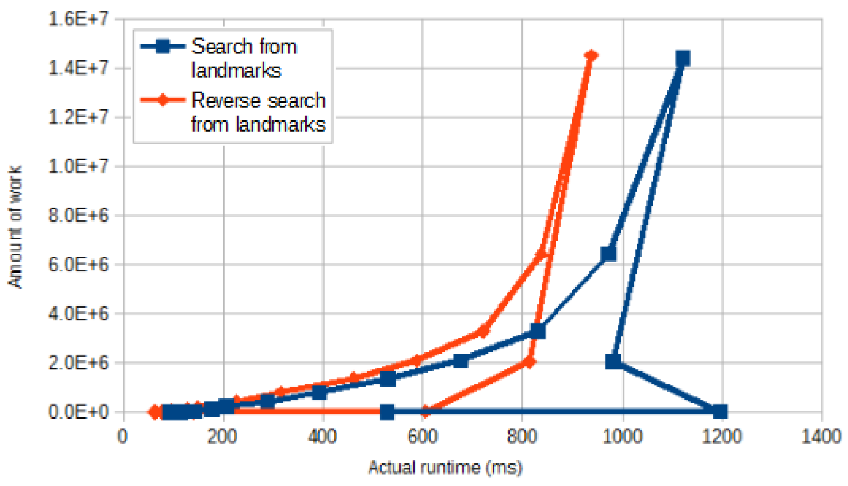


FIGURE 7.6: Comparison of actual runtime and observed work

Especially the first supersteps take more time than expected based on the number of paths received. This is probably because the first paths are almost always an improvement to known paths, since none of the best paths have been found yet. At later iterations, a smaller percentage of the received paths are found to be an improvement.

7.3 Alternative landmarks

In the previous landmark algorithm, the runtime of the preparation phase was relatively high. The algorithm first selected a number of landmarks from the graph, after identifying the vertices with highest degree. Then, all shortest paths from the landmarks were calculated in one phase, and all the shortest paths to the landmarks in the next phase. Since the landmarks did not need to match the vertices selected as sources, the results of these phases were only used for pruning the actual run. In some cases, total runtime including the preparation phase was higher than without landmarks. Therefore, we explore a different algorithm that should reduce the amount of additional work.

The algorithm requires a distribution of the sources into batches of increasing size. For comparison to the previous version, we use a first batch of 2 and a second batch of 23. The first batch is calculated normally without additional pruning, in the same way as the two-phase algorithm without landmarks. After a batch is completed, all vertices try to find the upper bounds on the top N distances for each source in the next batch.

Like the original landmark algorithm, the landmark phase takes two supersteps. In the first superstep, each new source S first selects one arbitrary vertex V from its neighbours that are in previous batches (Line 4 to 9). The edge length `S->V` is then made available to all vertices via an aggregator (Line 11). In the second superstep, each vertex D loads its list of distances for the (V,D) pair (Line 21) and uses the `S->V` edge to find the upper bound for (S,D). The Nth distance of (V,D) is added to the (S,V) edge (Line 24) and the result is stored as pruning information, since D is only interested in paths smaller than or equal to |`S->V->D`|.

Algorithm 5 Pseudocode for the alternative landmark algorithm

```
 1: if first step then
 2:     if vertex.batch == [next batch] then
 3:         msg = {}
 4:         for edge in vertex.getEdges() do
 5:             if edge.targetVertex.batch < [next batch] then
 6:                 msg[edge.targetVertex] = edge.distance
 7:                 break
 8:             end if
 9:         end for
10:         if not msg.isEmpty then
11:             aggregate(REVERSE_PATHS, {vertex.id, msg})
12:         end if
13:     end if
14: else
15:     reversedPaths = getAggregatedValue(REVERSE_PATHS)
16:     for (src → messageContent) in reversedPaths do
17:         upperBound = Float.MAX_VALUE
18:         for (src_neighbour → distFirstEdge) in messageContent do
19:             distsSecondPart = vertex.value[distances from src_neighbour]
20:             if distsSecondPart.size == TOPN then
21:                 upperBound = min(upperBound, distsFirstEdge + distsSecondPart[TOPN - 1])
22:             end if
23:             if upperBound < Float.MAX_VALUE then
24:                 vertex.value[upperBound for src] = upperBound
25:             end if
26:         end for
27:     end for
28: end if
```

7.3.1 Experiments

The main differences between the two landmark versions are:

- Selecting the best vertices globally, vs. selecting them from the set of work

- Using landmarks at any distance, vs. landmarks at one-hop distance
- Using all landmarks for each source, vs. selecting one landmark for each source (if within one-hop range)
- Sending the top-N list of distances, vs. sending the (single) edge weight

It is expected that the new version will use less communication when exchanging pruning information. Instead of exchanging large tables of distances to landmarks, only a single edge weight needs to be exchanged for a new source. Performance is expected to be very much dependent on the structure of the graph, and the selection of pairs. Therefore, we tested a number of different queries. In all cases, there are 16 workers, the number of sources is 25, the number of destinations is 10993, and the dataset is the set of geographical distances between friends. For comparison, both algorithms first calculate information for 2 sources. The distribution over batches is therefore 2/25 for the first algorithm and 2/23 for the second algorithm (since the new version does not separate landmarks from the set of work).

The four sets of sources are as follows:

1. Random selection of 25 sources
2. Selection of 25 sources located in one city
3. Random selection of 1 person, together with 24 friends located within 1 distance (star-like structure)
4. Fully connected subgraph of 25 sources, added to the original dataset. Each edge within the subgraph is set to 0.1 distance. This is especially relevant for social networks, which have highly connected communities.

For each of the sets, we evaluate the two-phase algorithm and the two landmark algorithms in terms of 'units of work' (Figure 7.7) and runtime (Figure 7.8). In the figure that shows the runtime, runtime is split into the main calculation (finding all the requested shortest paths using on pruning information) and the preparation and path reconstruction phases. Preparation includes finding the distances to and from the landmarks, and calculating the upper bounds for pruning. Increasing the number of sources, while keeping the number of landmarks constant, should therefore mainly increase the runtime of the main calculation.

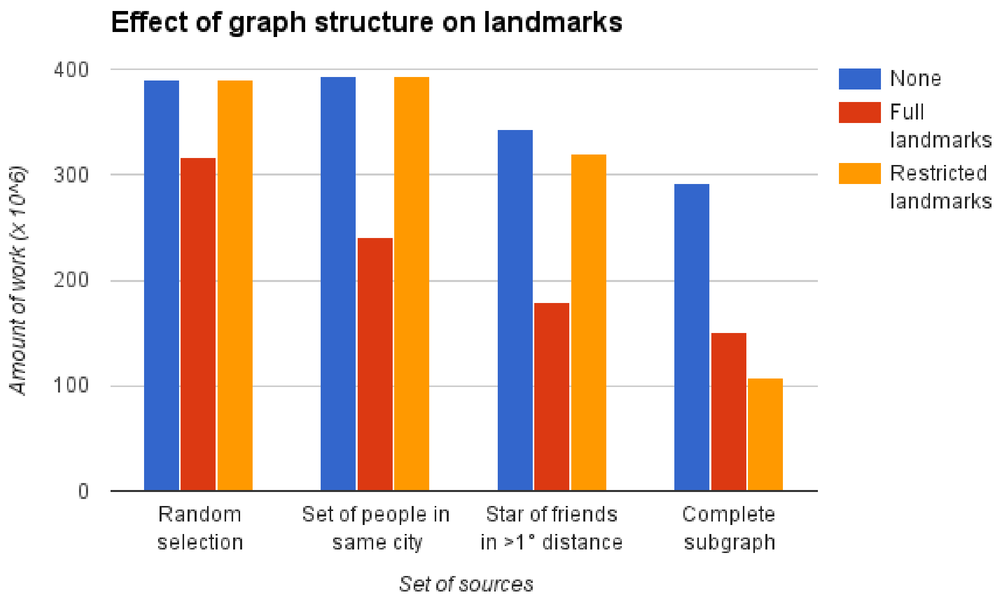


Figure 7.7: Performance of landmark-based algorithm in different graph structures

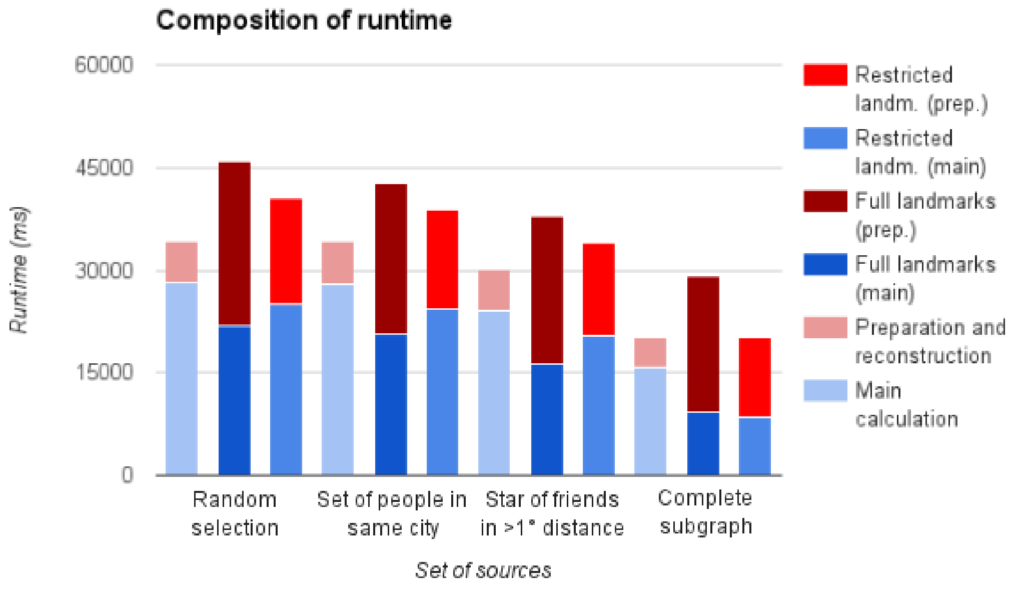


Figure 7.8: Composition of runtime for different landmark-based algorithms

7.3.2 Discussion

From the results above, it seems that the first three sets of sources are not very well connected. In the complete subgraph, the new version of the landmark algorithm performs quite well, in terms of work reduction. When many sources are adjacent to each other, the new version is able to reduce the work by up to 63%. In other cases, the original landmark algorithm performs better, reducing the work by at least 18%. This is much lower than in the experiment using different edge weights. When using edge weights based on the time persons were connected, a reduction of 40-50% was found. In the previous setting, high-degree vertices were more likely to be on a path between two arbitrary vertices. Another interesting point is that the new landmark algorithm still has higher total runtime than the version without landmarks. This is not caused by the new phase, where pruning information is exchanged, since this phase takes less than 1 s. Instead, the first batch of 2 sources is relatively slow, compared to the other batch of 23 sources.

To find the relation between the number of sources and runtime, some experiments with the original (two-phase) algorithm were repeated in the new setting (geographic distances in the LDBC social network). All values are excluding path reconstruction. Speedup is clearly higher when more sources are selected, and smaller batches found to be are relatively slow.

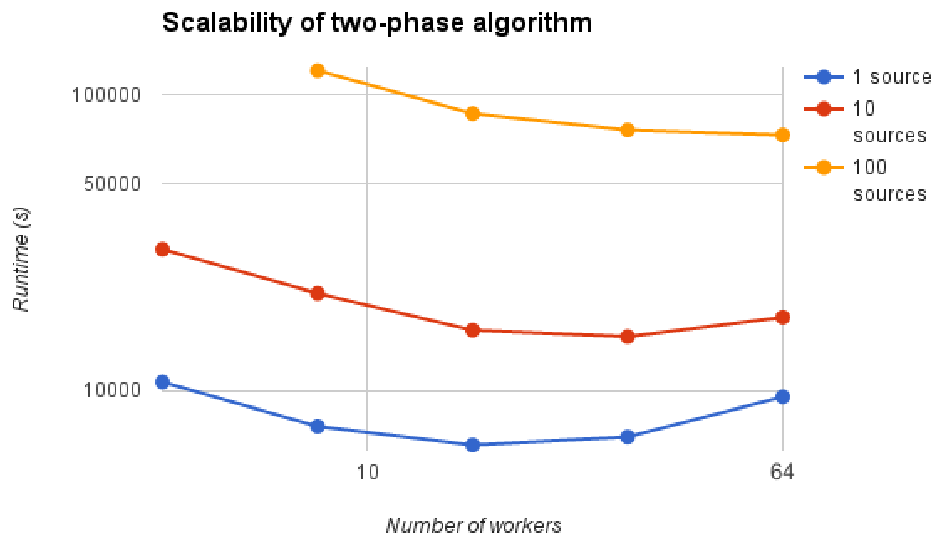


Figure 7.9: Scalability of two-phase algorithm

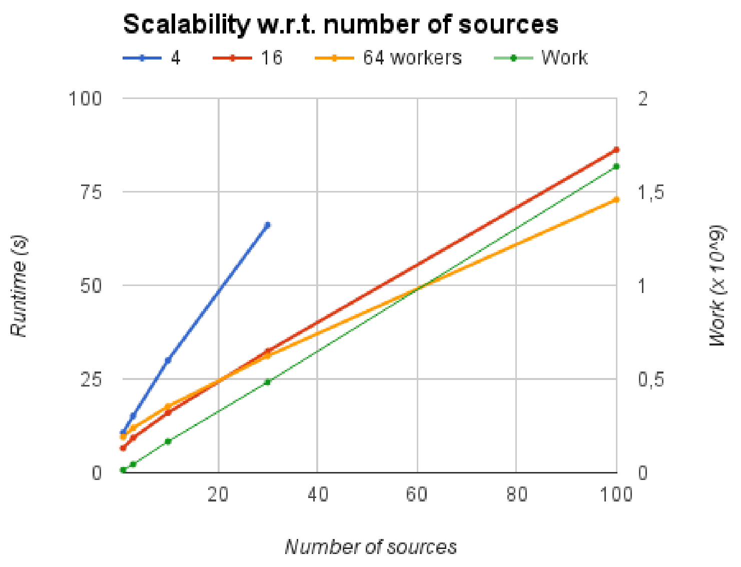


Figure 7.10: Scalability for different number of sources

7.4 Reasoning about nearby pairs

In an undirected graph, the top N shortest paths for a given pair (A, B) can in some cases be used to find shortest paths for a related pair (A', B). Depending on the graph and the query, it is possible that the full top N results for all pairs (A', x) can be found in this way. Then, there is no need for a separate search starting at source vertex A', reducing the runtime of the computation.

The reasoning used to find results for neighbours is as follows. Assume the top N+K is known for the pair (A, B), and there is an edge between A and A' with weight e.

For each top N path (A', B), we can make a path (A, A', B) by adding distance e. Since we know all paths (A, B) smaller than $\mathit{Dist}(A, B, N + K)$ (by assumption), we can construct all paths (A', B) smaller than $\mathit{Dist}(A, B, N + K) - e$. If there was a smaller path (A', B) for which the corresponding path (A, A', B) was not in the top N+K of (A, B), this would contradict the definition of top N shortest paths. In this way, we find a set of shortest paths for (A, B).

If cycles are acceptable in the definition of paths, this is all we need to do. Otherwise, two cases need to be distinguished: paths (A, B) that do not pass through A', and paths (A, A', B) that do. The former can be extended to (A', A, B) by adding e to the distance; the latter are shortened to (A', B) by subtracting e. Experiments show that the described method of reasoning is more useful in this cycle-free definition of paths. In the original definition of paths, the top N+K distances are very close to each other, and it is not possible to find the full top N for the related pairs. Therefore, some methods to use this new definition of paths are explored.

7.5 Exploration of methods to avoid cycles

In order to find more interesting results, it may be desired to return only paths that do not contain a cycle. An easy solution would be to check paths afterwards, remove all paths containing cycles, and compute additional paths when necessary. However, this is not considered a realistic solution, as there may be many cycles. For example, there could be two cycle-free paths `A->B` of lengths 10 and 20, and a cycle `B->B` of length 0.001. Then the second path of length 20 would only be found after removing the paths of length 10+0.001, 10+0.001+0.001, etc. Therefore, this method can not always return results in reasonable runtime (for $N > 1$).

The alternative approach is to avoid traversing cycles in the first place. The most intuitive method is to have all vertices read the intermediate paths they receive during the computation. Upon receipt of a path, each vertex would then check if adding itself to the path would result in a cycle, i.e. if the path already contains the current vertex. Reading the full path is not compatible with the optimization described in section 6.1, since the optimization consisted of only storing predecessors in the paths. Sending the full paths re-introduces a large increase in communication and memory usage: instead of keeping two values for each path (distance and predecessor), we also need to store and exchange each individual vertex in the path.

7.5.1 Bloom filters

To prevent sending the full paths, we can use Bloom filters [25] to estimate if a vertex is contained in a path. Bloom filters can be viewed as a reduced representation of sets, where enough information is kept to prove for most items they are *not* in the set, without any false negatives. The reduction in information stored comes at the cost of introducing false positives: items may incorrectly 'seem to be' in the set. In other words, the filter distinguishes between items that are certainly not in the set, and items for which it is unknown.

This is implemented using 256-bit filters, having the same size as a full path of length 4 (64 bits per vertex ID). As the average path length was found to be much higher (Section 5.3.3), communication is reduced significantly in this way. Using Java's hashCode method on the vertex ID, a set of four bytes is obtained for each vertex. The first three of them are used as indices for the filter. For example, if a vertex identifier has a hash of 0x35A9B24C, we check and set the bits of 0x35, 0xA9 and 0xB2 in the filter.

In addition, we need a way of backtracking paths to verify if they actually contain a cycle, for paths that are suspected to contain one. Therefore, a different type of message is introduced, called CheckCycleMessage (CCM). It contains the potential path (as a Bloom filter), the vertex ID that is suspected to be in the path, and the source vertex ID the path belongs to. Such a message is first created at the vertex detecting a potential cycle, and send backwards along the path. When a CCM ends up at the vertex that created it, a cycle is detected and the message is ignored. When it ends up at the source vertex, no cycle was present after all, so a special kind of message is sent back to the vertex ID that was suspecting a cycle. Another possible condition is that the path can not be backtracked further, because a part has been replaced by an improved version. In that case, the message is ignored as well, since it is not in the top N.

To backtrack a path back to the source, the predecessors and bloom filters are not enough to reconstruct the path at runtime. The path reconstruction method described in Section 6.1 depends on stability of the routing tables containing the predecessors. During runtime, the lists may change at any superstep, so that the best path from A to B may not match the best path known at C from A through B to C. In this inconsistent state, backtracking would often find an improved path that is different from the original path the Bloom filter was built for. Therefore, we introduced locally unique path identifiers: each vertex assigns a unique identifier to each partial path it stores. For each path it forwards, it includes in the message which local path it is based on. During a backtracking step, the recipient can then use this identifier to refer to the exact path.

7.5.2 Proof of incorrectness

Although the method described in this section may intuitively seem to work, it has been found to be incorrect. The problem is that removing cycles breaks the situation of overlapping sub-problems: originally, each vertex could find its own top N for a source, based on the top N lists for all of its neighbours. Top N lists at each vertex could only get better as the computation runs, decreasing in distance and increasing in usefulness. When introducing a special condition such as cycle-free paths, this is no longer the case. In the new situation, paths may suddenly become useless for one neighbour, while being perfectly useful for another neighbour. This is illustrated with the following counter-example.

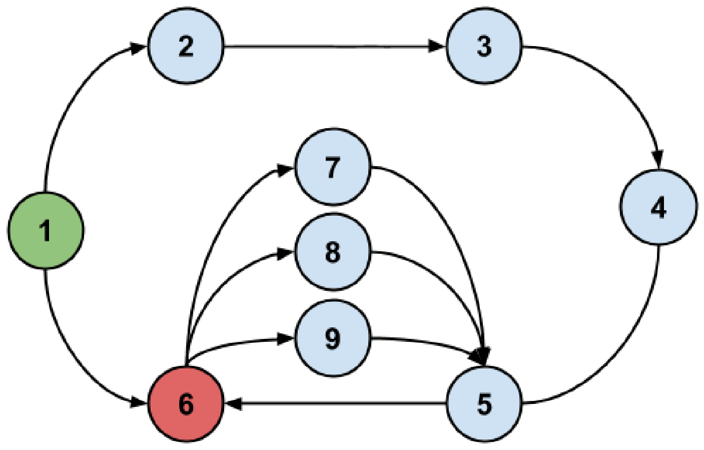


FIGURE 7.11: Counter-example for intuitive cycle-free approach

In this example, the query is about one pair: the top 3 shortest paths from 1 to 6, that do not contain a cycle. For simplicity, all edges are considered to have equal weight. We can observe in the graph that only two cycle-free paths exist, regardless of length: the one-hop path from 1 to 6, and the five-hop path through vertices 2, 3, 4 and 5. The correct result would therefore at least need to contain those two paths.

Applying the algorithm, each vertex starts improving its own list of paths from source vertex 1. Each vertex first tries to build a list of length 3, and then replaces longer paths as better paths are found. Since paths are forwarded over one hop per superstep, it requires at least five supersteps to inform vertex 6 of the long path. However, the path is unfortunately pruned at vertex 5, before vertex 6 is able to learn about its existence.

This happens during execution of superstep 4 at vertex 5: at that moment, vertex 5 already knows three paths from source 1: through vertices 6 and respectively 7, 8 and 9. All three have a length of 3.0. During superstep 4, the vertex received a new message describing the path through 2, 3 and 4, of total length 4.0. Since three better paths are known already, the new path is deemed not interesting and ignored. Vertex 6, being dependent on information received from 1 and 5, only receives paths of length 1.0 and 4.0, of which the latter all contain a cycle. In the end, only the one-hop path is found.

7.5.3 Detecting potential data loss

In the previous section, the method to find the best N acyclic paths has been shown to be incorrect for $N > 1$. The discussion does not apply to the first path in each top N, which can by definition never contain a cycle of non-zero distance. (If it had a cycle of positive total distance, removing the cycle would produce a shorter path, so it could not have been the shortest path. If it contained one of negative total distance, the same holds for repeating the cycle.) Although the paths at position ≥ 2 may not contain the actual best paths, they can still be used as an upper bound.

For some purposes, these results could be useful as an approximation or upper bound. However, they would be more useful if it is possible to detect situations in which a vertex is unable to get enough useful information from neighbours. The dependent results could then be marked as uncertain. Depending on the application, uncertain results could result in a warning or be calculated again using a different, potentially slower algorithm.

One option for this slower algorithm is to increase the number of paths per pair, moving the cut-off limit for intermediate paths from N to $2{*}N$ per pair per vertex, increasing the limit until enough certain results are available. Note that this approach does have some of the drawbacks as the one rejected at the beginning of Section 7.5. Still, this approach is clearly preferable, as many paths with cycles are avoided during the computation, instead of afterwards.

Two methods for detecting invalid results are compared. Both are designed to detect all invalid results, thereby also marking some valid results as potentially invalid. Effectively, the goal is to indicate when not all potential paths were covered during the search. They are inspired by the idea of adding flags to intermediate paths, indicating they are not certain.

The first method is to track removal of paths containing cycles. As each invalid result is caused by the removal of an intermediate path containing a cycle, vertices could inform each other when paths in the top N were removed. By keeping track of the length of the shortest removed path, each vertex can guarantee correctness of paths shorter than this limit. After all, removing a longer path does not lead to a smaller path being lost. This method, however, marks all paths after the first removed path as potentially invalid, and is therefore not very useful.

The second method is slightly more complicated. In this version, each vertex V tracks for each source S its limit of visibility, which represents the highest distance below which all acyclic paths (S, V) are known locally or pending to be explored. Initially, no paths are known and all paths are pending to be explored, so all limits are set to Infinity. If the N^{th} path is found or improved, the limit is decreased to its distance. If a lower limit is received from an incoming edge, the local limit is decreased to that one. At the end of each superstep, each vertex that decreased its limit broadcasts the new value over all outgoing edges, together with the distance lists sent by the algorithm. When the computation is ready, all results are compared to the limit of visibility for that pair. Paths with distance above the limit of visibility are marked as uncertain results.

Both versions are tested on a query of 25 by 25 pairs, for 5 paths per pair. The first method marks 353 results as potentially invalid (56%); the second one marks just 30 out of 625 (5%). This shows that even when many top N paths are removed because of cycles, it is in most cases still possible to find a new valid top N. When N is increased to 10 or 20 for intermediate paths, only 4 and 2 of the 625 top-5's are still uncertain, respectively.

When running the second version on the counter-example above, the system detects that paths may be missing for the paths (1,6), (1,7), (1,8) and (1,9). For the path (1,6), the small path of distance 1 is found, but the long one of distance 5 is not. Based on the limits of visibility at the end of the algorithm, the system is not certain that a path is missing, but does know that a missing path must be of distance 4.0 or higher.

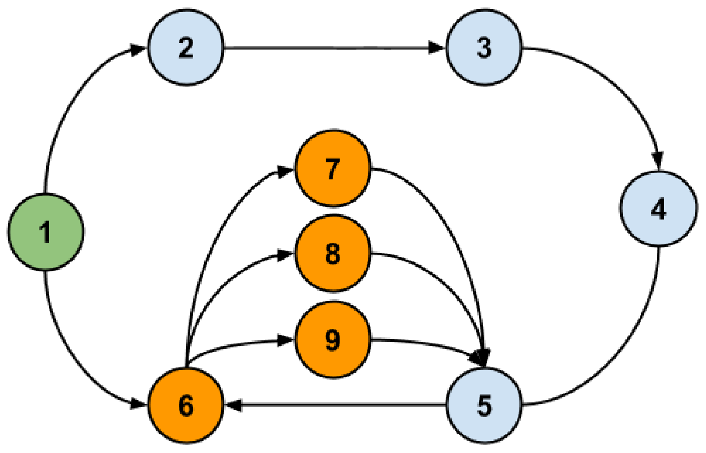


FIGURE 7.12: Counter-example with results of error detection

After running the algorithm, each vertex V knows for each source S a set of top N paths and a limit of visibility. All paths above this limit are not certain to be optimal. Therefore, vertices that do not have enough paths below the limit can contact their neighbours, who then in turn check if they do have enough information to satisfy the original request. Such a message would contain the source S, the (distance) limit up to which paths are desired, and the reverse path from the destination, over which the request has travelled from the original vertex V. For example, vertex V could have a top 3 of [10, 11, 15] and a limit of 12. Then to each potential predecessor P a request would be sent `S->..->V`, with the missing part `..` of length < 15. Vertex P subtracts the length of edge (P, V) (e.g. 2) from the length 15, and checks if its own limit is at least $15 - 2 = 13$. If so, vertex P is able to satisfy the request, otherwise it will send a new request to its own incoming neighbours. This process is expected to end in relatively few steps, as few vertices had uncertain top N values in our test described above.

Chapter 8

Conclusion

Existing query languages have different methods to support shortest paths. However, most do not have a special shortest path operator or function. Two related types of functionality that are provided by many languages are closure operators (for reachability) and recursion mechanisms. The former is often not powerful enough to express the shortest path problem in a way that can be detected and optimized, as discussed in Chapter 3. The latter is very powerful, but often more difficult to write from a user's perspective. Basically, the user then has to write the shortest path algorithm itself in the query language, and hope that it is still executed efficiently.

Cypher does have a built-in feature for shortest paths, but it is very restricted, as features like cheapest paths using edge weights are not supported. Therefore, we proposed an extension to Cypher providing a CHEAPEST SUM construct. A new operator to support this construct was implemented in Lighthouse. Few modifications were needed to the rest of the Lighthouse system, one notable exception being that binding tables can now contain columns with variable-length paths, as well as vertex identifiers and distances.

We also explored a number of possible algorithms for efficiently finding weighted shortest paths in a very large graph. In order to keep the algorithm scalable, and to integrate it with the graph pattern matching engine Lighthouse, we implemented the algorithm in Giraph. The new algorithm enables Lighthouse to match patterns that contains a query for shortest paths. Queries may include a top N parameter to return a number of best alternatives for each pair, may describe both the shortest distances or the shortest paths themselves, and can describe a weight function that is calculated ad-hoc during query execution. Because of this last requirement, many existing algorithms using an index are not suitable for our application.

A number of one-to-all algorithms were developed and compared. First, an algorithm based on the single-source example was implemented, in which each vertex informs all neighbouring vertices as it finds a new or improved path to itself. All other algorithms are based on this principle, and introduce additional features to make the implementation run faster, to improve pruning, or reduce communication between workers. The second one-to-all algorithm uses a parameter Δ to gradually increase the upper limit below which paths are allowed to be traversed. This reduced both runtime and communication. The third algorithm was using aggregation to determine how to increase the upper limit in a flexible way, removing the need to find a good parameter setting. This is very effective for pruning, but more research is needed to find a fast implementation. Currently the overhead in aggregating and postponing messages causes runtime to be slightly higher.

Independently of the method chosen for postponing messages, there are a number of optimizations that are possible in any version. One is to split the algorithm in either two or three phases, and reconstruct the paths at the last phase. This reduces communication in the first phase to either distances with predecessors (for two-phase), or even just the distances (for three-phase). We found that reconstructing paths improves performance and communication. Reconstructing predecessors does reduce communication further, but is not worth the additional runtime incurred.

Another way to reduce communication and improve performance is to avoid using the MapWritable class of Hadoop. By using typed maps when possible and some custom classes where necessary, we can avoid writing typing information in the process of serialisation. Performance is further improved by making use of aggregation to terminate a search after finding a specific destination; and by using message combiners, which are one of the features in the Pregel model.

Finally, we also explored a many-to-many algorithm based on the concept of landmarks. Two different approaches were implemented and tested, with some differences in the way landmarks were selected (from all vertices or only from the query), and differences in how landmarks were selected by vertices. In the first version, all vertices used all landmarks, while in the second version they only use one landmark at one-hop distance. The second version works better in graphs with a high clustering coefficient, while the first version works better in most other graphs, including the LDBC-SNB dataset.

In summary, the best performance is to be expected by combining a number of features:

- Aggregation to explore vertices in order of increasing weight.
- Path reconstruction to avoid sending paths that are found to be sub-optimal.

- Using a message combiner if the average vertex degree is higher than the number of workers.
- Using typed maps and vertex states instead of the general MapWritable.
- Aggregation to prune and terminate the search when paths to the destination are final.
- An implementation of landmarks exchanging less data than our first version, but more than our second version.

For future work, a natural extension would be to implement variable length paths with a minimum and maximum number of hops. It would also be interesting to implement other graph-based algorithms. With regard to the shortest path algorithm, it could be explored further if it is possible to directly find exact distances (instead of upper bounds), using the information of previous batches. The parallel approximation of Dijkstra could also be improved further, to find a useful aggregation metric that does not incur as much overhead as the current one.

Bibliography

[1] Sînziana Maria Filip. A scalable graph pattern matching engine on top of Apache Giraph. Master's thesis, VU University Amsterdam, 2014.

[2] Avery Ching and Christian Kunz. Apache Giraph, 2013.

[3] 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. ACM, 2010.

[4] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. The linked data benchmark council: A graph and RDF industry benchmarking effort. *ACM SIGMOD Record*, 43(1):27–31, 2014.

[5] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[6] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Ozsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[7] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[8] Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[9] Kamesh Madduri, David A Bader, Jonathan W Berry, and Joseph R Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *ALENEX*, volume 7, pages 23–35. SIAM, 2007.

[10] Joseph R Crobak, Jonathan W Berry, Kamesh Madduri, and David A Bader. Advanced shortest paths algorithms on a massively-multithreaded architecture. In

Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–8. IEEE, 2007.

[11] Jesper Larsson Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21(9):1505–1532, 1995.

[12] I-Lin Wang, Ellis L Johnson, and Joel S Sokol. A multiple pairs shortest path algorithm. *Transportation science*, 39(4):465–476, 2005.

[13] Philip N Klein. Multiple-source shortest paths in planar graphs. In *SODA*, volume 5, pages 146–155, 2005.

[14] Sergio Cabello and Erin W Chambers. Multiple source shortest paths in a genus g graph. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 89–97. Society for Industrial and Applied Mathematics, 2007.

[15] Philip M Merlin and Adrian Segall. A failsafe distributed routing protocol. *Communications, IEEE Transactions on*, 27(9):1280–1287, 1979.

[16] Tim Furche, François Bry, Sebastian Schaffert, Renzo Orsini, Ian Horrocks, Michael Krauss, and Oliver Bolzer. Survey over existing query and transformation languages, revision 2.0. deliverable i4-d1. *Institute for Informatics, Ludwig-Maximilians-Universität München*, 2006.

[17] Loredana Afanasiev, Torsten Grust, Maarten Marx, Jan Rittinger, and Jens Teubner. An inflationary fixed point operator in XQuery. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1504–1506. IEEE, 2008.

[18] Sebastian Schaffert and François Bry. Querying the web reconsidered: A practical introduction to xcerpt. In *Extreme Markup Languages®*. Citeseer, 2004.

[19] Steve Harris and Andy Seaborne. Sparql 1.1 query language. *W3C Recommendation*, 21, 2013.

[20] Maurizio Atzori. Computing recursive SPARQL queries. In *Semantic Computing (ICSC), 2014 IEEE International Conference on*, pages 258–259. IEEE, 2014.

[21] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of RDF query languages. In *The Semantic Web–ISWC 2004*, pages 502–517. Springer, 2004.

[22] Andrew Eisenberg and Jim Melton. SQL: 1999, formerly known as SQL3. *ACM Sigmod record*, 28(1):131–138, 1999.

[23] Mark W Goudreau, Kevin Lang, Satish B Rao, Torsten Suel, and Thanasis Tsantilas. Portable and efficient parallel computing using the BSP model. *Computers, IEEE Transactions on*, 48(7):670–689, 1999.

[24] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013.

[25] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.