UNIVERSITÀ DELLA CALABRIA

UNIVERSITA' DELLA CALABRIA

Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica

Dottorato di Ricerca in
Information and Communication Engineering for Pervasive Intelligent Environments

CICLO XXIX

Efficient Incremental Algorithms for Handling Graph Data

Settore Scientifico Disciplinare ING-INF/05

Coordinatore: Prof. Felice Crupi

Tutor: Prof. Sergio Greco

Dottorando: Ximena Quintana

To God. For allowing me to reach this point and for each blessing that He has managed to crystallize each of my goals, in addition to his infinite goodness and love.

To my father Bolivar. For his life because he has been an example of perseverance and sacrifice, he has been the engine that has driven this professional achievement.

To my mother Nancy. For having supported me at all times, for her advice, her values, for her constant motivation, but more than anything, for her immense love.

To my sister Lorena. For all the love shown and to have been a real support in the difficult moments; To my nephews Anthony and Antonella who despite their young age their love has been greater than distance and time separated.

To my grandparents Arturo, Zoila, Julio and Aurorita. that although only one of them is not in the sky the four have been my angels during all this time.

To my teachers Professor Sergio Greco and Cristian Molinaro. For their great support and for promoting the development of this professional training.

Preface

There has been a significant growth of connected data in the last decade. Enterprises that have changed the world like Google, Facebook, and Twitter share the common thread of having connected data at the center of their business. Such data can be naturally modeled as graphs.

In many current applications, graph data are huge and efficiently managing them becomes a crucial issue. Furthermore, one aspect that many current graph applications share is that graphs are *dynamic*, that is, they are frequently updated. In this setting, an interesting problem is the development of incremental algorithms to maintain certain kind of information of interest when the underlying data is changed. In fact, incremental algorithms avoid the recomputation of the information of interest from scratch; rather, they tend to minimize the computational effort to update a solution by trying to identify only those pieces of information that need to be updated. In contrast, non-incremental algorithms need to recompute new solutions from scratch every time the data change, and this can be impractical when data are huge and subject to frequent updates.

In this thesis, two classical graph theory problems are considered: the maximum flow problem and the shortest path/distance problem. For both of them efficient incremental algorithms are proposed.

The maximum flow is a classical optimization problem with a wide range of applications. Nowadays, it is successfully applied in social network analysis for link spam detection, web communities identification, and others. In such applications, flow networks are used to model connections among web pages, online voting systems, web communities, P2P and other distributed systems. Thus, networks are highly dynamic.

While many efficient algorithms for the maximum flow problem have been proposed over the years, they are designed to work with *static* networks, and thus they need to recompute a new solution from scratch every time an update occurs. Such approaches are impractical in scenarios like the aforementioned ones, where updates are frequent.

To overcome these limitations, we propose efficient *incremental* algorithms for maintaining the maximum flow in dynamic networks. Our approach identifies and acts only on the affected portions of the network, reducing the computational effort to update the maximum flow. We evaluate our approach on different families of datasets, comparing it against state-of-the-art algorithms, showing that our technique is significantly faster and can efficiently handle networks with millions of vertices and tens of millions of edges.

The second problem considered in this thesis is computing shortest paths and distances, which has a wide range of applications in social network analysis [46], road networks [63], graph pattern matching [22], biological networks [55], and many others. It is both an important task in its own right (e.g., if we want to know how close people are in a social network) and a fundamental subroutine for many advanced tasks.

For instance, in social network analysis, many significant network metrics (e.g., eccentricity, diameter, radius, and girth) and centrality measures (e.g., closeness and betweenness centrality) require knowing the shortest paths or distances for all pairs of vertices. There are many other domains where it is needed to compute the shortest paths (or distances) for *all* pairs of vertices, including bioinformatics [52] (where all-pairs shortest distances are used to analyze protein-protein interactions), planning and scheduling [51] (where finding the shortest paths for all pairs of vertices is a central task to solve binary linear constraints on events), and wireless sensor networks [15] (where different topology control algorithms need to compute the shortest paths or distances for all pairs of vertices).

Although many algorithms to solve this problem have been proposed over the years, they are designed to work in the main memory and/or with static graphs, which limits their applicability to many current applications where graphs are highly dynamic.

In this thesis, we present novel efficient incremental algorithms for maintaining all-pairs shortest paths and distances in dynamic graphs. We experimentally evaluate our approach on several real-world datasets, showing that it significantly outperforms current algorithms designed for the same problem.

Main Contributions. As for the maximum flow problem, the main contributions are:

- We propose efficient incremental algorithms to maintain the maximum flow after vertex insertions/deletions and edge insertions/deletions/updates. Our algorithms are designed to effectively identify only the affected parts of the network, in order to reduce the computational effort for determining the new maximum flow.
- We provide complexity analyses.
- We report on an experimental evaluation we conducted on several families of datasets with millions of vertices and tens of millions of edges. Experimental results show that our approach is very efficient and outperforms state-of-the-art algorithms.

As for the shortest path/distance problem, the main contributions are:

- We consider the setting where graphs and shortest paths are stored in relational DBMSs and propose novel algorithms to incrementally maintain all-pairs shortest paths and distances after vertex/edge insertions, deletions, and updates. The proposed approach aims at reducing the time needed to update the shortest paths by identifying only those that need to be updated (it is often the case that small changes affect only few shortest paths). To the best of our knowledge, [50] is the only disk-based approach in the literature for incrementally maintaining all-pairs shortest distances. In particular, like our approach, [50] relies on relational DBMSs.
- We experimentally compare our algorithms against [50] on five real-world datasets, showing that our approach is significantly faster. It is worth noticing that our approach is more general than [50] in that we keep track of both shortest paths and distances, while [50] maintain shortest distances only (thus, there is no information on the actual paths).

Organization. The thesis is organized as follows. In Chapter 1, basic concepts and notations on relational and graph databases are introduced.

In Chapter 2, the incremental maintenance of the maximum flow in dynamic flow networks is addressed.

In Chapter 3, the incremental maintenance of all-pairs shortest paths (and distances) in dynamic graphs is addressed.

Finally, conclusions are drawn.

Rende, *Ximena Quintana*
July 2017

Contents

1

Relational and Graph Databases

A database is a set of data belonging to the same context and stored systematically. *Database Management Systems* (DBMSs) allow us to store and then access the data. Data can be structured according to different data models. This chapter briefly recalls the basics of relational databases and graph databases.

1.1 Relational Databases

The existence of alphabets of *relation symbols* and *attribute symbols* is assumed. The *domain* of an attribute A is denoted by $Dom(A)$. The *database domain* is denoted by Dom. A *relation schema* is of the form $r(A_1, \ldots, A_m)$ where r is a relation symbol and the A_i's are attribute symbols (we denote the previous relation schema also as $r(U)$, where $U = \{A_1, \ldots, A_m\}$). A *relation instance* (or simply *relation*) R over $r(U)$ is a subset of $Dom(A_1) \times \cdots \times Dom(A_m)$. Each element of R is a *tuple*. Given a tuple $t \in R$ and a set $X \subseteq U$ of attributes, we denote by $t[X]$ (resp. $R[X]$) the projection of t (resp. R) on X. Given the relation schemata $r(U), s(V), \ldots$ we will refer to their respective instances as $R, S, \ldots$. A *database schema* DS is a set $\{r_1(U_1), \ldots, r_n(U_n)\}$ of relation schemata. A *database instance* (or simply *database*) DB over DS is a set $\{R_1, \ldots, R_n\}$ where each R_i is a relation over $r_i(U_i)$, $i = 1..n$. The set of constants appearing in DB will be called *active domain* of DB. In the following, we will also refer to the instance of the relation r in a database DB as $DB[r]$.
A conjunctive query Q is of the form $\exists Y\, \Phi(X, Y)$ where Φ is a conjunction of atoms (an atom is of the form $p(t_1, \ldots, t_n)$ where p is a relation symbol and each t_i is a term, that is a constant or a variable), X and Y are sets of variables with X being the set of free variables of Q. The result of applying Q over a database DB is denoted by $Q(DB)$.

Integrity constraints express semantic information about data, i.e. relationships that should hold among data. They are mainly used to validate database transactions.

Given a relation schema $r(U)$, a *functional dependency* fd over $r(U)$ is of the form $X \to Y$, where $X, Y \subseteq U$. If Y is a single attribute, the functional dependency is said to be in *standard form* whereas if $Y \subseteq X$ then fd is *trivial*. A relation R over $r(U)$ *satisfies* fd, denoted as $R \models fd$, if $\forall t_1, t_2 \in R \;\; t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$ (we also say that R is *consistent* w.r.t. fd). A *key dependency* is a functional dependency of the form $X \to U$. Given a set FD of functional dependencies, a *key* of r is a minimal set K of attributes of r s.t. FD entails $K \to U$. Each attribute in K is called *key attribute*. A *primary key* of r is one designated key of r. In the following, we will refer to the functional dependencies in FD over a schema $r(U)$ also as $FD[r]$.
Given two relation schemata $r(U)$ and $s(V)$, a *foreign key constraint* fk is of the form $r(W) \subseteq s(Z)$, where $W \subseteq U, Z \subseteq V, |W| = |Z|$ and Z is a key of s (if Z is the primary key of s we call fk a *primary foreign key constraint*). Two relations R and S over $r(U)$ and $s(V)$ respectively, *satisfy* fk if for each tuple $t_1 \in R$ there is a tuple $t_2 \in S$ such that $t_1[W] = t_2[Z]$ (we also say that R and S are *consistent* w.r.t. fk).

1.2 Graph Databases

Although graph databases are a newer technology, they have had a strong development in recent years and today there is a variety of graph database systems. Graph databases use graph structures with nodes, edges, and properties to represent and store data. General graph databases that can store any graph are different from specialized graph databases such as triple-stores and network databases. There are two properties of graph databases one should consider when investigating graph database technologies:

1. **The underlying storage:** Some graph databases use native graph storage that is optimized and designed for storing and managing graphs. Not all graph database technologies use native graph storage. However, some serialize the graph data into a relational database, an object oriented database, or some other general-purpose data store.
2. **The processing engine:** Some definitions require that a graph database use index-free adjacency, meaning that connected nodes physically point to each other in the database.

A graph database represents information as nodes of a graph and its relations with the edges thereof. An illustrative example is reported in Figure 1.1.

- **Nodes** represent entities and can be tagged with labels representing their different roles in their domain

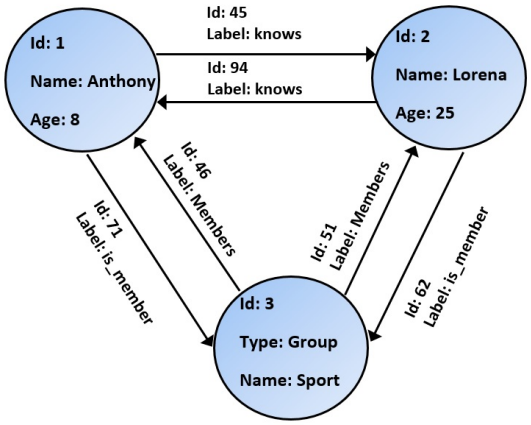


Fig. 1.1: Graph with properties.

- **Edges** express relationships and connect nodes. Significant patterns emerge when we consider the connections of nodes, properties, and edges. Note that even when edges are directed, relationships can always be navigated regardless of direction.
- **Properties** are pertinent information associated with nodes and edges.

A simple way to comprehend graph databases is to imagine social networks. In this scenario we can imagine users as nodes and connections as edges. On a social media platform relationships between users are complex and through the application of this model one can get many kinds of relationships.

In the rest of this chapter, we discuss two popular graph database systems: Neo4j and OrientDB.

1.2.1 Neo4j

Neo4j is a graph database developed by Neo Technology, open source, and written in Java.

Architecture of Neo4j. Neo4j interaction can take place at various levels, depending on the needs of the application. Figure 1.2 shows three levels: the bottom level relative to the data storage disk, the top level that exposes functions to interact with the database, and the intermediate level contains the management system. The transactional management is optimized for graph data model.

Neo4j uses the *Cypher* query language to manipulate data and issue queries. Many APIs are available that allow users to make requests through Cypher. Cypher generates the execution plan, finds start node, traverse through relationships and retrieves the results. The Traversal API are of

particular interest, as they offer functionalities for graph traversal. Traversal happens from node to node via edges (relationships). Core API provides functionalities for initiating embedded graph databases that receive client connections. It also provides capabilities to create nodes, relationships and properties [56].

Neo4j also provides indexes. Neo4j recently introduced the concept of labels and their sidekick, schema indexes. Labels are a way of attaching one or more simple types to nodes (and relationships), while schema indexes allow to automatically index labelled nodes by one or more of their properties. Those indexes are then implicitly used by Cypher as secondary indexes and to infer the starting point(s) of a query. This new indexing system allows for indexing on an attribute of all nodes/relationships with a specific label and automatically maintains the index as creation, deletion, and edit updates are made.

The memory is managed efficiently through two cache, one at the level of file system that keeps the parts of files stored as records (File System cache), and a faster and higher level that keeps portions of the graph (Object Cache). Caches in Neo4j are just part of the system memory used when Neo4j instances are created and queries are being performed on those instances. Transaction Management and Transaction log keep tracks of transactional consistency and atomicity, while their record being maintained in the log. Record Files or Store Files are those where Neo4j stores the graph data. Each store file contains data for specific part of the graph (e.g nodes, relationships, properties, etc.). Some of the store files commonly seen are

- neostore.nodestore.db
- neostore.relationshipstore.db
- neostore.propertystore.db
- neostore.propertystore.db.index
- neostore.propertystore.db.strings
- neostore.propertystore.db.arrays

Neo4j Features. Among the features of Neo4j there are high performance in search operations between nodes [45], the ability to adapt to the field of semantic networks [49], and the expressiveness of the query language (Cypher). For the purpose of fully maintain data integrity and ensure the proper transactional behavior, Neo4j supports the ACID properties.

For users, developers and database administrators, Neo4j provides the following features:

- Materializing of relationships at creation time, resulting in no penalties for complex runtime queries.
- Constant time traversals for relations in the graph both in depth and in breadth due to efficient representation of nodes and relationships.

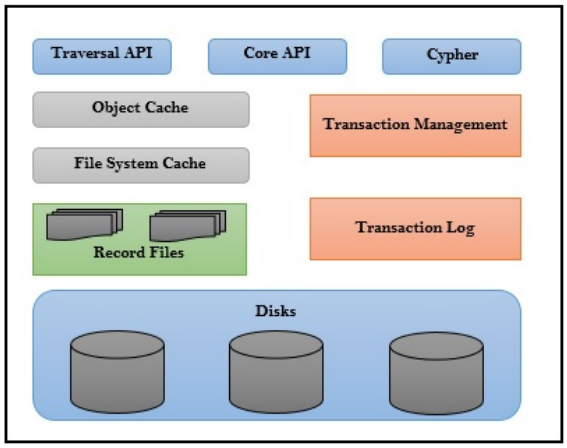


Fig. 1.2: Neo4j Architecture

- All relationships in Neo4j are equally important and fast, making it possible to materialize and use new relationships later on to "shortcut" and speed up the domain data when new needs arise.
- Compact storage and memory caching for graphs.

Neo4j provides a web front end for manual database querying with Cypher and for development and testing purposes, showing nodes and edges (cf. Figure 1.3). The interface presents returned results by interactive graph visualization. On one hand it has a lightweight API that allows it to be included into Java code and run as embedded database. If it shall be used as a standalone database, clients communicate via a RESTful [23] web service using an API similar to its Java API. It supports transactions also provides a user management.

Cypher. Cypher is a declarative query language. It enables one to tell what should be selected, inserted, updated, or deleted from a graph database. It is possible to run Cypher queries via CLI (Neo4j Shell), Java API, or by REST API.

The basic idea of the Cypher query language is to express patterns to be matched over the graph the user wants to query.

Nodes are enclosed by parentheses, relationships are enclosed by brackets. Nodes are connected together by –, $\longrightarrow$, or $\longleftarrow$ chars.

A pattern can also match nodes and relationships with specific labels. The labels are written after the variable name separated with a colon. Nodes and relationships can be matched by their properties. Properties are written into braces as key-value pairs. Multiple properties are separated by a comma.

In the following, we briefly mention the main clauses that can be used in Cypher.

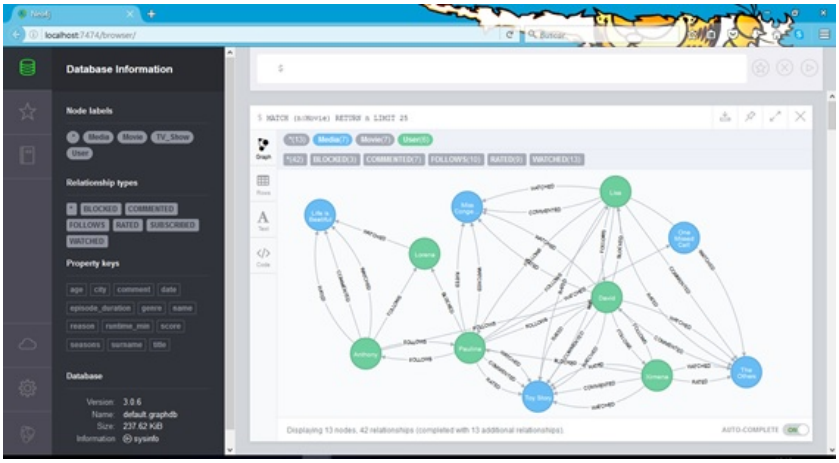


Fig. 1.3: Neo4j's web front end.

The MATCH clause is used to express the graph pattern to match. This is the most common way to get data from the graph. A matched result set may be processed by the RETURN clause which may run a projection over the result set. Figure 1.4 shows an example of a Cypher query with MATCH and RETURN clauses in action.

```
1 MATCH (node) RETURN node.property;
2 MATCH (nodeA)-[edgeB]->() RETURN
  nodeA.propertyA, nodeB.propertyB;
3 MATCH (nodeA)--(nodeC) RETURN nodeA.propertyA,
  nodeC.propertyC;
4
```

Fig. 1.4: MATCH and RETURN Clauses.

The WHERE clause is inspired by SQL. Nodes and relationships can be filtered by conditions using this clause. The query in Figure 1.5 returns all name of movies that were liked by a user, but that user must not like "The Conjuring" movie.

Cypher allows for ordering and pagination by the ORDER BY, LIMIT, and SKIP clauses.

The WITH clause allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next. Figure 1.6 reports a Cypher query limits matched paths to m node to the last one and then returns all adjacent nodes.

```
MATCH (movie:Movie) WHERE movie.duration > 1
RETURN movie.name;
MATCH (user:User)-[:LIKES]->(movie:Movie)
WHERE NOT (user)-[:LIKES]->(:Movie { name:
'The Conjuring' })
RETURN movie.name;
```

Fig. 1.5: WHERE Clause.

```
MATCH (a { name: 'Paulina' })--(b)
WITH b ORDER BY b.name DESC LIMIT 1
MATCH (b)--(c) RETURN o.name;
```

Fig. 1.6: WITH Clause.

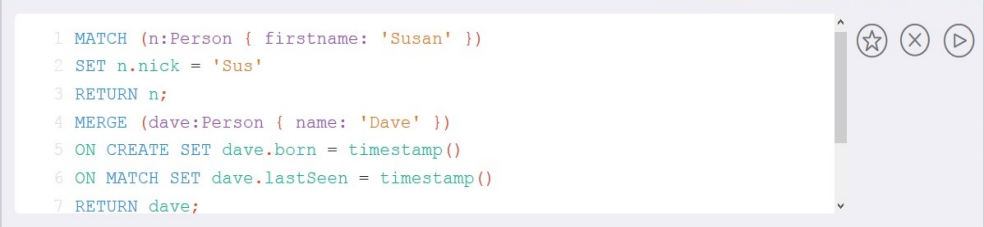
The UNION clause combines the result of multiple queries. The UNION ALL clause is used to leave duplicates in the result set. An example is reported in Figure 1.7.

```
MATCH (s:Singer) RETURN s.name AS name
UNION ALL
MATCH (s:Sing) RETURN s.title AS name;
```

Fig. 1.7: UNION Clause.

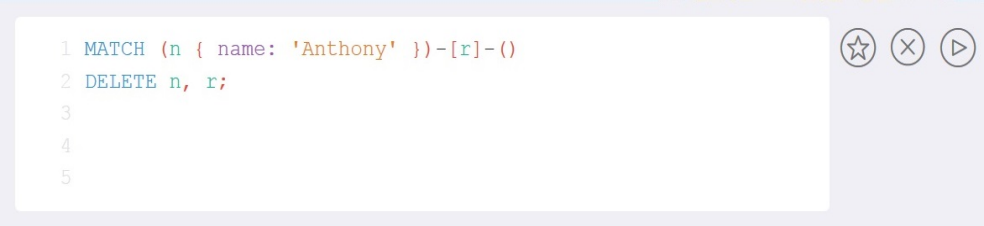
The SET clause updates labels on nodes and properties on nodes and relationships. Setting property to the NULL removes the property. There are also ON MATCH SET and ON CREATE SET for additional control over what happens depending on whether the node was found or created, respectively. An example is reported in Figure 1.8.

The DELETE clause deletes graph elements, such as nodes, relationships or paths. As an example, the query in Figure 1.9 removes the nodes with property name Anthony and all the node's relationships.



```
MATCH (n:Person { firstname: 'Susan' })
SET n.nick = 'Sus'
RETURN n;
MERGE (dave:Person { name: 'Dave' })
ON CREATE SET dave.born = timestamp()
ON MATCH SET dave.lastSeen = timestamp()
RETURN dave;
```

Fig. 1.8: SET Clause.



```
MATCH (n { name: 'Anthony' })-[r]-()
DELETE n, r;
```

Fig. 1.9: DELETE Clause.

1.2.2 OrientDB

OrientDB [61] is an open source NoSQL database management system written in Java. It is a multi-model database, supporting graph, document, key/value, and object models.

The *Graph Model* is a data model that allows us to store data in the form of nodes connected by edges. The vertex and edge components are the central pieces of the graph model.

In the *Document Model* the data is stored in documents and groups of documents are called "collections".

The *Key/Value Model* means that data can be stored in the form of key/value pairs where the values can be simple or complex types. It can support documents and graph elements as values.

The *Object Model* has been inherited by object-oriented programming and supports inheritance between types, polymorphism and direct binding from/to objects.

The following terminology is used in OrientDB:

- **Record.** The record is the smallest unit that the system can load and store in the the database. Records can be stored as Document, Record Bytes, Vertex or Edge.
- **Record ID.** When OrientDB generates a record, each record has its own self-assigned unique ID within the database called Record ID or RID. The RID looks like: ⟨cluster-id⟩:⟨cluster-position⟩ where ⟨cluster-id⟩ is the

cluster identifier and ⟨cluster-position⟩ is the position of the data within the cluster.

- **Documents.** Documents are defined by schema classes with defined constraints or without any schema.
- **Class.** The concept of class is derived from the object-oriented paradigm and it is very similar to a relational table where classes can be schema-less, complete or mixed schema. One class can inherit all attributes from another. Each class has its own grouping, although it must have at least one default cluster. When one runs a query on a class, it propagates to the cluster that is part of the same class.
- **Cluster.** A cluster or group is a place where the records are stored. We can compare the cluster concept to a table in the relational database model. OrientDB by default creates a cluster for each class. All the records of a class are stored in the same cluster having the same name of the class. Although the default strategy is to create a cluster for each class, a class can have multiple clusters.
- **Relationships.** OrientDB supports two kinds of relationships: referenced and embedded.
  - **Referenced relationships.** The reports in OrientDB are handled natively, without making costly join performed in a relational DBMS. OrientDB stores the direct link between objects in relation to each other.
  - **Embedded relationships.** It means it stores the relationship within the record that embeds it. This type of relationship is stronger than the reference relationship.

The table of data types is shown below.

TYPE	DESCRIPTION
Boolean	Handles only the values True or False.
Integer	32-bit signed integers.
Short	Small 16-bit signed integers.
Long	Big 64-bit signed integers.
Float	Decimal numbers.
Double	Decimal numbers with high precision
Date-time	Any date with the precision up to milliseconds.
String	Any string as alphanumeric sequence of chars.
Binary	Can contain any value as byte array.
Embedded	The record is contained inside the owner. The contained record has no RecordId.
Embedded list	The records are contained inside the owner. The contained records have no RecordIds and are reachable only by navigating the owner record.
Embedded set	The records are contained inside the owner. The contained records have no RecordId and are reachable only by navigating the owner record.
Embedded map	The records are contained inside the owner as values of the entries, while the keys can only be strings. The contained records have no RecordId and are reachable only by navigating the owner Record.
Link	Link to another Record. It's a common one-to-one relationship
Link list	Links to other Records. It's a common one-to-many relationship where only the RecordIds are stored.
Link set	Links to other records. It's a common one-to-many relationship.
Link map	Links to other records as value of the entries, while keys can only be strings. It's a common one-to-many relationship. Only the RecordIds are stored.
Byte	Single byte. Useful to store small 8-bit signed integers.
Transient	Any value not stored on database.
Date	Any date as year, month and day.
Custom	Used to store a custom type providing the Marshall and Unmarshall methods.
Decimal	Decimal numbers without rounding.
LinkBag	List of RecordIds as specific RidBag.
Any	Not determinate type, used to specify collections of mixed type, and null..

Table 1.1: OrientDB Data Types.

The OrientDB Console is a Java Application. OrientDB supports the following console modes:

- **Interactive Mode.** This is the default mode. The Console starts in interactive mode. We can execute commands and SQL statements, the Console loads data, and once done the console is ready to accept commands.
- **Batch Mode.** Running the console in batch mode takes commands as an argument or as a text file.

OrientDB is implemented in Java, and therefore available to a good standard API for interfacing through this language. There are three models to take advantage of APIs, depending on the model with which it wants to work: Graph API, Documents API, and Object API.

- **Graph API:** These APIs allow one to work on graphs, one of the most flexible data structures.
- **Document API:** These APIs are the most immediate to use because they even match one of the most common uses of a NoSQL database. They allow you to convert Java objects into documents and save them in OrientDB.
- **Object API:** These APIs are based on documents and help one work with real persistent Java objects, using the reflection mechanism.

The stratification of the functionality of the APIs above is represented by the following figure:

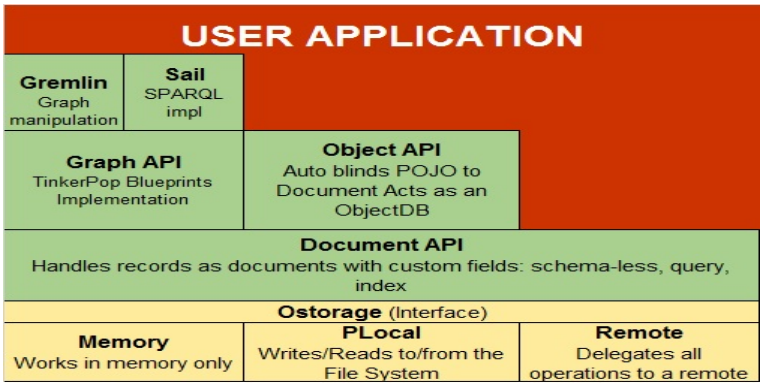


Fig. 1.10: Component Architecture.

The operations that OrientDB can perform over a database are listed below:

- Create Database
- Alter Database
- Backup Database
- Restore Database
- Connect Database

- Disconnect Database
- Info Database
- List Database
- Freeze Database
- Release Database
- Config Database
- Export Database
- Import Database
- Commit Database
- Rollback Database
- Optimize Database
- Drop Database

To handle the records OrientDB offers the following commands

- Insert Record
- Displays Records
- Load Records
- Reload Record
- Export Record
- Update Record
- Truncate Record
- Delete Record

OrientDB allows users to handle classes and cluster by means of commands to create, alter, truncate and drop.

The OrientDB commands to act on properties are:

- Create Property
- Alter Property
- Drop Property

The OrientDB commands to handles vertices and edges are:

- Create Vertex
- Move Vertex
- Delete Vertex
- Create Edge
- Update Edge
- Delete Edge

OrientDB offers also indexing mechanisms and support for transactions. Furthermore, OrientDB supports JDBC and has a Python Interface.

2

Incremental Maintenance of the Maximum Flow

The maximum flow problem is a classical optimization problem with a wide range of applications. Nowadays, it is successfully applied in social network analysis for link spam detection, web communities identification, and others. In such applications, flow networks are used to model connections among web pages, online voting systems, web communities, P2P and other distributed systems. Thus, networks are highly *dynamic*, that is, subject to frequent updates.

While many efficient algorithms for the maximum flow problem have been proposed over the years, they are designed to work with *static* networks, and thus they need to recompute a new solution from scratch every time an update occurs. Such approaches are impractical in scenarios like the aforementioned ones, where updates are frequent.

To overcome these limitations, in this chapter we propose efficient *incremental* algorithms for maintaining the maximum flow in dynamic networks [36, 38]. Our approach identifies and acts only on the affected portions of the network, reducing the computational effort to update the maximum flow. We evaluate our approach on different families of datasets, comparing it against state-of-the-art algorithms, showing that our technique is significantly faster and can efficiently handle networks with millions of vertices and tens of millions of edges.

2.1 Introduction

The maximum flow problem arises in settings as diverse as communication systems, image processing, scheduling, distribution planning, and many others [3, 33]. Even if the problem has a long history, revolutionary progress is still being made and the number of applications is continuously growing—e.g., see [33]. Interestingly, this problem is nowadays successfully applied in social network analysis for link spam detection [58], for the identification of web communities [25, 42], and to defend against sybil attacks [64, 62]. In such

applications, flow networks are used to model connections among web pages, online voting systems, web communities, P2P and other distributed systems. Thus, networks are highly *dynamic*, that is, subject to frequent updates.

Even though many efficient algorithms for the maximum flow problem have been proposed over the years, they are designed to work with *static* networks, and thus they need to recompute a new solution from scratch every time the network is modified. However, the aforementioned scenarios call for *incremental* algorithms, as it is impractical to compute a new solution from scratch every time an update occurs.

To overcome these limitations, we propose novel *incremental* algorithms for maintaing the maximum flow in dynamic networks.

Contributions. Specifically, we make the following main contributions:

- We propose efficient incremental algorithms to maintain the maximum flow after vertex insertions/deletions and edge insertions/deletions/updates. Our algorithms are designed to effectively identify only the affected parts of the network, in order to reduce the computational effort for determining the new maximum flow.
- We show correctness of the algorithms and provide complexity analyses.
- We report on an experimental evaluation we conducted on several families of datasets with millions of vertices and tens of millions of edges. Experimental results show that our approach is very efficient and outperforms state-of-the-art algorithms.

2.2 Related Work

The maximum flow problem is a classical optimization problem and many algorithms to solve it have been proposed over the years [26, 20, 21, 32, 13, 8, 48, 41, 29, 30, 9, 10, 31]. The first solutions to this problem were mainly based on finding augmenting paths in the residual network [26, 20, 21]—roughly speaking, these are paths from the source to the sink along which it is possible to send additional flow.

Approaches based on the push-relabel algorithm [32] have been proposed in [13, 29, 30]. Push-relabel algorithms do not maintain a valid flow during their execution. In particular, some vertices might have a positive flow excess. By maintaining a label for each vertex denoting a lower bound on its distance to the sink along non-saturated edges, the excess is then pushed toward vertices with smaller label or, eventually, sent back to the source.

The highest-level push-relabel implementation introduced in [13], named HIPR, was for a long time a benchmark for maximum flow algorithms. Optimizations have been introduced in the partial augment-relabel algorithm (PAR) [29]. The same data structures and heuristics of PAR are used by the two-level push-relabel algorithm (P2R) [30]. These solutions use both the

global update [32] and the gap relabeling [10] heuristics to improve the performance of the push-relabel method.

There are also algorithms that solve the minimum cut and the maximum flow problems without maintaining a feasible flow [41, 31]. The solution proposed in [41], named HPF, terminates its execution with the min-cut and a pseudoflow (which means that some vertices might have a positive or negative flow excess). However, it is possible to convert the pseudoflow into a maximum feasible flow by computing flow decomposition in a related network. As shown in [24, 10], the time spent in flow recovery is typically small compared to the time to find the min-cut.

Different from the above mentioned approaches, the algorithm of Boykov and Kolmogorov (BK) [8] has no strongly polynomial time bound. However, given its practical efficiency, it is probably the most widely used algorithm in computer vision. BK is based on augmenting paths. It builds two search trees, one from the source and the other from the sink, and reuses them.

The main limitation of all the aforementioned algorithms is that, when dealing with dynamic networks, they need to recompute a new maximum flow *from scratch* every time the network changes, which is impractical in applications where updates occur frequently.

Recently, some solutions have been proposed that consider the maximum flow problem in a dynamic setting [48, 31]. They mainly focus on solving a given series of maximum flow instances, each obtained from the previous one by relatively few changes in the input. The solution introduced in [48] is an extension of the BK algorithm (E-BK) to the dynamic setting. In [31], the Excesses Incremental Breadth-First Search (E-IBFS) algorithm is proposed. E-IBFS maintains a pseudoflow (like HPF), and thus an additional step has to be performed if a flow is required.

2.3 Preliminaries

In this section, we briefly recall the maximum flow problem and introduce notation and terminology used in the rest of this chapter (we refer the reader to [3] for a comprehensive treatment of the topic).

A *flow network* is a tuple $N = (V, A, s, t, c)$, where

- (V, A) is a directed graph with vertex set V and edge set A;
- s and t are two distinguished vertices in V called the *source* and the *sink*, respectively;
- c is a capacity function $c : A \to \mathbb{R}^+$ assigning a positive real capacity $c(u, v)$ to each edge (u, v) in A.

For convenience, we define $c(u, v) = 0$ for every $(u, v) \notin A$, and assume that there are no self-loops.

A *flow* in N is a function $f : V \times V \to \mathbb{R}$ satisfying the following two properties:

- $0 \leq f(u,v) \leq c(u,v)$ for all $u, v \in V$;
- $\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$ for all $u \in V \setminus \{s,t\}$.

The first property above says that the flow along an edge (u,v) cannot exceed its capacity, and the second property says that for each vertex (other than the source and the sink), its incoming flow must equal its outgoing flow.

The *value* of f is $|f| = \sum_{v \in V} f(v,t) - \sum_{v \in V} f(t,v)$, that is, it is the net flow into the sink. The maximum flow problem consists in finding a flow of maximum value.

Given a flow f in N and two vertices $u, v \in V$, the *residual capacity* is defined as $r_f(u,v) = c(u,v) - f(u,v) + f(v,u)$. The *residual network* of N induced by f is the directed graph $N_f = (V, E_f)$, where $E_f = \{(u,v) \in V \times V : r_f(u,v) > 0\}$. A simple path (i.e., a path without repeating vertices) from s to t in N_f is called an *augmenting path*.

2.4 Incremental Maximum Flow Computation

In this section, we present algorithms for the incremental maintenance of the maximum flow. We first propose an algorithm to handle edge insertions and capacity increases (Section 3.4.1), and then address edge deletions and capacity decreases (Section 3.4.2).

It is worth noting that insertions and deletions of *vertices* can be straightforwardly reduced to our setting and thus can be handled by our algorithms too: vertex insertions (resp. deletions) are handled by inserting (deleting) all edges that are incident from/to the inserted (resp. deleted) vertices.[1] As a consequence, *our algorithms can handle arbitrary sequences of edge insertions/deletions/updates (the latter to be understood as capacity updates) and vertex insertions/deletions.* Since the insertion of an edge (a,b) s.t. either a or b (or both) is a new vertex does not affect the maximum flow—i.e., the flow along (a,b) will be zero and remain the same elsewhere—we assume that both a and b belong to the current flow network.

Given a flow network $N = (V, A, s, t, c)$, a pair of vertices $(a,b) \in V \times V$, and a real value w, we use $update(N, (a,b), w)$ to denote the flow network $N' = (V, A', s, t, c')$ such that c' is the same as c except that $c'(a,b) = \max\{0, c(a,b) + w\}$ and $A' = \{(u,v) \in V \times V \mid c'(u,v) > 0\}$. Thus, when $w > 0$, if $(a,b) \in A$ then the edge capacity is increased by w, otherwise the new edge (a,b) with capacity w is inserted into the network. When $w < 0$, the capacity of (a,b) is decreased by w, which corresponds to deleting the edge altogether when $c'(a,b) = 0$.

We start by introducing two functions which will be used by both our algorithms: ASP (Augmenting Shortest Path) and UF (Update Flow).

[1] Vertices without incident edges can be neglected for our purposes.

ASP takes as input a residual network $N_f = (V, E_f)$ and two vertices $x, y \in V$, and it returns two vertex-indexed arrays, *prev* and *flow*, containing the shortest path from x to y (if it exists) and the maximum additional flow that can be pushed along that path, respectively. Essentially, ASP performs a breadth-first search from x, and for each vertex v reached during the visit, it keeps track of the maximum additional flow (denoted $flow[v]$) that v can receive from x along the discovered shortest (in terms of number of edges) path connecting them, together with v's predecessor in such a path (denoted $prev[v]$). If y is reached, then it is possible to build the whole path from x to y by following the chain of predecessors in *prev* starting from y. Otherwise (y is not reachable from x), the algorithm stops when there are no more vertices reachable from x. In the function, Q is a first-in, first-out queue. Also, we use $adj[u]$ to denote the set of u's adjacent vertices (in N_f).

The UF function takes as input a flow network N, a flow f in N, an additional flow value Δf, two vertices x, y, and an array of predecessors *prev* (which will be computed by ASP in Algorithms 3 and 2). The function simply increases f by Δf along the path from x to y stored in *prev*.

2.4.1 Edge Insertions and Capacity Increases

In this section, we present our algorithm to incrementally maintain the maximum flow after inserting a new edge or increasing the capacity of an exiting one. Algorithm 3 takes as input a flow network N, a maximum flow f in N, an edge (a, b), and a positive capacity w, and computes a maximum flow in $update(N, (a, b), w)$.

First of all, notice that lines 2–27 are executed only if (a, b) is a new edge or an existing one that has been used to its full capacity, because if this condition does not hold, then the insertion/update of (a, b) has no effect on the maximum flow. The algorithm works as follows. First, it computes the new flow network N (line 2). Then, it looks for a (shortest) path from s to a in the residual network by calling the ASP function (line 3). If such a path exists, an analogous search is performed from b to t (line 6). If also such a path exists, then there is an augmenting path p from s to t and the maximum flow is increased on the edges of p (lines 9–12).

After that, if (a, b) has still positive residual capacity, Algorithm 3 repeats the search of augmenting paths in N_f as follows (lines 16–27). If additional flow can be sent from s to a along the previously discovered path, the algorithm looks for a path from b to t (lines 17–18); otherwise (no additional flow can be sent from s to a along the previously discovered path), if additional flow can be sent from b to t along the previously discovered path, the algorithm looks for a path from s to a (lines 20–21); otherwise (no more flow can be sent along the previously discovered paths), the algorithm looks for both a path from s to a and a path from b to t (lines 23–27). The search continues until either the residual capacity of (a, b) gets to zero (i.e., no more additional flow

Function 1 Augmenting Shortest Path (ASP)

Input: A residual network $N_f = (V, E_f)$,
two vertices $x, y \in V$.
Output: Arrays *prev* and *flow*.

```
1: flow[1..|V|];
2: prev[1..|V|];
3: for each v ∈ V do
4:    flow[v] := −1;
5:    prev[v] := NIL;
6: flow[x] := +∞;
7: if x = y then
8:    return ⟨prev, flow⟩;
9: Q := {x};
10: while Q ≠ ∅ do
11:    u := Q.dequeue();
12:    for v ∈ adj[u] s.t. flow[v] = −1 do
13:       flow[v] := min{flow[u], r_f(u, v)};
14:       prev[v] := u;
15:       if v = y then
16:          return ⟨prev, flow⟩;
17:       Q.enqueue(v);
18: return ⟨prev, flow⟩;
```

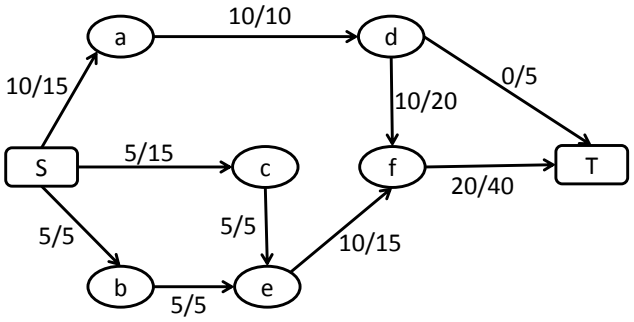


Fig. 2.1: A flow network and a maximum flow.

can be pushed along (a, b)), or no other path from s to a or from b to t can be found.

The following example shows how Algorithm 3 works.

Example 2.1. Consider the flow network and the maximum flow in Figure 2.1. Each edge label x/y states that y is the edge capacity and x is the current flow along the edge. Suppose the edge (c, d) with capacity 15 is added to the network.

Figure 2.2 shows the augmenting paths computed by Algorithm 3. Specifically, Figure 2.2 (left) highlights the path from the source to c (blue-colored

Function 2 Update Flow (UF)

Input: A flow network $N = (V, A, s, t, c)$,
a flow f in N,
an additional flow value Δf,
two vertices $x, y \in V$,
an array of predecessors *prev*.
Output: A flow in N.

```
1:  b := y;
2:  while b ≠ x do
3:     a := prev[b];
4:     if (a, b) ∈ A then
5:        diff := Δf − (c(a, b) − f(a, b));
6:        f(a, b) := f(a, b) + min{Δf, c(a, b) − f(a, b)};
7:        if diff > 0 then
8:           f(b, a) := f(b, a) − diff;
9:     else
10:       f(b, a) := f(b, a) − Δf;
11:    b := a;
12: return f;
```

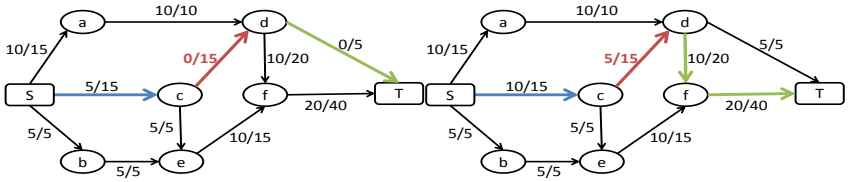


Fig. 2.2: First (left) and second (right) iteration of Algorithm 3.

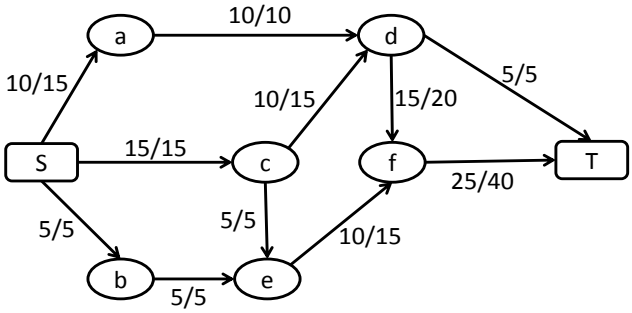


Fig. 2.3: Updated flow network and maximum flow.

edge), whose residual capacity is 10, and the path from d to the sink (green-colored edge), whose residual capacity is 5. The red edge is the new inserted

Algorithm 1 Edge-Insertion-Maintenance (EIM)

Input: A flow network $N = (V, A, s, t, c)$,
a maximum flow f in N,
a pair of vertices (a, b),
a capacity $w \in \mathbb{R}^+$.
Output: A maximum flow for $update(N, (a, b), w)$.

```
1: if c(a,b) − f(a,b) = 0 then
2:    N := update(N, (a,b), w);
3:    ⟨prev_sa, flow_sa⟩ := ASP(N_f, s, a);
4:    Δf_sa := flow_sa[a];
5:    if Δf_sa > 0 then
6:       ⟨prev_bt, flow_bt⟩ := ASP(N_f, b, t);
7:       Δf_bt := flow_bt[t];
8:    while r_f(a,b) > 0 ∧ Δf_sa > 0 ∧ Δf_bt > 0 do
9:       Δf := min{r_f(a,b), Δf_sa, Δf_bt};
10:      f(a,b) := f(a,b) + Δf;
11:      f := UF(N, f, Δf, s, a, prev_sa);
12:      f := UF(N, f, Δf, b, t, prev_bt);
13:      Δf_sa := Δf_sa − Δf;
14:      Δf_bt := Δf_bt − Δf;
15:      if r_f(a,b) > 0 then
16:         if Δf_sa > 0 then
17:            ⟨prev_bt, flow_bt⟩ := ASP(N_f, b, t);
18:            Δf_bt := flow_bt[t];
19:         else if Δf_bt > 0 then
20:            ⟨prev_sa, flow_sa⟩ := ASP(N_f, s, a);
21:            Δf_sa := flow_sa[a];
22:         else
23:            ⟨prev_sa, flow_sa⟩ := ASP(N_f, s, a);
24:            Δf_sa := flow_sa[a];
25:            if Δf_sa > 0 then
26:               ⟨prev_bt, flow_bt⟩ := ASP(N_f, b, t);
27:               Δf_bt := flow_bt[t];
28: return f;
```

one and its residual capacity is 15. The flow across the resulting augmenting path is increased by 5.

Then, as both (c, d) and the path ending in c have still residual capacity after updating the flow, the algorithm looks for a new path in the residual network from d to the sink, and finds the one consisting of the green-colored edges in Figure 2.2 (right). This allows the algorithm to identify a new augmenting path, the one highlighted in Figure 2.2 (right), along which the flow is increased by 5.

After that, the path from the source to c is saturated, and since there is no other path from the source to c in the residual network, the algorithm

terminates. Figure 2.3 shows the updated network with its new maximum flow.

The following theorems state the correctness and time complexity of Algorithm 3.

Theorem 2.2. *Given a flow network N, a maximum flow f in N, an edge (a, b), and a capacity $w > 0$, Algorithm 3 returns a maximum flow in $update(N, (a, b), w)$.*

Theorem 2.3. *Algorithm 3 runs in $O(m \min\{nm, \Delta f\})$ time, where $n = |V|$, $m = |A|$, and Δf is the maximum flow value's increase.*

In light of the previous theorem, in the presence of small variations of the flow value, our algorithm performs very well, confirming its incremental nature, and, anyway, it never performs worse than the Edmonds-Karp algorithm.

2.4.2 Edge Deletions and Capacity Decreases

In this section, we present our algorithm to maintain the maximum flow after decreasing an edge capacity (recall that this case includes also the deletion of an edge). Algorithm 2 below takes as input a flow network N, a maximum flow f in N, an edge (a, b), and a negative capacity w, and computes a maximum flow in $update(N, (a, b), w)$.

The algorithm first computes the amount of flow that (a, b) cannot manage anymore, namely *excess* (line 1). If *excess* is positive (i.e., the flow currently assigned to (a, b) exceeds the new capacity), then lines 3–21 are executed. Specifically, the new network is computed (line 3). Then, the flow on (a, b) is set to its new capacity (line 4). After that, the following three phases are performed:

(i) as vertex b has more outgoing than incoming flow, the exceeding one (namely, *excess*) is sent back from t to b (line 5);

(ii) as vertex a has more incoming than outgoing flow, it tries to push this excess to t along new routes not going through (a, b) (lines 6–13);

(iii) if, after the previous phase, a has still some excess, this is eventually pushed back to s (lines 14–21).

Step *(i)* above uses the REF function to reduce the flow from a vertex to the sink. Specifically, REF takes as input a flow network N, a flow f in N, a vertex x of N, and a positive real value *excess*. The function decreases the flow from x to the sink by *excess* by iteratively searching (in a breadth-first manner) paths from x to the sink having a positive flow on every edge.

Example 2.4. Consider the flow network and the maximum flow in Figure 2.3. Suppose we delete the edge (e, f).

Algorithm 2 Edge Deletion Maintenance (EDM)

Input: A flow network $N = (V, A, s, t, c)$,
a maximum flow f in N,
an edge $(a, b) \in A$,
a capacity $w \in \mathbb{R}^-$.
Output: A maximum flow for $update(N, (a, b), w)$.

1: $excess := f(a, b) - \max\{0, c(a, b) + w\}$;
2: **if** $excess > 0$ **then**
3: $N := update(N, (a, b), w)$;
4: $f(a, b) := f(a, b) - excess$;
5: $f := \mathsf{REF}(N, f, b, excess)$;
6: $\langle prev_{at}, flow_{at} \rangle := \mathsf{ASP}(N_f, a, t)$;
7: $\Delta f_{at} := \min\{flow_{at}[t], excess\}$;
8: **while** $excess > 0 \wedge \Delta f_{at} > 0$ **do**
9: $excess := excess - \Delta f_{at}$;
10: $f := \mathsf{UF}(N, f, \Delta f, a, t, prev_{at})$;
11: **if** $excess > 0$ **then**
12: $\langle prev_{at}, flow_{at} \rangle := \mathsf{ASP}(N_f, a, t)$;
13: $\Delta f_{at} := \min\{flow_{at}[t], excess\}$;
14: $\Delta f_{as} := \min\{flow_{at}[s], excess\}$;
15: $prev_{as} := prev_{at}$;
16: **while** $excess > 0$ **do**
17: $excess := excess - \Delta f_{as}$;
18: $f := \mathsf{UF}(N, f, \Delta f, a, s, prev_{as})$;
19: **if** $excess > 0$ **then**
20: $\langle prev_{as}, flow_{as} \rangle := \mathsf{ASP}(N_f, a, s)$;
21: $\Delta f_{as} := \min\{flow_{as}[s], excess\}$;
22: **return** f;

Figure 2.4 (left) shows the effect of the deletion by highlighting the excess at nodes e and f (red-colored numbers). More specifically, vertex f has a negative excess as it is receiving less flow than that being sent. The REF function sends this flow back to f from the sink, through the only path connecting them—see green-colored edge in Figure 2.4 (right).

Then, the algorithm tries to push the excess at e toward the sink. Figure 2.5 (left) shows the path (blue-colored edges) computed by function ASP: 5 units can be pushed to the sink along this path.

After that, as there is still some excess at e, the algorithm looks for other paths from e to the sink in the residual network, but none is found. As a result, the exceeding flow at e has to be pushed back to the source. Even if the last ASP search did not succeed in finding a path from e to the sink, it found a path from e to the source, which is highlighted in Figure 2.5 (right). The excess at e is decreased along such a path. Figure 2.6 shows the updated network with its new optimal flow.

Function 2 Reset Exceeding Flow (REF)

Input: A flow network $N = (V, A, s, t, c)$,
a flow f in N,
a vertex $x \in V$,
$excess \in \mathbb{R}^+$.
Output: A flow in N.

```
 1: if x ≠ t then
 2:   while excess > 0 do
 3:     flow[1..|V|];
 4:     prev[1..|V|];
 5:     for each v ∈ V do
 6:       flow[v] := −1;
 7:       prev[v] := NIL;
 8:     flow[x] := excess;
 9:     Q := {x};
10:     while Q ≠ ∅ do
11:       u := Q.dequeue();
12:       for v ∈ V s.t. f(u, v) > 0 ∧ flow[v] = −1 do
13:         flow[v] := min{flow[u], f(u, v)};
14:         prev[v] := u;
15:         if v = t then
16:           go to line 16;
17:         Q.enqueue(v);
18:     b := t;
19:     while b ≠ x do
20:       a := prev[b];
21:       f(a, b) := f(a, b) − flow[t];
22:       b := a;
23:     excess := excess − flow[t];
24: return f;
```

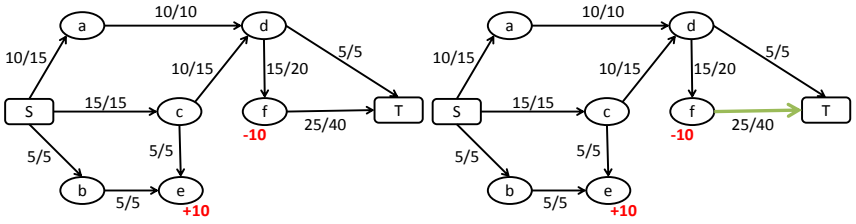


Fig. 2.4: Flow network with excess (left) and paths considered by the REF function (right).

The following theorems state the correctness and the time complexity of Algorithm 2.

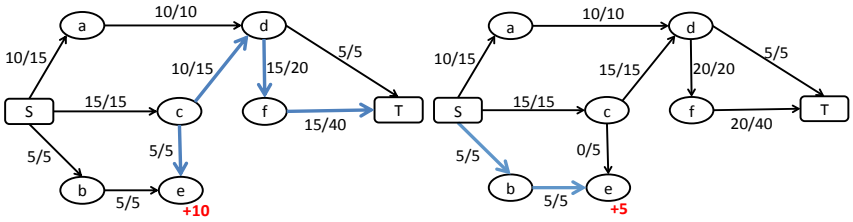


Fig. 2.5: Augmenting path from e to the sink (left) and augmenting path from e to the source (right).

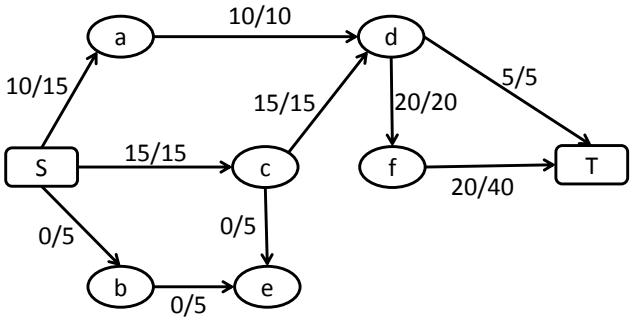


Fig. 2.6: Updated flow network.

Theorem 2.5. *Given a flow network N, a maximum flow f in N, an edge (a, b), and a capacity $w < 0$, Algorithm 2 returns a maximum flow in update$(N, (a, b), w)$.*

Theorem 2.6. *Algorithm 2 runs in $O(m \min\{nm, excess\})$ time, where $n = |V|$ and $m = |A|$.*

It is worth noting that Theorem 2.6 says that our algorithm performs very well in the presence of small variations of the flow value, that is, when *excess* is small, confirming its incremental nature. Moreover, in any case, the algorithm performs no worse than the Edmonds-Karp algorithm.

2.5 Experimental Evaluation

In this section, we report on an experimental evaluation we performed to compare our approach against state-of-the-art algorithms on different families of datasets.

All experiments were run on an Intel i7 3770K 3.5 GHz, 12 GB of memory, running Linux Mint 17.1.

Datasets. We used a variety of networks taken from the following repositories: *The Maximum Flow Project Benchmark* [2] and *Computer Vision*

Datasets				Execution times (secs)								
Family	Name	# of vertices	# of edges	HIPR	PAR	P2R	HPF	BK	E-BK	E-IBFS	EIM	EDM
multiview	gargoyle-med	8,847,362	44,398,548	46.27	22.17	18.81	10.06	88.54	92.16	8.94	**0.72**	**2.37**
	camel-med	9,676,802	47,933,324	56.14	33.84	27.88	14.26	17.95	17.33	6.88	**0.83**	**4.56**
3D segmentation	bone-sx6c100	3,899,396	23,091,149	6.58	1.76	1.79	1.68	3.46	3.54	1.47	**0.06**	**0.19**
	liver6c100	4,161,604	22,008,003	16.47	7.90	7.81	6.66	7.74	7.69	3.32	**0.17**	**0.85**
	babyface6c100	5,062,504	29,044,746	27.45	14.77	14.05	16.06	6.38	6.50	**3.28**	**0.76**	5.19
	bone6c100	7,798,788	45,534,951	13.62	3.33	3.46	3.02	4.07	4.16	2.07	**0.10**	**0.71**
	adhead6c100(64bit)	12,582,916	74,245,555	35.04	11.91	12.43	16.45	17.76	18.41	7.91	**0.76**	**1.97**
surface fitting	bunny-med	6,311,088	38,739,041	18.33	10.72	12.27	6.60	1.05	1.08	0.84	**0.74**	**0.78**
lazy-brush	lbrush-mangagirl	593,032	2,379,695	2.81	1.69	1.38	0.45	0.28	0.28	**0.22**	**0.02**	0.70
	lbrush-elephant	2,370,064	9,492,348	15.17	9.96	9.59	9.82	1.79	1.89	2.46	**0.15**	**1.67**
	lbrush-bird	2,372,116	9,505,709	16.31	9.03	9.28	10.81	2.72	2.80	**2.42**	**0.16**	5.25
	lbrush-doctor	2,373,272	9,522,523	11.98	7.54	6.28	2.52	1.15	1.15	**0.91**	**0.07**	0.97
PUNCH	punch-us22p	1,674,084	4,718,690	4.76	**1.67**	1.76	3.41	28.51	29.11	6.09	**0.14**	24.51
	punch-us22u	1,674,084	4,718,690	4.02	1.36	1.61	1.34	4.35	4.42	1.40	**0.15**	**0.16**
	punch-eu22p	2,020,897	5,715,868	10.88	4.05	3.70	2.52	3.41	3.54	**1.41**	**0.26**	2.76
	punch-eu22u	2,020,897	5,715,868	8.45	2.80	2.55	1.02	1.20	1.24	0.61	**0.190**	**0.090**
bisection	cal	1,800,723	4,434,236	8.81	3.68	4.05	1.28	0.51	0.53	**0.33**	0.34	**0.12**

Table 2.1: Execution times (secs).

Datasets [1]. We chose these datasets as they have been extensively used in the literature to compare maximum flow algorithms (e.g., they have been used in [31, 24]). In particular, we chose the largest networks of different families. Details are reported in (the first four columns of) Table 2.1. Most of the networks have millions of vertices (up to 12.5M vertices) and tens of millions of edges (up to 74M edges).

Algorithms. We compared our algorithms against state-of-the-art algorithms for the maximum flow computation, namely HIPR [13], PAR [29], P2R [30], HPF [41], BK [8], E-BK [48], and E-IBFS [31] (see Section 3.2 for a discussion of them). We used the implementations of the algorithms provided by their authors. Our algorithms were written in Java.

Results. For each of the datasets listed in Table 2.1, we randomly chose 250 edges to be inserted and 250 edges to be deleted. The EIM (resp. EDM) column reports the average time taken by our insertion (deletion) algorithm to handle the 250 edge insertions (deletions). The average time taken by each of the competitors to handle the edge insertions slightly differs from the average time taken to handle the edge deletions (the difference was on the order of 10^{-2} seconds or less), so we report only one time. Thus, each competitor's running time in Table 2.1 is to be interpreted as both the average time to handle the 250 insertions and the average time to handle the 250 deletions. The reason of the negligible difference between the two average times is that a one-edge modification does not affect much the overall running time, which is mainly determined by the size of the whole network.

To ease readability, for each dataset, the lowest running times are highlighted in blue. As for edge insertions, EIM is always the fastest algorithm,

except for the last dataset, where E-IBFS is faster, even though the difference is negligible. In most of the cases (around 75% of the datasets), EIM is more efficient than the second-fastest algorithm by at least one order of magnitude.

Regarding deletions, EDM is the fastest algorithm in most of the cases. In the remaining cases E-IBFS is the fastest one, with the only exception of the punch-us22p dataset. However, we point out that E-IBFS computes only the *value* of a flow, rather than the flow function, while EDM computes the flow function (which gives the flow for each edge). For the punch-us22p dataset, the fastest algorithm is PAR (which is based on the push-relabel approach), while algorithms based on augmenting paths showed poor performances in this case.

Finally, we observe that handling edge deletions showed to be more expensive than handling edge insertions.

2.6 Discussion

While many algorithms have been proposed for the maximum flow problem, many current applications deal with dynamic networks and thus call for incremental algorithms, as it is impractical to compute the new maximum flow from scratch every time updates occur.

We have proposed efficient incremental algorithms to maintain the maximum flow in evolving networks. Experimental results have shown that our approach outperforms state-of-the-art algorithms and can easily handle networks with millions of vertices and tens of millions of edges.

We point out that our algorithms can be directly applied to incrementally maintain the minimum cut too, because of the well-known max-flow/min-cut theorem (i.e., the maximum flow value is equal to the minimum cut capacity), thereby widening further the range of applications.

As directions for future work, we plan to generalize our algorithms to handle batches of updates at once, and leverage graph database systems.

3

Incremental Maintenance of All-Pairs Shortest Paths in Relational DBMSs

Computing shortest paths is a classical graph theory problem and a central task in many applications. Although many algorithms to solve this problem have been proposed over the years, they are designed to work in the main memory and/or with static graphs, which limits their applicability to many current applications where graphs are highly *dynamic*, that is, subject to frequent updates.

In this chapter, we present novel efficient incremental algorithms for maintaining all-pairs shortest paths and distances in dynamic graphs [35, 37]. We experimentally evaluate our approach on several real-world datasets, showing that it significantly outperforms current algorithms designed for the same problem.

3.1 Introduction

Computing shortest path and distances has a wide range of applications in social network analysis [46], road networks [63], graph pattern matching [22], biological networks [55], and many others. It is both an important task in its own right (e.g., if we want to know how close people are in a social network) and a fundamental subroutine for many advanced tasks.

For instance, in social network analysis, many significant network metrics (e.g., eccentricity, diameter, radius, and girth) and centrality measures (e.g., closeness and betweenness centrality) require knowing the shortest paths or distances for all pairs of vertices. There are many other domains where it is needed to compute the shortest paths (or distances) for *all* pairs of vertices, including bioinformatics [52] (where all-pairs shortest distances are used to analyze protein-protein interactions), planning and scheduling [51] (where finding the shortest paths for all pairs of vertices is a central task to solve binary linear constraints on events), and wireless sensor networks [15] (where different topology control algorithms need to compute the shortest paths or distances for all pairs of vertices).

Though several techniques have been proposed to efficiently compute shortest paths/distances, they are designed to work in the main memory and/or with static graphs. This severely limits their applicability to many current applications where graph are dynamic and the graph along with the shortest paths/distances do not fit in the main memory. When graphs are subject to frequent updates, it is impractical to recompute shortest paths or distances from scratch every time a modification occurs.

To overcome these limitations, we propose efficient algorithms for the incremental maintenance of all-pairs shortest paths and distances in dynamic graphs stored in relational DBMSs.

Contributions. We consider the setting where graphs and shortest paths are stored in relational DBMSs and propose novel algorithms to incrementally maintain all-pairs shortest paths and distances after vertex/edge insertions, deletions, and updates. The proposed approach aims at reducing the time needed to update the shortest paths by identifying only those that need to be updated (it is often the case that small changes affect only few shortest paths).

To the best of our knowledge, [50] is the only disk-based approach in the literature for incrementally maintaining all-pairs shortest distances. In particular, like our approach, [50] relies on relational DBMSs. We experimentally compare our algorithms against [50] on five real-world datasets, showing that our approach is significantly faster. It is worth noticing that our approach is more general than [50] in that we keep track of both shortest paths and distances, while [50] maintain shortest distances only (thus, there is no information on the actual paths).

3.2 Related Work

Different variants of the shortest path problem have been investigated over the years: *single-pair shortest path* (SPSP)—find a shortest path from a given source vertex to a given destination vertex; *single-source shortest paths* (SSSP)—find a shortest path from a given source vertex to each vertex of the graph; *all-pairs shortest paths* (APSP)—find a shortest path from x to y for every pair of vertices x and y.

Variants of these problems where we are interested only in the shortest distances, rather than the actual paths, have been studied as well. We will use SPSD, SSSD, and APSD to refer to the *single-pair shortest distance*, *single-source shortest distances*, and *all-pairs shortest distances* problems, respectively.

In this section, we discuss the approaches in the literature to solve the above problems. We first consider non-incremental algorithms, and then incremental ones.

Non-Incremental Algorithms. Since the introduction of the Dijkstra's algorithm [19], a plethora of algorithms have been proposed to improve on its performance (see [59] for a recent survey).

Most of the recently proposed methods require a preprocessing step for building index structures to support the fast computation of shortest paths and distances [11, 44, 4, 34, 65, 53, 54, 66, 12, 27].

Specifically, [11, 44, 4] address the SPSD problem, and a similar approach for the SPSP problem has been proposed in [34]. [65] considers both the SPSP and SPSD problems. There have been also proposals addressing the approximate computation of SPSD [53, 54]. These methods are based on the selection of a subset of vertices as "landmarks" and the offline computation of distances from each vertex to those landmarks. [66] and [12] propose disk-based index structures for solving the SSSP and SSSD problems, while disk-based index structure for the SPSP and SPSD problems have been proposed in [27].

[28] addresses the SPSP problem and, like our approach, relies on relational DBMSs.

Besides the fact that most of the aforementioned techniques assume that graphs, shortest paths/distances, and auxiliary index structures fit in the main memory (which is not realistic in many current applications), the main limit of all the approaches mentioned above is that they need to recompute a solution from scratch every time the graph is modified, even if we make small changes affecting a few shortest paths/distances. In many cases this requires an expensive pre-processing phase to build the index structures that are necessary to answer queries. Then, of course, shortest paths/distances have to be computed again for all pairs. Thus, these methods work well if the graph is static. In contrast, if the graph is dynamic, the expensive pre-processing phase and the re-computation of all shortest paths/distances have to be done every time a change is made to the graph, and this is impractical for large graphs subject to frequent changes.

Incremental Algorithms. Several approaches have been proposed to incrementally maintain shortest paths and distances when the graph is modified. [43] introduces data structures to support APSP maintenance after edge deletions in directed acyclic graphs. [17] considers both edge insertions and deletions. [5] presents a hierarchical scheme for efficiently maintaining all-pairs approximate shortest paths in undirected unweighted graphs and in the presence of edge deletions. [47] proposes an algorithm for maintaining all-pairs shortest paths in directed graphs with positive integer weights bounded by a constant. Another approach for the incremental maintenance of APSPs is [46]. [14] proposes an algorithm for maintaining nearest neighbor lists in weighted

graphs under vertex insertions and decreasing edge weights. The approach is tailored for scenarios where queries are much more frequent than updates.

Approximate APSP maintenance in unweighted undirected graphs has been addressed in [5, 57, 40], while [6] considered edge deletions and weight increases in weighted directed graphs.

All the techniques above share the use of specific complex data structures and work in the main memory, which limits their applicability. In fact, as pointed out in [16, 18], memory usage is an important issue of current approaches, limiting substantially the graph size that they can handle (for instance, the experimental studies in [16, 18] could not go beyond graphs of 10,000 vertices).

Algorithms to incrementally maintain APSDs for graphs stored in relational DBMSs have been proposed in [50]. These algorithms improve on [39] (which addresses the more general view maintenance problem in relational databases) by avoiding unnecessary joins and tuple deletions. However, [50] can deal only with the APSD maintenance problem while [39] works with general views (both in SQL and Datalog).

To the best of our knowledge, [50] is the only disk-based approach for maintaining APSDs (incremental algorithms to find a path between two vertices in disk-resident route collections have been proposed in [7], but paths are not required to be shortest ones). We are not aware of disk-based approaches to incrementally maintain APSPs. [50] is the closest work to ours, but with significant differences.

First, our algorithms maintain both shortest paths and shortest distances, while [50] is able to maintain shortest distances only—knowing the actual (shortest) paths is necessary in different applications.

Second, while both [50] and our algorithms proceed by first identifying "affected" shortest paths (i.e., those whose distance might need to be updated after the graph is changed) and then acting on them, the two approaches differ in *how* this is done. Specifically, the strategy employed by our approach to identify affected shortest paths is different and more effective. For both edge insertions and deletions, we have two distinct phases, one looking forward and the other looking backward. This allows us to filter out earlier shortest paths that do not play a role in the maintenance process. On the other hand, [50] identifies affected shortest paths in a more blind way: for both edge insertions and deletions, all shortest distances starting at the inserted/deleted edge are combined with all shortest distances ending in the inserted/deleted edge, leading to computationally expensive joins. Moreover, in our deletion algorithm, the recomputation of deleted shortest distances is done in an incremental way, as opposed to [50], which performs this step by combining all shortest distances that are left after the deletion phase.

Third, as shown in our experimental evaluation (cf. Section 3.5), our algorithms are significantly faster than those of [50].

The algorithms proposed in this chapter generalize the ones presented in [35] in that the former are able to maintain both shortest paths and distances, while the latter maintain shortest distances only.

3.3 Preliminaries

In this section, we introduce the notation and terminology used in the rest of this chapter.

A *graph* is a pair (V, A), where V is a finite set of *vertices* and $A \subseteq V \times V$ is a finite set of pairs called *edges*. A graph is *undirected* if A is a set of *unordered* pairs; otherwise (edges are *ordered* pairs) it is *directed*. A (directed or undirected) *weighted graph* consists of a graph $G = (V, A)$ and a function $\varphi : A \to \mathbb{R}^+$ assigning a *weight* (or *distance*) to each edge in A.[1] We use $\varphi(u, v)$ to denote the weight assigned to edge (u, v) by φ.

A sequence $v_0, v_1, \ldots, v_n$ $(n > 0)$ of vertices of G is a *path* from v_0 to v_n iff $(v_i, v_{i+1}) \in A$ for every $0 \leq i \leq n-1$. The *weight* (or *distance*) of a path $p = v_0, v_1, \ldots, v_n$ is defined as $\varphi(p) = \sum_{i=0}^{n-1} \varphi(v_i, v_{i+1})$. Obviously, there can be multiple paths from v_0 to v_n, each having a distance. A path from v_0 to v_n with the lowest distance (over the distances of all paths from v_0 to v_n) is called a *shortest path* from v_0 to v_n (there can be multiple shortest paths from v_0 to v_n) and its distance is called the *shortest distance* from v_0 to v_n. We assume that the shortest distance from a vertex to itself is 0, and the shortest distance from a vertex v_0 to a distinct vertex v_n is not defined if there is no path from v_0 to v_n.

In the rest of the chapter we consider directed weighted graphs and call them simply graphs—the extension of the proposed algorithms to undirected graphs is trivial. Without loss of generality, we assume that graphs do not have self-loops, i.e., edges of the form (v, v) (the reason is that self-loops can be disregarded for the purpose of finding shortest distances).

We consider the case where graphs and shortest paths are stored in relational databases. Notice that this allows us to take advantage of full-fledged optimization techniques provided by relational DBMSs.

Specifically, the set of edges of a graph is stored in a ternary relation E containing a tuple (a, b, w) iff there is an edge in the graph from a to b with weight w. We call E an *edge relation*. Vertices without incident edges can be ignored for our purposes.

A relation SP of arity 4 is used to represent the shortest paths for all pairs of reachable vertices of the graph (obviously, for a pair of reachable vertices we keep track of only one shortest path). Specifically, a shortest path $v_0, v_1, \ldots, v_n$ $(n \geq 1)$ from v_0 to v_n with (shortest) distance d_{0n} is represented with the set of tuples $\{(v_0, v_i, d_{0i}, v_{i-1}) \mid 1 \leq i \leq n\}$ in SP, where d_{0i} denotes the shortest distance from v_0 to v_i. Clearly, it is always possible to build

[1] In this chapter, we consider positive real weights.

the whole path from v_0 to v_n by following the chain of predecessors in SP starting from v_n. Analogous approaches are employed by different algorithms for shortest path computation, such as the well-known Dijkstra's algorithms. Thus, a tuple (x, y, d, z) in SP can be read as follows: there is a shortest path from x to y with (shortest) distance d and z is the predecessor of y along the path. We also say that SP is *a shortest path relation for* E.

Notice that an edge relation can admit different shortest path relations (depending on which shortest path is stored for a pair of reachable vertices in case of multiple shortest paths for that pair). We assume we are given an arbitrary shortest path relation for an edge relation, and are interested in computing an arbitrary shortest path relation for the updated edge relation. More precisely, we consider the shortest path maintenance problem defined below.

Problem (All-Pairs Shortest Paths Maintenance). Given an edge relation E, a shortest path relation SP for E, and an edge e, compute a shortest path relation for $E \cup \{e\}$ (or $E \setminus \{e\}$).

The case where we want to compute a shortest path relation for $E \cup \{e\}$ (resp. $E \setminus \{e\}$) corresponds to the scenario where the original edge relation E is modified by adding a new edge (resp. deleting an edge). Indeed, we will also consider the case where the weight of an edge is updated; however, as shown in Section 3.4.3, our algorithms to deal with insertion and deletion can be used to address this case too. We are interested in solving the problem in an efficient *incremental* fashion, i.e., avoiding to compute the new shortest path relation from scratch.

Clearly, a solution to the all-pairs shortest paths maintenance problem immediately gives the all-pairs shortest distances.

In addition to the edge relation for a graph and the corresponding shortest path relation, our algorithms will use auxiliary relations of arity arity 4 to store tuples of the form (x, y, d, z), which we also call *distance tuples*, whose meaning is that there is a path from x to y with distance d, and z is the predecessor of y along the path.

We will use the relational algebra operators π (projection), $\bowtie$ (join), $\ltimes$ (left semi-join), $\times$ (Cartesian product), $\cup$ (union), and $\setminus$ (difference).

We will refer to the i-th attribute of a relation as $\$i$. For instance, the projection of a relation R on the first and third attribute is written as $\pi_{\$1,\$3} R$. We will use the generalized projection so that we can write expressions like $\pi_{a,\$1} R$, which is equivalent to $\{a\} \times \pi_{\$1} R$.

Given a tuple $t = (t_1, ..., t_n)$, the i-th element of t is denoted as $t[i]$. Given two tuples t_1 and t_2, we say that t_1 and t_2 are *similar*, denoted $t_1 \sim t_2$, iff $t_1[1] = t_2[1]$ and $t_1[2] = t_2[2]$. Intuitively, in our setting, two tuples are similar when they refer to (possibly different) paths between the same pair of vertices.

Below we define the operators *min*, *prune*, $\oplus$, $\ominus$, and $\sqcap$, which will be used in the proposed algorithms. Let R and S be relations containing distance tuples (and thus of arity 4).

The *min* operator is defined as follows:

$$min(R) = \{t \in R \mid t[1] \neq t[2] \wedge \nexists t' \in R \text{ s.t. } t' \sim t \wedge t'[3] < t[3]\}.$$

Thus, $min(R)$ returns all the distance tuples t in R with $t[1] \neq t[2]$ (i.e., t refers to a path whose endpoints are distinct vertices) and s.t. R does not contain a similar distance tuple with a strictly lower distance.

The binary operator *prune* is defined as follows:

$$prune(R, S) = \{t \in R \mid \nexists t' \in S \text{ s.t. } t' \sim t \wedge t'[3] \leq t[3]\}.$$

Thus, $prune(R, S)$ returns all the distance tuples t in R for which there is no similar distance tuple in S with a lower distance.

The binary operator $\oplus$ is defined as follows:

$$R \oplus S = R \cup \{t \in S \mid \nexists t' \in R \text{ s.t. } t' \sim t\}.$$

Thus, $R \oplus S$ returns a relation obtained by adding to R the distance tuples of S that are not similar to any of the distance tuples in R.

The binary operator $\ominus$ is defined as follows:

$$R \ominus S = R \setminus \{t \in R \mid \exists t' \in S \text{ s.t. } t' \sim t \wedge t'[3] = t[3]\}.$$

Thus, $R \ominus S$ returns the set obtained from R by deleting every distance tuple for which there exists a similar distance tuple in S with the same distance.

Finally, the binary operator $\sqcap$ is defined as follows:

$$R \sqcap S = \{t \in R \mid \exists t' \in S \text{ s.t. } t' \sim t \wedge t[3] = t'[3]\}.$$

Thus, $R \sqcap S$ returns all the distance tuples of R having a similar distance tuple in S with the same distance.

Notice that none of the above binary operators is symmetric. Also, all the operators above can be expressed in the relational algebra as well as in SQL.

3.4 Incremental Maintenance of All-Pairs Shortest Paths

In this section, we present algorithms for the incremental maintenance of all-pairs shortest paths (as already mentioned, they immediately provide all-pairs shortest distances too).

We first propose an algorithm to handle edge insertions (Section 3.4.1) and then address edge deletions (Section 3.4.2). After that, we show how such algorithms can be used to handle edge updates too (Section 3.4.3).

It is worth noting that insertions and deletions of *vertices* can be straightforwardly reduced to our setting and thus be handled by our algorithms too: vertex insertions (resp. deletions) are handled by inserting (deleting) all edges that are incident from/to the inserted (resp. deleted) vertices. *Therefore, our algorithms can handle arbitrary sequences of edge insertions/deletions/updates and vertex insertions/deletions.*

Algorithm 3 Edge-Insertion-Maintenance (EIM)

Input: Edge relation E,
Shortest path relation SP,
Edge $e=(a,b,w)$ s.t. $\nexists(a,b,d,z) \in SP$ with $d \leq w$.
Output: Shortest path relation for $E \cup \{e\}$.

1: $\Delta P_f = \pi_{a,\$2,\$3+w,\$4}(\sigma_{\$1=b} SP)$;
2: $\Delta SP_f = prune(min(\Delta P_f), SP)$;
3: $\Delta P_\ell = \pi_{\$1,b,\$3+w,a}(\sigma_{\$2=a} SP)$;
4: $\Delta SP_\ell = prune(min(\Delta P_\ell), SP)$;
5: $\Delta P_i = \pi_{\$1,\$6,\$3+\$7-w,\$8}(\Delta SP_\ell \times \Delta SP_f)$;
6: $e' = (a,b,w,a)$;
7: $\Delta SP = prune(min(\Delta SP_\ell \cup \Delta P_i \cup \Delta SP_f \cup \{e'\}), SP)$;
8: $SP^* = \Delta SP \oplus SP$;
9: **return** SP^*;

3.4.1 Edge Insertion

Algorithm 3 below deals with edge insertions. We point out that the algorithm (as well as Algorithm 2 presented in Section 3.4.2) is written in a form that eases presentation, without applying optimizations to relational algebra expressions. However, relational DBMSs have full-fledged query optimization techniques to easily optimize the code—indeed, this is one of the advantages of relying on a relational DBMS.

Given a shortest path relation SP for an edge relation E, and an edge $e = (a,b,w)$ to be added to E, Algorithm 3 computes a shortest path relation for $E \cup \{e\}$. The precondition $\nexists(a,b,d,z) \in SP$ with $d \leq w$ is imposed just because if it does not hold, then the insertion of e has no effect on the shortest path relation, and thus there is no need to recompute it.

The algorithm performs the following four steps.

1. *(Forward step).* First, the algorithm looks at paths (in the new graph) having e as the first edge to see if new paths improving on the current ones can be obtained (lines 1–2).
2. *(Backward step).* Then, the algorithm looks at paths (in the new graph) having e as the last edge to see if new paths improving on the current ones can be obtained (lines 3–4).
3. *(Combination step).* After that, the algorithm "combines" the paths obtained from the forward and backward steps (line 5)—more details on how the combination is done are provided in the following. The aim is to build paths having e as an intermediate edge, and that might improve on the current ones.
4. *(Final step).* Finally, a path consisting only of the inserted edge is built (line 6), and all the paths built so far that improve on the original ones are incorporated into the shortest path relation (lines 7–9).

We now go into the details of each of the steps above.

(Forward step). First, the algorithm computes the set ΔP_f of all the distance tuples obtained by concatenating e with shortest paths starting from vertex b (line 1), that is, tuples of the form (a, y, d, z) s.t. there is a path from a to y with distance d (in the updated graph), the predecessor of y in the path is z, the first edge along the path is e, and the path from b to y is a shortest one (w.r.t. the original edge relation).

Then, among these distance tuples, the algorithm selects only those that improve on current shortest distances, that is, those tuples (a, y, d, z) in ΔP_f s.t. either there is no shortest path from a to y in SP or there is one with distance greater than d (line 2).

(Backward step). Next, two analogous steps are performed. First, shortest paths ending in vertex a are concatenated with e (line 3), yielding a set ΔP_ℓ of tuples of the form (x, b, d, a) s.t. there is a path from x to b with distance d (in the updated graph), the predecessor of b in the path is a, the last edge along the path is e, and the path from x to a is a shortest one (w.r.t. the original edge relation). Then, among the tuples in ΔP_ℓ, the algorithm selects only those that improve on the shortest distances in SP (line 4).

(Combination step). Then, the distance tuples obtained at lines 2 and 4 (which correspond to path whose first or last edge is e, respectively) are combined via a Cartesian product (line 5). Specifically, this step computes a relation ΔP_i by combining each tuple (x, b, d_1, z_1) in ΔSP_ℓ with each tuple (a, y, d_2, z_2) in ΔSP_f so as to get a tuple $(x, y, d_1 + d_2 - w, z_2)$. Notice that the distance of tuples in ΔP_i is diminished of w because e is taken into account both in tuples of ΔSP_ℓ and in tuples of ΔSP_f.

(Final step). Then, a distance tuple e' representing the path that consists only of e is built (line 6). Finally, the algorithm selects those tuples in $\Delta SP_\ell \cup \Delta P_i \cup \Delta SP_f \cup \{e'\}$ that improve on the shortest paths in SP (line 7) and incorporates them into SP (line 8).

The following example illustrates how Algorithm 3 works.

Example 3.1. Consider the graph in Figure 3.1 (top). A shortest path relation SP for the graph is represented in Figure 3.1 (bottom) as follows: an edge from x to y labeled with (d, z) means that there is a tuple (x, y, d, z) in SP. Thus, for instance, the edge from b to a with label $(2, c)$ means that $(b, a, 2, c) \in SP$, that is, there is a shortest path from b to a with distance 2 and c is the predecessor of a along that path.

Suppose we add the edge $(a, d, 1)$ to the graph.

Figure 3.2 shows the path computed by the forward step (dotted edge), that is, we have $\Delta SP_f = \{(a, e, 2, d)\}$.

The paths computed by the backward step are shown in Figure 3.3 (dashed edges), that is, we have $\Delta SP_\ell = \{(c, d, 2, a), (b, d, 3, a)\}$.

Figure 3.4 shows the paths computed by the combination step (dotted-and-dashed edges), that is, we have $\Delta P_i = \{(c, e, 3, d), (b, e, 4, d)\}$.

As all the distance tuples computed in the previous steps correspond to shortest paths (in the updated graph) improving on the original shortest paths,

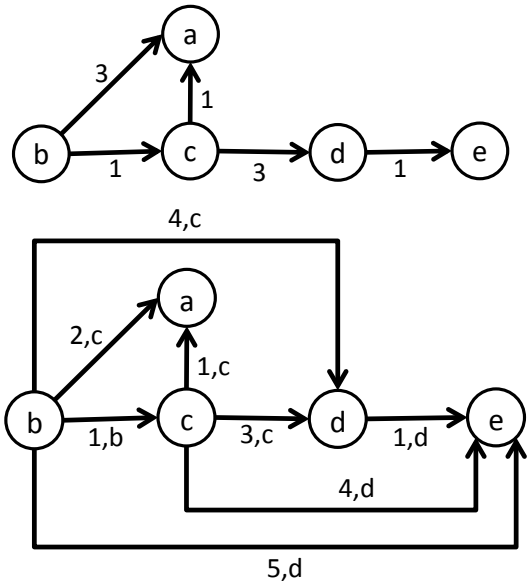


Fig. 3.1: A graph (top) and its shortest paths (bottom).

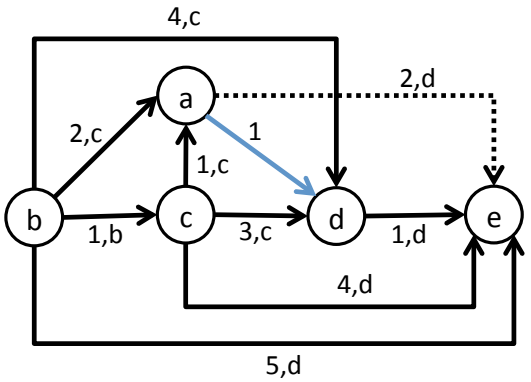


Fig. 3.2: Forward step of Algorithm 3.

they are incorporated into the shortest path relation at the final step. Notice that also $(a, d, 1, a)$ *is incorporated into the shortest path relation, as* d *was not reachable from* a *in the original graph.*

Figure 3.5 shows the updated graph and the new shortest path relation.

The following theorem states the correctness of Algorithm 3.

Theorem 3.2. *Given an edge relation* E*, a shortest path relation* SP *for* E*, and an edge* e*, Algorithm 3 computes a shortest path relation for* $E \cup \{e\}$.

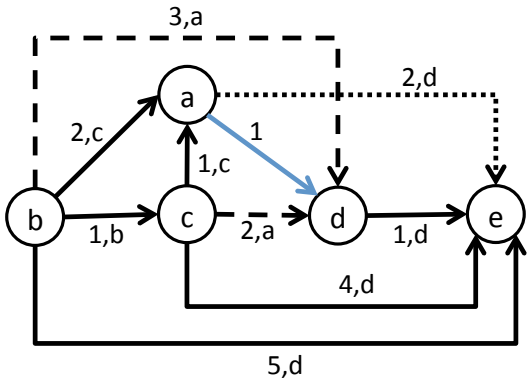


Fig. 3.3: Backward step of Algorithm 3.

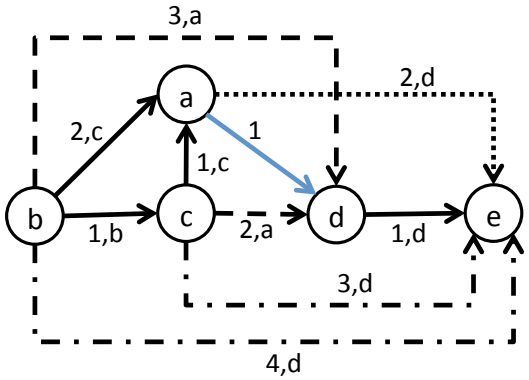


Fig. 3.4: Combination step of Algorithm 3.

Proof. Let $E_n = E \cup \{e\}$ and SP_n be a shortest path relation for E_n. Before proving the claim (i.e., showing that $SP^* = SP_n$) we make the following two observations, which will be used later on in the proof.

First, if a distance tuple (x, y, d, z) is in SP^* (resp. ΔP_f, ΔSP_f, ΔP_ℓ, ΔSP_ℓ, ΔP_i), then there is a path from x to y in E_n whose distance is d and where z is the predecessor of y along the path. Indeed, this property holds also for ΔP_f, ΔSP_f, ΔP_ℓ, ΔSP_ℓ, ΔP_i, and ΔSP.

Second, SP^* does not contain two distance tuples (x, y, d_1, z_1) and (x, y, d_2, z_2) s.t. $d_1 \neq d_2$ or $z_1 \neq z_2$ (to see why, it suffices to look at lines 7–8 and the definitions of *min*, *prune* and $\oplus$).

In the following, whenever the fourth element of a distance tuple (x, y, d, z) is irrelevant, we simply write the distance tuple as $(x, y, d, -)$.

Soundness ($SP^* \subseteq SP_n$). Let $(x, y, d, z) \in SP^*$. As noticed above, this means that there is a path from x to y in E_n; thus, there must be a shortest

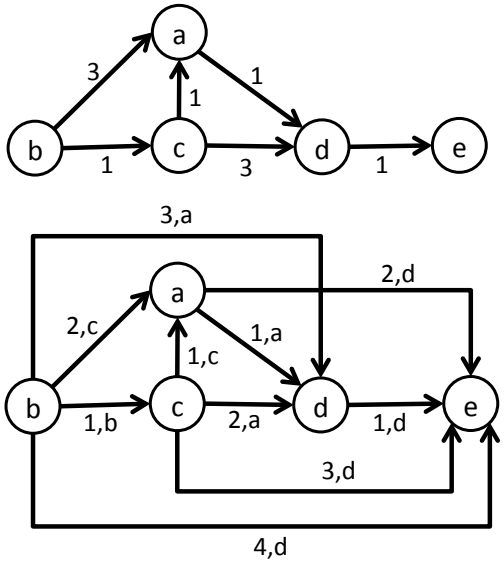


Fig. 3.5: Updated graph (top) and its shortest paths (bottom).

one too, which implies that a distance tuple $(x, y, d', -)$ is in SP_n. We show that $d = d'$. One of the following two cases must occur.

1. There is a shortest path p' from x to y in E_n (its distance is d') which does not go through e. Since p' goes only through edges in E, then there is a distance tuple (x, y, d', z') in SP. Since ΔSP contains distance tuples corresponding to paths in E_n that strictly improve on shortest distances in SP (line 7), then there is no distance tuple $(x, y, d'', -)$ in ΔSP. Since $SP^* = \Delta SP \oplus SP$ (line 8), then $(x, y, d', z') \in SP^*$. Hence, $d = d'$ (and $z = z'$).
2. Every shortest path from x to y in E_n (whose distance is d') goes through e. Let p' be one of such shortest paths. Then, e is either *(i)* the first edge of p', or *(ii)* the last edge of p', or *(iii)* an intermediate edge of p'. Notice that in case *(i)* the subpath of p' that goes from b to y is a shortest path in E_n and also in E (because p' does not go though e twice); in case *(ii)* the subpath of p' that goes from x to a is a shortest path in E_n and also in E; in case *(iii)* the subpath of p' that goes from x to a and the subpath of p' that goes from b to y are shortest paths in E_n and also in E. Now it is easy to see that a distance tuple for p' is computed at line 1 (resp. 3 and 5) when case *(i)* (resp. *(ii)* and *(iii)*) occurs. In particular, in case *(iii)*, since every shortest path from x to y in E_n goes through e, it must be the case that every shortest path from x to b in E_n has e as last edge, and every shortest path from a to y in E_n has e as first edge, and thus a distance tuple for p' is computed at line 5. Notice that since every shortest

path from x to y in E_n goes through e, this is the case where the insertion of e strictly improves the shortest distance from x to y. Thus, a distance tuple $(x, y, d', -)$ belongs to ΔSP (line 7) and SP^* (line 8). Hence, $d = d'$.

Completeness ($SP^* \supseteq SP_n$). Consider a distance tuple (x, y, d, z) in SP_n. If there is a shortest path p from x to y in E_n (its distance is d) that does not go through e, then $(x, y, d, z) \in SP$. This means that ΔSP does not contain a distance tuple $(x, y, d', -)$ because distance tuples in $\Delta SP_\ell \cup \Delta P_i \cup \Delta SP_f \cup \{e\}$ correspond to paths in SP_n and thus do not strictly improve on p (see line 7). Hence, $(x, y, d, z) \in SP^*$ (see line 8). If every shortest path from x to y in E_n goes through e, then it can be verified that $(x, y, d, z) \in SP^*$ by applying the same reasoning used in part *(2)* above. □

3.4.2 Edge Deletion

We now turn our attention to edge deletions, which are handled by Algorithm 2 below. It consists of two phases: a *deletion phase* (lines 5–20), which deletes from SP shortest paths that might go through the deleted edge; and a *recalculate phase* (lines 21–28), which recomputes the new shortest paths (if any) for the pair of vertices deleted in the first phase.

The precondition is imposed because if it does not hold, that is there is no distance tuple of the form (a, b, w, z) in SP, then there is a path from a to b not using e and with a distance strictly lower than w, which means that the deletion of e does not affect the shortest path relation, and thus there is no need to recompute it (recall that we consider positive weights).

First of all, the algorithm checks whether there is an alternative path in the updated graph from a to b with distance w (lines 1–2). If so, the distance tuple in SP for the shortest path from a to b is updated into (a, b, w, z'), where z' is the predecessor of b along the alternative path (line 3). Since there is an alternative path from a to b with the same distance as the weight of the deleted edge, the remaining shortest paths are not affected by the deletion, and the algorithm terminates (line 4).

If no alternative path is found, handling the edge deletion is much more involved, and the algorithm proceeds with the two aforementioned phases as follows.

Deletion phase. Similar to Algorithm 3, the deletion phase consists of different steps:

1. a *forward step*, looking at paths having e as the first edge (lines 5–9);
2. a *backward step*, looking at paths having e as the last edge (lines 10–14);
3. a *combination step*, combining paths of the previous two steps, whose aim is to look at paths having e as an intermediate edge (lines 15–18); and
4. a *deletion step*, (lines 19–20), where paths computed at the previous steps are deleted from the shortest path relation.

We now go into the details of each of the steps above.

Input: Edge relation E,
Shortest path relation SP,
Edge $e{=}(a, b, w)$ s.t. $\exists z.(a, b, w, z) \in SP$.
Output: Shortest path relation for $E_n = E \setminus \{e\}$.

1: $AP{=}\pi_{\$1,\$5,\$3+\$6,\$7}(\sigma_{\$1=a}E_n \underset{\$2=\$1}{\bowtie} \sigma_{\$2=b}SP)$;
2: **if** $\exists(a, b, w, z') \in AP$ **then**
3: $SP^* = \{(a, b, w, z')\} \oplus SP$;
4: **return** SP^*;
5: $\Delta P_f = \pi_{a,\$2,\$3+w,\$4}(\sigma_{\$1=b}SP)$;
6: $AP_f{=}\pi_{\$1,\$2,\$3,\$1}(\sigma_{\$1=a}E_n) \cup (\pi_{\$1,\$5,\$3+\$6,\$7}(\sigma_{\$1=a}E_n \underset{\$2=\$1}{\bowtie} SP))$;
7: $\Delta SP_f = (SP \sqcap \Delta P_f) \ominus AP_f$;
8: $NP_f = (AP_f \sqcap (SP \sqcap \Delta P_f))$;
9: $SP = NP_f \oplus SP$;
10: $\Delta P_\ell = \sigma_{\$2=b \wedge \$4=a}SP$;
11: $AP_\ell{=}\pi_{\$1,\$2,\$3,\$1}(\sigma_{\$2=b}E_n) \cup (\pi_{\$1,\$6,\$3+\$7,\$5}(SP \underset{\$2=\$1}{\bowtie} \sigma_{\$2=b}E_n))$;
12: $\Delta SP_\ell = \Delta P_\ell \ominus AP_\ell$;
13: $NP_\ell = (AP_\ell \sqcap \Delta P_\ell)$;
14: $SP = NP_\ell \oplus SP$;
15: $NP_i = (\pi_{\$1,\$6,\$3+\$7,\$8}((\sigma_{\$2=a}SP) \underset{\$2=\$1}{\bowtie} NP_f)) \sqcap SP$;
16: $SP = NP_i \oplus SP$;
17: $\Delta P_i = \pi_{\$1,\$6,\$3+\$7-w,\$8}(\Delta SP_\ell \times \Delta SP_f)$;
18: $\Delta SP_i = SP \sqcap \Delta P_i$;
19: $SP^- = \Delta SP_\ell \cup \Delta SP_f \cup \Delta SP_i \cup \{(a, b, w, z)\}$;
20: $SP = SP \setminus SP^-$;
21: $\Delta P{=}(\pi_{\$1,\$2,\$3,\$1}E_n \cup (\pi_{\$1,\$5,\$3+\$6,\$7}(E_n \underset{\$2=\$1}{\bowtie} SP))) \underset{\substack{\$1=\$1 \wedge \\ \$2=\$2}}{\ltimes} SP^-$;
22: $\Delta SP = min(\Delta P)$;
23: $SP^+ = \Delta SP$;
24: **while** $\Delta SP \neq \emptyset$ **do**
25: $\Delta P = \pi_{\$1,\$5,\$3+\$6,\$7}(E_n \underset{\$2=\$1}{\bowtie} \Delta SP) \underset{\$1=\$1 \wedge \$2=\$2}{\ltimes} SP^-$;
26: $\Delta SP = prune(min(\Delta P), SP^+)$;
27: $SP^+ = \Delta SP \oplus SP^+$;
28: $SP^* = SP \cup SP^+$;
29: **return** SP^*;

(Forward step). First, the algorithm computes all the distance tuples obtained by concatenating e with shortest paths starting from vertex b (line 5), that is, tuples of the form (a, y, d, z) s.t. there is a path from a to y with distance d, the first edge along such a path is e, the path from b to y is a shortest one (w.r.t. the original edge relation), and z is the predecessor of y. Such distance tuples are stored in ΔP_f—intuitively, they correspond to paths from a, using e as the first edge, and that might be affected by the deletion of e.

Then, alternative paths in the new graph from a to other vertices are computed and stored in AP_f (line 6).

After that, the distance tuples in ΔP_f that correspond to shortest paths (w.r.t. the original graph) and for which there are no alternative paths (in AP_f) with the same distance are stored in ΔSP_f (line 7). These distance tuples will be later deleted from SP.

On the other hand, the distance tuples in ΔP_f corresponding to shortest paths (w.r.t. the original graph) and for which there is an alternative path with the same distance are incorporated into SP (lines 8–9). This is done to properly update the predecessor of the destination vertex (i.e., the last element of the distance tuples).

(Backward step). Then, shortest paths having e as the last edge are similarly processed. Specifically, the algorithm selects all the distance tuples corresponding to shortest paths (w.r.t. the original edge relation) having b as destination vertex and e as the last edge (line 10). Such distance tuples are stored in ΔP_ℓ and correspond to paths to b, using e as the last edge, and that might be affected by the deletion of e.

After that, alternative paths to b (from other vertices) in the new graph are computed and stored in AP_ℓ (line 11).

Then, the distance tuples in ΔP_ℓ (recall that they correspond to shortest paths in the original graph) for which there is no alternative path (in AP_ℓ) with the same distance are stored in ΔSP_ℓ (line 12)—they will be later deleted from SP.

On the other hand, the distance tuples in ΔP_ℓ for which there is an alternative path with the same distance are incorporated into SP (lines 13–14). This is done to properly update the predecessor of the destination vertex (i.e., the last element of the distance tuples).

(Combination step). Then, the algorithm computes distance tuples obtained as the concatenation of distance tuples in SP ending in a with distance tuples in NP_f (the latter correspond to shortest paths from a). Among them, only those that correspond to shortest paths are kept in NP_i (line 15) and incorporated into SP (line 16). This step is done so that shortest paths in the original graph that possibly had e as an intermediate edge are replaced by alternative paths with the same distance in the updated graph.

After that, the distance tuples in ΔSP_ℓ computed in line 12 (corresponding to paths where the last edge is e) are combined with the tuples in ΔSP_f computed in line 7 (corresponding to paths where the first edge is e) via a Cartesian product (line 17). Analogous to line 5 of Algorithm 3, since edge e is taken into account twice, the distance of the tuples obtained from the Cartesian product is diminished of w. Among the so-obtained distance tuples, only those that correspond to shortest paths are kept (line 18).

(Deletion step). The distance tuples computed in lines 7, 12 and 18, together with the distance tuple (a, b, w, z), are stored in relation SP^- (line 19) and deleted from SP (line 20).

Recalculate phase. The second phase of Algorithm 2 (lines 21–28) computes the new shortest paths (when they exist) for the endpoints of the distance tuples in SP^-, and eventually adds them to SP.

More specifically, the algorithm first adds to E_n all the distance tuples obtained by concatenating edges in E_n with shortest paths in the updated shortest path relation SP (line 21). Indeed, only those distance tuples whose endpoints are in a tuple of SP^- are kept and stored in ΔP. Among these distance tuples, the minimum ones are selected and stored in ΔSP (line 22). These distance tuples are then added to SP^+ (line 23), which is the set that will be eventually added to SP.

Then, SP^+ is iteratively updated as follows (lines 24–27). The algorithm computes all the distance tuples obtained by concatenating edges in E_n with tuples in ΔSP, and keeps only those whose endpoints are in a tuple of SP^- (line 25). Among these, only distance tuples that improve on shortest distances in SP^+ are kept (line 26) and incorporated into SP^+ (line 27). When no more better distance tuples can be obtained, SP^+ is added to SP (line 28) and the result is returned (line 29).

It is important to notice that this way of computing the new shortest paths allows the algorithm to prune the search space, as computed paths that are not shortest ones are disregarded and no further explored.

An example illustrating how Algorithm 2 works is provided below.

Example 3.3. Consider the graph of Figure 3.6 (top) and the shortest path relation SP in Figure 3.6 (bottom). Once again, the shortest path relation is represented as follows: an edge from x to y labeled with (d, z) means that there is a tuple (x, y, d, z) in SP, that is, there is a shortest path from x to y with distance d and z is the predecessor of y.

Suppose we delete the edge $(a, b, 1)$ from the graph.

Since there is no alternative path in the updated graph from a to b, then Algorithm 2 performs the deletion phase.

The forward step identifies the two dotted paths in Figure 3.7 as possibly affected by the edge deletion.

Indeed, for the path from a to d there is an alternative one with the same distance, namely the one going through e. Thus, the predecessor of d is updated into e (i.e., $(a, d, 2, b)$ is updated into $(a, d, 2, e)$), see Figure 3.8. Since there is no alternative path from a to c with distance 4, the path from a to c will be later deleted by the algorithm (i.e., $(a, c, 4, b) \in \Delta SP_f$).

Then, the backward step identifies the dashed path in Figure 3.9 as possibly affected by the edge deletion. Since there is no alternative path from f to b, such a path will be later deleted by the algorithm (i.e., $(f, b, 3, a) \in \Delta SP_\ell$).

After that, the combination step realizes that, for the path from f to d, the predecessor of d has to be updated into e (Figure 3.10).

Moreover, the combination step identifies the dashed-and-dotted path in Figure 3.11 as possibly affected by the deletion. Indeed, since there is no al-

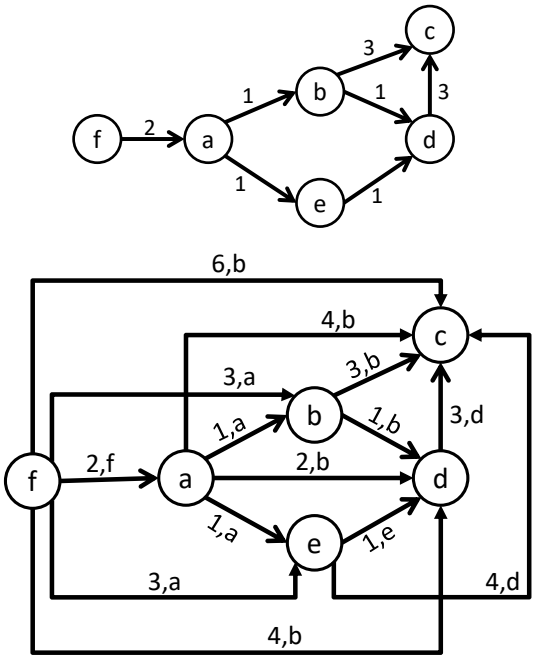


Fig. 3.6: A graph (top) and its shortest paths (bottom).

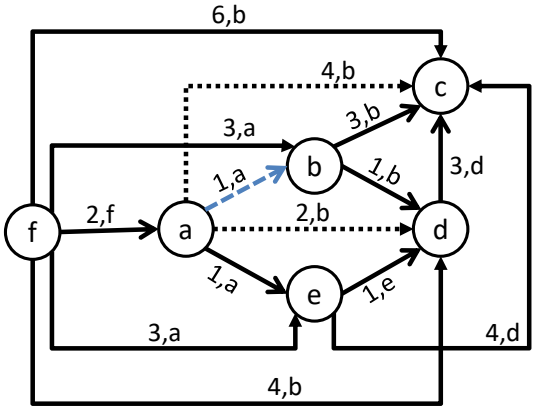


Fig. 3.7: Forward step of Algorithm 2 (1 of 2).

ternative path with the same distance, this path will be later deleted (i.e., $(f, c, 6, b) \in \Delta SP_i$).

In the deletion step of the deletion phase, all the non-solid paths in Figure 3.11 are deleted from the shortest path relation.

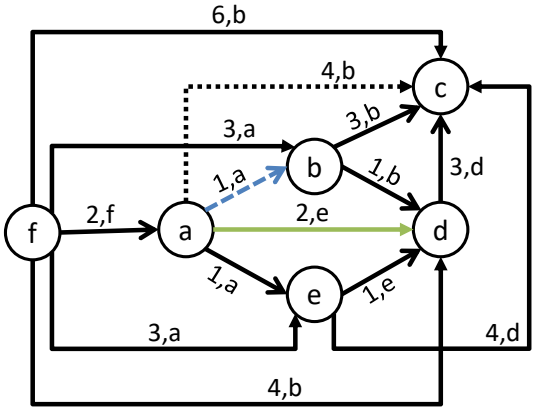


Fig. 3.8: Forward step of Algorithm 2 (2 of 2).

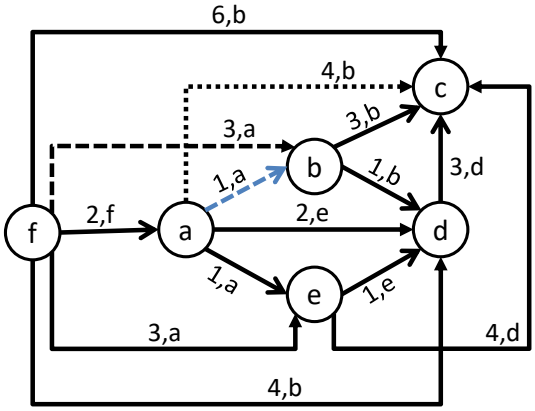


Fig. 3.9: Backward step of Algorithm 2.

Then, the algorithm tries to recompute the deleted shortest paths. Figure 3.12 shows the new shortest paths computed by the recalculate phase. At this point, Algorithm 2 terminates.

Figure 3.13 shows the shortest path relation for the updated graph.

The following theorem states the correctness of Algorithm 2.

Theorem 3.4. *Given an edge relation E, a shortest path relation SP for E, and an edge e, Algorithm 2 computes a shortest path relation for $E \setminus \{e\}$.*

Proof. Let $E_n = E - \{e\}$ and SP_n be a shortest path relation for E_n.

First of all, notice that each iteration of the **while** loop in lines 24–27 computes shortest paths that strictly improve on the currently computed ones. As we consider positive edge weights, the number of iterations is bounded by

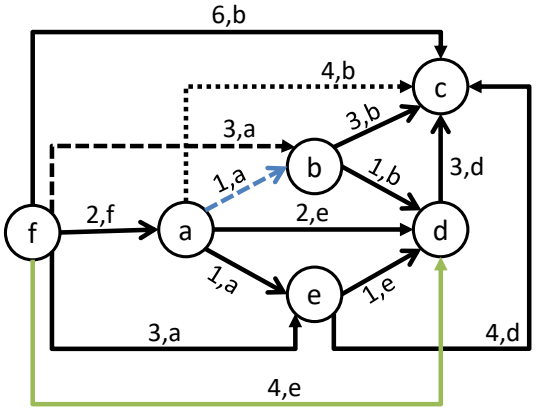


Fig. 3.10: Combination step of Algorithm 2 (1 of 2).

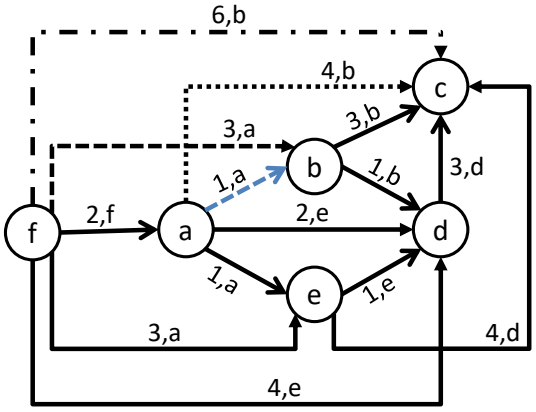


Fig. 3.11: Combination step of Algorithm 2 (2 of 2).

the number of simple paths (i.e., paths without multiple occurrences of the same vertex), which is finite, and thus the algorithm terminates.

In the following, whenever the fourth element of a distance tuple (x, y, d, z) is irrelevant, we simply write it as $(x, y, d, -)$. Likewise, we sometimes write $(x, y, -, -)$ in place of (x, y, d, z).

If a distance tuple (x, y, d, z) is in SP and there exists a distance tuple $(x, y, d, -)$ in SP_n, then there is a path from x to y with distance d in the updated graph. One of the following two cases occurs:

- None of the shortest paths from x to y in the original graph goes through e. Then, it is easy to see that (x, y, d, z) remains in SP and thus is in SP^*.
- At least one shortest path from x to y goes though e in the original graph. Since there is an alternative path with the same distance in the new graph, then (x, y, d, z) is updated into a distance tuple (x, y, d, z') either at line 9

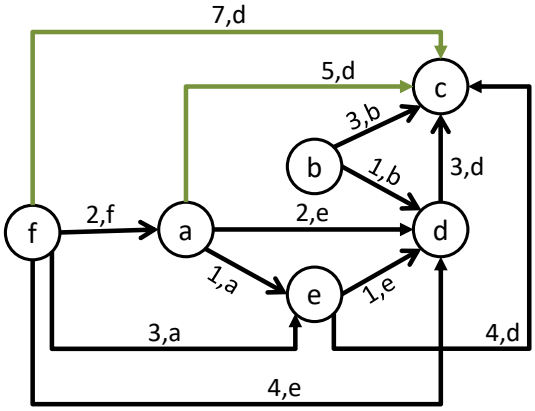


Fig. 3.12: Recalculate phase of Algorithm 2.

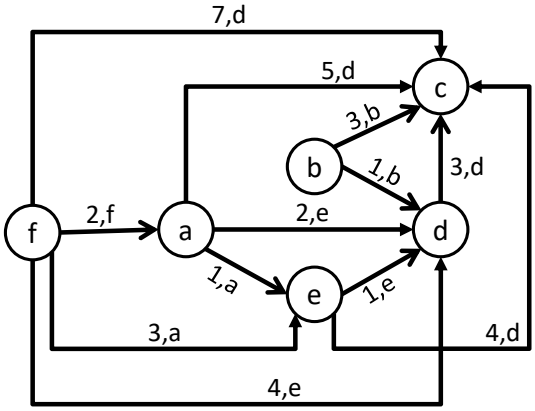


Fig. 3.13: Shortest paths of the updated graph.

(when an original shortest path from x to y has e as the first edge) or line 14 (when an original shortest path from x to y has e as the last edge) or line 16 (when an original shortest path from x to y has e as an intermediate edge).

Regarding distance tuples (x, y, d, z) is in SP s.t. there does not exists a distance tuple $(x, y, d, -)$ in SP_n, we now show the following property: if $(x, y, d, z) \in SP$ and there is no $(x, y, d, -)$ in SP_n, then $(x, y, d, z) \in SP^-$. Suppose $(x, y, d, z) \in SP$ and there is no $(x, y, d, -)$ in SP_n. Then, one of the following cases occurs.

(i) There is a shortest path from x to y in E whose first edge is e and there is no shortest path from x to y in E_n with distance d. In such a case $(x, y, d, z) \in \Delta SP_f$ (see lines 5–7) and thus $(x, y, d, z) \in SP^-$ (see line 19).

(ii) There is a shortest path from x to y in E whose last edge is e and there is no shortest path from x to y in E_n with distance d. In such a case $(x, y, d, z) \in \Delta SP_\ell$ (see lines 10–12) and thus $(x, y, d, z) \in SP^-$ (see line 19).

(iii) There is a shortest path from x to y in E where e is an intermediate edge and for the two subpaths starting from and ending in e conditions *(i)* and *(ii)* apply, respectively. In such a case $(x, y, d, z) \in \Delta SP_i$ (see lines 17–18) and thus $(x, y, d, z) \in SP^-$ (see line 19).

(iv) $e = (x, y, d, z)$. Then, $(x, y, d, z) \in SP^-$ (see line 19).

Because of the property above, if the shortest distance from x to y changes in the updated graph, then $(x, y, -, -)$ is deleted from SP. Then, it can be easily verified that a shortest path from x to y (if any) in the updated graph is correctly computed by the recalculate phase on lines 21–28. □

3.4.3 Edge Update

In this section, we show how our algorithms can be easily used to handle an edge update (i.e., the edge weight update).

Given an edge relation E, a shortest path relation SP for E, and an edge (a, b, w) whose weight w must be changed into w', we can compute a shortest path relation for $(E \setminus \{(a, b, w)\}) \cup \{(a, b, w')\}$ as follows:

1. if $w' < w$, then we call Algorithm EIM with input $E \setminus \{(a, b, w)\}$, SP, and (a, b, w');
2. if $w' > w$, then we call Algorithm with input $(E \setminus \{(a, b, w)\}) \cup \{(a, b, w')\}$, SP, and (a, b, w).

Thus, when the new weight w' "improves" on the old one w, we reduce the problem to an edge insertion, otherwise we reduce the problem to an edge deletion. As stated in the following theorem, this way of processing edge weight updates is correct.

Theorem 3.5. *Given an edge relation E, a shortest path relation SP for E, an edge $e = (a, b, w)$ in E, and a weight w', then*

1. *if $w' < w$, EIM$(E \setminus \{(a, b, w)\}, SP, (a, b, w'))$ returns a shortest path relation for $(E \setminus \{(a, b, w)\}) \cup \{(a, b, w')\}$;*
2. *if $w' > w$, EDM$((E \setminus \{(a, b, w)\}) \cup \{(a, b, w')\}, SP, (a, b, w))$ returns a shortest path relation for $(E \setminus \{(a, b, w)\}) \cup \{(a, b, w')\}$.*

Proof. Item 1 can be proved by proceeding in the same way as in the proof of Theorem 3.2. Likewise, Item 2 can be proved by proceeding in the same way as in the proof of Theorem 3.4. □

3.5 Experimental Evaluation

In this section, we report on an experimental evaluation we carried out to assess the validity of our approach.

3.5.1 Experimental Setup

Below we describe the algorithms and datasets used in the experimental evaluation.

Algorithms. We compared our algorithms against the ones proposed by Pang et al. [50], which are, to the best of our knowledge, the only disk-based approach for the incremental maintenance of all-pairs shortest distances—in particular, they rely on relational DBMSs too. Indeed, we also compared the algorithms presented in this chapter with our previous proposal [35], and the running times of the two approaches were nearly the same. Thus, in the following, we do not report the execution time of the algorithms proposed in [35]. Also, recall that while the algorithms presented in this chapter maintain both shortest paths and distances, the approach of [35] is able to maintain shortest distances only. To sum up, in this section we will consider the EIM algorithm (Algorithm 3), the EDM algorithm (Algorithm 2), and the insertion and deletion algorithms of [50] (denoted PDR).

Datasets. Experiments were carried out on the following real-world networks.

- **DIMES public data repository.**[2] This dataset provides monthly snapshots of Autonomous Systems on the Internet. Vertices represent Autonomous Systems and edges represent direct links between Autonomous Systems that were found for a given month. Since the original graphs were unweighted, we assigned unitary weight to every edge.
- **Road Network of North America (RNNA).**[3] This dataset is a road network of North America, thus edge weights stand for road lengths.
- **Twitter social network.**[4] This network contains information about who follows whom on Twitter. Vertices represents users and edges represent follow relationships. We assigned unitary weight to every edge, as the original graph was unweighted.
- **Gnutella peer-to-peer network.**[5] This dataset stores the Gnutella peer-to-peer file sharing network. Vertices represent hosts in the Gnutella network and edges represent connections between the Gnutella hosts. As the original graph was unweighted, we assigned unitary weight to every edge.
- **Instagram social network.**[6] This dataset was originally collected in 2014 and introduced in [60]. Vertices represents users and edges represent follow relationships. We assigned unitary weight to every edge (connecting a pair follower-followee).

[2] `http://www.netdimes.org/new/`
[3] `http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm`
[4] `http://konect.uni-koblenz.de/networks/munmun_twitter_social`
[5] `http://snap.stanford.edu/data/p2p-Gnutella31.html`
[6] `http://uweb.dimes.unical.it/tagarelli/data/`

The datasets' features are reported in Table 3.1. The number of shortest distances is the number of pairs of reachable vertices (which is also the number of stored shortest paths, as we store one shortest path for each pair of reachable vertices).

Datasets	# Vertices	# Edges	# Shortest distances
DIMES	17,148	53,833	$\approx 1.02 \times 10^8$
RNNA	175,813	179,179	$\approx 2.76 \times 10^8$
Twitter	465,017	834,797	$\approx 8.03 \times 10^8$
Gnutella	62,586	147,892	$\approx 8.84 \times 10^8$
Instagram	54,017	963,881	$\approx 9.3 \times 10^8$

Table 3.1: Properties of the considered datasets.

A couple of remarks on the size of the datasets are in order.

(1) As we deal with the problem of incrementally maintaining all-pairs shortest paths, the input of the algorithms consists of both a graph *and the corresponding shortest paths*[7], with the size of the latter being much bigger than the size of the former. The datasets we considered have up to hundreds of millions of shortest distances—indeed, the biggest dataset (namely, the Instagram network) has nearly one billion shortest distances.

(2) We also run the algorithms proposed in [16] and [46], which are state-of-the-art algorithms for the APSD maintenance working in the main memory, over the aforementioned datasets. We used the implementations of the algorithms provided by their authors, which work in the main memory. [16] run out of memory on all datasets, not being able to handle the graph, the shortest distances, and the employed data structures. [46] showed higher running times than our approach on the DIMES dataset, while run out of memory on the the remaining datasets.

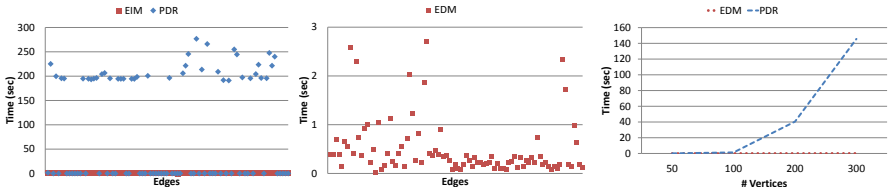
A couple of remarks on the size of the datasets are in order.

(1) As we deal with the problem of incrementally maintaining all-pairs shortest paths, the input of the algorithms consists of both a graph *and the corresponding shortest paths*[8], with the size of the latter being much bigger than the size of the former. The datasets we considered have up to hundreds of millions of shortest distances—indeed, the biggest dataset (namely, the Instagram network) has nearly one billion shortest distances.

(2) We also run the algorithms proposed in [16] and [46], which are state-of-the-art algorithms for the APSD maintenance working in the main memory, over the aforementioned datasets. We used the implementations of the algorithms provided by their authors, which work in the main memory. [16] run

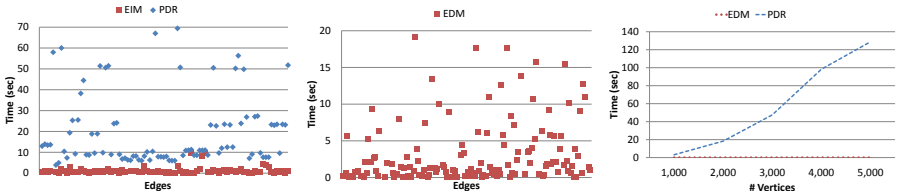
[7] Clearly, the input of PDR consists of a graph and the corresponding shortest distances, rather than the actual paths.

[8] Clearly, the input of PDR consists of a graph and the corresponding shortest distances, rather than the actual paths.



(a) Entire dataset – Insertion. (b) Entire dataset – Deletion. (c) Induced subgraphs – Deletion.

Fig. 3.14: DIMES Dataset.



(a) Entire dataset – Insertion. (b) Entire dataset – Deletion. (c) Induced subgraphs – Deletion.

Fig. 3.15: RNNA Dataset.

out of memory on all datasets, not being able to handle the graph, the shortest distances, and the employed data structures. [46] showed higher running times than our approach on the DIMES dataset, while run out of memory on the the remaining datasets.

Indeed, it was already as pointed out in [16, 18] that memory usage is an important issue of current approaches, limiting substantially the graph size that they can handle (for instance, the experimental studies in [16, 18] could not go beyond graphs of 10,000 vertices). Moreover, as shown in the following, the datasets are already too large for PDR, but can be efficiently maintained by our algorithms.

All experiments were run on an Intel i7 3770K 3.5 GHz, 12 GB of memory, running Linux Mint 17.1 and MySQL 5.5.43.

3.5.2 Results on the DIMES Dataset

In this section, we discuss the experimental results for the DIMES dataset. As the DIMES dataset is an evolving graph and its repository provides different snapshots of it, we could run experiments with *real* edge insertions/deletions—we considered the first two snapshots (namely, January and February 2007).

Runtimes for the insertion of 90 edges are reported in Figure 3.14a. PDR shows an unstable behavior, as its running times are around 200 secs (i.e.,

three orders of magnitude slower than EIM) for about half of the edges, and are similar to EIM for the remaining edges.

Results for the deletion of 90 edges are reported in Figure 3.14b. PDR is not reported as it did not terminate within one hour in all cases. On the other hand, we can see that EDM performs very well.

In order to allow PDR to answer in a reasonable amount of time and compare its behavior with EDM, we considered smaller induced subgraphs of the original DIMES graph. An *induced subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ contains all edges of E connecting two vertices in V'. To generate these graphs we started from a randomly chosen vertex in the DIMES dataset and visited connected vertices in a breadth-first fashion, adding to the graph in construction all edges of the original graph involving vertices so far encountered, until the desired size was reached.

Runtimes for both PDR and EDM are reported in Figure 3.14c (each running time is the average time over 10 edges), which shows that PDR becomes impractical very quickly.

3.5.3 Results on the RNNA Dataset

We now discuss the experimental results for the Road Network of North America.

Figure 3.15a shows the execution times for the insertion of 150 edges (randomly chosen). Algorithm EIM outperforms PDR being always faster with a difference from one to three orders of magnitude—in most of the cases the difference is at least two orders of magnitude.

Figure 3.15b shows the execution times for the deletion of 150 edges (randomly chosen). Like for the DIMES dataset, execution times for PDR are not reported as no answer was given within one hour in all cases. The EDM algorithm performs quite well being able to handle deletion in reasonable time.

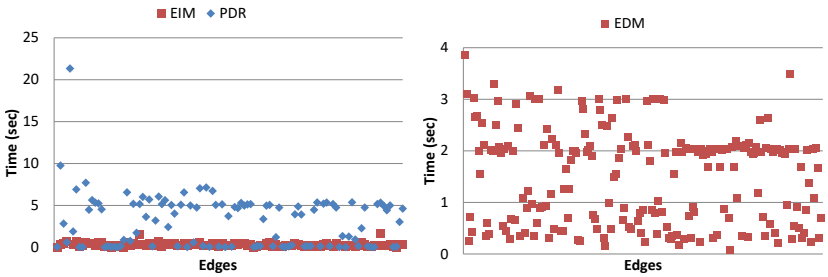
Once again, to compare PDR with EDM, we also considered smaller induced subgraphs of the original RNNA network, generated in the same way as for the DIMES dataset. However, in this case, the starting vertex was one with highest degree (this choice was not possible for the DIMES dataset because the resulting subgraphs were too large to be handled by PDR). As shown in Figure 3.15c, PDR's running times quickly diverge from those of EDM.

3.5.4 Results on the Twitter Dataset

In this section, we consider the Twitter network. Figures 3.16a and 3.16b show running times for the insertion and deletion of 200 randomly chosen edges, respectively.

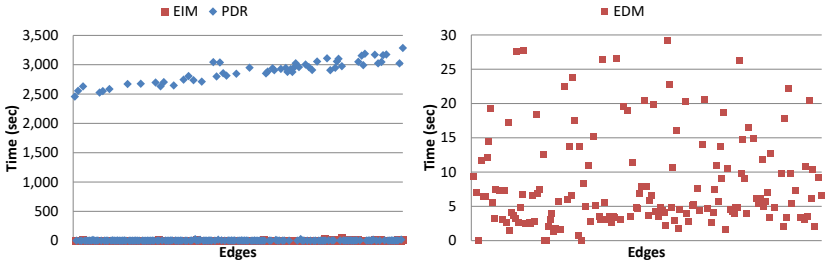
For insertion, EIM is faster than PDR by one order of magnitude for over 90% of the edges, while for remaining ones they have similar running times.

Regarding deletion, EDM was very fast, while PDR did not provided an answer within one hour.



(a) Entire dataset – Insertion. (b) Entire dataset – Deletion.

Fig. 3.16: Twitter Dataset.



(a) Entire dataset – Insertion. (b) Entire dataset – Deletion.

Fig. 3.17: Gnutella Dataset.

3.5.5 Results on the Gnutella Dataset

The running times for the insertion and the deletion of 200 randomly chosen edges over the Gnutella network are reported in Figures 3.17a and 3.17b, respectively.

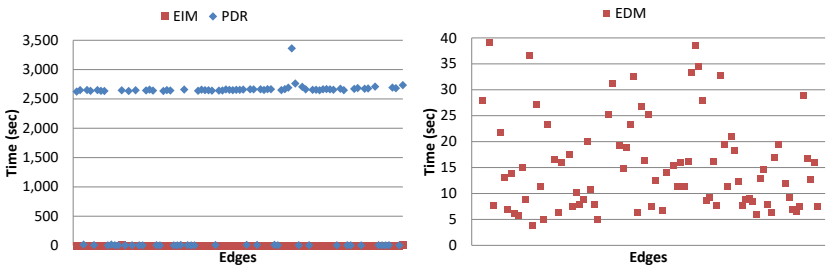
Similar to the DIMES Dataset, PDR shows an unstable behavior, as its running time is over 2500 secs (i.e., PDR is four orders of magnitude slower than EIM) for over one third of the edges, and is similar to EIM's running time for the remaining edges.

As for deletion, PDR did not terminate within one hour in all cases, while EDM has good performances.

3.5.6 Results on the Instagram Dataset

We now discuss the experimental results over the biggest network we considered, namely Instagram, which has nearly one billion shortest distances.

Figures 3.18a and 3.18b show running times for the insertion and deletion of 100 randomly chosen edges, respectively.



(a) Entire dataset – Insertion. (b) Entire dataset – Deletion.

Fig. 3.18: Instagram Dataset.

As for insertion, the difference between PDR and EIM is significant, with EIM being four orders of magnitude faster in most of the cases.

Regarding deletion, like for the other datasets, PDR was not able to answer within one hour, while EDM has good running times.

3.5.7 Experimental conclusions

From the experimental results reported in this section, we can draw the following conclusions.

We run state-of-the-art algorithms working in the main memory (namely, [46, 16]) over five real-world datasets and they run out of memory in all cases. This suggests the need of resorting to disk-based approaches for APSD and APSP maintenance.

To the best of our knowledge, [50] is the state-of-the-art approach for the all-pairs shortest distances maintenance problem working on the secondary memory (in particular, graphs are stored in relational DBMSs as in our approach). In all cases, our algorithms notably outperform the algorithms proposed in [50], being able to handle both insertions and deletions with very good performances over networks with hundreds of millions of shortest distances—indeed, [50] was not able to handle any deletion within one hour for every dataset.

3.6 Discussion

Computing shortest paths and distances is a fundamental problem in social network analysis and many other domains.

We have proposed efficient algorithms for the incremental maintenance of all-pairs shortest paths and distances.

Our experimental evaluation showed that current main-memory algorithms run out of memory even with reasonably sized graphs, and thus disk-based approaches are needed. Our algorithms rely on relational databases and

significantly outperform the state-of-the-art algorithms designed for the same setting.

Conclusions

There has been a significant growth of connected data in the last decade. Enterprises that have changed the world like Google, Facebook, and Twitter share the common thread of having connected data at the center of their business. Such data can be naturally modeled as graphs.

In many current applications, graph data are huge and efficiently managing them becomes a crucial issue. Furthermore, one aspect that many current graph applications share is that data are *dynamic*, that is, they are frequently updated. In this setting, an interesting problem is the development of incremental algorithms to maintain certain kind of information of interest when the underlying data is changed. In fact, incremental algorithms avoid the recomputation of the information of interest from scratch; rather, they tend to minimize the computational effort to update a solution by trying to identify only those pieces of information that need to be updated. In contrast, non-incremental algorithms need to recompute new solutions from scratch every time the data change, and this can be impractical when data are huge and subject to frequent updates.

In this thesis, we have considered two classical graph theory problems, namely the maximum flow and the shortest path/distance problems.

For the former, we have proposed efficient incremental algorithms to maintain the maximum flow after vertex insertions/deletions and edge insertions/deletions/updates. Our algorithms are designed to effectively identify only the affected parts of the network, in order to reduce the computational effort for determining the new maximum flow. We have provided complexity analyses and reported on an experimental evaluation we conducted on several families of datasets with millions of vertices and tens of millions of edges. Experimental results show that our approach is very efficient and outperforms state-of-the-art algorithms.

As for the shortest path/distance problem, we have considered the setting where graphs and shortest paths are stored in relational DBMSs and proposed novel algorithms to incrementally maintain all-pairs shortest paths and distances after vertex/edge insertions, deletions, and updates. The proposed

approach aims at reducing the time needed to update the shortest paths by identifying only those that need to be updated (it is often the case that small changes affect only few shortest paths).

We have experimentally compared our algorithms against state-of-the-art algorithms designed for the same setting on five real-world datasets, showing that our approach is significantly faster.

References

1. Computer Vision Datasets. `http://vision.csd.uwo.ca/data/`.
2. The Maximum Flow Project Benchmark. `http://www.cs.tau.ac.il/~sagihed/ibfs/benchmark.html`.
3. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows - Theory, Algorithms and Applications*. Prentice Hall, 1993.
4. T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proc. of International Conference on Management of Data (SIGMOD)*, pages 349–360, 2013.
5. S. Baswana, R. Hariharan, and S. Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 394–403, 2003.
6. A. Bernstein. Maintaining shortest paths under deletions in weighted directed graphs: [extended abstract]. In *Proc. of ACM on Symposium on Theory of Computing (STOC)*, pages 725–734, 2013.
7. P. Bouros, S. Skiadopoulos, T. Dalamagas, D. Sacharidis, and T. K. Sellis. Evaluating reachability queries over path collections. In *Proc. of International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 398–416, 2009.
8. Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(9):1124–1137, 2004.
9. M. Calautti, S. Greco, and I. Trubitsyna. Detecting decidable classes of finitely ground logic programs with function symbols. In *PPDP*, pages 239–250, 2013.
10. B. G. Chandran and D. S. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations Research*, 57(2):358–376, 2009.
11. L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao. The exact distance to destination in undirected world. *VLDB Journal*, 21(6):869–888, 2012.
12. J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *Proc. of International Conference on Management of Data (SIGMOD)*, pages 457–468, 2012.
13. B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

14. T. Crecelius and R. Schenkel. Pay-as-you-go maintenance of precomputed nearest neighbors in large graphs. In *Proc. of ACM Conference on Information and Knowledge Management (CIKM)*, pages 952–961, 2012.
15. A. Cuzzocrea, A. Papadimitriou, D. Katsaros, and Y. Manolopoulos. Edge betweenness centrality: A novel algorithm for qos-based topology control over wireless sensor networks. *Journal of Network and Computer Applications*, 35(4):1210–1217, 2012.
16. C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 369–378, 2004.
17. C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the ACM*, 51(6):968–992, 2004.
18. C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms*, 2(4):578–601, 2006.
19. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
20. E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Math Doklady*, 11:1277–1280, 1970.
21. J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.
22. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *Proc. of the VLDB Endowment (PVLDB)*, 3(1):264–275, 2010.
23. R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
24. B. Fishbain, D. S. Hochbaum, and S. Mueller. A competitive study of the pseudoflow algorithm for the minimum s-t cut problem in vision applications. *J. Real-Time Image Processing*, 11(3):589–609, 2016.
25. G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *KDD*, pages 150–160, 2000.
26. D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 2010.
27. A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *Proc. of the VLDB Endowment (PVLDB)*, 6(6):457–468, 2013.
28. J. Gao, J. Zhou, J. X. Yu, and T. Wang. Shortest path computing in relational dbmss. *IEEE Transactions on Knowledge and Data Engineering*, 26(4):997–1011, 2014.
29. A. V. Goldberg. The partial augment-relabel algorithm for the maximum flow problem. In *ESA*, pages 466–477, 2008.
30. A. V. Goldberg. Two-level push-relabel algorithm for the maximum flow problem. In *AAIM*, pages 212–225, 2009.
31. A. V. Goldberg, S. Hed, H. Kaplan, P. Kohli, R. E. Tarjan, and R. F. Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In *ESA*, pages 619–630, 2015.
32. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
33. A. V. Goldberg and R. E. Tarjan. Efficient maximum flow algorithms. *Commun. ACM*, 57(8):82–89, 2014.

34. A. V. Goldberg and R. F. F. Werneck. Computing point-to-point shortest paths from external memory. In *Proc. of Workshop on Algorithm Engineering and Experiments and Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO)*, pages 26–40, 2005.
35. S. Greco, C. Molinaro, C. Pulice, and X. Quintana. All-pairs shortest distances maintenance in relational DBMSs. In *ASONAM*, 2016.
36. S. Greco, C. Molinaro, C. Pulice, and X. Quintana. Efficient maximum flow maintenance on dynamic networks. In *Proceedings of the 26th International Conference on World Wide Web Companion, Perth, Australia, April 3-7, 2017*, pages 1383–1385, 2017.
37. S. Greco, C. Molinaro, C. Pulice, and X. Quintana. Incremental maintenance of all-pairs shortest paths in relational dbmss. *Social Network Analysis and Mining*, (to appear), 2017.
38. S. Greco, C. Molinaro, C. Pulice, and X. Quintana. Incremental maximum flow computation on evolving networks. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2017.
39. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. of International Conference on Management of Data (SIGMOD)*, pages 157–166, 1993.
40. M. Henzinger, S. Krinninger, and D. Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the O(mn) barrier and derandomization. In *Proc. of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 538–547, 2013.
41. D. S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, 2008.
42. N. Imafuji and M. Kitsuregawa. Finding web communities by maximum flow algorithm using well-assigned edge capacities. *IEICE Transactions*, 87-D(2):407–415, 2004.
43. G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988.
44. R. Jin, N. Ruan, Y. Xiang, and V. E. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proc. of International Conference on Management of Data (SIGMOD)*, pages 445–456, 2012.
45. S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *Social Computing (SocialCom), 2013 International Conference on*, pages 708–715. IEEE, 2013.
46. S. S. Khopkar, R. Nagi, A. G. Nikolaev, and V. Bhembre. Efficient algorithms for incremental all pairs shortest paths, closeness and betweenness in social network analysis. *Social Network Analysis and Mining*, 4(1):220, 2014.
47. V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 81–91, 1999.
48. P. Kohli and P. H. S. Torr. Dynamic graph cuts for efficient inference in markov random fields. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(12):2079–2088, 2007.
49. K. Nagi. A new representation of wordnet using graph databases on-disk and in-memory. *International Journal on Advances in Software Volume 6, Number 3 & 4, 2013*, 2013.

50. C. Pang, G. Dong, and K. Ramamohanarao. Incremental maintenance of shortest distance and transitive closure in first-order logic and SQL. *ACM Transactions on Database Systems*, 30(3):698–721, 2005.
51. L. Planken, M. de Weerdt, and R. van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. *Journal of Artificial Intelligence Research*, 43:353–388, 2012.
52. N. Przulj, D. A. Wigle, and I. Jurisica. Functional topology in a network of protein interactions. *Bioinformatics*, 20(3):340–348, 2004.
53. Z. Qi, Y. Xiao, B. Shao, and H. Wang. Toward a distance oracle for billion-node graphs. *Proc. of the VLDB Endowment (PVLDB)*, 7(1):61–72, 2013.
54. M. Qiao, H. Cheng, L. Chang, and J. X. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. In *Proc. of IEEE International Conference on Data Engineering (ICDE)*, pages 462–473, 2012.
55. S. A. Rahman, P. Advani, R. Schunk, R. Schrader, and D. Schomburg. Metabolic pathway analysis web service (pathway hunter tool at cubic). *Bioinformatics*, 21(7):1189–1193, 2005.
56. I. Robinson, J. Webber, and E. Eifrem. *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.", 2015.
57. L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM Journal on Computing*, 41(3):670–683, 2012.
58. H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of link spam detection by graph algorithms. In *AIRWeb*, 2007.
59. C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46:45:1–45:31, 2014.
60. A. Tagarelli and R. Interdonato. Time-aware analysis and ranking of lurkers in social networks. *Social Network Analysis and Mining*, 5(1):46:1–46:23, 2015.
61. C. Tesoriero. *Getting Started with OrientDB*. Packt Publishing Ltd, 2013.
62. D. N. Tran, B. Min, J. Li, and L. Subramanian. Sybil-resilient online content voting. In *NSDI*, pages 15–28, 2009.
63. L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *Proc. of the VLDB Endowment (PVLDB)*, 5(5):406–417, 2012.
64. H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: defending against sybil attacks via social networks. In *SIGCOMM*, pages 267–278, 2006.
65. A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *Proc. of International Conference on Management of Data (SIGMOD)*, pages 857–868, 2013.
66. A. D. Zhu, X. Xiao, S. Wang, and W. Lin. Efficient single-source shortest path and distance queries on large graphs. In *Proc. of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 998–1006, 2013.