Title
Query-based Graph Data Reduction

Author
Wang, Shao-Ting

Publication Date
2017

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Query-based Graph Data Reduction

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Shao-Ting Wang

Dissertation Committee:
Professor Phillip Sheu, Chair
Professor Jean-Luc Gaudiot
Professor Pai Chou

2017

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost I want to thank my advisor Phillip Sheu. It has been an honor to be his Ph.D. student. In the years of my Ph.D. studies, Professor Sheu trained me in many aspects, not only in academic learning, but also in social and ethical competencies. His enthusiasm for his research was motivational for me, making my Ph.D. experience productive and stimulating. I appreciate all his contributions of time and ideas on my dissertation and also the funding that enabled me to keep working on my research without having to worry about financial issues. Words cannot express how thankful I am to have such a supportive advisor.

I am thankful to the members in my Lab. Guigang Zhang helped me on the thesis structure, provided me ideas, and also helped me develop the use cases and examples. Jenny Jin helped me on polishing my dissertation, as I am not a native English speaker. Hao Xing, Bryan Chou, and Zheng Hu helped me rehearse my presentation on dissertation, and also gave me some advice. As a cooperator of our Lab, I am also thankful to the supports from NEC Solution Innovators, Ltd., Japan. Mr. Hiroyuki Shigematsu and Mr. Atsushi Kitazawa gave me many useful suggestions on my research directions. Mr. Masahiro Hayakawa helped me conduct experiments to verify my results. I appreciate to all the help.

For this dissertation I would also like to thank my committee members: Professor Jean-Luc Gaudiot and Professor Pai Chou. With their expert knowledge in this area, they provided many useful comments so I could refine my dissertation, and also enlightened me to some future research directions.

Finally I would like to thank my family and all my friends. Without your support and encouragement, I would not have been able to make this achievement. Thank you.

Shao-Ting Wang
University of California, Irvine
2017

ABSTRACT OF THE DISSERTATION

Query-based Graph Data Reduction

By

Shao-Ting Wang

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2017

Professor Phillip Sheu, Chair

Graphs are widely used nowadays to store complex data of large applications such as social networks, recommendation engines, computer networks, bio-informatics, just to name a few. With the rapid growth of the Internet, designing scalable systems to process huge amount of graph data efficiently has become a challenge. In order to store and process such data efficiently, as well as to save the time for transferring them, graph compression techniques are often used. Most of the existing graph data compression approaches are syntactic, focusing on graph structures and data reduction based on serialization or redundancy removal. While they can be applied uniformly by all applications, further reduction of graph data can be achieved by looking into the semantics of an application.

In this thesis we propose a query-based approach to graph data reduction which preserves only the information relevant to the class of queries needed for an application. We study several classical graph problems and their applications, and design graph reduction algorithms to generate a reduced graph from which we can still compute the same solutions as those from the original graph. We also introduce the concept of "lossy"

graph reduction for applications that may tolerate small but bounded errors in exchange for further data reduction. In addition, we design a synthesis algorithm that can combine existing graph reduction algorithms to generate a reduced graph for complex graph problems which include more than one constraint. With such we do not need to design a separate graph reduction algorithm for every multiple-constraint problem which can be expensive as the number of constraint combinations increases exponentially.

Chapter 1 Introduction

Nowadays lots of internet applications store their data as graphs, where a graph can be an undirected graph (Figure 1a), a directed graph (Figure 1b), a combination of both (Figure 1c), or a forest (Figure 1d). Note that a combination graph can also be represented by a directed graph as one undirected edge can be represented by two edges in both directions.

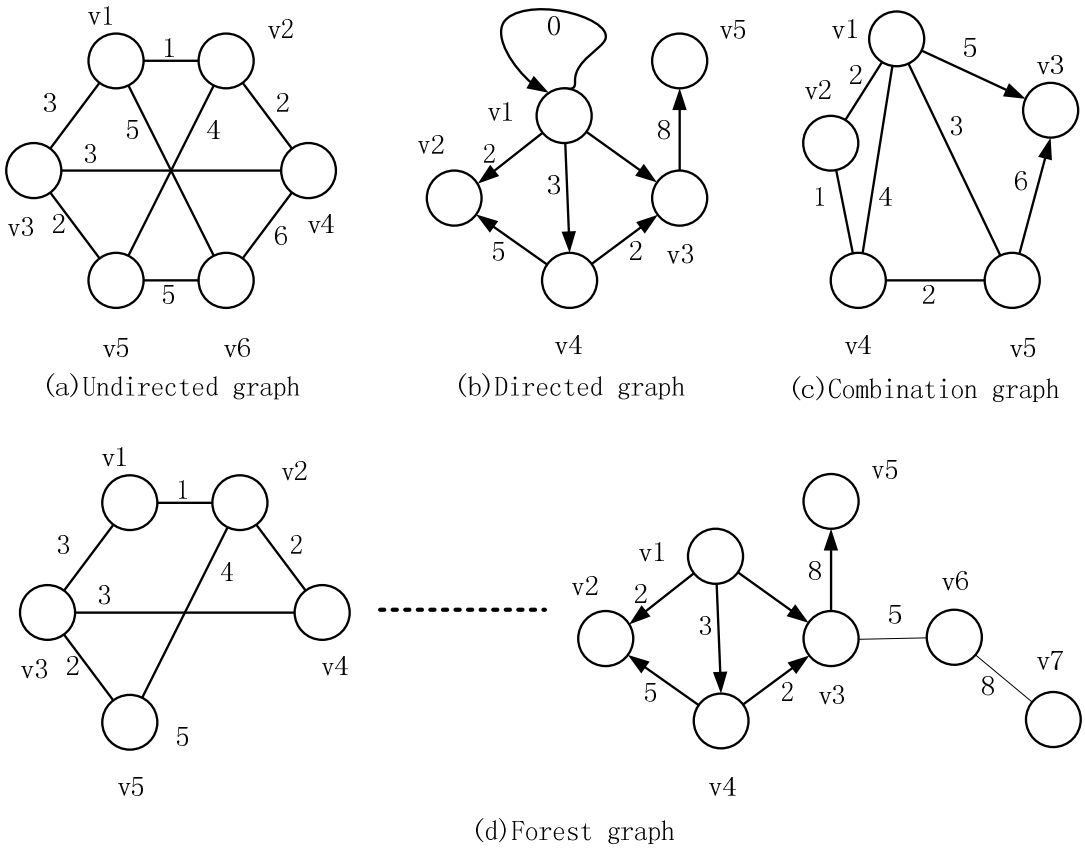


Figure 1. Possible graph applications

In order to process large graphs, it is necessary to compress them in degree. However, how to reduce a big graph including say billions of nodes and edges using an efficient method is a very huge challenge. And also, designing a good graph data reduction method which can not only save the storage size but also reduce the computing complexity is very important. Most of the

existing graph data compression approaches are syntactic, which means they focus on graph structure and reduce a graph by serialization or redundancy removal.

In this thesis we propose a semantic approach, namely query-based graph data reduction, which reduces a graph by preserving only the information relevant to the queries addressed by an application.

Given a graph *G* and a set of application queries *Q* against *G*, the Query-based Graph Reduction Problem (QGRP) considers the derivation of a new graph *G'* that retains the original answers for *Q* to reduce the storage and computational complexity. We can summarize the methods of QGRP into the following categories.

(1) Sub-Graph Reduction

Based on the definition, *G'* is a subgraph of *G* that can produce the same results for *Q*. Figure 2 shows one example. Given *G*=(*V*,*E*). *V*={*v1*,*v2*,*v3*,*v4*,*v5*,*v6*,*v7*}. If the answers to *Q* only require *G'*=(*V'*,*E'*), where *V'*={*v2*,*v4*,*v5*,*v7*} and *E'*={(*v2*,*v4*), (*v2*,*v5*), (*v5*,*v7*)}, then *G'* is a reduced subgraph of *G* for *Q*. Reduction of this type is usually lossy because *G'* may not be able to produce the same results for queries that are not included in *Q*.

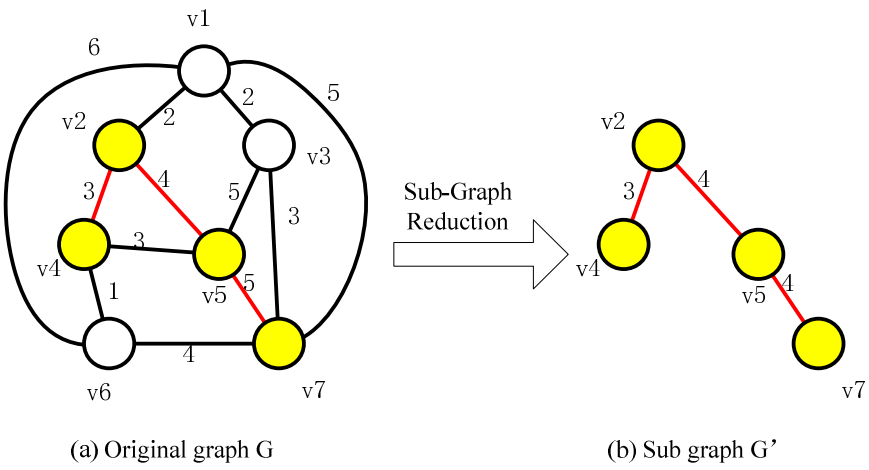


Figure 2. Sub-graph data reduction

(2) Modified Graph Reduction

Sometimes, we can remove a set of nodes/edges from *G* and add a smaller set of nodes/edges to preserve the answers for *Q*. Figure 3 shows one example of modified graph reduction, where an edge (*v2*,*v7*) is added to make up the removal of {*v1*,*v3*,*v*6} and the edges associated with them.

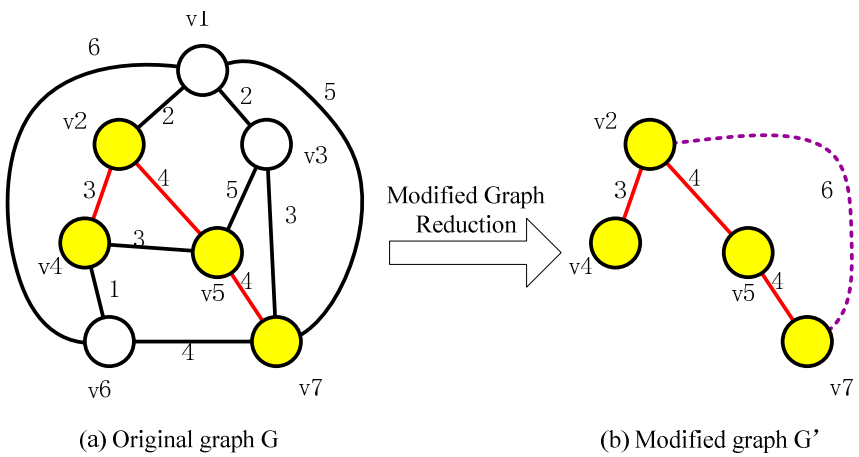


Figure 3. Modified-graph reduction

Much of the graph reduction methods in this category can still be considered as "lossless" graph reduction [26] (e.g., online reduction [27] and dynamic online network reduction [28]) in the sense that after reduction we can obtain the same query results based on the reduced graph and the original graph.

While syntactically all graph reduction methods are lossy because the original graph is reduced to a smaller graph (therefore some data is lost), in our research we interpret the term "lossy" more strictly, and semantically, referring to reduction methods that either lose the ability to answer some queries that may be answered from the original graphs correctly or losing the correctness of some results.

Our hypothesis is that further reduction may be possible even on top of query based reductions. Sometimes the accuracy of query results may not be strictly required (or expected such as with a search engine). Specifically if we allow approximate results, where the difference between the approximate results and the precise results is bounded, a graph may be further reduced.

The graph problems could become very complex if it contains multiple objectives and constraints. It is impractical to develop every lossy graph reduction algorithm for all such complex graph problems, as the number of graph problems grows exponentially with the number of constraints. Therefore, the reduction algorithm synthesis methods are introduced in my thesis, in order to synthesize existing algorithms to solve complex problem.

The rest of the dissertation is organized as follows: In Chapter 2, we introduce graph databases which are widely used to store the data in graph form, and also the existing approaches to graph data reduction problems. In Chapter 3, we define the Query-based Graph Reduction Problem (QGRP). The reduction algorithms for several classical problems and their variants related to QGRP are discussed in this chapter. In addition, we discuss reduction algorithm synthesis and incremental maintenance methods for QGRP. In Chapter 4, we define the Lossy Graph Reduction Problem (LGRP). The reduction algorithms for several classical problems and their variants related to QGRP are introduced. We also extend the Graph Reduction Problem to the Global Constraint Graph Reduction Problem which is more general, and discuss some related applications. In Chapter 5, we present the experiment results that compare the reduction rates with different sizes and types of data. The dissertation is concluded in Chapter 6 with a discussion of some future directions.

Chapter 2 Graph Database and the Graph Reduction Problem

Graph theory dates back to 1735, when Leonard Euler solved the Seven Bridges of Königsberg problem, “Is it possible to find a walk through the city that would cross each bridge once and only once?” Euler designed a mathematic model consisting of vertices and edges to solve this problem, laying the foundation of graph theory. Graph theory has since found many uses. One of the famous questions is the “Four Color Problem” posed by Francis Guthrie in 1852, but the problem wasn’t solved until 1969. Heinrich Heesch solved the problem with the assistance of a computer. Although graph theory has been researched for a long time, only recently has it been applied to storing and managing data.

2.1. Graph Databases

Extensive research has been done on graph database models which use graphs to provide representations of real-world entities and the relationships among them. We can conceptualize the notion of a graph database model into three basic components: data structures, transformation languages, and integrity constraints [1]. The following researches have been done on graph database models and we characterize them by the three different components:

Data Structure: Graphs are used to represent the data and/or the schema. Kuper and Vardi [2] use a directed graph to build a database schema, where leaves are data and internal nodes are relations. Kunii [3] defines a database schema using a labeled directed graph. Güting [4] encodes the whole database using graphs. Levene and Loizou [5] use a single labeled directed graph as the underlying data structure of the whole database. Paredaens et al. [6] formalize the database schemas, instances, and rules by directed labeled graphs. Gyssens et al. [7] treat object-oriented database schemas and instances as graphs, where the database objects are represented by nodes.

Amann and Scholl [8] organize the data into graphs. Hidders and Paredaens [9] represent schemas and instances by labeled graphs. Graves et al. [10] first formalize the representation of data structures using graphs, and then store them into a database. Finally Hidders [11] uses certain types of labeled graphs to describe the schema and instances.

Transformation Language: This is used to manipulate data and graph features like paths, neighborhoods, connectivity, graph patterns, subgraphs, and statistics. Graves et al. [10] define a flexible collection of operations and constructors to build and access a graph database. Güting [4] uses powerful graph manipulation primitives to query a graph database. Hidders and Paredaens [9] introduce a language based on pattern matching by finding a prototype of a graph from the database.

Integrity constraints: Data consistency is also an important feature that is needed to be enforced in a database. The constraints can be categorized as entity integrity, referential integrity, domain integrity, node/edge identities and path constraints. Graves et al. use labels with unique names [10] to maintain the data consistency, Kuper and Vardi [2] has typing constraints on nodes, Levene and Poulovassilis [5] use functional dependencies, and Klyne and Carroll [12] consider the domain and range of properties in their graph database model.

To conclude, graph database models can be defined as database models that use graph structures with nodes, edges and properties to store data. They use semantic queries and graph-oriented operations to access them. Appropriate integrity constraints should also be defined over the graph structure to enforce data consistency.

Comparing to other graph database models, there are two models that are mostly studied and widely used in the implementation of graph database, RDF and Property Graph.

RDF: Due to the rapidly growth of the number of Internet resources, there was an increasing need for encoding, exchanging and reusing the metadata, which is structured data about data, on the Internet. W3C [13] defines an infrastructure based on the graph model called Resource Description Framework (RDF) [12], and it has become a widely used standard in modern graph databases. There were several ancestors to the W3C's RDF. The history of metadata at the W3C dates back to 1995 with PICS [14], and PICS-NG Next Generation [15]. Some metadata communities contributed intellectual resource, including XML [16], XMLDATA [17], MCFXML [18], DC [19] and WF [20]. Those metadata communities brought together their needs, cooperating together, to build a robust and flexible architecture to support metadata on the web, and finally came out with RDF.

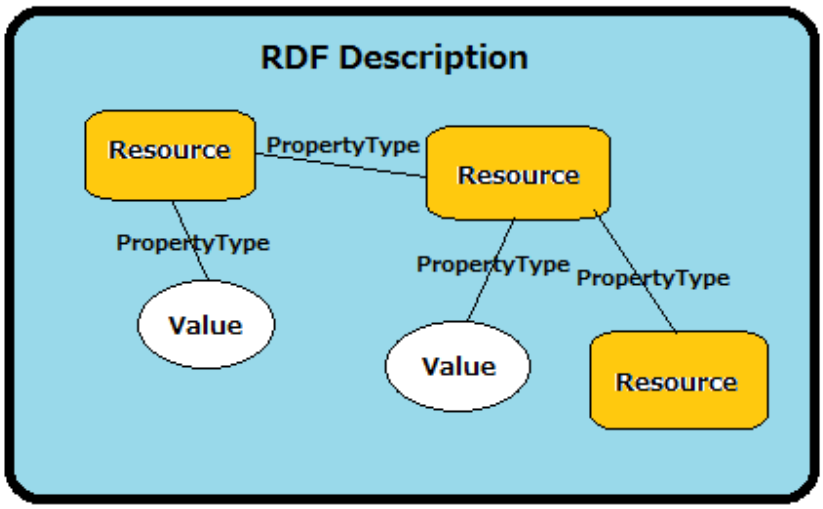


Figure 4. RDF Description

RDF provides a model for describing resources. Anything can be a resource, including physical things, documents, abstract concepts, numbers and strings, and can be uniquely identified by a Uniform Resource Identifier (URI). Resources have properties, and the properties associated with resources are identified by property-types, which have corresponding values. Properties express the relationships of values associated with resources. In RDF, values may be strings, numbers, other generic data types, or other resources which may also have their own properties. A collection of these properties of the same resource is called a description. Different people may have their own different federated descriptions for the same resource. At the core of RDF is a syntax-independent model for representing resources and their corresponding descriptions. Figure 4 is an example of RDF data structure in graph view. In an application, a set of terms used to describe the data and metadata is called RDF vocabulary. Sharing and reusing RDF vocabularies aids interoperability of the data. RDF vocabularies can be defined by RDF Schema (RDF-S) or Web Ontology Language (OWL).

Property Graph: In Property Graph [21], a node represents an entity, and a link represents a relationship between the entities. Nodes can be tagged with labels representing their different roles in the domain. Relationships provide directed, named semantically relevant connections between two node-entities. A relationship always has a direction, a type, a start node, and an end node. Both entities and relationships can have properties, which are key-value pairs; in contrast, RDF uses triples (subject, predicate/predicate, and object.) For example, in Property Graph, an entity which is a person could have properties name=John and age=29; a relationship may have quantitative properties like weights, costs, distances, or time intervals. Figure 5 illustrates Property Graph.

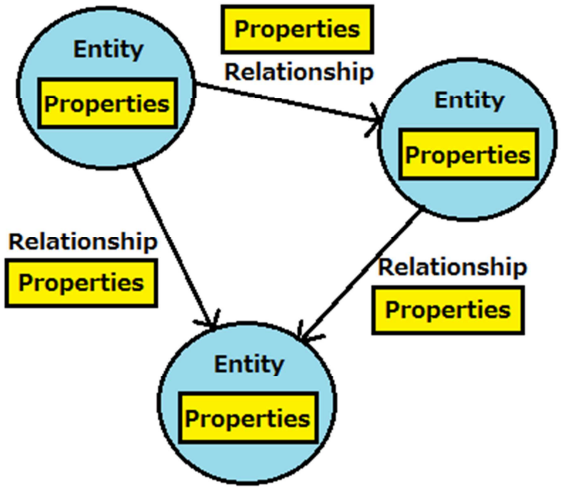


Figure 5. Property Graph

The difference between RDF and Property Graph is mainly as follows: Instead of using links as attributes in RDF, Property Graph attaches attributes in nodes, and also allows links to have attributes. In RDF, encoding, exchanging, and reusing the metadata on the Web is the objective, so URIs are used to define the resources, while Property Graph doesn't have this purpose. RDF is a standard recommended by W3C; however, different implementations may use different definitions in Property Graph, although TinkerPop [21] can be considered as a de facto standard.

2.2. Comparison of Graph Database with Other Database Models

We briefly compare the graph database model with two other most widely used database models in the following.

2.2.1. Relational Database

Relational database model [22] was introduced by Codd, where all data is represented in terms of tuples, grouped into relations. The relational model was a landmark development

because it gave the data modeling discipline a mathematical foundation, with relational algebra and logic. In the relational model, the data is organized into tables (relations), which are quite intuitive, as spreadsheets are widely used to store and maintain data in the real world. However, the relational database has some drawbacks. One is that the schema is fixed, which makes it difficult to extend. Another problem is known as “object relational impedance mismatch,” which is caused by the difference between the relational model and the object-oriented concept used by applications developed using object-oriented programming languages. Therefore, the two technologies do not work together seamlessly and often cause technical difficulties when developing applications. Finally, retrieving information from a relational database often needs to incorporate many join operations between tables, and it is very costly when the data stored in the tables is enormous. This is pretty normal in many big online companies such as Amazon and Facebook.

In order to solve these issues, the graph database has been brought into focus in recent years. The objects in the real world are naturally organized into graphs. For example, people and their relations in a social network are connected into a graph; the knowledge terms and their semantic interconnection (e.g. a dog is an animal) can be represented as a graph. The graph database stores the data in a more natural way, and all the information of an object is stored in a node and the edges connected to it, preventing the costly join operations. When developing object-oriented applications, objects can be organized into graph which prevents the mismatching issue. Moreover, the graph database can be schema-less for easily extension.

2.2.2. Object-Oriented Database

Object-oriented (O-O) database models [23] appeared in the eighties. People found that relational databases are not suitable for object-oriented applications, especially when they are data-intensive, such as CAD/CAM software, computer graphics, and information retrieval. To overcome this problem, O-O database models were introduced. In these models, data is represented as collections of objects that are organized into classes, which is the same as the object-oriented programming paradigm.

O-O database models are related to graph database models because of their explicit or implicit use of graph structures in definitions [24, 25]. The major difference between them is how they abstract the world. O-O database models view the world as complex objects with data, and they interact by methods. On the other hand, graph database models view the world as a network of relations, and focus on data interconnection. Another important difference is that an O-O database has the same schema as the application using it, while graph database can be schema-less, which is more flexible and can be used by multiple applications simultaneously.

2.3. Related Work to Graph Data Reduction Problem

Most existing approaches to graph data compression or reduction fall into the category of syntactical graph reduction. A partial list of references includes [26-31]. In this section, we summarize graph data reduction methods related to Internet applications that may be modeled as large graphs.

2.3.1. Logical Structure Based Graph Data Reduction

The following four methods are popular for reducing graph data.

(1) Replacing Dense Structure

Feder and Motwani [32] introduce a method to detect a complete bipartite graph and replace it with a virtual node. Beuhrer et al. [33] introduce an approach based on frequent item set mining to identify groups of nodes that share the same outgoing links, and replace them by a virtual node that points to those targets.

(2) Neighborhood Similarity

Boldi and Vigna [34] introduce the WebGraph framework to compress a Web graph based on locality and similarity. Maserrat [35] presents a compression technique to answer neighborhood queries efficiently by storing the Eulerian path or a generalization of the Eulerian path if it does not exist.

(3) Serialization

Fern´andez et al. [36] introduce HDT (Header-Dictionary-Triples) which is a binary serialization format optimized for RDF storage and transmission. Alvarez-Garc´ıa et al. [37] present the idea of K2-triple which partitions a dataset into subsets of pairs connected by a predicate, and the subsets are represented as a set of binary matrices.

(4) Logical Compression

Joshi et al. [38] introduce a logical compression technique. It automatically builds a set of inference rules from a dataset and removes all the triples that can be inferred using these rules. Data mining techniques are used to find these rules. Grimm et al. [39] find the logical redundancy within the ontology of RDF data and remove the redundancy to reduce the size of

data. Jeff Z. Pan et al. [40] present a compression method to replace a large graph pattern in RDF data by a smaller graph pattern based on logical rules.

2.3.2. Physical Storage Structure Based Graph Data Reduction

The former four graph data reduction methods (replacing dense structure, neighborhood similarity, serialization, and logical compression) mainly focus on logic structures. These reduction methods do not consider physical storage structures directly. Graph data reduction methods that consider physical storage structure include correlation matrix based graph data reduction and adjacency list based graph data reduction.

(1) Correlation matrix based graph data reduction

In most Internet applications, graph data can be stored using a correlation matrix. A method based on K2 [41, 42, 43, 44] can be very useful for reducing sparse matrices.

(2) Adjacency list storage based graph data reduction

An adjacency list uses pointers. Every row in an adjacency list consists of a node and its neighbor nodes. For a graph that includes many nodes whose neighbors are similar, we can reduce the redundant nodes to save space. Re-pair [45] is a reduction method that can be used in this situation. Many papers use the concept of Re-pair to reduce a graph [46, 47, 48, 49]. LZ78 [50] is an adjacency list based graph reduction as well.

2.3.3. Application Based Graph Data Reduction

Some methods have been proposed to reduce a graph that models a specific application:

(1) Web page based graph data reduction

Several algorithms such as K2-partitioned [51], AMN [52], BV [53, 54] and VNM [55] were proposed to reduce a graph that consists of web pages.

(2) Social Network Based Graph Data Reduction

Almost all social network applications can be modelled as a graph. DSM [56], FRS [57] and BFS [58] are proposed to reduce social network graph data.

2.3.4. Query-based Graph Data Reduction

Most existing methods that we discussed are syntactic – focusing on graph structures and data reduction based on serialization or redundancy removal. While they can be applied uniformly by all applications, some applications may only focus on special queries, e.g., adjacency queries, graph pattern queries and reachability queries. In this situation removing some information from the original graph may be allowed. For example, MPk [59] can be used to reduce a graph to focus on adjacency queries. QPGC [60] is efficient for special queries such as graph pattern queries and reachability queries.

2.4. Related Work to Lossy Graph Data Reduction

Lossy data reduction methods, also called lossy compression, for continuous sources have been well studied [64]. They are widely used in images, video and audio whose applications tolerant errors as human beings can still accept. Some of the approaches [65, 66, 67] use graphs in their algorithms for encoding and decoding the data. However, using the similar idea to reduce graph data in general is a new research direction, where only few publications can be found. Some work related to this direction is summarized in the following.

2.4.1. Approximate Query Processing.

Much research has been done for providing approximate answers to relational queries, such as samples [68], histograms [69] and wavelets [70]. These approximation techniques are more suitable for generating estimates for aggregated data, e.g., counts and averages. Some other approaches approximate queries on graph patterns [71] and similarity [72, 73] from a large graph [71] or a compressed graph [72, 73]. Approach [74, 75] by querying partial graph to retrieve approximate query results is also approached on reachability and pattern queries. The motivation behind these approaches is primarily on reducing the processing time, not reducing the dataset.

2.4.2. Clustering

In the data mining community, graph clustering has been widely used as a tool for summarization. With clustering, the original graph can be reduced to a graph of clusters, and approximate queries can be applied to retrieve some inner-cluster properties and relationship between clusters.

For some graphs with low density, clustering [76] is a good data reduction method, especially for high dimensional data reduction [77].

2.4.3. Graph Reduction for Reachability Query

The reachability problem is a very important problem in graph computing. A lot of applications need to compute the results based on k-reachability. Reachability reduction methods [78, 79, 80] have been reported in some papers. While most research in reachability reduction is lossless, Zhou et al. [81] describe a lossy network simplification method that reduces a graph with the minimum impact on graph connectivity. By maintaining the connectivity, most reachability and path queries can still be answered.

2.4.4. Lossy Graph Compression and Summarization

The research in this category may be most related to our interpretation of lossy graph reduction.

Navlakha et al. [82] describes a graph summarization method with bounded errors. A graph can be reduced by a two-part representation: summary and the corrections. The summary is an aggregate graph that groups the nodes into sets with edges representing relations between sets. The corrections part specifies a list of edge-corrections that should be applied to the summary to recreate the graph. The summarization can be lossless or lossy – here the term "lossy" means there are errors in finding neighbors of each node.

Toivonen et al. [83] compress a weighted graph by merging nodes and edges to achieve the target reduction rate while minimizing the difference on the edge weights between the original graph and the compressed graph.

The concept of lossy reduction in the above papers is based on graph structures, focusing on errors between the original graph and the reduced graph. In this thesis, we focus on query results – the errors from querying a reduced graph should be bounded.

2.5. Term definition

Below we first review some terms in the graph theory and the graph queries (problems) discussed in this thesis.

(1) Weighted Graph: A weighted graph G is a triplet (V, E, w), where w: $E \rightarrow eVal$ is a function mapping an edge to its weight, and $eVal$ is the set of possible values. For a weighted graph, $eVal$ would typically be real numbers.

(2) Path: A path p from node u to v in G is a sequence of nodes ($u=v_0, v_1, \ldots, v_n=v$) such that for every $i \in \{1, \ldots, n\}$, $(u_{i-1}, u_i) \in E$.

(3) Shortest Path: A shortest path from u to v is a path p= ($u=v_0, v_1, \ldots, v_n=v$) of which the sum $\sum_{i=1}^{n} w(e_{i-1,i})$ is minimal. We use the notation $\delta_G(u, v)$ to denote the weight of the shortest path from u to v. There may be multiple shortest paths between two nodes.

(4) Minimum Spanning Tree (MST): Given an undirected and weighted graph $G=(V,E,w)$, a spanning tree on $V_i \in V$ is a set of edges from E that connect all the vertices belong to V_i without any cycles. A minimum spanning tree is a spanning tree that has the minimum total weight. There may be multiple minimum spanning trees.

(5) Degree-Constrained MST (DC-MST): A degree-constrained MST is an MST of which the maximum vertex degree is limited by a constant k, i.e., degree(i) $\leq k$ for all $i \in V$.

(6) Weight-Constrained MST (WC-MST): A weight-constrained MST (WC-MST) is an MST of which the weight of each edge is not greater than a given constraint k.

(7) Distance-Constrained Reachability: A reachability query is that given 2 nodes x and y as the input, answer if x can reach y. A distance-constrained reachability query is a reachability query with a given distance constraint d, that is, if x can reach y within the distance d.

Chapter 3 Query-based Graph Data Reduction

3.1. Definition of Query-based Graph Reduction Problem (QGRP)

In this thesis, we focus on query based graph reduction problems whose objective is to reduce a graph by preserving only the information needed for a specific set of queries.

The query based graph reduction problem can be defined as follows. Given a weighted graph G and a set of queries Q, find a reduced graph G' that G' is a smallest subgraph of G with which for every q of Q, q(G) = q(G'), where q(G) is the answer to query q in G, and q(G') is the answer to q in G'. That is, we want to find a smallest reduced graph G' so that it can still answer all the queries for a specific graph problem.

3.2. QGRP for the Shortest Path (SP) Problem

In this section, we present our approaches to QGRP assuming the queries are confined to the shortest path problem.

3.2.1. GR-SP (Shortest Path)

(1) Definition of GR-SP

Given a graph *G*, assume an application is only interested in finding the shortest paths between any pair of two nodes. Can we reduce *G* to a smaller graph *G'* which preserves all the shortest paths between any pair of nodes? We call this problem the GR-SP (Graph Reduction-Shortest Path) Problem.

(2) Use Case of GR-SP

Using a road network as an example, there may be applications that are only interested in finding the shortest paths between any pair of locations.

(3) Algorithms of GR-SP

Our hypothesis is that if the weight of an edge between two nodes is greater than the weight of a shortest path between them, then we can remove the edge. It consists of the following two steps.

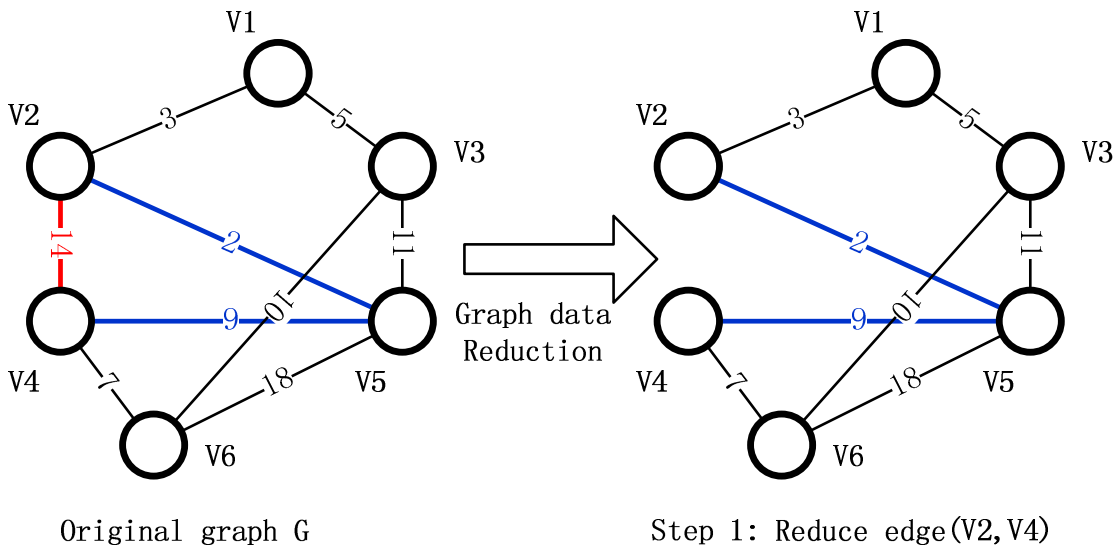
Step 1: Find the shortest path between any pair of nodes *a* and *b*, and the shortest distance is $\delta_G(a,b)$

Step 2: For any edge (*a*,*b*) that connects *a* and *b*, if the weight *w(a,b)* is greater than $\delta_G(a,b)$, remove the edge (*a*,*b*).

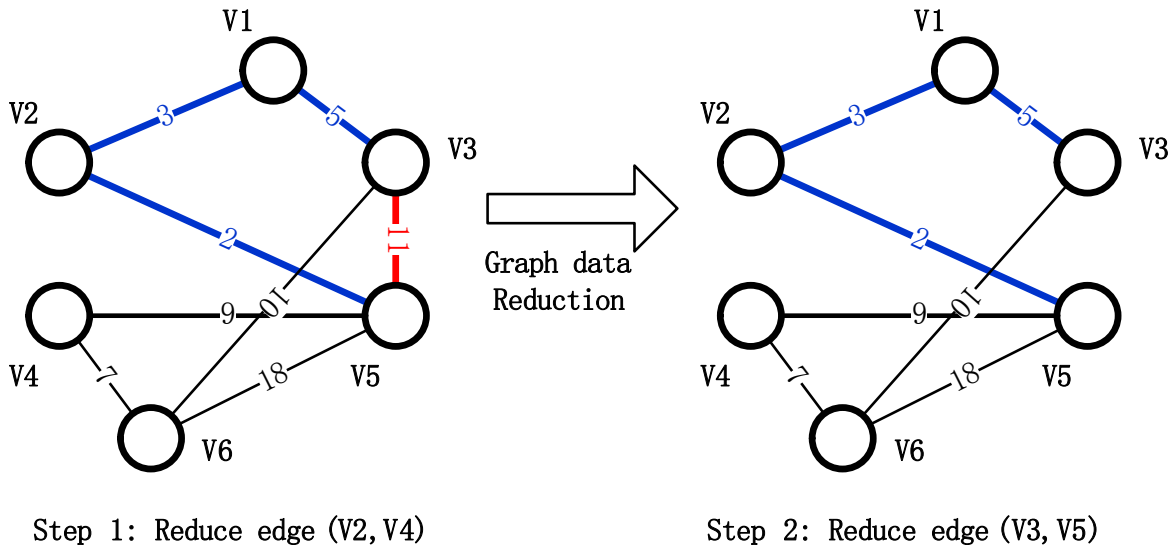
Figure 6 shows an example of the reduction process. In Figure 6(a), we can see that *w(v2,v4)*=14; however *w(v2, v5)* + *w(v5,v4)* = 2+9 = 11 < 14. So, the edge (*v2, v4*) can be deleted.

In Figure 6(b), we can see that *w(v3,v5)*=11; however *w(v3, v1)* + *w(v1,v2)* + *w(v2,v5)* = 5+3+2 =10 < 11. So (*v3, v5*) can be deleted.

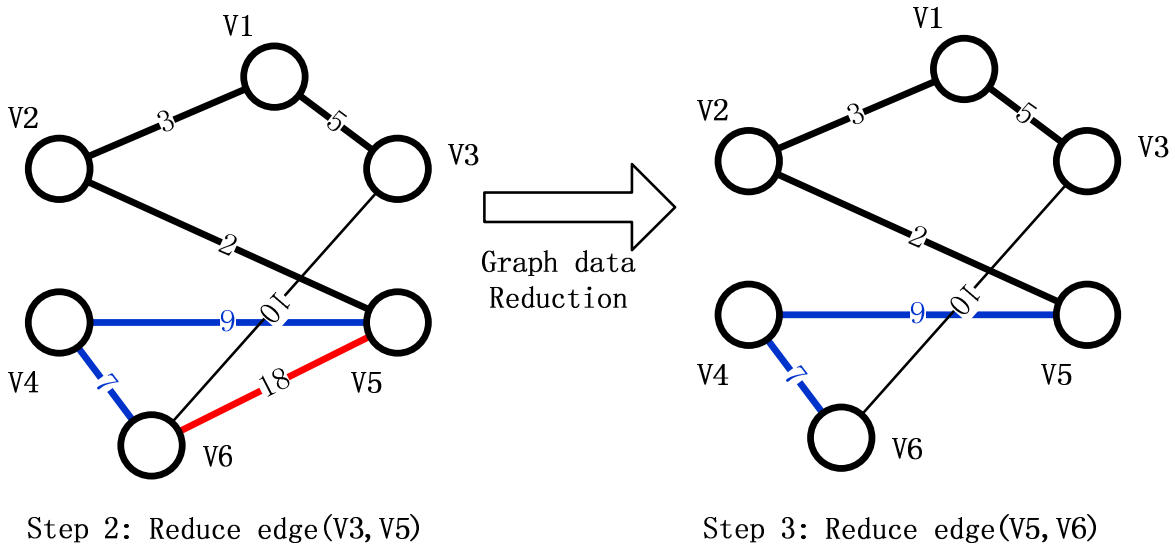
In Figure 6(c), we can see that *w(v5,v6)*=18; however *w(v5, v4)* + *w(v4,v6)* = 9+7 = 16 < 18. So (*v5, v6*) can be deleted.



(a) Delete (v2,v4)



(b) Delete (v3, v5)



(c) Delete edge (v5, v6)
Figure 6. GR-SP

The GR-SP algorithm can be designed based on the Dijkstra algorithm or the Floyd algorithm, i.e., when deciding an edge is to be removed or not, the shortest distance between the incident vertices can be calculated by the Dijkstra or Floyd algorithm. Due to the space limitation, only the Dijkstra algorithm is considered in this thesis.

GR-SP algorithm

GR-SP algorithm (Find a minimal reduced graph that preserves all the shortest paths between any pair of nodes from the original graph.)

Input: Graph *G*
Output: A reduced graph *G'*

```
G' ← G
N ← Number_of_Nodes(G)   //computing the graph G's nodes number.
 For i ← 0 to N do
    Dist[i] ← Dijkstra(G, i)  //use Dijkstra algorithm to compute the shortest path from source i  to other nodes.
    For j ← 0 to N do
        If edge E(i,j) exists and E(i,j)>Dist[i][j]
            Remove E(i,j) from G'
    EndFor
 EndFor
 Return G'
```

The time complexity of the GR-SP algorithm is $N*T(Dijkstra)$, where N is the number of nodes and $T(Dijkstra)$ is the time complexity of the Dijkstra algorithm. For most implementations, $T(Dijkstra)=O(N^2)$. With a min-priority queue implemented by a Fibonacci heap, the time complexity of Dijkastra algorithm can be improved to $O(E+NlogN)$, where E is the number of edges.

(4) Theorems and Proof for GR-SP

Theorem 1: The GR-SP algorithm produces a minimal subgraph *G'* from *G* that preserves all the shortest paths between any pair of nodes.

Proof: The theorem can be proved by the following two lemmas 1 and 2.

Lemma 1: The GR-SP algorithm produces a subgraph *G'* from *G* that preserves all the shortest paths between any pair of nodes.

Proof: Assume there exists a shortest path that is not preserved in $G'=(V', E',w')$. That is, there exists a shortest path $p = (u_0, u_1, \ldots, u_n)$, and for some $i \in \{1, \ldots, n\}$, $(u_{i-1}, u_i) \in E$ but $(u_{i-1}, u_i) \notin E'$.

Based on the GR-SP algorithm, if $(u_{i-1}, u_i) \notin E'$, $w(e_{i-1,i}) > \delta_G(u_{i-1}, u_i)$. Therefore there exists a path $(u_{i-1}=v_0, v_1, \ldots, v_m=u_i)$ from u_{i-1} to u_i which is shorter than $e_{i-1,i}$. As a result, we can generate a path $p' = (u_0, u_1, \ldots, u_{i-1}=v_0, v_1, \ldots, v_m=u_i, \ldots, u_n)$ which is shorter than p, that contradicts the assumption that p is the shortest. ■

Lemma 2. *G'* is minimal among all subgraphs that preserve all the shortest paths between any pair of nodes.

Proof: Assume there exists a subgraph $G''=(V'', E'', w'')$ which preserves the shortest paths between any pair of nodes and it is smaller than G'. That is, there exists an edge $(u,v) \in E'$ but $(u,v) \notin E''$.

Based on the GR-SP algorithm, if $(u,v) \in E'$, $w(e_{u,v}) = \delta_G(u, v)$ so (u,v) is one of the shortest paths between nodes u and v. Since G'' does not preserve (u,v), it contradicts the assumption that G'' preserves the shortest paths between any pair of nodes.■

If an application only needs to preserve at least one shortest path for any pair of nodes, we can remove more edges from the graph. Since not all the shortest paths are preserved, we call this graph a *weak* version of GR-SP. Note that in the later part of this thesis, some algorithms use GR-SP as a part (e.g., GR-MST). The GR-SP can also be replaced by the weak version of GR-SP, making the corresponding algorithms weak also. Due to space limitation, we will not define the weak version of these algorithms again later.

GR-SP algorithm (weak version)

GR-SP algorithm (weak version) (Find a minimal reduced graph that for any pair of nodes, at least one of the shortest paths is preserved**)**

Input: Graph G
Output: A reduced graph G'

```
G' ← G
N ← Number_of_Nodes(G); //computing the graph G's nodes number.
For i ← 0 to N do
    For j ← 0 to N do
        If the edge E(i,j) exists
            Remove E(i,j) from G';
            If Dijkstra(G', i, j) > E(i,j)
                add E(i,j) to G'
            EndIf
        EndIf
    EndFor
EndFor
Return G'
```

The time complexity of the GR-SP algorithm (weak version) is N^2*$T(Dijkstra)$, where N is the number of nodes and $T(Dijkstra)$ is the time complexity of the Dijkstra algorithm.

Theorem 1b: The GR-SP algorithm (weak version) produces a minimal subgraph G' from G that preserves at least one of the shortest paths between any pair of nodes.

Proof: The theorem can be proved by the following lemmas 1b and 2b.

Lemma 1b: The GR-SP algorithm (weak version) produces a subgraph G' from G that preserves at least one of the shortest paths between any pair of nodes.

Proof: For any pair of nodes in the original graph G, we can find a shortest path p for it. The algorithm only removes an edge if removing it does not affect the shortest distance between the nodes. Therefore, if an edge e in the path p does not exist in G', it means there exists another path p_e' in G' whose weight is smaller or equal to the edge weight $w(e)$. By the definition that p is shortest, it is not possible that p_e' is shorter, which means $p_e'=w(e)$. Therefore, by replacing e

by p_e' repeatedly until all the edges of the path are included in G', we can construct a path whose weight is equal to p so it is still a shortest path. ■

Lemma 2b. G' is minimal among all subgraphs that preserve at least one of the shortest paths between any pair of nodes.

Proof: Assume there exists a subgraph $G''=(V'', E'',w'')$ which preserves at least one of the shortest paths between any pair of nodes and it is smaller than G'. That is, there exists an edge $(u,v) \in E'$ but $(u,v) \notin E''$. By the definition of the algorithm, if $(u,v) \in E'$, (u,v) is the only shortest path between u and v because the shortest distance after removing (u,v) is greater than $w(u,v)$. Since G'' does not preserve (u,v), it contradicts the assumption that G'' preserves at least one of the shortest path between any pair of nodes.■

3.2.2. GR-SP2 (Shortest Path for 2 types of nodes)

(1) Definition of GR-SP2

Given a graph G with nodes in two colors, assume an application is only interested in finding the shortest paths between any pair of nodes having the same color. Can we reduce G to a smaller (and smallest) graph G' which preserves all the shortest paths between any pair of nodes with the same color? We call this problem the GR-SP2 Problem.

(2) Use Case of GR-SP2 (U.S Election)

Consider a social network for an election campaign. As shown in Figure 7, each node in the social network represents a person, and the weight of each edge may be the effort required for one person to influence the other (and vise-versa). An application may be only interested in finding a shortest path between any pair of republicans (blue nodes) or democrats (red nodes). If

the edge weight is a contact, a republican (democrat) may want to find the cheapest path for contacting another republican (democrat).

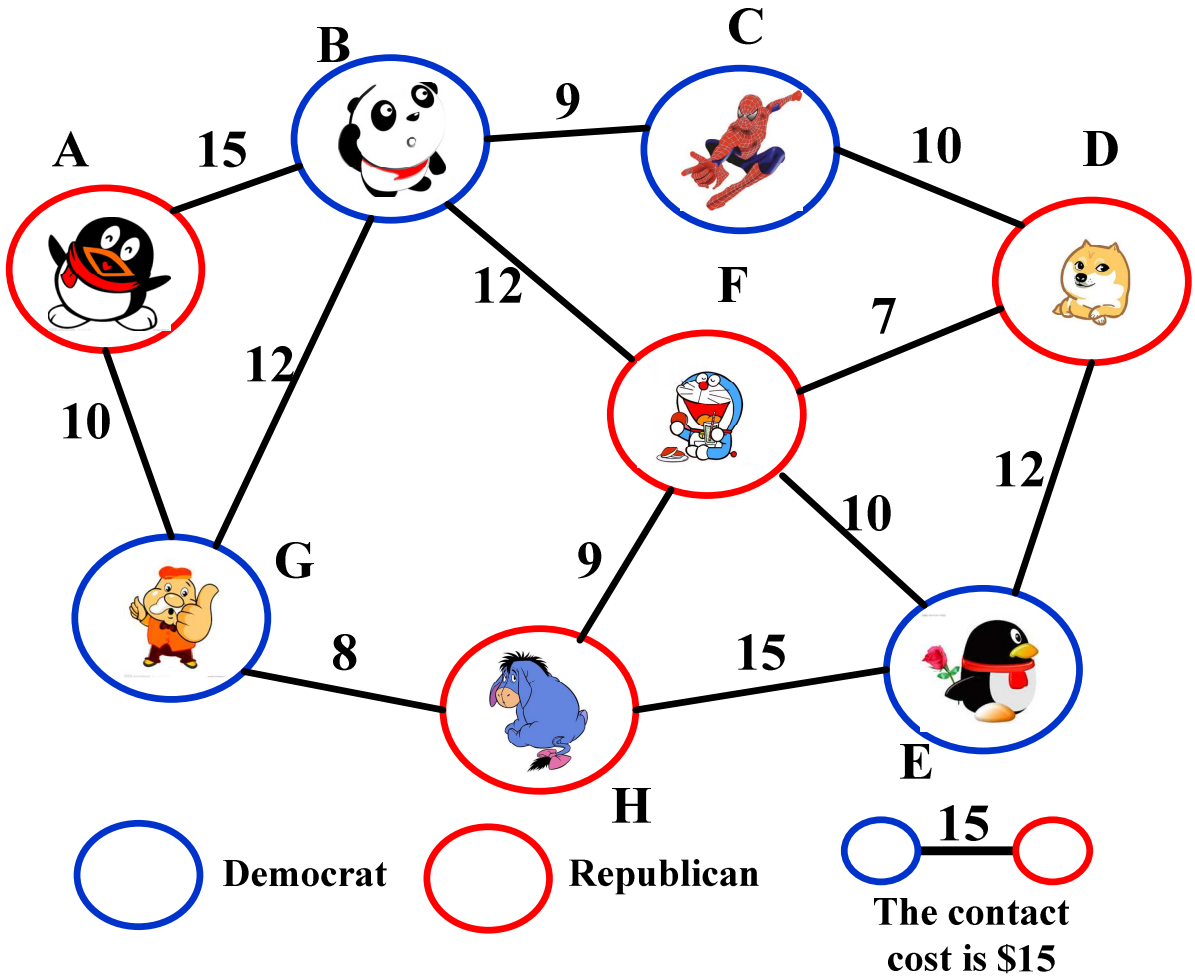


Figure 7. An example social network for U.S election

(3) Algorithms for GR-SP2

Our idea is to find a reduced graph *Gr* for the nodes in one color (say red), and a simplified graph *Gb* for the nodes in the other color (say black), and then merge them together.

Method to find *Gr* and *Gb*:

Step 1: Check if there is a direct edge between any two red nodes in *Gr* (or black nodes for *Gb*).

Step 2: For any two red nodes find the shortest paths between them.

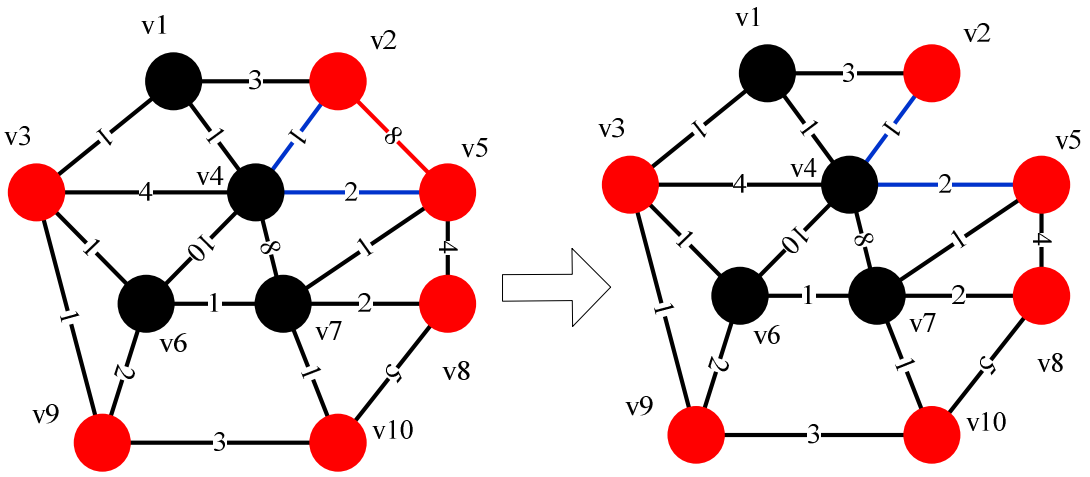
Step 3: Use a similar method as GR-SP to remove redundant edges.

Step 4: Remove any edge that does not belong to any shortest path.

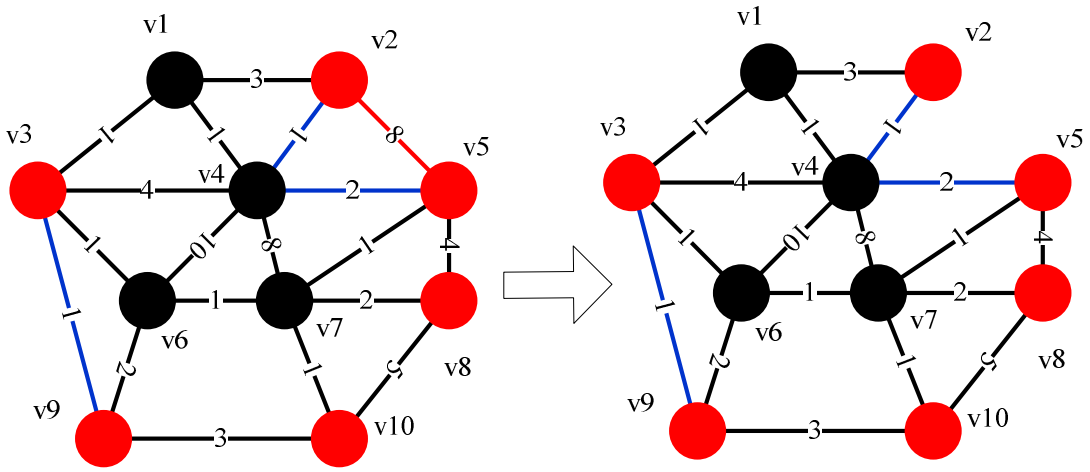
Figures 8-10 shows one example:

a) Compute *Gr*

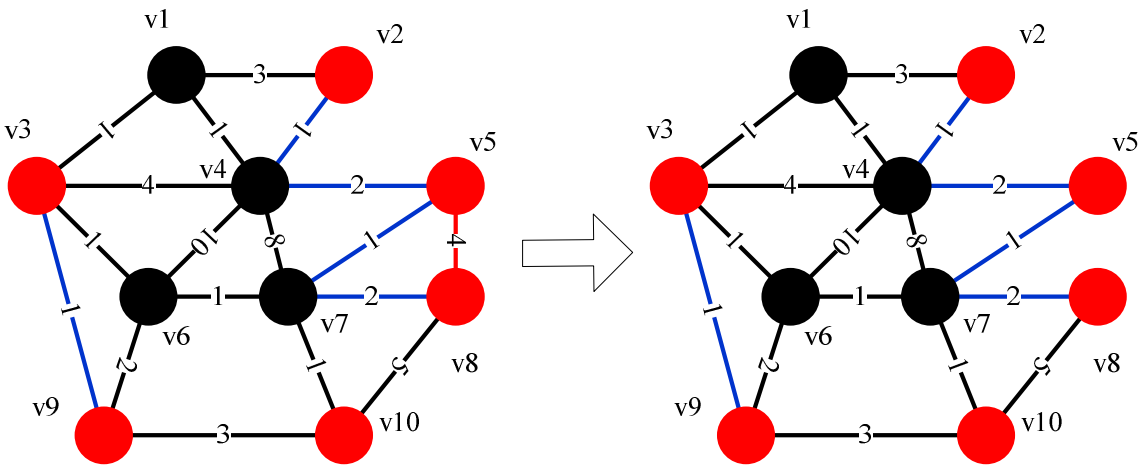
Figure 8 shows the reduction process to obtain *Gr* from *G*. In Figure 8, we can see (*v2, v5*), (*v3, v9*), (*v5, v8*), (*v8, v10*) and (*v9, v10*) connect two red nodes. So, we should focus on these edges and decide they should be retained in *Gr* by computing the shortest distance using Dijkstra or Floyd. Then, we remove any edge that does not belong to the shortest paths to obtain *Gr*. In Figure 8(a), $w(v2,v5)=8$; however $w(v2, v4) + w(v4,v5) = 1+2 = 3 < 8$. So (*v2,v5*) can be deleted. In Figure 8(b), $w(v3,v9) = 1$ which is the shortest path, so (*v3,v9*) should be retained. In Figure 8(c), $w(v5,v8)=4$ and $w(v5,v7)+w(v7,v8) = 1+2 = 3 < 4$. So (*v5,v8*) can be deleted. In Figure 8(d), $w(v8,v10)=5$ and $w(v8, v7) + w(v7,v10) = 2+1 = 3 < 5$. So (*v8,v10*) can be deleted. In Figure 8(e), $w(v9,v10)=3$ which is the shortest path, so (*v3,v9*) should be retained. Finally, in Figure 8(f), we delete all the redundant nodes and edges that do not belong to any shortest path between two red nodes, and obtain the sub-graph *Gr*.



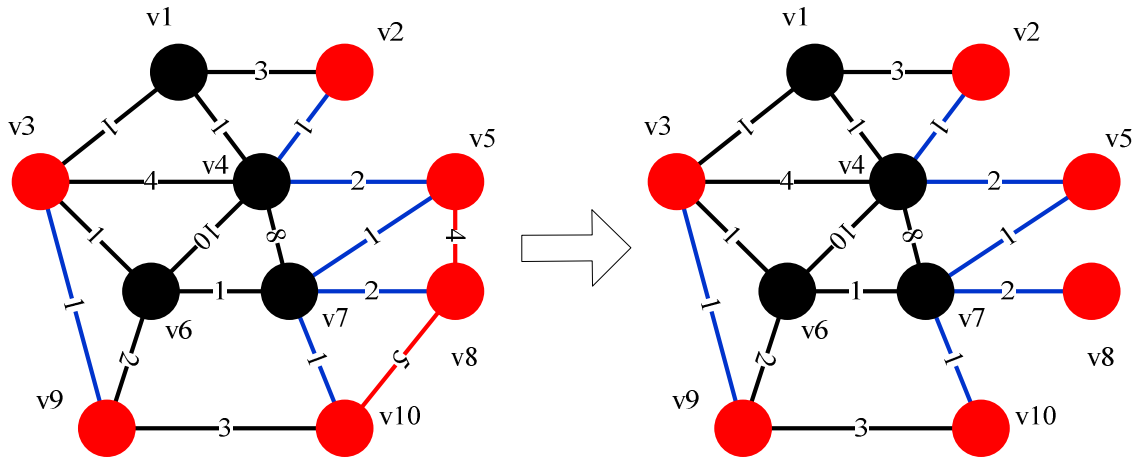
(a) Delete (v2,v5)



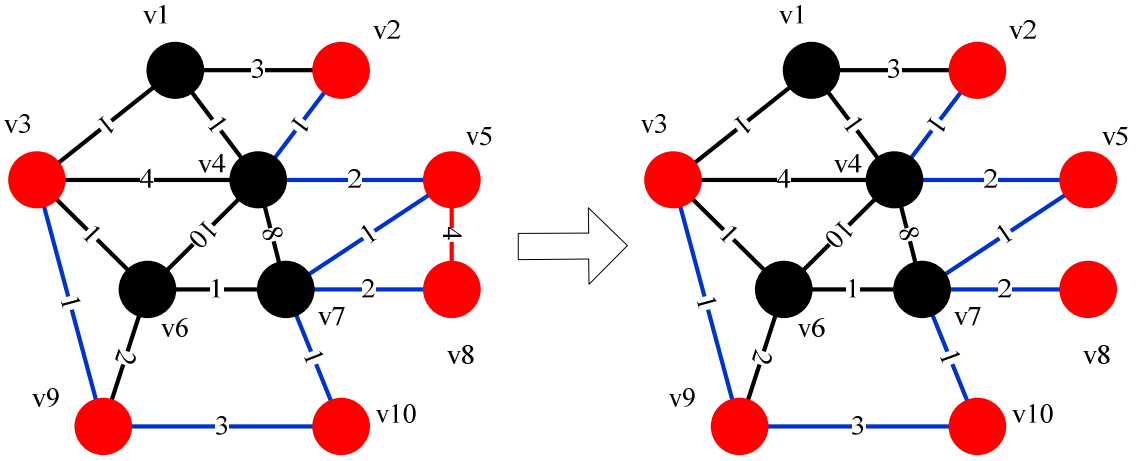
(b) Retain (v3,v9)



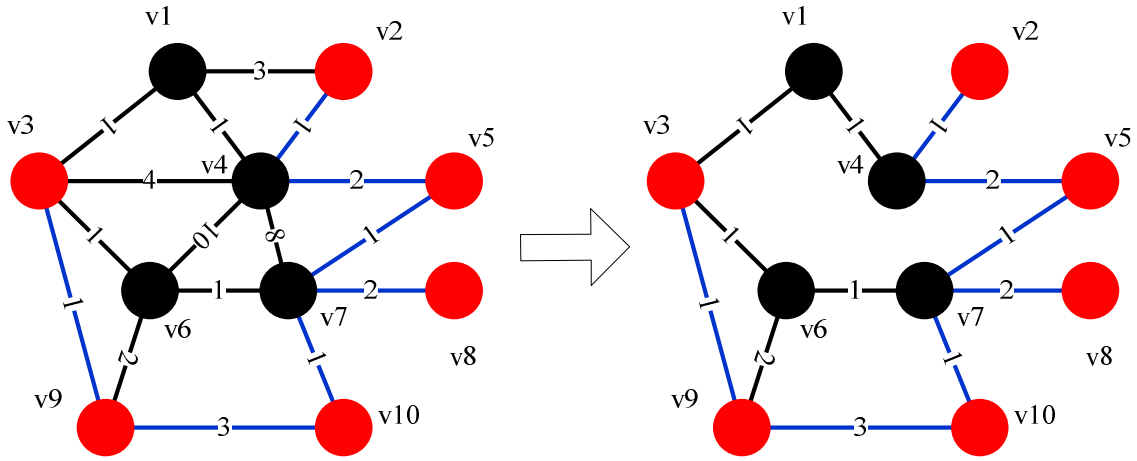
(c) Delete (v5,v8)



(d) Delete (v8,v10)



(e) Retain (v9,v10)



(f) Subgraph Gr

Figure 8. Compute Gr

b) Compute *Gb*

Figure 9 shows the reduction process to compute *Gb* using the same method.

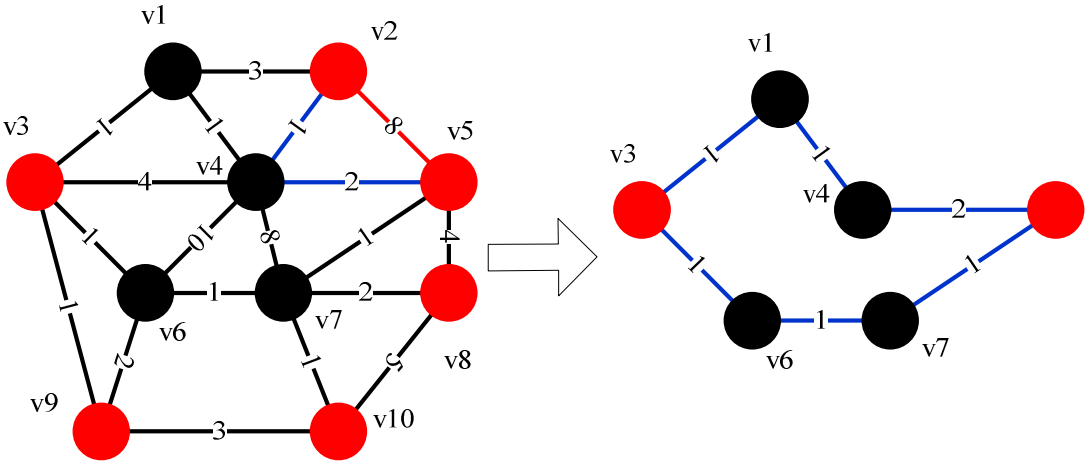


Figure 9. Compute Gb

c) Merge *Gr* and *Gb* to obtain *G'*

After *Gr* and *Gb* are obtained, we can merge them into a graph *G'*, as shown in Figure 10.

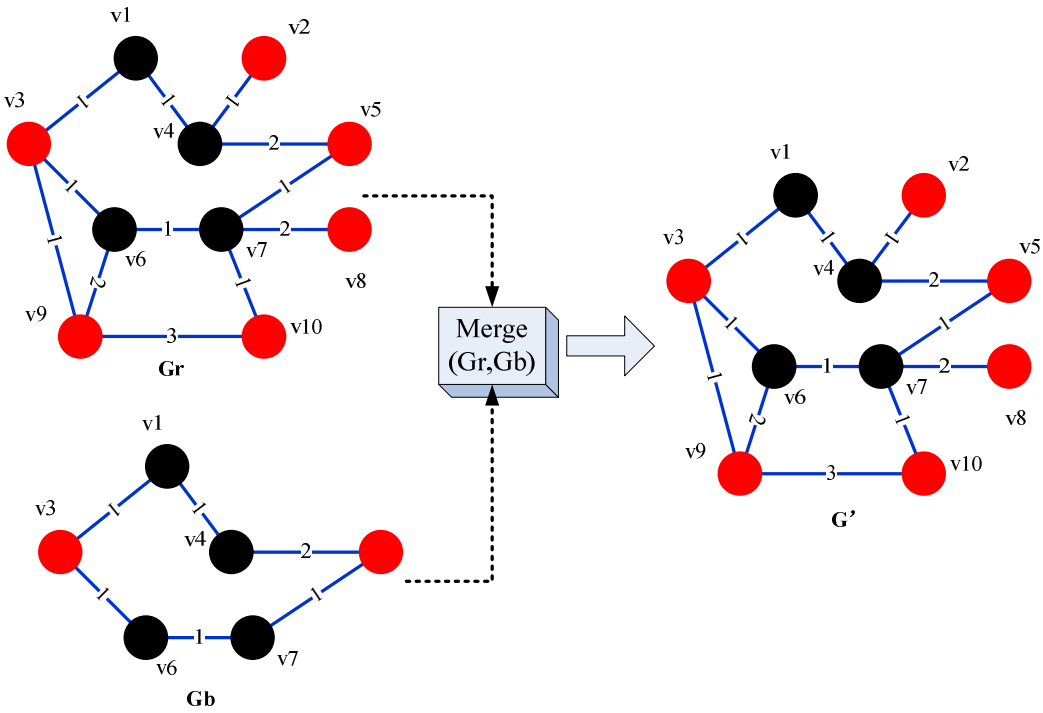


Figure 10. Compute G'

1) GR-SP2 algorithm

GR-SP2 algorithm (Find a minimal reduced graph that preserves all the shortest paths between any pair of nodes of the same colors, assuming there are 2 colors.)

Input: Graph *G*
Output: A reduced graph *G'* of the original graph *G*.

```
Initialize the original graph G.
Gr ← Gk_Subgraph_Algorithm(G, Red)
Gb ← Gk_Subgraph_Algorithm(G, Black)
G' ← Merge(Gr, Gb);
Return G'
```

2) Gk Subgraph Algorithm

Gk Subgraph Algorithm (Find a reduced subgraph that preserves all the shortest paths between any pair of nodes of the same color)

Input: Graph *G*, color *k*
Output: A reduced subgraph G_k of the original graph *G*.

```
G_k ←G
E_s ←∅
N ← Number_of_Nodes(G) //computing number of nodes in G.
 For i ← 0 to N do
    If i.color = k
      Dist[i] ← Dijkstra(G, i)//use Dijkstra algorithm to compute shortest paths from source i to other nodes.
    EndIf
    For j ← 0 to N do
       If j.color = k
         Insert all edges of shortest paths between i and j into E_s
         If the edge E(i,j) exists and E(i,j)>Dist[i][j]
             Remove E(i,j) from G_k
         EndIf
       EndIf
     EndFor
  EndFor
  For all E in G_k
    If E not in E_s
      Remove E from G_k
    EndIf
  EndFor
  Return G_k
```

The time complexity of the GR-SP2 algorithm is the same as that of the GR-SP algorithm.

(4) Theorems and Proof for GR-SP2

Theorem 2: G' is a minimal subgraph that preserves the shortest paths between any red nodes or any black nodes.

Proof: By Theorem 1, we can prove that the Gk subgraph algorithm generates the smallest subgraph that preserves all the shortest paths between two nodes of type *k*, because the algorithm is the same as GR-SP and it only removes the edges that do not belong to any shortest path tween two nodes of the same color. Since *Gr* preserves the shortest paths between any two red nodes, and *Gb* preserves the shortest paths between any two black nodes, the merged graph *G'* preserves the shortest paths between any pair of red nodes or black nodes, and it is the smallest among all such sub-graphs. ■

3.2.3. GR-SPk (Shortest Path for k types of nodes)

The GR-SP2 algorithm can be easily extended to graphs that consist of nodes in *k* colors.

Algorithm to find G_i, where $1 \leq i \leq k$

Step 1: Check if there is a direct edge between any two nodes of the same color in *Gi*.

Step 2: For any two nodes of the same color *i*, find the shortest path between them.

Step 3: Use GR-SP to remove the redundant edges.

Step 4: Remove any edge that does not belong to the shortest paths.

The GR- SPk algorithm can be described in the following.

GR-SPk algorithm (Find a minimum reduced graph that preserves all the shortest paths between any pair of nodes of the same color, where the number of colors is k and k>=3.)

Input: Graph *G*, number of colors *k*
Output: A reduced graph *G'*

1. Initialize the original graph *G*.
2. **For** $i \leftarrow 0$ to *k* **do**
3. $G_i \leftarrow$ Gk_Subgraph_Algorithm(*G*, *i*)
4. **EndFor**
5. $G' \leftarrow$ merge($G_1, G_2, \ldots, G_i$)
6. **Return** *G'*

The time complexity of the GR-SPk algorithm is the same as that of the GR-SP algorithm.

Theorem 3: *G'* is the minimal subgraph that preserves the shortest paths between any two nodes with the same color.

Proof: By Theorem 1, we know that each subgraph G_k is the minimal subgraph that preserves all the shortest paths between any two nodes in color k. By merging all the G_k*'s* the resulted graph *G'* preserves all the shortest paths between any two nodes in the same color. *G'* is minimal because all the subgraphs are minimal – removing any edge of *G'* means removing an edge from at least one of the subgraphs, and since each subgraph is already minimal we cannot remove any edge while still preserving all the shortest paths.■

3.3. QGRP for the Minimum Spanning Tree (MST) Problem

In this section, we present our approaches to QGRP based on the MST problem.

3.3.1. GR-MST

In this section, we will discuss how to compute a reduced GR-MST graph *G'* from a graph *G*.

(1) Definition of GR-MST

Given a graph *G*, assume an application is only interested in finding the minimum spanning trees, which can be solved by many existing algorithms, e.g., Prim's algorithm and Kruskal's algorithm. Can we reduce *G* to a smallest graph *G'* which preserves all the minimum spanning trees? We call this problem the GR-MST (Graph Reduction-Minimum Spanning Tree) Problem.

(2) Use Case of GR-MST (U.S Airport Network)

Figure 11 shows the U.S Airport Network. An application may only be interested in finding the MSTs for any subset of cities. The question is whether we can reduce the network that retains the same query results from the original network.

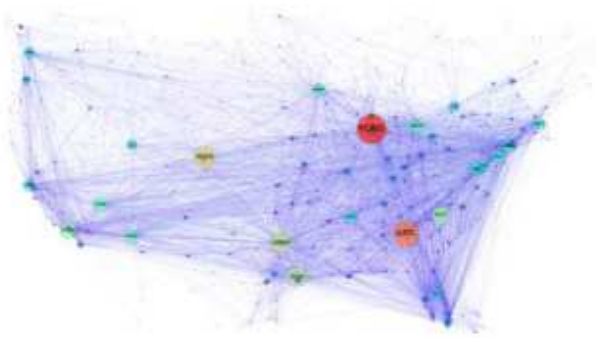


Figure 11. U.S Airport Network [61]

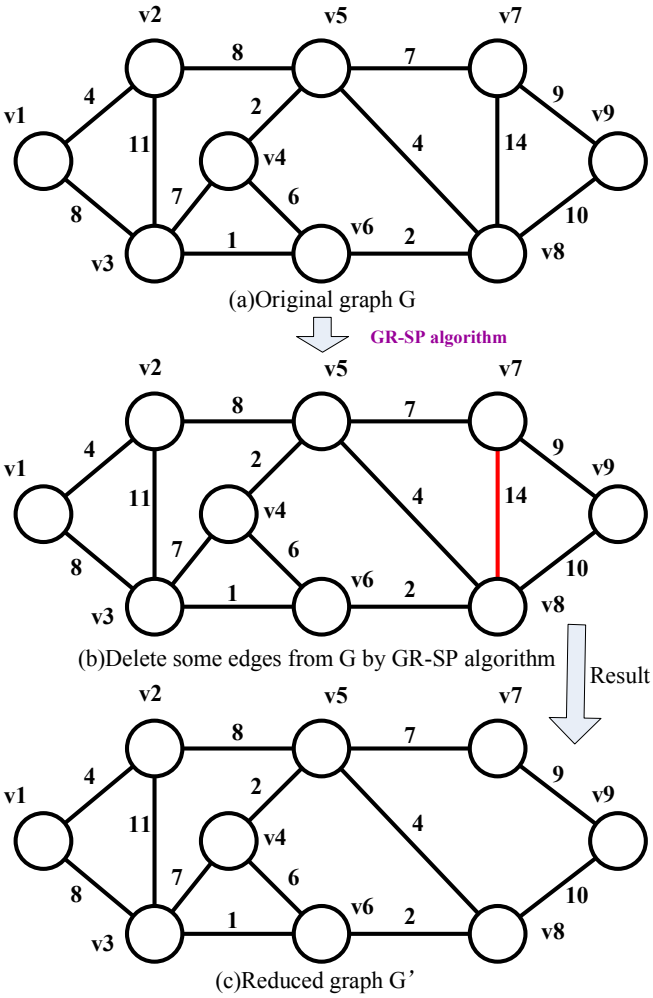
(3) Algorithms of GR-MST

We can apply the same idea as in the GR-SP algorithm to solve the GR-MST problem - In an MST any edge between two nodes must be the shortest path also, because if it is not the shortest, the total weight of the MST will not be the minimum.

The time complexity is the same as that of the GR-SP algorithm.

GR -MST algorithm (Find a minimal reduced graph that preserves all the MSTs among any subset of nodes.)
Input: Graph *G*
Output: A reduced graph *G'* from *G*
1. Initialize the original graph *G*.
2. *G'* ← GR-SP(*G*) //use the GR-SP algorithm
3. **Return** *G'*

Figure 12 shows an example of the reduction process. In Figure 12(b), we find that the edge *(v7,v8)* has a weight 14 but there is a shorter path *(v7,v5,v8)* whose weight is 4+7=11, so we remove the edge. For all the other edges, we are not able to find any shorter path, so we keep them. Finally we obtain the reduced graph in Figure 12(c).



(a)Original graph G

(b)Delete some edges from G by GR-SP algorithm

(c)Reduced graph G'

Figure 12. GR-MST

(4) Theorems and Proof for GR-SP

Theorem 4. The GR-MST algorithm produces a minimal subgraph *G'* from *G* that preserves all the minimum spanning trees of any set of nodes that belong to *G*.

Proof: The GR-MST algorithm is the same as the GR-SP algorithm, and the theorem can be proved by the following lemmas 3 and 4.

Lemma 3. The GR-SP algorithm produces a subgraph *G'* from *G* that preserves all the minimum spanning trees among any set of nodes that belong to *G*.

Proof: Assume we have a minimum spanning tree *T* of a set of nodes *N* which belong to *G*, and there exists an edge *E** in *T* that is not in *G'*. By the definition of the GR-SP algorithm, if *E** is in *G* but not in *G'*, there exists another path *P'* connecting the nodes of *E**, and all the edges of *P'* belong to *G'*. Therefore, we can replace *E** by *P'* from *T*, creating a spanning tree with the total weight smaller than *T*. That causes a contradiction which means the assumption that *T* is minimum is false; so G' preserves all of the minimum spanning trees of any set of nodes that belong to G. ■

Lemma 4. *G'* is minimal among all subgraphs that preserve all the minimum spanning trees among any set of nodes that belong to *G*.

Proof: Assume there exists a subgraph $G''= (V'', E'',w'')$ which preserves all the minimum spanning trees among any set of nodes and it is smaller than *G'*. That is, there exists an edge $(u,v) \in E'$ but $(u,v) \notin E''$.

Based on the GR-SP algorithm, assume $(u,v) \in E'$, $w(u,v) = \delta_G(u, v)$ so (u,v) is one of the shortest paths between nodes *u* and *v*. Assume the set of nodes we want to find the minimum spanning trees only consists of nodes *u* and *v*. Apparently one of the minimum spanning trees covering nodes *u* and *v* consists of only edge (u,v). Since *G''* does not preserve (u,v), it contradicts the assumption that *G''* preserves all the minimum spanning trees among any set of nodes that belong to *G*. ■

Figure 13 shows the MST trees computed from a graph *G* and Figure 14 shows the MST trees computed from the reduced graph *G'*. We can see the results are exactly the same.

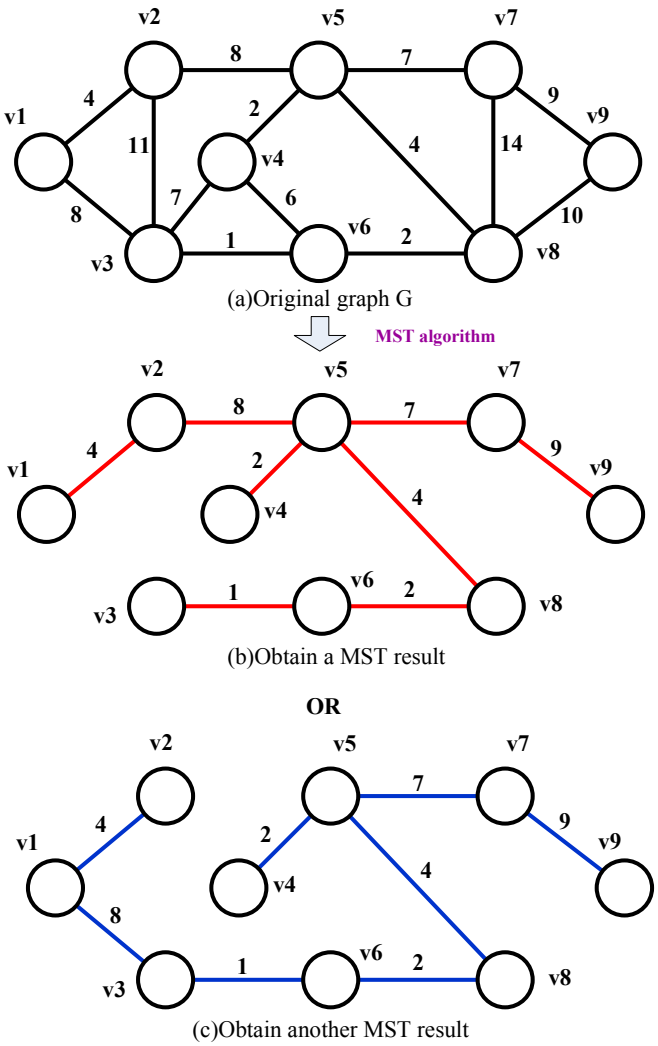


Figure 13. Compute MST from G

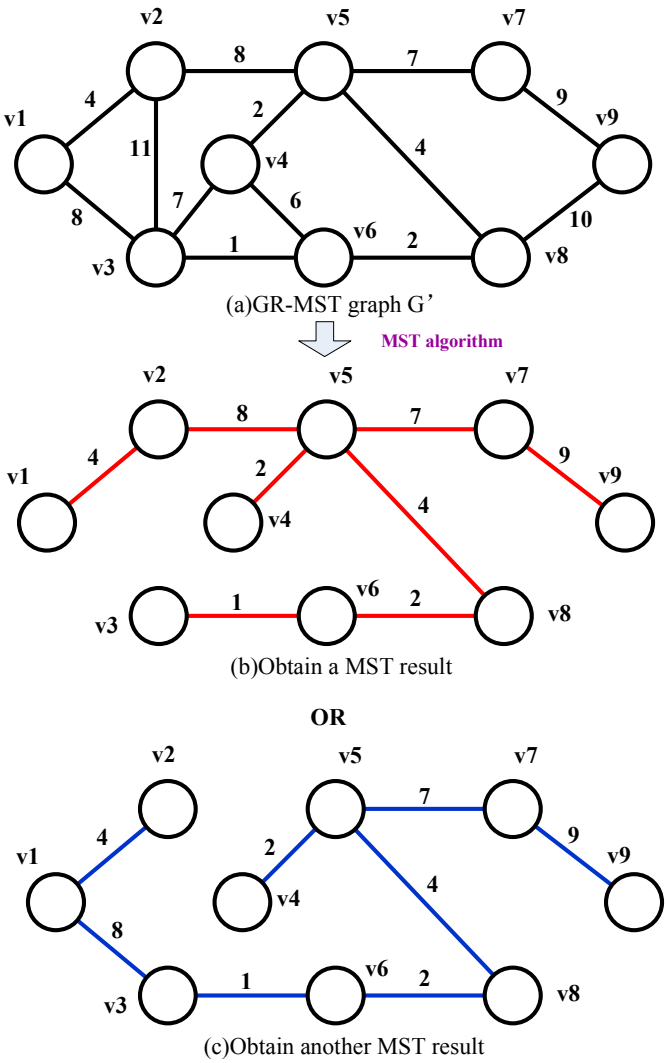


Figure 14. Compute MST trees from GR-MST graph *G'*

3.3.2. GR-WC-MST

In this section, we discuss how to compute a reduced GR-WC (Weight Constrained)-MST graph *G'* from a graph *G*.

(1) Definition of GR-WC-MST

Given a graph *G*, assume an application is only interested in finding the minimum spanning trees among any set of nodes with a weight constraint on the edges (i.e., the weight of each edge

has to be less than or equal to a constant *k*). Can we reduce *G* to a smallest graph *G'* which preserves all the minimum spanning trees with the weight constraint? We call this problem the GR-WC-MST (Graph Reduction-Weight Constraint-Minimum Spanning Tree) Problem.

(2) Use Case of GR-WC-MST (Delivery Network)

Figure 15 shows the delivery network of a transportation company, where each node designates a station, each link designates a transportation link, and its weight is the cost of the transportation. The company may only be interested in finding the minimum spanning trees among the nodes that are connected by edges whose weights are less than or equal to a constant.

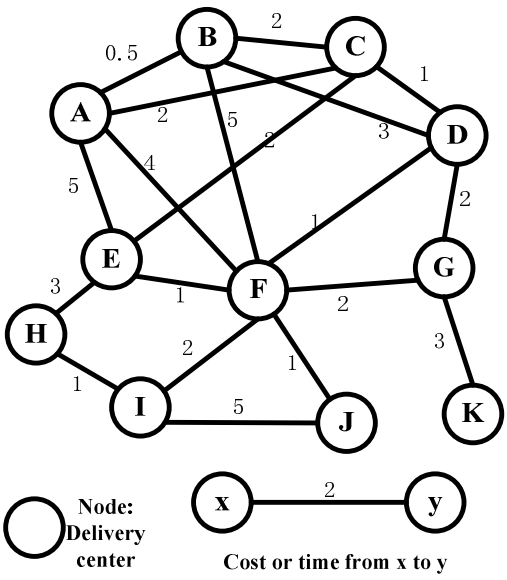


Figure 15. An example delivery network

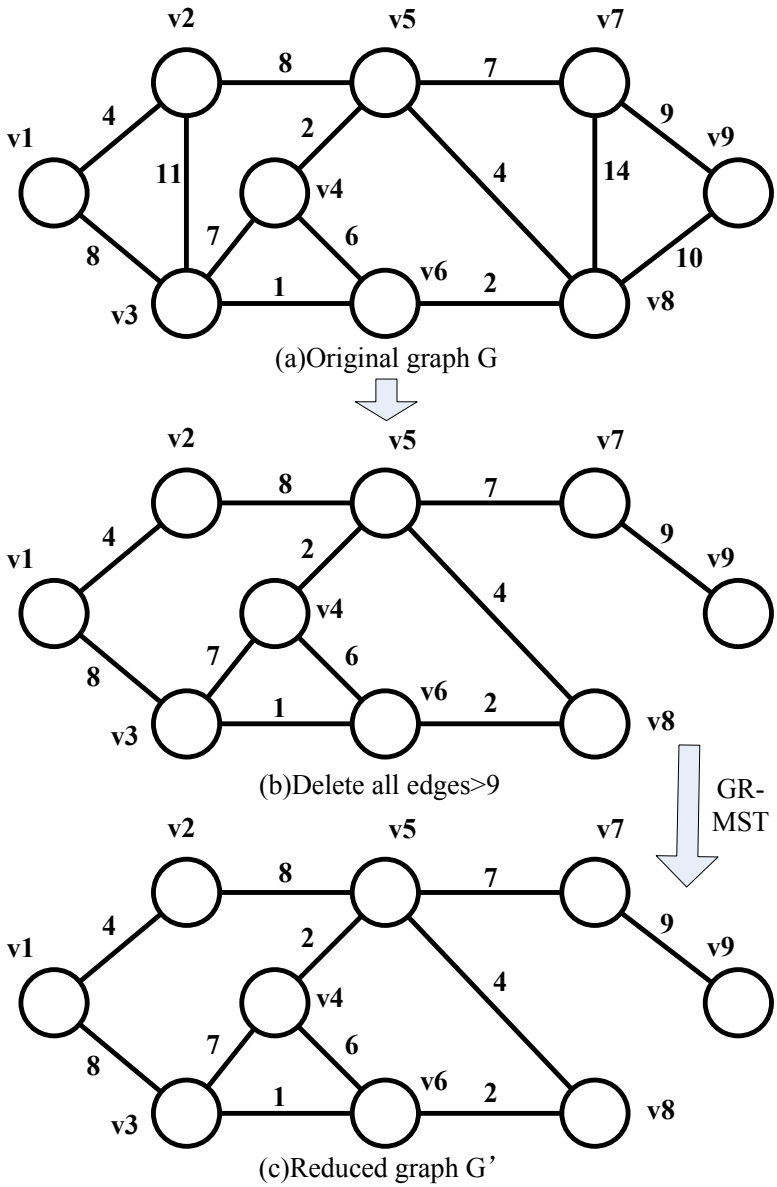
(3) Algorithms of GR-WC-MST

The GR-WC-MST algorithm can be summarized into the following two steps.

Step 1: Remove all the edges that exceed the constraint

Step 2: Apply RG-MST

Figure 16 shows the process of GR-WC-MST for an example network, where the weight of each edge should be less than or equal to 10. Figure 16(a) shows the original graph *G*. Figure 16(b) shows the result after removing all the edges that exceed the weight constraint. Figure 16(c) shows the result *G'* after applying the GR-MST algorithm.



(a)Original graph G

(b)Delete all edges>9

(c)Reduced graph G'

Figure 16. GR-WC-MST

The GR-WC-MST algorithm is described in the following. We first remove all the edges whose weight is more than *K*. Then we apply the GR-MST algorithm. Since the edges that violated the weight constraint are already removed, those edges will not be included in the reduced graph. Note that we can generalize the weight constraint (e.g., greater than or equal to *K*, between *m* and *n*), and the algorithm will still work.

GR –WC-MST algorithm (Find a smallest reduced graph *G'* from the original graph *G* that preserves all the MSTs among any subset of nodes of G where the weight of each link in G' is less than or equal to a constant *K*)

```
G' ← G
For e ← 0 to |E| do
   If w(e) < K, remove edge e from G'
EndFor
G' ← GR-MST(G')
```

The time complexity of the GR-WC-MST algorithm is the same as that of the GR-MST algorithm.

(4) Theorems and Proof for GR-WC-MST

Theorem 5. The GR-WC-MST algorithm produces a minimal subgraph *G'* from *G* that preserves all the weight-constrained minimum spanning trees among any set of nodes belonging to *G*.

Proof: We remove all the edges that have a weight violating the weight constraint, so the graph only contains spanning trees that do not violate the weight constraint, which also contains all the weight-constrained MSTs. Then we apply the GR-MST algorithm to generate *G'*, which reduces the graph but preserves all the MSTs; and these MSTs comply with the weight constraint. By Theorem 4, *G'* is minimal and all the MSTs are preserved, therefore Theorem 6 is proved. ■

3.3.3. GR-DC-MST

In this section, we discuss how to compute a reduced GR-DC (Degree Constrained)-MST graph *G'* from a graph *G*.

(1) Definition of GR-DC-MST

Given a graph *G*, assume an application is only interested in finding the minimum spanning trees among any set of nodes with a degree constraint. Can we reduce *G* to a smallest graph *G'* which preserves all the minimum spanning trees with the degree constraint? We call this problem the GR-DC-MST (Graph Reduction-Degree Constraint-Minimum Spanning Tree) Problem.

(2) Use Case of GR-DC-MST (Urban Road Network)

Consider a graph (shown in Figure 17) that corresponds to the design of an urban road network among a set of locations (nodes) where each edge is a road connecting two locations and the weight of an edge is the cost to build a road. It may be desired that no more than a certain number of roads (e.g., 4 roads) are allowed to meet at an intersection. We also want the overall cost of building the road network to be minimal. Therefore this application corresponds to a degree-constrained MST problem, and we want to compute the smallest reduced graph for the application.

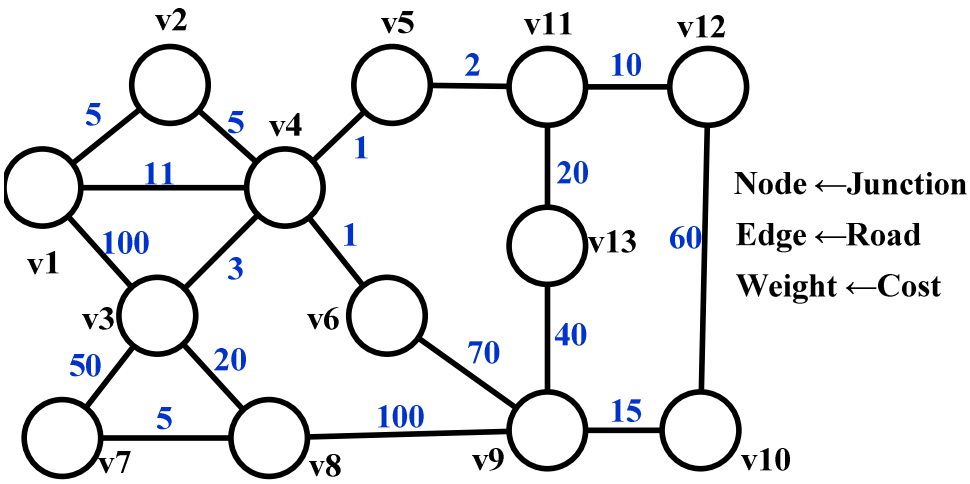


Figure 17. An urban road network

(3) Algorithms of GR-MST

The main idea behind the GR-DC-MST algorithm is as follows. We use the GR-MST algorithm to compute a reduced graph at the beginning. In the reduced graph, if there is any node that has more than *k* edges connected to it, where *k* is the degree constraint, we cannot simply remove the node and its incident edges. The reason is that some of these edges may be necessary to be included in a spanning tree if there is no other ways to connect the neighbors of the node that violates the degree constraint. Therefore, we find the shortest paths to connect its neighbors and add them to the reduced graph as alternative paths for finding the spanning trees.

The GR-DC-MST Algorithm is described as the following:

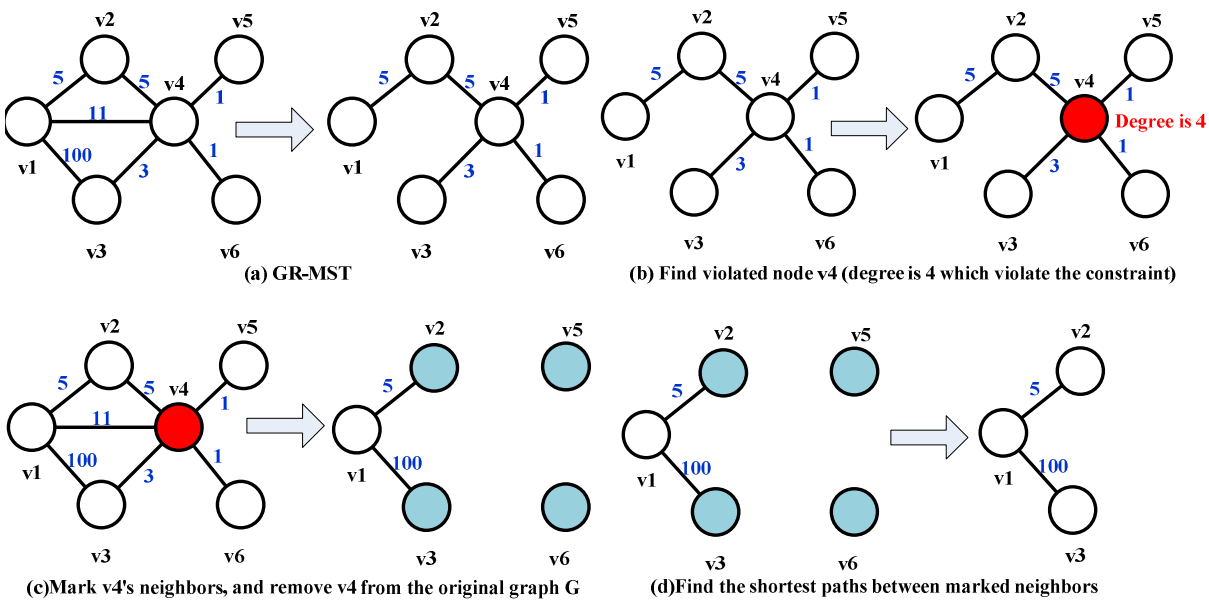
GR-DC-MST algorithm (Find a reduced graph *G'* from a graph *G* which preserves the MSTs among any subset of nodes with degree constraint *K*)

```
G' ← GR-MST(G) //Compute a reduction graph by the GR-MST algorithm.
G'' ← G
N_v ← Ø //initialize the condition violating nodes' set as an empty set
For i ← 0 to |N| do
  If node_degree(i) >K
    Insert node i to N_v
    //add the condition violating node to the set of condition violating nodes;
  EndIf
  For all node n in N_v do
```

```
        N_b ← n's neighbors in G''
        remove all the edges connected n from G''
        For each pair of nodes in N_b, find the shortest paths between them
        Add these paths to G'
      EndFor
    EndFor
    Return G'
```

The time complexity of the GR-DC-MST algorithm is $N^2 * T(\text{Dijkstra})$.

Figure 18 shows an example of the GR-DC-MST reduction process (where the degree constraint is 3). Figure 18(a) shows the result after executing the GR-MST(SP) algorithm. Figure 18(b) shows the process of finding the condition violating nodes. Figure 18(c) shows the process of removing the edges which are connected to the condition violating nodes. Figure 18(d) shows the process of finding the shortest paths between any pair of nodes that neighbor a condition violating node. Figure 18(e) shows the process of obtaining the GR-DC-MST graph G'. In Figure 18(f) we are able to find the DC-MST in *G'*, which cannot be found in the reduced graph after applying GR-MST to Figure 18(a).



(a) GR-MST

(b) Find violated node v4 (degree is 4 which violate the constraint)

(c)Mark v4's neighbors, and remove v4 from the original graph G

(d)Find the shortest paths between marked neighbors

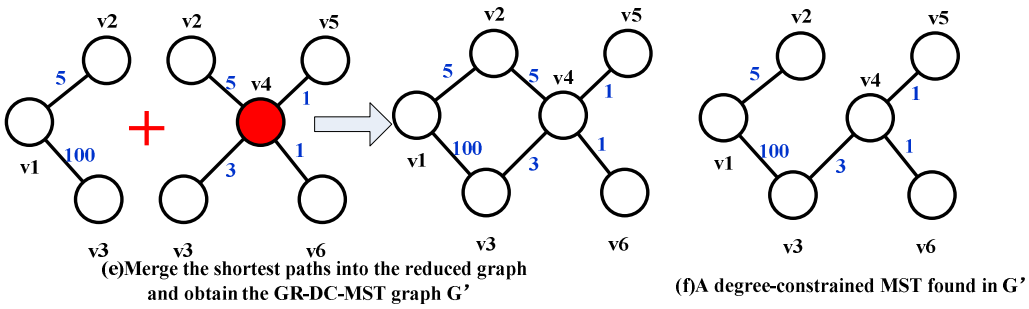


Figure 18. GR-DC-MST

(4) Theorems and Proof for GR-DC-MST

Theorem 6. The GR-DC-MST algorithm produces a subgraph *G'* from *G* that preserves the degree-constrained minimum spanning trees among any set of nodes belonging to *G*.

Proof: For any node in the reduced graph whose degree is more than *k*, we find the shortest paths between all pairs of its neighbors that do not include the violation node and add them into the graph. With such, we can always reduce the degree of that node by choosing an alternative shortest path between any pair of its neighbors. Therefore, if a degree-constrained MST exists, we are able to find it in the reduced graph. Note that, however, finding DC-MST in *G'* is still an NP-hard problem [63], as we do not know which combination of the alternative paths are the best. Testing all possible paths is still a combinatorial problem. Also note that, since we add all the alternative paths back to the reduced graph, *G'* may not be minimal as some alternative paths may not be necessary. ■

3.4. Synthesizing Reduction Algorithms for Integrated Problems

In the previous sections, we described the query-based graph data reduction algorithms for some graph problems. These problems mainly focus on one single optimization objective, or two objectives with one as the main goal and the other as a constraint. However, some applications may need to solve a complex graph problems which include the main goal and more than one

constraint. Developing every graph reduction algorithm for each of such complex graph problems is expensive, as the number of graph problems grows exponentially with the number of constraints. Therefore, it is desired to synthesize the existing graph reduction algorithms to generate a reduction graph for a new, complex, and integrated graph problem.

3.4.1. Stages in Graph Reduction Algorithms

The graph reduction algorithms we have discussed can be separated into three stages: pre-processing, the main algorithm, and post-processing, where pre-processing and post-processing are optional.

(1) Pre-processing

We define the pre-processing stage as the stage to remove all the edges that violate a constraint. That is, given the original graph G=(V,E), the pre-processing stage is to generate a subgraph $G_{pre}=G/E_{pre}$, where E_{pre} is the set of the edges and nodes that violate the condition, so that for any query q, $q(G)=q(G_{pre})$.

An example algorithm that has the pre-processing stage is GR-WC-MST. It first removes all the edges whose weight is greater than the weight constraint K; it then applies the GR-MST algorithm. Figure 19 shows the concept. The result of applying GR-MST to the preprocessed graph prevents the reduced graph from containing any unwanted edges, so we can compute a more compact graph which still contains all the WC-MSTs.

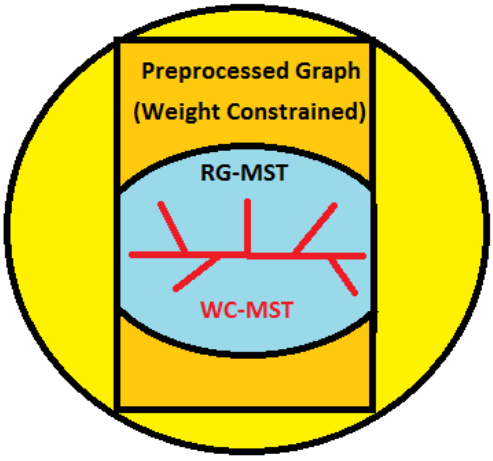


Figure 19. Pre-processing

(2) The main algorithm

The main algorithm applies the basic graph reduction method, which is usually an algorithm focused on a single objective without conditions. In order to do the synthesis, we require the reduced graph generated by the main algorithm to be a subgraph of the original graph. That is, let $G_m=(V_m, E_m)$ be the reduced graph generated by the main algorithm, $V_m = V$ and $E_m \subseteq E$.

(3) Post-processing

The post-processing stage is the stage after the main algorithm; it may add some edges E_{post} to restore the information needed to solve a problem with conditions. In general $E_{post} \subseteq E\text{-}E_{pre}$ because we should not add back the edges that were already removed by the pre-processing stage. Any query on the reduced graph after the post-processing should obtain the same result as the original graph, that is, for any query q, $q(G) = q(G_m \cup E_{post})$.

An example algorithm that has the post-processing stage is GR-DC-MST. It first applies the GR-MST algorithm to the original graph, and then adds the edges to form alternative paths between the nodes that neighbor the nodes that violate the degree constraint. Figure 20 shows the concept.

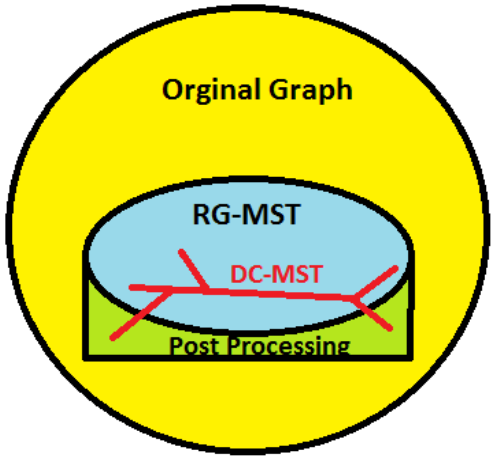


Figure 20. Post-Processing

3.4.2. Use Case of Reduction Algorithm Synthesis

Consider a wireless sensor network as an example. A wireless sensor node is usually composed of one or more sensors, a microcontroller, a wireless receiver and a power source, usually a battery. Figure 21 shows one example.

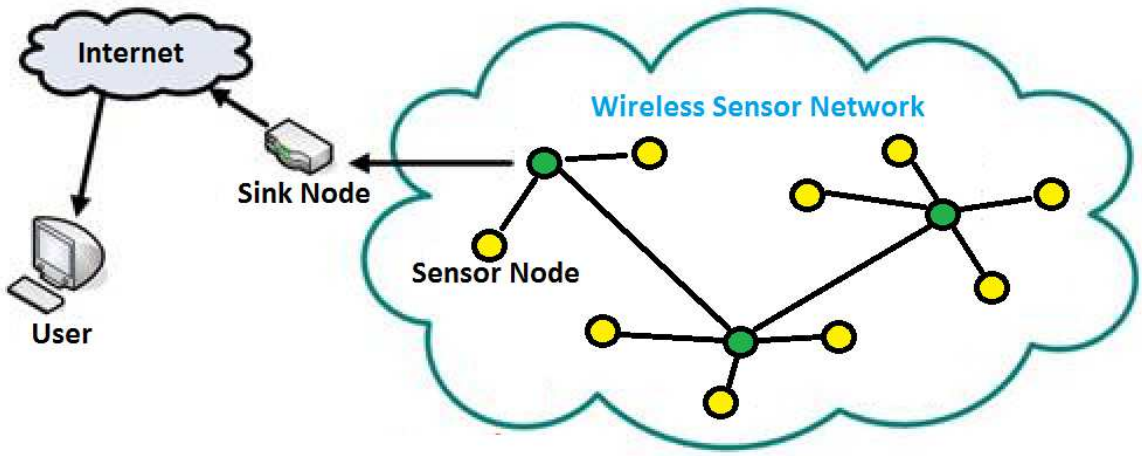


Figure 21. A wireless sensor network

In order to reduce the cost, there are limitations when designing a single wireless sensor; these include computation power, network bandwidth, and battery. It is common that the wireless sensors are randomly deployed to the environment, and after deployment it is difficult or not

possible to replace a battery of a node. Therefore, the lifetime of a wireless sensor network depends on the batteries. To design a communication network structure is important in order to save the energy as well as to receive the sensor results efficiently.

For reducing the overall energy consumption as well as receiving the sensor's data in a timely manner, a network is normally designed as a spanning tree that minimizes the overall data transition time. There are usually some sensor nodes collecting data from its neighbors as the head of a cluster, and the degree of these nodes should not exceed a pre-defined constraint K, as too many communication links and too much computation on a single node may consume too much energy that can reduce the lifetime. So it is a DC-MST problem. The data transition time between two nodes should not exceed some constraint also in order to make sure the nodes are synchronized and the data are up to date, which is a WC-MST problem. Due to the limitation of a wireless sensor, if the distance between two nodes is too far then the communication between them is not reliable and may suffer some data loss, which is another WC-MST problem on distance (reliability). As a result, the problem of designing a wireless sensor network is an integration of three existing problems in the context of this thesis, and we want to synthesize an algorithm from the reduction algorithms discussed in the previous sections.

3.4.3. Algorithm for Reduction Algorithm Synthesis

If a graph problem has multiple conditions (objectives), and the reduction algorithms for different objectives satisfy the definition of the three stages, i.e., pre-processing, the main algorithm, and post-processing, we are able to combine them in the three stages.

If we are only combining two algorithms, where one has the pre-processing stage, one has the post processing stage, and the main algorithm is the same, then we can easily combine them by

merging them in the order of the three stages. An example is to combine DC-MST and WC-MST, the combined GR-WC-DC-MST algorithm is described in the following:

GR-WC-DC-MST algorithm (Synthesize GR-WC-MST and GR-DC-MST)

1. Apply step 2-4 in GR-WC-MST to G and obtain G_{pre}
2. $G_m \leftarrow$ GR-MST(G_{pre})
3. Apply step 2-14 in GR-DC-MST to G_m and obtain G_{post}
4. **Return** G_{post}

Figure 22 shows the concept of combining the WC-MST and DC-MST algorithms into the GR-WC-DC-MST algorithm.

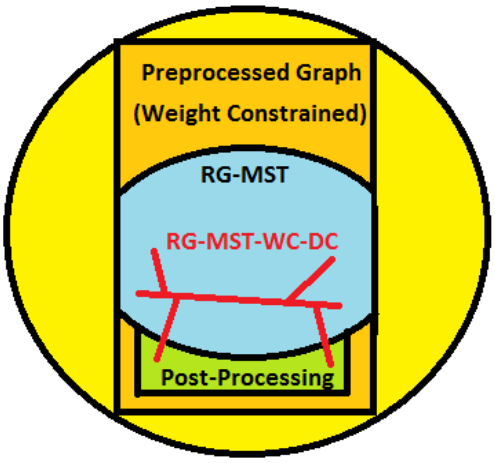


Figure 22. Combined GR-WC-DC-MST algorithm

If there are multiple algorithms, and they are only different in the pre-processing, the main algorithm, and the post-processing stages, we can synthesize them in each stage, and then combine the result of them in that order. The following shows the algorithm for synthesis in different stages.

(1) Pre-processing

Let E_{pre_1}, E_{pre_2}, $\cdots$, E_{pre_N} be the edge set that violate condition 1 to condition N, then the combined G_{pre} is $G/(E_{pre\_1} \cup E_{pre\_2} \cup \cdots \cup E_{pre\_N})$.

The Pre-processing synthesis algorithm is described in the following.

Pre-processing synthesis algorithm (Synthesize the pre-processing stages of multiple reduction algorithms)

Input: Graph G, set of edges obtained in the pre-processing stage for different algorithms E_{pre_1}, E_{pre_2}, $\cdots$, E_{pre_N}.
Output: synthesized graph G_{pre}

1. $G_{pre} \leftarrow G$
2. $E_{pre} \leftarrow \emptyset$
3. **For** $i \leftarrow 0$ to N **do**
4. $E_{pre} \leftarrow E_{pre} \cup E_{pre\_i}$
5. **EndFor**
6. Remove E_{pre} from G_{pre}
7. **Return** G_{pre}

Theorem 7. The Pre-processing synthesis algorithm produces a subgraph G_{pre} from G that preserves the result of the corresponding graph problem with multiple conditions if there exists an algorithm with he pre-processing stage for each condition.

Proof: Let define that for any query q on graph G, the result $q(G) = f(R_q)$ where R_q is a reduced subgraph of G based on query q, and f is the function that computes the result from R_q. Since we know that the query result cannot violate any condition, the set of edges that are found to violate a condition in the preprocessing stage will not be in R_q. That is, for condition 1, $E_{pre\_1} \not\subset R_q$ and $q(G)=q(G/E_{pre\_1})$. Similarly, $E_{pre\_2} \not\subset R_q$, … , and $E_{pre\_N} \not\subset R_q$. Therefore, $q(G) = q(G/(E_{pre\_1} \cup E_{pre\_2} \cup \cdots \cup E_{pre\_N}))$. ■

(2) The Main Algorithm

Usually the algorithms for different conditions imposed on the same problem will use the same main algorithm. However, if there are different main algorithms, we can still synthesize them by the main-algorithm synthesis algorithm.

Let G_{m_1}, G_{m_2}, $\cdots$, G_{m_N} be the reduced graphs generated by different main algorithms, the combined G_m will be $G_{m\_1} \cup G_{m\_2} \cup \cdots \cup G_{m\_N}$.

The Main-algorithm synthesis algorithm is described in the following.

Main-algorithm synthesis algorithm (Synthesize the main algorithms from multiple reduction algorithms)
Input: Graph G_{pre}, main algorithm M_1 to M_N.
Output: synthesized graph G_m
1. $G_m \leftarrow \emptyset$
2. **For** $i \leftarrow 0$ to N **do**
3. $G_{m\_i} \leftarrow M_i(G_{pre})$
4. $G_m \leftarrow G_m \cup G_{m\_i}$
5. **EndFor**
6. **Return** G_m

Theorem 8. The Main-algorithm synthesis algorithm produces a subgraph G_m of G that preserves the answers to the original queries. If we replace the result of each main algorithm G_{m_N} we can still obtain the correct result.

Proof: Let q be the query for the multiple conditions problem. $q(G)= q(G_{m\_1} \cup E_{post\_1})$ by the definition of the algorithm for condition 1, and $R_q \subseteq G_{m\_1} \cup E_{post\_1}$. Similarly $R_q \in G_{m\_2} \cup E_{post\_2}$, $\cdots$ and $R_q \subseteq G_{m\_N} \cup E_{post\_N}$, where R_q is the subgraph of query result of G. Therefore, $R_q \subseteq E_{post\_1} \cup G_{m\_1} \cup \cdots \cup G_{m\_N}$, $R_q \subseteq E_{post\_2} \cup G_{m\_1} \cup \cdots \cup G_{m\_N}, \cdots$, $R_q \subseteq E_{post\_N} \cup G_{m\_1} \cup \cdots \cup G_{m\_N}$

Therefore, let $G_m= G_{m\_1} \cup G_{m\_2} \cup \cdots \cup G_{m\_N}$, we can derive $q(G) = q(G_m \cup E_{post\_1}) = q(G_m \cup E_{post\_2}) =\cdots= q(G_m \cup E_{post\_N})$, so we can synthesize the main algorithm by the combined result G_m. ■

(3) Post-Processing

Let E_{post_1}, E_{post_2}, $\cdots$, E_{post_N} be the edge set that is added back in the post-processing stage of each algorithm, respectively, then the combined E_{post} will be $E_{post\_1} \cap E_{post\_2} \cap \cdots \cap E_{post\_N}$.

The Post-processing synthesis algorithm is described in the following.

Post-processing synthesis algorithm (Synthesize the Post-processing stage from multiple reduction algorithms)
Input: Graph G_m, edges from the post-processing stage for different algorithms E_{post_1}, E_{post_2}, $\cdots$, E_{post_N}.
Output: synthesized graph G_{post}
1. $G_{post} \leftarrow G_m$
2. $E_{post} \leftarrow \emptyset$
3. **For** $i \leftarrow 0$ to N **do**
4. $E_{post} \leftarrow E_{post} \cap E_{post\_i}$
5. **EndFor**
6. Insert E_{post} to G_{post}
7. **Return** G_{post}

Theorem 9. The Post-processing synthesis algorithm produces a subgraph G_{post} of G that preserves the result of the graph problem for multiple conditions when there exists an algorithm with the post-processing stage for each condition.

Proof: We know that in set theory, if $a \in X$ and $a \in Y$, $a \in X \cap Y$. We also proved that, $R_q \subseteq E_{post\_1} \cup G_m$, $R_q \subseteq E_{post\_2} \cup G_m, \cdots$, $R_q \subseteq E_{post\_N} \cup G_m$

Therefore, $R_q \subseteq (E_{post\_1} \cup G_m) \cap (E_{post\_2} \cup G_m) \cap \cdots \cap (E_{post\_n} \cup G_m)$

$R_q \subseteq G_m \cup (E_{post\_1} \cap E_{post\_2} \cap \cdots \cap E_{post\_n})$

Therefore, $q(\mathrm{G})=q(G_m \cup (E_{post\_1} \cap E_{post\_2} \cap \cdots \cap E_{post\_n}))=\mathrm{q}(G_{post})$. ■

Please note that we do not claim the combined reduced graph from our synthesis algorithms to be minimal. However, we know the combined graph will not be larger than the original graph, since we only perform set operations on the subgraphs and edges from the original graph.

3.4.4. Synthesis in general

It is possible that some reduction algorithms do not satisfy the definition of the three stages. For example, a Gomory-Hu tree [62] can be considered a reduced graph for a minimum s-t cut problem: given any pair of nodes *s* and *t*, find the weight of a minimum cut that separates the original graph *G* while *s* and *t* are in two different sets after the cut. The minimum *s*-*t* cut

between any pair of nodes in the original graph can be derived by the minimum edge on the path between the nodes of the Gomory-Hu tree. An application may be interested in finding the shortest path and the minimum s-t cut of any pair of nodes at the same time; a possible use case is clustering. If the shortest path between two nodes is long, and the minimum cut between them is small meaning they can be easily separated, the application may want to divide the graph into different clusters. In this case, we will need a general synthesis method because the Gomory-Hu tree algorithm does not satisfy the definition of the 3 stages, and the Gomory-Hu tree is not a subgraph of the original graph.

There is also a scenario in which a service provider wants to find a combined reduced graph for all graph problems. In this scenario, the combined reduced graph can be stored in the provider's main server, and each device (or service) only stores a specific reduced graph extracted from the combined graph in its own storage.

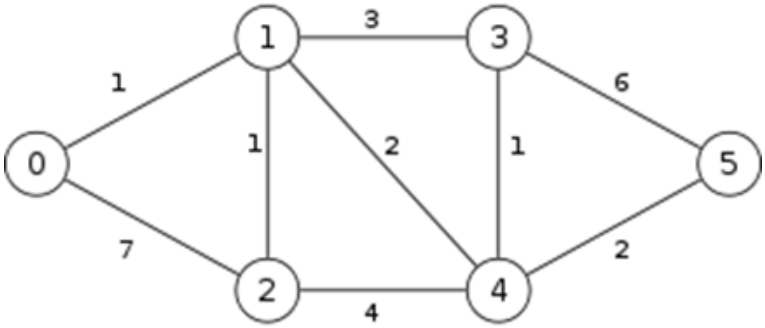
As a result, we develop a general synthesis method that can combine any reduced graphs even if it does not satisfy the three-stage definition. However the method does not guarantee that the combined graph is smaller than the original graph. The benefit of this method is that we can still easily separate the reduced graphs from the combined graph so an application can still be executed efficiently.

The general synthesis method consists of the following steps:

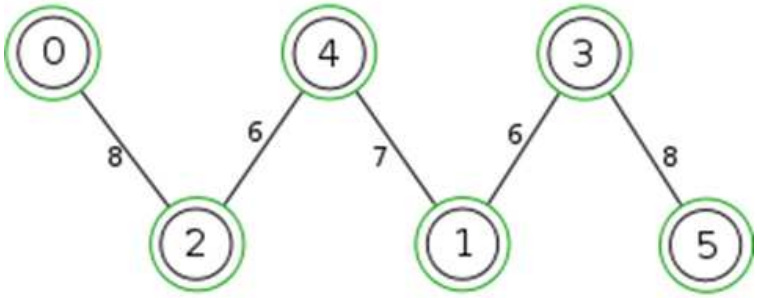
Step 1: Union the nodes. Find the nodes that are shared by the different reduced graphs after applying different reduction algorithms. If necessary, split a node. For instance, if algorithm 1 merges nodes 1 and 2 into one node, but algorithm 2 does not, then we need to split the merged node of algorithm 1 back to two nodes.

Step 2: Union the edges. If the weights of the edges are the same then we store it as a general edge. If the edges are different in different reduced graphs, add a tag to the edge to represent which weight is for which problem.

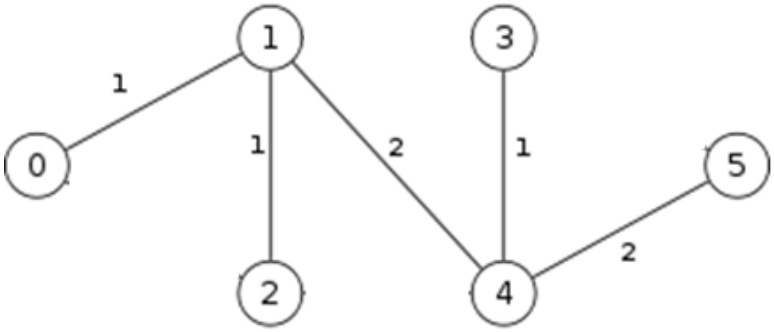
Figure 23 demonstrates the combination of Gomory-Hu tree and GR-SP. The tags of the edges are represented in different colors.



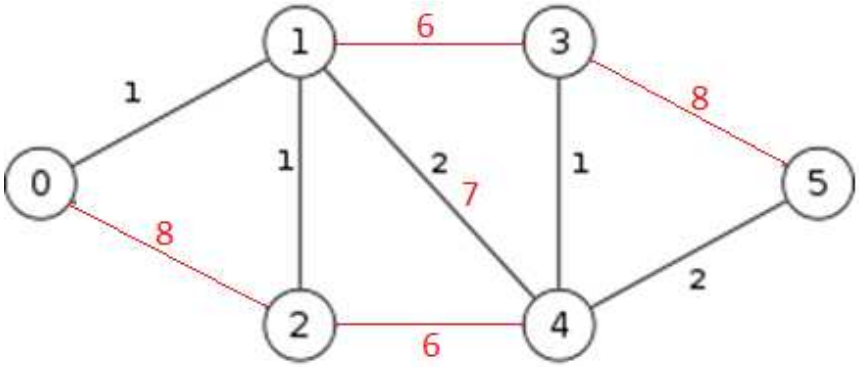
(a) The original graph G



(b) The Gomory-Hu tree



(c) The reduced graph from GR-SP



(d) The combined graph
Figure 23. Combination of Gomory-Hu Tree and GR-SP

3.5. Incremental Maintenance of Reduced Graphs

In Section 4, we discussed the GRP algorithms for the shortest path problem in details. However, due to the dynamic nature of many applications, the original graph may change, and the corresponding reduced graph should also change. If the change of the graph is major, we will need to re-compute the changed graph to obtain a new reduced graph. However in most cases the changes are minor, so it will be more efficient to compute the new reduced graph incrementally.

Incremental maintenance: Given a reduced graph G' and the change ΔG, can we compute a new reduced graph? The incremental maintenance should include four parts, which are edge insertion, node insertion, edge deletion and node deletion.

3.5.1. Edge Insertion for GR-SP

Assume we want to add an edge into the graph. Figure 24 shows an example that inserts an edge (*v2*,*v4*) with weight 4 between node *v2* and node *v4*.

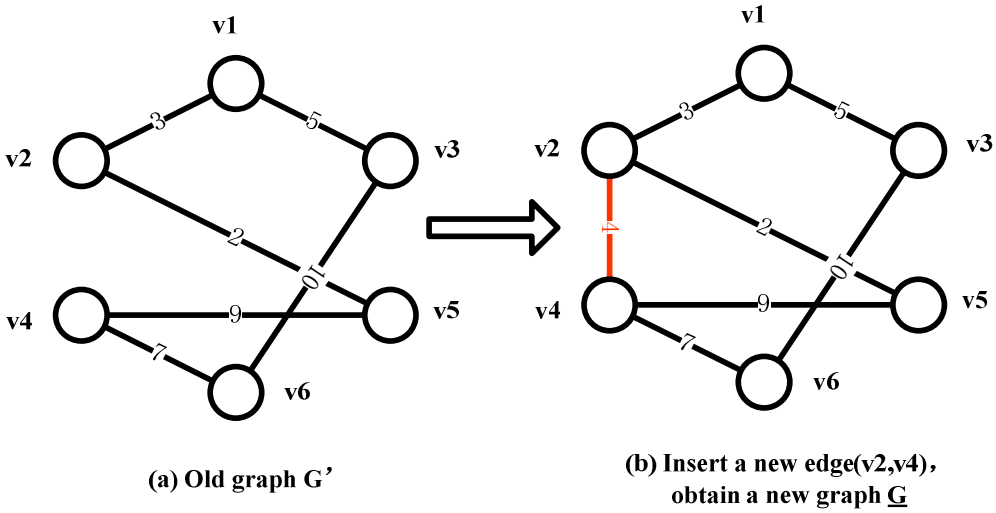


Figure 24. Insert an edge (*v2*,*v4*)=4

First we compute the shortest path between *v2* and *v4* from the old *G'*, which is *v2*-*v5*-*v4* and the distance is 11.

If the edge we want to add is longer than 11, then we do not add it to the new reduced graph because it is an unnecessary edge. In Figure 24(b), 4 is smaller than 11, so we insert the edge.

After inserting the edge (*v2*,*v4*), the new graph is G which is not a minimum graph. So, we need to convert G into a minimum reduced graph.

(1) Reduce_GP-SP (G→G'')

A solution is to apply GR-SP on G again. Figure 25 shows an example of reducing G to a minimum subgraph *G''*. We apply the GR-SP algorithm to reduce G. According to the GR-SP algorithm, we can see that the distance of edge (*v*4, *v*5) is 9. However, the shortest path from *v*4 to *v*5 has weight 4+2=6. So, we need to remove the edge (*v*4,*v*5), and obtain the reduction graph *G''* in Figure 25(b).

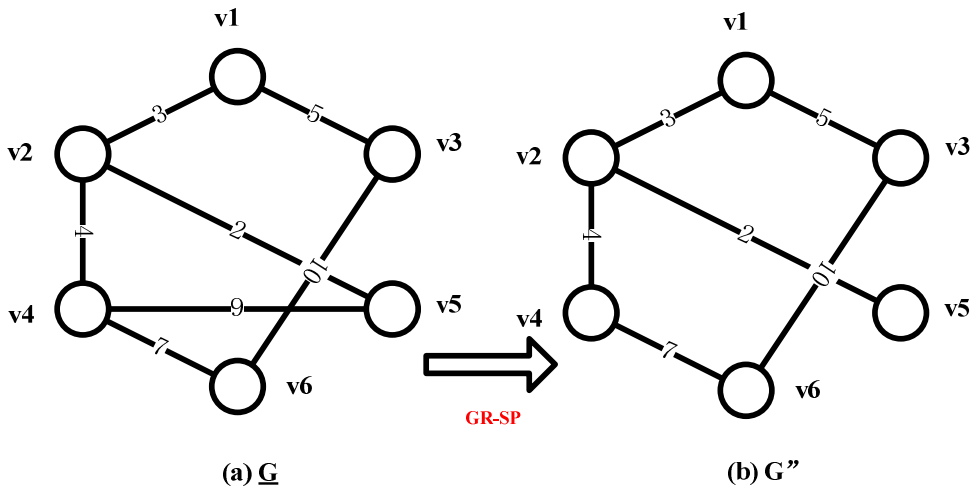


Figure 25. Compute *G’’* from *G* by the GR-SP algorithm

The Reduce_GP-SP (G→G’’) algorithm can be designed based on the GR-SP algorithms as follows.

Reduce_GP-SP (G→G’’) algorithm (Incremental maintenance on edge insertion based on GR-SP algorithm.)
Input: The original reduced graph *G’*, and inserting edge *e*
Output: Minimum spanning graph *G’’*
1. *G* ← *G’+e*
2. *G’’* ← GR-SP (*G*)
3. **Return** *G’’*

The time complexity of the Reduce_GP-SP (G→G’’) algorithm is the same as that of the GR-SP algorithm.

(2) Reduce_Cycle (G→G’’)

The method Reduce_GP-SP (G→G’’) is intuitive but may not be the best solution. This is because we have to recompute *G* by calling the GR-SP algorithm. This may not be needed because *G* was a minimum subgraph before inserting the new edge. We may use a new algorithm to obtain *G’’* from *G*. Figure 26 shows an example. The new method works as follows.

1) Find the shortest path that connects the incident nodes of the inserting edge in G'

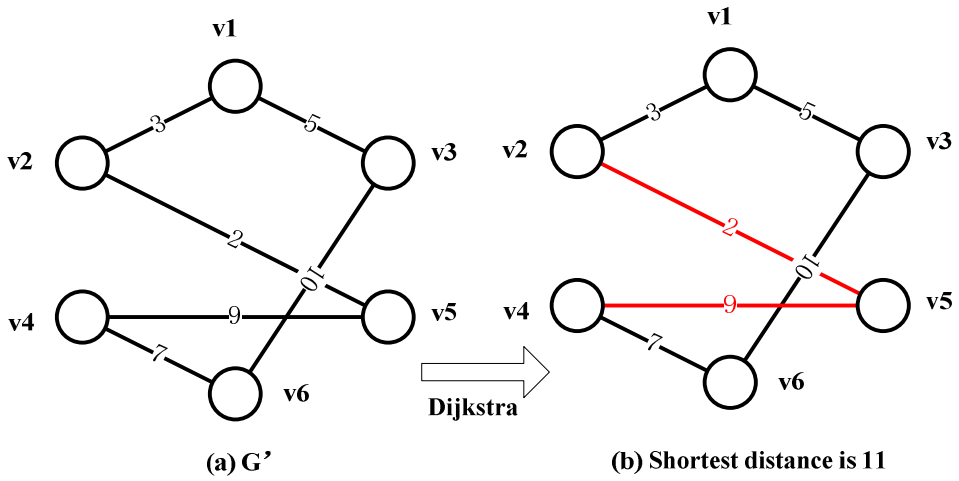


Figure 26. Compute the shortest distance between (v2, v4)

From Figure 26, we can obtain the shortest distance between *v2* and *v4* which is 11. If the inserting edge is longer than the shortest path, we do not insert it and keep the reduced graph unchanged as the new edge will not be a shortest path. If the inserting edge is shorter, we insert the edge and go to the next step.

2) Find the cycles formed by the inserting edge

After a new edge is added, as shown in Figure 27, a cycle *C* is formed.

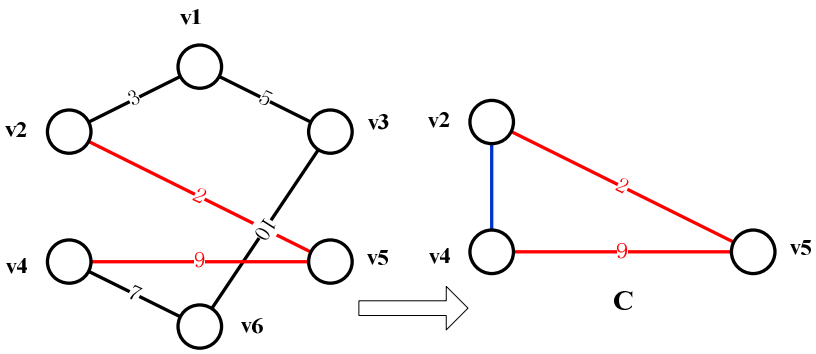


Figure 27. Detect a cycle C

3) For each cycle C, reduce C to C'

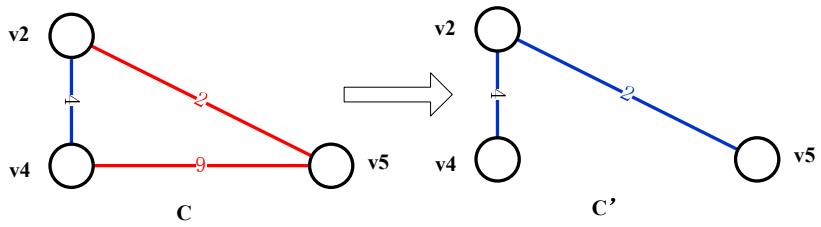


Figure 28. Compute C' from cycle C

We may be able to reduce an edge from cycle *C* - If the longest edge in the cycle is longer than the sum of the other edges, we can remove it. In Figure 28, *w(v4,v5)* is larger than *w(v2,v4)+w(v2,v5)* so (*v2,v4*) can be removed. The cycle *C* is "broken" into a tree C'. We then use *C'* to replace the cycle *C* and obtain *G''*, as shown in Figure 29.

4) Compute G''

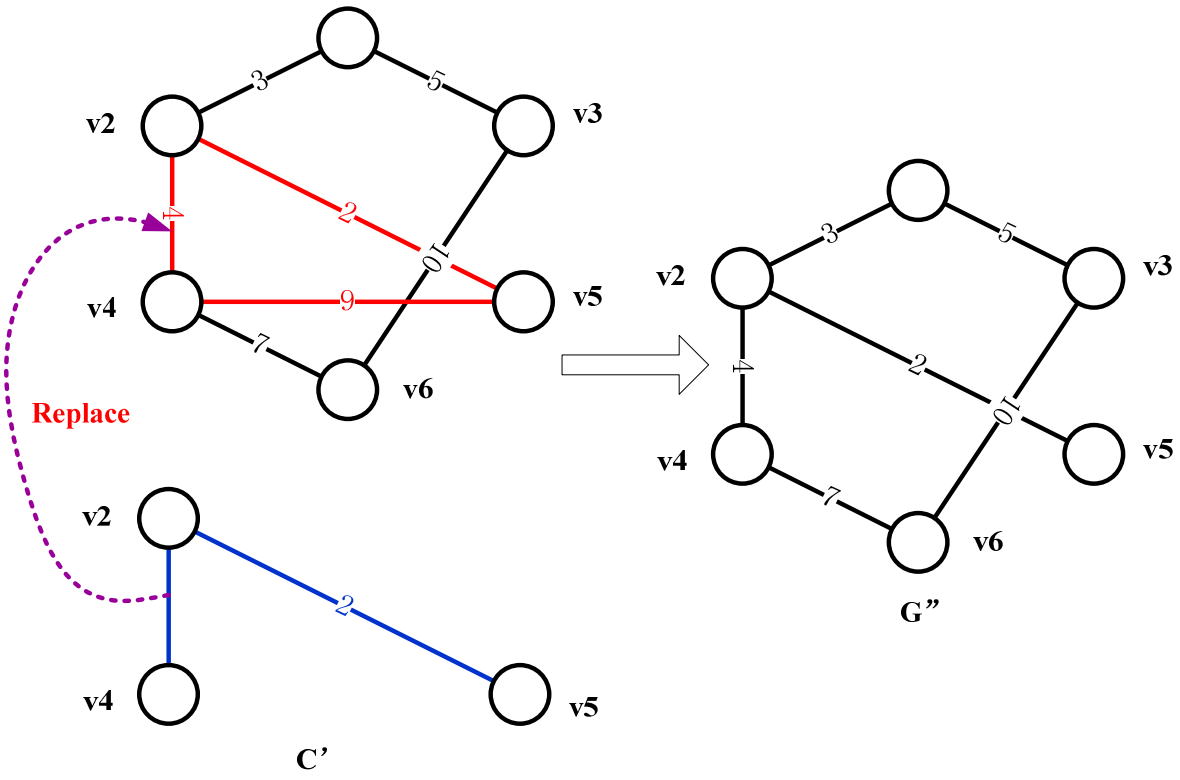


Figure 29. Compute G''

The steps of the method Reduce_Cycle (G→G’’) are summarized in the following.

Step 1: Find the shortest path of the two end nodes of the edge we want to insert in the old reduced graph

Step 2: If the distance of the shortest path between the two nodes is smaller than the edge to insert, do nothing. Otherwise, insert the edge and go to step 3.

Step 3: Check every cycles formed by the inserted edge. If the longest edge is greater than the sum of the other edges in the cycle, remove it.

The Reduce_Cycle (G→G’’) algorithm is summarized in the following.

Reduce_Cycle (G→G’’) algorithm

Reduce_Cycle (G→G’’) algorithm (Incremental maintenance on edge insertion based on cycle detection.)

Input: The original reduced graph *G’*, inserting edge *e*
Output: Minimum reduced graph *G’’*

```
I Let i, j be the incident nodes of the inserted edge e
d ← Dijkstra(G’,i,j); //find the shortest path of i,j from G’
If w(i,j) > d Return G’
Otherwise
G’’ ← G’ + e
For each cycle C formed by e in G’’ do
   Find the longest edge e’ in C
   t ← 0
   For each edge e_i in C except e’ do
      t ← t + w(e_i)
   EndFor
   If w(e’) > t, remove e’ from G’’
EndFor
Return G’’
```

The time complexity of the Reduce_Cycle (G→G’’) algorithm is T(Dijkstra).

Theorem 10: The Reduce_Cycle Algorithm generates a minimal reduced graph that preserves all the shortest paths after inserting an edge.

Proof: Let the original graph be *G*, the GR-SP reduced graph of *G* be *G'*, and the edge being inserted be $e=(a,b)$. By inserting *e* into *G*, the shortest paths between its two end nodes *a* and *b* may change in either of the following two ways:

w(e)> $\delta_G(a,b)$. In this case, inserting *e* will not cause any change on the shortest paths because the new edge is longer than the existing shortest path so any path that goes through *e* will not be the shortest.

w(e)<= $\delta_G(a,b)$. In this case, inserting *e* will cause the following shortest paths to form:

The edge *e* is a shortest path between nodes *a* and *b*. In this case we can replace the shortest path connecting *a* and *b* by the new edge *e*.

Any shortest path that contains an edge *e'* now has a shorter path *p** connecting the incident nodes of *e'* because of the new edge *e*. In this case, *p** contains *e* and we can replace *e'* by *p**.

By Theorem 1, we know *G'* preserves all the shortest paths and is minimal. We can know that the Reduce_Cycle algorithm generates the minimal subgraph *G''* preserving all the shortest paths of *G+e* by considering the cases mentioned in the above. This is because one of the following is true:

The algorithm does not insert the edge, so *G''=G'* which is still a minimal reduced graph since no shortest path is changed.

The new edge is inserted into the graph, so the shortest paths formed in 2a, with edge *e* being one of them, are preserved. Since every edge in *G'* is a shortest path because *G'* is minimal, each edge will either no longer be the shortest path or stay as a shortest path after inserting edge *e*. If it is no longer a shortest path, which is situation 2b, there must be another shortest path *p** containing *e*. Then *e'* and *p** together form a cycle, which is detected by the Reduce_Cycle

algorithm, and *e'* will be removed. Since *e'* is not a part of a shortest path anymore, *G''* still preserves every shortest paths. Since the edges not removed by the algorithm are not in situation 2b and still shortest paths; removing them will cause a contradiction to the definition that *G''* preserves all the shortest paths, so *G''* is minimal. ■

3.5.2. Edge Insertion for GR-SP2

In Section A, we only considered that all nodes in the graph are of the same type. In this section, we will consider edge insertion if there are two types of nodes in a graph. Figure 30 shows an example of inserting an edge (*v2*,*v5*) between two nodes of the same type (red nodes). Figure 31 shows an example of inserting an edge (*v2*,*v5*) between two nodes of different types (red node and black node).

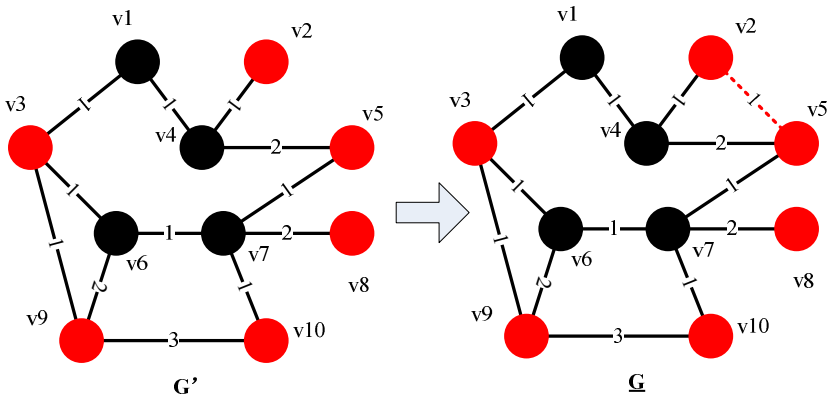


Figure 30. Insert an edge between two red nodes

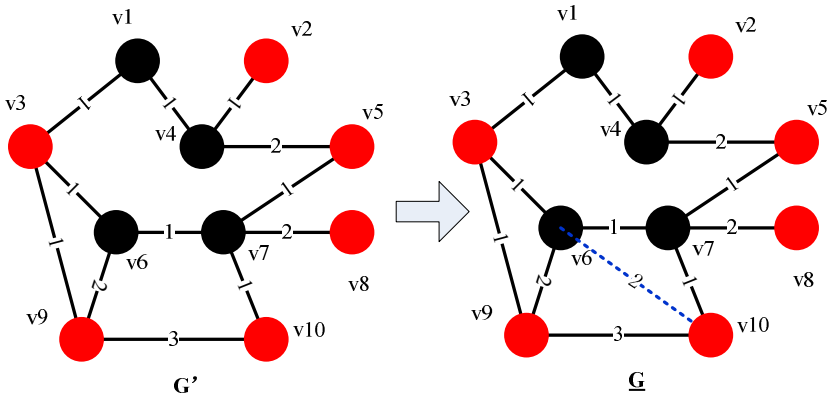


Figure 31. Insert an edge between a red node and a black node

1) Compute G'' from G using the GR-SP2 algorithm

If we insert an edge, whether the edge is between two red (or black) nodes or between a red node and a black node, we can simply apply the GR-SP2 algorithm to compute a reduce graph.

Figure 32 shows an example of reducing *G* to *G''* using the GR-SP2 algorithm. From *G'*, we can see that we want to add an edge (*v2*,*v5*) whose weight is 1. After we obtain *G'*, we apply the GR-SP2 algorithm to reduce the graph *G'* and obtain the reduced graph G''. Through this algorithm, we remove the edge (*v4*, *v5*).

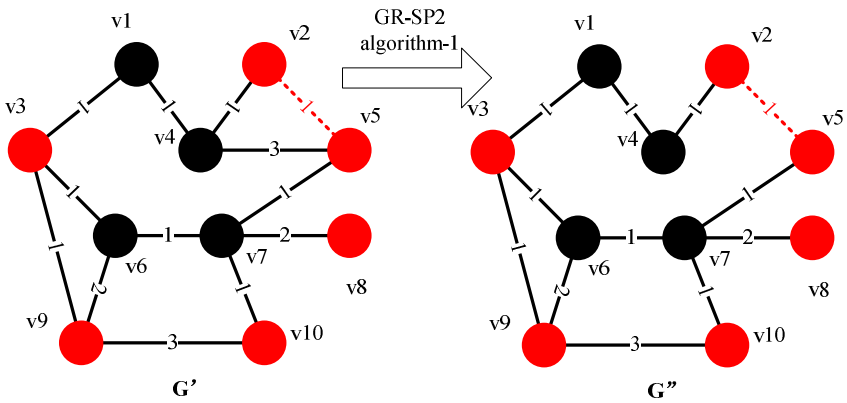


Figure 32. Reduce_GP-SP2 (G→G'')

The Reduce_GP-SP2 (G→G'') algorithm is summarized in the following.

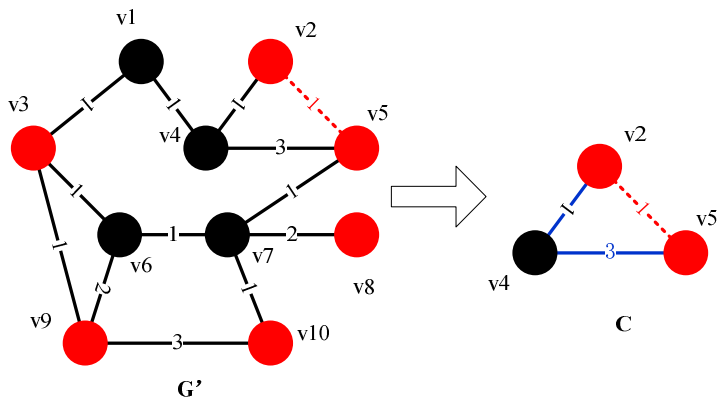
Reduce_GP-SP2 (G→G'') algorithm
Reduce_GP-SP2 (G→G'') algorithm (Find shortest path between any pair of nodes in a graph.)
Input: Graph G **Output**: A minimum reduced graph G''
1. *G* ← *G'* + *e* 2. *G''* ← GR-SP2 algorithm(*G*); 3. **Return** *G''*

The time complexity of the Reduce_GP-SP2 (G→G'') algorithm is the same as that of GR-SP2.

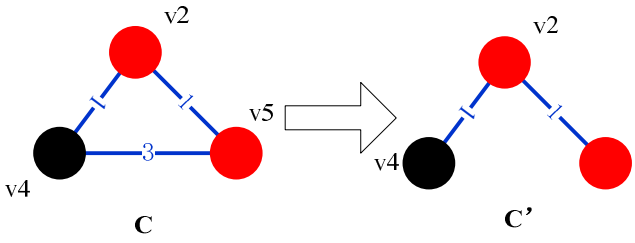
As is the case for one-color graphs, we can achieve further reduction by breaking the cycles. There are two situations to be considered: (1) The new edge is between two nodes of the same color, and (2) the new edge is between two nodes of different colors. These are discussed in (2) and (3) below.

2) Reduce_Cycle$^{2\text{-}1}$ (G→G'') if the new edge is between two nodes of the same color

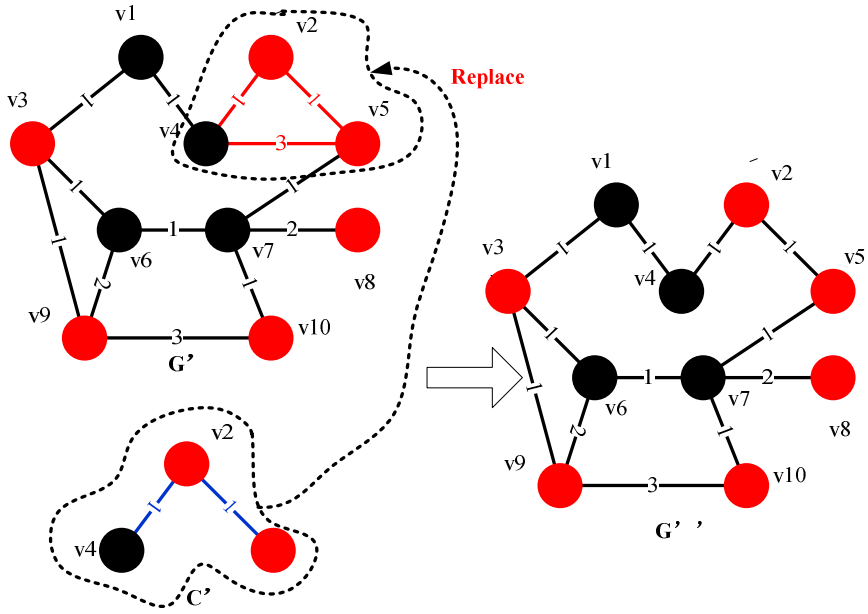
Again we may use a revised algorithm based on cycle detection to compute G'' from G to avoid re-computing G using the GR-SP2 algorithm. Figure 33 shows an example. In Figure 33(a), we need to add an edge (*v2*,*v5*), and its weight is 1. After connecting *v2* to *v5*, we have a cycle *C* formed as shown in Figure 33(a). In Figure 33(b), we obtain a tree *C'* from *C* using the Reduce_Cycle (G→G'') algorithm whose details were discussed earlier. Finally, in Figure 33(c), we replace the cycle by *C'* and obtain the reduced graph *G''*.



(a) Detect cycle C



(b) Obtain C' from C



(c) Obtain G''

Figure 33. Reduce_Cycle$^{2\text{-}1}$ (G→G'')

The Reduce_Cycle$^{2\text{-}1}$ (G→G'') algorithm for the same type of nodes is summarized in the following.

Reduce_Cycle$^{2\text{-}1}$ (G→G'') algorithm

Reduce_Cycle$^{2\text{-}1}$ (G→G'') (Incremental maintenance on inserting an edge between two nodes of the same color for a 2-color graph.)
Input: The original reduced graph *G'*, inserting edge *e* **Output**: Minimum reduced graph *G''*
1. Initialize the original graph *G'*. 2. Apply the Reduce_Cycle (G→G'') algorithm. 3. **Return** *G''*.

The time complexity of the Reduce_Cycle$^{2\text{-}1}$ (G→G'') algorithm is *T(Dijkstra)*.

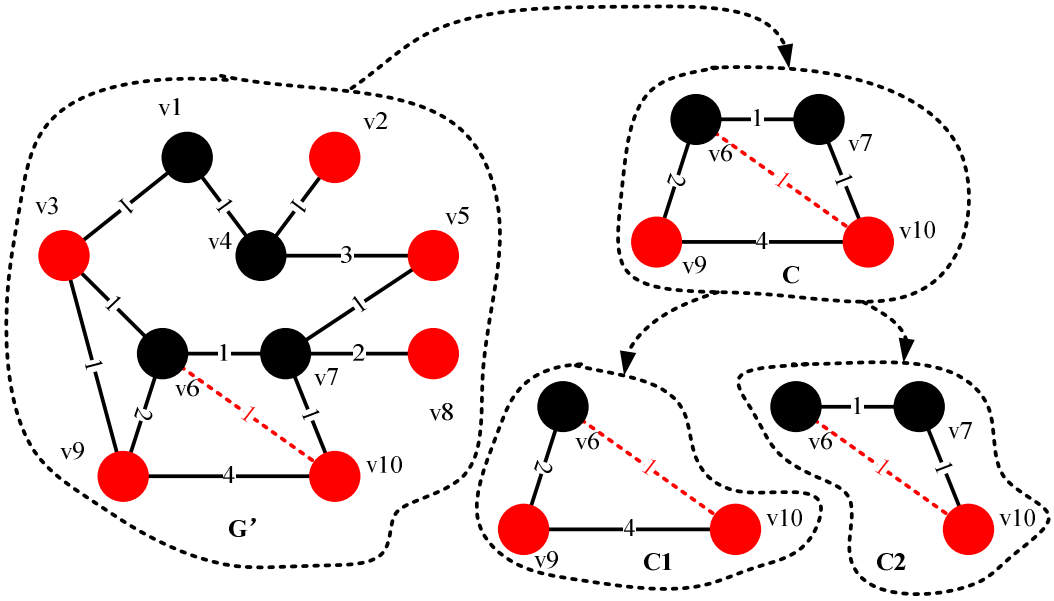
Theorem 11: The Reduce_Cycle$^{2\text{-}1}$ Algorithm generates a minimal reduced graph that preserves all the shortest paths between any pair of nodes of the same color after inserting an edge connecting two of them.

Proof: As in Theorem 10, the inserted edge may directly change the shortest path between two red nodes (or two black nodes), or affect other shortest paths so we may remove the longest edge in the cycle. Both situations have already been proved in Theorem 10. ■

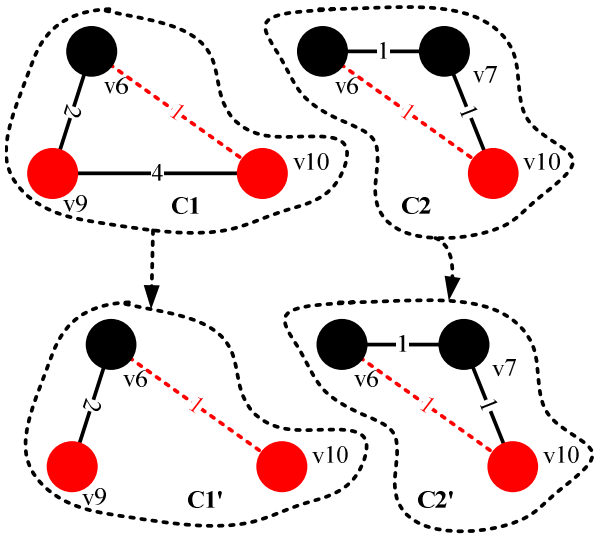
3) Reduce_Cycle$^{2\text{-}2}$ (G→G'') if the new edge connects two nodes in different colors

In the previous section, we assumed the new edge connects two nodes in the same color. If the new edge connects different types of node, what should we do? Figure 34 shows an example.

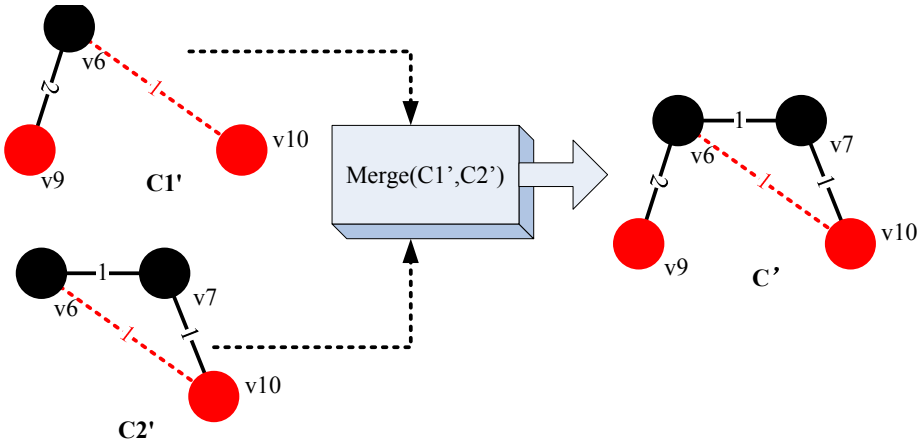
In Figure 34(a), we hope to add an edge between two different nodes *v6* and *v10*. After we add the edge (*v6, v10*), we detect two cycles *C1* and *C2*. Similar to the cases, for each cycle we check if the longest edge can be removed and obtain *C1'* and *C2'* as shown in Figure 34(b). In Figure 34(c), we obtain *C'* by merging *C1'* and *C2'*. Finally, in Figure 34(d), we replace the cycle using *C'* and obtain the reduced graph *G''*. Note that if none of the edges are removed from the cycles, it means the insertion of the edge may not be necessary, as we are not interested in the shortest paths between two nodes with different colors. In this special case, we have to check if the shortest paths from the incident nodes of the edge to all other nodes of the same color are affected or not. If not, the edge will not be inserted.



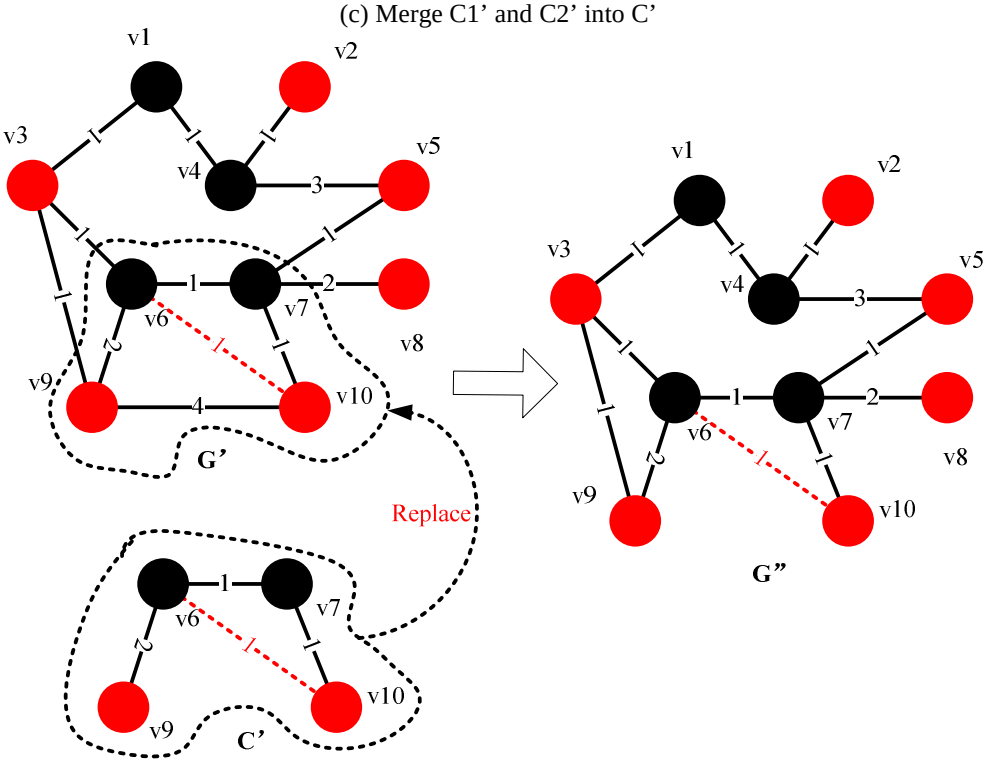
(a) Detect cycles C1 and C2



(b) Obtain C1' and C2'



(c) Merge C1' and C2' into C'



(d) Obtain the reduced graph G''

Figure 34. Reduce_Cycle$^{2\text{-}2}$ (G→G'')

The Reduce_Cycle$^{2\text{-}2}$ (G→G'') algorithm is summarized in the following.

Reduce_Cycle$^{2\text{-}2}$ (G→G’’) algorithm

Reduce_Cycle$^{2\text{-}2}$ (G→G’’) (Incremental maintenance on inserting an edge between two nodes of different colors in a 2-color graph.)

Input: The original reduced graph *G’*, inserting edge *e*
Output: Minimum reduced graph *G’’*

```
1.  Let i, j be the incident nodes of the inserted edge e
2.  d ← Dijkstra(G’,i,j); //find the shortest path of i,j from G’
3.  If w(i,j) > d Return G’
4.  Otherwise
5.  G’’ ← G’ + e
6.  For each cycle C formed by e in G’’ do
7.     Find the longest edge e’ in C
8.     t ← 0
9.     For each edge e_i in C except e’ do
10.       t ← t + w(e_i)
11.    EndFor
12.    If w(e’) > t, remove e’ from G’’
13. EndFor
14. If no edge is removed in the previous loop
15.    For each node n that has the same color of i do
16.     Check if Dijkstra(G’,i,n) = Dijkstra(G’’,i,n)
17.    EndFor
18.    For each node n that has the same color of j do
19.     Check if Dijkstra(G’,j,n) = Dijkstra(G’’,j,n)
20.    EndFor
21. EndIf
22. If all of the distances in the previous check are the same, remove e from G’’.
23. Return G’’
```

The time complexity of the Reduce_Cycle$^{2\text{-}2}$ (G→G’’) algorithm is usually T(Dijkstra), but may become N*T(Dijkstra) if lines 10 to 18 are executed, which happens when we cannot remove any edge from the cycles.

Theorem 12: The Reduce_Cycle$^{2\text{-}2}$ Algorithm generates a minimal reduced graph that preserves all the shortest paths between any pair of nodes of the same color after inserting an edge connecting two nodes in different colors.

Proof: Let the original graph be *G*, the GR-SP reduced graph of *G* be *G’*, and the edge being inserted be *e*=(*a*,*b*). By inserting *e* into *G*, the shortest paths will change in either of the following two ways:

$w(e) > \delta_G(a,b)$. In this case, inserting *e* will not cause any change to the shortest paths because the new edge is longer than the existing shortest path so any path that goes through *e* will not be the shortest.

$w(e) <= \delta_G(a,b)$. In this case, inserting *e* **may** cause the following shortest paths to form:

A shortest path is formed between nodes *a* and *b*. We can replace the path connecting *a* and *b* by the new edge *e*.

Any shortest path that contains an edge *e'* now has a shorter path *p** connecting its incident nodes because of the new edge *e*. In this case, *p** contains *e* and we can replace *e'* by *p**.

Situations 1 and 2b are the same as those in Theorem 10 and are already proven. However, situation 2a may contain no shortest paths, because it is not needed to preserve a shortest path between two nodes of different colors, so edge *e* itself is not a shortest path we need to preserve. If $w(e) <= \delta_G(a,b)$ but both 2a and 2b do not contain any of the shortest paths, which means no shortest paths is formed by the insertion, the algorithm will not insert *e*, so *G''* will still be the same as *G'* and is minimal. ■

3.5.3. Edge Insertion for GR-SPk (k>=3)

In Sections A and B, we discussed the edge insertion problem of GR-SP and GR-SP2. If there are more than two types of nodes, what should we do when the graph is updated by inserting a new edge?

The basic idea of the algorithm "edge insertion for GR-SPk (k>=3)" is the same as that of "edge insertion for GR-SP2", and we can still use the same algorithms for GR-SPk.

3.5.4. Node Insertion

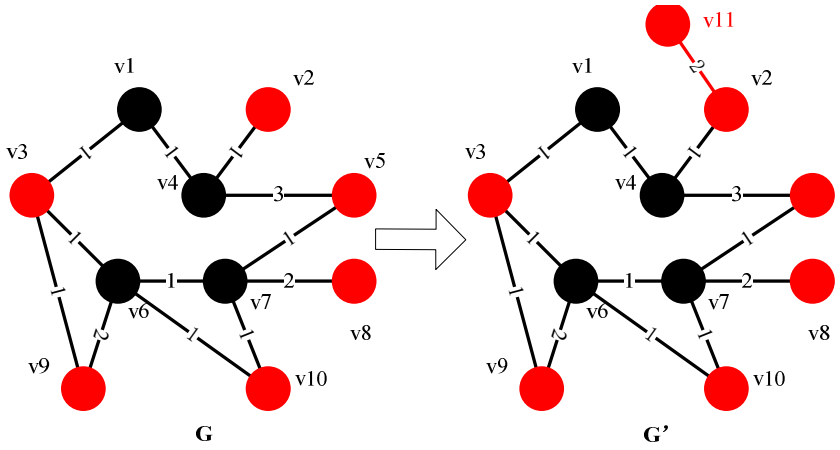
We can insert a node to a reduced graph with edges connected to it. In this case, the following two aspects need to be considered. The first is to insert a node with one incident edge and the other is to insert a node with more than one incident edge. Figure 35 shows a case that we insert a node with one edge, and Figure 36 shows another case that we insert a node with more than one edge.

1) Only one edge is connected to the new node

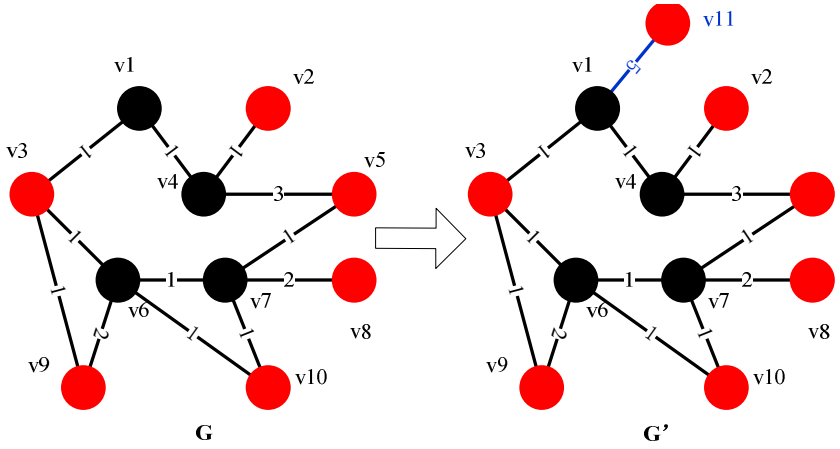
If the node has only one edge connected to it, we can simply insert it. This is because if we only add an edge to the new node, it cannot form a cycle. The new node and new edge cannot impact the fact that the graph is minimal.

In Figure 35(a), we show an example of adding a new node *v11* that is connected to an existing node *v2*, where *v2* and *v11* are of the same type (red nodes).

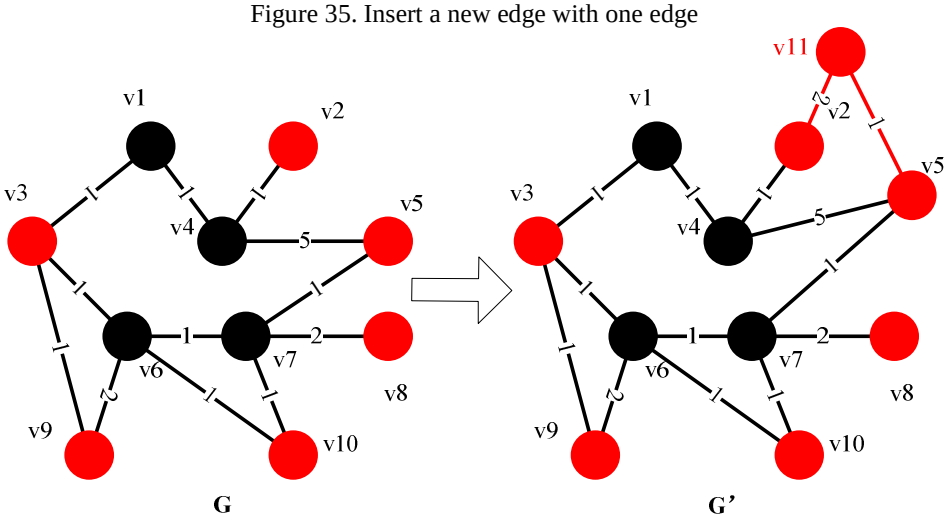
In Figure 35(b), we show an example to add a new node v11 that is connected to an existing node *v1*, where *v1* and *v11* are of different types (one is a red node and the other is a black node).



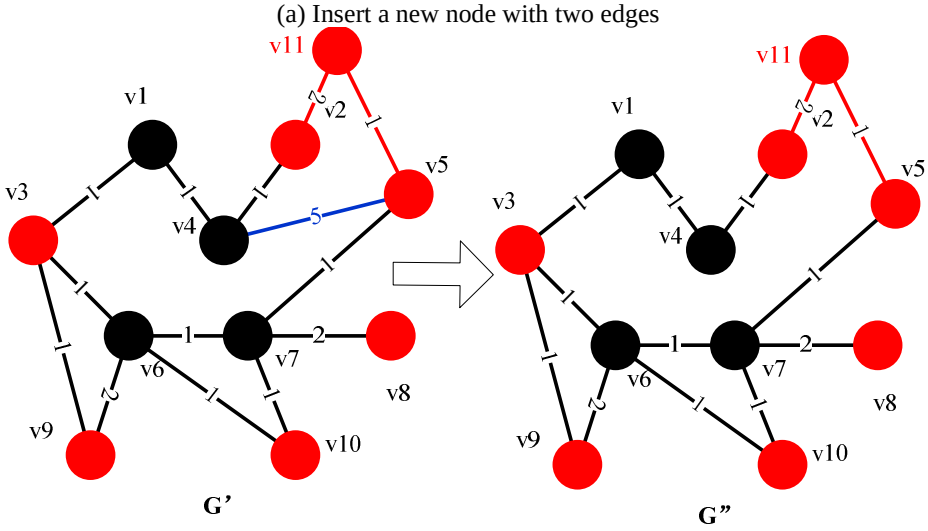
(a) Insert an edge (v2, v11) where v2 and v11 are of the same type



(b) Insert an edge (v1, v11) where v1 and v11 are of two different types.
Figure 35. Insert a new edge with one edge



(a) Insert a new node with two edges



(b) Reduce the G' and get the reduced G'' using reduction algorithm in section A and section B.
Figure 36. Insert a new edge with more than two edges

(2)There are more than one edges connected to the new node

If the node has more than one edge connected, we can randomly insert one edge first (may be the shortest one), and then insert the other edges using the edge insertion algorithm described earlier.

Figure 36 shows an example for the insertion and graph reduction process. In Figure 36(a), we compute an updated graph when inserting a new node with 2 edges. In Figure 36(b), we reduce *G'* and then reduce *G''* using the reduction algorithms in Section A and Section B.

In Figure 36(a), we add a new node *v11*, and *v11* is connected to node *v2* and node *v5*. The weight of the edge *(v2,v11)* is 2 and the weight of the edge *(v5,v11)* is 1.

Following is the algorithm to incrementally update a reduced graph when a node is being inserted.

Insert_Node (G→G'') algorithm

Insert_Node (G→G'') (Incremental maintenance on node insertion.)

Input: The original reduced graph *G'*, inserting node *n* with edge set *E* connecting to it
Output: Minimum reduced graph *G''*

```
G'' ← G'
e* ← minimum edge from E
Insert e* and n to G''.
For each edge e from E except e* do
   Apply suitable Reduce_Cycle algorithms to G'' with e
EndFor
Return G''
```

The time complexity is the same as that of the Reduce_Cycle algorithm.

Theorem 13: The Insert_Node Algorithm generates a minimal reduced graph *G''* that preserves all the shortest paths after inserting a node *n* with edge set *E* connecting to it.

Proof: If the edge set *E* contains only one edge $e^*=(n,v)$, the algorithm simply adds the node *n* and the edge *e** into *G'*. We prove the result *G''* preserves all the shortest paths by the following two cases:

1. For shortest paths not started or ended at node *n*: Because node n has only 1 degree, it cannot be an intermediate node of any path. As a result, the shortest paths in this case will not contain node *n*, and they will be exactly the same as in *G'*. Therefore *G''* also preserves them.

2. For shortest paths started or ended at node *n*: It is clear that $e^*=(n,v)$ is the only edge connecting to *n*, so the shortest paths must be e^* plus the shortest paths from *v* to other nodes, which is the same as in case 1 and proved to be preserved.

By the cases above, we prove that all the shortest paths are preserved if we insert a node with a single edge. Since the original reduced graph *G'* is minimal, and e^* is the only edge connecting *n*, *G''* will still be minimal.

If the edge set *E* contains more than one edge, we insert one edge first, and the other edges are inserted by the Reduce_Cycle algorithm. The correctness of the Reduce_Cycle algorithm has already been proven (Theorem 10 to 12). ■

3.5.5. Edge and Node Deletion

Unfortunately the information of the old reduced graph is not enough for deletion. This is because after a deletion, some unnecessary edges which are not the shortest before will become necessary now since the original shortest one is deleted. Therefore we have to run the GR-SP algorithm again after a deletion.

Chapter 4 Lossy Graph Data Reduction

4.1. Formal Definitions of LGRP

4.1.1. Lossy Graph Data Reduction

The Lossy Graph Reduction Problem (LGRP) is defined as follows. Given a graph G, a graph problem P, and a set of possible queries Q for the graph problem P, find a reduced graph G' whose size is smaller than G and it is able to answer the corresponding queries (in the original form or in a transformed form) in Q' but with some information lost.

Note that for lossless graph data reduction, the goal is:

- For any $q \in Q$, the corresponding query $q' \in Q'$ can be computed
- $q(G) = q'(G')$, where $q(G)$ is the result of q computed from G (e.g., a shortest distance, a Boolean value, etc.)

The information lost can be defined in two types:

- Some queries cannot be answered by G' (Query Loss): There exist some queries $q \in Q$ that cannot be computed from G'.
- Query results may have some error (Approximate Results): There exist some queries that $q(G) \mathrel{!=} q'(G')$.

4.1.2. Graph Problems Addressed

In this chapter, we study 3 basic graph problems:

(1) Distance-constrained reachability problem

Given two nodes *a* and *b*, can *a* reach *b* in distance *d*?

(2) Shortest path problem

Given two nodes *a* and *b*, what is the shortest distance between *a* and *b*, and what is the shortest path(s) between them?

(3) MST problem

Given a subset of nodes *N*, what is the minimum spanning tree(s) that includes all the nodes in *N*?

Some variations of these problems will also be discussed.

(1) 1+ε Lossy reduction

If an application can tolerate bounded errors for some queries in exchange for graph reduction, the reduction is called 1+ε lossy reduction, where ε ($0 < \varepsilon < 1$) is the bound for the ratio of errors.

(2) 2-degree-nodes-cut reduction

In some applications (to be discussed in Section IV), we may be only interested in queries on nodes with more than 2 incident edges (e.g., a map network) in order to improve the reduction ratio. Specifically we remove all nodes whose degree is 2. This is called 2-degree-nodes-cut reduction in this thesis.

(3) 2-degree-nodes reduction

Different from 2-degree-nodes-cut reduction, 2-degree-nodes reduction does not remove any node. Instead it retains all the nodes in graph G' but merge the 2-degree nodes to reduce the graph. The reduction rate is not as high as that of 2-degree-nodes-cut reduction, but it can still answer the queries related to 2-degree-nodes.

Compared to 1+ε lossy reduction, the 2-degree-nodes reduction is most suitable to graphs where the average degree of the nodes is low, and the weight of edges are geometric, e.g., road map data:

(a) Most of the query results are accurate if we query the shortest paths between a non-2-degree node (i.e., degree ≠ 2) and another non-2-degree node.

(b) Some query results are close to accurate if we query the shortest paths between a 2-degree node and another node.

(c) Some query results are not accurate if we query the shortest path between a 2-degree node and another 2-degree node.

4.2. Lossy Reduction for The Shortest Path Problem

4.2.1. Definition of LGR-SP

Given a graph G, assume an application is only interested in finding a shortest path between any pair of two nodes, which can be solved by many existing algorithms (e.g., the Dijkstra algorithm and the Floyd algorithm), a lossy reduction method reduces G to a smaller graph G' which preserves most of the shortest paths between any pair of nodes. We call this problem the LGR-SP (Lossy Graph Reduction-Shortest Path) Problem.

4.2.2. Use Case of LGR-SP (Road Network)

Using a U.S Road Network (Figure 37) as an example, some applications may be interested in finding the shortest paths between any two locations.

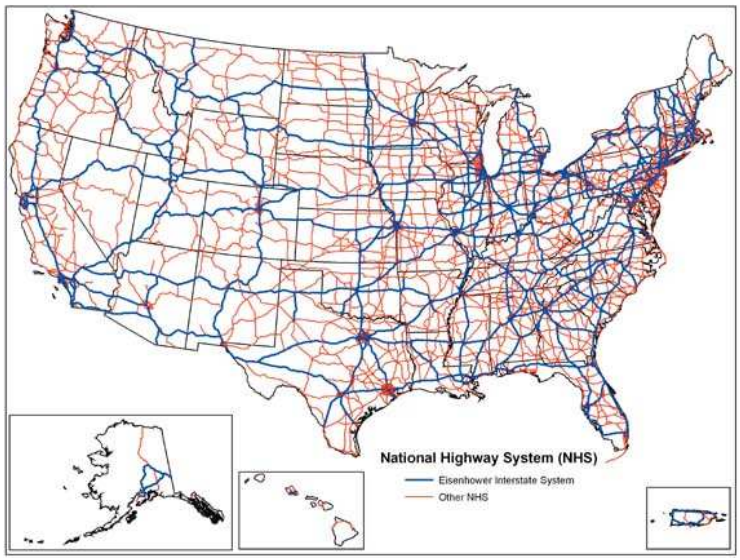


Figure 37. U.S Road Network [84]

A lossy graph reduction algorithm may allow us to compute a shortest path between any two locations tolerating from the reduced graph some small but bounded errors compared to the results obtained from the original graph.

4.2.3. Algorithms for LGR-SP

We describe three types of algorithms in this section.

(1) 1+ε LGR-SP Algorithm

The basic idea of the 1+ε LGR-SP algorithm is to remove the edge between any two nodes if we can find another shortest path connecting the two nodes whose weight is less than or equal to 1+ε times the weight of the edge. If the path is shorter than the edge, it is safe to remove the edge without causing any error; if the path is longer, the error is bounded by 1+ε, where ε can be decided by the user when applying the algorithm.

However, simply going through every edge and removing any edge that has another path not greater than $1+\varepsilon$ times of the edge weight may cause a problem. Assume an edge e is removed

because we find another path *p* whose weight is $(1+\varepsilon)w(e)$. Later we remove another edge e' as we find another *p'* whose weight is $(1+\varepsilon)w(e')$. It is possible that *e'* belongs to *p*, and removing e' would affect the path *p*, and the weight of the path connecting the incident nodes of *e* becomes $(1+\varepsilon)w(e) + \varepsilon w(e')$, so the error accumulates and becomes unbounded. Therefore, instead of edge removal, we build the reduced graph by inserting edges in increasing weight order. The algorithm checks the shortest one of the un-processed edges in each loop, and only if there is no path can be found to connect the incident nodes of the edge and does not violate the $(1+\varepsilon)$ bound, the edge can be inserted into the graph.

The 1+ε LGR-SP algorithm is described in the following.

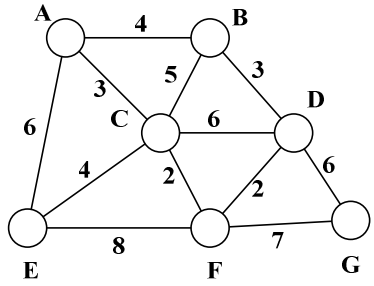
1+ε LGR-SP algorithm (Find a reduced graph that preserves the shortest paths between any pair of nodes in a graph with the error bound ε)

Input: Graph *G*
Output:1+ε Lossy reduction graph *G'*

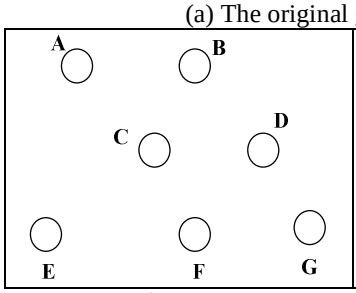
```
Let V be the nodes set of G, E be the edges set of G
G' ← (V, ∅) //G' contains only nodes in V without any edge
Sort E in ascending order of weight
For each edge e in E do
    Let nodes x, y be the incident nodes of e
    Dist=Dijkstra(G', x, y); //find the shortest distance of the corresponding nodes in G'
    If Dist does not exist or w(e) * (1+ε) < Dist
        Add e to G'
    EndIf
EndFor
Return G'
```

The time complexity of the 1+ε LGR-SP algorithm (based on Dijkstra) is n*T(Dijkstra).

Figure 38 shows an example that we apply the 1+ε LGR-SP algorithm to derive a reduced graph G' from the original graph G, where the value of ε is 0.2. For simplicity, we use the notion *AB* to designate *w(A,B)*, where *A* and *B* are two nodes.

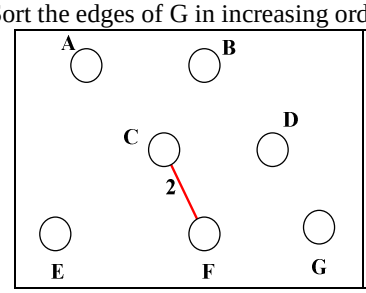


(a) The original graph G



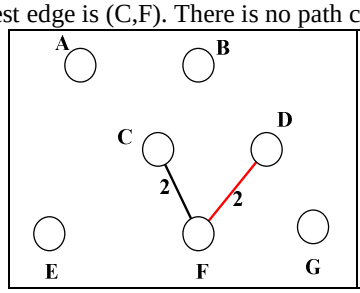
CF=2；DF=2；
AC=3；BD=3；
AB=4；CE=4；
BC=5；AE=6；
CD=6；DG=6；
FG=7；EF=8

(b) Sort the edges of G in increasing order, and create G’ without edges



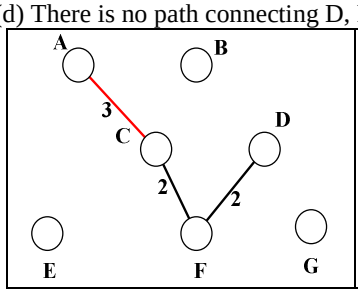
[CF=2]；DF=2；
AC=3；BD=3；
AB=4；CE=4；
BC=5；AE=6；
CD=6；DG=6；
FG=7；EF=8

(c)The shortest edge is (C,F). There is no path connecting C, F in G’. Insert (C,F)to G’.



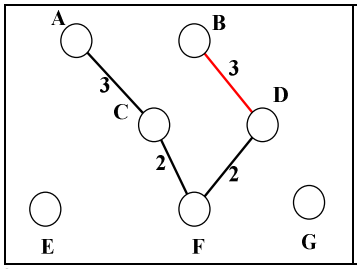
[DF=2]；AC=3；
BD=3；AB=4；
CE=4；BC=5；
AE=6；CD=6；
DG=6；FG=7；
EF=8

(d) There is no path connecting D, F in G’. Insert (D,F) to G’.



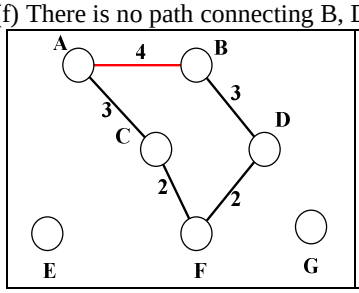
[AC=3]；BD=3；
AB=4；CE=4；
BC=5；AE=6；
CD=6；DG=6；
FG=7；EF=8

(e) There is no path connecting A, C in G’. Insert (A,C) to G’.



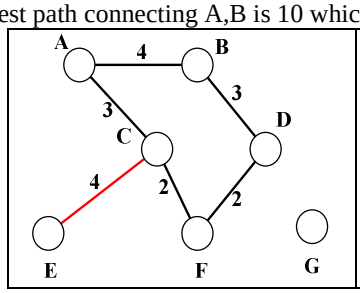
[BD=3]；AB=4；
CE=4；BC=5；
AE=6；CD=6；
DG=6；FG=7；
EF=8

(f) There is no path connecting B, D in G'. Insert (B,D) to G'.



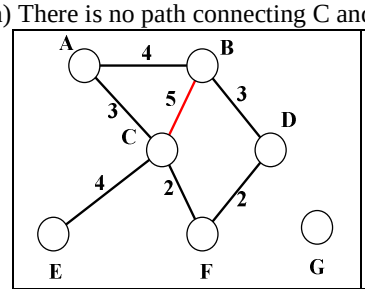
[AB=4]；CE=4；
BC=5；AE=6；
CD=6；DG=6；
FG=7；EF=8

(g) The weight of the shortest path connecting A,B is 10 which is greater than (1+0.2)*AB=4.8. Insert (A,B) to G'.



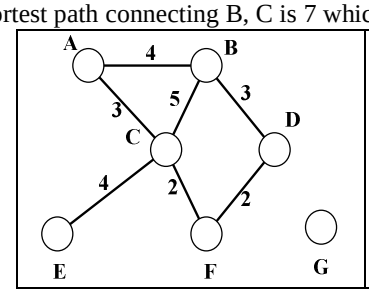
[CE=4]；BC=5；
AE=6；CD=6；
DG=6；FG=7；
EF=8

(h) There is no path connecting C and E in G'. Insert (C,E) to G'.



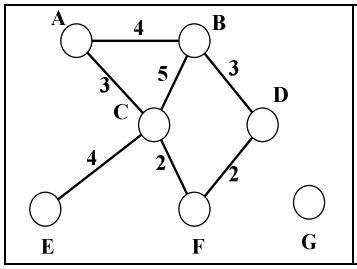
[BC=5]；AE=6；
CD=6；DG=6；
FG=7；EF=8

(i) The weight of the shortest path connecting B, C is 7 which is greater than (1+0.2)*BC=6. Insert (B,C) to G'.



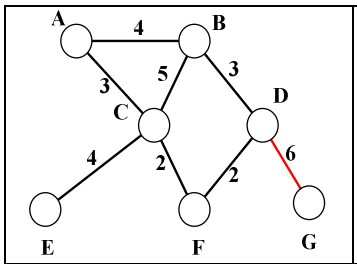
[AE=6]；CD=6；
DG=6；FG=7；
EF=8

(j) The weight of the shortest path connecting A, E is 7 which is smaller than (1+0.2)*AE=7.2. Ignore (A,E).



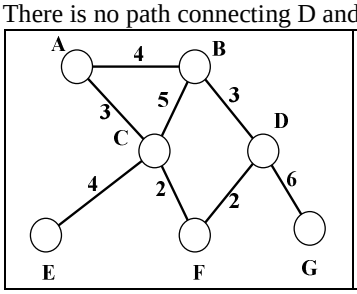
[CD=6]； DG=6；
FG=7；EF=8

(k) The weight of the shortest path connecting C, D is 4 which is smaller than (1+0.2)*CD=7.2. Ignore (C,D).



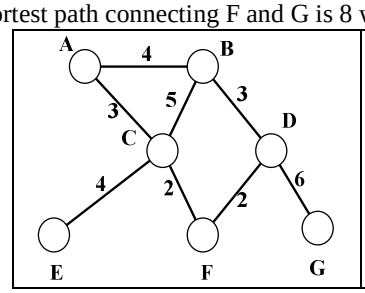
[DG=6]；FG=7；
EF=8

(l) There is no path connecting D and G in G'. Insert (D,G) to G'.



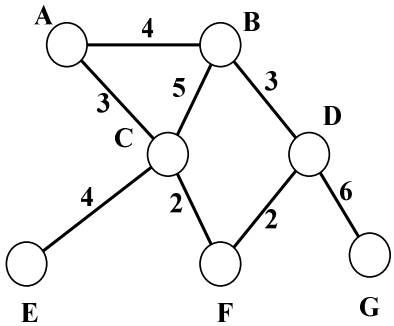
[FG=7]； EF=8

(m) The weight of the shortest path connecting F and G is 8 which is smaller than (1+0.2)*FG=8.4. Ignore (F,G).



[EF=8]

(n) The weight of the shortest path connecting E and F is 6 which is smaller than (1+0.2)*EF=9.6. Ignore (E,F).



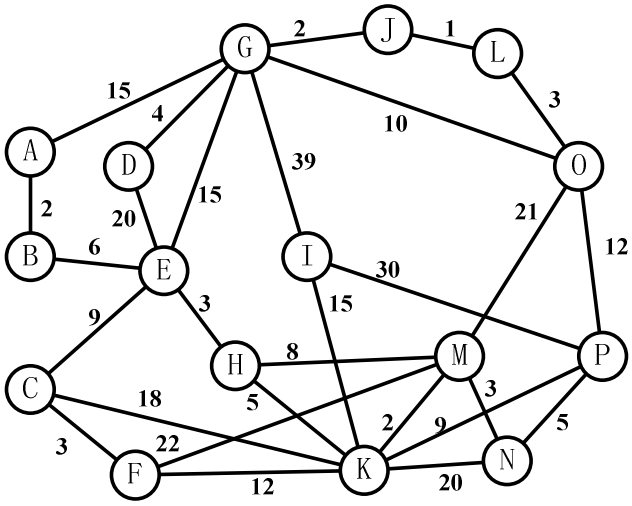
(o) The final reduced graph G'

Figure 38. 1+ε LGR-SP

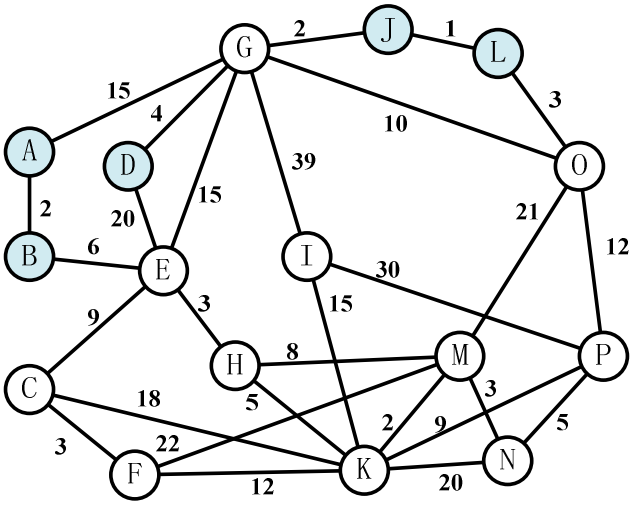
Figure 38(a) shows the original graph G. In Figure 38(b), we sort the edges based on the weights in ascending order. In Figure 38(c), we add the smallest edge (C,F) where CF=2 because the edge's weight is less than (1+ε) of the shortest distance. From Figure 38(d) to Figure 38(n), we add all edges, which are (C,F), (D,F), (A,C), (B,D), (A,B), (C,E), (B,C) and (D,G). Finally in Figure 38(o), we obtain the final reduced graph G'.

(2) 2-degree-nodes-cut LGR-SP Algorithm

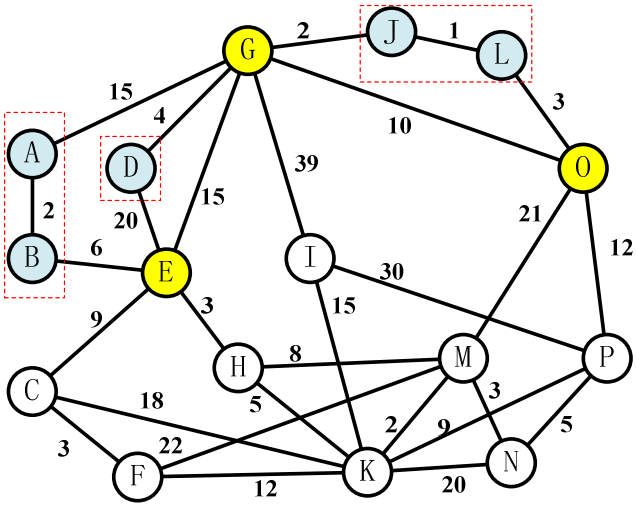
Figure 39 shows an example applying the 2-degree-nodes-cut LGR-SP algorithm to obtain a reduced graph.



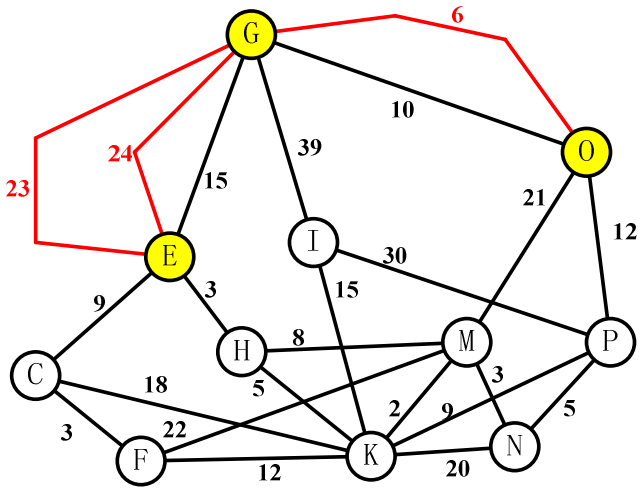
(a) Original graph *G*



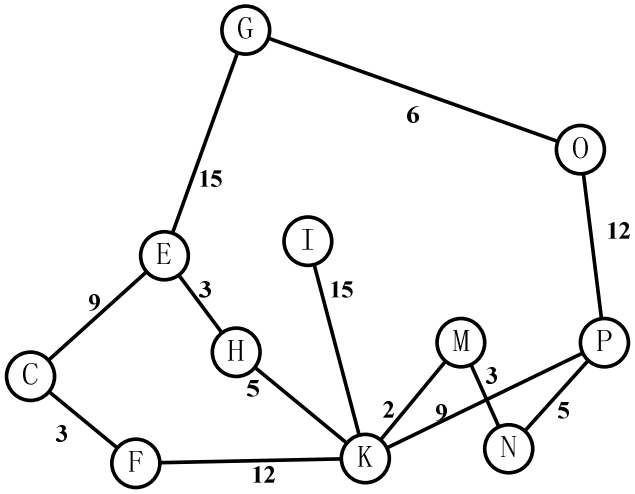
(b)Find the 2-degree nodes {A,B,D,J,L}



(c) Group the 2-degree nodes if they are neighbors. For each group, there will be two neighboring non-2-degree nodes.



(d) For each group, remove the nodes and the edges connecting to them. Reconnect the neighboring non-2-degree nodes by an edge whose weight is the same as the total weight of the edges removed.



(e) Apply GR-SP to obtain the 2-degree-nodes-cut reduction graph *G'*

Figure 39. 2-degree-nodes-cut LGR-SP

Figure 39(a) shows the original graph G. In Figure 39(b), we find all the 2-degree nodes, which are {A,B,D,J,L}. In Figure 39(c), we group the 2-degree nodes so that the nodes inside each group are neighbors and are connected as a path. For each group, we also find the two neighboring non-2-degree nodes at the two ends. (e.g., The group {A,B} has two neighboring non-2-degree nodes at the two ends: {E,G}). In Figure 39(d), for each group, the 2-degree nodes and the edges connected to them are replaced by a single edge, whose weight is the total weight of the edges removed. (e.g., nodes A and B and edges (E,B), (B,A), (A,G) are removed, and the edge (E,G) whose weight is 6+2+15=23 is inserted.) In Figure 39(e), we apply the GR-SP algorithm to remove the edges that are not the shortest paths and obtain the reduced graph *G'*.

As can be seen, the main idea of the 2-degree-nodes-cut LGR-SP algorithm is to remove all the 2-degree-nodes and reconnect their neighboring non-2-degree nodes by edges that produce the shortest distance between them.

The 2-degree-nodes-cut LGR-SP algorithm is summarized in the following.

2-degree-nodes-cut LGR-SP algorithm (**Find reduced graph that preserves the shortest paths between only pairs of non-2-degree nodes**)

Input: Graph *G*
Output: 2-degree-nodes-cut reduction graph *G''*

```
G' ← G;
N ← Find_2_degree_node(G); //N is the set of the nodes whose degree is 2
C ← Group_by_neighbor(N); // C is a set of groups, each group is a collection of nodes in N that each node
  in the group has at least one neighbor node in the same group, if it is not the only node in the group
For each group C_i in C do
    Find two non-2-degree nodes x and y that are connected to the two ends of the nodes in C_i
    //The 2-degree nodes in the same group are connected by a path, and the path is connected to exactly two
     of such non-2-degree nodes
    Remove all nodes in C_i and the edges connecting them from G;
    Insert edge (x,y) with the weight = total_edges_weight(C_i) to G';
    //Replace C_i by a single edge whose weight equals to the total weight of the weights of the edges removed
EndFor
G'' ← GR-SP(G');
Return G''
```

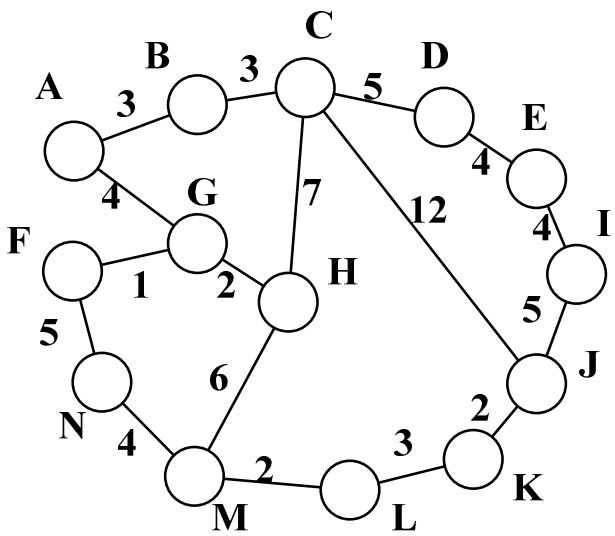
The time complexity of the LGR-SP algorithm is $O(n^2)$.

(3) Simple 2-degree-nodes reduction LGR-SP Algorithm

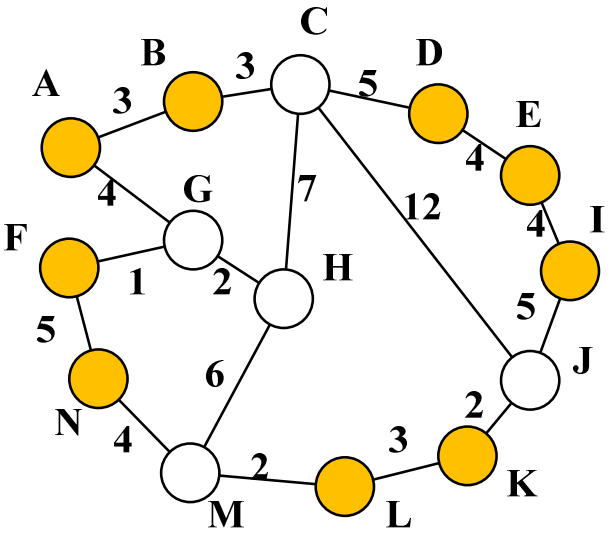
In Figure 39, the 2-degree-nodes-cut reduction graph G' does not preserve the information about 2-degree-nodes. If a query involves 2-degree nodes, e.g., the shortest distance between A and N in Figure 39(e), it cannot be answered. Therefore, we develop the simple 2-degree-nodes reduction LGR-SP algorithm.

The main idea of the algorithm is to preserve the information about the 2-degree-nodes and their closest non-2-degree nodes and the distances. Based on the information, the queries about 2-degree-nodes can still be answered but with some possible errors.

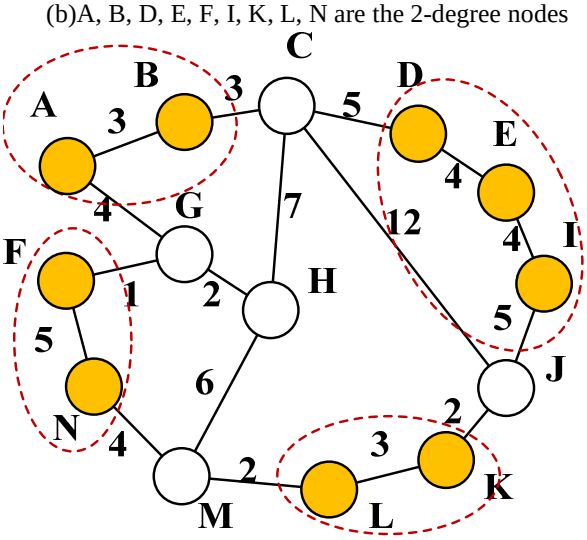
Figure 40 shows the method to obtain a reduced graph G' from the original graph G based on the simple 2-degree-nodes reduction LGR-SP algorithm.



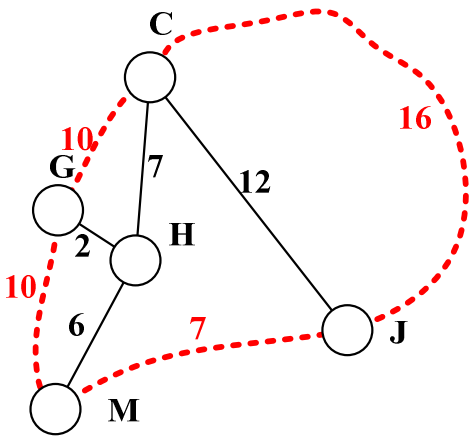
(a)Original Graph G



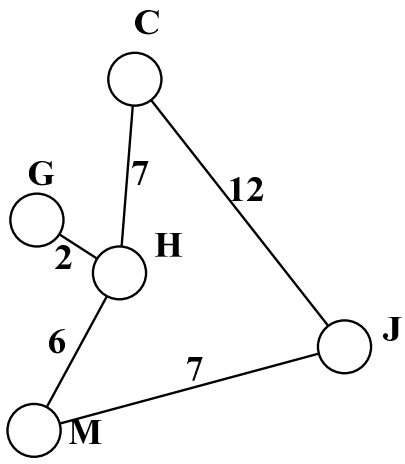
(b)A, B, D, E, F, I, K, L, N are the 2-degree nodes



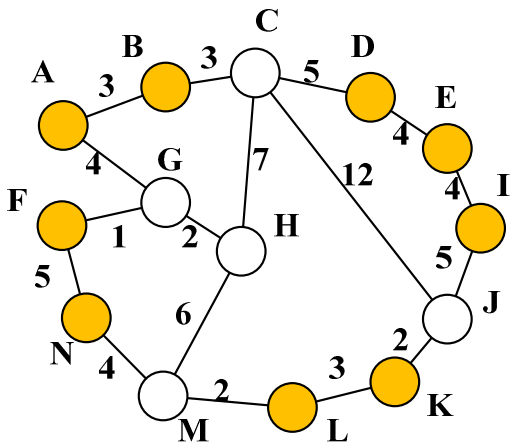
(c) Group them by neighborhood, and compute the total weights of edges (e.g., the total weight of the Group {A,B} is CB+BA+AG=10)



(d) Replace the 2 degree nodes by edges with the group weight. (e.g. CG=10 from group {A,B})



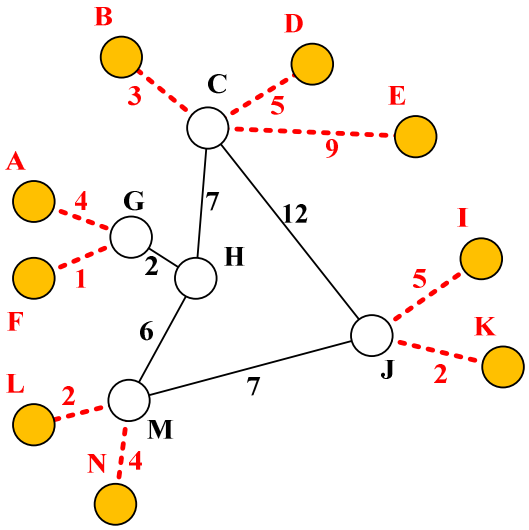
(e) Apply the GR-SP algorithm and obtain G'



Route Table

2-degree node	Closest non-2-degree node	Distance
A	G	4
B	C	3
D	C	5
E	C	9
F	G	1
I	J	5
K	J	2
L	M	2
N	M	4

(f)For each 2-degree node, find the closest non-2-degree nodes, and save the distances into a route table. (e.g., The closest non-2-degree nodes for A is G, and the distance is 4).



(g)Insert the 2-degree nodes with the edge to the closest non-2-degree nodes back to G'

Figure 40. Simple 2-degree-nodes reduction LGR-SP

Figure 40(a) shows the original graph. In Figure 40(b), all 2-degree nodes are found, which are A, B, D, E, F, I, K, L and N. in Figure 40(c), we compute the total edge weights of each group. The total weight for the Group {A,B} is CB+BA+AG=10, for the Group {F,N} is 10, for the Group {D,E,I} is 16, and for the group {L,K} is 7. We cut all 2-degree nodes and replace them by a set of edges, one for each group that has the weight equal to the total weight of the group, shown in Figure 40(d). We then apply the GR-SP algorithm in Figure 8(d) and obtain the graph of Figure 40(e). In Figure 40(f), we construct the route table for the 2-degree nodes based on the original graph G. The route table preserves some information about the 2-degree nodes so we can still answer queries on them with some possible errors. We may store the reduced graph in 40(e) and the route table separately. We may also combine all the 2-degree nodes and non-2-degree nodes and obtain the final reduction graph G' as shown in Figure 40(g).

There are two major drawbacks in this algorithm, still. First, if we do not store the route table separately in a compact way, the reduction is minor if we combine the 2-degree nodes and non-2-degree nodes as shown in Figure 40(g) (The number of edges is only reduced from 17 to 14). Second, the error for the queries related to 2-degree nodes may be very high (e.g., the shortest path between L and K in the reduced graph is 11, while it should be 3 in the original graph.) Therefore, we introduce the 2-degree-nodes reduction LGR-SP algorithm.

The simple 2-degree-nodes reduction LGR-SP algorithm is described in the following.

Simple 2-degree-nodes reduction LGR-SP algorithm

Input: Graph G

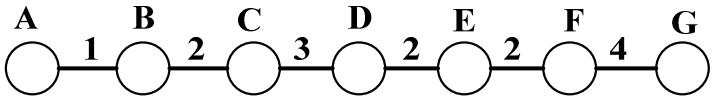
Output: An approximate reduced graph G''

```
G' ← G;
N ← Find_2_degree_node(G); //N is the set of the nodes whose degree is 2
C ← Group_by_neighbor(N); // C is a set of groups, each group is a collection of nodes in N that each node in
  the group has at least one neighbor node in the same group, if it is not the only node in the group
For each group C_i in C do
    Find two non-2-degree nodes x and y that are connected to the two ends of the nodes in C_i
    //The 2-degree nodes in the same group are connected by a path, and the path is connected to exactly two of
    such non-2-degree nodes
    Remove all nodes in C_i and the edges connecting them from G;
    Insert edge (x,y) with the weight = total_edges_weight(C_i) to G';
    //Replace C_i by a single edge whose weight equals to the total weight of the weights of the edges removed
    For each node in C_i, add the node x or y, whichever is closer to C_i, and the distance between it and C_i into
    the route table.
    Based on the route table, insert the edges to G'
EndFor
G''=GR-SP(G');
Return G''
```

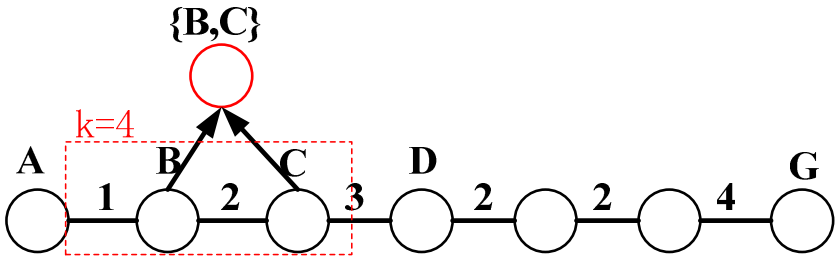
The time complexity of the simple 2-degree-nodes reduction LGR-SP algorithm is $O(n^2)$.

(4) 2-degree-nodes reduction LGR-SP Algorithm

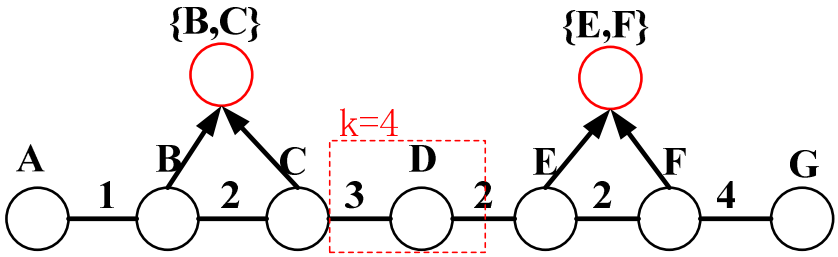
The 2-degree-nodes reduction algorithm is improved from the simple version by merging the 2-degree-nodes on each path which connects a pair of non-2-degree nodes into one or more nodes. The user can decide the value of *k*, which is the window size of node merging, making the distance between any two nodes within the same group not greater than *k*. As a simple example, in Figure 41, which is a part of the graph shown in Figure 42, the 2-degree nodes {B,C,D,E,F} are merged into three groups, {B,C}, {D}, {E,F}. In this example *k*=4, and the distance between any pair of nodes within each group should not be greater than 4. The merging however causes error on the distance queries to the merged node, but it increases the reduction rate. Increasing *k* will merge more nodes into one, which will increase the graph reduction rate with the tradeoff of introducing more errors in the query results.



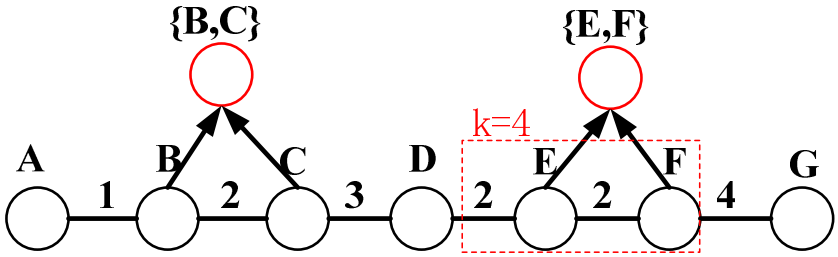
(a) Input path



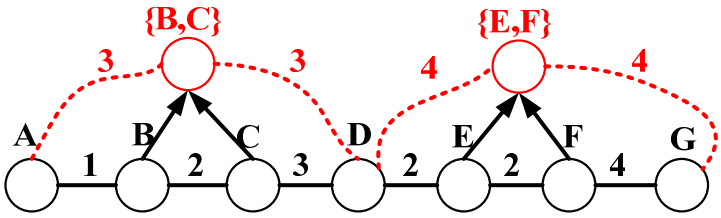
(b) Create the window beginning at A. Merge the nodes inside the window where the shortest distance between any pair of nodes does not exceed 4.



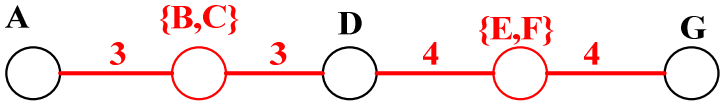
(c) Create the window beginning at the last node C of the previous group. Since D is the only node inside the window, merging is not needed.



(d) Repeat the same process until we reach the end of the path.



(e) Merge the nodes in each group. Construct a link between two groups whose weight is the distance between the last node of the previous group and the first node of the next group.

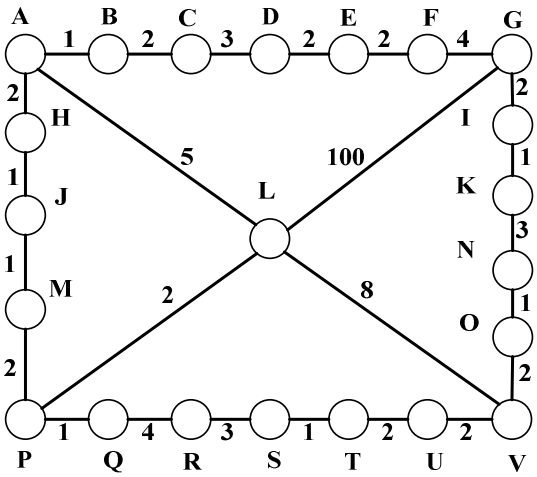


(f) Obtain the merged result.

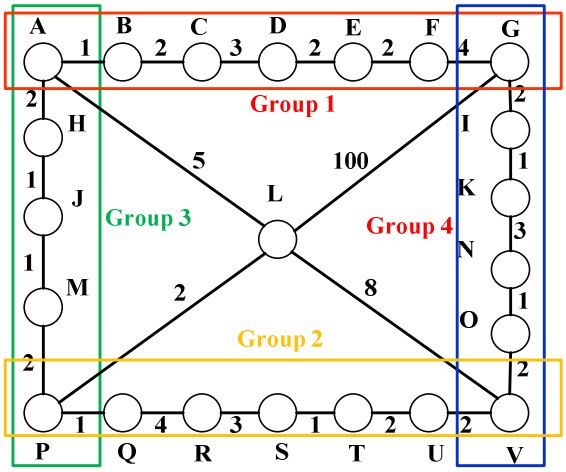
Figure 41. The k-Merge algorithm (window size k=4)

Figure 41 illustrates the k-merge algorithm, which is a part of the 2-degree-nodes reduction LGR-SP algorithm. It merges the 2-degree nodes in a path with a window of size 4. Figure 41(a) shows the input path. In Figure 41(b), we create the first window that starts from the beginning of the path, which is A, but A is not included in the window. We do not merge the two end nodes of the path, as they are not 2-degree nodes. The distance from A to C is $3<k=4$, and the distance from A to D is $6>k=4$, so B and C are inside the window, and D is not. We merge the nodes inside the window, which are B and C. Then we move the window to C which is the last node in the previous widow. In Figure 41(c), the distance between C and D is 3, but the distance between C and E is 5, so only D is inside the window and no merging is needed. We then move the window to D. We repeat the process in Figure 41(d) until we reach the end of the path. During the process we merge E and F. In Figure 41(e), for each merged group, we use the last node of the group to construct a link to the previous merged node (e.g., the distance between the merged node {B,C} and A is the distance between A and C which is AC =3) and a link to the next group (e.g., the distance between the merged node {B,C} and D is the distance between the last node of the group, which is C, and D which is CD =3.) Finally, we obtain the result of the k-merge algorithm in Figure 41(f).

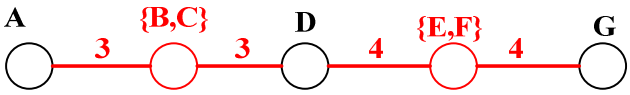
Figure 42 shows the method to compute a reduced graph G' from the original graph G through the 2-degree-nodes reduction LGR-SP algorithm which uses the k-merge algorithm to merge the nodes, where k=4.



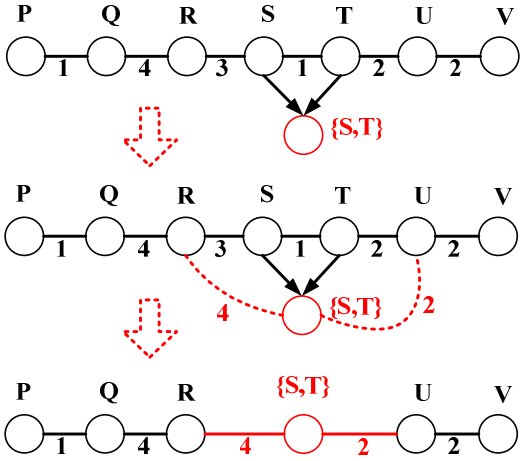
(a) Original graph G



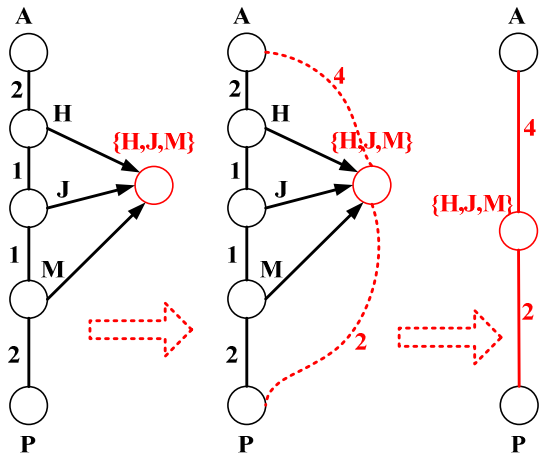
(b) Find all 2-degree linking groups



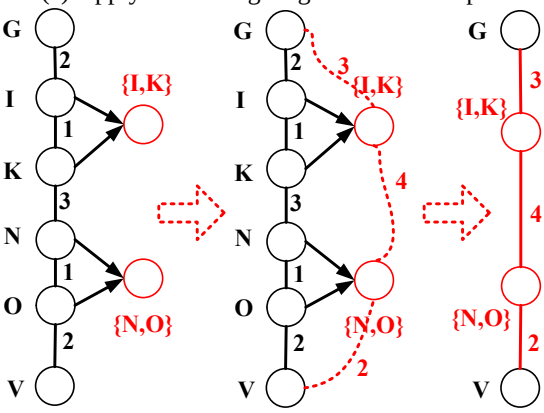
(c) Apply the k-merge algorithm to group 1; the details are explained in Figure 9.



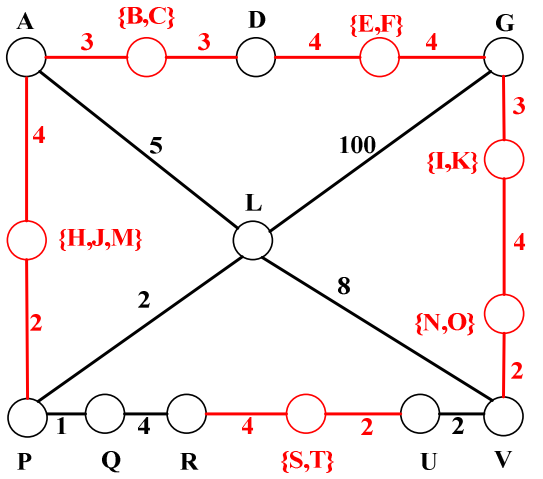
(d) Apply the k-merge algorithm to Group 2.



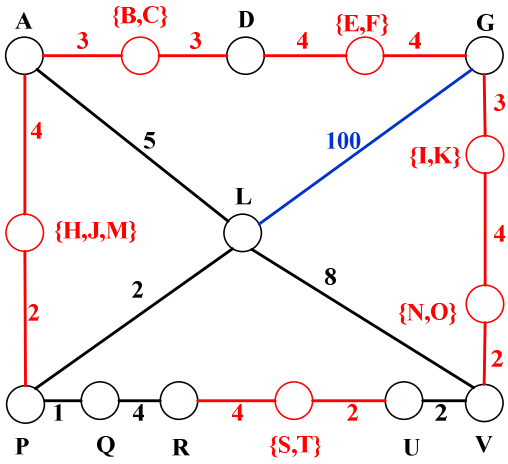
(e) Apply the k-merge algorithm to Group 3.



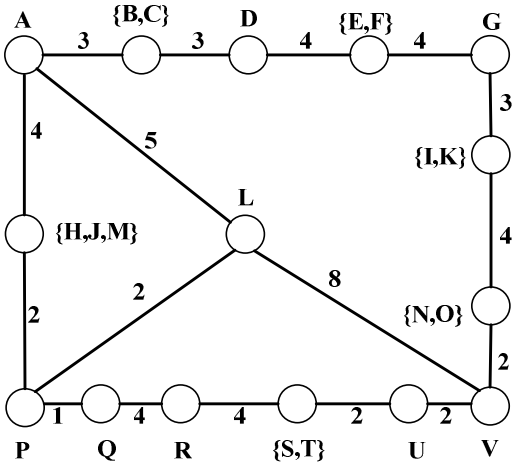
(f) Apply the k-merge algorithm to Group 4.



(g) Merge all results to the original graph G.



(h)Apply the GR-SP algorithm.



(i) Obtain the reduction graph G'

Figure 42. 2-degree-nodes reduction LGR-SP

Figure 42(a) shows the original graph G. In Figure 42(b), we find all 2-degree linking groups (where a 2-degree linking group is a maximum sequence of 2-degree nodes connected), and we can find 4 groups in Figure 42(b). Figure 42(c) shows the result of applying the k-merge algorithm to group 1; the details have already been explained in Figure 41. In Figure 42(d)(e)(f), we apply the k-merge algorithm to groups 2,3 and 4, respectively. In Figure 42(g), we merge all the results into the original graph G. Then, we apply the GR-SP algorithm in Figure 42(h), and obtain the reduction graph G' in Figure 42(i).

The 2-degree-nodes reduction algorithm is improved from the simple version by merging the 2-degree-nodes on each path which connects a pair of non-2-degree nodes. The user can decide the number *k*, which is the window size of merging the nodes, making the distance between any two groups of merged nodes not greater than *k*.

The 2-degree-nodes reduction LGR-SP algorithm and the k-merge algorithm it uses are summarized in the following.

k-merge-algorithm (**Merge the nodes in a path with a given distance threshold.**)

Input: path *p*, threshold *k*
Output: path *p'* that merges some nodes in *p*

```
p' ← Ø
N ← number_of_nodes(p)
V[k] ← nodes of p in travelling order //V[0] is the start node of the path and V[N-1] is the end node
pointer ← 0
Put V[0] into p'
For i ← 0 to N do
  If Dist(V[pointer], V[i]) >k
    Create a merge node of nodes V[pointer+1] to V[i-1] and put it into p'
    Set the edge weight between pointer and the merged nodes in p' to Dist(V[pointer],V[i-1])
    pointer ← i-1
  EndIf
EndFor
If pointer ≠ N-1
  Put V[N-1] into p' and set the distance of the edge to Dist(V[pointer],V[i-1])
EndIf
Return p'
```

The time complexity of the k-merge-algorithm is O(n).

2-degree-nodes reduction LGR-SP algorithm (**Find a reduced graph that approximates the shortest paths between any pair of nodes in a graph.)**

Input: Graph *G*, threshold *k*
Output:2-degree-nodes reduction graph *G''*

```
G' ← G;
N ← Find_2_degree_node(G); //N is the set of the nodes whose degree is 2
C ← Group_by_neighbor(N); // C is a set of groups, each group is a collection of nodes in N that each node in
  the group has at least one neighbor node in the same group, if it is not the only node in the group
For each group C_i in C do
  Find two non-2-degree nodes x and y that are connected to the two ends of the nodes in C_i
  Remove all nodes in C_i from G'
  Find the path p that connects x, C_i, and y from G
  Add k-merge(p, k) to G' //apply k-merge algorithm
EndFor
G''=GR-SP(G');
Return G''
```

The time complexity of the 2-degree-nodes reduction LGR-SP algorithm is $O(n^2)$.

4.2.4. Theorems and Proofs of LGR-SP

Theorem 14 (1+ε LGR-SP). *G'* is the lossy reduction graph of *G* that for every pair of nodes of *G*, the weight of the shortest path found from *G'* is at most 1+ ε times the weight of the shortest path found from *G*.

Proof: For any path $p = (u_0, u_1, \ldots, u_n)$ from G, we can construct a corresponding path *p'* from *G'* by the following method:

For $x = 0$ to n-1

if the edge (u_x, u_{x+1}) belongs to *G'*, put (u_x, u_{x+1}) into *p'*

if the edge (u_x, u_{x+1}) does not belong to *G'*, put the shortest path between u_x and u_{x+1} from *G'* into *p'*.

By the definition of our algorithm, if the edge (u_x, u_{x+1}) does not belong to *G'*, there exists a path *sp'* between u_x and u_{x+1} in *G'* whose weight is at most (1+ε) times greater than edge (u_x, u_{x+1}).

Therefore, the weight of *p'* is at most (1+ε) times greater than the weight of p because all of the components of *p'* are 1 to 1 mappings and the weight of each of them is at most (1+ε) times greater than the weight of the corresponding component of *p*. As a result, for any shortest path in *G*, we can construct a corresponding path in *G'* whose weight is at most (1+ε) times greater. ■

Theorem 15 (2-degree-nodes-cut LGR-SP). *G''* is a lossy reduction graph of *G* that for every pair of nodes of *G* that are not 2-degree nodes, the shortest path found from *G''* is the same as the shortest path found from *G*.

Proof: For any path $p = (u_0, u_1, \ldots, u_n)$ in *G*, for any sub-path $(u_k, \ldots, u_{k+i})$ in *p* where $u_k, \ldots, u_{k+i}$ are nodes with degree 2, we can find an edge (u_{k-1}, u_{k+i+1}) in *G'* whose weight is the same as

the total weight of the sub-path ($u_k,\ldots,u_{k+i}$) in G by the definition of the algorithm. As a result, for any pair of non 2-degree nodes, the shortest path found in G' is the same as in G. Since G'' is the GR-SP reduction from G', G'' also preserves all the shortest paths for any pair of nodes that are not 2-degree nodes. ■

Theorem 16 (k-merge algorithm). For any two nodes, the error between the weight of the shortest path in p and the distance of the shortest path of the corresponding nodes in p' generated by the k-merge algorithm is at most k.

Proof: By the definition of the k-merge algorithm, a series of nodes within a window whose size is k is merged into one node, and the distance is determined by the first node in the group. That is, for any node x in a path p, the distance from the start node to x is $a+e$ where a is the distance of the first node of the group that merged with x and x, and $e<k$. For any two nodes x_m and x_n, the distance between them is $a_m+e_m-a_n-e_n$, and the corresponding distance in p' is a_m-a_n, so the error is e_m-e_n. Since $e_m<k$, $e_n<k$, and $e_m-e_n<k$, the error is bounded by k. ■

Theorem 17 (2-degree-nodes reduction LGR-SP). *G'' is the lossy reduction graph of G* that for every pair of nodes in *G*, the error between the weight of the shortest path found from *G'' and the weight of the shortest path found in G is at most 2k*.

Proof: For the nodes that are not 2 degrees, there is no error for the shortest path between them because it is similar to 2-degree-nodes-cut LGR-SP and already proved in Theorem 2. The error comes from the 2-degree nodes. By Theorem 16, we know that if we want to find the shortest path between a 2-degree node and a non-2-degree node, the error is bounded by k.

Similarly, if we want to find the shortest path between the pair of nodes that both are nodes of 2 degrees, the error is bounded by *2*k*. ■

4.3. Lossy Reduction for the MST Problem

4.3.1. Definition of LGR-MST

Given a graph G, assume an application is only interested in finding the minimum spanning trees, which can be solved by many existing algorithms, e.g., the Prim's algorithm and the Kruskal algorithm. Can we reduce G to a smaller graph G' by a lossy reduction algorithm which can generate some approximate minimum spanning trees? We call this problem the LGR-MST (Lossy Graph Reduction-Minimum Spanning Tree) Problem.

4.3.2. Use Case of LGR-MST and Variations

(1) Use Case: Power Grid

Using a Power Grid Network as an example, shown in Figure 43, some applications may be interested in finding the MST trees.

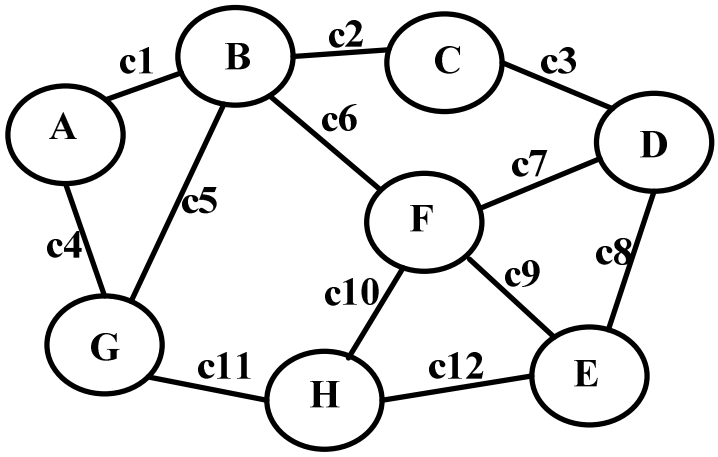


Figure 43. Power Grid

In this use case, every node is a community. The edge between any pair of nodes is the cost of high voltage wire from a node to another node. With a lossy reduction algorithm, we may compute some approximate MSTs from the original graph.

This use case can be solved with the 1+ε LGR-MST, 2-degree-nodes-cut reduction LGR-SP, 2-degree-nodes reduction LGR-MST, 1+ε LGR-WC-MST, 2-degree-nodes-cut LGR-WC-MST and 2-degree-nodes reduction LGR-WC-MST algorithms.

4.3.3. Algorithms of LGR-MST

We can apply the same idea as in the LGR-SP algorithm to solve the LGR-MST problem, as any edge in an MST should be a shortest path also. This is because if it is not the shortest, the total weight of the MST will not be minimal.

(1) 1+ε LGR-MST Algorithm

The 1+ε LGR-MST Algorithm is similar to the 1+ε LGR-SP Algorithm:

1+ε LGR-MST algorithm (Find a reduced graph that preserves a minimum spanning tree of any subset of nodes in a graph with error bound ε.)
Input: Graph *G* **Output**:1+ε Lossy reduction graph *G'*
1. Initialize *G* 2. *G'* ← Apply 1+ε LGR-SP algorithm on *G* 3. **Return** *G'*

The time complexity of the 1+ε LGR-MST algorithm is the same as that of the 1+ε LGR-SP algorithm.

(2) 2-degree-nodes-cut LGR-MST Algorithm

The 2-degree-nodes-cut reduction LGR-MST Algorithm is similar to the 2-degree-nodes-cut reduction LGR-SP Algorithm:

2-degree-nodes-cut LGR-MST algorithm (Find a reduced graph that preserves an MST among any subset of non-2-degree nodes.)
Input: Graph *G* **Output**: 2-degree-nodes-cut reduction graph *G'*
1. Initialize *G* 2. *G'* ← Apply 2-degree-nodes-cut LGR-SP algorithm on *G* 3. **Return** *G'*

The time complexity is the same as that of the 2-degree-nodes-cut LGR-SP algorithm.

(3) 2-degree-nodes reduction LGR-MST Algrorithm

The 2-degree-nodes reduction LGR-MST Algorithm is similar to the 2-degree-nodes reduction LGR-SP Algorithm:

2-degree-nodes reduction LGR-MST algorithm (Find a reduced graph that preserves an MST among any subset of nodes with error bound *N*k*.)
Input: Graph *G*, threshold *k* **Output**: 2-degree-nodes reduction graph *G'*
1. Initialize *G* 2. *G'* ← Apply 2-degree-nodes reduction LGR-SP algorithm with threshold *k* 3. **Return** *G'*

The time complexity is the same as that of the 2-degree-nodes reduction LGR-SP algorithm.

4.3.4. Algorithms of LGR-WC-MST

Given a graph G, assume an application is only interested in finding the MSTs among any set of nodes with a weight constraint on the edges (i.e., the weight of each edge has to be less than or equal to a constant *w*). The idea of the lossy reduction algorithms is to remove all the edges whose weight is greater than the weight constraint *w* and apply an LGR-MST algorithm.

(1) 1+ε LGR-WC-MST algorithm

The 1+ε LGR-WC-MST algorithm removes the edges that violate the constraint and apply the 1+ε LGR-MST algorithm:

1+ε LGR-WC-MST algorithm (Find a reduced graph for minimum spanning tree of any subset of nodes in a graph with the weight constraint *w*)
Input: Graph G, weight constraint w **Output**:**1+ε Lossy reduction graph G'**
1. Initialize *G'*; 2. *G'*=Remove all edges whose weight is greater than *w* from *G*. 3. Apply 1+ε LGR-SP algorithm on *G'*. 4. **Return** *G'*

The time complexity is the same as 1+ε LGR-SP algorithm.

(2) 2-degree-nodes-cut LGR-WC-MST

The 2-degree-nodes-cut LGR-WC-MST removes the edges that violate the constraint and apply 2-degree-nodes-cut LGR-MST.

The **2-degree-nodes-cut LGR-WC-MST** can be described as the following.

2-degree-nodes-cut LGR-WC-MST algorithm (Find a reduced graph that preserves an MST that does not violate the weight constraint *w* among any subset of non-2-degree nodes.)
Input: Graph *G*, weight constraint *w* **Output**:**2-degree-nodes-cut reduction graph G'**
1. Initialize *G'*; 2. *G'*=Remove all edges whose weight is greater than *w* from *G*. 3. Apply 2-degree-nodes-cut LGR-SP algorithm 4. **Return** *G'*

The time complexity is the same as that of the 2-degree-nodes-cut LGR-SP algorithm.

(3) 2-degree-nodes reduction LGR-WC-MST

In the 2-degree-nodes reduction LGR-WC-MST algorithm, after removing the edges that violate the weight constraint, we also need to make sure that the k-merge algorithm used in 2-degree-nodes reduction LGR-MST does not generate any result that violates the weight constraint. That is, the threshold k cannot be greater than the weight constraint *w*.

2-degree-nodes reduction LGR-WC-MST algorithm (Find a reduced graph that preserves an MST that does not violate the weight constraint *w* among any subset of nodes with error bound *N*k*.)
Input: Graph *G*, threshold k, weight constraint *w* **Output**:**2-degree-nodes reduction graph G'**
1. Initialize *G'*; 2. *G'*=Remove all edges whose weight is greater than *w* from *G*. 3. Apply the 2-degree-nodes reduction LGR-SP algorithm 4. **Return** *G'*

The time complexity is the same as 2-degree-nodes reduction LGR-SP algorithm.

4.3.5. Algorithms of LGR-DC-MST

Given a graph G, assume an application is only interested in finding the minimum spanning trees among any set of nodes with a degree constraint (i.e., the degree of each node has to be less than or equal to a constant *d*). The idea of the lossy reduction algorithms is to apply the LGR-MST algorithm first. If any of the nodes in the reduced graph has a degree more than *d*, it is possible that an MST not violating the degree constraint cannot be found in this graph. Therefore, we find the shortest paths that connect the neighbors of the nodes that violate the degree constraint to create alterative paths for constructing a DC-MST.

(1) 1+ε LGR-DC-MST algorithm

The 1+ε LGR-DC-MST algorithm applies the 1+ε LGR-MST algorithm and then adds alternative paths between the neighbors of the nodes that violate the degree constraint.

The **1+ε LGR-DC-MST** is described in the following.

1+ε LGR-DC-MST algorithm (Find a reduced graph that preserves an MST that do not violate the degree constraint *d* among any subset of nodes with a bounded error ε.)

Input: Graph *G*, degree constraint *d*
Output: 1+ε LGR-DC-MST reduction graph *G'*

```
G' ← Apply the 1+ε LGR-MST on G
G'' ← G
N_v ← ∅ //initialize the condition violating nodes' set as an empty set
For i ← 0 to |N| do
  If node_degree(i) > d
    Insert node i to N_v
    //add the condition violating node to the set of condition violating nodes;
  EndIf
For all node n in N_v do
  N_b ← n's neighbors in G''
  remove all the edges connected n from G''
    For each pair of nodes in N_b, find the shortest paths between them
    Add those paths to G'
  EndFor
EndFor
Return G'
```

The time complexity of the 1+ε LGR-DC-MST is the same as that of the 1+ε LGR-SP algorithm.

(2) 2-degree-nodes-cut LGR-DC-MST

The 2-degree-nodes-cut LGR-DC-MST algorithm applies the 2-degree-nodes-cut LGR-MST algorithm and then adds alternative paths between the nodes that neighbor the nodes violating the degree constraint.

The **2-degree-nodes-cut LGR-DC-MST** is described in the following.

2-degree-nodes-cut LGR-DC-MST algorithm (Find a reduced graph that preserves the MSTs that do not violate the degree constraint *d* among any subset of 2-degree nodes.)

Input: Graph G, degree constraint d
Output: 2-degree-nodes-cut reduction graph *G'*

```
G' ← Apply the 2-degree-nodes-cut LGR-MST on G
G'' ← G
N_v ← ∅ //initialize the condition violating nodes' set as an empty set
For i ← 0 to |N| do
  If node_degree(i) > d
    Insert node i to N_v
    //add the condition violating node to the set of condition violating nodes;
  EndIf
For all node n in N_v do
  N_b ← n's neighbors in G''
  remove all the edges connected n from G''
    For each pair of nodes in N_b, find the shortest paths between them
    Add those paths to G'
  EndFor
EndFor
Return G'
```

The time complexity of the **2-degree-nodes-cut LGR-DC-MST** time is the same as that of the 2-degree-nodes-cut LGR-SP algorithm.

(3) 2-degree-nodes reduction LGR-DC-MST

The 2-degree-nodes reduction LGR-DC-MST algorithm applies the 2-degree-nodes reduction algorithm and then finds alternative paths between the nodes that neighbor the nodes violating the degree constraint. It also applies the k-merge algorithm to the alternative paths.

2-degree-nodes reduction LGR-DC-MST algorithm (Find a reduced graph that preserves an MST that does not violate the degree constraint *d* among any subset of nodes with a bounded error *N*k*.)

Input: Graph *G*, threshold *k*, degree constraint *d*
Output: 2-degree-nodes reduction graph *G'*

```
G' ← Apply the 2-degree-nodes reduction LGR-MST on G
G'' ← G
N_v ← ∅ //initialize the condition violating nodes' set as an empty set
For i ← 0 to |N| do
  If node_degree(i) > d
    Insert node i to N_v
    //add the condition violating node to the set of condition violating nodes;
  EndIf
For all node n in N_v do
  N_b ← n's neighbors in G''
  remove all the edges connected n from G''
    For each pair of nodes in N_b, find the shortest paths between them
    Apply k-merge algorithm with threshold k on these paths
    Add those paths to G'
  EndFor
EndFor
Return G'
```

The time complexity is the same as that of the 2-degree-nodes reduction LGR-SP algorithm.

4.3.6. Theorems and Proofs of LGR-MST and Variations

Theorem 18 (1+ε LGR-MST). *G'* is a lossy reduction graph that for every subset of nodes, the weight of the MST found from *G'* is at most 1+ ε times than weight of the MST found from *G*.

Proof: By the proof in Theorem 14, we know that for any path in *G*, we can construct a corresponding path in *G'* whose weight is at most 1+ ε times greater. Therefore, we can divide the MST from *G* in multiple paths, and construct the corresponding paths in *G'*. Since the weight of every path in *G'* is at most 1+ ε times of the weight of the corresponding path in *G*, we can merge them into a spanning tree that covers all the given nodes with the total weight at most 1+ ε times greater. Since the MST found from *G'* is the minimum by definition, it is smaller than the spanning tree we construct, whose weight is at most 1+ ε times of the weight for the MST found in *G*. ■

Theorem 19 (2-degree-nodes-cut LGR-MST). *G'* is a lossy reduction graph that for any subset of nodes that are not 2-degree nodes, the MST found from *G'* is the same as the MST found from *G*.

Proof: Similar to Theorem 15, we can divide the MST in *G* to sub-paths that each connects two non-2-degree nodes. For any subpath $p = (u_0, u_1, \ldots, u_n)$ belonging to the MST of *G*, where only u_0 and u_n are not 2-degree nodes, we can find a corresponding edge in *G'* with the same weight. By replacing the path in *G* by the edge in *G'*, we are able to obtain a spanning tree that spans on the non-2-degree nodes in *G'* which has the same weight as the corresponding MST in *G*. ■

Theorem 20 (2-degree-nodes reduction LGR-MST). *G'* is a lossy reduction graph that for any subset of nodes where the total number of 2-degree nodes in the subset is *N*. The MST found among any subset of nodes from *G'* has at most $N*k$ errors compared to the corresponding MST found in *G*.

Proof: For any two nodes that are not 2 degrees, there is no error for the shortest path between them. The error comes from the 2-degree nodes. By Theorem 16, we know that the error of the shortest distance to a 2-degree node is bounded by *k*. Therefore, if there are *N* 2-degree nodes in the spanning tree, the error is bounded by $N*k$. ■

Theorem 21 (1+ε LGR-WC-MST algorithm). *G'* is a lossy reduction graph that for every subset of nodes of *G'*, the weight of each weight-constrained MSTs found from *G'* is at most 1+ ε times of the weight of the corresponding weight-constrained MST found from *G*.

Proof: The algorithm removes all the edges that violate the weight constraint before finding the MSTs. Therefore, any MST found from the reduced graph will not contain any edge whose weight is greater than *w*, so it is a weight-constrained MST. By Theorem 18, we know that its weight is at most 1+ ε of the weight for the corresponding weight-constrained MST found in G. ■

Theorem 22 (2-degree-nodes-cut LGR-WC-MST). *G'* is a lossy reduction graph that for any subset of nodes that are not 2-degree nodes, the weight-constrained MSTs found from *G'* is the same as the weight-constrained MSTs found from *G*.

Proof: The algorithm removes all the edges that violate the weight constraint before finding the MSTs. Therefore, each MST found from the reduced graph will not contain any edge whose weight is greater than *w*, so it is a weight-constrained MST. By Theorem 19, we know that it is the same as the weight-constrained MST found in *G*. ■

Theorem 23 (2-degree-nodes reduction LGR-WC-MST). *G'* is the lossy reduction graph that for any subset of nodes and the total number of 2-degree nodes in the subset is *N*, the weight-constrained MST over any subset of nodes found from *G'* has at most $N*min(k,w)$ errors compared to the corresponding weight-constrained MST found in G.

Proof: The algorithm removes all the edges that violate the weight constraint before finding the MSTs. Therefore, the MST found from the reduced graph will not contain any edge whose weight is greater than w, so it is a weight-constrained MST. By making the threshold *k* not greater than *w*, it guarantees that the distance between two merged nodes after applying the k-merge algorithm will not violate the weight constraint. Since the threshold used in the algorithm

is either *k* or w depending on which is greater, by Theorem 20 we know that the error is at most *N*min(k,w)*. ■

Theorem 24 (1+ε LGR-DC-MST algorithm). *G'* is a lossy reduction graph that for every subset of nodes, the weight of each degree-constrained MST found from *G'* is at most 1+ ε times of the weight for the corresponding degree-constrained MST found from *G*.

Proof: For any node in the reduced graph that is more than k degrees, we add the alternative paths to their neighbors. Therefore, if the MST found in *G'* has any node that is more than k degrees, we can always reduce the degree of that node by choosing the alternative paths instead. By Theorem 18, the weight of any degree-constrained MST found in *G'* is at most 1+ ε times of the weight for the corresponding weight-constrained MST found from *G*. ■

Theorem 25 (2-degree-nodes-cut LGR-DC-MST). *G'* is the lossy reduction graph that for any subset of nodes that are not 2-degree nodes, the degree-constrained MSTs found from *G'* are the same as the degree-constrained MSTs found from *G*.

Proof: We first apply the DC-MST algorithm, so it contains the degree-constrained MSTs found from *G*. Then we apply the 2-degree-node cut LGR-MST algorithm, so all the 2-degree nodes are removed. For any subpath in a degree-constrained MST of *G* which is composed of only 2-degree-nodes, we can find the corresponding edge in *G'* whose weight is the same as the total weight of the subpath. Therefore, by replacing the subpath in *G* by the edge in *G'*, we can construct the degree-constrained MST in *G'* with the same weight. ■

Theorem 26 (2-degree-nodes reduction LGR-DC-MST). *G'* is a lossy reduction graph that for any subset of nodes and the total number of 2-degree nodes in the subset is *N*, a degree-constrained MST over any subset of nodes found from *G'* has at most *N***k* errors to the corresponding degree-constrained MST found in *G*.

Proof: We first apply the 2-degree-node reduction LGR-MST algorithm to the graph. Then for any node that violates the degree constraint, we find the alternative paths in the original graph that connect the node's neighbors. If an alternative path contains 2-degree nodes, we apply the k-merge algorithm to it and merge the 2-degree nodes. By Theorem 16, we know that the error caused by k-merge algorithm is *k*. Since there are at most *N* 2-degree nodes we want to cover in the spanning tree, the error is at most *N***k*. ■

4.4. Lossy Reduction for the Reachability Problem

4.4.1. Definition of LGR-Reachability

Given a graph *G*, assume an application is only interested in knowing if node *x* can reach node *y* within distance *d*, which is often useful for fast distance estimation. Can we reduce *G* to a smaller graph *G'* which preserves the information that can answer distance constrained reachability queries with acceptable correctness? We call this problem the LGR-Reachability (Lossy Graph Reduction-Reachability) Problem.

4.4.2. Use Case of LGR-Reachability (Travel Planning)

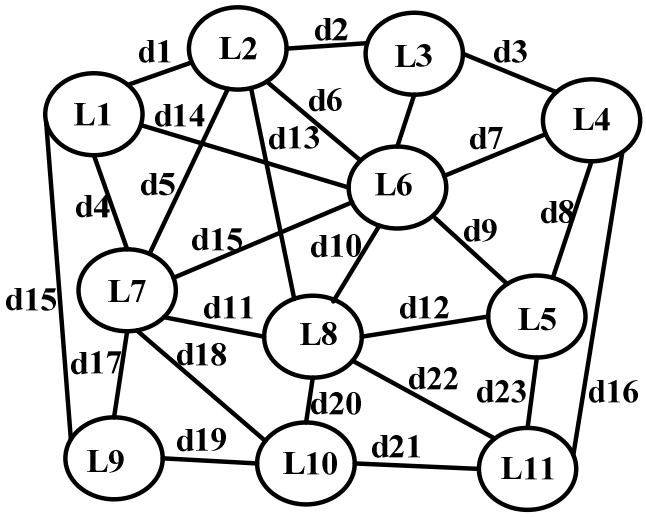


Figure 44. Travel Route Planning

In a travel planning system, we need to query the reachability from a location to another location and the distance needs to satisfy some reachability constraint, as shown in Figure 44 in which each node is a location and the weight of each edge is the distance from a location to another location.

4.4.3. Algorithms for LGR-Reachability

The algorithms used for the LGR-Reachability problem are similar to those for the shortest path problem. We list them in the following.

1+ε LGR-Reachability algorithm (Find a reduced graph for the distance-constrained reachability problem between any pair of nodes in a graph.)
Input: Graph *G* **Output**:1+ε Lossy reduction graph *G'*
1. Initialize *G* 2. *G'* ← Apply 1+ε LGR-SP algorithm on *G* 3. **Return** *G'*

2-degree-nodes-cut LGR-Reachability algorithm (Find a reduced graph for the distance-constrained reachability problem between any pair of non-2-degree nodes in a graph.)
Input: Graph *G* **Output**: 2-degree-nodes-cut reduction graph *G'*
1. Initialize *G* 2. *G'* ← Apply 2-degree-nodes-cut LGR-SP algorithm on *G* 3. **Return** *G'*

2-degree-nodes reduction LGR-Reachability algorithm (Find a reduced graph for the distance-constrained reachability problem between any pair of nodes in a graph.)
Input: Graph G, threshold k **Output**: 2-degree-nodes reduction graph *G'*
1. Initialize *G* 2. *G'* ← Apply 2-degree-nodes reduction LGR-SP algorithm on *G* 3. **Return** *G'*

4.4.4. Theorems and proofs of LGR-Reachability

Theorem 27 (1+ε LGR-Reachability). *G'* is a lossy reduction graph that for any pair of nodes *x* and *y*, if *x* can reach *y* within distance *d* in *G'*, then *x* can reach *y* within distance *d* in *G*.

Proof: By Theorem 14, for any pair of nodes we can find a path that connects them in *G'* whose weight is at most 1+ ε of the shortest distance found in *G*. Therefore, if the weight of the path found in *G'* is smaller than *d*, the distance of the shortest path found in *G* is also smaller than *d*. Note that there are still false negatives in *G'*. This is because if *x* cannot reach *y* within distance *d* in *G'*, it does not mean *x* cannot reach *y* within distance *d* in G. ■

Theorem 28 (2-degree-nodes-cut LGR-Reachability). *G'* is a lossy reduction graph that between any pair of non-2-degree nodes, the answer for the distance-constrained reachability problem from *G'* is the same as that from *G*.

Proof: By Theorem 15, we know that between any pair of non-2-degree nodes, the shortest path found in *G'* is the same as what is found in *G*. Therefore, the answer for the distance-constrained reachability problem will be the same, because if the distance of the shortest path between nodes *x* and *y* is *d*, then it means *x* can reach *y* within *d*. ■

Theorem 29 (2-degree-nodes reduction LGR-Reachability). *G'* is a lossy reduction graph that between any pair of nodes *x* and *y*, if *x* can reach *y* within distance $d+2k$ in *G'*, then *x* can reach *y* within distance *d* in *G*.

Proof: By Theorem 17, we know that the error is at most $2k$ of for a shortest path in *G'*. Therefore, the distance-constraint reachability problem can also have an error of at most $2k$. For any pair of nodes *x* and *y*, if *x* can reach *y* within distance $d+2k$ in *G'*, then *x* can reach *y* within distance *d* in *G*, which means the algorithm returns a more correct result if $d >> k$, and it will not work well if *d* is close to *k*, so adjusting the threshold *k* is important. ■

4.5. The Budget Cut Problem

The goal of the query-based graph data reduction is to reduce the size of a graph while preserving information to answer some specific queries from the application.

If the goal of graph data reduction is not the size of the graph, but other real-life functions, we can define some different types of problems. We name this kind of problems global constraint graph reduction problems.

There may be many different graph properties that can be the optimization goals for global constraint graph reduction problems, e.g., total weights, average degrees, centrality, …,etc. In this thesis, we will use the "budget cut problem" as an example. The goal of the budget cut problem is to reduce the budget, which is the total cost of a graph, while still preserving some desired properties in a graph.

4.5.1. The Budget Cut Problem

(1) Definition of the Budget Cut Problem

The basic problem is to reduce the budget and maximize the benefit.

Given a graph, each edge of the graph has two weights: profit and cost.

Given a number *N*, we want to remove *N* edges.

Different ways to evaluate the benefit will generate different problems. For example, benefit may be a liner function, e.g., (*a**total profit) – (*b**total cost), or it can be a ratio, e.g., (total profit) / (total cost).

In this thesis, we will focus on linear benefit functions in the budget cut problem and its variation. It should be noted that most optimization problems on ratios, e.g., minimum ratio spanning tree, are NP-hard problems [85].

(2) Use Case of Budget Cut Problem (Airline Operations)

An airline company wants to archive the biggest economical benefit by cutting some edges or canceling some nodes. In this use case, as shown in Figure 45.

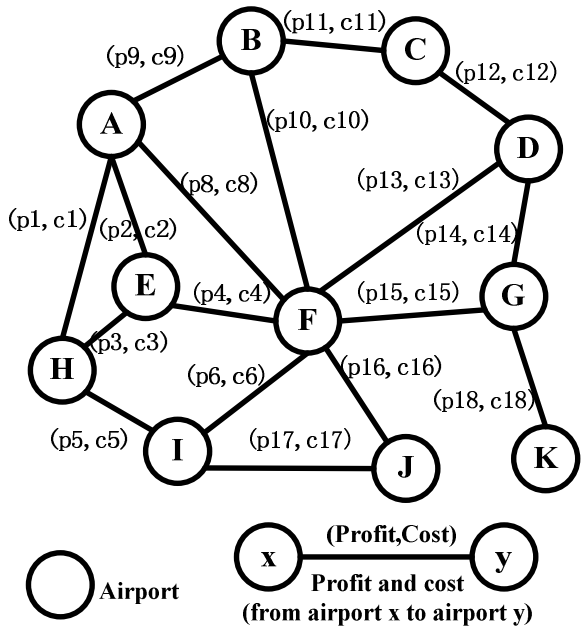
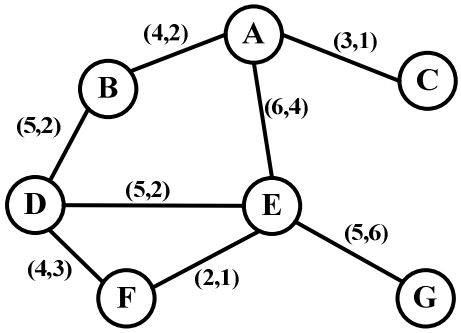


Figure 45. Airline Profit/Cost Network

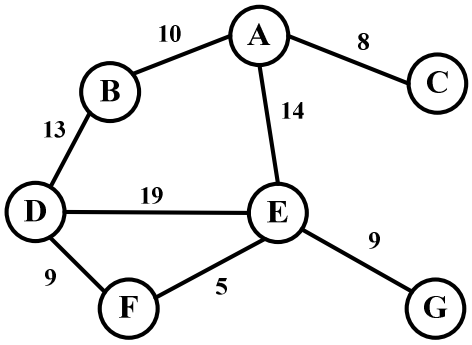
In this use case, each node is an airport and each edge is a flight between two airports. There are two parameters for each edge. One parameter is profit and the other parameter is cost.

(3) Algorithm for the Budget Cut Problem

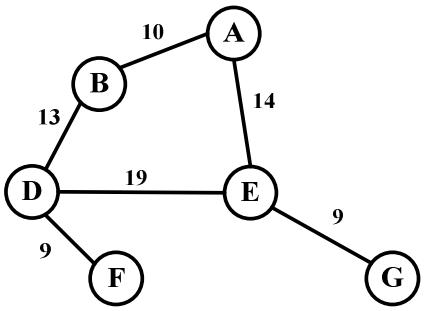
Figure 46 illustrates the algorithm for the Budget Cut Problem with an example. In this example, we hope to remove say 2 edges from the original graph G. Figure 46(a) shows the original graph. Each edge has two weights: profit and cost. Figure 46(b) shows the benefit of each edge using the benefit function 3*profit-cost in this example. Note that because the benefit function is linear, the overall benefit of the graph (3*total profit)-(total cost) can be calculated by summazing the the benefits of each edge. This makes the problem an optimization problem on the single objective: benefit. Then we remove the edges in the increasing order of benefit until we remove 2 edges. The result is shown in Figure 46(c).



(a) A profit and cost graph



(b) The corresponding benefit graph



(c)Reduced graph (n=2)
Figure 46. The Budget Cut Problem

The algorithm for the Budget Cut Problem calculates the benefit of each edge, sorts the edges in increasing order, and then removes the smallest edges. The GCGR-BC (Global Constraint Graph Reduction-Budget Cut) algorithm is described in the following.

GCGR-BC algorithm (Find a reduced graph for the Budget Cut Problem that maximizes the benefit after removing a given number of edges.)

Input: Graph *G*, a linear benefit function *f*, a number of removing edge *n*
Output: a reduction graph *G'*

```
G' ← G
For each edge e in G' do
   benefit(e) = f(profit(e),cost(e))
EndFor
Sort the edges in increasing order of benefit.
For i ← 1 to n do
   Remove the edge with minimum benefit from G'
EndFor
Return G'
```

The time complexity of the GCGR-BC algorithm is O(E log E) if using merge sort, where E is the number of edges.

(4) Theorem and Proof

Theorem 30: GCGR-BC algorithm produces a subgraph *G'* which has the maximum benefit compared to all subgraphs that have *n* edges fewer than the original graph.

Proof: We can prove this by induction. In the beginning, clearly n=0 holds because there is only one possibility of not removing any edge: the original graph itself. If $n=k$ holds, let G_k be the subgraph after removing k edges with the maximum benefit B_k. When $n=k+1$, assume that we

have a different subgraph G_{k+1}', which removes a different edge *e'* while our algorithm removes *e* to get G_{k+1}. Because the benefit is a linear function *f* of profit and cost, $B_{k+1}=B_k-f(profit(e), cost(e))$, and $B_{k+1}'=B_k-f(profit(e'),cost(e'))$. Comparing *e* and *e'* we have 3 cases:

1. $f(profit(e),cost(e))> f(profit(e'),cost(e'))$. This is not possible because the algorithm chooses *e* that has the minimum benefit.

2. $f(profit(e),cost(e))= f(profit(e'),cost(e'))$. In this case, $B_{k+1}= B_{k+1}'$

3. $f(profit(e),cost(e))< f(profit(e'),cost(e'))$. In this case, $B_{k+1}>B_{k+1}'$

Concluding the 3 cases, we know that B_{k+1} is greater than or equal to any possible B_{k+1}', so G_{k+1} is a subgraph with the maximum benefit. ■

4.5.2. Shortest Path Budget Cut Problem

(1) Definition of Shortest Path Budget Cut Problem

Similar to the Budget Cut Problem, each edge has profit and cost. Given a number *n*, we want to remove *n* edges, and maximize the profit, which is a linear function of profit and cost. However, in the Shortest Path Budget Cut Problem, each edge also has a third weight on it which is distance. We want to maximize the profit without increasing the distance of travel between any two nodes. In other words, we want to preserve all the shortest paths when we reduce the graph.

(2) Use Case of Shortest Path Budget Cut Problem

A logistics Company wants to archive the largest benefit by cutting some edges or canceling some nodes. In this use case, we focus on the budget constraint. Figure 47 shows the use case, where each node is a location and each edge is the logistic path from a location to another location. There are three parameters for each edge. One parameter is profit, another parameter is cost, and the last parameter is distance from a location to another location. The budget cut should not sacrifice the shortest distance between any two locations.

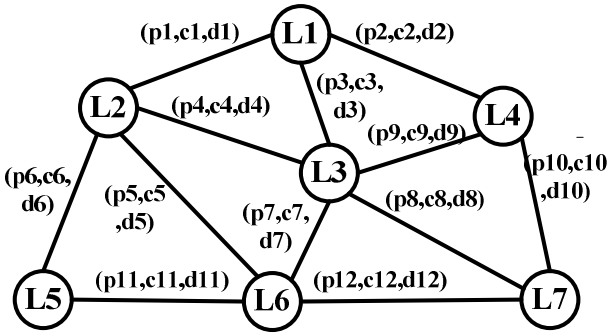
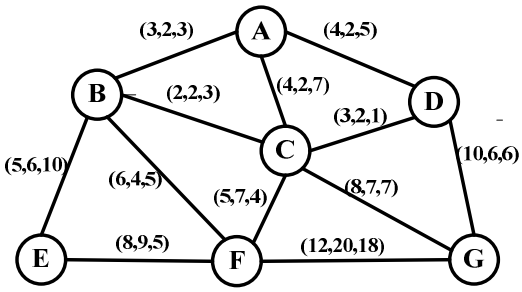


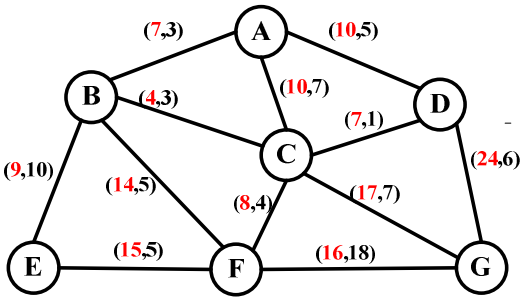
Figure 47. A logistics network

(3) Algorithm of Shortest Path Budget Cut Problem

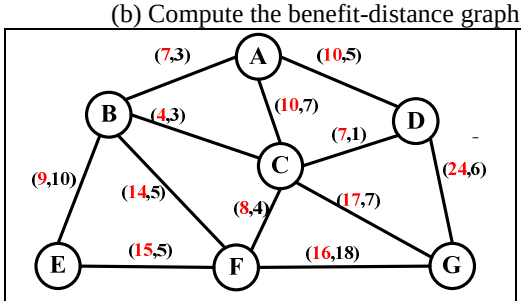
Figure 48 shows an example for the shortest path budget cut problem. In this example, we hope to remove 3 edges from the original graph G.



(a) Original graph

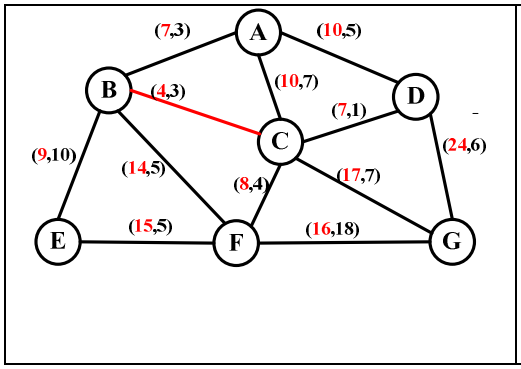


(b) Compute the benefit-distance graph



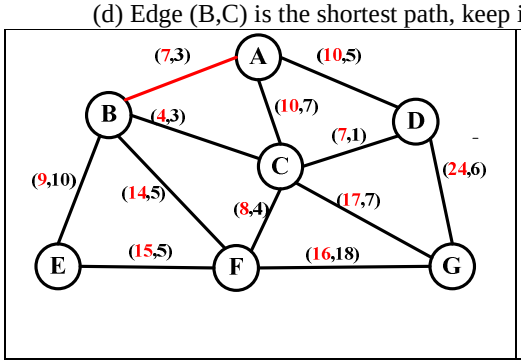
BC=4
AB=7
CD=7
CF=8
BE=9
AC=10
AD=10
BF=14
EF=15
FG=16
CG=17
DG=24

(c) Sorting according to benefit (in increasing order)



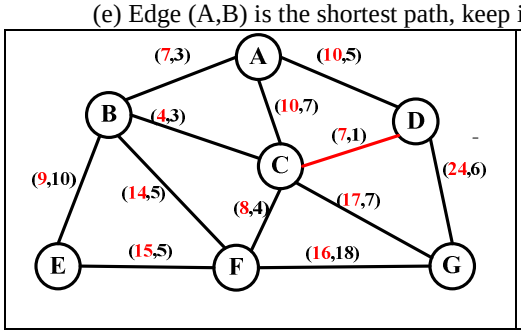
[BC]
AB
CD
CF
BE
AC
AD
BF
EF
FG
CG
DG

(d) Edge (B,C) is the shortest path, keep it.



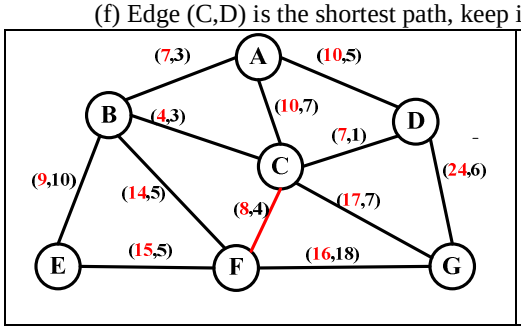
[AB]
CD
CF
BE
AC
AD
BF
EF
FG
CG
DG

(e) Edge (A,B) is the shortest path, keep it.



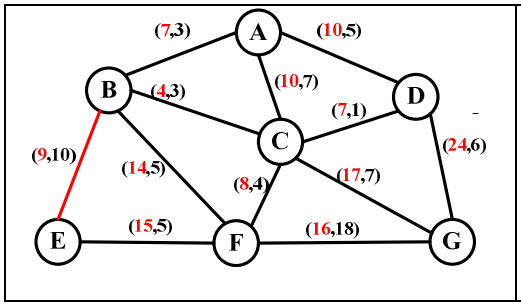
[CD]
CF
BE
AC
AD
BF
EF
FG
CG
DG

(f) Edge (C,D) is the shortest path, keep it.



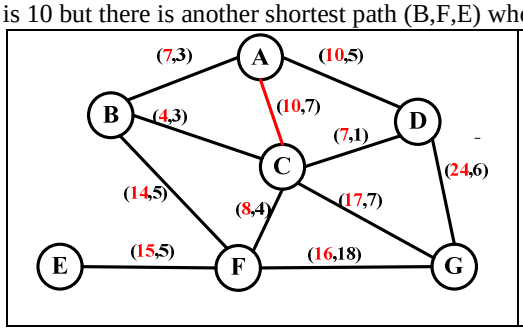
[CF]
BE
AC
AD
BF
EF
FG
CG
DG

(g) Edge (C,F) is the shortest path, keep it.



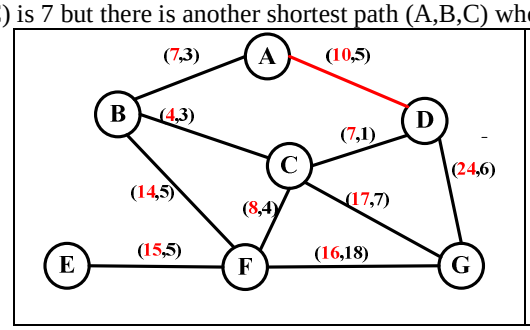
[BE]
AC
AD
BF
EF
FG
CG
DG

(h) Distance of (B,E) is 10 but there is another shortest path (B,F,E) whose weight is 10. Remove (B,E).



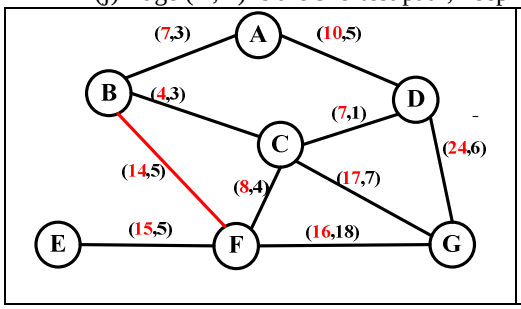
[AC]
AD
BF
EF
FG
CG
DG

(i) Distance of (A,C) is 7 but there is another shortest path (A,B,C) whose weight is 6. Remove (A,C).



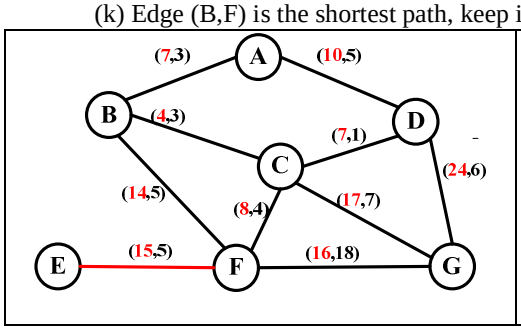
[AD]
BF
EF
FG
CG
DG

(j) Edge (A,D) is the shortest path, keep it.



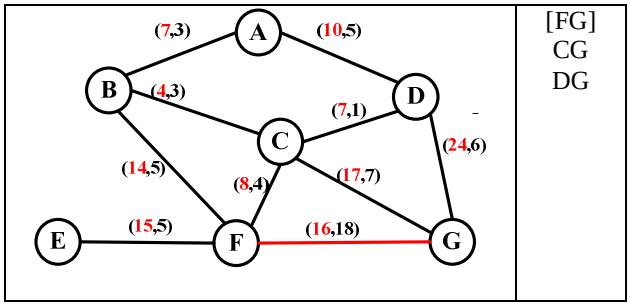
[BF]
EF
FG
CG
DG

(k) Edge (B,F) is the shortest path, keep it.

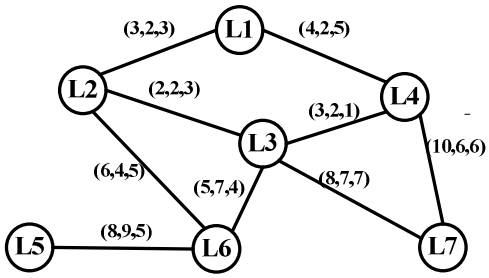


[EF]
FG
CG
DG

(l) Edge (E,F) is the shortest path, keep it.



(m) Distance of FG is 18 but there is another shortest path (F,C,G) whose weight is 11. Remove (F,G).



(n)Obtain the reduced graph G'

Figure 48. A shortest path budget cut problem

Figure 48(a) shows the original graph G. In Figure 48(b), we compute the benefit-distance graph from G. In Figure 48(c), we sort the edges in ascending order. In Figures 48(d),(e),(f),(g), we keep those edges because they are the shortest paths between the two ends. In Figures 48(h),(i), we remove those edges because we can find other shortest paths connecting their incident nodes. In Figures 48(j),(k),(l), we keep those edges because they are the shortest paths. In Figure 48(m) we remove the edge and since we have already removed three edges, we can end the algorithm. Finally we can obtain the reduction graph G in Figure 48(n).

The algorithm for the shortest path budget cut problem calculates the benefit of each edge and sorts the edges in increasing order. The algortihm then removes the minimum benefit edges if removing it does not increase the shortest distance between the incident nodes. The GCGR-BC-SP (Global constraint graph reduction-budget cut–shortest path) algorithm is summarized in the following.

GCGR-BC-SP algorithm (Find a reduced graph for the Shortest Path Budget Cut Problem that maximizes the benefit after removing a given number of edges without affecting the shortest distance between any pair of nodes.)

Input: Graph *G*, a linear benefit function *f*, the number of edges to be removed *n*
Output: a reduced graph *G'*

```
G' ← G
For each edge e in G' do
   benefit(e) = f(profit(e),cost(e))
EndFor
E ← Sort the edges in increasing order of benefit
For i ← 1 to n do
   e ← get_min(E); //get the minimum edge
   Remove e from G'
   If distance(e) < Dijkstra(G',e)
      Add e to G'. //Use Dijkstra algorithm to calculate the shortest distance between the incident nodes of e
      after removing e from G'. If the distance is increased, put e back
   EndIf
   Remove e from E
EndFor
Return G'
```

The time complexity of the **GCGR-BC-SP algorithm** is E*T(Dijkstra), where T(Dijkstra) is the time complexity of the Dijkstra algorithm.

(4) Theorem and Proof

Theorem 31: GCGR-BC-SP algorithm produces a subgraph *G'* which has the maximum benefit compared to all subgraphs that has *n* edges fewer than the original graph, without increasing the shortest distance between any pair of nodes.

Proof: We can prove this by induction. At the beginning, apparently n=0 holds because there is only one possibility of not removing any edge: the original graph itself. If $n=k$ holds, let G_k be the subgraph after removing k edges and it has the maximum benefit B_k, and all the shortest distances are preserved. When $n=k+1$, assume that we have a different subgraph G_{k+1}', which removes a different edge e' while our algorithm removes e to get G_{k+1}. Note that by the algorithm, removing e does not affect the shortest distance because the weight of the edge is smaller than the shortest distance the incident nodes after removal. So the shortest distance is still

preserved in G_{k+1}. Since the benefit is a linear function f of profit and cost, $B_{k+1}=B_k-f(profit(e),cost(e))$, and $B_{k+1}'=B_k-f(profit(e'),cost(e'))$. Comparing e and e' we have 3 cases:

1. $f(profit(e),cost(e))> f(profit(e'),cost(e'))$. By the algorithm, e has the minimum benefit among all the edges that does not increase the distance of the incident nodes. As a result, removing e' increases the distance between the incident nodes of e', so G_{k+1}'does not preserve the shortest distances.

2. $f(profit(e),cost(e))= f(profit(e'),cost(e'))$. In this case, $B_{k+1}= B_{k+1}'$

3. $f(profit(e),cost(e))< f(profit(e'),cost(e'))$. In this case, $B_{k+1}>B_{k+1}'$

Concluding the 3 cases, we know that comparing with any other subgraph, either G_{k+1}'violates the shortest distance preserving constraint, or $B_{k+1}$is greater than or equal to B_{k+1}', so G_{k+1} is a subgraph with the maximum benefit without increasing the shortest distance between any pair of nodes. ■

4.5.3. The K-connectivity Budget Cut Problem

(1) Definition of the K-connectivity Budget Cut Problem

Similar to the Budget Cut Problem, each edge has profit and cost. Given a number n, we want to remove n edges, and maximize the profit which is a linear function of profit and cost. However, in the K-connectivity Budget Cut Problem, we want to preserve the edge-connectivity in the graph, which is often useful for network reliability.

Assume the original graph has edge-connectivity c. That is, for any pair of nodes in the graph, we can find at least c separate paths connecting them. In the K-connectivity budget cut problem, the application specifies a connectivity constraint k which cannot be greater than c, and we want

the reduced graph after budget cut to have the edge-connectivity *k*. Note that when *k*=1 the problem is reduced to the maximum spanning tree problem.

(2) Use Case of K-connectivity Budget Cut Problem

A logistics Company wants to archive the largest benefit with the k-connectivity constraint by cutting some edges or canceling some nodes. Figure 49 shows an example.

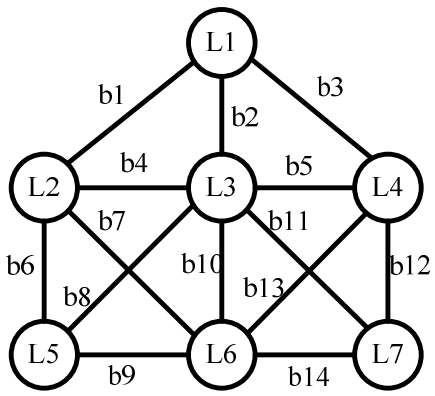
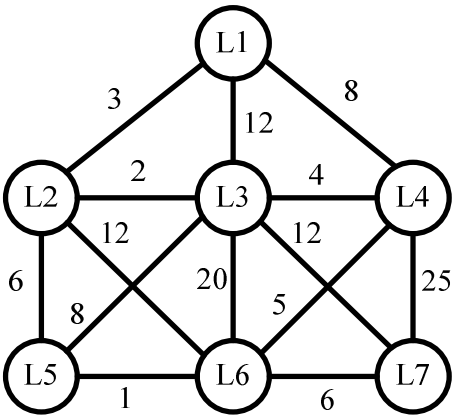


Figure 49. A logistics network

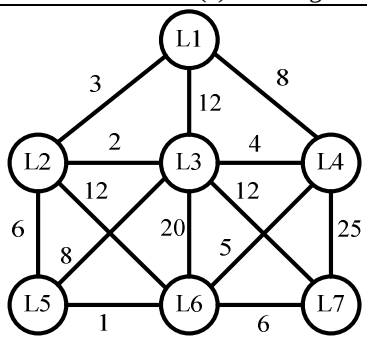
In this use case, each node is a location. Each edge is the logistic path from a location to another location. There is one parameter for each edge which is the benefit of the edge. The benefit can be calculated by a function.

(3) Algorithm for the K-connectivity Budget Cut Problem

Figure 50 shows an example of the 3-connectivity budget cut problem. In this example, we hope to remove 3 edges from the original graph G, and the reduced graph should preserve 3-connectivity.

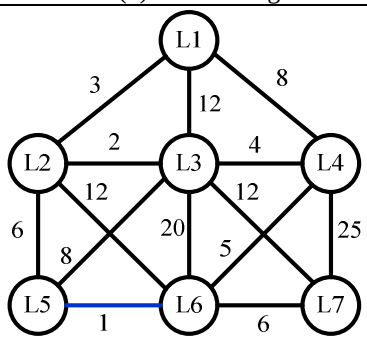


(a) The original graph



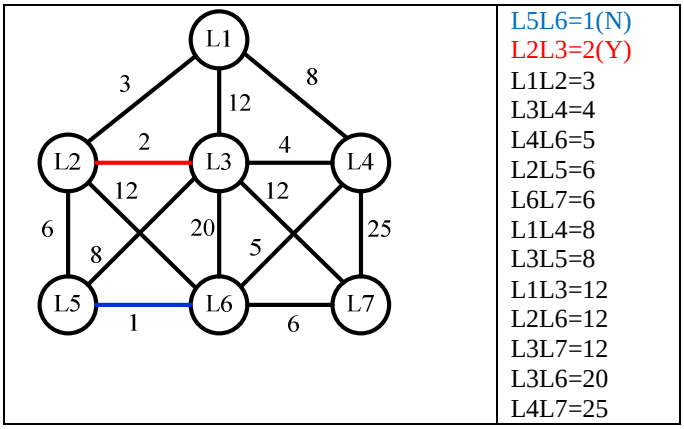
L5L6=1
L2L3=2
L1L2=3
L3L4=4
L4L6=5
L2L5=6
L6L7=6
L1L4=8
L3L5=8
L1L3=12
L2L6=12
L3L7=12
L3L6=20
L4L7=25

(b) Sort the edges in ascending order.

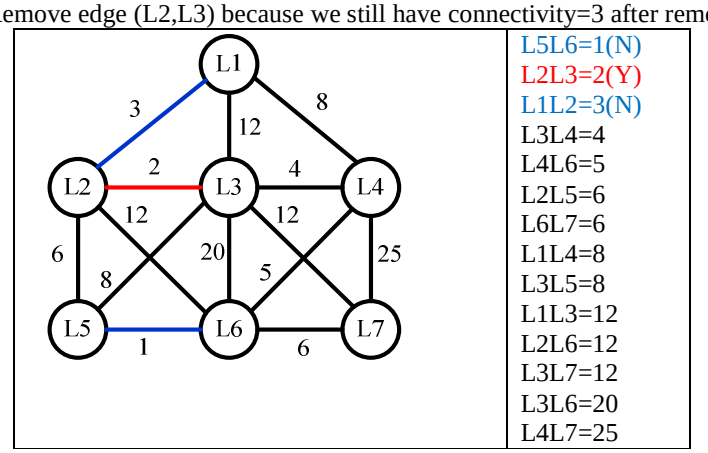


L5L6=1(N)
L2L3=2
L1L2=3
L3L4=4
L4L6=5
L2L5=6
L6L7=6
L1L4=8
L3L5=8
L1L3=12
L2L6=12
L3L7=12
L3L6=20
L4L7=25

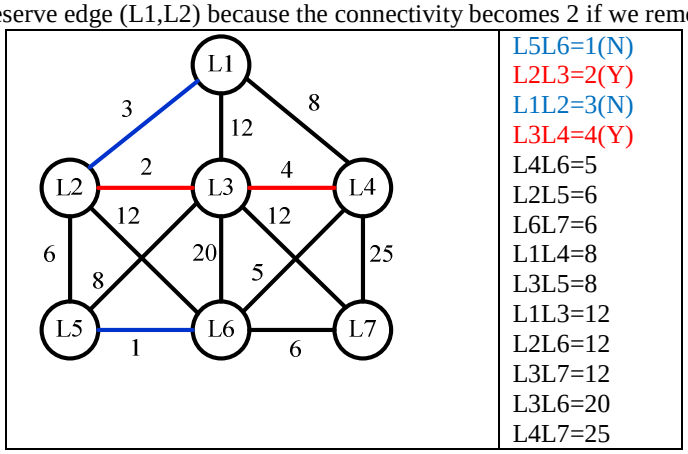
(c) Reserve edge (L5,L6) because the connectivity becomes 2 if we remove it.



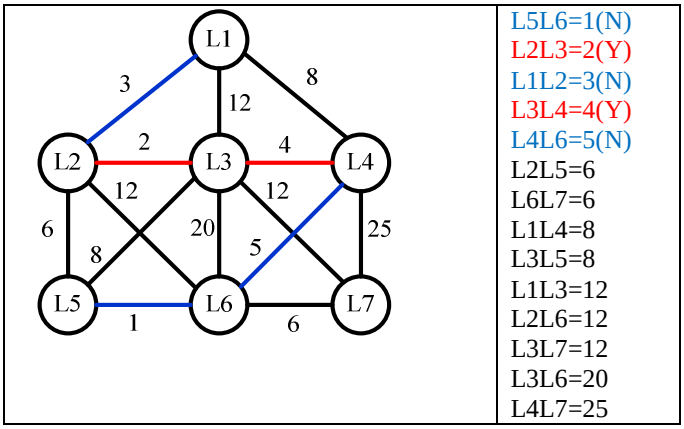
(d) Remove edge (L2,L3) because we still have connectivity=3 after removal.



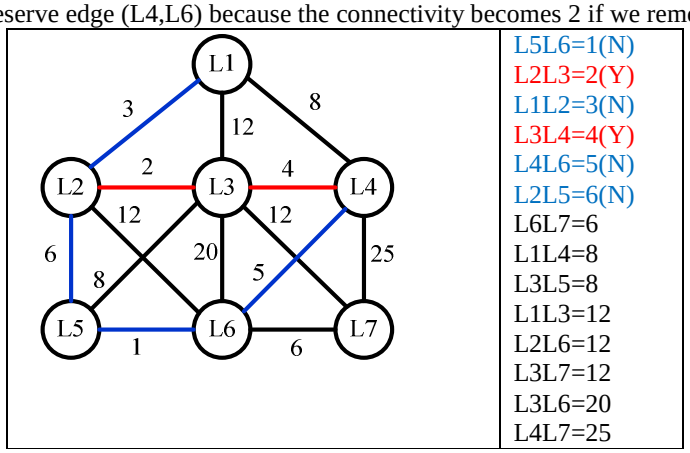
(e) Reserve edge (L1,L2) because the connectivity becomes 2 if we remove it.



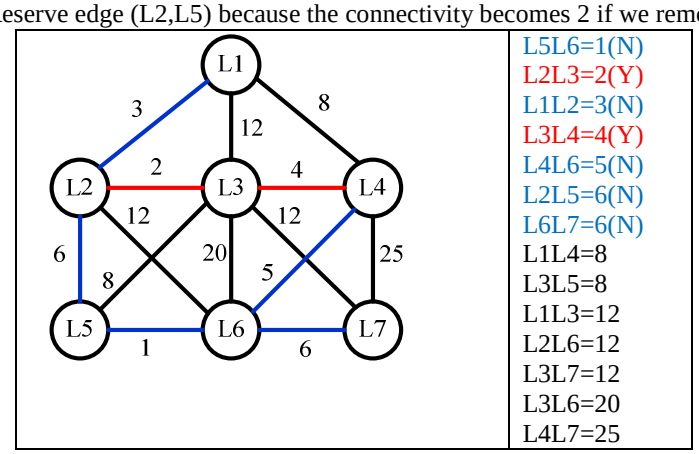
(f) Remove edge (L3,L4) because we still have connectivity=3 after removal.



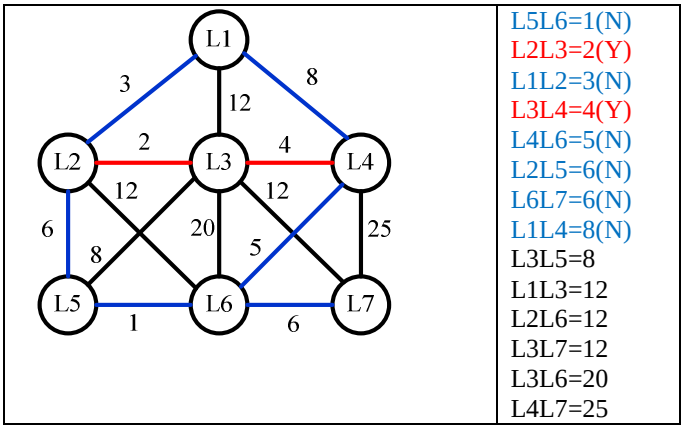
(g) Reserve edge (L4,L6) because the connectivity becomes 2 if we remove it.



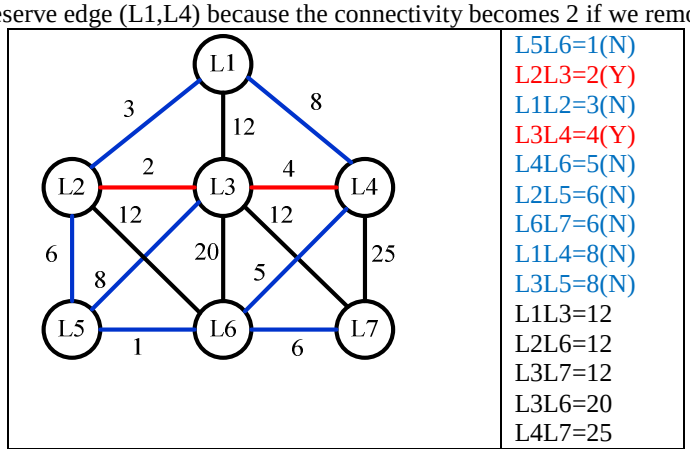
(h) Reserve edge (L2,L5) because the connectivity becomes 2 if we remove it.



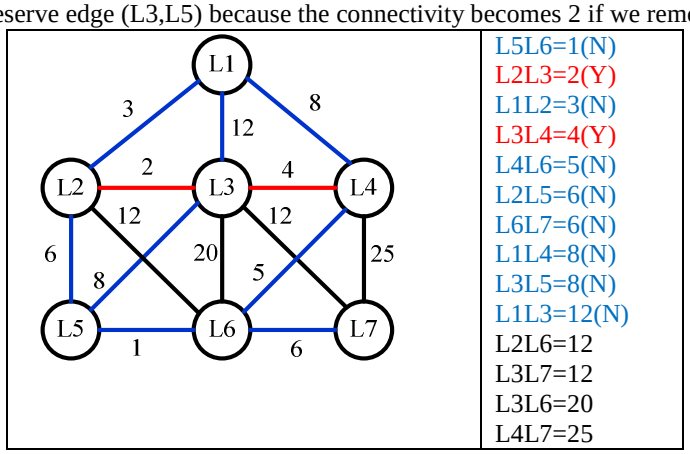
(i) Reserve edge (L6,L7) because the connectivity becomes 2 if we remove it.



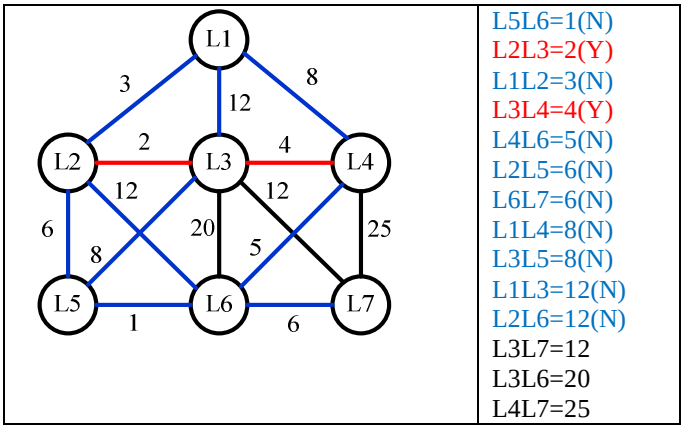
(j) Reserve edge (L1,L4) because the connectivity becomes 2 if we remove it.



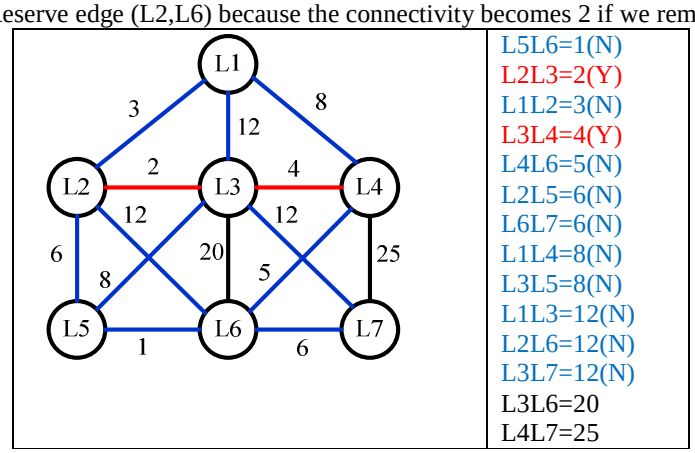
(k) Reserve edge (L3,L5) because the connectivity becomes 2 if we remove it.



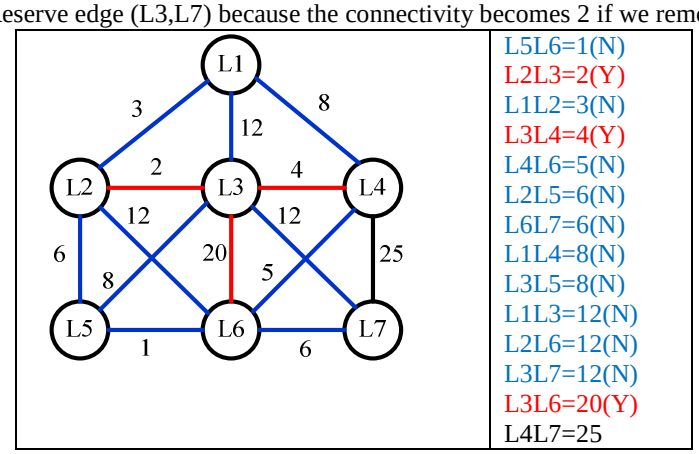
(l) Reserve edge (L1,L3) because the connectivity becomes 2 if we remove it.



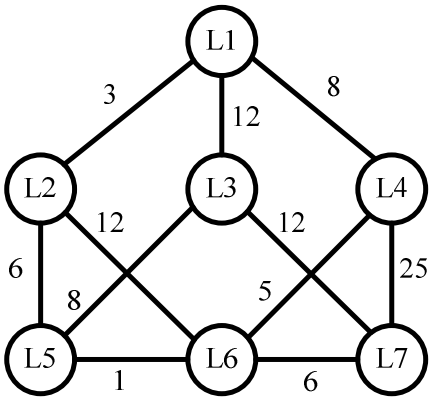
(m) Reserve edge (L2,L6) because the connectivity becomes 2 if we remove it.



(n) Reserve edge (L3,L7) because the connectivity becomes 2 if we remove it.



(o) Remove edge (L3,L6) because we still have connectivity=3 after removal.



(p) Compute the reduced graph G'
Figure 50. A 3-connectivity budget cut problem

Figure 50(a) shows the original graph G. In Figure 50(b), we sort the edges according in ascending order. In Figure 50(c), the edge (L5,L6) is reserved because if we remove the edge, the edge-connectivity between L5 and L6 becomes 2, which violates the connectivity constraint. In Figure 50(d), the edge (L2,L3) is removed because if we remove the edge, the edge-connectivity between L2 and L3 becomes 3, which does not violate the connectivity constraint so we can safely remove it. The following steps that reserve or remove an edge follow the same logic. In Figure 50(e), the edge (L1,L2) is reserved. In Figure 16(f), the edge (L3,L4) is removed. From Figure 50(g) to Figure 50(n), the edges (L4,L6), (L2,L5), (L6,L7), (L1,L4), (L3,L5), (L1,L3), (L2,L6), and (L4,L7) are reserved accordingly. In Figure 50(o), the edge (L3,L6) is removed. We stop the algorithm here because we have removed three edges. Finally, in Figure 50(p), we obtain the reduced graph G'.

The K-connectivity budget cut algorithm calculates the benefit of each edge, sorts the edges in increasing order. The algortihm then removes the minimum benefit edges if removing it does not make the connectivity of the graph smaller than *k*. The connectivity can be computed by any minimum s-t cut algorithm. The algorithm is described in the following.

GCGR-BC-KC algorithm (Find a reduced graph that maximizes the benefit after removing a given number of edges and preserves the k-connectivity constraint.)

Input: Graph G, a linear benefit function f, a number of edges to be removed n, a connectivity constraint k.
Output: a reduced graph G'

```
G' ← G
G_c ← modify G to assign 1 as the weight of each edge
For each edge e in G' do
   benefit(e) = f(profit(e),cost(e))
EndFor
E ← Sort the edges in increasing order of benefit
For i ← 1 to n do
   e ← get_min(E); //get the minimum edge
   If min_st_cut(G_c, e) > k
      Remove e from G' //Use the minimum s-t cut algorithm to calculate the minimum s-t cut of the incident
      nodes of e from G_c. If the number is greater than k, remove e
   EndIf
   Remove e from E
EndFor
Return G'
```

The time complexity of the GCGR-BC-KC algorithm is E*T(Mincut), where T(Mincut) is the time complexity of the Minimum s-t cut algorithm.

(4) Theorem and Proof

We will use Menger's Theorem to establish our theorem.

Menger's theorem [86] (edge version): Let G be an undirected graph. Let s and t be vertices of G. Then the maximum number of s, t-paths that are pairwise edge-disjoint equals the minimum number of edges that destroy all s, t-paths.

Theorem 32: The GCGR-BC-KC algorithm produces a k-connectivity subgraph *G'* which has the maximum benefit compared to all subgraphs with k-connectivity that have *n* edges fewer than the original graph.

Proof: We can prove this by induction. In the beginning, clearly n=0 holds because there is only one possibility of not removing any edge: the original graph itself. If $n=k$ holds, let G_k be the subgraph after removing k edges and it has the maximum benefit B_k, and G_k has k-connectivity. When $n=k+1$, assume that we have a different subgraph G_{k+1}' which removes a

different edge *e'* while our algorithm removes *e* to obtain G_{k+1}. By Menger's Theorem, our algorithm calculates the number of the edge-disjoint paths between the incident nodes of e by computing the minimum s-t cut. Removing the edge *e* will make the number of the edge-disjoint paths between the incident nodes decreased by 1. Since we only remove *e* if the number of the paths is greater than *k*, we still have at least *k* edge-disjoint paths after removing *e*, and the k-connectivity of the graph G_{k+1} is preserved. Since the benefit is a linear function *f* of profit and cost, $B_{k+1}=B_k-f(profit(e), cost(e))$, and $B_{k+1}'=B_k-f(profit(e'),cost(e'))$. Comparing *e* and *e'* we have 3 cases:

1. $f(profit(e),cost(e))> f(profit(e'),cost(e'))$. By the algorithm, *e* has the minimum benefit among all the edges that do not destroy the k-connectivity of the graph. As a result, removing *e'* makes the number of edge-disjoint paths between the incident nodes of *e'* smaller than *k*, so G_{k+1}' does not have k-connectivity.

2. $f(profit(e),cost(e))= f(profit(e'),cost(e'))$. In this case, $B_{k+1}= B_{k+1}'$

3. $f(profit(e),cost(e))< f(profit(e'),cost(e'))$. In this case, $B_{k+1}>B_{k+1}'$

Concluding the 3 cases, we know that compared to any other subgraph, either G_{k+1}' does not have k-connectivity, or $B_{k+1}$is greater than or equal to B_{k+1}', so G_{k+1} is a subgraph that has k-connectivity with the maximum benefit. ■

4.6. Synthesizing LGRP Algorithms

Some applications may need to solve complex graph problems which include more than one constraint. Developing every lossy graph reduction algorithm for all such complex graph problems is expensive, as the number of graph problems grows exponentially with the number of constraints. Therefore, it is desired to synthesize the existing lossy graph reduction algorithms to generate a lossy reduction graph for a new, complex integrated graph problem.

4.6.1. Stages in LGRP Algorithms

The algorithms discussed can be separated into three stages: pre-processing, the main algorithm, and post-processing, where pre-processing and post-processing are optional.

(1) Pre-processing

We define the pre-processing stage as the stage to remove all the edges that violate a constraint. That is, given the original graph G=(V,E), the pre-processing stage is to generate a subgraph $G_{pre}=G/E_{pre}$, where E_{pre} is the set of the edges that violate the condition, so that for any query q, $q(G)=q(G_{pre})$.

An example algorithm that has the pre-processing stage is 1+ε LGR-WC-MST. It first removes all the edges whose weight is greater than the weight constraint K; it then applies the 1+ε LGR-MST algorithm. The preprocessing stage removes all the edges that never appears in WC-MST as those edges violate the weight constraint. Therefore, any WC-MST queries on the graph after preprocessing obtain the same results as those can be obtained from the original graph.

(2) The main algorithm

The main algorithm applies the basic graph reduction method, which is usually an algorithm focused on a single objective without conditions. Examples are 1+ε LGR-SP, 2-degree-nodes-cut LGR-SP, and 2-degree-node reduction LGR-SP. These are three basic lossy reduction algorithms and other algorithms are developed based on them by adding the pre-processing and/or post-processing stages.

(3) Post-processing

The post-processing stage is the stage after the main algorithm; it may add some edges E_{post} to restore the information needed to solve problems with conditions, which is normally computed

from the original edges E, and if needed, with some modifications. The graph resulted from the post-processing stage may be used to answer the queries of interest with some possible errors.

An example algorithm that has the post-processing stage is 2-degree-nodes reduction LGR-DC-MST. It first applies the 2-degree-nodes reduction LGR-MST algorithm to the original graph, and then adds some edges that can provide alternative paths bypassing those nodes whose degree is more than the degree constraint. In the algorithm, the post-processing stage applies the k-merge algorithm to the alternative paths, and adds the merged paths back to the graph.

4.6.2. Algorithm for Reduction Algorithm Synthesis

If a graph problem has multiple conditions (objectives), and the reduction algorithms for different objectives satisfy the definition of the three stages: pre-processing, main algorithm, and post-processing, we are able to combine them in these three stages.

If we are only combining two algorithms, where one has the pre-processing stage, one has the post processing stage, and the main algorithm is the same, then we can easily combine them by merging them in the order of the three stages. An example is to combine 1+ε DC-MST and 1+ε WC-MST, the combined 1+ε LGR-WC-DC-MST algorithm can be described as follows:

1+εLGR-WC-DC-MST algorithm(Synthesize 1+ε LGR-WC-MST and 1+ε LGR-DC-MST)

1. Initialize the original graph G.
2. Apply step 2 in 1+ε LGR-WC-MST to G and obtain G_{pre}
3. $G_m \leftarrow$ 1+ε LGR-MST(G_{pre})
4. Apply steps 3-14 in 1+ε LGR-DC-MST to G_m and obtain G_{post}
5. **Return** G_{post}

If there are multiple algorithms for different constraints on the same type of problems, we can combine them if they use the same main algorithm. If they use different lossy main algorithms, since different lossy main algorithms will cause different kinds of errors, the interaction between

them will cause the combined error unbounded and unpredictable. Therefore, we only discuss the synthesis on pre-processing and post-processing stages.

(1) Pre-processing

Let E_{pre_1}, E_{pre_2}, $\cdots$, E_{pre_N} be the edge set that violate condition 1 to condition N, then the combined G_{pre} is G / ($E_{pre\_1} \cup E_{pre\_2} \cup \cdots \cup E_{pre\_N}$).

The Pre-processing synthesis algorithm can be described as follows.

Pre-processing synthesis algorithm (Synthesize the pre-processing stages of multiple reduction algorithms)

Input: Graph G, set of edges obtained in the pre-processing stage for different algorithms E_{pre_1}, E_{pre_2}, $\cdots$, E_{pre_N}.
Output: synthesized graph G_{pre}

1. $G_{pre} \leftarrow G$
2. $E_{pre} \leftarrow \emptyset$
3. **For** $i \leftarrow 0$ to N **do**
4. $E_{pre} \leftarrow E_{pre} \cup E_{pre\_i}$
5. **EndFor**
6. Remove E_{pre} from G_{pre}
7. **Return** G_{pre}

The same algorithm has already been proven by Theorem 7.

(2) Post-Processing

Let E_{post_1}, E_{post_2}, $\cdots$, E_{post_N} be the edge set that is added back in the post-processing stage of each algorithm, then the combined E_{post} will be $E_{post\_1} \cap E_{post\_2} \cap \cdots \cap E_{post\_N}$.

The Post-processing synthesis algorithm is described in the following.

Post-processing synthesis algorithm (Synthesize the Post-processing stage from multiple reduction algorithms)

Input: Graph G_m, edges from the post-processing stage for different algorithms E_{post_1}, E_{post_2}, $\cdots$, E_{post_N}.
Output: synthesized graph G_{post}

1. $G_{post} \leftarrow G_m$
2. $E_{post} \leftarrow \emptyset$
3. **For** $i \leftarrow 0$ to N **do**
4. $E_{post} \leftarrow E_{post} \cap E_{post\_i}$
5. **EndFor**
6. Insert E_{post} to G_{post}
7. **Return** G_{post}

The same algorithm has already been proven by Theorem 9.

Chapter 5 Experiments and Discussion

To evaluate the efficiency of the graph reduction algorithms, we experiment some algorithms with public data.

5.1. Experiment Environment

Database	Neo4j Community Edition: 3.0.6
Language	Java (jdk-8u02)
IDE	Eclipse (neon)

5.2. Evaluation Function

In our reduction algorithms, we only remove edges. Therefore, we can define the reduction ratio as follows.

$$\text{Reduction Ratio} = 1 - \frac{Num\ of\ Edges\ after\ Reduction}{Num\ of\ Edges\ before\ Reduction}$$

5.3. Experiment on Graph Data Reduction

5.3.1. Experiment with Map Data

We experiment the following graph reduction algorithms:

- GR-SP (Graph Reduction for Shortest Path)
- GR-MST-WC (Graph Reduction for Minimum Spanning Tree with Weight Constraint)
- GR-MST-DC (Graph Reduction for Minimum Spanning Tree with Degree Constraint)

(1) GR-SP

Table 1 shows the result of applying GR-SP on some map data. When we apply the algorithm to a public map dataset (http://www.dis.uniroma1.it/challenge9/download.shtml), the reduction

ratio is 0. For map data, it is expected that there is no other path between two nodes whose length is shorter than that of the edge connecting the two nodes.

Table 1. Results of map data

Map Data	Num of Edges Before Reduction	Num of Edges After Reduction	Reduction Ratio
New York	11,706	11,706	0%
San Francisco	13,763	13,763	0%
Colorado	11,552	11,552	0%

(2) GR-MST-WC

When we apply the algorithm with different values of *k* (weight constraint) to the public map data, we obtain the reduction ratios as shown in Figure 51. The reduction rate drops when *k* increases. It is expected because fewer edges violate the constraint when *k* increases. However the results show that the relationship is not linear, and the reduction rate becomes flatten when *k* is high. A possible reason is that the weight distribution is dense among small edges, and it becomes sparse among larger edges.

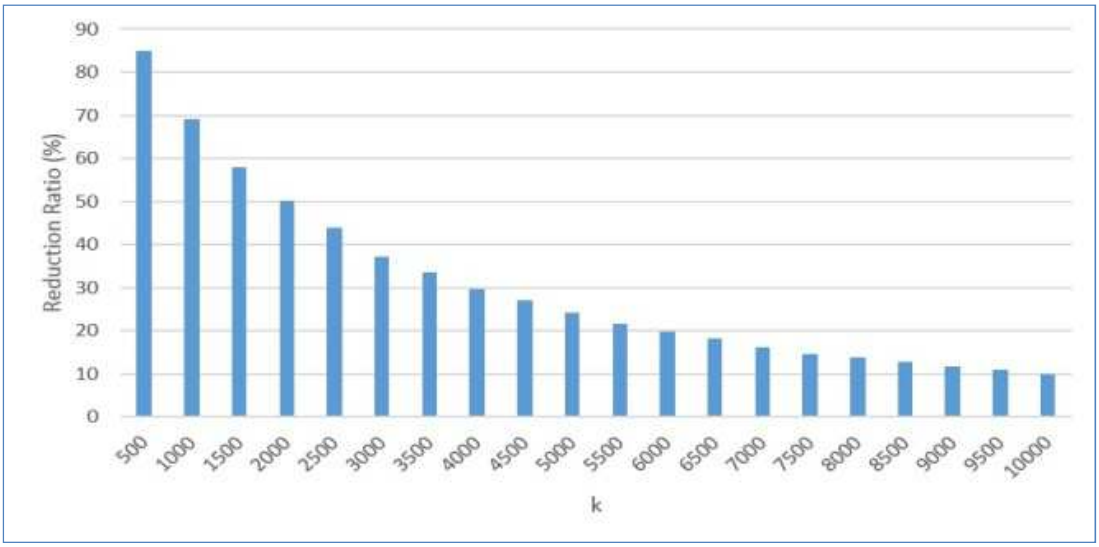


Figure 51. GR-MST-WC

(3) GR-MST-DC

When we apply the algorithm to the public map data, we obtain 0% reduction ratio. This is because GR-MST-DC depends on GR-SP, and GR-SP cannot reduce geometrical data such as map data.

5.3.2. Experiment with Social Network Data

As for social network data (the details are in table 2), we first download data from http://snap.stanford.edu/data/index.html#communities

Table 2. Experiment data set for social network

Name	Type	Nodes	Edges	Description
ego-Facebook	Undirected	4,039	88,234	Social circles from Facebook (anonymized)
ego-Gplus	Directed	107,614	13,673,453	Social circles from Google+
ego-Twitter	Directed	81,306	1,768,149	Social circles from Twitter
soc-Epinions1	Directed	75,879	508,837	Who-trusts-whom network of Epinions.com
soc-LiveJournal1	Directed	4,847,571	68,993,773	LiveJournal online social network
soc-Pokec	Directed	1,632,803	30,622,564	Pokec online social network
soc-Slashdot0811	Directed	77,360	905,468	Slashdot social network from November 2008
soc-Slashdot0922	Directed	82,168	948,464	Slashdot social network from February 2009
wiki-Vote	Directed	7,115	103,689	Wikipedia who-votes-on-whom network
wiki-RfA	Directed, Signed	10,835	159,388	Wikipedia Requests for Adminship (with text)

Next, we add a random value between 1 and 1000 as the weight of an edge since the datasets do not have weights on the edges. The weights can be considered as a contacting or data transferring cost between two people.

(1) GR-SP

When we apply the algorithm to the social network data with more than 10,000 nodes, we obtain more than 75% of reduction ratios as shown in Figure 52.

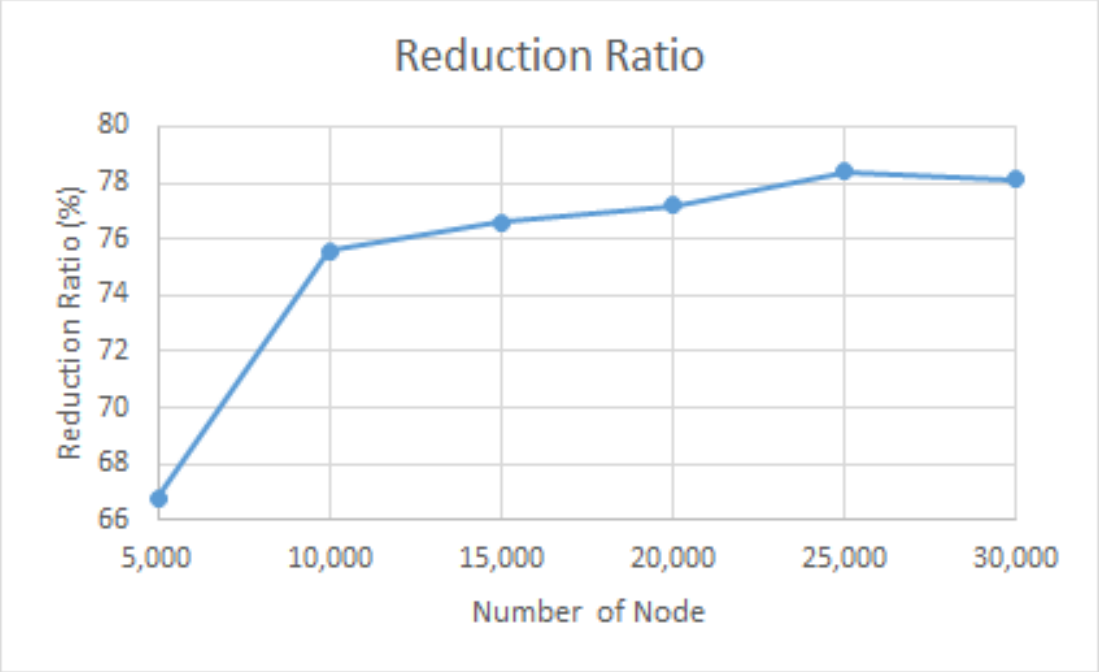


Figure 52. GR-SP with Social Network Data

5.3.3. Discussion

Comparing the results of graph reduction on map data and social network data, we can find a big difference in the reduction ratio. The reason is that the data structures between them are very different.

Figure 53 shows an example graph that models map data, we can see that the edges only connects the closest nodes, and most nodes have a small degree. As the map data are geometric, the edges are already the shortest paths between the nodes and there is almost nothing our algorithms can reduce.

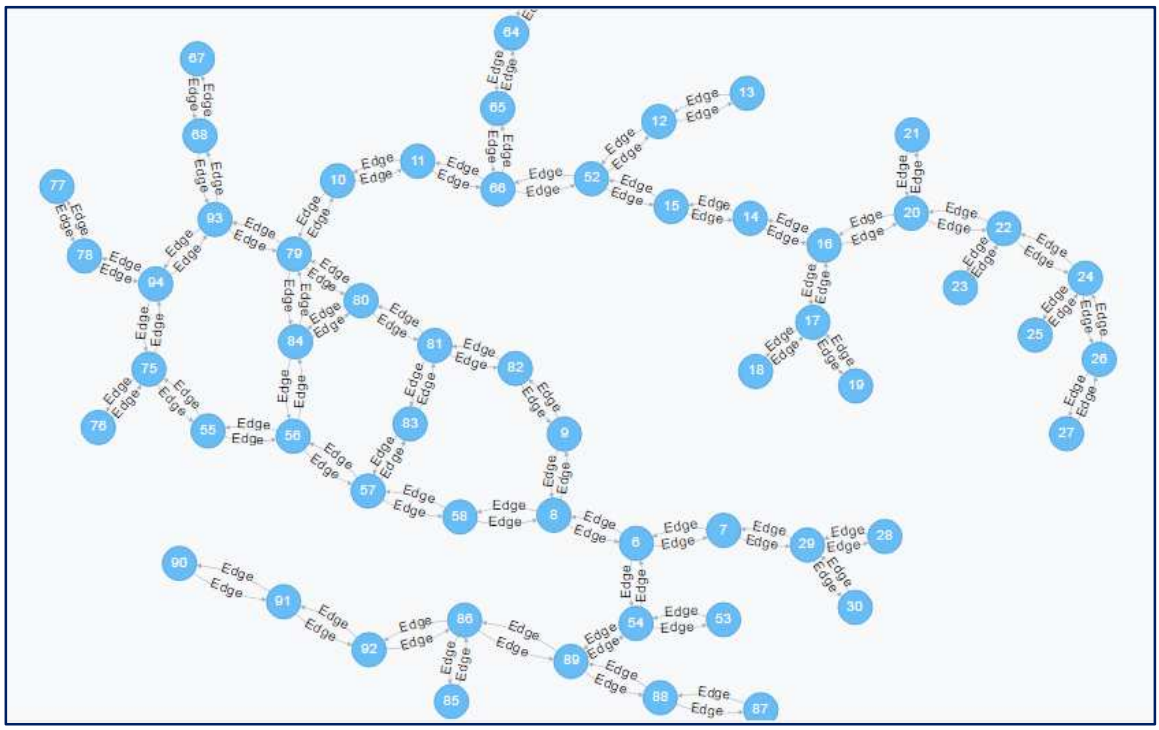


Figure 53. Map data

Figure 54 shows an example social network in which one node may be connected to many other nodes, and the distance between them is not geometric. Therefore our reduction algorithms are able to provide a good reduction ratio. Table 3 summarizes the differences between the two different types of graphs.

Table 3. Summary of two different types of data

	Map Data	Social Network Data
Reduction Ratio	Low (0~0.008)	High (66.8~78.4)
Data Type	Geometric	Not Geometric
Average Degree	Low (2.75)	High (7.06)

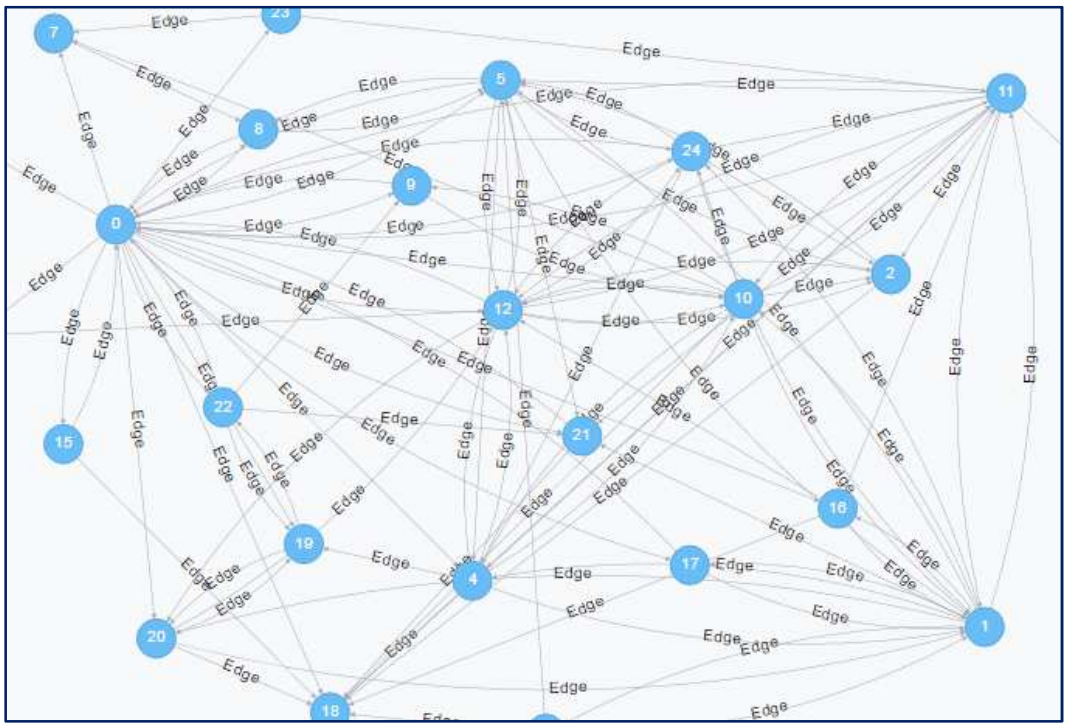


Figure 54. Social network data

5.4. Experiment on Lossy Graph Data Reduction

5.4.1. Experiment with Map Data

We apply the 1+ε LGR-SP algorithm on the same map data used in section 5.3.1. We first set ε to be 0.2, and experiment with different data sizes. Figure 55 shows the reduction rate is 0.5% ~ 0.7% on the map data. Comparing to the result of GR-SP, it is not surprised that there is not much improvement due to the properties of the map data discussed in 5.3.3.

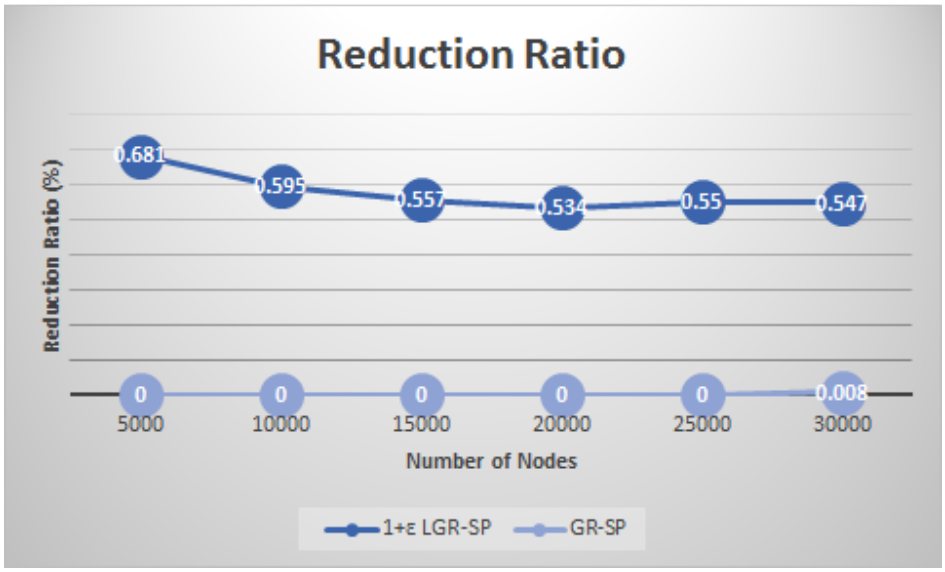


Figure 55. 1+ε LGR-SP on Map Data

Then we experiment different ε from 0.1 to 0.5 on 10,000 nodes. Figure 56 shows the results. When ε is 0.5, the reduction rate becomes 3.3%. It seems that reduction rate increases faster when ε is high. It is possibly due to the property of the map data. When ε is small, only the edges that have an alternative path which is slightly longer than the edge are removed, but it does not happen often in a real map: a road usually will not be built if there is another route with a similar distance.

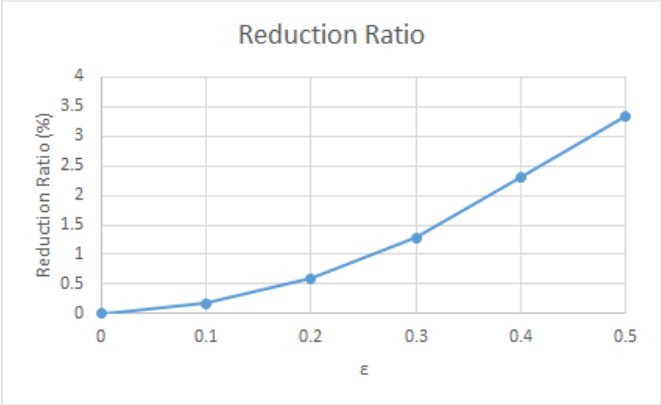


Figure 56. Different ε on map data (10,000 nodes)

5.4.2. Experiment with Social Network Data

We apply the 1+ε LGR-SP algorithm (ε = 0.2) on the same social network data used in Section 5.3.2. Figure 57 shows the results. Since the difference between 5,000 and 10,000 is high, we do more detailed experiments in this area. Table 4 shows the detailed results and explains the reason why the reduction rate is deviated between 1,000 and 10,000 nodes. This is due to the change of the average degree in the data. When the node number is 1,000, the average degree is 2.1, so the reduction rate is low. While with 10,000 nodes, the average degree is 7.1, so we obtain a higher reduction rate. Comparing to the results of GR-SP, the reduction rate of 1+ε LGR-SP is around 3% more, which is 15% improvement (i.e., the reduced data size is from 20% to 23% of the original data.)

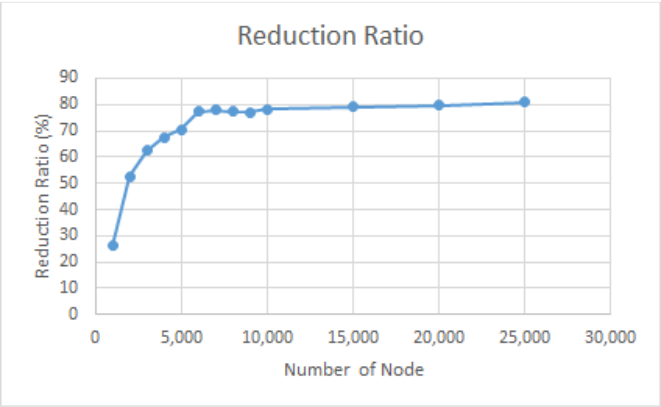


Figure 57. 1+ε LGR-SP on social network data (ε = 0.2)

Table 4. 1+ε LGR-SP on social network data (ε = 0.2)

Nodes	Edges	Edges Deleted	Reduction Rate(%)
1,000	2,150	560	26.1
2,000	6,531	3,433	52.6
3,000	11,793	7,373	62.5
4,000	18,867	12,752	67.6
5,000	27,825	19,604	70.5
6,000	42,965	33,229	77.3
7,000	51,281	39,910	77.8
8,000	55,891	43,154	77.2
9,000	60,432	46,474	76.9
10,000	71,234	55,722	78.2
15,000	115,128	91,128	79.2
20,000	164,180	131,025	79.8
25,000	227,071	183,677	80.9

Then we experiment different values of ε from 0.1 to 0.5 on 10,000 nodes. Figure 58 shows the results. The reduction rate is increased from 75.7% to 80.4% when we increase ε.

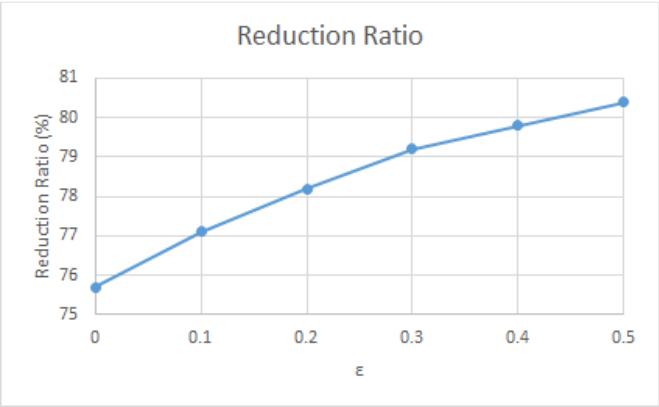


Figure 58. Different ε on social network data (10,000 nodes)

5.4.3. Verification

We verify the actual error rate when we apply the 1+ε LGR-SP algorithm. We randomly sample 100,000 pairs of nodes, and find the shortest paths between each pair in the original graph. Then we find the shortest paths for the same sampled pairs in the reduced graph, and compare them. Let the distance of the shortest path between a pair of nodes u and v in the original graph be $\delta_G(u,v)$, and the distance of the shortest path in the reduced graph be $\delta_{G'}(u,v)$. The error rate can be defined as $\frac{\delta_G(u,v)-\delta_{G'}(u,v)}{\delta_G(u,v)}$. Figure 59 shows the error ratios for different values of ε in the map data, and Figure 60 shows the error ratios for different values of ε in the social network data. The maximum error rate in the real data is smaller than ε, which means our algorithm is correct and actually sacrifices less (the actual error is smaller than the error bound ε) to achieve a higher reduction rate.

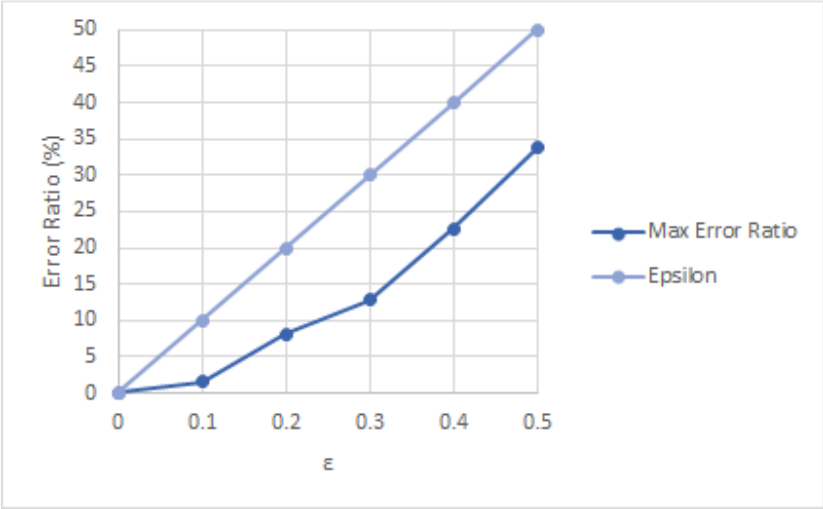


Figure 59. Actual error rates for different error bounds ε in map data

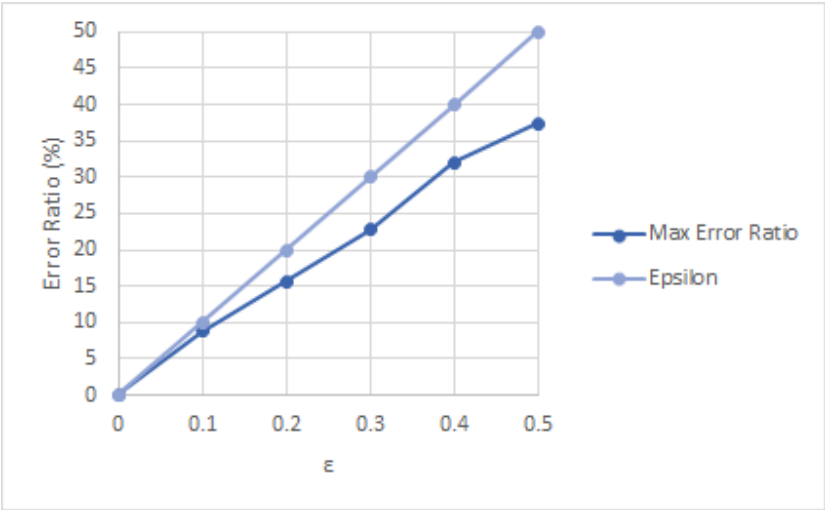


Figure 60. Actual error rates for different error bounds ε in social network data

Chapter 6 Conclusions and Future Work

In this thesis we address query-based graph data reduction, which reduces a graph by preserving only the information relevant to queries of interest to an application. Our method provides a semantic way to reduce the size of graph data, while other syntactical graph data compression techniques can still be performed on the reduced graph generated by our approach, which means our method can provide further reduction on top of other graph compression approaches.

Lossy graph data reduction may lose some information in order to improve the reduction ratio in some special applications. In order to solve the lossy reduction problem for graph data, we describe several types of lossy graph data reduction algorithms.

We discuss different types of problems and their variations, including the shortest path, MST, reachability, budget cut, shortest path budget cut, and the k-connectivity budget cut problems, and describe a set of graph reduction algorithms and lossy graph reduction algorithms designed for these problems. Incremental maintenance is discussed so when the original graph is modified, re-processing of the whole graph is not needed. We also introduce a synthesis method to combine existing graph reduction algorithms to generate a reduced graph for a complex graph problem that includes more than one constraint. The correctness of the graph reduction algorithms is proved theoretically and the error bounds of the lossy graph reduction algorithms are proved also.

We conduct experiments to compare the reduction rates of our algorithms on different sizes and different types of data. The results show that our algorithms are more suitable for non-geometric data such as social network data.

In the future, we plan to extend the work along the follow directions:

(1) Study the parallel processing algorithms for graph data reduction, as reducing very large graph data is time consuming.

(2) Study query-based graph data reduction and lossy graph reduction for different and possibly more complex problems, e.g., graph clustering.

(3) Apply similar techniques to query processing, which means developing different query processing methods to different types of queries.

Bibliography

[1] R. Angles and C. Gutierrez. Survey of graph database models. ACM Computing Surveys (CSUR), 40(1):1–39, 2008.

[2] G. M. Kuper and M. Y. Vardi. A new approach to database logic. In Proceedings of the 3th Symposium on Principles of Database Systems (PODS). ACM Press, 86–96, 1984.

[3] H. S. Kunii. DBMS with graph data model for knowledge handling. In Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: Today and Tomorrow. IEEE Computer Society Press, 138–142, 1987.

[4] R. H. Güting. GraphDB: modeling and querying graphs in databases. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB). Morgan Kaufmann, 297–30, 1994.

[5] M. Levene and G. Loizou. A graph-based data model and its ramifications. IEEE Trans. Knowl. Data Eng. 7, 5, 809–823, 1995.

[6] J. Paredaens and B. Kuijpers. Data models and query languages for spatial databases. Data & Knowledge Engineering (DKE) 25, 1–2, 29–53, 1998.

[7] M. Gyssens, J. Paredaens, and D. V. Gucht. A graph-oriented object model for database end-user interfaces. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data. ACM Press, 24–33, 1990.

[8] B. Amann and M. scholl. Gram: A Graph Data Model and Query Language. In European Conference on Hypertext Technology (ECHT). ACM, 201–211, 1992.

[9] J. Hidders and J. Paredaens. GOAL, A graph-based object and association language. Advances in Database Systems: Implementations and Applications, CISM, 247–265, 1993.

[10] M. Graves, E. R. Bergeman and C. B. Lawrence. A graph-theoretic data model for genome mapping databases. In Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS). IEEE Computer Society, 32, 1995.

[11] J. Hidders. Typing graph-manipulation operations. In Proceedings of the 9th International Conference on Database Theory (ICDT). Springer-Verlag, 394–409, 2002.

[12] G. Klyne and J. carroll. Resource description framework (RDF) concepts and abstract syntax. 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[13] The World Wide Web Consortium. http://www.w3.org/

[14] W3C PICS The Platform for Content Selection home page. http://www.w3.org/PICS

[15] W3C PICS-NG The Platform for Content Selection - Next Generation home page. http://www.w3.org/PICS/NG

[16] The W3C XML Extensible Markup Language Working Group home page. http://www.w3.org/XML/

[17] Andrew Layman, Jean Paoli, Steve De Rose, and Henry S. Thompson. XML-Data. June 20, 1997. http://www.microsoft.com/standards/xml/xmldata.htm

[18] R.V. Guha and Tim Bray. Meta content framework using XML. June 6, 1997. http://www.w3.org/TR/NOTE-MCF-XML/

[19] The Dublin Core home page. http://purl.oclc.org/metadata/dublin_core

[20] C. Lagoze, C. A. Lynch, and R. Daniel, Jr. The warwick framework: a container architecture for aggregating sets of metadata. June 21, 1996.

http://cs-tr.cs.cornell.edu:80/Dienst/UI/2.0/Describe/ncstrl.cornell/TR96-1593

[21] Property Graph Model. https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model

[22] E. F. Codd. A relational model of data for large shared data banks. Commun. ACM 13, 6, 377–387, 1970.

[23] W. Kim. Object-oriented databases: definition and research directions. IEEE Trans. Knowl. Data Eng. 2, 3, 327–341, 1990.

[24] LEVENE, M. Levene and A. Poulovassilis. The Hypernode model and its associated query language. In Proceedings of the 5th Jerusalem Conference on Information technology. IEEE Computer Society Press, 520–530, 1990.

[25] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. V. Den Bussche. Concepts for graph-oriented object manipulation. In Proceedings of the 3rd International Conference on Extending Database Technology (EDBT). LNCS, vol. 580, 21–38, 1992.

[26] Olivier Danvy and Ian Zerny. Three Syntactic Theories for Combinatory Graph Reduction. In María Alpuente, editor, Logic Based Program Synthesis and Transformation, 20th International Symposium, LOPSTR 2010, revised selected papers. LNCS, 6564, Springer, July 2010, 1–20. Invited talk.

[27] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. Journal of Functional Programming, 7(3):265–301, 1997.

[28] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. Journal of Functional Programming, 8(3):275–317, 1998.

[29] O. Danvy, K. Millikin, J. Munk, and I. Zerny. Defunctionalized interpreters for callby-need evaluation. In M. Blume and G. Vidal, editors, Functional and Logic Programming, 10th International Symposium, FLOPS 2010, number 6009 in Lecture Notes in Computer Science, pages 240–256, Sendai, Japan, Apr. 2010. Springer.

[30] O. Danvy. Defunctionalized interpreters for programming languages. In P. Thiemann,editor, Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08), SIGPLAN Notices, Vol. 43, No. 9, pages 131–142, Victoria, British Columbia, Sept. 2008. ACM Press. Invited talk.

[31] O. Danvy. From reduction-based to reduction-free normalization. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, Advanced Functional Programming, Sixth International School, number 5382 in Lecture Notes in Computer Science, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer. Lecture notes including 70+ exercises.

[32] T. Feder, R. Motwani. Clique partitions, graph compression and speeding-up algorithms. Proceedings of the twenty-third annual ACM symposium on Theory of computing. ACM, 1991. p. 123-133.

[33] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. Proceedings of the 2008 International Conference on Web Search and Data Mining. ACM, 2008.

[34] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. Proceedings of the 13th international conference on World Wide Web. ACM, 2004.

[35] Hossein Maserrat and Jian Pei. Neighbor query friendly compression of social networks. Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2010.

[36] J.D. Fern´andez, M.A. Mart´ınez-Prieto, C. Guti´errez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange. Journal of Web Semantics, 19:22–41, 2013

[37] S. Alvarez-Garc´ıa, N. Brisaboa, J.D. Fern´andez, M.A. Mart´ın ´ ez-Prieto, and G. Navarro. Compressed Vertical Partitioning for Efficient RDF Management. Knowledge and Information Systems, 2014.

[38] A.K. Joshi, P. Hitzler, and G. Dong. Logical linked data compression. Extended Semantic Web Conference. Springer Berlin Heidelberg, 2013.

[39] S. Grimm and J.Wissmann. Elimination of redundancy in ontologies. Extended Semantic Web Conference. Springer Berlin Heidelberg, 2011.

[40] Jeff Z. Pan et al. Graph pattern based RDF data compression. In: Joint International Semantic Technology Conference. Springer International Publishing, 2014.

[41] Brisaboa N R, Ladra S, Navarro G. k2-Trees for compact Web graph representation. In: Proc. of the 16th Int'l Symp. on String Processing and Information Retrieval (SPIRE). Heidelberg: Springer-Verlag, 2009. 18−30.

[42] González R, Grabowski S, Mäkinen V, Navarro G. Practical implementation of rank and select queries. In: Proc. of the 4th Workshop on Efficient and Experimental Algorithms (WEA). CTI Press and Ellinika Grammata, 2005. 27−38.

[43] Brisaboa NR, Ladra S, Navarro G. Compact representation of Web graphs with extended functionality. Information Systems, 2014,39:152−174.

[44] Brisaboa NR, Ladra S, Navarro G. DACs: Bringing direct access to variable-length codes. Processing and Management:an International Journal, 2013,49(1):392−404.

[45] Larsson NJ, Moffat A. Off-Line dictionary-based compression. Proc. of the IEEE, 2000,8(11):1722−1732.

[46] Claude F, Navarro G. A Fast and compact Web graph representations. ACM Trans. on the Web (TWEB), 2010,4(4):1−31.

[47] Claude F, Navarro G. Extended compact Web graph representations. In: Algorithms and Applications. Heidelberg: Springer-Verlag, 2010. 77−91.

[48] He M, Munro JI. Succinct representations of dynamic strings. In: Proc. of the 17th Int'l Symp. on String Processing and Information Retrieval (SPIRE). Heidelberg: Springer-Verlag, 2010. 334−346.

[49] Yu zhang, yanbin liu. Survey on Succinct Representation of Graph Data.2014,25(9):1937−1952.

[50] Ziv J, Lempel A. Compression of individual sequences via variable-rate coding. IEEE Trans. on Information Theory, 1978,24(5):530−536.

[51] Claude F, Ladra S. Practical representations for Web and social graphs. In: Proc. of the 20th ACM Int'l Conf. on Information and Knowledge Management (CIKM). New York: ACM, 2011. 1185−1190.

[52] Asano Y, Miyawaki Y, Nishizeki T. Computing and Combinatorics. Heidelberg: Springer-Verlag, 2008. 1−11.

[53] Boldi P, Vigna S. The WebGraph Framework II: Codes for the World-Wide Web. In: Proc. of the Conf. on Data Compression (DCC). Piscataway: IEEE, 2004.

[54] Boldi P, Vigna S. The Webgraph Framework I: Compression techniques. In: Proc. of the 13th Int'l Conf. on World Wide Web (WWW). New York: ACM, 2004. 595−602.

[55] Buehrer G, Chellapilla K. A scalable pattern mining approach to Web graph compression with communities. In: Proc. of the 2008. Int'l Conf. on Web Search and Data Mining (WSDM). New York: ACM, 2008. 95−106.

[56] Hernández C, Navarro G. Compression of Web and social graphs supporting neighbor and community queries. In: Proc. of the 6th ACM Workshop on Social Network Mining and Analysis (SNAKDD). New York: ACM, 2011. 1−10.

[57] Chierichetti F, Kumar R, Lattanzi S, Mitzenmacher M, Panconesi A, Raghavan P. On compressing social networks. In: Proc. of the 15th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD). New York: ACM, 2009. 219−228.

[58] Apostolico A, Drovandi G. Graph compression by BFS. Algorithms, 2009,2(3):1031−1044.

[59] Maserrat H, Pei J. Neighbor query friendly compression of social networks. In: Proc. of the 16th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD). New York: ACM, 2010. 533−542.

[60] Fan W, Li J, Wang X, Wu Y. Query preserving graph compression. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD). New York: ACM, 2012. 157−168.

[61] Vineet Mehta, Feanil Patel, Yan Glina, Matthew Schmidt, Ben Miller and Nadya Bliss.Characterization of Traffic and Structure in the U.S.Airport Network.2012 Conference on Intelligent Data Understanding.Pages:124-129.

[62] Gomory, R. E.; Hu, T. C. (1961). Multi-terminal network flows. Journal of the Society for Industrial and Applied Mathematics. 9.

[63] Garey, Michael R.; Johnson, David S. (1979), Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, ISBN 0-7167-1045-5. A2.1: ND1, p. 206.

[64] Amria raju, Navadeep singh. A Review on data compression techniques. International Journal of Applied Research in Computer Science and Software Engineering (IJARCSSE) vol5 Jan 2015.

[65] A. Gupta and S. Verd´u. Nonlinear sparse-graph codes for lossy compression. IEEE Trans. Inf. Theory, vol. 55, 2009.

[66] C. Gioran and I. Kontoyiannis. Lossy compression in near-linear time via efficient random codebooks and databases. arXiv preprint arXiv:0904.3340, 2009.

[67] R. Venkataramanan, T. Sarkar, and S. Tatikonda. Lossy compression via sparse linear regression: Computationally efficient encoding and decoding. CoRR, vol. abs/1212.1707, 2012.

[68] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala: Congressional Samples for Approximate Answering of Group-By Queries. SIGMOD Conference 2000: 487-498.

[69] H.V. Jagadish, Nick Koudas, S. Muthukrishnan,Viswanath Poosala, Kenneth C. Sevcik, Torsten Suel:Optimal Histograms with Quality Guarantees. VLDB 1998:275-286.

[70] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In VLDB, pages 111–122, 2000.

[71] Z. Liu, J.X. Yu, H. Cheng. Approximate homogeneous graph summarization. Information and Media Technologies 7(1):32-43, 2012.

[72] A. Ingber, T. Courtade, and T. Weissman. Quadratic similarity queries on compressed data. Data Compression Conference, 2013.

[73] A. Ingber, T. Courtade, and T. Weissman. Compression for quadratic similarity queries. IEEE Transactions on Information Theory, 61(5), 2729-2747, 2015.

[74] W. Fan, X. Wang, and Y. Wu, Querying big graphs within bounded resources," in SIGMOD, 2014, pp. 301-312.

[75] W. Fan, F. Geerts, Y. Cao, T. Deng, and P. Lu.Querying big data by accessing small data,in PODS, 2015.

[76] Christopher R. Palmer, Christos Faloutsos: Density Biased Sampling: An Improved Method for Data Mining and Clustering. SIGMOD Conference 2000: 82-92.

[77] Christopher Jermaine, Edward Omiecinski: Lossy Reduction for Very High Dimensional Data. Data Engineering, 2002. Proceedings. 18th International Conference on. IEEE, 2002. p. 663-672.

[78] Jeffrey Xu Yu, Jiefeng Cheng; Graph Reachability Queries: A Survey; In:Managing and Mining Graph Data - Advances in Database Systems Volume 40,2010, pp 181-215.

[79] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: A scalable index for reachability queryies in very large graphs. The VLDB Journal, pages 509–534, 2012.

[80] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying.An independent permutation labeling approach. PVLDB, pages 1191–1202, 2014.

[81] F. Zhou, S. Malher, and H. Toivonen. Network simplification with minimal loss of connectivity. In Proc of ICDM, 2010.

[82] S. Navlakha, R. Rastogi, and N. Shrivastava, Graph summarization with bounded error. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD ’08. New York, NY, USA: ACM, 2008,pp. 419–432.

[83] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka. Compression of weighted graphs. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, KDD '11, pp 965-973, 2011.

[84] https://en.wikipedia.org/wiki/National_Highway_System_(United_States)#/media/File:National_Highway_System.jpg

[85] Skiścim, Christopher C.; Palocsay, Susan W. The complexity of minimum ratio spanning tree problems. Journal of global optimization, 2004, 30.4: 335-346.

[86] Menger, Karl. Zur allgemeinen Kurventheorie. Fund. Math. 10: 96–115. 1927.