The Hong Kong Polytechnic University

Department of Computing

Decision Support Queries on Graph Data: Answering Which-Pair Queries

by

Ming-Hay Luk

A thesis submitted in partial fulfillment of the requirements

for the degree of

Master of Philosophy

March 2011

CERTIFICATE OF ORIGINATLITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

.....

Ming-Hay Luk

March 2011

Abstract

Decision support systems are computer aided tools that support the decision making process of an organization. By analyzing enormous volumes of data, decision support systems will provide insightful information to its users. The success of a decision support system relies heavily on both the accuracy of the information that it can provide and the time it requires to provide the information. Conventionally, decision support systems operate using numeric data. The prevalence of graph data has prompted decision support research to begin researching techniques for supporting graph data. Thus far, only graph summarizing and graph pattern mining has been addressed. However, these techniques do not allow for changes in the underlying graph data.

When the underlying data is represented by graphs, what types of decision support queries will users ask? This work presents "which" queries, a class of decision support queries that are specific to graph data, and illustrate their potential applications. When an organization makes a decision resulting in a change to the underlying graph data, new edges may eventually be added to the organization's graph data model. The addition of a new edge to the graph will affect how the objects, modeled in the graph, relate to each other. Determining how the object relationships change is a crucial factor that the organization needs to consider. However, determining object relationships is an expensive operation. Furthermore, the solution space that an organization is considering can be considerably large when using graph data; thus, efficient algorithms to

evaluate the fundamental "which" queries are also presented.

This work will develop a decision support system that effectively and efficiently answers the fundamental "which" queries. First, a model for "which" queries will be developed. Then, algorithms for efficiently answering fundamental "which" queries will be presented, which have been experimentally shown to be orders of magnitude faster than basic solutions. The results of the experiments on five real graph data sets will be presented. Finally, discussion of how the proposed algorithms can be used to answer other types of "which" queries will be presented.

Acknowledgements

I would like to thank all of my friends and family for their love and support while I have been away from home. Thank you all for talking me through various things and keeping me entertained, among many countless other things.

I would like to thank my academic supervisor, Dr. Eric Lo, for his help and guidance during my studies. Not only has he shown me the ropes when it comes to academic research in the computer science field, but he has also introduced me to many incredible people and given me opportunities to dabble with various projects, which I would have otherwise been unable to do.

Next, I would like to thank Dr. Ken Yiu for allowing me to work closely with him on various research projects. Additionally, his help with optimizing code, double checking ideas, and comments has been invaluable to the completion of this work.

Lastly, the comments and critiques on this work that were received from various other parties is also high appreciated. Thanks to Dr. Raymond Wong, Dr. Hong Va Leong, Dr. Ben Kao, Duncan Yung, Andy Ho, et al. for their comments and critiques on various parts of this work.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Decision support systems are information systems that support the complex decision-making process faced by businesses or organizations. When making important decisions, a decision support system provides insightful suggestions to its users. Decision support systems give suggestions by analyzing large volumes of data. In the real world, data is often represented by graphs, and many decisions are reflected as eventual changes to the graph data.

For example, consider an express mail company's delivery network: a graph with vertices representing locations (e.g., cities, warehouses) and edges representing connections (e.g., flights between cities). Subject to rapidly changing business environments, the company may need to revise its resource allocation policy regularly—with additional resources, the company can reduce its overall/average delivery time (increasing its competitive advantage) by establishing a new connection between two indirectly connected locations. In the example above, the company must decide "*which two locations should be connected by*

the new connection?" (Q1). In this situation, a decision support system that gives insightful suggestions such as "among all the possible choices, the maximum reduction in overall delivery time (55%) is achieved with a new connection between X and Y," will be very helpful in aiding the decision-making process. After said decision is made operational, a new connection (edge) is added to the delivery network (graph) reflecting the impact of the decision.

Alternatively, during tough economic times, the express mail company may need to scale back on its existing services. When the company needs to scale back, they may want to only temporarily cease operating certain routes. Following the above example, suppose that the express mail company wishes to temporarily shut down certain flights within their delivery network. As such, the compay may ask: "*which flight, if cut, has the minimal impact on overall delivery time?*" (Q2). Because the company is temporarily cutting an existing flight in their delivery network, this type of decision is eventually reflected by an existing connection (edge) being removed from the delivery network (graph).

Many other domains also demand similar kinds of decision support queries. For example, in urban planning the road network is often modeled as a graph, where vertices are road intersections, and edges are the different stretches of road themselves. Roadwork is constantly being performed to fix old sections of roads in the network, add new stretches of road to improve coverage to different places, stretches of road are expanded or rerouted to relieve traffic congestion, etc. As we can see, urban planning is a highly dynamic domain because the road network is constantly changing to support its traffic volume. To address transit times between locations on the road network, we may ask "*which stretch of road (edge) should we expand such that average travel times can be maximally*

reduced?" (Q3). Many other types of decision making queries are possible within the realm of urban planning. As noted above, when maintenance roadwork needs to be performed, certain stretches of road may need to be closed off to traffic completely and detour routes need to be constructed; in which case, determining short detour routes would benefit commuters.

Similar to urban planning, network planning is another field that can represent its communication network as a graph. For communication networks, graph edges can represent physical linkages, such as fiber optic lines, between the vertices, such as signal repeaters or access hubs. Because network maintenance is critical, we may ask "*which optic fibre, if broken (e.g., due to a natural disaster), would have the largest negative impact on the network communcation throughput?*" (Q4). By asking the previous decision making query, network planners can determine critical links in their network and take the proper actions to ensure these critical links are well maintained and proper backup solutions are in place to avoid adverse effects should these critical links fail.

Finally, consider one last domain that can make use of decision support queries when their underlying data can be represented as a graph. Within the domain of manufacturing, the production workflow can be modeled as a graph (a flow-graph with vertices as workstations, edges as conveyor-belts, and edge weights as belt capacities). Management may want to ask "*which two indirectly linked workstations should be linked with a new conveyor-belt such that the production rate increases the most?*" (Q5).

Conventional decision support systems mostly focus on numeric data [6]. Furthermore, recent graph-related decision support system research still focuses

on summarizing, OLAP, or mining graph data (e.g., [8, 33, 7, 19, 24]) and has not advanced to supporting the aforementioned types of "which" queries (Q1–Q5) on graphs. Answering "which" queries on graphs is an interesting, yet challenging, research topic: we can see from the above examples that real world decision-making problems are diverse; thus, "which" queries are equally diverse. For example, "which" query Q1 is related to *edge addition* and *shortest path distances*; Q4, however, is related to *edge deletion* and *network flows*. Thus, it is necessary to identify the core elements formulating "which" queries and also devise corresponding evaluation algorithms. Regarding evaluation algorithms for "which" queries, the challenge is that the solution spaces of "which" queries can be very large—for Q1, any pair of indirectly connected locations can potentially be the answer. Because efficiency is crucial to a decision support system's success, as users want to receive timely suggestions, straightforward algorithms (e.g., exhaustive search) are highly inapplicable.

Graph query processing is one of the most important topics in our field. However, to the best of our knowledge, the issue of "which" queries on graphs remains unaddressed. This work makes the following contributions:

1. We discuss the essential elements composing "which" queries and also provide the formulation of a "which" query (Chapter 3).

2. We present new algorithms that can efficiently evaluate the most fundamental type of "which" queries—those related to shortest path distances (Chapters 4 and 5) because they are useful in many applications (e.g., Q1) and also because their solutions can serve as a very good foundation for solving other types of "which" queries (in other types of "which" queries,

such as Q5, we know that the shortest path distances are directly related to the maximum flow value; thus, the algorithms proposed in Chapter 4 can be used to solve "which" queries such as Q5. Furthermore, "which" queries like Q2 and Q4 can be solved using the techniques proposed in Chapter 5).

3. The efficiency of our proposed methods is extensively evaluated using five real graph data sets and our experiments show that our methods are orders of magnitude faster than straightforward solutions (Chapter 6).

4. A case study of how the "which" query processing techniques proposed can be applied to the Bay Area.

5. A discussion of how our proposed techniques can be useful for solving other types of "which" queries (Chapter 8).

In Chapter 2, we discuss work in various domains related to "which" queries. Chapter 9 concludes this work.

Query Number	Query Description
Q1	Which two locations should be connected by the new connection?
Q2	Which flight, if cut, has the minimal impact on overall delivery time?
Q3	Which stretch of road should we expand such that average travel times can be maximally reduced?
Q4	Which optic fibre, if broken, would have the largest negative impact on the network communcation throughput?
Q5	Which two indirectly linked workstations should be linked with a new conveyor-belt such that the production rate increases the most?

Table 1.1. Example "Which" Queries

Chapter 2

Related Work

Graph query processing is one of the most important topics in the field of computer science [12]. Recently, the database community started to research efficient methods of managing and querying vast volumes of extremely large graphs. Important research topics include graph query algebra and languages (e.g., [2]) and indices for graph query processing (e.g., [3, 4, 9, 11, 16, 17, 18, 25, 26, 27, 28, 34, 35]). Conventional decision support systems mostly focus on numeric data, and only recently has started to address graph data. The decision support systems research related to graphs, which mainly focus in summarizing (e.g., [24]), OLAP (e.g., [8, 33]), or mining graph data (e.g., [7, 19]), does not address how to answer "which" queries.

To the best of our knowledge, supporting "which" queries on graphs is a novel topic that has not been addressed in the database and data mining fields. Nonetheless, the broad applications of "which" queries can be related to spatial database research, operations research, and network topology maintenance.

2.1 Spatial Database Research

Spatial database research focuses on numerical data with spatial relationship information attached to the data. Recently, spatial database research has begun to address graph data with certain embedded spatial relationships. Although graph data is starting to be addressed, query types similar to "which" queries remain largely unaddressed.

From spatial database research, optimal-location queries [13, 15] are the most similar type of queries to "which" queries. Optimal-location queries are a class of spatial decision support queries where users look for the best location, l, for a new facility such that the greatest *benefit* is obtained. The notion of benefit in an optimal-location query is defined by the user, and varies depending on the application and the context in which the query is posed. For example, [15] considers the benefit of a location as the total weight of its reverse nearest neighbors (i.e., the total weight of objects that are closer to l than to any other data point in the dataset).

Optimal-location queries are helpful in finding ideal locations for a new shop to attract the largest number of customers. However, "which" queries (e.g., Q1–Q5) that work on graphs are much more diverse than optimal-location queries, which only ask "*where to add a new point?*" Optimal-location queries are restricted to queries that have an answer that is just a single point, whereas "which" queries support queries where the answer can be sets of points and other more complicated component sets.

2.2 Operations Research

From operations research, optimizations to an existing system are the principle focus. We can see that "which" queries can also suggest how an existing system can be modified to become more efficient (e.g., Q3). Other types of "which" queries, such as Q5, have not been addressed by the operations field, to the best of our knowledge. In operations research, two types of problems are relevant to "which" queries: inverse optimization problems [29, 32] and reverse optimization problems [30, 31].

For an inverse optimization problem, users specify a *feasible solution* x as input. Then the problem's parameters are tuned, with as low of a cost as possible, such that x becomes the optimal solution. For example, in an *inverse maximum flow problem* [29], users must first specify a feasible flow, f, and then a set of edge weight adjustments are returned to the user. The edge weight adjustments that are returned are adjustments that will make f become the maximum flow.

In reverse optimization problems, a user inputs a *target value* v for a particular problem instance. Then the problem instance's parameters are tuned, again with as low of a cost as possible, such that the input target value v becomes either the optimal value for the problem instance, or if v cannot become the optimal value for the problem instance, v becomes an upper bound of the optimal value for the problem instance. For example, in *reverse shortest path problems* [30], an input of the desired shortest path distance d to a shortest path query q yields an output of a set of edge weight adjustments that make the shortest path distance of q shorter than d.

These existing operations research problems require the user to *explicitly state* the target that is to be tuned (e.g., a feasible maximum flow f or the desired shortest path distance d). In contrast, "which" queries do not require the user to specify the particular target. "Which" queries are given a set of targets, because the actual optimal target may be completely unknown to the user. As such, "which" queries will *tell* the user the target's identity and also the target's optimized value. So long as the user includes the actual optimal target within their best guess set of targets, the "which" query will find the optimal target and value.

2.3 Network Maintenance

Finally, networking research also has certain overlaps with "which" queries. Existing networking research mostly focuses on designing network topologies (e.g., [21, 10]) and localizing faulty links *after* some links break (e.g., [23]). "Which" queries, like Q4, are useful for network maintenance. For network maintenance, "which" queries can be used to help determine critical links in a network. By locating critical links, effective preventive maintenance measures can be implemented *before* any critical link fails.

Furthermore, for designing network topologies, "which" queries, not unlike Q1, can also help during the design stages. The design stages are an iterative process. As such, during each iterative step, the current design is analyzed and then modified in preparation for the next step. The quality of the current design can be analyzed with "which" queries, and "which" queries can help the designers decide how to modify the current network design for the next iterative step.

Chapter 3

Elements of "Which" Queries

"Which" queries are diverse and have many applications in different fields. Nonetheless, we can formulate a "which" query from four constituent parts:

Graph Component: Specifies the quantity and type (e.g., vertex or edge) of an object considered in the query. For example, Q1 and Q5 consider "*which non-adjacent vertices*" and Q2–Q4 consider "*which existing edge*." Other possible queries may include "*which vertex*," or "*which k-edges*," etc.

Operation: Specifies how the graph component will eventually affect the graph. For Q1 and Q5, a new edge is eventually *added*; for Q2 and Q4, an existing edge is eventually *deleted*; and for Q3, an existing edge's weight is eventually *updated*.

Measurement: Specifies the graph measurement involved in the decision process, such as *the distance sum of a set of shortest path (SP) queries* (e.g., Q1–Q3) or *the maximum flow value* (e.g., Q4 and Q5). Other graph measures,

such as *betweenness* [22] and *multi-commodity flow* [1], can also be used.

Optimization Goal: Specifies the optimization goal of the query. For Q1, the optimization goal is to find the pair of non-adjacent vertices (u_i, v_i) that can *minimize* the distance sum of a set of shortest path queries if vertices u_i and v_i are connected with an edge. In Q5, the optimization goal is to find the pair of non-adjacent vertices (u_i, v_i) that can *maximally* increase the maximum flow value if vertices u_i and v_i are connected with an edge.

In this work, we specify a "which" query in the general form of [Component]-[Operation]-[Measurement]-[Goal]. In fact, "which" queries for specific applications may demand its own specific formulation. For example, Q1 and Q5 consider connecting non-adjacent vertices with *new edges*, but the weights and capacities of the new edges are actually application-specific. Furthermore, the "real cost" of each operation should be considered, too. For example, the real cost (e.g., administration cost, operation cost) of adding a flight between locations X and Y would be different than a flight between locations A and B.

As the first work to look into this new type of query, we focus on two specific "which" queries, namely the "Which Edge-Add-SP-Min" and "Which k-Edges-Add-SP-Min" queries, and present efficient algorithms to answer them. Both queries aim to *minimize* (goal) the *distance sum of a set of shortest path queries* (measurement) by *adding edges* (operation and graph component). The first query considers adding one edge and it can solve decision-support problems like Q1 (Chapter 4). The second query considers *adding multiple,k (where $k > 1$), new edges* that can connect k pairs of non-adjacent vertices (Chapter 5). Although "Which Edge-Add-SP-Min" and "Which k-Edge-Add-SP-Min" queries

are the same for $k = 1$, when $k > 1$, the two are entirely different types of queries. For $k > 1$, the solution space of "Which k-Edge-Add-SP-Min" queries is exponential and finding an optimal solution requires exhaustively checking the entire solution space. These two queries are fundamental "which" query types, and solutions for evaluating them efficiently can serve as a good foundation for the other "which" queries (Chapter 8). Although the two "which" queries look similar, their problem complexities differ a lot; furthermore, these queries are good examples illustrating how an evaluation algorithm of one "which" query type helps in evaluating other "which" query types.

Chapter 4

"Which Edge-Add-SP-Min" Queries

In this chapter, we will present a possible formulation and efficient solutions for "Which Edge-Add-SP-Min" queries. Given a positively weighted graph $G = (V, E)$, there are potentially many non-adjacent vertices in G; however, the number of non-adjacent vertex-pairs to be considered is application-dependent in practice. In the delivery network example (Q1), if we consider only adding a new *flight*, only locations with airports need to be considered. So, we model the set of non-adjacent vertex-pairs P as user-given. Each vertex-pair (u_i, v_i) in P is associated with an implied *bridging edge* e_i (where e_i connects u_i to v_i, and $\|e_i\|$ is the edge length). In addition to the edge length, each bridging edge $e_i(u_i, v_i)$ is also associated with a given *cost* c^{e_i}, which models the real-world cost of connecting u_i to v_i by e_i in G (e.g., the operational cost of that flight). In what follows, we use the term "non-adjacent vertex-pairs" and "bridging edge"

interchangeably.

The measurement of "Which Edge-Add-SP-Min" queries is the distance sum of a workload of shortest path queries Q. Each query $q_j(s_j, t_j) \in Q$ is a distinct shortest path query with source s_j and destination t_j. In one extreme, Q may contain all-pairs shortest path queries.

Each query q_j is associated with an importance factor m_{q_j}—using urban planning as an example, assuming that only one resident makes a 100 mile trip from s_1 to t_1, $q_1(s_1, t_1)$, and 50 residents make a 5 mile trip from s_2 to t_2, $q_2(s_2, t_2)$, we may set $m_{q_1} = 1$ and $m_{q_2} = 50$. Thus, query importances can model the number of beneficiaries of a bridging edge. For instance, the bridging edge $e_1(u_1, v_1)$, which reduces the shortest path distance of q_1 from 100 to 40 miles, is not as beneficial as the bridging edge $e_2(u_2, v_2)$, which reduces the shortest path distance of q_2 from 5 to 1 mile, because only one resident makes the trip q_1. More precisely, let $sp_{q_j}^G$ be the shortest path distance of query q_j on the graph G, and let $G^{\{e_i\}} = (V, E \cup \{e_i\})$. Then the benefit of connecting u_i and v_i by e_i on a query $q_j(s_j, t_j)$, or simply the *benefit of bridging edge e_i on query q_j*, $b_{q_j}^{e_i}$, is the reduction in shortest path distance of q_j in $G^{\{e_i\}}$ versus G, accounting for the query's importance factor m_{q_j}, i.e.,

$$b_{q_j}^{e_i} = m_{q_j} \times (sp_{q_j}^G - sp_{q_j}^{G^{\{e_i\}}}) \tag{4.1}$$

In the example above, the bridging edge e_1, which shortens q_1 from 100 to 40 miles, has a benefit $b_{q_1}^{e_1} = 1 \times (100 - 40) = 60$; whereas the bridging edge e_2, which shortens q_2 from 5 miles to 1 mile, has a benefit $b_{q_2}^{e_2} = 50 \times (5 - 1) = 200$.

Recall that a "Which Edge-Add-SP-Min" query minimizes the weighted dis-

tance sum of the query workload Q (i.e., finds $e_i \in P$ that has the ***maximum benefit*** on Q). We define the benefit B^{e_i} of a bridging edge e_i on a query workload Q, or simply *the benefit of* e_i, as the sum of e_i's benefits with respect to each query $q_i \in Q$, i.e.,

$$B^{e_i} = \sum_{j=1}^{|Q|} b_{q_j}^{e_i} \tag{4.2}$$

PROBLEM FORMULATION. *Given a graph G, a set P of non-adjacent vertex pairs (u_i, v_i), their associated bridging edge e_i and edge cost c^{e_i}, and a workload of shortest path queries Q, find a vertex-pair $(u_i, v_i) \in P$ so that if they are connected by e_i, they have the maximum benefit to workload Q, inclusive of the cost of adding e_i to G, i.e.,* $\arg\max_{(u_i,v_i)\in P}(B^{e_i} - c^{e_i})$.

In this formulation, we assume the cost c^{e_i} of a bridging edge e_i (e.g., 1 million USD) has been normalized to match the unit of distance reduction (e.g., 10 miles), as in any ranking function in database query processing. In fact, the relationship between the edge benefit B^{e_i} and the edge cost c^{e_i} is flexible; in some applications we can consider a different formulation, for example, using another function $\arg\max_{(u_i,v_i)\in P} \frac{B^{e_i}}{c^{e_i}}$. The techniques that we propose efficiently calculate the benefit of an edge, B^{e_i}. As such, if the function that we are interested in is $\arg\max_{(u_i,v_i)\in P} \frac{B^{e_i}}{c^{e_i}}$, our techniques remain unchanged. A summary of frequently used symbols is given in Table 4.1.

To find the vertex-pair $(u_i, v_i) \in P$ with the *highest benefit* (on a query workload Q), we first examine a brute-force solution using Figure 4.1 as a running example.[1] The brute-force solution can be viewed as the process of calculating

[1]For ease of illustration, we assume query importance values of 1 and bridging edge costs of 0 in Figure 4.1. In fact, our solutions can deal with arbitrary values of m_{q_j} and c^{e_i}.

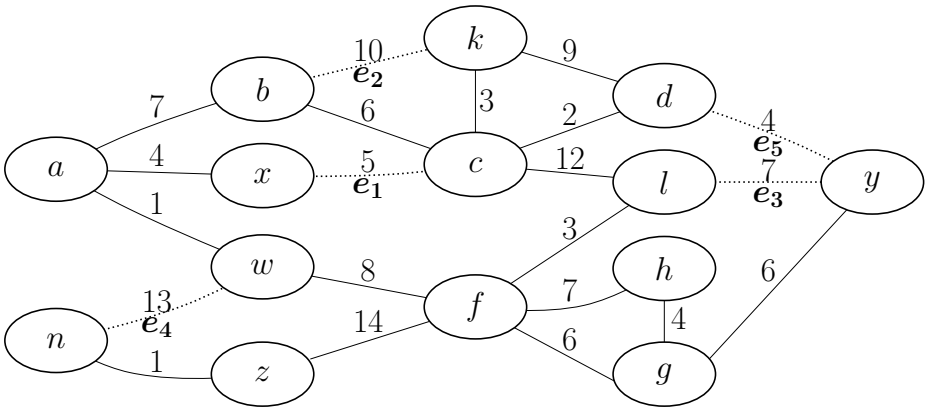
Symbol	Definition
P	Set of vertex-pairs
e_i	Bridging edge
$\|\|e_i\|\|$	Length of e_i
c^{e_i}	Cost of adding e_i to the graph G
$b_{q_j}^{e_i}$	Benefit of e_i with respect to q_j
B^{e_i}	Total benefit of e_i with respect to Q
UB^{e_i}	Upper bound benefit of e_i with respect to Q
LB^{e_i}	Lower bound benefit of e_i with respect to Q
Q	Workload of queries
q_j	Shortest path query
sp_{q_j}	Original shortest path distance for query q_j
m_{q_j}	Importance weighting for query q_j
UB_{q_j}	Upper bound benefit of q_j with respect to P
LB_{q_j}	Lower bound benefit of q_j with respect to P
G	Original input graph
θ	Threshold-pruning threshold value

Table 4.1. Table of Symbols

all the $b_{q_j}^{e_i}$ values in a search space of size $|Q| \times |P|$ (Figure 4.1(d)) and then returning the vertex-pair with the highest benefit B^{e_i}. To calculate the $b_{q_j}^{e_i}$ values, the brute-force solution temporarily bridges the vertex-pair $(u_i, v_i) \in P$ with e_i creating a new graph $G^{\{e_i\}}$. Then a shortest path algorithm SP (e.g., Dijkstra's algorithm) is used to compute the new shortest path distance of each query $q_j \in Q$ on $G^{\{e_i\}}$. After calculating e_i's benefit B^{e_i}, e_i is removed from $G^{\{e_i\}}$ to obtain the original graph G. This brute-force solution iteratively calculates the benefits of all edges and is not efficient because it invokes SP $|Q| \times |P|$ times.

If the all-pairs shortest path distances are available, the benefit of each bridging edge can be easily determined (details of this baseline solution are given in Chapter 10.1). However, maintaining the all-pairs shortest path distances requires $O(|V|^2)$ space, which is enormous[2] (this method ran out of memory in

[2]Of course, we can store all shortest path distances on disk, but doing so will result in excessive I/Os



(a) Input Graph G (bridging edges shown as dotted lines)

$e_i : (u_i, v_i)$	$\|e_i\|$	c^{e_i}
$e_1 : (c, x)$	5	0
$e_2 : (b, k)$	10	0
$e_3 : (y, l)$	7	0
$e_4 : (w, n)$	13	0
$e_5 : (d, y)$	4	0
$e_6 : (z, g)$	21	0

(b) Input Vertex-Pair Set P with e_i and c^{e_i}

Query q_j	sp_{q_j}	m_{q_j}
$q_1 : sp(a, d)$	15	1
$q_2 : sp(z, g)$	20	1
$q_3 : sp(h, y)$	10	1
$q_4 : sp(x, w)$	5	1
$q_5 : sp(k, c)$	3	1

(c) Input Query Workload Q

	P					
Q	$e_1 : (c, x) = 5$	$e_2 : (b, k) = 10$	$e_3 : (y, l) = 7$	$e_4 : (w, n) = 13$	$e_5 : (d, y) = 4$	$e_6 : (z, g) = 21$
$q_1 : sp(a, d)$=15	$b_{q_1}^{(c,x)}$=?	$b_{q_1}^{(b,k)}$=?	$b_{q_1}^{(y,l)}$=?	$b_{q_1}^{(w,n)}$=?	$b_{q_1}^{(d,y)}$=?	$b_{q_1}^{(z,g)}$=?
$q_2 : sp(z, g)$=20	$b_{q_2}^{(c,x)}$=?	$b_{q_2}^{(b,k)}$=?	$b_{q_2}^{(y,l)}$=?	$b_{q_2}^{(w,n)}$=?	$b_{q_2}^{(d,y)}$=?	$b_{q_2}^{(z,g)}$=?
$q_3 : sp(h, y)$=10	$b_{q_3}^{(c,x)}$=?	$b_{q_3}^{(b,k)}$=?	$b_{q_3}^{(y,l)}$=?	$b_{q_3}^{(w,n)}$=?	$b_{q_3}^{(d,y)}$=?	$b_{q_3}^{(z,g)}$=?
$q_4 : sp(x, w)$=5	$b_{q_4}^{(c,x)}$=?	$b_{q_4}^{(b,k)}$=?	$b_{q_4}^{(y,l)}$=?	$b_{q_4}^{(w,n)}$=?	$b_{q_4}^{(d,y)}$=?	$b_{q_4}^{(z,g)}$=?
$q_5 : sp(k, c)$=3	$b_{q_5}^{(c,x)}$=?	$b_{q_5}^{(b,k)}$=?	$b_{q_5}^{(y,l)}$=?	$b_{q_5}^{(w,n)}$=?	$b_{q_5}^{(d,y)}$=?	$b_{q_5}^{(z,g)}$=?
B^{e_i}	$\sum_{j=1}^{5} b_{q_j}^{(c,x)}$	$\sum_{j=1}^{5} b_{q_j}^{(b,k)}$	$\sum_{j=1}^{5} b_{q_j}^{(y,l)}$	$\sum_{j=1}^{5} b_{q_j}^{(w,n)}$	$\sum_{j=1}^{5} b_{q_j}^{(d,y)}$	$\sum_{j=1}^{5} b_{q_j}^{(z,g)}$

(d) The Search Space

Figure 4.1. Running Example

	P					
Q	e_1: (c,x)=5	e_2: (b,k)=10	e_3: (y,l)=7	e_4: (w,n)=13	e_5: (d,y)=4	B_{q_j}
$q_1 : sp(a,d)$=15	$b_{q_1}^{(c,x)}$=[0,10]	$b_{q_1}^{(b,k)}$=[0,5]	$b_{q_1}^{(y,l)}$=[0,8]	$b_{q_1}^{(w,n)}$=[0,2]	$b_{q_1}^{(d,y)}$=[0,11]	$\sum_{i=1}^{5} b_{q_1}^{e_i}$=[0,36]
$q_2 : sp(z,g)$=20	$b_{q_2}^{(c,x)}$=[0,15]	$b_{q_2}^{(b,k)}$=[0,10]	$b_{q_2}^{(y,l)}$=[0,13]	$b_{q_2}^{(w,n)}$=[0,7]	$b_{q_2}^{(d,y)}$=[0,16]	$\sum_{i=1}^{5} b_{q_2}^{e_i}$=[0,61]
$q_3 : sp(h,y)$=10	$b_{q_3}^{(c,x)}$=[0,5]	—	$b_{q_3}^{(y,l)}$=[0,3]	—	$b_{q_3}^{(d,y)}$=[0,6]	$\sum_{i=1}^{5} b_{q_3}^{e_i}$=[0,14]
$q_4 : sp(x,w)$=5	—	—	—	—	$b_{q_4}^{(d,y)}$=[0,1]	$\sum_{i=1}^{5} b_{q_4}^{e_i}$=[0,1]
B^{e_i}	$\sum_{j=1}^{4} b_{q_j}^{(c,x)}$=[0,30]	$\sum_{j=1}^{4} b_{q_j}^{(b,k)}$=[0,15]	$\sum_{j=1}^{4} b_{q_j}^{(y,l)}$=[0,24]	$\sum_{j=1}^{4} b_{q_j}^{(w,n)}$=[0,9]	$\sum_{j=1}^{4} b_{q_j}^{(d,y)}$=[0,34]	

Figure 4.2. Search Space after Search Space Reduction and Bounds Initialization (entries with "—" mean [0,0] benefits)

all of our experiments with 8GB of RAM).

In the remainder of this chapter, we will present our solutions, which our experiments have shown to be orders of magnitudes faster than the basic solutions discussed above and maintaining a linear, to V, consumption of memory. In Chapter 4.1, we present some simple pruning rules to reduce the search space. In Chapter 4.3.1, we describe how to use *thresholding*, a classic top-k query processing technique, as the solution framework for this type of "which" queries. In Chapter 4.3.2, we present two techniques for calculating the benefits of bridging edges without adding each edge to the graph G. Finally, in Chapter 4.3.3, we present two algorithms that exploit the aforementioned techniques.

4.1 Search Space Reduction

In some applications (e.g., Q1), the size of P can be very large since any pair of indirectly connected vertices of the input graph G can potentially be the answer. As such, Search Space Reduction uses the following lemma to reduce the number of vertex-pairs in P that need to be considered:

instead.

Lemma 1 *A vertex-pair (u_i, v_i) with bridging edge e_i can be removed from consideration if*

$$\|e_i\| \geq \max(sp_{q_1}, \ldots, sp_{q_{|Q|}})$$

Proof. Let $\lambda = \max(sp_{q_1}, \ldots, sp_{q_{|Q|}})$ be the shortest path distance of the longest query $q_j \in Q$. If $\|e_i\| > \lambda$, it is straightforward to see that e_i cannot reduce the shortest path distance for any query in Q and can be removed from consideration. If $\|e_i\| = \lambda$ and $\lambda = sp_{q_j}$, then even if u_i and v_i are the same vertices as s_j and t_j, respectively, the shortest path distance of q_j would remain unchanged; thus, the bridging edge e_i also has no benefit to any of the queries in Q and can be removed from consideration. □

The intuition of this lemma is very simple: if edge e_i is already longer than all query lengths, then no queries would include it in their shortest paths. Thus, B^{e_i} must be 0 and can be safely removed. Similarly, the following lemma removes some queries from consideration:

Lemma 2 *A query, q_j, can be removed from consideration if*

$$sp_{q_j} \leq \min(\|e_1\|, \ldots, \|e_{|P|}\|)$$

Proof. Let $\lambda = \min(\|e_1\|, \ldots, \|e_{|P|}\|)$ be length of the shortest bridging edge $e_i \in P$. It is straightforward to see that if all bridging edges $e_i \in P$ are not shorter than the length of a shortest path query q_j, e_i cannot reduce the shortest path distance for the query q_j. Thus, the q_j can be removed from consideration.

□

In our running example, Lemmas 1 and 2 respectively remove bridging edge $e_6(z, g)$ and query $q_5(k, c)$ from consideration.

4.2 Incremental Update Based Algorithm

The basic solutions mentioned above require temporarily modifying the graph data in order to calculate the benefit of a bridging edge. Only the all-pairs shortest path method mentioned above is free from the graph modification; however, the all-pairs shortest path method requires and exorbitant amount of space to store the shortest-path distances.

In this section, we will present an algorithm that does not require modifying the underlying graph data to calculate a bridging edge's benefit. In order to calculate the benefit of a bridging edge, we use a shortest path incremental update (SPI) algorithm [14] to determine how the addition of the bridging edge affects the shortest path distance of a given query. Thus, we will first discuss how a shortest path incremental update algorithm can be used to answer "Which Edge-Add-SP-Min" queries.

To begin, we first need to understand how the shortest path incremental update algorithm works. After understanding how the shortest path incremental update algorithm works, we will discuss how we can use it, incrementally, to answer "Which Edge-Add-SP-Min" queries.

4.2.1 Shortest Path Incremental Updates

A shortest path incremental update algorithm, as its name clearly indicates, is an algorithm that will efficiently update the shortest path distances of a single-source shortest path query. A shortest path incremental algorithm maintains a copy of the shortest path distances of all vertices from a specific source vertex. Then, when an edge is inserted into the graph, the shortest path incremental update algorithm is triggered to update the shortest path distances of all vertices affected by the inserted edge.

Given a newly inserted edge e, the shortest path incremental algorithm must determine which vertices need to have their shortest path distances updated. The shortest path distances from the source vertex s to the vertices of the edge e, vertices u and v, are already known. Furthermore, by inserting the edge e into the graph G, only vertices with a shortest path distance greater than v and u can be affected by the insertion of e. Thus, all vertices with shortest path distances less than v and u do not need to have their shortest path distances updated. Additionally, at most one vertex, either u or v, will have its shortest path distance affected by adding e to the graph G, which consequently affects all vertices with a shortest path passing through said vertex.

Three cases arise for determining whether u or v is affected by the inserted edge e. The first case is that u is affected by the new edge e. The second case is that v is the affected vertex, rather than u. The last case is that neither u or v is affected by the insertion of edge e.

First, let's examine the first case of u being affected by inserting e. For u to be affected by the edge e, this means that the shortest path distance from s

to v must be shorter than the shortest path distance from s to u. Furthermore, because e connects vertices u and v together, the shortest path distance from s to u must be larger than the shortest path distance from s to v plus the edge length of e. Thus, all vertices with a shortest path passing through u must also have their shortest path distances updated. The SPI algorithm will then invoke Dijkstra's algorithm to calculate the new shortest path distances for all affected vertices. This Dijkstra's invocation is done with the following conditions:

1. Dijkstra's algorithm is modified to initialize the shortest path distances of all the vertices with their shortest path distances from s.

2. The source vertex for this Dijkstra's expansion is set to u, because u is the first vertex affected by the inclusion of edge e.

3. The shortest path distance of u is set to the shortest path distance from s to v plus the edge length of e.

In the case of v being affected by the insertion of e, we can see that the same conditions hold for v as it did for u, in the first case. As such, the new shortest path distance of v is set to the shortest path distance from s to u plus the length of edge e, and the source vertex for the resulting Dijkstra's expansion is set to vertex v.

Now, the last case, where neither u or v are affected by e, means that no other vertices are affected by e as well. As such, the shortest path incremental update algorithm does not need to invoke Dijkstra's algorithm and can return immediately.

4.2.2 Using SPI for "Which Edge-Add-SP-Min" Queries

Now we will present how to modify SPI to answer "Which Edge-Add-SP-Min" queries. We propose an iterative shortest path incremental update algorithm (ISPI), which iteratively uses a shortest path incremental update algorithm instead of a shortest path algorithm to compute the benefit values. If an edge e_i is added to a graph G, the new shortest path distance between two query vertices s_j and t_j in $G^{\{e_i\}}$ can be calculated more efficiently using a SPI algorithm, which caches the shortest path tree obtained from a shortest path calculation on G.

ISPI works as follows: for each query q_j, we first compute its shortest path distance on G using any shortest path algorithm, then, the benefit of each bridging edge e_i on q_j is computed using the SPI algorithm. Compared with the brute-force solution, ISPI invokes a shortest path algorithm $|Q|$ times and a SPI algorithm $|Q| \times |P|$ times, which reduces the total execution time of ISPI. The pseudocode for ISPI can be found in Algorithm 1

Algorithm 1 ISPI

```
1:  function ISPI(G, Q, P)
2:      SearchSpaceReduction(Q, P)                    ▷ Chapter 4.1
3:      for q ∈ Q do
4:          Dijkstra(s, t)
5:          cache shortest path tree
6:          for e ∈ P do
7:              B^{e_i} += SPI(G, q, e)               ▷ Algorithm 2
8:              restore updated shortest path distances   ▷ Chapter 4.2.2.1
9:          end for
10:     end for
11:     return most beneficial edge
12: end function
```

4.2.2.1 Optimizations

We note that the shortest path incremental algorithm is defined for a single-source shortest path query. Because "Which Edge-Add-SP-Min" queries are single-pair shortest path queries, we modify the shortest path incremental algorithm to work under the constraints of a "Which Edge-Add-SP-Min" query.

Because SPI works on a single-source shortest path query, we need to impose extra conditions for determining which vertices require their shortest path distances to be updated. First, if Dijkstra's algorithm, as used in the update phase of SPI, visits the destination vertex t of the single-pair shortest path query, then all relevant vertices have had their shortest path distances updated. Because we are evaluating a single-pair shortest path query, once the new shortest path distance to t has been found, we can terminate updating other vertex distances.

We note that Dijkstra's algorithm may not be able to reach vertex t from either vertex u or v after the insertion of edge e into G, during the update phase of SPI. Thus, we can determine that e does not affect the shortest path distance from s to t if we update a vertex with a shortest path distance greater than the original shortest path distance from s to t. We can see the modified version of SPI, for use in answering "Which Edge-Add-SP-Min" queries, in Algorithm 2.

ISPI iteratively calls SPI to determine the benefit of each bridging edge in P. The updated shortest path distances calculated by SPI are only temporary for ISPI's usage. Thus, before each subsequent call to SPI, ISPI needs to restore the shortest path distances of any updated vertex to the cached shortest path distances.

Algorithm 2 SPI

```
1: function SPI(G, q, e)
2:     if sp_{s,u} + ‖e‖ < sp_{s,v} then
3:         root = u
4:     else if sp_{s,v} + ‖e‖ < sp_{s,u} then
5:         root = v
6:     else
7:         return sp_q
8:     end if
9:     initialize shortest path distances from cached values
10:    Dijkstra(root, t, sp_q)
11:    return shortest path distance to t
12: end function
```

If SPI immediately returns, because no vertices require their shortest path distances to be updated, we can immediately go to the next iteration. However, if some vertices have had their shortest path distances updated by SPI, then only those vertices need to be restored with the cached shortest path distances. As such, during each SPI call to Dijkstra's algorithm, we maintain a list of vertices that have been visited by Dijkstra's algorithm (i.e., the list of vertices that have had their shortest path distances updated by SPI). Then, before the next iterative call to SPI by ISPI, we restore the shortest path distances of all vertices in the list with their cached shortest path distances.

4.3 Top-K Based Algorithms

Using ISPI to incrementally compute the effects of a bridging edge can at times be rather expensive. As we can see from the SPI algorithm, the location of the bridging edge within the shortest path tree determines the number of vertices that are affected by inserting the selected bridging edge. Thus, if many bridging

edges are located very close to the source vertex of a query, and the bridging edge vertices only offer a marginal reduction in shortest path distances, the cost of the shortest path incremental update becomes high. Furthermore, because the incremental update is now more expensive due to more vertices having their shortest path distances updated, ISPI also becomes more expensive because more vertices have been touched and need to have their shortest path distances reset from the cached values for the next iteration.

Due to the aforementioned drawbacks of the ISPI algorithm, in this section we will present a set of algorithms that are independent on the number of vertices a bridging edge affects. The proposed algorithms below draw on Top-k query processing principles to efficiently compute a bridging edge's benefit.

4.3.1 Threshold Pruning

Notice that, currently, we are actually looking for only *one* vertex-pair that has the highest benefit. Our search space is only of size $|P|$; however, since the benefit of a vertex-pair is the amount reduced in the shortest-path distance sum, we still need to compute $|Q| \times |P|$ benefit entries, $b_{q_j}^{e_i}$. So, a question that naturally arises is *whether it is possible to stop early and/or skip some of the* $|Q| \times |P|$ *benefit entries?*

In fact, we can. Here, we propose to maintain *the upper and lower benefit bounds* of each vertex-pair throughout the search process. The value of the largest lower bound benefit of a vertex-pair achieved thus far is set as a threshold θ. During execution of the query, all vertex-pairs with an upper bound benefit less than θ are guaranteed to not be the answer and are pruned. Finally, we can

terminate the search if no vertex-pair has an upper bound benefit value greater than θ.

Take Figure 4.1 as an example. If bridging c and x by e_1 can shorten the shortest path of $q_1(a, d)$, e_1 at most reduces q_1's shortest path distance to 5, which is e_1's length. In this case, the benefit $b_{q_1}^{(c,x)}$ would be $m_{q_1} \times (sp_{q_1}^G - \|e_1\|) = 1 \times (15 - 5) = 10$. On the other hand, it is possible that $e_1(c, x)$ cannot shorten $q_1(a, d)$ at all. In this case, $b_{q_1}^{(c,x)} = 0$. So, after search space reduction (Chapter 4.1), we can initialize the upper and lower bound benefits of any bridging edge e_i to any query q_j as $m_{q_j} \times \max\big((sp_{q_j}^G - \|e_i\|), 0\big)$ and 0, respectively. Note that the benefit bounds of a bridging edge B^{e_i} can be derived from the benefit bounds of the corresponding column in the search space. Figure 4.2 shows the reduced search space of the running example after search space reduction and bounds initialization, which has been reduced to 14 entries from an original of $|Q| \times |P| = 5 \times 6 = 30$ entries. Throughout the search process, the benefit bound of a bridging edge can be dynamically tightened after a SP execution.[3]

In our example, assume that there is a bridging edge $e_7(a, d)$ with length $\|e_7\| = 5$, and that a Dijkstra's expansion has been executed for $q_1(a, d)$ on $G^{\{e_7\}}$, yielding $b_{q_1}^{e_7} = 10$. In this case, the lower benefit bound LB^{e_7} of e_7 is updated to 10, the upper benefit bound UB^{e_7} of e_7 remains as 30, and $\theta = 10$. Since θ is greater than e_4's upper bound benefit (Figure 4.2), e_4 can be pruned.

Our two proposed algorithms follow the thresholding framework detailed

[3]In this work, we use Dijkstra's algorithm to compute shortest path distances. If a shortest path index is available, our algorithms can call such an index instead. In any case, the key point is that our algorithms are designed to minimize those calls. As another example, if the input graphs are too large to fit in memory, we can use any disk-based shortest path algorithm (e.g., [5]) instead. Again, our algorithms are able to reduce those calls.

above and use the execution strategies below to calculate benefit values for bridging edges without the need to modify the underlying graph G. The first algorithm (Chapter 4.3.3.1) sticks to using the same strategy after it has determined the best strategy, based on the input information. The second algorithm (Chapter 4.3.3.2), adaptively changes the execution strategy based on the information collected during runtime. Both algorithms dynamically prune non-answers and stop early once the stopping condition is met.

4.3.2 Insertion Free Benefit Calculations

A basic solution is inefficient because it requires that each bridging edge e_i be temporarily inserted into the graph G followed by computing the benefit (distance reduction) for the inserted edge—requiring at most $|Q| \times |P|$ invocations of a shortest path to compute the benefit of bridging edges. Alternatively, a basic solution that uses the all-pairs shortest path distances would require an enormous amount of space to maintain the all-pairs shortest path distances.

In the following, we show how to use at most two Dijkstra's executions, without modifying G, to calculate (i) the benefit of a bridging edge with respect to *the entire workload* Q (Chapter 4.3.2.1) or (ii) the benefits of *every bridging edge* in P with respect to a particular query (Chapter 4.3.2.2). Mapping back to the search space illustrated in Figure 4.2, each invocation of the two methods can respectively determine the benefits of an entire column (B^{e_i}) or an entire row (B_{q_j}).

4.3.2.1 Edge Oriented Expansion (EOE)

We now introduce the Edge Oriented Expansion (EOE) method to calculate the benefit B^{e_i} of a bridging edge e_i using at most two Dijkstra's expansions, instead of $|Q|$. First, let us consider the following lemma:

Lemma 3 *Given a bridging edge $e_i(u_i, v_i)$, let T be the set of destination vertices of all queries $q_j \in Q$. Then, from a single Dijkstra's expansion, with source vertex v_i, we can confine the expansion distance to*

$$r^{e_i} = \max(sp_{q_1}, \ldots, sp_{q_{|Q|}}) - \|e_i\|$$

to obtain all necessary shortest path distances for calculating B^{e_i}. Furthermore, let the set of vertices visited by the expansion be N. If $T \cap N = \varnothing$, then $B^{e_i} \leq 0$.

Proof. For a bridging edge e_i to benefit a query $q_j \in Q$, we must have $sp_{q_j} > dist_{q_j}^{e_i}$. We then substitute the detour distance $dist_{q_j}^{e_i}$ with Equation 4.4. Noting that $sp_{(s_j,u_i)} \geq 0$ always holds, we arrive at $sp_{q_j} > \|e_i\| + sp_{(v_i,t_j)}$ and thus obtain $sp_{(v_i,t_j)} < sp_{q_j} - \|e_i\|$. Because $\max(sp_{q_1}, \ldots, sp_{q_{|Q|}}) - \|e_i\| = r^{e_i}$ and $sp_{q_j} \leq \max(sp_{q_1}, \ldots, sp_{q_{|Q|}})$, we derive that $sp_{(v_i,t_j)} < r^{e_i}$. Thus, r^{e_i} is the minimum distance that Dijkstra's expansion must traverse before being able to determine that e_i cannot have any benefit to the longest query in Q. As N is the set of vertices visited by a Dijkstra's expansion sourced from v_i and of distance r^{e_i}, $T \cap N = \varnothing$ implies that e_i cannot shorten any queries in Q. Extensions to the undirected graph case is straightforward. We use Equation 4.3 to calculate the $dist_{q_j}^{e_i}$ values. Furthermore, because G is undirected, T contains all the source and destination vertices for queries $q_j \in Q$. □

Lemma 3 confines the expansion distance. Furthermore, given a bridging edge $e_i(u_i, v_i)$, after executing the confined Dijkstra's expansion, if we determine that $B^{e_i} \leq 0$ by Lemma 3, then no further calculations are needed for e_i and another bridging edge can be examined. However, if we cannot conclusively determine $B^{e_i} \leq 0$ (i.e., $T \cap N \neq \varnothing$) in the first Dijkstra's expansion, another expansion is required to determine the true benefit of a bridging edge e_i. If $t_j \in T \cap N$, then it is possible that e_i has some benefit to q_j from the current information at hand. To determine the benefit of e_i to q_j, we can compute the shortest path distance for the query $q_j(s_j, t_j)$ using a detour path that includes the edge $e_i(u_i, v_i)$ as:

$$dist_{q_j}^{e_i} = \min\big(sp_{(s_j,u_i)} + \|e_i\| + sp_{(v_i,t_j)}, \\ sp_{(s_j,v_i)} + \|e_i\| + sp_{(u_i,t_j)}\big) \tag{4.3}$$

for an undirected graph, or for a directed graph:

$$dist_{q_j}^{e_i} = sp_{(s_j,u_i)} + \|e_i\| + sp_{(v_i,t_j)} \tag{4.4}$$

The shortest path distances from u_i are unknown and must be obtained from a Dijkstra's expansion using u_i as the source vertex and a distance of r^{e_i} (in an undirected graph). If G is a directed graph, then the shortest path distances from u_i are obtained by a reverse Dijkstra's expansion, which reverses all the edge directions in the graph, with source vertex u_i and distance r^{e_i}.

Note that the detour path distance $dist_{q_j}^{e_i}$ may be longer than the original shortest path distance sp_{q_j} because it deliberately passes through e_i. As such, the shortest path distance of q_j on the graph $G^{\{e_i\}}$ is given as $sp_{q_j}^{G^{\{e_i\}}} =$

$\min\left(dist_{q_j}^{e_i}, sp_{q_j}^{G}\right)$ and the benefit of e_i on query q_j can be determined using Equation 4.1. From at most two Dijkstra's expansions,

$dist_{q_j}^{e_i}$ of all queries q_j in Q can be obtained; thus, an entire column in the search space, or B^{e_i}, can be obtained using Equation 4.2.

A naive way of using EOE is to invoke EOE once for each vertex-pair in P and then identify the edge with the highest benefit. Despite Search Space Reduction and Threshold Pruning, this naive use of EOE finds the answer using at most $2|P|$ Dijkstra's expansions (independent of Q), without actually inserting each bridging edge to the graph G one-by-one. The pseudocode for EOE can be found in Algorithm 3. An invocation of EOE finds the exact benefit B^{e_i} of a bridging edge and only requires $O(|V|)$ memory to hold the necessary shortest path distances from each Dijkstra's expansion.

Algorithm 3 Function—Edge Oriented Expansion

```
1:  function EOE(e_i(u_i, v_i), Q)
2:      r^{e_i} = max(sp_{q_1}, ..., sp_{q_|Q|}) - ||e_i||        ▷ Lemma 3
3:      Dijkstra(v_i, r^{e_i})
4:      if T ∩ N ≠ ∅ then                                         ▷ Lemma 3
5:          ReverseDijkstra(u_i, r^{e_i})
6:      end if
7:      for all q_j ∈ Q do
8:          Calculate benefit b_{q_j}^{e_i}                       ▷ Equation 4.1
9:          LB^{e_i} += b_{q_j}^{e_i}
10:         UB^{e_i} = LB^{e_i}
11:     end for
12:     return average number of vertices visited
13: end function
```

4.3.2.2 Query Oriented Expansion (QOE)

Now, we present a counterpart of EOE, Query Oriented Expansion, which calculates the upper and lower bounds on the benefits of a query with respect to each bridging edge in P (an entire row in the search space, B_{q_j}) using at most two Dijkstra's expansions. QOE is important because when $|Q| \ll |P|$, we can fill the entire search space by calling the QOE procedure for each query, which requires at most $2|Q|$ Dijkstra's expansions (versus EOE's $2|P|$ Dijkstra's expansions).

QOE is similar to EOE, except that we carry out one Dijkstra's expansion using s_j as the source vertex (instead of v_i) and, if needed, a Dijkstra's expansion using t_j (instead of u_i) as the source making use of the following lemma:

Lemma 4 *Given a query $q_j(s_j, t_j)$, let U be the set of head vertices of $e_i \in P$. Then, from a single Dijkstra's expansion, with source vertex s_j we can confine the expansion distance to*

$$r^{q_j} = sp_{q_j} - \min\big(\|e_1\|, \ldots, \|e_{|P|}\|\big)$$

and obtain all necessary shortest path distances for calculating B_{q_j}. Furthermore, let the set of vertices visited by the expansion be N. If $U \cap N = \varnothing$, then $B_{q_j} \leq 0$.

Proof. For a query q_j to be benefited by a bridging edge e_i, we must have $sp_{q_j} > dist_{q_j}^{e_i}$. We then substitute the detour distance $dist_{q_j}^{e_i}$ with Equation 4.4. Also, note that $sp_{(v_i,t_j)} \geq 0$ must hold. We then derive $sp_{q_j} > sp_{(s_j,u_i)} + \|e_i\|$ and thus obtain $sp_{(s_j,u_i)} < sp_{q_j} - \|e_i\|$. Since $sp_{q_j} - \|e_i\| \leq sp_{q_j} - \min\big(\|e_1\|, \ldots, \|e_{|P|}\|\big) = r^{q_j}$, we derive that $sp_{(s_j,u_i)} < r^{q_j}$. Thus, r^{q_j} is the minimum distance that a

Dijkstra's expansion sourced at s_j must traverse before all bridging edges $e_i \in P$ can be determined to not have a benefit to q_j. Furthermore, $U \cap N = \varnothing$ implies that all bridging edges are of a distance no shorter than r^{q_j} from s_j, thus no bridging edges in P can reduce the shortest path distance of query q_j. Extension to the undirected graph case is straightforward. We use Equation 4.3 to calculate the $dist_{q_j}^{e_i}$ values. Furthermore, because e_i is undirected, U contains both vertices u and v for each $e_i \in P$. □

From a Dijkstra's expansion sourced at s_j and constrained to a maximum expansion distance r^{q_j}, if $U \cap N \neq \varnothing$, then a reverse Dijkstra's expansion sourced at t_j of distance r^{q_j} is required. The shortest path detour distances of query q_j that pass through each bridging edge e_i is calculated using Equation 4.3 for undirected graphs or Equation 4.4 for directed graphs. Finally, the benefit of each bridging edge e_i on query q_j (B_{q_j}) can be calculated using Equation 4.1 where $sp_{q_j}^{G^{\{e_i\}}} = \min\left(dist_{q_j}^{e_i}, sp_{q_j}^{G}\right)$. The upper and lower benefit bounds for e_i, UB^{e_i} and LB^{e_i}, are respectively updated.

The pseudocode for QOE can be found in Algorithm 4. A QOE invocation finds the benefits of $|P|$ bridging edges with respect to the same query. So, a naive way of using QOE is to invoke QOE once for each query in Q. After that, we can identify the edge with the highest benefit. Similar to the naive use of EOE, this naive method can find the answer using at most $2|Q|$ Dijkstra's expansions (independent of $|P|$) and only requires $O(|V|)$ memory space to hold the necessary shortest path distances from each Dijkstra's expansion.

Algorithm 4 Function—Query Oriented Expansion

1: **function** QOE($q_j(s_j, t_j), P$)
2: $r^{q_j} = sp_{q_j}$ - $\min\big(\|e_1\|, \ldots, \|e_{|P|}\|\big)$ ▷ Lemma 4
3: `Dijkstra`$\big(s_j, r^{q_j}\big)$
4: **if** $T \cap N \neq \varnothing$ **then** ▷ Lemma 4
5: `ReverseDijkstra`$\big(t_j, r^{q_j}\big)$
6: **end if**
7: **for all** $e_i \in P$ **do**
8: Calculate benefits $b^{e_i}_{q_j}$ ▷ Equation 4.1
9: LB^{e_i} += $b^{e_i}_{q_j}$
10: UB^{e_i} -= $m_{q_j} \times \big(sp_{q_j}$ - $\|e_i\|\big)$
11: **end for**
12: **return** average number of vertices visited
13: **end function**

4.3.3 Algorithms

We now present two algorithms that exploit the aforementioned techniques of search space reduction (Chapter 4.1), threshold pruning (Chapter 4.3.1), and insertion free benefit calculations (Chapter 4.3.2). While it is clear that both algorithms will follow the Threshold Pruning framework (Chapter 4.3.1) to prune non-answer entries and use search space reduction (Chapter 4.1) to reduce the search space, whether the algorithms should use EOE, QOE, or both, to perform insertion free benefit calculations has not yet been discussed. In fact, whether EOE or QOE is more efficient depends on many factors including the sizes of Q and P (search space size); and the characteristics of the input graph G, query workload Q, and bridging edges (which affect Dijkstra's execution times). Therefore, our first algorithm goes for a proactive approach that first invests two calls to QOE to estimate whether EOE or QOE has better efficiency for the current input, and then uses the lower cost option to finish processing the query. Our

second algorithm uses an adaptive approach to switch between EOE and QOE to answer the query.

4.3.3.1 Proactive Algorithm (PA)

The Proactive Algorithm operates in three phases. The first phase reduces the search space (Chapter 4.1). The second phase determines whether using EOE or QOE is more efficient on the input parameters. In the third phase, PA processes the "which" query and returns the final result. Algorithm 5 shows the pseudocode for PA.

Algorithm 5 Proactive Algorithm (PA)

```
1:  function PA(G, Q, P)
2:      SearchSpaceReduction(Q, P)                          ▷ Chapter 4.1
3:      ChooseEOEorQOE(G, Q, P)                             ▷ Algorithm 6
4:      if EstVertexVisit_EOE ≤ EstVertexVisit_QOE then     ▷ Use EOE
5:          InitialBounds(Q, P)                             ▷ Chapter 4.3.1
6:          while θ < maxUB do
7:              e = arg max_{e_i ∈ P}(UB^{e_1}, ... UB^{e_|E|})
8:              EOE(e, Q)                                   ▷ Algorithm 3
9:              maxUB = max(UB^{e_1}, ..., UB^{e_|E|})
10:             θ = max(LB^{e_1}, ..., LB^{e_|E|})
11:         end while
12:     else                                                ▷ Use QOE
13:         InitialBounds(Q, P)                             ▷ Chapter 4.3.1
14:         while θ < maxUB do
15:             q = arg max_{q_j ∈ Q}(UB_{q_1}, ... UB_{q_|Q|})
16:             QOE(q, P)                                   ▷ Algorithm 4
17:             maxUB = max(UB^{e_1}, ..., UB^{e_|E|})
18:             θ = max(LB^{e_1}, ..., LB^{e_|E|})
19:         end while
20:     end if
21:     return highest benefit vertex-pair
22: end function
```

The second phase of PA determines whether using EOE or QOE is more efficient based on the input parameters. The pseudocode of the estimation procedure (`ChooseEOEorQOE`) is found in Algorithm 6. The idea of the procedure is to formulate a power law equation $v = \alpha d^{\beta}$ that can roughly estimate the number of vertices visited v by Dijkstra's algorithm for a given expansion distance d. Since the running time of Dijkstra's algorithm is proportional to the number of vertices visited, the power-law equation can be used to estimate the running times of EOE and QOE. We can determine the values of α and β by running some queries in Q to obtain some values of v and d. Noting that $\log v$ is linear to $\log d$, we can select ϵ queries in Q to solve for α and β—consequently, we do not have to perform QOE for said queries if QOE is later chosen to be used. We apply linear regression to estimate the values α and β for the best-fit line. Because QOE may not be more efficient than EOE for the input instance, we only allocate a small cost for this sampling step (i.e., $\epsilon = 2$). For $\epsilon = 2$, we pick the longest and the shortest queries in Q for the estimation because these endpoints of the regression line are far apart. Using the derived power-law equation,PA estimates the worst case cost of EOE as the number of vertices visited by the longest query times $2|P|$ (because the expansion distance of EOE is bounded by the longest query (Lemma 3)). Similarly, PA estimates the worst case cost of QOE as the sum of the vertex visits of each remaining query in Q (i.e., substituting the query length of each remaining query as d).

After choosing the benefit calculation method (Line 3), the benefit bounds of the bridging edges and the queries are initialized (Line 5 or 13) according to Chapter 4.3.1 (N.B. it is not necessary to store the benefit bound of each individual entry, only B^{e_i}). The while loop Lines 6–11 for EOE and Lines 14–19

Algorithm 6 Function—Choosing EOE or QOE

1: **function** ChooseEOEorQOE(P, Q)
2: VerticesVisitedLQ = QOE$\big(\arg\max_{q_j \in Q}(sp_{q_1}, \ldots, sp_{q_{|Q|}})\big)$
3: VerticesVisitedSQ = QOE$\big(\arg\min_{q_j \in Q}(sp_{q_1}, \ldots, sp_{q_{|Q|}})\big)$
4: $\beta = \big(\log \frac{\text{VerticesVisitedLQ}}{\text{VerticesVisitedSQ}}\big) / \big(\log \frac{\max(sp_{q_1}, \ldots, sp_{q_{|Q|}})}{\min(sp_{q_1}, \ldots, sp_{q_{|Q|}})}\big)$
5: α = VerticesVisitedLQ $/\big(\max(sp_{q_1}, \ldots, sp_{q_{|Q|}})\big)^{\beta}$
6: EstVertexVisit_EOE $= 2 * |P| *$ VerticesVisitedLQ
7: **for all** $q_j \in Q$ **do**
8: EstVertexVisit_QOE $+= 2 * \alpha * (sp_{q_j})^{\beta}$
9: **end for**
10: **return** EstVertexVisit_QOE and EstVerticesVisit_EOE
11: **end function**

for QOE and the threshold θ are the realization of the thresholding framework (Chapter 4.3.1). One remaining consideration that needs to be addressed is which edge (for EOE) or which query (for QOE) should be carried out first? Note that the order of carrying out EOE and QOE has a performance impact because if the final answer is able to be examined early in the process, its high LB value will give a high value to θ, effectively pruning more EOE or QOE operations. In this regard, we notice that we should use the *upper bound principle* (UBP) [20] here. Specifically, in our context, UBP tells us that we should carry out EOE or QOE on edges or queries that have a larger upper bound first. For example, if EOE is used, and if there is a bridging edge e_1, with benefit bounds $[0, 30]$, and another bridging edge e_2 with benefit bounds $[0, 15]$, carrying out EOE on e_2 first requires carrying out EOE on e_1 later, because e_1's benefit is possibly better than e_2's. Following UBP, PA is designed to choose the (unconsidered) edge/query with the highest upper benefit bounds (Lines 7/15).

If QOE is chosen, PA requires $2|Q|$ Dijkstra's executions in the worst case; if EOE is chosen, PA requires $2+2|P|$ Dijkstra's executions in the worst case. Com-

pared with the algorithm that we present next, the performance of PA slightly depends on the estimation accuracy but it has a better worst case performance (in terms of number of Dijkstra's executions). PA's slight dependence on estimation accuracy is due to the fact that as long as PA is able to correctly determine whether EOE or QOE is more efficient, how accurate the cost estimation was is not critical. Nonetheless, our experiments show that the two proposed algorithms are nearly indistinguishable (because based on the input parameters, both algorithms converge to the same execution plan) in performance and both are orders of magnitude better than basic solutions.

4.3.3.2 Adaptive Algorithm (AA)

Our second proposed algorithm, Adaptive Algorithm, dynamically switches between using EOE and QOE based on the updated benefit bounds after each iteration. See Algorithm 7 for AA's pseudocode. Like PA, the search space is first reduced with `SearchSpaceReduction` (Line 2). Then, the benefit bounds of the bridging edges and queries are initialized (Line 3). For each iteration (Lines 4 to 16), either EOE or QOE is used to calculate exact benefit values of a bridging edge with respect to all queries (a column) or the benefit of all bridging edges with respect to a query q_j (a row). The decision of choosing whether EOE or QOE is used depends on the benefit bounds of a bridging edge (B^{e_i}) and a query (B_{q_j}) at run-time. If the upper bond principle (UBP) is used here, for each iteration, EOE is carried out on e_i if its upper bound benefit UB^{e_i} is the highest, otherwise QOE is carried out on q_j if its upper bound benefit UB_{q_j} is the highest. After an iteration, either a column (if EOE is used) or a row (if QOE is used) of entries in the search space is filled up and the benefit bounds of

all B_{q_j} (if EOE is used) or B^{e_i} (if QOE is used) are updated; iteration continues until the best vertex-pair is identified.

Algorithm 7 Adaptive Algorithm (AA)

```
1: function AA(G, Q, P)
2:     SearchSpaceReduction(Q, P)                          ▷ Chapter 4.1
3:     InitialBounds(Q, P)                                 ▷ Chapter 4.3.1
4:     while θ < maxUB do
5:         qUB = max(UB_{q_1}, ... UB_{q_|Q|})
6:         eUB = max(UB^{e_1}, ... UB^{e_|P|})
7:         if qUB ≤ eUB then                               ▷ Use EOE
8:             e = arg max(UB^{e_1}, ... UB^{e_|P|})
9:             EOE(e, Q)                                   ▷ Algorithm 3
10:        else                                            ▷ Use QOE
11:            q = arg max(UB_{q_1}, ... UB_{q_|Q|})
12:            QOE(q, P)                                   ▷ Algorithm 4
13:        end if
14:        maxUB = max(UB^{e_1}, ..., UB^{e_|P|})
15:        θ = max(LB^{e_1}, ..., LB^{e_|P|})
16:    end while
17:    return highest benefit vertex-pair
18: end function
```

Note that in AA, there are options other than UBP for choosing bridging edges or queries. To illustrate, using Figure 4.2 assume that QOE is chosen and carried out on q_1 and q_2 in the first two iterations; further assume that after the two iterations, $B_{q_1} = 36$ and $B_{q_2} = 61$, and the benefit bounds of bridging edges are tightened to: $B^{e_1} = [25, 30]$, $B^{e_2} = [15, 15]$, $B^{e_3} = [21, 24]$, $B^{e_4} = [9, 9]$, and $B^{e_5} = [27, 34]$. In the third iteration, if UBP is used, EOE should be carried out on e_5 next because e_5's upper bound benefit, which is 34, is the highest. However, if the lower bound benefits are non-zero, we can consider other options such as choosing the one with the *largest bound difference* (LBD), hoping that an EOE or QOE on that edge or query could reduce the uncertainty the most. In

the example, if LBD is used, then QOE on q_3 (with the largest bound difference of 14) is carried out instead of an EOE on e_5 (with a bound difference of 7). Note that LBD is not applicable to PA because the lower benefit bounds of an edge/query is always 0 or same as its upper benefit bound in PA.

AA requires at most $2\big(|Q| + |P| - 1\big)$ Dijkstra's executions, which is higher than PA in the worst case. Nonetheless, AA is not subject to any estimation error, and its decisions are made based on richer information (bounds are updated at every iteration). So, we regard AA as a competitive alternative to PA worth considering.

Chapter 5

"Which k-Edge-Add-SP-Min" Queries

The previous chapter focused on answering "Which Edge-Add-SP-Min" queries, which find the best vertex-pair in P. In some applications, we may want to find the best k (where $k > 1$) vertex-pairs in P, resulting in "Which k-Edges-Add-SP-Min" queries. In this case, we need an efficient way to determine the k bridging edges that have the maximum total benefit B_k^* on workload Q. Note that the

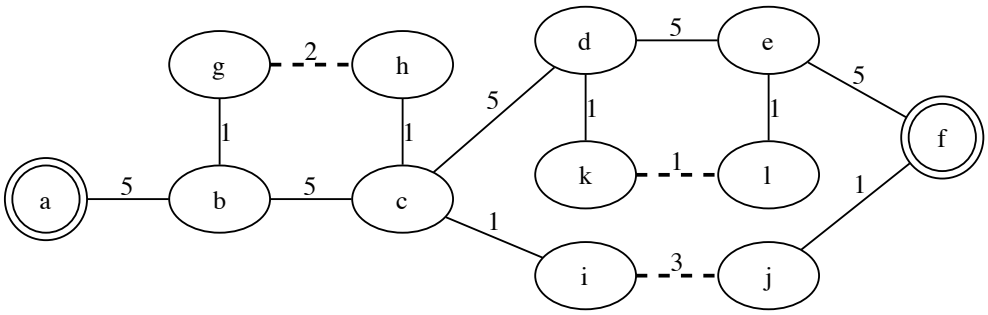


Figure 5.1. Which k-Edge-Add-SP-Min Query

search space of this type of "which" query consists of $|Q| \times \binom{|P|}{k}$ entries, which is exponential to k. We propose two heuristic algorithms that return approximate solutions.

The first heuristic algorithm (HA) is very straightforward—it returns the top-k vertex-pairs found by AA (or PA). Thus, we refer to it as TopK-HA. As a heuristic algorithm, there are cases where TopK-HA cannot return optimal solutions. Consider the example in Figure 5.1 with one shortest path query $q(a, f)$. The individual benefits of bridging edges (i, j), (k, l), and (g, h) on G are 10, 2, and 1 respectively (e.g., running q on $G^{\{(k,l)\}}$ has a distance reduction of 2). When $k = 2$, TopK-HA returns bridging edges (i, j) and (k, l) as the answer. However, the total benefit of (i, j) and (k, l) on G (i.e., running q on $G^{\{(i,j),(k,l)\}}$) is the same as the benefit of the top-1 bridging edge (i, j) (i.e., a benefit of $25 - 15 = 10$). The optimal solution, however, are bridging edges (i, j) (rank 1) and (g, h) (rank 3), whose total combined benefit is $25 - 14 = 11$. Pseudocode for TopK-HA can be found in Algorithm 8.

The second heuristic algorithm tries to avoid the above situation by using greedy heuristics. This algorithm, referred to as Greedy-HA, invokes AA (or PA) in k rounds. In each round, the best vertex-pair from the previous round is added to G. The example in Figure 5.1, Greedy-HA is able to find the optimal solution: first, it uses AA to find the best vertex-pair, (i, j) in G; then, i and j are connected by adding their bridging edge to G. In the second round, Greedy-HA uses AA to find the next best vertex-pair in $G^{\{(i,j)\}}$, which is (g, h). Pseudocode for Greedy-HA can be found in Algorithm 9.

Algorithm 8 TopK-HA

1: **function** TopK-HA(G, Q, P, k)
2: `SearchSpaceReduction`(Q, P) ▷ Chapter 4.1
3: `InitialBounds`(Q, P) ▷ Chapter 4.3.1
4: **while** $\theta < kmaxUB$ **do**
5: $qUB = \max(UB_{q_1}, \ldots UB_{q_{|Q|}})$
6: $eUB = \max(UB^{e_1}, \ldots UB^{e_{|P|}})$
7: **if** $qUB \leq eUB$ **then** ▷ Use EOE
8: $e = \arg\max(UB^{e_1}, \ldots UB^{e_{|P|}})$
9: `EOE`(e, Q) ▷ Algorithm 3
10: **else** ▷ Use QOE
11: $q = \arg\max(UB_{q_1}, \ldots UB_{q_{|Q|}})$
12: `QOE`(q, P) ▷ Algorithm 4
13: **end if**
14: $kmaxUB = k$-th largest UB^{e_i}
15: $\theta = k$-th largest LB^{e_i}
16: **end while**
17: **return** k highest benefit vertex-pairs
18: **end function**

Algorithm 9 Greedy-HA

1: **function** Greedy-HA(G, Q, P, k)
2: **while** $k > 0$ **do**
3: $e =$ `AA`(G, Q, P) ▷ Algorithm 7
4: $G = G + e$
5: $P = P$ - e
6: Update shortest path distances of Q
7: $k = k$ - 1
8: **end while**
9: **return** k highest benefit vertex-pairs
10: **end function**

In our experiments, both Greedy-HA and TopK-HA run in less than one minute with Greedy-HA being k times slower than TopK-HA but yielding very high quality solutions. From our experiments, we can see that for sampled edges, Greedy-HA find the optimal solution for all tested values of k, $k = 2$ to $k = 4$. For using synthetic edges, Greedy-HA finds solutions that approach the optimal solution as the value of k increases from $k = 2$ to $k = 4$. As part of our future work, we plan to derive the approximation ratio of both heuristic algorithms.

We note that there are other techniques for evaluating the best combination of k edges can be evaluated as well. These other techniques include hill climbing, simulated annealing, genetic algorithms, etc. Similar to the approximation ratio for the heuristic algorithms of TopK-HA and Greedy-HA, we leave the exploration and evaluation of hill climbing solutions, simulated annealing solutions, and other solutions as future work.

Chapter 6

Experiments

We evaluated our proposed solutions by running experiments on five real graphs (Table 6.1) of different sizes and types. Our real graph data sets can be obtained from the following locations. For the Argentina and San Francisco road networks, please visit http://www.maproom.psu.edu/dcw/. For the Facebook data set, please visit http://socialnetworks.mpi-sws.mpg.de/data/facebook-links.txt.gz. For the CAIDA data set, please visit http://www.caida.org/tools/measurement/skitter/router_topology/. And for the WebGraph data set, please visit http://law.dsi.unimi.it/webdata/eu-2005/.

Graph Name	Graph Type	Vertex Count	Edge Count	Average Fanout
Argentina	Road Network	85,287	88,357	2.072
San Francisco	Road Network	174,956	223,001	2.536
Facebook	Small World	63,731	817,090	25.642
CAIDA	Router Topology	190,914	607,609	6.365
WebGraph	World Wide Web	862,664	19,235,140	22.297

Table 6.1. Real Graph Data Set Properties

The query workloads were generated by randomly selecting source and destination vertices in the graph as queries. We used two methods to generate the input vertex-pair set P. The first is a synthetic method—non-adjacent vertex-pairs are randomly selected from the graph, and bridging edge lengths are randomly generated and do not exceed the longest edge length in the original graph. The second is a sampling method—bridging edges are randomly sampled from the real edges of the original graph, and the input graph is the reduced graph without the sampled edges. The synthetic method has the advantage that the size and property of the input graph are preserved, but the bridging edge lengths are synthesized. The sampling method has the advantage that the lengths of bridging edges are real, but the input graph have fewer edges than the original real graph. We have carried out experiments using both methods. All experiments were run on a 2.5 GHz Intel PC running Ubuntu with 8 GB of RAM.

6.1 Which Edge-Add-SP-Min Queries

We evaluated our proposed algorithms, ISPI, PA, and AA, against the two baseline solutions mentioned in the beginning of Chapter 4:

1. the brute-force solution (BF), and
2. the solution that calculates benefits by first computing the all-pairs-shortest-paths (APSP).

We implemented all algorithms in C++. APSP ran out of memory in all experiments; thus, we have omitted it from all the experimental results. In the

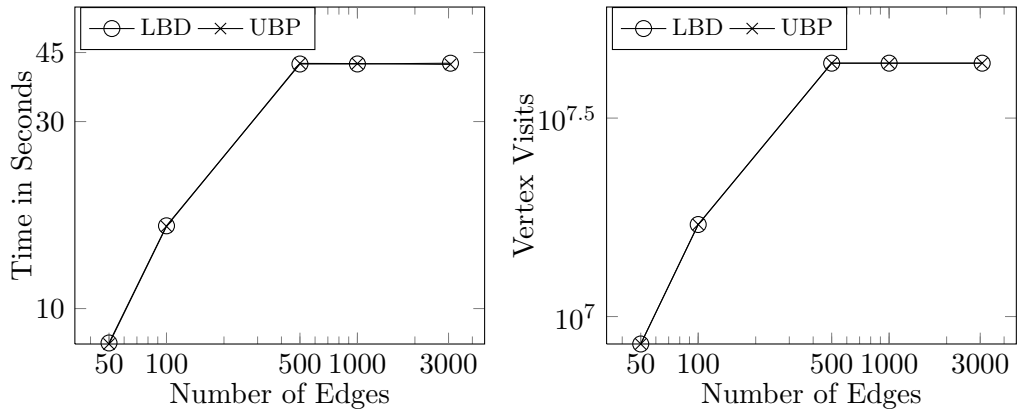
following, we present results that show the performance of the different algorithms under different sizes of input:

1. vertex-pair set P,
2. queries in the workload Q, and
3. different graph sizes.

The default sizes of P and Q are 500.

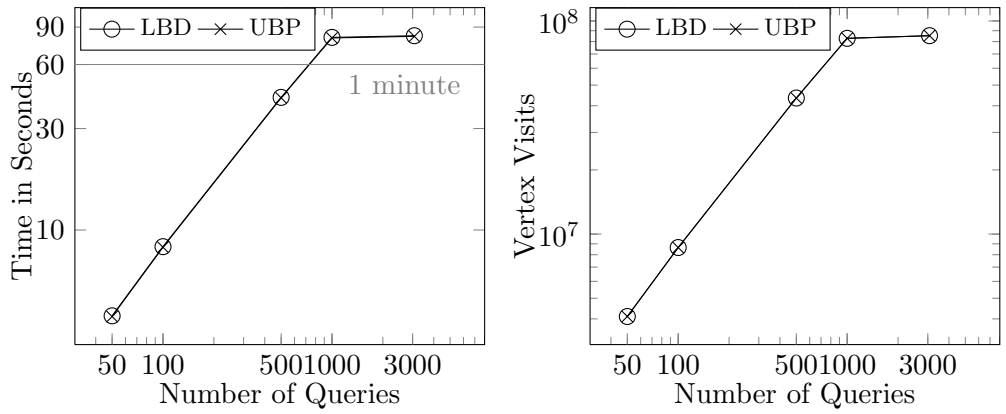
6.1.1 LBD vs UBP in AA

First, we will take a look at how the Largest Bound Difference compares with the Upper Bound Property when used in AA. We have experimented on the Argentina road network using both sampled and synthetic edges. Furthermore, we have also varied the sizes of P and Q. As we can see in Figure 6.1, Figure 6.2, Figure 6.3, and Figure 6.4, the use of LBD and UBP in AA are nearly identical. In fact, UBP has a very slight performance gain in terms of actual running times when compared with LBD because the UBP is simpler to implement.



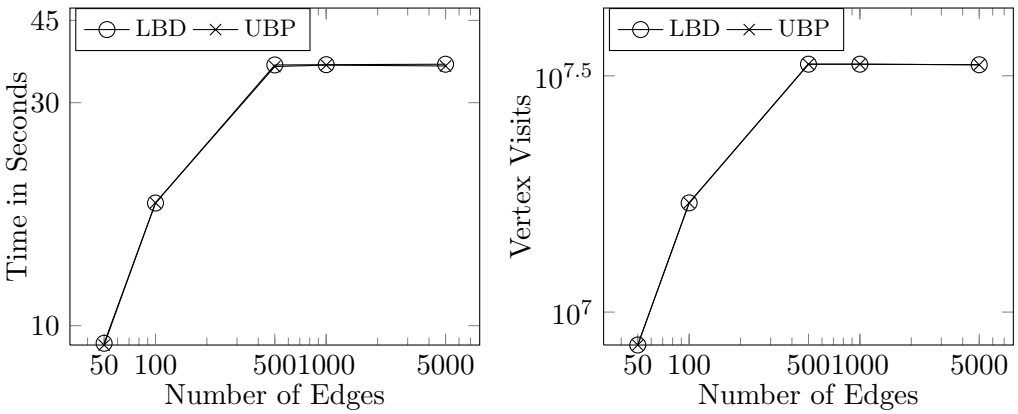
(a) Running time (b) Number of Vertices Visited

Figure 6.1. LBD vs UBP—Varying $|P|$ on the Argentina road network (sampled edges)



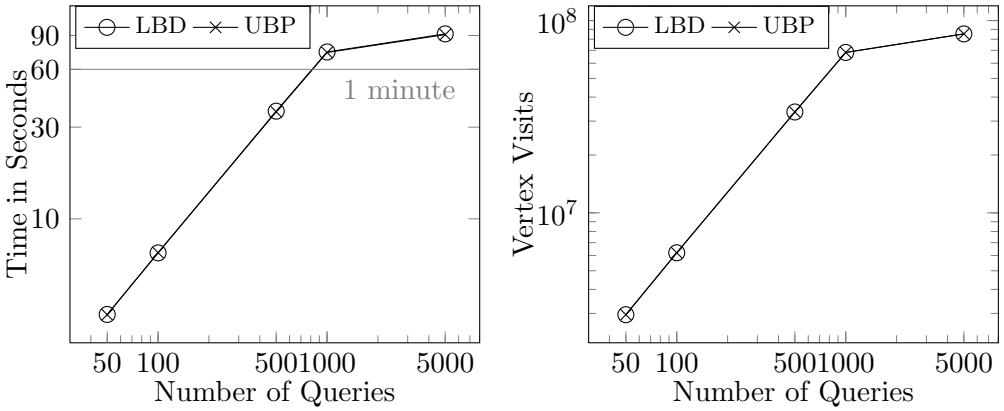
(a) Running time (b) Number of Vertices Visited

Figure 6.2. LBD vs UBP—Varying $|Q|$ on the Argentina road network (sampled edges)



(a) Running time (b) Number of Vertices Visited

Figure 6.3. LBD vs UBP—Varying $|P|$ on the Argentina road network (synthetic edges)



(a) Running time (b) Number of Vertices Visited

Figure 6.4. LBD vs UBP—Varying $|Q|$ on the Argentina road network (synthetic edges)

6.1.2 Results on Argentina Road Network

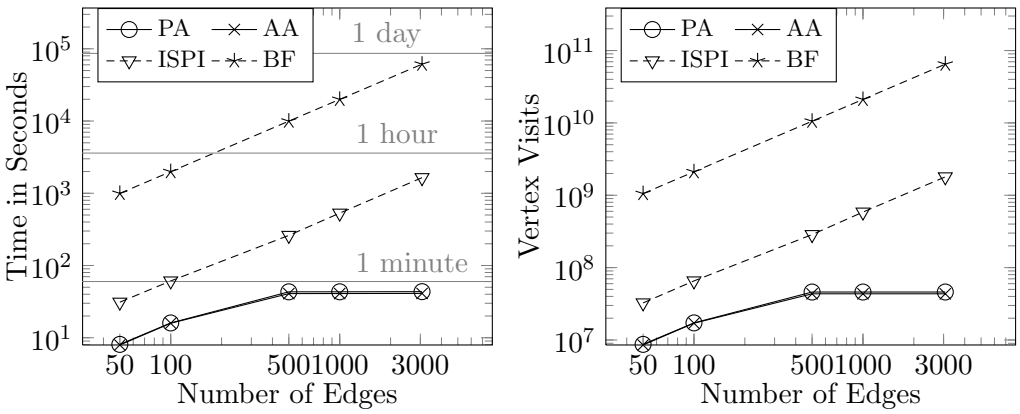
Since our experimental results in Chapter 6.1.1 show that the performance of AA is nearly identical when using UBP and LBD heuristics, the remaining experimental results of AA below use UBP.

6.1.2.1 Sampled Edges

Figure 6.5 shows the running times and number of vertices visited of all methods when varying the size of P on the Argentina Road Network. We can see that ISPI is approximately one and a half magnitudes times faster than BF in running. Furthermore, ISPI also visits approximately one and a half magnitudes less vertices than BF.

PA and AA have comparable performance and they are about two to three orders of magnitude faster than BF and about half to one order of magnitude faster than ISPI. We can see that starting from $|P| = 500$, PA plateaus with a constant running time as the size of P increases to 1000 and 3000. We can see that based on the input parameters, PA accurately estimates when using EOE or QOE is more efficient. Similarly, AA is able to determine that it is more efficient to use QOE when the input size of P exceeds 500.

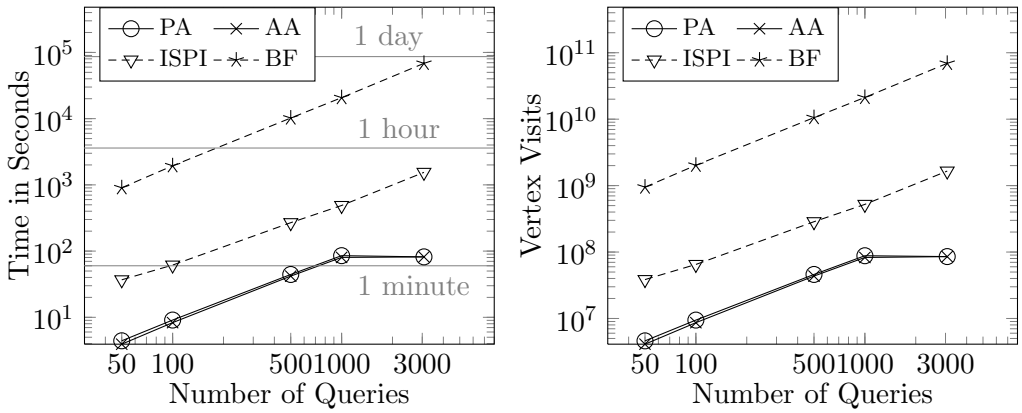
Figure 6.6 shows the running times and number of vertices visited of all methods when varying the size of Q on the Argentina Road Network. Again, PA and AA have comparable performance and they are one to three orders of magnitudes faster than the baseline methods. With the default value of $|P| = 500$, the running times of PA and AA plateau when $|Q| = 1000$. The presence of



(a) Running time (b) Number of Vertices Visited

Figure 6.5. Varying $|P|$ on Argentina Road Network—sampled edges

the plateau is due to PA estimating that when $|Q| < 1000$, using QOE is more efficient on the input, but for $|Q| \geq 1000$, EOE becomes more efficient than QOE. Again, AA is able to come to the same conclusion without the need to estimate the number of vertex visits as PA does.



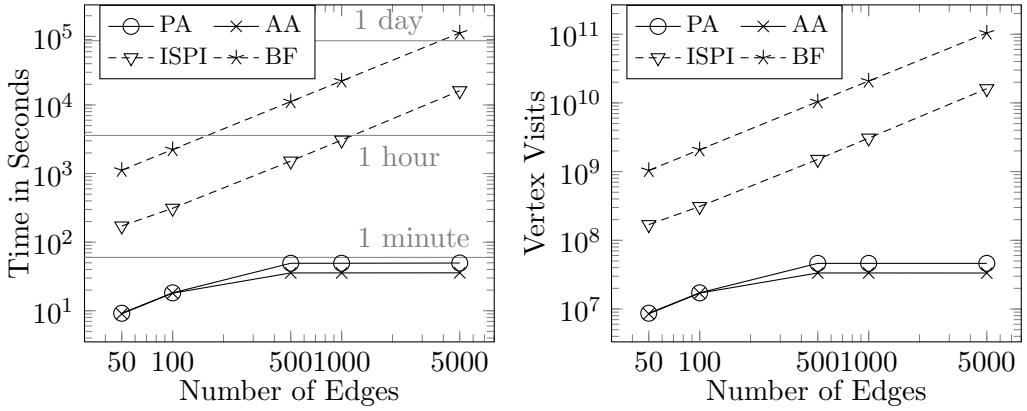
(a) Running time (b) Number of Vertices Visited

Figure 6.6. Varying $|Q|$ on Argentina Road Network—sampled edges

6.1.2.2 Synthetic Edges

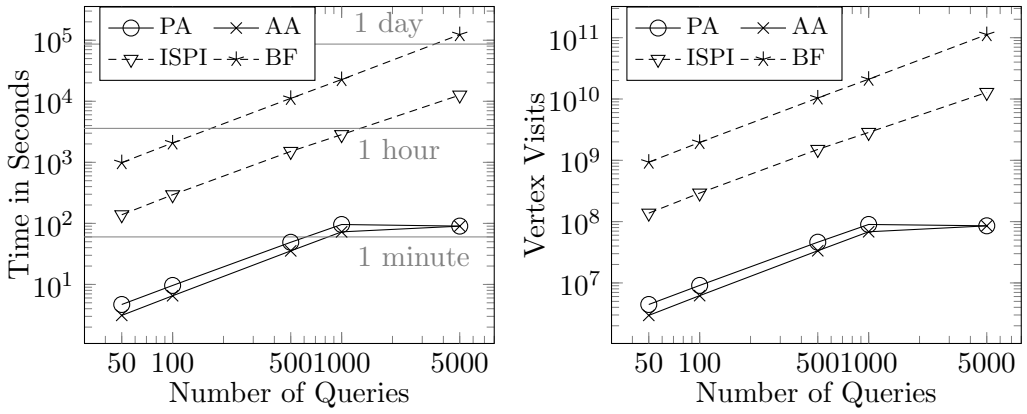
With synthetic edges, we see that ISPI does not perform as well. With sampled edges, ISPI outperforms BF by about 1.5 orders of magnitude; however, with synthetic edges, ISPI is only able to outperform BF by about one order of magnitude. The degraded performance of ISPI on synthetic edges is due to ISPI requiring many more invocations of the SPI algorithm to calculate an updated shortest path distance when compared to using sampled edges. As such, ISPI visits more vertices when synthetic edges are used, which also increases the running time of ISPI.

PA and AA both outperform ISPI and BF by at least one order of magnitude. Similar to the use of sampled edges, PA and AA are both able to determine an efficient strategy for answering the query (the plateau occurring at $|P| = 500$ when varying the size of P, and at $|Q| = 1000$ when varying the size of Q). In Figure 6.7 and Figure 6.8, we can see that AA actually runs faster than PA. This difference is due to AA being able to switch between using EOE and QOE during execution, thus AA is able to visit less vertices, which also effectively reduces the total running time needed.



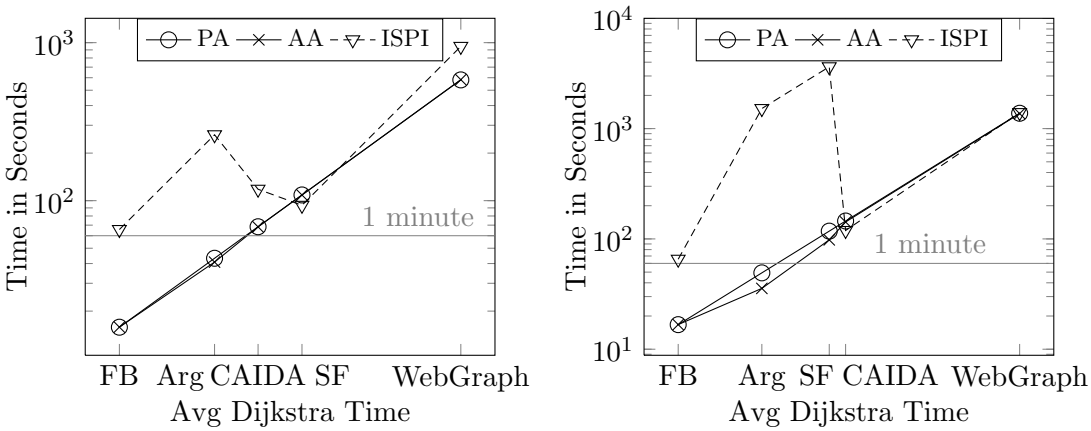
(a) Running time (b) Number of Vertices Visited

Figure 6.7. Varying $|P|$ on Argentina Road Network–synthetic edges



(a) Running time (b) Number of Vertices Visited

Figure 6.8. Varying $|Q|$ on Argentina Road Network–synthetic edges



(a) Sampled Edges (b) Synthetic Edges

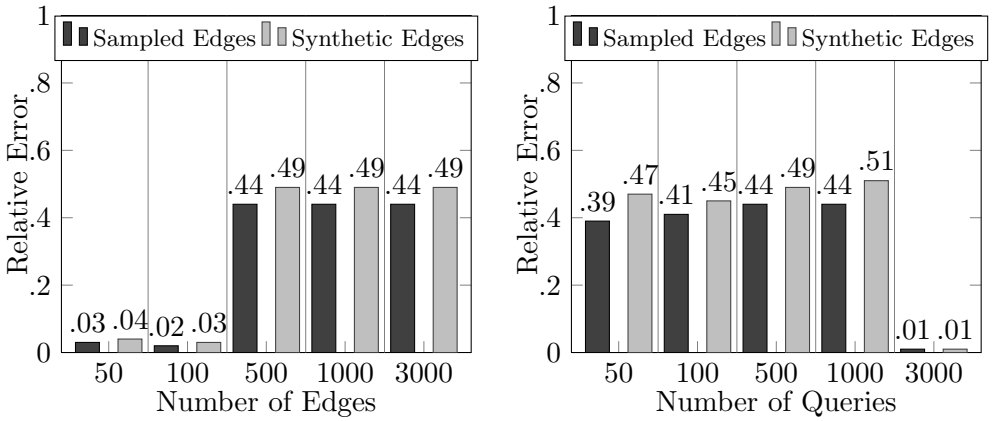
Figure 6.9. Scalability of ISPI, PA, and AA

6.1.3 Scalability of ISPI, PA, and AA

Figure 6.9 shows the scalability of ISPI, PA and AA on all five real data sets. As the graphs are different in sizes and density, we arrange the five data sets on the x-axis based on their average Dijkstra's running times. The results show that PA and AA are scalable to graphs of different sizes and densities. As for ISPI, ISPI does not scale according to graph sizes and densities.

6.1.4 Estimation Quality of PA on Argentina Road Network

In Figure 6.10, we see PA's quality in estimating the worst case number of vertices visited of the selected expansion method (EOE or QOE). Here, the absolute error is measured as the error between the estimated total number of vertices visited versus the actual number of vertices visited. We can see that PA's estimation error remains low with a maximum value of 0.51 and a minimum



(a) Varying $|P|$ (b) Varying $|Q|$

Figure 6.10. Estimation Error of PA

value of 0.01. The estimated number of vertices visited for using EOE is highly dependent on the longest query length, as such, PA's estimation quality is high when EOE is selected to be used (PA selects to use EOE when $|P| = 50$ and 100 in Figure 6.10(a) and $|Q| = 3000$ in Figure 6.10(b)). Although QOE estimations are more variable, because we need to sum up the estimated number of vertices visited for each query, we can see that the estimation error for QOE is still low, ranging from 0.39 to 0.51.

6.2 "Which k-Edge-Add-SP-Min" Queries

To evaluate the efficiency and the solution quality of TopK-HA and Greedy-HA in answering "Which k-Edge-Add-SP-Min" queries, we implemented a brute-force solution (BF-OPT) that exhaustively enumerates all possible combinations of k bridging edges to find the optimal solution. We compare the approximate

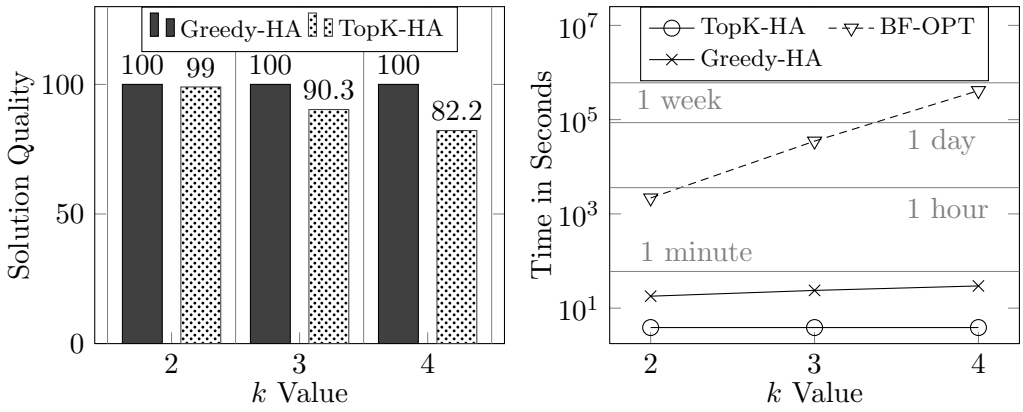
solutions returned by TopK-HA and Greedy-HA with BF-OPT. Since BF-OPT is extremely time-consuming, we performed our experiments on the Argentina road network and varied k from 2 to 4 with $|P| = |Q| = 50$ (BF-OPT was too slow on the other larger graphs).

6.2.1 Sampled Edges

Figure 6.11 shows the quality of our algorithms compared with BF-OPT and the running times of all algorithms. Greedy-HA returns optimal solutions for all three values of k. For $k = 2, 3, 4$, TopK-HA finds a solution that is 99%, 90%, and 82% of the optimal solution. Finally, TopK-HA maintains a constant running time, whereas Greedy-HA runs k times slower than TopK-HA.

6.2.2 Synthetic Edges

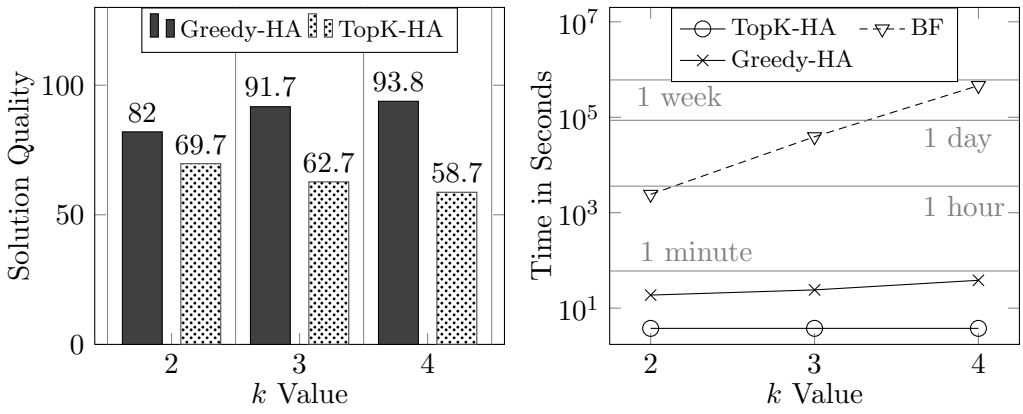
Figure 6.12 shows the results of the experiments on "Which k-Edge-Add-Min-SP" queries. We see that both Greedy-HA and TopK-HA provide answers of at least 58% of the optimal benefit, and in orders of magnitude shorter time. With Greedy-HA, as k increases, we can see that Greedy-HA's benefit increases. This is because Greedy-HA is able to find a large subset of the optimal edges (for $k = 2$, Greedy-HA finds one optimal edge, for $k = 3$, 2 optimal edges are found, and for $k = 4$, 3 optimal edges are found). In terms of running times between TopK-HA and Greedy-HA, again, TopK-HA runs in constant time, and Greedy-HA is only k times slower than TopK-HA.



(a) Solution Quality of TopK-HA and Greedy-HA

(b) Running Times

Figure 6.11. "Which k-Edge-Add-SP-Min" Queries—Sampled Edges



(a) Solution Quality of TopK-HA and Greedy-HA

(b) Running Times

Figure 6.12. "Which k-Edge-Add-SP-Min" Queries—Synthetic Edges

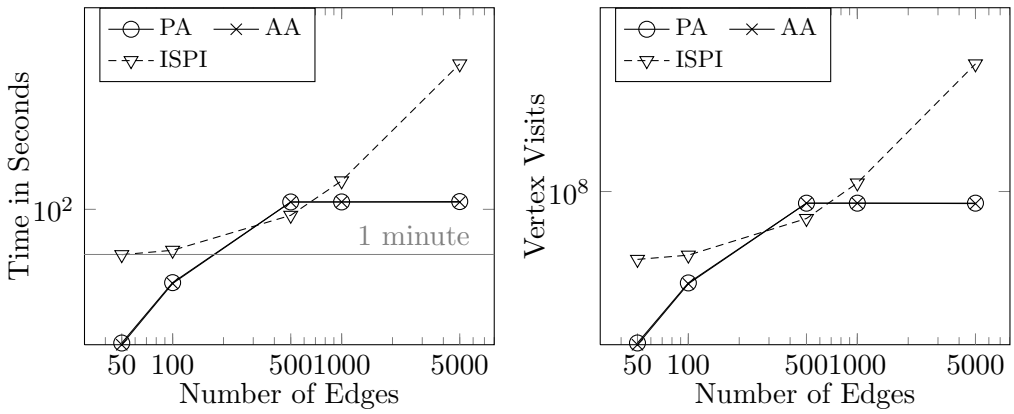
6.3 Experimental Results on Other Data Sets

Here, we present the experimental results (sampled edges and synthetic edges) on the remaining real data sets. We see that usually PA and AA have very similar performance curves and outperform ISPI. Our experiments show that AA generally performs better than PA because AA adaptively chooses between using EOE and QOE. However, occasionally PA performs better than AA because PA is able to estimate that using QOE, even though $|Q| > |P|$, is more efficient.

6.3.1 San Francisco Road Network

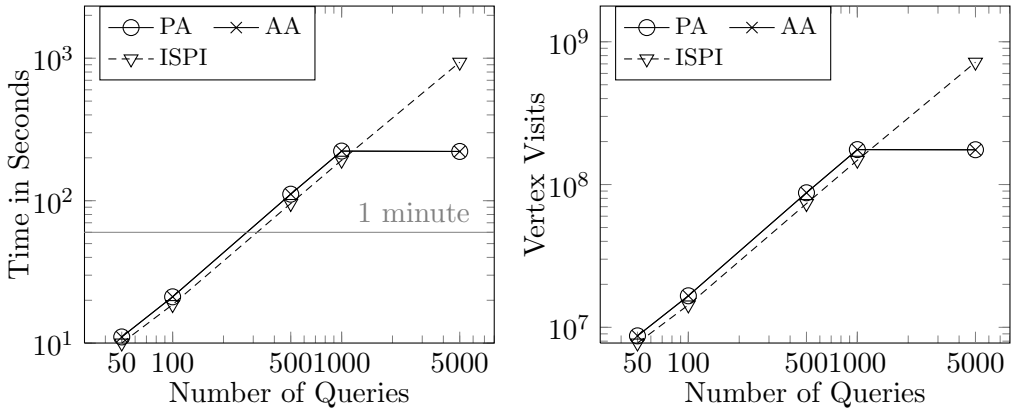
For the San Francisco road network data set, we can see in Figure 6.13 and Figure 6.14 that PA and AA maintain a performance that is very similar to their performance on the Argentina road network data set (Chapter 6.1.2). Furthermore, AA performs as well as, if not better, than PA throughout all of the experiments.

For ISPI, we can see that as we vary the size of $|P|$, using sampled edges, ISPI generally performs worse than PA and AA; however, when we vary the size of $|Q|$, again using sampled edges, ISPI typically outperforms PA and AA. We note again that ISPI is heavily dependent on the number of invocations to SPI. As we can see in Figure 6.13, when the number of SPI invocations is high, then ISPI will tend to perform worse than PA and AA; however, if the number of SPI invocations is low, as in Figure 6.14, then ISPI can perform better than PA and AA, because in the worst case PA and AA require 2 Dijkstra's expansion for a call to QOE or EOE.



(a) Running time while varying $|P|$

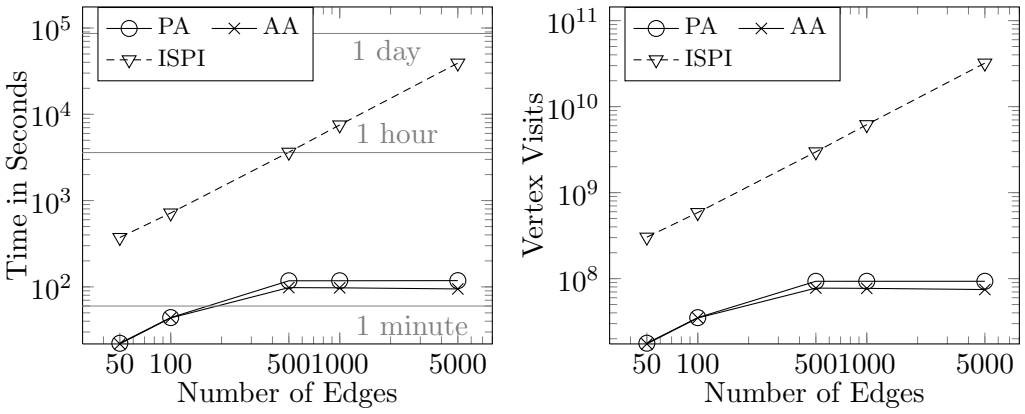
(b) Vertices visited while varying $|P|$



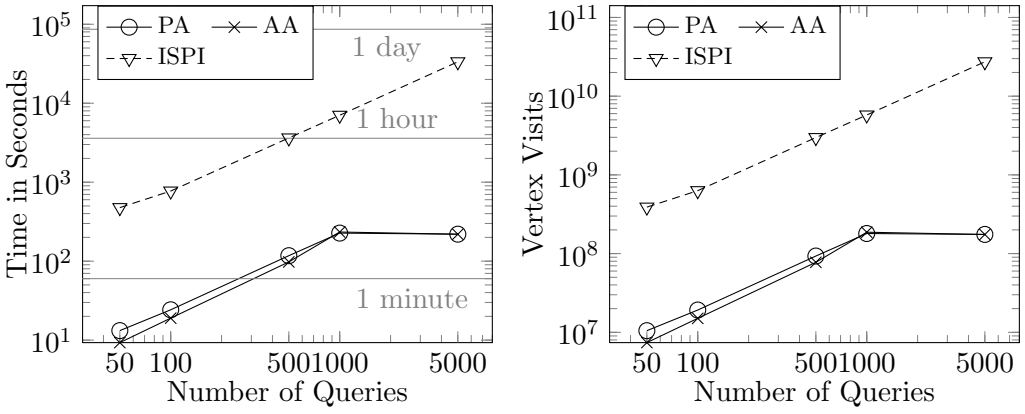
(c) Running time while varying $|Q|$

(d) Vertices visited while varying $|Q|$

Figure 6.13. San Francisco—Sampled Edges



(a) Running time while varying $|P|$ (b) Vertices visited while varying $|P|$



(c) Running time while varying $|Q|$ (d) Vertices visited while varying $|Q|$

Figure 6.14. San Francisco—Synthetic Edges

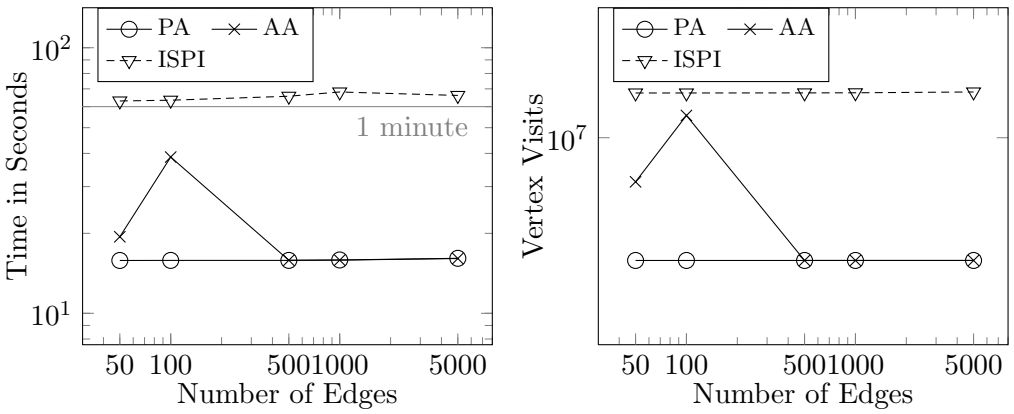
6.3.2 Facebook Social Network

In the Facebook data set, we can see that PA and AA generally outperform ISPI by about half an order of magnitude in running time and numbers of vertices visited. Here, we can see that generally PA and AA have identical performance.

At times, PA actually performs better than AA. This improved performance in PA is due, in part, to the fact that PA is able to estimate whether EOE or QOE is more efficient for the input parameters. Furthermore, because AA does not invest any effort into estimating EOE/QOE efficiency, AA may require extra Dijkstra's expansions when compared with EOE. We note again that AA does have a higher worst case complexity, $2\big(|Q|+|P|-1\big)$ compared with PA's $2+2|P|$.

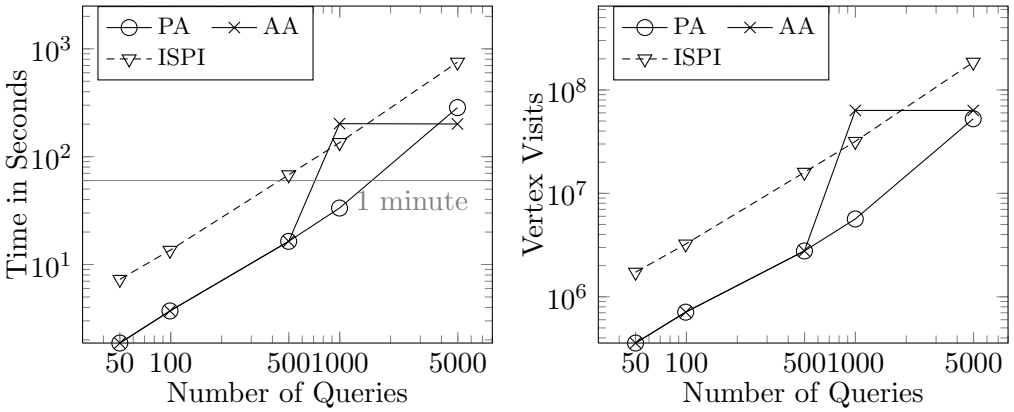
Although PA does outperform AA in certain cases on the Facebook data set, we can see that AA can outperform PA in terms of running time (Figure 6.15(c) and Figure 6.16(c)).

When AA performs worse than PA, we can see that only occasionally will AA also perform worse than ISPI. We can see that ISPI has a performance curve similar to PA on the Facebook data set. This is because PA uses QOE, much in the same way that ISPI calculates all shortest path distance tree for each query. However, we can see that ISPI is slower than PA and also visits more vertices than PA. ISPI is slower than PA because of the required SPI invocations to calculate the benefit values, in addition to the extra bookkeeping needed for ISPI to cache the shortest path distances and restore the original shortest path distances after each SPI invocation.



(a) Running time while varying $|P|$

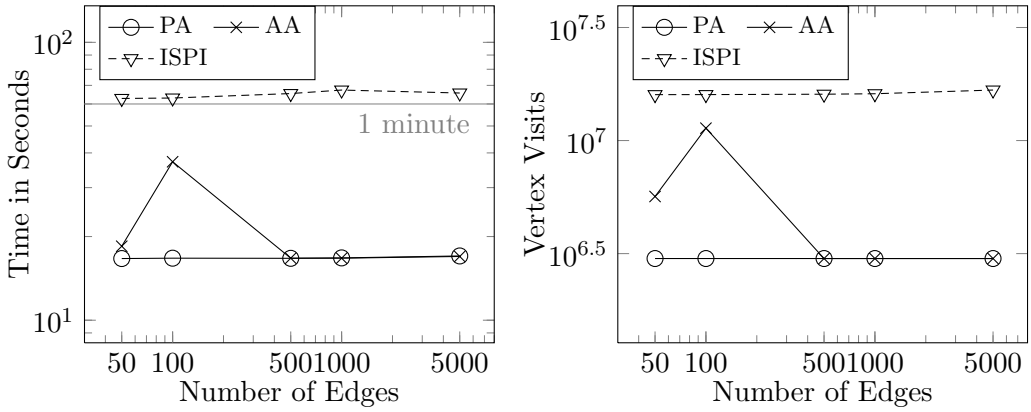
(b) Vertices visited while varying $|P|$



(c) Running time while varying $|Q|$

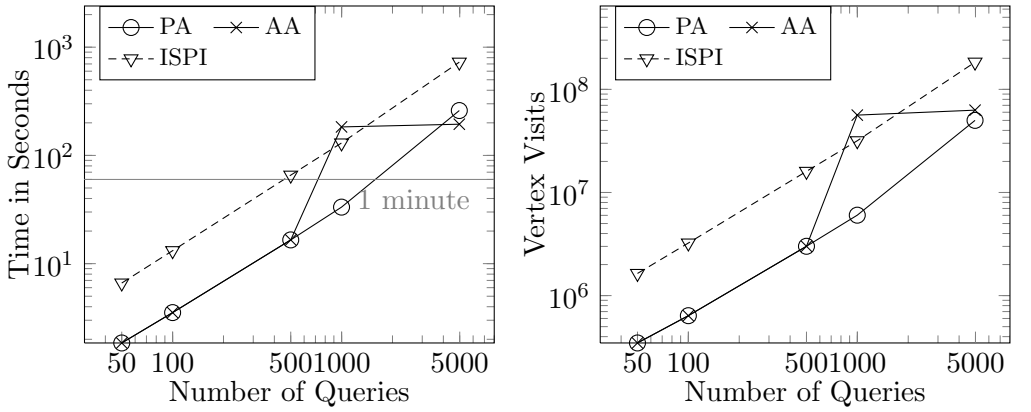
(d) Vertices visited while varying $|Q|$

Figure 6.15. Facebook—Sampled Edges



(a) Running time while varying $|P|$

(b) Vertices visited while varying $|P|$



(c) Running time while varying $|Q|$

(d) Vertices visited while varying $|Q|$

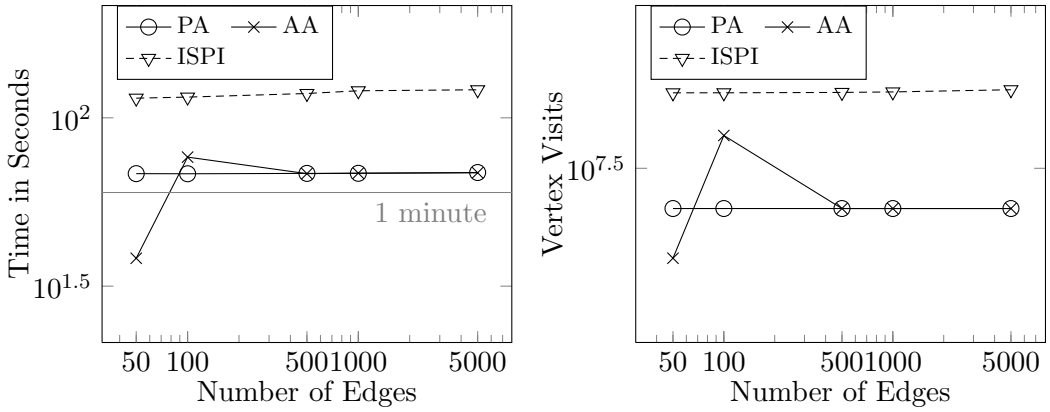
Figure 6.16. Facebook—Synthetic Edges

6.3.3 CAIDA Internet Router Topology

In the CAIDA data set, again, we can see that generally AA outperforms PA, or at least maintains a nearly identical performance to PA. The times that PA outperforms AA is due to AA's estimation step accurately estimating that QOE is more efficient than EOE on the input parameters. When AA outperforms PA, PA has estimated that QOE is more efficient, but AA has adaptively chosen a combination of EOE and QOE that is better than using QOE, as PA has determined.

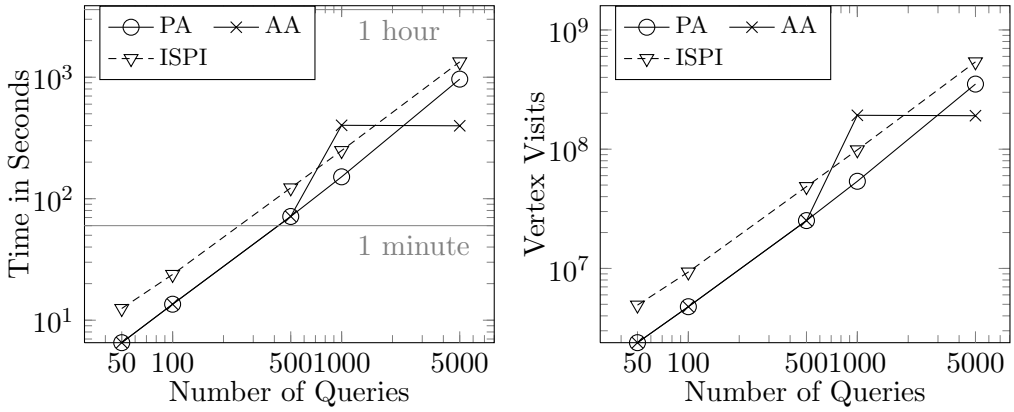
In the experiment varying the size of P using sampled edges, we can see that ISPI is usually worse than PA and AA by approximately half an order of magnitude in both running time and vertices visited. When we vary the size of Q using sampled edges, ISPI performs slightly better. Again, we can see that the extra bookkeeping and the extra vertices visited for SPI invocations in ISPI leads to a performance curve similar to PA, only not as efficient as PA.

In the case of synthetic edges, we can see that ISPI can actually perform better than PA because a majority of the bridging edges in P do not require a SPI invocation. As such, ISPI is able to visit less vertices, and as a result, the bookkeeping of the cached shortest path distances is reduced.



(a) Running time while varying $|P|$

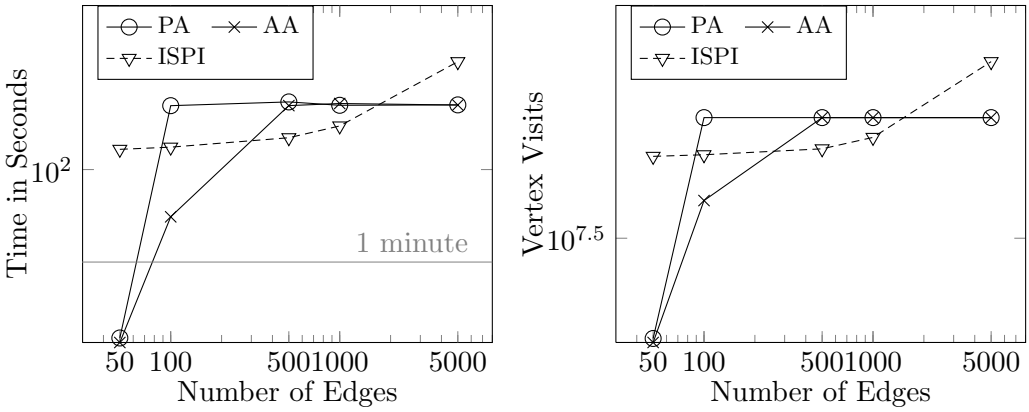
(b) Vertices visited while varying $|P|$



(c) Running time while varying $|Q|$

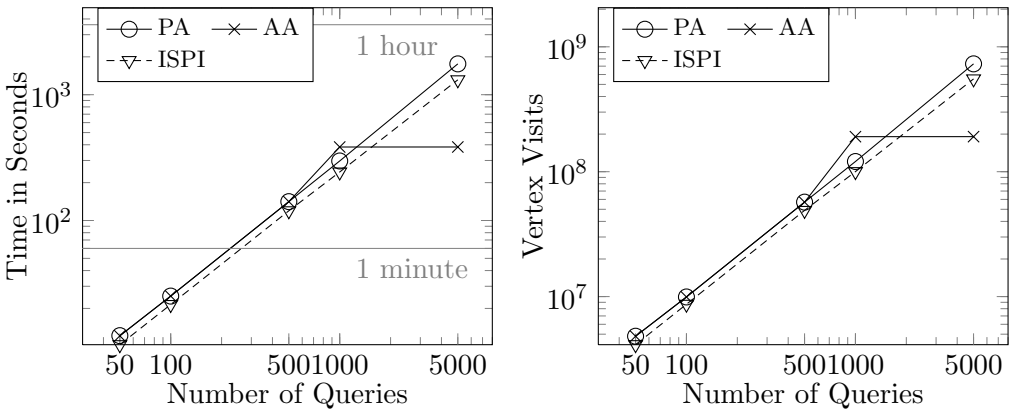
(d) Vertices visited while varying $|Q|$

Figure 6.17. CAIDA—Sampled Edges



(a) Running time while varying $|P|$

(b) Vertices visited while varying $|P|$



(c) Running time while varying $|Q|$

(d) Vertices visited while varying $|Q|$

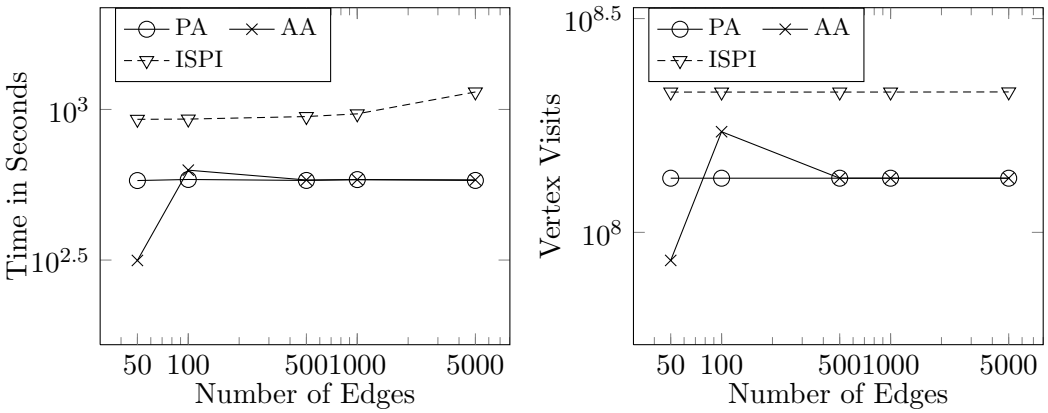
Figure 6.18. CAIDA—Synthetic Edges

6.3.4 WebGraph World Wide Web

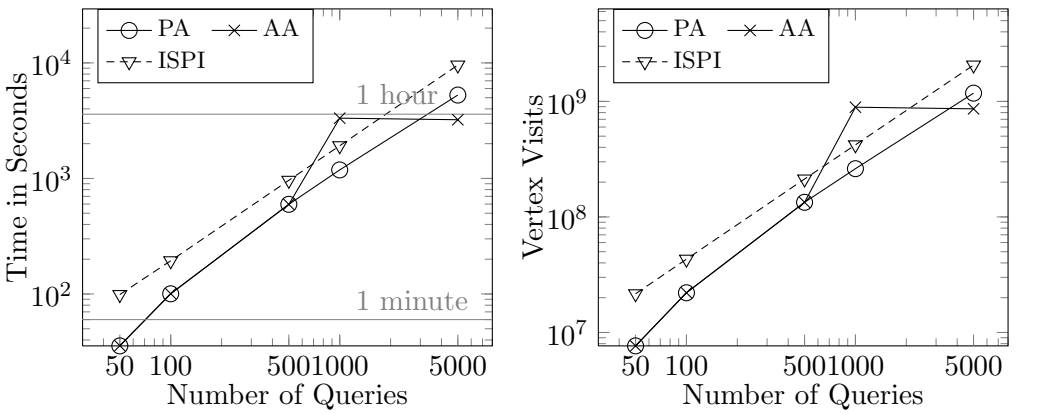
Finally, in the WebGraph data set, we see that PA, AA, and ISPI have similar performance curves as the CAIDA data set. We can see that ISPI is typically outperformed by PA and AA by about half an order of magnitude in running time and numbers of vertices visited on sampled edges. Furthermore, we can see that only very occasionally has AA been outperformed by PA.

When AA outperforms PA, AA is able to adaptively find a combination of EOE and QOE calls that are more efficient than PA selecting to only use QOE. The times that PA performs better than AA are rare, and AA is only marginally slower than PA.

In sampled edges, we see that ISPI has a similar performance curve as PA, but is consistently outperformed by PA because ISPI requires extra bookkeeping for the SPI invocations. Although ISPI is outperformed by PA, it is not significantly worse than PA, only slightly. When we examine the impact of synthetic edges, we can see that ISPI can even outperform PA when the number of SPI invocations are low.

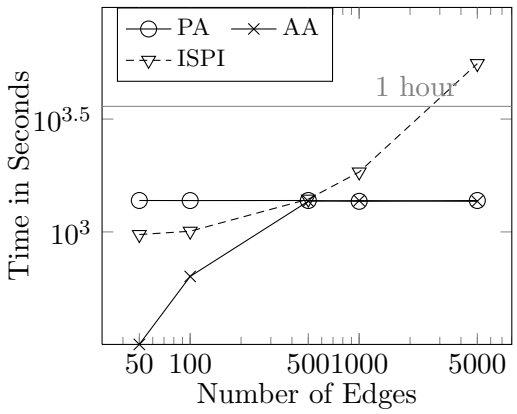


(a) Running time while varying $|P|$ (b) Vertices visited while varying $|P|$

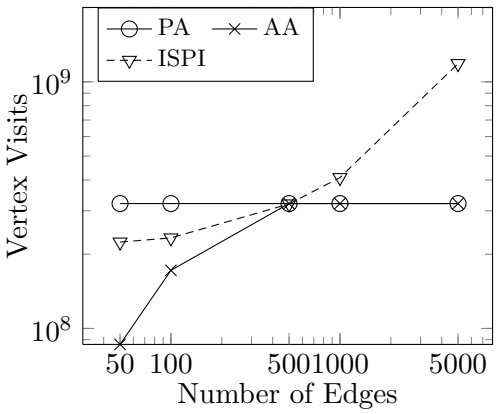


(c) Running time while varying $|Q|$ (d) Vertices visited while varying $|Q|$

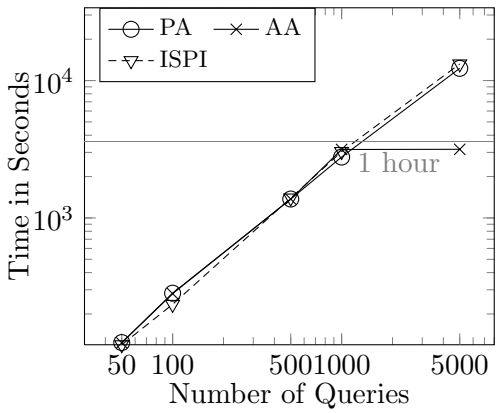
Figure 6.19. WebGraph—Sampled Edges



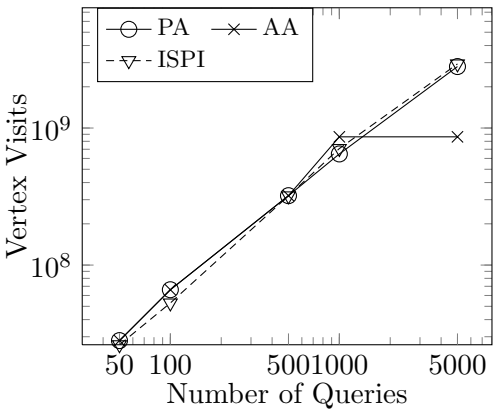
(a) Running time while varying $|P|$



(b) Vertices visited while varying $|P|$



(c) Running time while varying $|Q|$



(d) Vertices visited while varying $|Q|$

Figure 6.20. WebGraph—Synthetic Edges

Chapter 7

San Francisco Bay Area – A Case Study

Here we will present the findings of an intensive case study. In this case study, we have applied our proposed solution to a real life scenario. For this case study, we have used the San Francisco Bay area road network to determine the best locations for constructing a new bridge spanning the San Francisco Bay such that travel distances can be reduced the most.

In this case study, we concern ourselves only with the construction of new bridges spanning the San Francisco Bay. We have identified a number of possible locations for constructing new bridges, which resulted in 348 possible bridges to consider constructing. For all 348 candidate bridges, we have assumed that their construction costs are directly related to the bridge's span length; thus, longer bridges are more costly to construct, but may shorten travel distances more, when compared with shorter bridges.

To determine important travel destinations, we have used San Francisco Bay Area commuter statistics supplied by the Metropolitan Transportation Commission. From the Metropolitan Transportation Commission, we have mapped county to county commuter statistics back onto the San Francisco Bay Area road network graph. After remapping the county to county commuter statistics, we identified 377 queries to characterize the commuter data supplied from the Metropolitan Transportation Commission. For each of these 377 commuter queries, their relative importance to San Francisco Bay Area residents were computed from census data, which has also been collected by the Metropolitan Transportation Commission. The commuter census data can be found at `http://www.mtc.ca.gov/maps_and_data/datamart/census/county2county/table1coco.htm`.

In Figure 7.1, we can see the original San Francisco Bay Area road network that we have performed our case study on. Given the county to county commuter data that has been transformed into 377 queries, our search space reduction techniques (Chapter 4.1) are able to reduce search space by approximately 10%.

Using Greedy-HA, the most beneficial bridges to add into the San Francisco Bay Area road network are as follows. In Figure 7.2, the single most beneficial bridge is shown just south of the Dumbarton Bridge. This bridge would help commuters traveling between the extreme ends of the Silicon Valley at the southern end of Alameda County and the souther end of San Mateo County. Figure 7.3 shows the second most beneficial bridge to construct. This bridge connects south San Francisco to the mid to southern end of Alameda County. The third most beneficial bridge, as seen in Figure 7.4, connects central San Francisco to central Alameda County. Finally, the fourth most beneficial bridge to construct connects Alameda County to San Mateo County (Figure 7.5).

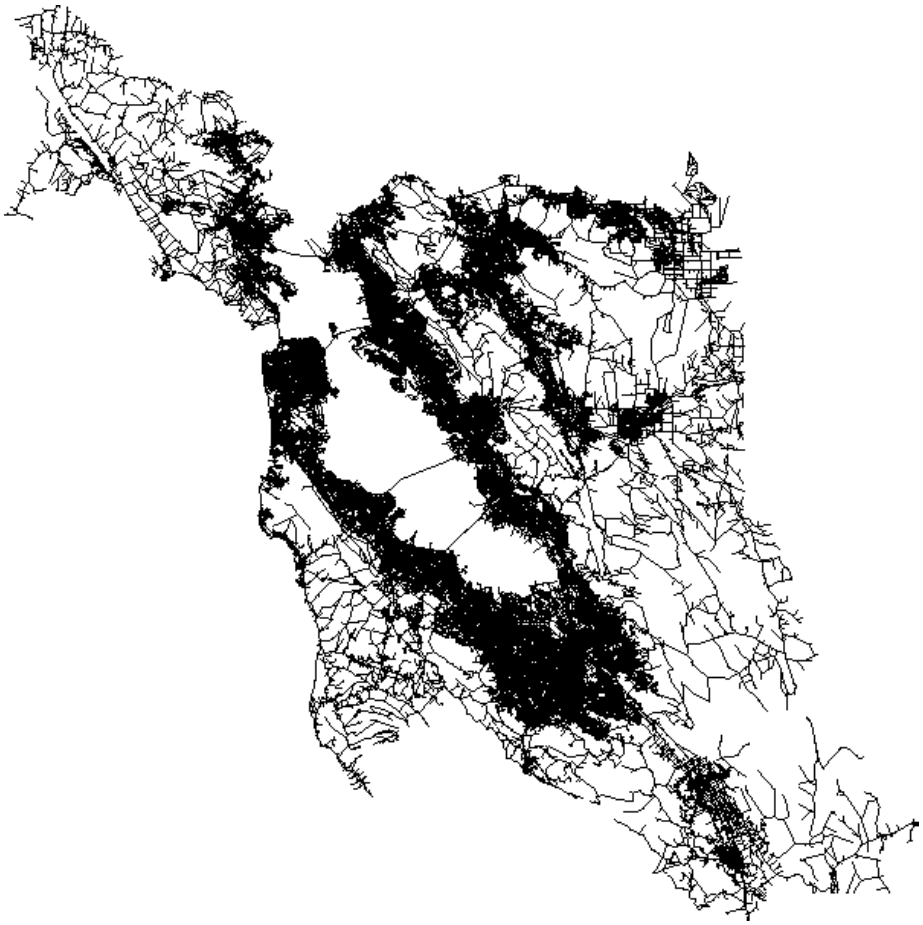


Figure 7.1. San Francisco Bay Area Road Network

In Figure 7.6, we can see how the San Francisco Bay Area road network would be changed if all four bridges were constructed, as calculated using Greedy-HA. We can see that TopK-HA selects to construct 3 bridges all south of the Dumbarton Bridge in close proximity to each other (Figure 7.7). As we can see, it makes more sense to construct the bridges suggested by Greedy-HA, even though Greedy-HA requires more time to determine the bridges to construct. Figure 7.8 shows the running times of the Greedy-HA and TopK-HA algorithms on for this case study. We can see that TopK-HA has a constant execution time,

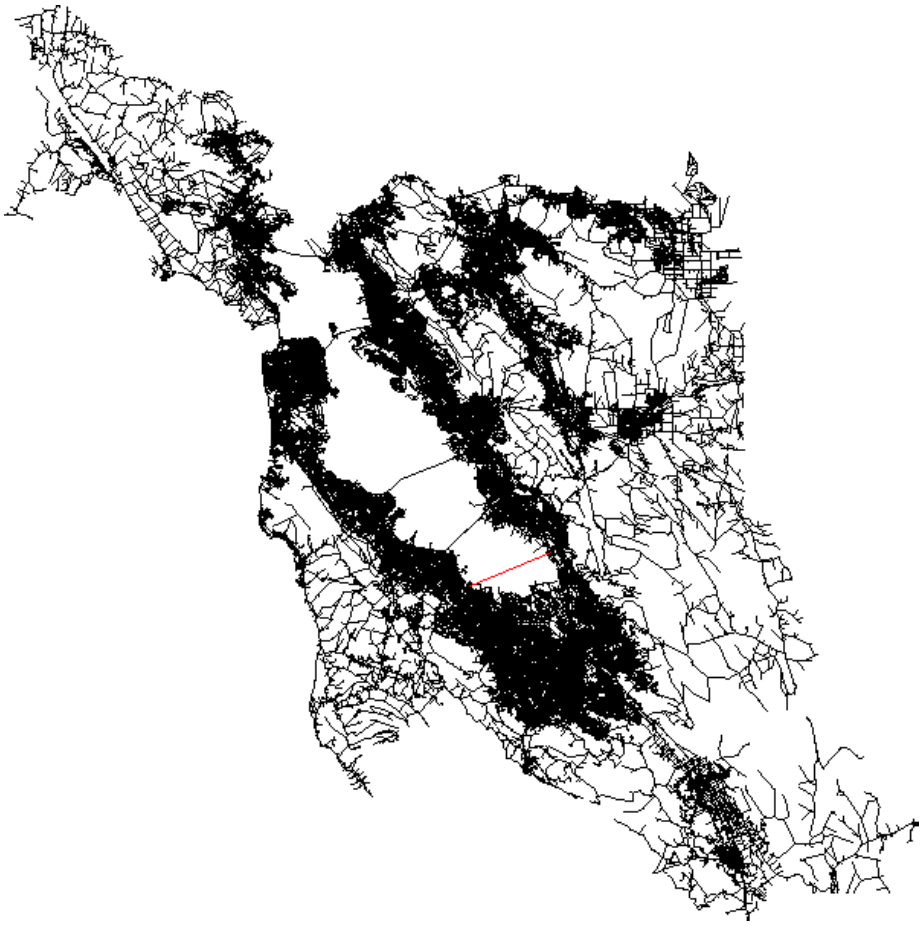


Figure 7.2. Most beneficial bridge location.

whereas Greedy-HA has a running tim that is linear to k. Even though Greedy-HA requires more time to calculate the best 4 bridges to construct, the running time of Greedy-HA is still highly reasonable. Furthermore, as we noted above, the bridges that Greedy-HA suggest to construct are much more useful than the bridges suggested by TopK-HA.

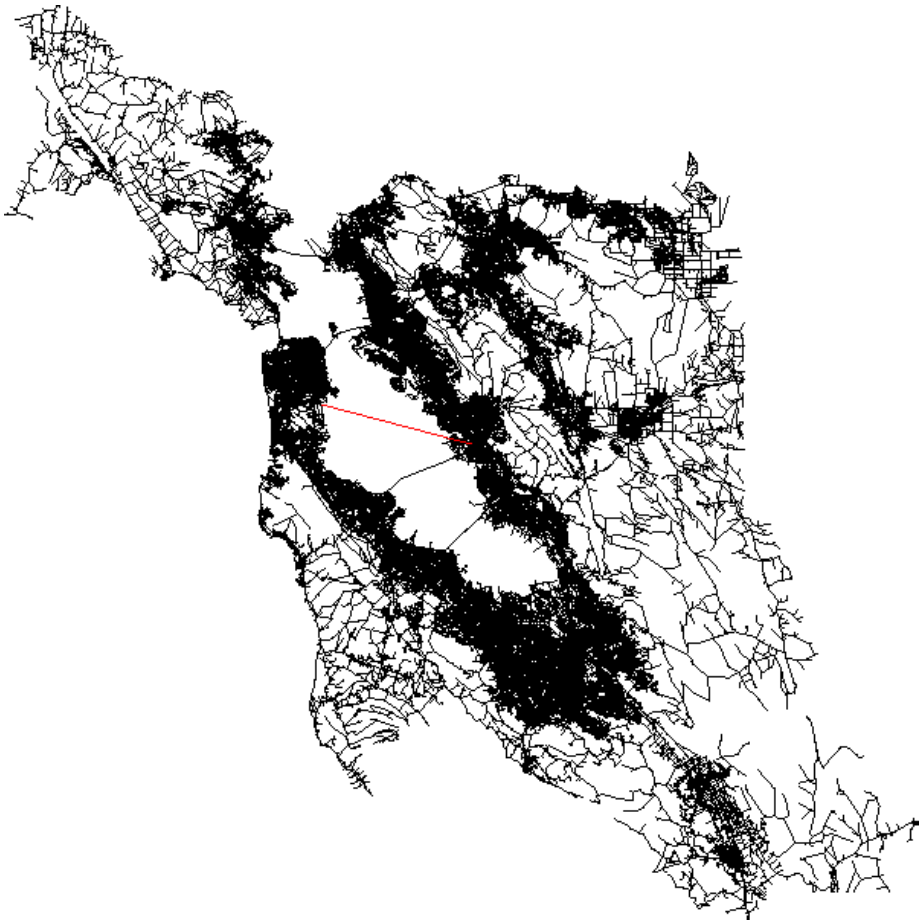


Figure 7.3. Second most beneficial calculated bridge location.

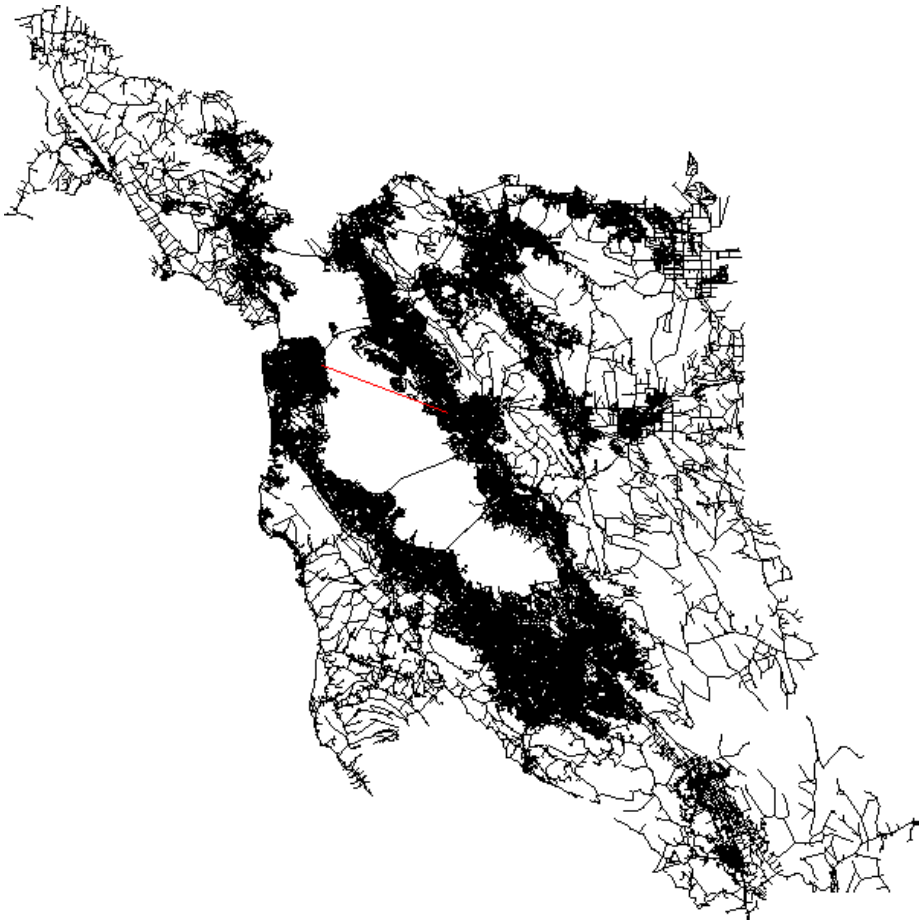


Figure 7.4. Third most beneficial calculated bridge location.

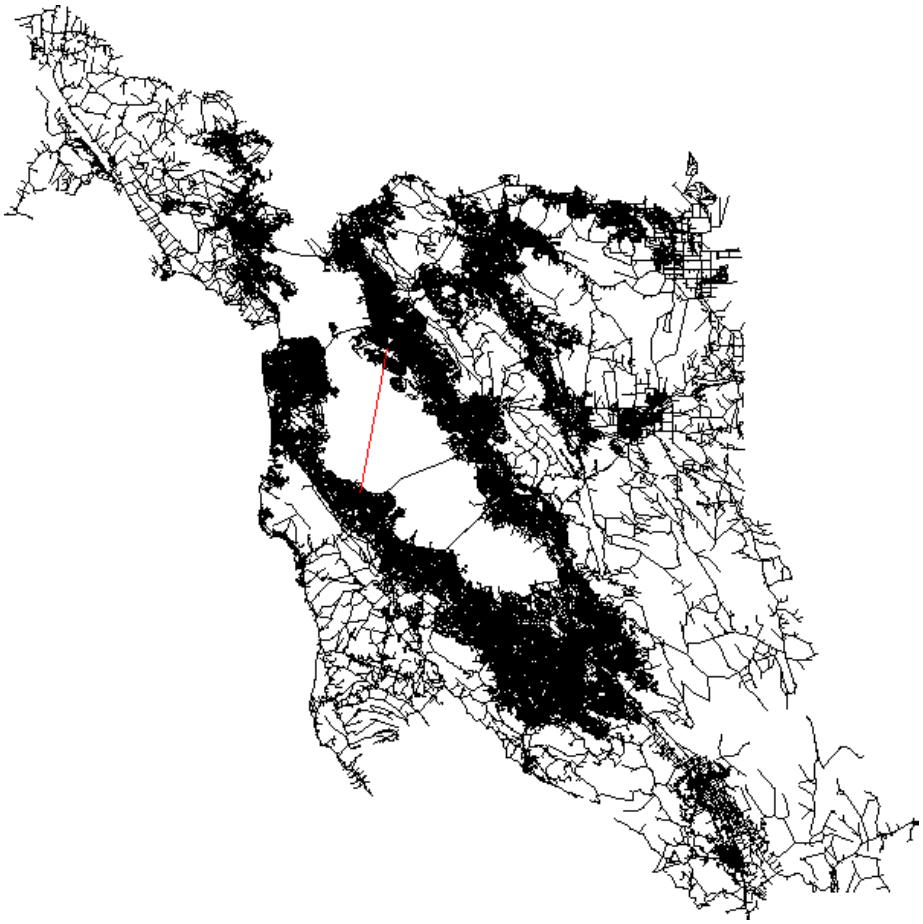


Figure 7.5. Fourth most beneficial calculated bridge location.

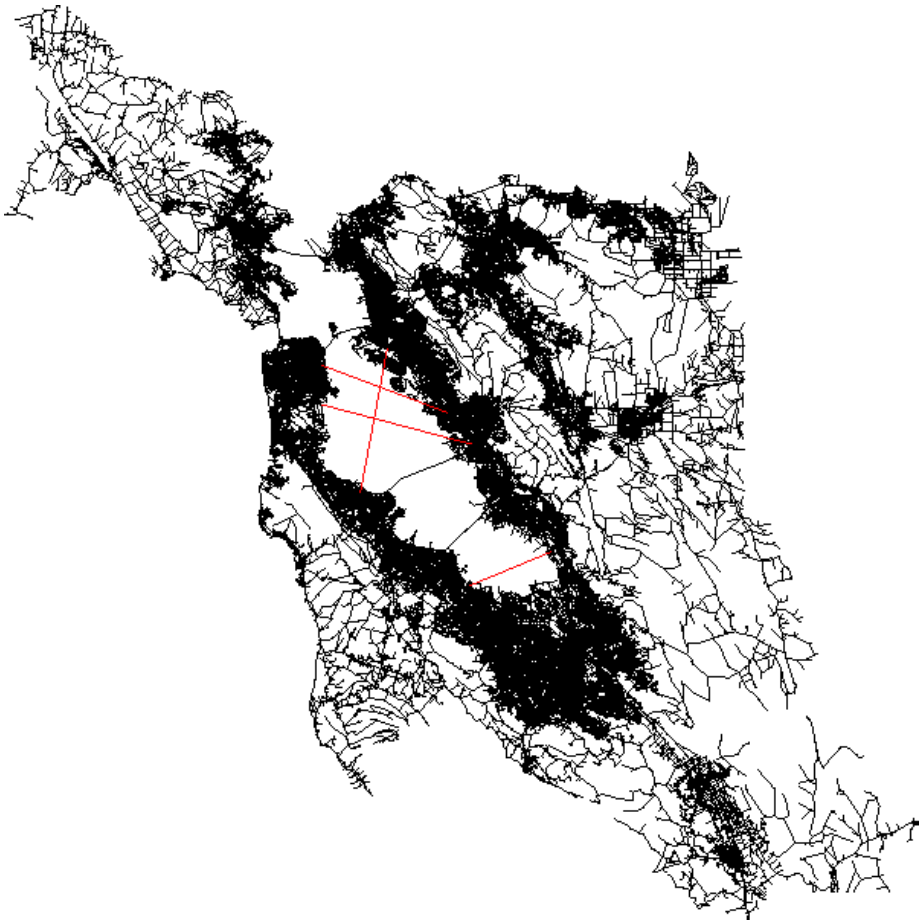


Figure 7.6. Top-4 calculated bridge locations.

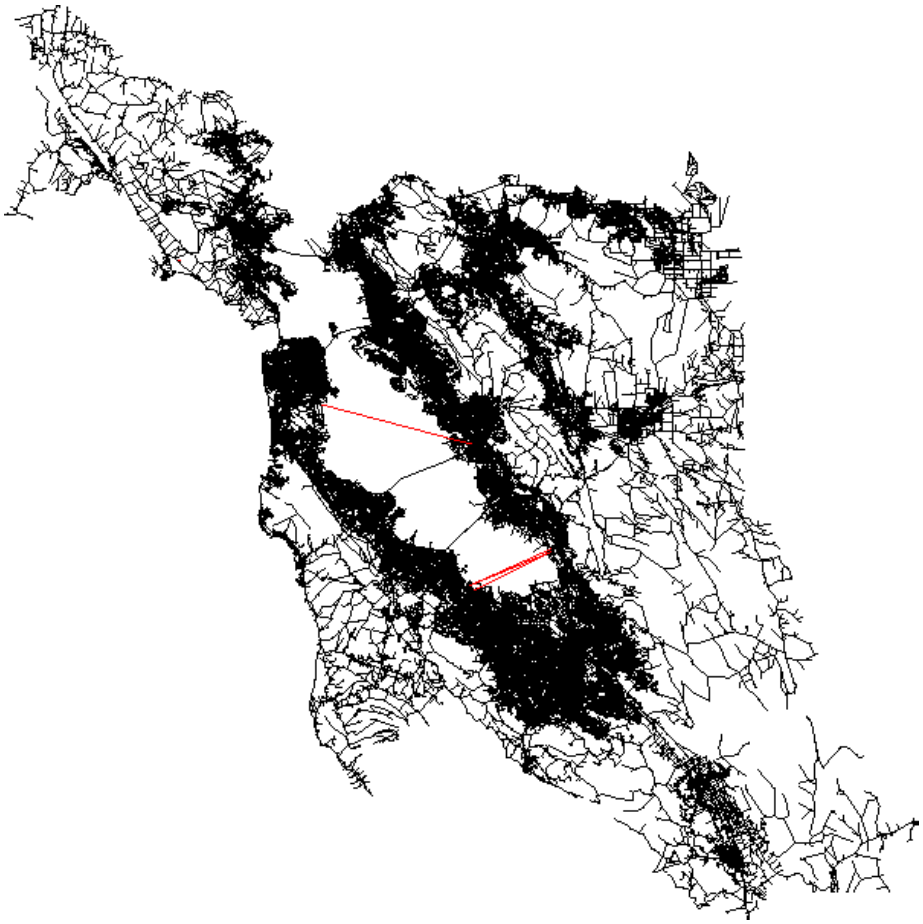


Figure 7.7. Top-4 calculated bridge locations using TopK-HA.

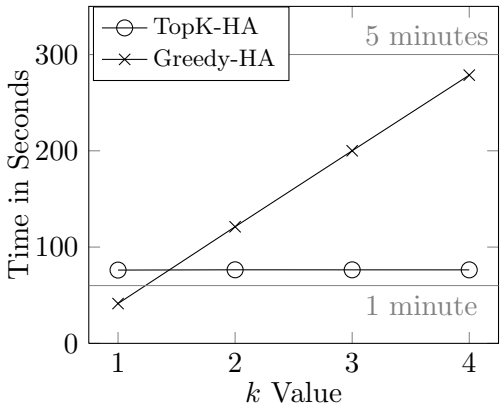


Figure 7.8. Running Time

Chapter 8

Discussing Other "Which" Queries

In this chapter, we briefly discuss how to support some other types of "which" queries. As we have mentioned before, "which" queries are diverse and have applications across many fields (such as urban planning or network planning). Because the applications of "which" queries is vast, we will discuss a few types of "which" queries below that we believe are highly useful across many domains.

First, we will discuss "Which Edge-Update-SP-Min" queries. Then, we will continue with a discussion of "Which Edge-Delete-SP-Min" queries. Finally, we will discuss "Which Edge-Add-MaxFlow-Max" queries.

8.1 "Which Edge-Update-SP-Min" Queries

"Which Edge-Update-SP-Min" queries (e.g., Q3) are very useful because the graph structure itself does not undergo any changes (i.e., the set of vertices V and set of edges E in the graph G remain unchanged). The only change is in the weights of the edges. In applications where the actual graph structure does not change, edge weight updates are the principle concern. For example, suppose that we are considering an application with an express mail company again. In certain situations, the express mail company may want to augment the existing delivery routes, rather than adding new routes or removing existing routes. In such a case, the express mail company is more concerned with how the existing network can be augmented to improve their overall delivery times (i.e., making a decision of adding more flights between certain cities because of existing infrastructure and the company does not have enough capital to add a completely new route).

Typically, edge weight update problems are solved by reducing the edge weight update operation to a sequence of operations. The first operation removes an edge e from the graph. Then, the second operation inserts a new edge, with the desired edge weight update value, into the graph. For "Which Edge-Update-SP-Min" queries, we can prove that our proposed framework above (the algorithms PA and AA for answering "Which Edge-Add-SP-Min" queries) also solve "Which Edge-Update-SP-Min" queries.

To answer "Which Edge-Update-SP-Min" queries, we model the "Which Edge-Update-SP-Min" query as such:

1. Let G be the input graph.
2. Let Q be the set of shortest path queries that we want to optimize.
3. Let P be the set of existing edges that we want to have edge weight updates performed on.
4. Let $\|e_i\|$ contains the desired updated edge weight value (contrast this with "Which Edge-Add-SP-Min" queries where $\|e_i\|$ contains the length of a new edge).

Now, we know that we want to update edge weights such that the distance sum of the shortest path queries is minimized. This is very similar to the optimization goal of "Which Edge-Add-SP-Min" queries. We note that without modifying the underlying graph structure, such as removing an existing edge, we can determine the benefit of an edge weight update by constructing a detour path using the desired edge weight update. Thus, Equation 4.3 or Equation 4.4, depending on what type of graph we are using, can be used in the same manner as "Which Edge-Add-SP-Min" queries to perform insertion free calculations of an edge's benefit. In fact, "Which Edge-Add-SP-Min" queries are a special case of "Which Edge-Update-SP-Min" queries.

8.2 "Which Edge-Delete-SP-Min" Queries

"Which Edge-Delete-SP-Min" queries (e.g., Q2) are more challenging to answer than either "Which Edge-Add-SP-Min" queries and "Which Edge-Update-SP-Min" queries. With edge removals from a graph, removing an edge that is

part of the shortest path of a query may result in the shortest path being longer; however, it is also possible that removing an edge on the shortest path of a query may not result in the shortest path distance changing due to the existence of another shortest path of the same length. Thus, it is unclear how long the shortest path distances are without removing the edges from the graph itself.

To answer "Which Edge-Delete-SP-Min" queries, we observe that one method that we can consider using is to reduce this query to a "Which ($|P|$-1)-Edge-Add-SP-Min" query. By reducing the "Which Edge-Delete-SP-Min" query to a "Which ($|P|$-1)-Edge-Add-SP-Min" query, we can use our "Which k-Edge-Add-SP-Min" framework to find an approximate answer to the query. First, we take the input graph G and remove all P edges to obtain the graph $G^{\{E-e_i:e_i \in P\}}$. Then, we use the "Which k-Edge-Add-SP-Min" framework and set $k = |P| - 1$ to find the $|P|-1$ most beneficial edges. The least beneficial edge (the remaining edge not found as the answer) is returned as being the edge, when removed from G, with the least impact on the shortest path query workload Q. The pseudocode for this proposed algorithm is outlined in Algorithm 10.

Algorithm 10 "Which Edge-Delete-SP-Min"

```
1: function DELETE(G, Q, P)
2:     k = |P|-1
3:     for e ∈ P do
4:         G = G - e
5:     end for
6:     for q(s,t) ∈ Q do
7:         sp_q = Dijkstra(s, t)
8:     end for
9:     Greedy-HA(G, Q, P, k)                    ▷ Algorithm 9
10:    return remaining vertex-pair
11: end function
```

8.3 "Which Edge-Add-MaxFlow-Max" Queries

Many graph problems are more computationally expensive than the shortest-path problem (e.g., the maximum flow problem can be solved by Edmonds-Karp's algorithm [1] in $O(|V||E|^2)$ time). Thus, for "which" queries like Q4 and Q5, it is critical to minimize the number of maximum flow algorithm calls, mirroring PA and AA minimizing calls to Dijkstra's algorithm. Note that certain maximum flow algorithms require finding an *augmenting path* (i.e., a *shortest path* from the source to the sink in the (residual) flow network). As such, we envision that the evaluation algorithms for "Which *-*-MaxFlow-*" queries can reuse some ideas presented in this work.

For example, a "Which Edge-Add-MaxFlow-Max" query (e.g., Q5) aims to increase the maximum flow value by connecting a vertex-pair with a new edge. Among all the considered vertex-pairs, we observe that if a new edge does not induce a residual flow, then the new edge cannot increase the maximum flow value. This resembles detecting whether a new edge can shorten query (s_i, t_i)'s distance in a "Which Edge-Add-SP-Min" query with source s_i and sink t_i in the flow network.

For a "Which Edge-Add-MaxFlow-Max" query, we have another important observation that allows us to reduce the search space. Specifically, among all the considered vertex-pairs, we observe that *only those vertex-pairs that span the minimum-cut can increase the maximum flow value*. This observation echos Lemma 1, which can be used to significantly reduce the number of vertex-pairs to be considered.

Furthermore, from the reduced search space, all remaining edges can increase the maximum flow value (i.e., in the context of shortest path distances of an augmenting path, all remaining edges can reduce the shortest path distance from s to t). Intuitively, the augmenting path that is the shortest, in length, increases the maximum flow value the most; thus, we can directly use the "Which Edge-Add-SP-Min" algorithms to find the shortest augmenting path, among all the possible augmenting paths remaining. The most beneficial edge of the "Which Edge-Add-SP-Min" query equivalently finds the edge with the largest increase in maximum flow value of the "Which Edge-Add-MaxFlow-Max" query. The pseudocode for this proposed algorithm is outlined in Algorithm 11.

Algorithm 11 "Which Edge-Add-MaxFlow-Max"

```
1: function EDGE-ADD-MAXFLOW-MAX(G, Q, P)
2:     Reduce P to only contain edges spanning the minimum cut
3:     AA(G, Q, P)                                   ▷ Algorithm 7
4:     return most beneficial edge
5: end function
```

8.4 "Which k-Edge-*-*-*" Queries

As we have seen in the above sections of this chapter, there are many other types of "which" queries that can be asked and solved using the "Which Edge-Add-SP-Min" query. By the same token as the "Which Edge-Add-SP-Min" query, the other types "which" queries mentioned in Chapter 8.1, Chapter 8.2, and Chapter 8.3 can also account of the case of k edges, where $k > 1$.

Similar to what we saw in the "Which k-Edge-Add-SP-Min" query, when $k > 1$, for the queries in Chapter 8.1, Chapter 8.2, and Chapter 8.3, we will

need to compute the optimal k edges from a set of ${}_{|P|}C_k$ possible edge combinations. Again, to exhaustively check all ${}_{|P|}C_k$ possible edge combinations would be woefully inefficient. As such, we can devise a plan similar to Top-K-HA and Greedy-HA (as we have done for the case of "Which k-Edge-Add-SP-Min" queries), to help solve the cases of $k > 1$ in the queries presented in Chapter 8.1, Chapter 8.2, and Chapter 8.3.

Chapter 9

Conclusion

This work presents "which" queries, a new class of decision-support queries, that are specific to graph data. In this work, we have presented various forms of "which" queries and their potential applications are also discussed. In particular, this work has discussed how to model "which" queries and has also identified a fundamental "which" query that can be used as a building block to answering other, more complicated, types of "which" queries.

In this work, we have specified a "which" query in the general form of [Component]-[Operation]-[Measurement]-[Goal]. Because "which" queries lead to eventual changes in the underlying graph data, we need to specify what graph component will undergo the eventual change. Additionally, how the graph structure itself changes needs to be indicated as well. Some "which" queries will lead to the graph structure changing (e.g., an addition or removal of an edge to the graph), while other "which" queries will not change the graph structure, but the attributes of the graph components themselves (e.g., edge weights are mod-

ified). Since "which" queries address a certain type of graph-based decision, we specify the type of decision made within the "which" query and also what the optimization goal of the decision is.

This work has identified the "Which Edge-Add-SP-Min" query as being the fundamental type of "which" query. As with most graph queries, the shortest path query is a fundamental type of query and also has uses in more complicated types of graph queries, such as the maximum flow query. We have presented one possible model for the query and also presented, in detail, algorithms that efficiently evaluate the"Which Edge-Add-SP-Min" query.

For answering "Which Edge-Add-SP-Min" queries, various techniques have been proposed to reduce the number of shortest path algorithm invocations, because these shortest path invocations are expensive operations. Given the input parameters, we first reduce the possible search space. After the search space has been reduced, we make use of a thresholding framework that allows us to terminate searching for the answer when the stopping criteria are met; thus, not every possible search space entry needs to be examined before the answer is returned to the user. This is a crucial component because decision support systems need to return answers to its users quickly, and any savings in the search is beneficial to a successful decision support system. Furthermore, we have proposed techniques to efficiently calculate the benefits of the bridging edges, without the need to modify the underlying graph data.

Our proposed algorithms for answering "Which Edge-Add-SP-Min" queries have been experimentally shown to be orders of magnitude faster than other basic solutions. Although both the proactive and adaptive algorithms generally

have similar performance, under certain instances one or the other algorithm performs better. Additionally, we have seen that the iterative shortest path incremental update algorithm is also a possible candidate for good performance, even though the iterative shortest path incremental update algorithm is sensitive to many factors, such as the number of shortest path incremental update invocations, and the number of vertices visited per shortest path incremental update invocation. However, between our three proposed algorithms, we suggest using the adaptive algorithm because it does not require a cost estimation step and in general performs better than the proactive algorithm and the iterative shortest path incremental update algorithm.

This work also discusses a possible solution to answering "Which k-Edge-Add-SP-Min" queries, where the query is interested in looking for the best $k > 1$ edges. We have proposed two heuristic algorithms for answering "Which k-Edge-Add-SP-Min" that are able to achieve high quality answers (either optimal or near optimal), and are orders of magnitude faster than a brute-force solution that checks every possible combination of k edges.

Finally, we have also discussed how other types of "which" queries can potentially be answered using the techniques and algorithms proposed in this work. We have discussed how "Which Edge-Update-SP-Min," "Which Edge-Remove-SP-Min," and "Which Edge-Add-MaxFlow-Max" queries can potentially be answered. We can see that, indeed, a number of techniques that have been proposed in this work can be used to answer other types of "which" queries. In some instances, the proposed algorithms in this work can be directly used to answer certain types of "which" queries.

Chapter 10

Appendix

10.1 APSP baseline algorithm

The all-pairs shortest path (APSP) baseline method first calculates the shortest path distances between all vertices $v \in V$ in a graph $G = (V, E)$. After the all-pairs shortest path distances are obtained, we can calculate the benefit of each bridging edge with respect to each query $q_j \in Q$ by using Equation 4.3 or Equation 4.4. After the search space has been computed, the most beneficial bridging edge e_i is computed and returned as the answer. Note that computing the all-pairs shortest path distances is very slow ($O(|V|^3)$, or $O(|V|^2 \log|V| + |V||E|)$ in a sparse graph). Furthermore, the memory space required to maintain the all pairs shortest path distances is $O(|V|^2)$ (the Argentina Road Network data set with $|V| = 85,287$ requires about 27 GB).

We note that all-pairs shortest path distances can be stored on disk. In this method, for each vertex, a single source Dijkstra's expansion is computed, thus

requiring $O(|V|)$ memory, and then these computed shortest path distances can be stored on disk. However, retrieving these distances from disk incurs high I/O cost.

Bibliography

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows : theory, algorithms, and applications.* Prentice Hall, 1993.

[2] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Survey*, 40(1):1–39, 2008.

[3] R. Bramandia, Byron Choi, and W-K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW 2008*, pages 845–854.

[4] R. Bramandia, Byron Choi, and W-K. Ng. Incremental maintenance of 2-hop labeling of large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 22:682–698, 2010.

[5] Edward Chan and Heechul Lim. Optimization and evaluation of shortest path queries. *The VLDB Journal*, 16:343–369, 2007.

[6] Surajit Chaudhuri and Umeshwar Dayal. Data warehousing and olap for decision support. In *SIGMOD*, pages 507–508, 1997.

[7] Chen Chen, Cindy X. Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. Mining graph patterns efficiently via randomized summaries. *PVLDB*, 2(1):742–753, 2009.

[8] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S. Yu. Graph OLAP: Towards Online Analytical Processing on Graphs. In *ICDM*, pages 103–112, 2008.

[9] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

[10] Kaikai Chi, Xiaohong Jiang, Susumu Horiguchi, and Minyi Guo. Topology design of network-coding-based multicast networks. *IEEE TPDS*, 19:627–640, 2008.

[11] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[13] Yang Du, Donghui Zhang, and Tian Xia. The optimal-location query. In *SSTD*, pages 163–180, 2005.

[14] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Incremental algorithms for the single-source shortest path problem. In *Foundation of Software Technology and Theoretical Computer Science*, pages 113–124. 1994.

[15] Yunjun Gao, Baihua Zheng, Gencai Chen, and Qing Li. Optimal-location-selection query processing in spatial databases. *TKDE*, 21:1162–1177, 2009.

[16] Ruoming Jin, Hui Hong, Haixun Wang, Ning Ruan, and Yang Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, pages 123–134, 2010.

[17] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.

[18] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608, 2008.

[19] Arijit Khan, Xifeng Yan, and Kun-Lung Wu. Towards proximity pattern mining in large graphs. In *SIGMOD*, pages 867–878, 2010.

[20] Chengkai Li, Kevin Chen-chuan, Chang Ihab, and F. Ilyas. Supporting ad-hoc ranking aggregates. In *In SIGMOD*, pages 61–72. ACM Press, 2006.

[21] Kok Yong Lim, Sieteng Soh, and S. Rai. Computer communication network upgrade for optimal capacity related reliability. In *Asia-Pacific Conference on Communications*, pages 1102 –1106, 5-5 2005.

[22] Tore Opsahl, Filip Agneessens, and John Skvoretz. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks*, 32(3):245 – 251, 2010.

[23] A. Pal, A. Paul, A. Mukherjee, M. Naskar, and M. Nasipuri. Fault detection and localization scheme for multiple failures in optical network. *Distributed Computing and Networking*, pages 464–470, 2008.

[24] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, pages 567–580, 2008.

[25] Silke Trissl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.

[26] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.

[27] Fang Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD*, pages 99–110, 2010.

[28] Yanghua Xiao, Wentao Wu, Jian Pei, Wei Wang, and Zhenying He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *EDBT*, pages 493–504, 2009.

[29] C. Yang and Jianzhong Zhang. Inverse maximum flow and minimum cut problems. *Optimization*, 40:147–170, 1997.

[30] Jianzhong Zhang and Yixun Lin. Computation of reverse shortest-path problem. *Journal of Global Optimization*, 25:243–261, 2003.

[31] Jianzhong Zhang, Zhenong Liu, and Zhongfan Ma. Some reverse location problems. *European Journal of Operation Research*, 124:77–88, 2000.

[32] Jianzhong Zhang, Zhongfan Ma, and Chao Yang. A column generation method for inverse shortest path problems. *Mathematical Methods of Operations Research*, 41(3), 1995.

[33] Ning Zhang, Yuanyuan Tian, and Jignesh M. Patel. Discovery-driven graph summarization. In *ICDE*, pages 880–891, 2010.

[34] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *PVLDB*, 2010.

[35] Lei Zou, Lei Chen, and M. Tamer ˙Distance-join: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.