This is to certify that the

dissertation entitled

RECURSIVE QUERY PROCESSING
IN LARGE DATABASES

presented by

Sungwon Jung

has been accepted towards fulfillment
of the requirements for

PhD degree in Computer Science

Major professor

Date December 18, 1995

Recursive Query Processing in Large Databases

By

Sungwon Jung

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DEGREE OF PHILOSOPHY

Department of Computer Science

1995

ABSTRACT

RECURSIVE QUERY PROCESSING IN LARGE DATABASES

By

Sungwon Jung

Recursive database structures are the fundamental processing elements in numerous important applications including navigation systems, recursive rule processing in knowledge base systems, and processing IS_A hierarchy in object-oriented systems. Query processing for these types of structures is the focus of numerous current database research projects, because traditional approaches are not suitable for this kind of query processing. In this thesis, we have investigated query processing methods for large recursive relational databases. We have used automobile navigation systems as one of the model applications. More general models for recursive queries using transitive closures have also been investigated.

In automobile navigation systems, a topographical road map can be defined by a large recursive relation, and "finding shortest paths" using this relation is a recursive query. We have developed a *HiTi* graph model for structuring large recursive relations to speed up the shortest path computation. The *HiTi* graph model provides a novel approach to abstracting and structuring a topographical road map in a hierarchical

fashion. We then proposed a new shortest path algorithm named $SPAH$, which utilizes the hierarchical abstraction of a topographical road map for its computation. Our performance analysis of $SPAH$ showed that it dramatically reduces the search space. Within the $HiTi$ graph framework, we also studied the parallel processing for *intra* and *inter* query shortest path problems. Our empirical analysis of these two problems revealed that the *inter* query shortest path problem provides more opportunity for scalable parallelism than the *intra* query shortest path problem.

We then studied recursive query processing for distributed fragments of recursive relations. The major performance issues involved were the description and location of recursive fragments in a distributed environment. Traditional approaches to describing and locating a fragment in non-recursive databases were not effective here. We have developed a new method for describing and locating recursive fragments based on the mathematical properties of lattices. We showed that lattice structures provide a good theoretical and practical basis for describing and locating recursive fragments. The performance of this lattice approach was analyzed both theoretically and experimentally in this thesis.

To my parents

ACKNOWLEDGMENTS

I wish to express my gratitude and appreciation to my thesis advisor Dr. Sakti Pramanik for his consistent guidance and encouragement right from the beginning, and his financial support. I am grateful for many discussions and invaluable comments he provided.

I would like to thank my guidance committee members, Dr. Moon Jung Chung, Dr. Jon Sticklen, and Dr. David Rovner, for their help and guidance.

I must express my heartful thanks to my father, BongJo Jung and my mother, JungHee Kim for their sincere pray during my studying here. I also thank my two sisters, Heewon Jung and Soonwon Jung, and my brother Sungkyu Jung for their encouragement.

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Description

Recursive relations have become one of the dominant types of relations in next generation databases such as deductive and object-oriented databases. The recursive relation can be easily viewed as a directed graph. Each instance of attributes participating in the transitive closure represents a node in the graph, and each tuple in the relation represents an arc [53]. The above recursive relation is usually called an *edge* relation, where each tuple corresponds to a labeled edge. Traditionally, the recursive relation refers to this edge relation. Queries are recursive if they are defined over recursive relations and their processing requires transitive closure computations. These recursive queries also must access detailed information about the nodes. Thus, we are defining another type of relation, called a node relation, where each tuple represents the detailed information of a node in the graph.

A significant amount of research has been done on recursive relations, primar-

ily in computing transitive closures [4, 42, 46, 49, 50, 57, 74, 83, 99, 102]. This is because every linearly recursive query requires transitive closure computation [52]. The shortest path problem is a special case of the transitive closure computation problem. The shortest path problem is fundamental for processing road map queries. Very large recursive relations are needed to store topographical road maps. Previously suggested transitive closure or graph traversal algorithms are not suitable for shortest path computations on topographical road maps because of their large search space.

In this thesis, we will first study an efficient database organization method that allows us to speed up the single pair shortest path (SPSP) computation from source to destination nodes on large topographical road maps. We will then investigate a fast SPSP algorithm that takes advantage of our proposed data organization method. The efficiency of our proposed SPSP algorithm and data organization method will be thoroughly analyzed on two dimensional grid graphs. Two dimensional grid graphs are considered to be typical examples of topographical road maps [64, 91]. Grid graphs not only closely model topographical road maps but also provide road map databases for controlled analysis of our algorithms.

This thesis wll then examine the parallel processing for *intra* and *inter* query SPSP problems on large topographical road maps. *Intra* query SPSP problems deal with parallelizing a single transaction of SPSP computation for further speedup, while *inter* parallel SPSP problems deal with computing multiple SPSP transactions in parallel. The *inter* query SPSP problem arises in the domain of automobile navigation systems where a single central server provides route information to automobile navigators. In

these systems, many vehicles send their SPSP computation requests to a central server and the server must be able to compute these multiple SPSP computation transactions within a reasonable amount of time. Thus, it is very useful to develop an efficient *inter* query parallel SPSP algorithm.

In a centralized database, the primary concern of recursive query processing is an efficient transitive closure computation (i.e., a shortest path computation for topographical road maps). However, in a distributed database system, the problems of describing and locating data are interrelated, and their optimization is an important goal for recursive query processing. One important optimization parameter for describing and locating distributed data is the data transmission cost, because this is a major performance bottleneck in distributed query processing. In distributed database systems, it is often convenient and beneficial to fragment and distribute data according to referencing frequency or locality. By storing only frequently accessed data at the site, we can significantly reduce the data maintenance and transmission cost. In recursive relations, frequently accessed data are likely to be related through the transitive relationships between them.

Most papers on parallel and distributed computation of transitive closure (including the shortest path computation) have used either predicates or hash functions to describe, distribute, and locate the data [32, 45, 47, 74, 84, 102]. However, the above two methods are not suitable for describing and locating the distributed fragments because they do not properly capture transitive relationships between data. In this thesis, we will investigate the problem of describing distributed fragments of the recursive relation that capture transitive relationships between data. Next, we will

show how to locally determine the location of the remote fragments of the recursive relation. By locally determining the location of remote fragments, we can avoid a significant amount of communication cost for recursive query processing.

1.2 Previous Works and Our Approach

Shortest path problems have been the fundamental problems in computer science. Therefore, a significant amount of research has been done on these problems, and excellent survey papers can be found in [23, 25, 31, 34, 81]. In the domain of automobile navigation systems where a database contains large recursive relations such as topographical road maps, one of the primary functions is to compute the shortest route from the current location (i.e. where a driver is) to the destination (i.e. where the driver wants to go). Thus, computing SPSP is also essential for processing of road map queries. As a database problem, the SPSP problem has been traditionally studied as a special case of single source and all pair shortest path problems in the context of recursive query processing [7, 8, 46, 65, 86, 101], transitive closure computation [3, 21, 22, 40, 48, 49, 52, 53, 57, 70, 103], and database query languages [2, 26, 71]. However, due to the large search space a topographical road map creates [64], these single source or all pair shortest paths algorithms are not suitable for computing SPSP on this kind of map.

Dijkstra first proposed an algorithm developed specifically for SPSP problems [24]. This algorithm further reduces the search space required by single-source and all-pair shortest path algorithms. However, the search space Dijkstra's algorithm has to

explore still remains very large when large topographical road maps are considered. Nicholson [77] proposed a scheme that finds SPSP by alternately building two trees, one rooted at the source node and the other rooted at the destination node. By exploring the search space from both ends simultaneously, his algorithm can reduce the explored search space of Dijkstra's algorithm by roughly half.

Another approach to reducing the explored search space is to use semantic information about the domain of topographical road maps. That is, each node in the graph corresponding to a topographical road map has its distinct coordinates represented in terms of longitude and latitude. By taking advantages of this information, we can estimate the cost of the shortest path between any two nodes. Either Euclidean or Manhattan distances are usually used for the estimation of shortest path cost between nodes. A* algorithm [30] is a typical example of this approach. A* algorithm using Manhattan distance estimation gives an optimal shortest path only when edge costs are uniform. This is because when edge costs are nonuniform, the Manhattan distance is usually overestimated. Unlike A* algorithm using Manhattan distance estimation, A* algorithm using Euclidean distance estimation always gives optimal shortest paths. The performance of A* algorithm was empirically analyzed on grid graphs as well as on actual topographical road maps by Pearl and Shekhar et. al. in [80, 91].

Pearl [80] showed that A* algorithm was more efficient than Breadth First Search (BFS) [20] when the database fit in main memory. However, Pearl's analysis did not examine the effect of path length and edge costs on the relative performance of A* algorithm. This was later analyzed by Shekhar et. al. [91]. Their analysis was done on

10×10, 20×20, and 30×30 grid graphs with different edge-cost distributions and one actual digitized Minneapolis road map consisting of 1089 nodes and 3300 edges. Their analysis shows that A* outperforms BFS if the path (source,destination) is much smaller than the diameter of the graph, or if the edge-cost distribution is skewed.

The explored search space can be further reduced by combining A* algorithm and Nicholson's two-tree building approach. That is, each tree is expanded by using A* algorithm. Mohr and Pasche [72] proposed a new SPSP algorithm named *OTO*, which is a hybrid of A* and Nicholson's algorithms. They provided an empirical analysis of *OTO*, Nicholson's, A*, and Dijkstra's algorithms on three grid graphs (i.e., 100×100 nodes with three different edge cost distributions) and one road map of Switzerland (i.e., 3937 nodes and 12500 edges). For their empirical analysis, they used Manhattan distance as the estimation of shortest path cost between nodes. Their analysis showed that *OTO* outperforms Dijkstra and A* algorithms. The *OTO* algorithm was also shown to perform far better than Nicholson's if the degree of edge cost variation is not high.

Agrawal and Jagadish [5] developed a data organization approach to further speed up SPSP computation. Their approach was to precompute some partially precomputed path information and then use it at run time to prune the explored search space. This data organization technique assumes that a graph (i.e., topographical road map) can be decomposed into a set of subgraphs. This assumption is reasonable in the domain of road maps, since a large road map can be easily decomposed into a set of small sub-road maps. While this approach gives much faster computation time than all the algorithms discussed above, it has additional storage overhead and

update complexity.

Although the approach of Agrawal and Jagadish speeds up the shortest path computation significantly, it does not optimize processing the shortest path queries based on road map navigation. The reason for this is that the shortest path obtained from their approach still includes all the detail intermediate nodes between the source and destination, and navigators do not usually need this information. This problem would be very serious if we needed to compute a shortest path on a large topographical road map on which the discretized interval is fine-grained, and the two end nodes on the path are far apart. To cope with this problem, we have developed a *HiTi* (Hierarchical mulTi) graph model of very large recursive relations (e.g., topographical road maps).

The *HiTi* graph provides a novel approach to abstracting and structuring road maps in a hierarchical fashion. It is modeled after the mechanism people use to select a route on a road map. That is, when a person wants to find the shortest route from a current location in one state to a destination point in another, he/she usually reads road maps in a deceasing map-scale order. For example, a driver who wants to find the shortest path from a current location in East Lansing to a destination in Los Angeles, will consult the highway road map for the entire U.S.A. first. The person will then find a more detailed highway connection by reading state road maps (i.e., Michigan, and California). Finally, the driver will consult city road maps of East Lansing and Los Angeles to find very detailed information about the starting point and destination. By maintaining road maps in this hierarchical structure, *HiTi* graphs provide the following three features:

1. They allow the computation of a shortest path without having to search through all the detailed maps of the intermediate states and cities.

2. They provide a basis for storing road maps at various levels of hierarchical abstraction. Thus, a more controlled storage management suitable for navigation systems (e.g., automobile navigation systems) with limited available storage can be built.

3. They are able to support various types of database queries on hierarchical abstractions of a road map.

HiTi graphs, therefore, provide a powerful framework for implementing road-map queries (e.g., shortest-path computation) as well as for storing very large topographical road maps. We developed a new shortest-path algorithm named *SPAH*, which utilizes *HiTi* graph structure. By taking advantage of hierarchical edge levels in the *HiTi* graph, *SPAH* significantly reduces the explored search space. The performance of *SPAH* can be further improved if Euclidean distance estimation is used. Although *SPAH* reduces the explored search space significantly, its performance improvement is limited by the computation power of a single processor.

In order to overcome this performance limitation, we studied parallel processing for shortest path problems. In this study, we considered not only *intra* but also *inter* query SPSP problems. *Intra* query parallel SPSP problems deal with parallelizing a single pair shortest path transaction, whereas *inter* query SPSP problems deal with parallelizing *multiple* single-pair shortest-path computations. Mohr and Pasche [72] proposed an *intra* parallel SPSP algorithm called *OTOpar*, which is a parallel im-

plementation of the *OTO* algorithm mentioned above. That is, two processors are used in *OTOpar*, and each processor executes A* algorithm with Manhattan distance estimation to build its corresponding tree. To the best of our knowledge, no previous work has been done on *inter* query SPSP problems.

The *OTOpar* algorithm is not scalable because its use of only two processors limits its performance improvement. Therefore, we need a parallel *intra* query SPSP algorithm that is more scalable and fine-grained than *OTOpar*. In this thesis, we proposed both *intra* query and *inter* query parallel SPSP algorithms named *PASPAH* and *MISPAH* respectively. These two algorithms are based on the *HiTi* graph structure. The *HiTi* graph structure provides the opportunity for fine-grained parallel processing for *intra* query SPSP problems. This is because multiple parallel shortest-path computations can be initiated based on the nodes at a higher level of abstraction of a topographical road map.

This far, we have discussed recursive query processing in a centralized database system where data are stored at a single location. In this environment, the primary goal of optimizing recursive query processing is to speed up either shortest path computation or, more generally, transitive closure computation. Location of the data in recursive relations is not an issue because they are all in one location. However, in a distributed database system, recursive relations are often fragmented and distributed over various sites to reduce the data maintenance cost and the data access cost. Each site stores the data (i.e., fragments) that are frequently accessed at that sites. In such a system, the reduction of transmission of data between sites is also very critical for the optimization of recursive query processing. Thus, we need an efficient fragment

description and location method to avoid the significant communication costs for query processing. This could be achieved by keeping remote fragment descriptions in local sites.

Most papers on parallel and distributed computation of transitive closure (including the shortest path) have used either predicates or hash functions to fragment, distribute, and identify the data in recursive relations [32, 45, 47, 74, 84, 102]. Huang et.al. [45] and Valduriez et.al. [102], studied the parallel computation of transitive closures where a hash function is applied to fragment a recursive relation. Ganguly et.al. [32] suggested a framework for the parallel processing of datalog queries where hash functions are also used to fragment recursive relations. Nejdl et.al. [74] studied evaluating recursive queries in distributed databases in which recursive relations are fragmented by predicates. However, these fragmentation criteria, defined by hash functions or predicates, are not suitable for fragmenting recursive relations in distributed environments. This is because they do not properly capture the referencing locality of data, which is of paramount importance to the success of distributed database systems. Locality often comes from the transitive relationships between data. Stated another way, relevant data, frequently accessed by recursive queries, are likely to be related in the transitive closures. Hence, in distributed databases, fragmentation criteria based on the transitive relationship are considered to be more suitable for recursive relations than traditional criteria.

Based on the criteria defined by transitive relationships, we developed a method to describe and locate the fragments of a recursive relation. Our method uses lattice structures. Lattice structures provide a powerful mathematical representation tool

for the description of recursive fragments. A distinct set of lattices describes each recursive fragment. This lattice representation provides a concise but flexible fragment description. It is concise in that a large recursive fragment is represented by a small set of lattices. At the same time, it is flexible in that it can represent any type of recursive fragment. Finding the optimal lattice descriptions of fragments is shown to be an NP-complete problem. We proposed an efficient heuristics for this problem. The performance of our heuristics was analyzed theoretically as well as empirically. The empirical analysis showed that our heuristics gives a near-optimal lattice description. The lattice approach creates unique update problems. We have extensively analyzed the performance of our proposed update algorithms.

1.3 Dissertation Outline

The structure of this thesis is as follows: Chapter 2 presents a HiTi graph model of large recursive relations for efficiently computing an optimal single pair shortest path. In Chapter 3, parallel processing for *intra* and *inter* single pair shortest path problems is discussed. Chapter 4 presents a methodology for describing and locating the distributed fragments of recursive relations in a distributed database. Chapter 5 concludes the thesis with a list of future research issues.

The following parts of this thesis have been published: The content of chapter 2 appears in [59]. The content of Chapter 4 appears in [82]. The content of Appendix A appears in [58]. Appendix B [60] and C [104] are being submitted. Chapter 3 will be submitted after additional work has been done.

Chapter 2

HiTi Graph Model of Large Recursive Relations for the Shortest Path Computation

In navigation systems, a primary task is to compute the minimum cost route from the current location to the destination. One of the major problems for navigation systems is that a significant amount of computation time is required to find a minimum cost path when the topographical road map is large. Since navigation systems are real time systems, it is critical that the path be computed while satisfying a time constraint. An efficient database organization method is proposed for structuring a topographical road map to speed up the computation of a minimum cost path. Data organization methods previously suggested either do not allow optimal minimum cost path generation or require searching all intermediate nodes on the minimum cost path.

In navigation systems, a navigator may not have to know all the intermediate nodes to go from the current location to destination. In this chapter, we propose a new graph model named *HiTi* (Hierarchical mulTi graph model), for efficiently computing an optimal minimum cost path. Multiple levels of geographical boundaries (e.g. cities, counties, and states) can be easily mapped into the hierarchical structure of the proposed graph model. Various levels of hierarchical abstractions can also serve as the basis for efficient storage management for large road maps. Thus, it provides the basis to develop a more controlled storage management suitable for navigation systems with limited available storage (e.g., navigation system inside an automobile). Based on *HiTi* graph model, we propose a new single pair minimum cost path algorithm. We empirically show that our proposed algorithm dramatically reduces the explored search space. Further, we empirically analyze our algorithm by varying both edge cost distribution and hierarchical level number of *HiTi* graphs.

2.1 Introduction

In navigation systems, a primary function is to find possible routes from the current location (e.g., where a driver is currently positioned) to the destination (e.g., where the driver wants to go) with a minimum expected cost. For this purpose, they use a topographical road map which is in the form of the following recursive relation :

topographical_road_map (source, destination, cost)

where the cost attribute indicates, for example, a minimum expected time of travel from point *source* to point *destination*. Another applicable cost can be the shortest distance between the two end points.

One of the major difficulties of navigation systems is the size of the topographical road map data. It requires about 2.4 Gbytes of storage to store a small $100\,mi \times 100\,mi$ map discretized at $100\,feet$ intervals [5, 64]. Thus, the size of data involved is very large when larger maps are considered. Minimum cost route computation with this large amount of road map data requires a significant amount of computation time. Since navigation systems are real time systems, it is critical to compute a minimum cost route satisfying a time constraint.

Previously suggested transitive closure or graph traversal algorithms [4, 46, 49, 50, 57, 83, 86, 99] are not directly applicable to **topographical_road_map (source, destination, cost)** for the computation of a minimum cost path due to the very large volume of data they have to search. Thus, we need an efficient database organization method for structuring the topographical road map to speed up the computation of

a minimum cost path. In this regard, two different approaches have been studied in the past. One approach is to develop a database structure which gives a suboptimal minimum cost path quickly. The other is to develop the database structure which gives an optimal shortest path.

For the suboptimal shortest path generation, Ishikawa et. al., Shapiro et. al., and Liu et. al. used road hierarchies (i.e., Freeways, Highways, ..., Side roads) to speed up minimum cost routes [51, 68, 89, 107]. They used multiple levels of hierarchical details of road maps to cut down the unnecessary search space. Ishikawa et. al. [51] applied Dijkstra algorithm to the hierarchically structured road maps. Shapiro et. al. [89] proposed a new graph structure, named *LGS (Level Graph Structure)* which models the road hierarchies theoretically. Based on LGS, Shapiro et. al. gave a new algorithm which generates approximate shortest paths rapidly. Their study showed that the length of the path produced by LGS converges rapidly to that of the actual minimum cost path as the distance between the source and destination nodes increases. Liu et. al. [68] studied integrating Dijkstra's algorithm with an AI knowledge-based approach and case-based reasoning for computing a minimum cost path.

For the optimal shortest path generation, Agrawal and Jagadish recently studied a data organization technique which precomputes and stores some partial path information [5]. They use the precomputed partial path information to prune the search space when computing a minimum cost path. While their approach speeds up the computation of a minimum cost path, it does not optimize processing minimum cost path queries based on road map navigation. The reason is that their approach still

requires searching all the intermediate nodes on a minimum cost path. In navigation systems, it is not necessary for a navigator to know all the intermediate nodes on a minimum cost path. This can be illustrated by the following example. Consider an automobile navigation system where a driver wants to find a minimum cost path from a place in East Lansing to a destination point in New York city. He/she needs to know more about detailed route information near East Lansing and New York city than those in other intermediate places (e.g., Toledo, and Pittsburgh). It is not likely that that person has to know about detailed route information within all the intermediate cities. The rational for this is based on the following two observations. First, a driver likes to have detailed route information near East Lansing to get into an interstate highway from a current location in East Lansing. Second, he/she needs to know detailed route information near New York city to get into the local road which leads to the destination point in New York city.

The above problem would be very serious if we need to compute a minimum cost path on a large topographical road map where its discretized interval is fine-grained and two end nodes on the path are far apart. To cope with this problem, we have developed a *HiTi* (Hierarchical mulTi) graph model of very large recursive relations (e.g., topographical road maps). In the following paragraphs, we will first describe *HiTi* graphs informally followed by a more formal description of the model.

Consider a topographical road map viewed as a directed graph $G(V, E)$. Nodes in V correspond to discretized grid points representing map objects in a road map. Edges in E correspond to the connections between the nodes in V. Then, regional bound-

aries partition a road map $G(V, E)$ into a set of Component ROad Maps (CROM) exclusively. Each CROM can be viewed as a subgraph with its boundary nodes defining the boundary of the CROM. Connectivity between CROMs is represented by the connectivity of their boundary nodes. One CROM has direct connections with another, if boundary nodes of the former are directly connected to boundary nodes of the latter. We call this kind of connectivity as *between* connections of CROMs. For each CROM, we precompute the cost for each pair of connected boundary nodes. If two boundary nodes of the same CROM are connected by a path solely contained within the CROM, we call this a *within* connection. By using the *between* and *within* connections, we can partition the entire road map. Examples of *between* and *within* connections of CROMs are shown in Figure 1.1 where dotted lines represent regional boundaries.

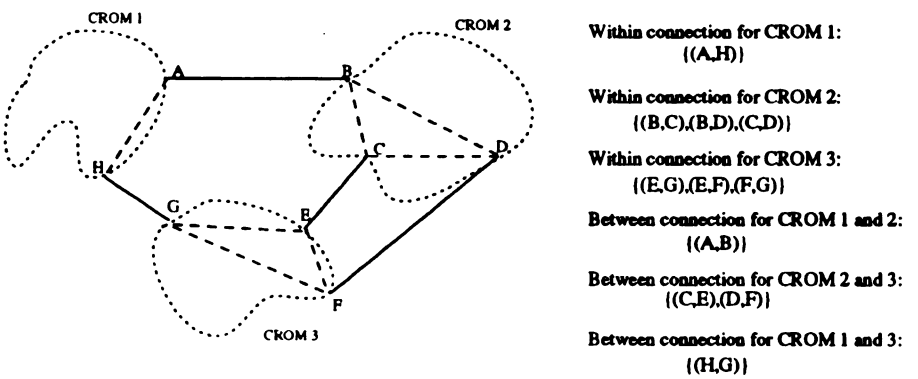


Figure 2.1: Examples of *within* and *between* connection of CROMs

HiTi graph is a graph whose nodes are the boundary nodes of the CROMs and edges are the *within* and *between* connections of CROMs. Note that a CROM can be

defined to contain a set of CROMs, thus creating a multi level hierarchy. Thus, we first need to determine a set of the lowest level CROMs which exclusively partition an entire road map. We call these CROMs as level 1 CROMs. Then, we can recursively construct a set of level k CROMs by grouping a set of geographically adjacent level *k-1* CROMs where $k \geq 2$. These sets of level *1, 2, ... ,k* CROMs form a complete balanced tree structure where the root node of the tree is a whole road map. For example, consider a topographical road map covering the entire area of U.S.A. The U.S.A. consists of 4 subregions which are *Eastern, Midwestern, Southern* and *Western* regions. These 4 regions consist of their subregions which are the states of U.S.A. Each state consists of a set of counties and each county consists of a set of cities. If a city road map is defined as basic i.e., level 1, CROMs, then level 2 CROMs are county road maps, level 3 CROMs the state road maps, and so on.

Given a complete balanced tree representing level *1, 2, ... ,k* CROMs, level 1 CROMs will be the leaf nodes, i.e., level 1 node of the tree. Each level 2 nodes of the tree will represent the level 2 CROMs and capture the *between* and *within* connections of the level 1 CROMs. Thus, in general, a complete balanced tree has 1 thru k levels corresponding to 1 thru k level CROMs. A level l ($\leq k$) node of the tree corresponds to *between* connections of level l CROMs plus level l *within* connections. Note that level l *within* connections capture the *between* and *within* connections of the level $l-1$ CROMs that corresponds to the child nodes of the tree. A *HiTi* graph is named the level k *HiTi* graph if its nodes and edges represent *boundary* nodes, *between* and *within* connections of level *1, 2, ... ,k* CROMs. The *HiTi* graph constructed for the

United States example above is a level 4 *HiTi* graph.

The rest of the paper is organized as follows. In section 2.2, a formal framework and description of *HiTi* graphs are discussed. In section 2.3, we propose and empirically analyzed a new single pair shortest path algorithm that takes advantage of *HiTi* graphs. Section 2.5 discusses the effects of updates on *HiTi* graphs. Finally, section 2.6 gives concluding remarks.

2.2 Formal Framework and Description of HiTi Graphs

Recursive relations have the generic form R (att_1, att_2, att_3) where attributes satisfy the following 3 conditions:

1. att_1 and att_2 are the key attributes of R and share the same domain

2. att_1 and att_2 are the recursive join attributes and their corresponding values are related by some transitive relationship

3. att_3 describes the transitive relationship between att_1 and att_2

Based on the above generic form, a recursive relation can be viewed as a directed graph $G(V, E)$. Each node in V is represented by the value of att_1. Each edge in E is represented by values of (att_1, att_2, att_3).

Suppose that a recursive relation $G(V, E)$ is partitioned into a set of subgraphs (i.e. $SG_1^1\ (V_1^1, E_1^1)$, $SG_2^1\ (V_2^1, E_2^1)$, ..., $SG_{n_1}^1\ (V_{n_1}^1, E_{n_1}^1)$) such that:

$$V_1^1 \cup V_2^1 \cup \cdots \cup V_{n_1}^1 = V, \quad E_1^1 \cup E_2^1 \cup \cdots \cup E_{n_1}^1 \subset E$$

$$V_i^1 \cap V_j^1 = \emptyset \text{ and } E_i^1 \cap E_j^1 = \emptyset \text{ where } 1 \leq i, j \leq n_1 \text{ and } i \neq j$$

We name $SG_i^1\ (V_i^1, E_i^1)$ a level 1 *subgraph* i and the connections between all level 1 *subgraphs* are captured in $PE^1 = E - (E_1^1 \cup E_2^1 \cup \cdots \cup E_{n_1}^1)$. Then, each $(x,y,z) \in PE^1$ has the property that $x \in V_i^1$ and $y \in V_j^1$ where $i \neq j$. Note that (x,y,z) corresponds to (att_1, att_2, att_3) where $x = att_1$, $y = att_2$, $z = att_3$.

Definition 2.2.1 *Let N_i^1 denotes a set of nodes in $SG_i^1\ (V_i^1, E_i^1)$ connected to/from nodes in other subgraphs $SG_j^1\ (V_j^1, E_j^1)$ where $i \neq j$ and $1 \leq i, j \leq n_1$. Specifically, $N_i^1 = \{x \mid (x,y,z) \in PE^1 \ \wedge \ x \in V_i^1\} \cup \{y \mid (x,y,z) \in PE^1 \ \wedge \ y \in V_i^1\}$. Set N_i^1 is called level 1* **boundary nodes** *of SG_i^1.*

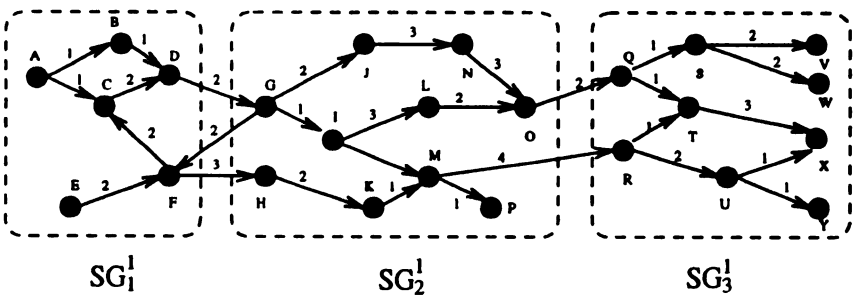


Figure 2.2: A digraph G and its level 1 subgraph SG_1^1, SG_2^1 and SG_3^1

For example, consider Figure 2.2 where a digraph G and its subgraphs SG_1^1, SG_2^1 and SG_3^1 are shown. Set N_2^1 of subgraph SG_2^1 is { G, H, M, O }. Each level 1 subgraph is described and identified by its boundary nodes, since they exclusively belong to one level 1 subgraph. Based on boundary nodes of SG_i^1, definition 2.2.2 gives formal definitions of level 1 *between* and *within* edge sets B_i^1 and W_i^1 of SG_i^1.

Definition 2.2.2 *Let* $\{ SG_j^1\ (V_j^1, E_j^1) \mid j = 1, n_1 \}$ *be a set of level 1 subgraphs of graph* $G(V, E)$ *with the corresponding* N_j^1. *Then, for each* SG_i^1 *where* $1 \leq i \leq n_1$, $B_i^1 = \{ (x, y, z) \mid (x, y, z) \in PE^1 \wedge x \in V_i^1 \}$ *and* $W_i^1 = \{ (x, y, f_z(x, y)) \mid (x, y) \in (N_i^1 \times N_i^1) \wedge (x \xrightarrow{f_z(x,y)} y \text{ in } SG_i^1) \wedge x \neq y \}$.

Each set B_i^1 contains an edge (x, y, z) if there is an edge (x, y, z) in PE^1 where x is in SG_i^1 and y is in a different level 1 subgraph. In other words, set B_i^1 contains connectivity information between level 1 subgraph SG_i^1 and other level 1 subgraphs of $G(V, E)$. Then, the following condition holds:

$$PE^1 = \cup_{j=1}^{n_1} B_j^1$$

From the above, it can be noted that edges in $\cup_{j=1}^{n_1} B_j^1$ exclusively partition graph $G(V, E)$ into a set of subgraphs $SG_1^1\ (V_1^1, E_1^1)$, $SG_2^1\ (V_2^1, E_2^1)$, ..., $SG_{n_1}^1\ (V_{n_1}^1, E_{n_1}^1)$.

Each set W_i^1 contains an edge $(x, y, f_z(x, y))$ if there exists a directed path from the boundary nodes x to y of SG_i^1. Function $f_z(x, y)$ gives an aggregated cost (i.e. from node x to y) with respect to z within SG_i^1. In other words, each edge in W_i^1 represents additional precomputed connectivity information of boundary nodes within

SG_i^1. Thus,

$$E \subseteq (\cup_{j=1}^{n_1} E_j^1 \cup \cup_{j=1}^{n_1} B_j^1 \cup_{j=1}^{n_1} W_j^1)$$

As an example, Table 2.2 shows the corresponding values of N_i^1, B_i^1 and W_i^1 of the three subgraphs shown in Figure 2.2. In the following example, $f_z(x, y)$ gives the cost of a minimum expected time of travel between node x and y.

i	N_i^1	B_i^1	W_i^1
1	{D,F}	{(D,G,2),(F,H,3)}	{(F,D,4)}
2	{G,H,M,O}	{(O,Q,2),(M,R,4),(G,F,2)}	{(G,O,6),(G,M,3),(H,M,3)}
3	{Q,R}	$\emptyset$	$\emptyset$

Table 2.1: N_i^1, B_i^1, and W_i^1 of SG_i^1 in Figure 2.2

So far, we have introduced level 1 subgraphs and their associated information such as level 1 *Boundary* node sets, and *between* and *within* edge sets. In general, based on these level 1 subgraphs and their associated information, we can recursively define level k subgraphs SG_i^k and their corresponding N_i^1, B_i^1, and W_i^1 for any $k \geq 2$. Their details are discussed in the following definitions.

Definition 2.2.3 *Let* $\Psi^{k-1} = \{ SG_i^{k-1} (V_i^{k-1}, E_i^{k-1}) \mid i = 1, n_{k-1} \}$ *be the set of all level k-1 subgraphs of graph* $G(V, E)$ *where* $k \geq 2$. *Then, a level k subgraph* $SG_i^k (V_i^k, E_i^k)$ *is defined as a subgraph induced by all nodes of level k-1 subgraphs in* $\Phi \subseteq \Psi^{k-1}$ *where* $|\Phi| \geq 2$.

After all level k subgraphs in Ψ^k are defined, the following three requirements must be satisfied:

1. Each level *k-1* subgraph in Ψ^{k-1} is a subgraph of some level k subgraph in Ψ^k.

2. All level k subgraphs in Ψ^k do not overlap among themselves. That is,

$$V_1^k \cup V_2^k \cup \cdots \cup V_{n_k}^k = V, \quad E_1^k \cup E_2^k \cup \cdots \cup E_{n_k}^k \subset E$$

$$V_i^k \cap V_j^k = \emptyset \text{ and } E_i^k \cap E_j^k = \emptyset \text{ where } 1 \leq i, j \leq n_k \text{ and } i \neq j$$

3. All edges belonging to level k *between* edge sets are removed from level *k-1* *between* edges sets. As a result, $(\cup_{j=1}^{n_{k-1}} B_j^{k-1} \cap \cup_{j=1}^{n_k} B_j^k) = \emptyset$.

By satisfying the preceding first two requirements, all subgraphs in $\cup_{j=1}^{k+1} \Psi^j$ are related to each other in a complete balanced tree structure where the root node of the tree symbolizes $G(V, E)$. This complete balanced tree is named a ***subgraph tree*** (ST). The root node of ST is considered as a level *k+1* subgraph and it has all level k subgraphs as child nodes. Recursively, each node symbolizing a level k subgraph has a set of level *k-1* subgraphs as child nodes. All leaf nodes of ST symbolizes level 1 subgraphs. This subgraph tree is named level *k+1* ST if the root of ST symbolizes level *k+1* subgraph (i.e. $G(V, E)$). The following Figure 2.3 shows an example of level 3 subgraph tree. This tree structure shows a level 3 subgraph SG_1^3 induced by the nodes in level 2 subgraphs (i.e. SG_1^2, SG_2^2 and SG_3^2). It also shows a level 2 subgraph SG_1^2 induced by the nodes in the two level 1 subgraphs SG_1^1 and SG_2^1.

For each level k subgraph SG_i^k of ST where $1 \leq i \leq n_k$, there exist corresponding level k ***boundary*** nodes (i.e. N_i^k) and level k *between* and ***within*** edge sets (i.e. B_i^k and W_i^k). The semantics of the level k ***boundary*** nodes, level k *between* and ***within*** edge sets are same as that of level 1 ***boundary*** nodes, level 1 *between* and ***within*** edge sets.

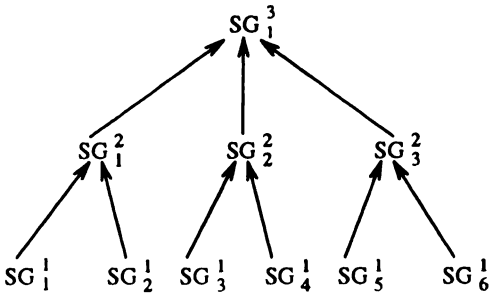


Figure 2.3: An example of level 3 subgraph tree ST

They are formally defined in definition 2.2.4.

Definition 2.2.4 *Let* $PE^k = E - (E_1^k \cup E_2^k \cup \cdots \cup E_{n_k}^k)$. *Then,* $N_i^k = \{\ x \mid (x,y,z) \in PE^k \wedge x \in V_i^k\ \} \cup \{\ y \mid (x,y,z) \in PE^k \wedge y \in V_i^k\ \}$, $B_i^k = \{\ (x,y,z) \mid (x,y,z) \in PE^k \wedge x \in V_i^k\ \}$ *and* $W_i^k = \{\ (x,y,f_z(x,y)) \mid (x,y) \in (N_i^k \times N_i^k) \wedge (x \xrightarrow{f_z(x,y)} y$ *in* $SG_i^k) \wedge x \neq y\ \}$. *Function* $f_z(x,y)$ *gives an aggregated cost (i.e. from node* x *to* y*) with respect to* z *within* SG_i^k.

As can be noticed from the above definition, $PE^k = \cup_{j=1}^{n_k} B_j^k$. Note that $\cup_{j=1}^{n_k-1} B_j^{k-1}$ $= \cup_{j=1}^{n_k-1} B_j^{k-1} - \cup_{j=1}^{n_k} B_j^k$. In order to show how N_i^k, B_i^k, and W_i^k are obtained, we use the digraph in Figure 2.4 as an example.

From Figure 2.4, we got PE^1 = { (4,6,2), (5,7,3), (5,20,7), (10,11,4), (10,12,6), (14,16,4), (15,14,3), (17,19,5), (17,20,4), (18,20,2), (23,25,4), (24,25,5) } and PE^2 = { (5,20,7), (10,11,4), (10,12,6), (17,19,5), (17,20,4), (18,20,2) }. By utilizing these PE^1 and PE^2, we can obtain *boundary* nodes, *within* and *between* edges of level 1 and 2 subgraphs. These are shown in Table 2.2. Note that in Table 2.2, *between* edge sets of SG_2^1 and SG_4^1 do not contain edge(s) in {(5,20,7), (10,11,4), (10,12,6)}

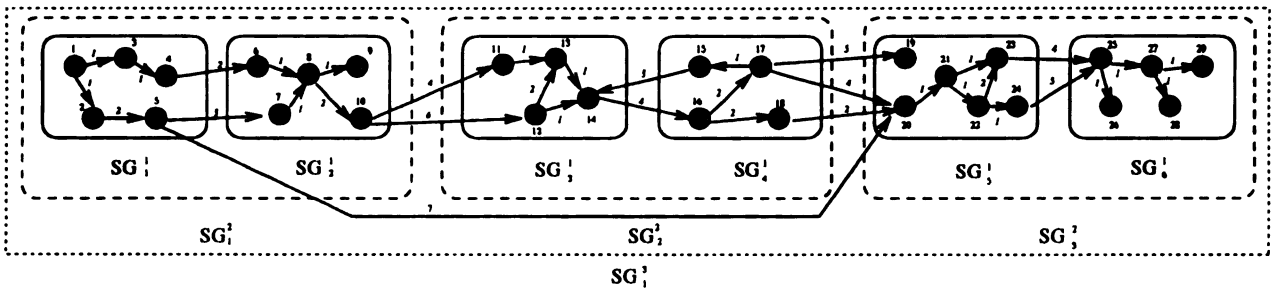


Figure 2.4: An example of a digraph $G(V, E)$ partitioned according to ST in Figure 2.3

and {(17,19,5), (17,20,4), (18,20,2) } respectively. This is due to the last requirement related to Definition 2.2.3. Thus, in general, given level *k+1* ST, edges in B_i^k of SG_i^k capture connections between *boundary* nodes of level *k-1* child subgraphs of SG_i^k and those of the rest of level *k-1* subgraphs. *Between* edges of level *k-1* child subgraphs of SG_i^k capture connections between *boundary* nodes of themselves.

i	N_i^1	W_i^1	B_i^1
1	{4,5}	$\emptyset$	{(4,6,2),(5,7,3) }
2	{6,7,10}	{(6,10,3),(7,10,3)}	$\emptyset$
3	{11,12,14}	{(11,14,2),(12,14,1)}	{(14,16,4)}
4	{15,16,17,18}	{(16,15,3),(16,17,2),(16,18,2)}	{(15,14,3)}
5	{19,20,23,24}	{(20,23,2),(20,24,3)}	{(23,25,4),(24,25,5)}
6	{25}	$\emptyset$	$\emptyset$
i	N_i^2	W_i^2	B_i^2
1	{5,10}	{5,10,6)}	{(5,20,7),(10,11,4),(10,12,6)}
2	{11,12,17,18}	{(11,17,8),(11,18,8),(12,17,7),(12,18,7)}	{(17,19,5),(17,20,4),(18,20,2)}
3	{19,20}	$\emptyset$	$\emptyset$
i	N_i^3	W_i^3	B_i^3
1	$\emptyset$	$\emptyset$	$\emptyset$

Table 2.2: *Boundary* nodes, *between* and *within* edges of level 1, 2, and 3 subgraphs in Figure 2.4

We have given basic definitions necessary to define a *HiTi* graph. However, these definitions (i.e. N_i^k, B_i^k, and W_i^k) do not provide an efficient way of obtaining their corresponding values. A primary reason for this inefficiency is that their definitions require obtaining their values by using V_i^k and E_i^k, which is computationally expensive. Thus, we need a more efficient way of computing these values. The following definitions provide a recursive way of obtaining the values of N_i^k, B_i^k, and W_i^k based on the values of N_i^{k-1}, B_i^{k-1}, and W_i^{k-1} for all level $k-1$ subgraphs in Ψ^{k-1}.

Definition 2.2.5 *Let* $D = \{ 1, 2, \ldots, n_{k-1} \}$ *where* $n_{k-1} = |\Psi^{k-1}|$. *Suppose level k subgraph* SG_i^k (V_i^k, E_i^k) *is induced by all nodes in level k-1 subgraphs* SG_t^{k-1}, $\forall$ $t \in I$ *where* $I \subset D$. *Then,* $N_i^k = \{ x \mid (x,y,z) \in \cup_{j=1}^{n_{k-1}} B_j^{k-1} \wedge x \in \cup_{t \in I} N_t^{k-1} \wedge y \notin \cup_{t \in I} N_t^{k-1} \} \cup \{ y \mid (x,y,z) \in \cup_{j=1}^{n_{k-1}} B_j^{k-1} \wedge y \in \cup_{t \in I} N_t^{k-1} \wedge x \notin \cup_{t \in I} N_t^{k-1} \}$.

Definition 2.2.6 *Let* $D = \{ 1, 2, \ldots, n_{k-1} \}$ *where* $n_{k-1} = |\Psi^{k-1}|$. *Suppose level k subgraph* SG_i^k (V_i^k, E_i^k) *is induced by all nodes in level k-1 subgraphs* SG_t^{k-1}, $\forall$ $t \in I$ *where* $I \subset D$. *Then,* $B_i^k = \{ (x,y,z) \mid (x,y,z) \in \cup_{j=1}^{n_{k-1}} B_j^{k-1} \wedge x \in \cup_{t \in I} N_t^{k-1} \wedge y \notin \cup_{t \in I} N_t^{k-1} \}$.

Definition 2.2.7 *Let* $D = \{ 1, 2, \ldots, n_{k-1} \}$ *where* $n_{k-1} = |\Psi^{k-1}|$. *Suppose level k subgraph* SG_i^k (V_i^k, E_i^k) *is induced by all nodes in level k-1 subgraphs* SG_t^{k-1}, $\forall$ $t \in I$ *where* $I \subset D$. *Then,* $W_i^k = \{ (x,y,f_z(x,y)) \mid (x,y) \in (N_i^k \times N_i^k) \wedge (x \xrightarrow{f_z(x,y)} y$ *in* $\cup_{t \in I}(W_t^{k-1} \cup B_t^{k-1}) \wedge x \neq y \}$.

By taking advantages of the above three definitions, we can efficiently obtain values of different levels of boundary node, *within*, and *between* edge sets. Based on *within* and *between* edge sets, the formal definition of *HiTi* graph is given as below:

Definition 2.2.8 *A HiTi graph is a directed labeled graph defined in terms of between and within edges associated with all nodes of subgraph tree ST. A HiTi graph defined over level k+1 subgraph tree ST is called level k HiTi graph. It is represented by* $H^k(P^k,A^k)$ *where* $P^k = \{ x \mid (x,y,z) \in \cup_{i=1}^{k}\cup_{j=1}^{n_i} B_j^i \} \cup \{ y \mid (x,y,z) \in \cup_{i=1}^{k}\cup_{j=1}^{n_i} B_j^i \}$ *and* $A^k = \{ (x,y,[l,W,f_z(x,y)]) \mid (x,y,f_z(x,y)) \in \cup_{j=1}^{n_l} W_j^l \text{ for } l = 1,k \} \cup \{ (x,y,[l,B,z]) \mid (x,y,z) \in \cup_{j=1}^{n_l} B_j^l \text{ for } l = 1,k \}$.

Note that the symbols *"l"* and *"W"*(*"B"*) in the definition of an edge in A^k represent the edge is level 1 *within*(resp. *between*) edge. An example of level 2 *HiTi* graph corresponding to the digraph in Figure 2.4 is shown in Figure 2.5.

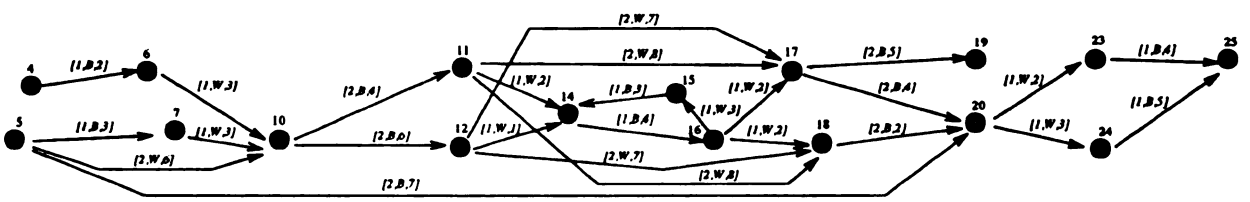


Figure 2.5: A level 2 HiTi graph corresponding to the digraph in Figure 2.4

We have introduced basic concepts and formal definitions of a level *k HiTi* graph in this section. It is easy to see that the overhead of a level *k HiTi* graph comes from the *within* edges in sets $\cup_{i=1}^{k}\cup_{j=1}^{n_i} W_j^i$. Thus, for the *HiTi* graph model to be practical, we introduce one requirement. That is, $|\cup_{i=1}^{k}\cup_{j=1}^{n_i} W_j^i|$ should be small compared with $|E|$ in $G(V,E)$. In practice, $|\cup_{i=1}^{k}\cup_{j=1}^{n_i} W_j^i|$ is likely to be very small. This is because $|W_j^i|$ can not be greater than $|N_j^i| \times |N_j^i|$ where the boundary nodes of SG_j^i are usually quite small compared to $|V_j^i|$.

2.3 Use of HiTi Graphs for Computing Shortest Route

In this section, we discuss the use of *HiTi* graphs for computing a minimum cost path. It is a simple bookkeeping problem to keep track of nodes on a minimum cost path; hence we will focus on an algorithm for simply finding a minimum cost of the path. We first introduce a set of basic notations which will be used in the rest of this paper.

Definition 2.3.1 *Let ST represent a level k subgraph tree. Assume set X consist of a set of subgraphs (i.e. nodes) of ST. Then,* $S_A^l(X) = \{\ y \mid y$ *is a level l subgraph which is an ancestor of each subgraph in X* $\}$, $S_A(X) = \cup_{l=1}^{k} S_A^l(X)$, *and* $S_C(X) = \{\ Y \mid Y$ *is a direct child node of a subgraph in X* $\}$. *Note that* $S_C(X) = X$ *for all leaf node subgraphs* SG_i^1 *in X.*

Definition 2.3.2 *Assume set X consist of a set of subgraphs. Then,* $S_B(X)$ *and* $S_W(X)$ *consist of* **between** *and* **within** *edge sets associated with all the subgraphs in X respectively.* $S_N(X)$ *consists of* **boundary** *node sets associated with all subgraphs in X.*

Definition 2.3.3 *Let* SG_i^1 *and* SG_j^1 *be two distinct level 1 subgraphs defined in a level k subgraph tree ST. Then,* $LUB_{SG}(SG_i^1, SG_j^1)$ *is the least leveled common ancestor subgraph of* SG_i^1 *and* SG_j^1.

The examples of the definition 2.3.1, 2.3.2 and 2.3.3 are illustrated through the level 4 subgraph tree shown in Figure 2.6. Assume that $X = \{ SG_8^1, SG_{11}^1 \}$. Then, $S_A^1(X) = \{ SG_8^1, SG_{11}^1 \}$, $S_A^2(X) = \{ SG_4^2, SG_5^2 \}$, $S_A(X) = \{ SG_8^1, SG_4^2, SG_2^3, SG_1^4, SG_{11}^1, SG_5^2 \}$, $S_C(X) = \{ SG_8^1, SG_{11}^1 \}$, $S_C(S_A^2(X)) = \{ SG_8^1, SG_9^1, SG_{10}^1, SG_{11}^1, SG_{12}^1 \}$, $S_B(X) = \{ B_8^1, B_{11}^1 \}$, $S_W(X) = \{ W_8^1, W_{11}^1 \}$, $S_N(X) = \{ N_8^1, N_{11}^1 \}$, and $LUB_{SG}(SG_8^1, SG_{11}^1)$ is SG_2^3.

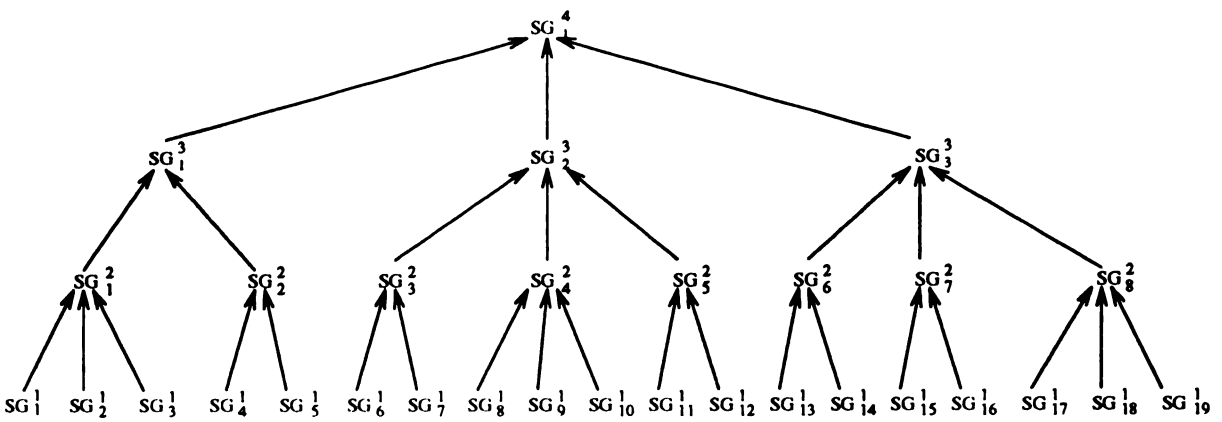


Figure 2.6: An example of level 4 subgraph tree ST

2.3.1 Shortest Path Algorithm SPAH

We now describe a Shortest Path Algorithm based on *HiTi* graph (*SPAH*). *SPAH* takes advantage of *HiTi* graphs to speed up the computation of a minimum cost path. Our algorithm explores at most *ES* search space necessary for computing a shortest cost path. The size of *ES* is shown in the The following theorem.

Theorem 2.3.1 *Let ST represent a level k+1 subgraph tree where level k HiTi graph $H^k(P^k, A^k)$ is defined. Assume that we want to compute a minimum cost path from $START$ node in SG_i^1 to $DEST$ node in SG_j^1 Then, the explored search space ES is at most the union of E_i^1, E_j^1 and $A'^k \subseteq A^k$ where $A'^k = S_B(S_C(S_A(\{SG_i^1, SG_j^1\}))) \cup S_W(S_C(S_A(\{SG_i^1, SG_j^1\})))$.*

Proof: Let l be the level number of $LUB_{SG}(SG_i^1, SG_j^1)$. Furthermore, let $ES_1 = E_i^1 \cup E_j^1 \cup S_B(S_C(\cup_{n=1}^{l} S_A^n(SG_i^1, SG_j^1))) \cup S_W(S_C(\cup_{n=1}^{l} S_A^n(SG_i^1, SG_j^1)))$ and $ES_2 = S_B(S_C(\cup_{n=l+1}^{k+1} S_A^n(LUB_{SG}(SG_i^1, SG_j^1)))) \cup S_W(S_C(\cup_{n=l+1}^{k+1} S_A^n(LUB_{SG}(SG_i^1, SG_j^1))))$. Then, we can represent ES as a union of ES_1 and ES_2. From ES, it is easy to see that at least ES_1 search space is necessary to compute a minimum cost path from the nodes $START$ and $DEST$. Hence, we only need to show that ES_2 is also a necessary search space. Assume that the shortest cost path from the nodes $START$ to $DEST$ is $x_0 = START \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_m \rightarrow x_{m+1} \rightarrow x_{m+2} \rightarrow x_{m+3} \rightarrow \cdots \rightarrow x_{m+q} = DEST$. It is possible that $(\{x_m\} \cup \cup_{p=m+2}^{m+q}\{x_p\}) \subset S_N(S_C(\cup_{n=1}^{l} S_A^n(SG_i^1, SG_j^1)))$ and $x_{m+1} \in S_N(S_C(\cup_{n=l+1}^{k+1} S_A^n(LUB_{SG}(SG_i^1, SG_j^1))))$ where $(x_m, x_{m+1}, [\zeta, B, z])$ and $(x_{m+1}, x_{m+2}, [\zeta, B, z])$ are level ζ *between* edges for some ζ such that $1 \leq \zeta \leq k$. □

From theorem 2.3.1, it is easy to see that how *HiTi* graphs can significantly reduce the search space. That is, without using *HiTi* graph, ES would be $\cup_{j=1}^{n_1} E_j^1$ where n_1 represents a total number of level 1 edges sets. Furthermore, theorem 2.3.1 allows more controlled storage management that is suitable for navigation systems (e.g.

automobile navigation system) where available storage is limited. This is possible because we do not need to have all level 1 edge sets to compute a minimum cost path. Instead, we only need a part of the level k *HiTi* graph and two level 1 edge sets (i.e. one for START node and the other for DEST node). Note that the major storage overhead of a topographical road map comes from the size of level 1 edge sets.

We take a level 4 subgraph tree in Figure 2.6 to give an example of theorem 3.1. Assume that we wants to find a minimum cost path from node $START$ in SG_1^1 to node $DEST$ in SG_{18}^1. Then, our search space ES consists of $E_1^1 \cup E_{18}^1 \cup S_B(S_C(S_A(SG_1^1)))$ $\cup\ S_W(S_C(S_A(SG_1^1)))\ \cup_{j=6}^{8}(S_B(SG_j^2) \cup S_W(SG_j^2))\ \cup_{j=17}^{19}(S_B(SG_j^1) \cup S_W(SG_j^1))$.

Within the search space ES, $SPAH$ traverses edges from node $START$ in SG_i^1 to node $DEST$ in SG_j^1 to find a minimum cost (i.e. with respect to cost z) path. Its edge traversal consists of two phases, the *ascending* and *descending* phases. The *ascending* phase corresponds to the period of edge traversal from node $START$ to the boundary nodes in $S_N(S_A^{l-1}(SG_j^1))$ where l is a level number of $LUB_{SG}(SG_i^1, SG_j^1)$. Similarly, the *descending* phase corresponds to the period of edge traversal from the nodes in $S_N(S_A^{l-1}(SG_j^1))$ to node $DEST$. During the *ascending* and *descending* phases, $SPAH$ traverses edges in a non-decreasing and non-increasing edge level order respectively. Note that each of the considered paths has its own thread of processing. Some of the paths are still in the ascending phase, whereas the others are in the descending phase. The following proposition 2.3.1 provides the theoretical basis of the detailed ascending method given in $SPAH$.

Proposition 2.3.1 *Given a level k HiTi graph, assume that l be the level number of* $LUB_{SG}(SG_i^1, SG_j^1)$ *where node* $START \in V_i^1$ *and* $DEST \in V_j^1$. *Assume that level* μ *to* τ *edges are incident on node* x *such that* $x \in (V_i^1 \cup S_N(S_C(S_A(SG_i^1))))$ *and* $0 \leq \mu \leq \tau$. *In order to find the shortest path from node* x *to any node in* $S_N(S_A^{l-1}(SG_j^1))$, *it is enough to follow only level* ρ *edges from node* x *where* $min(l-1, \tau) \leq \rho \leq \tau$.

Proof: Since level μ to τ edges are incident on node x, we only need to show that it is not necessary to follow level μ to $min(l-1, \tau) - 1$ edges from node x. Then, we need to consider two cases when $l - 1 \leq \tau$ and $l - 1 > \tau$. Let $SG_a^l - 1$ be $S_A^{l-1}(SG_j^1)$.

- **Case** $l - 1 \leq \tau$: In this case, we need to show that we do not have to visit the nodes adjacent to x by following level μ to $l - 2$ edges for the shortest path computation within SG_a^{l-1} from node x to the boundary nodes of SG_a^{l-1}. Since level $l - 1$ edges are incident on node x, node x is in $S_N(SG_a^{l-1})$. From the definitions of W_a^{l-1}, it is true that W_a^{l-1} contains all the shortest path connections between the boundary nodes of SG_a^{l-1} within SG_a^{l-1}. Thus, all shortest paths between node x and y within SG_a^{l-1} are also captured through level $l - 1$ *within* edges $(x, y, [l-1, W, z]) \in W_a^{l-1}$ where $y \in S_N(SG_a^{l-1})$, which proves this case.

- **Case** $l - 1 > \tau$: In this case, we need to show that we do not have to visit the nodes adjacent to x by following level μ to $\tau - 1$ edges for the shortest path computation within SG_a^{l-1} from node x to the boundary nodes of SG_a^{l-1}. Since level τ edges are incident on node x, node x is in $S_N(SG_b^\tau)$ where SG_b^τ is a descendant subgraph of SG_a^{l-1} in ST. Then, it is clear from the definition of

HiTi graphs that all paths generated from node x by following level μ to $\tau - 1$ edges must pass through level τ boundary node(s) of SG_b^τ. From the definitions of W_b^τ, it is true that W_b^τ contains all the shortest path connections between the boundary nodes of SG_b^τ within SG_b^τ. Thus, all shortest paths from node x to y within SG_b^τ are also captured through level τ *within* edges $(x, y, [\tau, W, z])$ $\in W_b^\tau$ where $y \in S_N(SG_b^\tau)$, which proves this case.

□

We take Figure 2.3, 2.4, and 2.5 to exemplify proposition 2.3.1. Assume that we want to find the shortest path from node 5 in SG_1^1 to node 25 in SG_6^1. Since $LUB_{SG}(SG_1^1, SG_6^1) = 3$, $l = 3$. Thus, we have $S_N(S_A^2(SG_6^1)) = N_3^2 = \{19,20,25\}$. From Figure 2.4 and 2.5, we can see that three edges are incident on node 5. They are: one level 1 *between* edge $(5, 7, 3)$, one level 2 *within* edge $(5, 10, 6)$, and one level 2 *between* edge $(5, 20, 7)$. Then, by the above proposition, in order to find the shortest path from node 5 to any node in {19,20,25}, we do not need to follow the level 1 *between* edge $(5, 7, 3)$ from node 5.

After obtaining a minimum cost path from $SPAH$ shown in Figure 2.7, a navigator (e.g. driver) may want to find a more fine-grained path connection on some edges (i.e. high level edges) with a level number greater than 0. This can be accomplished by specializing a high level edge. High level *within* edges are specialized by representing them in terms of lower level edges. For this purpose, we keep actual shortest path information for each corresponding *within* edges. A high level *between* edge cannot

```
begin
Step 1:
  Assume we have a level k HiTi graph defined on a level k + 1 ST;
  Find SG_i^1 and SG_j^1 to which START and DEST belong respectively;
  We maintain G(V, E) together with H^k(P^k, A^k) as adjacency lists;
  Let l be the level number of LUB_SG(SG_i^1, SG_j^1);
  for (p = k; p >= 1; p - -)
    Mark all boundary nodes in S_N(S_C(S_A^p(SG_j^1))) with level p - 1;
Step 2:
  λ(START) = 0;
  FSet = { START }; ESet = ∅;
  while (FSet ≠ ∅) {
    Select u from FSet with minimum λ(u) + f(u, DEST);
    /* the function f(u, DEST) estimates the cost of
       the shortest path from node u to DEST */
    FSet = FSet - { u }; ESet = ESet ∪ { u };
    if (u = DEST) stop;
    Let τ be the highest level number of the edges incident on node u;
    if (u is NOT marked) {      /* Start Ascending Phase */
      for each level ρ where min(l - 1, τ) ≤ ρ ≤ τ {
        for each edge u --z--> v with level ρ {
          if ((v ∉ FSet) and (v ∉ ESet)) {
            λ(v) = λ(u) + z;
            FSet = FSet ∪ { v }; }
          else {
            if (λ(v) > λ(u) + z) λ(v) = λ(u) + z; } } } }
    else { /* u is marked */      /* Start Descending Phase */
      Let β be the marked level number on node u;
      for each edge u --z--> v with the levels from β to τ {
        if ((v ∉ FSet) and (v ∉ ESet)) {
          λ(v) = λ(u) + z;
          FSet = FSet ∪ { v }; }
        else {
          if (λ(v) > λ(u) + z) λ(v) = λ(u) + z; } } } }
end
```

Figure 2.7: SPAH: Finding a Minimum Cost Path from $START$ to $DEST$

be specialized in terms of lower level edges. This is because all *between* edges in $\cup_{i=1}^{k}\cup_{j=1}^{n_i} B_j^i$ are the edges in $G(V, E)$.

2.4 Performance Analysis

The algorithm $SPAH$ can be classified as a variation of A* algorithm in that it uses the function $f(u, DEST)$ to estimate the cost of the shortest path from node u to $DEST$. In the domain of road maps, the function $f(u, DEST)$ computes the Euclidean distance between the node u and $DEST$. This is possible because the coordinates (i.e. longitude and latitude) of all nodes on a road map are assumed to be available. Assuming $(u.x, u.y)$ and $(DEST.x, DEST.y)$ are the corresponding coordinates of the nodes u and $DEST$, $f(u, DEST)$ computes $\sqrt{(DEST.x - u.x)^2 + (DEST.y - u.y)^2}$. Since Euclidean distance is not an over-estimated cost between node u and $DEST$, our algorithm finds the optimal shortest path. The detailed proof for the optimality of A* algorithm is found in [30]. The following Figure 2.8 shows A* algorithm which finds the shortest path from the nodes $START$ to $DEST$ on $G(V, E)$. Note that $l(x, y)$ represents the cost associated with each edge $(x, y) \in E$.

A* shortest path algorithm are shown to be more efficient than the breadth-first search single pair shortest path algorithm when the database can fit in main memory [64, 80]. For algorithm $SPAH$, this is the case since we showed in Theorem 2.3.1 that its explored search space is at most ES which is small enough to fit in main

memory.

```
begin
  For each node u ∈ V, λ(u) = ∞;
  Let λ(START) = 0, FSet = {START}, and ESet = ∅;
  while (FSet ≠ ∅) {
    Select a node u in FSet for which λ(u) + f(u, DEST) is minimum;
    FSet = FSet − {u} and ESet = ESet ∪ {u};
    If (u = DEST), then λ(u) is the shortest path cost and Stop;
    else {
      For every edge (u, v) in E, if λ(v) > λ(u) + l(u, v) {
        Let λ(v) = λ(u) + l(u, v);
        Let FSet = FSet ∪ {v} if v ∉ (FSet ∪ ESet); } } }
end
```

Figure 2.8: A* Algorithm: Finding a Minimum Cost Path from $START$ to $DEST$

For our empirical analysis of Algorithm $SPAH$, we create two dimensional grid graphs $G(V, E)$ with 4 adjacent nodes. Two dimensional grid graphs are considered as typical examples of road maps [64, 91]. In grid graph G, $|V|$ and $|E|$ are equal to 800×800 nodes and $4 \times 800 \times 799$ directed edges. From $G(V, E)$, we create a level 4 subgraph tree ST where each level 1 subgraph $SG_i^1(V_i^1, E_i^1)$ has $|V_i^1| = 100 \times 100$, $|E_i^1| = 4 \times 100 \times 99$. Thus, the level 4 subgraph tree ST consists of 64 level 1, 16 level 2, 4 level 3, and 1 level 4 subgraphs, which are shown in Figure 2.9.

We use the above level 4 subgraph tree ST to generate level 3 $HiTi$ graph for the empirical analysis of $SPAH$ in the following subsections.

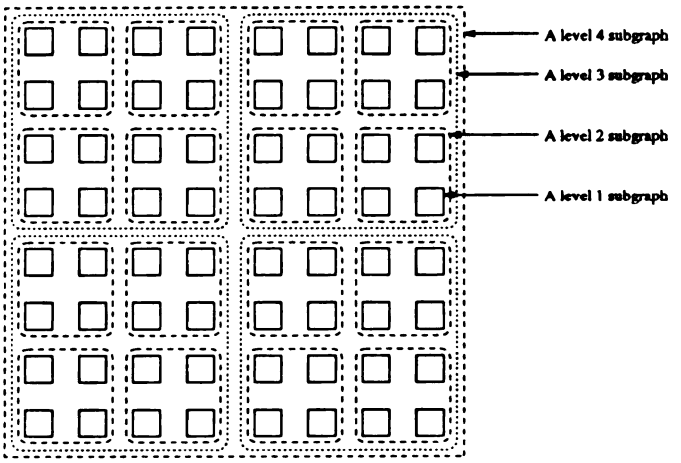


Figure 2.9: A 800 × 800 grid graph partitioned according to the level 4 ST

2.4.1 Comparison between *SPAH* and A* algorithm

To show the search space savings of *SPAH* over the traditional A* algorithm (presented in [91]), we create 5 level 3 *HiTi* graphs where $10 \leq |N_i^1| \leq 20$ and where the edge cost is generated based on a uniform distribution [100,120] with 5 different seeds. We represent $|N_i^1|$ as the total number of boundary nodes defined on level 1 subgraph SG_i^1. Next, we create 5 plain grid graphs which are simply unions of PE^1 and level 1 subgraphs used in the above 5 level 3 *HiTi* graphs. For each level 3 *HiTi* graph and plain grid graph created above, we compute 20 different shortest paths randomly prefixed pairs of source and destination nodes. Let H_n and A_n be the total number of edges visited by *SPAH* and A* respectively. Note that throughout this paper, we multiply the estimation of $f(u, DEST)$ by 100 to normalize the estimation with respect to the edge cost. The number 100 is used for this normalization because the Euclidean distance between two adjacent nodes is 1 and the edge cost between

two nodes is at least 100. We compare $SPAH$ and A* by observing the ratio A_n/H_n. these values are then averaged over the 5 different level 3 $HiTi$ and plain grid graphs with the same source and destination nodes. They are shown in Figure 2.10 where the numbers on x axis represent 20 ordered $< source, destination >$ pairs with path length increasing from 1 to 20.

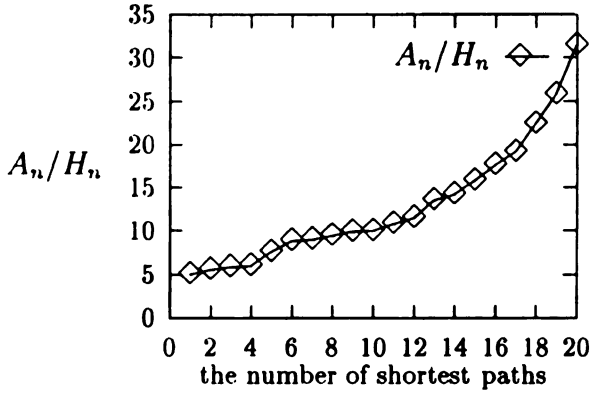


Figure 2.10: Performance comparison between A* and Algorithm $SPAH$

Figure 2.10 clearly shows how effectively Algorithm $SPAH$ cuts down the search space over the traditional A* algorithm. It is interesting to observe that the ratio A_n/H_n increases rapidly as the path lengths from source to destination increase. This occurs because the search space A* needs to explore grows exponentially whereas that of $SPAH$ grows very slowly due to the hierarchical structure of $HiTi$ graphs.

2.4.2 Effects of edge cost distribution

In this section, we studied the effects of edge cost distributions on the performance of $SPAH$. For this study, we generate 4 × 5 (i.e. 4 uniform distributions with 5 seeds) level 3 $HiTi$ graphs where $10 \leq |N_i^1| \leq 20$. Note that the 4 uniform distribu-

tions [100,120], [100,200], [100,300], and [100,500] correspond to 20%, 100%, 200%, and 400% variations of edge costs respectively. We apply $SPAH$ to each level 3 $HiTi$ graph by randomly creating 50 different $< source, destination >$ pairs and then averaging the cost. Let MA and MD symbolize $SPAH$ when the estimator $f(u, DEST)$ gives Euclidean distance and zero (i.e. no optimization based on Euclidean distance) respectively. Figure 2.11 shows the effect of edge cost distributions on the performance of $SPAH$ in terms of H_n (i.e. the total number of visited edges) of MA and MD. Note that the values of H_n are averaged over 5 seeds.

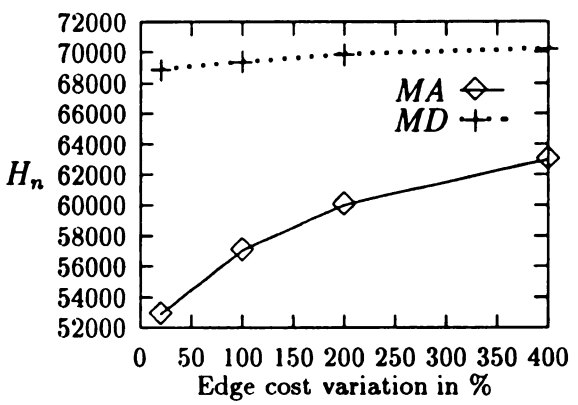


Figure 2.11: Effect of edge cost on the performance of MA and MD

When $f(u, DEST)$ gives Euclidean distance estimation, the performance of $SPAH$ deteriorates as the variation of edge cost is increased. The primary reason is that increasing the variation degrades the quality of Euclidean distance estimation to shortest path. This degradation of Euclidean distance estimation seems to have more severe impact on the performance of MA for the first part of edge cost variation (i.e. between 20% and 200%) than for the rest (i.e. between 200% and 400%). Unlike MA, MD gives a very stable performance with varying edge cost distribution.

2.4.3 Effects of the number of hierarchical levels

We examined the effects of different levels of *HiTi* graphs on the performance of *SPAH* (i.e. *MA* and *MD*). We constructed different levels of *HiTi* graph out of the same set of level 1 subgraphs. For this analysis, we create 2 (i.e. $4 \leq |N_i^1| \leq 8$ and $10 \leq |N_i^1| \leq 20$) $\times$ 5 (i.e. an edge cost distribution [100,200] with 5 seeds) level 3 *HiTi* graphs. Similarly, we create 2 $\times$ 5 level 1 and level 2 *HiTi* graphs. Then, we measure average H_n of *MA* and *MD* the same way as we did in section 3.2.2. They are shown in Table 3.1 and 3.2.

	level 1 *HiTi* graph	level 2 *HiTi* graph	level 3 *HiTi* graph
H_n for MA	57428	57336	57063
H_n for MD	72227	71953	69392

Table 2.3: Effects of levels of *HiTi* graphs on H_n when $10 \leq |N_i^1| \leq 20$

	level 1 *HiTi* graph	level 2 *HiTi* graph	level 3 *HiTi* graph
H_n for MA	50650	50370	50669
H_n for MD	59684	59642	60114

Table 2.4: Effects of levels of *HiTi* graphs on H_n when $4 \leq |N_i^1| \leq 8$

As we can see from table 2.3 and 2.4, higher level *HiTi* graphs do not necessarily guarantee the better performance when using *SPAH* than the lower level *HiTi* graphs. It depends on the the entire search space for *SPAH*. The entire search space for *SPAH* on a level k *HiTi* graph, represented by Υ^k, is formulated as follows:

$$\Upsilon^k = 2 \cdot |E^1| + \sum_{i=1}^{n_k} |W_i^k| + \frac{2}{n_k} \sum_{i=1}^{n_{k-1}} |W_i^{k-1}| + \frac{2}{n_{k-1}} \sum_{i=1}^{n_{k-2}} |W_i^{k-2}| + \cdots + \frac{2}{n_3} \sum_{i=1}^{n_2} |W_i^2| + \frac{2}{n_2} \sum_{i=1}^{n_1} |W_i^1| \quad for\ l = 1, k$$

Note that $|E^1|$ represent the total number of edges of a level 1 subgraph and n_l represent the total number of level l subgraphs where $1 \leq l \leq k$. If $\Upsilon^p < \Upsilon^q$ where $p > q$, then $SPAH$ is likely to perform better on the higher level p *HiTi* graph than on the lower level q *HiTi* graph. Otherwise, the higher level *HiTi* graph is not likely to provide a performance advantage over the lower level *HiTi* graph. Table 2.5 shows the values of Υ^1, Υ^2, and Υ^3 corresponding to level 1,2, and 3 *HiTi* graphs used in the tables 2.3 and 2.4.

	Υ^1	Υ^2	Υ^3
$10 \leq \lvert N_i^1 \rvert \leq 20$	99000	98737	98487
$4 \leq \lvert N_i^1 \rvert \leq 8$	82272	83364	83332

Table 2.5: Effects of levels of *HiTi* graphs on Υ^k

Table 2.5 verifies our conjecture on the performance of $SPAH$ for different levels of *HiTi* graphs. An interesting thing to note from the tables 2.3 and 2.4 is that the search space (i.e. H_n) does not vary significantly going beyond level 1 *HiTi* graph. This is because our experiments were done with the grid graphs having the following property. That is, the difference between Υ^i and Υ^{i+1} does not vary significantly as i increases. We believe that for most road maps this is the case. As a result, creating

higher level *HiTi* graphs does not contribute to the reduction in computation time. From this observation, we can conclude that level i *HiTi* graph is good enough for road map applications where the difference between Υ^i and Υ^{i+1} is small.

2.5 Updating HiTi Graph

There are two different types of updates for a recursive relation $G(V, E)$ (e.g. topographical road map). They are *edge addition*, and *edge deletion*. An edge addition (deletion) can connect (resp. disconnect) two existing nodes belonging to either a level 1 subgraph or two different level m subgraph where $1 \leq m \leq k$.

Since updating $G(V, E)$ may require modifying level k *HiTi* graph, special update problems associated with *HiTi* graph will need to be addressed. In the following subsections, we will discuss the update problems on level k *HiTi* graph.

2.5.1 Edge Deletion

Suppose edge (x, y, z) is deleted from E. Then, we have two possible cases to consider. The first case case is when (x, y, z) is in level m *between edge set* where $1 \leq m \leq k$. The second case is when edge (x, y, z) is in E_i^1 where $1 \leq i \leq n_1$.

The first case may change boundary nodes (i.e. x and y) to non-boundary nodes and it may invalidate existing *within* edges. Boundary node x (y) becomes non-boundary nodes if there is no other level m *between* edges incident on node x (resp.

y). Assume that node x and y become non-boundary nodes. We need to delete the nodes x and y from their corresponding *boundary* node sets. For this purpose, we first need to identify two level 1 subgraphs SG_i^1 and SG_j^1 where $x \in S_N(SG_i^1)$ and $y \in S_N(SG_j^1)$. Then, we delete node x from $S_N(\cup_{l=1}^m S_A^l(\{SG_i^1\}))$ and node y from $S_N(\cup_{l=1}^m S_A^l(\{SG_j^1\}))$. Next, we find all existing invalid *within* edges $(a, b, f_z(a, b))$ in $S_W(S_A(\{SG_i^1, SG_j^1\}))$. A *within* edge $(a, b, f_z(a, b))$ becomes invalid in the following three cases.

case 4.1.1 Nodes a or b is x or y.

case 4.1.2 A path from node a to b is disconnected.

case 4.1.3 The value of $f_z(a, b)$ is not correct.

For the case 4.1.1 and 4.1.2, we delete $(a, b, f_z(a, b))$ from the corresponding *within* edge set. In the case 4.1.3, simply replace the incorrect value of $f_z(a, b)$ with a correct one.

In the second case, the deletion of edge (x, y, z) from E_i^1 may invalidate some existing *within* edges in $S_W(S_A(\{SG_i^1\}))$. Thus, we need to identify all the existing invalid *within* edges. After the identification of all invalid *within* edges, we perform the same actions as we did in the cases 4.1.2 and 4.1.3.

2.5.2 Edge Addition

Suppose a new edge (x, y, z) is added to $G(V, E)$. Then, there are two cases we need to consider. The first case is that edge (x, y, z) becomes a new level m *between* edge

where $1 \leq m \leq k$. The second case is that edge (x, y, z) is added to E_i^1.

The first case changes non-boundary node x (y) to boundary node if there is no other level m *between* edges incident on node x (resp. y). Assume that node x and y become new boundary nodes. Then, it is necessary to add node x and y to their corresponding *boundary* node sets. For this purpose, we need to identify two level 1 subgraph SG_i^1 and SG_j^1 where $x \in V_i^1$ and $y \in V_j^1$. Then, we add node x to $S_N(\cup_{l=1}^m S_A^l(\{SG_i^1\}))$ and node y to $S_N(\cup_{l=1}^m S_A^l(\{SG_j^1\}))$. Next, we add edge (x, y, z) to level m *between* edge set $S_B(S_A^m(\{SG_i^1\})$. The addition of this new level m *between* edge may cause the following cases in $S_W(S_A(\{SG_i^1, SG_j^1\}))$.

case 4.2.1 Need to create new level m_1 ***within*** edges where $1 \leq m_1 \leq k$.

case 4.2.2 Need to update $f_z(a, b)$ of some existing level m_1 ***within*** edges where $1 \leq m_1 \leq k$.

In the second case, the addition of edge (x, y, z) to E_i^1 may cause the same cases as the cases 4.2.1 and 4.2.2 for edges in $S_W(S_A(\{SG_i^1\})$.

2.6 Conclusion

In this chapter, we developed a new graph, called a *HiTi* graph, to model very large topographical road maps. *HiTi* graphs provides a powerful formal framework for structuring topographical road map data in a hierarchical fashion. We have empirically shown that our proposed shortest path algorithm, based on *HiTi* graphs, significantly reduces the search space for computing the minimum cost path over a

very large topographical road map. Our algorithm is also empirically analyzed by varying edge cost distributions and the number of levels of *HiTi* graphs. Finally, we have investigated the problems of updating a *HiTi* graph in this paper. The applications of *HiTi* graph structure go beyond the domain of topographical road maps. It can be applied to any very large recursive relations where hierarchical abstractions of data are useful.

In addition to SPSP problem discussed in this chapter, there are quite a few other database research problems associated with the automobile navigation system. They include data models for storing large amounts of road map data, the determination of current location in the database, and physical storage structure of road maps. We give our survey on these problems in Appendix C of this thesis.

Chapter 3

HiTi Graph-based Parallel Shortest Path Algorithms

In this chapter, we study for both *intra* and *inter* query parallel processing for SPSP problems. Based on *HiTi* graph structure, we propose two parallel shortest path algorithms named *PASPAH* and *ISPAH* for intra and inter query SPSP problems, respectively. We empirically analyze the performance of *PASPAH* and *ISPAH* algorithms on two-dimensional grid graphs by implementing them on BBN GP1000 shared memory multiprocessor system. The analysis of *PASPAH* shows that the average execution time of *PASPAH* is not significantly better than those of *SPAH* and *MOTOpar*. From these performance, we conjecture that *inter* query shortest path problem has more potential for the parallel processing than *intra* query shortest path problem.

3.1 Introduction

A Single Pair Shortest Path (SPSP) computation is fundamental to topographical road map queries. In order to reduce the large search space of topographical road maps, a graph traversal approach is usually used. Although the graph traversal approach reduces the search space significantly, its search space is still large for most real world road maps. To cope with this limitation, two new approaches are proposed [5, 59]. Agrawal and Jagadish [5] studied SPSP computation in the context of problem of performing efficient search over disk-resident massive graphs. Jung and Pramanik [59] proposed a new graph model, named *HiTi* graph model for this problem. These two methods narrow down the search space very significantly at the cost of storing the partially precomputed path information. However, their performance improvement is still limited to a single processor case. For interactive applications, further reduction in response time is beneficial. Thus, efficient parallel SPSP algorithms have been developed to improve the performance further.

Little research has been done on parallel SPSP algorithms. Mohr and Pasche [72] presented a new parallel shortest path algorithm named *OTOpar* which is a parallel implementation of *OTO* algorithm. *OTOpar* uses two processors where each processor uses A* algorithm with Manhattan distance estimation to build its corresponding tree, one rooted at the source node and the other rooted at the destination node. Their empirical analysis shows that the average execution time of *OTOpar* is roughly half of *OTO*. However, *OTOpar* algorithm is not scalable at all. That is, their performance improvement is limited by only two processors. Thus, we need more scalable parallel

SPSP algorithms than *OTOpar*.

In this chapter, we study the parallel processing for *intra* as well as *inter* query shortest path problems. *Intra* query SPSP problem deals with parallelizing a single transaction of SPSP problem. The parallel SPSP algorithm proposed by Mohr and Pasche [72] belongs to this category. *Inter* query SPSP problem deals with parallelizing *multiple* single-pair shortest-path computations. *Inter* query SPSP problem arises in the domain of automobile navigation systems where many vehicles send their shortest route computation requests to a central server. Then the server must be able to handle these multiple SPSP computation requests satisfying a real time constraint.

We have developed two parallel SPSP algorithms. One is for *intra* query SPSP problem and the other for *inter* query SPSP problem. Both algorithms are based on *HiTi* graph introduced in Chapter 2. *HiTi* graph structure provides the opportunity for fine-grained parallel processing for *intra* query SPSP problem. This is because multiple parallel shortest-path computations can be initiated based on the nodes at a higher level of abstraction of a topographical road map. Note that *HiTi* graph structure can capture a hierarchical abstraction of the topographical road map and it is easy to identify the appropriate nodes at the higher level.

Even though we have done extensive analyses on parallel processing of *HiTi* graph, we also have explored the possibilities of using *HiTi* graph in a distributed environment. In a distributed environment, *HiTi* graph provides an efficient structure for computing a distributed transitive closure. That is, it allows a dynamic decomposition of a transitive closure computation. This dynamic decomposition in turn allows

us to minimize the communication costs between the sites involved for the transitive closure computation. Note that reducing the communication costs is a major optimization goal for distributed transitive closure computations. We give the detailed description of our proposed distributed transitive closure algorithm in the appendix A of this thesis.

The rest of the chapter is organized as follows. In section 3.2, we give the detailed description of *intra* query parallel shortest path algorithm and empirically compare its performance with those of the previous work. Section 3.3 discusses two *inter* query parallel shortest path algorithms. We give empirical performance analysis of these two *inter* query parallel algorithms in this section. Finally, section 3.4 gives the concluding remarks.

3.2 Intra Query Parallel Shortest Path Computation

The main performance overhead of the shortest path computation comes from the size of the search space it needs to explore. In this section, we describe a PArallel Shortest PAth algorithm based on *HiTi* graph (PASPAH). To show the basic idea behind *PASPAH*, we first graphically show the explored search space of the algorithms A* and *OTOpar* in Figure 3.1. Note that in the figure, the rectangle and eclipse represent the entire and the explored search space respectively. Arrows indicate the direction of exploring the search space. It is intuitively obvious that A* algorithm

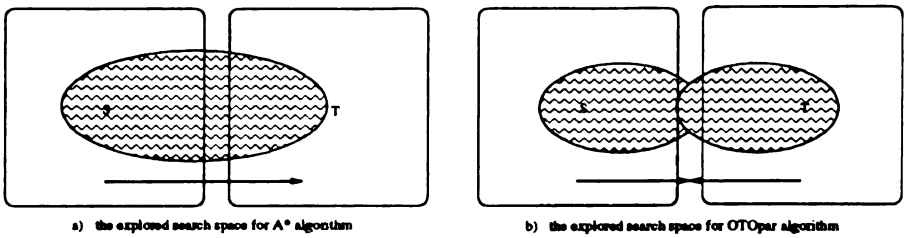


Figure 3.1: Relative size of explored search space for A* and *OTOpar*

explores the search space in an elliptical shape, which is shown in Figure 3.1 a. Since *OTOpar* explores the search space by applying A* algorithm from both the source (i.e. S) and destination (i.e. T) nodes, the size of its explored search space becomes smaller than that of A*. This is clearly illustrated in Figure 3.1 b. Based on this observation, we can infer that, if a middle node M lying on the shortest path is known, we can further narrow down the search space by applying A* algorithm from the middle node to both source and destination, simultaneously. Our *PASPAH* is based on this idea which is illustrated in Figure 3.2.

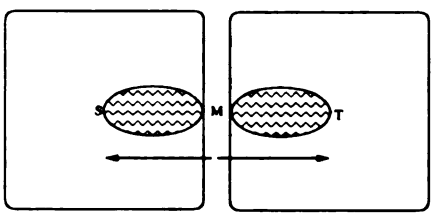


Figure 3.2: Explored search space size of *PASPAH*

3.2.1 Use of HiTi Graph for Identifying Middle Nodes

The problem that remains is how to identify the middle node without computing the shortest path. This problem can be partially resolved by taking advantage of HiTi graph structure, which we explain through Figure 3.3.

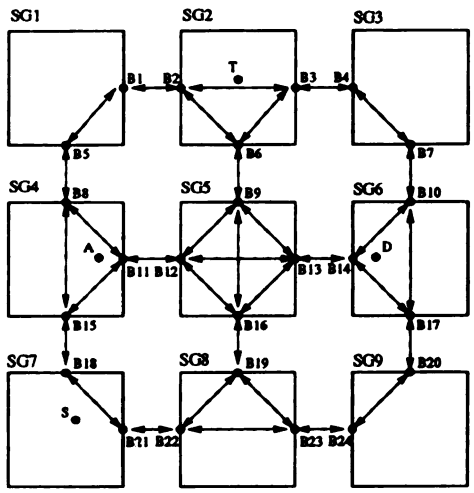


Figure 3.3: Level 1 HiTi graph created from level 1 subgraphs

In Figure 3.3, the rectangles symbolize the level 1 subgraphs $SG1$ to $SG9$ and their boundary nodes are represented by $B1$ through $B24$. Level 1 HiTi graph defined on these level 1 subgraphs is then easily represented by all the boundary nodes interconnected by the dotted arrows. Assume that we want to compute the shortest path from node S in SG_7 to node T in SG_2. It is clear from the figure that the shortest path from the nodes S to T must pass through at least one boundary node of SG_7 and one boundary node of SG_2. From this observation, we can approximate the two possible sets of middle node candidates for the shortest path. They are

$M_1 = \{B18, B21\}$ and $M_2 = \{B2, B3, B6\}$ where set M_1 is chosen in this example.

Then, the shortest path cost from the nodes S to T, represented by $PSP\_COST(S,T)$, can be computed by exploring the search space from each node in M_1 to S and T simultaneously. Let $SP\_COST(X,Y)$ represent the shortest path cost from the nodes X and Y. To obtain $PSP\_COST(S,T)$, we first compute $SP\_COST(B18,S)$, $SP\_COST(B18,T)$, $SP\_COST(B21,S)$, and $SP\_COST(B21,T)$ in parallel. Note that each processor use A* algorithm to compute $SP\_COST(X,S)$ and $SP\_COST(X,D)$ where $X \in M_1$. $PSP\_COST(S,T)$ is then obtained by selecting the minimum of { $SP\_COST(B18,S)$ + $SP\_COST(B18,T)$, $SP\_COST(B21,S)$ + $SP\_COST(B21,T)$ }. Note that we choose M_1 over M_2 since the size of set M_1 is smaller than that of M_2. This way, we can minimize the number of processors involved for the computation.

As it might be noticed from the above example, there is a possible performance bottleneck problem in this approach. We re-take Figure 3.3 to explain the bottleneck in the following example. Assume the shortest path computation from the nodes A in $SG4$ to D in $SG6$. Since the number of the boundary nodes of $SG4$ and $SG6$ are equal, we will randomly choose one of them. In this example, set $\{B8, B11, B15\}$ is selected. Then, in order to compute $SP\_COST(A,D)$, our method will compute $SP\_COST(B8,A)$, $SP\_COST(B8,D)$, $SP\_COST(B11,A)$, $SP\_COST(B11,D)$, $SP\_COST(B15,A)$, and $SP\_COST(B15,D)$ in parallel. As we can see from Figure 3.3, since node A is located a lot closer to node $B11$ than the nodes $B8$ and $B15$, the actual shortest path from A to D is likely to pass through

the boundary node $B11$. In such case, $SP\_COST(B11, A)$ and $SP\_COST(B11, D)$ computations will be done a lot faster than the others.

However, we have to wait until the rest of computations end, although we already got $SP\_COST(B11, A)$ and $SP\_COST(B11, D)$. This waiting time is required to guarantee the optimality of $SP\_COST(B11, A) + SP\_COST(B11, D)$ among the rest of results, which becomes the performance bottleneck of $PASPAH$. To cope with the performance bottleneck, it is necessary to minimize the waiting time. The minimization of the waiting time can be realized if we have a correct and efficient *stopping condition*. The *stopping condition* allows us to decide the optimality of $SP\_COST(B11, A) + SP\_COST(B11, D)$ before completing all the computation of $SP\_COST(B8, A)$, $SP\_COST(B8, D)$, $SP\_COST(B15, A)$, and $SP\_COST(B15, D)$.

3.2.2 Efficient Stopping Criteria

Before we explain *stopping condition*, we first examine how $SP\_COST(B8, A)$ in the above example is computed in A* algorithm. In A*, the search space is explored by permanently labeling a node x *the shortest path cost from* $B8$ *to* x until the next selected node x becomes the node A. The next unlabeled node x is selected from the search space such that $SP\_COST(B8, x) + EuDist(x, A)$ is minimum. Note that $EuDist(x, A)$ is the Euclidean distance estimation from the nodes x to A. Based on this observation, we now formally describe the *stopping condition* in the following proposition.

Proposition 3.2.1 *Given a directed graph $G(V,E)$, assume that the shortest path from the nodes S to T must pass through at least one node from set $B = \{ B_1, B_2, \ldots, B_n \}$. Let set $C \subseteq B$ where $SP\_COST(X,S)$ and $SP\_COST(X,T)$ are already computed for each node $X \in C$ at time t_0. For each $X \in B - C$, let $N(X,S,t_0)$ and $N(X,T,t_0)$ be the next node at time t_0 selected by A* algorithm for the computation of $SP\_COST(X,S)$ and $SP\_COST(X,T)$ respectively. Then, $PSP\_COST(S,T)$ is $cost_0 = \min_{X \in C} \{ SP\_COST(X,S) + SP\_COST(X,T) \}$ if $cost_0 \leq \min_{Y \in B-C} \{ SP\_COST(Y,N(Y,S,t_0)) + EuDist(N(Y,S,t_0),S) + SP\_COST(Y,N(Y,T,t_0)) + EuDist(N(Y,T,t_0),T) \}$.*

Proof: Assume that $cost_0$ is not $PSP\_COST(S,T)$ although $cost_0 \leq \min_{Y \in B-C}$ $\{ SP\_COST(Y,N(Y,S,t_0)) + EuDist(N(Y,S,t_0),S) + SP\_COST(Y,N(Y,T,t_0))$ $+ EuDist(N(Y,T,t_0),T) \}$. Then, at some time $t_1 > t_0$, there must be some node $Z \in B - C$ such that $SP\_COST(Z,S) + SP\_COST(Z,T) <$ $cost_0$. However, $SP\_COST(Z,S) + SP\_COST(Z,T)$ cannot be smaller than $SP\_COST(Z,N(Z,S,t_0)) + EuDist(N(Z,S,t_0),S) + SP\_COST(Z,N(Z,T,t_0)) +$ $EuDist(N(Z,T,t_0),T)$. This is because $SP\_COST(Z,N(Z,S,t)) + EuDist(N(Z,S,t),S)$ and $SP\_COST(Z,N(Z,T,t)) +$ $EuDist(N(Z,T,t),T)$ are always monotonically increasing in A* algorithm where $t_0 < t \leq t_1$. As a result, we have $cost_0 \leq \min_{Y \in B-C} \{ SP\_COST(Y,N(Y,S,t_0))$ $+ EuDist(N(Y,S,t_0),S) + SP\_COST(Y,N(Y,T,t_0)) + EuDist(N(Y,T,t_0),T) \} <$ $SP\_COST(Z,S) + SP\_COST(Z,T) < cost_0$, which is a contradiction. □

To exemplify how proposition 3.2.1 works, we show the *stopping condition* for the computation of $SP\_COST(A, D)$ in Figure 3.3. Assume that $SP\_COST(B11, A)$ and $SP\_COST(B11, D)$ are obtained at time t_0. Then, $PSP\_COST(A, D)$ is $cost_0 = SP\_COST(B11, A) + SP\_COST(B11, D)$ if $cost_0 \leq \min \{ SP\_COST(B8, N(B8, A, t_0)) + EuDist(N(B8, A, t_0), A), SP\_COST(B15, N(B15, D, t_0)) + EuDist(N(B15, D, t_0), D) \}$.

3.2.3 Formal Description of PASPAH

In the previous section, we described the basic idea as well as a theoretical foundation of *PASPAH* when level 1 HiTi graph is given. We now formally present *PASPAH* in Figure 3.4 where we use the same notations as those defined in the previous chapter 2.3. Our *PASPAH* is developed for shared memory multiprocessor systems with the following architecture assumptions:

- All processors have their own local memory.
- All Processors communicate with each others through globally shared memory.
- All processors share disk.

Based on the preceding assumptions, *PASPAH* accesses a level $k+1$ subgraph tree, level k HiTi graph $H^k(P^k, A^k)$, and level 1 subgraphs $\cup_{i=1}^{n_1} SG_i^1(V_i^1, E_i^1)$, which are stored in the globally shared memory.

begin
Step 1:
  for $(j = k + 1;\ j >= 1;\ j - -)$
    Mark all boundary nodes in $S_N(S_C(S_A^j(SG_s^1, SG_t^1)))$ with level $j - 1$;
  Let l be the level number of $LUB_{SG}(SG_s^1, SG_t^1)$;
  for $(j = l - 1;\ j >= 1;\ j - -)$
    Mark all boundary nodes in $S_N(S_C(S_A^j(SG_s^1, SG_t^1)))$ with level $j - 1$;
Step 2:
  If $(|S_N(S_A^{l-1}(SG_s^1))| \leq |S_N(S_A^{l-1}(SG_t^1))|)$ $SG_m^{l-1} = S_A^{l-1}(SG_s^1)$;
  else $SG_m^{l-1} = S_A^{l-1}(SG_t^1)$;
Step 3:
  for each boundary node x in $S_N(SG_m^{l-1})$
    Enqueue (x, S) and (x, T) to BQ;
Step 4:
  while $(|BQ| > 0)$ {
    Dequeue (x, y) from BQ;
    A next available processor performs $SP(x, y)$;
    /* $SP(x, y)$ computes the shortest path from x to y */
    /* It checks *stopping criteria* given in proposition 3.2.1
      when the computation is done */}
end

Figure 3.4: PASPAH: Find a shortest path from S in SG_s^1 to T in SG_t^1

PASPAH consists of four steps. At step 1, the boundary nodes of HiTi graph is marked with edge level numbers. This marking step is necessary to avoid traversing the unnecessary low level edges incident on the marked nodes. When a marked node is visited, the corresponding marked level (i.e. *low_level*) specifies the lowest level of the edges to be traversed from the marked node. Since the highest level (i.e. *high_level*) of the edges incident on the marked node can be easily identifiable, the nodes expanded directly from the marked node are only those reachable through the edges with the levels ranging from *low_level* to *high_level*. This marking step guarantees that the explored the search space of *PASPAH* is at most $E_s^1 \cup E_t^1 \cup S_B(S_C(\cup_{j=1}^{k+1} S_A^j(SG_s^1, SG_t^1))) \cup S_W(S_C(\cup_{j=1}^{k+1} S_A^j(SG_s^1, SG_t^1)))$.

At step 2, we determine the level l subgraph SG_m^{l-1} whose boundary nodes will be used as the middle nodes as explained in the previous section 3.2.1. The step 3 is self explanatory. In step 4, the shortest path computation $SP(x, y)$ for each pairs of nodes (x, y) in BQ is performed in parallel by the available processors. $SP(x, y)$ computes the shortest path from the nodes x to y by using $A*$ algorithm. Note that $A*$ algorithm in $SP(x, y)$ explores the neighbor nodes of the currently expanding node z by following only those edges whose level number is not less than the marked level number of the node z. Whenever each processor finish its computation, it exclusively checks the stopping condition described in the previous section 3.2.2. If the condition is satisfied, then it aborts all the computations being performed by other processors. Otherwise, it simply stops its computation.

3.2.4 Performance Evaluation

We empirically compare *PASPAH* with *OTO*, *OTOpar* [72] and *SPAH* [59] algorithms. *SPAH* is an improved A* algorithm which takes advantage of HiTi graph structure. For fair comparisons, we modified *OTO* and *OTOpar* so that they also utilize HiTi graph structure. In other words, in the modified *OTO* and *OTOpar* named *MOTO* and *MOTOpar* respectively, we use *SPAH* algorithm for building two trees from both source and destination nodes. Note that we use Euclidean distance for the lower bound estimation in *SPAH* rather than Manhattan distance of *OTOpar*. This is because Manhattan distance estimation does not guarantee the optimal shortest path generation.

We implemented *SPAH*, *MOTO*, *MOTOpar* and *PASPAH* on a BBN GP1000 shared memory multiprocessor system, which has a nonuniform memory architecture. The BBN GP1000 multiprocessor currently consists of 85 nodes, each one with 4MBytes of local memory, linked together by a high speed butterfly switch. In this system, the globally shared memory is the sum of the memories local to all processors. Thus, the size of available main memory increases with increasing number of nodes in the system. The BBN GP1000 system can have up to 250 processing nodes.

For our empirical analysis, we create two dimensional grid graphs $G(V, E)$ with 4 adjacent nodes as we did in chapter 2.4. In grid graph G, $|V|$ and $|E|$ are equal to 400×400 nodes and $4 \times 400 \times 399$ directed edges. From $G(V, E)$, we create a level 4 subgraph tree ST where each level 1 subgraph $SG_i^1(V_i^1, E_i^1)$ has $|V_i^1| = 50 \times 50$,

$|E_i^1| = 4 \times 50 \times 49$. The level 2 subgraph tree ST consists of 64 level 1, 16 level 2, 4 level 3, and 1 level 4 subgraphs.

Based on the above level 4 subgraph tree, we create level 1 HiTi graph which will be used throughout this section. To show performance comparison between *PASPAH*, *MOTOpar*, *MOTO*, and *SPAH*, we create 5 level 3 HiTi graphs where $3 \leq |N_i^1| \leq 8$ and where the edge cost is generated based on a uniform distribution [100,120] with 5 different seeds. Note that $|N_i^1|$ is the total number of boundary nodes defined on level 1 subgraph SG_i^1. For each level 1 *HiTi* graph, we compute two different sets SET I and SET II of 20 shortest paths randomly prefixed pairs of source and destination nodes. SET I consists of the pairs whose source node location within the corresponding level 1 subgraph is not far apart from the destination nodes. A simplified example of these two sets is shown in Figure 3.5.

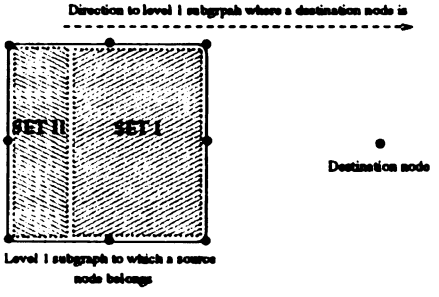


Figure 3.5: An example of sets I and II

We then compare the above 4 algorithms by observing their average execution times of two sets SET I and SET II over 5 different level 1 HiTi graphs. The performance of the four algorithms are given in Figure 3.6 a and b. As it is shown in

Figure 3.6, *PASPAH* requires 42 processors to achieve the maximum parallelism in this empirical analysis. Figure 3.6 a shows that *PASPAH* outperforms the rest of the algorithms. However, Figure 3.6 b shows that *PASPAH* does not perform significantly better than *SPAH* and *MOTOpar*[1]. This is because, when source nodes comes from SET II, the explored search space of *PASPAH* becomes almost same as those of *SPAH* and *MOTOpar*. Thus, the average execution time of *PASPAH* is not significantly better than *SPAH* and *MOTOpar*. From this observation, we conjecture that the parallel processing for *intra* query SPSP problem is not promising.

We have observed that *SPAH* performs better than *MOTO* and *MOTOpar*, which is a different result from the one given in [72]. Our empirical observation can be explained by the following two reasons. The first reason is that *HiTi* graph structure significantly reduces the advantage of using the two tree expansion approach given in [72]. In other words, the most of the nodes in the two level 1 subgraphs (i.e. where source and destination nodes are in) are already explored when two trees start to include the nodes in *HiTi* graph. This conjecture is verified in the appendix B of this thesis where the two tree expansion approach performs far better than than that of one tree expansion approach when *HiTi* graph is not used. The second reason is that the lower bound estimation of *MOTOpar* is Euclidean distance which provides a lot tighter bound than Manhattan distance of *OTOpar* in [72]. As a result, compared with *OTOpar*, *MOTOpar* takes much longer time to stop building the two trees before it finds the shortest path.

[1] In Figure 3.6 b, the minimum execution time of *PASPAH*, *SPAH*, *MOTOpar*, and *MOTO* are 8.36, 8.46, 9.92, and 15.56 seconds respectively.

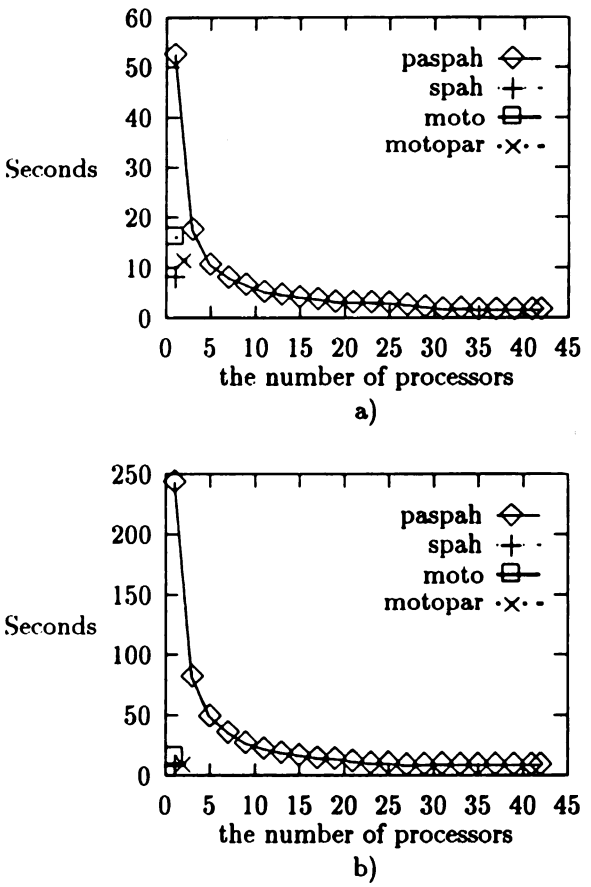


Figure 3.6: Performance of *PASPAH*, *SPAH*, *MOTOpar*, and *MOTO* over: a) SET I b) SET II

3.3 Inter Query Parallel Shortest Paths Computation

In this section, we study the parallel processing for computing *inter* query shortest path based on *HiTi* graph. Efficient and fast computation of multiple SPSP shortest paths are very critical for those automobile navigation systems where all route computations are performed on a single server. For this purpose, we propose a new *inter* query parallel shortest path algorithm in the following subsection.

3.3.1 Formal Description of ISPAH Algorithm

As it was already shown in section 3.2.4, $SPAH$ outperforms *MOTO* and *MOTOpar*. Due to the number of processors $PASPAH$ requires, $PASPAH$ is not appropriate for *inter* query parallel shortest paths computation. We therefore use $SPAH$ as the unit operation for parallelizing *inter* query shortest paths computation. In other words, $SPAH$ is performed in parallel for each shortest path computation request. The detail description of the *inter* query parallel shortest path algorithm, named Inter SPAH (*ISPAH*), is shown in Figure 3.7.

Algorithm *ISPAH* executes $SPAH$ in parallel on a shared memory multiprocessor system. For *SPAH* running on each processor, it accesses both local memory and globally shared memory for the computation. Algorithm *SPAH* accesses the globally shared memory only when it needs to accesses $G(V, E)$ or level k *HiTi* graph $H^k(P^k, A^k)$. Other than that, *SPAH* accesses the local memory. Note that there is

begin
Adjacency lists of $G(V, E)$ and level k *HiTi* graph $H^k(P^k, A^k)$
are stored in the globally shared memory;
Let R contain a set of source and destination nodes pairs (x,y);
for each $(x, y) \in R$ do in parallel {
Find SG_x^1 and SG_y^1 to which x and y belong respectively;
Let l be the level number of $LUB_{SG}(SG_x^1, SG_y^1)$;
for $(p = l - 1; p >= 1; p - -)$
Locally mark all boundary nodes in $S_N(S_C(S_A^p(SG_j^1)))$ with level $p - 1$;
Perform **Step 2** of $SPAH$ for (x, y); }
/* $SPAH$ is given in Figure 2.7 in section 2.3.1 */
end

Figure 3.7: ISPAH algorithm

no memory access contention for reading or writing a local memory. In the following section, we empirically analyze the performance of $ISPAH$.

3.3.2 Performance Evaluation

We implemented $ISPAH$ on a BBN GP1000 shared memory multiprocessor system. For our empirical analysis, we used the same grid graph as described in section 3.2.4. Based on these data sets, we computed 43 randomly prefixed shortest paths. The performance is then measured in terms of the average *speedup* T_n, where T_n represents the total time taken by n processors to compute M shortest paths. We express the *speedup* S_n of n processors as T_1/T_n. Figure 3.8 shows S_n as n is increased from 1 to 43.

As we can see from Figure 3.8, the speedup of $ISPAH$ increases almost linearly up to 10 processors, after 10 it is beginning to level off, and after 25 processors,

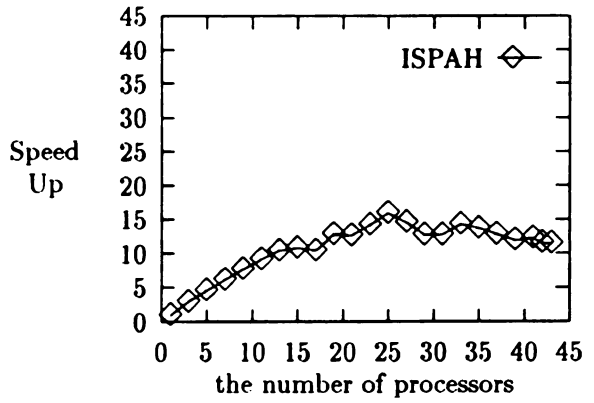


Figure 3.8: Performance of ISPAH

the performance is ever deteriorating. This occurs because of the globally shared memory access contentions. In $ISPAH$, all processors have to access a single $HiTi$ graph in a globally shared memory, which causes a severe memory access contention when more than 25 processors are used. In order to verify our conjecture of this memory access contention, we modified ISPAH so that an entire level k $HiTi$ graph $H^k(P^k, A^k)$ is replicated on the local memory of each processor. This revised $ISPAH$ is named a Modified ISPAH (MISPAH). We empirically analyzed $MISPAH$ the same way as we did for $ISPAH$. Figure 3.9 shows the *speedup* of $MISPAH$ up to 43 processors showing the advantage of replicating a level k $HiTi$ graph on each processor. Although $MISPAH$ performs better than $ISPAH$, it is scalable up to 41 processors. From this analysis, we conjecture that the parallel processing for *inter* query SPSP problems are much more promising than *intra* query SPSP problems.

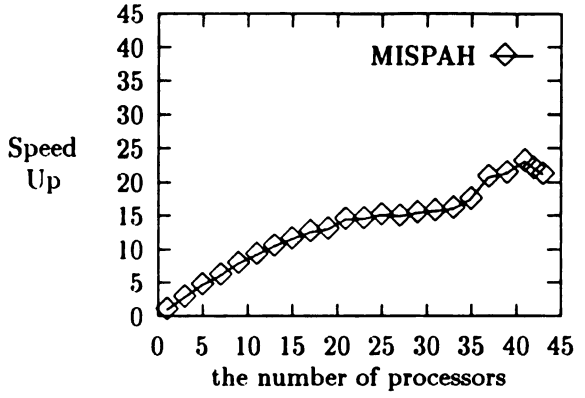


Figure 3.9: Performance of MISPAH

3.4 Conclusion

In this chapter, we studied the parallel processing for *intra* as well as *inter* query shortest path problems on topographical road maps. Based on *HiTi* graph structure, we first developed a parallel shortest path algorithm named *PASPAH* for *intra* query shortest path problem. *HiTi* graph structure provides an opportunity for developing more scalable parallel shortest path algorithms than two tree building approach used in *MOTOpar* algorithm. In *MOTOpar* algorithm, only two processors are used for building two trees, one from source and the other from destination. As a result, *MOTOpar* algorithm is not scalable at all. We empirically analyzed the performance of *PASPAH* by comparing its execution time on grid graphs with those of *MOTOpar* and *SPAH*. We used a BBN GP1000 shared memory multiprocessor system to implement *PASPAH*, *MOTOpar*, and *SPAH*. The BBN GP1000 has a nonuniform memory architecture. Note that *SPAH* presented in chapter 2 is the *HiTi* graph-based sequential shortest path algorithm. Our empirical analysis shows that, although

PASPAH is more scalable than *MOTOpar*, the average execution time of *PASPAH* is not significantly better than those of *MOTOpar* and *SPAH*. From this analysis, we conjecture that the parallel processing for *intra* query shortest path problem is not very promising.

For *inter* query shortest path problem, we proposed a *HiTi* graph-based parallel algorithm named *ISPAH*. We empirically analyzed the performance of *ISPAH* on the BBN GP1000 shared memory multiprocessor system by measuring its speedup on grid graphs as processors are increased. Our analysis shows that the speedup of *ISPAH* increases almost linearly for up to 10 processors. However, *ISPAH* is scalable up to 25 processors. This performance degradation occurs due to the severe memory access conflicts of processors. We then presented an improved version of *ISPAH* named *MISPAH* which reduces the memory access conflicts through partial data replication. The peformance analysis of *MISPAH* shows that it is scalable up to 41 processors. From this empirical analysis, we conjecture that *inter* query shortest path problem has more potential for the parallel processing than *intra* query shortest path problem.

Chapter 4

Description and Location of Distributed Fragments of Large Recursive Relations

In a distributed environment, it is advantageous to fragment a relation and store the fragments at various sites. In this chapter, based on the concept of lattice structures, we develop a framework to study the fragmentation problems of distributed recursive relations. Two of the fragmentation problems are how to describe and locate fragments. Description and location methods previously suggested are more suitable in parallel environments than in distributed databases. In this chapter, we propose a method to describe and locate fragments based on lattice structures. Finding lattice descriptions of fragments is shown to be an NP-complete problem. We analyze the performance of lattice approach both theoretically and experimentally. This is done

by creating a database of recursive relations. The empirical analysis shows that our proposed algorithms give near-optimal solutions.

4.1 Introduction

In distributed database systems, it is often convenient and beneficial to fragment and distribute data according to referencing frequency or locality. By having only frequently accessed data stored in each local site, we can reduce the data maintenance cost. For example, in a distributed database for designing automobiles or aircraft, it is desirable to store the relevant and frequently-accessed parts together. This data can be maintained in a single recursive relation such as a *parts/subparts relation.* There are three major problems in fragmenting recursive relations in distributed environment. The first is how to fragment a recursive relation. The second and third is how to describe and locate the fragments of the recursive relation. The focus of our research is to investigate the second and third problems.

Fragmentation in traditional (non recursive) databases is accomplished through the use of logical predicates, which the tuples of the fragments must satisfy. These logical predicates can easily capture the referencing locality of non-recursive data. They are formed by conjunction, disjunction, and negation statements defined on the attribute values of the tuples. The *fragmentation criterion*, or simply *criterion*, for a fragment F is a predicate which, when applied to a relation R, will determine which tuples of R belong to F. This fragmentation of non-recursive relations is investigated in [16].

Fragmenting recursive relations in distributed databases has not been extensively studied. Most papers on parallel and distributed computation of transitive closures

have applied the traditional fragmentation criteria to recursive relations [32, 45, 47, 74, 84, 102]. However, this criteria, defined by a predicate or a hash function, does not properly capture the referencing locality of data in recursive relations. Locality often comes from the transitive relationship between data. Stated another way, relevant data, frequently accessed by recursive queries, are likely to be related in transitive closures. Although the fragmentation based on the transitive relationship provides less efficient non-recursive query processing strategies for recursive relations than the traditional predicate-based fragmentation, it gives a very powerful basis for an efficient fragment description and location technique suitable for recursive query processing. An efficient processing of recursive queries is important for large recursive relations in distributed databases.

Based on the criteria defined by transitive relationships, Houtsma, *et al.*, studied the parallel computation of transitive closures in [42, 43] and designed strategies for fragmentation in [44]. However, Houtsma's paper [44] did not discuss how to describe and locate fragments in a distributed environment. In a distributed database, we need an efficient fragment description and location method because we can avoid a significant amount of communication cost for query processing by keeping remote fragment descriptions in local sites. In this paper, we focus only on acyclic recursive relations. Unless otherwise stated, *recursive relations* and *acyclic recursive relations* will be used interchangeably.

The rest of the chapter is organized as follows. Section 4.2 describes a fragment description method based on lattice structures. In Section 4.3, a fragment location

method is described. In section 4.4, we suggest methods to incrementally update description of fragments when a recursive relation is modified. The performance analysis of our update algorithm is also discussed in this section. Finally concluding remarks are given in Section 4.5.

4.2 Fragment Description by Lattice Structures

In this section we use the properties of partial order sets (posets) to define and locate fragments of acyclic recursive relations. The acyclic recursive relations, considered in this paper, have the generic form R ($attribute_1$, $attribute_2$, $attribute_3$, ..., $attribute_m$) where the attributes of R satisfy the following:

1. $attribute_1$ and $attribute_2$ are the key attributes of R and share the same domain

2. $attribute_1$ and $attribute_2$ are the recursive join attributes and their corresponding values are related by some transitive relationship

3. $attribute_3$ through $attribute_m$ describe the relationship between $attribute_1$ and $attribute_2$

The above relation R is associated with relation R' which maintains detail descriptions of $attribute_1$ of R. Thus, relation R' has the generic form of R' ($attribute_1'$, $attribute_2'$, $attribute_3'$, ...,$attribute_n'$) where the attributes of R' satisfy the following:

1. $attribute_1'$ of R' is the same as either $attribute_1$ or $attribute_2$ of R

2. $attribute_1'$ is the key attribute of R'.

3. $attribute_2'$ through $attribute_n'$ describe $attribute_1'$

Examples of relation R and R' are shown in Table 4.1 and Table 4.2 where R and R' correspond to *part/subpart* and *part* relations respectively.

part	subpart	quantity
engine	cylinder head	1
engine	fan belt	1
cylinder head	piston	4

Table 4.1: An example of relation R

part	color	weight	dimension	cost
engine	black	800	3 by 2 by 2	1000
cylinder head	gray	100	1 by 2 by 2	250
fan belt	black	30	1 by 5 by 1	250
piston	silver	70	1 by 1 by 1	100

Table 4.2: An example of relation R'

The above relations *part/subpart* and *part* together can be viewed as a directed acyclic graph (DAG) $G_R(V_R, E_R)$. Each node in V_R represent a tuple of the *part* relation and symbolized by the key attribute of the *part* relation. Each edge in E_R represent a tuple of the *part/subpart* relation and symbolized by the key attributes of the *part/subpart* relation. The values of attributes *part* and *subpart* are related by an *inclusion* (i.e. transitive) relationship. For example, *engine* directly *includes* *cylinder head* and *fan belt* while *piston* is indirectly *included* in *engine*. If we consider this *inclusion* relationship as a partial order, then DAG $G_R(V_R, E_R)$ is a poset. A

partial order, $x \preceq y$, is represented as a path in this DAG if there is a path of length 0 or more from node x to y.

In our fragmentation method, each site keeps E_R while V_R is fragmented exclusively and distributed over various sites. Thus, a fragment is a subset of nodes in V_R. Since each site stores E_R, our fragmentation technique is more effective when the ratio of $|E_R|$ to $|V_R|$ is small. When $|E_R|$ is much larger than $|V_R|$, the benefits of fragmenting E_R are likely to exceed the benefits of just fragmenting V_R. However, fragmenting E_R causes one problem in our fragment description and location method presented in the following sections. The problem is, if E_R is fragmented and distributed, then we cannot locally determine the location of remote fragments necessary for recursive query processing.

Definition 4.2.1 *Let $G_R(V_R, E_R)$ represent an acyclic recursive relation. Then, an* **F-subgraph** *for a fragment F is a subgraph $G_F(V_F, E_F)$ of the graph G_R induced by the nodes of fragment F.*

For example, Figure 4.1 shows a digraph $G_R(V_R, E_R)$ for a relation R. Suppose G_R has three fragments named F_1 = { 1, 2, 5, 8, 9 }, F_2 = { 3 ,6, 7, 10, 11, 12}, and F_3 = { 4, 13 }.

Figure 4.2 shows the three F-subgraphs G_1, G_2, and G_3 induced by the fragments F_1, F_2 and F_3, respectively. The F-subgraph is not likely to be a set of randomly unrelated nodes. More likely, it will be a connected component or a collection of several connected components. This is because, as mentioned before, the referencing locality of nodes in V_R often comes from the transitive relationships among the nodes.

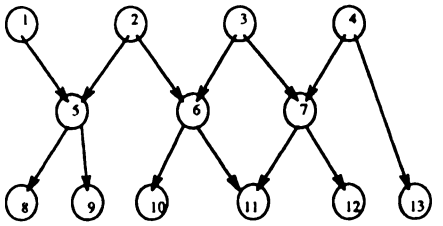


Figure 4.1: A digraph G for an acyclic relation

An F-subgraph is also a poset, since each subset of a poset is itself a poset. Therefore, the properties of posets can be used to describe and locate fragments.

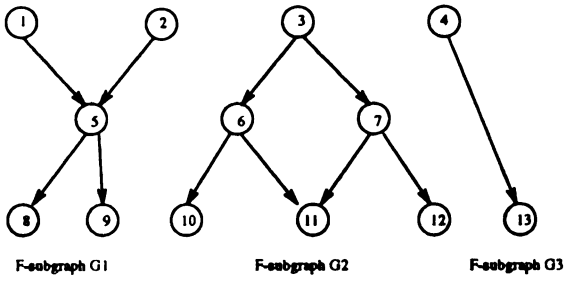


Figure 4.2: Three F-subgraphs

The descriptions of fragments are maintained in a fragment table at each site. A fragment table is a binary relation of the form (f, d) where f is a fragment identifier and d is the corresponding description. A naive description d of a fragment is defined as the set of key attribute values for nodes contained in the fragment. This naive description is quite flexible in the sense that any fragment can be described by grouping the key attribute values of its elements. It is also easy to maintain when updating fragments. However, this naive description suffers from the following serious drawbacks:

1. Fragment description size is large

2. Fragment description size grows linearly with respect to fragment size

3. Communication of fragments' descriptions with other sites is costly due to their large size

4. A large search space is needed to find the fragments containing certain nodes.

The above drawbacks can be very costly, especially when the original relation is very large. A more efficient approach is required for practical purposes. By taking advantage of E_R at each site, we can construct a concise but flexible fragment description method. This method is illustrated in the following subsections.

4.2.1 Maximal and Minimal Nodes Approach

We can represent a fragment by a set U of maximal nodes and a set of V minimal nodes of the F-subgraph $G_F(V_F, E_F)$. Let F be a fragment. Then

$$F = \{ z \mid z \in V_R \; where \; a \preceq z \preceq b \wedge a \in U \wedge b \in V \}$$

Thus, a node is in the fragment if it has an ancestor node in U and a descendant node in V. The fragment F is then represented by the notation $< U, V >$. This approach, however, is not *complete* because it cannot describe certain types of fragments. For example, consider the DAG $G_R(V_R, E_R)$ in Figure 4.3.

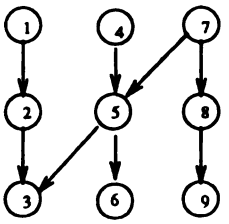


Figure 4.3: An Example of DAG $G_R(V_R, E_R)$

The fragment F = { 1, 2, 3, 7, 8, 9 } cannot be represented by this approach. This is because the fragment represented by $< \{1,7\},\{3,9\} >$ contains the additional node 5. In order for this maximal and minimal nodes approach to be applicable, we need the restriction which should be imposed on fragments. That is, for each fragment F represented by $< U, V >$, the intersection between $U \cup$ { all descendants of nodes in U } and $V \cup$ { all ancestors of nodes in V } should be equal to F. In the next subsection, we give a representation method based on the lattice approach which does not require the above restriction.

4.2.2 Lattice Approach

In this approach, a fragment is described by a set of lattices defined on the corresponding F-subgraph $G_F(V_F, E_F)$. A lattice is represented by L = $< x_L, y_L >$ where x_L is the least upper bound (LUB) and y_L is the greatest lower bound (GLB). Next, we present a list of definitions which will be used throughout the rest of the paper.

Definition 4.2.2 *Let $G_F(V_F, E_F)$ represent an F-subgraph. Then, any pair of nodes (x, y) in V_F is a* candidate lattice *$< x, y >$ for G_F if $x \preceq y$.*

Definition 4.2.3 *A candidate lattice* $< x, y >$ *is called a* **simple lattice** *if* $x \neq y$ *and there is no* z *such that* $x \preceq z \preceq y$, $z \neq x$ *and* $z \neq y$. *A candidate lattice of the form* $< x, x >$ *is called a* **trivial lattice**.

Definition 4.2.4 *Let* $G_R(V_R, E_R)$ *represent an acyclic recursive relation and* $G_F(V_F, E_F)$ *be an F-subgraph of* G_R. *A candidate lattice* $< x, y >$ *for* G_F *is* **suitable** *if the intersection between* { x } $\cup$ { *all descendants of* x } *and* { y } $\cup$ { *all ancestors of* y } *forms a nonempty subset of* V_F.

Definition 4.2.5 *Let* $L = < x, y >$ *be a suitable candidate lattice for* $G_F(V_F, E_F)$. *Then, lattice* L **covers** *node* $z \in V_F$ *if* $x \preceq z \preceq y$.

Definition 4.2.6 *Let* CL *be a set of suitable candidate lattices defined on the F-subgraph* $G_F(V_F, E_F)$. *Let* $LC \subseteq CL$. *Then, we say* LC *is a* **lattice cover** *for* G_F *if every node of* V_F *is covered by a member of* LC.

We will represent a fragment by a lattice cover. Unlike the maximal and minimal nodes approach, the lattice approach is *complete* in that it can describe any fragment by a set of *suitable candidate* lattices defined on the corresponding F-subgraph. For example, { $< 1, 3 >$, $< 7, 9 >$ } is a lattice cover precisely describing the fragment { **1**, 2, 3, 7, 8, 9 } shown in Figure 4.3. Thus we can define a fragment F by a lattice cover as follows:

$$F = \{ z \mid x \preceq z \preceq y \ \textit{for at least one} \ < x, y > \in LC \}$$

More than one lattice cover exists for a given F-subgraph. Figure 4.4a and 4.4b show two different lattice covers for the same F-subgraph G2 of Figure 4.2.

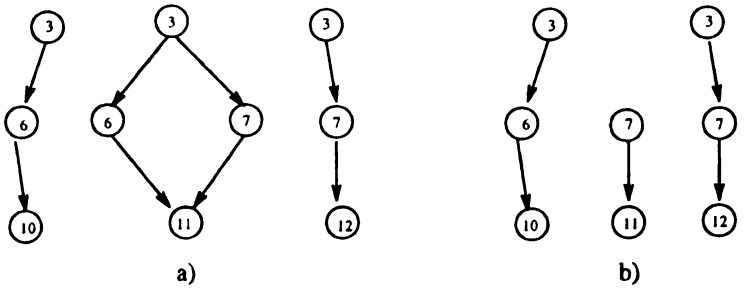


Figure 4.4: Two lattice covers for the F-subgraph G2

The above Figures 4.4a and 4.4b correspond to the lattice covers $LC_1 = \{ < 3,10 >, < 3,11 >, < 3,12 > \}$ $LC_2 = \{ < 3,10 >, < 7,11 >, < 3,12 > \}$ respectively.

So far, we have presented the basic idea for describing the fragment by using a set of suitable candidate lattices called a lattice cover. One important thing to note from this lattice approach is that our lattice representation scheme (i.e. $L = < x_L, y_L >$) has one representational ambiguity. That is, we cannot always uniquely reconstruct any lattice L by using its LUB x_L and GLB y_L alone. This problem will be shown from the lattice $L_1 = < 3,11 > = \{ 3 \preceq 6, 6 \preceq 11, 3 \preceq 7, 7 \preceq 11 \}$ in Figure 4.4a. From L_1, consider two lattices $L_2 = \{ 3 \preceq 6, 6 \preceq 11 \}$ and $L_3 = \{ 3 \preceq 7, 7 \preceq 11 \}$. Even though all three lattices L_1, L_2, and L_3 are represented by the same LUB and GLB (i.e. $< 3,11 >$), they are different lattices. However, this representational ambiguity is not a problem with our lattice approach. This is because in our lattice approach, lattice $L = < x_L, y_L >$ always refers to the one which maximally *covers*

nodes in $G_R(V_R, E_R)$. Thus, in the above example, lattice $< 3, 11 >$ refers to the lattice L_1.

The question that remains is what criteria will determine a good lattice cover for an F-subgraph. We define a good lattice cover as one that minimizes the search space for finding a lattice incorporating a given node. The search time will depend on the number of lattices in a lattice cover and not on the size of each lattice. In other words, the number of lattices in a good lattice cover should be as small as possible while the amount of overlap among the lattices is not important. An index called *lattice cover size* will be used to represent the number of lattices in a lattice cover.

In the next subsection, we show that finding a lattice cover with a minimal index is $NP - complete$. We then present a heuristics for finding a good lattice cover.

4.2.3 Lattice Cover Problem is NP – complete

The lattice cover problem, LCP, is defined as follows. Let P be a set of nodes where the partial order $\preceq$ is defined. For a given positive integer K, the problem is to determine the existence of a lattice cover LC whose cover size $\leq K$ such that every node of P belongs to at least one of the lattices in LC. To prove it is $NP - complete$, we will map the minimum cover problem [33] to LCP. The minimum cover problem, MCP, is stated as follows: Given a collection C of subsets of a finite set S with a positive integer $K \leq |C|$, is there a subset $C' \subseteq C$ with $|C'| \leq K$ such that every element of S belongs to at least one member of C'?

Before presenting a detailed proof for NP − completeness of LCP, we first explains the basic intuition on how to check if a given set of lattices is a lattice cover for the F-subgraph. Given a lattice in a lattice cover, finding a set of nodes covered by the lattice requires performing two transitive operations, one from the LUB (on the original edge graph) and the other from the GLB (on the inverted edge graph) of the lattice, and finding the intersection of the two node sets. Checking that a set of lattices is a lattice cover involves taking the union of the sets of nodes covered by the lattices and checking if this is identical to the sets of all nodes in a fragment. Next, we prove that LCP is NP − complete.

Lemma 4.2.1 *LCP is NP − complete*

Proof: *LCP* is in NP, since a nondeterministic algorithm may guess a set of suitable candidate lattices LC for a partial order set $(P, \preceq)$ such that $|LC| \leq K$, and then check in polynomial time whether LC is a lattice cover.

We transform MCP to LCP. Let C = { S_1, S_2,..., S_n } be a collection of subsets of a finite set S. The poset $(P, \preceq)$ is constructed as follows. For each subset $S_i \in C$, we create four nodes, M_{i_1}, M_{i_2}, N_{i_1}, and N_{i_2}. Next, we create direct partial orders for each i such that $M_{i_2} \preceq N_{i_1}$ and $M_{i_1} \preceq N_{i_2}$ where i = 1, 2, ... ,n. For every x $\in S_i$ where i = 1, 2, ... ,n, we create a direct partial order such that $M_{i_1} \preceq x \preceq N_{i_1}$. This newly constructed poset $(P, \preceq)$ has $2n$ maximal nodes (i.e. M_{1_1}, M_{1_2} M_{2_1}, M_{2_2}, ... ,M_{n-1_1}, M_{n-1_2}, M_{n_1}, M_{n_2}) and $2n$ minimal nodes (i.e. N_{1_1}, N_{1_2} N_{2_1}, N_{2_2}, ... ,N_{n-1_1}, N_{n-1_2}, N_{n_1}, N_{n_2}). From this poset $(P, \preceq)$, let set M and N consist of all maximal and minimal nodes of P. Then, we create CL as such CL = { $< x, y >$: $x \in M \wedge$

$y \in N \wedge x \preceq y$ }.

Then, in any lattice cover $LC \subseteq CL$, we must have at least $2n$ suitable candidate lattices to cover both the n maximal nodes (i.e. M_{1_2}, M_{2_2}, ... ,M_{n_2}) and the n minimal nodes (i.e. N_{1_2}, N_{2_2}, ... ,N_{n_2}). Furthermore, it is always possible to derive the above $2n$ suitable candidate lattices as $2n$ simple lattices. This is because we have $M_{i_2} \preceq N_{i_1}$ and $M_{i_1} \preceq N_{i_2}$ where $i = 1, 2, \ldots, n$. Let a set X denote the above $2n$ simple lattices.

It is obvious that the lattices in X also cover the remaining n maximal nodes (i.e. M_{1_1}, M_{2_1}, ... ,M_{n_1}) and n minimal nodes (i.e. N_{1_1}, N_{2_1}, ... ,N_{n_1}). Note that no lattice in X covers any node in S because the lattices in X are simple lattices. Therefore, if we can derive a lattice cover LC, where $|LC| \leq (K + 2n)$, from this poset $(P, \preceq)$, then we have a minimum cover $C' \subseteq C$ with $|C'| \leq K$.

To see that this transformation can be performed in polynomial time, it suffices to observe that the total number of partial orders in $(P, \preceq)$ are bounded by a polynomial of $O(|C| \cdot \max\{|S_i| : S_i \in C\})$. □

In order to demonstrate how the above transformation works, we will illustrate an example as follows. Let MCP have the instances such as S = { 1, 2, 3, 4, 5 } and C = { S_1, S_2, S_3, S_4, S_5 } where S_1 = { 1, 2, 3 }, S_2 = { 4, 5 },S_3 = { 1, 2 }, S_4 = { 3, 4 }, and S_5 = { 5 }, From this MCP, we need to construct a poset $(P, \preceq)$. For each subset $S_i \in C$, we first create four nodes M_{i_1}, M_{i_2}, N_{i_1}, and N_{i_2}, which are shown in Table 4.3.

i	S_i	M_{i_1}	M_{i_2}	N_{i_1}	N_{i_2}
1	$S_1 = \{ 1, 2, 3 \}$	A	B	K	L
2	$S_2 = \{ 4, 5 \}$	I	J	T	U
3	$S_3 = \{ 1, 2 \}$	C	D	N	O
4	$S_4 = \{ 3, 4 \}$	E	F	P	Q
5	$S_5 = \{ 5 \}$	G	H	R	S

Table 4.3: Newly created nodes for each set S_i for transforming from MCP to LCP

Next, we create direct partial orders for each set S_i where i = 1,2, ..., 5. For S_1, the direct partial orders created are A $\preceq$ L, B $\preceq$ K, A $\preceq$ 1 $\preceq$ K, A $\preceq$ 2 $\preceq$ K, and A $\preceq$ 3 $\preceq$ K. The rest of direct partial orders created for other sets in S are shown in Figure 4.5.

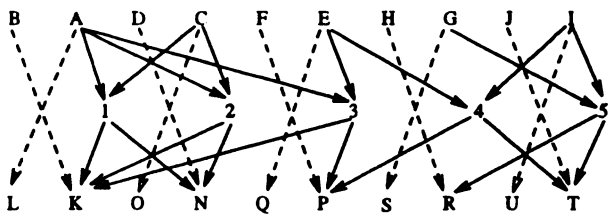


Figure 4.5: Partial order sets created from MCP to LCP

Note that in Figure 4.5, dashed arrows (undashed arrows) represent direct partial orders constructed for newly created nodes (resp. nodes in S_i). Then, set CL consists of suitable candidate lattices in $\{ < B,K >, < A,L >, < A,K >, < A,N >, < A,P >, < D,N >, < C,O >, < C,N >, < C,K >, < F,P >, < E,Q >, < E,K >, < E,P >, < H,R >, < G,S >, < G,R >, < G,T >, < J,T >, < I,U >, < I,R >, < I,T > \}$. It is easy to see from Figure 4.5 and CL that if we can derive a lattice cover $LC \subseteq CL$, where $|LC| \leq K + 2 \cdot 5$, then we have a minimum cover

$C' \subseteq C$ with $|C'| \leq K$.

4.2.4 A General Heuristic for Finding a Good Lattice Cover

In this subsection, we present a heuristics for finding a good lattice cover. The following proposition will establish the relationship between the lattice cover size and the number of maximal and minimal nodes in the F-subgraph $G_F(V_F, E_F)$.

Proposition 4.2.1 *Given an F-subgraph G_F with M maximal nodes and N minimal nodes, the index for any lattice cover of G_F must be at least* $\max(M, N)$.

Proof: Assume that $M \geq N$. Since two maximal nodes can not belong to the same lattice, we need at least M lattices to cover all those maximal nodes. Similarly, if $N \geq M$, we need at least N lattices to cover those minimal nodes. □

From the proof of proposition 4.2.1, it is obvious that all maximal (minimal) nodes in G_F must be used at least once as the LUB (resp. GLB) node of some lattice in any lattice cover of G_F. This is because a maximal (minimal) node cannot be covered by any lattice whose LUB (resp. GLB) node is not a maximal (resp. minimal) node. This property provides a good basis for choosing a set of suitable candidate lattices, i.e. CL, from all possible suitable candidate lattices.

We restrict all suitable candidate lattices to consist of those whose LUB and GLB nodes are chosen from maximal and minimal nodes (respectively) of F-subgraph. For example, consider the F-subgraph G2 in Figure 4.2. The suitable candidate lattice

$< 7, 11 >$ defined on G2 is not chosen as a member of set CL. This is because LUB node 7 is not a maximal node of the F-subgraph G2. The set CL consists of the lattices $< 3, 10 >$, $< 3, 11 >$ and $< 3, 12 >$. However, it is not always possible to derive a lattice cover for a given G_F from the subset of suitable candidate lattices we just described. We take the DAG G_R in Figure 4.6a as an example to illustrate this problem.

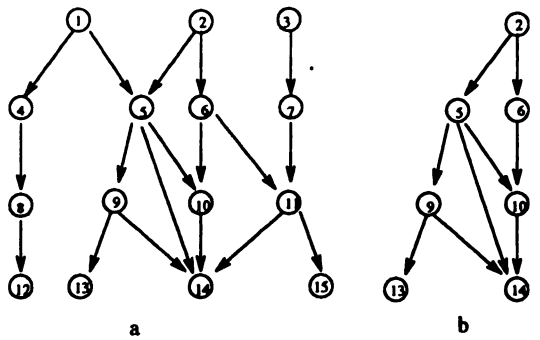


Figure 4.6: A digraph G and F-subgraph of G

From the graph G_R in Figure 4.6a and the fragment $F = \{ 2, 5, 6, 9, 10, 13, 14 \}$, we get the corresponding G_F shown in Figure 4.6b. It follows that $CL = \{ < 2, 13 > \}$ from which we cannot derive a lattice cover for G_F. This is because set CL cannot contain the unsuitable candidate lattice $< 2, 14 >$ which also includes additional node $11 \notin F$. Thus, we need to enumerate additional suitable candidate lattices to derive a lattice cover. For this purpose, we present Algorithm 1 in Figure 4.7. It obtains additional suitable lattices by partitioning an unsuitable lattice $< x, y >$ into a set of suitable lattices. The union of these suitable lattices then covers all nodes in $\{z \mid x \preceq z \preceq y \ \wedge \ z \in V_F\}$.

The input to the algorithm is V_F, unsuitable lattice $L =< x, y >$ and a subgraph $SG_I(V_I, E_I)$ induced by the elements of $L =< x, y >$. The elements of $L =< x, y >$ was obtained from the intersection between ({x} ∪ {all descendants of x}) and ({y} ∪ {all ancestors of y}) by using E_R. Note that the we represent E_I and E_R as adjacency lists.

begin

1. Let a subgraph $SG_I(V_I, E_I)$ corresponds to $L = < x, y >$;
2. Let $\Psi = \emptyset$;
3. Obtain $IV_I = V_I - V_F$ and $V_L = V_I - IV_I$;
4. Let $SG_L(V_L, E_L)$ be a subgraph induced by nodes in V_L;
5. Let $P_x = 0$, $S_1 = \{x\}$, and $i = 2$;
6. Let $S_2 = \{ v \mid (u, v) \in E_L$ for all $u \in S_1 \}$;
7. **If** $S_2 = \emptyset$ **then** goto **step** 10;
8. For each v in S_2, let $P_v = i$;
9. $i = i + 1$, $S_1 = S_2$, and goto **step** 6;
10. Find $\delta = \{ (u, v) \mid (u, v) \in E_L \ \wedge \ (u, w) \in E_I$ *where* $w \in IV_I\}$;
11. $E_L = E_L - \delta$ and $N = \{ u \mid (u, v) \in \delta \}$;
12. Find $\mathbf{w} \in N$ such that $P_{\mathbf{w}} = min\{ P_v \mid v \in N \}$;
13. Let M consist of all maximal nodes in E_L;
14. Let $\Psi = \Psi \cup \{< u, \mathbf{w} > | u \in M \wedge (u \preceq \mathbf{w}$ *is true in* $E_L)\}$;
15. From E_L, obtain $S_{<m,n>} = \{c | m \preceq c \preceq n\}$ for each $< m, n > \in \Psi$;
16. Obtain $V_L = V_L - \{ c \mid c \in S_{<m,n>}$ *where* $< m, n > \in \Psi \}$;
17. **If** $V_L = \emptyset$ **then** goto **step** 20;
18. Let E_L consist of those edges whose end nodes are in V_L;
19. Let N consists of all minimal nodes in E_L and goto **step** 12;
20. **return** (Ψ, $\{S_L | L \in \Psi\}$);

end

Figure 4.7: Algorithm 1: Partition an Unsuitable Candidate lattice $L =< x, y >$

We re-take the DAG G_R and G_F in Figure 4.6 as an example to demonstrate Algorithm 1. The input to this algorithm is a V_F = { 2, 5, 6, 9, 10, 13, 14 }, the unsuitable candidate lattice $< 2, 14 >$ and the subgraph $SG_I(V_I, E_I)$ corresponding to $< 2, 14 >$. Figure 4.8a shows $SG_L(V_L, E_L)$ obtained at step 4. The value P_v of

each node v in V_L is also shown beside node v in Figure 4.8a. Note that due to edge $(5, 14)$, P_{14} is initially set to 3 and then reset to 4.

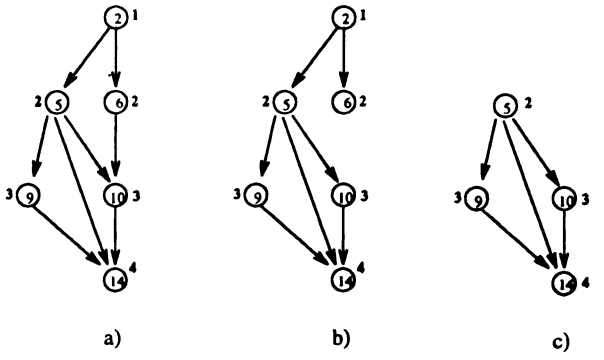


Figure 4.8: Changes of $SG_L(V_L, E_L)$ in Algorithm 1

At step 10, since node 6 connected to node 11 in IV_I, δ contains edge $(6, 10)$. We delete edge(s) in δ from E_L at next step. Thus, the structure of SG_L is changed, which is shown in Figure 4.8b. After the executions of step 12 to 15, we obtain $\Psi = \{ < 2, 6 > \}$ and change SG_L. This changed SG_L is shown in Figure 4.8c. Since V_L is still non-empty, steps 12 to 16 are repeated. During this repetition, set Ψ is changed to $\Psi = \{ < 2, 6 >, < 5, 14 > \}$. This set Ψ contains suitable candidate lattices partitioned from the unsuitable candidate lattice $< 2, 14 >$. Next, we discuss the time complexity of Algorithm 1.

The primary complexity of Algorithm 1 comes from traversing edges in E_L. Then, it is easy to see that Algorithm 1 will not traverse more than $|\Psi| \cdot |E_L|$ edges. Thus, the time complexity is $O(|\Psi| \cdot |E_L|)$.

We now present Algorithm 2 which generates a good lattice cover for an F-subgraph. It is given in Figure 4.9. The input to Algorithm 2 is the *F-subgraph* $G_F(V_F, E_F)$, edge set E_R, and set M and N. Set M and N denote the sets of maximal and minimal nodes of the *F-subgraph* respectively. Note that edge set E_R and E_F are maintained as adjacency lists.

Algorithm 2 first creates a set of suitable candidate lattices CL by using Algorithm 1 and the method we described in the early part of this subsection. Next, Algorithm 2 tries to identify *essential lattices* among the lattices in CL. The formal definition of essential lattices is given in the following definition 4.2.7.

Definition 4.2.7 *Let CL be a set of suitable candidate lattices for an F-subgraph $G_F(V_F, E_F)$. Suppose that $CL' = \{ LC_1, LC_2, \ldots, LC_k \}$ consists of all possible lattice covers for G_F where $LC_i \subseteq CL$ for $i = 1, 2, \ldots, k$. Then, a lattice in CL is* **essential** *if it belongs to any lattice cover LC_i for $i = 1, 2, \ldots, k$.*

Identifying all the essential lattices in CL was shown to be NP-Hard [73]. However, some of the essential lattices can be identified by checking each node in the F-subgraph. That is, if a node is covered by a single lattice in CL, then the lattice is *essential.* After finding all identifiable essential lattices in CL, Algorithm 2 compute a set of nodes, V, not covered by identified essential lattices in CL. It then repeatedly apply a greedy heuristics to the nodes in V to select a lattice from CL that covers the most remaining uncovered nodes in V (with ties broken arbitrary).

Figure 4.10 shows a digraph $G_R(V_R, E_R)$ consisting of two F-subgraphs corresponding to the fragments { 1, 2, ... ,14, 15 } and { 16, 17, ... , 21, 22 }. The

Stage 1

1. Let $LC = \emptyset$. For each node v in V_F, create the corresponding ancestor and the lattice set A_v and LS_v respectively. They are initialized to $\emptyset$. Create two sets ECL (Essential Candidate Lattice) and UCL (Unsuitable Candidate Lattice). Let $ECL = UCL = \emptyset$.

2. For each maximal node m in M, let $A_m = \{ m \}$. For every v which is a descendant of m in E_F, let also $A_v = A_v \cup \{ m \}$. Therefore, given a node v, its ancestor set A_v contains all the maximal nodes which are ancestors of v. For each maximal node m in M, let $A_m = \{ m \}$.

3. For each minimal node n in N, and each $m \in A_n$, the node pair (m, n) suggests a candidate lattice $< m, n >$. There are $\sum_{n \in N} |A_n|$ such pairs. Let CL denote the set of $\sum_{n \in N} |A_n|$ candidate lattices.

4. For each candidate lattice $L =< x, y >$ in CL, create the corresponding set S_L by using E_R. The elements of S_L are obtained from the intersection between ($\{ x \} \cup \{$ *all descendants of* $x \}$) and ($\{ y \} \cup \{$ *all ancestors of* $y \}$). If L is not suitable, move L from CL to UCL.

5. For each $L \in CL$, $LS_v = LS_v \cup \{ L \}$ if $v \in S_L$. Let $UCL' = \emptyset$. For each node v in V_F such that $LS_v = \emptyset$: $UCL' = UCL' \cup \{ L \mid L \in UCL \wedge v \in S_L \}$. If $UCL' = \emptyset$ goto **step 8**.

6. Apply Algorithm 1 to each unsuitable candidate lattice $L \in UCL'$. Let PCL consists of those suitable candidate lattices returned from Algorithm 1.

7. For each $L \in PCL$, $LS_v = LS_v \cup \{ L \}$ if $v \in S_L$. $CL = CL \cup$ PCL.

8. For all $v \in V_F$, if LS_v contains only one lattice L, then move L from CL to ECL.

9. Let $LC = LC \cup ECL$ and $V_F = V_F - \cup_{L \in ECL} S_L$.

Stage 2

$V = V_F$; $U = CL$;
while ($V \neq \emptyset$) {
    Select an $L \in U$ that maximizes $|S_L \cap V|$;
    $U = U - L$;
    $V = V - S_L$;
    $LC = LC \cup L$; }
LC is the lattice cover for the F-subgraph;

Figure 4.9: Algorithm 2: Find a Lattice Cover for an F-subgraph

F-subgraph $G_F(V_F, E_F)$ corresponding to the fragment { 1, 2, ... ,14, 15 } will be used to demonstrate Algorithm 2 as follows.

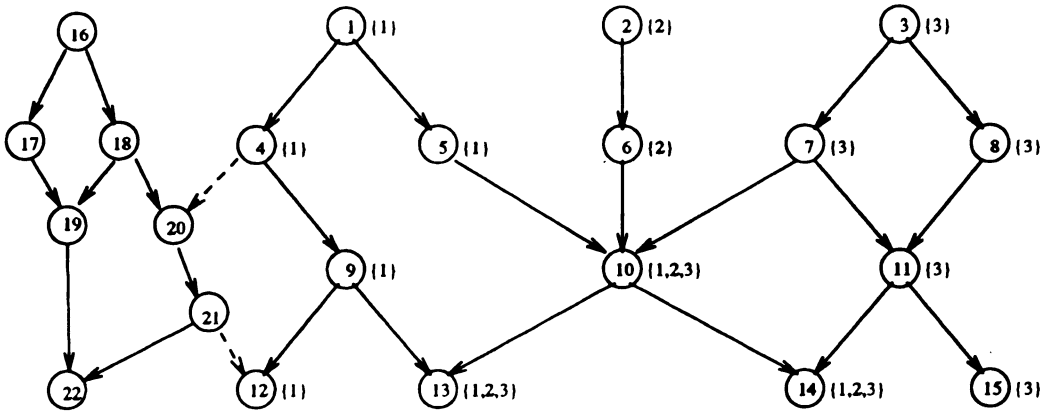


Figure 4.10: A digraph G consisting of two F-subgraphs

Stage 1

2. The ancestor set A_v of each node v in V_F is shown beside node v in Figure 4.10.

3. The set CL of candidate lattices is obtained: $CL = \{ < 1,12 >, < 1,13 >, < 1,14 >, < 2,13 >, < 2,14 >, < 3,13 >, < 3,14 >, < 3,15 > \}$.

4. Table 4.4 shows the nodes of each candidate lattice L in CL. The nodes of each lattice are computed by using the edge set E_R. Among the candidate lattices in CL, lattice $< 1,12 >$ is not suitable, since it also covers nodes 20 and 21. Thus, $CL = CL - \{< 1,12 >\}$ and $UCL = \{ < 1,12 > \}$.

5. In this step, we got $LS_1 = \{ < 1,13 >, < 1,14 > \}$, $LS_2 = \{ < 2,14 >, < 2,15 > \}$, $LS_3 = \{ < 3,13 >, < 3,14 >, < 3,15 > \}$, $LS_4 = \emptyset$, ... ,$LS_{15} = \{ < 3,15 > \}$. Then, by the definition of UCL', $UCL' = \{ < 1,12 > \}$.

L	S_L	L	S_L
$< 1,12 >$	S_L = { 1, 4, 9, 12, 20, 21 }	$< 2,13 >$	S_L = { 2, 6, 10, 13 }
$< 1,13 >$	S_L = { 1, 4, 5, 9, 10, 13 }	$< 3,13 >$	S_L = { 3, 7, 10, 13 }
$< 1,14 >$	S_L = { 1, 5, 10, 14 }	$< 3,14 >$	S_L = { 3, 7, 8, 10, 11, 14 }
$< 2,14 >$	S_L = { 2, 6, 10, 14 }	$< 3,15 >$	S_L = { 3, 7, 8, 11, 15 }

Table 4.4: Lattice L and Set S_L

6. By applying Algorithm 1 to unsuitable candidate lattice $< 1,12 >$, we got PCL $= \{ < 1,4 >, < 9,12 > \}$.

7. In this step, by using E_F, we got $S_{<1,4>} = \{ 1, 4 \}$, $S_{<9,12>} = \{ 9, 12 \}$, LS_1 $= \{ < 1,4 >, < 1,13 >, < 1,14 > \}$, $LS_4 = \{ < 1,4 >, < 1,13 > \}$, $LS_9 = \{$ $< 1,13 >, < 9,12 > \}$, $LS_{12} = \{ < 9,12 > \}$. And $CL = CL \cup \{ < 1,4 >,$ $< 9,12 > \}$.

8. The lattices $< 9,12 >$ and $< 3,15 >$ are essential because $LS_{12} = \{ < 9,12 >$ $\}$, $LS_{15} = \{ < 3,15 > \}$. Hence, $ECL = \{ < 9,12 >, < 3,15 > \}$ and $CL = \{$ $< 1,4 >, < 1,13 >, < 2,13 >, < 2,14 >, < 3,13 >, < 3,14 > \}$.

9. LC and V consist of $\{ < 9,12 >, < 3,15 > \}$ and { 1, 2, 4, 5, 6, 10, 13, 14 } respectively.

Stage 2

A greedy method is performed in *while* loop. At the first execution of while loop, lattice $< 1,13 >$ will be selected from CL. Then lattice $< 2,14 >$ will be selected in the next execution of the *while* loop and *Stage 2* will end. Therefore,

the lattice cover LC consists of $\{ < 9,12 >, < 3,15 >, < 1,13 >, < 2,14 > \}$, which in this example is the minimal lattice cover.

4.2.5 Near Optimal Lattice Cover Size

Cormen *et al.* [20] used the same greedy heuristics as the one in *stage 2* of Algorithm 2. They derived a logarithmic ratio bound for their algorithm. Our greedy approximation algorithm has the same performance ratio bound in the worst case as the one shown in [20]. This is because, in the worst case, our algorithm may not identify any essential lattice. As a result, only the greedy heuristics of *stage 2* is used for deriving a lattice cover.

Let $OPT(I^*)$ be the optimal solution and $LCP(I)$ the heuristic solution for LCP problem. Cormen *et al.* prove in [20] that $LCP(I) < (\ln(\max\{|S_L| : L \in CL\}) + 1) \cdot OPT(I^*)$. However, for our algorithm, this bound represents the worst case situation.

To measure the performance of Algorithm 2, we create random DAGs $G_R(V_R, E_R)$. The generation of DAGs is based on a uniform distribution in the range $[1, |V_R|]$ where $|V_R|$ represents the total number of nodes. Then, we create F-subgraphs by partitioning G_R. Each F-subgraph has a small number of maximal and minimal nodes compared to $|V_F|$ (i.e. about 1 % of $|V_F|$ on the average). Based on the above scheme, we create a set of random F-subgraphs $G_F(V_F, E_F)$ as follows:

- Size of V_F is varied from 1000, 1500, to 2000 nodes.
- For $|V_F| = 1000$, we create G_F by varying $|E_F|$ from 1500 to 10000 with an

interval 500 edges.

- For $|V_F| = 1500$, we create G_F by varying $|E_F|$ from 2500 to 10000 with an interval 500 edges.

- For $|V_F| = 2000$, we create G_F by varying $|E_F|$ from 3000 to 10000 with an interval 500 edges.

- For each fixed size of V_F and E_F, we create 10 G_F's using 10 different seeds.

For each F-subgraph created as above, we compute the lattice cover size $|LC_F|$ and the percentage ratio between the total number of essential lattices (i.e. $|ECL|$) and suitable candidate lattices (i.e. $|CL|$), by using Algorithm 2. Note that CL (ECL) is obtained in step 7 (resp. step 8) of Algorithm 2. These values are then averaged over the 10 F-subgraphs with same values of $|V_F|$ and $|E_F|$. These are shown in Figure 4.11.

Figure 4.11a shows that the lattice cover size $|LC_F|$ converges to the theoretically possible minimal lattice cover size as we increase $|E_F|$. The theoretically possible minimal lattice cover size is given in Proposition 2.1. From Figure 4.11b, we can clearly observe that the value of P approaches close to 100 % as we decrease $|E_F|$. This implies that the total number of essential lattices approaches close to the total number of the suitable candidate lattices, as we decrease $|E_F|$. Since $|ECL| \leq |LC_F| \leq |CL|$, Algorithm 2 gives LC_F close to the minimal lattice cover, as $|E_F|$ is decreased. Therefore, we can conclude that Algorithm 2 gives near optimal lattice covers for both dense and non-dense F-subgraphs. Note that dense F-subgraphs have much larger

number of edges per node than that for non-dense F-subgraphs.

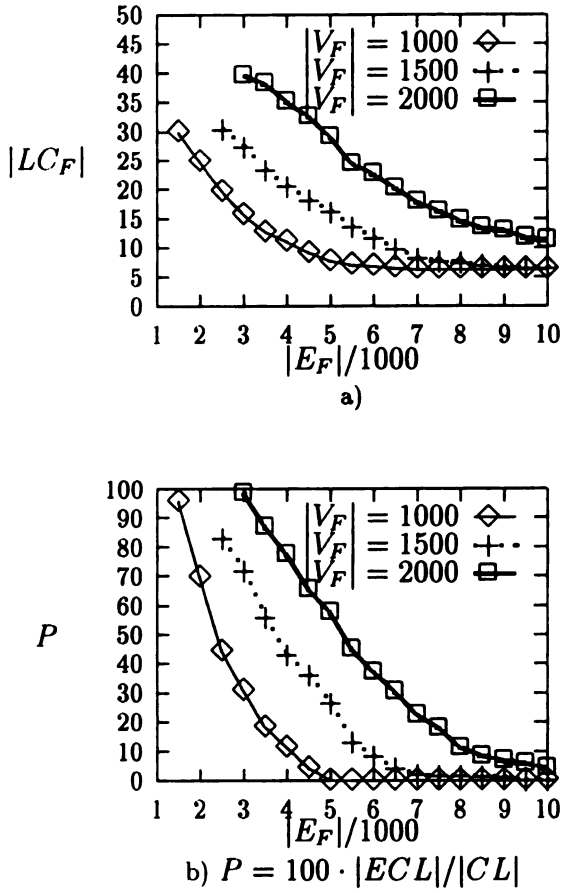


Figure 4.11: Performance analysis for Algorithm 2

Next, we discuss the complexity of Algorithm 2. Algorithm 2 consists of *stage 1* and *2*. The time complexity of *stage 1* is dominated by *step 4*. Let N_d (N_a) be the maximum number descendants (resp. ancestors) of node $x \in M$ (resp. N). Let e_t be the total number of edges to be traversed to find the above ancestor and descendant nodes. Then *step 4* will run in $O(e_t + (N_d + N_a) \cdot \sum_{n \in N} |A_n|)$. For *stage 2*, we can easily see that its time complexity is $O(|CL|^2 \cdot |V_F|)$. Thus, the time complexity of the algorithm is $O(e_t + (N_d + N_a) \cdot \sum_{n \in N} |A_n| + |CL|^2 \cdot |V_F|)$. Note that, as the denseness of F-subgraph is decreased, the time complexity will approach to $O(e_t +$

$(N_d + N_a) \cdot \sum_{n \in N} |A_n|)$. This is explained by the result shown in Figure 4.11b.

4.3 Finding Relevant Fragments

The fragments containing the nodes a query needs to access, are defined as *relevant fragments* of the query. Note these set of nodes may be a super set of the result nodes. In order to find relevant fragments of the query, each site needs to access its local fragment table and local edge set E_R. Before discussing methods of locating the relevant fragments, we first introduce a set of basic notations which will be used in the rest of this paper. They are defined as follows:

FT : the set of all lattices in the fragment table; Q : a set of nodes;

D_Q : $Q \cup \{$ all descendants of nodes in $Q \}$;

A_Q : $Q \cup \{$ all ancestors of nodes in $Q \}$;

$S_{lub} : \{ x \mid < x, y > \in FT \}$; $S_{lub_Q} : \{ x \mid x \in S_{lub} \wedge x \in A_Q \}$;

$S_{glb} : \{ y \mid < x, y > \in FT \}$; $S_{glb_Q} : \{ y \mid y \in S_{glb} \wedge y \in D_Q \}$;

Note that D_Q and A_Q are computed from E_R maintained in adjacency lists. The following proposition provides the basis for locating relevant fragments of the query.

Proposition 4.3.1 *Suppose Q consists of a set of nodes needed by a query. Let Λ be the set of lattices in FT which cover all nodes in Q. Then, the lattices in Λ are only those lattices in $\Upsilon = \{ < x, y > \mid x \in S_{lub_Q} \wedge y \in S_{glb_Q} \wedge < x, y > \in FT \}$.*

Proof: We need to show that $\Lambda \subseteq \Upsilon$ and $\Upsilon \subseteq \Lambda$ to prove $\Lambda \equiv \Upsilon$.

Case 1: To prove $\Lambda \subseteq \Upsilon$, we need to show $\forall < x, y > (< x, y > \in \Lambda \rightarrow < x, y > \in \Upsilon)$. Assume that there is a lattice $< u, v > \in \Lambda$ such that $< u, v > \notin \Upsilon$. Then, $< u, v >$ must cover at least one node $z \in Q$. By the definition of Υ, $u \notin S_{lub_Q}$ or $v \notin S_{glb_Q}$. However, since $u \preceq z \preceq v$ and $z \in Q$, $u \in A_Q$ and $v \in D_Q$. This implies that $u \in S_{lub_Q}$ and $v \in S_{glb_Q}$, which is a contradiction.

Case 2: To prove $\Upsilon \subseteq \Lambda$, we need to show $\forall < x, y > (< x, y > \in \Upsilon \rightarrow < x, y > \in \Lambda)$. Assume that there is a lattice $< u, v > \in \Upsilon$ such that $< u, v > \notin \Lambda$. Then, by the definition of Υ, $< u, v > \in FT$ and it must cover at least one node in Q. Since Λ contains all lattices in FT which cover all nodes in Q, Λ must contain $< u, v >$. This contradicts our assumption. □

Even though proposition 4.3.1 gives a general way of locating relevant fragments, it is an inefficient approach because it requires computing two transitive closures (i.e. finding all ancestors and descendants) of all nodes in Q. Thus, we need an efficient algorithm which minimizes transitive closure computations. Fortunately, for recursive queries, the nodes they need to access are likely to be a connected component or a collection of several connected components. In this case, we can minimize computing two transitive closures of all nodes in Q by taking advantage of the transitive relationships among the query nodes. Furthermore, if a recursive query processing requires a transitive closure computation on E_R, then either $Q \equiv D_Q$ or $Q \equiv A_Q$. Then, in either cases, no more transitive closure computation is necessary.

This will be proved in the following proposition 4.3.2 and 4.3.3.

Proposition 4.3.2 *Let* $D_Q \equiv Q$ *and* $\Pi = \{ < x, y > \mid y \in S_{glb_Q} \wedge < x, y > \in FT \}$. *Then,* $\Pi \equiv \Upsilon$.

Proof: We need to show that $\Pi \subseteq \Upsilon$ and $\Upsilon \subseteq \Pi$ to prove $\Pi \equiv \Upsilon$.

Case 1: To prove $\Upsilon \subseteq \Pi$, we need to show $\forall < x, y > (< x, y > \in \Upsilon \rightarrow < x, y > \in \Pi)$.

For all $< x, y > \in \Upsilon$, by definition of Υ, it is true that $y \in S_{glb_Q}$ and $< x, y > \in FT$.

Thus, all $< x, y > \in \Upsilon$ belongs to Π.

Case 2: To prove $\Pi \subseteq \Upsilon$, we need to show $\forall < x, y > (< x, y > \in \Pi \rightarrow < x, y > \in \Upsilon)$.

Assume that there is a lattice $< u, v > \in \Pi$ such that $< u, v > \notin \Upsilon$. Then, it must **be** true that $v \in S_{glb_Q}$, $< u, v > \in FT$ and $u \notin S_{lub_Q}$. By proposition 4.3.1, $< u, v >$ **cannot** cover any node in Q. However, $< u, v >$ covers the node $v \in Q$, since $v \in S_{glb_Q} \subseteq D_Q \equiv Q$. Thus, $< u, v > \in \Upsilon$, which is a contradiction. □

Proposition 4.3.3 *Let* $A_Q \equiv Q$ *and* $\Omega = \{ < x, y > \mid x \in S_{lub_Q} \wedge < x, y > \in FT \}$. *Then,* $\Omega \equiv \Upsilon$.

Proof: The proof is similar to the one for Proposition 4.3.2. □

Before presenting an efficient algorithm (i.e. Algorithm 3) for finding relevant fragments, we first introduces a function named **Block**(x, t), which is called within Algorithm 3. Note that parameter t represents either symbol "L" or "G". Function **Block**(x,G) (**Block**(x,L)) returns a set of GLB (resp. LUB) nodes from which a given

node x is reachable. This function is described in Figure 4.12.

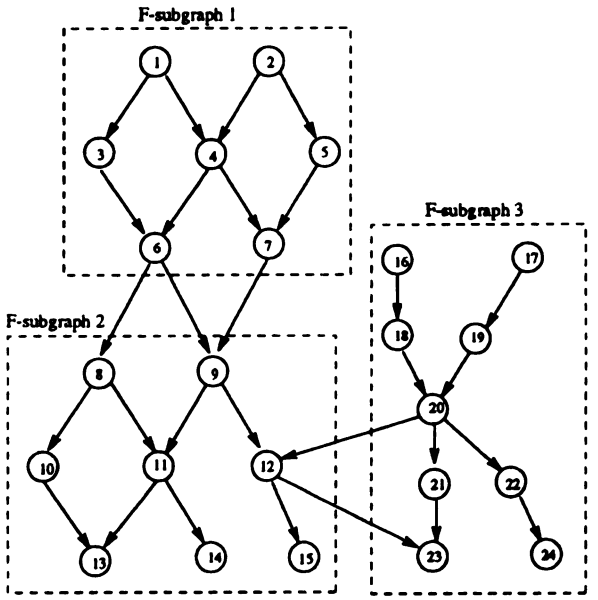
```
Block(x,t) /* x : node; t: either G or L; */
  begin
    current = {x}; Block = ∅;
    If t = G then s = S_glb;
    else s = S_lub;
    while ( current ≠ ∅ ) {
      temp = current ∩ s;
      Block = Block ∪ temp;
      current = current − temp;
      If t = G then
        current = {y|(x,y) ∈ E_R ∧ x ∈ current};
      else /* t = L */
        current = {x|(x,y) ∈ E_R ∧ y ∈ current}; }
    return (Block);
  end
```

Figure 4.12: Procedure Block(x, t)

To show how Block(x, t) works, we take Figure 4.13 as an example.

Figure 4.13a shows three F-subgraphs whose node information is stored at different sites. The fragment table corresponding to the DAG G_R of Figure 4.13a is shown in Figure 4.13b. Note that this fragment table is stored at each site. From Figure 4.13b, we have S_{glb} = { 6, 7, 13, 14, 15, 24, 20, 23 } and S_{lub} = { 1, 2, 8, 9, 16, 17, 21 }. Then, it is easy to see from the procedure that Block(4,G) = {6,7}, Block(6,G) = {6}, Block(23,L) = {9, 16, 17, 21}, and Block(21,L) = {21}.

We now describe Algorithm 3 in Figure 4.14. Let Q consists of a set of nodes a query need to access. Algorithm 3 first checks if either $Q \equiv D_Q$ or $Q \equiv A_Q$. If it is, using proposition 4.3.2 and 4.3.3, Algorithm 3 easily locates relevant fragments. Otherwise, the algorithm tries to find relevant fragments taking advantage of transitive



Fragment #	Lattice
1	<1,6>
1	<2,7>
2	<8,13>
2	<9,14>
2	<9,15>
3	<17,24>
3	<16,20>
3	<21,23>

b)

Figure 4.13: a) DAG for the whole recursive relation b) Fragment table

relationships among the nodes in Q.

We re-take the preceding Figure 4.13 to demonstrate Algorithm 3. Assume that two recursive queries RQ_1 and RQ_2 need to access nodes in Q_1 = { 8, 10, 11, 13, 14 } and Q_2 ={ 12, 18, 19, 20, 21 } respectively. We show how to find relevant fragments of RQ_1 and RQ_2 in the following two cases.

Case RQ_1: We have $L_Q = \{8\}$, $G_Q = \{13, 14\}$, E_Q = { (8,10), (8,11), (10,13), (11,13), (11,14) }, E_L = { (6,8), (9,11) }, and $E_G = \emptyset$. Since $E_G = \emptyset$, we obtain $LT = \{< 8, 13 >, < 9, 14 >\}$ at *step 3*. Thus, fragment 2 is the relevant fragment of the query RQ_1.

begin

1. Compute $L_Q = S_{lub} \cap Q$ and $G_Q = S_{glb} \cap Q$;
2. Compute $E_Q = \{(x,y) | x \in Q \wedge y \in Q \wedge (x,y) \in E_R\}$,
   $E_L = \{(x,y) | x \notin Q \wedge y \in Q \wedge (x,y) \in E_R\}$,
   $E_G = \{(x,y) | x \in Q \wedge y \notin Q \wedge (x,y) \in E_R\}$;
3. **If** $E_G = \emptyset$ **then** /* $Q \equiv D_Q$ */
   { $LT = \{< x,y > | y \in G_Q \wedge < x,y > \in FT\}$; Goto *step* 10; }
   If $E_L = \emptyset$ **then** /* $Q \equiv A_Q$ */
   { $LT = \{< x,y > | x \in L_Q \wedge < x,y > \in FT\}$; Goto *step* 10; }
4. **For** each node $x \in Q$, $L_x = G_x = \emptyset$;
5. $B_{G_Q} = \{x | (x,y) \in E_G\} \cup G_Q$;
   $B_{L_Q} = \{y | (x,y) \in E_L\} \cup L_Q$;
6. **For** each $x \in B_{G_Q}$, $G_x = Block(x, G)$;
   For each $x \in B_{L_Q}$, $L_x = Block(x, L)$;
7. **For** each $x \in B_{G_Q}$
   For every y which is an ancestor of x within E_Q
   $G_y = G_y \cup G_x$;
8. **For** each $x \in B_{L_Q}$ {
   $L' = L_x$; $G' = G_x$;
   $LT_x = \{< a,b > | a \in L_x \wedge b \in G_x \wedge < a,b > \in FT\}$;
   while ($LT_x = \emptyset$) { /* No lattice covering node x is yet found */
   $L' = \{u | y \in L' \wedge (u,y) \in E_R\}$;
   $G' = \{v | x \in G' \wedge (x,v) \in E_R\}$;
   For each $u \in L'$, $L_x = L_x \cup Block(u, L)$;
   For each $v \in G'$, $G_x = G_x \cup Block(v, G)$;
   $LT_x = \{< a,b > | a \in L_x \wedge b \in G_x \wedge < a,b > \in FT\}$; } }
9. $LT = \cup_{x \in B_{L_Q}} LT_x$;
10. All lattices in LT describe the fragments containing nodes in Q;

end

Figure 4.14: Algorithm 3: Find the Relevant Fragments

Case RQ_2: We have $L_Q = \{21\}$, $G_Q = \{20\}$, E_Q = { (18,20), (19,20), (20,12), (20,21) }, E_L = { (9,12), (16,18), (17,19)}, and E_G = { (12,15), (12,23), (21,23), (20,22) }. **At** *step 5*, we obtain $B_{G_Q} = \{12, 20, 21\}$ and $B_{L_Q} = \{12, 18, 19, 21\}$. At next step, **function** $Block(x, G)$ and $Block(x, L)$ gives $G_{12} = \{15, 23\}$, $G_{20} = \{20\}$, $G_{21} = \{23\}$, $L_{12} = \{9\}$, $L_{18} = \{16\}$, $L_{19} = \{17\}$, and $L_{21} = \{21\}$. Since nodes 18, 19, and 20 are **the** ancestors of each node $x \in B_{G_Q}$ within E_Q, G_{18} = { 15, 20, 23 }, G_{19} = { 15, 20, **23** } and G_{20} = { 15, 23 } are computed at step 7. In the following step, we identify **all** the lattices covering nodes in B_{L_Q}. By the definition of LT_x, $LT_{12} = \{< 9, 15 >\}$, $LT_{18} = \{< 16, 20 >\}$, $LT_{19} = \emptyset$, and $LT_{21} = \{< 21, 23 >\}$. Since $LT_{19} = \emptyset$, we need **to** execute the *while* loop. In that loop, we got $L' = \emptyset$ and $G' = \{22\}$. Then, we **compute** G_{19} = { 15, 20, 23 } $\cup$ {24}, which gives $LT_{19} = \{< 17, 24 >\}$. Thus, the **lattices** in { $< 9, 15 >$, $< 16, 20 >$, $< 17, 24 >$, $< 21, 23 >$ } describe the relevant **fragments** (i.e. fragment 2 and 3) of RQ_2. Next, we discuss the time complexity of **Algorithm** 3.

The time complexity of Algorithm 3 is varied depending the following two cases. **The** first case is when E_G or E_L is equal to $\emptyset$. In this case, the complexity is $O(|E_Q \cup E_L \cup E_G| + |LT| \cdot |FT|)$. The second case is when the first case is not true. Let e_L **represent** the total number of edges belong to lattice L. Then, Algorithm 3 in the **second** case do not traverse more than $O(\sum_{L \in LT} |e_L|)$ edges. Thus, the complexity is $O(\sum_{L \in LT} |e_L| + |LT| \cdot |FT|)$.

4.4 Updating Recursive Relations

There are four different types of updates on a fragmented recursive relation $G_R(V_R, E_R)$. They are *node addition*, *node deletion*, *edge addition*, and *edge deletion*. A new node can be added to V_R or an existing node can be deleted from V_R. This node addition and deletion require updating V_F at the corresponding site. An edge addition (deletion) can connect (resp. disconnect) two existing nodes belonging to either same fragment or two different fragments. This edge update requires revising the local copy of edge set E_R at each site.

These 4 different types of updates must be performed by following the two update constraints given below:

1. A new edge (x, y) can be added to E_R if node x and y already exists in V_R.

2. A node x can be deleted from V_R after all edges incident on node x are deleted from E_R.

The preceding two constraints are introduced to make each type of update *atomic*.

The updates may also require revising the descriptions of some fragments (i.e. lattice covers). If we have to recompute new lattice covers for those fragments affected by updates, the overhead of maintaining the fragment description would be high. However, we can avoid this re-computation by using an efficient incremental update algorithm. This is presented in the following subsections.

4.4.1 Edge Update

Suppose that an edge (x, y) is added (deleted) to (resp. from) DAG $G_R(V_R, E_R)$ at site k. Then, by using Algorithm 3, site k can easily locate which fragments contain node x or y. Assume that fragments F_i at site i and F_j at site j contain node x and y respectively. Note that i,j and k are not necessarily distinct. Based on the above assumption, we describe the detailed update algorithms for edge addition and deletion.

Edge Addition

Site k sends the update information (i.e. an addition of edge (x, y)) to site i and j. After receiving the update information, site i (j) performs the procedure AddEdge((x, y),i) (resp. AddEdge((x, y),j)). This procedure is described in Figure 4.15. The DAG $G_R(V_R, E_R)$ in Figure 4.16 is used as an example to show the execution of procedure AddEdge((x, y),s). Figure 4.16 shows two F-subgraphs whose nodes are stored at site 1 and 2.

Assume that we add edge (13,15) first and then (9,7) to E_R. Site 2 performs AddEdge($(13, 15)$,2) because both node 13 and 15 are available locally. Since A is an empty set, we obtain $L_{13} = \{< 12, 13 >\}$ and $L_{15} = \{< 15, 15 >\}$. Lattices $< 12, 13 >$ and $< 15, 15 >$ are merged into a new lattice $< 12, 15 >$ because lattice $< 12, 15 >$ is suitable. Thus, lattice cover LC2 is changed to $\{< 8, 10 >, < 12, 15 >\}$.

Next, we consider edge $(9, 7)$ addition to E_R. Since site 1 and 2 stores node 9 and

```
AddEdge((x,y),s) /* (x,y): edge; s: site number; */
  begin
  Let LC_s be a lattice cover for a fragment F_s at site s;
  Let E_{F_s} be an edge set corresponding to F_s;
  Let Q_1 = {x}, Q_2 = {y};
  By using E_{F_s}, compute S_{lub_{Q_1}} and S_{glb_{Q_2}};
  Obtain A = { <u,w> | u ∈ S_{lub_{Q_1}} ∧ w ∈ S_{glb_{Q_2}} ∧ <u,w> ∈ LC_s };
  If (A == ∅) {
  /* No lattice becomes unsuitable by the addition of edge (x,y) */
    Obtain L_x = { <a,x> | <a,x> ∈ LC_s };
    Obtain L_y = { <y,b> | <y,b> ∈ LC_s };
    E_R' = E_R ∪ {(x,y)};
    For each <a,x> in L_x
      For each <y,b> in L_y
        If (lattice <a,b> is suitable) {
        /* Suitability checking is done by using E_R' */
          LC_s = LC_s - {<a,x>,<y,b>};
          LC_s = LC_s ∪ {<a,b>}; } }
  else { /* Some lattices in A may become unsuitable */
    For each <u,w> in A
      If (<u,w> is not suitable) {
      /* Suitability checking is done by using E_R' */
        LC_s = LC_s - {<u,w>};
        Apply Algorithm 1 to lattice <u,w>;
        Let LC_s include those lattices returned from Algorithm 1; } }
  end
```

Figure 4.15: Procedure AddEdge($(x, y), s$)

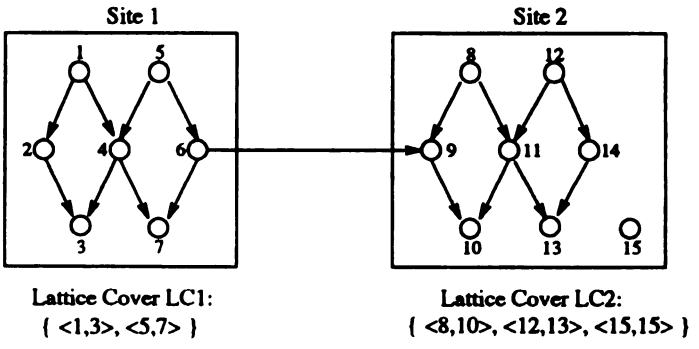


Figure 4.16: A DAG consisting of two F-subgraphs

7 respectively, the site 1 (2) performs AddEdge((9,7),1) (resp. AddEdge((9,7),2)). For site 2, no change is made on lattice cover $LC2$ because of $A = L_9 = L_7 = \emptyset$. For site 1, we got $A = \{ < 5,7 > \}$. Lattice $< 5,7 >$ is not suitable because it includes additional node 9 stored at site 2. Therefore, it is necessary to partition lattice $< 5,7 >$ into a set of suitable lattices. By applying Algorithm 1, we partition the unsuitable lattice $< 5,7 >$ into suitable lattices $< 5,6 >$ and $< 4,7 >$. Thus, we obtain $LC1 = \{ < 1,3 >, < 5,6 >, < 4,7 > \}$.

After performing AddEdge((x,y),i) (AddEdge((x,y),j)), site i (resp. j) notifies to site k its fragment description (i.e. lattice cover) changes. Upon receiving the replies from site i and j, site k updates its local fragment table and E_R. Then, site k broadcasts these changes to all other sites. They in turn update their local fragment table and E_R.

We now discuss the time complexity of procedure AddEdge((x,y),s). Its complexity is determined by the total number of edges to be traversed in E_R'. Let $LUB_x = \{ l \mid < l,g > \in (A \cup L_x) \}$ and $GLB_y = \{ g \mid < l,g > \in (A \cup L_y) \}$ where set A, L_x and L_y are shown in Figure 4.15. We denote a_x (a_y) as the total number of edges to be traversed to compute all descendants (resp. ancestors) of each node in LUB_x (resp. GLB_y). Then, it is easy to see that the time complexity of procedure AddEdge((x,y),s) is $O(a_x + a_y)$.

Edge Deletion

Site k checks if fragments F_i and F_j are two distinct fragments. If they are, then no update is necessary on the fragment table. Site k updates only its local E_R and broadcast this change to all other sites. They in turn updates their local E_R.

If F_i and F_j are same fragment (i.e. $i = j$), then site k sends the update information (i.e. a deletion of edge (x, y)) to site i. After receiving the update information, site i performs the procedure DeleteEdge((x, y),i). This procedure is described in Figure 4.17.

We re-use DAG $G_R(V_R, E_R)$ in Figure 4.16 as an example to show the execution of procedure DeleteEdge((x, y),s). Assume that we delete edge (8,9) first and then (8,11) from E_R. Thus, site 2 first executes DeleteEdge((8, 9),2) and then DeleteEdge((8, 11),2). For DeleteEdge((8, 9),2), we got $A = \{< 8, 10 >\}$. Since S_1 = { 8, 10, 11 } and S_2 = {8, 9, 10, 11 }, $LC2$ is { $< 8, 10 >$, $< 9, 10 >$, $< 12, 13 >$, $< 15, 15 >$ }. For DeleteEdge((8, 11),2), we obtain $A = \{< 8, 10 >\}$. Lattice $< 8, 10 >$ becomes unsuitable because S_1 is an empty set. It is easy to see from the procedure how $LC2$ is changed to { $< 8, 8 >$, $< 11, 10 >$, $< 9, 10 >$, $< 12, 13 >$, $< 15, 15 >$ }.

After performing DeleteEdge((x, y),i), site i notifies to site k its fragment description (i.e. lattice cover) changes. Upon receiving the reply from site i, site k updates its local fragment table and E_R. Then, site k broadcasts these changes to all other sites. They in turn update their local fragment table and E_R. In the next paragraph,

we discuss the time complexity of procedure DeleteEdge((x,y),s).

```
DeleteEdge((x,y),s) /* (x,y): edge; s: site number; */
  begin
  Let LC_s be a lattice cover for a fragment F_s at site s;
  Let E_{F_s} be an edge set corresponding to F_s;
  Let E'_{F_s} = E_{F_s} - { (x,y) }, Q_1 = {x} and Q_2 = {y};
  By using E_{F_s}, compute S_{lub_{Q_1}} and S_{glb_{Q_2}};
  Obtain A = { <u,w> | u ∈ S_{lub_{Q_1}} ∧ w ∈ S_{glb_{Q_2}} ∧ <u,w> ∈ LC_s };
  If (A ≠ ∅) { /* Lattice in A may be affected by the deletion of edge (x,y) */
    For each lattice <u,w> in A {
      By using E'_{F_s}, compute S_1 = { v | u ⪯ v ⪯ w };
      If (S_1 == ∅) { /* Lattice <u,w> is not suitable */
        LC_s = LC_s - { <u,w> };
        LC_s = LC_s ∪ { <u,x>, <y,w> }; }
      else { /* lattice <u,w> is still suitable */
        By using E_{F_s}, compute S_2 = { v | u ⪯ v ⪯ w };
        If (|S_2| ≠ |S_1|) {
        /* The number of nodes lattice <u,w> covers is decreased */
          If (u ≠ x) LC_s = LC_s ∪ { <u,x> };
          If (y ≠ w) LC_s = LC_s ∪ { <y,w> }; } } } }
  end
```

Figure 4.17: Procedure DeleteEdge($(x,y),s$)

The time complexity of procedure DeleteEdge((x,y),s) is determined by the total number of edges to be traversed in E_{F_s}. Let $LUB_x = \{ u \mid < u,w > \in A \}$ and $GLB_y = \{ w \mid < u,w > \in A \}$ where set A is referred in Figure 4.17. We denote d_x (d_y) as the total number of edges to be traversed to compute all descendants (resp. ancestors) of each node in LUB_x (resp. GLB_y). Then, it is easy to see that the time complexity of procedure DeleteEdge((x,y),s) is $O(d_x + d_y)$.

4.4.2 Node Addition and Deletion

Suppose that a new node x is added to DAG $G_R(V_R, E_R)$ at the site k. In this case, we assume that site k wants to store node x in its fragment. Site k first creates a new trivial lattice $< x, x >$. It then updates its local fragment table by adding $< x, x >$ to its fragment description. Site k broadcasts this change to all other sites. They in turn update their local fragment table.

Suppose that a node x is deleted from DAG $G_R(V_R, E_R)$ at the site k. Then, by using Algorithm 3, site k can easily locate which fragment contains node x. Assume that fragment F_i at site i contains node x. Note that i and k are not necessarily distinct. Further note that, by the *2nd* update constraint, all the edges incident on node x are already deleted. Therefore, except for a trivial lattice $< x, x >$, no lattice in the description of fragment F_i covers node x. Site k updates its local fragment table by deleting lattice $< x, x >$ from the description of fragment F_i. It then broadcasts this change to all other sites. They in turn update their local fragment table.

4.4.3 Performance Analysis of Update Algorithms

In this section, we discuss the performance analysis of our update algorithms given in the previous section. We use random F-subgraphs for this analysis. Random F-subgraphs $G_F(V_F, E_F)$ are created the same way as that given in Section 4.2.5.

Let LC_F be a lattice cover for an F-subgraph $G_F(V_F, E_F)$. After each update on G_F, the size of LC_F may change. We denote this changed LC_F by SLO (Semi Local

Optimal)-LC_F. Besides SLO-LC_F, we consider two other LC_Fs after each update on G_F. They are TOP (Theoretically OPtimal)-LC_F and NGO (Near Global Optimal)-LC_F. NGO-LC_F is obtained by applying Algorithm 2 to G_F. In order to show the performance of our update algorithm, it is ideal to compare the size of SLO-LC_F with that of TOP-LC_F. However, TOP-LC_F is computationally very difficult to achieve due to the NP-completeness of LCP. Thus, as an alternative, we compare the size of SLO-LC_F with that of NGO-LC_F.

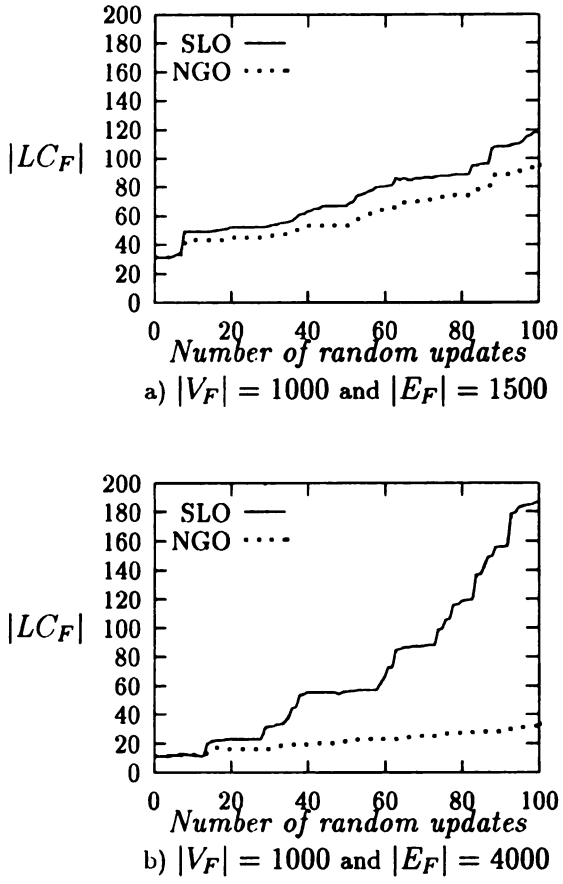
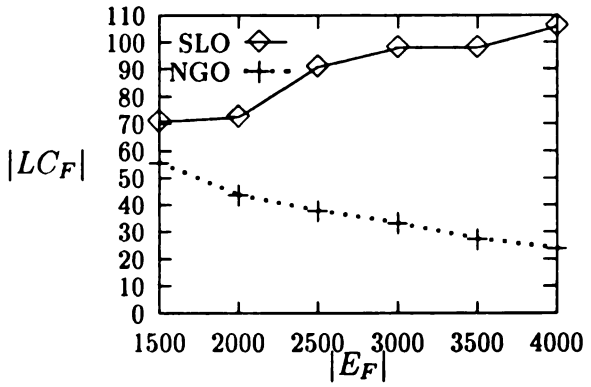


Figure 4.18: Effects of updates on $|LC_F|$ for a) non-dense and b) dense F-subgraphs

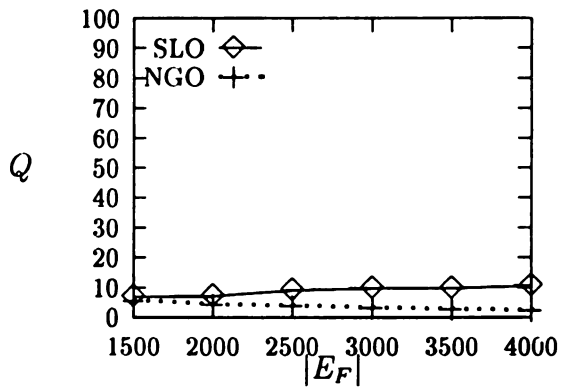
Figure 4.18 shows the effect of random updates on the size of LC_F for a non-dense as well as a dense F-subgraph. Note that random updates consist of *node addition*,

node deletion, *edge addition*, and *edge deletion* where each type of update is randomly chosen. Figure 4.18a shows that, for a non-dense F-subgraph, the performance of our update algorithms deteriorates very slowly. For a dense F-subgraph, the performance deteriorates faster than a non-dense F-subgraph. This is shown in Figure 4.18b. However, in the following paragraph, we will show that this performance degrade will not cause the frequent application of Algorithm 2 to compute NGO-LC_F.

We have measured the impact of updates on $|LC_F|$ for varying denseness of F-subgraphs. In this experiment, $|LC_F|$ is measured after 50 random updates. We averaged the value of $|LC_F|$ for 10 F-subgraphs using 10 different seeds, as we did in section 4.2.5. The results are shown in Figure 4.19. From Figure 4.19a, it is easy to see that the performance of our update algorithms is good for non-dense F-subgraphs while it is not as good for dense F-subgraphs. For a non-dense F-subgraph, SLO-LC_F remains near optimal for a large number of random updates. This is seen in Figure 4.18a and 4.19a. Thus, we do not have to frequently apply Algorithm 2 to obtain NGO LC_F. For a dense F-subgraph G_F, we do not need to frequently obtain NGO LC_F, either. This is justified from Figure 4.19b, which shows that the percentage ratio between $|LC_F|$ and $|V_F|$ for SLO-LC_F grows very slowly with increasing denseness of F-subgraph. Therefore, we conclude that our update algorithms perform well without requiring frequent application of Algorithm 2 for both dense and non-dense F-subgraphs.



a) $|V_F|$=1000



b) $|V_F|$=1000 and $Q = 100 \cdot |LC_F|/|V_F|$

Figure 4.19: Snapshots of $|LC_F|$ and Q after 50 random updates

4.5 Conclusion

In this chapter, we developed a distributed database model for studying the problem of fragmentation of recursive relations. We used the mathematical properties of lattices to investigate fragmentation schemes. We showed that the lattice structures are powerful representation tools for distributed fragments of recursive relations. Both theoretical and empirical performance analysis of the lattice approach are provided. The empirical analysis verifies the near optimality of the performance of our proposed algorithm. We also showed that lattice approach provides an efficient way of locating the relevant fragments of a recursive query.

There are special problems with updating fragmented recursive relations which do not occur in updating fragmented non-recursive relations. We discussed these problems and their solutions. The empirical performance analysis of these solutions are also discussed.

Chapter 5

Conclusion and Future Research

5.1 Contribution of This Dissertation

We have examined recursive query processing strategies for large recursive relations. Recursive query processing requires transitive closure computations. As a special case of transitive closure computation, the shortest path computation is essential for recursive query processing for the large recursive relations of topographical road maps.

In this thesis, we proposed an efficient recursive data organization method called the *HiTi* graph model. The *HiTi* graph model allows structuring of large recursive relations for hierarchical abstraction. Based on the *HiTi* graph structure, we developed a new fast shortest path algorithm named *SPAH*. *SPAH* uses the precomputations captured by the *HiTi* graph to speed up SPSP computation. These

precomputations are not used in the traditional SPSP algorithms, including A*. By using these precomputations, *SPAH* dramatically reduces the explored search space. This is shown through our empirical analysis of *SPAH* on grid graphs. We also studied *intra* and *inter* query parallel processing for shortest path computations within the *HiTi* graph model framework.

The traditional *intra* query parallel shortest path algorithm known as *OTOpar* is not scalable and was shown to be inefficient on the *HiTi* graph structure. We proposed a new parallel *intra* query shortest path algorithm named *PASPAH*. We implemented *PASPAH* and *MOTOpar* on a BBN GP1000, which has a nonuniform memory architecture. The BBN GP1000 multiprocessor currently consists of 85 nodes, linked together by a high speed butterfly switch. In this system, the globally shared memory is the sum of the memories local to all processors. Our empirical analysis of *PASPAH* on grid graphs showed that it is more scalable and performs better than *MOTOpar*. However, our analysis also showed that the average execution time of *PASPAH* is not significantly better than that of *SPAH* and *MOTOpar*. Note that *MOTOpar* is a version of *OTOpar* modified to utilize the *HiTi* graph structure.

Next, we proposed a new *inter* query parallel shortest path algorithm called *ISPAH*. The performance evaluation of *ISPAH* was also performed on the BBN GP1000 shared-memory multiprocessor system by using the same grid graphs as those for *PASPAH*. The empirical analysis of *ISPAH* revealed that *ISPAH* is scalable up to 25 processors although the speedup of *ISPAH* increases almost linearly for up to 10 processors. One of the major performance bottlenecks of *ISPAH* results from

the severe shared-memory access conflicts between processors. We then proposed an improved version of *ISPAH* named *MISPAH*, which reduces the memory-access conflicts through partial data replication. As a result of this replication, *MISPAH* is scalable up to 41 processors. From this analysis, we conjecture that the parallel processing for *inter* query SPSP problems is more promising than for *intra* query SPSP problems.

We then studied the problems of describing and locating the fragments of large recursive relations in a distributed database. These two problems are not primary query optimization issues in a centralized database. In a centralized database, distribution and location of the recursive data are easily handled through hashing or predicates. Thus, its query optimization is primarily done on shortest path or transitive closure computations. However, in a distributed database, the above two problems are the major performance bottlenecks for recursive query processing. This is because the cost of processing recursive queries is strongly dependent on the cost of transmitting information between sites in a distributed database. Traditional use of hashing is not suitable for distributed recursive databases, since they cannot capture the referencing locality of data. We used the mathematical properties of lattices to describe and identify the distributed recursive fragments. We found that lattice structures provide a complete and sound description of recursive fragments. Our theoretical and empirical performance analysis of the lattice approach verifies its effectiveness in distributed databases.

5.2 Future Research

- Shortest path computation on topographical road maps requires accessing the road segments on the disk. Since representation of topographical road maps requires spatial data structures, we need spatial index schemes to enhance the performance of shortest path computation. Traditionally, R, R+, R*, k-d, k-d-b, MD, and GDB trees are used for indexing general spatial objects. However, the above methods are not optimized for the identification of road segments. Thus, more efficient spatial index schemes for the road segments need to be developed.

- Physically clustering adjacent road segments together in the same disk sector is an important research issue for the performance of shortest path computation. If adjacent road segments are not clustered, this will cause very frequent page faults which in turn will slow down the shortest path computation.

- In navigation systems, a shortest path may need to be recomputed because of changing traffic conditions, such as roadblocking. Instead of recomputing an entire shortest path, it may be possible to optimize recomputation based on previously computed path information.

- The *MISPAH* algorithm given in section 3.3.2 is scalable up to 41 processors. More research is needed for further improvement.

APPENDICES

Appendix A

HiTi Graph Based Distributed Transitive Closure Algorithm

In a distributed database, *HiTi* graph can be used for describing and identifying local relations (i.e. *level* 1 subgraphs). That is, each boundary node of a *level* 1 subgraph uniquely describes and identifies the corresponding local relation. Furthermore, the structure of *HiTi* graph allows dynamic decomposition of a transitive closure computation. By utilizing the above two advantages, we developed an efficient distributed transitive closure computation. We first describe two primary algorithms named $TC.1$ and $TC.2$ which will be used for our proposed distributed transitive closure algorithm. Their descriptions are given in Figures A.1 and A.2.

/* U : Relation $U(x, y)$; j: site number; */
/* k : the highest level number of a subgraph tree ST; */
begin
  $i = 1$; $X = \{SG_j^i\}$; $skip$ = TRUE;
  BN_j^i = boundary node information of $S_C(S_A^{i+1}(X))$;
  $R_j^i = \pi_{U.x,U.y,BN_j^i.sg,BN_j^i.site}(U \bowtie_{y=x} BN_j^i)$;
  if $R_j^i = \emptyset$ **then** { $i = i - 1$; **return** $(\emptyset, i)$; }
L1:
  $WB_j^i = (S_B(S_C(S_A^{i+1}(X))) \cup S_W(S_C(S_A^{i+1}(X))))$;
  $R = \pi_{R_j^i.x,WB_j^i.y,R_j^i.sg,R_j^i.site}(R_j^i \bowtie_{y=x} WB_j^i)$;
  while $(R \neq \emptyset)$ {
    $skip$ = FALSE; $R_j^i = R_j^i \cup R$;
    $R = \pi_{R.x,WB_j^i.y,BN_j^i.sg,BN_j^i.site}(R \bowtie_{y=x} WB_j^i) - R_j^i$; }
  $R_j^i = \pi_{R_j^i.x,R_j^i.y,BN_j^i.sg,BN_j^i.site}(R_j^i \bowtie_{y=x} BN_j^i)$;
  if $(i = k - 1)$ **then return**$(R_j^1, R_j^2, \ldots, R_j^i, i)$;
  BN_j^{i+1} = boundary node information of $S_C(S_A^{i+2}(X))$;
  $R_j^{i+1} = \pi_{R_j^i.x,R_j^i.y,BN_j^i.sg,BN_j^i.site}(R_j^i \bowtie_{y=x} BN_j^{i+1})$;
  if ($skip$ = TRUE) **then** $R_j^i = \emptyset$;
  if $R_j^{i+1} = \emptyset$ **then return**$(R_j^1, R_j^2, \ldots, R_j^i, i)$;
  else { $X = S_A^{i+1}(X)$; $i = i + 1$;
        $skip$ = TRUE; **goto L1**; }
end

Figure A.1: Algorithm TC.1: AscendingPhase (U, j, k)

These two algorithms use three different types of relations such as $WB_j^i(x, y)$, $BN_j^i(x, sg, site)$, and $R_j^i(x, y, sg, site)$. Relation $WB_j^i(x, y)$ consists of tuples t where t corresponds to either a *level* i *between* or *within* edge. Relation $BN_j^i(x, sg, site)$ contains tuples t where $t.x$ corresponds to a *level* i boundary node, $t.sg$ corresponds to the notation of the *level* i subgraph to which $t.x$ belongs, and $t.site$ corresponds to the site number storing the *level* 1 subgraph to which $t.x$ belongs. We assume that the subscript of each *level* 1 subgraph corresponds to a site number. For example, the local relation corresponding to SG_3^1 is stored at site 3. Relation R_j^i is used as

a working relation for computing a distributed transitive closure. Thus, each site j has $R_j^i(x, y, sg, site)$ for each *level* i where $1 \leq i \leq k$. Note that we use the same notations as those defined in the chapter 2.3 in $TC.1$ and $TC.2$ algorithms.

```
/* Z : Relation Z(x, y, sg, site); */
/* j: site number; i: level number; */
begin
  i = i − 1; skip = TRUE; X = {SG_j^1};
  X = S_A^{i+1}(X);
  BN_j^i = boundary node information of S_C(X);
  Z = π_{Z.x,Z.y,BN_j^i.sg,BN_j^i.site}(Z ⋈_{y=x} BN_j^i);
  WB_j^i = (S_B(S_C(X)) ∪ S_W(S_C(X)));
  R = π_{Z.x,WB_j^i.y,Z.sg,Z.site}(Z ⋈_{y=x} WB_j^i);
  while (R ≠ ∅) {
    skip = FALSE; Z = Z ∪ R;
    R = π_{R.x,WB_j^i.y,R.sg,R.site}(R ⋈_{y=x} WB_j^i) − Z; }
  if (skip = FALSE) then
    Z = π_{Z.x,Z.y,BN_j^i.sg,BN_j^i.site}(Z ⋈_{y=x} BN_j^i);
  return(Z);
end
```

Figure A.2: Algorithm TC.2: DescendingPhase (Z, j, i)

Algorithm AscendingPhase() traverses *HiTi* graph by following edges with non-decreasing level number. To demonstrate Algorithm TC.1, we use the digraph and its corresponding *HiTi* graph shown in the Figures A.3 and A.4 respectively.

For site 1, which stores the local relation corresponding to SG_1^1, we have WB_1^1 = { (4,6), (5,7), (6,10) }, WB_1^2 = { (10,11), (11,17), (17,19), (17,20) }, BN_1^1 = { (4, SG_1^1, 1), (5, SG_1^1, 1), (6, SG_2^1, 2), (7, SG_2^1, 2), (10, SG_2^1, 2) }, and BN_1^2 = { (10, SG_1^2, 2), (11, SG_2^2, 3), (17, SG_2^2, 4), (19, SG_3^2, 5), (20, SG_3^2, 5) }. Assume that at site 1, AscendingPhase(U, 1, 3) is invoked where U = { (4,4) }. The outputs of

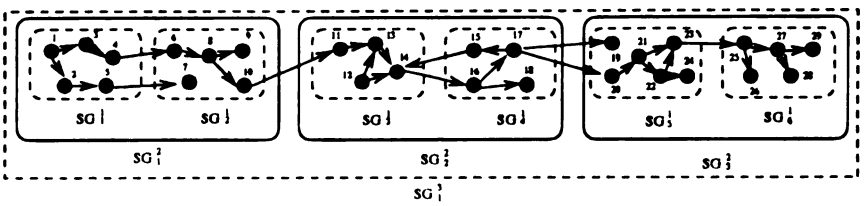


Figure A.3: $G(V, E)$ partitioned according to level 3 subgraph tree

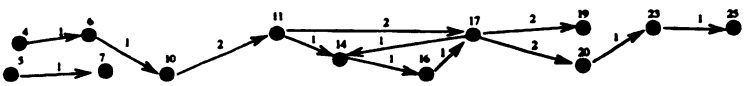


Figure A.4: A level 2 HiTi graph created from Figure A.2

AscendingPhase(U, 1, 3) are stored in relations R_1^1 and R_1^2 where R_1^1 = { (4, 6, SG_2^1, 2), (4, 10, SG_2^1, 2) } and R_1^2 = { (4, 11, SG_2^2, 3), (4, 17, SG_2^2, 4), (4, 19, SG_3^2, 5), (4, 20, SG_3^2, 5) }.

In contrast to AscendingPhase (U, j, k), DescendingPhase (Z, j, i) traverses *HiTi* graph by following edges with a given level number. We use *HiTi* graph shown in A.4 as an example to demonstrate Algorithm TC.2. Assume that at site 3, DescendingPhase(Z, 3, 2) is called where Z = { (4, 11, SG_2^2, 3), (4, 17, SG_2^2, 4) }. Set X is initialized with SG_3^1 and it becomes to contain SG_2^2. Then, site 3 has BN_3^1 = { (11, SG_3^1, 3), (14,SG_3^1, 3), (15, SG_4^1, 4), (16, SG_4^1, 4), (17, SG_4^1, 4) }, and WB_3^1 = { (11,14), (14,16), (16,15), (16,17), (15,14) }. After the execution of *while* loop and the *join* with BN_3^1, Z = { (4, 11, SG_3^1, 3), (4, 17, SG_4^1, 4), (4, 14, SG_3^1, 3), (4, 16, SG_4^1, 4) }.

Based on the above two algorithms, we discussed how to compute transitive closure in distributed database systems. In our distributed transitive closure algorithm, we have two types of sites where a transitive closure is computed. They are *coordinator* and *participant* sites. A site is named a *coordinator* if a transitive closure computation is initially started at this site. The coordinator decomposes a transitive closure computation into a set of sub-computations if needed. A site is called a *participant* if it performs the sub-computation by the request of either a *coordinator* or another *participant*. We give two algorithms TC.3 and TC.4 in Figures A.5 and A.6 where the former is for a *coordinator* and the latter for a *participant*.

```
/* H : a set of nodes; */
/* TC: transitive closure of nodes in H */
/* k : the highest level number of a subgraph tree ST; */
begin
  D1 = π_{H.x,E_j^1.y}(H ⋈_{y=x} E_j^1); D2 = ∅;
  while ( D1 ≠ ∅ ) {
    D2 = D2 ∪ D1;
    D1 = π_{D1.x,E_j^1.y}(D1 ⋈_{y=x} E_j^1) − D2; }
  AscendingPhase(D2, j, k) returns (R_j^1, R_j^2, ..., R_j^i, i);
  for (m = i downto 1 by -1) {
    Partition R_j^m such that each partition Γ
    share the same value of R_j^m.sg;
    Let Σ consists of Γs of R_j^m;
    for each Γ ∈ Σ {
      s = a majority value of Γ.site;
      SendParticipant (Γ, m, j, s); } }
  TC = D2 ∪ Wait_Result_From_Participant(R,⋆);
  /* ⋆ represents a site number */
end
```

Figure A.5: Algorithm TC.3: CoordinatorSite $j(H,TC,k)$

```
/* Γ : Relation Γ(x, y, sg, site); */
/* m: level number; j: coordinator site number; */
begin
  Receive(Γ, m, j, ⋆);
  /* ⋆ represents a site number */
  if (m = 1) {
    D = π_{Γ.x,E_j^1.y}(Γ ⋈_{y=x} E_j^1); R = π_{x,y}(Γ);
    while ( D ≠ ∅ ) {
      R = R ∪ D;
      D = π_{D.x,E_j^1.y}(D ⋈_{y=x} E_j^1) − R; }
    SendCoordinator(R, j); }
  else {
    DescendingPhase (Γ, j, m) returns Z;
    Partition Z such that each partition ϒ
    share the same value of Z.sg;
    Let Σ consists of ϒs of Z;
    for each ϒ ∈ Σ {
      Check if any value of ϒ.site is s;
        If yes then h = s;
        else h = a majority value of ϒ.site;
      If (h ≠ s) then notify coordinator site j
      that site h will send the coordinator the
      partial results of the transitive closure;
      SendParticipant (ϒ, m − 1, j, h); } }
end
```

Figure A.6: Algorithm TC.4: ParticipantSite $s(\Gamma, m, j)$

To demonstrate Algorithm TC.3, we also use a recursive relation $G(V, E)$ and *HiTi* graph shown in Figures A.3 and A.4 respectively. Assume that a recursive query "Find all descendants of node 3" is issued at site 1. Then, site 1 becomes *coordinator* and algorithm TC.3 is executed at site 1. Algorithm TC.3 first computes a transitive closure of node 3 (i.e. $H = \{ 3 \}$) with respect to SG_1^1 (V_1^1, E_1^1). As a result, $D_2 = \{ (3,4) \}$ and AscendingPhase (D_2, 1, 3) is called. After the execution of AscendingPhase (D_2, 1, 3), we got $i = 2$, $R_1^1 = \{ (3, 6, SG_2^1, 2), (3, 10, SG_2^1,$

2) } and R_1^2 = { (3, 11, SG_2^2, 3), (3, 17, SG_2^2, 4), (3, 19, SG_3^2, 5), (3, 20, SG_3^2, 5) }. R_1^2 is partitioned into two relations Γ_1 = { (3, 11, SG_2^2, 3), (3, 17, SG_2^2, 4) } and Γ_2 = { (3, 19, SG_3^2, 5), (3, 20, SG_3^2, 5) }. Next, *coordinator* site 1 sends its decomposed sub-transitive closure computation requests to site 3 and 5 by executing SendParticipant (Γ_1, 2, 1, 3) and SendParticipant (Γ_2, 2, 1, 5). Note that for Γ_1, site 4 could be chosen as a *participant* since there is no majority site number of elements in Γ_1. For R_1^1, since all tuples share the same $R_1^1.sg$ (i.e. SG_2^1), we have only one Γ_1 = R_1^1 in Σ and SendParticipant (Γ_1, 1, 1, 2) is executed. Algorithm TC.4 is executed at a participant site upon receiving the sub-transitive closure computation request (e.g. SendParticipant (Γ_1, 1, 1, 2)) from a coordinator.

To show how participant sites work, SendParticipant (Γ_1, 2, 1, 3) and SendParticipant (Γ_1, 1, 1, 2) mentioned in above paragraph are considered as examples. For SendParticipant (Γ_1, 2, 1, 3), since m = 2, site 3 first calls DescendingPhase (Γ_1, 1, 2) and gets Z = { (3, 11, SG_3^1, 3), (3, 14, SG_3^1, 3), (3, 16, SG_4^1, 4), (3, 17, SG_4^1, 4) }. Relation Z is partitioned into two relations Υ_1 = { (3, 11, SG_3^1, 3), (3, 14, SG_3^1, 3) } and Υ_2 = { (3, 16, SG_4^1, 4), (3, 17, SG_4^1, 4) }. For Υ_1, site 3 sends its decomposed transitive closure computation requests to itself by executing SendParticipant (Υ_1, 1, 3, 3). For Υ_2, site 3 first notifies the coordinator site 1 that site 4 will send partial results of the transitive closure. Site 3 then sends its decomposed sub-transitive closure computation to site 4 by executing SendParticipant (Υ_2, 1, 3, 4). As for SendParticipant (Γ_1, 1, 1, 2), since m = 1, site 2 computes a transitive closure with respect to SG_2^1 (V_2^1, E_2^1). Relation R contains { (3,6), (3,8), (3,9), (3,10)

} and site 2 sends R back to the coordinator site 1 by executing SendCoordinator (R, 1).

Appendix B

Empirical Analysis of Computing Shortest Path for Road Map Queries

B.1 Performance comparison of shortest path algorithms

In this section, we empirically compare the performance of Dijkstra's and Nicholson's shortest path algorithms on road map graphs. Their pseudo-codes are described in Figure B.1 and B.2 respectively. For the detailed demonstration of Nicholson's algorithm, readers can refer to Nicholson's paper in [77]. The inputs to the following algorithms are graph $G(V, E)$, source and destination nodes s and d respectively.

Note that $l(x, y)$ represents the cost associated with each edge $(x, y) \in E$.

1. For each node $u \in V$, $\lambda(u) = \infty$.
2. Let $\lambda(s) = 0$, $FS = \{s\}$, and $ES = \emptyset$.
3. Select a node u in FS for which $\lambda(u)$ is minimum.
4. $FS = FS - \{u\}$ and $ES = ES \cup \{u\}$
5. If $(u = d)$, then $\lambda(u)$ is the shortest path cost and Stop.
6. For every edge (u, v) in E, if $\lambda(v) > \lambda(u) + l(u, v)$,
   (a) Let $\lambda(v) = \lambda(u) + l(u, v)$.
   (b) Let $FS = FS \cup \{v\}$ if $v \notin (FS \cup ES)$.
7. Go to step 3.

Figure B.1: Dijkstra's shortest path algorithm

To compare the performance of Dijkstra's and Nicholson's algorithms on road map graphs, we create two dimensional grid graphs $G_R(V_R, E_R)$ with 4 adjacent nodes. The grid has $N * N$ nodes where each row and column includes N nodes. Thus, in a graph G_R, $|V_R|$ and $|E_R|$ are equal to $N * N$ and $4 * N * (N - 1)$ respectively. These two dimensional grid graphs are considered as typical examples of road maps [64]. Based on the above scheme, we create a grid graph G_R where $|V_R| = 200 * 200$ and $|E_R| = 4 * 200 * 199$. The cost of each edge in E_R is generated based on a uniform distribution [1, 100] with 10 different seeds. As a result, we have 10 different grid graphs G_Rs.

For each grid graph created above, we compute 50 different shortest paths for 50 randomly prefixed pairs of source and destination nodes. Let $DIJ(NIC)$ be the total number of nodes visited at step 6 (at step 5.(a).i and 6.(a).i) by Dijkstra's (Nicholson's) algorithms. Note that we also count the nodes re-visited for $DIJ(NIC)$. We compare the two algorithms by observing the ratio DIJ/NIC. These values are

1. For each node $u \in V$, $\lambda_s(u)=\lambda_t(u)=\infty$. Let $FS_s = FS_t = \emptyset$ and $x = y = 0$.
2. For each edge (s, u) in E, $\lambda_s(u) = l(s, u)$ and $FS_s = FS_s \cup \{u\}$.
3. For each edge (d, v) in E, $\lambda_t(v) = l(d, v)$ and $FS_t = FS_t \cup \{v\}$.
4. Let $C_s = \min_{\lambda_s(u)>x} \lambda_s(u)$ and $C_t = \min_{\lambda_t(v)>y} \lambda_t(v)$ where $u \in FS_s$ and $v \in FS_t$.
5. If $(C_s \leq C_t)$,
   (a) For each m for which $\lambda_s(m) = \min_{\lambda_s(u)>x} \lambda_s(u)$ for all $u \in FS_s$
      i. For every edge (m, u) in E, if $\lambda_s(u) > \lambda_s(m) + l(m, u)$,
         A. $\lambda_s(u) = \lambda_s(m) + l(m, u)$
         B. if $u \notin FS_s$, $FS_s = FS_s \cup \{u\}$.
   (b) Reset $x = \min_{\lambda_s(u)>x} \lambda_s(u)$.
6. If $(C_t < C_s)$,
   (a) For each m for which $\lambda_t(m) = \min_{\lambda_t(v)>y} \lambda_t(v)$ for all $v \in FS_t$
      i. For every edge (m, v) in E, if $\lambda_t(v) > \lambda_t(m) + l(m, v)$,
         A. $\lambda_t(v) = \lambda_t(m) + l(m, v)$
         B. if $v \notin FS_t$, $FS_t = FS_t \cup \{v\}$.
   (b) Reset $y = \min_{\lambda_t(v)>y} \lambda_t(v)$.
7. If $\min_{u \in FS_s \cup FS_t}(\lambda_s(u) + \lambda_t(u)) \leq (\min_{\lambda_s(w)>x} \lambda_s(w) + \min_{\lambda_t(v)>y} \lambda_t(v))$ where $w \in FS_s$ and $v \in FS_t$, the shortest path cost is $\min_u(\lambda_s(u) + \lambda_t(u))$ and Stop.
8. Go to step 4.

Figure B.2: Nicholson's shortest path algorithm

then averaged over the 10 different grid graphs with same source and destination nodes. These are shown in Figure B.3.

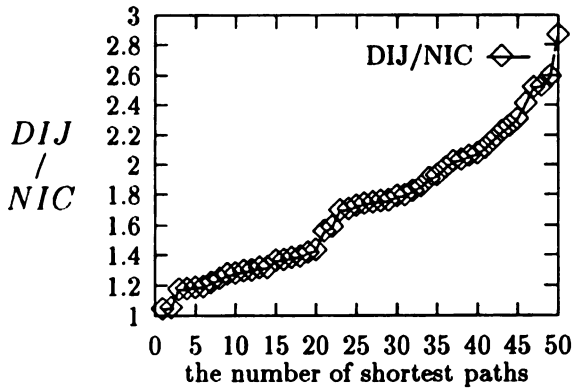


Figure B.3: Performance comparison between Dijkstra and Nicholson's algorithms

Figure B.3 clearly shows that Nicholson's algorithm visit far less nodes than Dijkstra's when they are applied to grid graphs. It is interesting to observe that the ratio DIJ/NIC is small for shorter paths. This happens when source and destination nodes are closely located in the graph. However, even in this case, Nicholson's algorithm still performs slightly better than Dijkstra's. Therefore, we can conclude from Figure B.3 that Nicholson's algorithm is superior to Dijkstra's when graphs capture road maps.

Next, we present our shortest path algorithm in Figure B.4 which finds shortest path by simultaneously constructing two trees, one rooted at the source node and the other at the destination node. This simultaneous expansion of two trees is the primary difference between our and Nicholson's algorithms. Note that the two trees in Nicholoson's algorithm are expanded alternatively. That is, if one tree has the next shortest route than the other, the one with the next shortest route is expanded. This alternating tree expansion may cause one tree to be expanded more than the

1. For each node $u \in V$, $\lambda_s(u)=\lambda_t(u)=\infty$ and Marked$(u) = 0$.
2. Let $\lambda_s(s) = \lambda_t(d) = 0$, Marked$(s)=1$ and Marked$(d)=2$.
3. Let $FS_s=\{s\}$, $FS_t=\{t\}$ and $E_s = E_t = \emptyset$.
4. Select m_s and m_t respectively from FS_s and FS_t such that $\lambda_s(m_s)$ and $\lambda_t(m_t)$ are minimum. Let $C_1 = \lambda_s(m_s) + \lambda_t(m_t)$.
5. Let $FS_s = FS_s - \{m_s\}$, $ES_s = ES_s \cup \{m_s\}$, $FS_t = FS_t - \{m_t\}$, $ES_t = ES_t \cup \{m_t\}$.
6. Obtain $I = \{ u \mid \text{Marked}(u) = 2 \text{ for all } u \in FS_s \}$.
7. If $I \neq \emptyset$, $C_2 = \min_{u \in I}(\lambda_s(u) + \lambda_t(u))$. Otherwise $C_2 = \infty$.
8. If $(m_s = m_t)$, the shortest path cost is $\min(C_1, C_2)$ and Stop.
9. If $(C_2 \leq C_1)$, the shortest path cost is C_2 and Stop.
10. If (Marked$(m_s) = 0$), Marked$(m_s) = 1$. If (Marked$(m_t) = 0$), Marked$(m_t) = 2$.
11. For every edge (m_s, u) in E,
    (a) If $\lambda_s(u) > \lambda_s(m_s) + l(m_s, u)$,
        i. $\lambda_s(u) = \lambda_s(m_s) + l(m_s, u)$
        ii. If $u \notin (FS_s \cup ES_s)$, $FS_s = FS_s \cup \{u\}$
12. For every edge (m_t, v) in E,
    (a) If $\lambda_t(v) > \lambda_t(m_t) + l(m_t, v)$,
        i. $\lambda_t(v) = \lambda_t(m_t) + l(m_t, v)$
        ii. If $v \notin (FS_t \cup ES_t)$, $FS_t = FS_t \cup \{v\}$
13. Go to step 4.

Figure B.4: Our two tree expansion shortest path algorithm

other. As a result, this unbalanced tree expansion may delay finding the shortest path. However, in our approach we expand the shortest paths of the two trees. In order to show the performance comparison between these two algorithms, we used the same data set and methods as we used for those between Dijsktra's and Nicholson's. This is shown in Figure B.5.

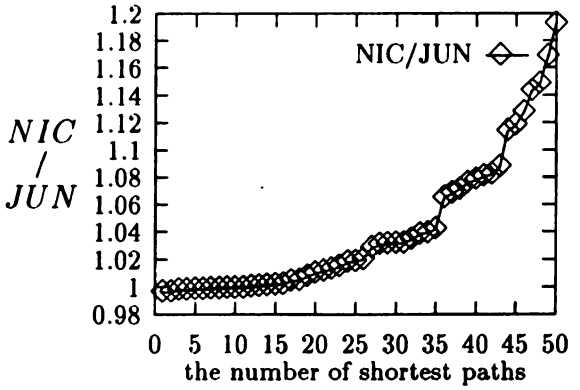


Figure B.5: Performance comparison between Nicholson's and Our algorithms

In Figure B.5, JUN represents the total number of nodes visited (including the re-visitation) at step 11.(a) and 12.(a) by our algorithm. Figure B.5 clearly verifies our conjecture that Nicholson's alternating tree expansion delays finding the shortest path. From this figure, we can conclude that our algorithm on the average always performs better than Nicholson's.

B.2 Conclusion

In this appendix, we studied a single pair shortest path problem on the domain of road map queries. We have compared Nicholson's shortest path algorithm with that of Dijkstra. Our empirical analysis shows that Nicholson's two tree expansion approach

outperforms Dijkstra's one tree expansion method for road map queries. We proposed a more efficient two tree expansion shortest path algorithm than Nicholson's. This is justified through empirical analysis.

Appendix C

A Survey of Database Problems in Intelligent Transportation System

C.1 Introduction

Intelligent Transportation Systems, ITS, are increasingly being used to automate the navigation function of the automobile. The navigation infrastructures such as Global Positioning Systems, GPS, and beacons have a profound impact on the design and development of modern navigation systems. Along with this, ITS also requires powerful software systems including very large database system support. The development of these infrastructures requires substantial resources involving coordination on a mammoth scale. The U.S. government has played a major role in providing the necessary leadership in the development of these infrastructures. Software systems and databases have been developed by governmental as well as non-governmental agencies

such as Universities and corporations. These infrastructures and the software systems are built on top of large digital map databases. Much work has been done in the area of Geographic Information Systems to create large databases of digital maps. In this paper we survey the database work which is relevant to ITS. We also discuss other important work that is necessary for an understanding of ITS.

ITS must be able to look up information on a digital map in order to automate navigation. A digital map not only stores location information about objects, but also shows the relative locations of objects. These objects are of differing dimensions such as points, line segments, and regions [106]. Multidimensional objects require more complex modeling and processing than those required in traditional applications. A database design for ITS must also capture the spatial relationships between these objects. One important spatial relationship is that of neighbor. The clustering of neighbor objects in the same area of the disk is an important database research issue [93]. A competing topic is the dispersion of stored neighbor objects to different disks to enable multiprocessing [55].

The database structures for these very large databases must be effective in quickly accessing multidimensional objects. For example, the map database stored for a system developed locally for the Twin Cities in Minnesota is approximately 10 Mbytes [92]. A system which uses country or continent wide data will access a much larger database. Efficient organization of large multidimensional databases for fast access is a paramount database research issue in this area. Accessing large databases of multidimensional objects requires efficient spatial indexing schemes. In section C.2

we discuss such indexing schemes. The currently available commercial navigation systems use one or more of these index methods. However the actual details of the indexing scheme used by particular current commercial systems is not known because they are proprietary.

The currently available position systems, such as GPS and dead reckoning, play an important role in the design of these database systems. If the GPS system is used it automatically determines latitude, longitude, and altitude [1]. However, the returned position is not accurate. If dead reckoning is used, the position of a vehicle is known precisely at certain locations. As the vehicle travels, the new location is calculated. Dead reckoning systems often use a beacon to find the initial known position. The position of the vehicle is then calculated as a displacement from the known position as the vehicle moves. This calculation of displacement is also prone to errors. An important issue in designing ITS is to minimize these errors. The correct determination of a vehicle's current location is essential for the successful operation of an automated navigation system. Positioning technologies such as GPS and dead reckoning are often combined to deliver the desired accuracy needed by ITS [61, 67].

In ITS, quick real time response to large database queries is essential. The simplest query in ITS is "where am I?" This query is answered partially by the positioning technology which determines the coordinates of the vehicle. These coordinates are usually mapped to an object in the database such as a landmark, a road, or a region. An important database design issue is the representation of the inherent hierarchy of the data such as roads and landmarks within cities, and cities within counties. An

example query using this hierarchy is "List all counties in Michigan which intersect Interstate Highway I94." A number of typical road map queries have been discussed in [92].

Another issue of importance is the placement of the data; on board the vehicle or at a central site. If the data is on board the vehicle, then each vehicle must have a copy of the necessary data in order to answer the queries. Real time updates through a central system is expensive. These updates must be communicated in real time for up-to-date queries at the vehicle. If the data is maintained at some central site, the vehicle must access this remote database in order to receive the data or answers to its queries. In this model the maintenance of the database is simpler but at the cost of increased communication overhead.

The remainder of this paper is organized as follows. In section C.2 of this paper we describe the data model for ITS, and discuss classes of road map queries for ITS in section C.3. In section C.4 we briefly review the existing positioning technologies. Section C.5 describes a number of commercially available intelligent transportation systems.

C.2 Data Models for Vehicle Navigation Systems

ITS databases contain information about objects such as landmarks, roads, and geographic regions We can classify these objects as 0-Dimensional, 1-Dimensional, 2-Dimensional, and 3-Dimensional objects [106]. 0-Dimensional objects are points and

represent landmarks such as building sites or intersections of roads. A typical 1-Dimensional object is a road segment, and it is a primary object for road map queries. Roads are formed from straight line segments which in turn form the network of roads. Examples of 2-Dimensional objects are geographic regions such as cities, counties, and states. 3 dimensional objects are necessary for recognizing certain types of landmarks. An object such as a building must be described by latitude, longitude, and altitude. Efficient spatial indexing schemes for these multidimensional objects are critical to the performance of ITS queries.

C.2.1 Indexing Schemes for accessing multidimensional road objects

Indexing schemes for ITS databases can be classified into 2 groups based on the type of object indexed. They are point indexes for point objects, and indexes for higher dimensional objects. We first discuss point indexes. In order to index objects such as line segments or regions, each vertex is a point to be indexed. Alternatively, the object may be considered as a point in a higher dimensional space. Point indexing schemes include quad-trees, oct-trees, point quad-trees, k-d trees, k-d-B-trees, and the grid file.

A quad-tree is an indexing scheme for objects in a planar region. A planar object consists of a collection of points in the plane. The quad-tree indexes the planar object by dividing the planar region into regular quadrants. This is a recursive process, and

halts when all points in a quadrant are of the same type, i.e., part of the object or not part of the object. The oct-tree is a generalization of the quad-tree to 3 dimensional space. The quad-tree can be further generalized to any k-dimensional space. Quad-trees are efficient mechanisms for computer vision, but not for ITS. The quad-tree (or oct-tree) is not height balanced. The point quad-tree (or point oct-tree) picks points of interest at which the plane (or 3-dimensional space) is subdivided into unequal size quadrants. This gives an index tree which is closer to being height balanced [88].

The k-d tree is a generalization of the point quad tree. A k-d tree is a binary tree in which k levels of the tree are necessary to partition the k-dimensional region into 2^k regions. One significant difference between k-D trees and point quad-trees is that in k-d trees these 2^k regions do not meet in a single point [88].

The grid file approach partitions a k-dimensional space into non-uniformly sized k-dimensional grid blocks in order to index the data points located within the blocks [76]. This grid file approach is not suited to indexing spatial objects of higher dimension than points. The primary reason is that k-dimensional objects are mapped into points in 2k-dimensional space in the grid file approach. As a result, most grid blocks will not be used. Various versions of the grid file have been proposed in order to improve their performance [28, 63, 96, 12]. Freeston [28] has developed a nested and balanced grid (BANG) file which has a tree structured directory. This tree structured directory has the self height balancing B-tree property. Kriegel and Seeger [63] have proposed a method called PLOP-Hashing to eliminate the grid directory. This is a multidimensional extendible hashing scheme which expands the pages of the file

corresponding to grid blocks in a piecewise linear fashion. However, the problem with PLOP-Hashing is that it can not fully capture natural ordering of higher dimension spatial objects. Six and Widmayer [96] present a multilayer paradigm for transforming point index structures (such as the grid file) into index structures for k-dimensional intervals. This paradigm allows the preservation of the natural ordering of interval data. Blanken et. al. [12] propose a generalized grid file which more fully preserves the natural ordering between geometric objects. The generalized grid file integrates the B^+-tree, the grid file, and the k-d-B-tree.

The second group of indexing schemes are better suited to indexing k-dimensional objects in k-dimensional space. The most important indexing scheme is the R-Tree [38]. The R-tree is a generalization of the B-tree to higher dimension objects. All objects are in the leaves of the tree, and the leaves are all at the same level. The R-tree functions by enclosing objects in rectangles. Nodes of the R-tree are formed by enclosing groups of rectangles in a larger rectangle which becomes an R-tree node. A problem with the R-tree structure is that these rectangle nodes in the tree overlap, which causes the lookup procedure to follow multiple tree paths. Several modifications to the R-tree have been developed to alleviate this problem, such as the R+-tree, the R*-tree and the packed R-tree [10, 35, 79, 87]. These alterations to the R-tree prevent overlap (R+-trees) or minimize the overlap by using a complicated algorithm (R*-trees). A weakness of the R-trees is that rectangles are the stored object. Another weakness is that R-trees have poor storage utilization. Objects such as diagonal line segments and convex objects are not well represented by bounding rectangles.

The cell tree is a modification of the R-tree in which the bounding regions do not need to be rectangular [36]. Jagadish uses bounding polyhedra in the P-tree [54] to eliminate the space wasted by bounding rectangles. Jagadish has also proposed a line segment index scheme based on the Hough transform. This index scheme always outperforms minimum bounding rectangle index schemes for retrieving line segments which go through a specific point or intersect a specific line segment.

Several structures improve the worst case storage utilization found in R-trees. The MD-tree [75], like the R-tree, is a generalization of the B-tree to higher dimensions utilizing bounding rectangles. The storage utilization of the MD-tree has a worst case of 66.7% due to the dynamic readjustment of split regions. The GBD-tree [78] is similar to the R-tree, but uses extra data to increase the storage utilization. GBD-trees also speed the processes of insertion and deletion over R-trees, but at a higher memory cost and at a higher CPU cost for large extent data. Freeston has proposed a generalization of n-dimensional B-trees in which the worst case performance is controlled [29]. This solution recursively partitions the data space, guarantees logarithmic access and update, and has a worst case 33% storage utilization. The hB-tree proposed by Lomet and Salzberg [69] has competitive storage utilization. The hB-tree differs from B^+-trees in that the index terms are organized into k-d trees.

Kolovson and Stonebraker propose segment indexes [62] as an extension to database indexing structures such as R-trees. These segment intervals improve the search performance on interval data with non-uniform length distributions.

C.2.2 Large Database Issues

The amount of stored data is massive. If the data is kept entirely at the vehicle, then the vehicle system is responsible for all queries and updates. This requires a powerful system in the vehicle, both in terms of storage space and processing power in order to access the information. External communications will be minimum for this configuration. If the data is stored entirely at some site outside of the vehicle, then the vehicle system will be primarily a communications system. The external processor(s) must be extremely powerful since all the vehicles in the processing area will be sending their information requests to that external site. A third alternative is to distribute the data and processing between the external site and the vehicle. In this case, the vehicle can download the needed data from an external system, then answer queries on that downloaded data. Troy Michigan has a system developed by Siemans which uses this distributed approach.

There are currently a number of systems under development and testing. These systems are restricted to a small geographic area. In order to develop a system which is viable in a large scale system, such as the continental United States, large database issues such as the clustering of data must be explored. Shekhar et. al., [94] discuss a partitioning technique based on max cut partitioning of a similarity graph whose nodes are data items. The similarity is the probability that the data items will be accessed together. This technique assigns the data items to distinct disks for a speed up using parallel processing. The size of the database is indicated by Shekhar, et.al [92], in that the storage size of the data for the Twin Cities metropolitan area

in Minnesota is 10 Mbytes. Jagadish discusses clustering of data so that objects which are close in multidimensional space are close together after mapping into a one dimensional [55] in order to minimize disk accesses.

C.2.3 Update

There are two different types of algorithms which are useful in automated navigation. A real-time system must be continuously updated according to the changes in road conditions. This also implies that the shortest route is dynamic; that is, the route planned at the beginning of the trip may not be the route finally taken. Methods to alter an already computed path using the computed path information could be faster than completely determining the path from the new current location.

For long trips, a pre-computed path may be used as part of the overall trip. If it can be determined that the route from source to destination must pass through certain locations, then shortest path information which is already computed between those locations can be used in order to improve the speed of the route calculation.

Updating the multidimensional objects in the database can be costly. The cost depends on the storage mechanism used for the objects. There is also the problem of changing the stored information about the relative locations of objects. For example, if a road is closed, all the information about the intersecting roads must be updated.

C.3 Queries

On a 0-D object, a typical query is to find the landmark (or nearest landmark) for a particular location. Locations are to be described by latitude and longitude. The queries on 1-D objects are more common, since they include all the find the path types of queries. Some of these queries are: find the shortest path between two locations; find the shortest path between two locations going through some list of other locations; find the shortest path between two locations not going through any locations in some list. There is also the query to determine which road segment goes through a particular location. The database must include the connectivity information about road segments in order to answer these queries. One practical problem that arises is that there is as yet no accurate large area map. The most common starting point for automated navigation is the Tiger file produced by the U.S. Census Bureau. These files have only road map accuracy.

For 2-D objects, typical queries are: which region includes a particular landmark, which region includes a particular road segment. Another query type is to determine the neighboring regions of a particular region.

Of all these types of queries, the predominating ones are the find the path queries. These types of queries are usually recursive, and can be answered by many types of algorithms, such as transitive closure, recursive query processing, partial transitive closure, depth first search, and A* [2, 3, 4, 7, 8, 21, 22, 26, 30, 40, 42, 46, 48, 49, 50, 52, 53, 57, 59, 70, 99, 101, 103]. The interface between the system and the driver

must be as unobtrusive as possible. A distracting system could become the cause of accidents. Some possibilities include voice recognition or point to menu choices for input. For output, voice is the least distracting; reading a map with a blip to indicate a location on a map is more distracting.

C.3.1 Optimization

Storing the data in an hierarchic fashion can help optimize the query processing. One method of storing road map data which has been explored is storing the road information in a hierarchy. That is, interstate expressways are at level 1, state highways are at level 2, and county roads are at level 3. An example of the use of this in finding the path algorithms is that a path from a source to a destination uses level 2 and level 3 roads in the immediate area of the source and destination; level 1 roads are used to find a path between these areas. This type of method does not guarantee an optimal path, however, the amount of processing to find a good path is greatly reduced.

In general, the queries must access objects of various types. Indexing schemes such as k-d-trees and point quad trees are efficient at finding point data (regions can be transformed into higher dimension points). These methods suffer from an unbalance in the tree heights. This leads to multiple disk accesses for a single object. The methods based on B-trees, such as the R-tree, are better in terms of finding a leaf node in the tree which contains the desired search object, however, multiple leaves may be discovered in the search due to overlap. The MD-tree and GDB-tree were developed to decrease the number of disk accesses in a search for an object. Shekhar

and Liu have presented a clustering method for storing objects such that connected objects are likely to be stored in the same disk sector [93] Shekhar and Yang have described MoBiLe Files in which the population density of objects determines the sector location for storage [90].

The number of processors available to answer queries is a vital factor in optimization. For a single processor system, storing neighboring objects in the same bucket is efficient, since all probable desired objects can be retrieved with a single disk access of that bucket. In a multiprocessor environment, it is more efficient to store the neighboring objects in different buckets on different disks. All probable desired search objects can be accessed in parallel.

C.3.2 Query Types

We divide the types of queries into two categories, shortest path, which has been extensively covered in the literature, and all other types of queries which we discuss in this section. One query is to retrieve certain information about an object in the database. For example, return the name of a road at a particular latitude and longitude; list the nearest landmark(s) (building, road intersection, etc.) to a certain location. Geographical data is inherently hierarchical. Queries are often asked in this context: list all the restaurants in a particular city; list all the roads in a particular county which are currently undergoing repair. Another classification of queries is by neighborhood. List all the national parks near I96; list all public colleges within 200 miles of home. A query such as list all the roads which intersect Maple street

within a particular county will use hierarchic information as well as the connectivity information which is necessary to answer shortest path queries.

C.4 Positioning Technology

One of the main components of a navigation system is the current position determination system. There are a number of hardware systems available for determining the current position. Typical examples of them include global positioning system (GPS), differential global positioning system (DGPS), Loran-C, dead reckoning, and map matching. The above systems can be used either independently or combined in the navigation system to enhance the accuracy of the vehicle's position. In this section, we discuss the above systems.

The U.S. government has placed satellites in orbit around the earth. GPS functions by sampling signals from several satellites. The hardware systems are able to use this sampled information to determine the absolute current position of the vehicle in latitude and longitude. GPS is inaccurate and the error can be as great as 100m [61]. Many papers have been published which show methods of reducing this error [1, 27, 100].

DGPS is a combination of on-board GPS hardware and GPS hardware at a base station. Here, the precise location of the base station is known. It is assumed that the positional error, determined by on-board GPS hardware, is identical to that determined at the base station. By adjusting the reported position of the vehicle according

to the error at the base station, the error factor can be reduced to as little as 5m. Problems with DGPS are that the base station must communicate with the vehicle, requiring additional capabilities on the receivers [27].

Loran-C is a system for sending location information by radio signals from land based sites. Loran-C signals are propagated along the earth's surface. The vehicle hardware samples signals from multiple transmitters, and determines a location. The positional error of Loran-C is 500m, but can be reduced to 100m. This compares favorably with that of GPS [66].

Dead Reckoning is a method of locating one's current position on a map, given an accurate starting position and the vectors representing headings and velocities of vehicles. The instruments that are used in dead reckoning to measure changes in a vehicles direction and speed includes:

- Wheel odometers for measuring distance, these are subject to changes in tire inflation, tire tread wear and slippage.
- Compasses for determining direction, these are subject to errors due to the magnetic field of the vehicle and changes in the earth magnetic field in different locations.
- Gyroscopes for measuring angular velocities of the vehicle, they suffer from changing temperatures which cause drifting problems.
- Accelerometers for measuring changes in acceleration, both Gyroscopes and accelerometers cannot accurately measure small changes in movement [67].

It is possible to precisely determine the current position if a complete and accurate information about a vehicles movements and its starting point is given. However, in reality, the information gathered from the instruments for a vehicle's movements

is likely to be inaccurate. As a result, the dead reckoning method accumulates errors as the vehicle continues to move [67]. This error accumulation is the primary disadvantage of the dead reckoning method.

Map matching is a method of correcting errors in vehicle's position caused by either GPS or dead reckoning systems. It tries to locate a logical path on the map that closely resembles the physical path traveled. For example, if after traveling 5 miles from the starting point the vehicle turns 90 degrees to the right, the map-matching algorithm will search for a road to fit the new direction.

C.5 Current Systems

In this section, we describe currently operational vehicle navigation systems in Europe, U.S.A. and Japan. The current vehicle navigation systems can be classified based on 3 different positioning technologies. They are systems based on dead reckoning with map matching, GPS, and the combined use of GPS and dead reckoning with map matching.

The systems based on the dead reckoning with map matching include Ali Scout, Euro-Scout, Autoguide, Carin, and ETAK Travelpilot [14, 15, 18, 85, 97, 98]. Among these system, ETAK Travelpilot is a stand-alone system which does not require any infrastructure [14, 18]. Its map databases are stored in one CD-ROM disk. Travelpilot provides two major functions to a driver. One is to display to the driver a road map of the area surrounding the car. The other is to locate destinations by the

street address, and to indicate those destination on the map. The main disadvantage of this system is that it does not provide a route guidance to the driver. The rest of the systems [15, 85, 97, 98] are based on the infrastructure called *Beacon*. Beacons are usually installed at selected traffic signal lights and other strategic locations. They broadcast traffic information to the vehicles, such as the current location and the description of the surrounding road networks. As a result, these beacon-based navigation systems provide most recent real time route guidance to the drivers. Interesting thing to note is that Ali-Scout and Euro-Scout do not require the car be equipped with CD-ROMs maintaining road maps. In other words, vehicles receive all necessary map data from the Beacons.

The GPS-based system are GuideStar, Milemaster, Satellite Cruising System, and NV-1 [41, 95]. Milemaster [41] is based on the text-based routes databases while others are based on the digitized road maps. Unlike beacon-based systems, GPS-based systems do not provide the most up-to-date real time routing guidance to the driver.

Advance, Multi Vision, and Socrates are the typical examples of the navigation systems using GPS as well as dead reckoning with map matching [13, 108]. Compared with the other systems based on either GPS or dead reckoning with map matching alone, these system provide very accurate current position of the vehicles.

Two of the most popular providers of digital maps and software for map access are ETAK and NavTech[][]. ETAK is a provider of a map database and software for accessing the database. ETAK has been instrumental in a number of ITS projects such

as Pathfinder ATIS in the Los Angeles area; TravTek in greater Orlando, Florida; Minnesota Guidestar; TRiMS (Trip Reduction Information Management System); and others. ETAK actually uses two separate databases: one database for drawing maps and another network database with connectivity information for path calculations. The network database is organized in connectivity-based clusters to reduce access time. ETAK uses a modified Moore's algorithm for path finding.

NavTech developed their own data model which has some of the features of the relational data model to support a navigation system. The physical organization of database is based on the k-d trees which uses hierarchical notions of parcels, zones, and regions. Parcels are lowest logical entity and it defines a geographical area with a certain number of road segments (always less than some maximum) [39]. The stored regions represent physically overlapping geographic areas.

The NavTech database is compiled from a set of standard files on IBM mainframe computer, for a particular region. It contains sufficient detail, accuracy, and breadth to support the advanced transportation and navigation applications. The database contains navigation and digital cartography information partitioned into regions. The data includes *detailed city* and *inter-town* databases including traffic control data such as time dependent restriction on road accesses. The NavTech database consists of 7 database layers which are *customer specific, points of interest, cartography, geo-political, navigation, path, and geometry* layers [19]. NavTech allows the database to be compiled into different standards such as the European standard called GDF(Geographic Data File) [17].

Bibliography

[1] M. Abousalem and E. Krakiwsky, "A Quality Control Approach for GPS-Based Automatic Vehicle Location and Navigation Systems" *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)* pp. 466-470, 1993.

[2] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *IEEE Transactions on Software Engineering*, Vol. 14, No. 7, 1988.

[3] R. Agrawal, A. Borgida, and H. Jagadish, "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases", In *Proc. ACM-SIGMOD 1989 Intl. Conf. on Management of Data*, pp. 253-262, 1989.

[4] R. Agrawal, S. Dar, and H. Jagadish, "Direct Transitive Closure Algorithms: Design and Performance Evaluation", In *ACM Transactions on Database Systems*, Vol. 15, No.3, pp, 427-458, September 1990.

[5] R. Agrawal, and H. Jagadish, "Algorithms for Searching Massive Graphs", In *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No.2, pp. 225-238, April 1994.

[6] S. Azuma, K. Nishida, and S. Hori, "The Future of In-Vehicle Navigation Systems", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 537-542, 1994.

[7] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations", *On Knowledge Base Management Systems - Integrating Database and AI systems*, Spring Verlag, 1985.

[8] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", In *Proc. ACM-SIGMOD 1986 Intl. Conf. on Management of Data*, pp. 16-52, 1986.

[9] J. Bander and C. White, "A New Route Optimization Algorithm for Rapid Decision Support", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 709-727, 1991

[10] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R* -tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. ACM-SIGMOD 1990 Intl. Conf. on Management of Data*, pp. 322-331, 1990.

[11] B. Becker, H. Six, and P. Widmayer, "Spatial Priority Search: An Access Technique for Scaless Maps", *Proc. ACM-SIGMOD 1991 Intl. Conf. on Management of Data*, pp. 128-137, 1991.

[12] H. Blanken, A. Ubema, P. Meek, and B. van den Akker, "The Generalized Grid File: Description and Performance Aspects", *Proc. IEEE 6th Int'l Conf. on Data Engineering*, pp. 380-388, 1990.

[13] D. Boyce, A. Kirson, and J. Schofer, "Design and Implementation of ADVANCE: The Illinois Dynamic Navigation and Route Guidance Demonstration Program", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 415-426, 1991

[14] J. Buxton, S. Honey, W. Suchowerskyj, and A. Tempelhof, "The Travelpilot: A Second-Generation Automotive Navigation System", *IEEE Transactions on Vehicular Technology*, Vol. 40, No. 1, pp. 41-44, February 1991.

[15] I. Catling and B. McQueen, "Road Transport Informatics in Europe – Major Programs and Demonstrations", *IEEE Transactions on Vehicular Technology*, Vol. 40, No. 1, pp. 132-140, February 1991.

[16] S. Ceri, M. Negri, and G. Pelagatti, "Horizontal Partitioning in Database Design", In *Proc. ACM SIGMOD 1982 Intl. Conf. on Management of Data*, 1982.

[17] H. Claussen, W. Lichtner, L. Heres, P. Lahaije, and J. Siebold, "GDF, A Proposed Standard for Digital Road Maps to be Used in Car Navigation Systems", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 324-330, 1993.

[18] H. Claussen and R. GmbH, "Status and Directions of Digital Map Databases in Europe", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 25-28, 1993.

[19] *Navtech Company Document*, Navigation Technologies, 740 East Arques Avenue, Sunnyvale, CA USA 94086-3833

[20] T. Cormen, C. Leiserson, and R. Rivest, "The Set-Covering Problem", In *Introduction to Algorithms*, MIT Press, Cambridge, Mass. and McGraw-Hill, New York, 1990, Chapter 37.3, pp. 974-978.

[21] I. Cruz and T. Novell, "Aggregate Closure: An Extension of Transitive Closure", *Proc. IEEE 5th Int'l Conf. on Data Engineering*, 1989.

[22] S. Dar and H. Jagadish, "A Spanning Tree Transitive Closure Algorithm", *Proc. IEEE 8th Int'l Conf. on Data Engineering*, 1992.

[23] N. Deo and C. Pang, "Shortest-Path Algorithms: Taxonomy and Annotation", *Networks*, Vol. 14, pp. 275-323, 1984.

[24] W. Dijkstra, "A Note on Two Problems in Connection with Graphs", In *Numer. Math.*, Vol. 1, pp. 269-271, 1959

[25] S. Dreyfus, "An Appraisal of Some Shortest-Path Algorithms", *Operation Research*, Vol. 17, pp. 395-412, 1969

[26] J. Eder, "Extending SQL with General Transitive Closure and Extreme Value Selections", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 4, 1990.

[27] C. Feijoo, J. Ramos, and F. Perez, "A System for Fleet Management using Differential GPS and VHF Data Transmission Mobile Networks" *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 445-448, 1993.

[28] M. Freeston, "The BANG file: A New Kind of Grid File", *Proc. ACM-SIGMOD 1987 Int'l. Conf. on Management of Data*, pp. 260-269, 1987.

[29] M. Freeston, "A General Solution of the n-dimensional B-tree Problem", *Proc. ACM-SIGMOD 1995 Int'l. Conf. on Management of Data*, pp. 80-91, 1995.

[30] D. Galperin, "On the optimality of A*", In *Artificail Intelligence*, Vol. 8, No. 1, pp. 69-76, 1977.

[31] G. Gallo and S. Pallottino, "Shortest Path Methods: A Unifying Approach", *Math. Programming Study*, Vol. 26, pp. 25-36, 1984.

[32] S. Ganguly, A. Silberschatz, and A. Tsur, "A Framework for the Parallel Processing of Datalog Queries", In *Proc. ACM-SIGMOD 1990 Intl. Conf. on Management of Data*, pp. 143-152, 1990.

[33] M. Garey and D. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", *W.H. Freeman and Company, New York*, 1979, pp. 222.

[34] B. Golden and T. Magnanti, "Deterministic Network Optimization - A Bibliography", *Networks*, Vol. 7, pp. 149-183, 1977.

[35] D. Greene, "An Implementation and Performance Analysis of Spatial Data Access Methods", *Proc. IEEE 5th Int'l Conf. Data engineering*, pp. 606-615, 1989.

[36] O. Günther, "The Design of the Cell Tree: An Object-Oriented Index Structures for Geometric Databases", *Proc. IEEE 5th Int'l Conf. Data engineering*, pp. 598-605, 1989.

[37] O. Günther and H. Noltmeier, "Spatial Database Indices for Large Extended Objects", *Proc. IEEE 7th Int'l Conf. Data engineering*, pp. 520-526, 1991.

[38] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", *Proc. ACM-SIGMOD 1984 Intl. Conf. on Management of Data*, pp. 47-57, 1984.

[39] J. Guzolekm and E. Koch, "Real-Time Route Planning in Road Networks", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 165-169, 1989.

[40] J. Han, G. Quadah, and C. Chaou, "Processing and Evaluation of Transitive Closure Queries", *Proc. Int'l Conf. on Extending Database Technology*, 1988.

[41] S. Hoffman and C. Stewart, "Text-based Routing: An Affordable Way Ahead ?", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 45-48, 1993.

[42] M. Houtsma, P. Apers, and S. Ceri, "Distributed Transitive Closure Computations: The Disconnection Set Approach", In *Proc. of the 16th VLDB Conference*, pp. 335-346, 1990.

[43] M. Houtsma, F. Cacace, and S. Ceri, "Parallel Hierarchical Evaluation of Transitive Closure Queries", In *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, pp. 130-137, Dec. 1991.

[44] M. Houtsma, P. Apers, and G. Schipper, "Data Fragmentation for Parallel Transitive Closure Strategies", In *Proc. IEEE 9th Int'l Conf. Data Engineering*, pp. 447-456, 1993.

[45] Y. Huang and J. Cheiney, "Parallel Computation of Direct Transitive Closures", In *Proc. IEEE 7th Int'l Conf. Data Engineering*, pp. 192-199, 1991.

[46] K. Hua, J. Su, and C. Hua, "Efficient Evaluation of Traversal Recursive Queries Using Connectivity Index", In *Proc. IEEE 9th Int'l Conf. Data Engineering*, pp. 549-558, 1993.

[47] G. Hulin, "Parallel Processing of Recursive Queries in Distributed Architectures", In *Proc. of the 5th Intl Conf. on VLDB*, pp. 87-96, 1989.

[48] Y. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators", *Proc. of the 13rd VLDB Conference*, 1987.

[49] Y. Ioannidis and R. Ramakirishnan, "Efficient Transitive Closure Algorithms", In *Proc. of the 14th VLDB Conference*, pp. 382-394, 1988.

[50] Y. Ioannidis, R. Ramakirishnan, and L. Winger "Transitive Closure Algorithms Based on Graph Traversal", In *ACM Transactions on Database Systems*, Vol. 18, No.3, pp. 513-576, September 1993.

[51] K. Ishikawa, M. Ogawa, S. Azume, and T. Ito, "Map Navigation Software of the Electro Multivision of the '91 Toyota Soarer", *Int. Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, IEEE, (1991) 463-473.

[52] H. Jagadish, R. Agrawal, and L. Ness, "A Study of Transitive Closure As a Recursion Mechanism", In *Proc. ACM-SIGMOD 1987 Intl. Conf. on Management of Data*, pp. 331-344, 1987.

[53] H. Jagadish, "A Compression Technique to Materialize Transitive Closure", In *ACM Transactions on Database Systems*, Vol. 15, No.4, pp, 559-598, December 1990.

[54] H. Jagadish, "Spatial Search with Polyhedra", *Proc. IEEE 6th Int'l Conf. Data engineering*, pp. 311-319, 1990.

[55] H. Jagadish, "Linear Clustering of Objects with Multiple Attributes", *Proc. ACM-SIGMOD 1990 Intl. Conf. on Management of Data*, pp. 332-342, 1990.

[56] H. Jagadish, "On Indexing Line Segments", *Proc. ACM-VLDB 1990 Intl. Conf. on Very Large Data Bases*, pp. 614-625, 1990.

[57] B. Jiang, "A Suitable Algorithm for Computing Partial Transitive Closures in Databases", In *Proc. IEEE 6th Int'l Conf. Data engineering*, pp. 264-271, 1990.

[58] S. Jung and S. Pramanik, "An Efficient Representation of Distributed Fragments of Recursive Relations" In *Proceedings of the Fourth International Conference on Database Systems for Advanced Applications*, pp. 394-407, 1995.

[59] S. Jung and S. Pramanik, "HiTi Graph Model of Topographical Road Maps in Navigation Systems", In *Proc. IEEE 12th Int'l Conf. on Data Engineering*, 1996.

[60] S. Jung and S. Pramanik, "Empirical Analysis of Computing Shortest Path for Road Map Queries", submitted to *Information Processing Letters.*

[61] W. Kao, "Integration of GPS and Dead Reckoning Navigation Systems" *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, 1991, pp. 635-643

[62] C. Kolovson and M. Stonebraker, "Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data", *Proc. ACM-SIGMOD 1991 Int'l. Conf. on Management of Data*, pp. 138-147 1991.

[63] H. Kriegel and B. Seeger, "PLOP-Hashing: A Grid File without Directory", *Proc. IEEE 4th Int'l Conf. Data engineering*, pp. 369-375, 1988.

[64] R. Kung, E. Hanson, Y. Ioannnidis, T. Sellis, L. Shapiro, and M. Stonebraker, "Heuristic Search in Data Base System", In *Proc. 1st Int. Workshop Expert Database Systems*, pp. 96-107, Oct. 1984.

[65] Y. Kusumi, S. Nishio, and T. Hasegawa, "File Access Level Optimization Using Page Access Graph on Recursive Query Evaluation", *Proc. Int'l Conf. on Extending Database Technology*, 1988.

[66] G. Lachapelle, B. Townsend, H. Gehue, and M. Cannon, "GPS versus Loran-C for Vehicular Navigation in Urban and Mountainous Areas" *IEEE VNIS* pp. 456-459, 1993.

[67] T. Lezniak, R. Lewis, and R. McMillen. "A Dead Reckoning/Map Correlation System for Automatic Vehicle Tracking", *IEEE Transactions on Vehicular Technology*, Vol. 26, No. 1, pp. 47-60, February 1977.

[68] B. Liu, S. Choo, S. Lok, S. Leong, S. Lee, F. Poon, and H. Tan, "Intergrating Case-Based Reasoning, Knowledge-Based Approach and Dijkstra Algorithm for Route Finding", *Proc. Tenth Conf. Artificial Intelligence for Applications (CAIA '94)*, (1994) 149-155.

[69] D. Lomet and B. Salzberg, "A Robust Multi-Attribute Search Structure" *Proc. IEEE 5th Int'l Conf. Data engineering*, pp. 296-304, 1989.

[70] H. Lu, K. Mikikilineni, and J. Richardson, "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation", *Proc. IEEE 3rd Int'l Conf. on Data Engineering*, 1987.

[71] M. Mannino and L. Shapiro, "Extensions to Query Languages for Graph Traversal Problems", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 3, pp. 353-363, 1990.

[72] T. Mohr and C. Pasche, "A Parallel Shortest Path Algorithm", *Computing*, Vol 40, pp. 281-292, 1988.

[73] S. Moran and Y. Perl, "The Complexity of Identifying Redundant and Essential Elements", In *Journal of Algorithm*, Vol 2, pp. 22-30, 1981.

[74] W. Nejdl, S. Ceri, and G. Wiederhold, "Evaluating Recursive Queries in Distributed Databases", In *IEEE Transactions on Knowledge and Data Engineering*, Vol 5, pp. 104-121, February 1993.

[75] Y. Nakamura, S. Abe, Y. Ohsawa, and M. Sakauchi, "A Balanced Hierarchical Data Structure for Multidimensional Data with Highly Efficient Dynamic Characteristics", *IEEE Trans. on Knowledge and Data Engineering*, Vol 5, No. 4, pp. 682-693, August 1993.

[76] J. Nievergelt and H. Hinterberger, "The Grid File: An Adaptable, Symmetric Multikey File Structure", *ACM Transactions on Database Systems*, Vol 9, No. 1,pp. 38-71, March, 1984.

[77] T. Nicholson, "Finding the shortest route between two points in a network", *Computer Journal*, Vol. 9, (1966) 275-280.

[78] Y. Ohsawa and M. Sakauchi, "A New Tree Type Data Structure with Homogeneous Nodes Suitable for a Very Large Spatial Database", *Proc. IEEE 6th Int'l Conf. Data engineering*, pp. 296-303, 1990.

[79] D. Papadias, Y. Theodoridis, T. Sellis, and M. Egenhofer, "Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees", *Proc. ACM-SIGMOD 1995 Int'l. Conf. on Management of Data*, pp. 92-103, 1995.

[80] J. Pearl, In *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison Wesley, Reading, Mass., 1984.

[81] A. Pier, "Bibliography on Algorithms for Shortest Path, Shortest Spanning Tree, and Related Circuit Routing Problems (1956-1974)", *Networks*, Vol. 5, pp. 129-149, 1975.

[82] S. Pramanik, and S. Jung, "Description and Identification of Distributed Fragments of Recursive Relations", accepted for *IEEE Transactions on Knowledge and Data Engineering.*

[83] G. Qadah, L. Henschen, and J. Kim, "Efficient Algorithms for the Instantiated Transitive Closure Queries" In *IEEE Transactions on Software Engineering*, Vol 17, No. 3, pp. 296-309, March 1991.

[84] Ries and Epstein, "Evaluation of Distribution Criteria for Distributed Database Systems" In *U. C. Berkley, Tech. Report UCB/ERL M78/22*, May 1978.

[85] D. Roper and G. Endo, "Advanced Traffic Management in California", *IEEE Transactions on Vehicular Technology*, Vol. 40, No. 1, pp. 152-158, February 1991.

[86] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola, "Traversal Recursion: A practical approach to supporting recursive applications" In *Proc. IEEE 3rd Int'l Conf. Data Engineering*, pp. 580-590, 1987.

[87] N. Roussopoulos and D. Leifker, "Direct Spatial Search on Pictorial Databases Using Packed R-trees", *Proc. ACM-SIGMOD 1985 Int'l. Conf. on Management of Data*, pp. 17-31, May 1985.

[88] H. Samet, *Applications of spatial data structures : computer graphics, image processing, and GIS*, Addison-Wesley series in computer science, 1990

[89] J. Shapiro, J. Waxman, and D. Nir, "Level Graphs and Approximate Shortest Path Algorithms", In *Networks*, Vol. 22, pp. 691-717, 1992.

[90] S. Shekhar and T. Yang, "Motion in a Geographical Database System", *Proc. 2nd Symp. Design and Implementation of Large Spatial Databases*, 1991.

[91] S. Shekhar, A. Kohli, and M. Coyle, "Path Computation Algorithms for Advanced Traveller Information System (ATIS)", In *Proc. IEEE 9th Int'l Conf. Data engineering*, pp. 31-39, 1993.

[92] S. Shekhar and D. Liu, "Genesis and Advanced Traveler Information System (ATIS): Killer Application for Mobile Computing?", *NSF MOBIDATA Workshop on Mobile and Wireless Information System*, Nov. 1994.

[93] S. Shekhar and D. Liu, "CCAM: A Connectivity-Clustered Access Method for Aggregate Queries on Transportation Networks: A Summary of Results", *IEEE 11th Int'l Conf. Data engineering*, pp. 410-419, 1995.

[94] S. Shekhar and D. Liu, "A Similarity Graph Based Approach to Declustering Problems and its Application towards Parallelizing Grid Files", In *IEEE 11th Int'l Conf. on Data engineering*, 1995.

[95] M. Shibata and Y. Fujita, "Current Status and Future Plans for Digital Map Databases in Japan", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 29-33, 1993.

[96] H. Six and P. Wildmayer, "Spatial Searching in Geometric Databases", *Proc. IEEE 4th Int'l Conf. on Data Engineering*, pp. 496-503, 1988.

[97] H. Sodeikat, "EURO-SCOUT is Facing the German 1994 Market", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 551-556, 1994.

[98] R. Tomkewitsch, "Dynamic Route Guidance and Interactive Transport Management with ALI-SCOUT" *IEEE Transactions on Vehicular Technology*, Vol. 40, No. 1, pp. 45-50, February 1991.

[99] I. Toroslu and G. Qadah, "The Efficient Computation of Strong Partial Transitive-Closures", In *Proc. IEEE 9th Int'l Conf. Data Engineering*, pp. 530-537, 1993.

[100] H. Tsuji, H. Maeda, A. Shibata, and F. Morisue, "Evaluation of Location System Combining a GPS Receiver with Inertial Sensor", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, 1991, pp. 645-649

[101] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices", *Int'l Conf. on Expert Database Systems*, 1986.

[102] P. Valduriez and S. Khoshafian, "Parallel Evaluation of the Transitive Closure of a Database Relation", In *Intl. Journal of Parallel Programming*, pp 19-42, 1988.

[103] P. Valduriez and S. Khoshafian, "Transitive Closure of Transitively Closed Relations", *Int'l Con. on Expert Database Systems*, Benjamin Cumming, 1989.

[104] D. Vineyard, S. Jung, and S. Pramanik, "A Survey of Database Problems in IVHS", submitted to *IEEE Transactions on Knowledge and Data Engineering.*

[105] H. Wang and B. Zhang, "Route Planning and Navigation System for an Autonomous Land Vehicle", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 135-141, 1992.

[106] R. Weiland, "Standards for Navigable Databases: A Progress Report" *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 185-192, 1991.

[107] T. Yang, S. Shekhar, B. Hamidzadeh, and P. Hancock, "Path Planning and Evaluation in IVHS Databases", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 283-290, 1991

[108] F. Zijderhand and J. Biesterbos, "Functions and Applications of SOCRATES: A Dynamic In-Car Navigation System with Cellular-Radio Based Bi-directional Communication Facility", *IEEE Int'l Conf. on Vehicle Navigation and Information Systems(VNIS IVHS)*, pp. 543-546, 1994.