The Pennsylvania State University

The Graduate School

College of Engineering

PATH RECOMMENDATION FOR ROAD NETWORKS

A Thesis in

Computer Science and Engineering

by

Talal Ahmed Shahid

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2009

The thesis of Talal Ahmed Shahid was reviewed and approved* by the following:

Wang-Chien Lee
Associate Professor of Computer Science and Engineering
Thesis Advisor

Daniel Kifer
Assistant Professor of Computer Science and Engineering
Thesis Commitee Member

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer and Engineering

** Signatures are on file in the Graduate School

Abstract

Finding paths (that can be driving directions, flight itineraries, etc.) from a source to a destination upon a spatial network is a common but non-trivial activity in our daily life. In this thesis, we focus on finding driving directions on a road network. With the availability of high-precision digital maps, on-line path search becomes very convenient like in the case of MapQuest, GoogleMap and Yahoo!Map; while Garmin and TomTom navigation systems provide on-road directions. By specifying a source and a destination, a user can promptly retrieve an optimal path usually in terms of travel distance from the existing services and products. However, for many applications, one single path is inadequate due to various application needs, security and privacy concerns, and system performance. In this thesis, we study the design and implementation of a path recommendation system that provides multiple paths based on *k-mincost path search* and *skyline path search*. A k-mincost path search returns k *different paths* whose costs are the minimum among all possibile paths between a given source and destination. A Skyline path search returns a set of *non-dominated paths* when multiple types of costs such as travel time, mileage, road toll etc are considered simultaneously. Here, a path P is said to be dominated by another P' if P' is smaller than P for at least one type of cost and P' is not larger than P for all the other types of costs. However, both the k-mincost and the skyline path searches are not simple extensions of any existing shortest path search methods. To address the need of new solutions to these searches, we examine the challenges behind these searches and explore their properties, based on which, we propose efficient search algorithms, namely, *Passage Counting Algorithm* and *Progressive Skyline Search Algorithm*, for k-mincost path search and skyline path search, respectively. Both of our proposed algorithms significantly outperform existing related approaches demonstrated through our extensive simulations and our comprehensive analysis.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Dr. Wang-Chien Lee, my advisor and instructor for introducing to me to the exciting field of Road Networks, and for being receptive and enthusiastic to the idea of innovation and introducing me to all the right tools that allowed me to complete my research work.

I would also like to specially thank Ken C. K. Lee for his consistent guidance to me in planning and writing this thesis. Without the long countless discussions with him, this thesis would not have been a possibility.

I would like to thank Yuan Tian for helping me in shaping up and finalizing the crude ideas I had in my head into tangible concepts. She has been a great help in making things happen.

Last but definitely not the least, I would like to thank my parents and siblings. Whose consistent guidance, love, encouragement and support throughout my lifetime has been the sole factor behind each and every one of my achievements.

I dedicate this thesis to my parents.

1. Introduction

1.1 Background and Motivation

"Do you have experience in finding a path from one place to another?" "How did you finally select your path?" "How do you usually judge the suitability of a found path?" When faced with such questions, typically everyone has an answer "yes" to the first question. For the second question, some would say they like looking up a paper map; some others may say they look it up on popular websites like MapQuest, GoogleMap or Yahoo! Map etc (that we collectively call path service providers) and yet some others would say they find their routes on the GPS navigators installed in their cars. With the ever increasing popularity of the Internet and affordable price of the GPS navigation tools, we believe that most people use path search service providers and GPS navigators.

In reply to the third question above, we may, however, get a wide variety of answers. Typical path search service providers and GPS navigators always return one path that is optimal typically in terms of travel time or travel distance. This single path is then the only option was the users to either select or discard. In general we call this as *mincost path* hereafter. To a logistics company, it may be important to arrange a fleet of trucks in different mincost paths to avoid unexpected traffic that would delay the entire delivery or to avoid overloading a certain part of a path. To some people, privacy protection is the top requirement in using public services. Instead of single mincost paths, they may look for some alternative paths such that their trajectories become not easy to predict or track. To some others, "goodness" of a path could depend upon some other critirea like the number of traffic signals, bridges, scenic views and so on. As a result, existing path search services and GPS navigators are not sufficient. Morever, using just one criterion such as travel time might not be completely appropriate for some travel needs. For instance, the returned quickest path can cover a toll road that is undesirable to budget travelers. On the other hand, a cheapest path may take a very long time to drive. This reflects that path search based on a single type of costs is ineffective. In a nutshell, there

is a need to have a path recommendation system that provides multiple suitable paths. Among those suitable paths, users can pick one or some for their travels. This motivates the research study in this thesis.

1.2 Goal of This Research

In this research, we design and implement a path recommendation system, call *PathAdvice*, which accepts a source address, a destination address and at least one of the preferred costs and returns multiple mincost paths. In developing this system, we propose and investigate two classes of mincost paths, namely, they are k-mincost paths and skyline paths. With respect to one type of cost, say, mileage, k-mincost paths are k different paths that have the shortest distance among all possibility between a source and a destination on a network. When k is set to 1, a returned mincost path is exactly the one returned by conventional systems. On the other hand, skyline paths are a set of non-dominated paths for multiple types of costs under consideration at the same time. A path P is said to be dominated by another path P' if P' is smaller than P for one of the considered costs and P' is not larger than P for all the remaining of the costs. When only one criterion is consider, one path is expected to be returned and it is equivalent to one mincost path. Although they are generalized version of a mincost path, existing shortest path search algorithms cannot be straightforwardly extended to support these searches. In order to support these searches, the main challenges in realizing *PathAdvice*, we develop efficient algorithms to search k-mincost paths and skyline paths. Also, we conduct extensive experiments to evaluate the efficiency of the proposed algorithms.

1.3 Problem Formulation

With the goal to develop efficient search algorithms to find k-mincost path and skyline path, we first formalize these searches. Formally, a road network is modeled as a weighted directed graph, $G(N, E)$ in which N is a set of nodes and E is a set of edges. Each edge (n, n') connects a node n to another node n' and it is associated with d types of costs. We use $C_i(n, n')$ to denote the i-th type of cost for an edge (n, n') where $1 \leq i \leq d$. In this work, we consider all the edge costs are non-negative and no cycle with zero edges is formed in the graph. Given a node s and a node t that represents a source and a destination, respectively, a path $P(s, t)$ is a sequence of edges, i.e., $< (n_1 = s, n_2), (n_2, n_3), (n_{|P(s,t)|-1}, n_{|P(s,t)|} = t) >$. Here, $|P(s, t)|$ represents the number

of nodes involved in $P(s,t)$. Then, the i-th cost of $P(s,t)$, denoted by $C_i(P(s,t))$, is $\sum_{i=1}^{|P(s,t)|-1} C_i(n_i, n_{i+1})$. Certainly, between s and t, there would exist many possible paths. Here, we denote all of them by $\mathcal{P}(s,t)$. Based on the definition of G and $\mathcal{P}(s,t)$ for a source s and a destination t, we introduce k-mincost paths in Definition 1.3.1. It retrieves k paths whose costs are the smallest among all possible paths between s and t. Also Observation 1 states the generalization of k-mincost to the conventional shortest path.

Definition 1.3.1. k-mincost paths. *Given a weight directed graph G, a source node s, a destination node t, the i-th cost to be considered and a request number of returned path k. k-mincost paths denoted by $\mathcal{P}'(s,t)$, are a subset of paths $\mathcal{P}(s,t)$. Every path in $\mathcal{P}'(s,t)$ has a lower cost than the others in $\mathcal{P}(s,t)$ and the size of $\mathcal{P}'(s,t)$ is k.*

. Formally, $|\mathcal{P}'(s,t)| = k \wedge \forall P'(s,t) \in \mathcal{P}'(s,t), \notin P(s,t) \in (\mathcal{P}'(s,t) - \mathcal{P}'(s,t)) C_i(P'(s,t)) > C_i(P(s,t))$.

Observation 1. *1-mincost path is the conventional shortest path.*

Further, with the same setting, we first state dominance relationship between paths in Definition 1.3.2, based on which we define skyline paths as stated in Definition 1.3.3. In case that only one type of costs is considered, skyline paths contain only one path and it is the conventional shortest path as stated in Observation 2.

Definition 1.3.2. Path Dominance. *Given two paths $P(s,t)$ and $P'(s,t)$ for the same pair of a source s and a destination t, $P(s,t)$ is said to be dominated by $P'(s,t)$ (denoted by $P(s,t) \vdash P'(s,t)$) if for at least one type of costs, $P'(s,t)$ is smaller than $P(s,t)$ and for the rest of the costs, $P'(s,t)$ is not greater than $P(s,t)$. Formally, $P(s,t) \vdash P'(s,t) = \exists i C_i(P'(s,t)) < C_i(P(s,t)) \wedge \forall j C_j(P'(s,t)) \leq C_j(P(s,t))$.*

Definition 1.3.3. *Given a weight directed graph G, a source node s, a destination node t and multiple types of costs to be considered. Skyline paths, denoted by $\mathcal{S}(s,t)$, are a subset of paths not dominated by any other path in $\mathcal{P}(s,t)$. Formally, $\mathcal{S}(s,t) = \{P(s,t) | P(s,t) \in \mathcal{P}(s,t) \wedge \notin P'(s,t) \in \mathcal{P}(s,t) P(s,t) \vdash P'(s,t)\}$.*

Observation 2. *skyline path for only one type of cost is the conventional shortest path.*

From the definitions, to determine k-mincost paths and skyline paths from a source s to a destination t, it seems to determine $\mathcal{P}(s,t)$ in the first place and then to compare individual paths. It appears to be very expensive search for paths between s and t and thus becomes a challenging problem. As will be discussed in later chapters, we explore some related path properties such that not all possible paths are required to be examined. In other words, we selectively explore a search space for result paths. Finally, our approaches can quickly determine the result paths and provide an efficient search.

1.4 Overview of our Approach

We provide single criteria-multipath and multicriteria-multipath search algorithms. We utilize a road network dataset to lay the foundation of our graphical prototype to provide path recommendation features. We provide evaluation results using our prototype to report the performance of our proposals.

1.5 Contributions

In this research, our contributions are summarized below:

1. We surveyed the existing path search service and identified the demand of multiple resulted paths for a given pair of source and destination due to various application needs, security and privacy concerns.

2. We define k-mincost path and identify its challenges. Accordingly, we analyze the path properties and propose an efficient search algorithm called Passage Counting Algorithm.

3. We define path dominance relationship in presence of multiple selection criteria in path search. Then, we define skyline paths that is a set of non-dominated paths among all possible paths. Correspondingly, we explore the skyline properties and effective path search strategies to efficiently identify possible paths. We devise an efficient skyline path search algorithm called Progressive Skyline Search algorithm.

4. We prototype a path recommendation system *PathAdvice* to prove the concepts and to show the conventional shortest path, k-mincost path and skyline path search functionality.

5. We conduct an extensive evaluation on our proposed algorithms against some representative works. The results shows the superiority of our algorithms for a wide range of settings.

1.6 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 reviews various related works. Chapter 3 and Chapter 4 provide our proposed solutions to the k-mincost path search algorithm and the Skyline path search algorithm respectively. Chapter 5 discusses

implementation details about the developed prototype *PathAdvice*. Chapter 6 reports the evaluations results and Chapter 7 concludes this thesis and states our future work.

2. Background and Related Work

In this chapter, we will review the shortest path search problem and some of its well known solutions. Then, we will discuss the research work reported in the literature related to the k-mincost path search problem and skyline path path search problem that we are studying. Further, we discussed some other related work.

2.1 Shortest Path Search Problem

Finding a shortest path routed from a source s to a destination t is a typical single-source single-destination path search problem. The shortest path is a sequence of edges whose total distance (i.e., can be generalized as any kind of cost) is the shortest among all possible paths for the same pair of s and t. One of widely used approaches to determine the shortest path for a given network with no negative edge distance and no zero cycle is Dijkstra's algorithm [11]. Based on a shortest path property as stated in **Property 1**, Dijkstra's algorithm gradually explores a network such that the distance of paths from the source to all reached nodes are guaranteed to be the shortest. Logically, a node n should be visited first by any expansion with the shortest accumulated distance (i.e., the distance from the source s to n). Later, n may be visited by other expansions that should have longer accumulated distances, those expansions should be discarded. This network expansion continues until t is reached. The first path formed from s to t is thus the result shortest path.

Property 1. *Given a shortest path from s to t via m, a subpath (i.e., a portion of the shortest path) from s to m should have the smallest distance from s to m. Similarly, the subpath from m to t is also the shortest path from m to t.*

Proof. Assume there is the shortest path $P(s,t)$ from s to t via m and its distance is $D_{s,t}$. Besides, the distance of a subpath from s to m is $D_{s,m}$ and that from m to t is $D_{m,t}$. Thus $D_{s,t} = D_{s,m} + D_{m,t}$. If the subpath from s to m is the not shortest path between them that implies their shortest path distance is less than $D_{s,m}$, Thus the shortest path

distance from s to t should also $< D_{s,t}$. As a result, $P(s,t)$ is not the shortest path, that violates our assumption. ■ □

To realize the Dijkstra's algorithm, a priority queue is used to order all node visits in non-decreasing order of accumulated distance with respect to the source s. Because the traversal is decided based on currently the best candidates among all pending queue entries in the priority queue, this approach is also considered to be the best first traversal strategy. In addition, each node is associated with a boolean flag to indicate if the node is already visited or not. Then, any expansion to a node whose flag has been set to visited is discarded from further traversal. (Figure 2.1 outlines the pseudo code of th this algorithm.)

Algorithm Dijsktra (Graph G, a source node s, and a destination t)
Local: Priority Queue Q
Begin
1. $enqueue(Q, (s, \langle s \rangle, 0))$;
2. While (Q is not empty)
3. $(n, P, d) \leftarrow dequeue(Q)$; // where n is a node to visited, P is a path formed, d is the distance,
4. If (n is visited) *continue*; // skip this expansion
5. If ($n == t$) return P; // path is found (termination condition 1)
6. foreach neighbor of n, n' do
7. $enqueue(Q, n', P \cup < n' >, d + c(n, n'))$; // further expansion.
8. report no path is found; // no path is found (termination condition 2)
End.

Figure 2.1: Dijsktra's algorithm.

In brief, the algorithm starts with initializing a priority queue with the source node s together with an initial path (i.e., $\langle s \rangle$) and the distance (i.e., 0) (STEP 1). Then the search iteratively examines the head entry in the priority queue whose distance is guaranteed to be the shortest among all the entries in the queue, thus minimum network expansion (STEP 2-7). In case that a node n to be visited is already visited, the search skip that traversal (STEP 4). Otherwise, the traversal explores n's neighbors n' (that are assumed to be maintained as a adjacent list) (STEP 6-7). Here, the node to be

visited, the formed path and accumulated distance will be $n', P \cup \angle n' \rangle and\ d + c(n, n')$, respectively (LINE 7). The search terminates when either one of conditions is satisfied. The first condition is when the destination node t is reached (STEP 5). The second one is when the priority queue becomes empty that implies no possible paths formed from s to t (STEP 8). Based on the Dijsktra's algorithm, other improved versions such as A* and B* use additional heuristic distances to reduce the search space for paths. These heuristic distances are based on a strong assumption the path distances can be approximated by other observed metrics like Euclidean distance or certain path distances are pre-computed and maintained. However, for many cases, path costs (e.g., road rolls) cannot straightforwardly be derived and maintaining some path distances will incur high storage cost and maintenance cost in presence of network updates. On the other hand, Bellman-Ford algorithm [4] is suitable to graphs with negative edge costs. Since in road networks, edge costs would not be negative, we consider Dijsktra's algorithm in this study.

2.2 Multiple Shortest Path Search

The k path problem is a variant of the shortest path problem where k best paths between two nodes in the network are found. Related works for this problem include [40, 48, 31, 13, 24, 18, 17, 43, 31, 22, 33]. Broadly speaking, there are two divisions of the k-shortest path problem that have been addressed in previous works, namely, the constrained k-shortest paths problem and non-constrained k-shortest paths problem. For constrained one there are extra limitations are imposed on resulted paths e.g. disjointness, looplessness, with only positive or negative edges, etc. A disjointed set of paths has no common vertices between any two paths; loopless paths contain no cycle; paths with positive and negative edges are self-explanatory. On the other hand, for unconstrained version of the problem, there are no restrictions resulted paths as long as they are identical. In general, having constraints on path types makes the k shortest path search problem much more harder as compared with the unconstrained counterpart. [22] proved that simply picking the k-lowest cost paths out of all paths does not guarantee the loopless path constraint as the lowest k paths can contain loopless as well as paths with loops.

Further, algorithms for contrained and unconstrained problems can be divided into two groups namely, the sequential algorithms and the labelling algorithms.

Deviation algorithms have usually three stages. In the first stage, the shortest paths between the source and destination nodes are found by any shortest path search algorithm (e.g. Dijkstra's algorithm). In the second stage, other candidate paths are found by

introducing deviations in the shortest paths found in the first stage. In the third stage out of all the deviation paths produced between the two nodes, the best ones are are selected. The best ones are then selected as the shortest path of the first stage and the second and third stages are repeated. These stages are repeated in order until the k shortest paths are achieved. Two major sequential algorithms such algorithms are [48] and [13]. Yen's k loop-less shortest path ranking algorithm [48] poineers the topic and it is one algorithm that has not been challenged by any other work for a long time, although there have been attempts[31, 22]. A deviation of a path P with node set $\eta_P = \{s = n_1, n_2, n_3, ..., n_r = t\}$ can be described as a loopless sub path C that has two parts C_1 and C_2; C_1 having same sequence of node as the parent path, i.e., $\eta_C 1 = \{s = n_1, n_2, n_3, ..., n_z\}$, where $1 < z \leq r-1$, and C_2 is another shortest path between n_z and destination node t, having intermediate nodes that are not in the exact same order as the remaining nodes of the parent. C_1 and C_2 are connected at a deviation point n_z where $1 < z \leq r-1$. Starting from the node n_2, all nodes upto n_{r-1} are set as the deviation node sequentially and a set of the generated paths between n_1 and n_r is determined. For each deviation node n_z, all the nodes in the set $\{s = n_1, n_2, n_3, ..., n_z - 1\}$ and edge $e(n_z, n_z + 1)$ are removed from the network to ensure that the second part of the child path is different from the parent. From the generated paths set then the best path is selected and in turn all of its nodes are set as deviation nodes to generated more paths. Finally, after identifying the first k shortest cost paths, the algorithm finishes in $O(kn(m + nlog(n)))$ time for a network with 'm' edges and 'n' nodes. There are several problems with this approach. The algorithm lacks the ability to find the optimal paths(except for the first one) as the generated paths are effectively assumed to be limited the parent paths. Also, after all deviation child paths have been found for a parent path they are added to candidate path set and the alorithm then needs to search for the next parent path which proves to be costly if the deviation nodes are large in number or k is large. One possible improvement to Yen's algorithm in [31], deviation nodes are iterated back form the destination node to the source node instead of going from the source node to the destination node. This saves many changes made to the network while loopless condition is satisfied but the worst case complexity though better than two other implementations done in the same paper, is still the same as $O(kn(m + nlog(n)))$. Later on in [40], Yen extended his work to one-to-all and all-to-one k-shortest paths as well. Several researchers [31, 40, 22, 33] have tried to improve the performance of his algorithm. In the worst case, Yen's algorithm takes $O(kn^3)$ to calculate the paths out of which $O(n^2)$ is used for finding the first shortest path. In [13], Eppstein et al. has translated the problem from that of k-shortest path

between two nodes in the network, to single source k-shortest path problem where paths are may not be loopless. A single source shortest path tree for the target node is first found. Sidetracks through deviations are then produced by adding edges that are not a part of initial shortest path tree. To reduce overheads of storage, the Eppstein uses an instrinsic path representation. After execution in $O(m + nlog(n))$, the k paths are extracted one by one out of an efficient minimum heap, consisting of all shortest paths found. Using a heap requires $O(log\ i)$ time to extract the ith top path from the set which effectively reduces Yen's 3rd stage overhead. Fox in [14] has given another innovative technique with $O(m + knlogn)$ time complexity. It is easy to see that for very large values of k, deviation algorithms can be time consuming. In order to improve the time complexity of finding shortest paths for each node in Yen's agorithm [24] and [19] were proposed. In these algorithms, the graph was divided into partitions. Instead of having each node as a deviation point each partition boundary was taken as a deviation point. Both algorithms have the worst case time complexity of $O(k(m+nlogn))$ which is better than the worst case of Yen's algorithm. [6, 30, 39] have tried improving Yen's algorithm by using heuristic approaches, but the asymptotic worst case bound for finding k simple shortest paths in a directed graph remains unbeaten [22].

The second class of algorithms are the based on the *Labelling* technique. The labelling algorithms can be further divided into label setting and label correcting techniques. Label correcting algorithms progress by randomly selecting any un-visited node in the network, assigning it a label(usually distance) and then iteratively re-accessing the labels of any visited nodes for label updation. On the other hand, the Label setting algorithms select the best un-visited node in the network and then link the node to its parent node with the help of a pointer, forming a spanning tree structure that has the source node as the root. It is intuitive to see that while label correcting algorithms find the top-k paths after all the operations have finished and when no more label updates are needed, the label setting algorithms are able to find paths while the algorithm is still in execution and terminate when the first k paths have been found. This second approach is similar to Dijkstra's approach.

Labelling algorithms can extract results on-the-fly. Such algorithms can prove very useful in systems where results need to be provided to the user in real time. In [18], single source k-shortest path problem has been addressed by maintaining k path length vectors for all n nodes of the graph. The approach needs to maintain k lists of candidate nodes (candidate lists), one for each path. At each iteration, a node from the kth list is selected and the k distance vector is updated. If the node selected is the one with

the lowest distance value, the resulting algorithm is a label setting algorithm. In this case the entry and exit of a node from the list will occur only once. However, if the overhead of finding the minimum distance node is mimimized and any random node is selected, a label-correcting algorithm is resulted and in this case each node might enter and exit the candidate list many times. So in both cases(label correcting and label setting), there is a drawback whether a minimum node is selected or otherwise. The algorithm terminates when the candidate node lists are empty. Overall, an optimal solution is produced in a finite number of iterations, however, the space requirements of maintaining k copies of distance vectors and k candidate lists is large.In [17], the same idea has been improved but hardware support in the form of a shared memory processor has been utilized to perform asynchronous label correction. The idea is to remove multiple nodes from the candidate lists and update the the vector asynchronously in the memory. In [43] the algorithm computes an implicit representation of the k shortest paths to a given destination node from every node of a graph by exploiting the PRAM (Parallel Random and Access Machine), leading to the dependency of the algorithm on hardware. The brute force methods in [17] and [43] speed up the path search but are dependent on the underlying hardware, which might not be always available.

2.3 Skyline Path Search

In place of a single search criterion, skyline paths (also known as Pareto-optimal paths in some literatures) find a set of paths that are not dominated by any other paths when multiple criteria are considered at the same time. We first discuss the skyline query [5] that is related to skyline path search. Given a set of data points in a multidimensional space, a data point r is said to be dominated by another r' if r' is strictly better than r for at least one dimension and r' is not worse than r for all the rest of the dimensions. Intuitively we can determine the skyline by blindly comparing all possible pairs of data points to determine which data points are not dominated. However it is not time efficient. Recently a number of research works done towards answering this skyline query for a huge dataset maintained in a database efficiency. As classified in [27], the existing works can be classified into sorting-based, divide-and-conquer and hybrid approaches. Sorting-based approaches [9] use monotone scores derived from all the attributes to order the access of points in a way that data points are never dominated by those sorted behind them in the access order. In these sorting-based approaches, data points should be a part of skyline if they are not dominated by others started before it in the access

order. However, even though it reduces some of comparison, the search still needs to examine data points individually and it does not have pruning capability. Divide-and-conquer approaches [5] recursively divides a huge datasets into smaller datasets that can fit into main memory. Then the local skyline is derived for each small dataset. Later, the (global) skyline is determined based on all those local skylines. The performance of these approaches is really depending on the partitioning strategies. Finally hybrid approaches [25, 37, 27] organizes a datasets as clusters and access data points and clusters according to a monotone score (e.g. the distances to the origin of the search space). Whenever a cluster is determined to be dominated by some data points, all enclosed data points in the clusters are waived from detailed examination, thus improving the search performance.

Although skyline path search and skyline query are developed based on the idea of dominance, skyline query cannot be directly used to answer a skyline path search. Skyline query assumes that all data points are available and indexed. However, in a network, there could be a huge number of possible paths between a given source and destionation. Logically, it is inefficient to determine all paths in prior and then to apply skyline query to find non-dominated paths. For a network with no negative edge costs, the path costs will only increases along the path from the source. Thus, if a subpath $SP(s, m)$ from s to m is already dominated by another subpath $SP(s, m)$ for the same pair of nodes, the final path extended from $SP(s, m)$ should also be dominated. As a result, any extension for $SP(s, m)$ can be safely discarded. Observing this idea, MOPA [20] was proposed. First, the authors used the accumulated monotonic score of nodes to be visited to arrange access the visit order of the nodes. In general, it adopts the best first strategy. Second, for each node n, it remarks a history of non-dominated visits associated with the multiple costs. Whenever a new visit reaches the node, it is checked against the history. If the visit is not dominated, the traversal continues to n's neighbors. Otherwise, it is terminated at n. Nevertheless, MOPA cannot avoid multiple visits to the same nodes due to a potentially huge number of non-dominated subpaths. Recall that clustering of candidates as exploited by hybrid skyline search algorithms can improve the search performance. In this work, we devise a novel skyline path search algorithm based on this idea. We shall present it in the Chapter 4.

2.4 Other Related Work

For any dynamic road network in which edge costs would change from time to time, shortest path searchs will become schedule problem like finding a route departed from a source within a time window e.g. between 7am to 9am and arrive the destination and this route is expected to have the shortest travel time. Existing works such as [23] address this issue based on the predicted traveling time for each edge in the CapCod Network. On the other hand, the shortest path would become invalid in presence of edge costs and thus efficient shortest path update becomes an interesting research problem. Existing works such as [28] proposes an impact scope associated with a path to detect the affecting edges to the corresponding path and to recompute the affected path. While our study focuses on the static network, we shall extend our path recommendation system for the dynamic network as our future work.

In later sections we explain how we used [23] to formulate time estimation in the protoyped system *PathAdvice*.

3. K-mincost Path Search

In this chapter, we identify some important properties of k-mincost path that allows us to develop an efficient k-mincost path search algorithm.

3.1 Problem Analysis

Figure 3.1 shows an example in which 3-mincost paths are found from a source s to a destination t. They are $\{s, a, b, c, d, e, t\}$, $\{s, a, b, c, d, f, t\}$ and $\{s, a, b, c, g, f, t\}$ (in order of their path distances). As we can see, edges (s, a), (a, b) and (b, c) are common to all the resulted three mincost paths and edge (c, d) is common to two of the resulted path. In the example, we can observe that any edge in a graph for k-mincost path search can be at most included by k separate mincost paths. Further, all traversals that include (c, d) must traverse node c before d since c has a shorter path distance to s than d. This implies two important findings: (1), each node should be first visited by the k-mincost subpaths from s if the best-first search strategy is utilized; and (2) the rest of visits at a node that should have their subpath distances greater than the first k visits can be pruned.

3.2 Passage Counting Algorithm

Based on our analysis, we present a novel search algorithm called *Passage Counting Algorithm*. The basic idea of this algorithm follows the principle of best-first graph

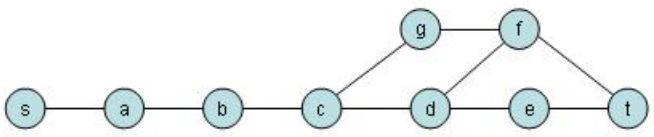

Figure 3.1: Sample graph

traversal in which the nodes are accessed according to the distance from the source s and each node n is associated with a visiting counter, v_n, initialized to 0. Every time when node n is visited, its visitor v_n is incremented by one. In case a subpath reaches node n whose v_n equals k, the extension of that subpath will then be ceased at n. The algorithm is outlined in Figure 3.2.

Algorithm PassageCountingAlgorithm(Graph a, Source node s, Destination t,Requested Number of Paths k)
Local PriorityQueue Q; a visited count $v_n = 0$ (forall n)
Output $\mathcal{P}$
Begin
1. $enqueue(Q, (s, \{\}, 0))$;
2. While (Q is not empty)
3. $(n, P, d) \leftarrow dequeue(Q)$;
4. $if(v_n = k)$ continue; // a node has already visited by k or more subpaths
5. $if(n \in P) continue$; // a node is revisited by the same subpath (no cycle path).
6. $if(n = t)$ // the node to be visited is the destination
7. $if((P \cup n) \in \mathcal{P}) continue$; // no duplicate result path
8. $\mathcal{P} \leftarrow \mathcal{P} \cup (P \cup n)$; // collect the result path
9. $if(|\mathcal{P}| == k) terminate$;// terminate when k result paths are found
10. $v_n = v_n + 1$;
11. foreach neighbor n' of n do
12. $enqueue(Q, (n', P \cup n, d + c(n, n')))$;
13. report less than k result paths found
End

Figure 3.2: Passage Counting Algorithm.

Intuitively, the Passage Counting Algorithm appears very similar to Dijkstra's algorithm as both of them use best-first traversal strategy. In fact some of the detailed operations are very different. The first and the most important is the use of v_n to guard unnecessary traversals (in Line 4) and its manipulation (in Line 10). Second, due to allowing multiple visits to nodes, it is possible for some subpaths to revisit nodes that they have already visited before. In other words, a cyclic subpath is formed. To tackle this case, we perform a check against repeated node accesses (as in Line 5). Last but not

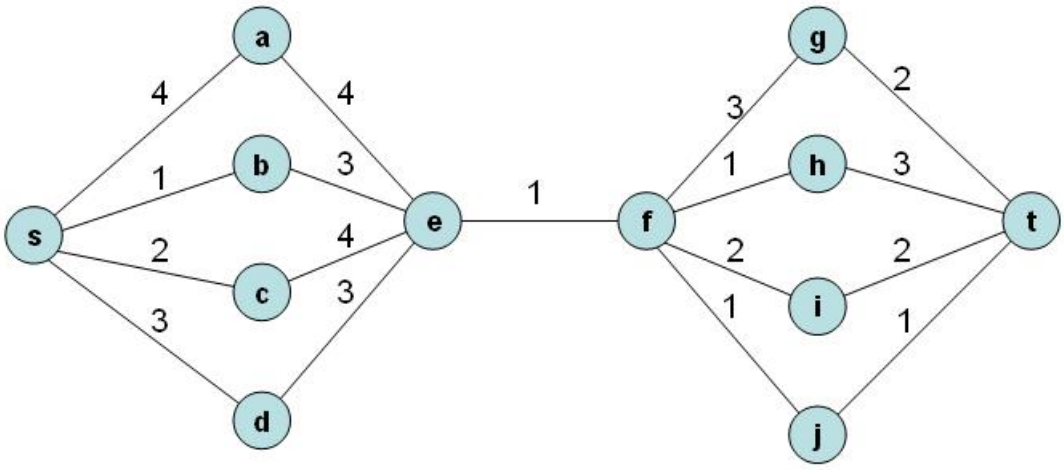

Figure 3.3: Example for k-mincost path search

the least, we incorporate all found qualified paths that reach the destination t (Line 6-9). Since a path found may be identical to one in those found paths in P, we perform check a path against individuals in P (Line 7). If it is not a duplicate to any existing path, it is incorporated to P (Line 8). Finally, the search ends when k resulted paths are found (Line 9) or all possible paths are examined indicated by an empty queue (Line 13)

3.3 Running Example

Suppose that we take the Figure 3.3 as a scenario to illustrate the Passage Counting algorithm where the top 3 paths between nodes s and t, are needed i.e. $k = 3$. Initially the counters associated to all nodes are set to 0 and the zero-length path at source s will be placed in the priority Q.

The step by step execution trace of the Passage Counting algorithm in shown in Table 3.1. At each step the updated counters for each node from a to j and t are shown. For each iteration the mincost sub path P with its cost is shown in the 'Lowest cost P' column. All one edge extensions of P are shown in the 'Extensions' column where C (X) is the accumulated cost of P extended by one edge to node X. The finalized path contains the finalized list of top 3 paths between s and t. In the table it can be seen that all counters v_n are initialized to zero. The nodes adjacent to node s are then used to extend the zero-cost path by one edge. Each extensions is then enqueued into the priority based on the accumulated cost function C .

We can see that at each iteration, the mincost path is dequeued from the priority queue. In iteration 2, this path is {s,b}, having a cost of 1. This path is then extended

It.	a	b	c	d	e	f	g	h	i	j	t	Lowest cost P	Extensions	Finalized Paths
1	0	0	0	0	0	0	0	0	0	0	0	{s},C=0	$\tilde{C}$(a)=4,$\tilde{C}$(b)=1,$\tilde{C}$(c)=2,$\tilde{C}$(d)=3	
2	0	1	0	0	0	0	0	0	0	0	0	{s,b},C=1	$\tilde{C}$(e)=4	
3	0	1	1	0	0	0	0	0	0	0	0	{s,c},C=2	$\tilde{C}$(e)=6	
4	0	1	1	1	0	0	0	0	0	0	0	{s,d},C=3	$\tilde{C}$(e)=6	
5	1	1	1	1	0	0	0	0	0	0	0	{s,a},C=4	$\tilde{C}$(e)=8	
6	1	1	1	1	1	0	0	0	0	0	0	{s,b,e},C=4	$\tilde{C}$(f)=5	
7	1	1	1	1	1	1	0	0	0	0	0	{s,b,e,f},C=5	$\tilde{C}$(g)=8,$\tilde{C}$(h)=6,$\tilde{C}$(i)=7,$\tilde{C}$(j)=6	
8	1	1	1	1	2	1	0	0	0	0	0	{s,c,e},C=6	$\tilde{C}$(f)=7	
9	1	1	1	1	3	1	0	0	0	0	0	{s,d,e},C=6	$\tilde{C}$(f)=7	
10	1	1	1	1	3	1	0	1	0	0	0	{s,b,e,f,h},C=6	$\tilde{C}$(t)=9	
11	1	1	1	1	3	1	0	1	0	1	0	{s,b,e,f,j},C=6	$\tilde{C}$(t)=7	
12	1	1	1	1	3	1	0	1	1	1	0	{s,b,e,f,i},C=7	$\tilde{C}$(t)=8	
13	1	1	1	1	3	2	0	1	1	1	0	{s,c,e,f},C=7	$\tilde{C}$(g)=10,$\tilde{C}$(h)=8,$\tilde{C}$(i)=9,$\tilde{C}$(j)=8	
14	1	1	1	1	3	3	0	1	1	1	0	({s,d,e,f},C=7	$\tilde{C}$(g)=10,$\tilde{C}$(h)=8,$\tilde{C}$(i)=9,$\tilde{C}$(j)=8	
15	1	1	1	1	3	3	0	1	1	1	1	{s,b,e,f,j,t},C=7	Not explored further	{s,b,e,f,j,t}
16	1	1	1	1	3	3	0	1	1	1	1	{s,a,e},C=8	Pruned:Node e Expired	{s,b,e,f,j,t}
17	1	1	1	1	3	3	0	1	1	1	2	{s,b,e,f,i,t},C=8	Not explored further	{s,b,e,f,j,t},{s,b,e,f,i,t}
18	1	1	1	1	3	3	0	1	1	1	3	{s,b,e,f,h,t},C=9	Not explored further	{s,b,e,f,j,t},{s,b,e,f,i,t},{s,b,e,f,h,t}

Table 3.1: Example run for Passage Count Algorithm

by one edge to adjacent nodes of node b, in this case which is the edge (b, e) to adjacent node e, resulting in a path {s,b,e}. We know that in classical Dijkstra's algorithm the best cost node is marked as visited as soon as it is accessed through any path. After being marked as a visited node, the node is not visited again since it has already been accessed with the lowest possible cost. In our algorithm, since at the time of dequeue the path {s,b} is the path with lowest cost in the queue, we only increment the counter of node b node by one. Unlike Dijkstra's algorithm, a visit in our algorithm is only a 'partial' visit. Each increment in the counter means that the node has now been officially (partially) visited one more time. As the algorithm marks consequent partial visits to nodes utilizing increasing lowest cost order, each following increment has a higher cost for the visiting path.

In iteration 15, a mincost path reaching the destination node t is dequeued. The v_n value of node t, is incremented. Since this is the destination node, no further edges for this path are explored. The algorithm continues. At this point, the next lowest cost path at the top of the queue is $\{s, a, e\}$. After the path is dequeued, node e is examined for the number v_n of partial visits already done and is found to be equal to k. This v_n value being equal to k, thus, extensions of node e are hence not produced. The algorithm simply ignores the {s,a,e} path and the next lowest cost path is then dequeued and the algorithm continues.

In each of the iterations 17 and 18, the mincost paths dequeued end at node t. For each of the dequeued path only the v_t value is incremented and no further edges are explored.The termination condition of the algorithm is when the counter value v_t, is found to be equal to k indicating that the best k, have already reached the final destination.

At this point since v_t is equal to 3, the algorithm will be terminated.

3.4 Discussion

In general in the Passage Count algorithm, a basic extension of Dijkstra's algorithm has been done. Whereas the original algorithm finds only one path to the destination node, our extended version of the algorithm is able to find multiple paths with a 'partial visit' concept. Whereas Dijkstra's algorithm marks next best visited node as 'visited' and no more visits are allowed to that node, our algorithm only marks a partial visit by incrementing the counter by one for each new visit until the counter has reached its maximum possible value. After the counter of a node has expired all paths originated from this node are ignored. In the absence of counter limits placed on the v_n value of a node, the algorithm will spend time doing un-needed path creation, enqueues and dequeues. The limit on the value v_n prunes the paths that will not provide a path that is one of the top k mincost paths.

We described in Chapter 2 the *label setting* and *label correcting* algorithms. By those explanations we can see that Passage Counting Algorithm by nature is a label setting algorithm. Unlike label correcting algorithms that choose any unvisited node instead of the best unvisited node in the unvisited set, our label setting algorithm chooses the next best unvisited or partially unvisited node for inclusion into a path. This comes with the overhead of maintaining a priority queue of nodes. On the other hand because label correcting algorithms randomly choose the next univisited node to correct its label and only requires a simple queue, it can be argued that label correcting algorithms provide a more efficient solution. However, for finding top-k paths label correction for intermediate and destination nodes will be needed an indefinite number of times before the algorithm terminates when finally the queue is empty. However, in our label setting algorithm after a discrete number of visits, the algorithm is able to terminate. Hence providing a limited number of processing for a node ($\leq k$ times).

Passage Counting Algorithm is in more stark contrast with the best known Yen's sequential algorithm. Yen's approach first finds all possible sets of paths, and then repeatedly picks the best one from each set. In Passage Counting algorithm, the best results are available immediately after they path reach the node t, which in the Yen's case is not possible since the top first path can not be decided until all paths have been found and the top k, amongst them are chosen. Yen's algorithm and other sequential algorithms rely on choosing the best paths from a set of cadidate paths. Therefore, in cases where

the k is much smaller than size of the candidate paths set many false hits occur. However, because of pruning techniques to avoid un-needed operations, all top k paths are returned by Passage Count algorithm in a much quicker time than Yen's algorithm. As compared to Yen's time of $O(kn(m+nlog(n)$, Passage Counting algorithm has a complexity equal to $O(knlog(n))$ which is broken down into k times the underlying Dijkstra's algorithm time. Although, both algorithms are linear growth algorithms, this is a significant improvement over Yen's approach. Morever our algorithm provides results that are optimal paths which is not the case in Yen's algorithm because of Yen's premature termination immediately after the first k paths are found. These paths are not neccessarily the optimal k paths and if the termination condition of Yen's algorithm is relaxed paths found after the first k paths might have have a smaller cost. Evaluations in Chapter 6 demonstrate these results.

It is to be noted that Passage Counting algorithm is applicable for the case where loopless unique paths are you be found. For paths with loops, this algorithm will not be able to provide the correct results. This type of setting is applicable and very relevant to the road networks scenarios.

4. Skyline Path search

In this section, we discuss another path recommendation criterion, i.e. the *skyline paths*, which represent the paths that are at least not worse than any other path connecting the same source and destination when multiple types of costs are considered at the same time. The concept of skyline query has been studied extensively [9, 36, 27] but to the best of our knowledge, this is the first study that applies the skyline semantics on path search. Below, we will first analyse the properties of skyline paths, then based on these properties, we propose a novel algorithm that computes skyline paths for a give source-destination pair efficiently.

4.1 Problem Analysis

Recall that in Section 1.3, we have already present the concept of dominance and skyline paths. Assume each path is associated with k types of costs $C_1, \cdots, C_j$ $(1 \leq j \leq k)$ a path $P(s,t)$ dominates a path $P'(s,t)$ (s,t are the source and destination of both P and P') iff $\exists 1 \leq j \leq k$, $C_j(P(s,t)) < C_j(P'(s,t))$ and $\forall 1 \leq j \leq k$, $C_j(P(s,t)) \leq C_j(P'(s,t))$, denoted as $P(s,t) \vdash P'(s,t)$. If $P(s,t)$ is a skyline path then we have $\forall P'(s,t) \in \mathcal{P}(s,t)$, $P'(s,t) \nvdash P(s,t)$.

Once the source node s and the destination t is specified, a naive way of finding all skyline paths is to first find out all the paths from s to t, and then apply conventional skyline algorithms (e.g. SFS [9], BBS [36], ZSKY [27], etc.) to identify non-dominated paths by treating each path as an individual data point in the k-dimensional space, where each dimension represents a type of road costs. However, this approach is very inefficient, because computing all the paths from s to t itself is extremely expensive. Also, this brute-force method redundantly computes many paths that are clearly dominated by others. In fact, if we can utilize some properties of the skyline paths, such exhaustive computation can be avoided without affecting the correctness of results. Hence, in order to derive an efficient skyline path search algorithm, let us first look at the properties that the skyline paths have.

Since $C_j(P(s,t)) = \sum_{(n,n')\in P(s,t)} C_j(n,n')$, and $\forall(n,n') \in E$, $C_j(n,n') \geq 0$, we can observe the monotonic cost property as follows.

Property 2. Monotonic cost. *$\forall P'(u,v) \subseteq P(s,t)$ (we say that $P'(u,v)$ is a subpath of $P(s,t)$), $C_j(P'(u,v)) \leq C_j(P(s,t))$.*

Proof. $\because C_j(P(s,t)) - C_j(P'(u,v)) = \sum_{(n,n')\in P(s,t)\wedge(n,n')\notin P'(u,v)} C_j(n,n') \geq 0,$
$\therefore C_j(P(s,t)) \geq C_j(P'(u,v))$. ■

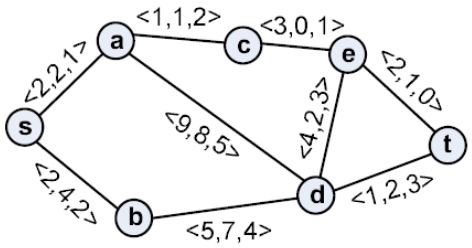

Figure 4.1: Illustration of skyline path properties

As Figure 4.1 shows, in the simple graph, the path $P_1(s,t) = \langle s,b,d,t\rangle$ has a total cost of $\langle 8,13,9\rangle$. The cost of its subpaths are $C(P_1'(s,b)) = \langle 2,4,2\rangle$ and $C(P_1''(s,d) = \langle 7,11,6\rangle$, which are both smaller than $C(P_1(s,t))$.

According to the definition of skyline paths, the dominance relationship can only exist between two paths that share the same source and destination. For two paths that are not connecting the same source and destination, we say that they are *incomparable*. This is an additional condition of the incomparability definition in conventional skyline problems.

Property 3. Incomparability. *Given two paths $P(s,t)$ and $P'(u,v)$, they are incomparable if 1) $u \neq s$ or $v \neq t$, or 2) $u = s$, $v = t$ and $P(s,t) \nvdash P'(u,v) \wedge P'(u,v) \nvdash P(s,t)$.*

For example, two paths $P_1(s,a)$ and $P_2(s,b)$, as shown in Figure 4.1, are incomparable because $a \neq b$. Meanwhile, even if $P_3(s,d) = \langle s,a,d\rangle$ and $P_4(s,d) = \langle s,b,d\rangle$ share the same source s and the same destination d, they are incomparable because neither can $P_3(s,d)$ dominate $P_4(s,d)$, nor the opposite ($C(P_3(s,d)) = \langle 11,10,6\rangle$, $C(P_4(s,d)) = \langle 7,11,6\rangle$).

On the other hand, if two paths are comparable (i.e., one path dominates another), we can also observe the fact that as both paths are extended further toward the destination following exactly the same way, the newly extended paths will keep the dominance relationship. This is called *transitivity* of dominance.

Property 4. Transitivity. *Let $P_1(u,v)$ and $P_2(u,v)$ be two different paths from u to v ($P_1(u,v)$ and $P_2(u,v)$ have at least one edge not in common). Let $P'_1(v,w)$ and $P'_2(v,w)$ be the extensions of $P_1(u,v)$ and $P_2(u,v)$, respectively, such that two extended paths, $P''_1(u,w)$ and $P''_2(u,w)$ are formed as $P''_i(u,w) = P_i(u,v) + P'_i(v,w)$ ($i = 1, 2$, "+" means to concatenate two sequent paths). If $P_1(u,v) \vdash P_2(u,v)$ and $P'_1(v,w) \equiv P'_2(v,w)$, then we have $P''_1(u,w) \vdash P''_2(u,w)$.*

Proof. Since $P''_i(u,w)$ is the concatenation of $P_i(u,v)$ and $P'_i(v,w)$, we have $C_j(P''_i(u,w)) = C_j(P_i(u,v)) + C_j(P'_i(v,w))$ ($i = 1, 2$, $1 \le j \le k$). $P'_1(v,w) \equiv P'_2(v,w) \Rightarrow C_j(P'_1(v,w)) = C_j(P'_2(v,w))$ for all $1 \le j \le k$. Because $P_1(u,v) \vdash P_2(u,v)$, from the definition of dominance, we know that there is a j_0, $1 \le j_0 \le k$, $C_{j_0}(P_1(u,v)) < C_{j_0}(P_2(u,v))$ and for any other $1 \le j \le k$, $C_j(P_1(u,v)) \le C_j(P_2(u,v))$. So,

$$\begin{aligned} C_{j_0}(P''_1(u,w)) &= C_{j_0}(P_1(u,v)) + C_{j_0}(P'_1(v,w)) \\ &< C_{j_0}(P_2(u,v)) + C_{j_0}(P'_2(v,w)) \\ &= C_{j_0}(P''_2(u,w)) \end{aligned}$$

Similarly, for any other $1 \le j \le k$, we can easily prove that $C_j(P''_1(u,w)) \le C_j(P''_2(u,w))$. Thus, by definition we have $P''_1(u,w) \vdash P''_2(u,w)$. ■

Again, we illustrate this property in Figure 4.1. Consider two paths $P_1(s,e) = \langle s, a, c, e$ and $P_2(s,e) = \langle s, b, d, e\rangle$, we have $C(P_1(s,e)) = \langle 6, 3, 4\rangle$ and $C(P_2(s,e)) = \langle 11, 13, 9\rangle$, so $P_1(s,e) \vdash P_2(s,e)$. Then, let $P'_1(e,t) \equiv P'_2(e,t) = \langle e, t\rangle$, $P_1(s,e)$ and $P_2(s,e)$ are extended to $P''_1(s,t) = \langle s, a, c, e, t\rangle$ and $P''_2(s,t) = \langle s, b, d, e, t\rangle$. Because $C(P''_1(s,t)) = \langle 8, 4, 4\rangle$, $C(P''_2(s,t)) = \langle 13, 14, 9\rangle$, we still have $P''_1(s,t) \vdash P''_2(s,t)$, the dominance relationship is unchanged with the same path extension. So as early as we have computed $P_1(s,e)$ and $P_2(s,e)$, we can already determine that it is impossible for $P''_2(s,t)$ to be a skyline path without explicitly go through the path and calculate its total cost.

Based on the above properties, we can safely eliminate a large number of paths before reaching the destination node, as long as these paths are dominated by any other paths during the network traversal process. Our algorithm is derived from these important properties, and we elaborate the details in the next subsection.

Briefly, our proposal for Skyline path search extends the conventional skyline search problem and hence allows us to use the same properties in devising our algorithm. We note that during the course of path search, a node can be visited by different paths, each having a different cost vector. For each node, if the cost vector of every visiting path is considered as a multidimensional data point and stored, we can achieve a dataset similar

Notation	Explanation
$C_i(n, n_0)$	ith type of cost for an edge (n, n_0)
$C(n, n_0)$	Monotonic score for an edge, calculated by adding components $\sum_{i=1}^{j} C(P(n, n_0))$ where j is the number of total component costs
$C_i(P(r, s))$	ith component in the cost vector of path $P(r, s)$
$\|P(r, s)\|$	the number of nodes involved in path $P(r, s)$ from node r to node s.
$\mathcal{L}(n)$	Labels set stored for node n
$\|L(n)\|$	number of Labels in set $\mathcal{L}(n)$
$L_i(n, P(r, s))$	ith label in set $L(n)$, where $0 < i <= \|L(n)\|$. It the accumulated cost vector of $P(r, s)$ and edge (s, n) calculated by adding their corresponding cost components $\{C_1(P(r, n)) + C_1(s, n), C_2(P(r, n)) + C_2(s, n), ..., C_k(P(r, n)) + C_k(s, n)\}$ where k is the number of total component costs and n is the adjacent node of r
$C_c(P(s, n))$	Is the set of accumulated cost components of path P $\{\sum_{i=1}^{m} C_1(P(s, n)), \sum_{i=1}^{m} C_2(P(s, n)), ..., \sum_{i=1}^{m} C_k(P(s, n))\}$ where $m = \|P(s, n)\|$, k is the number of component costs, n is head node of the path P and n is connected in P through m

Table 4.1: Notations and Descriptions

to conventional Skyline problem datasets for that node. In our approach, we utilize the conventional Skyline point properties like *dominance*, *incomparibility* and *transitivity* to identify the Skyline cost vectors for a node from amongst all the cost vectors from other visiting paths. We then limit the further exploration of the graph to the only those paths that have Skyline cost vectors. All other dominated paths are not explored since in a network with static non-decreasing edge cost functions, these paths will never be able to provide lower costs than their Skyline counterparts. It is however important to mention here that Skyline points are maintained w.r.t. to a particular node only as cost vectors on different nodes are *incomparable*.

4.2 Skyline Path Search Algorithm

In the table 4.1 we can see some notations that will be used throughout this chapter.

The basic idea of the Skyline path algorithm is similar to Dijkstra's algorithm which is to visit the nodes in a best first cost order. We use the combined monotonic score $C(P(s, n))$ as our cost function, where s is the source node and n is any node besides t, the final destination node . (See corallary below)

Definition 4.2.1. Monotonic Score *Represented by $C(P(r, s))$, the Monotonic Score of path P is calculated by adding all cost components$\sum_{j=1}^{n} \sum_{i=1}^{k} C_j(P(r, s))$ where k is the number of total component costs and n is the number of edges in the path.*

Observation 3. Monotonic Score of dominated points *A dominated path has a higher monotic cost than its dominating path*

This means that in a set of paths ordered in ascending order of monotonic costs, the dominating paths come before before the dominated paths. We use this important property also taken from conventional skyline problem and put it to our use.

Apart from the greedy approach of Dijkstra's algorithm there are several other ideas that we propose in our algorithm in order to map the classical algorithm to Skyline paths problem.

Definition 4.2.2. Path Label *A Path Label is the accumulated cost vector represented by* $\langle C_i, C_i + 1, ..., C_k \rangle$ *of* $P(r, s)$ *and edge* (s, n) *calculated by adding their corresponding cost components* $\{C_1(P(r, n)) + C_1(s, n), C_2(P(r, n)) + C_2(s, n), ..., C_k(P(r, n)) + C_k(s, n)\}$ *where* k *is the number of total component costs and* n *is the adjacent node of last node* r *of a path with source node as* s*.*

Each node n in the network maintains a set of non-dominant labels $\mathcal{L}(n)$. In other words, this set represents cost vectors of all skyline paths that reached the node n at one time or the other. The ith label $L_i(n, P(r, q))$ where $i = |\mathcal{L}(n)| + 1$ will be added to the set only if the cost vector of $P(r, q)$, is non-dominated or incomparable to all existing labels in $\mathcal{L}(n)$. After dequeueing the least monotonic score path $P(s, n)$ from the priority queue its label l_P is compared with the label set of its head node n. There are two possibilities for an extension's label analysis. l_P can either be a:

1. Dominated label: A label that is dominated by atleast one label in the node label set $\mathcal{L}(n)$.

2. Incomparable label: A label that is better for some components and worse against for labels in node label set $\mathcal{L}(n)$.

Based on these cases the algorithm decides whether to consider l_P for addition into the existing set of labels in $\mathcal{L}(n)$ or to just ignore it. If l_P is dominated by any one of the labels in set $\mathcal{L}(n)$, then l_P is ignored as it is not a skyline path for the node n. In a non-decreasing cost graph, the path with l_P as its label will never have an extension that dominates the dominating path labels in $\mathcal{L}(n)$. Finally, if l_P is incomparable to all the labels in $\mathcal{L}(n)$, then it is a Skyline point and is included in $\mathcal{L}(n)$.

4.3 Pseudocode Explanation

Figure 4.2 shows details of our algorithm. Skyline paths are desired between the source node s and destination node t while oonsidering a subset c of all possible criterias. The algorithm starts by initializing label sets for nodes as empty sets, a paths priority queue and an output set {SP. An entry in the priority queue Q for a path $P(s, n)$ contains the set of nodes in P, the head node n, the monotonic score $C(P(s, n))$ and accumulated cost

Algorithm SkyLinePathAlgorithm(Graph a, Source node s, Destination t,Criteria Set c)
Local PriorityQueue Q;
Output PathSet $\mathcal{SP}$;
$\mathcal{L}(n) = \{\}$; (forall n)
$\mathcal{SP} = \{\}$
Begin
1. $Enqueue(Q, (s, \{s\}, C(P(s,s)), C_c(P(s,s)))$
2. While (Q is not empty)
3. $l = (n, P, x, y) \leftarrow dequeue(Q)$;
4. If $\exists l' \in \mathcal{L}(n), l' \vdash l$, then go to Step 2;//label dominated by an existing path from s to n
5. If $(n \neq t)$
6. If $\exists l' \in \mathcal{L}(t), l' \vdash l$, then go to Step 2;
7. For each neighbor n' of n do // if the label l is not dominated
8. if n' is already in P then continue; // avoid loops
9. $Enqueue(Q, (n', P \cup \{n'\}, x + C(n,n'), y + C_c(n,n')))$;
10. End For
11. $else\ \mathcal{SP} = \mathcal{SP} \cup P$;
12. $\mathcal{L}(n) \leftarrow \mathcal{L}(n) \cup l$;
13. End while;
14. Return $\mathcal{SP}$
End.

Figure 4.2: Skyline Path Search Algorithm.

vector $C_c(P(s,n))$. The priority queue Q is sorted based on the monotonic score of each path $C(P(s,n))$. The monotonic score is calculated by adding all the cost components $C_j(P(s,n))$ where $j \in c$, $n \in N$, $n \neq s$.

Initially a seed path with a single node s is enqueued into the priority queue Q at step 1. This path has a monotonic score of 0 and all its component costs are zero as well. On each iteration here onwards, the path with the lowest monotonic score $C(P(s,n))$ is dequeued from Q as shown in step 3. In the first iteration, path $P(s,s)$ will be dequeued as it is the only path in Q.

The label l of the dequeued path $P(s,n)$ is computed in step 3 and compared with the label set $\mathcal{L}(n')$ of the P's head node n in Step 4. If any label in label set $\mathcal{L}(n')$ dominates l, we simply ignore the path and go back to step 2. In a network with non-decreasing monotoni cost functions, P with a dominated label l at node n will never extend further to form a path that will dominate any extensions of paths already present within in the label set $\mathcal{L}(n')$.

In the othercase, the label l can be non-dominated for the labels in $\mathcal{L}(n')$. In this case l will be *incomparable* to the existing labels in the node label set. An incomparable pair of labels is better in some criteria and worse in the remaining criteria. If the new label is incomparable to all labels in the label set, then it is also to be added to the label set as this label could be one of the skyline labels we need to track in future path extensions. It is worth mentioning here that being dominated by any *one* label in $\mathcal{L}(n')$ disqualifies the path P from continuing further, whereas being incomparable to *all* the labels in $\mathcal{L}(n')$ only qualifies the path to have its label included in set $\mathcal{L}(n')$. The main idea is to allow only skyline paths at a node to extend further in the network. All paths that are not skyline paths at their head nodes are halted immediately.

All paths that are filtered through step 4 are paths that have incomparable labels and are to be included in the label set of their respective head node label sets. These paths can have two possibilities for the head node i.e. They can either have t as the head node, meaning they have already reached the destination or they can have any other node n as the head node where $n \neq t$ and $n \neq s$ meaning they are still searching. Paths that have already reached the destination are not required to be extended anymore. We first discuss the case where the path has not reached the final destination (where $n \neq t$) and hence can proceed further. In Step 6 the path is checked for dominance against the labels in the set $\mathcal{L}(t)$. Further processing of any dequeued path is only useful and allowed if it has an incomparable label for label set $\mathcal{L}(t)$. Since the network is non-decreasing any path to an intermediate node if dominated by the label set of final destination, will not

be able to dominate the existing labels at t. This is the additional check in our algorithm for pruning out useless paths. All paths that are not pruned out at step 6 are to be extended. For the skyline path P, this is achieved by appending to P the edges that exist between n and its adjacent nodes as long as they are not already in P(step 8). By doing this check, we avoid any cycles in the paths. Each resulting path formed by appending a new one new edge (n, n') to P is then to be enqueued into the priority queue for later processing. To calculate the monotonic score $C(P(n'n))$ of the new path we add in step 9 the monotonic score of the path $P(s, n)$ and edge (n, n'). Moreover in the same enqueue, in order to calculate the cummulative cost vector $C_c(P(s, n'))$ of the new path we treat the (n, n') as a path having a cost vector $C_c(P(n, n'))$ and add both the vectors. The new path is enqueued into the priority queue Q at a position based on its monotonic score.

On the other hand, for paths that have already reached the node t, instead of extending them further we add them to our result set SP. Step 11 performs this function. In step 4, we decide whether the dequeued path P is to be included as a skyline path for its head node. All paths that pass this condition are skyline and in step 12 we include their label into the label set of their respective head nodes. The algorithm that goes onto the next iteration and dequeues the next lowest monotonic cost path for its respective processing.

Finally, when the Q is empty our algorithm terminates and at this point set SP has all the paths that are Skyline going from node s to node t. These paths are the final result of the algorithm and are returned in step 14.

4.4 Running Example

We now present a running example of the proposed algorithm. In figure 4.3, a unidirectional graph can be seen with 3 dimensional costs assigned to each edge. The skyline paths required are between nodes s and t while considering costs set $c = C_1, C_2, C_3$. For each iteration we show the path with the lowest monotonic score, its extensions, the state of the label set at its head node and finalized Skyline paths SP set at t. Each path in the table 4.4is represented by $\langle C_1, C_2, C_3, C_1 + C_2 + C_3, n\rangle$ where C_1, C_2, C_3 represent the three costs considered, $C_1 + C_2 + C_3$ represents the monotonic score of the path at head node n.

As we explained earlier, in the first iteratio we will dequeue the seed path that has just one node s. Since there are no labels in any label set of the network, this path will not have to be compared with any label from the sets $\mathcal{L}(s)$ or $\mathcal{L}(t)$. The zero label will be included in the label set of node s as it can be seen in table 4.4. Before the label is added

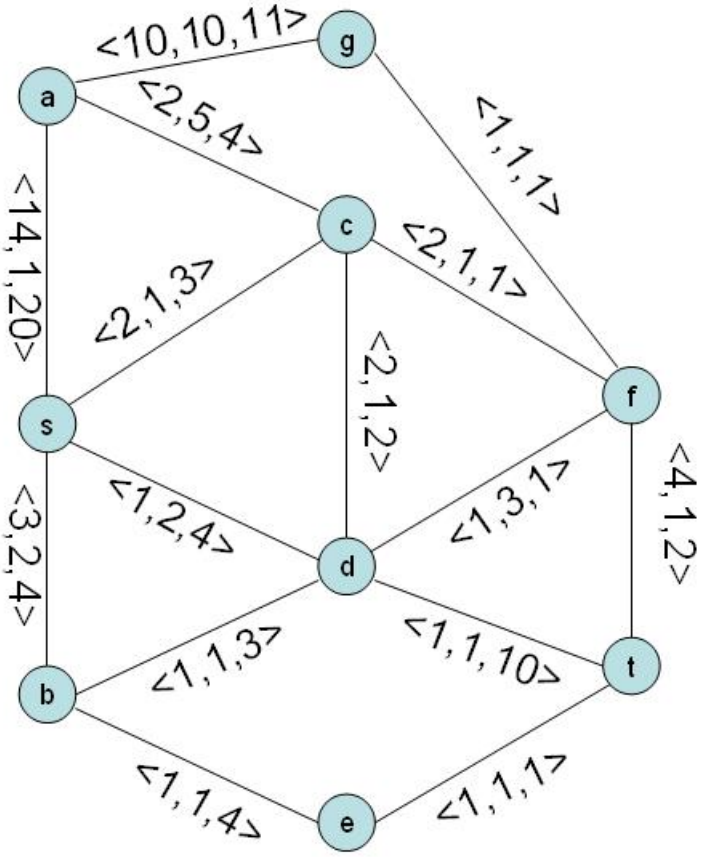

Figure 4.3: Graph for Illustration of Skyline Running Example

to the set, all paths resulting from appending all outgoing edges of the head node (s), namely $\{s, a\}$,$\{s, b\}$,$\{s, c\}$ and $\{s, d\}$ will also be enqueued into the priority queue. These paths can be seen in the second column of the table. Paths will placed in the priority queue in increasing order of monotonic score.

In iteration 2, path $\{s, c\}$ has the lowest monotonic score. We know that set $\mathcal{L}(c)$ has no labels and there are no paths yet to reach the node t, label of path <2,1,3> is accepted as a skyline label at node c. Again before we add this label to $\mathcal{L}(c)$, we extend the path $\{s, c\}$ with all the outgoing nodes that result in non-cyclic paths and enqueue them into the priority queue. The algorithm then continues in a similar fashion until the 4 iteration. At iteration 5, path $\{s, c, d\}$ has the lowest score in the priority queue. After the dequeue we compare the cost vector generated by this path <4,2,5> with the labels already in the set $\mathcal{L}(d)$. The label set $\mathcal{L}(d)$ has just one label at this time which is <1,2,4> of path $\{s, d\}$. We compare the corresponding entries of these two labels and find out that the label of path $\{s, c, d\}$ is dominated by the label of path $\{s, d\}$ (C_1 4>1, for C_2 2=2 and for C_3 5>4). We conclude this by definition of *dominance* defined earlier according to which a path dominates another if it better in atleast one cost and not worse in all remaining costs than another path. The path $\{s, c, d\}$ is hence not included in the set $\mathcal{L}(d)$ and none of its extensions are included in the priority queue. Similar scenario happens in iteration 6 where path $\{s, d, c\}$ is found to be dominated by already existing labels in set $\mathcal{L}(c)$.

The algorithm continues uptil iteration 13. In iteration 14 the path $\{s, c, f, t\}$ has the lowest score in the queue and is dequeued. This path is special as it has the head node as the destination node. We check to see if the label of this path <8,3,6> is dominated by any label in the set $\mathcal{L}(t)$. $\mathcal{L}(t)$ is empty hence this is the first skyline path to reach the final destination. Instead of extending this path by the adjacent nodes of node t, we simple add it to the final result set SP. No enqueues will be done in this iteration. According to our algorithm after finding the first path to node t, we have the ability to prune out paths that have not reached t but are dominated by the label set $\mathcal{L}(t)$. In our network with only non-decreasing monotonic cost functions, these pruned paths will never be able to extend and form paths that will dominate the paths with labels in $\mathcal{L}(t)$.

Similarly more paths will be added to the set SP in iterations 16, 17, 18 and 19 and their labels will be added to the set $\mathcal{L}(t)$. In iteration 20 the path $\{s, a\}$ will be dequeued. We compare the label of this path <14,1,20> with labels in set $L(a)$ and find it to be incomparable. Moreoever, this label is also incomparable to the label set $\mathcal{L}(t)$. Hence we include this label in the label set $\mathcal{L}(a)$. Furthermore this path has two extensions $\{s, a, g\}$

It.	Lowest score P	Enqueued Extensions	Head Node label set	Skyline paths at t
1	{s},C=0	<14,1,20,35,a>,<3,2,4,9,b>,<2,1,3,6,c>,<1,2,4,7,d>	{s}	{}
2	{s,c},C=6	<4,2,4,10,f>,<4,6,7,17,a>,<4,2,5,11,d>	$\{\{s,c\}\}$	{}
3	{s,d},C=7	<2,5,5,12,f>,(<2,3,7,12,b>,<2,3,14,19,t>,<3,3,6,13,c>	$\{\{s,d\}\}$	{}
4	{s,b},C=9	<4,3,8,15,e>,<4,3,7,14,d>	$\{\{s,b\}\}$	{}
5	{s,c,f},C=10	<8,3,6,17,t>, <5,5,5,15,d>	$\{\{s,c,f\}\}$	{}
6	{s,c,d},C=11	(NE) - Dominated label in $\mathcal{L}(d)$	$\{\{s,d\}\}$	{}
7	{s,d,c},C=12	(NE) - Dominated label in $\mathcal{L}(c)$	$\{\{s,c\}\}$	{}
8	{s,d,f},C=12	<4,6,6,14,c>,<6,6,7,19,t>	$\{\{s,c,f\},\{s,d,f\}\}$	{}
9	{s,d,b},C=12	<3,4,11,18,e>	$\{\{s,b\},\{s,d,b\}\}$	{}
10	{s,b,d},C=14	(NE) - Dominated label in $\mathcal{L}(d)$	$\{\{s,d\}\}$	{}
11	{s,b,e},C=15	<5,4,9,18,t>	$\{\{s,b,e\}\}$	{}
12	{s,c,f,d},C=15	(NE) - Dominated label in $\mathcal{L}(d)$	$\{\{s,d\}\}$	{}
13	{s,d,f,c},C=16	(NE) - Dominated label in $\mathcal{L}(c)$	$\{\{s,c\}\}$	{}
14	{s,c,a},C=17	(NE) - No adjacent Node	$\{\{s,c,a\}\}$	{}
15	{s,c,f,t},C=17	(NE) - Destination Node	$\{\{s,t\}\}$	$\{\{s,t\}\}$
16	{s,d,b,e},C=18	<4,5,12,21,t>	$\{\{s,b,e\},\{s,d,b,e\}\}$	$\{\{s,t\}\}$
17	{s,b,e,t},C=18	(NE) - Destination Node	$\{\{s,t\},\{s,b,e,t\}\}$	$\{\{s,t\},\{s,b,e,t\}\}$
18	{s,d,t},C=19	(NE) - Destination Node	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\}\}$	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\}\}$
19	{s,d,f,t},C=19	(NE) - Destination Node	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\},\{s,d,f,t\}\}$	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\},\{s,d,f,t\}\}$
20	{s,d,b,e,t},C=21	(NE) - Destination Node	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\},\{s,d,f,t\},\{s,d,b,e,t\}\}$	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\},\{s,d,f,t\},\{s,d,b,e,t\}\}$
21	{s,a},C=35	<24,11,31,66,g>,<16,6,24,46,c>	$\{\{s,c,a\}\{s,a\}\}$	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\},\{s,d,f,t\},\{s,d,b,e,t\}\}$
22	{s,a,c},C=46	(NE) - Dominated label in $\mathcal{L}(c)$	$\{\{s,c\}\}$	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\},\{s,d,f,t\},\{s,d,b,e,t\}\}$
23	{s,a,g},C=66	(NE) - Dominated label in $\mathcal{L}(t)$	{}	$\{\{s,t\},\{s,b,e,t\},\{s,d,t\},\{s,d,f,t\},\{s,d,b,e,t\}\}$

Table 4.2: Example run for Skyline Path Search algorithm

and $\{s,a,c\}$ with labels <24,11,31> and <16,6,24> respectively. Both these paths are enqueued into the queue however, on their respetive dequeues both of these paths are found to be dominated by the labels in $\mathcal{L}(t)$ and hence are not extended further.

At this point the priority queue is empty. At the end of the execution when the queue is empty, we have a set of skyline paths <2,3,14>, <8,3,6>, <6,6,7>, <5,4,9> and <4,5,12> starting at node s and ending at node t. Any path amongst these is able to provide atleast one cost that is better than atleast one other path in this set.

4.5 Discussion

Finding all possible multicriteria paths between two nodes of a network is a challenging problem. Finding multicriteria paths that do not dominate each other is an even harder problem. The problem has an exponential growth of number of candidate paths for the final skyline path set as the distance between the source and the destination nodes is increased. Our algorithm firstly arranges the more important paths in an order such that the most likely paths are considered first. Secondly, our algorithm counters the exponential growth by using conventional skyline semantics. By progressively applying

conventional skyline semantics like *transitivity* and *incomparability* to all the possible candidate paths between the source and destination, the algorithm prunes out the extra candidates paths and reduce the search space and avoiding un-needed processing.

The sorted priority queue of paths based on monotonic score ensures that dominating paths for any node in the network with lower costs are extended before the dominated paths. By extended the dominating paths first and consequently letting them reach nodes earlier than the dominating paths for the same nodes, we increase the chances of extracting the dominating paths first and hence save cost to extend the dominated paths. On reaching a dominated path that was initially lying in the priority queue behind the dominating paths, its simple dequeue and label comparison with the label set at the head node is all the required processing needed for its removal from the queue. Moreover with only dominating paths allowed at each node, the label set at head nodes is comparatively smaller and the dominance check hence is not time consuming. Therefore, all dominated cadidate paths are halted with little overhead as soon as they are found to be dominated by any other path.

We utilize two pruning methods. The first of these is mentioned above. The second pruning method starts as soon as the first path that reaches the final destination node t. As we have seen in figure 4.2 a dominance check with the label set of destination node is the first check after any path with any destination node is dequeued from the priority queue of paths. If this path is dominated by the label set of t, the path is pruned. Although this check is based on *imcomparable* paths, having the same source but different destinations, we know that since t is the desired destination node of all paths, we can justify this pruning approach. Using two pruning techniques greatly improve the performance of our algorithm as we have seen through evaluations.

The skyline path search algorithm is a novel idea. We have now transformed the case of finding multidimensional skyline points to multidimensional skyline points. In Chapter 6, experiments and evaluations of our Skyline Path search algorithm show an efficiently performing algorithm consistent 3-phase behaviour with regards to time and resources. In the third and the longest lasting part, our algorithm is able to "dampen" the time taken and database accesses and all of them grow at a very small rate as the candidate number of paths increases.

By using two efficient data structures for implementing our approach, we have gained performance efficiency. This will be more clear in chapter 6.

5. Path Advice 1.0

In this chapter we review our prototype path recommendation system called *PathAdvice*. At a higher level *PathAdvice* is an integrated suite with different modules, details of which will be provided in the upcoming sections.

5.1 System Overview

PathAdvice is a location based software suite that provides path recommendations to its users in a quick and a user friendly manner. As compared to the current path search service providers, *PathAdvice* has enhanced spatial search capabilities and can provide itinerary suggestions that are more comprehensive and applicable for the real word. *PathAdvice* uses traffic information on an hour-to-hour basis to estimate on road travel times for each of its suggested routes and hence empowering the user with decision capability over real-time information. *PathAdvice* is a flexible and scalable set of softwares that allows administrators to modify and upgrade information being used for location based services.

Figure 5.1 provides the higher level architecture of *PathAdvice*. *PathAdvice* consists of three basic levels of implementation. The front end, logic and database layer. Object Oriented model has been adopted for all three layers. Dividing the three major functionalities into three sections has reduced coupling amongst the system modules and with platform independent Java based implementation, these layers can act as "plug and play" modules for other systems as well.

The database layer is responsible for storing and providing road network information to the logic layer. The logic layer consists of logical objects that perform various functionalities including algorithm execution, caching, mathematical conversions etc. Moreover, this layer also is also responsible for moulding data into a form that can be utilized by the front end layer to display information to the user and by the database layer to store in tables. The front end layer is the main interface of user interface where user interaction is the main goal.

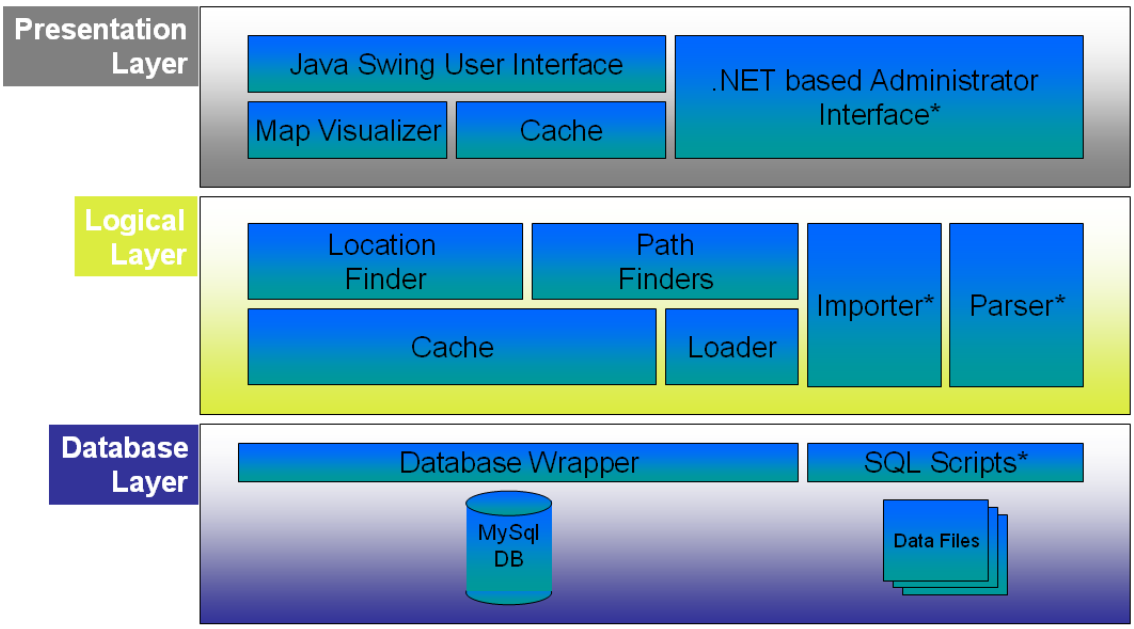

Figure 5.1: Path Advice System Architecture

PathAdvice comes with two usage modes namely the *Management Mode* and the *User Mode*. The former is to be used for performing back end activities like importing, updating data and the latter is to be used by people who are interested in doing path search through *Path Advice*. Note: Modules marked with a '*' in figure 5.1 indicate the *Management Modules* of *PathAdvice*.

5.2 System Operation

In Figure 5.1 we can see many modules inside each of the layers, we will describe these in detail below.

5.2.1 Database Layer

TIGER/Line dataset

We start off by introducing TIGER/Line data set that was used as the underlying source for road networks for *PathAdvice*. TIGER (Topologically Integrated Geographic Encoding and Referencing system)/Line is accepted as a standard when it comes to being a vast source of information for the Road Networks in USA. The design of the TIGER database adapts the theories of topology, graph theory, and associated fields of mathematics to provide a disciplined, mathematical description for the geographic structure of the United

States and its territories. The topological structure of the TIGER data base defines the location and relationship of streets, rivers, railroads, and other features to each other and to the numerous geographic entities for which the Census Bureau tabulates data from its censuses and sample surveys. It is designed to assure no duplication of these features or areas.

The TIGER/Line files are a digital database of geographic features, such as roads, railroads, rivers, lakes, legal boundaries, census statistical boundaries, etc. covering the entire United States. The data base contains information about these features such as their location in latitude and longitude, the name, the type of feature, address ranges for most streets, the geographic relationship to other features, and other related information. These advantages make the use of this data set as the most comprehensive source of information.

The TIGER/Line dataset has divided the different elements of road networks into neatly separated types. The record types are each separated from each other using unique identification which greatly helps for implementation purposes. For instance facts on roads - whether they are under ground or in open, water pipes routes, rail road tracks, the address ranges on each side of the road and the zip codes related to them, whether the road is a high way or a small neighborhood road or whether it is a restricted road or a non restricted one, are all mentioned in record type 1 of the data set. There are 18 other types of records each having its own focus.

Database Wrapper

This module is responsible for extracting data out of the database after the TIGER/Line data has been imported. The wrapper is the primary link of the top layers with the Database. Using Mysql Java connector, stored procedure calls are made to the database tables. With a large dataset to handle and limited heap space for java processes available, this module plays a very vital. To improve avoid heap space and efficiency threatened by many database calls propagated down from the front end and logic layers, this module adjusts its database connection mechanism. This is done through Singleton connection streams maintained for each of the object type which are flushed and reinitialized as per need basis.

Database Schema

In figure 5.2 we show the schema for *PathAdvice* database. We can see that there are three types of tables designed. The TIGER/Line tables are mapped with exact properties

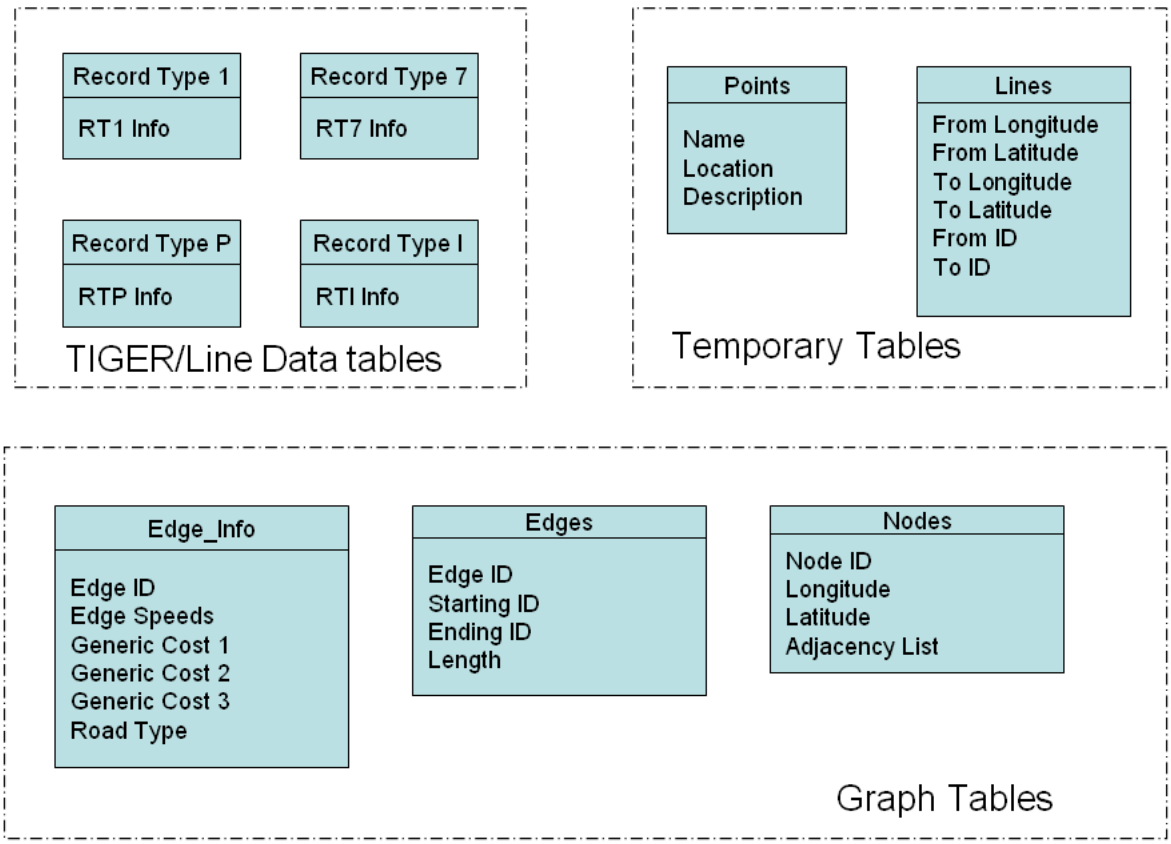

Figure 5.2: Relational Database Schema

set of record types in the TIGER/Line dataset. These tables are filled by Importer modules (See section 5.2.2) after the TIGER/Line dataset has been parsed by Parser module (See section 5.2.2). These purely dataset tables are then processed to extract out temporary tables consisting of basic extracted information of the TIGER/Line dataset. Temporary tables are used to improve performance while constructing the spatial graph which is an extremely time consuming process. For this purpose, temporary tables are indexed on in-built MySQL spatial data libraries to greatly improve lookup activities. Lastly, the permanent relational database tables are filled in using information from both TIGER/Line tables as well as temporary tables. During this step the adjacency list of nodes is also prepared that contains all nodes which are connected to outgoing edges of a node. During User mode all three types of tables are used in combination to provide information for the user.

The schema has been designed to be scalable. With 19 record types available in TIGER/Line dataset this database is capable of storing all of them. Basic graph information stored in the Graph tables can be used in combination with as many different record types as desired.

5.2.2 Logical Layer

Parser

Parsing the large amount of Data in TIGER/Line is considerably tedious with information about the 19 record types provided in the form of values and their corresponding offsets in the text files. Each record type is stored in a text file for each county of every state in America. The data in TIGER/Line has been compressed and hence the information is not readily available by just simply having the dataset. *PathAdvice* includes Parser module that automates the parsing of this complex and large dataset according to the specification provided. XML configuration files have been used in this module to define the formats of inputs from the TIGER/Line dataset. TIGER/Line dataset is an ever growing and changing dataset and any furture changes in TIGER/Line data formats can easily be handled by changing just the Parser configuration files. Further the dependency of *PathAdvice* hence can be easily changed to any other dataset if there is a need.

Outputs from Parser are text files. The module recursively goes through the data of each county in a state-wise manner and picks the neccessary information that is to be stored in the database. Parser filters data that is not needed, for instance by making a few changes in XML configuration files this module can seperate out the road data from all of the other types of data like train tracks, water locations. Parser module has a friendly user interface developed in .NET technology and administrators of *PathAdvice* can parse large amounts of data in TIGER/Line with a few simple clicks.

Importer

Once the data has been brought into the form of text files, the Importer module imports data about roads into *PathAdvice* relational database format. Besides being the link between the Parser and the relational database, the basic operation of this module is extraction of a connected graph representing road networks. Using each road intersection as a node and each road itself as the edge, the module populates temporary tables indexed with MySQL libraries. After this step the temporary tables are used by the module to prepare a uni-directional graph and stores it into the relational database in a quick manner. The importer module consists of SQL scripts.

Cache

As the road data is a large dataset, retrieval of all information in one burst is not efficient. We discussed the solution of this problem in section 5.2.1. We know that *PathAdvice*

has adjustable streaming connections to the MySql database that retrieved spatial information in chunks and not in one burst. This reduces the in-memory space requirement of the system and increases the speed of map formation on the display window. While the data is being streamed in, soft cache is used to store the results. This cache not only helps in combining data into one set after it is retrieved in chunks from the Database Wrapper, but also this cache acts as the primary datasource for later retrievals and hence improves performance. One example of such data is the road information for roads that are to be drawn on the screen for the user. Not all roads that lie within the screen of the user are retrieved at once. They are retrieved in multiple chunks and stored in the cache. Now, whenever there is a need to display a different part of the map on the screen e.g. a Zoom out or a click towards South, this cache can quickly provides us with the road information and the user experience is much smoother.

Path Finders

To the end user, *PathAdvice* is a software that caters to their travel needs. To normal users performance of the software and accuracy of results is the more important aspect rather than the behind-the-scene methodologies. With its layered and modularized design, *PathAdvice* allows the inclusion of any path search implementations in the logic layer as long as they utilize the underlying graph structure. For including any improvements in the path search implementation or replacement of current algorithms with any other search method no other changes are required in the front-end, logic and database layers. We have included *Passage Counting Algorithm* and *Skyline Search Algorithm* as the two sub modules of Path Finders in this version of *PathAdvice*.

Loader

In Chapter 2, we briefly mentioned the work [23]. The *CapCod* paper simplifies the traffic modelling of road networks greatly by using speed patterns observed at different times of the day for each road and then deducing as many linear travel time functions as required to represent time required to travel the full length of the road. Any point on any of the linear function for a road within the 24-hour period is the time taken to travel the particular road at that time instant of the day. This set of travel time functions for all edges are called the *CapCod* network.

In *PathAdvice*, we similate speed for each hour in the 24-hour for each road in the network. The speeds are generated based on the type of road under consideration e.g. Highways speeds lie anywhere above 50 mph and less than 70 mph whereas the smaller

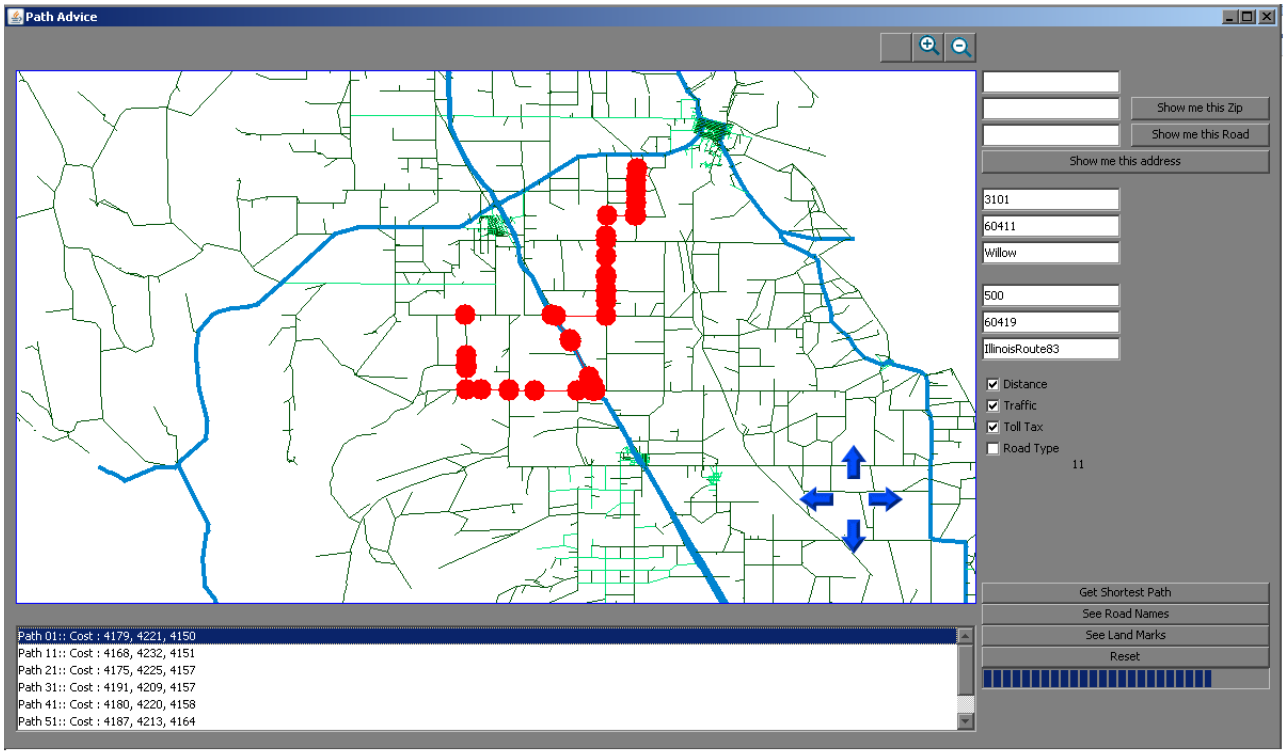

Figure 5.3: User Interface

neighbourhood roads have been given speeds les than 20 mph. Speed patterns are loaded by the *Loader* into a specialized data structure for *CapCod* network when *PathAdvice* is first started. This network is then stored in the memory throughout the running of the application and is utilized to estimate travel times for each of the route the user is presented as a result of his path search.

5.2.3 Presentation Layer

Map Visualizer

This module allows the user to view real world maps with the help of Java Swing libraries. The module provides basic functionalities of map manipulation like zooming in/out, navigation in the east, west, north and south directions and display of information about the road network. The longitude and latitude provided by the database are converted to coordinates on the viewing panel and appear on the display panel depending on the zoom level set by the user. The points that lie outside the boundaries are not queried from the cache, enhancing the visual performance.

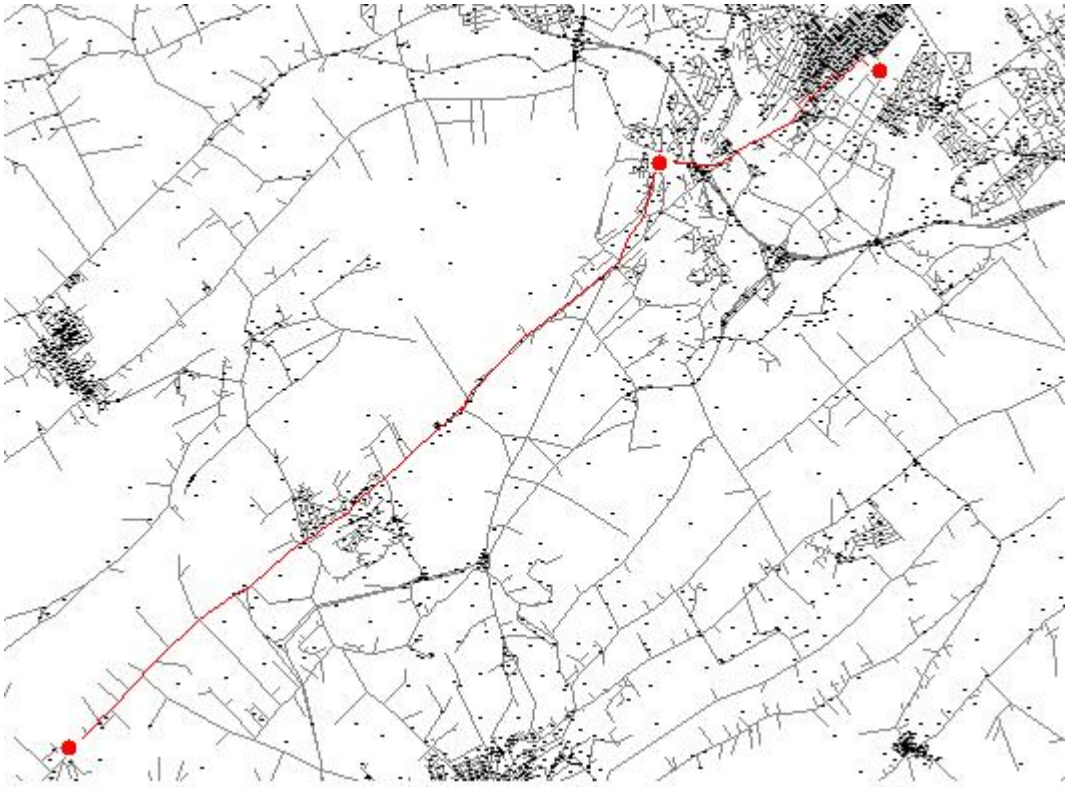

Figure 5.4: Centre County Road Network

Map Components

The implementation divides the map into different display components that combine together to produce the required functionality. Below are some basic details about these components.

Nodes

The Importer module converts the graph into set of nodes and edges during the execution. The nodes in the context of the map are the road junctions that are given a unique id. Each road in the road network is the edge between these nodes. The nodes from TIGER/Line data are mapped onto the user screen using latitude and longitude values with 7 digit accuracy. Each node maintains an adjacency list that has all its one hop neighbors in it and we do not need to retrieve information about node's adjacent lists by parsing the complete set of nodes. This however, adds pre-computation costs.(See section 5.2.2)

Edges

The edges in the map can consist of water areas, roads, rail tracks or underground pedestrian passages etc. However, there is a need to separate out the roads from all records during the Importing of data. This is done though placing checks on the road types in the TIGER/Line. The TIGER/Line gives a special type to each type of edges. For Example the roads in the network are given CFCC values from A1 to A74. These different types of roads indicate roads that are tunnels, are restricted or normal highways or neighborhood roads etc. TIGER/Line dataset gives each edge starting and ending points in the form of coordinates in the longitude and latitude plane.

Landmarks

The TIGER/Line comes with information about the well known locations within the roads as well e.g. Parks, Churches, Libraries etc. These locations are named as landmarks. Each landmark latitude and longitude value and is overlapped with any address. Through the relational database design, customized landmarks can also be added and re-used for a particular user.

Addresses

The TIGER/Line provides extensive information about the address ranges that lie on the roads. Each road has different sets of addresses on each side. This includes having different Zip Code for each side.

Users of the system can view addresses on their screens using the Display module. Linear interpolation has been used to map the address on one appropriate side of the road on the map screen. For example if address numbers start from A_i and end at A_j on one side of the road having a Euclidian distance x between them on the user screen then the address at point p of the edge would be at $(P - A_i)/A_j$ from the starting point A_i.

During path finding the node nearest to the address location on the edge is taken as the representative of the address.

Java Swing

Apart from displaying the map on the screen this module provides the user of the system the ability to interact with the system and add their information. The user is able to provide specifications using check boxes (for different cost criterias)and text boxes and then requests the system to return one of more paths. If the user selects multiple costs, the

Skyline Search Algorithm is active otherwise with one cost selected the *Passage Counting Algorithm* is applied and results are provided back to the user. Figure 5.3 shows the application lay out. The user can then see all of the possible results on the list box and double clicking on each of the paths displays the path on the map of the user.

Cache

The front end has the ability to cache resulted paths that are retrieved using any of the approached from the Logical Layer. For multiple paths it is important that the request is not sent repeatedly to the database for retrieving each of the returned paths. Whenever user clicks the search button all results retrieved are stored in the cache and requests for viewing and reviewing returned paths are then service from the cache. This greately helps in improving the user interface performance.

5.2.4 User Friendly Features

PathAdvice is a user friendly environment which provides path finding functionalities to the user. A few of the features are mentioned below

- Location Search : Ability to search addresses by address number, Street and Zipcode on the Map.
- Map Navigation : Ability to navigate the map in all directions along with zooming in and out.
- Path Search : Ability to provide source and destination addresses to the system and performing search based on single or multiple criterias.
- Path Display : Ability to see all resultant paths on the map
- Time Estimation : Ability to see estimated time for each return path considering traffic information

All in all *PathAdvice* is the corner stone in our implementation which allows us to view the results of our algorithms and also allow users of the system to view the recommedations in a simplified, practical manner.

PathAdvice has a completely modular approach based on Object Oriented Design. The flexible and scalable design of the system allows administrators to add new application features, algorithms and data without any change to the existing structure. For any processing on the same dataset, different types of networks can be adopted by the

'Loader' as required, with all other modules remaining un-affected. Similarly, any different approach for path finding can be adopted by simply modifying the 'Path Finder' only. With Cache implemented in different modules, the performance of the application is improved and repeated accesses to the database are avoided.

With user mode of the application, the average users of *PathAdvice* can easily provide path search criterias and obtain recommendations for their journey. The criterias include number of paths to be returned or as many as 4 different criterias to be considered for path search. Moreoever, users can compare the resultant paths based on the current traffic information available. These more real world like features make *PathAdvice* a novel work in the area.

6. Evaluation and Performance

In this chapter we provide the evaluations of Passage Count algorithm(PCA) and Skyline Path Algorithm (SPA) described in the last two chapters.

6.1 Environment Set up

PathAdvice installed on a Pentium machine with 1.66Ghz processor and 1 GB of RAM (See Chapter 5) was used as the testing environment. We tested our results on two real world road networks.

- Centre County, Pennsylvania, USA with 15000 nodes and 40000 edges (From here on called N1)
- Colusa County, California, USA with 6000 nodes and 15000 edges. (From here on called N2)

The network was assumed to be uni-directional. Distance of each edge in the network was calculated by estimating the Eucledian distance between its starting and ending nodes. Further, for the purpose of experiments, we simulated 3 other costs randomly to each edge.

6.2 Passage Count Algorithm vs. Yen's Algorithm

6.2.1 Quantitative Analysis

. Yen's algorithm (from here on referred to as YA), is one of the most cited multiple path search algorithms. The algorithm was the first algorithm that reduced the time complexity to find multiple shortest paths between a pair of nodes, to a linear function. All previous efforts before this algorithm took polynomial time for the same problem. YA is a classical algorithm for the K-shortest path problem.

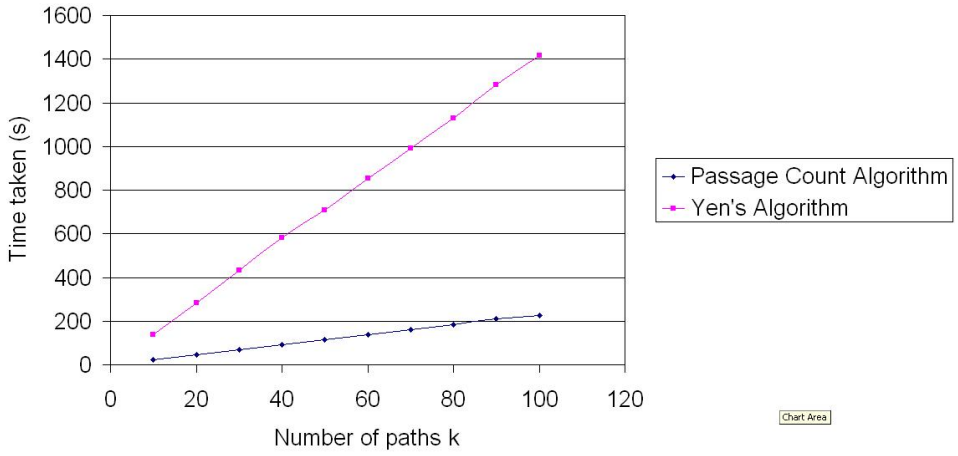

Figure 6.1: Yield time comparison in network N1 for longer paths

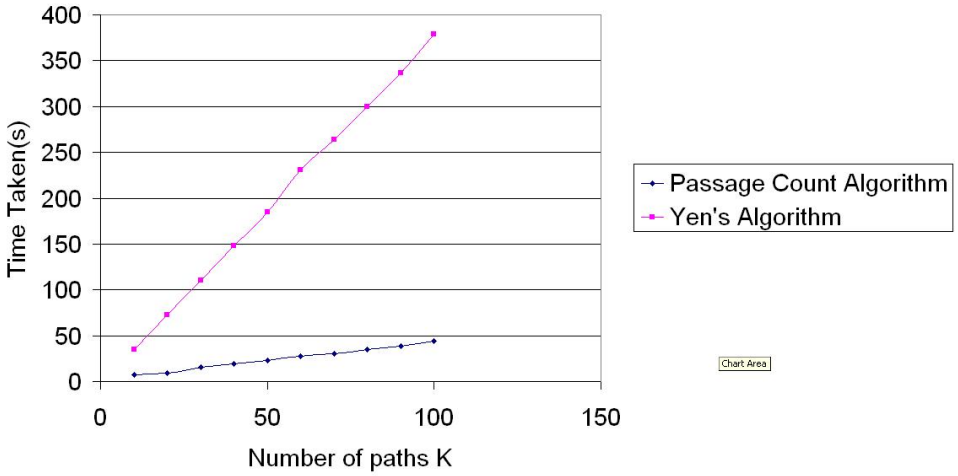

Figure 6.2: Yield time comparison in network N1 for shorter paths

We already have explained in previous chapters that to find the top-k mincost paths, PCA utilizes the greedy approach(Dijkstra's approach) whereas YA is based on finding mincost paths repeatedly with deviations and then choosing the lowest mincost path from amongst the shortest paths set to add to the final set of k paths. For finding mincost path in YA, any any well known single shortest path algorithm (Dijkstra/A*) can be used. We have used Dijkstra's algorithm as the underlying mincost path search algorithm for YA for our experiments. The results presented below provide a comprehensive quantitative and qualitative comparison between YA and PCA.

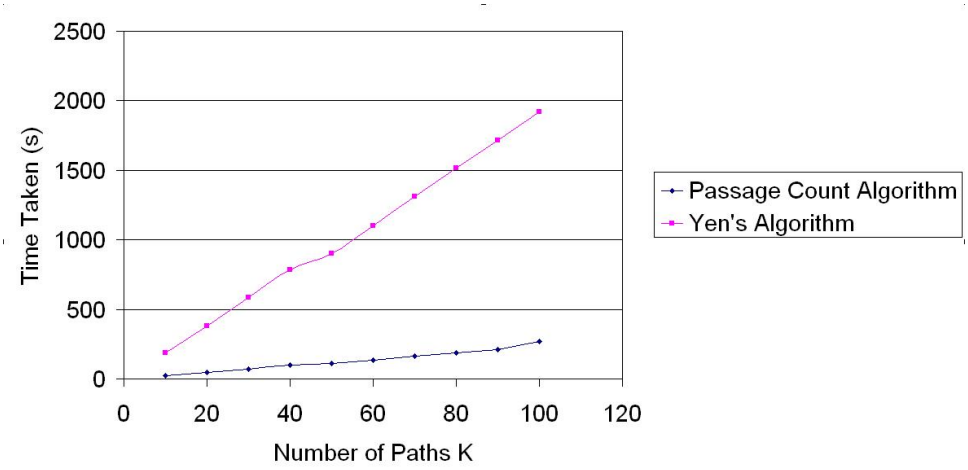

Figure 6.3: Yield time comparison in network N2 for longer paths

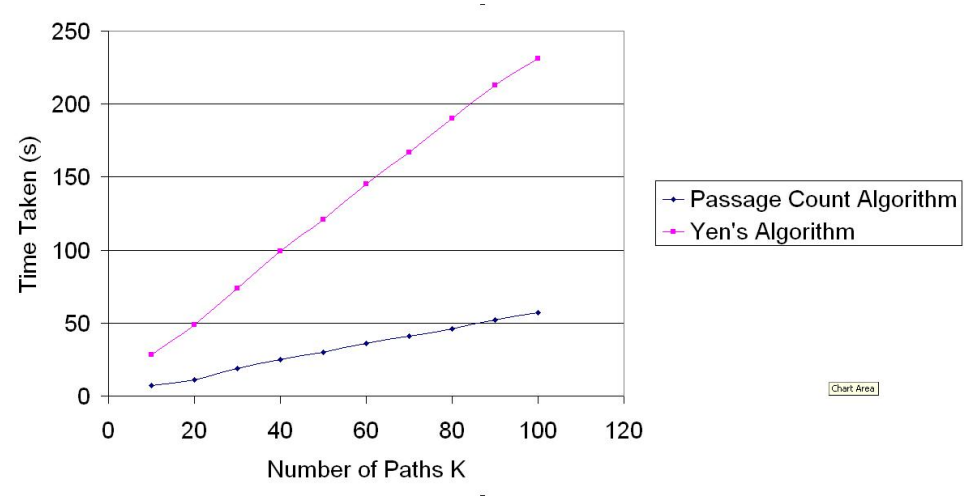

Figure 6.4: Yield time comparison in network N2 for shorter paths

Running Time Analysis

We define the Yield time for k-shortest path algorithm as the running time taken to identify all the top k paths. Yield time for any algorithm holds the key to an algorithms performance not only because it highlights the time complexity of the algorithm but it also identifies if the algorithm can be considered useful in real time systems that have to interact with the users. The first evaluation performed was the comparison between Yield time of YA and PCA for finding k number of mincost paths between two nodes in a graph. Two different scenarios were considered for both networks in this experiment, first being the case where the source and the destination nodes are nearby in proximity (less than 40 nodes away) and other second being the case when they are far away from each other (more than 100 nodes away). Value of k, the number of required mincost paths was varied between 10 and 100 and the results can be seen in Figures 6.1, 6.2, 6.3 and 6.4. Both algorithms show a linear increase rate but we can see that for both promixity cases (nearby and far), in both networks, and for all values of k, PCA is showing a smaller increase rate than YA. The reason for the large difference in Yield Time is due to the fact that YA is bound to apply dijkstra's algorithm many times during just one iteration (once for each node in the chosen best path from the priority queue). If the chosen path is long, the repeated deviation path calculation using Dijkstra's algorithm will cause many delays. PCA however, only applies the Dijkstra's algorithm k times and hence greatly reduces the number of times Dijkstra's algorithm is applied.

Database I/O Analysis

In scenarios with larger dataset such as our case of geo-spatial road network data with alot of information, another important aspect for algorithm comparison is the number of database I/O the algorithm requires.

Along with the yield time of the algorithm, it is important that the algorithm does not excessively require database retrievals that prove to be expensive especially when multiple users are using the system. We again indulge in comparing YA with PCA based on the number of database accesses required to find k number of paths between two nodes in a graph. In Figures 6.5 and 6.6, we can see the number of database retrieval calls (limited to adjacency list retrievals for this experiment) made by each algorithm to find K number of paths. Here again we can see that the increase in number of database accesses is linear for both the algroithm w.r.t. the value of k. An improvement of a factor equal to approximately 12 was found in PCA retrievals as compared to the YA retrievals for the N1 whereas for N2 we observe a Database accesses ratio to be around 6.

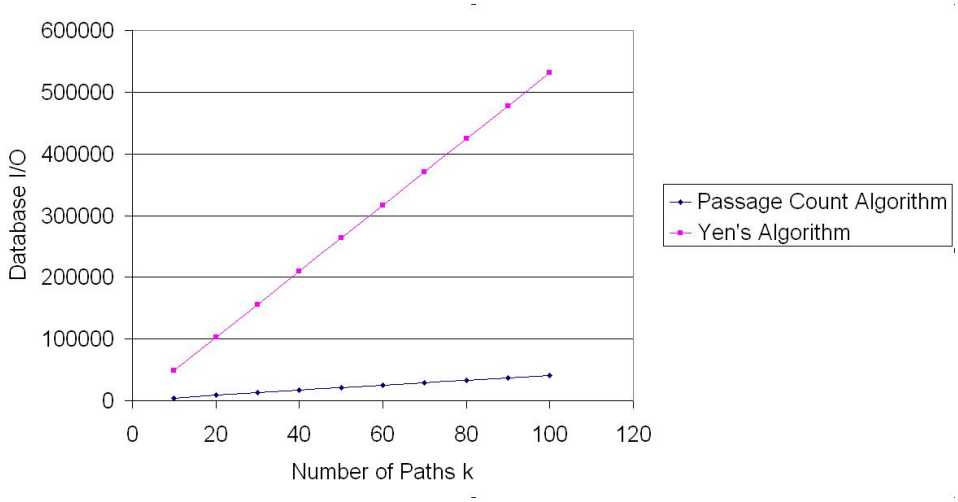

Figure 6.5: Database Accesses in network N1

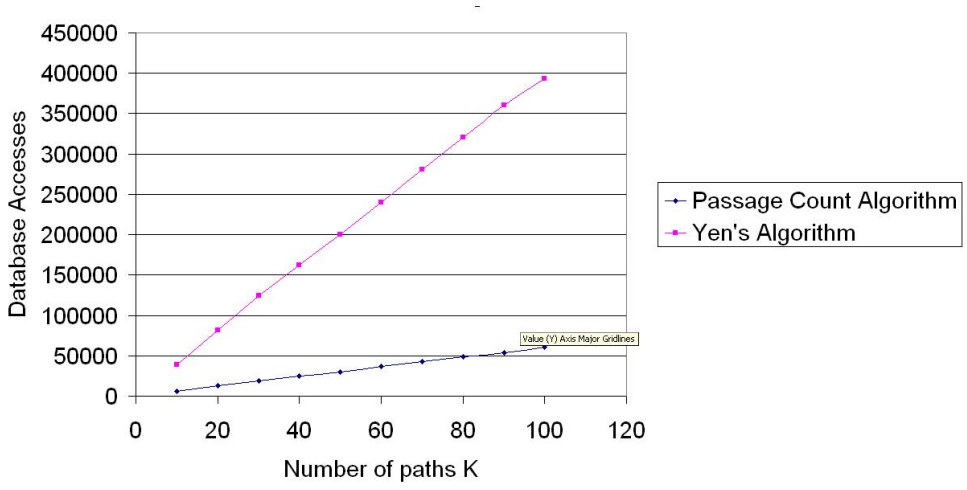

Figure 6.6: Database Accesses in network N2

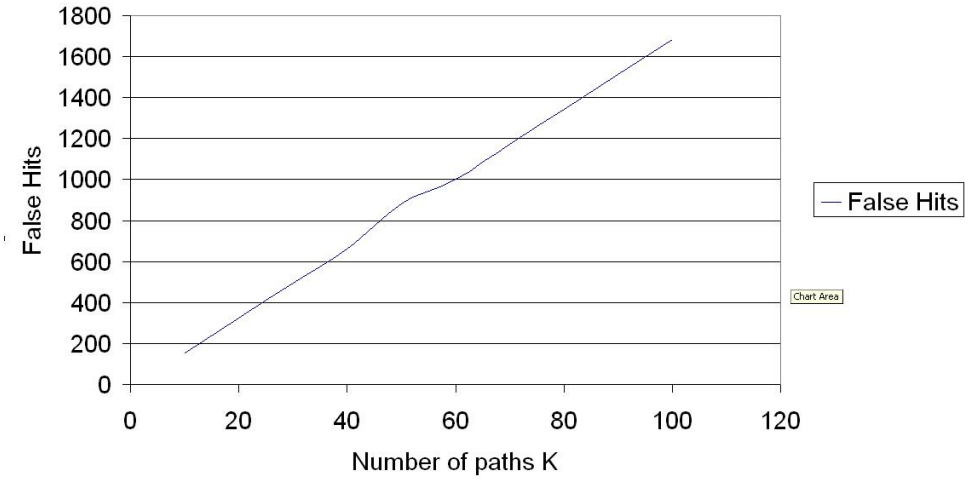

Figure 6.7: False Hits in Yen's algorithm for network N1

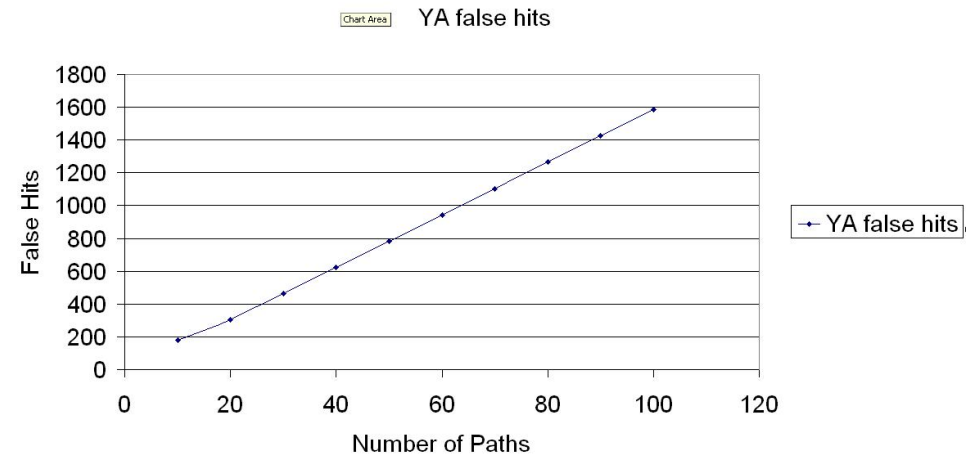

Figure 6.8: False Hits in Yen's algorithm for network N2

For both networks it was observed that the ratio of retrievals has an increasing trend for larger values of k as YA makes more and more retrievals for successive larger values of K from 10 to 100 to find deviation paths using multiple Dijkstra application per iteration. However with PCA, only k Dijkstra applications are required and this is clear by the low gradient curve for PCA. This clearly indicates the effectively reduced dependency of PCA on database calls.

6.2.2 False positives in Yen's Algorithm

Yen's algorithm maintains a priority list of paths from the source node to the destination node as it continues to explore the deviations. Each path in the priority list is a candidate

path which may or may not be a path of the final k paths that YA returns. We now refer to Figure 6.7 and Figure 6.8 which shows the maximum number of candidate paths maintained in the priority queue for different values of k. We can see that as k was varied from 10 to 100, the number of candidate paths that needed to be calculated rose at an approximate rate of 17 new paths per increment of K. This means that the Dijkstra's algorithm was applied 17 times to produce one more path in the final path set. This in contrast to the PCA that has the worst case requirement of just *one* more application of Dijkstra's algorithm for finding *one* more top-k path. In PCA no false hits are encountered as the algorithm has been designed in a manner that the first k paths reaching the final destination node are the indication that the algorithm should terminate as the final results will be complete with those first k paths only. Whereas in YA many paths between source s and destination t are found before the best paths can be chosen from amongst them and hence there is a huge overhead.

6.2.3 Qualitative Analysis

Another aspect of the algorithm evaluation is the quality of results produced. For the k-shortest path problem, the results returned should be best k ones over the complete range of possible paths going from the source node and ending at the destination node. When we consider the result set produced by Yen's algorithm we observe that the results could included non best paths. We know that the algoritm proceeds by exploring deviations from the source node onwards, exploring exhaustively all paths that lead up to the destination node. At the first look, this approach seems a good approach for finding the top-k paths however, when we consider the termination condition of YA, we can see that the exhaustive path search is carried out *only* until the first set has less than k paths. The specific iteration in which the kth path is achieved, the algorithm immediately terminates. This means that the algorithm does exhaustive deviation exploration until only a particular node. All deviations that lie beyond this node are ignored by YA. In a graph with arbitrary edge costs, any edges that lie beyond the termination node could may as well be *less* costly edges. If deviations of such nodes are explored there is a possibility of finding paths that infact shorter than the first k-paths that were deduced before the termination node. Overall, the results of YA hence are calculated without considering all paths between the source and destination nodes. PCA on the other hand is optimal in nature since Dijkstra's greedy approach exhaustively explores the edge space between any two points. In other words PCA advances its path to nodes only after finding that the particular node is better than all the possible nodes. Because of this complete exhaustive

search of PCA, it becomes possible to choose the top-k paths out of the all the possible paths. PCA hence provides paths that are optimal. This is hence another improvement of PCA over YA.

6.2.4 Discussion

Through our tests we have seen that PCA beats YA both show a linear increase in time and database I/O as the value of k is increased. We also saw that PCA beats YA by a large margin when it comes the usage of resources. With rising value of k PCA, keeps a constant and low growth late which greatly overshadows YA's performance. We have also seen that PCA has no false hits while calculating the top-k paths out of as many as infinite possible paths between a source and a destination pair in the graph. YA however, has many false hits and alot of effort is wasted on finding deviations paths which may or may not contribute to the final path set. We have seen that in YA to find one more top path, many false hits have to be incurred. Moreover, through PCA we provide the actual top-k *optimal* paths over all the possible paths.

6.3 Skyline Path Search Algorithm

6.3.1 Quantitative Analysis

To test the performance of Skyline Path Search Algorithm, we performed in depth experiments on the same two networks N1 and N2. As we are concerned with multicriteria paths, each edge in the network was assigned 3 additional costs through randomly generated numbers. One other cost utilized was the length of the edge which is the cost that is used in conventional mincost algorithms and our *Passage Counting Algorithm* as well.

Before providing our results there are two important observations worth mentioning that we kept in mind while performing our experiments

- For the skyline path search problem, the number of resulted paths can not be determined beforehand. This means that the number of skyline (non-dominated) paths at the destination varies according to the network topology. The number of paths finally found hence can vary from a minimum of 1 to maximum of as many as all possible paths between the source and the destination node. This is in contrast with the top-k path problem addressed earlier that always has k number of paths in the resulted set.

Range Name	Description
R_s	Paths with 25 or less edges
R_m	Paths with more than 25 and less than 50 edges
R_l	Paths with more than 50 and more edges

Table 6.1: Dataset Ranges based on path length

- It is intuitively to see that nodes that are far away in the network will have more unique paths between them. With more unique paths and each one having a different cost vector, finding dominating paths for longer paths is far more time consuming than for the smaller paths. Path length hence is a direct indicator of problem complexity.

In order to test our algorithm against rising complexity we consider different *average* path lengths to consider a range of cases. Average path length is defined as the average number of edges per each skyline path found between a source and destination. This average was found at the termination of the algorithm by dividing the combined length of all skyline paths with the number of skyline paths found.

Based on the average length of paths we divide the our dataset roughly into three ranges. See Table 6.3.1.

Running Time Analysis

The first experiment was done to test the behaviour of yield time. The yield time here indicates the total time taken by our algorithm to completely explore the search space and finally return the skyline paths set at the destination node.

We tested 3 different sets of criteria (2,3 and 4 number of criterias) for finding paths between different nodes. Refer to Figure 6.9 and Figure 6.10. We can see that the curves for all three criterias, for both networks are 'S' shaped. We can also see in the Figure 6.9 that for the maximum number of criterias (4), the yield time is highest. and with lowest number of criterias(2), the time taken to explore the priority list is smallest.

Moreover, by looking closely we see in the both the figures, yield time curve has 3 parts each corresponding to our ranges define in table 6.3.1. The first part is the for relatively shorter paths (R_s). The second part is the shown by relatively sharper increase in yield time for path lengths having slightly longer lengths (R_m). The third and final part of the curve is for the paths in the R_l range where yield time increases negligible with path lengths. The first trend can be explained by the lower number of possible

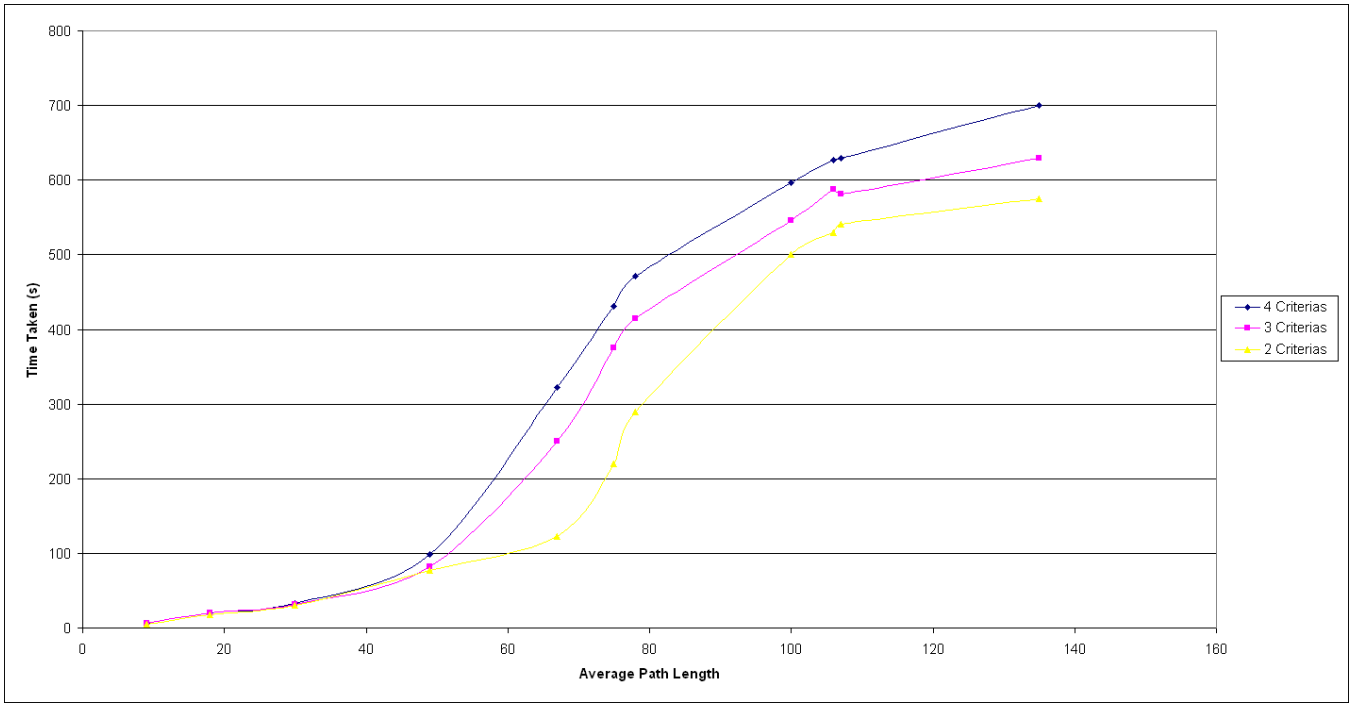

Figure 6.9: Running Time for N1

unique paths from the source to the destination node because of the close proximity of the nodes in the network. The priority queue of the paths hence is explored quickly and the algorithm terminates soon. The sharp increase in the second part of the curve can be explained by the larger number of unique paths from the source to the destination. With a larger queue these intermediate length paths have more time consuming yield times than the smaller counter parts. However, both the first and second parts of the curve are relatively shorter. The third part of the curve is much longer than the earlier two paths. This part of the curve has a reduced gradient than the previous parts which indicates that the yield time does not increase exponentially after the second part ends.

The decrease in the curve's gradient in the third part of the curve for the range R_l is primarily due to the pruning technique we have employed in the algorithm. As a reminder, our algorithm starts pruning the paths in the queue as soon as the first skyline path to the destination as found. This allows us to remove the paths that are in the queue but dominated by the paths found at the destination node. Furthermore, with further exploration, more Skyline paths are found by the algorithm which enhance the pruning ability of the algorthm and we see a lower rise in yield time for paths that are far off. Because there is larger space to explore for paths in the R_l range, this pruning effect becomes much more effective than for paths in R_s and R_m.

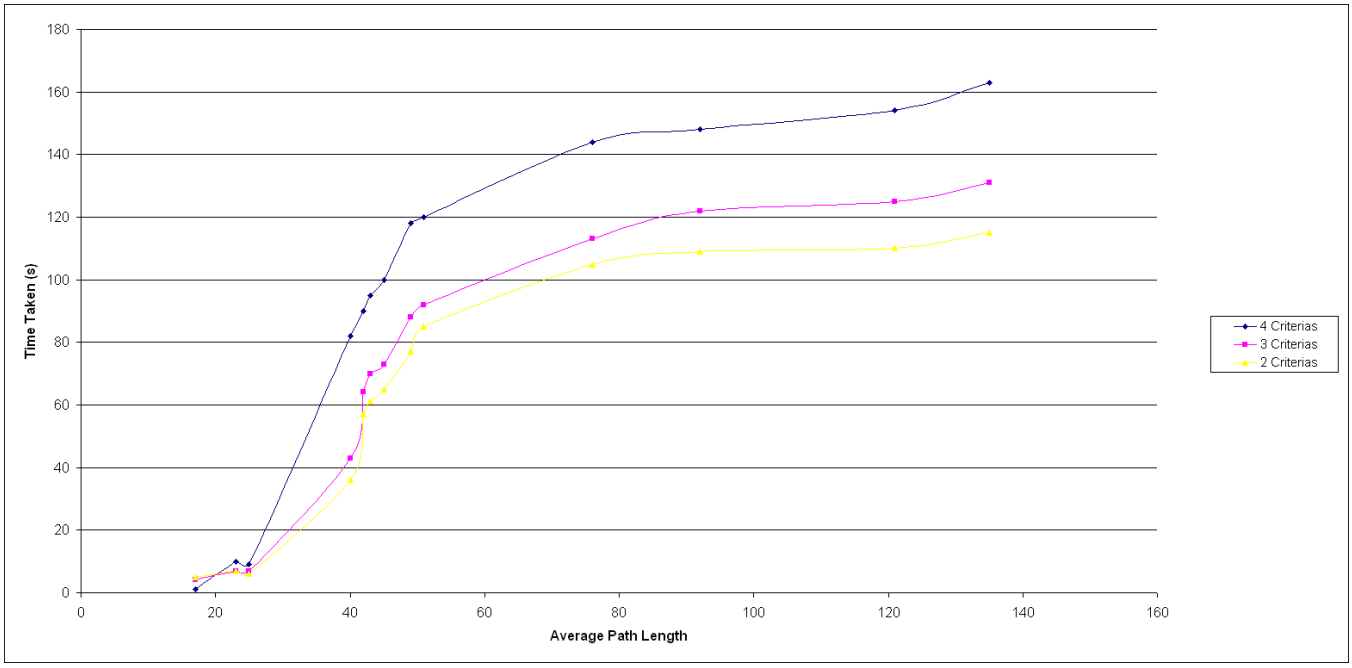

Figure 6.10: Running Time for N2

Database I/O Analysis

We know that the 'goodness' of algroithm with a large dataset can not be judged completely without considering the Data handling calls required by it. In Figure 6.11 and 6.12 we see three curves for N1 and N2 for different scenarios of the experiment. We provide the database access count of the algorithm over the course of its execution for Skyline paths with 2,3 and 4 criteria scenarios. As expected, with more criterias considered for the Skyline path more data retrieval calls are required. The shape of the curve again indicates that 3 different behaviours shown by the yield curves in the past sections. Starting at a lower increase rate for the smaller paths, the database retrieval count indicates the lesser number of calls needed to retrieve adjacent nodes and corresponding edges (R_s). Slightly longer path lengths (R_m) have an increasing rate of retrievals and finally with pruning effect we effectively reduced the number of calls required to find longest paths(R_l). These three trends compliment the Yield time and Database I/O access curves reported in the previous sections.

Skyline Analysis based on Data Set type

We know that while considering multiple types of costs for the network the different costs can have different behaviours compared to each other. Two different costs of an edge can have independent, correlated and anti-correlated relations amongst them. From

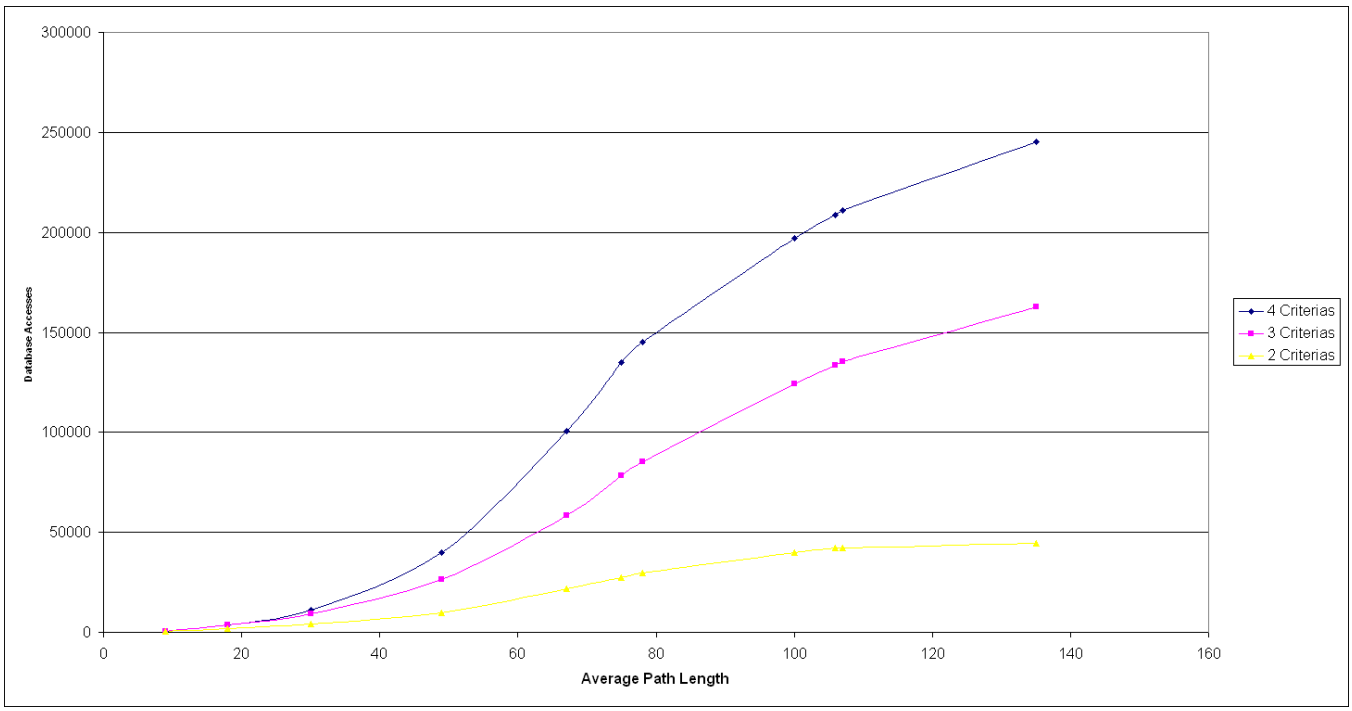

Figure 6.11: Database Accesses for N1

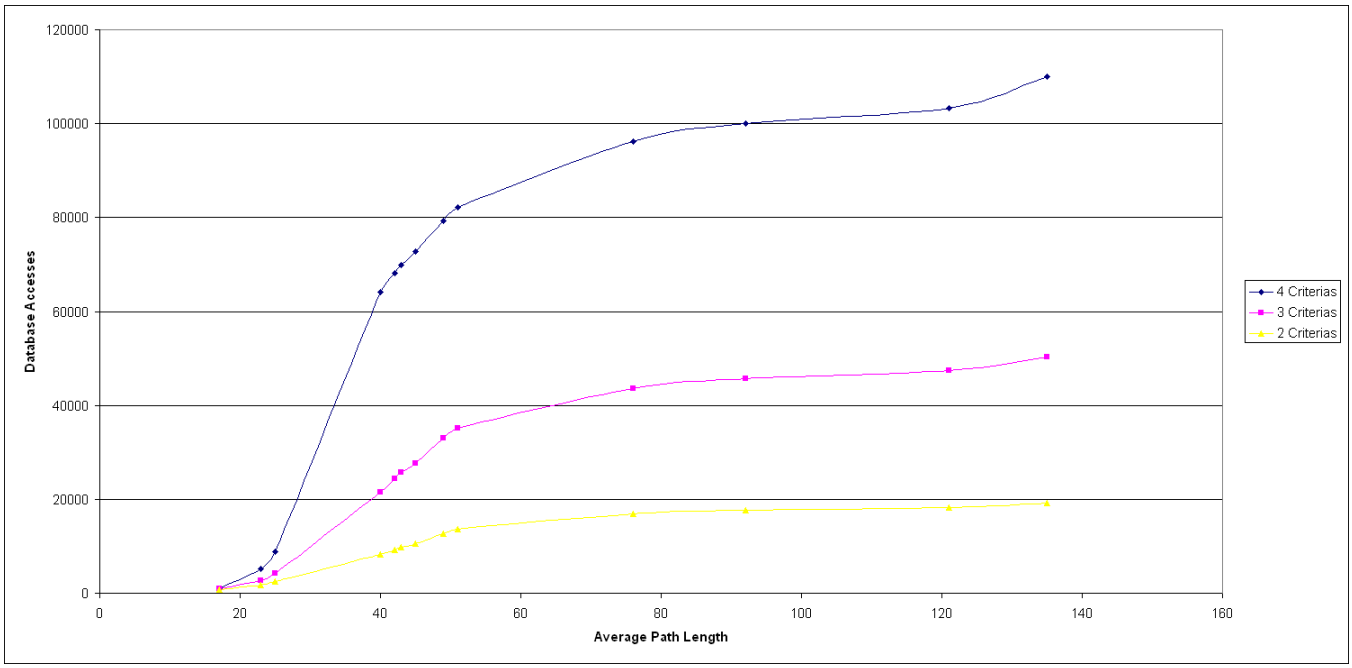

Figure 6.12: Database Accesses for N2

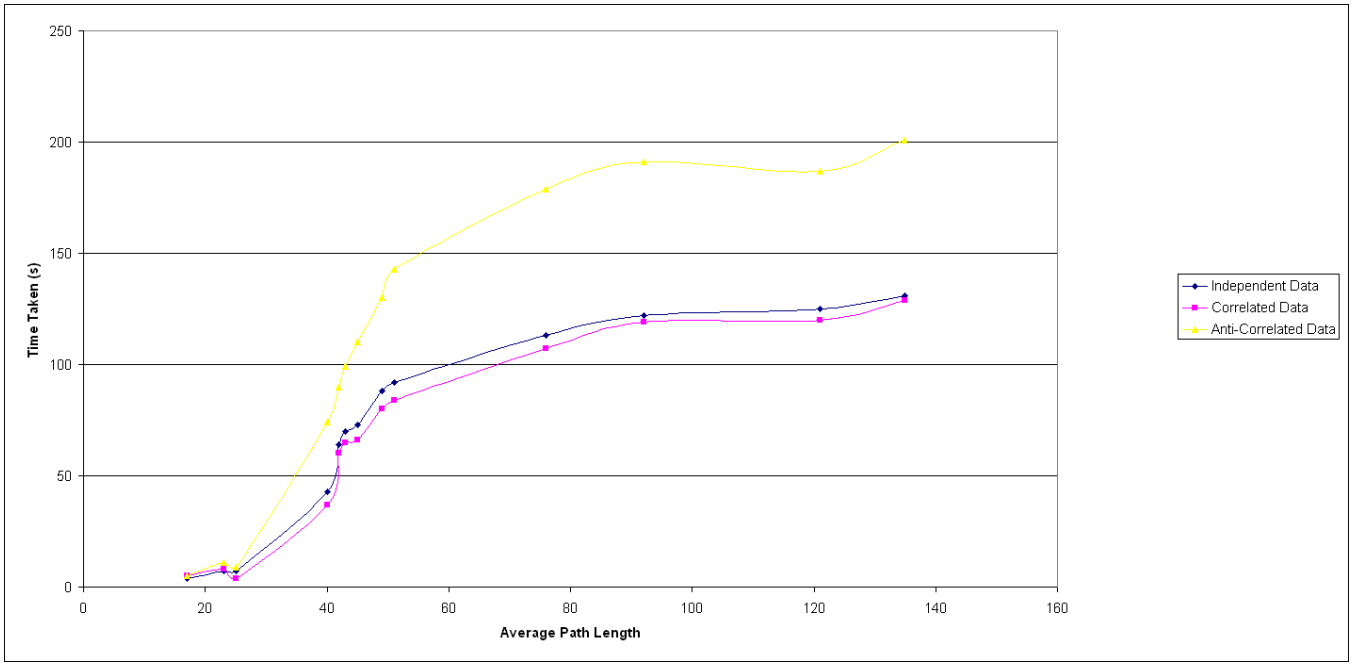

Figure 6.13: Running Time for N2 based on Dataset

conventional Skyline problem we know that the dataset type has a marked effect on the performance of an algorithm. We tested our algorithm with 3 different dataset types by generating independent, correlated and anti-correlated corresponding costs for each edge in the network.

- Two *Independent* costs vary randomly and have no relation to other costs for the edge. For example in the case of road networks the number of traffic signals is independent of how many toll plazas the road has.

- Two *Correlated* costs increase or decrease at the same time e.g. in a longer road will have a higher number of traffic signals and a smaller road will have lesser number of traffic signals.

- Two *Anti-Correlated* costs always move in opposite directions e.g. a road with higher speed limit will have a lower travel time.

Dataset with Anti-correlated data and dataset with correlated data form the two extreme cases of data for the Skyline problem. Intitutively with correlated costs, domination in terms of one costs also signals the domination in the correlated cost. Whereas in case of anti-correlated costs, dominating in terms of one cost will indicate dominance in terms of the other. Independent costs fall in the middle of the two extremes where

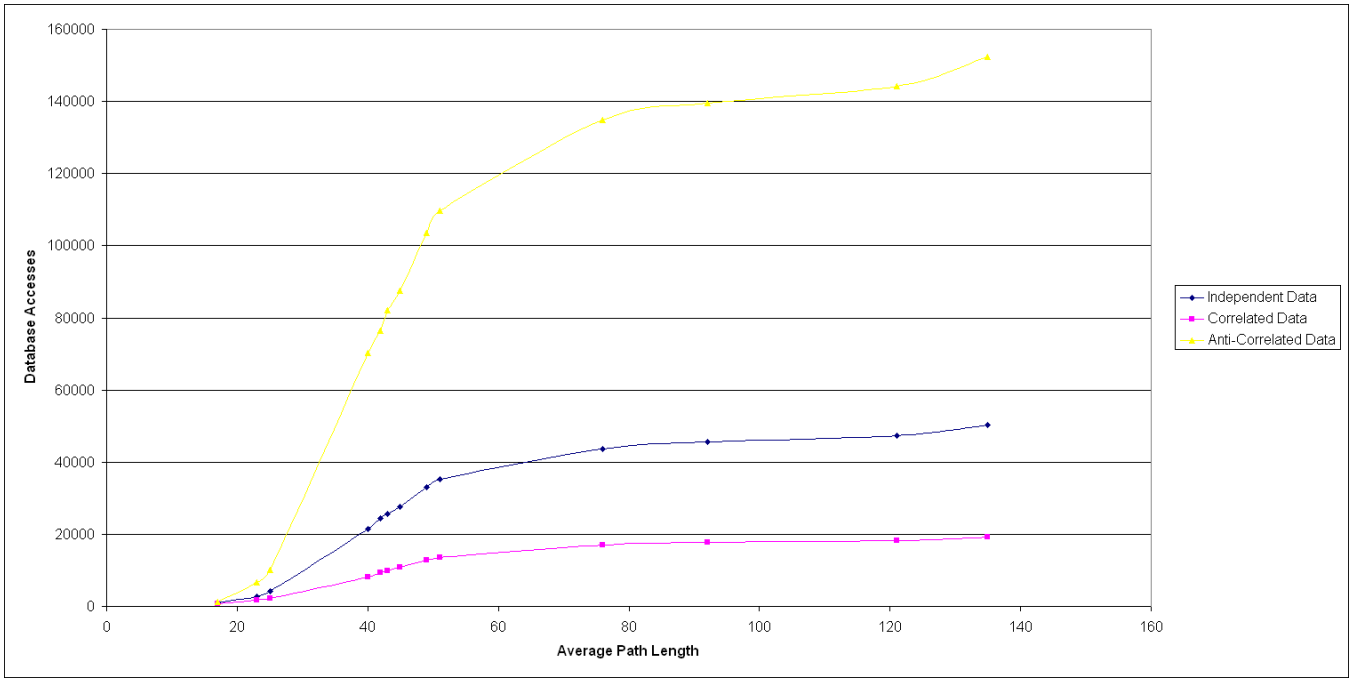

Figure 6.14: Database Accesses for N2 based on Datasets

some costs might show directly proportional data (correlated) and others might show inversly proportional data(anti correlated) data.

Three different experiments w.r.t to each dataset were then performed on network N2. The results are shown in figures 6.13 and 6.14. As expected we can see that in all three experiments, the yield time as well as the database I/O has the more demaning curve for the Anti-Correlated data. Whereas the correlated data because of its simplistic dataset distribution has the minimum yield time as well as database calls. The independent dataset lies in the middle of these first two cases. Generally, the independent dataset is the one which is most representitative of real world roads costs where some costs are directly proportional to each other and others are inversely proportional.

6.3.2 Discussion

We know that finding all possible multicriteria paths between two nodes of a network has to examine an exponentially growing number of paths. We have seen that in all evaluations of the *Skyline Path Search Algorithm* that the curves for yield time and database access all show approximately similar curves as we vary the average path length between the source and destination nodes is increased. All of the yield time and resource usage curves can be divided into three parts depending on the average path length namely the small (R_s), intermediate (R_m) and long(R_l). Smaller paths show smaller yield time

and resource usage whereas the medium length paths expectedly show a more rapid increase as more multi paths between the source and destination exist. We use "within node domination" tests to prune the paths that are dominated. All the paths that are dominated are not processed by the algorithm, reducing the candidate path set to only useful paths.

For paths with longer lengths, the effect of increasing path length is neutralized as second type of pruning starts as soon as we find the first path that reaches the final destination node t. This can be observed by the flattening out of the yield time and resource usage curves for paths longer than medium length. This pruning becomes more effective as more maths reach the destination node, tightening further the "destination domination" test and hence increasing the chances for earlier removal of dominated paths from the candidate set.

As compared to previous works, we provide dominance tests between paths that have the same source and destination. This results in quicker dominance tests as the node labels set are small than all node label sets combined.

We have seen that based on Dataset types, our algorithm behaves exactly like the conventional skyline algorithms. Because the anti-correlated data has inversely proportional dataset, the shape of the distribution curve has a negative gradient ans is similar to the curve for conventional skyline points in any data set(See Appendix A). Because of the similarity of the two curves many more points in the dataset have a probality of lying on the skyline curve. In terms of path search this means that many paths can contribute to the skyline points curve. This makes the case for finding the skyline paths the most demanding of the three dataset cases. We have observed this behaviour in yield time as well as resource curves for anti-correlated data in the previous sections. Taking the case of other extreme i.e. the correlated data, we have observed that the yield time and resource curves are lower than the cases for other two datasets. This indicates that correlated data is the least demanding of the three cases. We can explain this by the distribution curve of correlated data which tends to have a positive gradient which is in contrast with the skyline point curve allowing lesser number of dataset points to become the dominating points. In terms of path search, this means a smaller number dominating paths will eventually lead up to the destination node. With a smaller number of dominating paths, the effort spent is less and hence the curves of correlated data are the lowest ones amongst the three dataset types. Lastly, the independent dataset, can be a mixture of correlated and anti-correlated points and hence produces the middle curve in our path search algorithm.

7. Conclusions and Future work

In this thesis we have presented two new approaches namely, the *Passage Counting Algorithm* and *Skyline Paths Search Algorithm* for efficient mincost paths and dominant paths search respectively between a pair of nodes in any spatial network. *Passage Counting Algorithm* approach is able to provide **multiple mincost paths** instead of just one mincost path. *Skyline Paths Search Algorithm* approach is able to consider many cost funcions at a time to provide **multiple cost paths**. Evaluations show that both of our approaches are able to provide quantitatively and qualitatively better results than previously existing approaches.

Further, we designed and implemented a path recommedation system called *PathAdvice*. This system utilizes the TIGER/Line dataset for maintaining spatial data for road networks in the United States. The prototype system maps both of our new path search approaches to a real world road network scenario where users can provide their current locations and desired destinations along with any other criteria to get possible routes. The system is flexible and can easily incorporate more cost functions, algorithms, road network features as required. This is in contrast with the existing path services available on the world wide web and GPS devices, that only cater single criteria to find a single mincost path.

We have shown with extensive evaluations and performance testing our approach for finding these multiple paths with multiple costs is much more efficient than the previously known approaches. We take real world networks and test them through simulated cost functions and show the enhanced qualitative as well as quantitative improvement of our approached.

Passage Counting Algorithm and *Skyline Paths Search Algorithm* both provide unique paths as results. In future we plan to extend the Passage Count algorithm by introducing an l-unique path restriction on the resulted paths. This restriction will place a limit l on the maximum number of common edges that can exist between any pair of paths from amongst the resulted paths.

Moreover as future work we plan to further improve the functionalities of *PathAdvice*.

Drag and Drop functionalities and the ability to manipulate the map displayed on the screen using mouse pointers are some of these. We also plan to add personalization of the system for different users so each user can maintain a profile within the system and does need to provide information about major source addresses(home or office) or destionations(a cinema, shopping mall) and preferences (distance, cost, gas, time, traffic, speed etc) for each of the search.

Bibliography

[1] Hee-Kap Ahn, Helmut Alt, Tetsuo Asano, Sang Won Bae, Peter Brass, Otfried Cheong, Christian Knauer, Hyeon-Suk Na, Chan-Su Shin, and Alexander Wolff. Constructing optimal highways. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 7–14, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[2] Ravindra K. Ahuja, James B. Orlin, Stefano Pallottino, and Maria G. Scutell. Dynamic shortest paths minimizing travel times and costs. *Networks*, 41:205, 2003.

[3] Oswin Aichholzer, Franz Aurenhammer, and Belén Palop. Quickest paths, straight skeletons, and the city voronoi diagram. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*, pages 151–159, New York, NY, USA, 2002. ACM.

[4] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[5] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[6] A. W. Brander and M. C. Sinclair. A comparative study of k-shortest path algorithms. In *In Proc. of 11th UK Performance Engineering Workshop*, pages 370–379, 1995.

[7] Ismail Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Records*, 1645:170–175, 1998.

[8] Y. L. Chen. Finding the k quickest simple paths in a network. *Inf. Process. Lett.*, 50(2):89–92, 1994.

[9] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 717–816. IEEE Computer Society, 2003.

[10] Brian C. Dean. Shortest paths in fifo time-dependent networks: Theory and algorithms.

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. *In Numerische Mathematik*, 1:269–271, 1959.

[12] Bolin Ding, Jeffrey Xu Yu, and Lu Qin. Finding time-dependent shortest paths over large graphs. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 205–216, New York, NY, USA, 2008. ACM.

[13] David Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[14] B. L. Fox. Calculating kth shortest paths. *INFOR–Canad. J. Op. Res. & Inf. Proc.*, 11:66–70, 1973.

[15] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. Adaptive fastest path computation on a road network: a traffic mining approach. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 794–805. VLDB Endowment, 2007.

[16] F Guerriero, V. Musmanno Lacagnina, and A. Pecorella. An Efficient Node Selection Strategies in Label-Correcting Methods for the K Shortest paths problem. *Report Parcolab,Department of Electronics, Informatics and Systems, University of Calabria*, (2):88–101, June 1996.

[17] Francesca Guerriero and Roberto Musmanno. Parallel asynchronous algorithms for the K shortest paths problem. *J. Optimization Th. & Appl.*, 104(1):91–108, January 2000.

[18] Francesca Guerriero, Roberto Musmanno, Valerio Lacagnina, and Antonio Pecorella. A class of label-correcting methods for the k shortest paths problem. *Oper. Res.*, 49(3):423–429, 2001.

[19] E. Hadjiconstantinou and Nicos Christofides. An efficient implementation of an algorithm for finding K shortest simple paths. *Networks*, 34(2):88–101, September 1999.

[20] Harrison K. J. Hallam, Christina and J. A Ward.

[21] P. E. Hart, Nilsson, N. J, and Raphael B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 1:100–107, 1968.

[22] John Hershberger, Matthew Maxel, and Subhash Suri. Abstract finding the k shortest simple paths: A new algorithm and its implementation, 2003.

[23] Evangelos Kanoulas, Yang Du, Tian Xia, and Donghui Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 10, Washington, DC, USA, 2006. IEEE Computer Society.

[24] N. Katoh, T. Ibaraki, , and H. Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12:411–427, 1982.

[25] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[26] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18:401405, 1972.

[27] Ken C. K. Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. Approaching the skyline in z order. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 279–290. VLDB Endowment, 2007.

[28] Xiaolei Li, Jiawei Han, Jae-Gil Lee, and Hector Gonzalez. Traffic density-based discovery of hot routes in road networks. *Advances in Spatial and Temporal Databases*, pages 441–459, 2007.

[29] Feng Lu and Yanning Guan. An optimum vehicular path solution with multi-heuristics. In *International Conference on Computational Science*, pages 964–971, 2004.

[30] E. Martins, M. Pascoal, , and J. Santos. A new algorithm for ranking loopless paths. *Technical report,Universidade de Coimbra, Portugal*, 1998.

[31] Ernesto Q.V. Martins and Marta M.B. Pascoal. A new implementation of Yens ranking loopless paths algorithm. *Centro de Informtica e Sistemas*, page 121133, 2003.

[32] Giacomo Nannicini, Philippe Baptiste, Gilles Barbier, Daniel Krob, and Leo Liberti. Fast paths in large-scale dynamic road networks. *CoRR*, abs/0704.1068, 2007.

[33] Lars Relund Nielsen, Daniele Pretolani, and Kim Allan Andersen. Finding the k shortest hyperpaths using reoptimization. CORAL Working Papers L-2004-04, University of Aarhus, Aarhus School of Business, Department of Business Studies, November 2004.

[34] Lars Relund Nielsen, Daniele Pretolani, and Kim Allan Andersen. K shortest paths in stochastic time-dependent networks. CORAL Working Papers L-2004-05, University of Aarhus, Aarhus School of Business, Department of Business Studies, November 2004.

[35] Ariel Orda and Raphael Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37:607–625, 1990.

[36] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 467–478, New York, NY, USA, 2003. ACM.

[37] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[38] Marta M. B. Pascoal, Eugénia, and Ao C. Jo˙ An algorithm for ranking quickest simple paths. *Comput. Oper. Res.*, 32(3):509–520, March 2005.

[39] A. Perko. Implementation of algorithms for k shortest loopless paths. *Networks*, 16:149160, 1986.

[40] Daniele Pretolani, Lars Relund Nielsen, and Kim Allan Andersen. A note on multicriteria adaptive paths in stochastic, time-varying networks. CORAL Working Papers L-2006-11, University of Aarhus, Aarhus School of Business, Department of Business Studies, November 2006.

[41] J. Scott Provan. A polynomial-time algorithm to find shortest paths with recourse. *Networks*, 41:223–234, 2003.

[42] J. B. Rosen, S. Z. Sun, and G. L. Xue. Algorithms for the quickest path problem and the enumeration of quickest paths. *Comput. Oper. Res.*, 18(6):579–584, 1991.

[43] Eric Ruppert. Finding the k shortest paths in parallel. *Algorithmica*, 28(2):242–254, October 2000.

[44] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In Gerth Stlting Brodal and Stefano Leonardi, editors, *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[45] Kiseok Sung, Michael G. H. Bell, Myeongki Seong, and Soondal Park. Shortest paths in a network with time-dependent flow speeds. *European Journal of Operational Research*, 121(1):32–39, February 2000.

[46] Yi-Hwa Wu, Harvey J. Miller, and Ming-Chih Hung. A gis-based decision support system for analysis of route choice in congested urban road networks. *Journal of Geographical Systems*, 3(1):3–24, 2001.

[47] Cai X, Kloks T, and Wong CK. Time-varying shortest path problems with constraints. *Networks*, 29:141–9, 1997.

[48] Jin Y. Yen. Finding the K shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

[49] Jin Y. Yen. Finding the lengths of all shortest paths in n -node nonnegative-distance complete networks using 12n3 additions and n3 comparisons. *J. ACM*, 19(3):423–424, 1972.