Constrained Shortest Path Computation[⋆]

Manolis Terrovitis[1], Spiridon Bakiras[2], Dimitris Papadias[2], and Kyriakos Mouratidis[2]

[1] Electrical and Computer Engineering Department,
National Technical University of Athens, Greece
mter@dblab.ece.ntua.gr

[2] Computer Science Department, Hong Kong University of Science and Technology,
Clear Water Bay, Hong Kong
{sbakiras, dimitris, kyriakos}@cs.ust.hk

Abstract. This paper proposes and solves *a-autonomy* and *k-stops* shortest path problems in large spatial databases. Given a source s and a destination d, an a-autonomy query retrieves a sequence of data points connecting s and d, such that the distance between any two consecutive points in the path is not greater than a. A k-stops query retrieves a sequence that contains exactly k intermediate data points. In both cases our aim is to compute the shortest path subject to these constraints. Assuming that the dataset is indexed by a data-partitioning method, the proposed techniques initially compute a sub-optimal path by utilizing the Euclidean distance information provided by the index. The length of the retrieved path is used to prune the search space, filtering out large parts of the input dataset. In a final step, the optimal (a-autonomy or k-stops) path is computed (using only the non-eliminated data points) by an exact algorithm. We discuss several processing methods for both problems, and evaluate their efficiency through extensive experiments.

1 Introduction

Shortest path computation has been studied extensively in graph theory and computer networks, assuming in-memory processing. However, the emergence of time-critical applications that require processing of voluminous spatial datasets necessitates the design of efficient shortest path algorithms for disk-resident data. In this paper we study two variations of the problem, and demonstrate how spatial access methods can be exploited to speed up processing. In particular, we consider the existence of a large collection of data in a Euclidean space, where each point is accessible from any other point in the database, with a cost equal to their distance. In this context, identifying the shortest path between two points is trivial; it is always the straight line connecting them. Nevertheless, real-world applications impose constraints that complicate the computation of the answer. We propose solutions to the following variations of the problem: (i) the *a-autonomy shortest path*, and (ii) the *k-stops shortest path*.

Definition 1 (**a-autonomy path**). *Let DB be a collection of points in the Euclidean space and a be a constant. An a-autonomy path from source s to destination d is a sequence of points (path) $s \rightarrow p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow d$, where each intermediate point*

[⋆] Supported by grant HKUST 6180/03E from Hong Kong RGC.

belongs to DB, and the distance between any two consecutive points in the path is not greater than a.

The parameter a is called the autonomy constraint. Informally, it expresses the maximum distance that one can travel without a stop. Assume, for example, that an airplane must perform a flight from city A to city B, whose distance is D. If a is the autonomy of the plane and $D > a$, which intermediate airports should we choose to use as refueling bases, in order to minimize the overall flight distance? The a-autonomy problem also arises in the area of mobile communications. A signal can be successfully received when the distance between the sender and the receiver is no greater than a constant a. Given a source s and a destination d, we want to determine the intermediate communication centers that a message has to pass from, so that the latency (which is proportional to the overall covered distance) is minimized.

Definition 2 (k-stops path). *Let DB be a collection of points in the Euclidean space and k be a constant. A k-stops path from source s to destination d is a sequence $s \rightarrow p_1 \rightarrow \ldots \rightarrow p_k \rightarrow d$, where each intermediate point belongs to DB, and the number of intermediate points is exactly k.*

As an instance of the k-stops shortest path problem, assume that a delivery vehicle loads some goods at point s, and has to drive to its terminus at point d. In its course it has to deliver the goods to k of the company's customers, where k depends on its cargo capacity. Which k customers should it choose to serve in order to minimize the total traveled distance? The k-stops shortest path is also related to the prize collecting traveling salesman problem [1], where the salesman must choose k out of the total N cities to visit.

To the best of our knowledge, there is no previous work on the aforementioned problems in the context of spatial databases. On the other hand, naïve solutions, such as exhaustive search, are inapplicable to large datasets, due to their prohibitive CPU and I/O cost. In this paper, we propose algorithms for both the a-autonomy and the k-stops shortest paths. Specifically, given s and d, we initially compute an approximate solution (i.e., a sub-optimal path connecting s and d) that satisfies the input constraint. To obtain this approximate answer we design fast heuristics that utilize an existing data-partition index on the input dataset DB. The length of the retrieved path is used to prune the search space, filtering out large parts of the input dataset. Finally, an exact algorithm computes the optimal a-autonomy or k-stops shortest path among the remaining points. The proposed methodology reads only a fraction of DB from the disk, and has very low cost.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the general framework and states our basic pruning criterion. Sections 4 and 5 focus on the a-autonomy and the k-stops problem, respectively. Section 6 experimentally evaluates our techniques with real datasets, and Section 7 concludes the paper with a discussion on future work.

2 Related Work

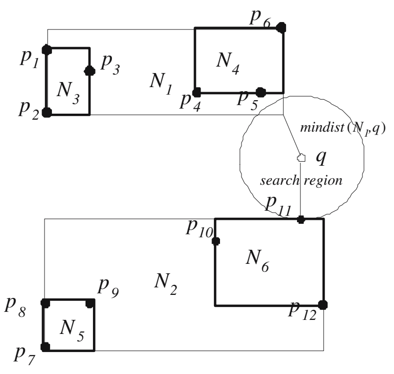
Section 2.1 describes R-trees and algorithms for nearest neighbor (NN) search. Section 2.2 presents existing methods for shortest path computation and related problems.

2.1 R-Trees and Nearest Neighbor Queries

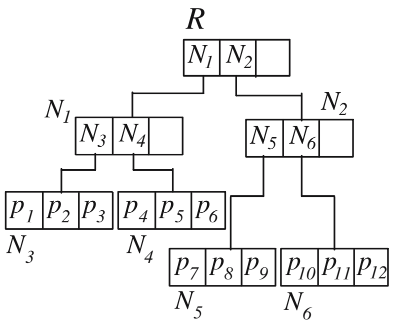
Although our techniques can be used with any data-partition method, here we assume R-trees [2,3] due to their popularity. Figure 1 shows an R-tree for point set $DB = \{p_1, p_2, \ldots, p_{12}\}$ with a capacity of three entries per node. Points that are close in space (e.g., p_1, p_2, p_3) are clustered in the same leaf node (N_3). Nodes are then recursively grouped together with the same principle up to the top level, which consists of a single root. Given a node N and a query point q, the $mindist(N, q)$ corresponds to the closest possible distance between q and any point in the sub-tree of node N. Figure 1(a) shows the $mindist$ between point q and node N_1.

The first NN algorithm for R-trees [4] searches the tree in a depth-first (DF) manner, by recursively visiting the node with the minimum $mindist$ from q. In Figure 1, for example, DF accesses the root, followed by N_1 and N_4, where the first potential nearest neighbor is found (p_5). During backtracking to the upper level (node N_1), the algorithm prunes entries whose $mindist$ is equal to or larger than the distance ($best\_dist$) of the nearest neighbor already retrieved. In the example of Figure 1, after discovering p_5, DF backtracks to the root level (without visiting N_3), and then follows the path N_2, N_6 where the actual NN p_{11} is found.

The DF algorithm is sub-optimal, i.e., it accesses more nodes than necessary. On the other hand, the best-first (BF) algorithm of [5] achieves the optimal I/O performance, visiting only nodes intersecting the circle centered at the query point q with radius equal to the distance between q and its nearest neighbor. These nodes have to be examined anyway in order to avoid false misses. In Figure 1(a), for instance, BF visits only the root, N_1, N_2, and N_6 (whereas DF also visits N_4). BF maintains a heap H with the entries encountered so far, sorted by their $mindist$. Starting from the root, it inserts all the entries into H (together with their $mindist$), e.g., in Figure 1(a), $H = \{< N_1, mindist(N_1, q) >, < N_2, mindist(N_2, q) >\}$. Then, at each step, it visits the node in H with the smallest $mindist$. Continuing the example, the algorithm retrieves the contents of N_1 and inserts all its entries in H, after which $H = \{<$



(a) Points and node extents



(b) The corresponding R-tree

Fig. 1. Example of an R-tree and a NN query

$N_2, mindist(N_2, q) >, < N_4, mindist(N_4, q) >, < N_3, mindist(N_3, q) >\}$. Similarly, the next two nodes accessed are N_2 and N_6 (inserted in H after visiting N_2), in which p_{11} is discovered as the current NN. At this time, the algorithm terminates (with p_{11} as the final result) since the next entry (N_4) in H is farther (from q) than p_{11}. BF (as well as DF) can be easily extended to kNN queries, where k >1. Additionally, BF is incremental, implying that it can output the NNs in ascending order of their distance to the query without a pre-defined termination condition.

An interesting variation of the NN search is the *aggregate nearest neighbor* (ANN) query. Given a set of query points $Q = \{q_1, q_2, \ldots, q_m\}$ and an object p, the aggregate distance $adist(p, Q)$ is defined as a function f over the individual distances $|p, q_i|$ between p and each point $q_i \in Q$. Assuming, for example, n users at locations $q_1, \ldots, q_n$ and $f = sum$, an ANN query outputs the data object p that minimizes $adist(p, Q) = \sum_{q_i \in Q} |p, q_i|$, i.e., the sum of distances that the users have to travel in order to meet at the position of p. Similarly, if $f = max$, the ANN query reports the object p that minimizes the maximum distance that any user has to travel to reach p. In turn, this leads to the earliest time that all users will arrive at the location of p (assuming that they move with the same speed). Finally, if $f = min$, the result is the object p which is closest to any user, i.e., p has the smallest $adist(p, Q) = \min_{q_i \in Q} |p, q_i|$. Assuming that the data set is indexed by an R-tree, the *minimum bounding method* [6] applies best-first NN search, with the difference that each encountered node N is inserted into the heap H with key equal to $f(mindist(N, q_i), mindist(N, q_2), \ldots, mindist(N, q_m))$. We use this technique, as a module of the proposed algorithms, in Sections 4 and 5.

2.2 Shortest Path Computation and Related Problems

To the best of our knowledge, a-autonomy and k-stops shortest paths have not been studied before. On the other hand, there is extensive work on shortest path algorithms for main memory and disk-resident graphs. The most popular algorithm of the former category is proposed by Dijkstra [7]. This technique expands the input graph starting from the source node until it reaches the destination. It uses a priority queue to store the encountered nodes with key equal to their graph distance from the source. In every step, the node with the smallest key is de-queued, and its adjacent (non-visited) nodes are en-queued. The procedure terminates when a complete path (connecting the source and the destination) is found. A* search [8] uses heuristics in order to direct the graph expansion and prune the search space, assuming that the Euclidean distance between two nodes lower bounds their graph distance. The difference from Dijkstra's algorithm is that the key of each en-queued node is the sum of its graph distance from the source and its Euclidean distance from the destination. Other main memory methods include the Bellman-Ford [9,10] and Floyd [11] algorithms.

Shortest path computation techniques for disk-resident data, such as HiTi [12] and HEPV [13], are based on partial materialization. They partition the graph into sub-graphs that fit in memory, and each sub-graph is abstracted as a graph node. The sub-graphs are grouped recursively into higher level nodes, thus forming a hierarchy. All the distances between the sub-graph boundary nodes are computed and stored in the upper level. To answer a shortest path query, the algorithms (i) determine the lowest-level sub-graph containing the source and destination, and (ii) utilize the materialized information along the two search paths to retrieve the result. [14,8] analyze the performance of several secondary memory adaptations of shortest path algorithms.

Papadias et al. [15] propose a storage scheme for large graphs and algorithms for nearest neighbors, range search and distance joins. Their methods combine connectivity and location information about the data objects (indexed by R-trees) to guide the search. Kolahdouzan and Shahabi [16] use the concept of network Voronoi cells and materialization to speed-up query processing. Shahabi et al. [17] find approximate nearest neighbors in road networks by transforming the problem to high dimensional space. Jensen et al. [18] discuss nearest neighbor queries for points moving in a network. Shekhar and Yoo [19] find all the nearest neighbors along a given route. Yiu and Mamoulis [20] study clustering problems in spatial networks.

The most related paper to our work is [21] that uses thematic spatial constraints to restrict the permitted paths (e.g., "find the shortest path that passes only through rural areas"). Although the problem is similar, in the sense that it also deals with constrained shortest path computation in spatial databases, the thematic restrictions are very different from our autonomy and cardinality constraints. Summarizing, all the existing techniques are inapplicable to the proposed problems. In the sequel, we discuss algorithms for a-autonomy and k-stops shortest paths, starting with the general framework.

3 General Framework and Pruning Criterion

The proposed techniques follow the methodology of Figure 2. The first step applies heuristics to efficiently retrieve a path, not necessarily optimal, that satisfies the given constraint (on a or k). The second step uses the length of this path to prune the search space and eliminate the majority of the data points. Finally, the third step computes the actual shortest path (subject to the constraints) using only the non-eliminated points.

```
Algorithm Find Shortest Path
// Input: the source $s$, the destination $d$, the dataset $DB$, the parameter $a$ or $k$
// Output: the constrained (i.e., $a$-autonomy or $k$-stops) shortest path
1. Find a sub-optimal solution with a fast algorithm
2. Use the length of the obtained path to prune parts of the workspace
3. Compute the exact ($a$-autonomy or $k$-stops) shortest path using only the
   non-eliminated data points
```

Fig. 2. The general processing methodology

Whereas steps 1 and 3 are problem-dependent, the pruning criterion is common for both *a-autonomy* and *k-stops* shortest paths. Consider that in Figure 3, we already have a path $s \rightarrow p_1 \rightarrow p_2 \rightarrow d$ with length l satisfying the given constraint (a or k). Our goal is to use this path in order to restrict the search space. For example, point p_3 cannot belong to a better path because $|s, p_3| + |p_3, d| > l$, where $|s, p_3|$ and $|p_3, d|$ are the distances of p_3 from s and d, respectively. In general, any data point p that may be part of a path with length equal to or shorter than l must satisfy the condition $|s, p| + |p, d| \leq l$, i.e., it must lie in the ellipse with foci at points s and d, and sum of

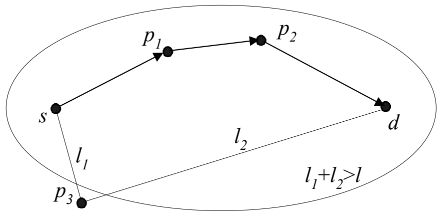


Fig. 3. Pruning example

distances from the foci equal to l. Such points are efficiently retrieved by a range query (in the shape of the ellipse) on the data R-tree. Based on the above, the goal of the first step of our framework is to retrieve a nearly-optimal path, so that the area of the ellipse, and the number of data points therein, is minimized. Then, the third step computes the actual shortest path using only these points. In the following sections we discuss steps 1 and 3 for *a-autonomy* and *k-stops* shortest paths.

4 a-Autonomy Shortest Paths

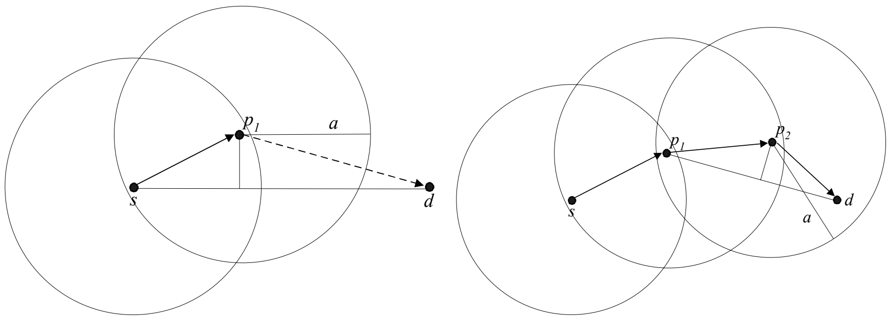
Section 4.1 describes alternative ways to obtain a good approximate solution for the a-autonomy problem, while Section 4.2 deals with the optimal path computation.

4.1 Fast Sub-optimal Path Computation

The shortest route between the source s and the destination d in the Euclidean space is obviously the line segment $\overline{sd}$ connecting them. If the distance $|s, d|$ between s and d is greater than the autonomy of the problem a, we have to introduce intermediate stops. An intuitive strategy for choosing the stops is to select points that cause the least diversion from the optimal route (i.e., the line segment $\overline{sd}$). In particular, the point p in DB that lies closest to $\overline{sd}$ is chosen as a part of the path. The process continues recursively with segments $\overline{sp}$ and $\overline{pd}$ if their individual lengths exceed a. This Least Diversion Method (LDM) terminates when (i) a complete path fulfilling the autonomy constraint is found, or (ii) when all possible solutions are examined. In the latter case, there is no solution for the given value of a and the required path is *infeasible*.

We illustrate the functionality of LDM using the example of Figure 4(a). Initially, LDM retrieves the NN of the line segment $\overline{sd}$, which is point p_1. Assuming that the distance $|s, p_1|$ is less than a, $s \rightarrow p_1$ is accepted as a component of the path. On the other hand, if $|p_1, d|$ is greater than a, the process is repeated for the line segment $\overline{p_1 d}$. Continuing the example, in Figure 4(b) the NN of $\overline{p_1 d}$ is point p_2. Since both distances $|p_1, p_2|$ and $|p_2, d|$ are smaller than the autonomy a, the components $p_1 \rightarrow p_2$ and $p_2 \rightarrow d$ are inserted into the path, and LDM terminates with $s \rightarrow p_1 \rightarrow p_2 \rightarrow d$ as the result.

Figure 5 contains a divide-and-conquer version of LDM. The first call has input parameters s, d, and an empty list $path$. The algorithm recursively examines points according to their distance from $\overline{sd}$. If some point cannot lead to a feasible solution (lines 11, 12), LDM backtracks, and continues with the next NN of the line segment



(a) Retrieval of p_1 (b) Retrieval of p_2

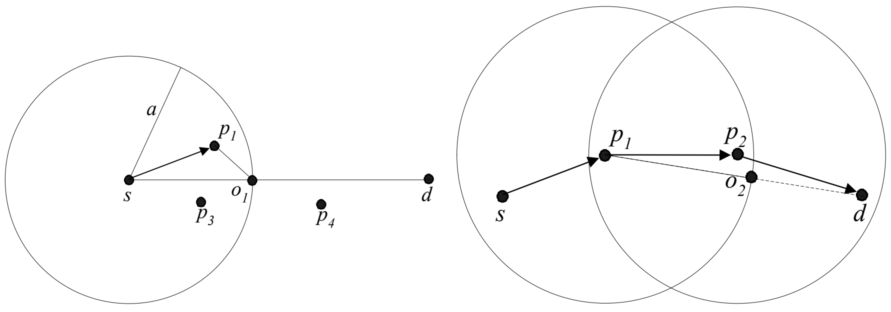
Fig. 4. LDM example

```
Algorithm LDM($p, p', path$)
// $p$ and $p'$ are two intermediate points, and $path$ is the path constructed so far
1.  If $|p, p'| > a$
2.     Find the next nearest neighbor $p_{NN}$ of line segment $\overline{pp'}$ in $DB - \{path\}$
3.     If no $p_{NN}$ is found // i.e., all possible $p_{NN}$ have been unsuccessfully examined
4.        Return false // Backtracking
5.     Else
6.        LDM($p, p_{NN}, path$) // Recursion
7.        LDM($p_{NN}, p', path$)
8.        If both above calls of LDM return true
9.           Add $p_{NN}$ to $path$
10.          Return true
11.       Else // Selection of $p_{NN}$ cannot lead to a valid path
12.          Go to line 2 and continue with the next NN of $\overline{pp'}$
13. Else // i.e., $|p, p'| \leq a$
14.    Add $p$ to $path$
15.    Return true
```

Fig. 5. The least diversion method for the a-autonomy problem

in line 2. Upon termination, if the returned result is *false*, the problem is infeasible. Otherwise, the obtained path is stored in $path$. The nearest neighbor of a line segment (in line 2) is retrieved in a way similar to the best-first NN search discussed in Section 2.1. The difference is that the $mindist$ between the query line segment and an MBR is computed according to the method of [22].

LDM may incur relatively high cost because each NN query (to a line segment) may visit numerous nodes (that intersect, or are near, the segment). Furthermore, since LDM does not aim at minimizing the intermediate points in the path, the number of such queries may be large. Motivated by this we propose a Greedy Heuristic Method (GHM) that (i) applies point (instead of line segment) NN queries and (ii) tries to minimize the number of intermediate points. GHM is based on the observation that an optimal set of intermediate points would lie on $\overline{sd}$, and that the distance between any consecutive



(a) Retrieval of p_1 (b) Retrieval of p_2

Fig. 6. GHM example

```
Algorithm GHM($p, path$)
// $p$ is an intermediate point, and $path$ is the path constructed so far
1.  If $|p, d| > a$
2.      Set $o$ to be the point in the line segment $\overline{pd}$ with distance from $p$ equal to $a$
3.      Find the next NN of $o$ ($p_{NN}$) in the circle with center at $p$ and radius $a$
4.      If no $p_{NN}$ is found // i.e., all possible $p_{NN}$ have been unsuccessfully examined
5.          Return false // Backtracking
6.      Else
7.          GHM($p_{NN}, path$) // Recursion
8.          If the above call of GHM returns true
9.              Add $p_{NN}$ to $path$
10.             Return true
11.         Else // Selection of $p_{NN}$ cannot lead to a valid path
12.             Go to line 3 and continue with the next NN of $o$
13. Else // i.e., $|p, p'| \leq a$
14.     Add $p$ to $path$
15.     Return true
```

Fig. 7. The greedy heuristic method

pair would be equal to the autonomy a. Since such points do not necessarily exist in the database, it tries to use their NNs.

Figure 6(a) illustrates GHM with an example. Ideally, the first intermediate point o_1 would lie on the line segment $\overline{sd}$ at distance a from s. GHM retrieves the NN p_1 of o_1 among the points that are directly reachable by s (i.e., the points falling in the circle centered at s with radius a) and inserts it into the path. To compute the second intermediate point, it determines the ideal point o_2 that lies on $\overline{p_1 d}$ at distance a from p_1 (see Figure 6(b)). Then, it retrieves the NN of o_2 (i.e., p_2) among the points that are directly reachable from p_1, and inserts it into the path. The distance $|p_2, d|$ is smaller than a, and GHM terminates with $s \to p_1 \to p_2 \to d$ as the result.

Figure 7 shows the pseudo-code of GHM. The NN computation in line 3 is an instance of a constrained NN query [23], since the retrieved points must fall in a specified region (i.e., the circle centered at p with radius a). The search algorithm follows the best-first paradigm and, thus, it is incremental. In particular, it inserts into a search heap only R-tree entries N where $mindist(N, p) \leq a$, with sorting key $mindist(N, o)$. It reports only points p_{NN} with $|p, p_{NN}| \leq a$ in ascending order of their distance from o.

Note that the paths obtained by LDM and GHM are possibly different and sub-optimal. For instance, LDM first exploits paths containing the first NN p of the segment $\overline{sd}$, but the best path does not necessarily contain this point (even if there is a path passing from p that satisfies the autonomy constraint). Similarly, in Figure 6(a), GHM will not discover the best path $s \rightarrow p_3 \rightarrow p_4 \rightarrow d$ because p_3 is not the NN of o_1. Therefore, both LDM and GHM only constitute fast filter steps before the exact computation, which is discussed next.

4.2 Optimal Path Computation

After obtaining a sub-optimal path, we perform a query on the R-tree to retrieve the data points that may lead to better solutions. As discussed in Section 3, if l is the length of the sub-optimal path (returned by LDM or GHM), potential candidates lie in the ellipse defined by points s, d, and the value of l. Assume that the corresponding set of points is $DB_{eps} \subseteq DB$. To identify the optimal path, we process DB_{eps} with a modified version of A* search, which takes into account the autonomy constraint and the approximate solution available. In order to apply the algorithm, we consider that the retrieved points form a graph, such that (i) two points (nodes) are connected, if their Euclidean distance

```
Algorithm **Optimal Path Computation**
// $s$: source point, $d$: destination point, $a$: autonomy
// $DB_{eps}$: the subset of $DB$ after the pruning step
1.  Initialize a min-priority queue $Q$
2.  Insert $s$ into $Q$ with key $dist(s) = |s, d|$
3.  While $Q$ is not empty
4.    Get the next entry $< e, dist(e) >$ in $Q$
5.    If $e \neq d$ // expand the graph around point $e$
6.      For each point $p$ inside the circle centered at $e$ with radius $a$
7.        If $dist(e) - |e, d| + |e, p| + |p, d| \geq l$
8.          Go back to line 6 and continue with the next point
9.        If $p$ has not been de-queued before
10.         If $p$ is not currently in $Q$ // i.e., $p$ is visited for the first time
11.           En-queue $< p, dist(e) - |e, d| + |e, p| + |p, d| >$ in $Q$
12.         Else // i.e., $p$ was visited before and it is contained in $Q$
13.           Let $dist(p)$ be the key of $p$ in $Q$
14.           If $dist(p) > dist(e) - |e, d| + |e, p| + |p, d|$
15.             Update the key of $p$ in $Q$ to be $dist(e) - |e, d| + |e, p| + |p, d|$
16.   Else // i.e., $e = d$
17.     Return the corresponding path as the result, and terminate
18. Return the path at hand as the result // i.e., no better path was found
```

Fig. 8. The optimal a-autonomy path computation over the un-pruned part of the dataset

does not exceed the autonomy a, (ii) the cost of an edge connecting two points equals their distance.

Figure 8 illustrates the pseudo-code for the optimal path computation module. Line 6 guarantees that the path returned is valid by considering as reachable only points within distance a from the de-queued entry. The knowledge of a path with length l is used to reduce the search space in line 7. Note that the pruning condition $(dist(e) - |e, d| + |e, p| + |p, d| \geq l)$ for a considered point p also takes into account $|p, d|$, which constitutes a lower bound for the length of the shortest path between p and d, in accordance with the A* algorithm.

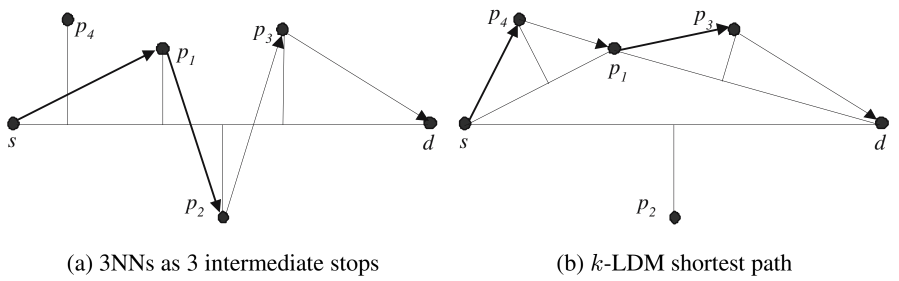
5 k-Stops Shortest Paths

Section 5.1 presents two heuristics for the efficient retrieval of sub-optimal k-stops paths. Section 5.2 describes an algorithm for computing the optimal answer.

5.1 Fast Sub-optimal Path Computation

A naïve heuristic for computing a good initial path is to select the k closest points to the line segment that connects s and d. In certain cases, however, this may lead to a poor solution. Consider, for instance, that in Figure 9(a) we want to compute the shortest path that passes through three intermediate stops. The four NNs of $\overline{sd}$ are p_1, p_2, p_3, p_4 (in this order). The path $s \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow d$ containing the first 3NNs is relatively long since p_2 is on the opposite side (of $\overline{sd}$) with respect to p_1 and p_3. In order to avoid this problem we follow the least diversion paradigm. Figure 9(b) illustrates the adaptation of LDM (called k-LDM) to the k-stops problem on the example of Figure 9(a). First, k-LDM adds to the path the NN (p_1) of line segment $\overline{sd}$. Then, it retrieves the point with the minimum distance from line segments $\overline{sp_1}$ and $\overline{p_1d}$. Among the NNs (p_3 and p_4) of $\overline{sp_1}$ and $\overline{p_1d}$, p_3 is inserted into the path. The process is repeated for the NN of $\overline{sp_1}$, $\overline{p_1p_3}$, $\overline{p_3d}$, and LDM terminates with $s \rightarrow p_4 \rightarrow p_1 \rightarrow p_3 \rightarrow d$ as the result.

The k-LDM algorithm is illustrated in Figure 10. It is worth mentioning that the best NN computation in line 3 is not performed by individual NN queries for each edge of the path, because this approach would lead to multiple traversals of the R-tree of DB.

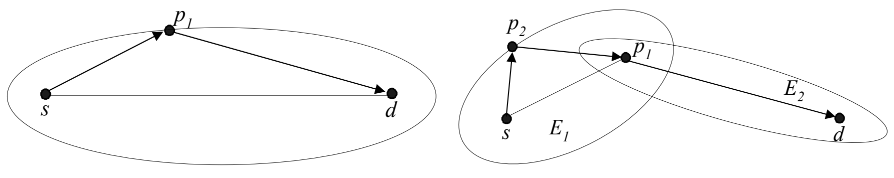


(a) 3NNs as 3 intermediate stops (b) k-LDM shortest path

Fig. 9. k-LDM motivation and example

```
Algorithm $k$-LDM$(s, d)$
// $s$ and $d$ are the source and destination, and $path$ is the path constructed so far
1. Initialize $path$ to $s \rightarrow d$
2. For $i = 1$ to $k$
3.     Find the point $p$ in $DB$ with the min distance from any line segment in $path$
4.     Add $p$ to $path$
5. Return $path$
```

Fig. 10. The least diversion method for the k-stops problem



(a) Retrieval of p_1 (b) Retrieval of p_2

Fig. 11. LOM example

To compute an intermediate point with a single traversal, line 3 is implemented using an adaptation of the aggregate nearest neighbor search discussed in Section 2.1. In particular, the query set Q now consists of the edges in the path constructed so far, and the aggregate distance of a point is defined as the distance from its closest line segment in Q. The algorithm follows the best-first paradigm, by inserting each encountered node N in the R-tree of DB into the search heap H with key equal to $\min_{p \rightarrow p' \in path} mindist(N, \overline{pp'})$, i.e., the minimum $mindist$ between N and any of the edges in the path.

An alternative to k-LDM is the Local Optimum Method (LOM). Given any pair of points p and p', if we want to select one intermediate point $o \in DB$ so that the length of the path $p \rightarrow o \rightarrow p'$ is minimized, then o is by definition the point that minimizes the sum of distances from p and p' (i.e., $|p, o| + |o, p'|$). Based on this observation, given a path with fewer than k points, LOM chooses as the next point o the one that minimizes the sum of distances from any pair of consecutive points in the current path. In other words, it selects $o \in \{DB - path\}$ that minimizes $\min_{p \rightarrow p' \in path} |p, o| + |o, p'|$. Figure 11 gives an example for a 2-stops query. The first intermediate point is p_1, since it minimizes the sum of distances from s and d. Geometrically, this implies that the ellipse in Figure 11(a) does not contain any other point. The second intermediate point p_2 is computed as the point in $DB - \{s, d, p_1\}$ that minimizes the quantity $\min(|s, p_2| + |p_2, p_1|, |p_1, p_2| + |p_2, d|)$. Consequently, in Figure 11(b) the ellipse E_1 does not contain any point other than p_2, while ellipse E_2 (defined by foci p_1 and d, and length $|p_1, p_2| + |p_2, d|$) is empty.

Figure 12 shows the LOM algorithm. Similar to the implementation of k-LDM, step 3 is performed with an ANN algorithm, in order to avoid multiple traversals of the R-tree of DB. The query set Q contains all the edges in the current path, and the aggregate distance of a point is defined as the minimum sum of distances from the

```
Algorithm LOM($s, d$)
// $s$ and $d$ are the source and destination, and $path$ is the path constructed so far
1. Initialize $path$ to $s \rightarrow d$
2. For $i = 1$ to $k$
3.     Find the point $p$ in $DB$ with the min sum of distances from the endpoints
4.     of any line segment in $path$
5.     Add $p$ to $path$
6. Return $path$
```

Fig. 12. The local optimum heuristic method for the k-stops problem

endpoints of any line segment in Q. The ANN search traverses the R-tree of DB in a best-first manner, inserting each encountered node N into the search heap H with key equal to $\min_{p \rightarrow p' \in path} mindist(N, p) + mindist(N, p')$, i.e., the minimum sum of $mindist$ between N and the endpoints of any of the edges in the path.

5.2 Optimal Path Computation

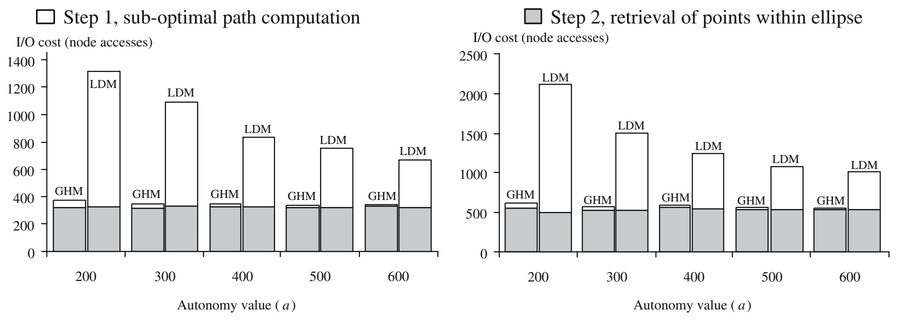
The optimal path computation involves an implementation of Bellman-Ford's algorithm. This algorithm works iteratively, and calculates the shortest paths in increasing number of hops. In particular, during step i it computes the shortest paths consisting of exactly i hops between the source node and every other node of the graph. The complexity of Bellman-Ford's algorithm is $O(mE)$ for a graph with E edges and a maximum path length of m hops, rendering it very expensive for dense graphs. Notice that in the k-stops problem formulation there is no autonomy constraint, and therefore, the number of edges is $O(N^2)$ (where N is the number of points after the pruning step). Consequently, the running time of the algorithm is expected to be $O(kN^2)$, and a good first solution is crucial for achieving low cost.

6 Experimental Evaluation

In this section we experimentally evaluate the performance of our methods, in terms of I/O and CPU cost. We use the two real spatial datasets TCB and LA (available at *www.rtreeportal.org*), containing 450K and 1.3M points, respectively. Both datasets are normalized to fit in a [0, 10000]^2 workspace. The block size of the R-trees is set to 2 KBytes. For each simulation, we select two random points from the dataset and compute the constrained shortest path using the proposed methods. In order to reduce randomness, each result is obtained by averaging over the measurements of 10 simulations. For all experiments we use a Pentium 3.2 GHz CPU with 1 GByte memory. Section 6.1 focuses on the a-autonomy problem, while Section 6.2 on k-stops shortest paths.

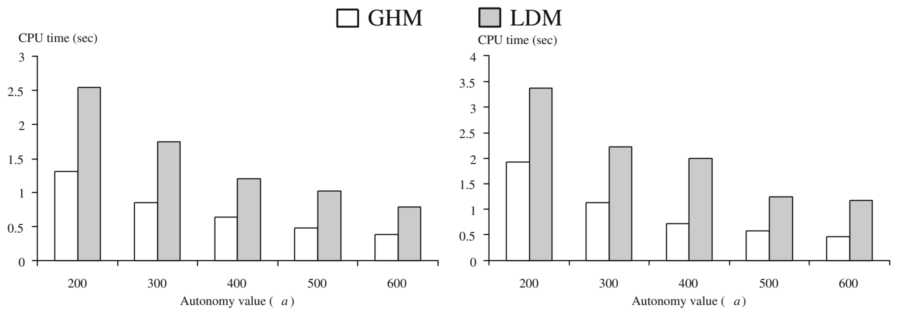
6.1 Evaluation of a-Autonomy

We first study the effect of the autonomy value a using the GHM and LDM heuristics. We fix the distance between the source and destination points to 3000 and vary a from



(a) TCB (b) LA

Fig. 13. Total I/O cost vs. autonomy a



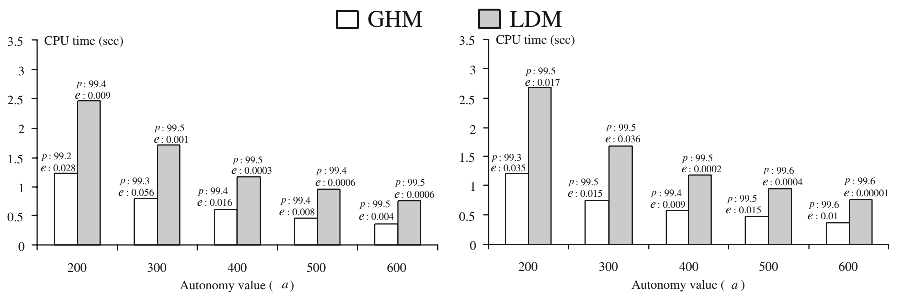
(a) TCB (b) LA

Fig. 14. Total CPU time vs. autonomy a

200 to 600. Figure 13 shows the total I/O cost for datasets TCB and LA, and its break-down into the sub-optimal path computation (step 1) and the elliptical range search (step 2) that collects the points passing the criterion of Section 3. As a increases, the cost drops because the path consists of fewer intermediate points. Regarding the final range search, the methods incur similar overhead. On the other hand, obtaining the initial path with GHM incurs significantly fewer node accesses compared to LDM. This happens because GHM performs (cheap) point NN queries, as opposed to the (expensive) linear NN queries of LDM. Furthermore, since GHM aims at reaching the destination with the minimum number of steps, it performs fewer NN searches than LDM.

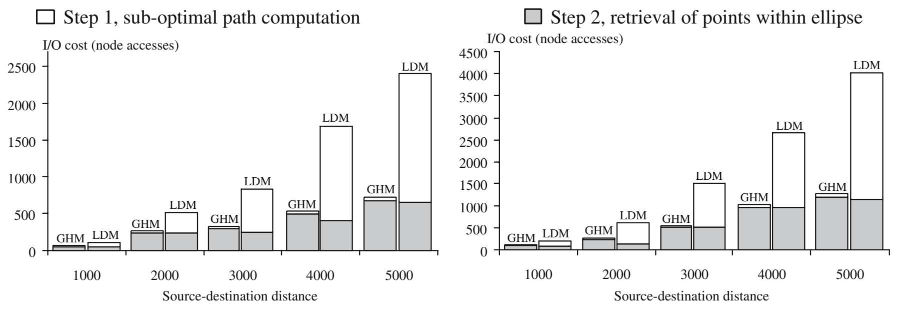
Figure 14 depicts the total CPU time for the previous experiment. The running time decreases with a because both the sub-optimal and the optimal path consist of fewer points. The performance gain of GHM is similar to the I/O gain for the reasons explained in the context of Figure 13. An important remark concerning both methods is that the initial path computation dominates the total CPU time because, as discussed shortly the number of non-eliminated points (that participate in the selection of the optimal path) is small.

We now present some interesting measurements regarding the cost and accuracy of the sub-optimal path computation. Figure 15 depicts the CPU time for calculating



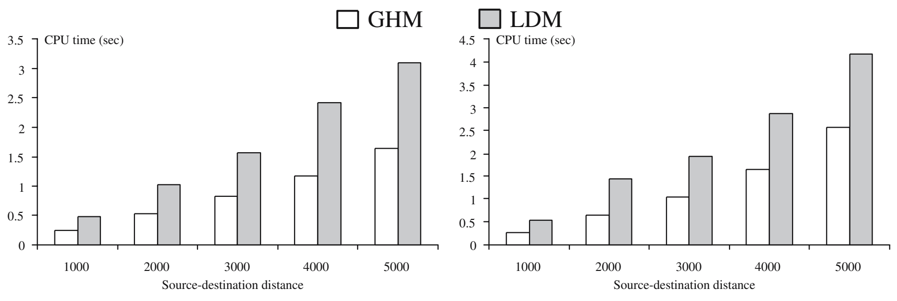
(a) TCB (b) LA

Fig. 15. CPU time and quality of the sub-optimal path vs. autonomy a



(a) TCB (b) LA

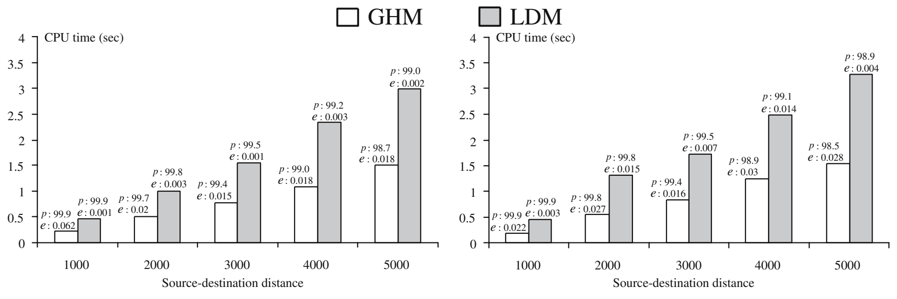
Fig. 16. Total I/O cost vs. distance of endpoints



(a) TCB (b) LA

Fig. 17. Total CPU time vs. distance of endpoints

the initial path, the deviation percentage e of the achieved path length compared to the optimal one, and the percentage p of the dataset that is pruned according to the criterion of Section 3. LDM provides a better quality path than GHM, at the expense of higher CPU cost. Both methods, however, produce a very accurate result (with less than 0.04% deviation from the optimal path length) and are able to prune over 99% of the database.



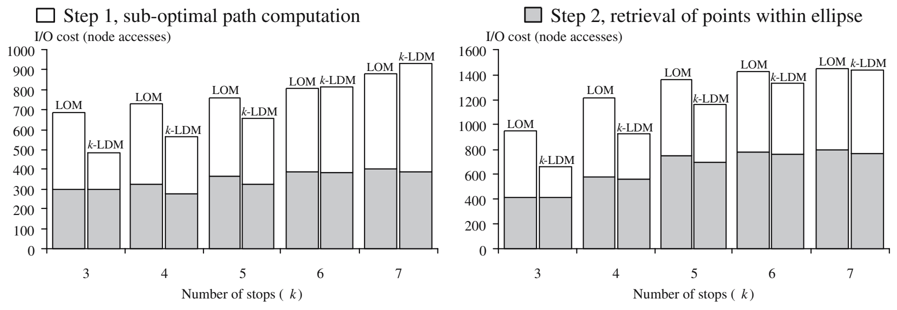
(a) TCB (b) LA

Fig. 18. CPU time and quality of the sub-optimal path vs. distance of endpoints

Next, we investigate the effect of the distance between the source and destination. We set the autonomy variable a to 300, and vary the (Euclidean) distance between the endpoints of the path from 1000 to 5000. The total number of node accesses and the overall CPU time are illustrated in Figures 16 and 17, respectively. As expected, a larger distance implies higher I/O and CPU cost. LDM is affected more because of the numerous linear NN queries. Figure 18 focuses on the sub-optimal path computation step. As the distance between the endpoints of the path increases, the pruning percentage p decreases, since the ellipse of the criterion of Section 3 grows. On the other hand, the deviation e from the optimal solution is very small in all cases.

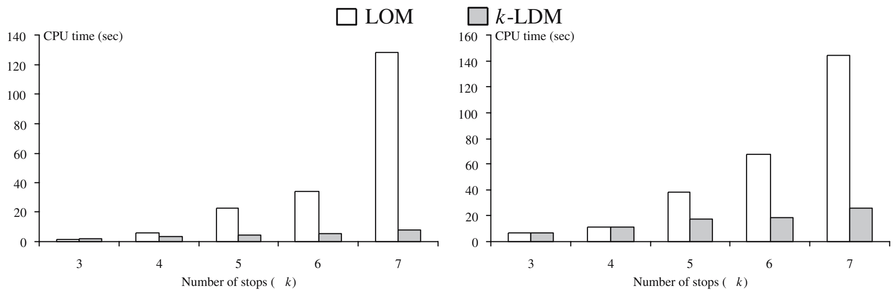
6.2 Evaluation of k-Stops

In the following experiments we evaluate the performance of the k-stops shortest path algorithms. First, we study the effect of k on the LOM and k-LDM techniques. We fix the distance between the source and destination points to 3000 and vary the number k of intermediate stops from 3 to 7. Figure 19 shows the total I/O cost, and its breakdown into the initial sub-optimal path computation and the retrieval of points passing the pruning criterion. The node accesses of LOM remain relatively stable, while for k-



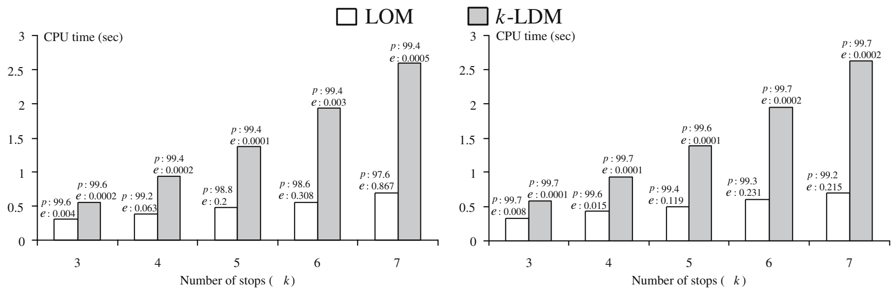
(a) TCB (b) LA

Fig. 19. Total I/O cost vs. number k of required stops



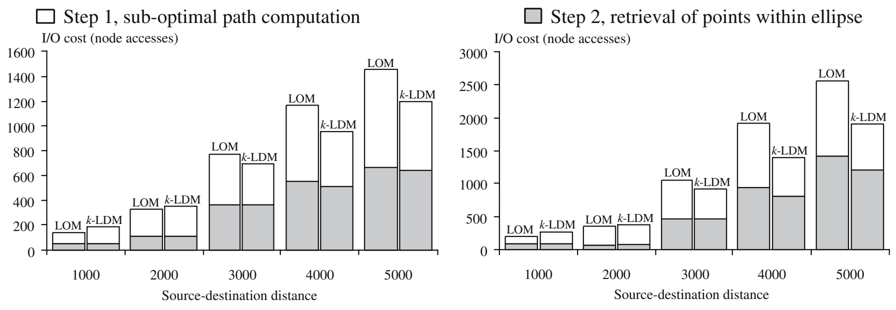
(a) TCB (b) LA

Fig. 20. Total CPU time vs. number k of required stops



(a) TCB (b) LA

Fig. 21. CPU time and quality of the sub-optimal path computation vs. number k of required stops

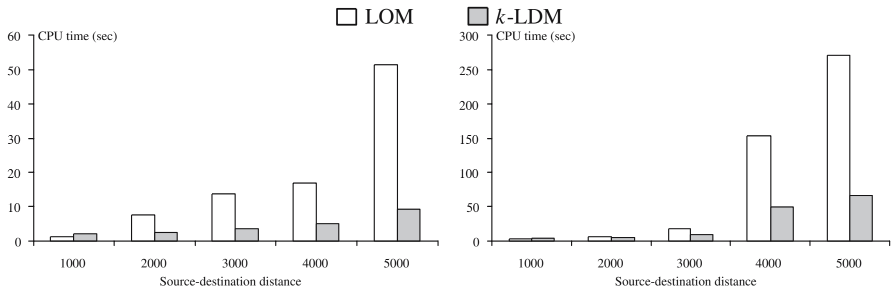


(a) TCB (b) LA

Fig. 22. Total I/O cost vs. distance of endpoints

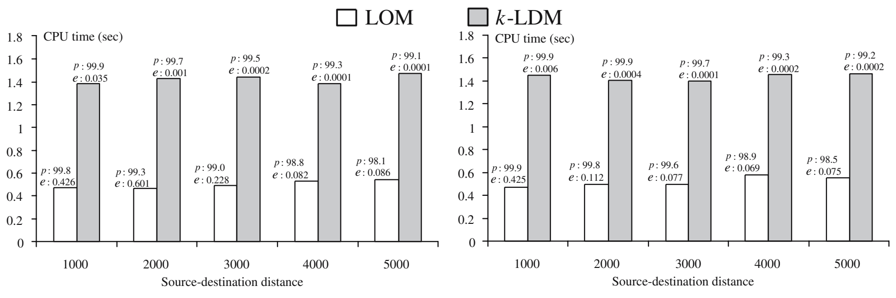
LDM they increase linearly with k. Initially, k-LDM incurs fewer node accesses than LOM, but this changes for larger values of k. Regarding the elliptical range search for points passing the pruning criterion, its I/O cost is similar for both methods.

Figure 20 illustrates the total running time for the previous experiment. k-LDM is considerably faster than LOM, and the performance gap increases sharply with k. This



(a) TCB (b) LA

Fig. 23. Total CPU time vs. distance of endpoints



(a) TCB (b) LA

Fig. 24. CPU time and quality of the sub-optimal path vs. distance of endpoints

fact indicates that the k-LDM technique retrieves a better sub-optimal solution, and thus achieves more effective pruning than LOM. Recall that the time complexity of the optimal path computation is $O(kN^2)$, where N is the number of points that pass the criterion of Section 3. It follows that even a slightly worse initial solution can increase significantly the overall running time. Regarding the detailed behavior of each method, the CPU time for LOM is dominated by the optimal path computation, while for k-LDM approximately 10%-50% of the CPU time is spent on the initial path computation.

Figure 21 verifies the above observation, by comparing the accuracy and the CPU time of the sub-optimal path computation for k-LDM and LOM. The path returned by k-LDM is at most 0.003% longer than the optimal one in all cases. LOM is not as effective, especially for large values of k. Its solution deviates from the optimal by up to 0.86%. On the other hand, LOM is considerably faster than k-LDM, and it is a good choice for providing approximate results to time-critical applications that can tolerate a certain amount of inaccuracy in return for a fast response.

Finally, in Figures 22, 23 and 24 we investigate the impact of the distance between the two endpoints of the path. We set the number k of stops to 5, and vary the distance between the source and destination from 1000 to 5000. As expected, when the distance increases, both the I/O and CPU costs are higher. The reason is that the ellipse covers a larger area of the workspace, and prunes fewer nodes and points (also verified by the pruning percentages p in Figure 24). Furthermore, the I/O cost increases because the edges of the path are longer, and the ANN queries of both k-LDM and LOM access a larger part of the index. As shown in Figure 24, the running time of the heuristics remains relatively stable because the required number of intermediate points is constant (i.e., k).

7 Conclusion

This paper formulates and solves a-autonomy and k-stops shortest path queries, in spatial databases. Assuming a large collection of points in the Euclidean space indexed by a data-partitioning access method, we propose several techniques for the efficient computation of the constrained shortest paths. Our methods exploit the spatial information provided by the index, in order to produce very fast an initial sub-optimal path. The length of this path is then used to prune the workspace, according to a geometric criterion. The optimal path is retrieved by utilizing an exact shortest path algorithm on the non-eliminated data points. Our experimental results on real spatial datasets demonstrate that the proposed techniques are able to prune over 98% of the database for all examined settings, thus leading to very low response times.

A promising direction for future work concerns a top-K version of the constrained shortest path problem, where instead of a single path, we are asked to compute the best K paths according to some input constraint (e.g., a-autonomy or k-stops). Furthermore, in this paper we consider that all points are equivalent. It would be interesting to study cases where the data points have different properties. For instance, in autonomy problems it may be beneficial to visit a point that incurs relatively high diversion, if it can provide a large benefit (e.g., in terms of refueling capacity).

References

1. Balas, E.: The prize collecting travelling salesman problem. Networks **19** (1989) 621–636
2. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: SIGMOD. (1990) 322–331
3. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD. (1984) 47–57
4. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD. (1995) 71–79
5. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM TODS **24** (1999) 265–318
6. Papadias, D., Tao, Y., Mouratidis, K., Hui, C.: Aggregate nearest neighbor queries in spatial databases. ACM TODS (2005, to appear)
7. Dijkstra, E.: A note on two problems in connection with graphs. Numerische Mathematik **1** (1959) 269–271
8. Shekhar, S., Kohli, A., Coyle, M.: Path computation algorithms for advanced traveller information system (ATIS). In: ICDE. (1993) 31–39
9. Bellman, R.: On a routing problem. Quarterly of Applied Mathematics **16** (1958) 87–90
10. Ford, L., Fukelson, D.: Flows in networks. Princeton University Press (1962)
11. Floyd, R.: Shortest path. Communications of the ACM **5** (1962) 345

12. Jung, S., Pramanik, S.: HiTi graph model of topographical roadmaps in navigation systems. In: ICDE. (1996) 76–84
13. Jing, N., Huang, Y., Rundensteiner, E.A.: Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. IEEE TKDE **10** (1998) 409–432
14. Jiang, B.: I/O-efficiency of shortest path algorithms: An analysis. In: ICDE. (1992) 12–19
15. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: VLDB. (2003) 802–813
16. Kolahdouzan, M.R., Shahabi, C.: Voronoi-based K nearest neighbor search for spatial network databases. In: VLDB. (2004) 840–851
17. Shahabi, C., Kolahdouzan, M.R., Sharifzadeh, M.: A road network embedding technique for k-nearest neighbor search in moving object databases. In: GIS. (2002) 94–100
18. Jensen, C.S., Kolárvr, J., Pedersen, T.B., Timko, I.: Nearest neighbor queries in road networks. In: GIS. (2003) 1–8
19. Shekhar, S., Yoo, J.S.: Processing in-route nearest neighbor queries: a comparison of alternative approaches. In: GIS. (2003) 9–16
20. Yiu, M.L., Mamoulis, N.: Clustering objects on a spatial network. In: SIGMOD. (2004) 443–454
21. Huang, Y., Jing, N., Rundensteiner, E.A.: Integrated query processing strategies for spatial path queries. In: ICDE. (1997) 477–486
22. Tao, Y., Papadias, D., Shen, Q.: Continuous nearest neighbor search. In: VLDB. (2002) 287–298
23. Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A.: Constrained nearest neighbor queries. In: SSTD. (2001) 257–278