Shortest Path Computation on Air Indexes

Georgios Kellaris Kyriakos Mouratidis

Singapore Management University
80 Stamford Road
Singapore 178902
{gkellaris, kyriakos}@smu.edu.sg

ABSTRACT

Shortest path computation is one of the most common queries in location-based services that involve transportation networks. Motivated by scalability challenges faced in the mobile network industry, we propose adopting the *wireless broadcast model* for such location-dependent applications. In this model the data are continuously transmitted on the air, while clients listen to the broadcast and process their queries locally. Although spatial problems have been considered in this environment, there exists no study on shortest path queries in road networks. We develop the first framework to compute shortest paths on the air, and demonstrate the practicality and efficiency of our techniques through experiments with real road networks and actual device specifications.

1. INTRODUCTION

Greater productivity, more convenient communication and everyday need for data-on-demand are just some of the reasons that mobile devices are becoming increasingly popular. The vast majority of these devices is equipped with positioning systems, which gave rise to an expanding industry of location-based services. Users of these devices can easily request for the nearest business or service to their location, navigate to a target address, locate personal contacts on a map or receive alerts such as warnings of traffic jams.

To answer a location-based query, such as navigation to a specific address, the device typically either (i) pre-loads the map data and runs the query locally, or (ii) it connects to a location server through a GSM/3G/Wi-Fi service provider. The option of storing the map information locally imposes heavy requirements on the already limited storage of the mobile device and it is viable only for few pre-selected maps. In case the user travels to a new city/country, this option would fail. Moreover, storing maps locally may result in routing decisions based on outdated network information.

On the other hand, given the growing number of users and services, the option of querying online location servers is already facing scalability limitations, a situation expected to worsen in the near future. While coping with increasing query loads necessitates continuous infrastructure upgrades at the side of service providers, a greater challenge is network congestion. According to [13] the number of mobile subscribers has risen by 250% in the last 3.5 years, while data traffic volume from handheld devices is growing by more than 10 times per year. The main concern of mobile service providers in the Mobile World Congress 2010 [1] was that the number of data-capable phones is growing faster than network capacity, so network overload is considered an immediate risk, and data traffic management is becoming a major priority for the telecommunications industry.

A promising solution to the above problem is the wireless broadcast model [6]. In this model the location server repeatedly broadcasts the data on the air (using GSM, 3G, Wi-Fi, HD radio, or even a Bluetooth network), while the clients tune in the broadcast channel and process their queries locally. Since the server's hardware requirements are low, multiple servers could be installed at different locations to provide coverage in large areas. The main advantage of the broadcast model is that it can support an arbitrary number of users/queries, since no processing takes place at the server and the network overhead is irrelevant to the number of clients. A side benefit is that user privacy is guaranteed, as the location server is unaware of user positions and queries; this has been a serious concern recently [2].

Wireless broadcasting has been considered for spatial processing in Euclidean space (e.g., [12, 16, 17]). However, there is currently no work on road networks. Motivated by the fact that in most location-based applications movement is constrained by a transportation network, in this paper we develop broadcasting schemes for network data. In particular, we study *shortest path computation*, the most common query in road networks. Our contributions can be summarized as follows:

1. We adapt traditional shortest path algorithms to the broadcast model, and identify their weaknesses in this setting.

2. We present two novel methods, namely *Elliptic Boundary* (EB) and *Next Region* (NR), which exploit the broadcast environment's characteristics and take into account the technical limitations of mobile devices.

3. We demonstrate the efficiency of our schemes through extensive experiments with various road networks and with real-world device specifications.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

2. RELATED WORK

2.1 Road Networks and Shortest Path Query

A road network is a directed weighted graph $G = (V, E)$. V is the set of nodes v_i, each of the form $< id_i, x_i, y_i >$, denoting the identifier and the coordinates of the node. E is the set of edges (v_i, v_j), each modeled as a triplet $< id_i, id_j, w_{ij} >$ that contains the identifiers of the connected nodes v_i and v_j and the weight w_{ij} of the edge; w_{ij} may correspond to the edge's length, travel time, toll fee, etc.

The shortest path between a source node v_s and a target node v_t is the edge sequence connecting v_s and v_t with the minimum sum of edge weights. Algorithms for shortest path queries are categorized as follows: (a) those with no pre-computation, where only the road network information is available, and (b) those with pre-computation, where shortest paths between some or all node pairs are pre-calculated and appropriate information is materialized in order to speed up the search.

Without pre-computation: A common algorithm is Dijkstra's [3]. Initially, nodes adjacent to v_s are pushed into a min-heap with their graph weights from v_s as sorting keys. The top node v in the heap is popped in every iteration and expanded, i.e., its adjacent nodes v' are en-heaped with key equal to that of v plus the weight of edge (v, v'). The process stops when v_t is popped. The shortest path is returned by tracing backwards the expansions that lead to v_t.

A* search [5] improves on Dijkstra's algorithm but requires a lower bound LB(v, v_t) to be known for the graph distance between an encountered node v and the target node v_t. The difference from Dijkstra is that the key of each en-heaped node v is increased by LB(v, v_t). We ignore A* in the following since we assume general road networks (where no a priori lower bounds exist).

With pre-computation: *ArcFlag* [10] first partitions the network nodes. A bit vector (flag) is assigned to every edge, where each bit corresponds to a partition; in the flag of edge (v_i, v_j) the bit for a partition is 1 if there is at least one node v in the partition where the shortest path from v_i to v traverses (v_i, v_j). Search (e.g., Dijkstra) only considers edges whose bit for v_t's partition is 1.

Landmark [4] chooses some anchor nodes (called landmarks) and pre-computes for each node v its graph distances to all anchor nodes. A distance vector is then created from the distances to the anchor nodes. From the distance vectors of two nodes, a lower bound can be derived for their graph distance. This bound is then used by A* algorithm to guide the search.

In *HiTi* [9] the graph is partitioned by a grid of cells. The resulting sub-graphs are recursively grouped into higher level sub-graphs forming a tree. Every node that has at least one adjacent node that lies in a different partition is defined as *border node*. For each level, the shortest paths among all border nodes are pre-computed and stored in the HiTi index. A shortest path query is answered by first selecting the necessary sub-graphs in the HiTi hierarchy, and then performing a search (e.g., Dijkstra) inside them only. *HEPV* [7] works similarly to HiTi.

The *shortest path quad-tree scheme* (SPQ) [14] stores for each node v a colored quad-tree, built on the Euclidean coordinates of the other graph nodes. The nodes v' for which the shortest path (from v) passes through the same incident edge of v are assigned the same color. Search first traverses the edge (v_s, v) that corresponds to the color of v_t in the quad-tree of v_s. The process is repeated for v and its quad-tree, and so on, until v_t is reached.

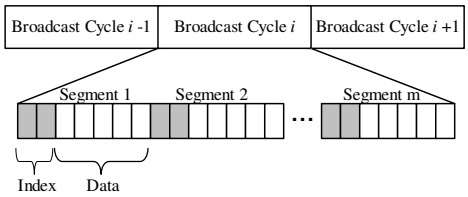


Figure 1: (1,m) scheme

2.2 Wireless Broadcasting and Air Indexes

In this model the server repeatedly transmits identical *broadcast cycles*, each containing the entire database and potentially some indexing information (called *air index*). The broadcast cycle consists of fixed-size *packets*, defining the smallest information unit transmitted.

The most common organization of the broadcast cycle is the (1,m) *interleaving scheme* [6], exemplified in Figure 1; the data tuples are placed into m equi-sized *data segments* interleaved by m copies of the index (e.g., a B-tree). To process a query, the client tunes in the broadcast channel and waits until the next index is broadcast; the larger m is, the shorter the wait for the index. The client receives the index, performs its point/range selection, and then waits until the data segments that contain the result tuples are broadcast; the larger m is, the longer the wait for the data. The optimal balance between the wait for index and the wait for data, is achieved for $m = \sqrt{\frac{data\_packets}{index\_packets}}$. During any wait, the client sleeps (i.e., stops listening to the channel), thus preserving energy.

The performance factors considered in this setting are *tuning time* and *access latency*. Tuning time is the number of packets necessary for query processing, which essentially determines the energy consumption at the client (see Section 3.1 for this rationale, supported by current device specifications). Access latency is the time (expressed in number of packets) elapsed between the user request and the time when all result tuples are received.

[6] considers point and range selections, and hence B-tree air indexes. [16, 17] address range and nearest neighbor queries on spatial data (in Euclidean space), based on space-filling curves. [12] addresses the same queries, relying on an implicit grid partitioning. None of the above applies to our problem, as they cannot capture the constraints imposed by a road network, and they rely on Euclidean concepts (space-filling curves, rectangles, and circles) impossible, or at least non-trivial, to interpret into the network distance space. A detailed review of the above methods can be found in Appendix A.

3. PRELIMINARIES

3.1 Performance Factors

The main performance factors involved in our problem are: (i) tuning time, (ii) memory requirements (at the client side), (iii) access latency, (iv) CPU time (at the client side), and (v) pre-computation time (at the server side).

Power consumption is essentially determined by tuning

time (i.e., number of packets received). To exemplify the relation, consider that the (widely used) 802.11 WaveLAN card consumes 1.65W, 1.4W, and 0.045W in transmit, receive, and sleep states, respectively [8]. Computations (i.e., the CPU) also consume power, but their effect is outweighed by communication; almost 98% of the market's mobile devices are integrated with an Advanced RISC Machines (ARM) processor whose very characteristic is energy preservation (with a typical peak consumption of 200mW).

Mobile devices have much smaller RAM memory than desktop or laptop computers. More importantly, only a fraction of this memory is available to applications; space is primarily reserved for the operating system and user data, while application development platforms, like the ubiquitous Java 2 Micro Edition (J2ME), impose a further limit on the working memory (termed *heap size*). As we show in Section 7 based on actual device specifications, memory requirements are a major concern in our problem, yielding several methods inapplicable to large or even moderately sized road networks.

Access latency relates to the responsiveness of the system. Technical limitations aside, the time between asking a query and receiving the shortest path determines the user experience; it is essential for the success of the model that this time is within acceptable limits. As absolute time depends on the network speed, access latency is measured in number of packets between query posing and the last packet received for shortest path computation.

Another performance factor is CPU time, i.e., the time taken by calculations at the client side. As explained above, calculations do not significantly affect power consumption. Also, CPU time is negligible compared to access latency (in absolute terms) for real network speeds, as shown in the experiments. It remains, however, a necessary consideration.

The last factor pertaining to the efficiency of a solution is pre-computation time. This is the time required to form the broadcast cycle, including possible shortest distance/path pre-calculations. Although it is a one-off cost, and therefore not one of our main design criteria, it must be reasonable.

Among the above, our design decisions take into consideration primarily the first three performance factors, although the last two are not ignored either.

3.2 Adaptation of Existing Approaches

In this section we attempt to adapt existing shortest path algorithms to the broadcast setting, and identify drawbacks thereof. Assume for simplicity that the user location (i.e., the source of the shortest path) and her destination are located at two network nodes v_s and v_t, as opposed to some arbitrary location on an edge. Consider first Dijkstra's algorithm. It does not require any pre-computation, and therefore the broadcast cycle includes only the road network information, i.e., the adjacency lists of all nodes. Assume that the client knows at which node it is located at (i.e., it knows v_s) and that it somehow knows when to wake up in order to receive the adjacency information for a specific node. The device should first wake up to receive the adjacency information of v_s and en-heap all its adjacent nodes. Letting v be the node at the head of the heap, Dijkstra next needs to listen to v's adjacency list. As this information may have already been broadcast, the device might need to wait for the next broadcast cycle. The access latency of this approach is unacceptable, as the device may have to wait for as many cycles as the number of nodes popped by Dijkstra. Since there is no way to tell in advance what the source and the destination of user queries will be, it is not possible to organize the nodes in the broadcast cycle in a certain order so that every needed node appears later than the previously deheaped one. The situation could be improved if the network was partitioned into regions and the adjacency lists for the same region were broadcast together; still, however, when Dijkstra search reaches the borders of the current region, the device might have to wait for the next cycle in order to receive network information about the neighboring regions.

Due to the above problem we abandon the idea of selective tuning in Dijkstra's adaptation. To achieve reasonable access latency, the client could listen to the *entire* broadcast cycle, and then process the query locally in the complete network. Access latency this way never exceeds one cycle. Particularly, since the broadcast cycle is the shortest possible (as no indexing/extra information is broadcast), this method is expected to be very competitive in terms of access latency. On the other hand, it has very large memory requirements (equal to the size of the entire network) and incurs long tuning time (all packets of the cycle are received).

ArcFlag, Landmark, and SPQ would broadcast for each edge/node (in addition to its adjacency information) a bit vector, a distance vector and a colored quad-tree, respectively. All three would face the same issues with selective tuning as Dijkstra: information of next edge/node to visit may have already been transmitted, requiring wait for the next cycle and causing unacceptable delays. Again, somehow partitioning the network would not cure the problem. The only viable option is that the device listens to the entire cycle and performs processing in the entire network. Compared to Dijkstra's adaptation, these three methods would be inferior according to all performance factors in Section 3.1 (since now the broadcast cycle includes extra information and, thus, is longer), except CPU time at the client.

HiTi is the only approach that could effectively achieve selective tuning (thus reducing tuning time), since it uses an index structure to determine the needed regions of the network in advance. For this pruning of the search space to be possible, however, the client should receive the entire index. As we show in the experiments, the index size can be several times larger than the actual network, due to the numerous pre-computed distances stored. This leads to a long broadcast cycle (and thus access latency), large tuning time, and high memory requirements at the client. Note that even for moderately sized networks, the index cannot fit in the heap size of the mobile device used in Section 7.

4. ELLIPTIC BOUNDARY (EB) METHOD

Intuitively, in order to efficiently process shortest path queries we have to partition the road network into regions and use an index structure to guide the search through them. To satisfy the requirements in Section 3.1, the index should be particularly concise, much more so than existing indexes designed for disk-resident networks. The *Elliptic Boundary* method (EB) follows this approach.

Its crux is to first provide the client with an upper bound of the shortest path distance between v_s and v_t. This bound is used to prune (i.e., to avoid listening to) network information about nodes that lie too far away from v_s and v_t to affect the shortest path search. This is achieved by partitioning the network into regions, and placing in the index

of EB information about the minimum and maximum possible distance from any partition to any other. EB owes its name to the fact that the search area is reminiscent of a network-based ellipse with foci the regions of v_s and v_t.[1]

4.1 Index of EB

The index of EB includes two components. The first defines partitions and provides a mapping of nodes into regions; the second specifies minimum/maximum distances between regions.

To commence query processing, the client must first identify the regions R_s and R_t of the source v_s and destination v_t, respectively. This process is bound to the partitioning method used. A straightforward approach is to superimpose a Euclidean regular grid (of equi-sized rectangular cells) over the network, and consider the part of the network inside each cell as a region. In this approach the client could trivially map the Euclidean coordinates of v_s and v_t to regions R_s and R_t, requiring only knowledge of the grid granularity (e.g., $k \times m$ cell partitioning), and of the total spatial extent of the grid. The drawback of this approach is that some regions would contain too few nodes (or be empty), while others would be too full. This would reduce the benefits of partitioning and impede the search.

As shown in [11], a simple, yet very effective partitioning method (in terms of facilitating shortest path search) is kd-tree partitioning. To illustrate, consider the example in Figure 2. Initially, the network is divided into two regions by a straight line parallel to x-axis. In our case, this line is $y = 10$ (see number next to horizontal line), corresponding to the median y coordinate of all network nodes. Each of the resulting two regions is divided by a line parallel to y-axis, corresponding to the median x coordinate of their contained nodes; for the upper region this line is $x = 9$ and for the lower it is $x = 11$ (values 9 and 11 are illustrated next to the corresponding vertical lines). The process continues recursively, alternating between the two axes, until a desired number of partitions (i.e., kd-tree leaves) is reached. In Figure 2 and for the case of 16 regions, the kd-tree shown on the right implicitly defines the partitions created; the value in each node corresponds to the x or y value used for splitting the corresponding region into its children.

In the kd-tree case, the client needs more information to identify R_s and R_t than with a regular grid. Specifically, it needs to reconstruct the tree in Figure 2. To achieve this, the first component of the EB index includes the splitting values of the kd-tree, transmitted in breadth-first order; this information suffices to implicitly define the partitions. In our example, the first component of the index is sequence $< 10, 9, 11, 16, 15, 7, 6, 5, 4, 12, 13, 7, 8, 14, 15 >$. In the general case, if there are n partitions, the sequence includes $n-1$ values. Note that this is preferable over explicitly transmitting the x and y extent of each region (that would require $4n$ values). Another remark is that kd-tree splitting is also used to implicitly enumerate (name) the regions; we establish the convention that the leftmost region of the leftmost leaf is R_1, and increment the region number for its sibling. We apply this rule to the second leaf node (R_3, R_4), etc, and derive the region numbers shown in the figure.

As mentioned before, in order to allow selective tuning,

[1]We stress that the search space is by no means an ellipse or any other geometric shape, as we assume no relationship between network distance and Euclidean distance.

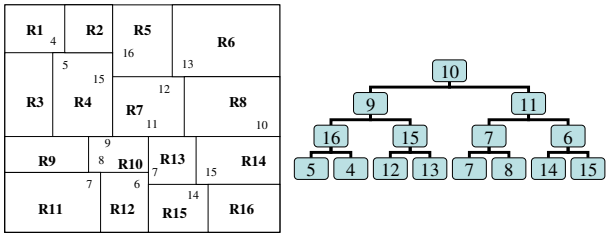


Figure 2: Kd-tree

the second component of the EB index provides the minimum and maximum distances between every pair of regions. This component is essentially an $n \times n$ array A, where n is the total number of partitions. Each row represents a potential source region and each column a destination region. Every cell A_{R_i,R_j} contains two values; the minimum and the maximum distance from *any* border node in R_i to *any* border node in R_j (recall that border nodes are those whose at least one adjacent node lies in a different partition). To create array A, EB needs to pre-compute the shortest path distances for all possible pairs of border nodes from different regions (we show experimentally that this pre-computation cost is manageable even for very large networks).

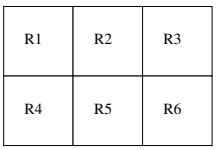
The two components described above constitute the index of EB. The network information (i.e., the adjacency lists of nodes) need also be included in the broadcast cycle; the adjacency lists of nodes that belong to the same region are placed contiguously in the cycle, while region ordering abides by the assigned region numbers. For the client to be able to receive the node information for a specific region R_i, the index is appended with pointers to the data of each region. Essentially, a column is appended to array A, providing the offset (in number of packets) where the region data for the corresponding row will start being transmitted.

To keep the access latency low, we replicate the index m times in the broadcast cycle, following the (1,m) scheme and setting m according to the analysis in [6]. However, we force the index copies to appear *between regions*, as opposed to fixed, evenly spaced intervals, so that adjacency data for the same region are not cut in by index packets. Every packet, regardless of its contents, includes a pointer (offset) to the next copy of the index in the broadcast cycle.

The region data received can be reduced based on a classification of the network nodes. Let S be the set of precomputed shortest paths, i.e., paths between any pair of border nodes that belong to different regions. Let R be a region to be received, other than R_s and R_t. We observe that the shortest path from v_s to v_t may only pass via nodes $v \in R$ that appear in at least one path in S. We call nodes that appear in S *cross-border*, and the rest *local*. Each region's data are divided into the cross-border segment and the local segment, so that if $R \notin \{R_s, R_t\}$ the client may listen only to the former and ignore the latter. Note that both segments are needed for R_s and R_t, as local nodes may appear in the path from v_s to a border node of R_s, or from a border node of R_t to v_t. In our experiments, this optimization reduces tuning time by around 20%.

4.2 Client-side Processing in EB

Posed a query, the client tunes in the broadcast channel, and listens to the current packet. It retrieves the pointer to the next index and sleeps; it wakes up when the index starts



	R1		R2		R3		R4		R5		R6	
	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
R1			1	5	6	8	1	4	3	7	8	9
R2	1	5			1	3	2	5	1	2	2	4
R3	4	6	1	3			5	8	2	4	1	4
R4	1	3	2	4	5	8			2	3	4	7
R5	3	6	1	3	2	4	1	3			1	2
R6	7	9	2	4	1	2	5	6	1	3		

Figure 3: EB method's index

being broadcast, and receives it in its entirety. Regions R_s and R_t are determined as described in Section 4.1. The second component of the index (array A) is then used to derive an upper bound UB for the shortest path distance from v_s to v_t; the maximum value stored in A_{R_i,R_j} serves as UB. The correctness of this upper bound can be easily seen, since any path from source to destination has to pass through at least one border node of each R_s and R_t.

The next step is to determine which regions must be received. Based again on A, the client needs to listen to only those regions R for which $mindist(R_s, R)+mindist(R, R_t) \leq UB$, i.e., the sum of minimum distance from R_s to R plus the minimum distance from R to R_t is no larger than UB. Upon deciding which regions are necessary, it sleeps and wakes up when their data (contained nodes and adjacency lists thereof) are broadcast. When all necessary regions are broadcast and received, the client performs a Dijkstra search in their union (a sub-graph of the network) and reports the computed shortest path; this is guaranteed to be the correct answer in the entire network. Observe that access latency does not exceed one broadcast cycle.

In Figure 3, the source v_s is in region R_1 and the destination v_t in region R_5. The maximum shortest path distance is $UB = 7$ (see A_{R_1,R_5}). Hence, the client need only receive the regions of the source and destination, plus R_2 ($1+1 < 7$) and R_4 ($1 + 2 < 7$). Region R_3 and region R_6 are not necessary since the sum of their minimum distances is larger than UB ($6 + 2$ and $8 + 1$, respectively). The pseudo-code of the EB method can be found in Appendix B.

5. NEXT REGION (NR) METHOD

While EB allows for selective tuning, its search space (i.e., the set of regions received by the client) may still be large. The problem is exacerbated when the source and destination are far away, as EB's network-based ellipse includes an increasing number of regions. In the extreme case where v_s and v_t are located in the furthest regions, it is possible that EB needs to receive all regions. Note that in this degenerate case, performance may actually be worse than Dijkstra's algorithm, since the broadcast cycle of EB is longer.

In this section we present the *Next Region* method (NR), which avoids the above problem. The server again precomputes the shortest paths between all border nodes of different regions, but now the index keeps for each pair of R_i, R_j the identifiers of intermediate regions appearing in any shortest path between any of their border nodes. These regions are guaranteed to contain the shortest path from any node in R_i to any node in R_j. To exemplify, consider that the index includes an $n \times n \times n$ array A of boolean values, where n is the number of network regions. The bit in cell A_{R_i,R_j,R_k} is 1 if and only if there exists a shortest path from R_i to R_j that traverses R_k. Given array A, the client knows in advance which regions are necessary for query processing.

Consider the example in Figure 4. The source is in region R_1, which has two border nodes; the destination is in R_{16},

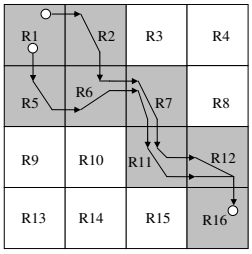


Figure 4: Shortest paths from R_1 to R_{16}

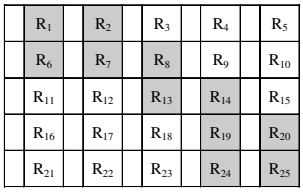


Figure 5: Needed regions

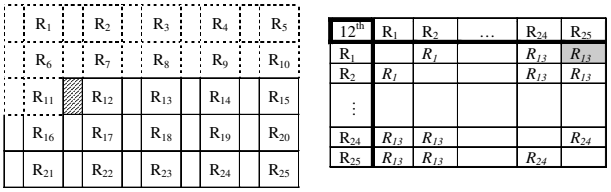
which has a single border node. There are two shortest paths between the border nodes of R_1 and R_{16}. One of them traverses regions R_2, R_6, R_7, R_{11} and R_{12}, and the other regions R_5, R_6, R_7, R_{11} and R_{12}. The NR index records that any shortest path from R_1 to R_{16} may only pass through the union of the above region sets (shown gray in the figure), and the corresponding bits in A are set to 1.

Array A as described above has n^3 size, and would lead to a large index. This, in turn, implies a long broadcast cycle, especially if the $(1,m)$ scheme is applied. In Section 5.1 we show how we can retain the pruning effectiveness of NR while both (i) keeping the broadcast cycle short and (ii) achieving low access latency. We stress that the reduced access latency is provided by local, region-specific indexes, broadcast immediately before the corresponding regions. This eliminates the need for $(1,m)$ interleaving, and is fundamentally different from the common practice in the literature of having a global index, and replicating identical copies of it in the broadcast cycle.

5.1 Index of NR

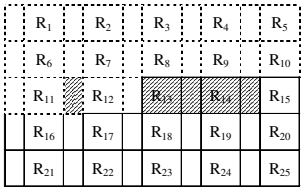
The first component of the index in NR is the same as EB, and is used to identify the source and destination regions R_s and R_t. Regarding the second component, the main idea is that, since the device will have to wake up every time a needed region is broadcast, it does not need to know all the required regions in advance. It suffices, instead, to only know when the *next* required region will be broadcast. When the client receives that region, it also listens to the adjacent local index in order to determine the next required region, and so on. This way, we keep the broadcast cycle small, we enable the client to receive only the relevant parts of (instead of the entire) indexing information, and allow the device to commence query processing shortly after tuning in for the first time, without employing the $(1,m)$ scheme.

Specifically, the index A^m of region R_m is an array with n rows and n columns. A^m is placed in the broadcast cycle immediately before R_m's data. Every cell $A^m_{R_i,R_j}$ indicates the next region R_{nxt} in the broadcast cycle that is needed for a shortest path from R_i to R_j. Note that R_{nxt} could be R_m itself. Figure 5 shows the structure of the cycle, where the unlabeled slot before each region corresponds to its index.



12th	R1	R2	…	R24	R25
R1		*R1*		*R13*	*R13*
R2	*R1*			*R13*	*R13*
⋮					
R24	*R13*	*R13*			*R24*
R25	*R13*	*R13*		*R24*	

Figure 6: NR algorithm; receiving A^{12}



15th	R1	R2	…	R24	R25
R1		*R1*		*R19*	*R19*
R2	*R1*			*R19*	*R19*
⋮					
R24	*R19*	*R19*			*R24*
R25	*R19*	*R19*		*R24*	

Figure 7: NR algorithm; receiving A^{15}

5.2 Client-side Processing in NR

Posed a query, the device tunes in the channel, receives the current packet, and waits until the subsequent index is broadcast (for this to be possible, every packet in the cycle includes a pointer (offset) to the subsequent index). The client receives this index, and finds out what the next required region R_{nxt} is. It wakes up when R_{nxt} is broadcast and keeps listening until A^{nxt+1} is also received. From A^{nxt+1}, it determines the next needed region, and so on. Note that if the end of the current broadcast cycle is reached, another starts, and processing continues as if it was the same cycle. When the latest index received indicates that R_{nxt} is a region that the client already possesses, listening stops and a Dijkstra search computes the shortest path over all collected regions.

Similarly to EB, the access latency in NR does not exceed one broadcast cycle. Regarding tuning time and memory requirements, we expect NR to be superior to EB, as the client listens only to a subset of the regions necessary in EB. The same holds for CPU time at the client. Pre-computation cost is identical to EB (assuming the same partitioning), as the same shortest paths among border nodes are computed.

To illustrate, consider the broadcast cycle in Figure 5. The user wants to find the shortest path from a source in R_1 to a destination in R_{25}. The needed regions for this shortest path computation are shown in gray color, but the client does not know this in advance. Assume that the query is posed while R_{11} data are broadcast, which points the client to index A^{12}. Index A^{12} (shown in Figure 6) indicates that R_{13} is the next needed region, so the device sleeps and wakes up to receive R_{13} and also the adjacent index A^{14}. A^{14} indicates that R_{14} is also required, so the client continues to receive data from the channel, until A^{15} points to R_{19}, as shown in Figure 7. The device sleeps until R_{19} is broadcast, and so on. The process continues this way until R_8 is received and index A^9 points to the already available R_{13}; listening stops and the shortest path is computed. Algorithm 2 in Appendix B formalizes this process.

So far we have assumed that source and destination are network nodes. In practice this may not always be the case, i.e., the source/destination could be at arbitrary locations on the network. EB and NR work as described, the difference being that the border nodes of a region are now defined as the intersections of its network edges with the splitting lines of the kd-tree (i.e., with the boundary of the region).

6. PRACTICAL CONSIDERATIONS

6.1 Memory-bound Processing

Both NR and EB methods receive all the needed regions before processing the shortest path query. In case however a device has very limited memory, it is possible to reduce space requirements if the client pre-computes some shortest paths within each needed region R as soon as R is received. The region data can be discarded, and only the local shortest paths (called *super-edges*) can be kept in memory.

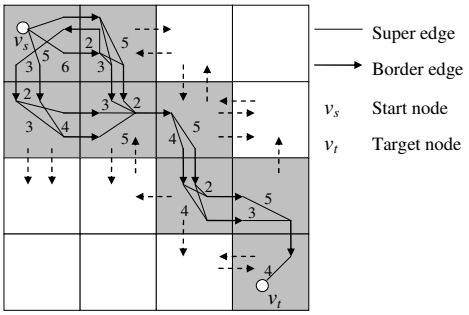


Figure 8: G' graph

Specifically, if R is not R_s or R_t, a super-edge is computed for each pair of R's border nodes. Especially for R_s and R_t, we include v_s and v_t to the border node set. This produces a graph G' through the needed regions, that includes only super-edges and border edges (i.e., edges between border nodes); the costs of super-edges equal the costs of the shortest paths they represent. An example is given in Figure 8, where received regions are shown in gray, the arrows are border edges, the line segments correspond to super-edges, and the numbers next to super-edges are their costs.

When all needed regions have been received and their super-edges pre-computed, the client executes Dijkstra's algorithm on G' to find the shortest path from v_s to v_t. In the returned path, super-edges are replaced by their corresponding path. A further optimization in the last step is to ignore border nodes adjacent only to irrelevant (white) regions, along with their super-edges, as they cannot be part of the shortest path; edges of such border nodes are shown as dashed arrows in the figure.

For the above technique to be fully effective, pre-computing the super-edges in a region must be faster than receiving the next region's data. This is always the case for typical 3G network speeds and the widespread ARM processor used in our evaluation. In our experiments, this mechanism reduces peak memory consumption roughly by 35%.

6.2 Dealing with Packet Loss

When receiving data through a wireless channel, packets may be lost due to bad reception, noise and/or network errors [15]. A practical method should be able to deal with this situation. In Dijkstra's algorithm, if a packet is lost, search is not guaranteed to return the shortest possible path; the missing nodes could lead to a shorter distance than that in an incomplete graph. Therefore, the client has to receive lost packets in the next broadcast cycle.

In ArcFlag, Landmark, and SPQ it is important to separate the adjacency lists of nodes from pre-computed infor-

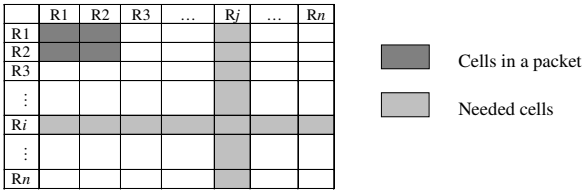


Figure 9: Cell placement into packets

mation. The reason is that missing the pre-computed data for a node/edge, can potentially be overcome if the adjacency data are intact (although the reverse does not hold). To minimize the chances that both pieces of information are lost, we avoid placing them in the same packet. In HiTi this separation is already in place, since there is a distinct index.

Specifically, if a bit vector is missing in ArcFlag, all its bits are assumed to be 1, and processing is possible as long as the adjacency data are available. In Landmark, if a node's distance vector is lost, the lower bound of its distance from v_t is assumed to be 0. In SPQ, loss of a quad-tree implies that all adjacent edges of the specific node have to be considered by the search. In HiTi, a lost index packet can be reconstructed if the underlying data (raw adjacency lists or lower index level) are available. In all cases, missing any needed adjacency data still requires waiting for the next cycle.

Similarly, in EB and NR loss of needed region data demands receiving them in the next cycle. If a packet of their first index component (used for locating v_s and v_t) is lost, the next index is received. Loss in their second (and larger) index component can be effectively dealt with. Consider EB first, and let R_s and R_t be the i^{th} and j^{th} region, respectively. A central observation is that all minimum/maximum distances required for EB pruning are contained in the i^{th} row and j^{th} column of array A, shown light gray in Figure 9. If the lost packet contains any of these values, the client must wait for the next index. To minimize the chances that this happens, each index packet corresponds to a square $(w \times w)$ area of cells in A; e.g., the dark gray cells would be placed in the same packet. We use squares since, among all rectangles that cover the same number of cells, a square intersects the smallest number of rows and columns.

In NR a single value is necessary from each local array A^m (i.e., $A^m_{R_s,R_t}$), so the above optimization is not meaningful. This very fact, however, means that even if an A^m packet is missed, the probability that it contains the necessary value is small. In the event that value $A^m_{R_s,R_t}$ is missed, NR must listen to the adjacent region R_m and to the next index A^{m+1}. Note that it is not possible to simply ignore R_m and directly proceed to index A^{m+1}. In Figure 7, assume that the necessary value from A^{15}, which in reality points to R_{19}, is lost. The client has no means to infer whether R_{15} is needed, unless it waits for A^{15} to be broadcast again (recall that every region's index is unique). To avoid this cycle-long delay, R_{15} is received anyway, and included in the final Dijkstra search.

A general remark is that the lower the tuning time of a method, the less it degrades with packet loss. Hence, we expect our approaches to be significantly more resilient to losses than those in Section 3.2, with NR the least affected.

7. EVALUATION

In this section we evaluate alternative methods w.r.t. the performance factors in Section 3.1. Client-side algorithms were implemented in the proliferous Java 2 Micro Edition (J2ME) platform, using Java ME SDK v3.0. The device (memory, CPU, etc) is simulated inside the SDK as a generic GPS-enabled clamshell phone supporting the current J2ME standards: CLDC-1.1 and MIDP-2.1. The default heap memory of this generic device is 8MB. Server-side processing (i.e., pre-computation for broadcast cycle creation) was implemented in C++ and run on a 3GHz machine. We use 5 real road networks, the default being Germany with 28,867 nodes and 30,429 edges.

Table 1: Broadcast cycle length

Method	Packets	Sec. (2Mbps)	Sec. (348Kbps)
Dijkstra (DJ)	14019	6.845	40.284
NR	14260	6.963	40.977
EB	15299	7.470	43.962
Landmark (LD)	21236	10.369	61.022
ArcFlag (AF)	29233	14.274	84.003
SPQ	52337	25.555	150.391
HiTi	58138	28.387	167.061

Our evaluation considers Dijkstra's algorithm, ArcFlag, Landmark, EB and NR. HiTi and SPQ are excluded as their space requirements exceed our device's heap size even for the smallest of our networks; for further indications about their impracticality see Table 1 and discussion thereof.

In ArcFlag, EB, and NR, the regions are produced by kd-tree partitioning. We fine-tuned ArcFlag, EB, and NR w.r.t. the number of partitions (choosing 16, 32, and 32 for the default network, respectively) and Landmark w.r.t. the number of landmarks (the best being 4); the fine-tuning experiment can be found in Appendix C. In each experiment we process 400 shortest path queries between randomly selected source and destination nodes. The packet size is set to 128 bytes; packet size affects methods in a similar way, thus experiments for different sizes are omitted.

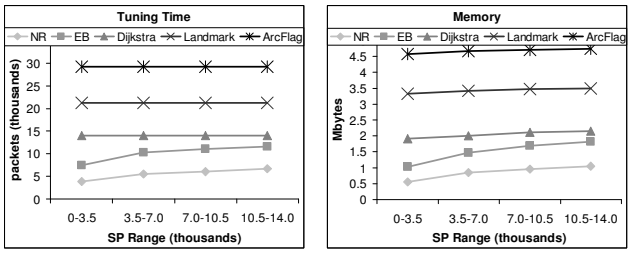
In our default network, shortest path pre-computation at the server takes around 62 seconds for EB and NR, 58 seconds for ArcFlag, and 1 second for Landmark. This one-off cost is reasonable for all methods. Measurements for other networks are given in Appendix C.

Table 1 shows the size of the broadcast cycle in packets. To express its duration in absolute terms, we also show the time it takes to broadcast over a 2Mbps and a 384Kbps channel, which are typical in 3G networks for static and moving devices. The reported times also serve as an indication of the access latency in seconds. Dijkstra has the smallest cycle (containing only network data), followed closely by NR and EB; this verifies that our methods satisfy the design objective of broadcasting very little indexing information, as opposed to any pre-computation-based competitor. Observe that the extra information of HiTi and SPQ is several times larger than the network itself. In addition to high space requirements, this implies an excessive tuning time (as the client needs to receive the entire index/all the quad-trees) and a long access latency.

The broadcast cycles of Dijkstra, ArcFlag, and Landmark also indicate their space consumption, since an entire cycle must be received for query processing. Their memory requirements exceed the available heap size of our device for larger networks; Table 2 presents which methods managed to run in the various networks. Germany was chosen as the default, because it is the largest network where all our three competitors work. Complete performance investigation for different networks is given in Appendix C.

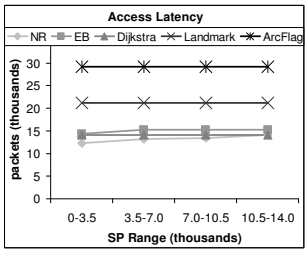
Table 2: Method applicability per network

	Nodes	Edges	AF	LD	DJ	EB	NR
Milan	14021	26849	√	√	√	√	√
Germany	28867	30429	√	√	√	√	√
Argentina	85287	88357			√	√	√
India	149566	155483				√	√
S. Francisco	174956	223001					√

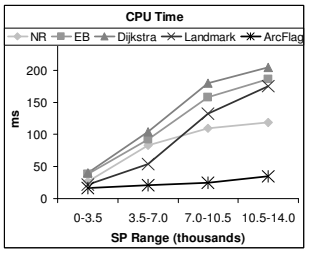


(a) Tuning Time

(b) Memory



(c) Access Latency



(d) CPU Time

Figure 10: Effect of shortest path length

In Figure 10 we present the tuning time, memory requirements, access latency and CPU time in Germany network versus the length of the shortest path. The 400 queries/paths computed are classified into 4 length categories. We note that the diameter of the network is 14,383 while the 400 paths have lengths from 597 to 13,249. NR is by far the best method in tuning time and memory consumption, and EB is the runner-up, which verifies their pruning effectiveness. On the other hand, the deficiencies of EB w.r.t. NR are obvious for long paths, as anticipated. Competitors perform poorly since they listen to (and store) the entire broadcast cycle; the problem is more serious for ArcFlag and Landmark, whose cycles are particularly long. An interesting result is that NR achieves lower access latency even than Dijkstra, although the latter has the shortest possible cycle; this is because NR receives only a subset of the broadcast packets, which usually does not span the entire cycle. This is not the case for EB, as its longer cycle outweighs this effect. In terms of CPU time, Landmark is faster than NR and EB. However, as explained in Section 3.1, this is the least important factor, as its effect on power consumption is minimal compared to tuning time, and insignificant w.r.t responsiveness too (CPU time is in the order of milliseconds, while access latency in the order of seconds).

We evaluated the competing schemes for various packet loss rates; results show that NR and EB are quite robust, and retain their performance advantages over competitors. Also, we investigated the effectiveness of the technique in Section 6.1 in reducing space consumption for memory-bound devices. We found that client-side pre-computation lowers peak memory utilization by around 35% for both EB and NR. Detailed results for the above are given in Appendix C.

8. CONCLUSIONS

The continuing diffusion of mobile devices enables a flourishing market of location-based services, but also poses serious scalability concerns. This motivates the wireless broadcast model, where a server repeatedly transmits the entire database, while clients selectively tune in the communication channel and process their queries locally.

In this paper, we propose the first study on shortest path computation in this model. We first adapt existing shortest path approaches to the broadcast setting, and identify their deficiencies. Then, we propose two novel methods tailored to the technical peculiarities of the model and of real-world handheld devices. Finally, we empirically demonstrate the efficiency of our techniques using real road networks and actual device specifications. A promising direction for future work is to consider on-air processing of spatial queries in road networks, e.g., range and nearest neighbor retrieval.

9. REFERENCES

[1] Mobile World Congress: Network Breaking Point, February 17 2010.

[2] C.-Y. Chow, M. F. Mokbel, and W. G. Aref. Casper*: Query processing for location services without compromising privacy. *ACM TODS*, 34(4), 2009.

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

[4] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, pages 156–165, 2005.

[5] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE TSSC*, 4(2):100–107, 1968.

[6] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on air: Organization and access. *IEEE TKDE*, 9(3):353–372, 1997.

[7] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE TKDE*, 10(3):409–432, 1998.

[8] E.-S. Jung and N. H. Vaidya. An energy efficient MAC protocol for wireless LANs. In *INFOCOM*, 2002.

[9] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE TKDE*, 14(5):1029–1046, 2002.

[10] E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In *9th DIMACS Implementation Challenge - Shortest Paths*, 2007.

[11] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup Dijkstra's algorithm. *J. Exp. Algorithmics*, 11:2.8, 2006.

[12] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of spatial queries in wireless broadcast environments. *IEEE TMC*, 8(10):1297–1311, 2009.

[13] J. Pigg. Mobile backhaul: Will the levees hold?, 2009.

[14] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.

[15] I. Stojmenovic. *Handbook of wireless networks and mobile computing*. John Wiley & Sons, 2002.

[16] B. Zheng, W.-C. Lee, and D. L. Lee. Spatial queries in wireless broadcast systems. *Wireless Networks*, 10(6):723–736, 2004.

[17] B. Zheng, W.-C. Lee, K. C. K. Lee, D. L. Lee, and M. Shao. A distributed spatial index for error-prone wireless data broadcast. *VLDB J.*, 18(4):959–986, 2009.

APPENDIX A: SPATIAL AIR INDEXES

For range and k nearest neighbor (kNN) queries, [16] presents the *Hilbert curve index* (HCI) method, which is based on the $(1, m)$ interleaving scheme. The data objects (2-dimensional space) are mapped onto a Hilbert curve (1-dimensional space) and indexed with a B^+-tree. This index is then broadcast on the air.

To execute a range query, a client initially determines the first and the last points on the Hilbert curve that fall in the query window. Then, it retrieves all objects with Hilbert values between them. Finally, it maps the collected objects back to the 2-dimensional space and checks which of them are actually inside the query window.

In the case of kNN processing, the method consists of two steps. In the first step, the position of the user is mapped onto the Hilbert curve and the k objects with Hilbert values closest to it are determined. In the second step, the client calculates the maximum Euclidean distance between its location and these k objects. Using this distance as radius, a range query is processed around the client's location (in the way described before) in order to retrieve the candidate neighbors. Eventually, the distances of these candidates are computed and the k closest are reported as the actual kNNs.

The *distributed spatial index* (DSI) [17] aims at minimizing the access latency of HCI at the cost of longer tuning time. The objects are sorted on their Hilbert values and placed into equi-sized *frames*. Each frame also contains an index, which points to subsequent frames along with the minimum Hilbert value inside them. The index does not point to all subsequent frames. Instead, it points 2^n frame positions afterwards, where $n = 0, 1, 2, 3, ...$ This way, fast access to both nearby and distant frames is possible. The client listens to an index and finds the furthest frame where the minimum Hilbert value does not exceed the required Hilbert value. This process is repeated until the required Hilbert value is found. The processing of queries is similar to HCI, exploiting again the Hilbert curve's properties.

In the *broadcast grid index* (BGI) method [12], the data objects are initially partitioned using a regular grid, i.e., a grid with equi-sized cells. Each grid cell holds the coordinates of objects that lie inside it, plus their total number. This information forms the index of BGI. The full object information is divided into m segments using the $(1, m)$ scheme, and a copy of the index precedes each segment.

In order to compute a kNN query, the algorithm performs two steps. In the first step, the client receives index information for the cells. Using the extent and number of objects in each cell, it calculates an upper bound d_{max} of the radius around its position that contains at least k objects. In the second step, the device receives from the broadcast cycle only the objects within distance equal to or smaller than d_{max}. As the client receives index information about more cells or more object coordinates, it reduces the upper bound d_{max} incrementally, excluding more unnecessary packets each time. BGI can additionally support continuous kNN queries; these are standing queries that require continuous re-evaluation as the data objects move.

APPENDIX B: CLIENT-SIDE ALGORITHMS

Below are the pseudo-codes that summarize query processing at the client side in EB and NR (Algorithms 1 and 2, respectively).

Algorithm 1 EB Algorithm

```
1: connect();
2: receive_network_data();
3: Let R_s=find_source_region();
4: Let R_t=find_destination_region();
5: sleep_until_index_broadcast();
6: connect();
7: Let index=receive_index();
8: for i = 1 to number_of_regions do
9:     if index[R_s][i].min + index[i][R_t].min ≤ index[R_s][R_t].max then
10:        needed_regions.add(i);
11: while needed_regions.size > 0 do
12:    Let next_region = needed_regions.pop(next_region_to_be_broadcast);
13:    sleep_until_broadcast(next_region);
14:    connect();
15:    get_region(next_region);
16: calculate_SP();
```

Algorithm 2 NR Algorithm

```
1: connect();
2: receive_network_data();
3: Let R_s=find_source_region();
4: Let R_t=find_destination_region();
5: sleep_until_index_broadcast();
6: connect();
7: Let first_index=receive_index();
8: Let first_region=first_index[R_s][R_t];
9: sleep_until_broadcast(first_region);
10: connect();
11: get_region(first_region);
12: Let current_index=indexof(first_region + 1);
13: next_region = current_index[R_s][R_t];
14: while first_index ≠ current_index and first_region ≠ next_region do
15:    sleep_until_broadcast(next_region);
16:    connect();
17:    get_region(next_region);
18:    Let current_index=indexof(next_region + 1);
19:    Let next_region = current_index[R_s][R_t];
20: calculate_SP();
```

APPENDIX C: EXTENDED EVALUATION

This section includes additional experiments that were either omitted or simply summarized in Section 7.

C.1 Method Fine-tuning

To fine-tune the methods, we measure their performance for various numbers of partitions (for ArcFlag, EB and NR) and landmarks (for Landmark). Figure 11 plots the results; the x-axis corresponds to the number of regions or landmarks, depending on the method. Dijkstra's algorithm is included for comparison purposes only.

For EB and NR, too few partitions imply looser pruning, and therefore receipt of more region packets. Too many, however, lead to large indexes, and thus to listening more index packets. On the contrary, access latency strictly increases for more regions, because the broadcast cycle grows. Using 32 partitions seems to strike the best balance for both EB and NR. There is a single measurement for ArcFlag (for 16 regions), since for more regions its space requirements exceed the client's heap size. Regarding Landmark, we use 4 landmarks, as at least its CPU improvements are visible. Its trends demonstrate clearly that conventional precomputation methods, in general, are unsuitable to wireless broadcasting, due to their voluminous indexes/bit vectors/distance vectors.

C.2 Pre-computation Time

In Table 3 we present the pre-computation time needed to form the broadcast cycle at the server for different road networks. These times are reasonable for all approaches even for the largest networks, as this is a one-off cost paid during the setup of the system. Note that EB and NR have the same cost as they need to pre-compute the exact same shortest paths (when starting with the same regions).

C.3 Performance for Different Networks

To test the methods for different networks, we fine-tune them every time as above. Figure 12 shows their performance. Missing values correspond to cases where the respective approach run out of memory at the client, as explained in Table 2. The only method that works for all networks is NR. Moreover, even for the largest network, it uses less than half of the available heap size.

C.4 Memory-bound Processing

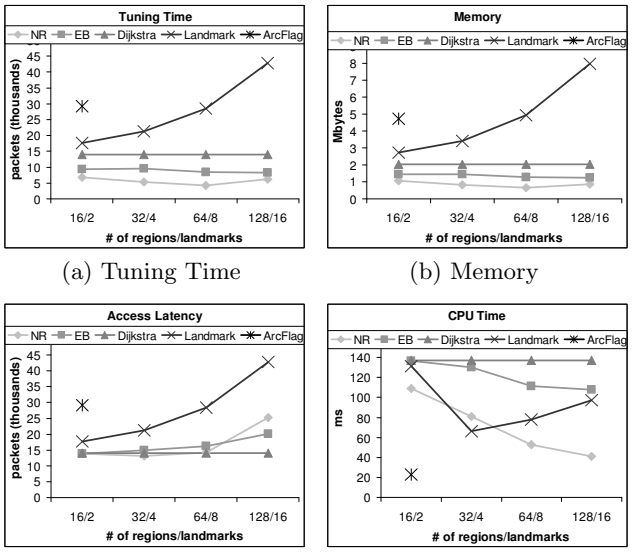
In Section 6.1 we mentioned that it is possible to further reduce the memory requirements of EB and NR if the client pre-computes several shortest paths as soon as some regions are received. The memory requirements with and without this technique are depicted in Figure 13(a). Figure 13(b) shows the cost of shortest path calculation after all necessary data are received. The downside of this technique is that it requires extra CPU effort at the client (and thus consumes extra power) for pre-computations in the region receiving phase.

Table 3: Pre-computation time (sec)

	EB/NR	ArcFlag	Landmark
Milano	55.516	68.500	0.453
Germany	61.797	58.11	1.031
Argentina	454.766	308.859	2.641
India	2348.421	753.219	4.359
San Francisco	6332.468	2164.75	5.312

C.5 Robustness to Packet Loss

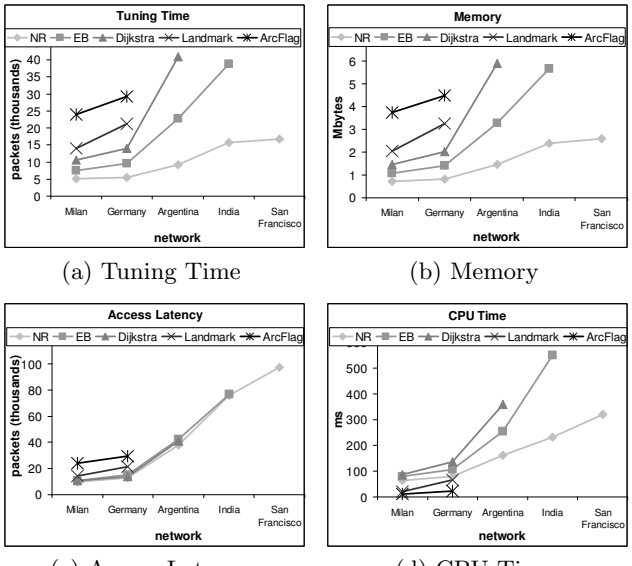
In Figure 14 we measure the effect of packet loss on performance, varying the loss rate from 0.1% to 10% (according to [15], in practice the packet loss rates range up to 10%). Only tuning time and access latency are considered, as memory requirements and CPU time are essentially unaffected by losses. The relative performance of the methods is similar to previous experiments, with NR being the clear winner for all loss rates in terms of both tuning time and access latency.



(a) Tuning Time (b) Memory

(c) Access Latency (d) CPU Time

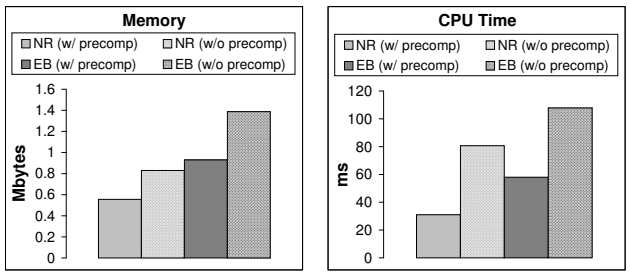
Figure 11: Fine-tuning



(a) Tuning Time (b) Memory

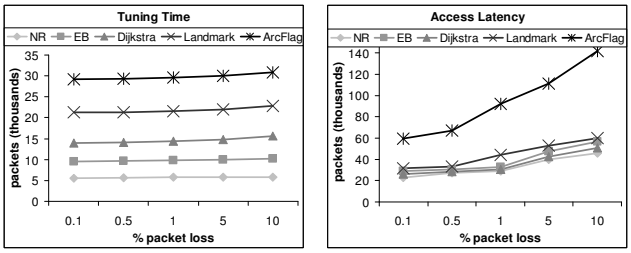
(c) Access Latency (d) CPU Time

Figure 12: Different networks



(a) Memory (b) CPU Time

Figure 13: Client-side pre-computation scheme



(a) Tuning Time (b) Access Latency

Figure 14: Effect of packet loss