Maintaining Connectivity in Dynamic Multimodal Network Models

Petko Bakalov, Vassilis J. Tsotras *
University of California
Riverside, CA 92521
pbakalov, tsotras@cs.ucr.edu

Erik Hoel, Wee-Liang Heng, Sudhakar Menon
Environmental Systems Research Institute (ESRI)
Redlands, CA 92373
ehoel,wheng,smenon@esri.com

Abstract

Network data models are frequently used as a mechanism to describe the connectivity between spatial features in many emerging GIS applications (location-based services, transportation design, navigational systems etc.) Connectivity information is required for solving a wide range of location based queries like finding the shortest path, service areas discovery, allocation and distance matrix computation. Nevertheless real life networks are dynamic *in nature since spatial features can be periodically modified. Such updates may change the connectivity relations with the other features and connectivity needs to be reestablished. Existing approaches are not suitable for a dynamic environment, since whenever a feature change occurs, the whole network connectivity has to be reconstructed from scratch. In this paper we propose an efficient algorithm which* incrementally *maintains connectivity within a dynamic network. Our solution is based on the existing functionality (tables, joins, sorting algorithms) provided by a standard relational DBMS and has been implemented, tested and will be shipped with the ESRI ArcGIS 10 product.*

1 Introduction

Network data models have been proposed as an efficient way to represent connectivity information among spatial features in geographic information systems [13] [6] [12] [9]. Connectivity information for such features is explicitly represented by network elements that are found in an associated logical network. Many of the proposed designs have also been implemented in existing operational systems like Arc/Info [14] and TransCAD [3] where the network model is persisted inside a centralized database server. Using connectivity information, these systems can then be utilized to solve a wide range of problems, typical for the transportation networks like finding the shortest path between points of interest, finding optimal resource allocation and many others.

In order to provide more accurate results and better service to the end users, a network model has to meet a growing set of sophisticated (functional and system) requirements, needed for real world transportation systems modeling. Some primary functional requirements for a robust network data model are: (i) Ability to model *multi-modal* transportation networks [20]. A multi-modal transportation network is a network which includes more than one mode of transportation (like freeways and railroads). In such transportation systems, the users can change their transportation mode (drive a car to the train station and then take a train) but such changes can occur only in a set of connectivity points shared by both transportation modes. (ii) Ability to model different *hierarchical levels*. The level hierarchy occurs naturally in transportation systems: consider for example the road network where the freeways are typically associated with the highest hierarchy level followed by the highways and the city streets. (iii) Ability to model (simple two-part) *turns* and (more complex multi-part) *maneuvers* [21]. The presence of such turns in the network can have great impact on the movement inside the network [1]. (iv) Ability to capture a big set of real life *constraints* [7]. For example, restrictions like one way streets and weight-height limitations have to be reflected in the network model.

In addition to these functional requirements a network data model has to meet also a set of standard system requirements for performance, multi-user support, interoperability and scalability. To satisfy the system requirements the network model has to be persisted inside a commercial relational database (using only the standard DBMS tools and operators).

Nevertheless, a common weakness in previously proposed network model designs is that they do not consider *dynamic* modifications in the network. Such modifications occur often in many real life scenarios where spatial features can be frequently modified (e.g., features are updated, deleted or inserted). Even a single feature update can change the connectivity relations among other features

[1]This work was partially supported by NSF grant IIS-0534781.

and connectivity needs to be reestablished. To address this problem, all existing approaches reestablish the connectivity of the *whole* network from scratch. While the correct result is finally produced, this process is very time consuming (networks are typically very large e.g. 50 million linear features in a continental wide transportation networks) and can be prohibitive for many applications (for example, networks supporting on-line navigation queries or location-based services). A new and effective mechanism to maintain the correctness of the network model in dynamic environment is thus needed.

In this paper we present an *incremental* algorithm for maintaining the correctness of the connectivity information in the presence of modifications. This algorithm has been implemented on top of a relational database for ESRI's ArcGIS geographic information system.

We introduce the notion of a $dirty\ area$ inside a network data model which tracks feature edits. This concept have been used previously in topologies [10]. A dirty area is a region inside the network spatial extend where the network features have been modified but the correctness of their connectivity information has not been verified. The network data model is assumed correct only when it is free of dirty areas. A dirty area is incrementally reduced by a process called $rebuilding$. The rebuilding may happen over the entire dirty area in the network, or it may affect only portions of it. The end user specifies which portions of the dirty area wants to be cleaned and the rebuild process analyzes and re-establishes the connectivity information there. Allowing users the ability to rebuild only portions of the dirty area is a practical requirement in scenarios involving very big seamless networks. The user may clean only these portions of the network extend which are of interest to a given application or query, thus avoiding the costly total rebuild.

Effectively dirty areas can be viewed as a mechanism to support transaction functionality over complex network data (graphs etc.). The database starts with a consistent state (i.e. without any dirty areas), then the updates are applied, the dirty areas are cleaned and eventually the database returns to a consistent state when all dirty areas are cleaned (the "end" of the transaction).

The rest of the paper is organized as follows: Section 2 provides a brief description of the network model including logical structure and physical design. Section 3 discuses the physical implementation. Section 4 provides in depth description of our rebuilding algorithm. Section 5 discusses experiences with the actual implementation of our solution while section 6 concludes the paper.

2 Network Model Basics

We proceed with a brief description of the network model introduced in [9]. The term *feature* in a GIS system is used as a graphical representation of a real world object. Line features are used to represent objects with line geometry; for example, freeways, railways etc. Point features are used when the location of an object is important (but not the object's exact shape); for example, street intersections or railway stations. A *network model* is a graph used for storing connectivity information about spatial *features* with line or point geometry. Lines are represented as $edges$ while points are noted as $junctions$. Edges and junctions do not have shape attributes and are used to represent connectivity information. They are also termed as $network\ elements$.

While a spatial feature can be represented with multiple network elements (for example, a line can be segmented into multiple edges in the graph), a network element has at most one spatial feature associated with it with the appropriate geometry type. In particular, an edge element is associated with exactly one line feature while a junction element is associated with at most one point feature. Relations between network elements represent connections between their corresponding features. For example, consider two line features intersecting at a given point. In the network model this will be represented as one junction (associated to the intersection) connected with four edges (each edge associated with a line segment).

A network element is also described by a set of numerical or boolean values called $network\ attributes$. These attributes are used to store properties of interest from the real world object whose representation in the system is the feature associated with this network element. For example, consider an edge associated with a street segment; this edge may have as attributes the length of the street segment, or, the travel time across this segment, etc. Network attributes typically provide the means for optimization (shortest path, minimal travel time, etc.) during the analysis of the network model. They are stored along with the network elements in order to minimize the number of tables that have to be accessed and thus achieve better performance.

Because of their very large data size (e.g., tens of millions of features), network models are usually located in a centralized server, persisted either in a RDBMS or in a file system. The process of analysis is done on the client side of the architecture by components called *network solvers*. Solvers implement a wide range of graph algorithms (like finding shortest paths etc. [16] [18] [2]) and obtain data from the server utilizing specialized access methods (e.g. forward star cursor [5] [17] [19]).

Turns. In addition to the edge and junction elements, a network can also represent real life movement constraints. This is performed by special network elements called *turns*. A turn is anchored to a specific junction (the junction where the turn starts) and controls the movement between two edges (edge-In, edge-Out) connected to this junction. The presence of turns can have great impact on the movement

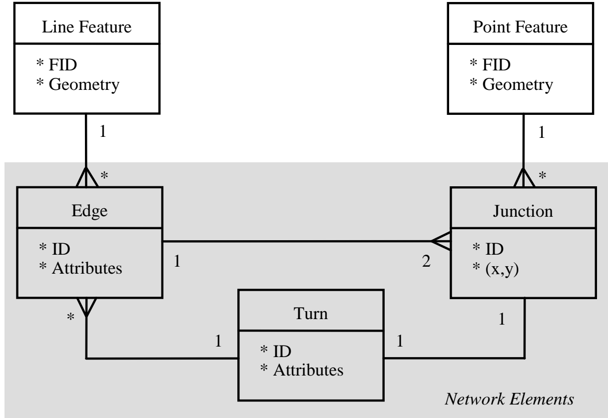
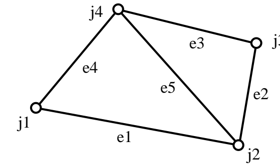


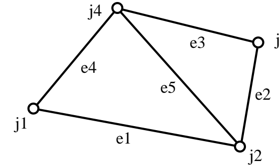
Figure 1. ER network model design



Edge Table			
id	Attributes	jid1	jid2
e1	…	j1	j2
e2	…	j2	j3
e3	…	j3	j4
e4	…	j4	j1
e5	…	j4	j2

Junction Table	
id	Attributes
j1	…
j2	…
j3	…
j4	…

Figure 2. Network tables as suggested from the ER



Junction Table									
id	edge1	junct1	edge2	junct2	edge3	junct3	edge4	junct4	turns
j1	e2	j2	e4	j4	null	null	null	null	no
j2	e1	j1	e2	j3	e5	j4	null	null	no
j3	e2	j2	e3	j4	null	null	null	null	no
j4	e3	j3	e4	j1	e5	j2	null	null	no
	Adjacency pair 1		Adjacency pair 2		Adjacency pair 3		Adjacency pair 4		

Figure 3. Actual junction table used

inside the network (and the analysis process) so the ability to model this aspect is critical [1]. In our system, turns are explicitly maintained in a relation (the *turn table*) that stores the turn's anchored junction, the edges associated with it and a user specified penalty (the cost associated with traveling on this turn). We choose this approach because it can be easily optimized for the most commonly used client access patterns [19].

Modeling Connectivity. Connectivity is described by a set of rules specifying how the features, modeling real world objects are connected to each other. A simple connectivity model, called *end point connectivity*, is based on the spatial coincidence of the point features and the endpoints of the line features (that is, two lines are connected only if their endpoints coincide). It works relatively well for planar datasets where the line features do not cross each other. However, for applications that deal with non-planar data, another connectivity model has been introduced, the *mid-span connectivity model*, where connections are established based not only on the end-point collocation but also on the spatial collocation on the mid-span vertexes along the line features. Since both models are widely used, they are supported by our algorithms.

3 Physical Implementation

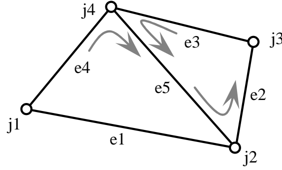
We now discuss our physical implementation of a network model within a relational database. Figure 1 shows the normalized ER diagram of a typical network model design. Similar designs have been used in many research or commercial implementations [11] [8] [15] [4]. In this ER diagram, network elements are represented explicitly with entity sets while the connectivity information is stored as the *from* and *to* relationships between the edge and junction entity types. This representation of connectivity comes naturally from the mathematical formalization of a graph where an edge is defined as a binary relation on junctions. Translating this ER into a relational schema, would result in creating an edge table with attributes *from* and *to* (the endpoints of an edge) which are foreign keys to the junction table (i.e., refer to the junction ID column) (see Figure 2).

This approach however, has some disadvantages from a performance point of view. The major one is revealed during the process of network traversing when the solver algorithms move inside the graph from one junction to another following the connecting edges. Every time a junction j_i is explored, we need to discover the set of edges associated with that junction. This is done by issuing a separate SQL query in the edge table checking whether j_i appears in the "from" or "to" attributes. This is very time consuming, given that the solver algorithms typically examine many junctions.

Instead we use a different implementation for the junction and turn tables. We represent the connectivity information as a set of *adjacency pairs* of the form $\langle edgeId, junctionId \rangle$, stored inside the junction table (see Figure 3). For example, junction j_1 is connected through edge e_1 to junction j_2, etc. This approach is tailored to-

Table 1. Frequency distribution on junction degrees for different datasets.

Degree	1	2	3	4	5	6	7	8
Count	147,860	43,737	375,340	145,400	2,689	234	11	1
Percent	20.7%	6.1%	52.5%	20.3%	0.4%	0.03%	0.01%	0.001%



Turn Table

jid	turnId	edge1	edge2	turnId	edge1	edge2	….	edge1	edge2
j4	t1	e4	e5	t2	e3	e5	….	null	null
j2	t3	e5	e2	null	null	null	….	null	null
…	….	….	….	….	….	….	….	….	….
…	….	….	….	….	….	….	….	….	….

Turn record 1 — Turn record 2 — Turn record 5

Figure 4. Turn table example

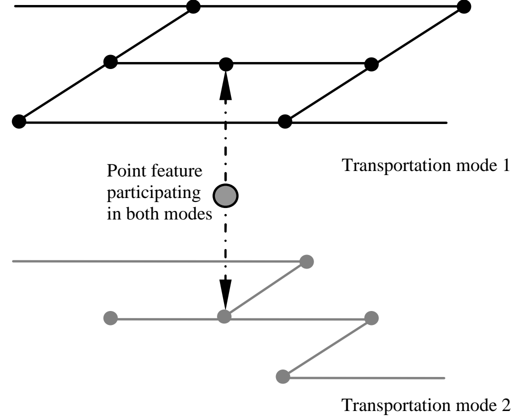


Figure 5. Connectivity model for multi-modal networks

wards navigation as it is designed to answer the most common type of adjacency queries during the network analysis process. That is, all connections to given junction j_i appear in the record of this junction. The junction table uses fixed-length records for direct access purposes; this implies a fixed number of adjacency pairs per record (see figure 3). Through an empirical evaluation over several real datasets (see table 1) it was found that 99.5% of the junctions have four or less adjacent junctions; hence we fixed the number of adjacency pairs in the junction table to four. If a junction requires more than 4 adjacency pairs a special overflow table is created.

Similarly, in a typical network traversal process, the graph is traversed from junction to junction. It is thus advantageous, at each junction to know all the turns (and their penalties) anchored at this junction. This has influenced the way we implement the turn table (see figure 4). If there are any turns anchored at a junction j_i, the turn table will have a record with primary key j_i which also contains all the turns anchored on j_i. Information about a turn is stored in the form of a *turn triplet* $\langle turnId, edge{-}InId, edge{-}OutId\rangle$. Again from empirical evaluation, we have set a fixed length of five turns per junction.

4 Connectivity Maintenance

We view the maintenance of the network connectivity as a two phase process: (i) the initial establishment of connectivity where the connectivity graph is derived from the features participating in the network through a process called *geometric analysis*, and, (ii) its incremental rebuilding where the connectivity information in the graph is kept persistent with the modifications which occur in the feature space. The rebuilding is done through a process of analysis in both, the connectivity graph and feature space. During this process the connectivity established in the previous iteration is reused as much as possible.

The applications which will benefit form the presence of the incremental connectivity maintenance functionality are the ones where the features involved in the network model are modified frequently. The network model however is expected to maintain the correctness of the connectivity information about the features despite these frequent modifications.

Moreover, during this process we have to consider another important aspect of the connectivity model which is the ability to represent *multi-modal* networks. An example of a multi-modal network appears in transportation systems, where various transportation modes (roads, bus lines etc.) are linked together. To satisfy this requirement we use *connectivity groups* [9] where each connectivity group represents the set of features associated with a given mode of transportation. For example, in a transportation system with two modes (e.g. road network and railway system) there will be two connectivity groups of features. The connectivity is established locally for each group using either the end-point or the mid-span connectivity model (see Figure 5). To create connections between the two connectivity

Algorithm 1 Initial Build

Require: $\mathcal{F}$: The set of all features participating in the model
Ensure: Network model with established connectivity

1: Set $VertexTable \leftarrow \emptyset$, vt_{prev} = 0;
2: **for** each feature f_i in $\mathcal{F}$ **do**
3: $\quad\mathcal{V}$ = GetFeatureVertexes(f_i);
4: $\quad$**for** each vertex v_i in $\mathcal{V}$ **do**
5: $\quad\quad$Insert Into $VertexTable$ values (v_i.x,v_i.y,f_i.id);
6: $\quad$**end for**;
7: **end for**;
8: SortByXY($VertexTable$);
9: **while** GetConnectivityGroup($ConnGroup$) == $true$ **do**
10: $\quad j_i$ = CreateJunction($ConnGroup$.x,$ConnGroup$.y);
11: $\quad$**for** each vertex v_i in $ConnGroup$ **do**
12: $\quad\quad$Update $VertexTable$ set jid = j_i.id;
13: $\quad$**end for**;
14: **end while**;
15: SortByFeatureId($VertexTable$);
16: **for** each record vt_i in $VertexTable$ **do**
17: $\quad$**if** vt_i.fid = vt_{prev}.fid **then**
18: $\quad\quad$CreateEdgeBetweenJunctions(vt_i.jid,vt_{prev}.jid);
19: $\quad vt_{prev} = vt_i$;
20: **end for**;

Table 2. Vertex table description

Column	Description
x	The x coordinate of the vertex
y	The y coordinate of the vertex
fid	Feature identifier
jid	Junction identifier

groups we allow point features to participate in more than one connectivity group. In our example if a point feature (e.g railway station) participates in both connectivity groups it will connect them together in its role as a junction element in the graph (Figure 5).

4.1 Initial Establishment of Connectivity

The algorithm for initial establishment of connectivity takes as input all the features (e.g. all representations of real world objects) in the system. The pseudo code of the algorithm is shown on Figure 1.

The first step in the algorithm is to extract information about the vertexes of all features participating in the network (lines 1-7). The extracted vertex coordinates and the feature identifier are stored in a temporary table called $VertexTable$ (see table 2).

The next step is to sort the content of the vertex table by coordinate values in order to group the coincident vertexes together (line 8) and extract and analyze each group of coincident vertexes according to the connectivity model specified by the user (line 9). For each discovered connectivity group a new junction element is created whose identifier is stored in the column jid in the vertex table for all vertexes participating in that connectivity group (lines 10-13). The content of the vertex table then is resorted by feature identifier so that the vertexes for each line feature are grouped together (line 15). The vertex table is scanned sequentially and for each pair of adjacent vertexes on the same line feature a new edge is created (lines 16-20).

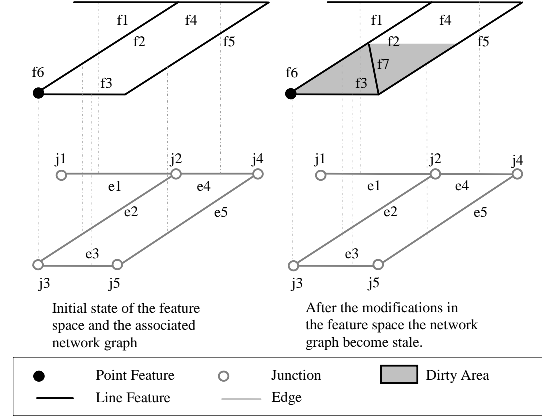


Figure 6. Dirty area example

4.2 The Incremental Approach

We proceed with the description of the incremental rebuilding algorithm and the infrastructure needed to support it (i.e. dirty areas). Finlay we discuss how turn features a rebuild (using the notion of dirty object).

4.2.1 Dirty Areas

In order to track the feature modifications we employ the concept of $dirty\ areas$. A dirty area corresponds to the spatial region within the network where features have been modified and the connectivity model for them has not been re-established. To simplify the computation of the dirty area we define it as a union of the envelopes (e.g. the bounding box around the feature geometry) of the features which have been modified. All dirty areas are stored in a special table called *dirty area table*.

In the initial state, the network model has no features, the underlying network has no elements, and the dirty area is empty. When edits are made to the features or new features are loaded, the dirty area is modified (or new dirty area is created in case of a new feature) to encompass the extend of the feature envelope. Consider the example shown in Figure 6. The left side shows the initial state of the feature space and the network (below). The network correctly represents the connectivity information about the features in the fea-

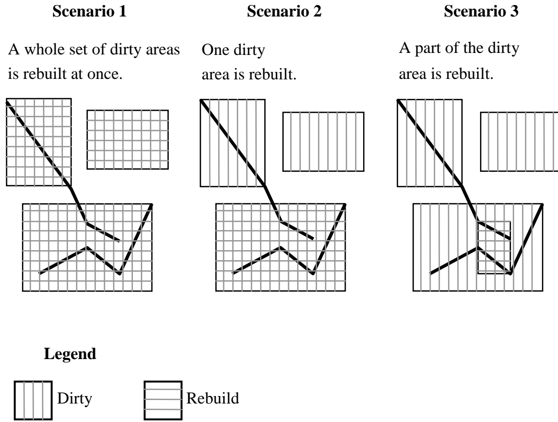


Figure 7. Rebuilding dirty areas

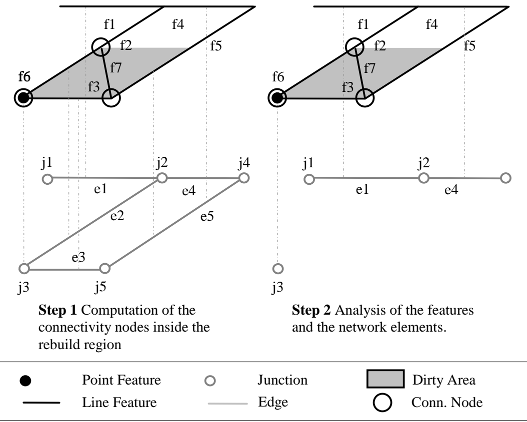


Figure 8. Step 1 and 2 of the incremental rebuild algorithm

ture space. The right side depicts the feature space and the network model after a modification (a new feature $f7$ has been added). However this change has not been propagated to the network model and as a result the model becomes stale and does not describe the connectivity in the feature space correctly (the new line feature $f7$ has not been reflected in the model). To keep track of the modifications which occur in the feature space and which are not propagated to the network model a dirty area is created around feature $f7$ covering the whole feature.

A dirty area (shown as shaded region) is reduced by the process of rebuilding. The rebuilding may occur over a set of dirty areas (figure 7 scenario 1), or it may happen over single dirty area (scenario 2), or it may even involve only parts of dirty area (scenario 3).

The users have the ability to specify *rebuild region* - a spatial extend inside the dirty area which has to be rebuild. When the rebuilding process covers just a subset of the dirty area we have a *partial rebuild*. The portions of the dirty area which are not covered by the rebuild region are referred as *gray region*. When the rebuild process ends, the extend of the rebuild region will be removed from the original dirty area. The network model is assumed correct and up to date when there are no dirty areas.

4.2.2 The Rebuild Algorithm

For simplicity we first describe the rebuilding algorithm when there are no line features that intersect both the rebuilding region and the gray region. The extension of the algorithm to handle partial line features is discussed later in this section.

As with the initial connectivity establishment algorithm, the incremental rebuild constructs a vertex information table and uses it to create the junction and edge elements inside the network model. We can view the network model as containing historical connectivity information for the points

Table 3. Vertex table description

Node	Features connected to this node
1	f_2, f_7
2	f_2, f_3, f_6
3	f_3, f_5, f_7

and lines intersecting the rebuilding region established before the modifications in the feature space. The goal of the rebuild algorithm is to replace that historical information with the current connectivity data.

The input to the algorithm is the set of line features that intersect the rebuilding region. This can be provided by a simple spatial query in the feature space. To illustrate the algorithm we use the simple example shown on figure 6. For simplicity we assume that the rebuild region is identical with the dirty area shown with gray on figure 6. The features which intersect with the rebuild region are line features f_2, f_3, f_5, f_7 and point feature f_6. The first step of the algorithm (lines 2-8) computes the connectivity nodes inside the rebuild region. This is done in a way similar to the initial establishment of connectivity, but now the rebuild algorithm only looks at the line vertexes which belong to the rebuild region and which are stored in the vertex table (it is possible that only parts of the line feature are covered by the rebuild region). All other vertexes are ignored. In our example, we have three connectivity nodes as shown in table 3.

In the second step of the algorithm (lines 9-19), both the features and the network elements are analyzed. For each line feature in the rebuilding region, the algorithm deletes the associated edge elements from the network model. The junction elements connected to those edge elements are analyzed to determine if they satisfy at least one condition from the set of *saving conditions* explained below. If none of these conditions is satisfied, then the junction is deleted.

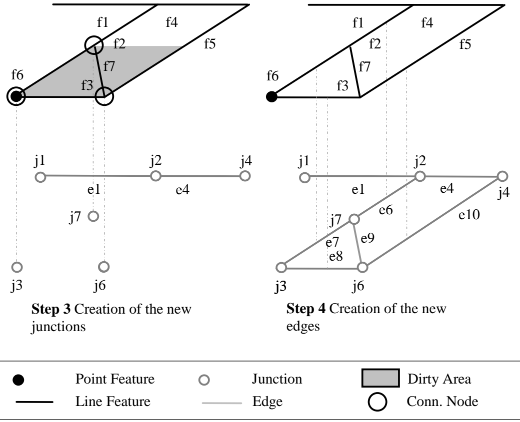


Figure 9. Step 3 and 4 of the incremental rebuild algorithm

The saving conditions are used to determine junction elements that are:

- **Junctions outside the dirty area.** These junctions belong to edges which are partially covered by the rebuild region. They are saved and reused later as connection points through which the rebuild portion of the network is snitched together with the rest of the model.
- **Junctions which have point feature associated.** Since every point feature has a junction associated in the network model such junctions are saved for later reuse. There is no scenario where this junction can become obsolete. It might happen though that we have to update the junction properties like the x and y coordinates.

If a junction is saved for later reuse, the connectivity information for it is stored in the *saved junction table*.

In our example we have edges e_2, e_3 and e_5 associated with line features f_2, f_3 and f_5 which intersect the rebuild region. Those edge elements are removed from the network. The junctions connected to those edges (j_2, j_3, j_4 and j_5) are analyzed to determine if they satisfy one of the saving conditions. Junctions j_2 and j_4 are outside of the rebuild region and thus satisfy the first saving condition and will be used later for resnatching. Junction j_3 has a point feature associated with it and therefore satisfies the second saving condition. Junction j_5 does not satisfy any of the saving conditions and is thus deleted. The connectivity information stored for the saved junctions is shown on table 4.

The third step in the rebuild algorithm (lines 20-28) involves creation of the new junction elements inside the rebuild area. For each connectivity node computed in the first step, there should be a junction element in the network. Care should be taken as some of the processed point features participating in a connectivity node (more specifically

Table 4. Saved junction table

Jct.	j_2	j_2	j_2	j_3	j_3	j_4	j_4
Ft.	f_1	f_2	f_4	f_2	f_3	f_4	f_5

those which were present in the previous iteration) may already have an associated junction element. Instead of creating new junctions, we reuse the saved junction elements. The connectivity information for the newly created junctions is added to the connectivity information for the saved junctions in the $junction\ table$. In our example, connectivity nodes 1 and 3 do not include any point feature so we have to create new junction elements j_6 and j_7 for them. Connectivity node 2, however, includes a point feature f_6 which has a associated junction j_3 and there is no need to create a new one.

The last step (lines 29-34) is the recreation of the edge elements in the rebuild region. For this purpose we use the information for the newly created junctions inside the rebuild region and the saved junctions from step 2 in the saved junction table (those are the junctions used for resnatching with the rest of the network). The information in this table is sorted using the feature id as a primary key so that the junctions that belong to the same feature are grouped together (see table 5). The sorted table is then scanned and for each pair of junctions that belong to the same feature a new edge is created.

The description of the algorithm up to this point covers the basic scenario where there are no features which intersect both: the rebuild region and the gray region (so called ”partial” line features). Nevertheless a point feature cannot intersect both the rebuild and the gray regions since the regions are nonoverlaping. The problem with those line features is that we cannot save the junction at the feature endpoint outside the rebuild region for resnatching if it is in a gray region since this is an indication that the end point has not been processed and junction element may not exist for it.

The incremental rebuild algorithm is thus extended to handle the case where there are ”partial” line features. For each line feature that intersects the rebuilding region, information about its endpoints in the gray region is added to the set of connectivity nodes computed in the first step of the algorithm. During this process we ignore all other feature geometries that may be present there. As a result the connectivity node for this end point may be inaccurate. However, the node is in the part of the dirty area remaining after the current rebuild, and we will correct the inaccuracy with a subsequent rebuild there.

An example is shown in Figure 10 where a new feature f_3 is added to the network model and creates a dirty area. Only a portion of this area is rebuild (the darker region) and

Algorithm 2 Incremental Build

Require: $\mathcal{F}$: The set of line features in the rebuild region
Ensure: Network model with correct connectivity

1: Set $VertexTable \leftarrow \emptyset$, vt_{prev} = 0;
2: **for** each feature f_i in $\mathcal{F}$ **do**
3: $\mathcal{V}$ = GetFeatureVertexes(f_i);
4: **for** each vertex v_i in $\mathcal{V}$ **do**
5: **if** v_i in $rebuildregion$ **then**
6: Insert Into $VertexTable$ values (v_i.x,v_i.y,f_i.id);
7: **end for**;
8: **end for**;
9: **for** each line feature f_i in $\mathcal{F}$ **do**
10: $\mathcal{E}$ = GetAssociatedEdges(f_i);
11: **for** each vertex e_i in $\mathcal{E}$ **do**
12: $|_{\{\nabla\wr\Updownarrow}$ = GetFromJnct(e_i);
13: $|_{\sqcup\wr}$ = GetToJnct(e_i);
14: **if** j_{from} can be saved **then**
15: Insert Into $VertexTable$ values (v_i.x,v_i.y,f_i.id,j_{from}.id);
16: **if** j_{to} can be saved **then**
17: Insert Into $VertexTable$ values (v_i.x,v_i.y,f_i.id,j_{to}.id);
18: **end for**;
19: **end for**;
20: SortByXY($VertexTable$);
21: **while** GetConnectivityGroup($ConnGroup$) == $true$ **do**
22: **if** $ConnGroup$ has junction **then**
23: continue;
24: j_i = CreateJunction($ConnGroup$.x,$ConnGroup$.y);
25: **for** each vertex v_i in $ConnGroup$ **do**
26: Update $VertexTable$ set jid = j_i.id;
27: **end for**;
28: **end while**;
29: SortByFeatureId($VertexTable$);
30: **for** each record vt_i in $VertexTable$ **do**
31: **if** vt_i.fid = vt_{prev}.fid **then**
32: CreateEdgeBetweenJunctions(vt_i.jid,vt_{prev}.jid);
33: $vt_{prev} = vt_i$;
34: **end for**;

only the new feature f_3 intersects with this rebuild region. During the first step of the algorithm both end points of feature f_3 are considered as connectivity nodes though none of them is actually inside the rebuild region. The other features participating in these connectivity nodes (features f_1 and f_2) are not analyzed. The information about the connectivity nodes at the endpoints of the feature f_3 is not complete (we miss the fact that there are junctions associated with these connectivity points). We thus create extra junctions j_5 and j_6 and the edge e_3 will be disconnected form e_1 and e_2. However all these side effects will be fixed in a later iteration of the rebuild algorithm when the dirty areas around the end points of feature f_3 are cleaned.

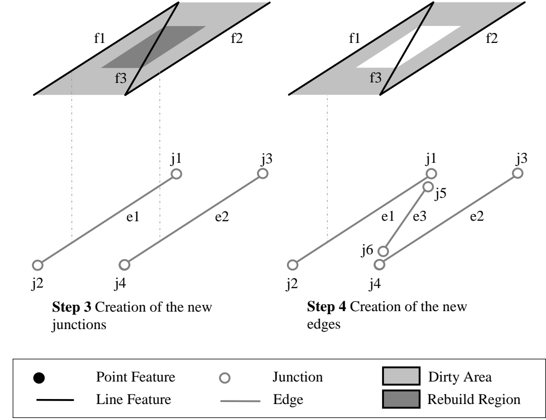


Figure 10. Partial rebuild

Table 5. Sorted junction table

Jct.	j_2	j_7	j_3	j_3	j_5	j_4	j_6	j_4	j_6	j_7
Ft.	f_2	f_2	f_2	f_3	f_3	f_5	f_5	f_5	f_7	f_7

4.2.3 Rebuilding Turn Features

We now discuss how the turn features participating in the network are rebuild. The problem comes from the fact that the turn features (or objects like relationships and so on) are defined as a relation between two or more line features and typically do not have geometrical properties. In our implementation of the network model the information about the turns is stored in a separate turn table with references to the line feature table.

The lack of spatial properties makes it difficult to determine if a turn feature is inside a rebuild region or not. It is possible to find the line features inside the rebuild region and check for each one of them if it participates in a turn feature. However this will require: (1) a query with spatial range predicate (to find the line features in the rebuild area) and (2) numerical join for each column in the turn table containing line features identifiers (to find turns associated with given line feature). This will be inefficient because of the multiple join operators involved and the fact that in the existing geodatabase engines the execution of a spatial predicate and a numerical join are not pipelined.

Instead we propose to extend the dirty area concept to cover network elements without geometrical properties. We thus introduce the notion of *dirty object*. We define a dirty object to be an object without geometrical properties whose modifications has not been propagated to the network. When an object is modified we store its identifier in a *dirty object table*. During the rebuild process we attempt to recreate all objects in the Dirty Table into the Network. If we succeed, we remove the recreated dirty object from

the table. If we fail, we keep it there till it is successfully recreated. For the specific task of rebuilding turn features, we mark turn as a dirty when (i) the turn feature is directly modified (Insert, Update, Delete) or (ii) the associated line features are modified (Update, Delete) or (iii) the associated network turn element is deleted (This may happen during the rebuild process).

5 Implementation Experiences

In this section we discuss our experience with implementing ant testing the algorithms maintaining the connectivity inside a multimodal dynamic network. The proposed algorithms have been implemented and shipped with the ESRI ArcGIS.

We first present a performance evaluation of both the initial build and the incremental rebuild algorithms for establishing several different sized networks; the results are shown in Table 6. Here the initial algorithm is used to build a network with the specified size while the incremental rebuild algorithm is used to establish the connectivity for the same dataset by artificially creating a dirty area which covers the *whole* feature space. Hence, this experiment serves as a worst case scenario for the incremental rebuild algorithm (in practice dirty areas are small percentage of the whole space).

Clearly the initial build algorithm is much faster (about ten times) than the incremental build algorithm for building the whole network. The main reason is that in the initial build algorithm only the feature space is analyzed while during the incremental rebuilding we analyze both the feature space and the logical network storing the connectivity. For each edge in the incrementally rebuild region we have an extra query which retrieves the associated edge as well as the endpoint junctions for this edge element. Furthermore the junction elements are analyzed to determine the number of edges associated to them (see algorithm 2 for details). This requires another forward star query for each junction.

In the general case (i.e., when the dirty area is part of the whole spatial extent) the initial build algorithm can also take advantage of sequential scanning over the feature tables retrieving the rows one by one. In the incremental rebuild algorithm however we analyze only the features that intersect with the rebuild region. Thus we have to perform a range query over the feature space. To improve the performance of this operation we utilize a tree-based spatial index for the incremental algorithm.

However having the incremental rebuild functionality is extremely useful in great number of practical scenarios. This is due to the fact that the amortized cost of maintaining an incrementally rebuildable network in practice is far less than an ordinary network that must be periodically rebuilt from scratch. Typically the user modifications are clustered in the spatial domain. They also involve relatively small number of features. On average a regular rebuild operation which involves 20 or less modified features takes around one second to complete. Figure 11 depicts the total cost of maintaining connectivity using three different approaches, under an increasing number of rebuilds. We used the Durango dataset described in table 6 for this experiment.

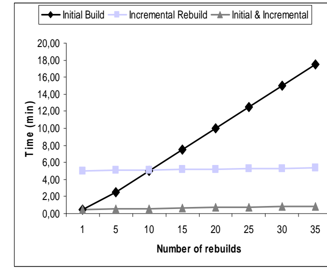


Figure 11. The total cost of maintaining connectivity

The first approach uses only the initial build algorithm to establish and maintain the connectivity. Hence this represents how current systems will reestablish connectivity by rebuilding the whole network from scratch. Thus the time this algorithm takes is linear to the number of rebuilds. The second approach uses only the incremental rebuild algorithm for both the establishment and the maintenance of the network (that is, the incremental algorithm was used to establish even the initial connectivity, similarly to Table 6). Observe that the incremental approach is practically independent of the number of rebuilds (given that in this experiment the dirty areas are relatively small, as it is usually the case in practice). The third approach is a combination which uses the initial build algorithm to establish the connectivity and then uses the incremental rebuild for network maintenance (this corresponds to the way our system is implemented). Clearly the last approach is the most advantageous one combining, the fast initial network establishment with the fast incremental maintenance.

We also examined how the incremental algorithm behaves as the size of the dirty area increases. The Durango dataset was again used. Table 7 shows the results. Here the dirty area is expressed as a percentage of the total network spatial extend. Even though typically the dirty areas cover less than 1% of the spatial extent, in this experiment we created artificial modifications to cover up to 15% of that space. As it can be seen, the advantages of the incremental rebuild algorithm are present even for large covers (typically around 10-15%). We experimented both with localized as well as scattered modifications over the spatial extent. Even though scattered updates are expected to take more time (more range queries, non sequential I/O) the performance of the incremental algorithm is not greatly affected, making it a robust solution for both environments.

Table 6. Vertex table description

Dataset	Features	Vertexes	Net. Elements	Build Time	Rebuild time
Durango	13,523	33,556	22,349	0.5 minutes	3 minutes
Inland empire	123,753	254,354	254,637	1.1 minutes	10 minutes
Philadelphia metro area	233,445	454,457	435,257	1.5 minutes	13 minutes
Paris metro area	403,040	834,024	734,234	3 minutes	27 minutes

Table 7. Rebuild time

Type of modifications	1%	5%	10%	15%
Localized updates	4 sec	10 sec	17 sec	32 sec
Scattered updates	4 sec	11 sec	18 sec	33 sec

6 Conclusion

Using existing functionality provided by a relational DBMS (tables, joins etc.) we presented algorithms and appropriate infrastructure (dirty areas, dirty objects and management policies for them) needed to incrementally maintain connectivity of a dynamic multimodal network. This is required by applications which frequently edit their network data. The proposed solution provides for fast maintenance while existing approaches need to rebuild connectivity from scratch. We are currently extending our work to support a versioning mechanism for network datasets. As future work we plan to extend our network model maintenance algorithms within a distributed database environment; moreover, we will examine how to incorporate features without geometry (so called *aspatial features*) into the model.

References

[1] Tom Caldwell. On finding minimum routes in a network with turn penalties. *Commun. ACM*, 4(2):107–108, 1961.

[2] Hyung-Ju Cho and Chin-Wan Chung. An efficient and scalable approach to cnn queries in a road network. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 865–876. VLDB Endowment, 2005.

[3] Caliper Corporation. *TransCAD Transportation GIS Software Ref. Manual*. Caliper Corporation, 1996.

[4] K. Dueker and A. Butler. Gis-t enterprise data model with suggested implementation choices. *Journal of the Urban and Regional Information Systems*, 10(1):12–36, 1998.

[5] J. Evans and E. Minieka. *Optimization Algorithms for Networks and Graphs*. Dekker, Marcel Incorporated, 1992.

[6] M. Goodchild. Geographic information systems and disaggregate transportation modeling. *Geographical Systems*, 5(1–2):19–44, 1998.

[7] S.-L. Shaw H. Miller. *Geographic Information Systems for Transportation: Principles and Applications*. Oxford University Press US, 2001.

[8] C. Hage, C. Jensen, T. Pedersen, L. Speicys, and I. Timko. Integrated data management for mobile services in the real world. In *Proc. of Very Large Data Bases (VLDB)*, pages 1019–1030, 2003.

[9] E. Hoel, W.L. Heng, and D. Honeycutt. High performance multimodal networks. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 308–327, 2005.

[10] Erik G. Hoel, Sudhakar Menon, and Scott Morehouse. Building a robust relational implementation of topology. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 508–524, 2003.

[11] C. Jensen, T. Pedersen, L. Speicys, and I. Timko. Data modeling for mobile services in the real world. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 1–9, 2003.

[12] P. Longley, M. Goodchild, D. Maguire, and D. Rhind. *Geographical Information Systems, Principles, Techniques, Applications and Management*. Wiley Publishing Company, 1999.

[13] M. Mainguenaud. Modelling of the geographical information system network component. *Int. Journal of Geographical Information Systems, Vol 9(6)*, pages 575–593, 1995.

[14] S. Morehouse. Arc/info: a geo-relational model for spatial information. *Proceedings of AUTOCARTO 8. ASPRS, Falls Church Virginia*, pages 388–397, 1985.

[15] Oracle Corp. *Oracle Database 10g: Oracle Spatial Network Data Model: Technical white paper*, May 2005.

[16] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *Proc. of Very Large Data Bases (VLDB)*, pages 802–813, 2003.

[17] B. Ralston. Gis and its traffic assignment: Issues in dynamic user-optimal assignments. *Geoinformatica*, 4(2):231–243, 2000.

[18] Cyrus Shahabi, Mohammad R. Kolahdouzan, and Mehdi Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *GIS '02: Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 94–100. ACM Press, 2002.

[19] Shashi Shekhar and Duen-Ren Liu. Ccam: A connectivity-clustered access method for networks and network computations. *IEEE Transactions on Knowledge and Data Engineering*, 09(1):102–119, 1997.

[20] F. Southworth and B. Peterson. Intermodal and international freight network modeling. *Geographic Information Systems in Transportation Research*, 8:147–166, 2000.

[21] Stephan Winter. Modeling costs of turns in route planning. *Geoinformatica*, 6(4):345–361, 2002.