Spatial database for fire department

Mahboobeh Abdal Mahmood Abadi 1, Mehran Nasseri 1

1 ICT Kista campus, KTH, Stockholm, Sweden
{maama, mehrann}@kth.se

Abstract. The project aimed to develop a geographical information system GIS. It helps users in emergency call center to locate the on-fire building on the map and then finding the nearest hydrant and fire station. It also finds shortest path with lowest cost, based on travelling distance, from both hydrant and fire station to the on fire building; also it considers blocked streets if any. Moreover, the application is able to provide statistical information about fires that happened in a specific region on specific time period, include number of times a building was on fire and which station dispatched to rescue. Additionally, by clicking on each building fire fighters can find out about the number of floors and residents in the building, so they can make more precise decision about the number of vehicles, and rescuers that must be dispatched to the location.

1 Introduction

When it comes to rescue people who are in danger, it is important to make a correct decision in timely manner. It is vital to facilitate emergency centers with an instructive information system to store data and extract useful information from it. Such system can help rescuers to have timely reaction to the incident and save as much life as possible.

To achieve this objective we developed a GIS application that utilizes oracle 11g to create and manage geo-spatial database. The Spatial option in 11g is intended to make spatial data administration easier to GIS applications. Once data is stored in database, it can be easily managed, retrieved, and associated to the other data.

Furthermore, the Jmapviewer library is used to import map tiles from the ‘open street map’ to the application and paint map markers based on the user queries.

2 Implementation

2.1 Database

The application stores and retrieve data from six tables named as follow:

- Buildings (building_ID, Building_name, onfire, no_residents, floors, geom): Building_name indicates the address. Onfire gets binary values to represent whether the building is on fire right now or not.
- Firestations (building_ID, Station_name, geom)
- Waterresources (waterresourcesindex, address, geom)
• Fire history (building_id, station_id, fire_date): in which building_id references buildings table and station_id references Firestations table.
• Tables to create network data model and perform shortest distance analysis
  — Fire_Node$(node_id, node_name, node_type, active, partition_id, geometry)
  — Fire_link$(link_id, link_name, start_node_id, end_node_id, link_type, active, link_level, geometry, cost):
    These tables represent data about the streets. Start_node_id and end_node_id references Fire_Node$ table to show start and end of street. The ‘cost’ column corresponds to length of street, which is used to calculate the shortest path using the Dijkstra algorithm. The ‘active’ column indicates if a street is blocked. The partition_id is used to divide data into partitions and load them to memory on demand by the dijkstraShortestPath(PointOnNet, PointOnNet, NetConstraint) method.

Data feeding.
Another special application was developed to find and store geometry data of the buildings, fire stations, hydrants, street nodes and links. The application gets the geographical coordination (Lon/Lat) of the clicked point on the map and stores them in the appropriate text files (one text file for each table). In case of finding links (streets geometries) sequence of right/left clicks illustrate connections between clicked points on the map. The application also calculates and stores the distance between each connected nodes. The distances are then imported in the cost column of the Fire_link$ table to be used to calculate the shortest path. DataSupplier class reads data from these files to populate the database.

Indexing geometry data.
All geometry data is indexed using R-tree index. R-tree indexing preferred over quad tree indexing since [1]:
• R-tree has shown better performance in finding nearest neighbors and ‘within’ query, which are used a lot in the application
• R-tree index can be used to index whole world and requires less storage compare to quad tree.
• Heavy updates have no effect on performance of quad tree, while it may reduce the performance of R-tree until rebuilding the index. Given that the application does not require heavy updates, the R-tree will have better trade off.

2.2 Functionalities
Query tab.
The application enables the emergency center to locate the fire on the map and find the closest hydrant, and fire station to the on fire building. To locate the fire user needs to select the ‘locate fire’ radio button and enter the address. The address field enabled with auto address completion to make address entry much faster and more accurate. After clicking the submit button the map zooms on the building, mark the building with a red dot. The application also shows the distance in Km and travel time from the nearest fire station in the info text area, and finally draws the shortest path.
between fire station and building on the map with a blue line. The distance and travel
time is calculated based on cost column of the Fire_linke$ table. Dijkstra algorithm is
used to find shortest path. Furthermore, by left clicking the red dot a pop up window
will show information regarding number of residents and floors of the building. Such
information can help to decide which type of truck should be dispatched, how many
ambulances may be needed and what extra facilities may be required (e.g. ladder,
rescue mattress, etc.).

To find the closest hydrant user should select the `find closest hydrant` and then
left clicking on any building will query database for two nearest hydrants. It also finds
and draws the shortest path along with information about distance and travel time.

To find the closest hydrant or fire station the application finds two nearest of each
to the building. Then it draws shortest navigation line on streets toward the one with
shortest travel time using dijkstra. The idea is that the nearest function SDO_NN can
find the nearest hydrant/fire station by assuming the direct path between them. There-
fore, the nearest node returned by SDO_NN is not always nearest node considering the
traveling distance and obligatory turns on streets. For example, assume that there is a
hydrant 10 meters far from the bounding box of the building but located behind it. So
you have to travel around the building for let say 200 meters to access it. Assume that
there is another hydrant 50 meters far from the building’s bounding box and you have
to travel only 70 meters (considering few turns on streets) to access it. The SDO_NN
function will return first one as nearest even though it is not the nearest in reality due
to obligatory turns on streets. Therefore, the application decides to draw shortest path
on the streets that direct to the second hydrant.

After extinguishing the fire the user can remove the on fire building from the list
of on fire buildings and update the `onfire` column of the buildings table by selecting
extinguished radio button, inserting address, and pushing submit button.

Moreover, the `locate fire` query is responsible to record the history of the fires in
`fire_history` table. The records include the date of fire, building id, and the fire sta-
tion id that put out the fire.

The last provided feature in this tab includes **indicating blocked streets** due to
maintenance or accident. The user can simply select the closed street radio button and
locate the blocked street by writing the address in provided text area. After pushing
the submit button the specified street would be set to be inactive, in table fire_link$.
As a result the shortest path analyzer will try to find the shortest path to destination
that is not passing through the blocked street.

**Load on demand (LOD)**

Since running the dijkstra algorithm requires many query to database the java net-
work data model (ndm) provides LOD feature to load network data, include nodes
and links, to the memory on demand. Therefore, dijkstra queries will be sent to the
memory instead. To prevent loading all network data into memory, LOD uses parti-
tion_ID column in fire_node$ table to load only those partitions of network data that
are required for shortest path analysis. Because of this, after any update in the
fire_node$ and fire_link$, due to blocking and unblocking a street, we update the data
in the memory as well. This achieved by writing and invoking a method called
networkupdater() after each update in database.
Report tab.

In the report tab user can select among four check boxes to get information about the location of hydrants, fire stations, on fire buildings, and buildings that was on fire.

There are two possible range available to query the database namely, whole region and limited range. In case of **whole region** the application finds and marks all objects on the map. However, if the **range** is selected the user should draw a polygon on the map, so the database will only return objects within that polygon. To draw the polygon user should left click, on different locations, on the map. Each left click will draw a line between previous click and the current one. Finally a right click will close the polygon by connecting last clicked point to the first one. After pushing the search button and sending the query to the database the result will be shown by dot markers on the map. The markers for the on fire buildings are red, for the fire stations are pink, for the hydrants are blue, and for the buildings that were on fire are green.

Additionally if the ‘**extinguished**’ check button be selected the user can define the date interval for the search query. Therefore, only buildings that were on fire on a specific period of time will be returned. This information will be shown on the info text area, which includes number of times a building was on fire, which fire station dispatched to rescue and in which dates.

3 Further work

To improve the application we can add more constraint columns than just ‘cost’ and ‘active’ to the fire_link$. These can be turn restriction, one way street, max speed, and etc. These constraints then should be taken into account when calculating the shortest path. At the moment the traveling distance (column cost) and blocked street due to maintenance (column active) is the only factor that is considered in calculation of the shortest path.

In addition, finding only two or more nearest node to the source and then exploring the shortest path among them is not an ideal solution. Our experiment confirmed a situation in which, the fifth nearest node can have shorter path than the first nearest node. One idea to solve this problem can be to get the shortest path between building and nearest hydrant/fire station and set this as maximum cost. Then calculate shortest path to the second nearest one. Whenever the cost exceeds the maximum cost we should abort the dijkstra algorithm from further analyzing the rest of the path to the destination. However, if the cost to the destination is less than the maximum, it should be set as new maximum cost. Albeit, question still remains that for how many nearest nodes to the source we have to do this in order to find the closest one in terms of traveling distance. Developing an efficient algorithm to find the nearest node based on traveling time can be another interesting project.

Moreover, before implementing any application it is vital to perform a comprehensive requirement analysis to define the functional and non functional requirements. This must be done by conducting a qualitative research by interviewing end users and managers to understand what they need. For such application it is also important to work on user friendliness and information quality of the application. Failing to show the correct information, difficult navigation, complexity of working with application, and etc can greatly reduce effectiveness and efficiency of such application.
References