GSB/RF
November 17, 2003

Project 3 — A GIS system

1 Introduction

The goal of this project is to implement a rudimentary Geographical Information System. The intent is solve a realistic problem on a real-world set of data.

The data will be the 2002 TIGER/Line files from the US Bureau of Census. The files contain a large collection of annotated points, lines and polygons, which together form a detailed map of the United States.

The basic task is to make a program which can graphically display the data and which can find shortest routes in the data. For sake of overview, the visualization task and the route finding task will be described separately.

The project is intentionally left quite open. We give a required minimum, but students are free to make a more advanced solution. Some ideas for possible enhancements are mentioned later. Some of the choices left open may be resolved by experimentation. Such experimentation is encouraged (and so is its documentation in the report).

The program should be implemented in C, C++, or Java (others on request).

2 The TIGER/Line Data

The TIGER/Line data set is produced by the US Census Bureau (`www.census.gov`). The following description of TIGER/Line is taken from the Bureau's web page for the TIGER/Line data set (`www.census.gov/geo/www/tiger`).

> The TIGER/Line files are a digital database of geographic features, such as roads, railroads, rivers, lakes, political boundaries, census statistical boundaries, etc. covering the entire United States. The data base contains information about these features such as their location in latitude and longitude, the name, the type of feature, address ranges for most streets, the geographic relationship to other features, and other related information. They are the public product created from the Census Bureau's TIGER (Topologically Integrated Geographic Encoding and Referencing) data base of geographic information. TIGER was developed at the Census Bureau to support the mapping and related geographic activities required by the decennial census and sample survey programs.
>
> These files are not graphic images of maps, but rather digital data describing geographic features. To make use of these data, a user must have mapping or Geographic Information System (GIS) software that can import TIGER/Line data. The Census Bureau does not provide these data in any vendor-specific format. With the appropriate software a user can produce maps ranging in detail from a neighborhood street map to a map of the United States. To date, many local governments have used the TIGER data in applications requiring digital street maps. Software companies have created products

for the personal computer that allow consumers to produce their own detailed maps. There are many other possibilities.

The 2002 TIGER/Line data set is available from the web site of the U. S. Census Bureau:

`www.census.gov/geo/www/tiger/tiger2002/tgr2002.html`

The 2002 TIGER/Line data set consists of 3.7 Gb of compressed spatial data, which expands to estimated 33 Gb data. The data for New York state (NY) (`/tiger2002/NY/`) is 99 MB and uncompresses to 900 MB.

The documentation for the data files can be found at:

`www.census.gov/geo/www/tiger/tiger2002/tgr2002.pdf`

This document is rather long (305 pages). The necessary and sufficient parts for this project are:

Chapter	I	1	2	3	5	6
Pages	1-2, 8	1–10, 15–17	1–3	25–43	6, 9	1–3

The rest is mostly concerned with information relevant for census making, such as address ranges and county borders, and with changes from previous releases.

A local copy of the 2002 TIGER/Line files is available at `/users/gerth/tiger2002/`.

3 Visualization

The goal of this part of the project is to build a data structure holding a suitable subset of the TIGER/Line data, and to make a program which takes a specification of a rectangle – a *window* – and graphically displays the part of the data contained within the window.

3.1 The Minimal Requirements

- The data should consist of the lines for the roads, the railroads and the waterways (Feature Class A, B and H, see Section 3 of `tgr2002.pdf`). Only the actual lines are mandatory, i.e. no annotation needs to be shown, and no polygons defined by the lines need to be considered as separate entities. The necessary information for this is contained in the records of type 1 and 2. The complete set of points for a polygon line (a *chain* in TIGER terms) is obtained by merging the information in the files for the two types of records. For this, they files should first be sorted according to the TLID key.
- The data should be stored in a static R-tree. The R-tree should be build in a bottom-up fashion on the set of lines sorted according to the position of the centers of their bounding boxes along the Z-curve. This is equivalent to sorting on a key obtained by interleaving the bits of the x- and y coordinates of the centers.
- A query program should be made which takes two points and displays the lines or part of lines contained in the rectangle spanned by these two points. The program does not have to be interactive.

- The method of generating graphical output may be chosen rather freely. A graphical library may be used for direct on-screen output, or Postscript or LaTeX-files may be generated. Other methods may be approved on request.
- The amount of data should be such that the data structure storing them takes up more than 0.5 Gb of disk space. The amount should be adjusted by choosing a proper number of (neighboring) states. The state New York should be included.

3.2 Some Possible Enhancements

- Store more types of data (e.g. landmark and polygon features).
- Show annotations (names, etc.).
- Make an interactive query program, with navigational features.
- Make a zoom facility which adjusts the level of detail according to the size of the window. This may be implemented by grouping the subclasses of the feature classes according to some judgment about what level of detail they belong to. Only the classes for the appropriate level should then be shown.

4 Route Finding

The goal of this part of the project is to add capability for finding shortest paths in the maps visualized in the previous task.

The basic task is to implement Dijkstras algorithm on the graph defined by the road data of the TIGER/Line files, and to try to do this in an I/O-efficient way. Other tasks necessary is to extract the graph from the data, and to implement facilities for defining a departure and a destination point, and for returning the result.

The underlying assumption for the solution described here is that the graph is too large for internal memory, but the working set (the *front* below) of Dijkstras algorithm hopefully is not.

4.1 The Minimal Requirements

- The basic algorithm should be the well-known algorithm of Dijkstra, which finds single source shortest paths in a weighted graph with no negative edge weights. It explores the graph in a closest-nodes-first manner from a source node. During execution, it maintains the nodes partitioned into three sets: the *not yet encountered*, the *finished*, and the *front* in between. It is often implemented using a priority queue supporting updates of priorities, with the priority being the shortest known distance to the node so far. The following is the pseudo-code for a version which stops as soon as a specified target node is finished, and which keeps only the front in the priority queue. This version should be used.

```
v := source node
INSERT(v,0)        (* parameters: node, priority *)
while v ≠ target node
    (v, d) := DELETEMIN()        (* v is finished, d is true shortest distance *)
    for each neighbor u of v
        if u not yet encountered
            INSERT(u, d + cost(v, u))        (* move u to front *)
        else
            if u still in priority queue and d + cost(v, u) < priority of u
                UPDATE(u, d + cost(v, u))        (* shorter path found *)
```

To traverse the actual shortest path afterwards, each node should have a backpointer to the previous node on some shortest path from the source node. This should be updated whenever the priority is set or updated in the algorithm.

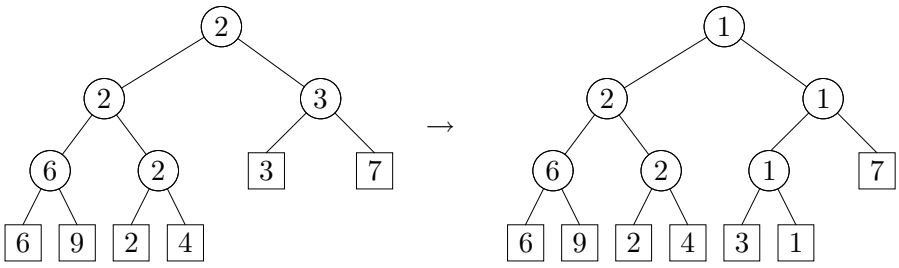
- For greater efficiency, you should add the so-called A^*-heuristic [1]. This consist of assigning all nodes v in the graph a potential $\Psi(v)$ equal to the Euclidean distance from v to the destination node. During execution, all edge costs is then adjusted as follows: if a directed edge (v, u) has original cost c, an adjusted cost of

$$\Psi(u) - \Psi(v) + c$$

  is used. Any path shortest with respect to the adjusted costs will also be a shortest path with respect to the original costs. Furthermore, no adjusted edge costs will be negative, so Dijkstras algorithm still works.

- As priority queues normally do not support searching for elements, the UPDATE operation must as argument take a pointer to the position of the element in the queue. A simple and space efficient priority queue is the heap, implemented in an array. However, when pointers to positions are required, the bubble-up and -down routines require these pointers to be updated for the affected elements.

  Therefore, a tournament-version of the heap should be used. This is still a binary tree implemented in an array. The internal nodes form a heap-shaped tree. The leaves contain the elements and their priorities, and the internal nodes contain the value of the minimal priority in any leaf in its subtree. When a change happens at a leaf, the information on the path to the root should be updated in an bottom-up fashion, but unlike the bubble-up and -down routines, this will not change the position of any leaf element. This is all that is done during UPDATE. Insertions are done by making the leaf of lowest index an internal node, inserting the previously contained element and the new elements as its children. The following figure illustrates this:



Only the pointer to the moved leaf has to be updated. The DeleteMin operation is performed by searching from the root using the information in the internal nodes. When the leaf in question is found, it is replaced by the leaf with the largest index, and the reverse of the process above is performed. Two pointers should be updated, as should the information on two leaf-to-root paths.

The array should be given some initially, reasonably large size (like 1Mb). If the current size is exceeded, the heap should be rebuilt with an array twice as large.

- The nodes of the graph should consist of the *start* and *end* nodes of the complete chains which are part of roads. Roads are here defined as all line features of Class A, except for the subclasses A5 and A7. The complete chains themselves constitute the edges of the graph—we consider the graph as directed, so each chain represents two edges. This information is contained in records of Type 1.

- The cost of an edge should be the length of the chain represented, seen as an polygonal line. In other words, it should be the sum of the lengths of the line segments it is made of. To calculate this, the information in records of Type 2 must be taken into account, so the two types of files should be sorted on TLID key, and then "merged", just as in the first part of the project.

- The line segments of a chain are seen as straight lines in $\mathbb{R}^3$ (as the roads are on a sphere, this is not true, but the approximation is probably very good, as the line segments are short compared to the circumference of the earth), so the length of a segment is the Euclidean distance between its two endpoints. Recall that the Euclidean distance between (x_1, y_1, z_1) and (x_2, y_2, z_2) is
$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}.$$

- In the TIGER/Line data, points are described by latitude and longitude, which has to be transformed into points in $\mathbb{R}^3$. This is done by mapping the point with latitude $s \in [-90; 90]$ and longitude $t \in [-180; 180]$ to
$$\begin{aligned} x &= r \cdot \cos s \cdot \cos t, \\ y &= r \cdot \cos s \cdot \sin t, \\ z &= r \cdot \sin s, \end{aligned}$$
where r is the radius of the earth, which is 6378 km (at equator, that is, but we assume that the earth is a perfect sphere - in reality, the radius at other latitudes differs from this value, but never by more than 1 percent). Note that the values for latitude and longitude in the TIGER/Line data should be divided by 10^6 to get degrees (and don't forget that the standard math library in C uses radians (360 degrees equal 2π radians)). Of course, all numbers should be doubles, to keep the precision as high as possible.

- The graph should be represented by a list of nodes, with each node followed by its adjacency list. The nodes should be sorted according to their order along the Z-curve (i.e. sorted on a key made by bit-interleaving the latitude and longitude coordinates), to utilize the fact that a good deal of spatial locality can be expected from a graph representing roads. This list is kept in an external file, which is mapped to virtual memory by `mmap` before use.

  The specific contents of this graph representation is, for each node v:

  - Its latitude and longitude coordinates.
  - A pointer (an index) to its location in the priority queue—this is set to nil after a DeleteMin.
  - A timestamp for the last time the above pointer was set. This is needed because the contents of pointers could come from a previous invocation of Dijkstras algorithm. A timestamp from before the current start of the algorithm indicates a not yet encountered node.
  - A back pointer for traversing the shortest path afterwards.
  - The number of neighbors.
  - For each neighbor u: the cost of the edge (v, u) and the index of the byte in the file representing the graph where the information for the node u starts.

  In the priority queue, nodes are represented by their index in the file.

- The graph representation is made once by a preprocessing step and saved for future shortest path calculations. This is done by a series of sorts and scans:
  - The information in Type 1 and Type 2 records is merged using their TLID keys as in the first part of the project, and simultaneously the edge costs is calculated. Two copies for each edge is made, one for each orientation.
  - The resulting stream of edges is sorted according to Z-curve position of the start nodes of the edges.
  - The sorted stream is traversed, and for each node, the eventual index (position of first byte) is calculated. The result is a stream of pairs of Z-curve positions and indexes in file.
  - Sort the stream of edges according to Z-curve position of the end nodes of the edges. During a "merge" with the previous stream, Z-curve positions of end nodes are exchanged with indexes.
  - Finally, the stream of edges is again sorted according to Z-curve position of the start nodes of the edges. This stream is scanned and the final graph representation can be made.

- The program from the first part should be augmented with facilities for the user to specify source and target nodes. Possible ways of doing this could be:
  - The user specifies a point on the map, either by the mouse (if a GUI is used) or by giving coordinates at a shell prompt. The program then tries to find the closest node by doing a binary search (according to position along the Z-curve) in the graph representation to find the position clicked on, and then finds the physically closest node within a fixed number (say, one hundred) of nodes surrounding that point in the file. An auxiliary file containing just the indexes (in order) of the nodes in the graph representation will be helpful here (as the start of a nodes information is not detectable in the representation itself).
  - Alternatively, the nearest neighbor (which is a node in the road graph) of a specified point can be searched for (by a backtracking search process) in the R-tree from the first part of the project. Then the leaves of the R-tree should store indexes from the graph representation.

  - If using a GUI, the nodes (which are endpoints of some line segments) can be made "clickable" by representing them graphically by widgets knowing their index in the graph representation.

- The result should be specified by indicating the shortest path to the user. This could be by coloring the path in a GUI, or by listing the coordinates of its nodes at the prompt.

4.2 Some Possible Enhancements and Experiments

Among the possible enhancements and experiments are the following. Further creativity is encouraged.

- Test to what extent the A^*-heuristic improves performance (by removing it and comparing performance).
- Nodes of degree two are redundant. Count their number, and if it is large, reduce the size of the file by contracting these nodes (by an appropriate series of sorts and scans).
- Exchange the use of a Z-curve with the use of a Hilbert curve, and measure which performs best.

5 Deadline

The deadline for the entire project is Friday December 19, 2003.

References

[1] R. Sedgewick and J. S. Vitter. Shortest Paths in Euclidean Graphs. *Algorithmica*, 1:31–48, 1986.