GSB/RF
November 23, 2000

Project 3B — Route Finding

1 Introduction

The goal of this part of the project is to add capability for finding shortest paths in the maps visualized in the first part of Project 3.

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.

2 The 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.

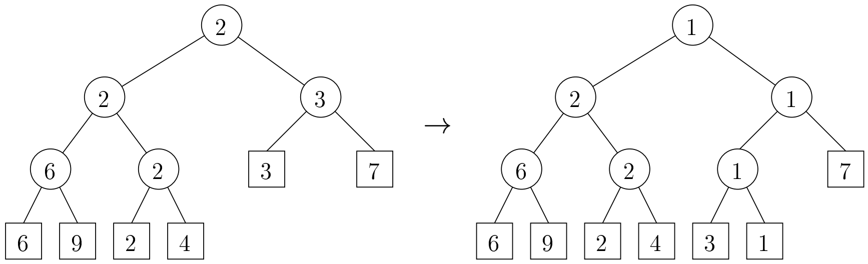
- For greater efficiency, you should add the so-called A^*-heuristic [1]. This consist of assigning all nodes v in the graph a potential $\Psi(v)$ equal to the Euclidean distance from v to the destination node. During execution, all edge costs is then adjusted as follows: if a directed edge (v, u) has 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 merging of the information in the Type 1 and Type 2, and the simultaneously calculation of the edge costs. 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 these nodes should store the an index 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.

3 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.

- Accessing the graph representation in a read-only manner is probably more efficient. Try to implement the algorithm in this way. Hints: Use a hash table to make the priority queue searchable (store the pointers for the current members of the priority in a hash table indexed by latitude and longitude coordinates). When inserting elements, move a copy of their neighbor list along, for use in the last two lines of the pseudo-code of the algorithm. During a DELETEMIN, the neighbors of the removed node still in queue should have this node removed from their neighbor list. In this way, we do not need to record the status of finished nodes anywhere (recall, we try to use internal memory only for the *front*). Finding the actual path afterwards can be done by adding all edges, which in the algorithm triggered an UPDATE operation, to a stream (i.e. whenever the algorithm sets the backpointer, an edge is added to the stream). Scanning through this stream will allow the path to be output in time bounded by the time already used. If this solution is implemented, the information stored in the graph representation can be reduced accordingly.

4 Extended Deadline

As a consequence of the addition of this second part to Project 3, the deadline for the entire project is extended to Monday, January 15, 2001. The report should describe both parts of the project.

References

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