Automatic Packet Reporting System: Building a Large Scale Geospatial Database

James Jefferson Jarvis KB0THN
kb0thn@aprsworld.net

Abstract

The purpose of my research is collecting and analyzing position data from amateur radio's Automatic Packet Reporting System (APRS). APRS equipped vehicles transmit their latitude, longitude, course, and speed. This data is received via VHF radio and aggregated into the APRS Internet stream.

I designed and implemented a system that automatically collects and catalogs the entire APRS Internet stream into a relational database. Because APRS data is expressed in various formats, I wrote a parser that translates the data to a common format. My software collected more then 52,000,000 data points in 3 months.

I developed software to analyze position data. One program determines what polygon latitude / longitude points falls inside, and locates a station within a political boundary. For use in detecting roadway traffic problems, another program searches digital maps for the distance to the nearest road. Overall I wrote approximately 10,000 lines of computer source code. Currently I am investigating using neural networks to detect characterize roads and be able to detect anomalies, such as traffic jams.

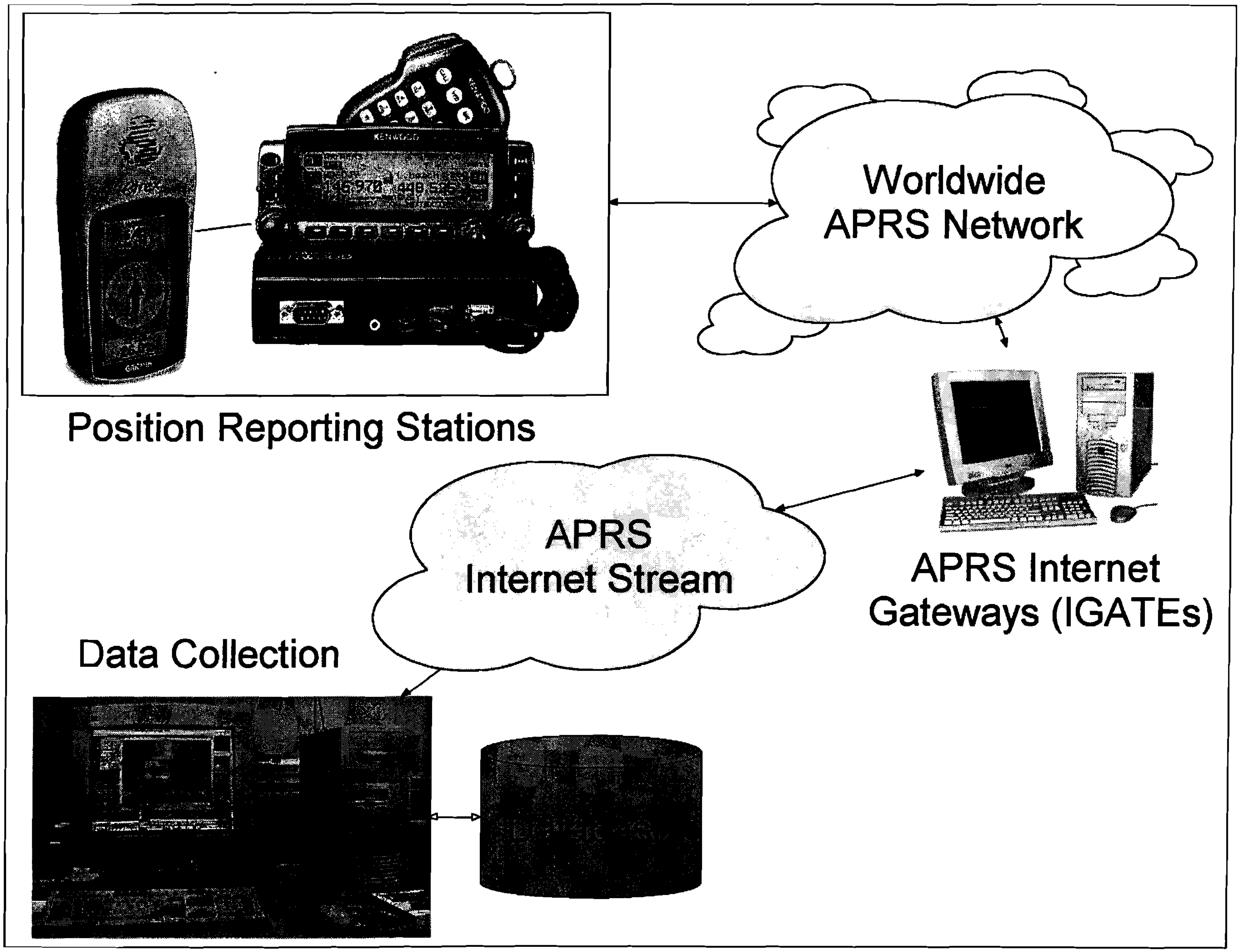
Keywords: APRS, datamining, database, GIS

Introduction

Thousands of amateur radio operators drive around the United States each day transmitting their latitude and longitude to anyone listening. Tens of thousands of companies use Automatic Vehicle Locating technology to dispatch and track their fleets. Police, fire, and emergency medical service groups use Global Position System (GPS) tracking equipment for their day-to-day operations. A wealth of geospatial data is generated each day from these activities and many more. The purpose of my project is two-fold: to develop techniques and a system for effectively collecting this data, and to analyze the data to provide useful output products.

The first system, used by amateur radio operators, is called Automatic Packet Reporting System or APRS. APRS was developed in 1992 by Bob Bruniga as a means of tracking Naval Academy boats during summer cruises. Since then it has evolved to become a complex mesh of protocols for transmitting position, weather, messages, and status information by way of digital packet radio. In most cases APRS uses GPS for determining the user's precise latitude, longitude, and elevation anywhere on earth. The position is digitally encoded into a packet radio signal then transmitted by VHF radio. Other information can be included with each transmission such as course of movement, speed, messages, and text-based status information.

From 1992 until 1996 APRS was a collection of radio-based networks spread out across the country. For people to communicate with each other they had to both be within range of the same radio network. In 1996 Steve Dimse developed a means of combining all of the different networks into a common APRS network on the Internet. In each area with a radio-based network there would be an IGATE computer running special software to receive the messages from the radio and transfer them to Dimse's Internet


Worldwide
APRS Network
Position Reporting Stations
APRS
Internet Stream
APRS Internet
Gateways (IGATEs)
Data Collection

server. Once the data was combined at the Internet server, anybody could connect to it and receive an aggregate stream of all of the information from all of the networks. When Dimse's system first came online he was aggregating data from San Francisco, Los Angeles, Nashville, Atlanta, Miami, and New Jersey. The data represented, at most, a few hundred users, accounting for a few thousand packets each day. Since 1996 APRS has grown in leaps and bounds. Amateur radio operators all around the world are building APRS networks. Today the APRS Internet stream aggregates over 600,000 packets per day from thousands of networks and users around the world.

Procedure

To facilitate analysis of the APRS data I designed several systems for storing and retrieving the data delivered by the APRS Internet stream. From the beginning of my research it was obvious to me that I would have to write a lot of custom software. I selected the Linux operating system for my research. Linux is a free, open source code, Unix-like operating system that works extremely well on a variety of hardware platforms. I avoided using proprietary software because I did not want to be locked into a particular software vendor that could not meet all of my needs. By using open source software I was free to learn from, and build upon, thousands of different programs. I chose to write most of my programs in the programming languages C and PHP and store data using MySQL relational database software.

APRS Database Design

I designed five C structures for use in the parser: a raw packet structure, a message structure, a position structure, a status structure, and a weather structure. The structures all closely resemble the table layout in the MySQL database. When the parsers are passed the raw packet information they are also passed a structure to fill with the extracted data. The individual parsing routines are responsible for converting the many different APRS formats into the common database structure. In many cases the parse routines call other functions help process the data.

After the packet is parsed and the appropriate structure is filled, the structure is inserted in the database. I designed the database to have each packet be keyed on a system-generated unique ID. Every packet is inserted into the raw table that contains the unparsed packet along with date and time, source, destination, and routing information. If the packet was successfully parsed into one of the five structures, then the unique packet ID and parsed values are inserted into the appropriate database table.

First Attempts

My first attempt at collect the whole APRS Internet stream was the spring of 2001. I wrote a program in C called net2db. Net2db would connect an APRS Internet server and store in the MySQL database each packet along with a timestamp of when my program received it. With net2db I was collecting over 3 packets per second and handing each one to the database. After about an hour the network connection would time out and the program would hang, so I had the system automatically restart net2db every 10 minutes. Although this system worked for collecting raw packets it had a number of problems. Besides text-indexing the whole packet field, there was no way to index the database, making it very computationally expensive to retrieve specific packets. Coupled with an inefficient database design and a five-year-old machine it was nearly impossible to retrieve data using this technique.

APRS Parser

To replace net2db, I wrote a full-fledged APRS parser that would not only store the raw APRS packets in a retrievable form, but it would also decompose packets into generic data structures. A parser is a program of function that breaks data down into specific elements. I named the parser message, because that was the first parsing function I wrote. APRS is relatively complicated protocol because it was never really designed. Instead the protocol was written around existing implementations.

I designed the parser to be tolerant in the way which it parses packets. I wanted it to make an attempt to parse the packet, but not to crash if it could not. This decision was not made lightly, but I eventually concluded that with a large dataset it would not be statistically significant to have erroneous information occasionally stored in the database.

After initializing everything and establishing connections to the Internet server and MySQL database the main function simply passes each received packet to the routine that begins the parsing. Every packet is run through a parser that separates the source, destination, routing information and the body of the message. The process function determines what parsing routine to hand the packet to based on the first information character in the body of the packet.

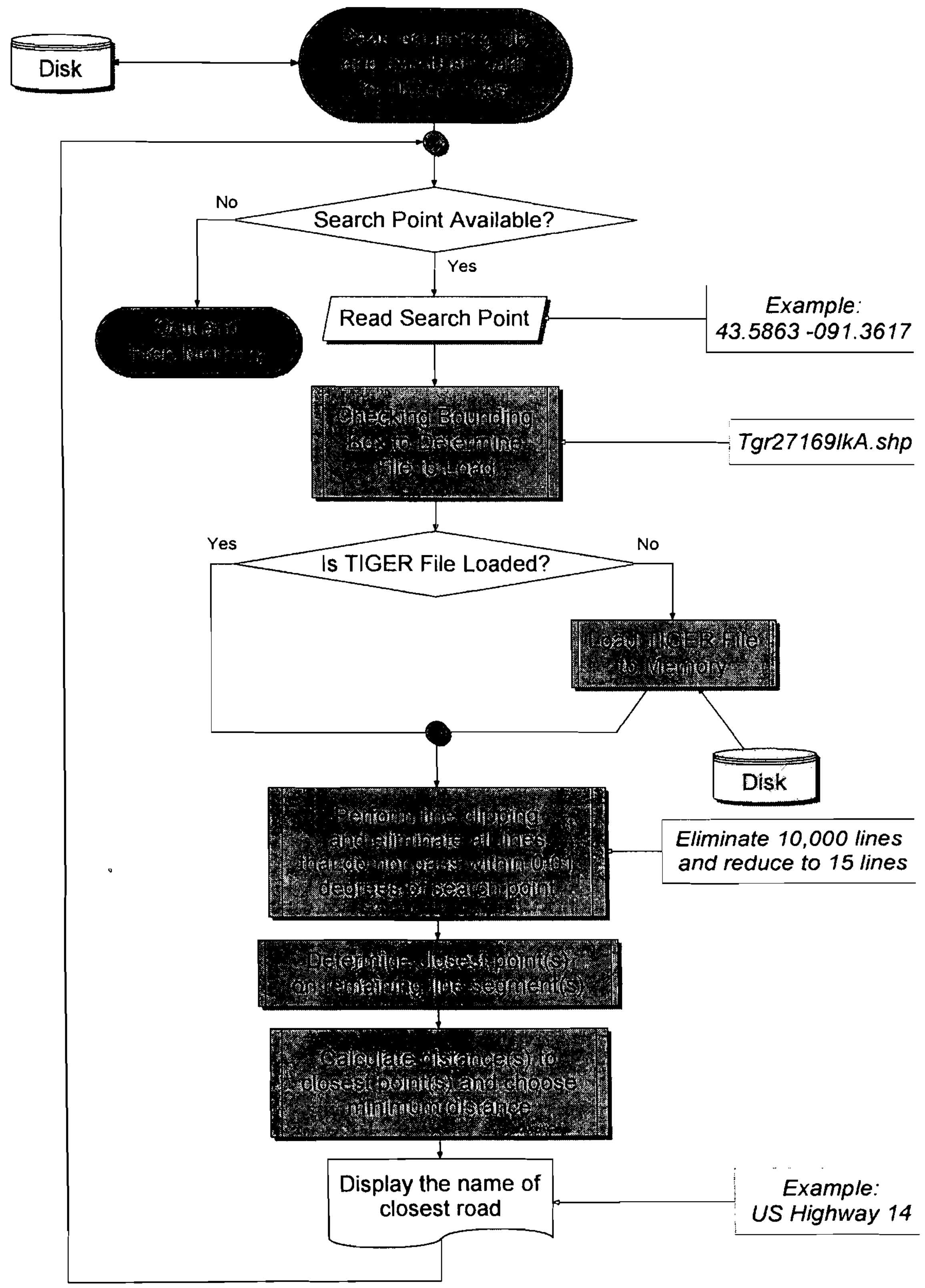
Locating the Nearest Road

Effectively modeling traffic flow has been a long-standing goal of the computer science community. I think that it may be possible, with enough diverse data, to actually monitor traffic for anomalies. The first step in recognizing traffic problems is to determine what roads vehicles are on. I wrote a program called shpdump-arc, which takes a latitude / longitude pair as input and provides the distance to, and name of, the closest road. I used US Census Bureau TIGER maps as my roadway base maps. The TIGER maps define roads as a series of line segments. The TIGER maps come in 7 gigabytes of map files.

Geometrically, the closest distance between a point and a line will be an imaginary line running perpendicular to the original line and intersecting the search point. By adapting this general principle to work with line segments, it is possible to calculate the distance from a search point to every other line segment on the map. Since the line segments represent roads, the line segment associated with the smallest calculated distance will correspond with the nearest road. If the point is not within a few meters of a road then the point cannot be considered as being on the road. Therefore only roads near the search point need be considered. I used a line clipping algorithm to decrease the lines segments searched. Line clipping is the process of determining what line segments fall within a search area and then searching only those segments. Determining what road is closest to the search point is simply finding the line-segment with the smallest distance to the search point.

Determining what road a vehicle is a major step in identifying congestion and other traffic problems. Neural networks could be used determine whether or not the current vector of the vehicle is consistent with the characteristics of previously examined data about the road. In the simplest case, a vehicle stopped in the middle of freeway would be a good indicator of an abnormal traffic situation.

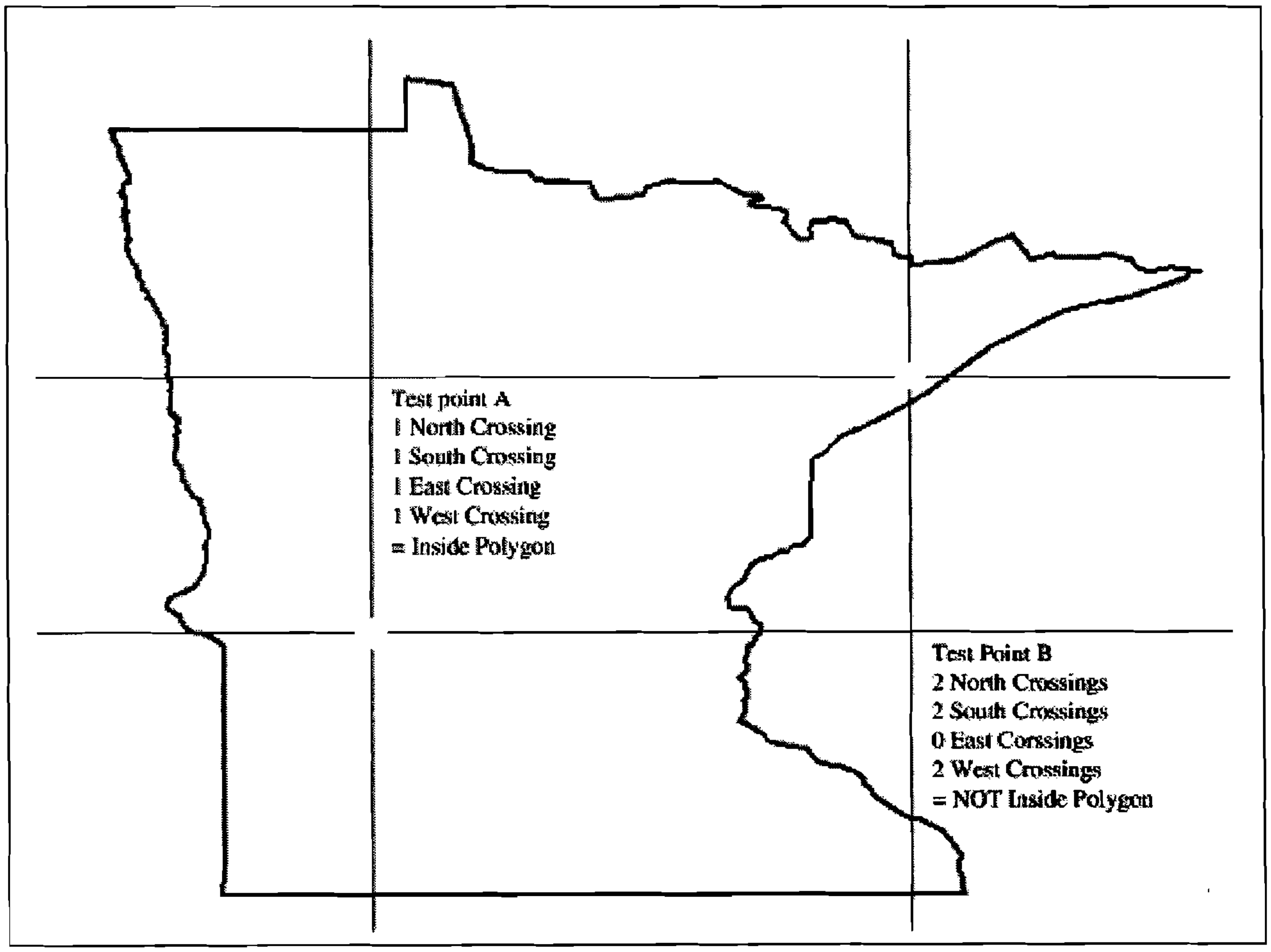
Nearest Road Search


Disk
No
Search Point Available?
Yes
Read Search Point
Example:
43.5863 -091.3617
Checking Bounding Box to Determine File to Load
Tgr27169lkA.shp
Yes
Is TIGER File Loaded?
No
Load TIGER File to Memory
Disk
Perform line clipping and eliminate all lines that do not pass within 0.01 degrees of search point
Eliminate 10,000 lines and reduce to 15 lines
Determine closest point(s) on remaining line segment(s)
Calculate distance(s) to closest point(s) and choose minimum distance
Display the name of closest road
Example:
US Highway 14

Polygon Searching

I wrote a program called shpdump-poly that can very rapidly determine what polygon a point lies within. This test is performed by drawing an imaginary horizontal and vertical line through the test point. By counting the number of times the line crosses the polygon it is possible to determine if the point falls within or outside the polygon. If, in each of the four cardinal directions, the line crosses the polygon an odd number of times then the point is within the polygon. In order to optimize processing time I load each vertex of the polygon and the associated bounding box into memory. Because bounding box checking is more efficient then point in polygon testing, each test point is first checked to see if it falls within the bounding box. If the point falls within the bounding box it is passed to the slower point in polygon test to determine if it is really in the polygon.

While working on shpdump-poly I hypothesized that performance increases could be accomplished by pre-sorting the polygons so the shapes with the highest likelihood of containing the polygon would be searched first. My hypothesis appeared to be correct because when I implemented pre-sorting based on population I was able to decreased execution time by 7%. Subsequent analysis shows that the correlation between the population of each state in the United States and the number of packets originating from that state is r=0.837243. On a Pentium III class machine, the program can place 500,000 latitude / longitude points into a their corresponding state of the United States in 15 seconds.



Map of Observed Average Speeds for Los Angeles

The map to the left shows the average of speed vehicles traveling in Los Angeles County. Road segments that are colored a shade of red represent sections where data has been collected. The brighter the shade of red the faster the observed speed. This map is based on data I collected from approximately 200 mobile APRS stations over a three month period.

1) My database was populated with three months of data from APRS stations worldwide. Data was collected using the APRS parser program. Three months of data represent approximately 54,000,000 packets stored in the database.

2) Packets from moving APRS stations in the vicinity of Los Angeles County were extracted from the database using this SQL query:

> *SELECT latitude, longitude, packet_id FROM position WHERE speed>=10 AND (latitude >= 32.806 AND latitude <= 34.823) AND (longitude >= -118.945 AND longitude <= -117.649) ORDER BY latitude;*

The WHERE clause of the query limits the data to stations moving more than nine kilometers/hour and that fall within the bounding box around Los Angeles. In this case there were approximately 135,000 data points that met these criteria. Each data point consists of the latitude, longitude, and database packet identifier

3) Because the bounding box around Los Angeles allows some packets from stations outside of Los Angeles, the data is sorted by county using the polygon searching algorithm. Using shpdump-poly the data is sorted to one file per county. This further sorting reduced the data points to approximately 106,000.

4) Once the data is separated, each latitude / longitude point is run through my nearest road searching algorithm, shpdump-arc, to find the closest road. For each data point where shpdump-arc finds a corresponding road, a single line is printed. That line is comprised of the original search latitude, longitude, packet identifier, and a unique identifier for the road segment.

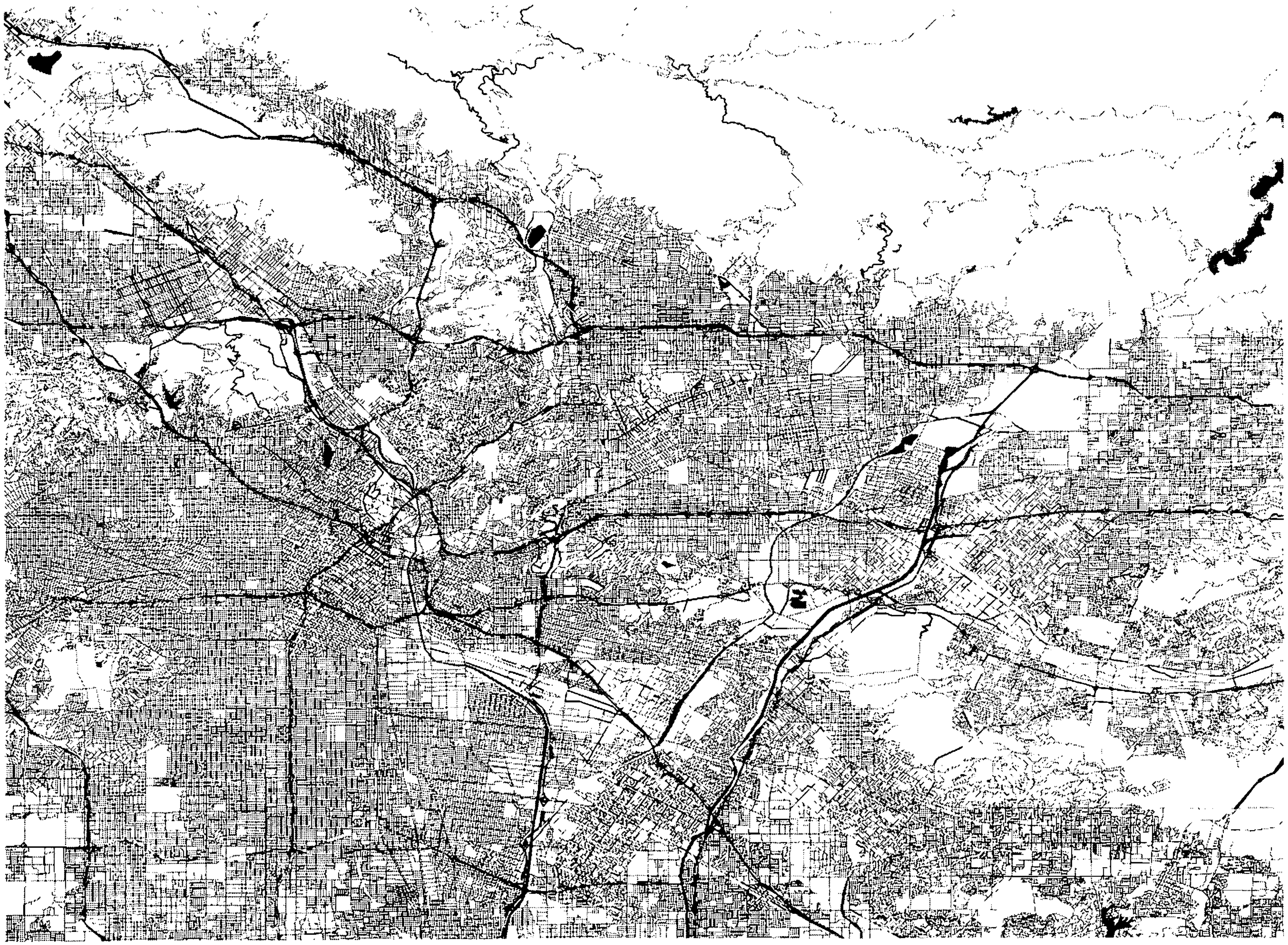
5) The speed at each data point is retrieved from the database based upon the unique packet identifier. This query would retrieve a single speed:

> *SELECT speed, source FROM position WHERE packet_id=35851920;*

Once the speed is determined a SQL INSERT is performed that puts the new record into the database:

> *INSERT INTO speeds (tlid, speed, source, packet_id) VALUES (141792250, 94, "N6NKU", 35851920);*

6) The important database tables are now the speed table containing the observed data points, and the roads table containing the name and unique identifier of every road segment on the map. To find all of the road segments where data has been observed, I use an equi-join between the two tables, based on the road segment. This SQL query finds all of the roads in Los Angeles County and it returns the minimum, maximum, average, and standard deviation of the observed speeds:



See a full-size version of this map at: **http://db.aprsworld.net/mapserver/la.php**

```
SELECT speeds.tlid, AVG(speeds.speed) AS "speedAvg", MIN(speeds.speed) AS
"speedMin", MAX(speeds.speed) AS "speedMax", STD(speeds.speed) AS "speedStddev",
COUNT(speeds.speed) AS "num" FROM speeds, roads WHERE roads.fips=06037 AND
speeds.tlid = roads.tlid GROUP BY speeds.tlid;
```

7) For display of the data I use a standard GIS file format called a shapefile. A shapefile consists of three individual files. Two specify the geometry of the shapes and the third file is a database of information associated with each shape. The map of Los Angeles County is a shapefile, however I had to add fields to the shapefile's database to store the speed information for each road segment. I wrote a program, join2dbf, which takes the calculated data from the equi-join above and appends it onto the appropriate record in the map.

After running join2dbf I now have a complete map of Los Angeles that contains the average speed, minimum speed, maximum speed, and standard deviation from each road segment with observed data points.

The map to left will never change, however the programs and techniques that I used to create it could be run continuously to create a dynamic map of speeds. Such a map could be used to identify areas of traffic congestion or used by drivers to avoid slow moving roads.

Results and Conclusions

1) APRS and Automatic Vehicle Location technologies provide data that can be harnessed for many other applications besides simple vehicle tracking. The amount of data generated by these technologies is enormous.

I was able to create tools and techniques for effectively storing and retrieving position information for thousands of vehicles generating millions of pieces of information.

- The database and APRS parser has scaled well and demonstrated its robust nature.
- The datamart web interface provides a convenient means for verifying database integrity and visualizing station locations.
- The polygon searching program is useful for separating data by region or within political boundaries. It can be used as a filter before other data analysis.

2) Automatic detection of traffic congestion has the potential to reduce investments by metropolitan areas in camera systems and public service personnel for monitoring traffic problems.

By observing traffic over a long term it is possible to calculate and map the average speed on roadways. Once such a base map is made, it is possible to detect traffic anomalies, such as traffic jams, by comparing the "live" observed data to previously gathered data.

- The parser that I wrote continuously adds data to the database which has the effect of constantly adjusting the minimum, maximum, and average speeds of the road.

3) Because large fleets of vehicles are being equipped with vehicle locating technologies, there are many new opportunities for automatically collecting near real-time vehicle movement data on a large scale.

Even with a low density of mobile APRS stations, I was able to develop software for observing vehicle movement on many roadways. In the case of Los Angeles, only 200 stations over three months provided enough data to calculate average road speeds on many highways and some smaller roads. With hundreds of thousands of mobile stations, it would be possible to observe data on many roads with a large number of observations for each road. More observations would also make it possible to compute traffic maps for different time periods during the day, including rush hour versus normal traffic.

Acknowledgements

I would like to thank Dr. Mark Ordal of the Minnesota Academy of Math and Science for his generosity of providing me lab space with a suitable high-speed Internet connection, as well as for the evenings that he stayed late to allow me to keep working. Adam Frisch KB0VYO's Beowulf cluster was responsible for putting the idea in my mind of processing millions of pieces of data. Adam has also spent days working with me to setup computers and keep equipment running. This project would not be possible without the dedicated individuals that have designed and built the APRS networks. I would like to extend a big thank you to the men and women of the open source and free software communities because "If I have seen further it is by standing on ye shoulders of Giants." – Isaac Newton

References

Dimse, Steve. (1997). "APRServe: An Internet Backbone for APRS." Available online: <http://www.aprs.net/aprserve.dcc.html>

Wade, Ian. (2000). " APRS Protocol Specification Version 1.0.1." Available online: <ftp://ftp.tapr.org/aprssig/aprsspec/spec/aprs101/APRS101.pdf>