

Cycle Sense: correlating outside data sources

Senglong Taing, Isaac Kim, Min Mun, Eric Howard

Group/Lab Name - URL

Introduction: Gathering and applying outside data

Gathering Data

- Find existing Websites that have desirable data and gather data from the Web.
- Store data in sensorbase. This allows all the data to be seen in one place rather than having to browse the web and several websites.

Applying Data

- Mapmatching – given a point or a bike route we can find what road or path was taken by the bike.
- Supplying bike routes with information such as biking community events, so that the users may want to decide what route to take based on events information

Problem Description: Gathering and applying data effectively

Gathering Data

- Gathering and storing data about elevation, traffic incidents, and Cycling events manually wastes time and energy.

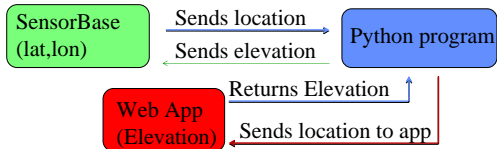
Applying Data

- Mapmatching – Trying to match a coordinate to the closest road is often incorrect due to GPS error or other factors. Matching using a *nearest road* algorithm often produces long incorrect paths.
- Events data lack longitude-latitude points that are necessary for the integration of events into the mapping routes.

Proposed Solution: Web source, Cross-sections, Spatial Data, Events

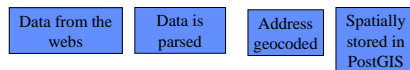
Gathering Data from Web Apps

- Use python to gather data from Web
 - Using HTTP Post and python, we get longitude and latitude coordinates from a web database (sensorbase).
 - Using python code and HTTP Get we can get data such as, elevation, traffic info, and cycling events from the web.
- Store Data
 - Send the additional data back to sensorbase with an HTTP Post.



Spatial Data Management

- Events' addresses are geocoded for their longitude-latitude points via google's geocode API.
- Longitude-latitude geometric points are spatially contained in zip code areas through which biking paths intersect.

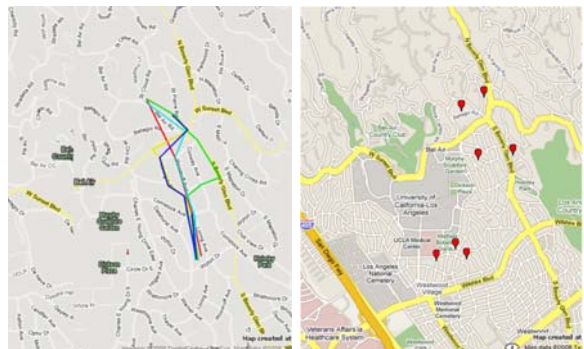


Events Display

- Generated biking traces are accompanied with events located in the zip code areas they intersect.

Matching Cross-Sections

- Instead of simply matching to the closest road, using the same closest road algorithm, match cross-sections within a certain range of the GPS point. Connect *cross-sections* and ignore anything else in-between.
- In the bottom picture the nearest road algorithm implies that the driver went all the way around to Lemarsh, made a U-turn, and went back to Topanga Canyon. This is obviously wrong. By matching cross-sections however, the correct path is easily found.



- Comparison of the results when both algorithms are run on the picture on the right. (errors underlined)
 - Nearest Road Algorithm returns: ['Sawtelle', 'Sardis', 'Sawtelle', 'National', 'Sawtelle', 'National', 'National', 'Sepulveda', 'Sepulveda', 'Sepulveda', 'Sepulveda']
 - Cross-Section Algorithm returns: ['Sawtelle', 'Sawtelle', 'National', 'Sepulveda', 'Sepulveda']
- The Cross-Section algorithm works perfectly while the Nearest Road algorithm mixes up roads and references roads that are not part of the path.

- The user is presented with a number of potential routes between two locations inputted by the user.
- Along those traces, seven events are found in the zip code areas the traces intersect.
- Our algorithm's implementation provides the user with the number of nearby events and suggest the route closest to them.