mqtime: A Stata Tool for Calculating Travel Time and Distance Using Mapquest Web Services

John Voorheis*

University of Oregon

September 8, 2014

Abstract

This note describes `mqtime`, a new Stata library which provides functionality to perform a variety of mapping tasks, including calculating travel time, distance (driving, biking or on foot) and estimated fuel use. `mqtime` uses an overlooked free and open-source mapping service provided by Mapquest. This service has significantly more attractive terms of use than widely used alternatives (e.g. Google Maps), which limit use to a few thousand queries per day. Hence `mqtime` makes analysis with even very large datasets practical. We also provide a convenient function for geocoding character addresses to geographic coordinates (or reverse geocoding geographic coordinates to character addresses.)

1   Introduction

In a variety of applications, the distance between two locations can be a crucial factor in explaining behavior. There are a number of ways to calculate distances, and these methods have varying degrees of realism. One can, for instance, calculate a straight-line, or "great circle" distance between two points based on their latitude and longitude coordinates. If one is modeling transportation, however, the straight-line distance may be quite different from the distance actually traveled e.g. by car or bicycle. Calculating the true driving distance is a much more complex task than calculating straight-line distance, but when accurate measures of distances are important to a particular analysis, this extra complexity may be warranted.

One way to calculate distances is to make use of third-party mapping services. Indeed, the `traveltime` library (written by Adam Ozimek and Daniel Miles) was written to use Google's Maps service. Unfortunately, `traveltime` was written for the now-obsolete v2 of the Google Maps API. Since the Spring 2013 transition to the new Google Maps API v3, `traveltime` has been rendered non-functional. Additionally, before the API change, Google implemented restrictions which severely limited the number of requests that a researcher could make each day. An extension of the `traveltime` library, `traveltime3`, has been written which supports the Google Maps v3 API, but it is still subject to the same rate limits.

`mqtime` is an attempt to provide off-the-shelf travel time calculation functionality without the disadvantages associated with tools that utilize the Google Maps API. This utility takes advantage of a valuable but heretofore overlooked service provided by MapQuest. MapQuest does provide an API that accesses its commercial mapping service (the same service one would access through www.mapquest.com). However, Mapquest also provides a second API which accesses the OpenStreetmaps service. The OpenStreetmaps (OSM) project is a partially crowd-sourced project to produce and maintain a publicly available, open-source street map covering as much of the world as possible. More information about the OSM project can be found at http://www.openstreetmap.org.

`mqtime` (and the associated helper function `mqgeocode`) is written to mimic the syntax of `traveltime`, thereby easing the learning curve for users who are already familiar with the `traveltime` tool. However, the under-the-hood functionality of making the API requests and then parsing them into a format that is Stata-readable differs significantly. The `insheetjson` library provides functionality to parse the type of data object (a JSON object) returned by the Mapquest service.[1] The approach in `mqtime` can be easily extended to make use of other similar API services, which are increasingly provided by companies, and are an underutilized resource for economists.

The chief advantage that `mqtime` offers over the functionality of `traveltime` (and the `traveltime3` successor) is freedom from the strict usage limits imposed by Google. Google

---

[1]The Mapquest APIs are written with web-oriented languages like Javascript and Python in mind. It is trivially easy for an experienced programmer to write a Python script which accomplishes the same thing as `mqtime`, but the goal in this case is to make the information as accessible as possible for general Stata users.

places a firm limit of 2,500 API requests (e.g. directions or geocoding) per day per IP address. These limits can be very restrictive, even for medium-sized datasets. A dataset used by one of the testers of `mqtime` involves approximately 500,000 unique address pairs. Processing these data with the Google Maps API would take over six months. The terms of use for the Mapquest APIs are much more permissive. The commercial MapQuest API allows up to 5,000 API requests per day, while the OSM API has no preset limit. By utilizing the Mapquest APIs, `mqtime` allows for the processing of large datasets in reasonable amounts of time.[2]

2 The `mqgeocode` Command

Although the Openstreetmaps API will take as input either text address information or latitude/longitude coordinates, users may wish to generate latitudes/longitudes from a text address for other purposes (e.g. for calculating simpler straight-line distances or for use within GIS software.) Geocoding of addresses can be accomplished with GIS rather easily, but doing so requires leaving the comfortable confines of Stata. `mqgeocode` is provided as a convenience utility. Like the `mqtime` command (see next section), the `mqgeocode` command will first attempt to geocode using the Openstreetmaps API. If this geocoding fails, `mqgeocode` will then query the commercial Mapquest API. By default, `mqgeocode` will display a running total of the number of addresses that have been geocoded, as well as the number of requests made to the commercial Mapquest API. Users with data that

2.1 Syntax

```
mqgeocode [in], [address(string) | lat(string) long(string)]
  [outaddress(string) api_key(string)]
```

2.2 Options

address(*string*) specifies the variable holding the plain text addresses (e.g. "123 First Ave., Eugene, OR 97402") to be geocoded. Cannot be combined with options **lat** and **long**

lat(*string*) and **long**(*string*) specify the variables holding the latitudes and longitudes to be reverse geocoded. Cannot be combined with option **address**

outaddress(*string*) specifies the variable name to be used for the output, which will be either a plain text string ("Ann Arbor, MI") or a string holding a latitude, longitude pair ("32.377588,-86.301882")

---

[2]In testing, we timed the average request at about 0.5 seconds, most of which is HTTP overhead, so our tester's dataset would require several days to complete.

`api_key`(*string*) specifies a user's API key. By default the `mqtime` command will use the built in API key associated with the author. However, if users wish to use their own, they may do so. See comments following for more discussion.

2.3 Remarks

Traditional geocoding is relatively straightforward – it requires only that the user has created a variable holding the full text addresses. If a user has separate variables for e.g., City and State, concatenating them is trivial:

$$\texttt{gen } \textit{newvar} \texttt{ = } \textit{cityvar} \texttt{ + "," + } \textit{statevar}$$

In general, `mqgeocode` will do only basic string formatting (most importantly replacing spaces with %20), so the user will need to ensure that there are no disallowed characters or spelling errors. The API is permissive in terms of which address formats it will accept, including `city, state`; `address, city, state, zipcode`; and `address, zipcode`. For certain addresses located in unincorporated areas, the latter seems to perform better.

When reverse geocoding, `mqgeocode` will return a text address at the most granular level available (this will range, in practice, from the zip code level to the exact address.) In either the traditional or reverse geocoding case, the geocoded variable is returned as a single string. If separate latitude and longitude variables are required in the traditional geocoding case, the `outaddress` variable can be split using

$$\texttt{split } \textit{latlongvar}, \texttt{ p(",")}$$

2.4 Example

Suppose we think that the distance between state capitol buildings is important for some application. Then if we have a dataset of capitol building addresses that looks like:

Our first step is to create new variable:

```
gen capitol = address + "," + city + "," + state
```

And then we can geocode these addresses using:

```
. mqgeocode in 1/10, address(capitol) outaddress(coords)
Observation 1 of 10 geocoded.
Observation 2 of 10 geocoded.
...
```

When execution is complete, the data will contain latitude and longitude for each observation:

```
                    address          city         state
      |-------------------------------------------------|
  1.  |      600 Dexter Ave.    Montgomery      Alabama |
  2.  |          120 4th St.        Juneau       Alaska |
  3.  |     1700 W. Washington      Phoenix      Arizona |
  4.  |     300 W. Markham St.   Little Rock     Arkansas |
  5.  |         1315 10th St      Sacramento   California |
      |-------------------------------------------------|
  6.  |      200 E. Colfax Ave.      Denver      Colorado |
  7.  |       2210 Capitol Ave.    Hartford   Connecticut |
  8.  |     411 Legislative Ave.       Dover     Delaware |
  9.  |         402 S. Monroe St.  Tallahassee    Florida |
 10.  |  206 Washington St. SW       Atlanta      Georgia |
      +-------------------------------------------------+
```

```
                         capitol                        coords
      |----------------------------------------------------------------|
  1.  |        600 Dexter Ave.,Montgomery,Alabama     32.377588,-86.301882 |
  2.  |            120 4th St.,Juneau,Alaska     58.301945,-134.410453 |
  3.  |          1700 Washington,Phoenix,Arizona     33.448409,-112.045406 |
  4.  | 300 W. Markham St.,Little Rock,Arkansas     34.748601,-92.273452 |
  5.  |        1315 10th St,Sacramento,California     38.576718,-121.494911 |
      |----------------------------------------------------------------|
  6.  |        200 E. Colfax Ave.,Denver,Colorado     39.739994,-104.986134 |
  7.  |    2210 Capitol Ave.,Hartford,Connecticut     41.762511,-72.681198 |
  8.  |        411 Legislative Ave.,Dover,Delaware     39.156598,-75.520208 |
  9.  |      402 S. Monroe St.,Tallahassee,Florida     30.437994,-84.280724 |
 10.  |      206 Washington St. SW,Atlanta,Georgia     33.749747,-84.38857 |
      +----------------------------------------------------------------+
```

3 The `mqtime` Command

3.1 Syntax

mqtime [in], [start_x(*string*) start_y(*string*) end_x(*string*) end_y(*string*)
 start_add(*string*) end_add(*string*) api_key(*string*) km mode(*string*)]

3.2 Options

start_x(*string*), start_y(*string*) specify the variables holding the x, y (i.e. longitude,
latitude) coordinates of the *origin* location. Cannot be used with option start_add.

end_x(*string*), end_y(*string*) specify the variables holding the x, y (i.e. longitude, lati-
tude) coordinates of the *destination* location. Cannot be used with option end_add.

`start_add(`*string*`)` specifies the variable holding the plain text address of the *origin* location. Cannot be used with options `start_x, start_y`.

`end_add(`*string*`)` specifies the variable holding the plain text address of the *destination* location. Cannot be used with options `end_x, end_y`.

`api_key(`*string*`)` specifies a user's API key. By default the `mqtime` command will use the built in API key associated with the author. However, if users wish to use their own, they may do so. See comments following for more discussion.

`km` specifies whether the distances returned will be in kilometers or miles (default is miles.)

`mode(`*string*`)` specifies which mode of travel is to be used. Must be one of "walking", "bicycle" or "transit" (the default is driving, if unspecified.)

3.3 Comments

By default `mqtime` will first attempt to query the Mapquest OSM API for each origin/destination pair. As noted earlier, the OSM service has no pre-set rate limit, and is therefore much more amenable to the processing of large datasets. However, it does have slightly less coverage than the commercial mapping alternatives.[3] The commercial Mapquest API has considerably better coverage, although it does have stricter usage limits. By default, `mqtime` will query the commercial Mapquest API only if it receives a route failure error from the OSM API.

By default, `mqtime` will also use the API key associated with the author of this note (Voorheis). This means, however that all requests to the commercial API will be pooled for all users. If users find themselves needing to make a relatively large number of requests for which the OSM service fails, it may be advantageous to request a separate API key. Noncommercial API keys are free, and can be requested at the Mapquest developer site.[4] To make sure `mqtime` employs a user's own API key, the `api_key` option must be specified. The actual API key is a long, randomly generated string, so it may be easiest to attach it to a local macro, for instance:

<div align="center">

`local api_key = "`*your_key_here*`"`

</div>

Users can then instruct `mqtime` to use this API key:

<div align="center">

`mqtime, ... api_key("'api_key'")`

</div>

With this modification, `mqtime` will make requests employing the user's own personal API key.

---

[3]In testing, I found that the OSM failed to generate a route for approximately 2% of origin/destination pairs.

[4]http://developer.mapquest.com/web/info/account/app-keys

The `mqtime` program takes the user's inputs, and builds a URL that is the desired API request. This URL will return a JSON (Javascript Object Notation) data object.[5] This JSON object is a text file containing nested key:value pairs. Many general purpose programming languages have built-in functionality to parse these objects, but Stata has no base functionality to deal with them. Fortunately, Erik Lindsley's `insheetjson` ado can parse JSON files into Stata-readable data. `mqtime` makes a call to the `insheetjson` ado to parse the JSON object and return the relevant information (travel time, driving distance and fuel use), and discards the rest.

Users can specify the origin and destination locations as either latitude/longitude or as a text address. The two locations need not be in the same format. For example, if one had latitude and longitude for a list of origins, and text addresses for a list of destinations, one could execute

<div align="center">

`mqtime, start_x(`*lngvar*`) start_y(`*latvar*`) end_add(`*addvar*`)`

</div>

The other options are largely self-explanatory, with the exceptions of the `api_key` option. The non-driving travel modes available in `mode` may be slightly less accurate than the driving information for the OSM service. For multimodal (transit) mode, the user needs to specify a time of day (if no time is provided, the API request will be for the time at runtime.)

`mqtime` will by default create four new variables `travel_time, distance, fuelUsed, service`. Travel time is returned in minutes by default, and distance is returned in miles (or kilometers if `km` is specified). The estimated fuel use variable is generated based on Mapquest's estimate of fuel use in gallons for the trip.[6] The OSM API returns the fastest route based on posted speed limits, but does not take into account realtime traffic information. The commercial Mapquest API can give traffic-adjusted travel time, but this is not yet built into `mqtime`.

3.4 Examples

First, we can return to the state capitol building example to illustrate how `mqtime` can query the API to obtain driving directions from text addresses. Suppose we want to generate directions from the 10 state capitol buildings used in the previous example to 10 other arbitrarily matched capitol buildings. Our data then look like:

To generate the driving time, etc, we can then execute

```
mqtime, start_add(capitol_origin) end_add(capitol_destination)
Processed 1 of 10
Processed 2 of 10
```

---

[5]An example URL (requesting the driving route between Eugene, OR and Springfield, OR) is http://open.mapquestapi.com/directions/v2/route?key=YOUR_KEY_HERE& from=Eugene,OR&to=Springfield,OR&outFormat='json'&narrative='none'

[6]As of now, the fuel use estimate is returned as the Mapquest default (based on). It is possible to supply the Mapquest API with specific fuel efficiency assumptions, but this is not yet implemented in `mqtime`.

```
+-----------------------------------------------------------------------------+
|                    capitol_origin                        capitol_destination |
|-----------------------------------------------------------------------------|
 1. |       600 Dexter Ave.,Montgomery,Alabama       488 N 3rd St,Harrisburg,Pennsylvania |
 2. |              120 4th St.,Juneau,Alaska          82 Smith St.,Providence,Rhode Island |
 3. |            1700 Washington,Phoenix,Arizona    1100 Gervais St.,Columbia,South Carolina |
 4. | 300 W. Markham St.,Little Rock,Arkansas       500 E. Capitol Ave,Pierre,South Dakota |
 5. |        1315 10th St,Sacramento,California       600 Charlotte Ave,Nashville,Tennessee |
    |-----------------------------------------------------------------------------|
 6. |        200 E. Colfax Ave.,Denver,Colorado       1100 N. Congress Ave,Austin,Texas |
 7. |    2210 Capitol Ave.,Hartford,Connecticut      350 N. State St,Salt Lake City,Utah |
 8. |       411 Legislative Ave.,Dover,Delaware         115 State St,Montpelier,Vermont |
 9. |      402 S. Monroe St.,Tallahassee,Florida          1000 Bank St.,Richmond,Virginia |
10. |    206 Washington St. SW,Atlanta,Georgia     416 Sid Snyder Ave. SW,Olympia,Washington |
    +-----------------------------------------------------------------------------+
```

```
...
Processed 10 of 10
```

When execution is complete we can check that `mqtime` has generated the correct data; we can see that all of the routes were mapped without error, and none required the backup commercial Mapquest API.

```
+-------------------------------------------+
|travel_time  distance   fuelUsed   service |
|-------------------------------------------|
 1. |   816.05    882.256     42.06       OSM |
 2. | 4559.617   4208.583    187.13       OSM |
 3. |  1917.85   2057.966     96.07       OSM |
 4. |   929.15    1004.32     47.94       OSM |
 5. | 1996.067   2279.258    114.96       OSM |
    |-------------------------------------------|
 6. | 939.3834   1061.885     53.38       OSM |
 7. |   2041.8    2275.139    111.56       OSM |
 8. | 477.6333   476.2961     21.58       OSM |
 9. |      713    760.513     35.63       OSM |
10. | 2458.733   2712.527    130.95       OSM |
    +-------------------------------------------+
```

4   Conclusion

Taking advantage of `insheetjson`'s ability to parse JSON files (the industry standard of data provision from an API) we have shown how to generate travel times, distances and estimated fuel use. This is accomplished by using a convenient but under-utilized service provided by Mapquest. Unlike previous implementations, `mqtime` can process even very large datasets without running afoul of commercial entities' terms of use. `mqtime` can also be easily patched to stay up to date with API changes.

There are several features available in either the commercial Mapquest or the OSM API of which `mqtime` does not yet take full advantage. Future revisions of the `mqtime` codebase, conditional on user input, will seek to incorporate these features. Chief amongst these potential additional features are: 1) providing vehicle MPG to calculate more precise fuel use, 2) taking advantage of real-time traffic data to provide more precise travel times, and 3) the ability to more precisely request directions (e.g. avoiding toll roads).

References

Mapquest platform terms of use. URL `http://developer.mapquest.com/web/info/terms-of-use`.

Openstreetmaps legal faq. URL `http://wiki.openstreetmap.org/wiki/Legal_FAQ#I_would_like_to_use_OpenStreetMap_maps._How_should_I_credit_you.3F`.

Stefan Bernhard. Traveltime3: Stata command to retrieve travel time and road distance between two locations. Statistical Software Components, Boston College Department of Economics, July 2013. URL `http://ideas.repec.org/c/boc/bocode/s457670.html`.

Adam Ozimek and Daniel Miles. Stata utilities for geocoding and generating travel time and travel distance information. *The Stata Journal*, 11(1):106–119, 2011.

About The Author

John Voorheis is a PhD student in Economics at the University of Oregon. His research focuses on measuring income inequality and poverty at sub-national geographic scales, and examining the effects of local income distribution on economic outcomes of interest. He is also interested in learning and developing new tools for economists to take advantage of the "Big Data" revolution.