MQtime: A Tool For Calculating Travel Time and Distance in Stata

John Voorheis*

University of Oregon

November 6, 2013

Abstract

This note describes a new Stata library which provides functionality to calculate driving distance and travel time using an overlooked free (as in speech and beer) and open-source mapping API provided by Mapquest, which has much more attractive terms of use for researchers than comparable alternatives. We also provide a convenience function for geocoding or reverse geocoding string geographic data.

1 Introduction

In a variety of applications, the distance between two locations can be a crucial bit of data. There are a number of ways of calculating said distances, which have varying degrees of realism. One can, for instance, calculate a straight line distance between two points given latitude and longitude coordinates. If one is modeling transportation, however, the straight line distance may be quite different from the actual distance traveled e.g. by car or bicycle. Calculating the true driving distance, however, is a much more complex task than calculating straight line distance.

One way to complete this task is to make use of third party mapping services. Indeed, the TRAVELTIME library (written by Adam Ozimek and Daniel Miles) was written to do this exactly, using Google's Maps service. Unfortunately, Google recently moved to a new version their API (application programming interface) and this change has obviated TRAVELTIME's approach. In the mean time, there has been no off the shelf solution available for Stata using either the new Google Maps API or another mapping service. Additionally, even before the API change, Google had implemented restrictions which limited the number

*Thanks to Sonja Kolstoe and Jason Query for providing datasets and testing, and to Trudy Cameron for encouragement and advice. None of this would be possible without the work of Adam Ozimek and Daniel Miles (for writing the original TRAVELTIME) and Erik Lindsley (for writing INSHEETJSON). All mistakes are my own.

of requests that a researcher could make to a few thousand per day.

MQtime is an attempt to provide just this sort of off the shelf travel time calculation functionality without the disadvantages associated with tools that utilize the Google Maps API. This is accomplished by taking advantage of an otherwise overlooked service provided by MapQuest. MapQuest provides an API that accesses its commercial mapping service (the same service one would access through www.mapquest.com) which has similar rate limits to those imposed by Google. However, Mapquest also provides a second API which accesses the OpenStreetmaps service, and imposes no preset limits on queries to this API. The OpenStreetmaps (OSM hereafter) project is a partially crowdsourced project to make 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) are written to mimic the syntax of Traveltime, in order to ease the learning curve for users who are familiar with the tool. 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 sort of data object (a JSON object) returned by the Mapquest service[1]. The approach in MQtime is then easily extensible to other similar API services, which are increasingly provided by companies, and are an underutilized resource for economists.

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 crow-flies distances.) This can be accomplished with, e.g. ArcGIS or the equivalent rather easily, but doing so requires leaving the comfortable confines of Stata. MQgeocode is provided as a convenience function. Unlike the MQtime command (see next section), the MQgeocode command makes use of only the OpenStreetmaps API, although it is easily modified to use the commercial API.

2.1 Syntax

```
MQgeocode [in], [address(string) lat(string) long(string) outaddress(string)]
```

2.2 Options

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

[1]It should be noted that the APIs are written with web-oriented languages like Javascript and Python in mind; in fact, it is trivially easy to write a Python script which accomplishes the same thing as MQtime.

`lat`(*string*) and `long`(*string*) specifies the variables holding the latitude 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 ("45.52,-122.71")

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, using, e.g.

`gen` *newvar* `=` *cityvar* `+ "," +` *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

`split` *latlongvar*`, p(",")`

2.4 Example

Consider a list of State capitol building addresses, and assume further we are using any information gleaned for good and not evil. If the information originally 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.
...
```

After which we should have latitude and longitudes 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, latitude) 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 not (default is miles.)

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

3.3 Comments

`MQtime` will by default 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.[2] The commercial Mapquest API has considerably better coverage, although it does have stricter rate limits. Mapquest limits each API key to 5000 directions requests per day.[3] As such, `MQtime` will, by default, query the commercial Mapquest API if it receives a route failure error from the OSM API.

By default, `MQtime` will use the API key associated with the author. 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 on the Mapquest developer site: http://developer.mapquest.com/web/info/account/app-keys. In order to make sure `MQtime` uses a user's own API key, the `api_key` option must be specified. Since the actual API key is a randomly generated string, it may be easiest to attach it to a local macro.

The `MQtime` program takes the user's inputs, and forms a URL that is the API request. This URL will return a JSON (Javascript Object Notation) data object.[4] This JSON object is a text file containing nested key:value pairs. Web-facing languages like Python or Javascript have built-in functionality to parse these objects, but Stata has no base functionality to deal with them. Luckily, Erik Lindsley's INSHEETJSON ado can parse JSON

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

[3] In contrast to the Google Maps API, for instance, which rate limits on the basis of IP address.

[4] 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'

files into Stata readable data. `MQtime` makes a call to the INSHEETJSON program to parse through 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. So, if, for instance, one had latitude and longitude for a list of origins, and text addresses for a list of destinations, one could execute

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

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 either miles or kilometers (if `km` is specified). The estimated fuel use variable is generated based on Mapquest's estimate of fuel use for the trip. 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 to illustrate how `MQtime` can query directions from text addresses. Suppose we want to generate directions from the 10 state capitol buildings used in the previous example to 10 other capitol buildings. Our data looks like then to generate the driving time, etc, we can execute

	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

```
MQtime, start_add(capitol_origin) end_add(capitol_destination)
```

```
Processed 1 of 10
Processed 2 of 10
...
Processed 10 of 10
```

After which we can check that `MQtime` has generated the correct data: So we can see that

```
    +-----------------------------------------+
    |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 |
    +-----------------------------------------+
```

all of the routes were mapped without error, and none required the backup commercial Mapquest API.

4 Conclusion

Taking advantage of user-written functionality 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 an under-appreciated service provided by Mapquest. Unlike previous implementations, `MQtime` can process even very large datasets without running afoul of commercial entities' terms of use, and can be easily patched to stay updated with API changes.

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