

OSAT: Open Source Accessibility Toolset:

Accessibility Analysis Process Overview

**Transportation Systems**

Brian Cheng

David Greczner

Mike McGurrin

August, 12, 2011

© Noblis 2011

# License Information

© Noblis Corporation, 2011. This documentation is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Table of Contents

[License Information 2](#_Toc302398623)

[Introduction 4](#_Toc302398624)

[Overview 4](#_Toc302398625)

[Process Summary 4](#_Toc302398626)

[Overview 10](#_Toc302398627)

[Necessary Files 10](#_Toc302398628)

[Useful Software/Plug-ins 10](#_Toc302398629)

[Obtaining Necessary Files 11](#_Toc302398630)

[Outline of the Process 12](#_Toc302398631)

[Detailed Process 13](#_Toc302398632)

[Preparing for and Running the Analyses 13](#_Toc302398633)

[Build graph 20](#_Toc302398634)

[Visualize graph (VizGui) 20](#_Toc302398635)

[Analyzing and Visualizing the Results 24](#_Toc302398636)

[Folder/File Structure (as of summer 2011) 27](#_Toc302398637)

[Future Work 30](#_Toc302398638)

[Overview of Classes (documented summer 2010): 31](#_Toc302398639)

[Overall and Transit Accessibility 31](#_Toc302398640)

[Automotive Accessibility Calculator 36](#_Toc302398641)

[Common Questions 38](#_Toc302398642)

# Introduction

## Overview

The Open Source Accessibility Toolset is a process, along with a set of software routines, that, in conjunction with open data and other open source software packages, allows the calculation of various transportation accessibility metrics. The toolset has been used to calculate accessibility at both the Traffic Analysis Zone (TAZ) and is being enhanced to handle Block Group level analysis (incomplete at present).

The primary focus of the effort has been on calculating access to jobs via transit, however the same procedures can be used to calculate access to other needs or services, provided the user has the necessary data sets.

Similarly, the toolset includes procedures for calculating accessibility metrics for automobile travel, however the utility is limited by the lack of open data on automotive travel times or roadway speeds during congested periods.

The toolset is built around OpenTripPlanner, an open source multi-modal trip planner which runs on both Linux and Windows operating systems. ([github.com/openplans/OpenTripPlanner/wiki/](file:///C:\Users\m14661\AppData\Local\Microsoft\Windows\Temporary%20Internet%20Files\Content.Outlook\FQEE0XPQ\github.com\openplans\OpenTripPlanner\wiki\)) . It also uses OpenStreetMap ([www.openstreetmap.org/](http://www.openstreetmap.org/)). These are independent projects and Noblis has no connection to either of these projects other than as a user. An additional key input is the set of transit schedule files, in GTFS format, that OpenTripPlanner uses.

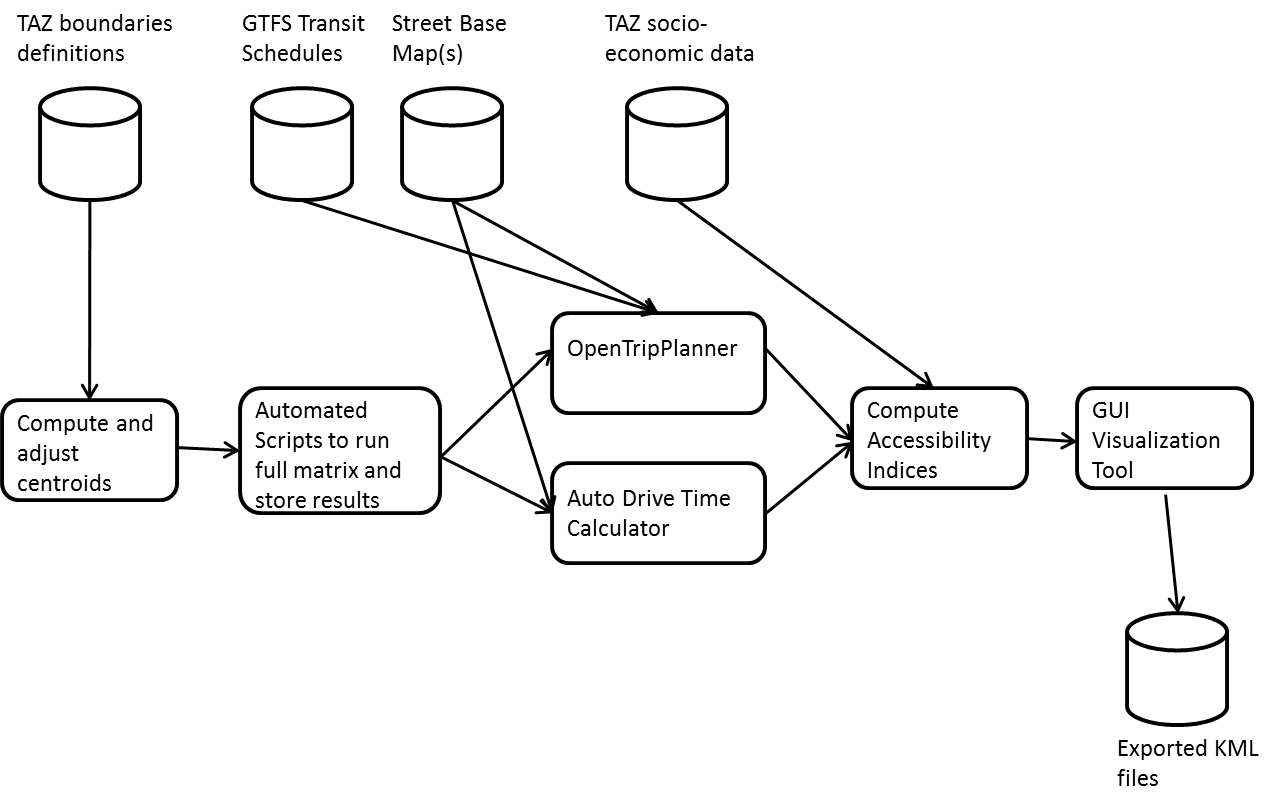
# Process Summary

This section provides a very brief overview of the process. It can be referred back to as a guide when reading the detailed sections of the documentation.

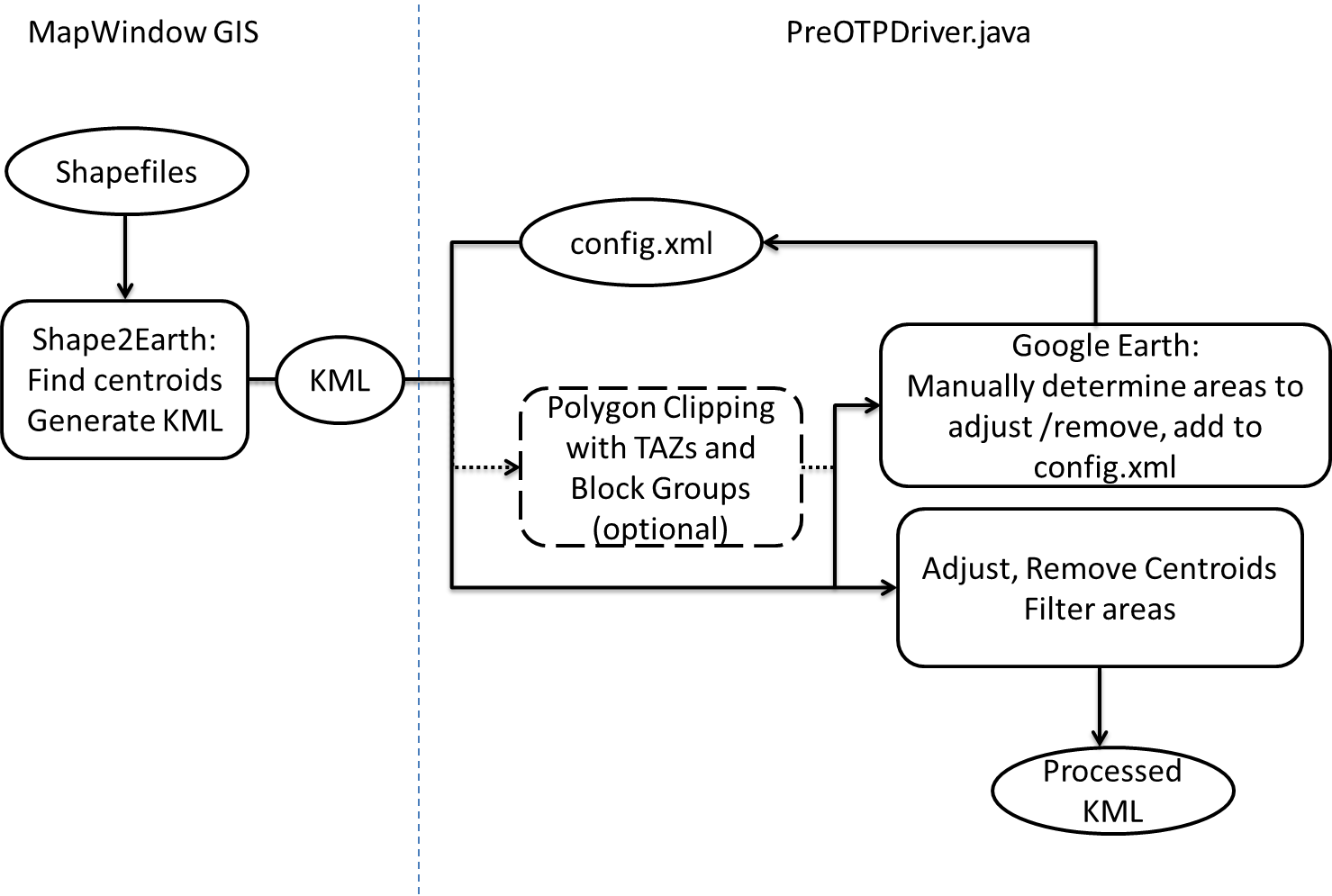
To calculate accessibility, a region is divided up into smaller polygons, e.g., traffic analysis zones or census block groups, and the travel time, by mode of interest, is then calculated between every pair of polygons. To calculate travel times, all trips are assumed to begin and end at the centroid of the polygons. Then, the appropriate demographic data, e.g., employment and population, is factored in to compute the accessibility metric for each polygon. The results can be graphically displayed and if desired, a region-wide average accessibility metric can be calculated. Figure 1 shows an overview of the process, and Figure 2 shows the process of processing the shape file information and determining the centroids.

The next steps are the core of the process: building the graphs of the transportation network and calculating travel times. For transit trips, the process is built around Open Street Maps and OpenTripPlanner. For automotive travel, TIGER files and our own routing routine are used. These steps are shown in Figure 3.

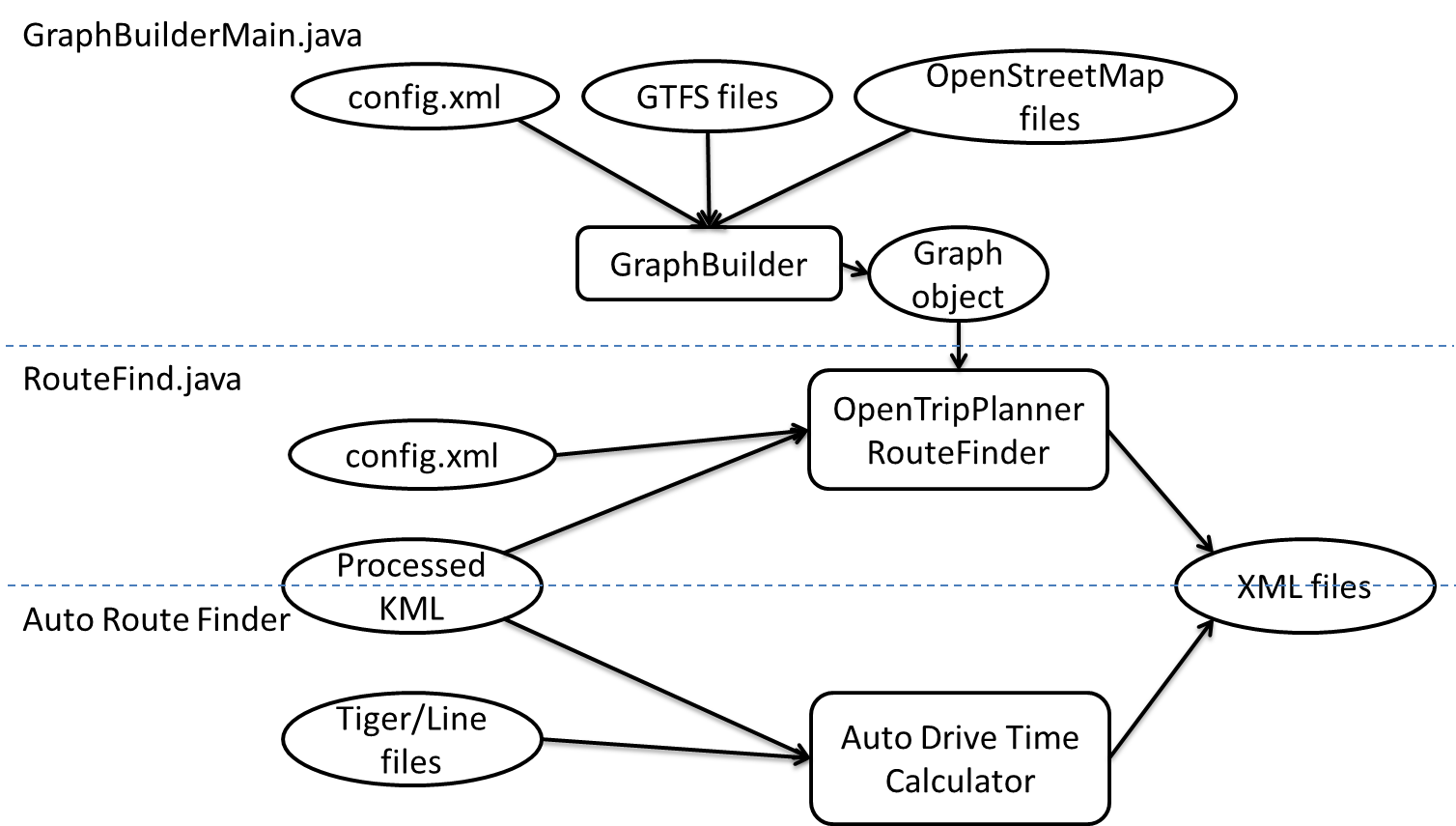
Once the travel times are calculated for all O/D pairs, one can add in the demographics data to calculate the measures of interest. In our initial analyses, we have looked at access to jobs, so we have focused on population and employment values for each polygon (either a traffic analysis zone or block group). In addition, a graphical tool can be used to build 2D or 3D geographic overlays to be visualized using Google Earth. For example, one can plot a heat map of the modal accessibility gap while displaying population as the height of polyhedrons extruded from the polygons. These steps are depicted in Figure 4.



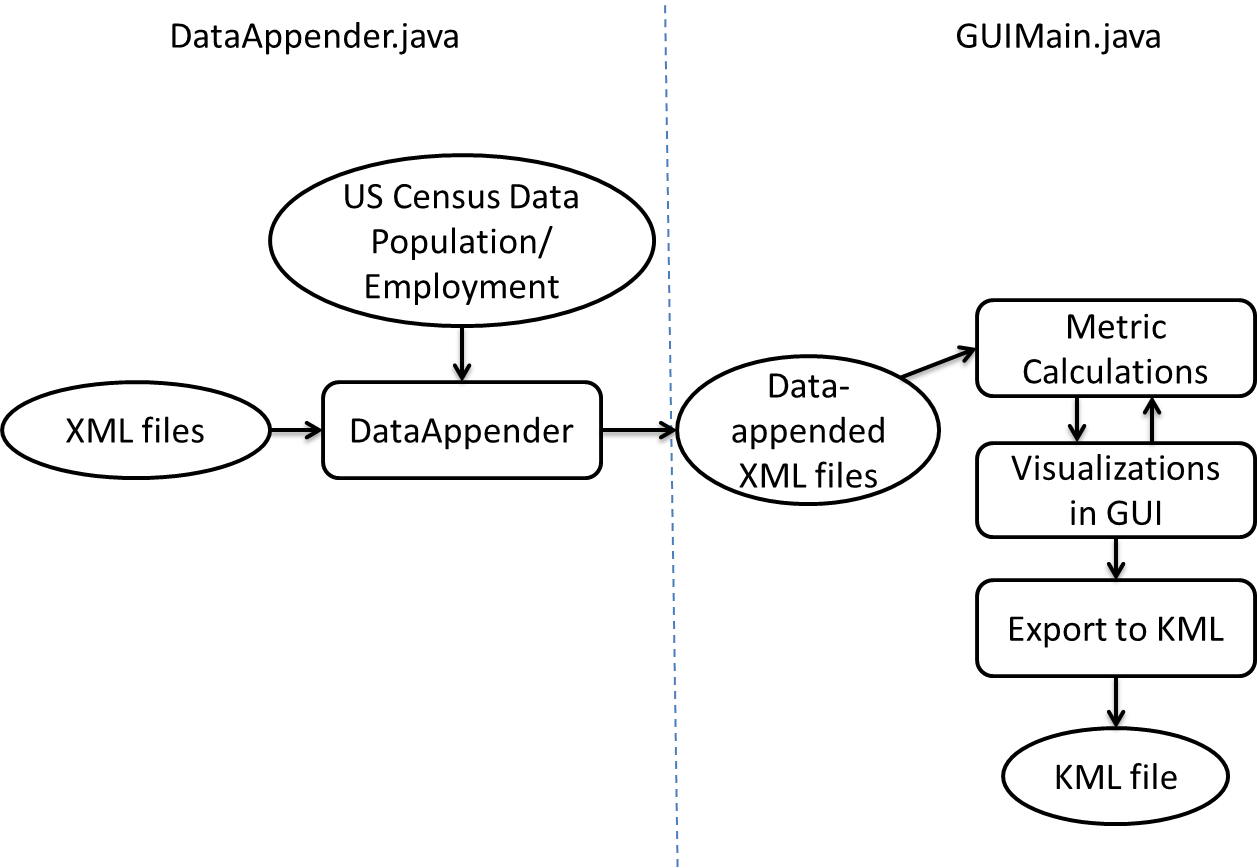
**Figure 1: Process Overview**



**Figure 2: Centroid Generation/Processing**



**Figure 3: Route Finding/Trip Time Calculations**



**Figure 4: Metric Calculations/Visualizations**

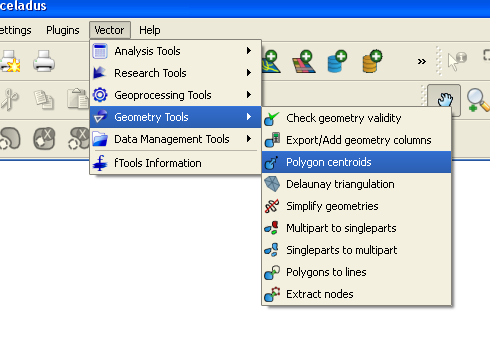
# Overview

## Necessary Files

* Shapefiles (.shp) for the boundaries of the Traffic Analysis Zones (TAZ) and Block Groups in a region you are looking at
* OpenStreetMap (.osm) file for the region you are looking at. Should be large enough to contain the full area of each TAZ and Block Group
* General Transit Feed Specification (GTFS) file set for each public transit agency which provides service in the area
* Census data with regards to population, and employment (or other parameters of interest) are needed for each TAZ and Block Group in order for accessibility metrics to be calculated. Employment data is needed by workplace area, rather than residence area.

## Useful Software/Plug-ins

* MapWindow GIS + Shape2Earth plug-in – Used for calculating centroids and converting ESRI Shapefiles into KML files. Note: The US Census Bureau provides population-based centroids for block groups defined by the 2000 and 2010 Census which can be used as an alternative. 2010 centers of population by Block Group can also be found at [www.census.gov/geo/www/2010census/centerpop2010/blkgrp/bgcenters.html](http://www.census.gov/geo/www/2010census/centerpop2010/blkgrp/bgcenters.html)
* Osmosis – Used to trim or merge OSM files
* Google Earth – Helpful for visualizing KML files
* Java JDK
* Eclipse IDE for Java EE Developers – Environment for setting up OpenTripPlanner and using other tools
* OpenTripPlanner (version 0.4.2) – Used to computer door-to-door transit times. (Changes to OpenTripPlanner may require changes to one or more of the tools in this toolset)
* The tools provided by this project – Preparation of KML files for OpenTripPlanner routing, accessibility metric calculation, visualization
* Optional: Quantum GIS – Can be used for calculating centroids. Quantum GIS is another tool that can be used to calculate the centroids of each TAZ or Block Group. After opening QGIS and loading the Shapefile (add Vector Layer), which contains the shapes for each of the TAZs or Block Groups, there is a feature built into the program that will allow you to output the centroids as a Shapefile.



## Obtaining Necessary Files

*Shapefiles* – Shapefiles of the TAZs or Block Groups are typically provided by the local metropolitan planning organization (MPO) for the area. For example, the Shapefiles for King County were obtained via the Puget Sound Regional Council website at <http://psrc.org/data/gis/shapefiles>. The Shapefiles for the District of Columbia (DC) were obtained via <http://data.dc.gov/>. They may also be obtained from the U.S. Census Bureau. NOTE: Make sure you pick Shapefiles that you can match up census data with. For example, at the time of this writing, there is no employment data available for block groups defined by the 2010 Census, so we had to use block groups as defined by the 2000 Census.

*OpenStreetMap Files* – Depending on the size of the region that you are looking at, two methods can be used to obtain the proper OSM file. If the region is small, it can be found using the online tool provided by OpenStreetMap, <http://www.openstreetmap.org/>. Under the export tab, you can zoom in on the region that you want and form a bounding box which can then be downloaded to your local system. If the region is too large, the site will not let you download the file. Instead, you will have to use the site <http://downloads.cloudmade.com/north_america/united_states/>, which allows you to download the map file for an entire state. In the event you aren’t actually calculating the accessibility for the entire state, you can (and should) cut out the unnecessary regions using an open source tool called Osmosis (<http://wiki.openstreetmap.org/wiki/Osmosis>). Using the command prompt, the following two commands are useful in manipulating large OSM files:

**Merging OSM Files**

osmosis --rx file1.osm --rx file2.osm --merge --wx merged.osm

**Extraction of a Rectangular Region from an OSM File (King County Coordinates)**

osmosis --rx bigFile.osm --bounding-box left=-122.541068 bottom=47.084174 right=-121.065708 top=47.780329 --wx smallFile.osm

*General Transit Feed Specification Files* – For our analysis of DC, we used GTFS files for WMATA and DC Circulator. For King County, we used GTFS files for King County Metro Transit, Sound Transit, and Community Transit. GTFS files can sometimes be obtained via the particular transit agency’s website. For example, WMATA GTFS files can be found at <http://www.wmata.com/rider_tools/developer_resources.cfm>. For the other agencies, you can use GTFS Data Exchange at <http://www.gtfs-data-exchange.com/>. Lists of publically available GTFS files can be found at this website as well as at <http://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds>.

*Census Data* – Census data for a variety of metropolitan areas is available either from the local MPO, the US Census Bureau, or other government agencies. If you cannot locate the proper files for the resolution of interest (TAZ or block group), the data can also be aggregated from block level data obtained from the following sources: <http://factfinder.census.gov/servlet/DownloadDatasetServlet?_lang=en>  
<http://lehd.did.census.gov/led/>

Note on Census Block Groups: They can change from census to census. To identify a block group, you usually need its 6-digit census tract as well. For example, a block group can be identified as Census Tract 001100, Block Group 2. For our purposes, we would store that as “0011002.” The census tract and block group identifier is just a part of the 15-digit identifier for blocks. The format is as follows:

WWXXXYYYYYY**Z**ZZZ

where W’s are digits representing the state, X’s are digits representing county, Y’s are digits representing census tract, and Z’s are digits representing block. The bolded Z is the digit representing the block group.

## Outline of the Process

*Preparing for and Running the Analyses*

1. Find Centroids / Convert Shapefiles to KML (the centroids are used as the O/D for trips)
2. Manipulate KML files to adjust/remove centroids where needed
3. Download OpenTripPlanner (OTP)
4. Build a Graph Using OTP
5. Calculate Public Transit Trip Times for each O/D

*Analyzing and Visualizing the Results*

1. Append Census Data of interest
2. Calculate the accessibility metric for each TAZ or Block Group, roll-up to regional metric if desired.
3. Visualize Data

# Detailed Process

## Preparing for and Running the Analyses

1. **Find the Centroids / Convert Shapefiles to KML**



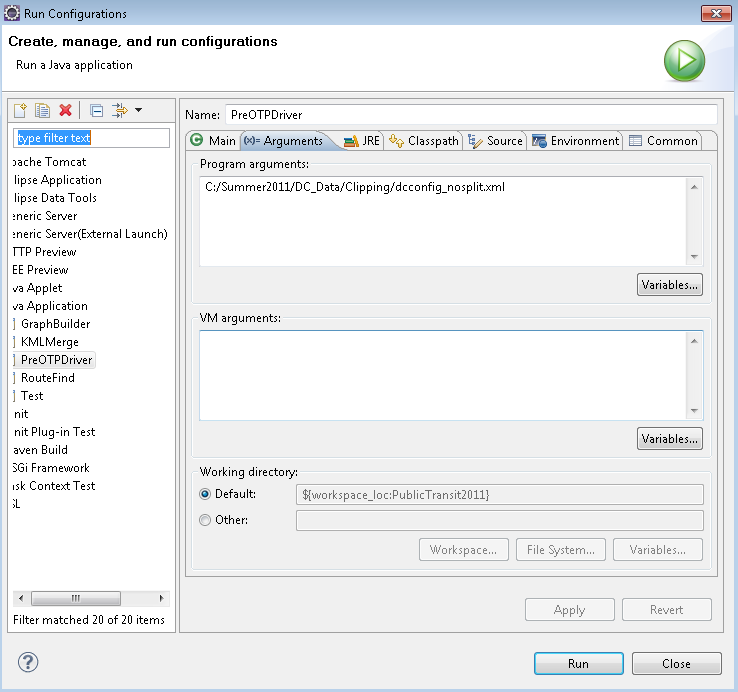
* Open MapWindow GIS. Make sure that Shape2Earth is one of the drop-down menus. If not, under Plug-ins, select Shape2Earth.
* Open the Shapefile with View -> Add Layer
* Find the Legend box. The just-added layer should show up in there under Data Layers. Right-click the layer and select View Attribute Table Editor
* Check the columns to see if there is one that can act as a unique identifier like the SHAPE\_\_ID column, but it cannot be the SHAPE\_\_ID column. If such a column exists (usually the case for TAZs), remember what that column is called. Otherwise, in the editor, click Tools -> Generate or Update MWShapeID Field, then hit apply to finish creating a unique identifier in a column called MWShapeID.
* Close out of the Attribute Table Editor and go to Shape2Earth -> Export to KML
* In the KML tab, you can adjust the opacity of the polygons that will be exported
* In the properties tab, check “Create Labels,” which serves the dual purpose of labeling and calculating the centroid of each polygon, which can then be read in from the KML files. If you want to use centroids from another source, such as population-based centroids provide by US Census, you can write code to overwrite these values later, when they are read into objects called GraphNodes (explained below).
* Under the GIS Data tab, select the column that acts as the unique identifier for the name. For “Select Attributes to Write to KML,” if you are working with Block Groups, select the census tract and block group attributes to be written to the KML, so that census data can be linked later.
* Shape2Earth is limited to exporting 500 shapes at a time unless you use the pay version, so if you have more than 500 shapes and are using the free version, use the Query tab to select up to 500 shapes (shift+click). The KML files can be merged later. If you selected shapes, you need to go to the Options tab and choose “Export Selected.”
* Now press the “Save as KML” button or File->Export->SaveAs KML/KMZ and save the file.
* The file KMLMerge.java can be used to merge KML files produced by Shape2Earth. To run it, pass in a config.xml, putting the paths to each KML file inside tags for <merge>, and specifying the output file path with <output>.

NOTE: In order for other tools to be able to use the exported KML files, make sure that the correct information from the Shapefiles is exported. If you don’t do this, the program won’t know how to link up the census data! For TAZs, the data field needs to be included as the name field in the KML. For Block Groups, a unique identifier needs to be created for use as the name, and the census tract and block group attributes also need to be exported to KML, where they will appear in the description field.

As mentioned above, when using block groups one can also use the population centers by block group data provided by US Census for 2000 and 2010. In the util package, PopulationCentroids.java produces a KML file so you can visualize the centroids. Since the file does not have the block group boundaries, you can open another KML file that you produced using a Shapefile (from the appropriate census). Now you have an overlay of the block groups and also the geometric centroids. If you want to work with population-based centroids, you can write some code to read the population-based centroids KML file and set new centroids for the GraphNodes with the setCentroid() function.

1. **Manipulate KML Files**

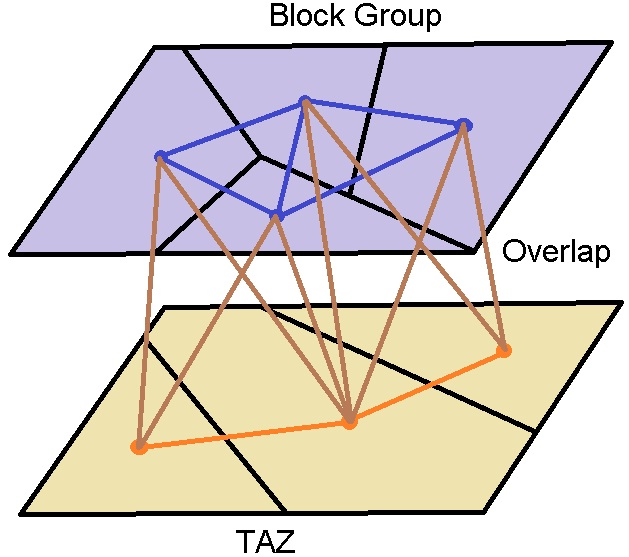
This section talks about the Java code that takes the KML produced by Shape2Earth and massages it into the KML file that is used by the RouteFind.java to calculate transit times with OpenTripPlanner. Almost everything in the following sections is contained in PreOTPDriver.java in the preotp package. GraphNode is an object that was created to represent TAZs, Block Groups, or subareas. Here is a step-by-step look at what the user needs to do and what the code is doing. More detailed explanation will come afterward.

* 1. User writes a XML configuration file Details can be found in the section below titled PreOTPDriver.java
  2. User runs PreOTPDriver.java, passing in the configuration file as the argument  
     
  3. Program reads configuration file
  4. Program reads in KML files for TAZs and Block Groups
  5. If polygon clipping is wanted, the program adjusts the coordinates of the polygon vertices to avoid special cases that throw off the clipper. Adjacencies are determined, and the clipping algorithm is run. Clipping can be used to break up Block Groups into smaller areas that align with TAZ boundaries. However this greatly increases the number of areas, and the number of route calculations increases as the square of the number of areas. It is not recommended.
  6. If the option for manually removing or adjusting centroids was selected, the program manually removes/adjusts centroids based on the information provided in the configuration file.
  7. The program filters out GraphNodes if the area is too small (as may occur if polygon clipping is used) or the centroid is too far from a transit stop. It renumbers the GraphNodes so there are no gaps in the numbering, but should maintain the TAZ, census tract, and Block Group to which it belongs.
  8. The program writes the KML file that can be used by RouteFind.java to calculate door-to-door transit times with OpenTripPlanner.

*GraphNodes.java*

When the KML files are read, it reads the number identifier of the polygon and the coordinates of the centroid. It also reads in the vertices of the polygon. Sometimes a polygon has an inner boundary that results from an enclosed polygon. We break the outer polygon into two polygons so there are no more inner boundaries. This information is used to create a GraphNode, which can then calculate things like area. The GraphNode keeps track of TAZ, Census Tract, and Block Group. If you clip the polygons, the GraphNode will know all three of those values. If you don’t clip the polygons, it will only know either its TAZ or Census Tract/Block Group, depending on what you are working with.

The idea behind using a graph is that knowing which areas are adjacent or are overlapping makes polygon clipping more efficient. We make each TAZ or Block Group or subarea a node in the graph. There are two layers in the graph: one for TAZs and one for Block Groups. The edges represent an adjacent TAZ or Block Group, or an overlap between a TAZ and a Block Group (see picture). Finding adjacent areas only requires iterating through all the vertices once. Finding overlapping areas is slower, but knowing adjacent areas limits the number of areas you need to check for overlaps. Once you have this information, you only need to clip TAZs with Block Groups that are known to overlap with the TAZ.



*PreOTPDriver.java*

Input: XML Configuration File

Tags:

<split> - 1 if you want polygon clipping, 0 if you do not

<keepTAZ> - only used if split is zero, 1 means unclipped TAZs, 0 means unclipped Block Groups

<manualadjust> - 1 if you want manual adjustments of centroids to happen, 0 if you don’t

<kmlpath\_taz>,<kmlpath\_bg>,<output> - file paths to TAZ and Block Group KML files, and file path to where you want the output KML file

<round> - only used if split is 1, determines how you want to round the coordinates. A -5 means you round to the nearest 10-5

<areaFilter> - sets the area threshold for eliminating subareas that are too small. The argument should be between 0 and 1, and represents the fraction of the area of the smallest TAZ or Block Group, so 0.5 means eliminating subareas smaller than half the smallest TAZ or Block Group

<gtfs> - file path to GTFS stops.txt. You can have more than one GTFS tag if you are dealing with multiple transit agencies

<maxDist> - the maximum straight line distance a centroid can be from a stop. If the distance is greater than this limit then the centroid and hence the area does not have transit access. If you want to set it to about 800 meters, the argument here would be 0.008 because we are dealing with longitudes and latitudes.

<remove> - content is the ID of the GraphNode to be removed. To determine which IDs to remove, set <manualadjust> to 0, look at the output KML in Google Earth, put the IDs you want in the configuration file, and set <manualadjust> to 1

<adjust id=”x”> - the attribute is the ID of the GraphNode to be adjusted. The content holds the new centroid coordinate. To determine which IDs to adjust, set <manualadjust> to 0, look at the output KML in Google Earth, put the IDs you want into the configuration file, and set <manualadjust> to 1

Output: KML file that is ready for use with RouteFind.java and OpenTripPlanner

*Aspects of PreOTPDriver.java*

Finding Adjacent Nodes

This step is only used if you wish to do polygon clipping so that Block Group level analysis can be rolled back up to TAZ’s, since Block Groups often cross TAZ boundaries. If this is not desired, this does not need to be done.

Finding adjacent TAZs or Block Groups within the layer is easy to do because adjacent areas will share at least one vertex. You can use a HashMap and iterate once through all the vertices to find adjacent areas.

Finding overlaps between TAZs and Block Groups uses the adjacent areas just found. For each TAZ vertex, we find a Block Group that contains it, and vice versa. For example, let’s say you know a TAZ vertex is in a certain Block Group. When you move to the next vertex in that TAZ, you should check the Block Group you just found first because it is likely to contain the new vertex, and if necessary, do a breadth-first-search from that Block Group until you find one that contains the vertex.

Polygon Clipping

This is used if it is desired to be able to roll Block Group results back up to TAZ’s.

Weiler-Atherton Polygon Clipper (what I’ve implemented)  
<http://cs.fit.edu/~wds/classes/graphics/Clip/clip/clip.html>  
<http://pierloic.free.fr/clip/clip.html>  
Issues: Not robust. Parallel lines, intersections near vertices cause problems.

Vatti Polygon Clipper  
<http://en.wikipedia.org/wiki/Vatti_clipping_algorithm>  
<http://www.cs.man.ac.uk/~toby/alan/software> (not completely open-source)

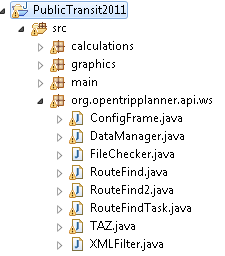
Improvement on Vatti  
<http://davis.wpi.edu/~matt/courses/clipping/>

Filtering/Adjusting/Removing Nodes

After clipping the polygons, we filter out polygons that are too small to worry about. We also read in the stops from GTFS files and remove polygons whose centroid is not near a transit stop. The size and distance thresholds are set in a config file. After that, regions that contain water may need to be removed or have their centroids adjusted manually. In the config file, you can put in the number identifier of the polygons you want to remove and put in new coordinates for centroids.

1. **Download and Setup for OpenTripPlanner**

As useful as OpenTripPlanner may be, it is also the most likely to cause the greatest amount of annoyance and confusion. OTP is implemented in Java, and can be loaded in Eclipse IDE. Eclipse can be downloaded from <http://www.eclipse.org/downloads/packages/eclipse-classic-360/heliosr>. Full details on how to get the project up and running can be found at <http://opentripplanner.org/wiki/Install>. Once the project is set up in Eclipse, you’re going to want to include the opentripplanner projects in the buildpath (Right click on your project -> Build Path -> Configure Build Path…). To calculate public transportation trip times, make sure you have ConfigFrame.java, RouteFind.java, TAZ.java, and XMLFilter.java in your project.

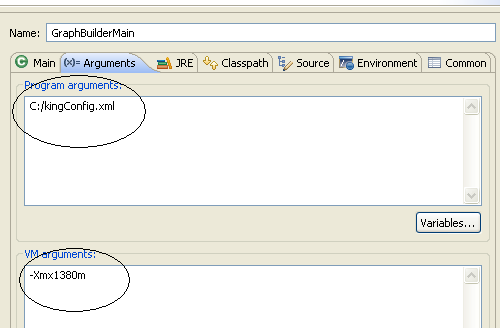


OpenTripPlanner is in the midst of transitioning from SVN to GitHub, and the installation process for GitHub is not yet documented on their website. As a result, there are some issues with including the opentripplanner projects in the buildpath. We believe the issue is Maven-related, but as of August 11, 2011, we have not been able to fix it. We have not been able to run GraphBuilder or RouteFind.java since switching to GitHub.

1. **Build a Graph Using OpenTripPlanner** (<http://opentripplanner.org/wiki/GraphBuilder>)

The directions to create the Graph.obj file needed to use OpenTripPlanner can also be found on the website noted above.

Once you have OpenTripPlanner set up in Eclipse and have access to the GTFS files that you would like to incorporate into your graph, you need to write a configuration file for use with GraphBuilder. Two working configuration files for DC and King County are included in the files. To build the Graph object, you need to pass the path to the configuration file as an argument to the main class GraphBuilderMain.java. You’ll also want to alter the Java Virtual Machine (JVM) arguments to increase the amount of memory you want to allocate towards the program. The OSM files are often fairly large and the program consumes quite a bit of memory in the process of building the Graph object. If not enough memory is allocated, the program will throw an exception and exit execution. The maximum you can set the memory on a 32-bit processor is 2GB in theory, but in reality is closer to 1.5GB. We have not had memory issues with DC or King County since switching the OSM Provider to StreamedFileBasedOpenStreetMapProviderImpl from FileBasedOpenStreetMapProviderImpl. It may be worth exploring RegionBasedOpenStreetMapProviderImpl, which lets OpenTripPlanner download OSM data based on the stops in your GTFS files. Depending on the size of the GTFS and OSM files, the execution of GraphBuilder can take anywhere from 10 to 30 minutes.



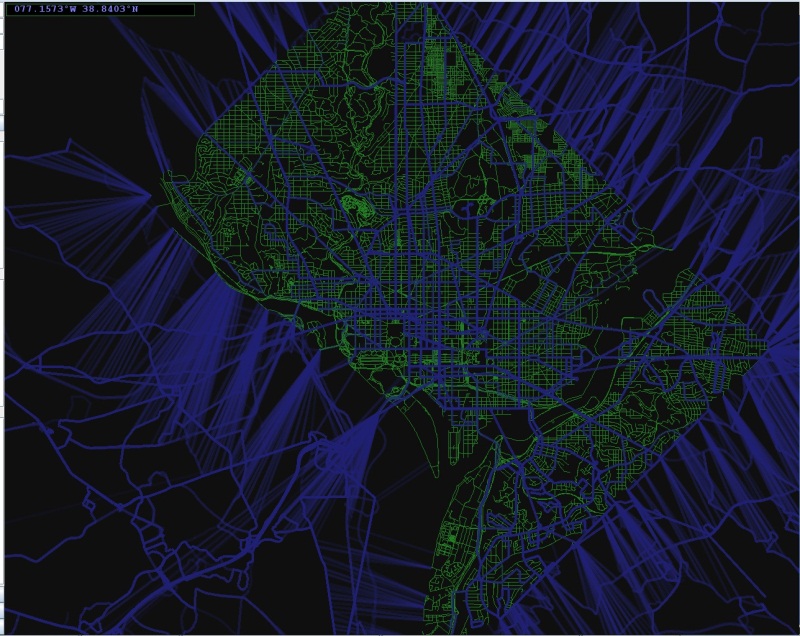
I suggest first using the GraphBuilder and VizGui jar files to build and visualize graphs. This allows you to use VizGui to visualize the graph and make sure it’s what you want. Here are the commands for using the jar files to build and visualize graphs:

### Build graph

java –Xmx1500M –jar C:\PathToGraphBuilder.jar C:\PathToGraphBuilderConfig.xml

### Visualize graph (VizGui)

java –Xmx1500M –jar C:\PathToVizGUI.jar C:\PathToGraph.obj



If the graph is what you want, you can now build the graph in Eclipse with GraphBuilderMain.java by setting the run configurations like in the screenshot, with a configuration file and JVM settings. You must build the graph in Eclipse if you want the graph to be read correctly by RouteFind.java. Conversely, the VizGui jar cannot read graphs created from within Eclipse.

If you are satisfied with your graph, you might also want to consider investing some time to let GraphBuilder optimize the graph for non-transit modes (walking, biking, etc…). It will take longer to build the graph, but route finding will speed up. To give a sense of the potential performance improvement, for DC, we spent an extra half hour to fully optimize for walking. This doubled our route-finding speed, so when we were running about 250,000 trip calculations, we saved about 16 hours of computation. The current configuration files are set to fully optimize walking since we only consider walking and transit. When tinkering with graphs, you’ll want to comment out the highlighted section in the screenshot, and uncomment that section when you make a final graph.

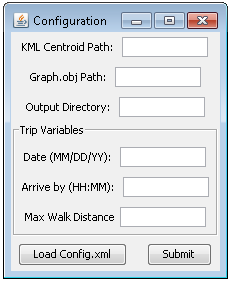


Sometimes the GTFS file contains information for more than one transit agency. For example, King County Metro GTFS also includes Sound Transit and King County Marine Division. Sometimes the GTFS file may cause problems. If that happens, try a GTFS file from a different date. You can use a GTFS validator if you wish (<http://code.google.com/p/googletransitdatafeed/wiki/FeedValidator>), but there will almost certainly be many warnings and no errors. Checking the King County Metro GTFS with a GTFS validator there are 370,000 warnings. Using the 2010 King County Metro GTFS works, and there are only 180,000 warnings. I don’t think you can mix and match GTFS files from different time periods for the graph, so if you use a GTFS file from Summer 2010, other GTFS files should be from around then also.

1. **Calculate Trip Times**

**5.1 Public Transit Trip Times**

If you have the source files set up in Eclipse, the file you’ll want to use to calculate public transit trip times is RouteFind.java. Once you run the program, a box will pop up asking you to select the proper directories to run from. You’ll also have the option of loading the information from a configuration file, config.xml. A sample configuration file is included in the files provided to show the proper formatting necessary for the program to work. After you submit either your own settings, or the setting specified in the configuration file, the program will proceed to read through the centroid KML file and find the routes between every point in the file.



The current setting that the program uses specifies that the user must *arrive-by* the given argument and if you want to change this or any other settings, you’ll want to alter the settings in the initRequest() method in RouteFind. Once your settings are correct and the program runs, it will incrementally output each TAZ or Block Group to the specified directory. If the execution is interrupted or needs to be stopped for any reason, you can add a simple for-loop in the source to get you back to the point you were previously at and continue execution. If an exception is thrown during the course of execution, the most likely cause of the problem is that the point needs to be closer to either a road or one of the transit stops. If the first TAZ or Block Group outputs with no issues, then the rest of the routes should be calculated without issues as well considering if they can both reach the same point, then they can obviously reach each other. For each centroid, an XML file is created, containing the transit times to the other centroids and some empty fields where census data can be added. The code is not aware of whether a centroid came from a TAZ or block group or subarea. It just calculates the shortest trip time using centroids as origins and destinations. Don’t be alarmed that the output file names and the variables in the source code make it appear to only work with TAZs. If you put in a KML with Block Group centroids, you will get back transit times between all the Block Groups.

Trip calculations can range from <100ms to >2 seconds per trip. In the long run for DC, it comes out to be about 240ms per trip, or 4 minutes to calculate 1000 trips. For King County, which has a larger graph, it comes out to be about a second per trip. Runtime also depends on your processor.

RouteFind2.java and RouteFindTask.java are the result of my brief attempt at trying to run multiple threads with no prior experience. It runs, but I am not seeing any performance improvement. Here are some of my thoughts (might be way off): I am not sure if multiple cores are being used. Maybe having just one graph is a bottleneck if most of the time is spent route-finding on the graph and only one thread can access the graph at a time. However, having multiple graphs requires more memory, and I run out of memory running with two copies of the graph.

Other possible errors during routing:

* TransitTimesException – pick a different date closer to when the GTFS files are dated, because the current date is probably not in the valid range for the GTFS data
* 2011-07-20 11:18:57,145 WARN [StateEditor.java:192] : A state's time is being incremented by a negative amount while traversing edge org.opentripplanner.routing.edgetype.Alight(<Sound\_61080\_KCM\_18114747\_88835011\_A> -> <Sound\_61080\_arrive>)

2011-07-20 11:18:57,145 ERROR [StateEditor.java:73] : Defective traversal flagged on edge org.opentripplanner.routing.edgetype.Alight(<Sound\_61080\_KCM\_18114747\_88835011\_A> -> <Sound\_61080\_arrive>)

I think this error is the result of faulty GTFS files. If I try out the 2011 King County Metro GTFS file by itself, it doesn’t work, so the issue isn’t caused by the meshing of different GTFS files by GraphBuilder. This error doesn’t happen if I use an older GTFS file from 2010.

Utility: FixXML.java

FixXML.java is some code snippets that were used to correct XML files generated by RouteFind.java. One of the two issues that were fixed with this utility was later resolved in the code. The other issue has to do with what time zone you are in. The visualization tool reads in start and end times as Date objects, but the parser does not accept certain time zones (e.g. AKDT – Alaska Daylight Time), so if you are outputting XML files with a time zone that is not accepted, you will have to use this utility to change those time zones to one that works, like EDT – Eastern Daylight Time. The inputs and outputs are hardcoded, so take a look at the code.

**5.2 Automobile Trip Times**

**Necessary Files:**

• TIGER/Line files (.RT1 and .RT2) for the area you are analyzing

**Obtaining Necessary Files:**

TIGER/Line Files – These files can easily be obtained from the census bureau at http://www.census.gov/geo/www/tiger/tiger2006se/tgr2006se.html. The files are organized by county, and you may have to use a lookup table to locate the county or region that you would like to download.

**Calculate Automotive Trip Times:**

If you are running the source files for the automotive trip calculator, the main class is Interface.java. Once the program is run, it will ask for input and output directories as well as other configurations and has the option to just load the configuration stored in config.xml. As the program begins to run, if an alert message pops up saying that one off the two points may need to be moved, then the duration of the trip was considered to be abnormally short and such trips are often associated with points being misplaced or not properly apart of the network of roads. If the two points are actually just very close to one another, you can just ignore the message and continue execution of the code.

As the program runs, it will append the driving time data to the previously generated XML files created by OpenTripPlanner. Due to this dependency, transit trip times need to be calculated first before automotive trip times can be calculated. However, if necessary, the source files can easily be altered to create a default XML file and append the data as you’d like.

## Analyzing and Visualizing the Results

1. **Append Census Data**

For TAZs:

Assuming that you have access to demographic data for each of the TAZs they can be added to the XML data files using the file DataAppender.java. In order to use this program you need to have an Excel file (.xls, **not .xlsx**) with the TAZ in the first column and the data which you want to append in the second column. If the data you wish to alter requires alterations for all of the destinations from a particular TAZ, then the TAZ’s in the first column correlate to the destinations of the selected TAZ which will be altered. For other pieces of data, such as population data, the TAZ’s in the first column will be associated with individual XML files and each one listed will be altered and updated. As a default setting, any census data added will be listed as from 2010 in the XML file. Of course, this can be altered to change values for varying years. When adding census data, please note that it is not necessary to list all TAZ’s in the XML and that the program will only alter those that are specified in the Excel file.

For Block Groups:

For Block Groups, DataAppender2011.java is the file that loads population and employment data into the XML files produced by RouteFind.java. The population data is in text file format and the employment data is in CSV files. You need to make sure that the census data matches up with the block groups you are using. For example, the block groups from the 2000 Census and 2010 Census are different, so if you have data from different censuses, the block groups will not match up correctly. The file paths to the population and employment data are hardcoded, as are the directory with the XML files and the output directory. Make sure to change these to work with your system. The data appended will always be labeled as from the year 2010, but that is because the visualization tool is hardcoded to look for data tagged with the year 2010.

1. **Calculate the accessibility metric for each TAZ or Block Group, roll-up to regional metric if desired.**

Accessibility metrics are calculated by the visualization tool. Two types of accessibility metrics can be calculated.

Cumulative Opportunity:

Ci = ∑ Oj Bj

Ci Opportunities (jobs in this study) reachable from zone i

Oj Opportunities (jobs) at zone j

Bj A binary value equal to 1 if zone j is reachable within a predetermined time from zone I, otherwise equal to 0

This measures the number of jobs available to the residents of a given TAZ within a given transportation time. It has the advantages of being both simple to calculate and easy to understand. The units are number of jobs. However, the function uses a step function: all jobs within the travel time limit are counted equally, regardless of how quickly they can be reached, and jobs just one minute beyond the travel time limit are not counted at all. The gravity model addresses these limitations.

Gravity Model:



Ai Accessibility at a given zone, i

Oj Opportunities, in this case jobs, at a particular zone, j

Cij The time the mode of transit took to get from TAZ i to TAZ j

A downside of the gravity model is that it yields a unitless index whose only meaning is as a measure of relative value.

Modal Accessibility Gap:

In addition, the Modal Accessibility Gap (MAG) index between transit and automotive accessibility (12) is also calculated:



Ap Transit Accessibility

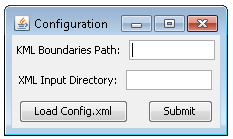
Ac Automotive Accessibility

The values for the MAG range between -1 and +1. At zero the two modes provide equal accessibility.

In the code, accessibility is first calculated in GUIMain.java while subsequent calculations are done in ModelCalculator.java.

1. **Visualize Data**

Input: XML Configuration File loaded from a GUI



Output: KML file

If you run the program via the source code, the main file is GUIMain.java. Once the program is run, a window will open and ask for the path to the KML file of the boundaries of each TAZ as well as the directory which contains all of the XML data files. Alternatively, you can use the load config.xml button to load the settings. The configuration file is actually called configviz.xml to avoid confusion with the config.xml for RouteFind. All the shapes in the KML file get drawn so if not all of the XML files match up, they will just be colored black and left out of the equation. So if you load the program and all of the shapes are black, then somewhere along the lines the zone number and the XML file are not matching up. If one of the metrics shows up as either all green or all red, there may be an outlier in your data, or the data itself may be missing and basing the calculations off a default value of 0. Currently when you choose to extrude the current model there is a placeholder for a future metric that is currently just called name. If it is chosen it will just extrude by the color of the current model, though the file ModelCalculator.java could easily be altered to add a new metric to extrude by. If you plan on altering any of the metrics or adding a metric, you’ll need to add a button for it in ControlPanel.java and then have the program do the actual calculations in ModelCalculator.java.

When you click around on the map of the GUI and the map seems to distort itself where it looks like it is only half there and it is at the bottom of the screen, or really any other painting error, it just seems like a glitch in Java’s graphics or the way it is interpreting the current values. Just zoom in and then out and the screen will be back to its normal viewing state. Another little glitch with the graphics is at the start of the program when the splash screen is displayed. Sometimes it will look like the frame never closes. However, what is really happening seems to be some sort of repainting issue with the current frame and despite the frame actually being closed, it’s image gets painted on the other frame. If you do something to that frame such as scrolling up, which forces a repaint, you’ll notice that the image goes away.

The current version of the visualization tool works for transit only, but it is more robust with handling missing data. First, you don’t need a XML file for every TAZ or Block Group, so if you are in the middle of running transit calculations, you can visualize what you have. Second, an area doesn’t need to be reachable from all other areas. If there is no information, the area will show up as black. Third, errors won’t happen if data isn’t appended. If you click on an area with no data, the areas that do have data will turn blue.

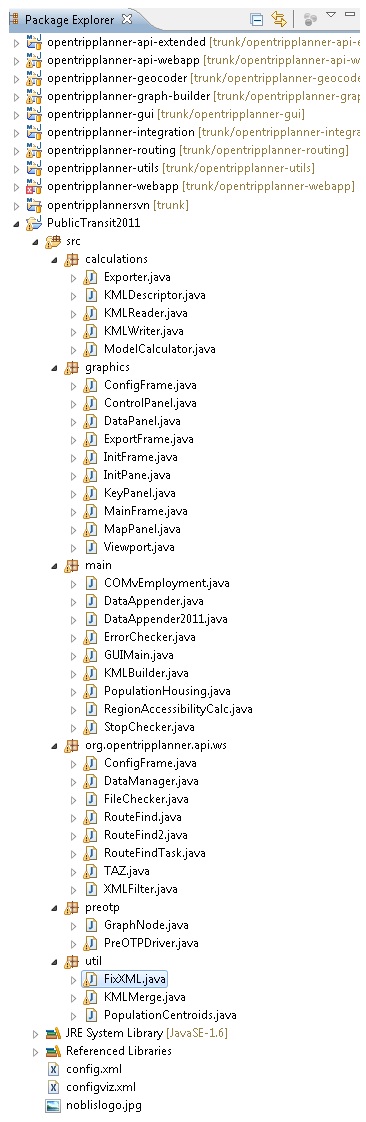
There are some missing libraries (e.g. PushingPixels), but it only affects the appearance of the GUI and none of the data visualization. Commenting out code where necessary took care of that. Automotive display should be easy to restore (it was commented out when the tool could not handle missing data).

# Folder/File Structure (as of summer 2011)

/Summer2011  
 /DC\_Data  
 /CensusData – census data files  
 /Clipping – config.xml for PreOTPDriver.java, output of PreOTPDriver.java  
 /GraphBuilder – config.xml for GraphBuilder, OSM files, Graph object files  
 /GTFS – GTFS files  
 /OTPOutput – XML files created by RouteFind.java  
 /Shape2Earth – KML files created by Shape2Earth, merged KML file  
 /Shapefile - Shapefiles  
 /Downloads – GTFS validator, JAR files, some libraries  
 /King\_Data – same structure as DC\_Data  
 /Pics – some screenshots

/workspace – Eclipse project, source code, configuration file

***Eclipse Project Explorer Screenshot***



# Future Work

The use of a database would be very helpful for this project. Everything could be kept in a database, so instead of reading KMLs/XMLs and keeping track of file paths, it would just be database queries.

Making a graph with just GTFS files might be one way to speed up runtime because you wouldn’t have all the street nodes/edges. You would first need to calculate walk times to transit stops for each centroid, but that is less calculations. If the OTP graph traversal times with the GTFS-only graph are significantly faster, there could be some major savings.

To keep runtimes down, parallel processing can be used. Threads do not have to communicate with each other. Each thread can produce one of the OTP output XML files.

# Overview of Classes (documented summer 2010):

Overall and Transit Accessibility  
-calculations

* Exporter

Takes the current model being visualized and using JDOM creates a KML files which is then outputted to the specified directory. Used in both the main GUI tool as well as KMLBuilder to export the models in bulk when necessary.

* KMLDescriptor

The primary purpose of this class is to provide a means of describing what model is being selected to be outputted as KML. Essentially just uses whether each button is selected in the control panel and stores that information so that it can be used later for output.

* KMLReader

This class opens a KML file which contains the coordinates of the boundaries for each TAZ and parses the file in order to create a mapping between each zone and the Shape for the particular zone.

* KMLWriter

Primarily used with KMLBuilder, originally created in order to convert an XML file containing information about KML files and goes ahead and builds the specified files. Later altered to work with KMLBuilder to read to\_build.xml and outputs the KML files as listed in the configuration files.

* ModelCalculator

Using which buttons are selected in the control panel as input, determines which metric or model is being selected and using stored TAZ information calculates the metric and normalizes the value and stores the information as a color and maps it to the particular TAZ.

-graphics

* ConfigFrame

Creates the frame which pops up at the start of execution of the GUI tool and passes the data, whether it is the data loaded from the configuration file or the data selected manually, along to GUIMain.

* ControlPanel

Creates the panel that appears on the right hand side of the GUI and allows the user to select which model should be calculated. Calls on the ModelCalculator to calculate the model and passes that information along to the map to be painted. If adding a metric to be calculated, you’ll want to add a button in this class.

* DataPanel

Creates the panel on the top right hand side of the GUI which displays the information for the selected TAZ

* ExportFrame

Creates the frame that pops up when the user selects to export the current visualization to KML. Using the model that has been calculated and what the user selects to extrude the metric by, uses the Exporter class to output the data to the specified path

* InitFrame

Creates the splash screen which is displayed while the XML files are loading and properly increments the progress bar as each file is loaded

* InitPane

Reads in the Noblis logo as a BufferedImage and draws the image on the panel which will be added to InitFrame and displayed while the XML files are being loaded

* KeyPanel

The only purpose of this class is to create a panel which draws the range of colors which will be used in the heat maps so that the user has an understanding of what is a high and what is a low value

* MainFrame

This class contains all of the code which is used to take all the elements of the GUI and put them together and display them. If you want to change the arrangement of the GUI, you’ll most likely want to start in this class.

* MapPanel

Creates a panel which is used to draw each of the shapes for the TAZs and using the color mappings provided by the ModelCalculator fills the shapes accordingly. Also contains code for how the screen coordinates are transformed into world coordinates

* Viewport

Takes all of the shapes for each TAZ and using the union function of each shape, determines the most northern, eastern, western, and southern points to use as a boundary for setting the viewport and drawing the shapes to the screen

-main

* COMvEmployment

Used to develop data which could be copy and pasted into Excel and analyzed, prints out a comparison of accessibility and cumulative opportunity values for both public transit and automotive modes of transit for each TAZ

* DataAppender

Helper program to allow the user to upload values from an Excel file and append the values to particular XML data files

* ErrorChecker

Sample class used to show how points gathered from WMATA’s online trip planner were compared against actual values. Would need to change path for the actual file to read and compare.

* KMLBuilder

Helper program which allows the user to add configurations to a list and ultimately output, in bulk, particular KML files to work with

* PopulationHousing

Not part of any of the programs, merely serves as an example of how to possibly aggregate block data and then append that data to the specified directory of XML files.

* RegionAccessibilityCalc

Not a part of the main visualization, just a standalone program which reads the XML data files and calculates the average percentage of jobs reachable from each TAZ within a region

* GUIMain

The main class which runs the GUI visualization program. Asks for a configuration via a ConfigFrame, which then initializes data via InitFrame and finally opens the MainFrame to display the data stored in the specified directory of XML files.

* StopChecker

Serves as a standalone example program which takes in acreage data for each TAZ as well as loads the list of transit stops and determines the average number of stops per acre for each TAZ. Could be used to determine which TAZs to use in an analysis

-org.opentripplanner.api.ws

* ConfigFrame

Essentially the same as the ConfigFrame listed above, although it is altered to account for the change in required input from the user.

* DataManager

Not essential to calculating transit trip times, however it can be used to load an XML dataset and just prints out the information stored for transit trips in a frame graphically.

* FileChecker

Again, not an essential program, though can be used to load a directory and it’ll check whether any transit routes require a walking distance of over one mile

* RouteFind

The main file for calculating public transportation trip times. Loads the Graph.obj and then uses the files contained in OpenTripPlanner to find the fastest route and stores the data in TAZ objects and also outputs all of the information to a specified directory.

* TAZ

Primary file for storing all transit route data. Anytime you need the ability to store more information, retrieve information, or alter the way that the information is outputted as XML, you’ll want to do it in this file

* XMLFilter

A file filter used when reading a directory and seeking out files whose extension is .xml.

Python Script:

ie.py – This is just an example of a file I used to gather data from WMATA’s online trip planner and then later compare against the results of OpenTripPlanner. You could write a similar script for other sites, however this particular script will only work with WMATA at the current time.

**Configuration Required for Main Classes:**

**Interface** – A Configuration file is optional to run this program, however you will need to know the path for the input directory, which is the directory of the XML files to be appended. You’ll also need to specify an output directory, which can either be the same as the input directory, or different depending on where you want the data to end up. You’ll also need to specify a KML file which contains the coordinates for the centroids and the directory which has the TIGER/Line files. If you are using a configuration file, make sure that it is in the same folder as the src folder of the project so it becomes visible to the program.

**GeoEncoder** – Configuration file not necessary, though you will be prompted for the TIGER/Line directory as well as the path to the KML file with the centroids.

**RouteFind** – Configuration file is not necessary to run this program, though make sure you have already created Graph.obj and have a KML file with the coordinates of each centroid.

**DataManager** – Because this is a standalone program only meant to take a peek at the data outputted, it currently takes no configuration and if you want to see a different XML directory loaded, you’d have to manually change the file name as noted in the code.

**FileChecker** – Another standalone program which does not require a configuration file, though you would have to change the directory in the code to load the proper directory that you want.

**COMvEmployment** – This file requires the use of the configuration file, and loads whatever files are specified in the document.

**DataAppender** – Does not use the configuration file, though will prompt the user for a directory to load the XML data files, as well as an excel file which contains the data to be added. Please note that the Excel file should be .xls and not .xlsx because the latter will often cause an error at runtime. Also note that the first column should be the TAZ number to be altered and the second column should be the value that will actually be added.

**GUIMain** – The use of the configuration file is optional, though it will require access to the XML data files, a KML file containing the boundaries of each TAZ and a path for the logo to be displayed as a splash screen.

**KMLBuilder** – This file requires the use of the configuration file and whatever is specified as the current paths/directories will be used.

**PopulationHousing** – For this program to work, you need to alter the paths in the source code to the actual files that you are using for the conversion between TAZs and blocks and the actual block data to be aggregated.

**RegionAccessibilityCalc** – Strictly uses the configuration file, so make sure to specify the fields properly.

**StopChecker** – Standalone program which does not use the configuration file. Though in the source code you will need to specify the path to the KML file of the boundaries of each shape as well as the stops.txt file from a particular transit agency’s GTFS file.

**GraphBuilderMain** – You must pass this program a path to a configuration file that tells GraphBuilder exactly how to build the graph. Using the sample provided, you can alter the file to include your own particular GTFS files and OSM data.

**VizGui** – Requires you to specify the location of the graph object as an argument passed to the program.

## Automotive Accessibility Calculator

-main

* ConfigFrame

Very similar to the above listed configuration frames, though has been altered to only allow for input of the necessary files and directories

* GeoEncoder

Not used in the automotive calculator, and is a standalone program. Used to read in a KML file and a TIGER/Line directory and output a text file which associates a road segment with each TAZ which could then be used in the ie.py to check data against an online trip planner

* Interface

Main file which is used to run the automotive trip calculator and append the data to the specified XML files. Loads the database of roads using TIGER/Line files and then parses the KML file to create a mapping between each zone and its centroid. It then takes those mappings and finds the fastest route between each of them and alters the appropriate XML file

* RTFilter

File filter used to filter a directory down to only files ending with .RT1 and .RT2

-mapping

* Compass

Calculates the direction of a RoadSegment

* Edge

Class used to model the edge of a graph which consists of RoadSegments, the object contains “way” data which could be used to model one way roads.

* Node

Models a vertice in a graph which tracks which edges it connects to and also stores the incoming edge which is used for Dijkstra’s algorithm. If you want to limit which points can be connected by edges, you’ll want to use this class as a possible means.

* Point

Converts a point from TIGER files into latitude and longitude coordinates.

* RoadSegment

Stores information about a particular RoadSegment including starting and ending points and the name of the actual road.

* ShapePoints

Very similar to the Point class, however does not divide the input latitude and longitude points and is used for storing the shapes of the TAZs

-routing

* Database

Interface used to define the methods for the DefaultDatabase.

* DatabaseFilter

Interface used to define the methods for DefaultDatabaseFilter.

* DefaultDatabase

An implementation of Database consisting of road segments from the TIGER/Line database from the US Census. The TIGER/Line data consists of a separate set of files for each county in the US. In the file of Type 1, each record represents a segment of a one-dimensional feature such as a road, river, or political boundary. It is a road if its 3-character Census Feature Class Code (CFCC) begins with ‘A’.

* DefaultDatabaseFilter

Simple filter for querying the database of RoadSegments for a particular street.

* DefaultRouteStep

Aggregates a number of RoadSegments of the same name together into a single path. This is typically used after the result from Dijkstra is obtained and you then want to see the whole route without looking at individual small segments. Calculates the estimated time and total distance for the given step.

* Dijkstra

Implementation of Dijkstra’s algorithm for finding the shortest path between two points in a graph

* GeoDatabase

Another implementation of a Database which is used to add in shape points for each road from RT2.

* RouteFailureException

Signals when a route cannot be found between the two specified points in the graph.

* RoutePlanner

Primary class for calculating the faster route between two points. This class is the one that initializes the node pool for Dijkstra to use, calls Dijkstra, and aggregates the output into a route. This is the file you’d want to change if you want to alter the speed limits for particular roads.

* RouteStep

Interface which defines the methods for a RouteStep and also implements the turn penalty and rules for turning within the graph. If you want to alter the turn penalty, you’ll have to do it here.

# Common Questions

**When I attempt to run one of the programs, it runs for a while, and then I get an error that it has run out of heap space?**

This error means that you need to allocate more memory to the Java Virtual Machine, so you’ll have to alter the Run Configurations of the program and set the value to something higher, such as “-Xmx1500M” which will allocate 1500 megabytes of RAM to the program.

**In the GUI visualization tool, all of the TAZs are black?**

This means that somewhere along the line the name of the TAZ is getting lost in translation. You’ll want to make sure you are using the proper inputs and if so, add some output statements to make sure that they are hashing to the same values.

**I clicked on the screen and the TAZ went black?**

This means that you probably right-clicked on the zone, and under the current implementation, when a zone is right-clicked, it is extruded from accessibility calculations and the map is re-colored accordingly.

**A pop-up window told me that there was an error with the configuration file?**

If this is at the start of the execution of the program, you may want to double check that the configuration files you are using are correct and that all necessary opening and closing brackets were used in the XML file.

**How do I change the speed limits for each road?**

As noted, you can make this change in RoutePlanner.java which contains the implementation for speed limits.

**How do I alter the turn-based penalties?**

You can access the method which determines the penalties in Routestep.java.

**I’m getting an error during execution of either of the route calculators?**

Chances are the point you are currently using is not close enough to a road or transit stop and if you move it slightly towards the nearest road, that will more than likely solve your issue.

**I get a NumberFormatException for input string: “null” when running GUIMain.**

This can happen if the data files have missing data (e.g., population). DataAppender will run fine, but GUIMain will throw this exception when it finds the empty XML element.