**TMG Bill Toolbox – Calculate Fitness**

**Namespace:** bill\_toolbox.fitness

**Versions:** 1.0, 1.1, 1.2

**Authors:** Yuan Hong (Bill) Sun, Bilal Yusuf

**Date Completed:** July 26th, 2017 (1.2)

**Programming Language:** Python 2.7

1. **Background**
   1. **General Concept**

In path choice modelling, one needs to compare the observed paths taken in real life from a survey to the paths generated by the computer simulation. This is to analyze the similarity of the modelled paths and the observed paths, to ensure the correctness of the modelling.

For a path choice analysis of the Greater Toronto Area (GTA), Emme Modeller [1] was used in conjunction with XTMF [2] to allow for repeated transit assignments using different sets of parameters, and the path choices generated by each assignment is compared to the observed paths gathered by Transportation Tomorrow Survey (TTS) [3] in 2012. Following each assignment, the paths generated by Emme are collected, and compared to the observed paths using this program. XTMF uses a particle swarm optimization program to generate different values for the parameters that optimize the similarity between the two data sets.

The Emme base network is constructed using 2011 data, with zones and centroids defined relative to the population density. All the paths are defined to go from the centroid of one zone to another. Each zone in the GTA is represented by integers ranging from 1 to 9999, depending on the region [6]. Every path is defined by its origin-destination (OD) pair, as well as all the segments of public transit routes taken (all the other path details are omitted for simplicity). A proportion value, ranging from 0.0 to 1.0, represents the probability that a path is taken for a certain origin-destination pair.

* 1. **Data Comparison Methods**

Three methods have been improvised to represent the similarity of the Emme-generated path data verses the TTS observed path data. However, the final method (logarithmic fitness sum) is currently the preferred method of comparison.

1. **Percentage of observed paths that appear in Emme-generated paths**

For each OD pair, every observed path is compared with the list of Emme-generated paths for that OD pair to check if the exact observed path exists. The closer the final value is to 100%, the better.

1. **Average max. percent similarity of all observed paths with their closest-matched Emme path**

For each OD pair, every observed path is compared with the list of Emme-generated paths for that OD pair to check for the closest match by segments of transit routes taken (order is considered).

For example, if the observed path is [‘T001’, ‘T002’, ‘T003’], and the two Emme-generated paths for the same OD pair are [‘T000’, ‘T002’, ‘T003’, ‘T004’] and [‘T001’, ‘T005’, ‘T004’], then the maximum percent similarity is 2/3 as two out of three segments of the observed path appear in the first Emme path in the same order (‘T002’ & ‘T003’). Moreover, if the observed path appears exactly in the list of Emme-generated paths for the same OD pair, then the maximum percent similarity is 1.0.

The sum of all the percentages is divided by the total number of observed paths. The closer the final value is to 100%, the better.

1. **Logarithmic fitness sum:**

For every OD pair, the probability that the observed paths are chosen is computed (exact method will be discussed in section 2 below). A small beta value (β), usually around 0.1, is taken. The fitness sum should be negative. The smaller the magnitude of the final value (i.e. closer to 0), the better.

1. **Overview**
   1. **Program Specifications**

The Calculate Fitness program is programmed using Python 2.7, the default programming language used by Emme 4. It is developed to be used alongside XTMF, by calling it from an “EmmeTool” module. This program cannot be ran separately on Python, or ran directly using Emme.

The program takes exactly five arguments when called from XTMF:

1. Location of Emme to TTS transit line conversion file as a string
2. Location of TTS observed path file as a string
3. Location of Emme-generated path file as a string
4. Location of file that stores the fitness value as a string
5. Beta value (β) as a float
   1. **Structure Overview**

The program is divided into several helper functions that perform different tasks, rather than being put under the main \_m.tool () class structure. These functions differ in Version 1.0, 1.1, and 1.2, and are explained in detail below (Section 3.2). The main class structure, Calculate\_fitness (\_m.Tool ()), calls and executes the functions above from the \_\_call\_\_ () function.

* 1. **Versions**

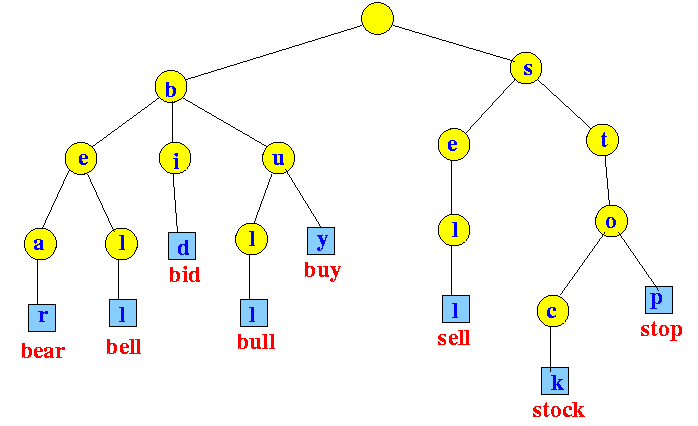
The Calculate Fitness program has three operational versions, to satisfy the needs of the path analysis.

* + 1. **Version 1.0**

This is the original implementation using lists and dictionaries. All three methods of comparing the paths mentioned above (Section 1.2) are implemented.

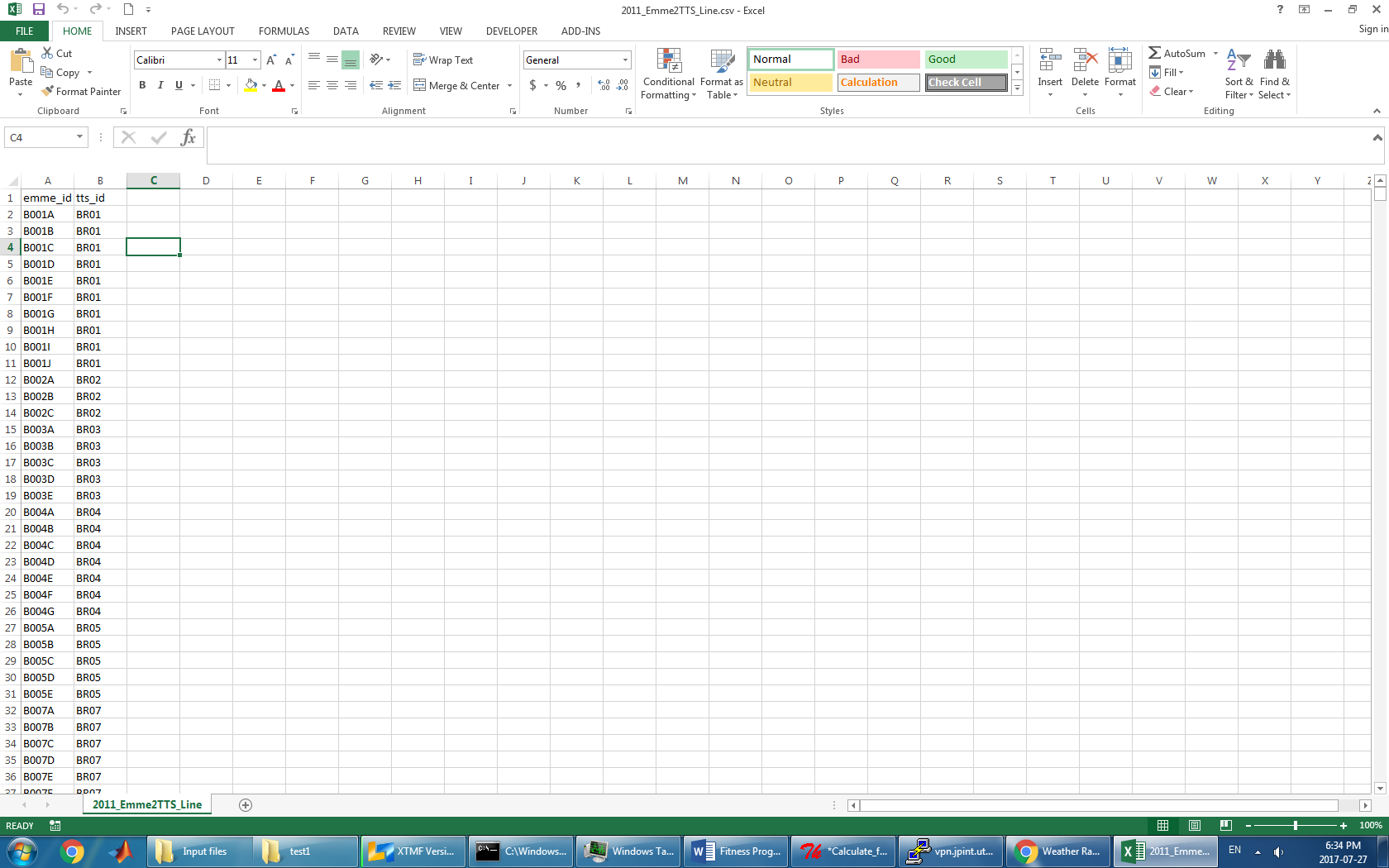
In this version, when the fitness value was computed, the probability of path chosen is defined as the number of times that each observed path appears in the Emme-generated paths over the total number of Emme-generated paths for that specific OD pair.

* + 1. **Version 1.1**

In Version 1.0, all Emme-generated paths are stored as lists. Since many paths are either repeated, or share some of the segments, this method wastes a lot of memory and storage space. To improve the efficiency of storing more Emme-generated paths, a trie data structure (shown on the right), similar to the one used for a word-search algorithm [4] is used. All the other functions from Version 1.0 have been modified to adapt to this structure.

* + 1. **Version 1.2**

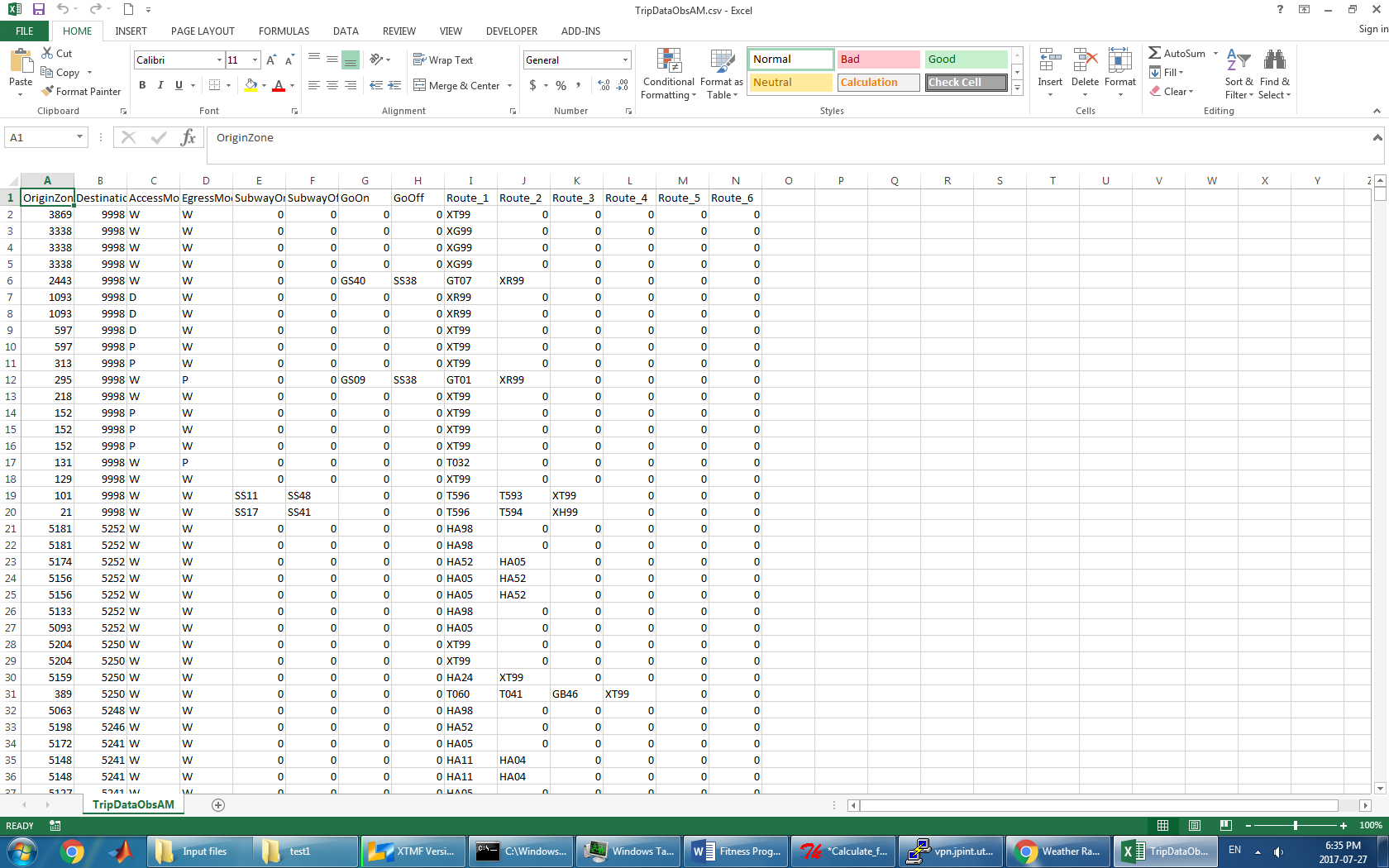
In Versions 1.0 and 1.1, the definition of the probability of path chosen for calculating fitness is not precise because it does not realistically reflect the probability that an observed path is chosen in Emme. Because only the transit segments of each path are stored (see Section 3.1.3), some of the Emme paths stored into the memory are repeated (the raw paths were all unique when they were first generated). As a result, the probability of path chosen has been changed to be the sum of proportions of all the Emme paths that match exactly with the observed path. The original list and dictionary structures are used since tries are not feasible for storing the proportion of each Emme-generated path.

1. **Program Details**
   1. **Input Data Details**
      1. **Emme to TTS Line ID File (.csv):**

For every transit route, the route ID in Emme and TTS are shown. This is needed to convert the transit route IDs in the Emme-generated paths. A sample data sheet is shown on the right.

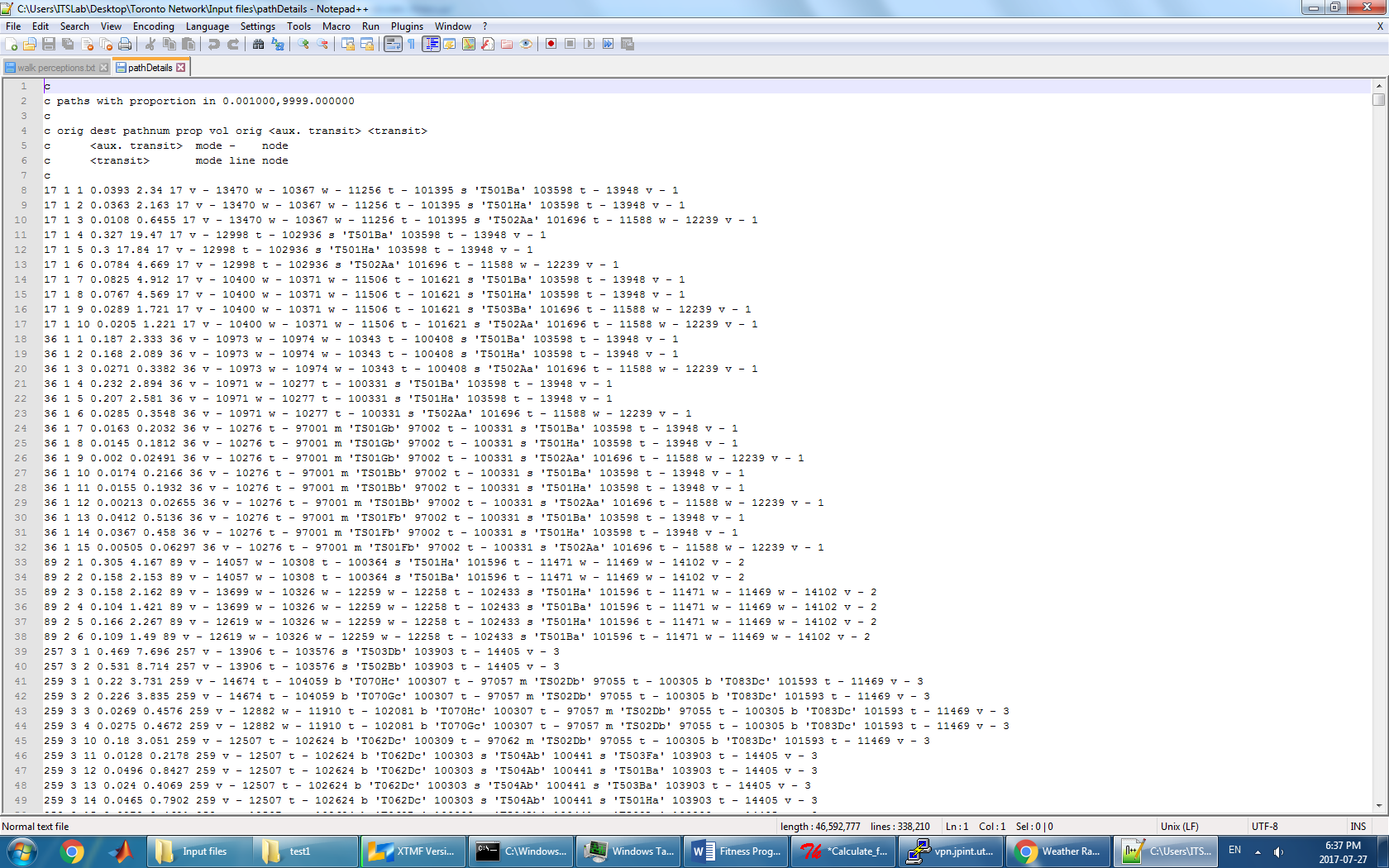
* + 1. **TTS Observed Paths File (.csv):**

The first two columns are the origins and destinations. The next 6 columns (Access mode, Egress mode, Subway on, Subway off, Go on, Go off) are ignored, and then the transit routes (Route\_1 - Route\_6) are stored. A sample data sheet is shown below.



* + 1. **Emme-Generated Paths File (.txt):**

The first two columns are the origins and destinations. The third and fourth columns are proportions and volumes, respectively. Each path segment includes mode, transit line ID (or a dash if the mode [5] of the segment is not transit), and the destination node. Each individual path generated by Emme is unique. However, when reading in the file, only the segments with transit modes are considered, and only the transit routes are stored. This is to cut down the amount of data stored in memory and increase runtime efficiency. A sample data sheet is shown below.



* 1. **Code and Structure Details**

In all three versions, the functions convert\_id () and read\_obs\_paths () read in the Emme ID to TTS ID file and the TTS observed paths file, respectively. The TTS to Emme IDs are stored as a dictionary, with the Emme IDs as the keys and their associated TTS IDs as the values. The TTS observed paths are also stored in a dictionary structure, with the OD pairs as the keys, and the list of all the paths for each OD pair as the values, each path being a sublist. Only the transit routes are stored for each path.

* + 1. **Version 1.0**

In the original implementation, the function read\_EMME\_paths () reads in the Emme-generated paths. They are stored in a dictionary structure, with the OD pairs as the keys, and the list of all the paths for each OD pair as the values, each path being a sublist. Only the transit routes are stored for each path.

The function compare\_paths () computes the percentage of observed paths that appear in the Emme-generated paths. The function compare\_path\_similarity () computes the average maximum percent similarity of all the observed paths with their closest-matched Emme path. The function calculate\_fitness () computes the logarithmic fitness sum.

* + 1. **Version 1.1**

In the trie implementation, two class structures, Node () and Trie () are defined. Each node contains a label, data, and a dictionary of its children. The nodes make up a trie structure. A trie is defined by its head, the topmost node. For each path, the transit routes would be stored in the nodes of a trie. Paths with repeated segments would share the same nodes, which is much more efficient than storing every single segment in a list structure. At the end of each path, another node stores the number of times that a path has appeared. The tries are stored in a dictionary, with the OD pairs as the keys, and its associated paths, in a trie structure, as the values.

Some helper functions are included: Trie.add () adds a path into the trie. Trie.has\_path () checks if the trie contains a specific path. Trie.count\_path () gives the number of times that a specified path has appeared in the trie. Node.count\_total\_paths () recursively counts the total number of paths in a trie.

* + 1. **Version 1.2**

In the latest version, since the fitness function is updated to include the sum of proportions (see section 2.3.3), the proportion values must be stored. However, a trie structure would not be feasible for storing this data since all the repeated paths are not distinguished, and all the paths have a different proportion. Therefore, a list and dictionary structure similar to Version 1.0 was used. For every path, the proportion values are stored as the first item in the list, in front of all the transit routes. All the other functions remain unchanged from Version 1.0.

**Appendix**

1. Emme

<https://www.inrosoftware.com/en/products/emme/>

1. XTMF

<http://www.ecf.utoronto.ca/~miller/TMG-XTMF-Documentation.pdf>

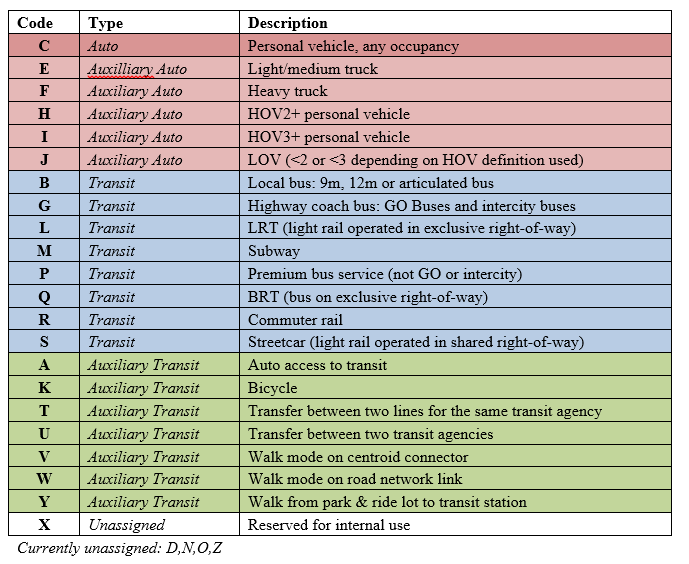
1. TTS

<http://www.transportationtomorrow.on.ca/>

1. Trie and word-search algorithm:

<https://en.wikipedia.org/wiki/Trie>

1. TTS Transit modes:



1. TTS Zone Regions:

|  |  |
| --- | --- |
| **Zones** | **Region** |
| 1-1000 | Toronto |
| 1001-2000 | Durham |
| 2001-3000 | York |
| 3001-4000 | Peel |
| 4001-5000 | Halton |
| 5001-6000 | Hamilton |
| 6001-7000 | Niagara |
| 7001-8000 | Waterloo |
| 8001-9000 | Brantford, Brant, Guelph, Wellington, Orangeville, Dufferin, Orillia, Simcoe, Barrie, Kawartha Lakes, Peterborough County, City of Peterborough |

Source: <http://www.dmg.utoronto.ca/pdf/tts/2011/dataguide2011.pdf>