The Path Most Travelled : Travel Demand Estimation Using Big Data Resources

Jameson L. Toole[a],[**], Serdar Colak[b],[**], Bradley Sturt[a], Lauren P. Alexander[b], Alexandre Evsukoff[c], Marta C. González[a,b]

[a]*Engineering Systems Division, MIT, Cambridge, MA, 02139*
[b]*Department of Civil and Environmental Engineering, MIT, Cambridge, MA, 02139*
[c]*COPPE/Federal University of Rio de Janeiro, Brazil*

Keywords: mobility, location based services, congestion, road networks, mobile phone data

---

1. Algorithms

[*]corresponding author: serdarc@mit.edu
[**]These authors contributed equally to this work.

ALGORITHM 1: Parsing OpenStreetMap Networks

1: {OSM files are XML based and contain way and node objects}

2: $ways$ = set of ways in an OSM file

3: $nodes$ = set of nodes in an OSM file

4: $graph$ = an empty graph

5: ————————————————————————————

6: {Add each pair of consecutive nodes to the edge list}

7: **for** way **in** $ways$ **do**

8:     **for** $i = 0$ **to** $i = way.nodes.size() - 2$ **do**

9:         $graph.addNode(way.nodes[i])$

10:         $graph.addNode(way.nodes[i + 1])$

11:         $graph.addEdge(way.nodes[i], way.nodes[i + 1])$

12: ————————————————————————————

13: {Simplify the network by merging road segments }

14: **for** way **in** $ways$ **do**

15:     $startNode = way.nodes[0]$

16:     **for** $node$ **in** $way.nodes$ **do**

17:         **if** all edges into and out of $node$ are segments of the same way **then**

18:             $graph.removeNode$(node)

19:             remove all edges to or from $node$

20:         **else**

21:             $endNode = node$

22:             $graph.addEdge(startNode, endNode)$

23:             $FillEdgeAttributes()$

24:             $startNode = endNode$

25: ————————————————————————————

26: {Notes}

27: $^{*}FillEdgeAttributes()$ fills in missing data such as speed limits or number of lanes based on way attributes

28: $^{*}graph.addNode(node)$ and $graph.addEdge(node1, node2)$ only add objects if they do not already exist

2

29: $^{*}graph.removeNode(node)$ also removes all edges containing that node

30: *when simplifying the network, proper geographic lengths are kept even when nodes are deleted

---

ALGORITHM 2: Stay Point Algorithm - Step 1 - Initialize

1: {Each user object has a number of attributes}

2: $call$ = a call object with an associated latitude, longitude, stay index

3: $calls$ = vector of a user's calls ordered by timestamp

4: $candidateSet$ = empty set of consecutive calls that meet criteria for a stay

5: $candidateStays$ = a vector of centroids from candidate sets

6: δ = distance threshold between consecutive calls (in meters)

7: τ = time threshold between entry into and exit from the stay (in seconds)

8: ds = a grid size for the agglomerative clustering algorithm (in meters)

9: $stayCalls$ = an empty vector of calls from stay points

10: {Notes}

11: $^{*}Centroid(callSet)$ returns an object whose latitude and longitude are the centroid of all points in the input

12: $^{*}DistanceBetweenCalls(call1, call2)$ returns the geographic distance between calls in meters

13: $^{*}TimeBetweenCalls(call1, call2)$ returns the time between call in seconds

ALGORITHM 3: Stay Point Algorithm - Step 2 - Candidate Stays

1: {For each user, loop through all calls and find candidate stays}

2: $candidateIndex = 0$

3: $candidateSet = \{\}$

4: **for** $i = 0$ **to** $i = calls.size() - 2$ **do**

5:     **if** $DistanceBetweenCalls(calls[i], calls[i+1]) < \delta$ **then**

6:         $candidateSet.append(calls[i+1])$

7:     **else**

8:         **if** $TimeBetweenCalls(candidateSet[0], candidateSet[end]) > \tau$ **then**

9:             **for** $call$ **in** $candidateSet$ **do**

10:                 $call.stayIndex = candidateIndex$

11:             $candidateStay = Centroid(candidateSet)$

12:             $candidateStays.append(candidateStay)$

13:         $candidateSet = \{calls[i]\}$

14:         $candidateIndex = candidateIndex + 1$

ALGORITHM 4: Stay Point Algorithm - Step 3 - Agglomerative Clustering

---

1: $grid$ = construct a uniform grid that covers all of a user's calls with cell dimensions $ds \times ds$

2: $stayIndex = 0$

3: **for** grid cells containing a $candidateStay$ **do**

4:     $candidateStays = \{list\,of\,\text{candidateStay}\,in\,cell\}$

5:     $stay = Centroid(candidateStays)$

6:     **for** $call$ made from a $candidateStay$ in this cell **do**

7:         $call.longitude = stay.longitude$

8:         $call.latitude = stay.latitude$

9:         $call.stayIndex = stayIndex$

10:         $stayCalls.append(call)$

11:     $stayIndex = stayIndex + 1$

ALGORITHM 5: Stay Point Algorithm - Step 4 - Final Pass

1: {Final pass to add any remaining calls to the stay}

2: **for** $i = 0$ **to** $i = calls.size()$ **do**

3:     **if** $call$ not part of a stay and DistanceBetweenCalls($call$, $stay$) ¡ δ for any $stay$ **then**

4:         $call.longitude = stay.longitude$

5:         $call.latitude = stay.latitude$

6:         $call.stayIndex = stayIndex$

7:         $stayCalls.append(call)$

8: Sort $stayCalls$ by timestamp

9: ———————————————————

5

ALGORITHM 6: OD Creation Algorithm - Step 1 - Home / Work Expansion

---

1: {Data objects}

2: $tracts$ = census tract data objects containing demographic variables

3: $OD(o, d, p, t) = 0$ for origin o, destination d, purpose p, and period t

4: _____

5: {Detect home and work for all users and compute expansion factors}

6: **for** $user$ **in** $users$ **do**

7:     $user.stays$ = vector of calls at stay points sorted by time

8:     $user.home$ = index of stay point visited the most between 8pm and 7am on weekdays

9:     $user.work$ = index of non-home stay point visited the most between 7am and 8pm on weekdays

10:     **if** user visits work less than once per week **then**

11:         $user.work$ = null

12:     **for** $stay$ **in** $user.stays$ **do**

13:         $stay.label$ assigned as $home$, $work$, or $other$

14:     $user.weekdays$ = number of weekdays a user records a stay

15:     $user.workdays$ = number of weekdays a user records a stay at work

16:     $tract[user.home].numUsers = tract[user.home].numUsers + 1$

17: **for** $tract$ **in** $tracts$ **do**

18:     $tract.expansionFactor = tract.population/tract.numUsers$

ALGORITHM 7: OD Creation Algorithm - Step 2 - Trip Counting

1: {Count and expand trips}

2: **for** $user$ **in** $users$ **do**

3:     $trips =$ empty vector to store trips taken by a user

4:     **for** $i = 1$ **to** $i = user.stays.size()$ **do**

5:         $s0 = user.stays[i-1]$

6:         $s1 = user.stays[i]$

7:         **if** $s0 == S1$ **then**

8:             continue

9:         **if** $s0$ and $s1$ are on the same effective day **then**

10:             $trip =$ new trip from $s0$ to $s1$

11:             $trip.purpose = PurposeFromLabels(s0, s1)$

12:             $trip.workday =$ true if workday for user, false otherwise

13:             $trip.departure = GetConditionalDepartureTime(s0, s1)$

14:             $trips.append(trip)$

15:         **else** $s0$ and $s1$ are not on the same effective day

16:             $morning =$ create trip from home to first recorded stay

17:             $night =$ create trip from last recorded stay to home

18:             $trips.append(morning)$

19:             $trips.append(night)$

20:   **for** *trip* **in** *trips* **do**

21:       $o = trip.origin$

22:       $d = trip.destination$

23:       $p = trip.purpose$

24:       $t = trip.departure$

25:       **if** $trip.workday ==$ true **then**

26:           $flow = tract[user.home].expansionFactor/user.workdays$

27:       **else**

28:           $flow = tract[user.home].expansionFactor/user.weekdays$

29:       $OD(o, d, p, t) = OD(o, d, p, t) + flow$

30: ──

31: {Notes}

32: $^*PurposeFromLabels(s0, s1)$ returns a trip purpose (HBW, NHB, HBO) based on the label of origin and destination stays

33: $^*GetConditionalDepartureTime(s0, s1)$ returns a departure time based on the observation times at origin and destination

34: *an effective day is defined as a period between 3am today until 3am on the next consecutive morning

8

ALGORITHM 8: Incremental Traffic Assignment

$graph$ = road network

$OD(p, t)$ = origin-destination matrix for purpose p and time window t

B = a bipartite network containing roads and census tracts

$incrSize$ = vector of increment sizes, e.g. [0.4, 0.3, 0.2, 0.1]

$nBatches$ = number of threads to use

for $i = 0$ **to** $i < incrSize.size()$ **do**

    for $b = 0$ **to** $b < nBatches$ **do**

        create new thread

        $batch = GetBatch(OD, b)$

        for all o, d pairs in $batch$ **do**

            $flow = OD[o, d].flow \cdot incrSize[i]$

            $route = A^*(o, d, graph)$

            for all segment s in $route$ **do**

                $s.flow = s.flow + flow$

                $B_{e \to o} = B_{s \to o} + flow$

    wait for all threads to finish

    for segment s in $graph$ **do**

        $s.cost \leftarrow s.freeFlowTime \cdot (1 + \alpha(\frac{s.volume}{s.capacity})^\beta)$

---

* GetBatch(OD, B) returns only the subset of OD pairs pertaining to a batch

* $A^*(o, d, graph)$ returns the shortest path between o and d if a path exists