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

Jameson L. Toole\textsuperscript{a,\textdagger}, Serdar Colak\textsuperscript{b,\textdagger}, Bradley Sturt\textsuperscript{a}, Lauren P. Alexander\textsuperscript{b}, Alexandre Evsukoff\textsuperscript{c}, Marta C. González\textsuperscript{a,b}

\textsuperscript{a}Engineering Systems Division, MIT, Cambridge, MA, 02139
\textsuperscript{b}Department of Civil and Environmental Engineering, MIT, Cambridge, MA, 02139
\textsuperscript{c}COPPE/Federal University of Rio de Janeiro, Brazil

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

1. Algorithms
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

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])

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

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
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: $\delta$ = distance threshold between consecutive calls (in meters)
7: $\tau$ = 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 $i = \text{calls.size}() - 2$ do
5:     if $\text{DistanceBetweenCalls(calls}[i],\text{calls}[i + 1]) < \delta$ then
6:         candidateSet.append(calls[i + 1])
7:     else
8:         if $\text{TimeBetweenCalls(candidateSet[0], candidateSet[end])} > \tau$ 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 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:   end for
12:   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 $i = \text{calls.size()}$ do
3:   if call not part of a stay and DistanceBetweenCalls(call, stay) $\leq \delta$ for any stay then
4:     call.longitude = stay.longitude
5:     call.latitude = stay.latitude
6:     call.stayIndex = stayIndex
7:     stayCalls.append(call)
8:   end if
9: end for
10: Sort stayCalls by timestamp
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: {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

6
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)
for trip in trips do
  o = trip.origin
  d = trip.destination
  p = trip.purpose
  t = trip.departure
  if trip.workday == true then
    flow = tract[user.home].expansionFactor/user.workdays
  else
    flow = tract[user.home].expansionFactor/user.weekdays
  OD(o, d, p, t) = OD(o, d, p, t) + flow

{Notes}

*PurposeFromLabels(s0, s1) returns a trip purpose (HBW, NHB, HBO) based on the label of origin and destination stays
*GetConditionalDepartureTime(s0, s1) returns a departure time based on the observation times at origin and destination
an effective day is defined as a period between 3am today until 3am on the next consecutive morning
ALGORITHM 8: Incremental Traffic Assignment

\(\text{graph} = \text{road network}\)
\(\text{OD}(p, t) = \text{origin-destination matrix for purpose } p \text{ and time window } t\)
\(B = \text{a bipartite network containing roads and census tracts}\)
\(\text{incrSize} = \text{vector of increment sizes, e.g. } [0.4, 0.3, 0.2, 0.1]\)
\(nBatches = \text{number of threads to use}\)

\[\text{for } i = 0 \text{ to } i < \text{incrSize.size()} \text{ do}\]
  \[\text{for } b = 0 \text{ to } b < nBatches \text{ do}\]
    \[\text{create new thread}\]
    \[\text{batch} = \text{GetBatch}(\text{OD}, b)\]
    \[\text{for all } o, d \text{ pairs in } \text{batch} \text{ do}\]
      \[\text{flow} = \text{OD}[o, d].\text{flow} \cdot \text{incrSize}[i]\]
      \[\text{route} = A^*(o, d, \text{graph})\]
      \[\text{for all segment } s \text{ in } \text{route} \text{ do}\]
        \[s.\text{flow} = s.\text{flow} + \text{flow}\]
        \[B_{s\rightarrow o} = B_{s\rightarrow o} + \text{flow}\]
    \[\text{wait for all threads to finish}\]
  \[\text{for segment } s \text{ in } \text{graph} \text{ do}\]
    \[s.\text{cost} \leftarrow s.\text{freeFlowTime} \cdot (1 + \alpha \left(\frac{s.\text{volume}}{s.\text{capacity}}\right)\beta)\]

* \(\text{GetBatch(OD, B)}\) returns only the subset of \(\text{OD}\) pairs pertaining to a batch
* \(A^*(o, d, \text{graph})\) returns the shortest path between \(o\) and \(d\) if a path exists