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

```
{OSM files are XML based and contain way and node objects}
ways = set of ways in an OSM file
nodes = set of nodes in an OSM file
graph = an empty graph

{Add each pair of consecutive nodes to the edge list}
for way in ways do
    for i = 0 to i = way.nodes.size() − 2 do
        graph.addNode(way.nodes[i])
        graph.addNode(way.nodes[i + 1])
        graph.addEdge(way.nodes[i], way.nodes[i + 1])

{Simplify the network by merging road segments }
for way in ways do
    startNode = way.nodes[0]
    for node in way.nodes do
        if all edges into and out of node are segments of the same way then
            graph.removeNode(node)
            remove all edges to or from node
        else
            endNode = node
            graph.addEdge(startNode, endNode)
            FillEdgeAttributes()
            startNode = endNode

{Notes}
*FillEdgeAttributes() fills in missing data such as speed limits or number of lanes based on way attributes
*graph.addNode(node) and graph.addEdge(node1, node2) only add objects if they do not already exist
*graph.removeNode(node) also removes all edges containing that node
*when simplifying the network, proper geographic lengths are kept even when nodes are deleted
```

ALGORITHM 2: Stay Point Algorithm - Step 1 - Initialize

{Each user object has a number of attributes}
$call$ = a call object with an associated latitude, longitude, stay index
$calls$ = vector of a user's calls ordered by timestamp
$candidateSet$ = empty set of consecutive calls that meet criteria for a stay
$candidateStays$ = a vector of centroids from candidate sets
δ = distance threshold between consecutive calls (in meters)
τ = time threshold between entry into and exit from the stay (in seconds)
ds = a grid size for the agglomerative clustering algorithm (in meters)
$stayCalls$ = an empty vector of calls from stay points
{Notes}
$^*Centroid(callSet)$ returns an object whose latitude and longitude are the centroid of all points in the input
$^*DistanceBetweenCalls(call1, call2)$ returns the geographic distance between calls in meters
$^*TimeBetweenCalls(call1, call2)$ returns the time between call in seconds

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

```
{For each user, loop through all calls and find candidate stays}
candidateIndex = 0
candidateSet = {}
for i = 0 to i = calls.size() − 2 do
    if DistanceBetweenCalls(calls[i], calls[i + 1]) < δ then
        candidateSet.append(calls[i + 1])
    else
        if TimeBetweenCalls(candidateSet[0], candidateSet[end]) > τ then
            for call in candidateSet do
                call.stayIndex = candidateIndex
            candidateStay = Centroid(candidateSet)
            candidateStays.append(candidateStay)
        candidateSet = {calls[i]}
        candidateIndex = candidateIndex + 1
```

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

```
grid = construct a uniform grid that covers all of a user's calls with cell dimensions ds × ds
stayIndex = 0
for grid cells containing a candidateStay do
    candidateStays = {listof candidateStay incell}
    stay = Centroid(candidateStays)
    for call made from a candidateStay in this cell do
        call.longitude = stay.longitude
        call.latitude = stay.latitude
        call.stayIndex = stayIndex
        stayCalls.append(call)
    stayIndex = stayIndex + 1
```

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

```
{Final pass to add any remaining calls to the stay}
for i = 0 to i = calls.size() do
    if call not part of a stay and DistanceBetweenCalls(call, stay) ¡ δ for any stay then
        call.longitude = stay.longitude
        call.latitude = stay.latitude
        call.stayIndex = stayIndex
        stayCalls.append(call)
Sort stayCalls by timestamp
```

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

{Data objects}
$tracts$ = census tract data objects containing demographic variables
$OD(o, d, p, t) = 0$ for origin o, destination d, purpose p, and period t

---

{Detect home and work for all users and compute expansion factors}
for $user$ **in** $users$ **do**
  $user.stays$ = vector of calls at stay points sorted by time
  $user.home$ = index of stay point visited the most between 8pm and 7am on weekdays
  $user.work$ = index of non-home stay point visited the most between 7am and 8pm on weekdays
  if user visits work less than once per week **then**
    $user.work$ = null
  for $stay$ **in** $user.stays$ **do**
    $stay.label$ assigned as $home$, $work$, or $other$
  $user.weekdays$ = number of weekdays a user records a stay
  $user.workdays$ = number of weekdays a user records a stay at work
  $tract[user.home].numUsers = tract[user.home].numUsers + 1$
for $tract$ **in** $tracts$ **do**
  $tract.expansionFactor = tract.population / tract.numUsers$

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

```
{Count and expand trips}
for user in users do
    trips = empty vector to store trips taken by a user
    for i = 1 to i = user.stays.size() do
        s0 = user.stays[i − 1]
        s1 = user.stays[i]
        if s0 == S1 then
            continue
        if s0 and s1 are on the same effective day then
            trip = new trip from s0 to s1
            trip.purpose = PurposeFromLabels(s0, s1)
            trip.workday = true if workday for user, false otherwise
            trip.departure = GetConditionalDepartureTime(s0, s1)
            trips.append(trip)
        elses0 and s1 are not on the same effective day
            morning = create trip from home to first recorded stay
            night = create trip from last recorded stay to home
            trips.append(morning)
            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

$graph$ = 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