Geo-Spatial API - Algorithms

Vocabulary:

- **Edge** - a path (set of road segments) having vertices at the starting and ending points.
- **Graph** – a pair of sets V (vertices) and E (edges) – denoted as **G(V,E)**
- **Household** - a geographic location (point) having coordinates and adjacent **paths** /road segments and is a private case of **Vertex**.
- **Path** - the set of mutually adjacent road segments having an Vertex (a household or a site) at the starting point of the first road segment and the ending point of the last segment. **Path** forms an **Edge**.
- **Road Segment** - a road cluster that may have adjacent vertices (households or sites) or other **road segments**
- **Service Area** – a geographic polygonal area identified by shape coordinates and containing the **road segments**, **households**, and **sites** that are covered by service.
- **Site** – a geographic location (point) having adjacent **paths**/road segments and is a private case of **Vertex**
- **Threshold** – maximum distance between the origin **site** and the destination **households**.
- **Vertex** - a **household** or a **site** having adjacent **edges** (**paths**)
- **Abbreviations used**: **hh – household, sg – road segment, st – site, sa – service area, adj - adjacent**

Initial Graph Conversion Algorithm Outline

Input:

- Set of **Road Segments** **S** *sg* within the **service area S** *sa*
- Set of **Households** – **S** *hh* within the **service area S** *sa*
- Set of **Sites** - **S** *st* within the **service area S** *sa*
- **Service Area - S** *sa* – the search area that may fully or not contain the **households** within the **threshold distance** from the specified **site**.
- Hash Table of paths adjacent to a vertex **- H***adj* . Initially empty, we need this to keep the progress track.
- **V** – initially empty list of vertices – we use this for recursion
- **E** – initially empty list of edges – we use this for recursion
- NOTE: The data contained in **S** *sa,* **S** *sg,* **S** *hh,* **S** *st* can be populated using both Django ORM querying or a RESTful API call (we suggest doing it by calling the Postgres as we work on the server and have access to Django ORM).

Output:

- **Graph G(V,E)** where **V** is set of **vertices** (**households**, **sites**) and
  E is a set of **edges** (**road segment** chains leading from one **vertex** to another

Graph Conversion Algorithm Overview:

- Go through the set of road segments **S** *sg* and link adjacent road segments to each other until you discover a **household** or a **site** at the end of a **road segment**. Once you have a linked set of **road segments** starting and ending with either **household** or a **site** vertex, we push this linked set as an **edge** into **E**.

- For each Household in **S** *hh*
  1. Create a vertex **v,** discover adjacent **edges** (**road segment** chains or **paths** leading from **vertex v** to another), set **v**.*type* to **household,** set **v**.*degree* to the amount of adjacent **edges** from **E,** set **v**.*distance =* infinity.

2. Push **v** into **V**.

- For each Site in **S** *st*
  1. Create a vertex **v,** discover adjacent **edges** (**road segment** chains or **paths** leading from one **vertex** to another), set **v**.*type* to **site** or **household** ,set **v**.*degree* to the amount of adjacent **edges** from **E,** set **v**.*distance* = 0.
  2. Push **v** into **V**.

- Return the compound **Graph G(V,E)**

PSEUDOCODE FOR GRAPH GENERATION:

```
GET_ADJACENT(v , S sg)            # Helper function to get the segments adjacent
    ADJACENT = []                   # to a vertex
    FOR EACH s IN S sg :
        IF v.coordinates = s.coordinates
        ADJACENT.push(s)
    RETURN ADJACENT


DESTINATION_VERTEX (v, L, V)  #  Helper function to discover vertices on both ends
      FOR EACH vt IN V:
          IF vt.coordinates ! = v.coordinates AND IS_REACHABLE(vt, L) : # traverse the list from hash
               RETURN vt
      RETURN NIL


IS_REACHABLE(v, L): # Discover if the destination vertex is reachable by traversing the segments
    FOR EACH sg in L:
        IF sg.coordinates == v.coordinates
              RETURN TRUE
        END IF
    END FOR
RETURN FALSE
LINK_TO_PATH(sg, L) :  # Helper function trying to append segment to path if there's a match
      FOR EACH s in L:
            IF s.node_to_id == sg.node_from_id OR sg.node_from_id==s.node_to_id:
                  L.append(sg)
```

```
                    L.total_distance= L.total_distance +sg.length
 RETURN L
CONVERT_TO_GRAPH (S sa, S sg, S hh, S st, Hadj, V, E):  #This is the main graph conversion function


    IF NOT S sg, S hh, S st   in   S sa
        return NIL


    IF V==NIL:              # Initial call -  no vertices
      FOR EACH hh IN S hh :      # we first convert households to vertices and  move all vertices into V
          v = new Vertex
          v.type = household.
          v.degree  =  0
          v.distance =  0
          v.coordinates = hh.coordinates
          V.push (v)

     FOR EACH st IN S st :       # we then convert sites to vertices and move all vertices into V
         v = new Vertex
         v.type = site.
         v.degree  =  0
         v.coordinates = hh.coordinates
         v.distance =  0
         V.push (v)
    END IF



    FOR EACH  v IN  V :                  # Lets go through each vertex in V
        Sadj = Hadj[ [v]                  #  Lets check if the hash has been already initiated for specific vertex
        IF Sadj == NIL                   # Initial call - no values
            Sadj= GET_ADJACENT (v , S st)                      # get the adjacent segments
        FOR EACH sg IN Sadj   :                        # let's go and check for neighbor vertice
             v =   DESTINATION_VERTEX (v, sg, Sadj , V)   # neighbor vertices
             IF v != NIL          # if some exist  - let's assign an edge and push into the hash
                 Hadj[ [v] = LINK_TO_PATH(sg, Hadj[ [v] ) # we add the segment to the list in the HASH
```

```
            e = new Edge(Hadj[

            e.weight = Hadj[ [v].total_distance

          E.push(e)      # one more edge added


        ELSE     # no neighbor vertices exist - let's chain the segment to the v's list in HASH

            Hadj[ [v] = LINK_TO_PATH(sg, Hadj[ [v] ) # we add the segment to the list in the HASH


            CONVERT_TO_GRAPH (S sa, S sg, S hh, S st, Hadj, E) # recursively call with new H

      END FOR  # no segments vertices - exist

    END FOR # no more vertices - exit

    RETURN G(V,E)  # all vertices explored - return graph
```

Housholds Computation Algorithm Outline

Input:

- User input – Site-Threshold pairs (S,T)
- Graph **G(V,E)** constructed in the previous step.
- **Service Area - S** *sa* – the search area

Output:

- **Number of households for the specified sites-threshold pairs**

Number of Households Computation Algorithm Overview:

- For each pair (s,t) in (S,T) identify if there's a site with a smaller distance - Apply Kruskal algorithm to identify the minimal weight spanning tree for all the sites and households. Assign the result to the site with lowest value.
- Initialize a Hash Table **H**v for vertices storing id and checked Boolean values to prevent duplications
- Traverse the Graph with **A*** (A-star) algorithm using the site, identified in previous step, as an origin. Proceed with calculating distances for vertices by storing key-value pairs in a Hash Table **H**. Stop traversing where the threshold becomes smaller than total path of the route.
- Through traversing the route add 1 to the count if the unique vertex id has not been checked yet – have no 1 in **H**v[v].
- Store the counted amount in the Hash Table for the specific inquired (S,T) pair.
- Render the results as JSON

PSEUDOCODE FOR KRUSKAL's MINIMAL WEIGHT SPANNING TREE ALGORITHM:

```
KRUSKAL(G):
  A = ∅
  foreach v ∈ G.V:
  MAKE-SET(v)
  foreach (u, v) ordered by weight(u, v), increasing:
     if FIND-SET(u) ≠ FIND-SET(v):
        A = A ∪ {(u, v)}
        UNION(u, v)
  return A
```

PSEUDOCODE FOR A* TRAVERSAL ALGORITHM:

```
initialize the open list
initialize the closed list
assign list a unique id
assign H(list.id).counter = 0

put the starting node on the open list (you can leave its f at zero)

while the open list is not empty
    find the node with the least f on the open list, call it "q"
    pop q off the open list
    generate q's 8 successors and set their parents to q
    for each successor
        if successor is the goal, stop the search
        successor.g = q.g + distance between successor and q
        successor.h = distance from goal to successor
        if successor.h > threshold terminate
        successor.f = successor.g + successor.h
        if (successor.type == household) H(list.id).counter += 1

        if a node with the same position as successor is in the OPEN list \
            which has a lower f than successor, skip this successor
        if a node with the same position as successor is in the CLOSED list \
            which has a lower f than successor, skip this successor
        otherwise, add the node to the open list
    end
    push q on the closed list
end
```