import sys

import cairo

import math

from xml.sax import make\_parser, handler

from loadOsm import \*

class GetRoutes(handler.ContentHandler):

"""Parse an OSM file looking for routing information, and do routing with it"""

def \_\_init\_\_(self):

"""Initialise an OSM-file parser"""

self.routing = {}

self.nodes = {}

self.minLon = 180

self.minLat = 90

self.maxLon = -180

self.maxLat = -90

def startElement(self, name, attrs):

"""Handle XML elements"""

if name in('node','way','relation'):

if name == 'node':

"""Nodes need to be stored"""

id = int(attrs.get('id'))

lat = float(attrs.get('lat'))

lon = float(attrs.get('lon'))

self.nodes[id] = (lat,lon)

if lon < self.minLon:

self.minLon = lon

if lat < self.minLat:

self.minLat = lat

if lon > self.maxLon:

self.maxLon = lon

if lat > self.maxLat:

self.maxLat = lat

self.tags = {}

self.waynodes = []

elif name == 'nd':

"""Nodes within a way -- add them to a list"""

self.waynodes.append(int(attrs.get('ref')))

elif name == 'tag':

"""Tags - store them in a hash"""

k,v = (attrs.get('k'), attrs.get('v'))

if not k in ('created\_by'):

self.tags[k] = v

def endElement(self, name):

"""Handle ways in the OSM data"""

if name == 'way':

last = -1

highway = self.tags.get('highway', '')

railway = self.tags.get('railway', '')

oneway = self.tags.get('oneway', '')

reversible = not oneway in('yes','true','1')

cyclable = highway in ('primary','secondary','tertiary','unclassified','minor','cycleway','residential', 'service')

if cyclable:

for i in self.waynodes:

if last != -1:

#print "%d -> %d & v.v." % (last, i)

self.addLink(last, i)

if reversible:

self.addLink(i, last)

last = i

def addLink(self,fr,to):

"""Add a routeable edge to the scenario"""

# Look for existing

try:

if to in self.routing[fr]:

#print "duplicate %d from %d" % (to,fr)

return

# Try to add to list. If list doesn't exist, create it

self.routing[fr].append(to)

except KeyError:

self.routing[fr] = [to]

def initProj(self,w,h, lat,lon, scale=1):

"""Setup an image coordinate system"""

self.w = w

self.h = h

self.clat = lat

self.clon = lon

self.dlat = (self.maxLat - self.minLat) / scale

self.dlon = (self.maxLon - self.minLon) / scale

def project(self, lat, lon):

"""Convert from lat/long to image coordinates"""

x = self.w \* (0.5 + 0.5 \* (lon - self.clon) / (0.5 \* self.dlon))

y = self.h \* (0.5 - 0.5 \* (lat - self.clat) / (0.5 \* self.dlat))

return(x,y)

def markNode(self,node,r,g,b):

"""Mark a node on the map"""

self.ctx.set\_source\_rgb(r,g,b)

lat = self.nodes[node][0]

lon = self.nodes[node][1]

x,y = self.project(lat,lon)

self.ctx.arc(x,y,2, 0,2\*3.14)

self.ctx.fill()

def markLine(self,n1,n2,r,g,b):

"""Draw a line on the map between two nodes"""

self.ctx.set\_source\_rgba(r,g,b,0.3)

lat = self.nodes[n1][0]

lon = self.nodes[n1][1]

x,y = self.project(lat,lon)

self.ctx.move\_to(x,y)

lat = self.nodes[n2][0]

lon = self.nodes[n2][1]

x,y = self.project(lat,lon)

self.ctx.line\_to(x,y)

self.ctx.stroke()

def distance(self,n1,n2):

"""Calculate distance between two nodes"""

lat1 = self.nodes[n1][0]

lon1 = self.nodes[n1][1]

lat2 = self.nodes[n2][0]

lon2 = self.nodes[n2][1]

# TODO: projection issues

dlat = lat2 - lat1

dlon = lon2 - lon1

dist2 = dlat \* dlat + dlon \* dlon

return(math.sqrt(dist2))

def doRouting(self, routeFrom, routeTo):

"""Wrapper around the routing function, which creates the output image, etc"""

size = 800

scalemap = 5 # the bigger this is, the more the map zooms-in

# Centre the map halfway between start and finish

ctrLat = (self.nodes[routeFrom][0] + self.nodes[routeTo][0]) / 2

ctrLon = (self.nodes[routeFrom][1] + self.nodes[routeTo][1]) / 2

self.initProj(size, size, ctrLat, ctrLon, scalemap)

surface = cairo.ImageSurface(cairo.FORMAT\_RGB24, self.w, self.h)

self.ctx = cairo.Context(surface)

# Dump all the nodes onto the map, to give the routes some context

self.ctx.set\_source\_rgb(1.0, 0.0, 0.0)

self.ctx.set\_line\_cap(cairo.LINE\_CAP\_ROUND)

for id,n in self.nodes.items():

x,y = self.project(n[0], n[1])

self.ctx.move\_to(x,y)

self.ctx.line\_to(x,y)

self.ctx.stroke()

# Do the routing itself

self.doRoute(routeFrom, routeTo)

# Highlight which nodes were the start and end

self.markNode(routeFrom,1,1,1)

self.markNode(routeTo,1,1,0)

# Image is complete

surface.write\_to\_png("output.png")

def doRoute(self,start,end):

"""Do the routing"""

self.searchEnd = end

closed = [start]

self.queue = []

# Start by queueing all outbound links from the start node

blankQueueItem = {'end':-1,'distance':0,'nodes':str(start)}

for i in self.routing[start]:

self.addToQueue(start,i, blankQueueItem)

# Limit for how long it will search (also useful for debugging step-by-step)

maxSteps = 10000

while maxSteps > 0:

maxSteps = maxSteps - 1

try:

nextItem = self.queue.pop(0)

except IndexError:

print ("Failed to find any route")

return

x = nextItem['end']

if x in closed:

continue

self.markNode(x,0,0,1)

if x == end:

print ("Success!")

self.printRoute(nextItem)

return

closed.append(x)

try:

for i in self.routing[x]:

if not i in closed:

self.addToQueue(x,i,nextItem)

except KeyError:

pass

else:

self.debugQueue()

def debugQueue(self):

"""Display some information about the state of our queue"""

print ("Queue now %d items long" % len(self.queue))

# Display on map

for i in self.queue:

self.markNode(i['end'],0,0.5,0)

def printRoute(self,item):

"""Output stage, for printing the route once found"""

# Route is stored as text initially. Split into a list

print ("Route: %s" % item['nodes'])

listNodes = [int(i) for i in item['nodes'].split(",")]

#Route having lat and long saved in text file

file = open("Route.txt","w")

for i in listNodes:

file.write("%f , %f\n"% (

self.nodes[i][0],

self.nodes[i][1]))

# Display the route on the map

last = -1

for i in listNodes:

if last != -1:

self.markLine(last,i,0.5,1.0,0.5)

self.markNode(i,0.5,1.0,0.5)

last = i

# Send the route to an OSM file

fout = open("route.osm", "w")

fout.write("<?xml version='1.0' encoding='UTF-8'?>");

fout.write("<osm version='0.5' generator='route.py'>");

for i in listNodes:

fout.write("<node id='%d' lat='%f' lon='%f'>\n</node>\n" % ( \

i,

self.nodes[i][0],

self.nodes[i][1]))

fout.write("<way id='1'>\n")

for i in listNodes:

fout.write("<nd ref='%d' lat='%f' lon='%f' />\n" % ( \

i,

self.nodes[i][0],

self.nodes[i][1]))

fout.write("</way>\n")

fout.write("</osm>")

fout.close()

def addToQueue(self,start,end, queueSoFar):

"""Add another potential route to the queue"""

# If already in queue

for test in self.queue:

if test['end'] == end:

return

distance = self.distance(start, end)

# Create a hash for all the route's attributes

queueItem = {}

queueItem['distance'] = queueSoFar['distance'] + distance

queueItem['maxdistance'] = queueItem['distance'] + self.distance(end, self.searchEnd)

queueItem['nodes'] = queueSoFar['nodes'] + ","+str(end)

queueItem['end'] = end

# Try to insert, keeping the queue ordered by decreasing worst-case distance

count = 0

for test in self.queue:

if test['maxdistance'] > queueItem['maxdistance']:

self.queue.insert(count,queueItem)

break

count = count + 1

else:

self.queue.append(queueItem)

# Show on the map

self.markLine(start,end,0.5,0.5,0.5)

# Parse the supplied OSM file

print ("Loading data...")

obj = GetRoutes()

parser = make\_parser()

parser.setContentHandler(obj)

parser.parse(sys.argv[1])

print ("Routing...")

# Do routing between the two specified nodes

"""Format a route (as list of nodes)"""

data = LoadOsm("giki.osm")

node1 = data.findNode(float(sys.argv[2]),float(sys.argv[3]),'car')

node2 = data.findNode(float(sys.argv[4]),float(sys.argv[5]),'car')

print("NODE 1 : %d" %node1)

print("NODE 2 : %d" %node2)

obj.doRouting(int(node1), int(node2))