'''

A\*, path planner solution

note xml includes all nodes for all partially-present ways

uses a bounding box to ignore nodes outside the region, should be safe

'''

from Tkinter import \*

import struct

import xml.etree.ElementTree as ET

from Queue import \*

import math

# bounds of the window, in lat/long

LEFTLON = 18.055

RIGHTLON = 18.125

TOPLAT = 42.675

BOTLAT = 42.635

WIDTH = RIGHTLON-LEFTLON

HEIGHT = TOPLAT-BOTLAT

# ratio of one degree of longitude to one degree of latitude

LONRATIO = math.cos(TOPLAT\*3.1415/180)

WINWID = 800

WINHGT = (int)((WINWID/LONRATIO)\*HEIGHT/WIDTH)

TOXPIX = WINWID/WIDTH

TOYPIX = WINHGT/HEIGHT

#width,height of elevation array

EPIX = 3601

# approximate number of meters per degree of latitude

MPERLAT = 111000

MPERLON = MPERLAT\*LONRATIO

def node\_dist(n1, n2):

''' Distance between nodes n1 and n2, in meters. '''

dx = (n2.pos[0]-n1.pos[0])\*MPERLON

dy = (n2.pos[1]-n1.pos[1])\*MPERLAT

return math.sqrt(dx\*dx+dy\*dy) # in meters

class Node():

''' Graph (map) node, not a search node! '''

\_\_slots\_\_ = ('id', 'pos', 'ways', 'elev')

def \_\_init\_\_(self,id,p,e=0):

self.id = id

self.pos = p

self.ways = []

self.elev = e

self.waystr = None

def \_\_str\_\_(self):

if self.waystr is None:

self.waystr = self.get\_waystr()

return str(self.pos) + ": " + self.waystr

def get\_waystr(self):

if self.waystr is None:

self.waystr = ""

self.wayset = set()

for w in self.ways:

self.wayset.add(w.way.name)

for w in self.wayset:

self.waystr += w.encode("utf-8") + " "

return self.waystr

class Edge():

''' Graph (map) edge. Includes cost computation.'''

\_\_slots\_\_ = ('way','dest')

def \_\_init\_\_(self, w, src, d):

self.way = w

self.dest = d

self.cost = node\_dist(src,d)

if d.elev > src.elev:

self.cost += (d.elev-src.elev)\*2

if self.way.type == 'steps':

self.cost \*= 1.5

class Way():

''' A way is an entire street, for drawing, not searching. '''

\_\_slots\_\_ = ('name','type','nodes')

# nodes here for ease of drawing only

def \_\_init\_\_(self,n,t):

self.name = n

self.type = t

self.nodes = []

class Planner():

\_\_slots\_\_ = ('nodes', 'ways')

def \_\_init\_\_(self,n,w):

self.nodes = n

self.ways = w

def heur(self,node,gnode):

'''

Heuristic function is just straight-line (flat) distance.

Since the actual cost only adds to this distance, this is admissible.

'''

return node\_dist(node,gnode)

def plan(self,s,g):

'''

Standard A\* search

'''

parents = {}

costs = {}

q = PriorityQueue()

q.put((self.heur(s,g),s))

parents[s] = None

costs[s] = 0

while not q.empty():

cf, cnode = q.get()

if cnode == g:

print ("Path found, time will be",costs[g]\*60/5000) #5 km/hr on flat

return self.make\_path(parents,g)

for edge in cnode.ways:

newcost = costs[cnode] + edge.cost

if edge.dest not in parents or newcost < costs[edge.dest]:

parents[edge.dest] = (cnode, edge.way)

costs[edge.dest] = newcost

q.put((self.heur(edge.dest,g)+newcost,edge.dest))

def make\_path(self,par,g):

nodes = []

ways = []

curr = g

nodes.append(curr)

while par[curr] is not None:

prev, way = par[curr]

ways.append(way.name)

nodes.append(prev)

curr = prev

nodes.reverse()

ways.reverse()

return nodes,ways

class PlanWin(Frame):

'''

All the GUI pieces to draw the streets, allow places to be selected,

and then draw the resulting path.

'''

\_\_slots\_\_ = ('whatis', 'nodes', 'ways', 'elevs', 'nodelab', 'elab', \

'planner', 'lastnode', 'startnode', 'goalnode')

def lat\_lon\_to\_pix(self,latlon):

x = (latlon[1]-LEFTLON)\*(TOXPIX)

y = (TOPLAT-latlon[0])\*(TOYPIX)

return x,y

def pix\_to\_elev(self,x,y):

return self.lat\_lon\_to\_elev(((TOPLAT-(y/TOYPIX)),((x/TOXPIX)+LEFTLON)))

def lat\_lon\_to\_elev(self,latlon):

# row is 0 for 43N, 1201 (EPIX) for 42N

row = (int)((43 - latlon[0]) \* EPIX)

# col is 0 for 18 E, 1201 for 19 E

col = (int)((latlon[1]-18) \* EPIX)

return self.elevs[row\*EPIX+col]

def maphover(self,event):

self.elab.configure(text = str(self.pix\_to\_elev(event.x,event.y)))

for (dx,dy) in [(0,0),(-1,0),(0,-1),(1,0),(0,1),(-1,-1),(-1,1),(1,-1),(1,1)]:

ckpos = (event.x+dx,event.y+dy)

if ckpos in self.whatis:

self.lastnode = self.whatis[ckpos]

lnpos = self.lat\_lon\_to\_pix(self.nodes[self.lastnode].pos)

self.canvas.coords('lastdot',(lnpos[0]-2,lnpos[1]-2,lnpos[0]+2,lnpos[1]+2))

nstr = str(self.lastnode)

nstr += " "

nstr += str(self.nodes[self.whatis[ckpos]].get\_waystr())

self.nodelab.configure(text=nstr)

return

def mapclick(self,event):

''' Canvas click handler:

First click sets path start, second sets path goal

'''

print "Clicked on "+str(event.x)+","+str(event.y)+" last node "+str(self.lastnode)

if self.lastnode is None:

return

if self.startnode is None:

self.startnode = self.nodes[self.lastnode]

self.snpix = self.lat\_lon\_to\_pix(self.startnode.pos)

self.canvas.coords('startdot',(self.snpix[0]-2,self.snpix[1]-2,self.snpix[0]+2,self.snpix[1]+2))

elif self.goalnode is None:

self.goalnode = self.nodes[self.lastnode]

self.snpix = self.lat\_lon\_to\_pix(self.goalnode.pos)

self.canvas.coords('goaldot',(self.snpix[0]-2,self.snpix[1]-2,self.snpix[0]+2,self.snpix[1]+2))

def clear(self):

''' Clear button callback. '''

self.lastnode = None

self.goalnode = None

self.startnode = None

self.canvas.coords('startdot',(0,0,0,0))

self.canvas.coords('goaldot',(0,0,0,0))

self.canvas.coords('path',(0,0,0,0))

def plan\_path(self):

''' Path button callback, plans and then draws path.'''

print "Planning!"

if self.startnode is None or self.goalnode is None:

print "Sorry, not enough info."

return

print ("From", self.startnode.id, "to", self.goalnode.id)

nodes,ways = self.planner.plan(self.startnode, self.goalnode)

lastway = ""

for wayname in ways:

if wayname != lastway:

print wayname

lastway = wayname

coords = []

for node in nodes:

npos = self.lat\_lon\_to\_pix(node.pos)

coords.append(npos[0])

coords.append(npos[1])

#print node.id

self.canvas.coords('path',\*coords)

def \_\_init\_\_(self,master,nodes,ways,coastnodes,elevs):

self.whatis = {}

self.nodes = nodes

self.ways = ways

self.elevs = elevs

self.startnode = None

self.goalnode = None

self.planner = Planner(nodes,ways)

thewin = Frame(master)

w = Canvas(thewin, width=WINWID, height=WINHGT)#, cursor="crosshair")

w.bind("<Button-1>", self.mapclick)

w.bind("<Motion>", self.maphover)

for waynum in self.ways:

nlist = self.ways[waynum].nodes

thispix = self.lat\_lon\_to\_pix(self.nodes[nlist[0]].pos)

if len(self.nodes[nlist[0]].ways) > 2:

self.whatis[((int)(thispix[0]),(int)(thispix[1]))] = nlist[0]

for n in range(len(nlist)-1):

nextpix = self.lat\_lon\_to\_pix(self.nodes[nlist[n+1]].pos)

self.whatis[((int)(nextpix[0]),(int)(nextpix[1]))] = nlist[n+1]

w.create\_line(thispix[0],thispix[1],nextpix[0],nextpix[1])

thispix = nextpix

thispix = self.lat\_lon\_to\_pix(self.nodes[coastnodes[0]].pos)

# also draw the coast:

for n in range(len(coastnodes)-1):

nextpix = self.lat\_lon\_to\_pix(self.nodes[coastnodes[n+1]].pos)

w.create\_line(thispix[0],thispix[1],nextpix[0],nextpix[1],fill="blue")

thispix = nextpix

# other visible things are hiding for now...

w.create\_line(0,0,0,0,fill='orange',width=3,tag='path')

w.create\_oval(0,0,0,0,outline='green',fill='green',tag='startdot')

w.create\_oval(0,0,0,0,outline='red',fill='red',tag='goaldot')

w.create\_oval(0,0,0,0,outline='blue',fill='blue',tag='lastdot')

w.pack(fill=BOTH)

self.canvas = w

cb = Button(thewin, text="Clear", command=self.clear)

cb.pack(side=RIGHT,pady=5)

sb = Button(thewin, text="Plan!", command=self.plan\_path)

sb.pack(side=RIGHT,pady=5)

nodelablab = Label(thewin, text="Node:")

nodelablab.pack(side=LEFT, padx = 5)

self.nodelab = Label(thewin,text="None")

self.nodelab.pack(side=LEFT,padx = 5)

elablab = Label(thewin, text="Elev:")

elablab.pack(side=LEFT, padx = 5)

self.elab = Label(thewin, text = "0")

self.elab.pack(side=LEFT, padx = 5)

thewin.pack()

def build\_elevs(efilename):

''' read in elevations from a file. '''

efile = open(efilename)

estr = efile.read()

elevs = []

for spot in range(0,len(estr),2):

elevs.append(struct.unpack('>h',estr[spot:spot+2])[0])

return elevs

def build\_graph(elevs):

''' Build the search graph from the OpenStreetMap XML. '''

tree = ET.parse('dbv.osm')

root = tree.getroot()

nodes = dict()

ways = dict()

waytypes = set()

coastnodes = []

for item in root:

if item.tag == 'node':

coords = ((float)(item.get('lat')),(float)(item.get('lon')))

# row is 0 for 43N, 1201 (EPIX) for 42N

erow = (int)((43 - coords[0]) \* EPIX)

# col is 0 for 18 E, 1201 for 19 E

ecol = (int)((coords[1]-18) \* EPIX)

try:

el = elevs[erow\*EPIX+ecol]

except IndexError:

el = 0

nodes[(long)(item.get('id'))] = Node((long)(item.get('id')),coords,el)

elif item.tag == 'way':

if item.get('id') == '157161112': #main coastline way ID

for thing in item:

if thing.tag == 'nd':

coastnodes.append((long)(thing.get('ref')))

continue

useme = False

oneway = False

myname = 'unnamed way'

for thing in item:

if thing.tag == 'tag' and thing.get('k') == 'highway':

useme = True

mytype = thing.get('v')

if thing.tag == 'tag' and thing.get('k') == 'name':

myname = thing.get('v')

if thing.tag == 'tag' and thing.get('k') == 'oneway':

if thing.get('v') == 'yes':

oneway = True

if useme:

wayid = (long)(item.get('id'))

ways[wayid] = Way(myname,mytype)

nlist = []

for thing in item:

if thing.tag == 'nd':

nlist.append((long)(thing.get('ref')))

thisn = nlist[0]

for n in range(len(nlist)-1):

nextn = nlist[n+1]

nodes[thisn].ways.append(Edge(ways[wayid],nodes[thisn],nodes[nextn]))

thisn = nextn

if not oneway:

thisn = nlist[-1]

for n in range(len(nlist)-2,-1,-1):

nextn = nlist[n]

nodes[thisn].ways.append(Edge(ways[wayid],nodes[thisn],nodes[nextn]))

thisn = nextn

ways[wayid].nodes = nlist

print len(coastnodes)

print coastnodes[0]

print nodes[coastnodes[0]]

return nodes, ways, coastnodes

elevs = build\_elevs("N42E018.HGT")

nodes, ways, coastnodes = build\_graph(elevs)

master = Tk()

thewin = PlanWin(master,nodes,ways,coastnodes,elevs)

mainloop()