**import** **cv2**

**import** **numpy** **as** **np**

**import** **sys**

**import** **math**

*# from matplotlib.path import Path*

*# ============================================================*

*# Globals*

*# ============================================================*

np.set\_printoptions(threshold=np.nan) *# Set such that full image array is printed out*

sys.setrecursionlimit(150000) *# Reset python's default recursion limit (1000)*

*# 1. Basic Infrastructure*

map\_labeled = cv2.imread('ass3-labeled.pgm', 0) *# Load labeled map as grayscale*

map\_campus = cv2.imread('ass3-campus.pgm', 1) *# Load campus map(for display) as color*

map\_binary = cv2.cvtColor(map\_campus,cv2.COLOR\_BGR2GRAY) *# Convert campus map to grayscale for contouring*

MAP\_H = len(map\_binary)

MAP\_W = len(map\_binary[0])

buildings = []

num\_buildings = 0

monument = {}

*# 2. Spatial Relationships*

n\_table = []

e\_table = []

s\_table = []

w\_table = []

near\_table = []

*# 3a. User Interface*

drawing = False *# true if mouse is pressed*

mode = True *# if True, generate path. Press 'm' to toggle to curve*

ix,iy = -1,-1

click\_count = -1

clicks = []

*# Green, Red, Blue, Teal, Yellow, Orange, Magenta*

*# colors = [(0,255,0),(0,0,255),(255,0,0), (255,255,0), (0,255,255),(0,128,255),(255,0,255)]*

*# All Blue (user clicks)*

colors = [(255,0,0),(255,0,0),(255,0,0),(255,0,0),(255,0,0),(255,0,0),(255,0,0),(255,0,0)]

color = colors[0] *# Default*

*# 3b. Cloud Ambiguity*

cloud = {}

called = {}

recursive\_calls = 0

pix = 2 *# Number of pixels to check in each direction for cloud generation*

*# 4. Path Generation*

*# S1G1: Broadway Gates -^ Mudd*

*# S2G2: Pupin -v Alma Mater*

*# S3G3: Carman -> Hartley*

*# S4G4: Kent <- Mathematics*

*# S5G5: Butler ^- Physical Fitness Center*

*# S6G6: Journalism -^ Uris*

*# S7G7: Avery ^- Shapiro*

*# S8G8: Lawn ^- Low*

S\_LIST = [(8,320),(35,4),(78,477),(232,285),(132,443),(52,398),(203,160),(135,369)]

G\_LIST = [(205,51),(137,291),(257,374),(36,178),(88,68),(134,97),(143,37),(172,212)]

paths = [] *# Will contain all the sequences of instructions for each 8 paths*

path\_parens = []

path\_no\_parens = []

itinerary\_num = 0

user\_responses = []

counter = 0

*# ============================================================*

*# The "What"*

*# ============================================================*

**def** load\_names(filename):

*"""Load files from text file in order"""*

names = {}

infile = open(filename, 'rU')

**while** True:

**try**:

line = infile.readline().replace('"', '').split('=')

n = line[0]

name = line[1].rstrip('**\r\n**')

names[n] = name

**except** **IndexError**:

**break**

**return** names

**def** analyze\_what(names):

*"""Find information about buildings and save in list of dicts"""*

**global** num\_buildings, buildings

num\_buildings = len(names)

buildings = list(np.zeros(num\_buildings))

areas = measure\_areas()

*# print areas*

*# Find contours in binary campus map image*

*# Contours is a Python list of all the contours in the image*

*# Each contour is a np array of (x,y) boundary points of each object*

contours,hierarchy = cv2.findContours(map\_binary,cv2.RETR\_LIST,cv2.CHAIN\_APPROX\_SIMPLE)

**for** cnt **in** contours:

building = {}

idx = id\_building(cnt)

**if** idx **is** None:

**continue**

building['number'] = idx

building['name'] = names[str(idx)]

building['area'] = areas[str(idx)]

mbr, centroid, extent, xywh = measure\_building(cnt,building['area'])

building['mbr'] = mbr

building['centroid'] = centroid

building['extent'] = extent

building['xywh'] = xywh

*# Note: this was used by analyze\_shapes and analyze\_extents*

*# building['cnt'] = cnt*

buildings[(idx-1)] = building

max\_area, min\_area = analyze\_areas(buildings) *# add True arg to print results*

find\_monument()

**for** building **in** buildings:

location = describe\_location(building)

size = describe\_size(building, max\_area)

shape = describe\_shape(building)

**if** 'description' **not** **in** building:

description = []

**else**:

description = building['description']

**if** building['area'] **is** min\_area: *# replace with extrema*

description.append('smallest')

**else**:

description.append(size)

description.extend(shape)

description.extend(location)

building['description'] = description

*# Reduce descriptions*

find\_extrema()

find\_ambiguity()

*# multiple = describe\_multiplicity*

*# analyze\_extents(buildings)*

*# analyze\_shapes(buildings)*

**def** measure\_areas():

*"""Count areas for each building"""*

areas = {}

**for** x **in** xrange(MAP\_W):

**for** y **in** xrange(MAP\_H):

pixel = map\_labeled[(y,x)]

**if** str(pixel) **in** areas:

areas[str(pixel)] += 1

**else**:

areas[str(pixel)] = 1

**return** areas

**def** id\_building(cnt):

*"""Identify what building a contour represents by its pixel value"""*

*# To get all the points which comprise an object*

*# Numpy function gives coordinates in (row, col)*

*# OpenCV gives coordinates in (x,y)*

*# Note row = x and col = y*

mask = np.zeros(map\_binary.shape,np.uint8)

cv2.drawContours(mask,[cnt],0,255,-1)

pixelpoints = np.transpose(np.nonzero(mask))

*#pixelpoints = cv2.findNonZero(mask)*

*# Use color to determine index, which will give us name*

color = cv2.mean(map\_labeled,mask=mask)

*# print color*

**if** (color[0] > 0.9):

idx = int(round(color[0], 0))

**return** idx

**else**:

**return** None

**def** measure\_building(cnt, area, print\_rect=False):

*"""Use OpenCV to create a bounding rectangle and find center of mass"""*

*# Let (x,y) be top-left coordinate and (w,h) be width and height*

*# Find min, max value of x, min, max value of y*

x,y,w,h = cv2.boundingRect(cnt)

xywh = (x,y,w,h)

mbr = [(x,y),(x+w,y+h)]

roi = map\_campus[y:y+h,x:x+w]

*# To draw a rectangle, you need T-L corner and B-R corner*

*# We have mbr[0] = T-L corner, mbr[1] = B-R corner*

**if** print\_rect:

cv2.rectangle(map\_campus,(x,y),(x+w,y+h),(200,0,0),2)

*# print " Minimum Bounding Rectangle: ({0},{1}), ({2},{3})".format(x,y,(x+w),(y+h))*

*# Calculate centroid based on bounding rectangle*

cx = x+(w/2)

cy = y+(h/2)

centroid = (cx, cy)

*# DRAW CENTROIDS!*

*# cv2.circle(map\_campus, centroid, 3, (255,255,0), -1)*

*# To draw a circle, you need its center coordinates and radius*

*# print ' Center of Mass:', centroid*

rect\_area = w\*h

extent = float(area)/rect\_area

*# Discarded methods*

*# Image moments help you to calculate center of mass, area of object, etc.*

*# cv2.moments() gives dictionary of all moment values calculated*

*# M = cv2.moments(cnt)*

*# Centroid is given by the relations*

*# cx = int(M['m10']/M['m00'])*

*# cy = int(M['m01']/M['m00'])*

*# centroid = (cx, cy)*

*# Contour area is given by the function cv2.contourArea(cnt) or*

*# area = M['m00']*

*# print ' Area:', area*

*# area = cv2.contourArea(cnt)*

*# x,y,w,h = cv2.boundingRect(cnt)*

*# rect\_area = w\*h*

*# extent = float(area)/rect\_area*

*# print ' Extent:', round(extent, 3)*

*# label = str(idx) + ' : ' + str(area) + ' : ' + str(extent)*

*# cv2.putText(map\_campus, str(idx), (cx,cy), cv2.FONT\_HERSHEY\_SIMPLEX, 0.3, 255)*

*# check curve for convexity defects and correct it*

*# pass in contour points, hull, !returnPoints return indices*

*# hull = cv2.convexHull(cnt,returnPoints = False)*

*# defects = cv2.convexityDefects(cnt,hull) # array*

*# if len(hull) > 3 and len(cnt) > 3 and (defects is not None):*

*# for i in range(defects.shape[0]):*

*# s,e,f,d = defects[i,0]*

*# start = tuple(cnt[s][0])*

*# end = tuple(cnt[e][0])*

*# far = tuple(cnt[f][0])*

*# # print start, end, far*

*# cv2.line(map\_campus,start,end,[0,255,0],1)*

*# cv2.circle(map\_campus,far,3,[255,0,255],-1)*

*# this just draws the rect again*

*#cv2.drawContours(map\_campus, contours, 0, (0,0,255), 1)*

*# find corners - this method is buggy*

*# dst = cv2.cornerHarris(map\_binary,3,3,0.2)*

*# dst = cv2.dilate(dst,None)*

*# map\_campus[dst>0.01\*dst.max()]=[0,0,255]*

**return** mbr, centroid, extent, xywh

**def** analyze\_areas(buildings, print\_results=False):

*"""Sort buildings by area, determine cutoff for size and return max"""*

*# num\_buildings = len(buildings)*

sorted\_buildings = sorted(buildings, key=**lambda** k:-k['area'])

indices = [(sorted\_buildings[i]['number']-1) **for** i **in** range(num\_buildings)]

areas = [(sorted\_buildings[i]['area']) **for** i **in** range(num\_buildings)]

max\_area = areas[0]

avg\_area = sum(areas)/num\_buildings

min\_area = areas[-1]

*# Print results to analyze cutoffs for size categories*

**if** (print\_results):

**print** 'Analyzing building areas...'

ratios = [round(float(areas[i])/max\_area,3) **for** i **in** range(num\_buildings)]

ratio\_diffs = [round((ratios[i+1]-ratios[i]),3) **for** i **in** range(num\_buildings-1)]

ratio\_diffs.insert(0,0)

max\_area\_ratios = [round(max\_area/areas[i],3) **for** i **in** range(num\_buildings)]

**print** 'Max Area:', max\_area

**print** 'Average:', avg\_area

**print** 'Min Area:', min\_area

**print** 'Area**\t**Ratio r**\t**Diff r**\t**Max r**\t**Building'

**for** i **in** xrange(num\_buildings):

idx = indices[i]

**print** areas[i], '**\t**', ratios[i], '**\t**', ratio\_diffs[i], '**\t**', max\_area\_ratios[i], '**\t**', idx+1, buildings[idx]['name']

**return** max\_area, min\_area

**def** analyze\_extents():

*"""Sort buildings by extent and determine cutoff for rectangles"""*

**print** 'Analyzing building extents (area/mbr) and convexity...'

*# num\_buildings = len(buildings)*

sorted\_buildings = sorted(buildings, key=**lambda** k:-k['extent'])

indices = [(sorted\_buildings[i]['number']-1) **for** i **in** range(num\_buildings)]

**for** i **in** indices:

building = buildings[i]

convex = cv2.isContourConvex(building['cnt'])

**print** round(building['extent'],4), '**\t**', convex, '**\t**', i+1, building['name']

**def** analyze\_shapes(buildings):

*"""Sort building by shape similarity (not very good results)"""*

**print** 'Analyzing shape similarity with cv2.matchShapes...'

*# num\_buildings = len(buildings)*

shape\_sim = {}

**for** i **in** xrange(num\_buildings):

**for** j **in** xrange(i+1, num\_buildings):

cnt1 = buildings[i]['cnt']

cnt2 = buildings[j]['cnt']

ret = cv2.matchShapes(cnt1,cnt2, 1,0.0)

shape\_sim[(i,j)] = ret

sorted\_sim = sorted([(value,key) **for** (key,value) **in** shape\_sim.items()])

**for** sim **in** sorted\_sim[:40]:

bldg1 = sim[1][0]

bldg2 = sim[1][1]

**print** round(sim[0],4), '**\t**', buildings[bldg1]['name'], '&', buildings[bldg2]['name']

**def** describe\_size(building, max\_area):

ratio = float(building['area'])/max\_area

**if** ratio > 0.7: *# cutoff at College Walk*

**return** 'colossal'

**elif** ratio > 0.4: *# cutoff at Journalism & Furnald*

**return** 'large'

**elif** ratio > 0.16: *# cutoff at Philosophy*

**return** 'medium'

**elif** ratio > 0.1: *# cutoff Earl Hall*

**return** 'small'

**else**:

**return** 'tiny'

**def** describe\_shape(building,draw\_points=False):

*"""Describe shape based on corner and midpoint counts"""*

descriptions = []

xywh = building['xywh']

corners\_count, midpoints\_count, xywh2 = count\_points(building,xywh,draw\_points)

*# print building['number'], building['name']*

*# print ' Tolerance', tolerance*

*# print '', corners\_filled, 'Corners Count', corners\_count*

*# print '', midpoints\_filled, 'Midpoints Count', midpoints\_count*

*# Difference between height and width should be small enough*

*# Decided not to use absolute value as differnce is relative*

*# Also check that building fills out most of the MBR*

*# Ruling out Journalism & Furnald, and Chandler & Havemeyer*

x,y,w,h = unpack(xywh)

**if** (abs(h-w) <= max(h,w)/5) **and** (building['extent'] > 0.7):

is\_square = True

**else**:

is\_square = False

*# Used this method to check accuracy of my rectangle check*

*# if (cv2.isContourConvex(building['cnt'])):*

*# print 'Rectangle'*

*# Check shape conditions:*

*# [] must have all corners and midpoints filled*

*# + should have empty corners and all midpoints*

*# I should have all corners but only 2 midpoints*

*# C should have all corners but one midpoint missing*

*# L should have 3 corners and only 2 midpoints*

*# T should have 2 corners but all midpoints*

*# Anything else is classified as 'irregular'*

**if** (corners\_count == 4 **and** midpoints\_count == 4):

*# because if it square, rectangular would be redundant*

**if** (is\_square):

descriptions.append('square')

**else**:

descriptions.append('rectangular')

**elif** (corners\_count == 0 **and** midpoints\_count == 4):

**if** (is\_square):

descriptions.append('squarish cross-shaped')

**else**:

cc, mc, xywh2 = count\_points(building,xywh2,draw\_points)

**if** (cc%2 == 1): *# Not symmetrical*

descriptions.append('bell-shaped')

**else**:

descriptions.append('cross-shaped')

**elif** (corners\_count == 4 **and** midpoints\_count == 2):

descriptions.append('I-shaped')

**elif** (corners\_count == 4 **and** midpoints\_count == 3):

descriptions.append('U-shaped')

**elif** (corners\_count == 3 **and** midpoints\_count == 2):

descriptions.append('L-shaped')

**elif** (corners\_count == 2 **and** midpoints\_count == 4):

descriptions.append('almost rectangular')

**else**:

descriptions.append('irregularly shaped')

*# Check orientation conditions:*

*# If width is > 1.5 \* height, "wide", E-W oriented*

*# If height is > 1.5 \* width, "tall", N-S oriented*

*# Decided not to include symmetrically oriented*

*# if (w > 1.5 \* h):*

*# descriptions.append('oriented East-West')*

*# elif (h > 1.5 \* w):*

*# descriptions.append('oriented North-South')*

*# print ' Description', descriptions*

**return** descriptions

**def** unpack(tup):

**if** len(tup) **is** 4:

**return** tup[0],tup[1],tup[2],tup[3]

**elif** len(tup) **is** 5:

**return** tup[0],tup[1],tup[2],tup[3],tup[4]

**def** count\_points(building,xywh,draw\_points):

x,y,w,h = unpack(xywh)

*# Tolerance based on ratio of min(w,h) as building sizes vary*

tolerance = min(w,h)/10

*# Shift x,y,w,h so corners and midpoints are closer to center*

*# Else they may report false negative on the MBR perimeter, esp*

*# for bumpy buildings*

x += tolerance

y += tolerance

w -= 2\*tolerance

h -= 2\*tolerance

*# Extract four corners*

nw = (x,y)

se = (x+w,y+h)

ne = (x+w,y)

sw = (x,y+h)

*# Extract midpoints on every wall face*

n = (x+(w/2),y)

e = (x+w,y+(h/2))

s = (x+(w/2),y+h)

west = (x,y+(h/2))

corners = [nw,se,ne,sw]

midpoints = [n,e,s,west] *# west because it overwrites width*

corners\_filled = [] *# nw, ne, se, sw*

midpoints\_filled = [] *# n, e, s, west*

**for** corner **in** corners:

**if** map\_labeled[tuple(reversed(corner))] == building['number']:

corners\_filled.append(1)

**if** draw\_points:

cv2.circle(map\_campus, corner, 1, (255,255,0), -1)

**else**:

corners\_filled.append(0)

**if** draw\_points:

cv2.circle(map\_campus, corner, 1, (0,0,255), -1)

**for** midpoint **in** midpoints:

**if** map\_labeled[tuple(reversed(midpoint))] == building['number']:

midpoints\_filled.append(1)

**if** draw\_points:

cv2.circle(map\_campus, midpoint, 1, (0,255,0), -1)

**else**:

midpoints\_filled.append(0)

**if** draw\_points:

cv2.circle(map\_campus, midpoint, 1, (0,0,255), -1)

*# Count the number of corners and midpoints for each building*

*# Not necessary to consider order at this point*

corners\_count = corners\_filled.count(1)

midpoints\_count = midpoints\_filled.count(1)

**return** corners\_count, midpoints\_count, (x,y,w,h)

**def** find\_monument():

**global** monument, buildings

**for** idx **in** xrange(num\_buildings):

**if** buildings[idx]['xywh'][2] > MAP\_W - 10:

monument = buildings[idx]

buildings[idx]['description'] = ['longest']

**return**

**def** describe\_location(building):

**if** building['number'] **is** monument['number']:

**return** []

location = []

marker = monument['centroid'][1] *# cy for College Walk*

h = building['mbr'][1][1] - building['mbr'][0][1]

w = building['mbr'][1][0] - building['mbr'][0][0]

*# Reduce h/w shift so buildings are positioned properly*

h = int(h \* 0.7)

w = int(w \* 0.7)

cx = building['centroid'][0]

cy = building['centroid'][1]

*# Draw lines*

*# if building['number'] is 10:*

*# cv2.line(map\_campus,(0,marker/2),(MAP\_W,marker/2),[0,255,0],2)*

*# cv2.line(map\_campus,(0,marker),(MAP\_W,marker),[0,255,0],2)*

*# cv2.line(map\_campus,(0,h),(MAP\_W,h),[0,255,0],2)*

*# cv2.line(map\_campus,(0,MAP\_H-h),(MAP\_W,MAP\_H-h),[0,255,0],2)*

*# cv2.line(map\_campus,(int((MAP\_W/2)-w),0),(int((MAP\_W/2)-w),MAP\_H),[0,255,0],2)*

*# cv2.line(map\_campus,(int((MAP\_W/2)+w),0),(int((MAP\_W/2)+w),MAP\_H),[0,255,0],2)*

*# cv2.line(map\_campus,(w,0),(w,MAP\_H),[0,255,0],2)*

*# cv2.line(map\_campus,(MAP\_W-w,0),(MAP\_W-w,MAP\_H),[0,255,0],2)*

*# Locate buildings on borders or central axis*

**if** (cx < w) **and** (cy < h):

location.append('northwest corner')

**elif** (cx > MAP\_W-w) **and** (cy < h):

location.append('northeast corner')

**elif** (cx > MAP\_W-w) **and** (cy > MAP\_H-h):

location.append('southeast corner')

**elif** (cx < w) **and** (cy > MAP\_H-h):

location.append('southwest corner')

**elif** (cy < h):

location.append('northernmost')

**elif** (cy > MAP\_H-h):

location.append('southernmost')

**elif** (cx > MAP\_W-w):

location.append('easternmost')

**elif** (cx < w):

location.append('westernmost')

*# For buildings not on north/south borders, locate whether on*

*# upper/central/lower campus*

**if** (cy > marker) **and** (cy < MAP\_H-h): *# southernmost already weeded out*

location.append('lower campus')

**elif** (cy > h) **and** (cy < marker/2):

location.append('upper campus')

**elif** (cy < marker) **and** (cy > marker/2) **and** (cx < MAP\_W-w) **and** (cx > w): *# central\_axis(cx,w):*

location.append('central campus')

*# For buildings not on east/west borders*

*# if (cx > (MAP\_W/2)-w) and (cx < (MAP\_W/2)+w) and (cx > w) and (cx < MAP\_W-w):*

*# location.append('on central axis')*

**return** location

**def** central\_axis(cx,w):

**if** (cx > (MAP\_W/2)-w) **and** (cx < (MAP\_W/2)+w) **and** (cx > w) **and** (cx < MAP\_W-w):

**return** True

**return** False

**def** counting\_dict(dic,key):

**if** key **in** dic:

dic[key] += 1

**else**:

dic[key] = 1

**return** dic

**def** find\_extrema():

*"""Find singularly defining characteristics and remove other details"""*

**global** buildings

characteristics = {}

**for** idx **in** xrange(num\_buildings):

bldg1 = buildings[idx]

description = bldg1['description']

**for** characteristic **in** description:

*# print characteristic*

count = 0

*# Add to counting dictionary*

characteristics = counting\_dict(characteristics, characteristic)

**for** jdx **in** xrange(num\_buildings):

bldg2 = buildings[jdx]

**if** (idx != jdx) **and** (characteristic **in** tuple(bldg2['description'])):

count += 1

**if** count **is** 0 **and** characteristic != 'almost rectangular' **and** characteristic != 'southernmost':

*# 'Found extrema!', characteristic*

extrema = [characteristic]

bldg1['description'] = extrema

buildings[idx] = bldg1

**return** characteristics

**def** find\_ambiguity():

**global** buildings

**for** idx **in** xrange(num\_buildings):

bldg1 = buildings[idx]

**for** jdx **in** xrange(num\_buildings):

bldg2 = buildings[jdx]

**if** idx != jdx **and** bldg1['description'] == bldg2['description']:

**if** is\_north(bldg1,bldg2):

bldg2['description'].insert(0,'more northern')

bldg1['description'].insert(0,'more southern')

**elif** is\_south(bldg1,bldg2):

bldg1['description'].insert(0,'more northern')

bldg2['description'].insert(0,'more southern')

buildings[idx] = bldg1

buildings[jdx] = bldg2

*# print 'Ambiguity between', bldg1['name'], 'and', bldg2['name']*

**def** print\_info():

*"""System output for part 1"""*

**for** building **in** buildings:

**print** building['number'], ':', building['name']

**print** ' Minimum Bounding Rectangle:', building['mbr'][0], ',', building['mbr'][1]

**print** ' Center of Mass:', building['centroid']

**print** ' Area:', building['area']

**print** ' Description:', building['description']

*# ============================================================*

*# The "Where"*

*# ============================================================*

**def** analyze\_where(buildings):

*"""Find all binary spatial relationships for every pair,*

*and apply transitive reduction."""*

**global** n\_table, e\_table, s\_table, w\_table, near\_table

n\_table = np.zeros((num\_buildings, num\_buildings),bool)

e\_table = np.zeros((num\_buildings, num\_buildings),bool)

s\_table = np.zeros((num\_buildings, num\_buildings),bool)

w\_table = np.zeros((num\_buildings, num\_buildings),bool)

near\_table = np.zeros((num\_buildings, num\_buildings),bool)

**for** s **in** xrange(0, num\_buildings):

**for** t **in** xrange(0, num\_buildings):

**if** s != t:

source = buildings[s]

target = buildings[t]

n\_table[s][t] = is\_north(source,target)

s\_table[s][t] = is\_south(source,target)

e\_table[s][t] = is\_east(source,target)

w\_table[s][t] = is\_west(source,target)

near\_table[s][t] = is\_near(source,target)

**print** 'North relationships:'

count = print\_table(n\_table, num\_buildings)

**print** 'South relationships:'

count += print\_table(s\_table, num\_buildings)

**print** 'East relationships:'

count += print\_table(e\_table, num\_buildings)

**print** 'West relationships:'

count += print\_table(w\_table, num\_buildings)

**print** 'Near relationships:'

count += print\_table(near\_table, num\_buildings)

**print** 'Total count:', count

n\_table, s\_table, e\_table, w\_table, near\_table = transitive\_reduce(n\_table, s\_table, e\_table, w\_table, near\_table)

**print** 'After transitive reduction...'

**print** 'North relationships:'

count = print\_table(n\_table, num\_buildings)

**print** 'South relationships:'

count += print\_table(s\_table, num\_buildings)

**print** 'East relationships:'

count += print\_table(e\_table, num\_buildings)

**print** 'West relationships:'

count += print\_table(w\_table, num\_buildings)

**print** 'Near relationships:'

count += print\_table(near\_table, num\_buildings)

**print** 'Total count:', count

print\_table\_info(n\_table, buildings, 'North')

print\_table\_info(s\_table, buildings, 'South')

print\_table\_info(e\_table, buildings, 'East')

print\_table\_info(w\_table, buildings, 'West')

print\_table\_info(near\_table, buildings, 'Near')

**return** n\_table, s\_table, e\_table, w\_table, near\_table

**def** analyze\_single\_where(source, direction, buildings):

*"""Analyze relations for single building"""*

*# Try 11 Lowe and then 21 Journalism*

*# num\_buildings = len(buildings)*

**for** target **in** xrange(0, num\_buildings):

**if** source != target:

s = buildings[source]

t = buildings[target]

**if** (direction == "north"):

triangulate\_FOV(s,t,-1,0,1,draw=True)

**elif** (direction == "east"):

triangulate\_FOV(s,t,MAP\_W,-1,1.2,draw=True)

**elif** (direction == "south"):

triangulate\_FOV(s,t,-1,MAP\_H,1,draw=True)

**elif** (direction == "west"):

triangulate\_FOV(s,t,0,-1,1.2,draw=True)

**def** is\_north(s,t):

*"""Find out if 'North of S is T'"""*

*# Form triangle to north border: (x,0)*

**return** triangulate\_FOV(s,t,-1,0,0.8)

**def** is\_south(s,t):

*"""Find out if 'South of S is T'"""*

*# Form triangle to south border: (x,MAP\_H)*

**return** triangulate\_FOV(s,t,-1,MAP\_H,0.8)

**def** is\_east(s,t):

*"""Find out if 'East of S is T'"""*

*# Form triangle to east border: (MAP\_W,y)*

**return** triangulate\_FOV(s,t,MAP\_W,-1,1.5)

**def** is\_west(s,t):

*"""Find out if 'West of S is T'"""*

*# Form triangle to west border: (0,y)*

**return** triangulate\_FOV(s,t,0,-1,1.5)

**def** triangulate\_FOV(s,t,x,y,slope,draw=False):

*"""Create a triangle FOV with 3 points and*

*check if t is within triangle"""*

*# Check if input is a building (if so, leave it)*

*# or an int (if so, change to a building)*

**if** type(s) == int **and** type(t) == int:

s = buildings[s]

t = buildings[t]

**if** y **is** 0:

fov = 'north\_fov'

**elif** y **is** MAP\_H:

fov = 'south\_fov'

**elif** x **is** MAP\_W:

fov = 'east\_fov'

**elif** x **is** 0:

fov = 'west\_fov'

**if** fov **not** **in** s:

*# 0. Find (x,y) for source and target*

p0 = s['centroid']

p4 = t['centroid']

*# 1. Determine slopes m1 and m2*

*# if (s['number'] == 21):*

*# slope = 3*

m1 = slope

m2 = -slope

*# print "m1, m2", m1, m2*

*# 2. Find b = y - mx using origin and slope*

b1 = p0[1] - m1\*p0[0]

b2 = p0[1] - m2\*p0[0]

*# print "b1, b2", b1, b2*

*# 3. Calculate 2 other points in FOV triangle*

*# Direction is determined by what x or y values*

*# are given for p1 and p2*

**if** (x == -1): *# y given, so North/South direction*

x1 = int((y-b1)/m1)

x2 = int((y-b2)/m2)

*# print "x1, x2", x1, x2*

p1 = (x1,y)

p2 = (x2,y)

**elif** (y == -1): *# x given, so East/West direction*

y1 = int((m1\*x) + b1)

y2 = int((m2\*x) + b2)

*# print "y1, y2", x1, y2*

p1 = (x,y1)

p2 = (x,y2)

**if** (draw == True):

cv2.line(map\_campus,p0,p1,(0,255,0),2)

cv2.line(map\_campus,p0,p2,(0,255,0),2)

*# Mandatory: Add new FOV to building dictionary for reuse*

s[fov] = (p0,p1,p2)

idx = s['number'] - 1

buildings[idx] = s

*# If FOV has been pre-calculated, just use the points to check*

**else**:

p0 = s[fov][0]

p1 = s[fov][1]

p2 = s[fov][2]

p4 = t['centroid']

*# 4. Check whether target centroid is in the field of view*

**if** is\_in\_triangle(p4,p0,p1,p2):

**if** (draw == True):

cv2.circle(map\_campus, p4, 6, (0,255,0), -1)

**return** True

*# Special case for campus-wide College Walk, add centroids*

**if** (t['number'] == monument['number']):

mid = t['centroid']

p5 = (MAP\_W/5,mid[1])

p6 = (MAP\_W\*4/5,mid[1])

**if** is\_in\_triangle(p5,p0,p1,p2):

**if** (draw == True):

cv2.circle(map\_campus, p5, 6, (0,255,0), -1)

**return** True

**elif** is\_in\_triangle(p6,p0,p1,p2):

**if** (draw == True):

cv2.circle(map\_campus, p6, 6, (0,255,0), -1)

**return** True

**return** False *# if not in FOV, return false*

*# 4. Check whether target centroid is in the field of view*

**if** is\_in\_triangle(p3,p0,p1,p2):

**if** (draw == True):

cv2.circle(map\_campus, p3, 6, (0,255,0), -1)

**return** True

*# Special case for campus-wide College Walk, add centroids*

**if** (t['number'] == monument['number']):

mid = t['centroid']

p5 = (MAP\_W/5,mid[1])

p6 = (MAP\_W\*4/5,mid[1])

**if** is\_in\_triangle(p5,p0,p1,p2):

**if** (draw == True):

cv2.circle(map\_campus, p5, 6, (0,255,0), -1)

**return** True

**elif** is\_in\_triangle(p6,p0,p1,p2):

**if** (draw == True):

cv2.circle(map\_campus, p6, 6, (0,255,0), -1)

**return** True

**return** False *# if not in FOV, return false*

**def** same\_side(p1,p2,a,b):

cp1 = np.cross(np.subtract(b,a), np.subtract(p1,a))

cp2 = np.cross(np.subtract(b,a), np.subtract(p2,a))

**if** np.dot(cp1,cp2) >= 0:

**return** True

**else**:

**return** False

**def** is\_in\_triangle(p,a,b,c):

**if** same\_side(p,a,b,c) **and** same\_side(p,b,a,c) **and** same\_side(p,c,a,b):

**return** True

**else**:

**return** False

**def** shift\_corners(building, shift):

*# Shift should be negative if you want to tuck in points*

x,y,w,h = unpack(building['xywh'])

*# Shift x,y,w,h so corners and midpoints are closer/farther to center*

x -= shift

y -= shift

w += 2\*shift

h += 2\*shift

**return** x,y,w,h

**def** extract\_corners(x,y,w,h):

nw = (x,y)

ne = (x+w,y)

se = (x+w,y+h)

sw = (x,y+h)

**return** nw,ne,se,sw

**def** draw\_rectangle(nw,ne,se,sw):

cv2.line(map\_campus,nw,ne,(0,128,255),2)

cv2.line(map\_campus,ne,se,(0,128,255),2)

cv2.line(map\_campus,se,sw,(0,128,255),2)

cv2.line(map\_campus,sw,nw,(0,128,255),2)

**if** (diagonal):

cv2.line(map\_campus,nw,se,(0,128,255),2)

**def** draw\_triangle(p1,p2,p3):

cv2.line(map\_campus,p1,p2,(0,128,255),2)

cv2.line(map\_campus,p2,p3,(0,128,255),2)

cv2.line(map\_campus,p3,p1,(0,128,255),2)

**def** get\_near\_points(building,shift):

**if** 'near\_points' **not** **in** building: *# or building['number'] > num\_buildings:*

*# Extract four corners: nw,ne,se,sw*

x1,y1,w1,h1 = shift\_corners(building,shift)

p1,p2,p3,p4 = extract\_corners(x1,y1,w1,h1)

p0 = building['centroid']

points = (p1,p2,p3,p4,p0)

*# draw\_rectangle(p1,p2,p3,p4)*

*# Add new points to source*

building['near\_points'] = points

idx = building['number'] - 1

buildings[idx] = building

**else**:

points = building['near\_points']

**return** points

**def** is\_near(s,t,draw=False):

*"""Near to S is T"""*

**if** type(s) == int **and** type(t) == int:

s = buildings[s]

t = buildings[t]

shift = 15 *# Empirically chosen*

s\_points = get\_near\_points(s,shift)

t\_points = get\_near\_points(t,shift)

s1,s2,s3,s4,s0 = unpack(s\_points)

t1,t2,t3,t4,t0 = unpack(t\_points)

*# Check whether any corner in expanded target rectangle*

*# lies inside one of the two triangles that form the*

*# source rectangle*

**for** pt **in** t\_points:

**if** is\_in\_triangle(pt,s1,s2,s3) **or** is\_in\_triangle(pt,s3,s4,s1):

*# Optional*

**if** (draw):

**if** is\_in\_triangle(pt,s1,s2,s3):

draw\_triangle(s1,s2,s3)

**else**:

draw\_triangle(s3,s4,s1)

*# draw\_rectangle(t1,t2,t3,t4)*

cv2.circle(map\_campus, s0, 6, (0,128,255), -1)

cv2.circle(map\_campus, t0, 6, (0,128,255), -1)

cv2.circle(map\_campus, pt, 6, (0,128,255), -1)

cv2.circle(map\_campus, pt, 3, (0,255,255), -1)

*# Mandatory*

**return** True

**return** False

**def** transitive\_reduce(n\_table, s\_table, e\_table, w\_table, near\_table):

*"""Output should use building names rather than numbers"""*

**for** t **in** range(0, num\_buildings):

**for** s **in** range(0, num\_buildings):

**if** n\_table[s][t]:

**for** u **in** range(0, num\_buildings):

**if** n\_table[t][u]:

n\_table[s][u] = False

**if** s\_table[s][t]:

**for** u **in** range(0, num\_buildings):

**if** s\_table[t][u]:

s\_table[s][u] = False

**if** w\_table[s][t]:

**for** u **in** range(0, num\_buildings):

**if** w\_table[t][u]:

w\_table[s][u] = False

**if** e\_table[s][t]:

**for** u **in** range(0, num\_buildings):

**if** e\_table[t][u]:

e\_table[s][u] = False

*# If t is north of s we no longer need to say s is south of t*

*# Similarly, east west relationships can be inferred*

**for** s **in** range(0, num\_buildings):

**for** t **in** range(0, num\_buildings):

**if** n\_table[s][t] **and** s\_table[t][s]:

s\_table[t][s] = False

**if** e\_table[s][t] **and** w\_table[t][s]:

w\_table[t][s] = False

*# If relationship is reflexive, keep the smaller building's relationship*

**for** s **in** xrange(0, num\_buildings):

**for** t **in** xrange(0, num\_buildings):

source = buildings[s]

target = buildings[t]

**if** near\_table[s][t] **and** near\_table[t][s]:

**if** source['area'] > target['area']:

near\_table[s][t] = False

**else**:

near\_table[t][s] = False

**elif** near\_table[s][t] **and** **not** near\_table[t][s]:

**print** 'Near to', source['name'], 'is', target['name'], 'but not other way around'

**return** n\_table, s\_table, e\_table, w\_table, near\_table

**def** print\_table\_info(table, buildings, direction):

*# num\_buildings = len(buildings)*

*# Track printed source indices so they are only printed once*

printed = 0

**for** s **in** xrange(0, num\_buildings):

**for** t **in** xrange(0, num\_buildings):

**if** table[s][t]:

target = buildings[t]

source = buildings[s]

**if** printed < s:

printed += 1

**if** direction **is** 'Near':

**print** 'Near to', source['name'], 'is:'

**else**:

**print** direction, 'of', source['name'], 'is:'

**print** ' ', target['name']

**def** print\_table(table,num\_buildings):

count = 0

**print** ' ',

**for** s **in** xrange(num\_buildings):

**if** s < 9:

**print** '', s+1,

**elif** s == 9:

**print** '', s+1,

**else**:

**print** s+1,

**print** ''

**for** s **in** xrange(num\_buildings):

**for** t **in** xrange(num\_buildings):

**if** t == 0:

**if** s < 9:

**print** '', s+1, '',

**else**:

**print** s+1, '',

**if** table[s][t]:

count += 1

**print** 1, '',

**else**:

**print** ' ',

**if** t == num\_buildings-1:

**print** '**\n**',

**print** 'Number of true relationships:', count

**return** count

*# ============================================================*

*# User Interface*

*# ============================================================*

*# mouse callback function*

**def** click\_event(event,x,y,flags,param):

**global** ix,iy,drawing,mode,click\_count,color,itinerary\_num,map\_campus,counter

**if** event == cv2.EVENT\_LBUTTONDOWN:

drawing = True

ix,iy = x,y

clicks.append((ix,iy))

*# counter += 1*

*# ix, iy = intercept\_click(ix,iy)*

**if** mode == True: *# User tests*

change\_color() *# and increment click count*

*# print 'iter num: ', itinerary\_num*

*# print 'counter', counter*

*# print '<len(path\_parens[itinerary\_num])', len(path\_parens[itinerary\_num])*

*# print '==len(path\_parens[itinerary\_num]-1)', len(path\_parens[itinerary\_num])-1*

*# print '>len(path\_parens[itinerary\_num])', len(path\_parens[itinerary\_num])*

**if** counter < len(path\_parens[itinerary\_num]):

**print** 'Clicked location: ({},{})'.format(ix,iy)

counter += 1

*# print 'Click count:', len(clicks)*

**if** counter == len(path\_parens[itinerary\_num])-1:

end = G\_LIST[itinerary\_num]

**print** 'Final destination:', end

clicks.append((ix,iy))

counter += 1

**print** 'Distance: ', get\_euclidean\_distance(end, clicks[-1])

cv2.circle(map\_campus,end,6,(0,0,255),-1)

itinerary\_num += 1

user\_responses.append(clicks[-1])

**print**

**print** 'Good job! Next itinerary! Click any white space to begin.'

**print** '------'

*# Save results*

cv2.imwrite('iter'+str(itinerary\_num)+'.png', map\_campus);

**elif** counter > len(path\_parens[itinerary\_num]):

*# Reset counter*

counter = 0

color = (255,255,255)

*# Reload image*

map\_campus = cv2.imread('ass3-campus.pgm', 1)

cv2.imshow('Columbia Campus Map', map\_campus)

start = S\_LIST[itinerary\_num]

*# print new start*

cv2.circle(map\_campus,start,6,(0,255,0),-1)

print\_instructions()

**else**: *# Cloud Ambiguity*

*# Function to test ALL clouds for largest/smallest*

*# test\_clouds()*

idx = create\_building(ix,iy)

change\_color() *# and increment click count*

pixels = pixel\_cloud(ix,iy) *# Generate cloud of all similar pixels*

**elif** event == cv2.EVENT\_LBUTTONUP:

drawing = False

**if** mode == True:

cv2.circle(map\_campus,(ix,iy),6,color,-1)

**else**:

cv2.circle(map\_campus,(ix,iy),pix,color,-1)

*# white dot indicates original click location*

cv2.circle(map\_campus,(ix,iy),1,(255,255,255),-1)

*# if mode == True:*

*# # cv2.rectangle(map\_campus,(ix,iy),(x,y),(0,255,0),-1)*

*# else:*

*# cv2.circle(map\_campus,(x,y),pix/2,(255,255,255),-1)*

**def** intercept\_click(ix,iy):

*"""Helper function that intercepts click values and changes to desired test"""*

**if** click\_count%2 == 0: *# Target*

ix,iy = 50,430 *# Smallest*

*# ix,iy = 90, 400 # New Largest*

**else**: *# Source*

ix,iy = 70,210 *# Largest*

*# ix,iy = 130, 340 # Second Largest*

*# ix,iy = 10, 190 # Small*

**return** ix,iy

**def** change\_color():

**global** color, click\_count

*# alternate colors based on clicks*

**if** click\_count >= len(colors)-1: *# reset*

click\_count = 0

**else**:

click\_count += 1

color = colors[click\_count]

*# ============================================================*

*# Source and Target Description*

*# ============================================================*

**def** create\_building(x,y):

**global** buildings

*# idx = int(map\_labeled[y][x])*

*# add new x,y as a new building*

idx = len(buildings)

building = {}

building['number'] = len(buildings)+1

building['name'] = 'Building ' + str(len(buildings)+1)

building['centroid'] = (x,y)

building['xywh'] = (x,y,1,1)

buildings.append(building)

*# num\_buildings = len(buildings)*

**return** idx

**def** pixel\_cloud(x,y):

**global** color, cloud, recursive\_calls, called

*# Reset cloud every time this function is called*

cloud = {}

relationships = []

recursive\_calls = 0

called = {}

*# To copy numpy arrays:*

*# a = np.zeros((27,27),bool)*

*# b = np.zeros((28,28),bool)*

*# b[:-1,:-1] = a*

*# for num in xrange(0, num\_buildings-1-click\_count):*

**for** num **in** xrange(num\_buildings):

s = buildings[num]

t = buildings[-1] *# the newly added building*

*# Note these methods require xywh, centroid, number*

idx = int(map\_labeled[y][x]) - 1

*# near = xy\_near(s,x,y)*

*# near = is\_near(s,t) # Keep smaller (1 pixel) building's relationship*

near = is\_near(s,t) **or** is\_near(t,s)

relationships.append([is\_north(s,t), is\_south(s,t), is\_east(s,t), is\_west(s,t),near,num,idx])

*# relationships.append([is\_north(s,t), is\_east(s,t), is\_near(s,t),idx])*

*# print "Relationships:", relationships*

relationships, sorted\_indices = reduce\_by\_nearness(relationships)

*# print 'New relationships:', relationships*

*# Recursively generate ambiguity cloud based on pruned relationships and sorted indices*

flood\_fill(x,y,relationships,sorted\_indices)

*# Color in the cloud*

**for** xy **in** cloud:

col = xy[0]

row = xy[1]

*# map\_campus[row][col] = [0,255,0]*

*# Draw filled circle with radius of 5*

cv2.circle(map\_campus,(col,row),pix/2,color,-1)

description = ts\_description(x,y,relationships,sorted\_indices)

**print** description

cloud\_size = len(cloud) \* pix

**print** ' Size of cloud:', cloud\_size, '(recursive calls: **%d**)**\n**' %recursive\_calls

**return** cloud\_size

**def** reduce\_by\_nearness(relationships):

*# Experiment with limit*

*# Increasing it does not shrink ambiguity by much*

*# Users seem confused by more than 3 descriptions*

limit = 3

distances\_to = {}

**for** i **in** xrange(num\_buildings):

*# Only keep near relationships*

**if** relationships[i][4] == False:

*# Change all values to False (ignore)*

relationships[i][:5] = [False,False,False,False,False]

**else**:

*# Of the remaining 'near' relationships, sort by distance*

s = relationships[i][5]

t = -1 *# Last added building to list of buildings*

dist = get\_euclidean\_distance(s,t)

distances\_to[str(s)] = dist

*# Keep relationships only with three closest structures*

sorted\_distances = sorted(distances\_to.items(), key=**lambda** k:k[1])

*# Special case: if click is inside building, its color value - 1*

*# (its building index) should be at start of list*

click\_idx = relationships[0][-1]

**if** click\_idx == -1: *# Outside*

sorted\_indices = [int(tup[0]) **for** tup **in** sorted\_distances]

**else**: *# Inside*

sorted\_indices = [int(tup[0]) **for** tup **in** sorted\_distances **if** int(tup[0]) != click\_idx]

sorted\_indices.insert(0,click\_idx)

*# If there more than three structures indicated, set rest to be ignored*

**if** len(sorted\_indices) > limit:

**for** n **in** xrange(limit,len(sorted\_indices)):

idx = sorted\_indices[n]

relationships[idx][:5] = [False,False,False,False,False]

*# Prune the list of indices to contain only the limit*

sorted\_indices = sorted\_indices[:limit]

*# print 'Sorted distances:', sorted\_distances*

*# print 'Distances:', distances\_to*

*# print 'Sorted indices:', sorted\_indices*

*# print 'New relationships:', relationships*

**return** relationships, sorted\_indices

**def** flood\_fill(x, y, rel\_table, indices):

*"""Recursive algorithm that starts at x and y and changes any*

*adjacent pixel that match rel\_table"""*

**global** cloud, called, recursive\_calls

**if** (x,y) **in** called:

**return**

**else**:

recursive\_calls += 1

called[(x,y)] = ''

*# print recursive\_calls, ':', x,y*

rel = []

*# for num in range(0, num\_buildings-1-click\_count):*

**for** num **in** xrange(num\_buildings):

s = buildings[num]

t = buildings[-1]

t['centroid'] = (x,y) *# change centroid to new x,y*

t['xywh'] = (x,y,100,100)

**if** 'near\_points' **in** t:

**del** t['near\_points']

buildings[t['number']-1] = t

idx = int(map\_labeled[y][x]) - 1

*# Only check relevant relations*

**if** num **in** tuple(indices):

*# near = xy\_near(s,x,y)*

*# near = is\_near(s,t) # Keep smaller (1 pixel) building's relationship*

near = is\_near(s,t) **or** is\_near(t,s)

**if** (near):

rel.append([is\_north(s,t), is\_south(s,t), is\_east(s,t), is\_west(s,t),near,num,idx])

**else**:

rel.append([False,False,False,False,False,num,idx])

*# Else set all values to default False*

**else**:

rel.append([False,False,False,False,False,num,idx])

*# print 'Flood Fill Rel:', rel*

*# Base case. If the current x,y is not the right rel do nothing*

**if** rel != rel\_table:

**return**

*# Add pixel to list of clouds to be recolored and used later*

cloud[(x,y)] = ''

*# Recursive calls. Make a recursive call as long as we are not*

*# on boundary*

**if** x > (pix-1): *# left # originally 0*

flood\_fill(x-pix, y, rel\_table, indices)

**if** y > (pix-1): *# up # originally 0*

flood\_fill(x, y-pix, rel\_table, indices)

**if** x < MAP\_W-(pix+1): *# right # originally MAP\_W-1*

flood\_fill(x+pix, y, rel\_table, indices)

**if** y < MAP\_H-(pix+1): *# down # originall MAP\_H-`*

flood\_fill(x, y+pix, rel\_table, indices)

**def** test\_clouds():

*"""Check clouds of every other 10 pixels in the map*

*and lists the xy coordinates sorted by cloud size"""*

clouds = []

min\_cloud = (0,0,10)

max\_cloud = (0,0,10)

**for** x **in** xrange(MAP\_W):

**for** y **in** xrange(MAP\_H):

**if** (x%10 == 0) **and** (y%10 == 0):

idx = create\_building(x,y)

*# change\_color() # don't draw*

size = pixel\_cloud(x,y)

**if** (size < min\_cloud[2]):

min\_cloud = (x,y,size)

**elif** (size > max\_cloud[2]):

max\_cloud = (x,y,size)

clouds.append((x,y,size))

sorted\_clouds = sorted(clouds, key=**lambda** k:-k[2])

**print** 'Max cloud', max\_cloud

**print** 'Min cloud', min\_cloud

**print** 'Sorted clouds', sorted\_clouds

**def** index\_valid(x,y):

x = xy[0]

y = xy[1]

**if** (x > 0) **and** (x < MAP\_W) **and** (y > 0) **and** (y < MAP\_H):

**return** True

**else**:

**return** False

**def** what\_description(idx):

**global** buildings

what = 'the '

descr = buildings[idx]['description']

**for** i **in** xrange(len(descr)):

**if** i < len(descr)-1:

what += descr[i] + ', '

**else**:

what += descr[i] + ' structure'

**return** what

**def** ts\_description(x, y, relationships, sorted\_indices):

coordinates = ' Click (**%d**,**%d**)' %(x,y)

**if** click\_count%2 == 1:

**print** 'TARGET: ' *#+ coordinates*

*# description = 'Then go to the building that is '*

**else**:

**print** 'SOURCE: ' *#+ coordinates*

*# description = 'Go to the nearby building that is '*

*# Check if click point is outside or inside*

**if** (relationships[0][-1] == -1):

description = coordinates + ' is '

**else**:

description = coordinates + ' is INSIDE and to the '

*# print 'Sorted indices:', sorted\_indices*

*# print 'Relationships:', relationships*

*# for idx in range(0, num\_buildings-1):*

rel\_count = 0

**for** idx **in** sorted\_indices:

count = 0

**if** relationships[idx][0]:

description += 'NORTH of '

count += 1

**if** relationships[idx][1]:

description += 'SOUTH of '

count += 1

**if** relationships[idx][2]:

**if** count == 0:

count += 1

**else**:

description = description[:-4]

description += 'EAST of '

**if** relationships[idx][3]:

**if** count == 0:

count += 1

**else**:

description = description[:-4]

description += 'WEST of '

*# Implied nearness*

*# if relationships[idx][4]:*

*# if count == 0:*

*# descr += "near "*

*# count += 1*

*# else:*

*# desc += "and near "*

**if** count != 0:

description += what\_description(idx)

description += ' (**%s**), ' %buildings[idx]['name']

rel\_count += 1

**if** sorted\_indices[1] == -1: *# Only one descriptor*

**break**

**if** sorted\_indices[0] == -1 **and** rel\_count == len(sorted\_indices)-2:

description += 'and '

**elif** sorted\_indices[0] != -1 **and** rel\_count == len(sorted\_indices)-1:

description += 'and '

description = description[:-2] + '.'

**return** description

*# ============================================================*

*# Path Generation*

*# ============================================================*

**def** generate\_graph():

graph = {}

**for** s **in** xrange(0, num\_buildings):

distances = {}

**for** t **in** xrange(0, num\_buildings):

near = is\_near(s,t) **or** is\_near(t,s)

*# Only generate paths between near nodes*

**if** s != t **and** near:

distances[str(t)] = get\_euclidean\_distance(s,t)

graph[str(s)] = distances

*# print dist\_table*

**return** graph

**def** generate\_paths(graph):

**global** paths, S\_LIST, G\_LIST, buildings, path\_parens, path\_no\_parens

starting\_points = []

starting\_indices = []

terminal\_points = []

*# path\_descriptions = []*

path\_ends = []

*# Find description for starting point*

**for** xy **in** S\_LIST:

start = find\_closest(xy)

starting\_points.append(start)

idx = create\_building(xy[0],xy[1])

text = first\_step(idx,start,True)

path\_parens.append([text])

text = first\_step(idx,start,False)

path\_no\_parens.append([text])

buildings.pop()

**for** xy **in** G\_LIST:

end = find\_closest(xy)

terminal\_points.append(end)

idx = create\_building(xy[0],xy[1])

description = terminal\_guidance(idx,start)

path\_ends.append(description)

buildings.pop()

*# print "Starting points", starting\_points*

*# print "Terminal points", terminal\_points*

*# print "Graph", graph*

**for** i **in** xrange(len(starting\_points)):

start = starting\_points[i]

end = terminal\_points[i]

*# Convert ints because graph keys are strings*

dijkstra(graph, str(start), str(end),[],{},{})

*# print "\nGraph", graph*

*# Example: [[22, 19, 12, 8, 4, 0]] len: 6*

**for** i **in** xrange(len(paths)):

path = paths[i]

**for** j **in** xrange(len(path)-1):

s = path[j]

t = path[j+1]

text = step\_guidance(s,t,True) *# True)*

path\_parens[i].append(text)

text = step\_guidance(s,t,False)

path\_no\_parens[i].append(text)

path\_parens[i].append(path\_ends[i])

path\_no\_parens[i].append(path\_ends[i])

**print** 'Paths', paths

*# print 'Paths (parens):', path\_parens*

*# print 'Paths (no parens):', path\_no\_parens*

*# print 'Path endings:', path\_ends*

*# dijkstra(graph,'0','22')*

**def** get\_euclidean\_distance(source,target):

*"""Find the euclidean distance between two points*

*Based on an old Java program of mine:*

*private double getEuclideanDistance(Vertex v1, Vertex v2) {*

*double base = Math.abs(v1.x - v2.x); // x1 - x2*

*double height = Math.abs(v1.y - v2.y); // y1 - y2*

*double hypotenuse = Math*

*.sqrt((Math.pow(base, 2) + (Math.pow(height, 2))));*

*return hypotenuse;*

*"""*

*# Take min(w,h) of source building into account*

*# margin = (min(s['xywh'][2],s['xywh'][3])/2)*

**if** (type(source) == int):

*# Get building from indices*

s = buildings[source]

x1 = s['centroid'][0]

y1 = s['centroid'][1]

**else**:

x1 = source[0]

y1 = source[1]

**if** (type(target) == int):

t = buildings[target]

x2 = t['centroid'][0]

y2 = t['centroid'][1]

**else**:

x2 = target[0]

y2 = target[1]

base = abs(x1-x2)

height = abs(y1-y2)

hypotenuse = math.sqrt(math.pow(base,2)+(math.pow(height,2)))

**return** hypotenuse

**def** find\_closest(xy):

x = xy[0]

y = xy[1]

building\_idx = int(map\_labeled[y][x])-1

**if** building\_idx **is** **not** -1:

**return** building\_idx

**else**:

distances = np.zeros(num\_buildings)

**for** i **in** xrange(num\_buildings):

distances[i] = get\_euclidean\_distance(xy,i)

**return** distances.argmin()

**def** is\_inside(idx):

building = buildings[idx]

x = building['centroid'][0]

y = building['centroid'][1]

pixel = int(map\_labeled[y][x])-1

**if** pixel **is** -1:

**return** False

**else**:

**return** True

**def** dijkstra(graph,src,dest,visited=[],distances={},predecessors={}):

*"""Calculates a shortest path tree routed in src. Based on this tutorial:*

*http://geekly-yours.blogspot.com/2014/03/dijkstra-algorithm-python-example-source-code-shortest-path.html*

*I could have converted my Dijkstra program in Java into Python, but sorted\_indices*

*path-finding is not the emphasis of this assignment, I decided to spend more time*

*on the visual analysis component"""*

**global** paths

*# a few sanity checks*

**if** src **not** **in** graph:

**raise** **TypeError**(src, ': the root of the shortest path tree cannot be found in the graph')

**if** dest **not** **in** graph:

**raise** **TypeError**(dest, ': the target of the shortest path cannot be found in the graph')

*# ending condition*

**if** src == dest:

*# We build the shortest path and display it*

path=[]

pred=dest

**while** pred != None:

path.append(int(pred))

pred=predecessors.get(pred,None)

**if** path:

*# print('Shortest Path: '+str(path)+" (Cost: "+str(distances[dest])+')')*

correct\_order = []

**for** item **in** reversed(path):

correct\_order.append(item)

paths.append(correct\_order)

*# print paths*

**else**:

*# if it is the initial run, initializes the cost*

**if** **not** visited:

distances[src]=0

*# visit the neighbors*

**for** neighbor **in** graph[src] :

**if** neighbor **not** **in** visited:

new\_distance = distances[src] + graph[src][neighbor]

**if** new\_distance < distances.get(neighbor,float('inf')):

distances[neighbor] = new\_distance

predecessors[neighbor] = src

*# mark as visited*

visited.append(src)

*# now that all neighbors have been visited: recurse*

*# select the non visited node with lowest distance 'x'*

*# run Dijskstra with src='x'*

unvisited={}

**for** k **in** graph:

**if** k **not** **in** visited:

unvisited[k] = distances.get(k,float('inf'))

x=min(unvisited, key=unvisited.get)

dijkstra(graph,x,dest,visited,distances,predecessors)

**def** first\_step(start,target,parens,name=False):

*"""'Go to the building that is east and near (which is cross-shaped).*

*"""*

inside = is\_inside(start)

**if** inside:

text = 'You are inside a building'

**if** parens:

text += ' (**%s**)' %what\_description(target)

**if** name:

text += ' <**%s**>' %buildings[target]['name']

text += ' to the '

**else**:

text = 'You are outside. Go to the nearby building that is '

s = buildings[start]

t = buildings[target]

**if** is\_north(s,t): *# north of s is t*

**if** inside:

text += 'SOUTH'

**else**:

text += 'NORTH'

**elif** is\_south(s,t):

**if** inside:

text += 'NORTH'

**else**:

text += 'SOUTH'

**if** is\_east(s,t):

**if** inside:

text += 'WEST'

**else**:

text += 'EAST'

**elif** is\_west(s,t):

**if** inside:

text += 'EAST'

**else**:

text += 'WEST'

**if** **not** inside:

**if** parens:

text += ' (**%s**)' %what\_description(target)

**if** name:

text += ' <**%s**>' %buildings[target]['name']

text += '.'

*# print 'EAST:', is\_east(s,t), e\_table[s][t]*

*# print 'WEST:', is\_west(s,t), e\_table[t][s], w\_table[s][t]*

*# print text*

**return** text

**def** step\_guidance(s,t,parens,name=False):

*"""'Go to the building that is east and near (which is cross-shaped).*

*Then go to the building that is north (which is oriented east-to-west).*

*Then go to the building that is north and east (which is medium-sized and oriented north-to-south)*

*"""*

text = 'Now go to the nearby building that is '

count = 0

**if** n\_table[s][t]: *# north of s is t*

text += 'NORTH'

count += 1

**elif** n\_table[t][s] **or** s\_table[t][s]:

text += 'SOUTH'

count += 1

**if** e\_table[s][t]:

**if** count == 0:

text += 'EAST'

count += 1

**else**:

text == ' and EAST'

**elif** e\_table[t][s] **or** w\_table[s][t]:

**if** count == 0:

text += 'WEST'

count += 1

**else**:

text += ' and WEST'

*# if count == 0:*

*# if is\_north(s,t):*

*# text += 'north'*

**if** count != 0:

**if** parens:

text += ' (**%s**)' %what\_description(t)

**if** name:

text += ' <**%s**>' %buildings[t]['name']

text += '.'

*# print 'EAST:', is\_east(s,t), e\_table[s][t]*

*# print 'WEST:', is\_west(s,t), e\_table[t][s], w\_table[s][t]*

*# print text*

**return** text

**def** terminal\_guidance(start,target):

**if** is\_inside(target):

text = 'Your final destination is inside this building. Go '

**else**:

text = 'Your final destination is outside near this building. Go '

s = buildings[start]

t = buildings[target]

count = 0

**if** is\_north(s,t): *# north of s is t*

text += 'NORTH'

count += 1

**elif** is\_south(s,t):

text += 'SOUTH'

count += 1

**if** is\_east(s,t):

**if** count == 0:

text += 'EAST'

**else**:

text == ' and EAST'

**elif** is\_west(s,t):

**if** count == 0:

text += 'WEST'

**else**:

text += ' and WEST'

**if** is\_inside(target):

text += ' within the building'

text += '.'

*# print 'EAST:', is\_east(s,t), e\_table[s][t]*

*# print 'WEST:', is\_west(s,t), e\_table[t][s], w\_table[s][t]*

*# print text*

**return** text

**def** print\_instructions(parens\_first=True):

**global** path\_parens, path\_no\_parens

**if** parens\_first:

firsthalf = path\_parens

secondhalf = path\_no\_parens

**else**:

firsthalf = path\_no\_parens

secondhalf = path\_parens

**print** '**\n**ITINERARY ' + str(itinerary\_num+1)

**print** '------'

**if** itinerary\_num < 4:

itinerary = firsthalf[itinerary\_num]

**for** step **in** itinerary:

**print** step

**print** '------'

**else**:

itinerary = secondhalf[itinerary\_num]

**for** step **in** itinerary:

**print** step

**print** '------'

**def** print\_all\_instructions(parens\_first=False):

**global** path\_parens, path\_no\_parens

**if** parens\_first:

firsthalf = path\_parens

secondhalf = path\_no\_parens

**else**:

firsthalf = path\_no\_parens

secondhalf = path\_parens

**for** i **in** xrange(4):

**print** '**\n**ITINERARY ' + str(i+1)

**print** '------'

itinerary = firsthalf[i]

**for** step **in** itinerary:

**print** step

**print** '------'

**for** i **in** xrange(4):

**print** '**\n**ITINERARY ' + str(i+5)

**print** '------'

itinerary = secondhalf[i+4]

**for** step **in** itinerary:

**print** step

**print** '------'

*# ============================================================*

*# Main Invocation*

*# ============================================================*

**def** main():

**global** buildings, mode

*# Step 1. Generate 'what' for each building by analyzing image*

*# Note: images and buildings information are stored as global vars*

building\_names = load\_names('ass3-table.txt')

analyze\_what(building\_names)

print\_info()

*# Step 2. Generate 'where' lookup table for building relations*

analyze\_where(buildings)

*# Step 4. Generate path for user*

graph = generate\_graph()

generate\_paths(graph)

*# print\_all\_instructions(parens\_first=False)*

*# for path in path\_parens:*

*# print len(path)*

*# Step 3. Source and Target Description and User Interface set up in click event*

cv2.namedWindow('Columbia Campus Map')

cv2.setMouseCallback('Columbia Campus Map', click\_event)

**print** "**\n**Showing image...**\n**"

*# Step 4. Show first itinerary*

**print** '**\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n**'

start = S\_LIST[itinerary\_num]

cv2.circle(map\_campus,start,6,(0,255,0),-1)

print\_instructions()

**while**(1):

cv2.imshow('Columbia Campus Map', map\_campus)

k = cv2.waitKey(1) & 0xFF

**if** k == ord('m'):

mode = **not** mode

**if** mode:

modeval = 'Path Generation'

**else**:

modeval = 'Cloud Generation'

**print** 'Changing mode to', modeval,'(you pressed m)...**\n**'

**if** k == 27:

**break**

cv2.destroyAllWindows()

**if** \_\_name\_\_ == "\_\_main\_\_": main()