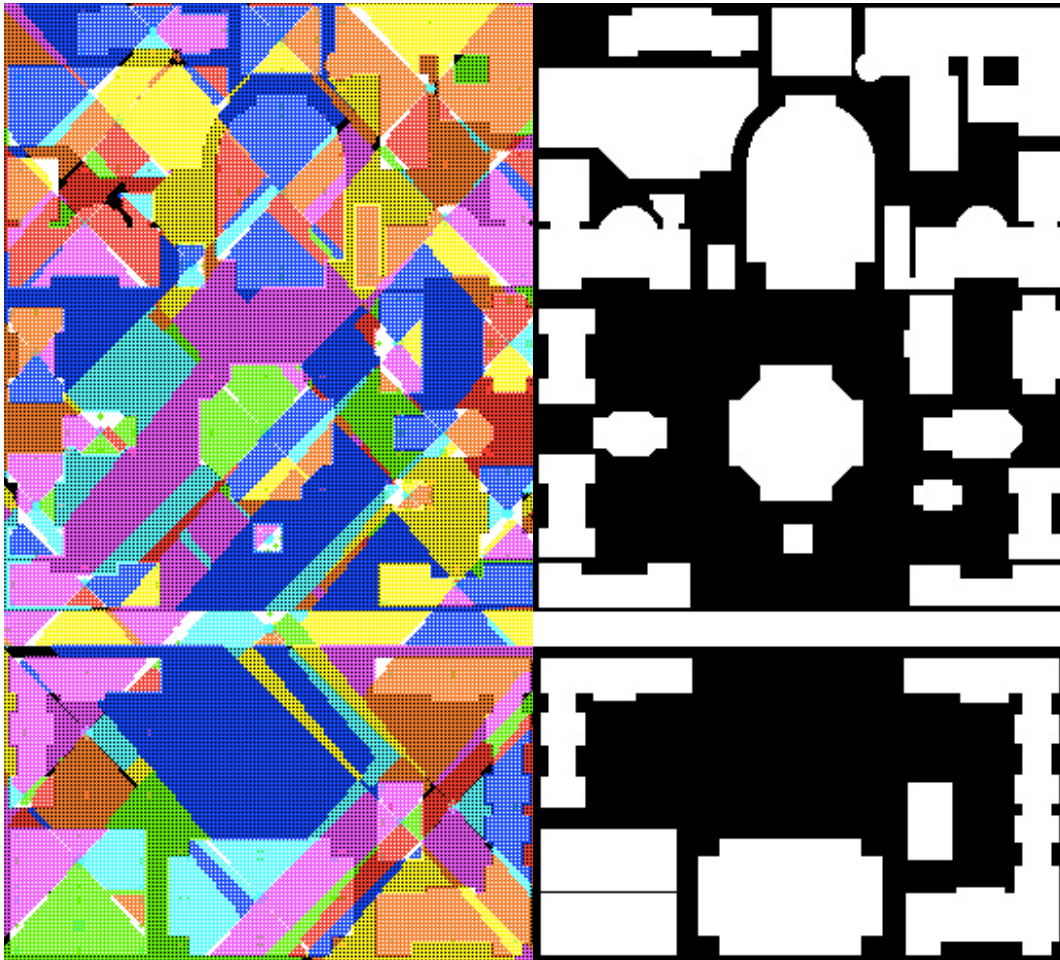


DESCRIPTION OF VISUAL RELATIONS



COMS W4735: VISUAL INTERFACES TO COMPUTERS

Assignment 3
due: 4/9/2015
Nina Baculinao
uni: nb2406

Introduction

The goal of the Description of Visual Relations system is to explore how well a visual image can be described in human language terms. The program takes as input a 2D image of the Columbia campus plus a "source" and a "target" location, and then produces as output non-numeric descriptions of these locations (the "what") and non-numeric directions to follow ("the where") from the source to the target.

Information to aid the construction of the program was provided by the following files:

- "ass3-campus.pgm": a binary image of the main campus as seen from above, where a large number (white) represents the buildings and zeros represent the space between them
- "ass3-labeled.pgm": an integer-valued image based on the first, in which each building is given an encoded integer, and all the pixels in the same building are encoded with the same integer
- "ass3-table.txt": a text file which translates the encoded integer into a string

Hardware and library specifications of the development environment are as follows:

- MacBook Air 11 inch running OSX Yosemite
- Python Standard Library 2.7.9
- OpenCV with a Python binding, v2.4.10.1
- NumPy 1.9.1

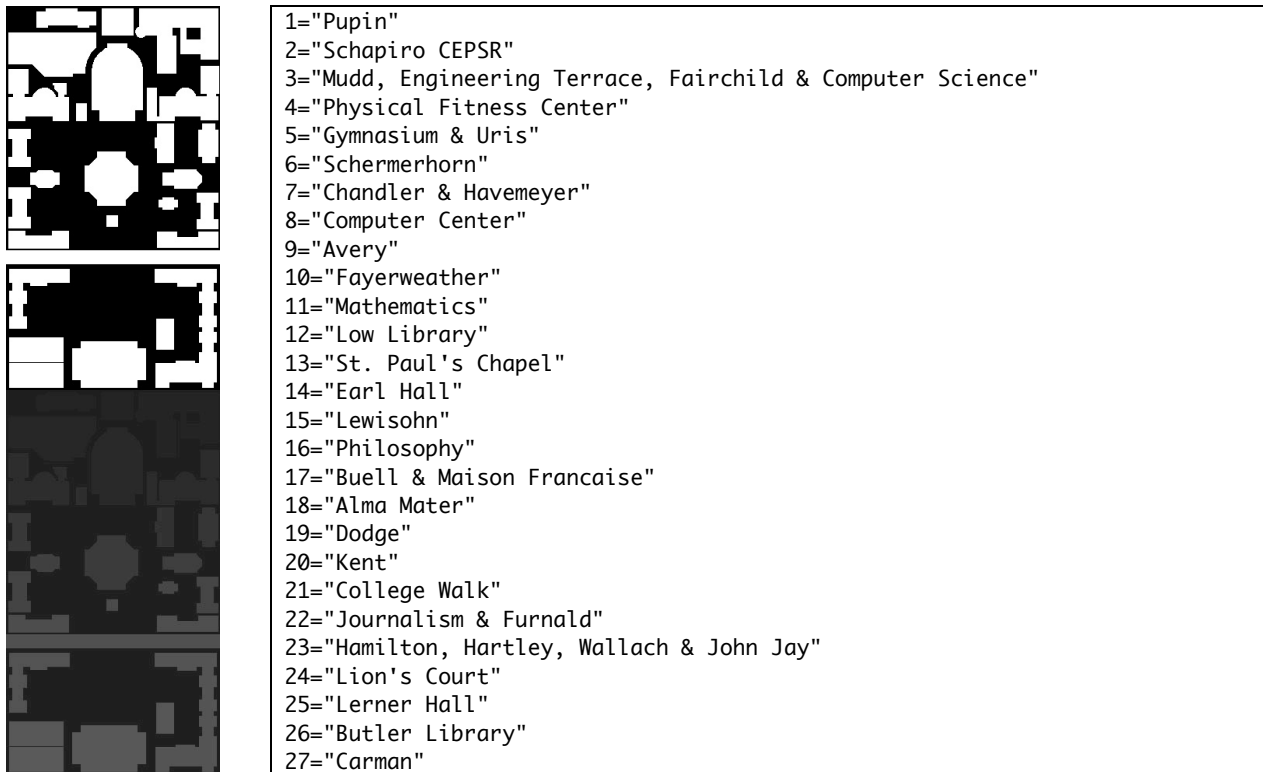


Figure 1. Top to bottom, left to right. Binary map of Columbia campus, grayscale campus map where pixel values encode building keys (brightened here), and text listing of building keys and name values

First part: describe how you determined each feature that you used. Give examples.

Second part: talk about how you defined "near" and about how you pruned. Output all the surviving relations.

Third part: talk about your clouds, and pictures of the largest/smallest clouds, show clouds for three interesting paths. Parentheses for describing them. Justify how you picked the paths.

Fourth part: talk about which search algorithm used, talk about ambiguity - picking between paths, output the experiments

For all steps, your writeup explains: what design choices you made and why, what algorithms you used, what you observed in the output, and how well you believe your design worked.

For step 1, you must show your computer vision processing of buildings into quantitative location and shape numbers. You must also show your choice of descriptive categories, and the algorithms that translate the numbers into descriptions.

For step 2, you must describe each of the 27 buildings with their pruned spatial relationships.

For step 3, you must show the S and T "clouds" for three interesting paths, and the set of directions, including the parentheses, that describe them. Show also the most confused place (biggest cloud) and least confused place (smallest cloud) that your system generates.

For step 4, you must record 16 experiments, display the accuracy of their results, and comment on how the different kinds of descriptions were or were not helpful.

Vocabulary of Shapes: The "What"

Overview

The first part of this system automatically computes features and descriptions for each building from the visual properties of the given data. In this subsection I attempt to give a broad outline of what this “front-end” vision processing entails, before delving into a fuller-fledged explanation of my choices of descriptive categories, and what computer vision algorithms I use to process the buildings into measurable location and shape numbers, then translate that quantitative data into qualitative features.

1. **<Given Data>** Read in building names and ID numbers from text file
 - a. Now we know the number of buildings in the map and the size of our buildings array (27)
2. Determine area of every building by taking advantage of the labeled map’s special properties and tallying the pixels for every building number using an efficient counting dictionary
3. Binarize the campus map and use OpenCV to find contours
4. **<Quantitative Processing>** For each contour (one for each building):
 - a. Identify which building the contour represents on the campus map by sampling the pixel values of the contour mask in the labeled map
 - b. Find the minimum bounding rectangle (mbr) and centroid
 - c. Store building as dictionary object with the following key-value pairs:
 - i. **Essential: number, name, area, mbr, centroid (cx, cy)**
 - ii. For convenience: extent, xywh tuple
 - d. Add the dictionary object to the buildings list index that matches its pixel ID number
5. **<Individual Analysis>** Analyze areas, extents and shapes of each building relative to one another
 - a. In order to determine “magic numbers” and “thresholds” for descriptive categories
 - b. This is performed by extracting these details into sorted list
 - c. Extract minimum area and maximum area
 - d. Identify College Walk as a useful monument for demarcating location
6. **<Qualitative Processing>** For each building in our list:
 - a. Describe its size
 - b. Describe its location
 - c. Describe its shape
 - d. Combine descriptions in a list and add **description** key-value pair to building dictionary
7. **<Minor Optimization>** Reduce and clarify descriptions
 - a. Find extrema descriptions (e.g. “smallest” structure no longer needs “tiny” as descriptor)
 - b. Find ambiguous descriptions (e.g. Mathematics and Lewisohn had identical lists)
8. **<Deliverable>** Print out resulting information

Summary of Chosen Descriptions

Size (area/max area)	Tiny, Small, Medium, Large, Colossal
Shape (midpoints, corners)	Geometric: Rectangle, Square, Cross, Bell, Irregular Alphabetic: Serif I-shaped, C-shaped, L-shaped, T-shaped
Orientation	N-S if narrow, E-W if long
Location	Upper, lower, central campus Northernmost, southernmost, easternmost, westernmost Northeast corner, Southeast corner, Southwest Corners
Extrema	Smallest, Largest, Longest

Basic infrastructure

As mentioned, the first thing to do was set up the basic infrastructure by loading the provided data. Because the maps were widely used in almost every single function, with many functions testing for success by drawing on the image display of map campus, I decided to make these global variables to avoid passing them in out. I originally had the list of buildings passed in the functions as a parameter, but this proved to be quite unwieldy especially when designing the mouse callback functions and the user interface, so I made it a global variables too.

```
# 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 = {}
```

Building names

The building names were provided to us in a text file, so I parsed each line and stored the names in a list to make translation of the encoded pixel integers into building names easy. Since all the building numbers were listed in order, I just passed in the name, as a building number/ID correlated with its list index + 1. E.g. 1="Pupin" --> names[0] = "Pupin"

```
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
```

Areas and identification from pixel values

As the “labeled” map was created explicitly to make identification easy – to find Building N we only had to scan the image for the occurrences of the encoded integer N. To find the area for each building, all I had to do was traverse every pixel in the image once and tally the occurrences of each N.

```
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
```

Previously I had used `cv2.contourArea(cnt)` and `cv2.moments(cnt)['m00']` just to find the area, but these were a) approximations of the binarized campus map using the Green formula and b) overly complicated considering we had a labeled image whose pixels correlated exactly with building size.

```
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
```

Since I still found `cv2.findContours` method a powerful way to quickly identify the various structures in the map, so I used contours to swiftly pinpoint the building locations, and then performed more detailed and less “approximate” analysis on those regions. What this OpenCV library function does is retrieve contours from the binary image using the Suzuki 1985 algorithm for detecting structures in a binary image by border following. The contours are returned as a list of vectors points, and are a useful tool for shape analysis and object detection and recognition. I use the cheaper mode of retrieval `CHAIN_APPROX_SIMPLE`, which essentially compresses line segments and leaves only their endpoints.

Once I have a contour, I identify the building. My method is a bit bloated, but it works. Essentially, what it does is take the contour, create a mask of those contour points, and then find the color mean average in that masked region. While I use the binary map (that I don’t care about altering) for drawing the contour mask, I end up actually querying the color pixels of the labeled image, since it has the same dimensions as

the binarized campus mask. I receive a color mean back as a float, because the contour as mention is an approximation that also samples some black pixels, so not all the integer values are exact (they are very close though, like 16.00001 vs. 16 flat). The one exception is Mudd, which has a black hole in it whose color values enter the color mean calculation, so the value is offset only slightly more from the original integer value of 3.

```
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
```

This main benefit of this function is that it allows me to discount the hole in Mudd. When OpenCV identified this as a contour, I was able to set a threshold and say that any contour with a mean color of less than 0.9 was not a building. Size would have been a little difficult as not-a-building condition, because the courtyard looks to my human eyes similar in size to Alma Mater. Moreover, identifying buildings by the pixel value of their centroid might have been problematic – while I could take the centroid of the Mudd hole and get back a 0 value to indicate this is not a building, buildings like the L-shaped Journalism and Furnald building or C-shaped Hartley, Wallach, JJ and Hamilton block would give back false negatives since their center of mass is empty, as you can see in the figure below.

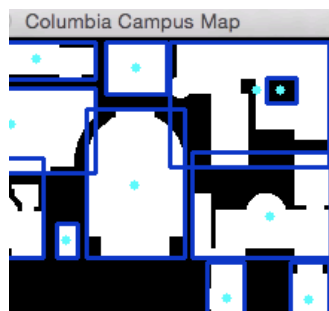
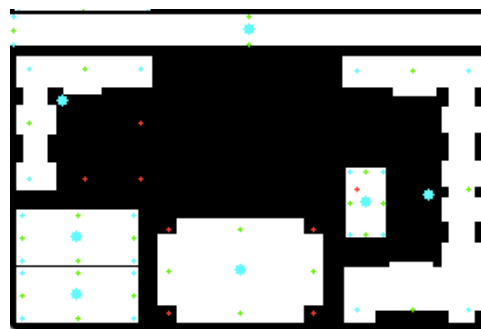


Figure 2. Contours and centroids in empty pixel land.

Left: courtyard mis-identified as its own building structure

Right: buildings where centroid pixel does not represent its building number



Finding mbr and centroid

After identifying the building the contour represents, I try to extract some essential measurements. First, I use `cv2.boundingRect(cnt)` to find the minimum x and y values, and the width and height of the bounding rectangle. I could have used the minimum bounding rectangle function but all the buildings in the map look to be at strict angles, so an upright bounding rectangular was effective enough.

```
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)

    # 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

    rect_area = w*h
    extent = float(area)/rect_area

    return mbr, centroid, extent, xywh
```

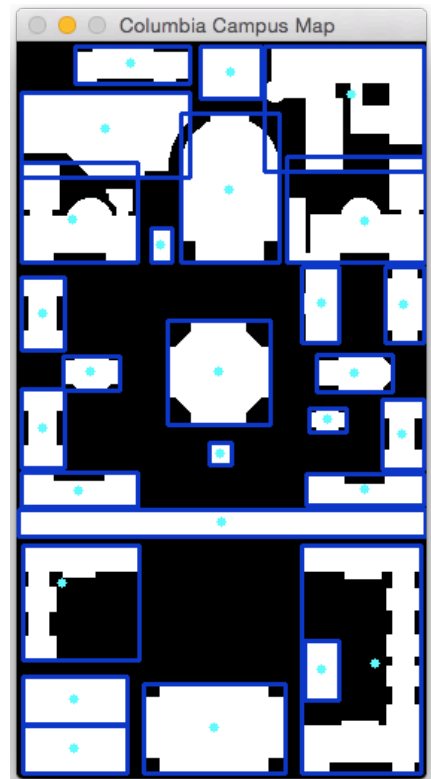


Figure 3. Drawn mbr and centroids.

To get the center of mass, rather than using image moments, I just used the bounding rectangle's dimensions to get the center point of the rectangle and the center of mass of the building (dividing the width and height by 2 and then adding them to their respective top left x,y coordinate to get the middle point coordinates). I also draw the bounding rectangles and centroids on the labeled map so I can verify that the results looked properly calculated by my human eyes. The results are shown above.

Besides the essential centroid and mbr requirements, I also return the extent (area/rect_area where rect_area is calculated by bounding box $w * h$) while thinking ahead about how I would be able to distinguish rectangles (higher extents) from other shapes (lower extents the more black space or less filled the bounding box is). I also included (x,y,w,h) as a tuple – while these values could be retrieved from the ((x,y),(x+w,y+h)) values from the mbr, this was more straightforward.

During this part, I also assessed and discarded a number of OpenCV functions for over complication and low accuracy. While `cv2.moments()` calculates moment values and is useful for center of mass and object area, again this is just an approximation. As mentioned already, their area of the contour is also an approximation. I tried using convex hull and defects, similar to what I successfully did with for recognizing hand gestures in assignment 1, but the return values were too inconsistent. I also experimented with the `cv2.cornerHarris` method for detecting corners but the results were very inconsistent and I discarded that method as well.



Figure 4. Harris Corner method with undesirable results

Analyzing sizes and extents

After extracting all the required “quantitative” measurements from the buildings, I had to analyze the results in order to determine a way to automate the system to product “qualitative” descriptions based on the numbers in the image array. I wrote the two following methods in order to analyze the building sizes relative to each other, and their extents, to see what information might be useful. The data is printed out in tabulated columns to system output, and I copied and pasted the output into it to a CSV file so I could color code and make connections.

```
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\tRatio \tDiff \tMax \tBuilding'
        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
```

In the building areas table on the next page, all the information is automatically printed out by my program except for the last two columns which contain my own human judgments of which buildings should go into which size category. The first column contains the plain Area for each building (i.e. the pixel counts). I end up using the values in the second column Ratio r (area/max_area) as cutoffs for the different size categories, and the different groups are separated by color. I tried to avoid hard coding these constraints and to find a way to algorithmically define where a grouping should break off by calculating values such as diff r, the difference between $r[i]$ and $r[i-1]$ (third column, yellow highlights show bigger differences) but the numbers didn’t point at any magical cutoff marks. Max r (max_area/area) was just the inverse of Ratio. In the fifth column, the colored building pairings in the fifth column represent very similarly sized buildings that should not be put into different size groups.

Analyzing building areas...

Max Area: 5855 Average: 2491 Min Area: 225

Area	Ratio r	Diff r	Max r	Building	Added Info	
5855	1	0	1	23 Hamilton, Hartley, Wallach & John Jay	Huge	Biggest
5831	0.996	-0.004	1.047	3 Mudd, Engineering Terrace, Fairchild & Computer Science	Huge	
5753	0.983	-0.013	1.061	5 Gymnasium & Uris	Huge	
5368	0.917	-0.066	1.126	4 Physical Fitness Center	Huge	
5282	0.902	-0.015	1.144	26 Butler Library	Huge	
4950	0.845	-0.057	1.269	21 College Walk	Huge	
3911	0.668	-0.177	1.552	6 Schermerhorn	Large	
3898	0.666	-0.002	1.588	12 Low Library	Large	
3613	0.617	-0.049	1.712	7 Chandler & Havemeyer	Large	
2615	0.447	-0.17	2.402	22 Journalism & Fernald	Large	
2240	0.383	-0.064	2.744	25 Lerner Hall	Medium	
2240	0.383	0	2.744	27 Carman	Medium	
1640	0.28	-0.103	3.815	1 Pupin	Medium	
1590	0.272	-0.008	3.953	19 Dodge	Medium	
1470	0.251	-0.021	4.3	20 Kent	Medium	
1435	0.245	-0.006	4.316	2 Schapiro CEPSP	Medium	
1307	0.223	-0.022	4.819	15 Lewisohn	Medium	
1191	0.203	-0.02	5.298	11 Mathematics	Medium	
1182	0.202	-0.001	5.307	10 Fayerweather	Medium	
1164	0.199	-0.003	5.385	9 Avery	Medium	
1087	0.186	-0.013	5.758	13 St. Paul's Chapel	Medium	
1085	0.185	-0.001	5.858	16 Philosophy	Medium	
920	0.157	-0.028	6.841	24 Lion's Court	Small	
759	0.13	-0.027	8.314	14 Earl Hall	Small	
340	0.058	-0.072	19.437	17 Buell & Maison Francaise	Tiny	
322	0.055	-0.003	20.524	8 Computer Center	Tiny	
225	0.038	-0.017	29.949	18 Alma Mater	Tiny	Smallest

Based on the building data I organized on the next page, I decided to use the following categories for size:

Ratio (area/max_area)	Size Category	Smallest Structure in Category (the Cutoff)
0.7 – 1	Colossal	College Walk
0.4 – 0.7	Large	Journalism and Fernald
0.16 – 0.4	Medium	Philosophy
0.1 – 0.16	Small	Earl Hall
0 – 0.1	Tiny	Alma Mater

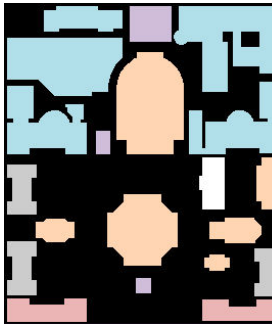
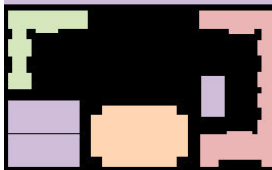
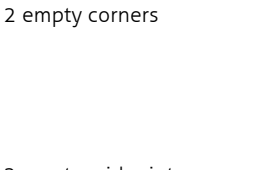
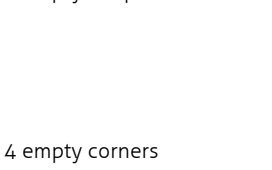
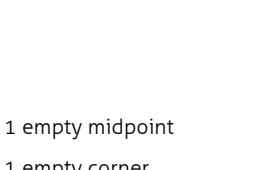
Analyzing shapes

Next, I used extents to analyze the shapes, with the initial thought that the first and easiest distinction to make was which buildings were rectangular vs. which buildings were more special shapes.

```
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']
```

While it is true that the top results that had the highest extents were mostly rectangular, there was a deceptively high exception – Butler Library, which was more cross shaped. I also tested OpenCV's `isContourConvex` based on the contour, which gave very accurate results for whether a building was rectangular or not, but I was not sure how this method worked because of a lack of documentation about it, and it also was not terribly useful to me to only be able to distinguish between rectangles and non-rectangles. The last two columns contain observations manually entered by me. First, I looked at the map and heuristically created shape categories, which I grouped buildings by (so many crosses!). In the last column (below the picture), I begin noticing that the shapes have certain corner/midpoint characteristics.

Analyzing building extents and convexity...

Extent	Convex	Building	Added Info		
0.9549	True	25 Lerner Hall	Rectangle		
0.9549	True	27 Carman	Rectangle	Square	
0.9477	True	2 Schapiro CEPsR	Rectangle		
0.941	True	21 College Walk	Rectangle		
0.9326	True	24 Lion's Court	Rectangle		
0.9	False	26 Butler Library	Symmetrical	Cross	
0.8882	True	8 Computer Center	Rectangle		
0.8711	True	18 Alma Mater	Near Rectangle		
0.8549	False	9 Avery	Irregular	Cross	
0.8494	False	5 Gymnasium & Uris	Symmetrical	Cross	
0.8356	False	10 Fayerweather	Symmetrical	C	
0.8278	False	19 Dodge	Symmetrical	C	
0.8228	False	20 Kent	Symmetrical	C	
0.8096	False	4 Physical Fitness Center	Irregular		
0.8078	False	14 Earl Hall	Irregular	Cross	
0.7996	False	13 St. Paul's Chapel	Symmetrical	Cross	
0.7992	False	1 Pupin	Irregular		
0.7925	False	15 Lewisohn	Symmetrical	H	
0.783	False	12 Low Library	Symmetrical	Cross	
0.7783	False	11 Mathematics	Symmetrical	H	
0.7614	False	16 Philosophy	Symmetrical	H	
0.755	False	17 Buell & Maison Francaise	Symmetrical	Cross	
0.6561	False	7 Chandler & Havemeyer	Irregular		
0.6531	False	3 Mudd	Irregular		
0.5597	False	6 Schermerhorn	Irregular		
0.452	False	23 Hamilton, H,W & JJ	Irregular	C	1 empty midpoint
0.4069	False	22 Journalism & Fernald	Irregular	L	1 empty corner

Choosing extrema and monument

From my area analysis, I already returned the max_area for the ratios as well as the min_area since I was sorting areas by size anyway. Even though the lower right structure is the largest one on the map, it is not obviously so and almost equal in size to Mudd, so I decided this was an ambiguous case and not to use “Biggest” as an extrema in my descriptions. However, it is quite easy to agree that Alma Mater is very small, so I decided to use that as an extrema. I also picked College Walk as not only the “longest” extremum, but also an important monument marker that spans the width of the entire campus and looks like a dividing line between lower campus, and upper campus. Indeed, while it is not apparent from this bird’s eye view, everything north of College Walk is elevated. In my mind, a useful building description should include three main facets: size, shape and location relative to the whole campus. College Walk as a monument would be very helpful to dividing this extremely narrow campus into three vertical layers since it divides the bottom third from the top two thirds.

```
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
```

Describing size

I have explained where I got my “magic numbers” and threshold ratios already in my analysis of areas. Here is the simple implementation to calculate describe size. It returns a string, which is added to a building dictionary’s description list.

```
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'
```

Describing shape

Describing the shape, however, is not as easy as the size. The way I approach it is to take the x,y,w,h points of a building, tuck them in closer to the centroid by a certain tolerance (calculated by as a ratio of min(w,h)/10), and count the number of corner points and midpoints that have the right pixel color (if it’s 0, that part of the building is empty). I use this shift or tolerance amount because for example, if I were to simply use the top left and bottom right corner of the bounding box for the Hamilton-Hartley building block, I would get back all 0’s because of its irregular shape.

To determine all the midpoints and corner points, I begin with the starting point of of the top right (x,y) and bottom left (x+w,y+h) pixel values of the mbr for each building. From there, I can calculate all the midpoints and corner points for every single building. On the right is a diagram that sums up all the calculations for each point as performed in my count_points() method.

The relevant functions used to determine shape are included below, while the relatively accurate pictorial results (green dots mean this part of the building is filled, while red dots represent empty corners/midpoints) is also shown in the right (aqua dots are still the centroid). The midpoints and corner counts are based on the observations I made during the extents and shape analysis, and the results are individually verified. The criteria is highlighted below.

```
def describe_shape(building, draw_points=False):
    """Describe shape based on corner and midpoint counts"""

    descriptions = []

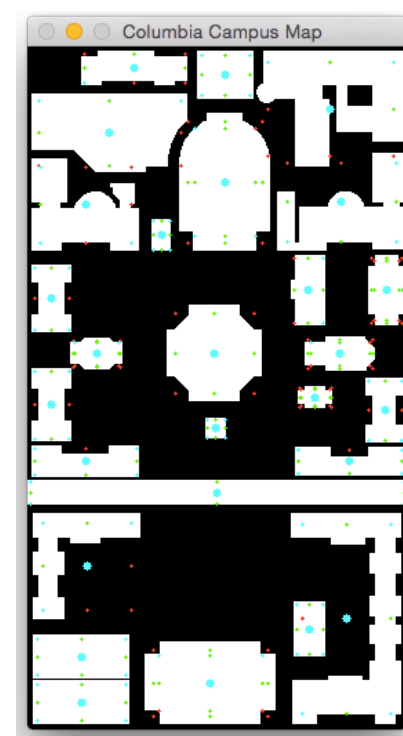
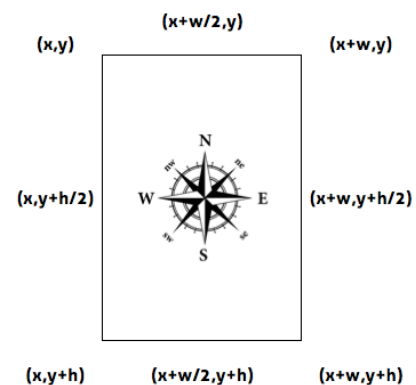
    xywh = building['xywh']
    corners_count, midpoints_count, xywh2 =
count_points(building, xywh, draw_points)

    # 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 difference 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
    # (check for bellshape: extra tolerance gives uneven corner count)
    # I should have all corners but only 2 midpoints
    # U 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')
```



1 Pupin
Tolerance 2
[0, 0, 0, 1] Corners Count 1
[1, 1, 0, 1] Midpoints Count 3
Description ['irregularly shaped', 'oriented East-West']

2 Schapiro CEPSP
Tolerance 3
[1, 1, 1, 1] Corners Count 4
[1, 1, 1, 1] Midpoints Count 4
Description ['square']

```

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

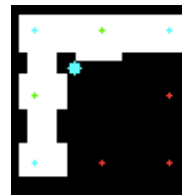
    # Extract four corners
    nw = (x,y)
    se = (x+w,y+h)
    ne = (x+w,y)
    sw = (x,y+h)

    # 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))

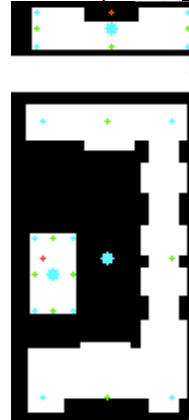
```



11 Mathematics
Tolerance 2
[1, 1, 1, 1] Corners Count 4
[1, 0, 1, 0] Midpoints Count 2
Description ['I-shaped',
'oriented North-South']



22 Journalism & Furnal
Tolerance 7
[1, 0, 1, 1] Corners Count 3
[1, 0, 0, 1] Midpoints Count 2
Description ['L-shaped']



20 Kent
Tolerance 2
[1, 1, 1, 1] Corners Count 4
[0, 1, 1, 1] Midpoints Count 3
Description ['U-shaped',
'oriented East-West']

23 Hamilton, Hartley, Wallach &
John Jay
Tolerance 8
[1, 1, 1, 1] Corners Count 4
[1, 1, 1, 0] Midpoints Count 3
Description ['U-shaped',
'oriented North-South']

```

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

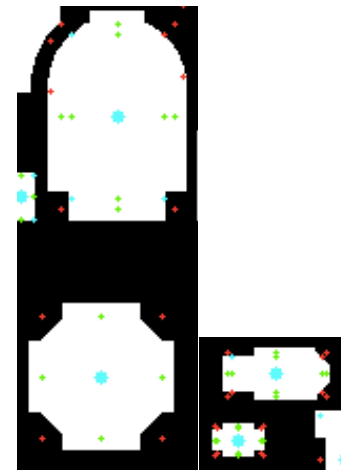
for corner in corners:
    if map_labeled[tuple(reversed(corner))] ==
building['number']:
        corners_filled.append(1)
        if draw_points:
            cv2.circle(map_campus, corner, 1, (255,255,0), -1)
    else:
        corners_filled.append(0)
        if draw_points:
            cv2.circle(map_campus, 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)

```



```

5 Gymnasium & Uris
Tolerance 6
[0, 0, 0, 0] Corners Count 0
[1, 1, 1, 1] Midpoints Count 4
[1, 0, 0, 0] Corners 2 Count 1
[1, 1, 1, 1] Midpoints 2 Count
4
Description ['bell-shaped',
'oriented North-South']
12 Low Library
Tolerance 6
[0, 0, 0, 0] Corners Count 0
[1, 1, 1, 1] Midpoints Count 4
Description ['squarish',
'cross-shaped']

```

Something else I would like to point out is the appearance of extra dots on the cross-shaped buildings. This is a later addition I made to distinguish between BELL-SHAPED (round) and CROSS-SHAPED (sharp) edges. As you can see, Uris and Lowe are quite different, but if but they both have 4 empty corners and 4 filled midpoints. However, Uris (and St. Paul's Chapel as well), both have asymmetrical sides. The way I deal with this is by adding tolerance/shift once again, and recalculating new midpoints and corners that are even closer to the building centroid. The result is some unevenness, and the bell shaped consistently end up with an odd number of filled corners (because it's hard to have perfect symmetry with rounded edges).

Also, you may notice that I commented out calculations for East-West (true if width is 1.5 times the height) and North-South orientations (true if height is 1.5 times the width) because this description proved to be extraneous, confusing with the direction set explored later, and my users asked me for clarification on what the orientation really meant. On the whole I was very pleased by the shape categories accuracy, and thought they removed the need for additional information in terms of orientation.

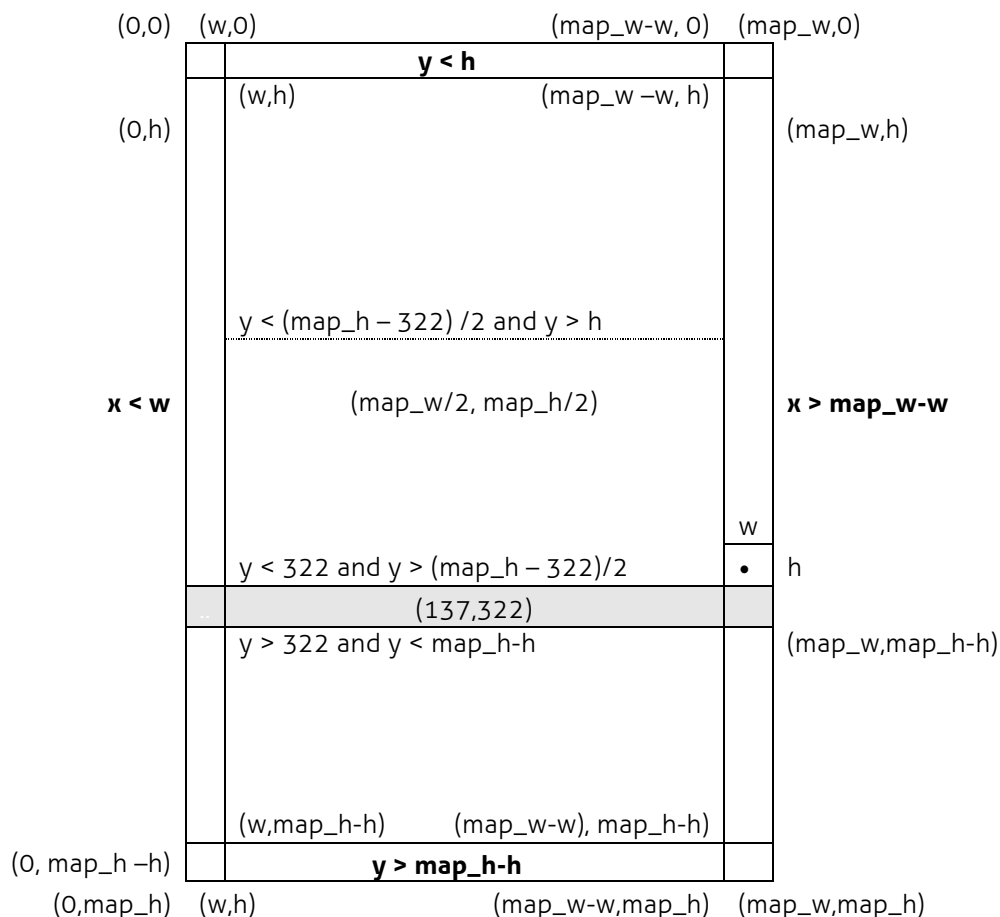
Describing location

To describe location, I divided campus into the following sections and used this diagram to determine if a building fell on any of the following noteworthy locations:

- Northwest corner
- Northeast corner
- Southeast corner
- Southwest corner
- Northernmost
- Southernmost
- Easternmost
- Westernmost
- Upper campus
- Central campus
- Lower campus

Since campus is very tall and narrow, northernmost and southernmost were pretty clear descriptors, but easternmost and westernmost were more ambiguous. Therefore, for the easternmost and westernmost buildings, I also checked if they belonged to lower campus (below our monument College Walk) or (upper/central campus). Since the word “central” is a bit ambiguous, I decided to only include buildings on the central vertical axis and the middle third of the map.

Note: in the diagram below, w and h vary building to building so the borders are flexible. Where a building's centroid (cx, cy) values lay in this dynamically calculated range of regions determined it's locative description.




```

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')

    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

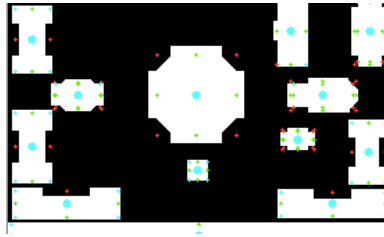
```

The three chopped regions:

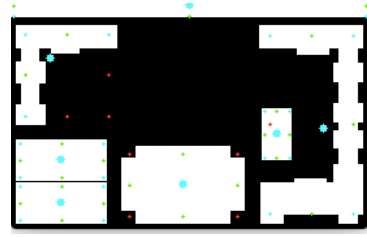
Upper



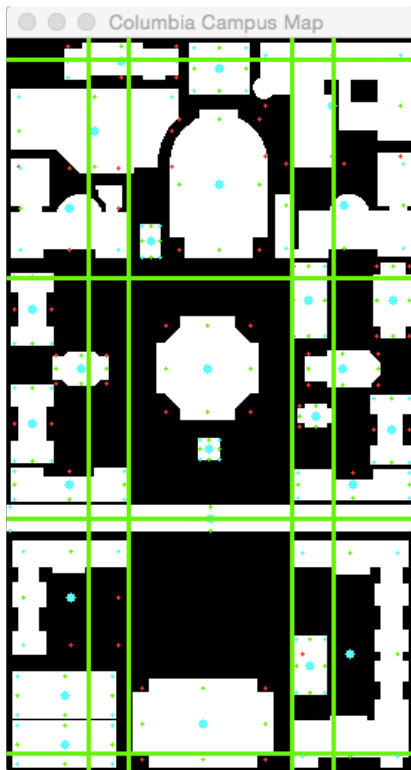
Central



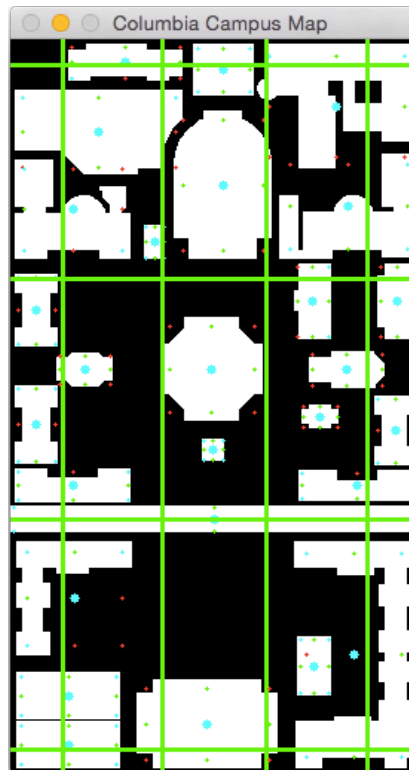
Lower



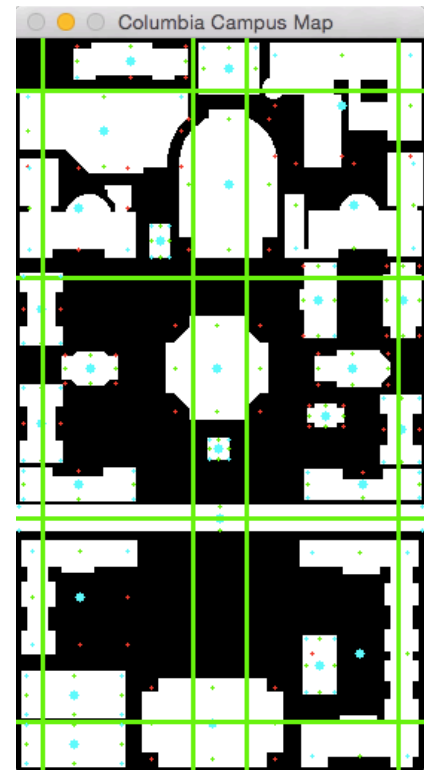
Some results follow below so you can understand what I mean by the borders being recalculated depending on which building is being checked for its descriptive location:



Kent, the U shaped building above College Walk on the right side of campus, is westernmost



Fayerweather, the narrow cross-shaped building, is also properly identified as westernmost



St. Paul's Chapel, the smaller bell shaped building, is correctly not assigned westernmost

Note: I wish I had color coded the appropriate buildings being checked and verified, as these natural language descriptions by me are not very good. Hopefully my system does better!

Description Disambiguation

In order to clean up my descriptions, I wrote two methods – one to find extrema (a piece of information that only applies to that building description, in which case all the other descriptors could be removed). The other method sought identical descriptions, and added a piece of information to make them more distinctive.

```

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']

```

The first function find_extrema() results in shorter descriptions like this:

<u>Building</u>	<u>Old description</u>	<u>New description</u>
Hamilton, Hartley, etc.	['colossal', 'irregularly shaped', 'northeast corner']	['northeast corner']
Alma Mater	['smallest', 'square', 'central campus']	['smallest']
Journalism & Furnald	['large', 'L-shaped', 'westernmost', 'lower campus']	['L-shaped']
Carman	['medium', 'rectangular', 'southwest corner']	['southwest corner']

While the second function find_ambiguity() clarifies identical descriptions like this:

<u>Building</u>	<u>Old description</u>	<u>New description</u>
Mathematics	['medium', 'l-shaped', 'westernmost']	['more northern', 'medium', 'l-shaped', 'westernmost']
Lewisohn	['medium', 'l-shaped', 'westernmost']	['more southern', 'medium', 'l-shaped', 'westernmost']

System Output

System output is enabled by the following simple method.

```

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']

```

And my system output for Part 1 is as follows:

```

1 : Pupin
    Minimum Bounding Rectangle: (39, 3) , (116, 28)
    Center of Mass: (77, 15)
    Area: 1640
    Description: ['medium', 'irregularly shaped', 'northernmost']
2 : Schapiro CEPsR
    Minimum Bounding Rectangle: (123, 3) , (164, 38)
    Center of Mass: (143, 20)
    Area: 1435
    Description: ['medium', 'square', 'northernmost']
3 : Mudd, Engineering Terrace, Fairchild & Computer Science
    Minimum Bounding Rectangle: (166, 3) , (273, 87)
    Center of Mass: (219, 45)
    Area: 5831
    Description: ['northeast corner']
4 : Physical Fitness Center
    Minimum Bounding Rectangle: (3, 34) , (116, 91)
    Center of Mass: (59, 62)
    Area: 5368
    Description: ['colossal', 'irregularly shaped', 'westernmost', 'upper campus']
5 : Gymnasium & Uris
    Minimum Bounding Rectangle: (110, 48) , (176, 148)
    Center of Mass: (143, 98)
    Area: 5753
    Description: ['colossal', 'bell-shaped', 'upper campus']
6 : Schermerhorn
    Minimum Bounding Rectangle: (181, 77) , (274, 148)
    Center of Mass: (227, 112)
    Area: 3911
    Description: ['large', 'irregularly shaped', 'easternmost', 'upper campus']
7 : Chandler & Havemeyer
    Minimum Bounding Rectangle: (3, 81) , (81, 148)
    Center of Mass: (42, 114)
    Area: 3613
    Description: ['large', 'irregularly shaped', 'westernmost', 'upper campus']
8 : Computer Center
    Minimum Bounding Rectangle: (90, 125) , (104, 148)
    Center of Mass: (97, 136)
    Area: 322
    Description: ['tiny', 'rectangular', 'upper campus']
9 : Avery
    Minimum Bounding Rectangle: (191, 151) , (216, 202)
    Center of Mass: (203, 176)
    Area: 1164
    Description: ['medium', 'almost rectangular', 'central campus']
10 : Fayerweather
    Minimum Bounding Rectangle: (247, 151) , (273, 202)
    Center of Mass: (260, 176)
    Area: 1182
    Description: ['medium', 'cross-shaped', 'easternmost']

```

- 11 : Mathematics
Minimum Bounding Rectangle: (3, 158) , (32, 207)
Center of Mass: (17, 182)
Area: 1191
Description: ['more northern', 'medium', 'I-shaped', 'westernmost']
- 12 : Low Library
Minimum Bounding Rectangle: (101, 187) , (170, 257)
Center of Mass: (135, 222)
Area: 3898
Description: ['squarish cross-shaped']
- 13 : St. Paul's Chapel
Minimum Bounding Rectangle: (201, 210) , (252, 235)
Center of Mass: (226, 222)
Area: 1087
Description: ['medium', 'bell-shaped', 'central campus']
- 14 : Earl Hall
Minimum Bounding Rectangle: (31, 211) , (69, 234)
Center of Mass: (50, 222)
Area: 759
Description: ['small', 'cross-shaped', 'central campus']
- 15 : Lewisohn
Minimum Bounding Rectangle: (3, 233) , (32, 286)
Center of Mass: (17, 259)
Area: 1307
Description: ['more southern', 'medium', 'I-shaped', 'westernmost']
- 16 : Philosophy
Minimum Bounding Rectangle: (245, 240) , (273, 287)
Center of Mass: (259, 263)
Area: 1085
Description: ['medium', 'I-shaped', 'easternmost']
- 17 : Buell & Maison Francaise
Minimum Bounding Rectangle: (196, 246) , (221, 262)
Center of Mass: (208, 254)
Area: 340
Description: ['tiny', 'cross-shaped', 'central campus']
- 18 : Alma Mater
Minimum Bounding Rectangle: (129, 269) , (144, 284)
Center of Mass: (136, 276)
Area: 225
Description: ['smallest']
- 19 : Dodge
Minimum Bounding Rectangle: (3, 289) , (81, 312)
Center of Mass: (42, 300)
Area: 1590
Description: ['medium', 'U-shaped', 'westernmost']
- 20 : Kent
Minimum Bounding Rectangle: (194, 290) , (273, 311)
Center of Mass: (233, 300)
Area: 1470
Description: ['medium', 'U-shaped', 'easternmost']
- 21 : College Walk
Minimum Bounding Rectangle: (1, 314) , (274, 332)
Center of Mass: (137, 323)
Area: 4950
Description: ['longest']
- 22 : Journalism & Furnald
Minimum Bounding Rectangle: (4, 338) , (82, 415)
Center of Mass: (43, 376)
Area: 2615
Description: ['L-shaped']
- 23 : Hamilton, Hartley, Wallach & John Jay
Minimum Bounding Rectangle: (191, 338) , (271, 491)
Center of Mass: (231, 414)

```

        Area: 5855
        Description: ['southeast corner']
24 : Lion's Court
        Minimum Bounding Rectangle: (193, 402) , (216, 442)
        Center of Mass: (204, 422)
        Area: 920
        Description: ['small', 'rectangular', 'lower campus']
25 : Lerner Hall
        Minimum Bounding Rectangle: (4, 426) , (74, 458)
        Center of Mass: (39, 442)
        Area: 2240
        Description: ['medium', 'rectangular', 'westernmost', 'lower campus']
26 : Butler Library
        Minimum Bounding Rectangle: (85, 431) , (180, 491)
        Center of Mass: (132, 461)
        Area: 5282
        Description: ['colossal', 'cross-shaped', 'southernmost']
27 : Carman
        Minimum Bounding Rectangle: (4, 459) , (74, 491)
        Center of Mass: (39, 475)
        Area: 2240
        Description: ['southwest corner']

```

Compact Spatial Relations: The "Where"

Overview

The second part of this system encodes every single building pair with the boolean relationship `is_north(s,t)`, `is_south(s,t)`, `is_east(s,t)`, `is_west(s,t)`, `is_near(s,t)` which can be read as “North of S is T” and “Near to S is T”, etc. I stored a lookup table for each of these 5 pairwise relationships in a numpy table, so that if I ever wanted to check on these pairwise relationships later, I could access them in constant time (similar to how I approached assignment 2 and the pairwise relationships between similar/dissimilar imagery).

```

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

```

North, East, South, West

Since it is probably more humanly accurate to have North be more complicated than simply comparing y coordinates alone, I generated a field of view for every one of the four cardinal directions. Since there was significant amount of calculation involved, I stored the points for the North, South, East and West FOVs in the building dictionary once they were generated so they could be reused later.

The field of view was calculated by taking the centroid of the Source building as one point in the triangle. I then generated slopes for each triangle side m1 and m2. If I was looking north, the y-coordinate of my two new points in the triangle would be on the 0 coordinate. If I was looking south, the y-coordinate of the two new points would be on the MAP_H value. Accordingly, if I was looking west, the x-value for the new points would be 0 (looking at the left axis) and the x-value for the new points in the triangle would be MAP_W on the right axis.

Once I've generated this Field of View (essentially, this triangle), I can easily figure out if the Target building's centroid is inside the triangle by using dot products and cross products in the Point in Triangle test¹. The full code for generating the triangle fields of view follow below, along with examples.

¹ <http://www.blackpawn.com/texts/pointinpoly/default.html>

```

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 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

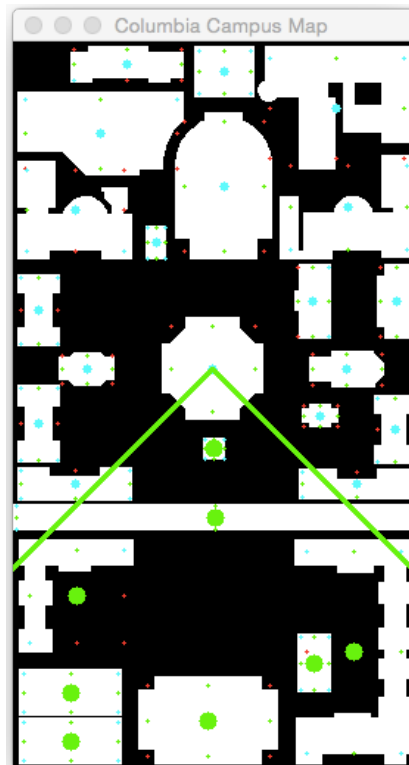
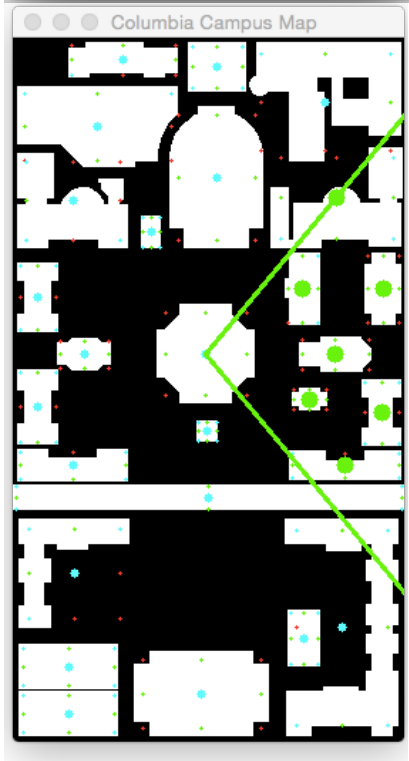
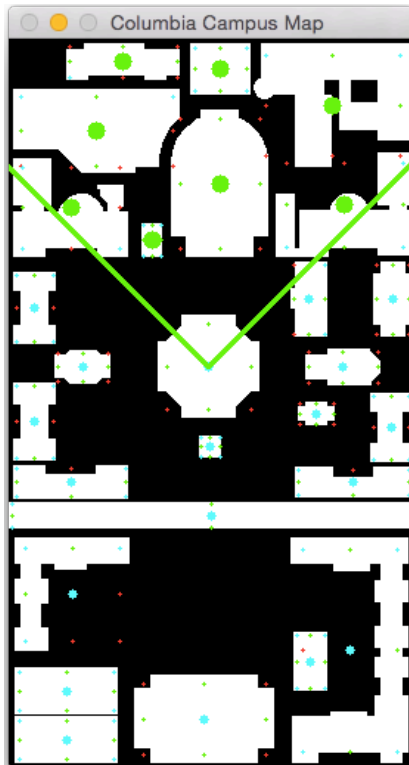
    # 4. Check whether target centroid is in the field of view
    if is_in_triangle(p3,p0,p1,p2):
        if (draw == True):
            cv2.circle(map_campus, p3, 6, (0,255,0), -1)
        return True

    # 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

```

All directions use the same logic to create a Field of View triangle for the Source building and determine which Target buildings centroids are in that triangle and thus in that direction. The only difference between the directional methods is the ideal slope and the known coordinates for the two new points. The third and fourth parameters in the `triangulate_FOV()` method are not really subject to subjectivity, but the last parameter, the slope, is the result of my own human judgment and testing.



```
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)

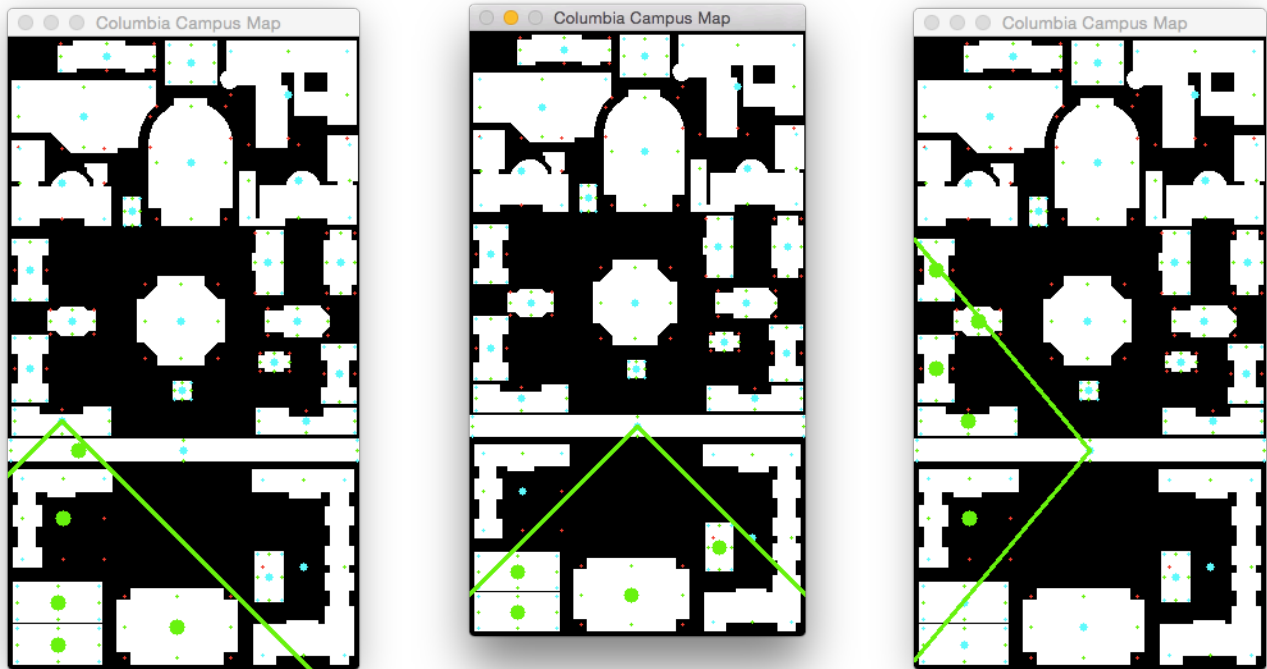
def is_east(s,t):
    """Find out if 'East of S is
    T'"""
    # Form triangle to east
    border: (MAP_W,y)
    return
    triangulate_FOV(s,t,MAP_W,-
    1,1.5)

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)
```

It makes sense that the North and South slopes are half the value of East and West since campus has much more MAP_H (map height) to survey than its narrow width.

Note that Schemerhorn is both North and East of Low Library, which hopefully you agree with. And even though the y-coordinate of Philosophy may be lower numerically than Low, I would consider it more West than South, and my system agrees with its creator.

Special FOV Target: College Walk



College Walk is a bit of a special case, so you may have noticed the additional condition I added in my triangulate method. It states that if College Walk is the Target building, then add two new “centroids” to it, one $1/5$ of the way down its length, and the other $4/5$ of the way down its length. This is because previously I was getting results like Dodge (rightmost picture) reporting that College Walk was only East and not South of it, when clearly it is South of everything that is not part of Lower Campus. This has to do with this monument’s uniquely disproportionate shape. In the rightmost picture, I have already added the condition, so you see the additional green circle near the starting point of the FOV – that is the new centroid, reporting in that it is indeed South of Dodge.

I contemplated whether College Walk as a Source building needed any special checking conditions, but it produced agreeable results from its current centroid, so I just left the check for the case where College Walk is the Target, and treat it as a normal case when College Walk is the Source.

```
# 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
```

Near

To calculate Near, I follow the method described in Abella's thesis of expanding the bounding boxes of the source and target and seeking intersection points (not elliptically however, but by rectangularly shifting the corners outwards, similar to what I did in my `corner_check` in my `describe_shape()` method). The shift is the minimum value between the width or height of the respective building, halved.

To determine if there is an intersection point, I reuse the Point in Triangle algorithm and see if any of the four corners or centroid of the Target building lie within one of the two triangles that compose the Source building's expanded bounding box.

```
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]

    w = s['xywh'][2]
    h = s['xywh'][3]
    s_shift = min(w,h)/2

    w = t['xywh'][2]
    h = t['xywh'][3]
    t_shift = min(w,h)/2

    s_points = get_near_points(s,s_shift)
    t_points = get_near_points(t,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 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

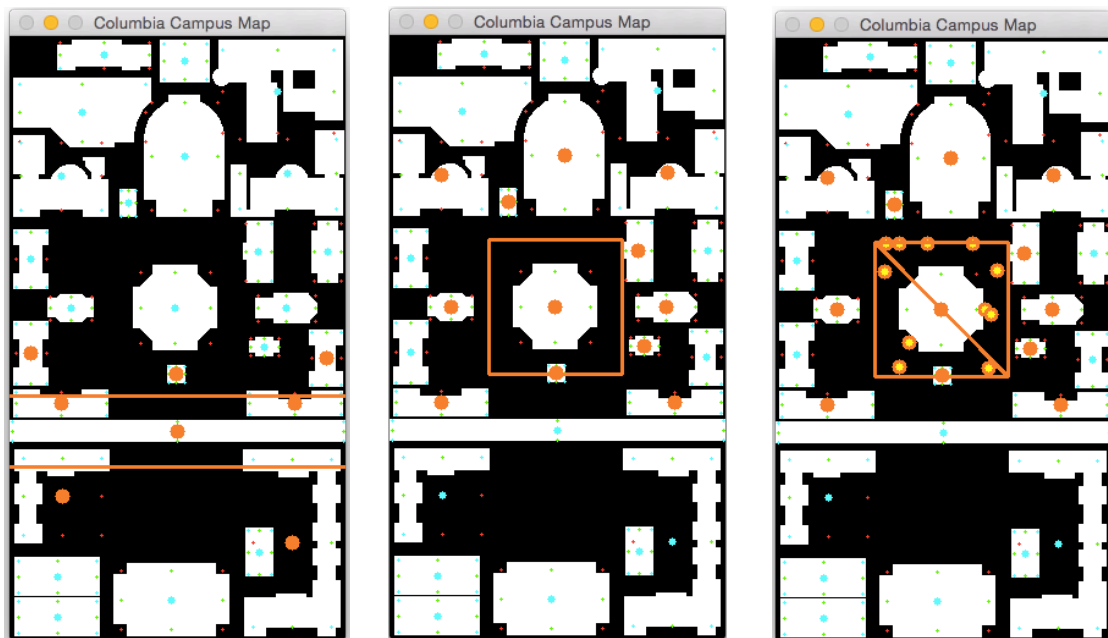
def extract_corners(x,y,w,h):
    nw = (x,y)
    ne = (x+w,y)
    se = (x+w,y+h)
    sw = (x,y+h)
    return nw,ne,se,sw

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)

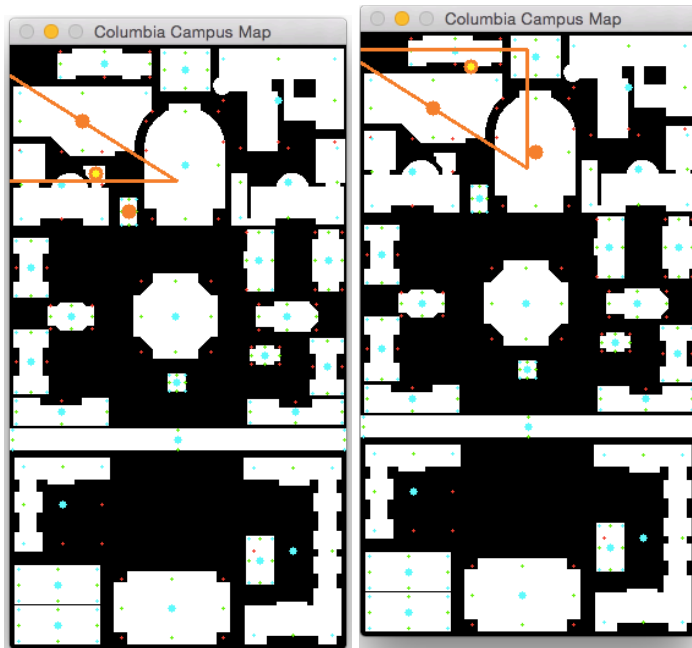
def draw_triangle(p1,p2,p3):
    cv2.line(map_campus,p1,p2,(0,128,255),2)
    cv2.line(map_campus,p2,p3,(0,128,255),2)
    cv2.line(map_campus,p3,p1,(0,128,255),2)

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

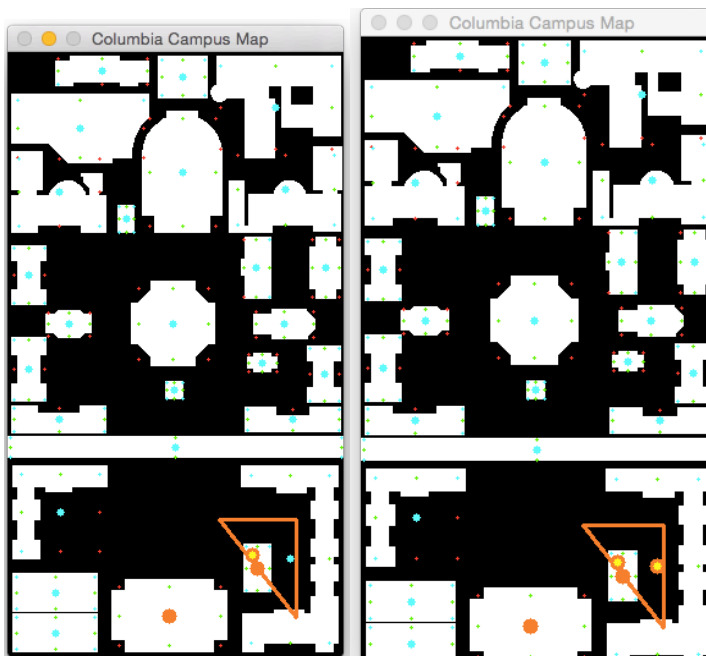
```



In the above sequence, I check all buildings to determine which are close to the Source building Low library. You can see that nearness to Low is determined by the presence of intersection points (the yellow and orange points) inside the expanded two triangles. The pure orange dots are the centroids of the qualified as “nearby” buildings.



In these two images to the left, I am checking in the first one whether the Computer Center is close to the Fitness Center (it is, and its intersection point is yellow and orange, while the appropriate triangle that returns True for `is_near()` statement is drawn). In the second image, I check if Uris is nearby to the Fitness Center, and yes it is, and again the intersection point is marked appropriate.



I also want to note the importance of not checking only the four corners, but also checking if the centroid intersects with the building. I observed this strange case where `is_near(Lion's Court, Hartley)` returned False, even though they are clearly right next to each other. This happened because Hartley's bounding box was so large that its corners could not possibly fit inside the smaller rectangle (or two triangles, however you want to think of it) of the smaller structure Lion's Court.

How Size Affects Nearness

Because of this determination of shift by building size and the depth of the centroid buried in the structure also as an effect of size, near is necessarily affected by the size of the building: it should be harder to be near to Alma Mater than to be near to Low. During transitive reduction, I reduce the buildings that are

is_near(s,t) and is_near(t,s) because the reciprocal relationship can be inferred, but I also print out eh relationships that are not reciprocal. It is important to note that it is indeed easier to be near to a larger structure with a greater expanded region in the Nearness algorithm than to a smaller structure which has not expanded that much.

Near to Schapiro CEPSR is Pupin but not other way around
Near to Mudd, Engineering Terrace, Fairchild & Computer Science is Pupin but not other way around
Near to Mudd, Engineering Terrace, Fairchild & Computer Science is Schapiro CEPSR but not other way around
Near to Mudd, Engineering Terrace, Fairchild & Computer Science is Physical Fitness Center but not other way around
Near to Physical Fitness Center is Pupin but not other way around
Near to Physical Fitness Center is Computer Center but not other way around
Near to Gymnasium & Uris is Schapiro CEPSR but not other way around
Near to Gymnasium & Uris is Mudd, Engineering Terrace, Fairchild & Computer Science but not other way around
Near to Gymnasium & Uris is Computer Center but not other way around
Near to Schermerhorn is Mudd, Engineering Terrace, Fairchild & Computer Science but not other way around
Near to Schermerhorn is Gymnasium & Uris but not other way around
Near to Schermerhorn is Avery but not other way around
Near to Schermerhorn is Fayerweather but not other way around
Near to Chandler & Havemeyer is Physical Fitness Center but not other way around
Near to Chandler & Havemeyer is Gymnasium & Uris but not other way around
Near to Chandler & Havemeyer is Computer Center but not other way around
Near to Chandler & Havemeyer is Mathematics but not other way around
Near to Avery is Gymnasium & Uris but not other way around
Near to Low Library is Gymnasium & Uris but not other way around
Near to Low Library is Schermerhorn but not other way around
Near to Low Library is Computer Center but not other way around
Near to Low Library is St. Paul's Chapel but not other way around
Near to Low Library is Earl Hall but not other way around
Near to Low Library is Buell & Maison Francaise but not other way around
Near to Low Library is Alma Mater but not other way around
Near to St. Paul's Chapel is Avery but not other way around
Near to St. Paul's Chapel is Fayerweather but not other way around
Near to Earl Hall is Mathematics but not other way around
Near to Philosophy is St. Paul's Chapel but not other way around
Near to Buell & Maison Francaise is St. Paul's Chapel but not other way around
Near to Dodge is Low Library but not other way around
Near to Kent is Low Library but not other way around
Near to Journalism & Furnald is Lewisohn but not other way around
Near to Journalism & Furnald is Dodge but not other way around
Near to Journalism & Furnald is College Walk but not other way around
Near to Journalism & Furnald is Lerner Hall but not other way around
Near to Journalism & Furnald is Carman but not other way around
Near to Hamilton, Hartley, Wallach & John Jay is Philosophy but not other way around
Near to Hamilton, Hartley, Wallach & John Jay is Kent but not other way around
Near to Hamilton, Hartley, Wallach & John Jay is College Walk but not other way around
Near to Hamilton, Hartley, Wallach & John Jay is Lion's Court but not other way around
Near to Hamilton, Hartley, Wallach & John Jay is Butler Library but not other way around
Near to Butler Library is Lerner Hall but not other way around
Near to Butler Library is Carman but not other way around

Transitive Reduction (NESW)

After generating pairwise relationships for every single building, it's time to filter out these "where" relationships leaving only the ones that cannot be inferred by the usual transitivity rules. The example given in the assignment specifications is that CEPsR satisfies both North(Low, CEPsR) and North(Butler, CEPsR), but you can drop the second one because we know from the map North(Butler, Low), so we can infer North (Butler, CEPsR) anyway. It's important to note there are cases where buildings can be North (or whatever direction) of multiple buildings after filtering, e.g. Avery and Fayerweather are both north of St. Paul's but not North of each other so the chain rule of inference can no longer apply. I accomplish this reduction through a series of loop checks, removing unnecessary intermediary points so that we have a sparser lookup matrix from which we can still infer the pruned relationships.

```
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
```

The second part of filtering is that if the target is north of a source we no longer need to say the inverse relationships that the source is south of the target. As you shall see in my System Output included below, this second part of the reduction makes the secondary South and West relationship tables very sparse.

Transitive Reduction (Near)

In terms of transitive reduction of near, this is a bit more tricky because the relationships are not as symmetric. I've already discussed the how size affects nearness and included that output list of non-symmetric relationships up there, where it is much easier to be near a bigger structure than to be near a small one. I didn't tamper with compass directions, because I thought that the reduction above was quite effective.


```

# 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

```

Note that this snippet of code is a continuation of the `transitive_reduce()` method above.

Results < 27*27*5

After calculating the $O(27*27*5)$ binary relationships I printed out only the True relationships in 5 separate tables. While I had **934** True relationships in the beginning across all the five lookup tables, after transitive reduction I had reduced the total count to **182**. The matrices with building numbers and True relationships indicated by the presence of “1” in the appropriate row and column are included below, along with the complete pruned list in English format .

The following two methods are used to print the system output. The first generates the table of numbers and indices. The second produces English language descriptions of the remaining 182 filtered relationships.

```

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

```

```
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']
```

System Output

Apologies for the miniscule font face. The main point is to see the reduction and patterns in the table representation anyway.

Original Relationship Table

Total count: 934

North relationships:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
1
2
3
4 1
5 1 1
6 1 1 1
7 1 1 1 1
8 1 1 1 1 1
9 1 1 1 1 1 1
10 1 1 1 1 1 1 1
11 1 1 1 1 1 1 1 1
12 1 1 1 1 1 1 1 1 1
13 1 1 1 1 1 1 1 1 1 1
14 1 1 1 1 1 1 1 1 1 1 1
15 1 1 1 1 1 1 1 1 1 1 1
16 1 1 1 1 1 1 1 1 1 1 1
17 1 1 1 1 1 1 1 1 1 1 1
18 1 1 1 1 1 1 1 1 1 1 1
19 1 1 1 1 1 1 1 1 1 1 1
20 1 1 1 1 1 1 1 1 1 1 1
21 1 1 1 1 1 1 1 1 1 1 1
22 1 1 1 1 1 1 1 1 1 1 1
23 1 1 1 1 1 1 1 1 1 1 1
24 1 1 1 1 1 1 1 1 1 1 1
25 1 1 1 1 1 1 1 1 1 1 1
26 1 1 1 1 1 1 1 1 1 1 1
27 1 1 1 1 1 1 1 1 1 1 1 1
```

Number of true relationships: 253

South relationships:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
1
2 1 1
3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
6 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
8 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
10 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
12 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
13 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
14 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
15 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
16 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
17 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
18 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
19 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
20 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
21 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
22 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
23 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
24 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
25 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
26 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
27 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Number of true relationships: 256

After Transitive Reduction

Total count: 182

North relationships:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
1
2
3
4 1
5 1 1
6 1 1 1
7 1 1 1 1
8 1 1 1 1 1
9 1 1 1 1 1 1
10 1 1 1 1 1 1 1
11 1 1 1 1 1 1 1 1
12 1 1 1 1 1 1 1 1 1
13 1 1 1 1 1 1 1 1 1 1
14 1 1 1 1 1 1 1 1 1 1 1
15 1 1 1 1 1 1 1 1 1 1 1
16 1 1 1 1 1 1 1 1 1 1 1
17 1 1 1 1 1 1 1 1 1 1 1
18 1 1 1 1 1 1 1 1 1 1 1
19 1 1 1 1 1 1 1 1 1 1 1
20 1 1 1 1 1 1 1 1 1 1 1
21 1 1 1 1 1 1 1 1 1 1 1
22 1 1 1 1 1 1 1 1 1 1 1
23 1 1 1 1 1 1 1 1 1 1 1
24 1 1 1 1 1 1 1 1 1 1 1
25 1 1 1 1 1 1 1 1 1 1 1
26 1 1 1 1 1 1 1 1 1 1 1
27 1 1 1 1 1 1 1 1 1 1 1 1
```

Number of true relationships: 52

South relationships:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
```

Number of true relationships: 7

1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
2	1	1			1	1			1	1			1			1											
3		1				1				1																	
4		1	1		1	1			1	1			1			1	1			1							
5		1			1				1	1						1											
6																											
7		1	1		1	1		1	1	1		1	1			1	1			1	1						
8		1			1	1			1	1		1				1	1			1							
9									1																		
10									1	1																	
11	1	1			1	1		1	1	1		1	1	1		1	1	1		1	1			1	1		
12					1				1	1		1				1	1			1	1						
13									1							1											
14		1		1	1				1	1		1	1			1	1	1		1	1			1	1		
15		1		1	1				1	1		1	1	1		1	1	1		1	1			1	1		
16																											
17									1							1											
18								1	1			1				1	1			1	1			1			
19		1			1			1	1		1	1				1	1	1		1	1			1	1		
20								1	1							1											
21									1			1				1	1			1			1	1			
22					1			1	1			1				1	1	1		1	1		1	1		1	
23																											
24																								1			
25									1			1				1	1			1	1		1	1		1	
26																							1	1			
27									1			1				1	1			1	1		1	1		1	

Number of true relationships: 170

[illegible]

Number of true relationships: 170

[illegible]

Number of true relationships: 85

1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
2		1			1					1																	
3			1			1					1																
4		1			1							1					1			1							
5			1			1			1				1														
6														1													
7		1					1						1					1			1						
8					1																						
9										1																	
10																											
11		1						1						1													
12					1				1												1						
13								1		1							1										
14					1							1							1						1		
15													1														
16														1													
17										1							1										
18									1												1						
19		1										1							1						1		
20													1														
21														1					1						1		
22						1									1					1							1
23																											
24																								1			
25																									1		1
26																										1	
27																						1					1

Number of true relationships: 55

[illegible]

Number of true relationships: 19

[illegible]

Number of true relationships: 49

North of Physical Fitness Center is:
Pupin
North of Gymnasium & Uris is:
Pupin

North of Gymnasium & Uris is:
Schapiro CEPSP
North of Schermerhorn is:
Schapiro CEPSP

North of Schermerhorn is:
Mudd, Engineering Terrace, Fairchild & Computer
Science
North of Chandler & Havemeyer is:
Schapiro CEPsR
Physical Fitness Center
North of Computer Center is:
Physical Fitness Center
Gymnasium & Uris
North of Avery is:
Gymnasium & Uris
Schermerhorn
North of Fayerweather is:
Pupin
Schermerhorn
North of Mathematics is:
Chandler & Havemeyer
North of Low Library is:
Schermerhorn
Chandler & Havemeyer
Computer Center
North of St. Paul's Chapel is:
Physical Fitness Center
Avery
Fayerweather
North of Earl Hall is:
Mudd, Engineering Terrace, Fairchild & Computer
Science
Computer Center
Mathematics
North of Lewisohn is:
Earl Hall
North of Philosophy is:
St. Paul's Chapel
North of Buell & Maison Francaise is:
Chandler & Havemeyer
Computer Center
St. Paul's Chapel
North of Alma Mater is:
Avery
Fayerweather
Low Library
North of Dodge is:
Low Library
Lewisohn
North of Kent is:
Philosophy
Buell & Maison Francaise
North of College Walk is:
Earl Hall
Buell & Maison Francaise
Alma Mater
North of Journalism & Furnal is:
St. Paul's Chapel
Dodge
College Walk
North of Hamilton, Hartley, Wallach & John Jay is:
Kent
College Walk
North of Lion's Court is:
Lewisohn
Kent
College Walk
North of Lerner Hall is:
Philosophy
Journalism & Furnal
North of Butler Library is:
Kent
Journalism & Furnal
North of Carman is:
Kent
Lerner Hall

South of Earl Hall is:
Hamilton, Hartley, Wallach & John Jay
South of Buell & Maison Francaise is:
Lerner Hall
South of Alma Mater is:
Journalism & Furnal
South of Dodge is:
College Walk
South of Kent is:
College Walk
South of College Walk is:
Lerner Hall
South of College Walk is:
Butler Library
Schapiro CEPsR
Gymnasium & Uris
East of Schapiro CEPsR is:
Mudd, Engineering Terrace, Fairchild & Computer
Science
Schermerhorn
Fayerweather
East of Physical Fitness Center is:
Schapiro CEPsR
East of Physical Fitness Center is:
Gymnasium & Uris
Buell & Maison Francaise
Kent
East of Gymnasium & Uris is:
Mudd, Engineering Terrace, Fairchild & Computer
Science
Schermerhorn
Avery
St. Paul's Chapel
East of Chandler & Havemeyer is:
Schapiro CEPsR
East of Chandler & Havemeyer is:
Computer Center
Low Library
East of Computer Center is:
Gymnasium & Uris
Buell & Maison Francaise
Kent
East of Avery is:
Fayerweather
East of Mathematics is:
Schapiro CEPsR
East of Mathematics is:
Computer Center
Earl Hall
East of Low Library is:
Schermerhorn
Avery
College Walk
East of St. Paul's Chapel is:
Fayerweather
Philosophy
East of Earl Hall is:
Gymnasium & Uris
Low Library
Alma Mater
Lion's Court
East of Lewisohn is:
Earl Hall
East of Buell & Maison Francaise is:
Fayerweather
East of Buell & Maison Francaise is:
Philosophy
East of Alma Mater is:
Avery
College Walk
East of Dodge is:
Mudd, Engineering Terrace, Fairchild & Computer

Science
 Low Library
 Alma Mater
 Lion's Court
 East of Kent is:
 Philosophy
 East of College Walk is:
 St. Paul's Chapel
 Buell & Maison Francaise
 Kent
 Lion's Court
 East of Journalism & Fernald is:
 Schermerhorn
 Alma Mater
 Butler Library
 East of Lion's Court is:
 Hamilton, Hartley, Wallach & John Jay
 East of Lerner Hall is:
 College Walk
 East of Lerner Hall is:
 Butler Library
 East of Butler Library is:
 Lion's Court
 East of Carman is:
 College Walk
 Butler Library
 West of Low Library is:
 College Walk
 West of St. Paul's Chapel is:
 Low Library
 West of St. Paul's Chapel is:
 Alma Mater
 West of St. Paul's Chapel is:
 Lerner Hall
 West of St. Paul's Chapel is:
 Carman
 West of Buell & Maison Francaise is:
 Low Library
 West of Buell & Maison Francaise is:
 Alma Mater
 West of Buell & Maison Francaise is:
 Lerner Hall
 West of Buell & Maison Francaise is:
 Carman
 West of Alma Mater is:
 College Walk
 West of Kent is:
 Low Library
 West of Kent is:
 Alma Mater
 West of Kent is:
 Lerner Hall
 West of Kent is:
 Carman
 West of College Walk is:
 Earl Hall
 West of College Walk is:
 Dodge
 West of College Walk is:
 Journalism & Fernald
 West of Hamilton, Hartley, Wallach & John Jay is:
 Alma Mater
 West of Hamilton, Hartley, Wallach & John Jay is:
 Lerner Hall
 Physical Fitness Center
 Gymnasium & Uris
 Near to Schapiro CEPSR is:
 Pupin
 Mudd, Engineering Terrace, Fairchild & Computer
 Science
 Physical Fitness Center

Near to Physical Fitness Center is:
 Gymnasium & Uris
 Near to Gymnasium & Uris is:
 Schapiro CEPSR
 Near to Gymnasium & Uris is:
 Mudd, Engineering Terrace, Fairchild & Computer
 Science
 Near to Schermerhorn is:
 Mudd, Engineering Terrace, Fairchild & Computer
 Science
 Gymnasium & Uris
 Avery
 Fayerweather
 Near to Chandler & Havemeyer is:
 Physical Fitness Center
 Gymnasium & Uris
 Near to Computer Center is:
 Gymnasium & Uris
 Chandler & Havemeyer
 Near to Avery is:
 Gymnasium & Uris
 Low Library
 Near to Mathematics is:
 Chandler & Havemeyer
 Near to Mathematics is:
 Lewisohn
 Near to Low Library is:
 Alma Mater
 Near to St. Paul's Chapel is:
 Avery
 Fayerweather
 Near to Earl Hall is:
 Mathematics
 Lewisohn
 Near to Lewisohn is:
 Dodge
 Near to Philosophy is:
 St. Paul's Chapel
 Buell & Maison Francaise
 Kent
 Near to Buell & Maison Francaise is:
 Low Library
 St. Paul's Chapel
 Near to Dodge is:
 Journalism & Fernald
 Near to Kent is:
 Buell & Maison Francaise
 Near to Kent is:
 Hamilton, Hartley, Wallach & John Jay
 Near to College Walk is:
 Lewisohn
 Philosophy
 Alma Mater
 Dodge
 Kent
 Journalism & Fernald
 Hamilton, Hartley, Wallach & John Jay
 Near to Journalism & Fernald is:
 Butler Library
 Near to Lion's Court is:
 Hamilton, Hartley, Wallach & John Jay
 Near to Lion's Court is:
 Butler Library
 Near to Lerner Hall is:
 Journalism & Fernald
 Butler Library
 Carman
 Near to Butler Library is:
 Hamilton, Hartley, Wallach & John Jay
 Near to Carman is:
 Butler Library

User Interface: Source and Target

Overview

In this part, I created a user interface to record clicks. In this particular step, the system takes two clicks, a Source and a Target, and creates a tiny 1-pixel building that exists in relation to the other building (note this building can also be 'in' another building). What's most interesting is generating an ambiguity cloud for describing this location only in terms of where.

Cloud Generation

Upon a user's click, my mouse callback function calls the following functions.

```
# 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
```

The first function creates a new building with a new centroid and 1 pixel width and height, so that it can be entered as a source/target parameter for all the other relationship boolean-returning functions. Change_color() just changes the color of our cloud. pixel_cloud() is called on the clicked point, and this is where things get interesting. First, it generates a relationship for this new "click building" with every other building on the map. Then it reduces the relationship by calling reduce_by_nearness(). After that, it calls a recursive flood_fill() to recursively iterate through all the pixels around it to find out how big (or small) the cloud of ambiguity is for this particular description.

```
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]

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 # original MAP_H-`
    flood_fill(x, y+pix, rel_table, indices)

```

The Search for the Biggest Cloud

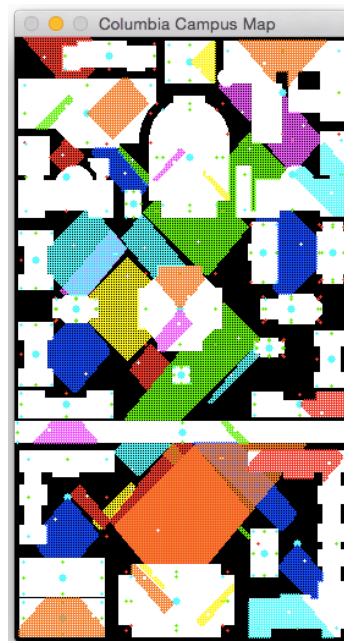
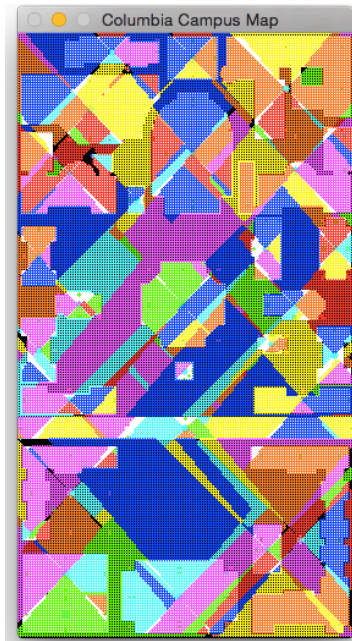
In order to increase performance, I choose to jump by a certain number of pixels (in this case, just 2) rather than checking every single pixel in the flood algorithm. This allowed me to test my clouds with the follow function (which only tests every 10 pixels and generates these 2-pixel jump clouds at each intersection).

```

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

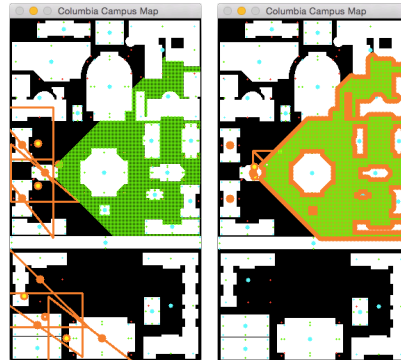
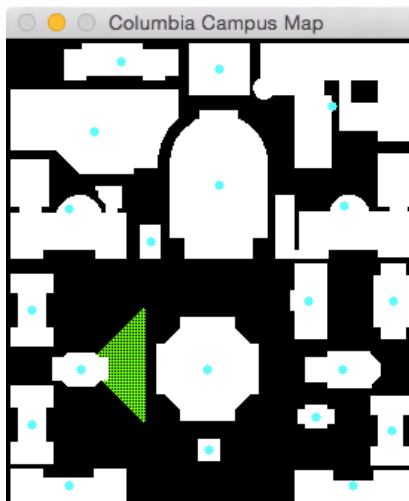
```

Below are the pretty results of brute-force checking every 10 pixels, and manually checking every pixel for the largest ambiguity clouds. The system outputs the cloud size and number of recursive calls for every single cloud generated, and my test_clouds() method returned a sorted list of clouds by size after about 25 minutes of runtime.

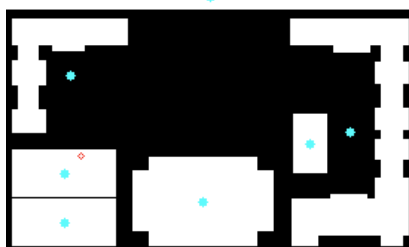


But alas, there were some bugs in my nearness algorithm, so after I fixed it, the position of the largest cloud was no longer at the same point. Nonetheless, I'd like to talk about it in my three interesting paths.

Three Interesting Paths



Original Largest Cloud – now average



Source:

Click (70,210) is east of the small, cross-shaped, central campus structure (Earl Hall), east of the more northern, medium, I-shaped, westernmost structure (Mathematics), and west of the squarish cross-shaped structure (Low Library).

Size of cloud: 674 (recursive calls: 419)

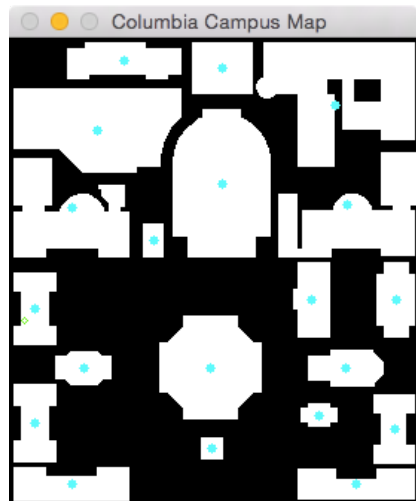
Target:

Click (50,430) is inside and to the north of and east of the medium, rectangular, westernmost, lower campus structure (Lerner Hall), south of the L-shaped structure (Journalism & Furnald), and west of the colossal, cross-shaped, southernmost structure (Butler Library).

Size of cloud: 2 (recursive calls: 5)

This is the current state of the formerly largest cloud returned by my test_cloud function. (Before pictures above as well). The description of Source is highlighted in green above, essentially it is East of Earl Hall, east of Mathematics, and West of Low. Originally, this cloud extended all the way to the Eastern border of campus, in a Field of View radar. The Nearness and West functions were clearly not working for that cloud to have traveled so far, and this had to do with my earlier performance decision to store dictionary values such as the FOV and nearness points for constant time lookup in the building dictionary. In the case of the “clicked” building, this was a very special building where the centroid essentially was always moving as the recursive flood_fill() algorithm worked its way through the map. However, in my faulty implementation, the original centroid always stayed the same, so it would continue returning true for the entire scope of that point. (The plaid pattern that test_cloud() created on the map was pretty, but that’s about the end of its advantages). The picture above makes sense, all the points in that triangle are indeed East of the two structuresn West of Lowe.

As for my smallest cloud (the Target), it stayed small. I had created an inside function, so the flood fill would not traverse past a building’s boundaries to go outside. This point appears in that very small intersection between Lerner’s North_FOV and East_FOV, so it doesn’t have much room for ambiguity.



New and More Accurate Largest Cloud (Target) and Smallest Cloud (Source)

Source:

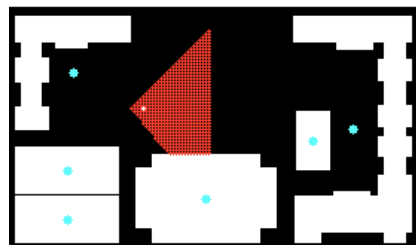
Click (10,190) is inside and to the south of and west of the more northern, medium, I-shaped, westernmost structure (Mathematics), west of the small, cross-shaped, central campus structure (Earl Hall), and north of the more southern, medium, I-shaped, westernmost structure (Lewisohn).

Size of cloud: 2 (recursive calls: 5)

Target:

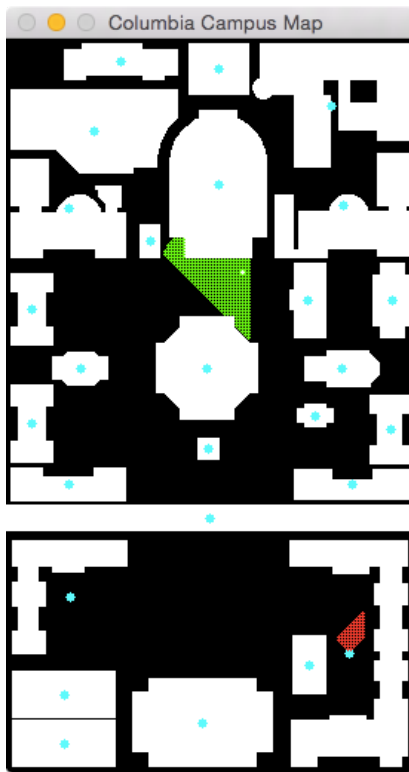
Click (90,400) is east of the L-shaped structure (Journalism & Furnald), east of the medium, rectangular, westernmost, lower campus structure (Lerner Hall), and north of the colossal, cross-shaped, southernmost structure (Butler Library).

Size of cloud: 1360 (recursive calls: 779)



The Source is now a small source, it may be hard for you to find but it is in Mathematics. It’s in the south and west – so again, there is this very specific region which lies between two adjacent directions, especially the closer you are to the center of mass from which this Field of Vision stems. The relationships it has with the next two closest buildings (Earl and Lewisohn) are identical to the relationships the Mathematics building itself has with them.

As for the Target From my manual clicking, this was the largest cloud I could find. You might note that all of these large clouds are in a triangular shape. That may have to do with the straight line definitions of my FOV technique for determining direction and my constraint of including only three neighbors, as too much information might overwhelm the user. The lawn is a fairly sparse area in terms of structural representations but it’s interesting to see how this small theoretically one pixel building generates its own FOV-shaped ambiguity cloud.



Source:

Click (159,157) is west of the medium, almost rectangular, central campus structure (Avery), south of the colossal, bell-shaped, upper campus structure (Gymnasium & Uris), and east of the tiny, rectangular, upper campus structure (Computer Center).

Size of cloud: 826 (recursive calls: 510)

Target:

Click (230,403) is north of the southeast corner structure (Hamilton, Hartley, Wallach & John Jay), east of the small, rectangular, lower campus structure (Lion's Court), and east of the colossal, cross-shaped, southernmost structure (Butler Library).

Size of cloud: 130 (recursive calls: 94)

The Source building is inside the East_FOV of the Computer Center, as well as South of the larger FOV of Uris. It's closest structure is Avery, that's why it's listed first in its description, but the cloud of ambiguity actually leans toward the Computer Center more because there is more combinatorial intersection between the relevant directional fields of view for Uris and the Computer Center.

This path is also interesting because of the somewhat incorrect directions it entails for Target. the target is defined as North of the Hamilton building block, but it's actually North of the centroid. It is east of Lion's Court and Butler library, but the cloud is shrunk by its closest proximity to Hamilton. Also, as a student who usually thinks of Hamilton and Hartley as different buildings, I would say this area is West of Hartley and South of Hamilton, but definitely not North of this block.

Natural Language Descriptors

These are the print functions for the Source and Target descriptions. They just stitch together "Where" and "What" features into hopefully readable English.

```
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

```

Creativity: Path Generation

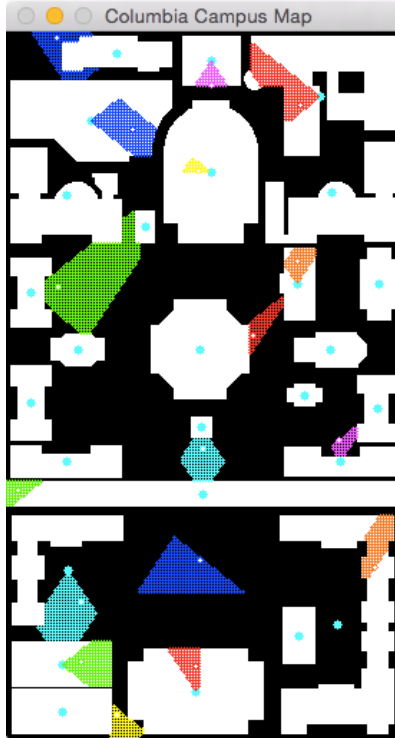
Overview

In this part, I finally get to describe a path from S and T, and save the results for a friend to see how well he/she can follow the purely “where” results and the combined “where” and “what” (but no name) results.

Dijkstra Algorithm

To generate the graph, I only chose buildings that were close together for connection. As soon as users completed my study, I realized how flawed this algorithm for this scenario and I would have chosen a different shortest-path algorithm to generate this blind path. Most likely, I would have implemented to a path that always takes the user to the closest building in this general direction. I would also have liked to utilize the ambiguity clouds in an evaluative yet efficient fashion to generate less ambiguous results, particularly for the directions without parentheses.

My Path Clicks and Clouds:



I first saved a series of 8 clicked paths using my own user interface.

I stored the printed output as the following global variables:

```
# 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)]
```

Through Euclidean distances and Dijkstra's algorithm I generated the following 8 paths (where the numbers represent building indices):

```
[20, 17, 11, 8, 5, 2], [0, 4, 8, 11, 17], [26, 25, 22],
[19, 20, 14, 10], [25, 21, 18, 14, 10, 6, 3], [21, 18,
14, 10, 6, 4], [8, 4, 1], [20, 17, 11]
```

```
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 Dijkstra 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)

```

Description Generation

Descriptions were generated by the following functions. In the mouse callback method, every click that indicated the end of a particular itinerary would change a global variable `itinerary_number`, and `print_instructions` would be called, generating the instructions for that particular itinerary.

```

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 '\nITINERARY ' + 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 '-----'

```

User Results

Unfortunately, only the first three images saved for some reason. Nonetheless, I will do my best to analyze based on the final distance of the clicks and the paths themselves.

For the first itinerary from the West Broadway gates of College Walk to Mudd, the confusion was clear without parentheses. The directions said go North and West, which should have been Low. However, the lack of “What” description just made my friend go a little bit to the West and North, and then continue following the steps directing him to go up. However, with the parenthetical information describing the building shape, a differen friend got very close to the final destination before the red circle showed up.

ITINERARY 1

You are inside a building (the longest structure) to the WEST.

Now go to the nearby building that is NORTH and WEST (the smallest structure).

Now go to the nearby building that is NORTH (the squarish cross-shaped structure).

Now go to the nearby building that is EAST (the medium, almost rectangular, central campus structure).

Now go to the nearby building that is NORTH (the large, irregularly shaped, easternmost, upper campus structure).

Now go to the nearby building that is NORTH (the northeast corner structure).

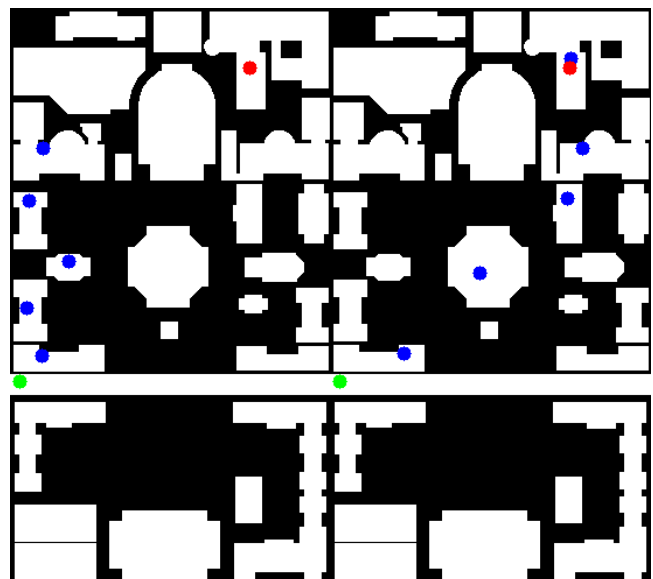
Your final destination is inside this building. Go SOUTH within the building.

Clicked location: (153,90)
Final destination: (205, 51)
Distance: 65.0

Clicked location: (27,130)
Final destination: (205, 51)
Distance: 194.743420942

Good job! Next Itinerary!

No Parentheses (Orange)
Left Image (Orange).



Access to Parentheses (Blue).
Right Image (Blue).

ITINERARY 2

You are outside. Go to the nearby building that is EAST (the medium, irregularly shaped, northernmost structure).

Now go to the nearby building that is SOUTH (the colossal, bell-shaped, upper campus structure).

Now go to the nearby building that is SOUTH (the medium, almost rectangular, central campus structure).

Now go to the nearby building that is WEST (the squarish cross-shaped structure).

Now go to the nearby building that is SOUTH (the smallest structure).

Your final destination is inside this building. Go SOUTH within the building.

Clicked location: (136,275)
Final destination: (137, 291)
Distance: 16.0312195419

Clicked location: (121,231)
Final destination: (137, 291)
Distance: 62.096698785

Good job! Next Itinerary!

ITINERARY 3

You are outside. Go to the nearby building that is WEST (the southwest corner structure).

Now go to the nearby building that is EAST (the colossal, cross-shaped, southernmost structure).

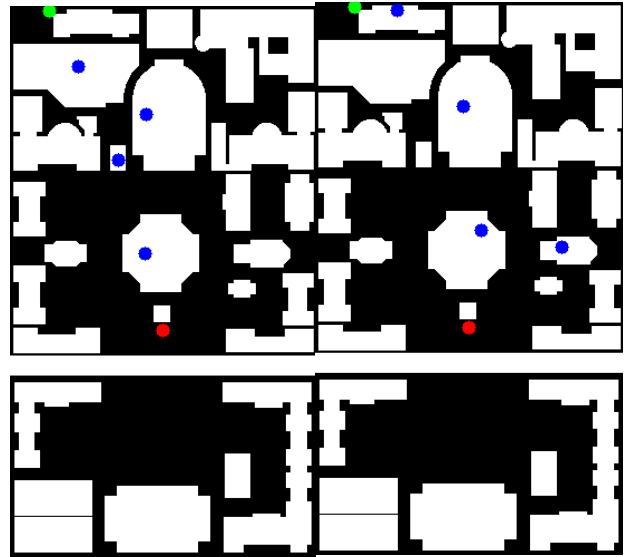
Now go to the nearby building that is EAST (the southeast corner structure).

Your final destination is inside this building. Go NORTH and WEST within the building.

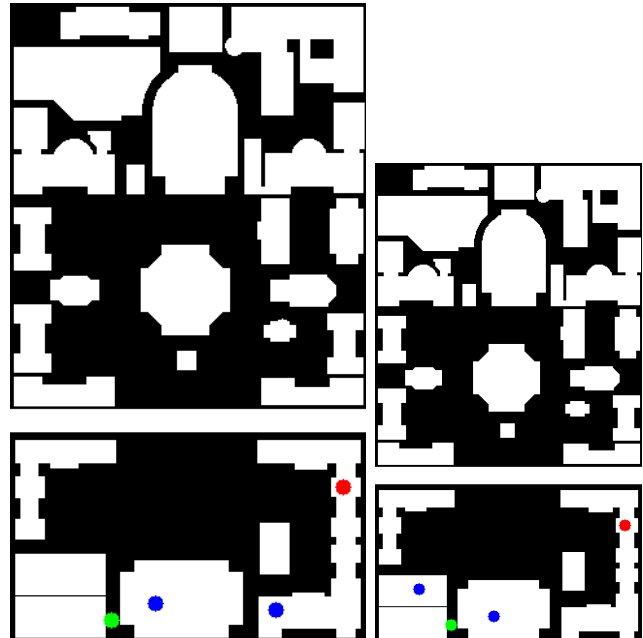
Clicked location: (218,473)
Final destination: (257, 374)
Distance: 106.404887106

Clicked location: (197,460)
Final destination: (257, 374)
Distance: 104.861813831

Good job! Next Itinerary!



For the second itinerary, again, the user with additional parenthetical information prevailed. However, the orange user in the righthand picture did quite well. He followed the directions to go south, but skipped the Computer Center and veered a little far by going to St. Paul's. However, Alma Mater and St. Paul's is very near and the Euclidean distance is under 100, so kudos to him.



For the third itinerary, both of the users performed similarly. They were both confused by the directions of going within a certain building. However this straight path from East to West seemed quite unambiguous, especially because there are less structures in lower campus, so having parentheses didn't give a huge advantage.

Code Listing

vismap.py

```
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),(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\tRatio r\tDiff r\tMax r\tBuilding'
        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 difference 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('suarish 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"""
    # TODO: Uncomment these and explain
    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_LBUTTONDOWN:
    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_earness(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 Dijkstra 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 '\nITINERARY ' + 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 '\nITINERARY ' + str(i+1)
        print '-----'
        itinerary = firsthalf[i]

```