Anthony Dubis - ajd2194
Homework 3

Step 0 - Data Representation

There are a few details worth covering here that will be relevant to the rest of the write-up.

While the foundation of the assignment is around computer vision, it is useful to represent the data as a graph when trying to solve many of the challenges. Specifically, the buildings can be thought of nodes, which can have attributes to describe themselves and spatial relationships to other buildings (nodes).

I completed this assignment using MATLAB_R2014b and its Image Processing Toolbox. While I don't consider MATLAB much of an object-oriented language, it offers some features that make a graph representation possible. To satisfy the assignment's needs, I created my own MATLAB *classdef* (definition of a class) called **Building**. Every building on campus, as well as the pixel points for Source and Target in **Step 3**, were represented as a Building instance.

Building instances have **properties** to easily access attributes such as the name, number, shape, size, and center of mass of the building. I also had properties for the north, south, east, west, and near relationships. These spatial properties each contained their own vector of numbers to relate the building instance to other building instances (an adjacency list). For example, if the "Alma Mater" building instance contained the vector [12 5] in its "north" property, that implied that "north of Alma Mater are building numbers 12 and 5." These building numbers could then be used to lookup the corresponding building instances and their data (used to traverse the graph).

Building instances also have **methods** associated with them. Some are simply helper methods that helped aggregate a building's properties (such as shape and size) to form a single string description. However, many others are typical setters and getters for accessing the building's properties. For example, when the building's bounding box is set, the setter will also create the building's "expandedBoundingBox" which is used when determining what other buildings are near.

The full list of properties and methods for the Building *classdef* can be seen in *Building.m*.

All of the buildings are stored in a collection called **bMap** that you'll see throughout this write-up and the code. **bMap** needs to be passed around when accessing adjacent nodes to lookup the actual building instance of a given building number.

With this knowledge of how the data is represented, we can now proceed with the other steps. Please note that the program kicks off from **hw3.m**.

Step 1 - Create the Building Instances and Determining the "What"

In this step, all of the Building (*a classdef explained in Step 0)* instances are created and we give each of them descriptions.

To assist with these, MATLAB's **regionprops()** function is used to get many of the basic attributes, such as center of mass, bounding box, area, and orientation. It does this by taking in a binary image, which was created from the *ass3-campus.pgm* file, and determining these properties for each connected component. These components are identified by looking up one of the pixels they contain in the *ass3-labled.pgm* file, which provides the building number. These building numbers are then matched with *ass3-table.txt* to give the name of the building.

The information from region props is then stored directly in the attributes of the corresponding building instances and is also used in helper methods to determine a building's size, shape, and region of the campus. These attributes are then combined into a single description by a helper method in the Building classdef called **getDescription**; in other words, **getDescription** gives us a single string describing the building.

Let's quickly explain each of these characteristics that aren't self-explanatory and/or didn't come straight from the **regionprops()** function.

<u>Determining a Building's Size Description</u> - **getSizeDescription.m**

The size of a building is dependent on the other buildings. In the creation of the buildings in **hw3.m**, the min and max area for all of the buildings is maintained. Once all of the buildings have been created, we then iterate through them again with this min and max in mind to label them with an appropriate size. The work for this is done in **getSizeDescription()**, which compares a building's area to the min and max. Specifically, it determines where the building's area falls in the **range** of areas represented by [min, max].

The possible labels and when they are described are as follows:

- **"Smallest in Size":** This is assigned when the area of a building is equal to the smallest area of all buildings.
- **"Tiny in Size"**: This labeled is assigned to buildings that fall in the **lower** 8.3% of the [min, max] range.
- **"Small in Size"**: This label was assigned to buildings that fall in the **lower** 8.3% to 25% of the [min, max] range.
- **"Medium in Size":** This label was assigned to buildings that fall in the **lower** 25% to 58.3% of the [min, max] range.

- **"Large in Size":** This label was assigned to buildings that had areas in the **upper** 41.7% of the [min, max] range.

These numbers were found by trial and error and what seemed visually reasonable. Ultimately, they resulted in the creation of segments of size (max - min) / 6 that would divided up the range from min to max. Portions of the range were assigned to the above labels by using different amounts of these segments (for example, the "tiny" label got the lowest ½ segment, the "small" label got the next smallest 1 ½ segment, and so on).

Notice that I didn't include a **"Largest in Size"** label like I did for smallest. I chose to do this because I didn't see the largest extrema label as useful because of how hard it is to tell differences in building size as the size grew. For example, I would have thought that one of the buildings 3, 4, or 5 were the largest just by looking at the map, but the largest actually turned out to be 23. I believe this inability to rank the larger sizes is due to the complex shapes that they come in. However, as building's get smaller, it's typically quite easy to say one is smaller than the other.

This realization can also be seen in how most of the segregation of buildings takes place in the lower portions of the range. The "Large" label takes almost the entire upper half of the range because it's much harder to visually distinguish these larger, oddly shaped areas.

<u>Determining a Building's Orientation</u> - **setOrientation()** in the Building classdef

Determine the orientation is straight forward. The orientation number given by **regionprops()** is between -90 and 90. If it is closer to one of the endpoints of this range, the building is classified as being oriented north-to-south. If it is closer to 0 (-45 < orientation < 45), the building is classified as being oriented east-to-west.

<u>Determining a Building's Shape</u> - **determineShape.m**

Trying to describe a building's shape in a way that is as unique as possible was a bit tricky. There are several building characteristics determined up front to make this classification:

- **Area and BoundingBox** - provided by **regionprops**()
- **BoundingBoxArea -** found by multiplying the height and width of the bounding box
- **Boundaries** - the boundaries of the building and any internal holes it might have
- The **numberOfLargeCircles** and **numberOfMediumCircles** the building contains (distinguishes 5, 6, and 7 from the others).

Boundaries was found using MATLAB's **bwboundaries()** function. It will find the boundary for all connected components in the image and any holes they may contain.

numberOfLargeCirlces and **numberOfMediumCircles** were determined by supplying two different radius ranges (one for medium size circles and one for larger circles) to MATLAB's **imfindcircles()** function, which tries to find circles in an image. This function is applied to an isolated image of each building during the **setImage()** setter method of the Building class during instance creation (see Building.m). It returns the number of circles it believes are in the supplied image, which is then assigned to the appropriate variable. The ranges to differentiate large and medium circles were formed by trial and error to classify building 5 differently than 6 and 7.

With these pieces of data, the function then proceeds to classify the specified building as one of the following shape descriptions:

Shape	Description
Contains a Large Curved Room	The building was found to have one or more large circles.
Contains a Medium Sized Curved Room	The building was found to have one or more medium sized circles.
Divides Campus	The building's bounding box stretches the entirety of the width or height of the campus.
Square	The area of the building equals the area of its bounding box and the sides of the bounding box are equal in length.
Rectangle	The area of the building equals the area of its bounding box but the sides of the bounding box are not equal.
I-Shaped	The building fills the corners of its bounding box, but has background pixels halfway down the edges of both its vertical sides.
Irregular shape with a hole	If the number of boundaries the building contains is greater than 1, it has a hole.
Close to a rectangle in shape	If the area of the building fills 85% or more of the bounding box's area, then it was considered to be "close" to a rectangle
L-Shaped	Divide a building's bounding box into four quadrants, like an x vs y graph with it's 0,0 origin in the middle. If the building does not contain pixels in one of these quadrants, then it contains a gap there and it is L-Shaped.
C-Shaped	With the L-Shaped definition in mind, take one of these quadrants

	and slide it halfway down each side (left and right) of the bounding box. If the building does not contain any pixels in one of these displaced quadrants, then there is a gap there and it is C-shaped.
Cross-Shaped	If the building does not fill any of the four corners of its bounding box, but it does extend to mid-points along each of the sides of its bounding box, it's considered cross-shaped.
Irregular shape	Any building that did not fall into one of the above categories was considered irregular shaped (just building 4).

<u>Determining a Building's Region</u> - **getRegion()**

I classified a building's region on the map by dividing the map's width and height into three segments to create nine equal sized regions. These regions are as follows:

Northwest	**North-Center**	**Northeast**
Mid-West	**Center**	**Mid-East**
Southwest	**South-Center**	**Southeast**

Even though Columbia's campus isn't oriented perfectly north-to-south, I label it as if it is since that is generally how people approach it.

The building's center of mass is used to determine which label it gets assigned. In other words, the region the centroid lands in is the label the building's region is given.

There is no "deadspace" as there was for assignment 1. I didn't view that as appropriate here for two reasons. First, in assignment 1, we wanted to make sure the region the gesture was being displayed in was deliberate to avoid wrong classifications. I didn't see that to be as much of an issue when describing a location on a map as it's probably deemed ok to be close enough.

Second, it would be pretty rare for someone to describe a building's location as "not sure." Even if they were on the fence, the person would say something like "well, it's kind of in the middle of campus, maybe off to the west side a little bit." We can give this level of fuzziness by continuing to break the map into a greater number of smaller comments, but stopping at nine seems reasonable.

Summary of Descriptive Data

The table below shows a summary of the descriptive data for each building. A couple of notes:

- Points are rounded to the nearest integer for display purposes.
- The center of mass is given as (x,y)
- The corners for the bounding box are formatted as follows: [topLeftX, topLeftY, bottomRightX, bottomRightY]
- The descriptions were generated by the **getDescription** instance method of the Building class. It concatenates the data generated in previous work in a relatively human-friendly way.

#	Name	Center of Mass	Area	Bounding Box	Description
1	"Pupin"'	[77,16]	1640	[40,4,117,29]	Medium in size, oriented west-to-east, close to a rectangle in shape, and in the northwest part of campus.'
2	"Schapiro CEPSR"'	[144,21]	1435	[124,4,165,39]	Small in size, oriented west-to-east, rectangle in shape, and in the north-center part of campus.'
3	"Mudd, Engineering Terrace, Fairchild & Computer Science"'	[224,36]	5831	[167,4,274,88]	Large in size, oriented west-to-east, irregular in shape with a hole, and in the northeast part of campus.'
4	"Physical Fitness Center"'	[60,59]	5368	[4,35,117,92]	Large in size, oriented west-to-east, irregular in shape, and in the northwest part of campus.'
5	"Gymnasium & Uris"'	[144,100]	5753	[111,49,177,149]	Large in size, oriented north-to-south, contains a large curved room, and in the north-center part of campus.'
6	"Schermerhorn"'	[234,122]	3911	[182,78,275,149]	Large in size, oriented west-to-east, contains a medium sized curved room, and in the northeast part of campus.'
7	"Chandler & Havemeyer"'	[39,121]	3613	[4,82,82,149]	Large in size, oriented west-to-east, contains a medium sized curved room, and in the northwest part of campus.'
8	"Computer Center"'	[98,137]	322	[91,126,105,149]	Tiny in size, oriented north-to-south, rectangle in shape, and in the north-center part of campus.'
9	"Avery"'	[205,177]	1164	[192,152,217,203]	Small in size, oriented north-to-south, close to a rectangle in shape, and in the mid-east part of campus.'
10	"Fayerweather"'	[261,177]	1182	[248,152,274,203]	Small in size, oriented north-to-south, close to a rectangle in shape, and in the mid-east part of campus.'
11	"Mathematics"'	[18,183]	1191	[4,159,33,208]	Small in size, oriented north-to-south, I-shaped, and in the mid-west part of campus.'
12	"Low Library"'	[136,223]	3898	[102,188,171,258]	Large in size, oriented north-to-south, cross shaped, and in the center part of campus.'

13	"St. Paul''s Chapel"'	[228,223]	1087	[202,211,253,236]	Small in size, oriented west-to-east, close to a rectangle in shape, and in the mid-east part of campus.'
14	"Earl Hall"'	[51,223]	759	[32,212,70,235]	Small in size, oriented west-to-east, close to a rectangle in shape, and in the mid-west part of campus.'
15	"Lewisohn"'	[18,260]	1307	[4,234,33,287]	Small in size, oriented north-to-south, I-shaped, and in the mid-west part of campus.'
16	"Philosophy"'	[259,264]	1085	[246,241,274,288]	Small in size, oriented north-to-south, I-shaped, and in the mid-east part of campus.'
17	"Buell & Maison Francaise"'	[209,255]	340	[197,247,222,263]	Tiny in size, oriented west-to-east, close to a rectangle in shape, and in the mid-east part of campus.'
18	"Alma Mater"'	[137,277]	225	[130,270,145,285]	Smallest in size, oriented west-to-east, square in shape, and in the center part of campus.'
19	"Dodge" '	[43,302]	1590	[4,290,82,313]	Small in size, oriented west-to-east, close to a rectangle in shape, and in the mid-west part of campus.'
20	"Kent"'	[234,302]	1470	[195,291,274,312]	Small in size, oriented west-to-east, close to a rectangle in shape, and in the mid-east part of campus.'
21	"College Walk"'	[138,324]	4950	[1,315,276,333]	Large in size, oriented west-to-east, divides campus, and in the center part of campus.'
22	"Journalism & Furnald"'	[32,365]	2615	[5,339,83,416]	Medium in size, oriented north-to-south, L-shaped, and in the southwest part of campus.'
23	"Hamilton, Hartley, Wallach & John Jay"'	[241,418]	5855	[192,339,272,492]	Large in size, oriented north-to-south, C-shaped, and in the southeast part of campus.'
24	"Lion''s Court"'	[205,423]	920	[194,403,217,443]	Small in size, oriented north-to-south, rectangle in shape, and in the southeast part of campus.'
25	"Lerner Hall"'	[40,443]	2240	[5,427,75,459]	Medium in size, oriented west-to-east, rectangle in shape, and in the southwest part of campus.'
26	"Butler Library"'	[133,461]	5282	[86,432,181,492]	Large in size, oriented west-to-east, close to a rectangle in shape, and in the south-center part of campus.'
27	"Carman"'	[40,476]	2240	[5,460,75,492]	Medium in size, oriented west-to-east, rectangle in shape, and in the southwest part of campus.'

Step 2 - Describing Spatial Relations - The "Where"

This step involves both the setting and pruning of spatial relationships. The code for each of these steps can be found in **setSpatialRelationships.m** and **pruneRelationships.m,** respectively, but I'll walk through the logic here.

Setting the Spatial Relationships - **setSpatialRelationships.m**

The program iterates through all possible ordered pairs of buildings in **hw3.m** and then calls the function **setSpatialRelationships()** for each. I say "ordered" pairs because I designed **setSpatialRelationships()** to work with one building as the reference and the other building we are trying to describe with respect to that reference. In other words, for any two buildings S and T, we want to call **setSpatialRelationships(S, T)** and **setSpatialRelationships(S, T)**.

setSpatialRelationships(S, T) takes in two buildings S and T and determines their relation to each other. Specifically, it determines if T is near to S based on S's size and it determines if T is north, west, south, and/or east of S.

Near(S,T)
Whether T is near S is determined by **near(S, T)**. This function checks to see if the expanded bounding box of S intersect the bounding box of T. This expandedBoundingBox attribute is determined when the building's true bounding box is set (see setBoundingBox() in Building.m). In general, it increases the height and width of the true bounding box by multiplying each by **2.1**. This creates a new, larger rectangle that is then centered over the original bounding box. Scaling by a constant factor means big buildings get a significantly larger expandedBoundingBox, while small buildings get a relatively smaller one. With regards to this function, this means that it is easier to be considered "near" to a big building than a smaller one since it is more likely that other building's will intersect with these enlarged boundaries.

The number **2.1** was found through observation. In other words, I looked at the map (as a human) and made some judgements about what building's I thought should be considered near to each other. Through this, I continued to increase the scaling factor until the results I was getting were satisfactory.

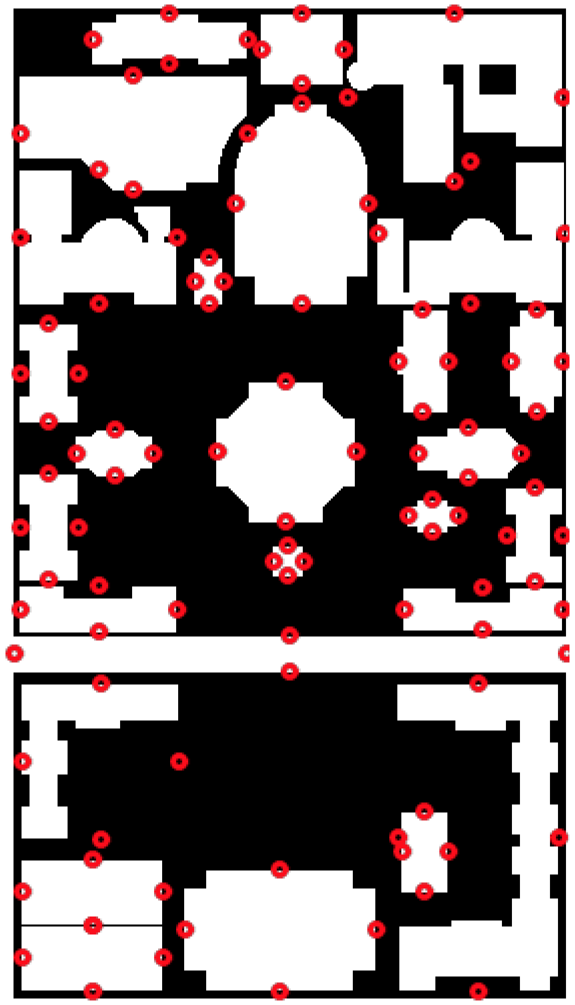
However, I set a couple of restrictions in place to ensure desired results. Specifically, I made it so that the width and height of the expanded bounding boxes had to be at least 30 pixels larger than the original. This allowed me to keep the scaling factor lower and still get the desired "near" relationships for smaller buildings (such as Low being considered as near to Alma Mater).

In addition, I set a maximum increase in the height and width of 70 pixels. This allowed me to increase the scaling factor without causing the expanded bounding box of larger buildings to become too large. For example, Hamilton should not be considered near to St. Paul's Chapel just because it's bounding box grew so much.

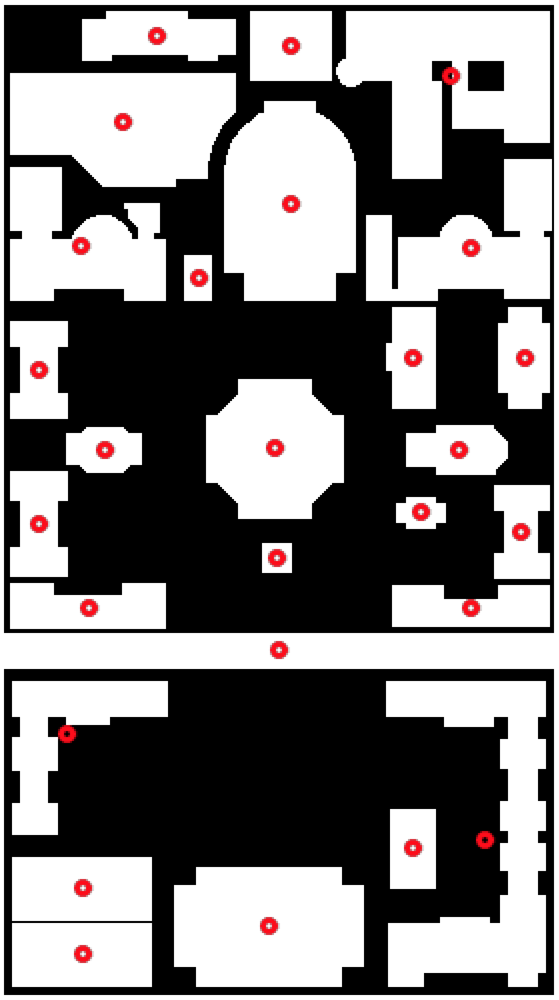
Overall, these min and max restrictions made sense. As humans, I feel like we could classify something as near to a reference object based on some sort of margin around it. While this margin would grow as the object did, it may not continue to expand out at the same rate. Instead, there's likely some maximal margin we start to form around its perimeter to decide if something is near. Conversely, the margin for smaller objects is likely much bigger relative to the object's size. Just because an object only takes up 1 square foot doesn't mean we would feel like we need to be one foot away to be near. Instead, it's reasonable to think that being within 10 feet of the object, especially in such a large campus/map, would imply that we are pretty close to it.

North(S,T), South(S,T), East(S,T), and West(S,T)

For determining north, south, east, and west relationships, I used what I called "spatial points" for each building. These are essentially the midpoints for each side of a building's bounding box. They were originally used for a different algorithm and I could have just as easily have used the bounding box's corners here, but I decided to stick with them all the same. You can see them mapped for each building below.


Spatial Points for each Building

The centroids (center of mass) were also used in these relationships. I have plotted those below.


Centroids for each building

To help explain these binary functions, let's walkthrough the **south(S, T)** function as the other three are very similar and can be quickly inferred. Remember, according the assignment specification, **south(S, T)** is true when "south of S is T." Finally, note that the origin for this map starts in the **top left corner**. The x-coordinate increases as we go right, and the y-coordinate increases as **we go down**.

There are two steps to determine if **south(S, T)** is true. First, the y-coordinate of S's centroid must be lower (higher on the map) than the y-coordinate of T's centroid. Second, the intervals formed by the west and east spatial points of each building must overlap.

For example, consider S = Pupin and T = Physical Fitness Center. The centroid of S does have a lower (higher on the map) y coordinate than T, so the first condition is satisfied. Also, the west-to-east ranges of these buildings overlap, so the second condition is satisfied. Therefore, T is considered to be south of S.

However, note that the Mathematics building (11) is not considered south of Pupin. Although the first condition holds, the second condition fails because the two do not have overlapping west-to-east ranges. However, we can see that Mathematics is south of the fitness center and Chandler & Havemeyer. These buildings **are** considered south of Pupin, we can also infer that Mathematics is south of Pupin (which is quite reasonable).

This is what of the biggest benefits of the overlapping range condition. As we get farther south (or any direction) from a reference, other buildings can be inferred as being south of the reference due to the west-to-east range of buildings in between. This actually helps establish relationships that are more human-like. For example, a human would likely consider all buildings south of College Walk as being south of Pupin. This is captured by the range condition and the inferred relationships!

North(S, T) is very similar except that the centroid condition is reverse. **East(S, T)** and **West(S, T)** are also similar except that the first condition looks at the x-coordinate of the centroid and the second condition looks for an intersection in the north-to-west ranges for each building.

All of these are broken down into useful helper functions in **setSpatialRelationships.m** to reduce the amount of repetitive and very similar code.

Pruning the Relationships - **pruneRelationships.m**

With the relationships set, we can now prune them to remove those that can be inferred in the **pruneRelationships()** function.

pruneRelationships() calls a recursive function **prune(bMap, current, original, direction)**. As mentioned in Step 0, **bMap** contains all of the Building objects are are accessed by building number (bMap('1') for Pupin). **original** is the Building object having its relationships pruned. Specifically, a relationship is moved from **original** if it can be inferred by a relationship of **current**, a child of **original.** Finally, the particular relationship we are looking at in this pruning is specified by **direction**, which is equal to 'north', 'south', 'east', or 'west'.

The **prune()** algorithm works its way down the graph with **original** as the start and highest parent node. Note that this graph is directed and acyclic with respect to just a single directional relationship. Also note that **prune()** is called with the **original** and **current** node set to the same building that is being pruned. In other words, you don't prune the north and south relationships at the same time. You'll want to consider one first and then the other.

prune() begins by calling itself recursively on all adjacent buildings of **current** with respect to the **direction** relationship. Once we reach a building that has no other adjacent buildings for that relationship to recurse on (for example, Pupin has no other building's that are 'north' of it), we go back up the recursive calls.

As we work our way back up, we remove relationships from **original** that can be inferred by going through **current**. We can safely do so because these are redundant and we can still establish the relationship by traversing the directed, acyclic graph formed by these directions when taken on their own.

When we eventually reach the **original** node (i.e. when the building numbers for **original** and **current** match), we are done pruning.

Note that I did not prune on the "near" relationships. In my mind, this didn't make much sense as the buildings were only being labeled as near to buildings that truly seemed near to them. This was normally a small handful, something that I thought was reasonable and more intuitive to keep attached to the Building objects rather than trying to infer them.

Resulting Relationships

The resulting relationships from the above processes are as shown below. As requested, building names are used rather than their numbers.

Source	North	East	South	West
"Pupin"'		"Schapiro CEPSR"'	"Physical Fitness Center"'	
"Schapiro CEPSR"'		"Mudd, Engineering Terrace, Fairchild & Computer Science"'	"Gymnasium & Uris"'	"Pupin", "Physical Fitness Center"'
"Mudd, Engineering Terrace, Fairchild & Computer Science"'		"Schermerhorn"'	"Gymnasium & Uris", "Schermerhorn"'	"Schapiro CEPSR", "Gymnasium & Uris"'
"Physical Fitness Center"'	"Pupin"'	"Schapiro CEPSR", "Gymnasium & Uris"'	"Gymnasium & Uris", "Chandler & Havemeyer", "Computer Center"'	"Chandler & Havemeyer"'
"Gymnasium & Uris"'	"Schapiro CEPSR", "Mudd, Engineering Terrace, Fairchild & Computer Science", "Physical Fitness Center"'	"Mudd, Engineering Terrace, Fairchild & Computer Science"'	"Low Library"'	"Physical Fitness Center", "Computer Center"'
"Schermerhorn"'	"Mudd, Engineering Terrace, Fairchild & Computer Science"'		"Avery", "Fayerweather"'	"Mudd, Engineering Terrace, Fairchild & Computer Science"'
"Chandler & Havemeyer"'	"Physical Fitness Center"'	"Physical Fitness Center", "Computer Center"'	"Mathematics"'	

Anthony Dubis - ajd2194
Homework 3

"Computer Center"'	"Physical Fitness Center"'	"Gymnasium & Uris"'	"Low Library"'	"Chandler & Havemeyer"'
"Avery"'	"Schermerhorn"'	"Fayerweather"'	"St. Paul''s Chapel"'	"Low Library"'
"Fayerweather"'	"Schermerhorn"'		"St. Paul''s Chapel"'	"Avery"'
"Mathematics"'	"Chandler & Havemeyer"'	"Low Library"'	"Earl Hall"'	
"Low Library"'	"Gymnasium & Uris", "Computer Center"'	"Avery", "St. Paul''s Chapel", "Buell & Maison Francaise"'	"Alma Mater"'	"Mathematics", "Earl Hall"'
"St. Paul''s Chapel"'	"Avery", "Fayerweather"'		"Philosophy", "Buell & Maison Francaise"'	"Low Library"'
"Earl Hall"'	"Mathematics"'	"Low Library"'	"Lewisohn"'	"Lewisohn"'
"Lewisohn"'	"Earl Hall"'	"Earl Hall", "Alma Mater"'	"Dodge" '	
"Philosophy"'	"St. Paul''s Chapel"'		"Kent"'	"Buell & Maison Francaise", "Alma Mater"'
"Buell & Maison Francaise"'	"St. Paul''s Chapel"'	"Philosophy"'	"Kent"'	"Low Library"'
"Alma Mater"'	"Low Library"'	"Philosophy"'	"College Walk"'	"Lewisohn"'
"Dodge" '	"Lewisohn"'	"Kent"'	"College Walk"'	
"Kent"'	"Philosophy", "Buell & Maison Francaise"'		"College Walk"'	"Dodge" '
"College Walk"'	"Alma Mater", "Dodge", "Kent"'		"Journalism & Furnald", "Hamilton, Hartley, Wallach & John Jay", "Butler Library"'	
"Journalism & Furnald"'	"College Walk"'	"Lion''s Court"'	"Lerner Hall"'	
"Hamilton, Hartley, Wallach & John Jay"'	"College Walk"'		"Lion''s Court"'	"Lion''s Court"'
"Lion''s Court"'	"Hamilton, Hartley, Wallach & John Jay"'	"Hamilton, Hartley, Wallach & John Jay"'		"Journalism & Furnald", "Butler Library"'
"Lerner Hall"'	"Journalism & Furnald"'	"Butler Library"'	"Carman"'	
"Butler Library"'	"College Walk"'	"Lion''s Court"'		"Lerner Hall", "Carman"'
"Carman"'	"Lerner Hall"'	"Butler Library"'		

Step 3 - Source/Target Descriptions and the User Interface

In this section, we talk about setting the source and target areas. Although these areas will be a "cloud" of pixels where the system describes each pixel in the same way. This process begins in the Step 3 section of **hw3.m**. For both the source (S) and the target (T), we'll let the user specify a point and each of these points will go through the following process.

<u>Creating a Building for a Point</u> - **buildingForPoint.m**

To take advantage of the graph we've setup for the buildings, the specified point will be turned into a building B in the function **buildingForPoint()**. In this function, a new Building object is created, which is located at the point the user clicked and is 1 pixel in size. B is then related to the actual buildings of Columbia's campus by calling the **setSpatialRelationships()** just as was discussed in **Step 2** for the original 27 buildings.

<u>Describing a Point Building</u> - **getBuildingSpatialDescription.m**

With B added to our graph with the appropriate relationships, we then describe it with relation to the actual buildings of the university. This takes place in the **getBuildingSpatialDescription()** function, which does two things. First, it determines if a building is nearby or if B is within an actual building. Second, it determines the most relevant directional relationships based on which are closest.

The function begins by calling **getNearbyDescription()**, which returns three possible results specifying what B is near. If B is not near any buildings, an empty string is returned. If B is located inside an actual building, a string specifying this is returned. Finally, if neither of the other two conditions are met, then the string returned gives a building that B is near.

With the "nearby description" in hand, the function then calls the helper function **getMostRelevantRelations()**. This function returns a string that describes B with respect to at most two other buildings of different directional types (so it won't say the point is <u>north</u> of Carmen and <u>north</u> of Lerner). The two directional descriptions are chosen based on the reference building proximity. Specifically, the closer one of the reference building's spatial points are to B, the more likely it is to be chosen as the most relevant relationship for that type (north, south, east, west). An example might make this more clear...

Consider a point right in front of Butler. This point will have several directional relationships with the buildings around it. For example, it is south of College Walk, west of Lion's Court, north of Butler, and potentially east of Lerner if we place it just right. However, the algorithm detects that the point is closest to Butler, so including "north of Butler" is a must. Now that we have a "north" description, the system will consider east, west, and south. Again, depending on where the point is at, the system could then find that Lion's Court is the next closest relation. So, the final output will be "north of Butler, west of Lion's Court."

The hope is that choosing the related buildings that are closest will give the best descriptions. I chose to put a max of two directional descriptions (plus the "nearby" description) because using more than that seemed excessive.

Getting the Point Cloud - **getEquivalenceClass.m**

Now that we have accepted the point, created its building, setup its relationships, and spatially described it, we can determine what other points around this would be described in the exact same way (it's "cloud" or equivalence class). This is accomplished in the **getEquivalenceClass()** function, which returns a list of points representing the cloud.

To find equivalently described points, the algorithm starts with the selected point and expands outward in the 2D space to find points with the same description. Specifically, a queue of points is initialized with just the user designated point. Then, while the queue is not empty, the next point is removed, it's relationships to the other buildings are set, and then it is given a spatial description as described above. If it matches the spatial description of the user designated point (the first iteration through the loop has to since it was the original point), the 4 points north, east, south, and west of the point are added to the queue for processing.

The algorithm has several optimizations. First, it keeps track of which points it has already visited in a binary matrix the size of the campus map. Each pixel in the matrix is initialized to 0 and is set to 1 when visited. Looking up pixels to determine if they have been accessed is then simply. Second, the algorithm checks to see if the point being considered is within the boundaries of the map so we don't expand too far outward. And third, the algorithm checks to see if the point currently being considered is located in a different container of the labeled map. As an example, if the current point is located outside of a building while the original point was within building 12, the two points will not be described the same.

Getting Ready for Traversal in Step 4

It's also worth mentioning in this step that the pixeling building relationships were pruned and the original 27 buildings had their relationships set against the target (and prune again). We only need relationships from the original 27 buildings to the target and not the source because we won't consider paths that revisit the source (it wouldn't make sense to tell the user to go back to where they started); however, we do need relationships that let us move from the building's to the target area since that's where we want to go.

Also, if the Target was located inside a building, I replaced Target with the building it was in. From a user perspective, I think this makes sense. The person probably doesn't care about the specific area within the building - they just want to get to the building itself. As we'll see in the next step, this will also make the directions more clear.

Plotting the Point Cloud

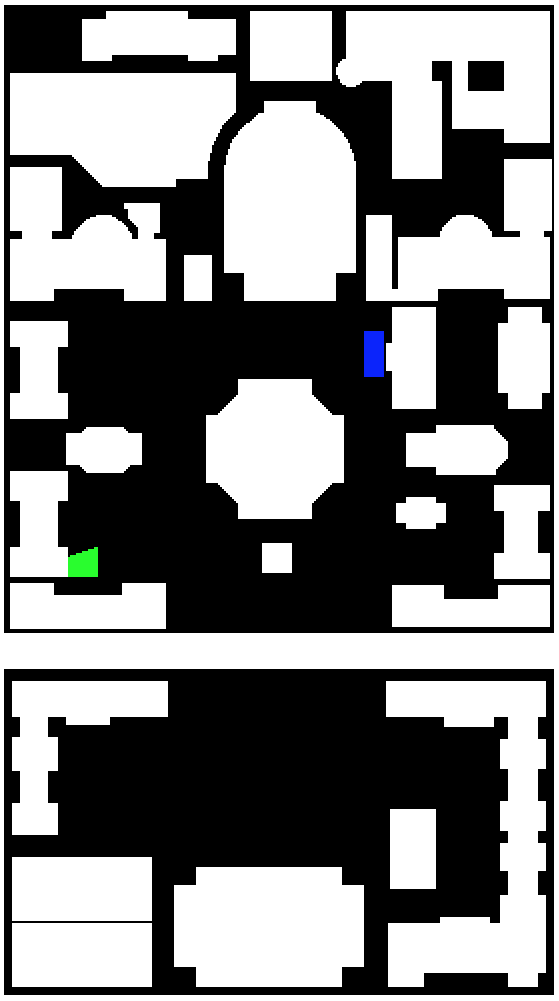
With the cloud of pixels for both the source and the target, we can color them on a map. The cloud of pixels representing the source will be colored in green while the cloud of pixels representing the target will be colored in blue.

Again, please assume pixel coordinates are given with the top left being the origin.

Example 1: Dodge to Avery

The source is in front of Dodge (42, 284). The generated description is: *near "Lewisohn", north of "Dodge" and east of "Lewisohn"*

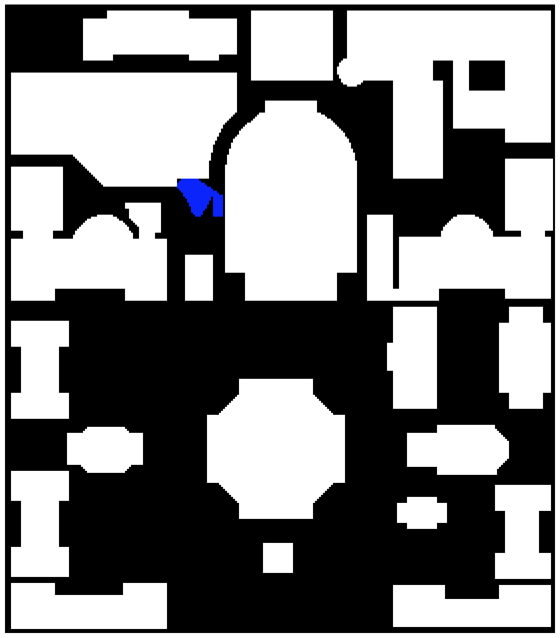
The target is in front of Avery (189, 176). The generated description is: *near "Avery", west of "Avery" and south of "Schermerhorn"*

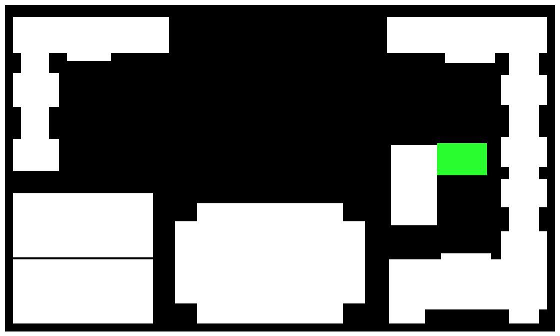


Example 2: Hamilton, Hartley, Wallach, & John Jay to the Fitness Center

The source is in front of the Hamilton buildings (235, 416). The generated description is: *near "Hamilton, Hartley, Wallach & John Jay", east of "Lion's Court" and west of "Hamilton, Hartley, Wallach & John Jay"*

The target is in front of the Fitness Center (92, 90). The generated description is: *Target Description is: near "Physical Fitness Center", west of "Gymnasium & Uris" and east of "Chandler & Havemeyer"*

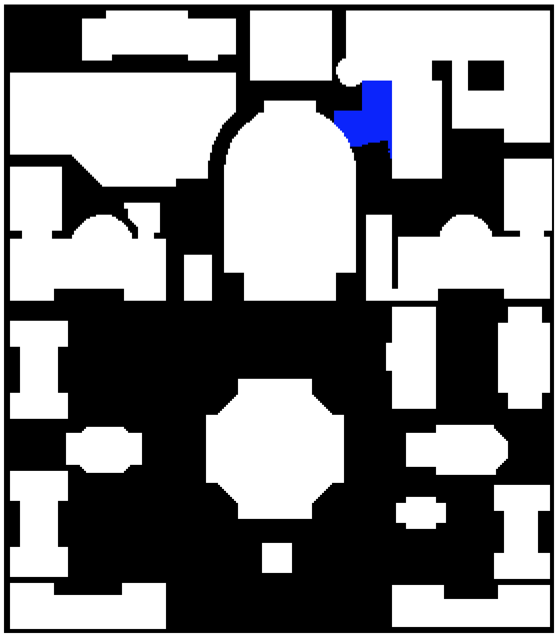


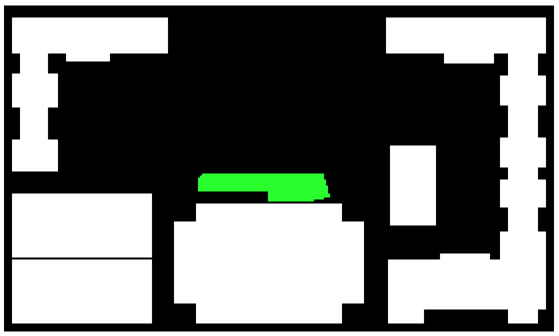


<u>Example 3: Butler to Mudd</u>

The source is the front of Butler (136, 426). The generated description is: *near "Butler Library", north of "Butler Library" and west of "Hamilton, Hartley, Wallach & John Jay"*

The target is the main entrance of Mudd (189, 45). The generated description is: *near "Mudd, Engineering Terrace, Fairchild & Computer Science", south of "Mudd, Engineering Terrace, Fairchild & Computer Science" and west of "Mudd, Engineering Terrace, Fairchild & Computer Science"*

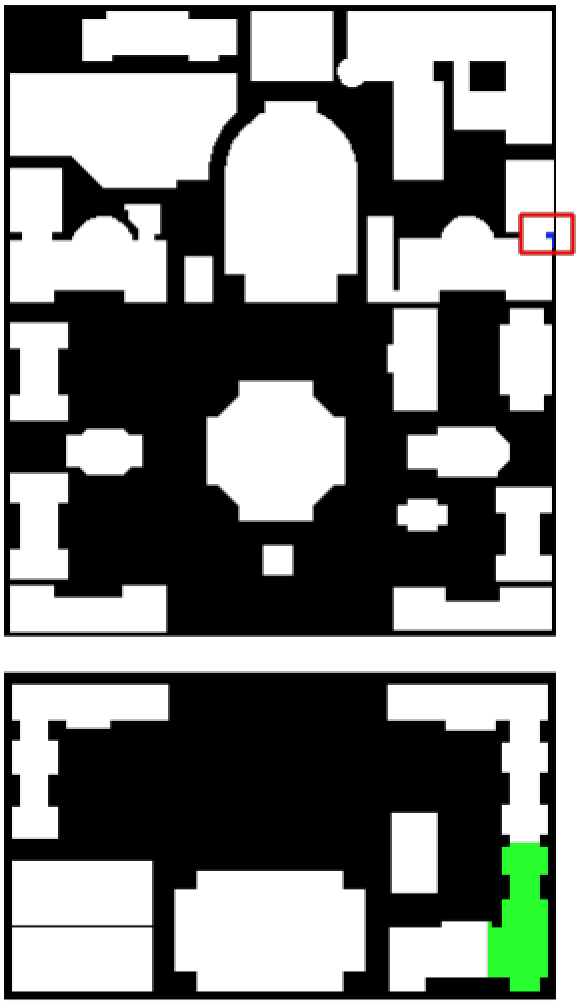




Anthony Dubis - ajd2194
Homework 3

Smallest and Largest Clouds

I experimented through a number of areas and found the largest area to be within Hamilton, Hartley, Wallach, & John Jay. Specifically, I selected point (259, 467), which resulted in the green pixel cloud. In addition, I found a small point off to the right of Schermerhorn (273, 115) that seemed to result in the smallest point. This is shown in blue (I also added a square around it because it is quite small).



The Hamilton conglomerate gives lots of space for a pixel cloud to grow. The relations are primarily built off of the building's centroid, which is roughly in the middle-right of its C shaped structure.

The smallest point is it's size because it is limited by the Schermerhorn building. It can't expand into the building because that would change its description from "near Schermerhorn" to "in Shermerhorn."

Step 4 - Path Generation

Now we have our graph setup with spatial relationships going **away from** S **to** relevant buildings and relationships going **into** T **from** relevant buildings. We can now traverse the graph to discover a path.

Getting the Path - **getPath.m**

I decided to find a shortest path by implementing a breadth first search (BFS) algorithm that first does a breadth first search and then reconstructs the path from S to T. This code is in the **getPath.m** file.

getPath() begins by calling **bfs()** which performs a breadth first search that ultimately returns a 29x3 matrix called **edgeTo**. 29 is the number of original buildings (27) plus the source and target. The ith row corresponds to the ith building where i = 28 is S and i = 29 is T. Each row contains the following four values: [fromBuilding#, directionalID, isNear]. In plain english, that means that in the ith row, this vector would tell us how we got to building i. Specifically, we got to building i by going from fromBuilding# in the direction represented by directionalID (1=N, 2=E, 3=S, 4=W). We can also tell if near(fromBuilding#, i) is true if isNear is 1. If they are not near, then isNear is 0.

The **bfs()** algorithm is essentially the standard BFS. It uses a Queue to maintain the order of the search so that we branch out in layers from S. For each node, it iterates through all of that nodes directional relationships (N, S, E, W). When these directional relationships takes us to an undiscovered node, it also checks to see if the two buildings are near to each other by using the reference building's near relationships. This means that I chose to ignore the near relationships with other buildings until I knew that the other buildings could also be described as a directional relationship.

An example might help here. If I were at Mudd and was at Low and telling a friend how to get to Avery, I wouldn't just say it's "near and somewhat rectangular." Low has near buildings all around it. How would the person know where to go? It would be much more useful to first establish that the building is **east** and near so the person had an idea of which direction to head in. So with this in mind, near relationships were ignored until I could also give a direction that supported it.

Finally, once this processing of an undiscovered adjacent node is finished, this adjacent node is added to the queue. The algorithm continues until the queue is empty or T is discovered, at which point it returns.

The algorithm of **bfs()** works by only letting us visit a building once, which is why the result of the BFS is a shortest path. It enforces this by seeing if the row (again, which represents the building number) in the **edgeTo** matrix has already been set. If it has been set, the building has already been traversed to by another building so we don't visit it again. If it hasn't been set, we fill out the appropriate details in the **edgeTo** matrix so no other building will be able to visit it. BFS continues to work in a way that branches out from S in layers. Assuming each edge is considered to be of weight 1, which I'm doing here, it will find a shortest path to T.

Note that each row i in the **edgeTo** matrix simply tells us which building we moved from to get to building i. This gets us halfway to what we want, so **reconstructPath()** is called to get us a more explicit path.

Reconstruct path starts with the T, the 29th row of the **edgeTo** matrix, and then works it's way back until it reaches the S, the 28th row of the **edgeTo** matrix. Along the way, it compiles the **path** into an Nx4 matrix where N is the number of steps it took to get from S to T. Each row of **path** represents the following [fromBuilding# directionTaken isNear toBuilding#]. Similar to the **edgeTo** matrix, the row describes the direction we took from the fromBuilding# to the toBuilding# and if near(fromBuilding, toBuilding) is true. Also, this **path** matrix is constructed in a way that the steps are ordered. So, row 1 corresponds to the move from S to buildingA, row 2 is the move from buildingA to buildingB, and so on until row N shows the final move from buildingX to T (the terminal guidance).

Describing the Path - **describePath.m**

Now that we have the **path** matrix with the details of the ordered steps, we can describe the directions in plain english to the user. This is done in the **describePath()** function.

describePath() takes the **path** matrix described above and bMap as its arguments. The **directions** are returned in an Nx1 array where N is the number of steps and each row is a string describing the move. In this step, we get to reuse the **getDescription()** function that is a part of the Building class to describe the buildings within the parentheses.

The function makes an effort to make the text more human-like. The general structure of the first N-1 steps is the following:

Go to the building that is [direction] [and near] (and that is [description])

Note that the [and near] part is optional. It will only be inserted if the **path** matrix says the two are near.

Finally, the Nth step has a couple of possibilities. If the specified Target is within a building, then the terminal guidance will be in the same format as above. However, if the Target is just an area outside of any building, the description will be someone different. Specifically, it will be structured as follows:

Go to the area that is [direction] [and near]

Again, the [and near] part is option, but it is highly likely since the algorithm typically ends at a building right next to T.

Specifying Eight Paths

Now we can specify eight (S, T) pairs and give directions from S to T. The pairs I have chosen are as follows:

1. From the entrance of Mudd to the front of Butler
2. From the John Jay area to inside the Fitness Center
3. From the entrance of Uris to the front of Kent
4. From Lerner Hall to Pupin
5. From inside Mathematics to inside Fayerweather
6. From Carmen to Alma Mater
7. From Shapiro to the area north of Lion's Court
8. From Butler to St. Paul's Chapel

You'll notice that some of these paths can expose some weakness in the system, rather than selectively choosing paths that resulted in super clear directions. I considered that a good thing and worth discussing for future improvements.

I'm going to include the user results right into the sections containing each path, its points, and its descriptions. I was lucky enough to have friends in town the weekend I was doing this project because they know just about nothing about Columbia's campus. **User 1** was given the "where-only" descriptions for paths 1-4 and the "full" descriptions for paths 5-8. **User 2** was given the "full" descriptions for paths 1-4 and the "where-only" descriptions for paths 5-8.

In the headers below, you'll notice the exact (x, y) points selected as the Source and Target are in (parentheses). Again, the origin is at the top left of the map, so y increases as you move down the map.

Also, there are three images associated with each path. The first just shows the source and target clouds. The second shows User 1's clicks. The third shows User 2's clicks.

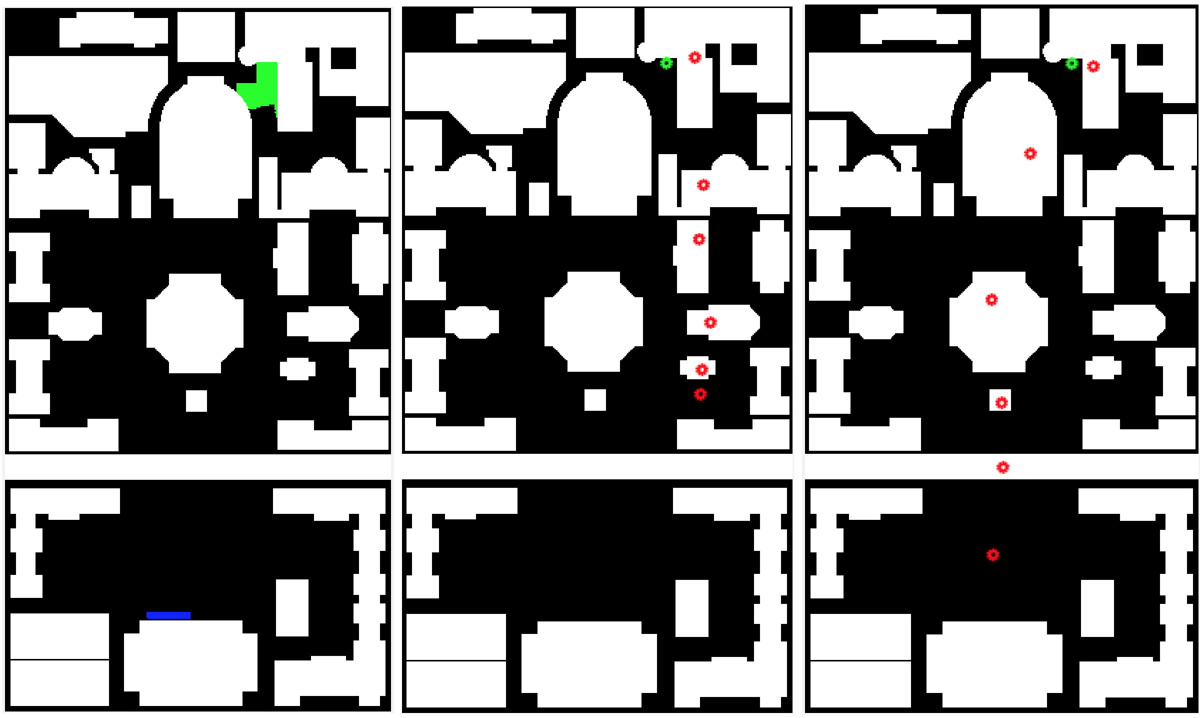
The program/code to collect and plot the user clicks can be seen in **userStudy.m**

Anthony Dubis - ajd2194
Homework 3

From the entrance of Mudd (187, 42) to the front of Butler (132, 429)

Full Directions
Go to the building that is north and near (and that is Large in size, oriented west-to-east, irregular in shape with a hole, and in the northeast part of campus.)'
Go to the building that is south and near (and that is Large in size, oriented north-to-south, contains a large curved room, and in the north-center part of campus.)'
Go to the building that is south (and that is Large in size, oriented north-to-south, cross shaped, and in the center part of campus.)'
Go to the building that is south and near (and that is Smallest in size, oriented west-to-east, square in shape, and in the center part of campus.)'
Go to the building that is south (and that is Large in size, oriented west-to-east, divides campus, and in the center part of campus.)'
Go to the area that is south.'

Where-Only Directions
Go to the building that is north and near
Go to the building that is south and near
Go to the building that is south
Go to the building that is south and near
Go to the building that is south
Go to the area that is south.'


S & T (left), User 1 where-only (center), User 2 full directions (right)

As we can see, User 1 really struggled to get to the destination. Without having the building descriptions, the user didn't realize that the first move south was supposed to go to Uris, which put them on a bad course for the rest of the path.

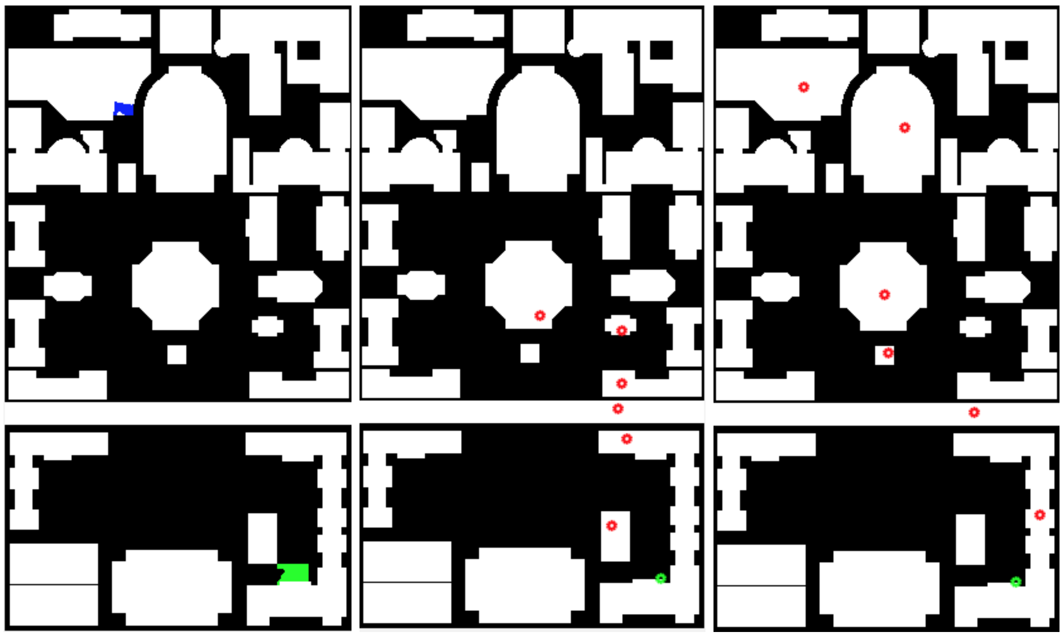
While User 2 got close, they didn't quite get as close to the entrance of Butler as we would like. More descriptions would be needed to quantify how far south into an area the user should go.

Anthony Dubis - ajd2194
Homework 3

From the John Jay area (241, 456) to inside the Fitness Center (95, 83)

Full Directions
'Go to the building that is north and near (and that is Large in size, oriented north-to-south, C-shaped, and in the southeast part of campus.)'
'Go to the building that is north and near (and that is Large in size, oriented west-to-east, divides campus, and in the center part of campus.)'
'Go to the building that is north (and that is Smallest in size, oriented west-to-east, square in shape, and in the center part of campus.)'
'Go to the building that is north and near (and that is Large in size, oriented north-to-south, cross shaped, and in the center part of campus.)'
'Go to the building that is north (and that is Large in size, oriented north-to-south, contains a large curved room, and in the north-center part of campus.)'
'Go to the building that is west and near (and that is Large in size, oriented west-to-east, irregular in shape, and in the northwest part of campus.)'

Where-Only Directions
'Go to the building that is north and near
'Go to the building that is north and near
'Go to the building that is north
'Go to the building that is north and near
'Go to the building that is north
'Go to the building that is west and near


S & T (left), User 1 where-only (center), User 2 full directions (right)

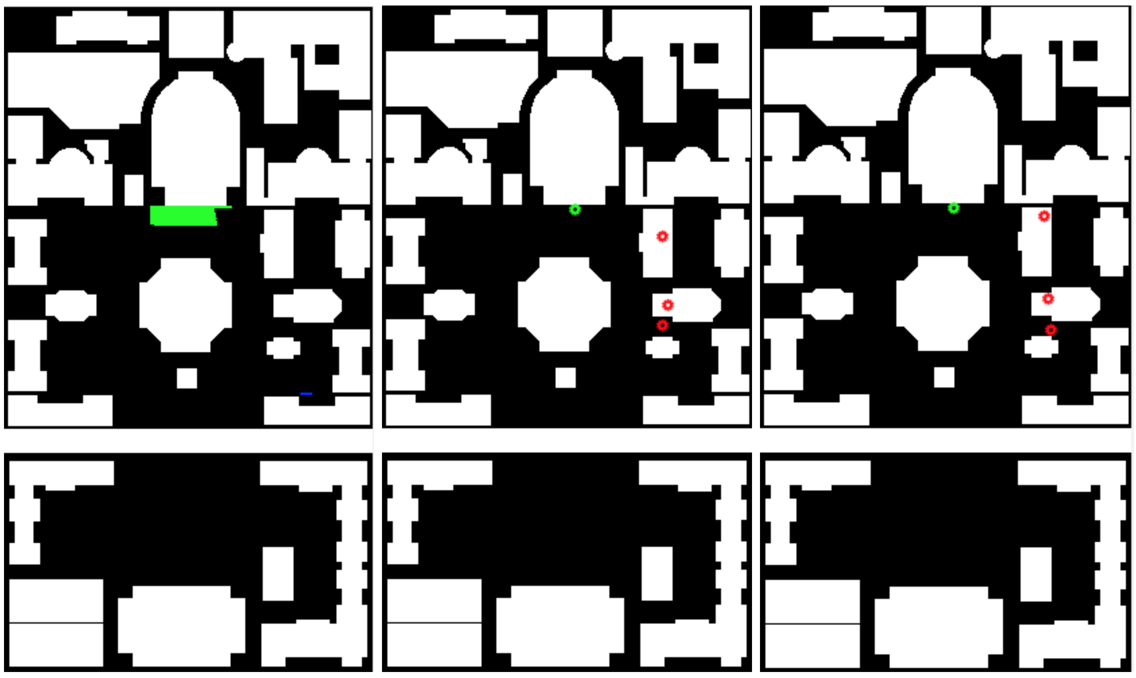
Again, without the building descriptions, User 1 didn't quite make the right building choices. Specifically, she got caught going through Lion's Den and then missed the route that went through Alma Mater.

User 2 made it!

From the entrance of Uris (144, 152) to the front of Kent (227, 289)

Full Directions
'Go to the building that is east (and that is Small in size, oriented north-to-south, close to a rectangle in shape, and in the mid-east part of campus.)'
'Go to the building that is south and near (and that is Small in size, oriented west-to-east, close to a rectangle in shape, and in the mid-east part of campus.)'
'Go to the area that is south.'

Where-Only Directions
'Go to the building that is east
'Go to the building that is south and near
'Go to the area that is south.'


S & T (left), User 1 where-only (center), User 2 full directions (right)

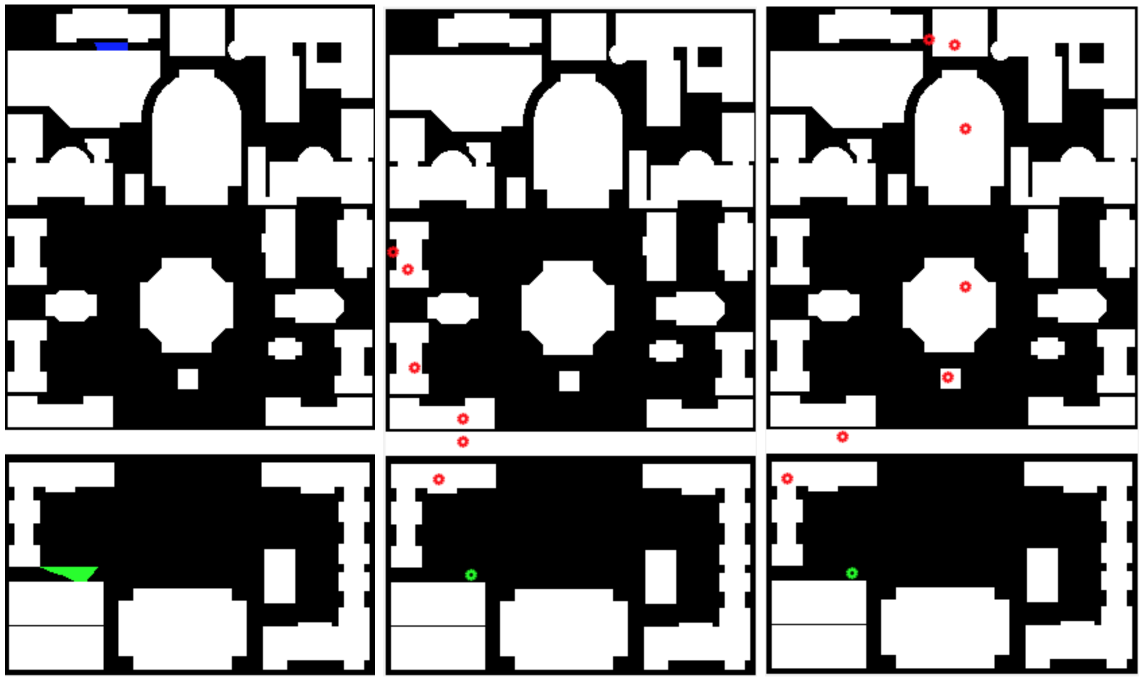
Interestingly, both users made quite similar moves whether they had the building descriptions or not. This is probably because the buildings are so close to each other that the building choice while going east or south was pretty obviously, but the system did a poor job of describing the appropriate buildings (it wasn't intended for the users to go through St. Paul's Chapel). Better descriptions for St. Pauls and Buell would have helped here.

Anthony Dubis - ajd2194
Homework 3

From Lerner Hall (64, 421) to Pupin (73, 29)

Full Directions
'Go to the building that is north and near (and that is Medium in size, L-shaped, and in the southwest part of campus.)'
'Go to the building that is north and near (and that is Large in size, oriented west-to-east, divides campus, and in the center part of campus.)'
'Go to the building that is north (and that is Smallest in size, oriented west-to-east, square in shape, and in the center part of campus.)'
'Go to the building that is north and near (and that is Large in size, oriented north-to-south, cross shaped, and in the center part of campus.)'
'Go to the building that is north (and that is Large in size, oriented north-to-south, contains a large curved room, and in the north-center part of campus.)'
'Go to the building that is north and near (and that is Small in size, oriented west-to-east, rectangle in shape, and in the north-center part of campus.)'
Go to the area that is west.'

Where-Only Directions
'Go to the building that is north and near
'Go to the building that is north and near
'Go to the building that is north
'Go to the building that is north and near
'Go to the building that is north
'Go to the building that is north and near
Go to the area that is west.'


S & T (left), User 1 where-only (center), User 2 full directions (right)

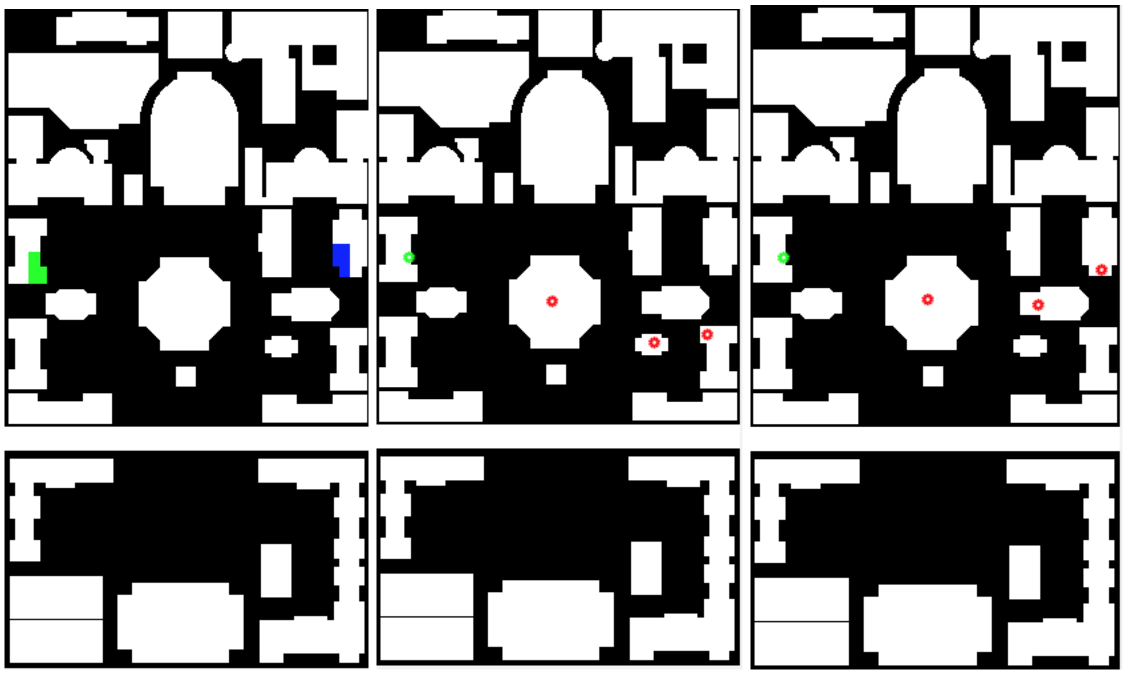
Once again, not having the building descriptions led User 1 to make choices that left her behind the Mathematics building. I think that because the starting point was at Lerner, going "north" still meant staying on the left side of the map (even though College Walk spans the entire map). It's just a human thing to do.

With the building descriptions, User 2 was able to properly veer eastward to Alma Mater and Low, and then up to a final destination that is pretty close to the entrance of Pupin.

From inside Mathematics (26, 189) to inside Fayerweather (251, 184)

Full Directions
'Go to the building that is east (and that is Large in size, oriented north-to-south, cross shaped, and in the center part of campus.)'
'Go to the building that is east and near (and that is Small in size, oriented north-to-south, close to a rectangle in shape, and in the mid-east part of campus.)'
'Go to the building that is east(and that is Small in size, oriented north-to-south, close to a rectangle in shape, and in the mid-east part of campus.)'

Where-Only Directions
'Go to the building that is east
'Go to the building that is east and near
'Go to the building that is east


S & T (left), User 1 full directions (center), User 2 where-only (right)

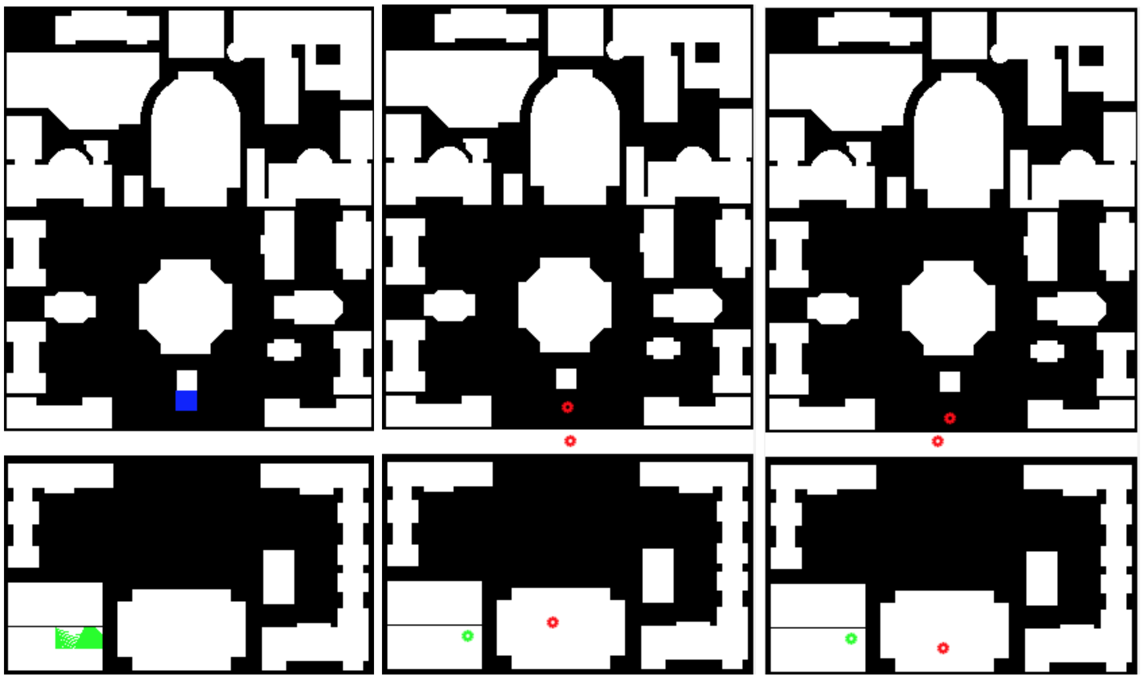
The directions here are clearly a bit vague. For example, going east from St. Paul could mean going slightly upwards to Fayerweather, or slightly downwards to Philosophy. User 1 took the wrong turns here despite having the building descriptions. I think this shows how the buildings in this area were not properly distinguish from each other - in other words, they had very similar descriptions.

User 2 with the where-only description actually made it to Fayerweather, but I think this is more out of lucky, 50-50 clicks.

From Carmen (64, 467) to Alma Mater (138, 288)

Full Directions
'Go to the building that is east (and that is Large in size, oriented west-to-east, close to a rectangle in shape, and in the south-center part of campus.)'
'Go to the building that is north (and that is Large in size, oriented west-to-east, divides campus, and in the center part of campus.)'
'Go to the area that is north.'

Where-Only Directions
'Go to the building that is east
'Go to the building that is north
'Go to the area that is north.'

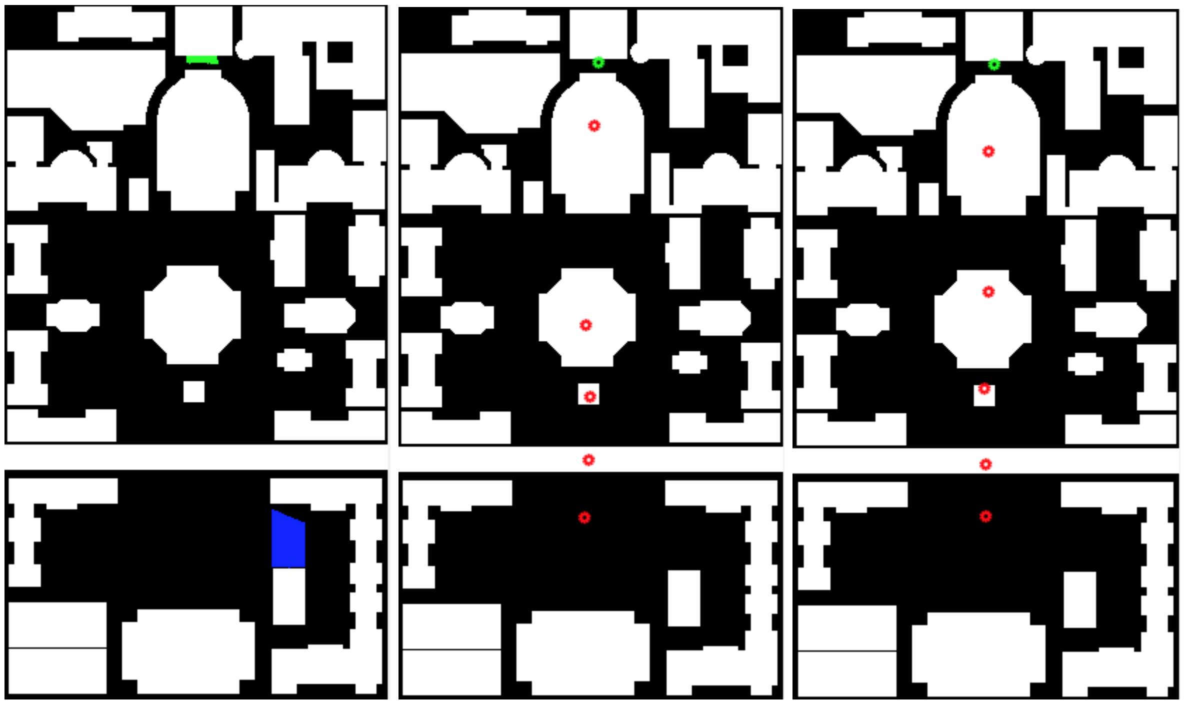

S & T (left), User 1 full directions (center), User 2 where-only (right)

Both users made great progress here! I think it was because the directions were fairly straight forward, even without building descriptions. Butler is the obvious choice when going east from Carmen, and going North a few times puts you right in the area in front of Alma Mater.

From Shapiro (144, 41) to the area north of Lion's Court (201, 398)

Full Directions
'Go to the building that is south and near (and that is Large in size, oriented north-to-south, contains a large curved room, and in the north-center part of campus.)'
'Go to the building that is south (and that is Large in size, oriented north-to-south, cross shaped, and in the center part of campus.)'
'Go to the building that is south and near (and that is Smallest in size, oriented west-to-east, square in shape, and in the center part of campus.)'
'Go to the building that is south (and that is Large in size, oriented west-to-east, divides campus, and in the center part of campus.)'
'Go to the area that is south.'

Where-Only Directions
'Go to the building that is south and near
'Go to the building that is south
'Go to the building that is south and near
'Go to the building that is south
'Go to the area that is south.'

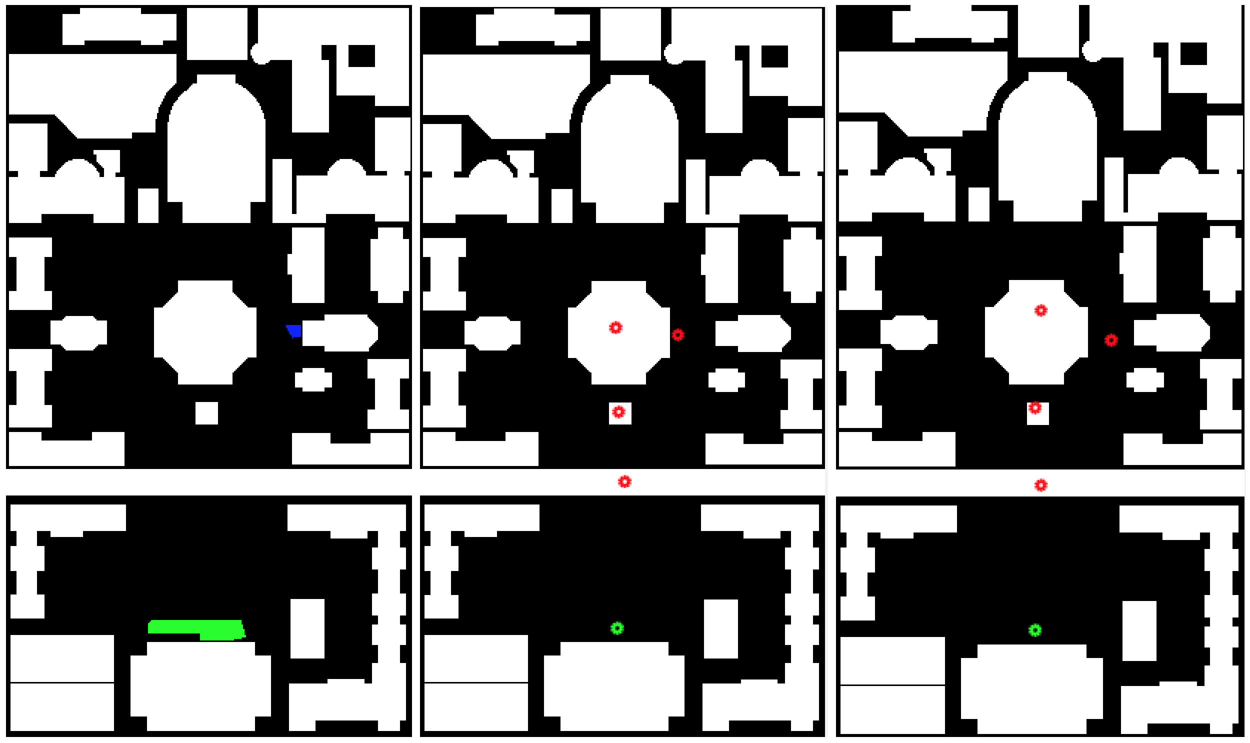

S & T (left), User 1 full directions (center), User 2 where-only (right)

Both users essentially followed the same path here, and it's not too hard to see why when the directions basically just say to keep going south. However, this did not get them all that close to the northern area of Lion's Court. The system probably should have taken them to Lion's Court instead and then moved them north (or found some way of saying go to the "southeast area from College Walk").

From Butler (135, 422) to St. Paul's Chapel (198, 223)

Full Directions
'Go to the building that is north (and that is Large in size, oriented west-to-east, divides campus, and in the center part of campus.)'
'Go to the building that is north (and that is Smallest in size, oriented west-to-east, square in shape, and in the center part of campus.)'
'Go to the building that is north and near (and that is Large in size, oriented north-to-south, cross shaped, and in the center part of campus.)'
'Go to the area that is east and near.'

Where-Only Directions
'Go to the building that is north
'Go to the building that is north
'Go to the building that is north and near
'Go to the area that is east and near.'


S & T (left), User 1 full directions (center), User 2 where-only (right)

Both users did well here! Again, though, these directions were pretty straight forward and the where-only descriptions were probably good enough (go north a few buildings, then go to the area in the east). Oh well, a win is a win.

Conclusion

Overall, we could get away with just having the “where-only” descriptions if they were relatively true-to-form. In other words, if the buildings/areas you were going to were truly in one of the four basic directions and didn’t have anything in the way, we could get there without issue. However, as soon as we started heading in directions that weren’t quite as clear (NE, SE, SW, NW), the descriptions became invaluable. However, as we saw in the Avery/Fayerweather area, the descriptions need to be clear enough to distinguish nearby buildings from each other.

Code

All code is my own - my only reference was MATLAB's online documentation

hw3.m

```
clc; clear; close all;

%% Step 0 - Get the campus map, a BW represention, and a labeled version

campus  = imread('supporting/ass3-campus.pgm');
BW      = im2bw(campus, 0);
% figure; imshow(BW); hold on;
labeled = imread('supporting/ass3-labeled.pgm');
labeled = correctedLabeled(labeled);

% Map the building numbers to their names
b_names = containers.Map();
strs = textread('supporting/ass3-table.txt', '%s', 'delimiter', '\n');
N = length(strs);
for i=1:N
    str = strs{i};
    len = length(str);
    eqls = strfind(str, '=');
    b_names(str(1:eqls-1)) = str(eqls+1:len);
end

%% Step 1 - Set the building features and descriptions

% Get the desired region properties for the buildings
stats = regionprops(BW, 'Area', 'BoundingBox', 'Centroid', ...
    'MajorAxisLength', 'MinorAxisLength', 'Orientation', 'PixelList', ...
    'ConvexHull', 'ConvexArea', 'Image');

% Set the building objects
bMap = containers.Map();
areaMin = double(intmax);
areaMax = 0.0;

for i=1:N
    b_stats = stats(i);
    b = Building;

    % Get a point within the building to set the building name and number
    pt = b_stats.PixelList(1,:);
    b.number = labeled(pt(2), pt(1));
    b.name = b_names(int2str(b.number));

    % Set basic properties
    b.area = b_stats.Area;
    b.centroid = b_stats.Centroid;
    b = b.setOrientation(b_stats.Orientation);
    b = b.setBoundingBox(b_stats.BoundingBox);
    b = b.setImage(b_stats.Image);
    b.shape = determineShape(b, labeled);
    b.region = getRegion(b.centroid, labeled);
    bMap(int2str(b.number)) = b;
```

Code

All code is my own - my only reference was MATLAB's online documentation

```
    areaMin = min(areaMin, b.area);
    areaMax = max(areaMax, b.area);
end

% Set the size attribute
for i=1:N
    b = bMap(int2str(i));
    b.buildingSize = getSizeDescription(b.area, areaMin, areaMax);
    bMap(int2str(i)) = b;
end


%% Step 2 - Set the spatial relationships between buildings
% Establish relationships between all buildings
for i=1:N
    for j=1:N
        if i == j; continue; end
        S = bMap(int2str(i));
        T = bMap(int2str(j));
        [S, T] = setSpatialRelationships(S, T);
        bMap(int2str(i)) = S;
        bMap(int2str(j)) = T;
    end
end

% Prune the graph to remove relationships that can be inferred
for i=1:N
    bMap = pruneRelationships(bMap, bMap(int2str(i)));
end

%% Step 3 - Setting and describing sources & targets
% Turn campus into RBG
rgb = campus(:,:,[1 1 1]);

% Get the source (S) and it's description *********************************
figure(); imshow(rgb);
sLoc = int16(ginput(1));
S = buildingForPoint(bMap, 28, 'Source', sLoc);
sDesc = getBuildingSpatialDesc(bMap('28'), bMap, labeled);
fprintf('Source Description is: %s\n', sDesc{1});

% Get the equivalence class of pixels surrounding S and color them green
cloud = getEquivalenceClass(S, bMap, sDesc, labeled);
while ~cloud.isEmpty()
    pt = cloud.remove();
    rgb(pt(2),pt(1),:) = [0 255 0];
end
imshow(rgb);

% Get S ready for graph traversal
bMap = pruneRelationships(bMap, bMap(int2str(28)));

% Get the target (T) and it's description *********************************
```

Code

All code is my own - my only reference was MATLAB’s online documentation

```
tLoc = int16(ginput(1));
T = buildingForPoint(bMap, 29, 'Target', tLoc);
tDesc = getBuildingSpatialDesc(bMap('29'), bMap, labeled);
fprintf('Target Description is: %s\n', tDesc{1});

% Get the equivalence class of pixels surrounding S and color them green
cloud = getEquivalenceClass(T, bMap, tDesc, labeled);
while ~cloud.isEmpty()
    pt = cloud.remove();
    rgb(pt(2),pt(1),:) = [0 0 255];
end
imshow(rgb);

% Get S ready for graph traversal
if labeled(T.centroid(2), T.centroid(1)) ~= 0
    % T is in a building - let's make the building the target
    newTarget = bMap(int2str(labeled(T.centroid(2), T.centroid(1))));
    newTarget.number = 29;
    bMap('29') = newTarget;
end

bMap = pruneRelationships(bMap, bMap(int2str(29)));
for i=1:N
    B = bMap(int2str(i));
    [B, T] = setSpatialRelationships(B, T);
    bMap(int2str(i)) = B;
end
% Reprune to get rid of relationships to T that can be inferred
for i=1:N
    bMap = pruneRelationships(bMap, bMap(int2str(i)));
end

%% Step 4

% Get the path
path = getPath(bMap, S, T);

% Describe the path
directions = describePath(bMap, path);
```

Code

All code is my own - my only reference was MATLAB’s online documentation

Building.m

```matlab
classdef Building
    % This class represents a building on a map, its description, and its
    % relationships to other buildings

    properties
        % Numeric properties
        number;
        name;
        area;
        centroid;
        orientation;
        boundingBox;
        spatialPts; % Used to determine N,S,E,W relationships
        expandedBoundingBox;
        image;
        corners;
        boundaries;
        numTurns;
        numLargeCircles = 0;
        numMediumCircles = 0;

        % Text descriptions
        buildingSize;
        oriented;
        shape;
        region;

        % Spacial relationships
        north;
        south;
        east;
        west;
        near;
    end

    methods
        function desc = getDescription(obj)
            desc = strcat(obj.buildingSize, {', '});
            if abs(obj.orientation) < 20 || abs(obj.orientation) > 70
                desc = strcat(desc, obj.oriented, {', '});
            end
            desc = strcat(desc, obj.shape, {', and in the '}, ...
                obj.region, {' part of campus.'});
        end

        function obj = setOrientation(obj, b_orientation)
            obj.orientation = b_orientation;
            if b_orientation < 45 && b_orientation > -45
                obj.oriented = 'oriented west-to-east';
            else
                obj.oriented = 'oriented north-to-south';
```

Code

All code is my own - my only reference was MATLAB’s online documentation

```
        end
    end

    function obj = setBoundingBox(obj, rec)
        obj.boundingBox = rec;

        % Set spatialPts
        pts = zeros(4,2);
        pts(1,:) = [rec(1)+rec(3)/2, rec(2)]; % North pt
        pts(2,:) = [rec(1)+rec(3), rec(2)+rec(4)/2]; % East pt
        pts(3,:) = [rec(1)+rec(3)/2, rec(2)+rec(4)]; % South pt
        pts(4,:) = [rec(1), rec(2)+rec(4)/2]; % West pt
        obj.spatialPts = pts;

        % Set expandedBoundingBox
        scale_factor = 2.1;
        minIncrease = 30;
        maxIncrease = 70;
        % Determine new width with min/max boundaries
        newW = rec(3) * scale_factor;
        newW = max(newW, rec(3)+minIncrease);
        newW = min(newW, rec(3)+maxIncrease);
        % Determine new height with min/max boundaries
        newH = rec(4) * scale_factor;
        newH = max(newH, rec(4)+minIncrease);
        newH = min(newH, rec(4)+maxIncrease);
        % Determine new origin
        newX = rec(1) - ((newW - rec(3)) / 2);
        newY = rec(2) - ((newH - rec(4)) / 2);
        obj.expandedBoundingBox = [newX, newY, newW, newH];
    end

    function obj = setImage(obj, image)
        obj.image = image;
        obj = obj.setBoundaries(bwboundaries(image));

        [centers, ~] = imfindcircles(image, [20, 35]);
        obj.numLargeCircles = size(centers,1);

        [centers, ~] = imfindcircles(image, [10, 20]);
        obj.numMediumCircles = size(centers,1);
    end

    function obj = setBoundaries(obj, boundaries)
        obj.boundaries = boundaries;
        n_turns = 1;

        for i=1:length(boundaries)
            vert_line = false;
            horz_line = false;
            bounds = boundaries{i};
            sz = size(bounds,1);
            prev = bounds(sz,:);
            for j=1:sz
```

Code
All code is my own - my only reference was MATLAB’s online documentation

```
                    curr = bounds(j,:);
                    if ~vert_line && ~horz_line
                        if     (prev(1) == curr(1)); horz_line = true;
                        elseif (prev(2) == curr(2)); vert_line = true;
                        else   n_turns = n_turns + 1;
                        end
                    elseif vert_line
                        if prev(2) ~= curr(2)
                            vert_line = false;
                            n_turns = n_turns + 1;
                        end
                    else
                        if prev(1) ~= curr(1)
                            horz_line = false;
                            n_turns = n_turns + 1;
                        end
                    end
                    prev = curr;
                end
            end
            obj.numTurns = n_turns;
        end
    end

end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

buildingForPoint.m

```
function [bld, bMap] = buildingForPoint(bMap, num, name, loc)
% Create and setup a new building object for a point

side = 1;

N = 27; % Number of buildings on campus.
bld = Building;
bld.number = num;
bld.name = name;
bld.centroid = loc;
bld = bld.setBoundingBox([loc(1)-(side/2) loc(2)-(side/2) side side]);
bMap(int2str(num)) = bld;
for i=1:N
    [bld, T] = setSpatialRelationships(bld, bMap(int2str(i)));
    bMap(int2str(num)) = bld;
    bMap(int2str(i)) = T;
end
end
```

correctedLabel.m

```
function [ corrected ] = correctedLabeled( L )
%Used to get labeled graph with buildings 1-27

label = 1;
for i=1:255
    if ismember(i,L)
        L(L == i) = label;
        label = label + 1;
    end
end

corrected = L;

end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

describePath.m

```matlab
function [ directions ] = describePath( bMap, path )
steps = size(path,1);
directions = cell(steps, 1);

for i=1:(steps-1)
    info = path(i,:);
    b = bMap(int2str(path(i,4)));

    % Start the move
    move = strcat({'Go to the building that is '}, ...
        getDirectionFromIdentifer(info(2)), {' '});

    % Add near if applicable
    if info(3) == 1
        move = strcat(move, {'and near '});
    end

    % Describe the building
    move = strcat(move, {'(and that is '}, ...
        b.getDescription, {')'});

    directions{i} = move;
end

% Give terminal directions
info = path(steps,:);

if ~strcmp(bMap('29').name, 'Target')
    % We are giving directions to a building
    terminal = strcat({'Go to the building that is '}, ...
        getDirectionFromIdentifer(info(2)));
    if info(3) == 1
        terminal = strcat(terminal, {' and near '});
    end

    % Give building description
    terminal = strcat(terminal, {'(and that is '}, ...
        bMap(int2str(29)).getDescription, {')'});
else
    % We are giving directions to an area
    terminal = strcat({'Go to the area that is '}, ...
        getDirectionFromIdentifer(info(2)));
    if info(3) == 1
        terminal = strcat(terminal, {' and near'});
    end
    terminal = strcat(terminal, '.');
end
directions{steps} = terminal;

end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

describeShape.m

```
function [ shape ] = determineShape( b, labeled )
% Determine the shape of the building
% b is a Building object

box = b.boundingBox;
buildingArea = b.area;
boundingArea = getBoxArea(box);

if b.numLargeCircles > 0
    shape = 'contains a large curved room';
elseif b.numMediumCircles > 0
    shape = 'contains a medium sized curved room';
elseif dividesCampus(b, labeled)
    shape = 'divides campus';
elseif buildingArea == boundingArea
    if isSquare(b.boundingBox)
        shape = 'square in shape';
    else
        shape = 'rectangle in shape';
    end
elseif isIShaped(b)
    shape = 'I-shaped';
elseif hasHole(b)
    shape = 'irregular in shape with a hole';
elseif buildingArea / boundingArea >= .85
    shape = 'close to a rectangle in shape';
elseif isLShaped(b)
    shape = 'L-shaped';
elseif isCShaped(b)
    shape = 'C-shaped';
elseif isCrossShaped(b)
    shape = 'cross shaped';
else
    shape = 'irregular in shape';
end
end



%% Helper functions

% Gets the area of the bounding box
function area = getBoxArea(p)
area = (p(3) * p(4));
end

function doesDivide = dividesCampus(b, labeled)
doesDivide = false;
img = b.image;
if (size(img,1) == size(labeled,1)) || (size(img,2) == size(labeled,2))
```

Code

All code is my own - my only reference was MATLAB’s online documentation

```
    doesDivide = true;
end
end

% Determines if the bounding box is a square
function isSqr = isSquare(p)
isSqr = p(3) == p(4);
end

function isI = isIShaped(b)
isI = false;
img = b.image;
x_sz = size(img,2);
y_org = round(size(img,1) / 2);

if fillsCorners(b)
    if ~(img(y_org, 1)) && ~(img(y_org, x_sz))
        isI = true;
    end
end
end

function fills = fillsCorners(b)
fills = false;
img = b.image;
x_sz = size(img, 2);
y_sz = size(img, 1);
if img(1,1) && img(1,x_sz) && img(y_sz,1) && img(y_sz,x_sz)
    fills = true;
end
end

function hole = hasHole(b)
hole = false;
numBoundaries = length(b.boundaries);
if numBoundaries > 1
    hole = true;
end
end

function isL = isLShaped(b)
isL = false;
img = b.image;

x_rng = round(size(img, 2) / 2 - 1);
y_rng = round(size(img, 1) / 2 - 1);

if (sum(sum(img(1:y_rng, 1:x_rng))) == 0) ...
        || (sum(sum(img(y_rng:y_rng*2, 1:x_rng))) == 0) ...
        || (sum(sum(img(1:y_rng, x_rng:x_rng*2))) == 0) ...
        || (sum(sum(img(y_rng:y_rng*2, x_rng:x_rng*2))) == 0)
    isL = true;
end
end
```

Code

All code is my own - my only reference was MATLAB's online documentation

```
function isC = isCShaped(b)
isC = false;
img = b.image;

x_rng = round(size(img, 2) / 2 - 1);
x_org = round(x_rng / 2);
y_rng = round(size(img, 1) / 2 - 1);
y_org = round(y_rng / 2);

if (sum(sum(img(y_org:y_rng, 1:x_rng))) == 0) ...
        || (sum(sum(img(y_org:y_rng, x_org:x_rng))) == 0)
    isC = true;
end
end

function crossShaped = isCrossShaped(b)
crossShaped = false;
if ~fillsCorners(b)
    img = b.image;
    y_sz = size(img,1);
    x_sz = size(img,2);
    y_sz_half = round(y_sz / 2);
    x_sz_half = round(x_sz / 2);
    if img(1, x_sz_half) && img(y_sz_half, 1) && img(y_sz_half, x_sz) ...
            && img(y_sz, x_sz_half)
        crossShaped = true;
    end
end
end
```

Code

All code is my own - my only reference was MATLAB's online documentation

getAdjacentNodes.m

```
function adj = getAdjacentNodes(building, dir)
switch dir
    case 'east';  adj = building.east;
    case 'west';  adj = building.west;
    case 'north'; adj = building.north;
    case 'south'; adj = building.south;
end
end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

getBuildingSpatialDesc.m

```
function [ desc ] = getBuildingSpatialDesc( b, bMap, labeled )
% Return the spatial description of building b based on the graph made up
% of the other buildings in bMap

% First, determine if this building is 'in' another or if you need to find
% what is nearby. If you just need to find what is nearby, grab anything
% from the "near" relationship set.
desc = getNearbyDescription(b, bMap, labeled);

% Next, find the two closets relationships. Closest is defined as being
% closest to the spatialPt. The two results with related buildings that are
% the closest to b should be used in the description. The returned matrix
% will have two rows with the first column corresponding to the distance
% (1=North, 2=East, 3=South, 4=West) and the second the building number. If
% the row contains all zeros, then a good relationship couldn't be found.
R = getMostRelevantRelationships(b, bMap);

% Now apped these relevant relationships to the desc
directionalDesc = getDirectionalDescriptionForRelations(bMap, R);

if isempty(desc)
    desc = directionalDesc;
else
    desc = strcat(desc, {', '}, directionalDesc);
end

end

function desc = getNearbyDescription(b, bMap, labeled)
cent = b.centroid;
val = labeled(cent(2), cent(1));
if val ~= 0
    desc = strcat('in', {' '}, bMap(int2str(val)).name, {' '});
else
    nearAdj = bMap(int2str(b.number)).near;
    if isempty(nearAdj)
        desc = '';
    else
        desc = strcat('near', {' '}, bMap(int2str(nearAdj(1))).name);
    end
end
end

function R = getMostRelevantRelationships(b, bMap)
north = getMostRelevantRelationship(b, bMap, 'north');
east  = getMostRelevantRelationship(b, bMap, 'east');
south = getMostRelevantRelationship(b, bMap, 'south');
west  = getMostRelevantRelationship(b, bMap, 'west');
R = [north; east; south; west];

% Sort them
```

Code

All code is my own - my only reference was MATLAB’s online documentation

```
[~,I] = sort(R(:,2));
R = R(I,:);

% Remove relationships that don't exist
while ~isempty(R) && R(1,2) == 0
    len = size(R,1);
    R = R(2:len,:);
end

% Return only the two most relevant
if size(R,1) > 2
    R = R(1:2,:);
end
end

% This is going to return a 1 x 3 vector. The first column will specify the
% the direction (1=N, 2=E, 3=S, 4=W). The second column will contain the
% distance between b and the nearest building with respect to the given
% direction (north, south, east, west). The third column will the building
% number.
function results = getMostRelevantRelationship(b, bMap, dir)
rid = getReverseDirectionIdentifier(dir);
results = [rid 0 0];

adj = getAdjacentNodes(b, dir);
len = length(adj);
if len > 0
    minDist = intmax;
    for i=1:len
        b2 = bMap(int2str(adj(i)));
        dist = distanceBetweenBuildings(b, b2);
        if dist < minDist
            results(2:3) = [dist b2.number];
            minDist = dist;
        end
    end
end
end

% Returns the minimum distance between b1 and b2 by comparing b1's centroid
% to b2's four spatial points
function dist = distanceBetweenBuildings(b1, b2)
dist = intmax;
b1Cent = double(b1.centroid);
b2Pts = double(b2.spatialPts);
for i=1:4
    dist = min(pdist([b1Cent; b2Pts(i,:)], 'euclidean'), dist);
end
end

function desc = getDirectionalDescriptionForRelations(bMap, R)
desc = '';
for i=1:size(R,1)
    seg = getDirectionFromIdentifer(R(i,1));
```

Code

All code is my own - my only reference was MATLAB’s online documentation

```
    seg = strcat(seg, {' '}, 'of', {' '}, bMap(int2str(R(i,3))).name);
    if i==1
        desc = seg;
    else
        desc = strcat(desc, {' and '}, seg);
    end
end
end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

getDirectionFromIdentifer.m

```
function [ direction ] = getDirectionFromIdentifer( id )
% Return direction string for identifier
switch id
    case 1
        direction = 'north';
    case 2
        direction = 'east';
    case 3
        direction = 'south';
    case 4
        direction = 'west';
end
end
```

getDirectionIdentifer.m

```
function [ id ] = getDirectionIdentifier( direction )
% Returns an integer identifier for a direction
switch direction
    case 'north'
        id = 1;
    case 'east'
        id = 2;
    case 'south'
        id =  3;
    case 'west'
        id =  4;
end
end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

getEquivalenceClass.m

```
function [ pts ] = getEquivalenceClass(bOrig, bMap, descOrig, labeled)
% Determines what other pixels share the same description as bOrig
fprintf('Computing the pixel cloud. Please wait...\n');

bldNum = labeled(bOrig.centroid(2), bOrig.centroid(1));
queue = java.util.LinkedList();
visited = zeros(size(labeled)); % Will contain 1 for points we've visited
pts = java.util.LinkedList(); % Points in the equivalance class

queue.add(bOrig.centroid);

while ~queue.isEmpty()
    % Get the next point
    pt = queue.remove()';

    % Check to see if the point is valid and we haven't visited it before
    if visited(pt(2), pt(1)) || isInvalidPt(pt,labeled) ...
            || changedPerspective(pt, bldNum, labeled)
        continue;
    end

    % Process the point
    [~, bMap] = buildingForPoint(bMap, 30, 'Dummy', pt);
    desc = getBuildingSpatialDesc(bMap('30'), bMap, labeled);
    if strcmp(descOrig, desc)
        pts.add(pt);
        % Add the four surrounding points
        queue.add([pt(1)-1, pt(2)]);
        queue.add([pt(1),   pt(2)-1]);
        queue.add([pt(1)+1, pt(2)]);
        queue.add([pt(1),   pt(2)+1]);
    end
    % Set the point to visited so we don't process it again
    visited(pt(2), pt(1)) = 1;
end

end

function isInvalid = isInvalidPt(pt,labeled)
isInvalid = false;
[y_sz, x_sz] = size(labeled);
x = pt(1);
y = pt(2);
if x < 0 || y < 0 || x > x_sz || y > y_sz
    isInvalid = true;
end
end

function didChange = changedPerspective(pt, bldNum, labeled)
didChange = (bldNum ~= labeled(pt(2),pt(1)));
end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

getPath.m

```
function path = getPath(bMap, S, T)

edgeTo = bfs(bMap, S, T);
path = reconstructPath(edgeTo);

end

function edgeTo = bfs( bMap, S, T )
% Does a breadth first search to discover the shortest path
% Returns a 29x3 matrix that shows how we got to each building
% Column 1 represents the building we came from, column 2 represents the
% direction we came (1=north, 2=east, 3=south, 4=west), column 3 is a flag
% that is 1 when the reference is near the building the edge points to

edgeTo = zeros(29,3);
tNum = T.number;

queue = java.util.LinkedList();
queue.add(S.number);
edgeTo(S.number,:) = [S.number, S.number, 1];

while ~queue.isEmpty()
    bNum = queue.remove();
    b = bMap(int2str(bNum));

    % North
    for i=b.north
        if edgeTo(i,1) == 0
            edgeTo(i,:) = [bNum, 1, 0];
            if any(b.near == i)
                edgeTo(i,3) = 1;
            end
            if i == tNum
                return;
            end
            queue.add(i);
        end
    end

    % East
    for i=b.east
        if edgeTo(i,1) == 0
            edgeTo(i,:) = [bNum, 2, 0];
            if any(b.near == i)
                edgeTo(i,3) = 1;
            end
            if i == tNum
                return;
            end
            queue.add(i);
```

Code

All code is my own - my only reference was MATLAB’s online documentation

```
        end
    end

    % South
    for i=b.south
        if edgeTo(i,1) == 0
            edgeTo(i,:) = [bNum, 3, 0];
            if any(b.near == i)
                edgeTo(i,3) = 1;
            end
            if i == tNum
                return;
            end
            queue.add(i);
        end
    end

    % West
    for i=b.west
        if edgeTo(i,1) == 0
            edgeTo(i,:) = [bNum, 4, 0];
            if any(b.near == i)
                edgeTo(i,3) = 1;
            end
            if i == tNum
                return;
            end
            queue.add(i);
        end
    end
end
end

function path = reconstructPath(edgeTo)
% Start at the target and work our way backwards through the selected edges
% The final path is a Nx4 matrix. N is the number of steps taken. The three
% columns are in the form: [build1 directionTake isNear build2]. In other
% words, it repesents the direction taken to go from building1 to
% building 2. The matrix is also ordered (the steps should be taken in
% sequence)
path = [];
bNum = 29; % The target
while true
    path = [[edgeTo(bNum,1) edgeTo(bNum,2) edgeTo(bNum,3) bNum]; path];
    bNum = edgeTo(bNum,1);
    if bNum == 28 % The source
        break;
    end
end
end
```

Code

All code is my own - my only reference was MATLAB's online documentation

getRegion.m

```
function [ region ] = getRegion( pt, img2D )
% Returns the region, which consists of two parts
% Part One: North, Middle, South
% Part Two: West, Center, East

[yMax, xMax] = size(img2D);
ySeg = yMax/3;
xSeg = xMax/3;
x = pt(1);
y = pt(2);

if x<xSeg && y<ySeg
    region = 'northwest';
elseif x<xSeg*2 && y<ySeg
    region = 'north-center';
elseif y<ySeg
    region = 'northeast';
elseif x<xSeg && y<ySeg*2
    region = 'mid-west';
elseif x<xSeg*2 && y<ySeg*2
    region = 'center';
elseif y<ySeg*2;
    region = 'mid-east';
elseif x<xSeg
    region = 'southwest';
elseif x<xSeg*2
    region = 'south-center';
else
    region = 'southeast';
end


end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

getReverseDirectionIdentifer.m

```
function [ id ] = getReverseDirectionIdentifier( direction )
% Returns the reverse integer identifier for a direction
% Useful when wording directions in certain ways
switch direction
    case 'north'
        id = 3;
    case 'east'
        id = 4;
    case 'south'
        id =  1;
    case 'west'
        id =  2;
end
end
```

getSizeDescription.m

```
function [ size ] = getSizeDescription(buildingArea, minArea, maxArea)
% Return a human readable description of the size of the building
% Smallest, tiny, small, medium, large
% Tiny and small are meant to differentiate between the smaller buildings
% Medium and large will capture greater range segments

range_sz = (maxArea - minArea) / 6;
size = 'Medium in size';

if buildingArea == minArea
    size = 'Smallest in size';
elseif buildingArea < (minArea + range_sz * .5)
    size = 'Tiny in size';
elseif buildingArea < (minArea + range_sz * 1.5)
    size = 'Small in size';
elseif buildingArea > (minArea + range_sz * 3.5)
    size = 'Large in size';
end

end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

outputStepOneData.m

```
function tab = outputStepOneData( bMap )

% Build Nums
bNums = [1:27]';

% Building names
bNames = cell(27,1);
for i=1:27
    bNames{i} = bMap(int2str(i)).name;
end

% Centroids
cents = cell(27,1);
for i=1:27
    cents{i} = round(bMap(int2str(i)).centroid);
end

% Area
areas = cell(27,1);
for i=1:27
    areas{i} = bMap(int2str(i)).area;
end

% Bounding box
bboxes = cell(27,1);
for i=1:27
    box = bMap(int2str(i)).boundingBox;
    bboxes{i} = round([box(1), box(2), box(1)+box(3), box(2)+box(4)]);
end

descriptions = cell(27,1);
for i=1:27
    descriptions{i} = bMap(int2str(i)).getDescription;
end

tab = table(bNums,bNames, cents, areas, bboxes, descriptions);

end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

outputStepTwoData.m

```
function tab = outputStepTwoData( bMap )

% Building names
bNames = cell(27,1);
for i=1:27
    bNames{i} = bMap(int2str(i)).name;
end

% Get north
northNames = cell(27,1);
for i=1:27
    desc = '';
    blds = bMap(int2str(i)).north;
    for j=1:length(blds)
        if j == 1
            desc = bMap(int2str(blds(j))).name;
        else
            desc = strcat(desc, {', '}, bMap(int2str(blds(j))).name);
        end
    end
    northNames{i} = desc;
end

% Get east
eastNames = cell(27,1);
for i=1:27
    desc = '';
    blds = bMap(int2str(i)).east;
    for j=1:length(blds)
        if j == 1
            desc = bMap(int2str(blds(j))).name;
        else
            desc = strcat(desc, {', '}, bMap(int2str(blds(j))).name);
        end
    end
    eastNames{i} = desc;
end

% Get south
southNames = cell(27,1);
for i=1:27
    desc = '';
    blds = bMap(int2str(i)).south;
    for j=1:length(blds)
        if j == 1
            desc = bMap(int2str(blds(j))).name;
        else
            desc = strcat(desc, {', '}, bMap(int2str(blds(j))).name);
        end
    end
    southNames{i} = desc;
```

Code

All code is my own - my only reference was MATLAB’s online documentation

```
end

% Get west
westNames = cell(27,1);
for i=1:27
    desc = '';
    blds = bMap(int2str(i)).west;
    for j=1:length(blds)
        if j == 1
            desc = bMap(int2str(blds(j))).name;
        else
            desc = strcat(desc, {', '}, bMap(int2str(blds(j))).name);
        end
    end
    westNames{i} = desc;
end

tab = table(bNames, northNames, eastNames, southNames, westNames);

end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

pruneRelationships.m

```
function [ bMap ] = pruneRelationships( bMap, b )
% Prune away relationships that can be inferred
% bMap is a key-value map of (string b_num : Building b)
% b is the Building object you're pruning
b = prune(bMap, b, b, 'north');
b = prune(bMap, b, b, 'south');
b = prune(bMap, b, b, 'west');
b = prune(bMap, b, b, 'east');
bMap(int2str(b.number)) = b;

end

function orig = prune(bMap, curr, orig, dir)
% Get the edges for the current node given the direction
adj_curr = getAdjacentNodes(curr, dir);
if isempty(adj_curr); return; end;

% Recurse on adjacent nodes
for i=1:length(adj_curr)
    orig = prune(bMap, bMap(int2str(adj_curr(i))), orig, dir);
end

% If the current building == the original, you're done
if curr.number == orig.number
    return;
end

% Prune adjacent nodes of the current from the original
adj_orig = getAdjacentNodes(orig, dir);
for i=1:length(adj_curr)
    % Remove adj_curr[i] from adj_orig if it's present
    len = length(adj_orig);
    index = find(adj_orig == adj_curr(i));
    if index > 0
        adj_orig = [adj_orig(1:index-1), adj_orig(index+1:len)];
    end
end
orig = setAdjacentNodes(orig, adj_orig, dir);
end

function building = setAdjacentNodes(building, adj, dir)
switch dir
    case 'east';  building.east = adj;
    case 'west';  building.west = adj;
    case 'north'; building.north = adj;
    case 'south'; building.south = adj;
end
end
```

Code

All code is my own - my only reference was MATLAB's online documentation

setSpatialRelationships.m

```
function [S, T] = setSpatialRelationships( S, T )
% Determines spatial relations for the source (S) and the target (T)
% S and T are Building objects
% Example of how to read north(S, T) : North of S is T
[S, T] = setSpatialRelationship(S, T, 'near');
[S, T] = setSpatialRelationship(S, T, 'east');
[S, T] = setSpatialRelationship(S, T, 'west');
[S, T] = setSpatialRelationship(S, T, 'north');
[S, T] = setSpatialRelationship(S, T, 'south');
end

function [S, T] = setSpatialRelationship( S, T, rel)
% Determines if the source (S) and the target (T) is spacially related
% S and T are Building objects
% rel is a string: 'east' 'west' 'north' or 'south'
% Example of how to read north(S, T) : North of S is T

switch rel
    case 'near'
        [S, T] = near(S, T);
    case 'east'
        [S, T] = east(S, T);
    case 'west'
        [S, T] = west(S, T);
    case 'north'
        [S, T] = north(S, T);
    case 'south'
        [S, T] = south(S, T);
end
end

function [S, T] = near(S, T)
    if rectint(S.expandedBoundingBox, T.boundingBox) > 0
        S.near = [S.near T.number];
    end
end

function doIntersect = doRangesIntersect(r1, r2)
adjustment = min(r1(1), r2(1));
r1 = r1 - adjustment;
r2 = r2 - adjustment;

if (r1(1) > r2(2) || r2(1) > r1(2)) % No intersection
    doIntersect = false;
else
    doIntersect = true;
end
end

function areInLine = areInLineHorizontally(S, T)
areInLine = false;
```

Code

All code is my own - my only reference was MATLAB’s online documentation

```
s_yrange = [S.spatialPts(1,2), S.spatialPts(3,2)];
t_yrange = [T.spatialPts(1,2), T.spatialPts(3,2)];

if doRangesIntersect(s_yrange, t_yrange)
    areInLine = true;
end
end

function areInLine = areInLineVertically(S, T)
areInLine = false;
s_xrange = [S.spatialPts(4,1), S.spatialPts(2,1)];
t_xrange = [T.spatialPts(4,1), T.spatialPts(2,1)];

if doRangesIntersect(s_xrange, t_xrange);
    areInLine = true;
end
end

function [S, T] = east(S, T)
if S.centroid(1) < T.centroid(1)
    if areInLineHorizontally(S, T)
        S.east = [S.east T.number];
    end
end
end

function [S, T] = west(S, T)
if S.centroid(1) > T.centroid(1)
    if areInLineHorizontally(S, T);
        S.west = [S.west T.number];
    end
end
end

function [S, T] = north(S, T)
if S.centroid(2) > T.centroid(2)
    if areInLineVertically(S, T);
        S.north = [S.north T.number];
    end
end
end

function [S, T] = south(S, T)
if S.centroid(2) < T.centroid(2)
    if areInLineVertically(S, T);
        S.south = [S.south T.number];
    end
end
end
```

Code

All code is my own - my only reference was MATLAB’s online documentation

useStudy.m

```
clc; clear; close all;
source = [187 42; 241 456; 144 152; 64 421; 26 189; 64 467; 144 41; ...
    135 422];

campus  = imread('supporting/ass3-campus.pgm');

for i=1:length(source)
    S = source(i,:);

    figure; imshow(campus); hold on;

    % Label the center
    start = plot(S(1), S(2), 'o', 'MarkerEdgeColor', 'g');
    set(start, 'MarkerSize', 6, 'LineWidth', 3);

    selectedPts = ginput

    for j=1:size(selectedPts,1)
        point = selectedPts(j,:);
        pt = plot(point(1), point(2), 'o', 'MarkerEdgeColor', 'r');
        set(pt, 'MarkerSize', 6, 'LineWidth', 3);
    end
    hold off;
end
```