Human Guided Machine Vision for Road Detection
Team AddNameHereWhenWeFinallyDecide

Kelsey DiPietro, Richard Frnka, Brian Hunter, Shaked Koplewitz, Khanh Nguyen, Scott Spencer

Problem Statement

Detection of roads from an aerial view is an important problem, yet one with an elusive general solution. Road detection is essential to developing and maintaining a database of roads, especially with the increased use of GPS devices. Humans are particularly good at picking out roads from an image, but can only do so at a limited speed. Computers operate at the other end of the spectrum, lacking in a general algorithm to detect roads, but able to do so quickly. To get the best of both worlds, it is useful to combine a computer program with human input to fill in where a computer missed a road.

The task was to create a program which finds roads in an aerial image and contains a user interface for a human to highlight specific roads and identify any missed roads. All user interface was to be done in a graphical component which allows the user to easily click on the image and have the computer do the rest of filling in and highlighting.

Initial Setup

The project was divided into three subprojects: A pre-processing part which involved initially segmenting the image into candidate road component and an object detection part which involved ranking the components for their likeliness to be a road and finding obstructions that blocked view of the road. After pre-processing was complete, there was a graphical user interface piece which would allow a human user to select and highlight roads and identify any roads missed by the pre-processing.

All programming was done using MATLAB which has built in libraries for image segmentation and graphical user interface. Only simple images consisting of a single road with no cars were considered initially, but as the segmentation methods were developed, more complicated road configurations were used. Since no single algorithm performed well on all roads, many were considered, each with a unique segmentation process.

Segmentation Methods

Below is a list of the segmentation functions with pseudocode and a brief description:

basicSegmentation

```
J ← grayscale image
BW ← edge(J,canny)
se0 ← structural horizontal line of length 3
se90 ← structural vertical line of length 3
BW ← dilate(BW,[se0,se90])
BW2 ← ∼BW
BW, BW2 ← fillHoles(BW), fillHoles(BW2)
BW, BW2 ← BW, BW2 minus small pieces
CC1 ← components(BW)
CC2 ← components(BW2)
return CC1 + CC2
```

This method is a bareboned road finder. It finds the edges, thickens them, fills the holes, cleans up noise, and returns the components.

blurSegmentation

```
J ← grayscale image
se ← structural square
Io ← erode(J,se)
Io ← dilate(J,se)
Ie ← erode(Io,se)
Iobr ← reconstruct(Ie,∼J)
Iobrd ← dilate(Iobr, se)
blur1 ← reconstruct(∼Iobrd,∼Iobr)
blur2 = ∼blur1
fgm, fgm2 ← localmax(blur1), localmax(blur2)
fgm, fgm2 ← fillHoles(fgm), fillHoles(fgm2)
CC1, CC2 ← components(fgm), components(fgm2)
return CC1 + CC2
```

This method blurs the image a significant amount to create a more even color distribution then looks for local maximums. It fills in any holes that were not local maxima and returns the components of the corrected image.

colorBasedSeg

```
J ← grayscale image
x,y ← dimensions(J)
blank ← zeroMatrix(x,y)
se ← structural square
Io ← erode(J,se)
Io ← dilate(J,se)
Ie ← erode(Io,se)
Iobr ← reconstruct(Ie,∼J)
Iobrd ← dilate(Iobr, se)
Iobrcbr ← reconstruct(∼Iobrd,∼Iobr)
CC ← components(null)
for i in range 0 to 255:
    C ← Iobrcbr == i
    C ← fillHoles(C)
    C ← C minus small pieces
    C ← components(C)
    for j in range 1 to length(C):
        CC += C
    rof
rof
return CC
```

Similar to blurSegmentation, this method first blurs the image to get a more even color distribution. It then gets components by taking pixels of each color index, filling the holes, cleaning up noise, and merging them into one component list to return.

Connected_Comp_Edges

```
I ← grayscale image
hy ← sobelFilter
Iy ← filter(I,hy)
Ix ← filter(I,hy')
se ← structural square
Io ← dilate(J,se)
Io ← erode(J,se)
Ie ← erode(Io,se)
Iobr ← reconstruct(Ie,∼J)
Iobrd ← dilate(Iobr, se)
Iobrcbr ← reconstruct(∼Iobrd,∼Iobr)
```

```
fgm ← localmax(Iobrcbr)
fgm ← edge(fgm,sobel)
fgm ← thicken(fgm)
se ← structural disk
cfgm ← erode(fgm,se)
cfgm ← dilate(cfgm,se)
inv ← ∼cfgm
CC ← components(inv)
return CC
```

A method that uses a Sobel filter and multiple dilations an erosions to smooth an image. It finds the edges using the Canny method and thickens them before finding the components.

majorSegmentation/ImageSegmentation

```
rgb ← color image
rgbblur ← gaussFilter(rgb)
gray ← grayscale(rgbblur)
edges ← edge(gray, Canny)
edges ← thicken(edges)
structElem ← structural square
edges ← dilation(edges, structElem)
edges ← erosion(edges, structElem) edges ← shrink(edges)
edges ← ẽdges
bw ← BW image with non-roads blacked out
bwEdged ← edges .* bw
filt ← bwEdged minus small pieces
filt ← thicken(filt)
filt ← fillHoles(filt)

return components(filt)
```

This method enhances external boundaries, darkens internal boundaries, and uses Canny edge detection to split the image up into components. This algorithm is used in the GUI as this tended to be the most robust segmentation algorithm. The GUI also uses two other algorithms which are modifications of this one, and in the performance table below, this general algorithm is simply referred to as ImageSegmentation.

ext_grad_seg

```
ibw ← grayscale image
se ← structural square
eg ← dilate(ibw,se) - ibw
rec ← reconstruct(eg,ibw)
f1 ← edge(rec,canny)
fgm ← localmax(rec)
fgm ← fgm minus small pieces
return components(fgm)
```

This method enhances external boundaries, darkens internal boundaries, and uses Canny edge detection to split the image up into components.

rgbSeg

```
J ← color image
x,y ← dimension(J)
apform ← L*a*b form of J
apd ← reshape apform to x*y by 2
cluster ← kmeans(apd) with k = 5
pixels ← reshape cluster to x by y
CC ← empty component list
for i in range 1 to 5:
    temp ← x by y zero matrix
    temp ← fillHoles(temp)
    temp ← (pixels == i) minus small pieces
    CC += components(temp)
rof
return CC
```

This method takes an rgb image and segments it into 5 parts using k-means. It then finds components for each individual cluster and merges them together in a master list.

thinRoad

```
I ← grayscale image
I2 ← ~I
x,y ← dimensions of I
BW1, BW2 ← I > 150, I2 > 150
```

```
BW1, BW2 ← fillHoles(BW1), fillHoles(BW2)
BW1, BW2 ← BW1, BW2 minus very small pieces
CC1, CC2 ← components(BW1), components(BW2)
return CC1 + CC2
```

This method segments a grayscale image and it's complement by separating by intensity and filling holes. It removes only very small components to accommodate for the small area of thin roads.

Computational Efficiency Comparison

All the above segmentation methods produce different connected component matrices, to varying levels of accuracy. We ran each algorithm separately and took note of the computational time required for each method, the number of connected components, and, using a very rough visual test, we measured the overall ability of each algorithm to detect the roads in an efficient manner.

All tests were ran using MATLAB2014b, on a standard laptop. It should be noted though that these times only reflect the computational time required for preprocessing. If an image is segmented properly, than the GUI runs very close to real time with little processing time. Therefore, we will take identifying all the roads properly for longer computational time over poor road identification and shorter preprocessing time.

We use the following two test images:

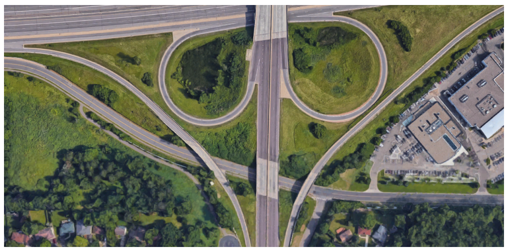


Figure 1: Highway Image

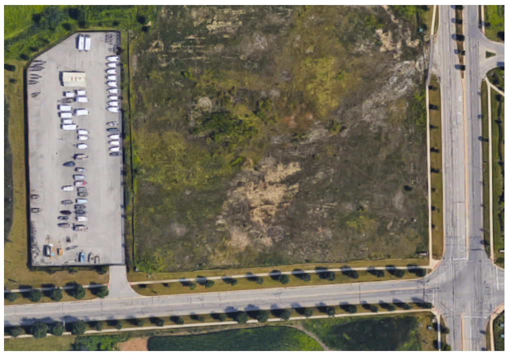
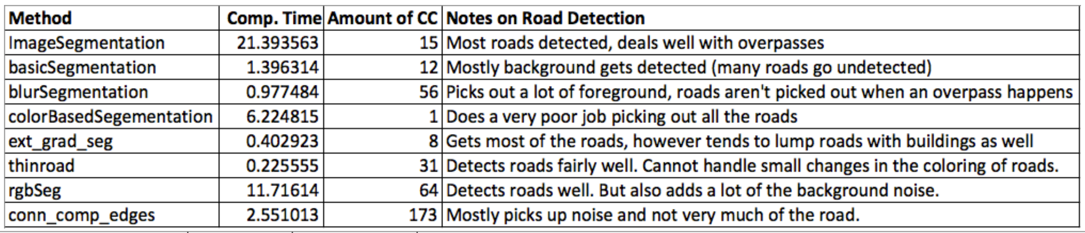
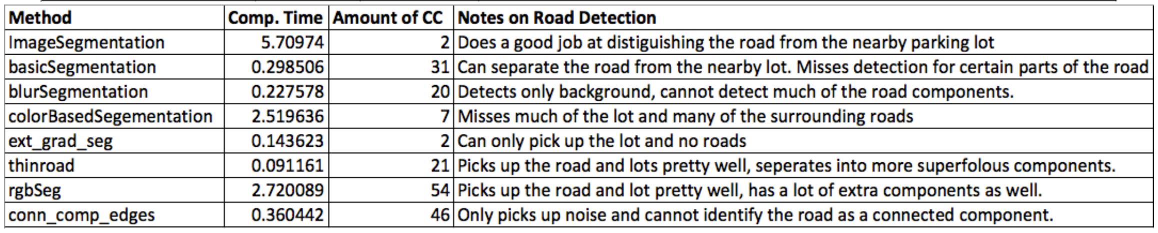


Figure 2: Residential Image

The following two tables generalize our results. While the ImageSegmentation filter takes the longest, it correctly identifies all the roads for both images based on a crude visual analysis. While the other algorithms have a shorter computational time, they either miss key road segments or add a lot of additional noise to the connected road components, which would increase the processing time for GUI component of the project. Therefore we strictly use the ImageSegmentation filter for all of our test cases.



Method	Comp. Time	Amount of CC	Notes on Road Detection
ImageSegmentation	21.393563	15	Most roads detected, deals well with overpasses
basicSegmentation	1.396314	12	Mostly background gets detected (many roads go undetected)
blurSegmentation	0.977484	56	Picks out a lot of foreground, roads aren't picked out when an overpass happens
colorBasedSegementation	6.224815	1	Does a very poor job picking out all the roads
ext_grad_seg	0.402923	8	Gets most of the roads, however tends to lump roads with buildings as well
thinroad	0.225555	31	Detects roads fairly well. Cannot handle small changes in the coloring of roads.
rgbSeg	11.71614	64	Detects roads well. But also adds a lot of the background noise.
conn_comp_edges	2.551013	173	Mostly picks up noise and not very much of the road.



Method	Comp. Time	Amount of CC	Notes on Road Detection
ImageSegmentation	5.70974	2	Does a good job at distiguishing the road from the nearby parking lot
basicSegmentation	0.298506	31	Can separate the road from the nearby lot. Misses detection for certain parts of the road
blurSegmentation	0.227578	20	Detects only background, cannot detect much of the road components.
colorBasedSegementation	2.519636	7	Misses much of the lot and many of the surrounding roads
ext_grad_seg	0.143623	2	Can only pick up the lot and no roads
thinroad	0.091161	21	Picks up the road and lots pretty well, seperates into more superfolous components.
rgbSeg	2.720089	54	Picks up the road and lot pretty well, has a lot of extra components as well.
conn_comp_edges	0.360442	46	Only picks up noise and cannot identify the road as a connected component.

Future work would entail doing some kind of weighted average using the best algorithms and implementing them for our test images. Also, as the charts show, some segmentation strategies work better for highway systems than they do for more residential areas, which is why we used the ImageSegmentation for our examples, as it performs robustly on both within the preprocessing step.

Canny Edge Detection and K-Means

Canny edge detection comes up frequently in the segmentation methods above. This edge detection algorithm has six steps, which are outlined here: Apply a Gaussian filter to reduce unwanted noise, compute the gradient of the image, finding an optimal cutoff threshold and apply it to the image, suppress non-maxima pixels, threshold the image twice more with a low and high threshold, and merge segments in the noisier image using the low noise image as a guide. MATLAB does feature many other edge detection algorithms, but Canny was found to be a strong candidate for road detection due to it's ability to suppress unwanted noise and find the true edge of the road.

The rgbSeg method used the k-means algorithm from machine learning as a way to divide an image by it's color value. K-means works by first assigning k random centroids among the set of data points, in this case pixels. Then at each step, every point is assigned to the group which has the closest centroid and after all points are grouped, new centroids are calculated. New group assignments and centroids are found until the groups stop changing, at which point the algorithm has converged to a local minimum. It is important to note that k-means will converge to a local minimum, but it is not guaranteed to be a global minimum. Depending on the amount of pre-processing time allowed, it may be beneficial in the rgbSeg method to repeat the k-means method multiple times and take the best solution, as this could help to ensure that the image is not segmented into an undesirable local minimum. However, in practice, a non-optimal clustering on the image did not hinder road detection to a significant extent in that if any additional pieces were grouped in the same cluster as the road, they were separated when splitting up the image into components.

User Interface

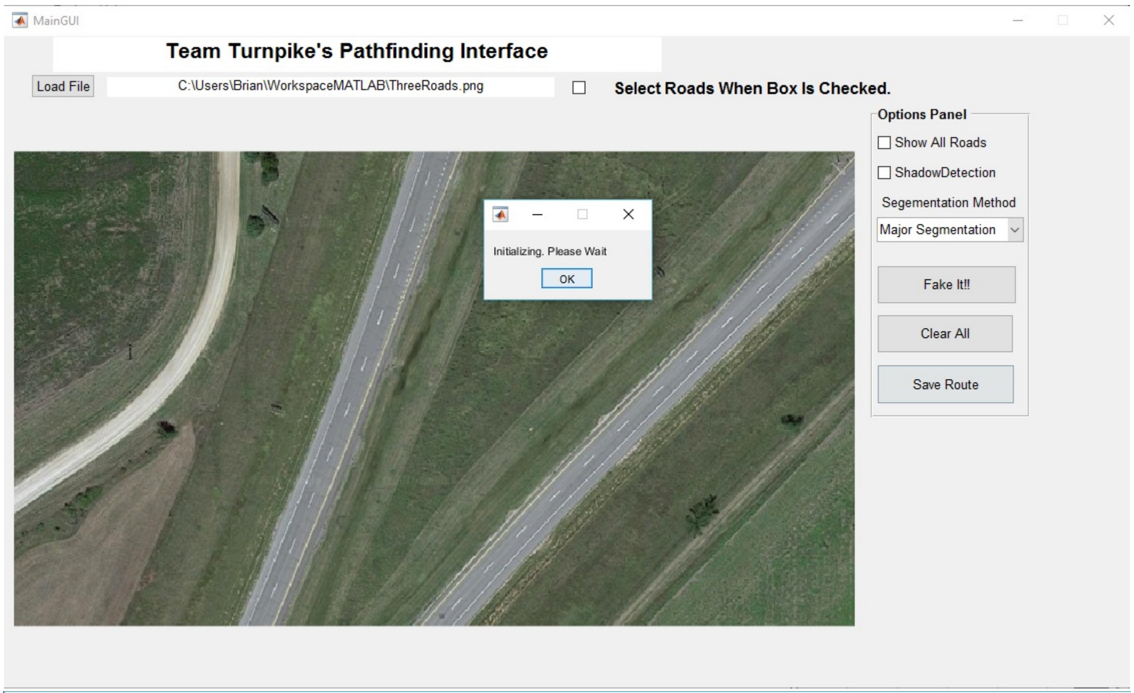
The Graphical User Interface (GUI) provides a way to interact with the project without having to sort through the code or do any kind of programming. Ideally, this would be presented to a client so they can point and click without worrying about any of the background processing.

Initially, the user loads whatever image they would like to analyze. Any new image that is loaded should clear the current process and refresh the interface. Once an image is loaded, pre-processing is done in the background. This runs the default segmentation (Major Segmentation) behind the scenes, and produces the components that are used throughout the rest of the program. The user sees a message box implying patience that vanishes once initialization is complete.

After pre-processing, the user can begin to interact with the rest of the GUI. One of the more helpful features is the Show All Roads button. This shows the results of the image segmentation method that is being used and gives a visual clue as to what can be considered a road. The user can check the Select Roads box to begin drawing their path. The GUI keeps track of what roads have been selected with a binary vector X, which is the size of the number of components. X is updated as users click on the components they want, and it is one of the major global paremeters that is passed through the GUI. With X and the component cell CC, it is easy to highlight the selected roads in the image by turning all of the pixels corresponding to selected components blue (0000FF).

The GUI has options to clear the current image and its parameters, save the file by exporting the current image to a new PNG, change the segmentation method to the options indicated above, and turn on shadow detection. The current shadow detection has thresholds adjusted to fit certain images, but ideally, the user could adjust these parameters in the future.

Included below are two pictures of the GUI in action. It loads the image and runs the pre-processing image segmentation to get the road components. As shown in the second image, it allows the user to select which roads to highlight. In the example, the user has selected the left and right roads, but has left the center road unselected.



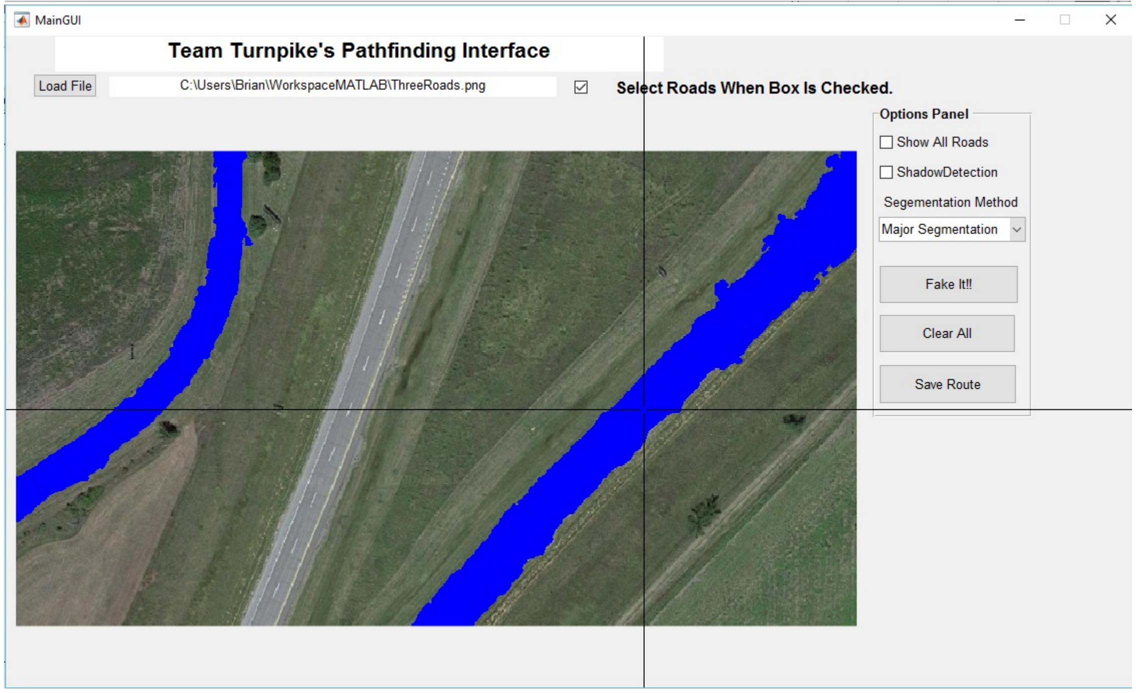
Team Turnpike's Pathfinding Interface

C:\Users\Brian\WorkspaceMATLAB\ThreeRoads.png

☐ Select Roads When Box Is Checked.

Segementation Method: Major Segmentation

Initializing. Please Wait



Team Turnpike's Pathfinding Interface

C:\Users\Brian\WorkspaceMATLAB\ThreeRoads.png

☑ Select Roads When Box Is Checked.

Segementation Method: Major Segmentation

Object and Road Detection

car_detection

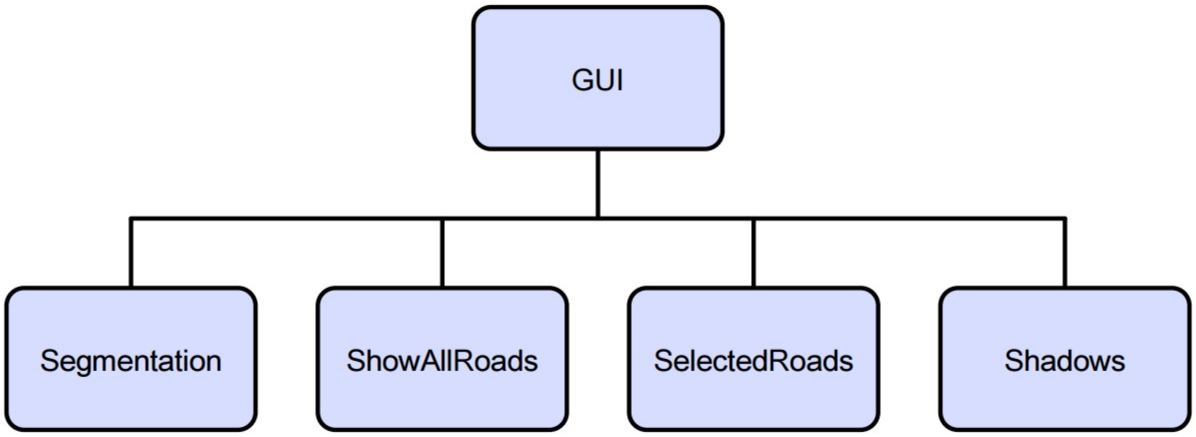
```
M1_B ← grayscale image
M2_B ← maximum intensity of each row of M1_B
T1_B ← mean(M2_B)
T2_B ← min(M2_B)
T3_B ← mean(T1_B,T2_B)
Image1_B ← 0 if M1_B(i,j) <T1_B; 1 if M1_B(i,j) >T1_B
Image2_B ← 0 if M1_B(i,j) <T2_B; 1 if M1_B(i,j) >T2_B
Image3_B ← 0 if M1_B(i,j) <T3_B; 1 if M1_B(i,j) >T3_B
New_Image1_B ← bitand(Image1_B, Image2_B)
New_Image2_B ← bitand(New_Image1_B, Image3_B)
Bright_cars ← bitor(New_Image1_B, New_Image1_B)
M1_D ← grayscale image
M2_D ← minimum intensity of each row of M1_D
T1_D ← mean(M2_D)
T2_D ← min(M2_D)
T3_D ← mean(T1_D,T2_D)
Image1_D ← 1 if M1_D(i,j) <T1_D; 0 if M1_D(i,j) >T1_D
Image2_D ← 1 if M1_D(i,j) <T2_D; 0 if M1_D(i,j) >T2_D
Image3_D ← 1 if M1_D(i,j) <T3_D; 0 if M1_D(i,j) >T3_D
New_Image1_D ← bitand(Image1_D, Image2_D)
New_Image2_D ← bitand(New_Image1_D, Image3_D)
Dark_cars ← bitor(New_Image1_D, New_Image1_D)
cars ← bitor(Dark_cars,Bright_cars)
CC ← bwconncomp(cars)
car_labels ← labelmatrix(CC)
return car_labels
```

Inspired by the idea of Sumalatha Kuthadi's master's thesis, this method finds bright colored cars and black colored cars using multiple thresholds. Bright colored cars' intensities are higher comparing to roads' intensity, and dark colored cars' intensities are lower comparing to roads' intensity. Both types of cars are combined to define all cars on the road. This method successfully determines all cars in a parking lot. Unfortunately, it won't work in more complicated pictures because it will focus on any other objects with brighter colors or darker colors as cars. As a result, this is not used in the implementation of the main algorithm.

Shadow detection code was written by Beril Sirmacek, and is permitted to redistribute and use in source code. This code was used in the algorithm listed in future work to remove shadows from images.

Code Call Chart

The GUI serves as the main function with which the user can interface. Since each segmentation function was made to run as it's own independent component, the GUI can make calls to any and all of these, and each function is self-contained. The GUI currently makes use of simpleSegmentation, majorSegmentation, and overpassSegmentation. The majorSegmentation algorithm is listed above and the other two methods are slight variations of it. Additionally, the GUI also calls shadows, showAllRoads, and selectedRoads to highlight various areas on the image. This yields a simple call chart displayed below.



For the shadow removal, shadow_detect served as the main method which called removeShadows to modify the image. Shadow detection is used in the GUI as the function shadows, mentioned above, but the removal function has not yet been included. Given the simplicity of a single main and function file, the call chart for these two is omitted.

Future Work

Our methods were able to properly segment and detect road paths for a narrow sample of test images. This sample includes roads without shadows, highway systems with simple inter-passes, and intersections where there is a strong change in intensity between the intersecting roads. An example of the later category is an intersection where the main road has been recently repaved, thus darker, than its corresponding intersection road. In addition, we chose to sample images from the same satellite scaling (i.e. a scale of 100 ft from Google Earth aerial view).

If given more time, we would like to extend our implementation to a wider host of sample problems. These future directions as well as possible strategies for tackling these more complications are given below.

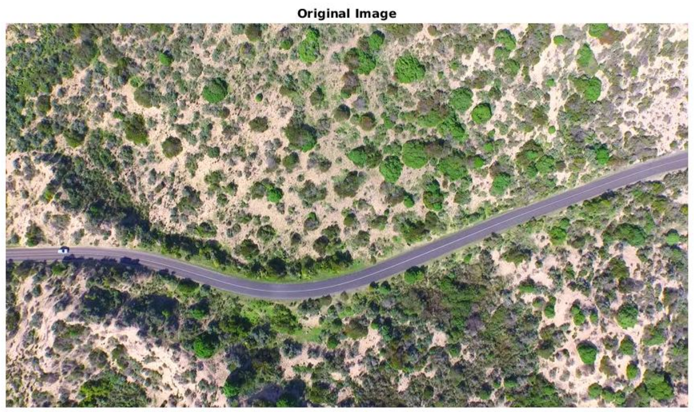
1. Car detection and removal: For our sample images, we considered roads that only had a few cars. Because the scale of the car was much smaller than the road, our algorithm was able to ignore the car and still return a connected component road. We would like to extend our algorithm to identifying roads that have more traffic on them, which would interfere with the connectivity of the segmentation. Ideally, we would be able to identify (using size training algorithms) all the cars in the image and either remove those cars or change them to the color of the road. This would allow for the segmentation algorithm to still see the road as a connected component to highlight, regardless of the traffic.

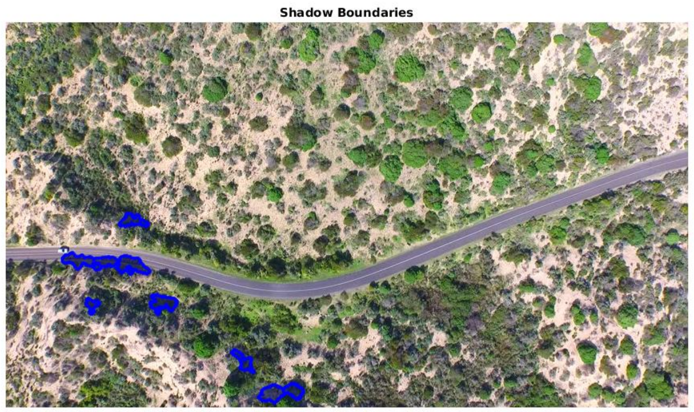
2. Shadow detection and removal: Similar to cars with high traffic, shadows pose a huge obstruction to our algorithm's ability to perform. Often times shadows from trees and buildings end up breaking up the road in separate connected components. Similar to the car detection, ideally we would use filter that could identify all shadows on the roads and change those shadows to the same intensity of the road and then feed that image to the segmentation component of our preprocessing algorithm. We have been able to do this for small shadows that only obscure part of the road, but have not been able to remove larger shadows that segment much of the road.

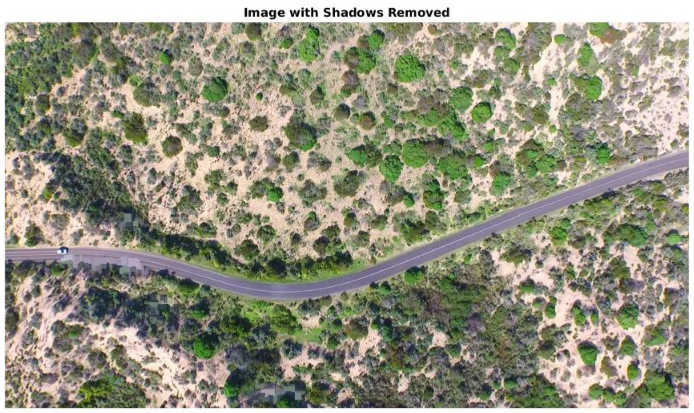
   Below is pseudocode for the shadow removal algorithm. It works by going around the boundary of detected shadows and dilates over points within 2 units to the left, right, above, and below. After dilating the boundary points, it finds the boundary of the new, smaller shadow and continues this process until all points of the shadow have been covered.

```
gr,gg,gb ← r,g,b components of image
gr,gg,gb ← gr,gg,gb embedded in a larger matrix
blank ← bw matrix with only shadows
x,y ← x,y coordinates of boundary pixels of shadows
while x,y not empty
    for i in range of length of x
        dilate gr,gg,gb at point x[i],y[i]
    rof
    blank ← blank minus boundary points
    x,y ← coordinates of boundary pixels in blank
elihw nosh ← inner matrices of gr,gg,gb combined
return nosh
```

   Included as an example is an image in which a shadow partially obstructs the view of the road. Shadows identified by the shadow detection method are highlighted in blue, and below is the result after the above function is run.







3. Extending the road detection algorithm: Currently our road detection takes connected components from image segmentation and returns the probability that each component is a road. This allows for limited processing to be wasted in the GUI if the user clicks a large connected component such as a forest. It would be helpful to add an additional component to this algorithm that would work in conjunction with the GUI, so that if the user clicks within

some region nearby the road, it will take them to the nearest road.

4. Detecting intersections and choosing directions: Our algorithm can only handle intersections in which there is a change in the intensity between intersecting roads. Ideally, we would want to be able to handle any type of intersection. Finding and segmenting an intersection consists of two parts: identifying the fact that we are at an intersection and segmenting that intersection into its component parts.

   (a) Identifying an intersection: Intersection identification could be done a local level for the given image. Given road segmentation, one could find an intersection by looking at the width of the road (which should stay constant). At an intersection, this constant width changes and the location of this change should be flagged as an intersection.

   (b) Breaking up the intersection: once a location is flagged as an intersection point, create lines in each direction of the intersection that cause an intensity change. This is equivalent to building a new edge, so that the algorithm detects the edge and breaks up the connected road component.