**ECE 479 – Winter 2019**

**Final Project – Dim PSU Greeter**

**Group Members: Stephen Poanessa, Ali \_\_\_\_, John \_\_\_\_\_ and Adel**

**Introduction**

**Implementation**

**Navigation of PSU FAB basement**

**List of phrases**

**Xbox kinnect**

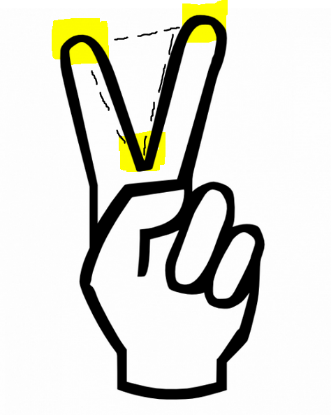
**LED Matrix**

**Introduction**

In order to allow easier integration of additional capabilities by future robotics students, the entire code base for DIM was rebuilt in a purely pythonic environment. Software written last quarter used ROS as the underlying framework for accessing specific python code. While this implementation worked, it was needless complicate, and would have been a daunting challenge for future students to attempt to understand and be able to adapt within a school quarter’s timespan.

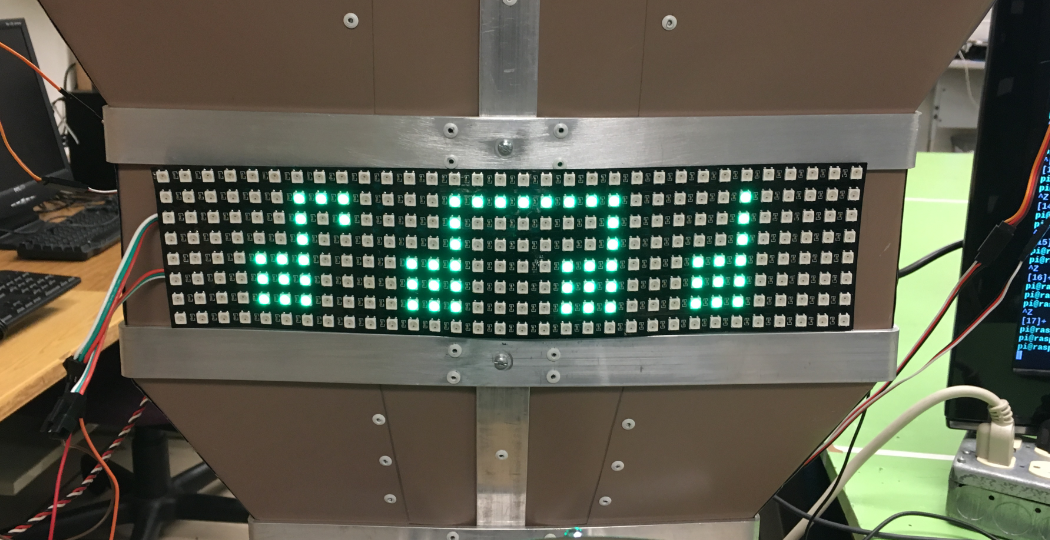
Given that DIM is supposed to be a greeter from a location behind the glass of the robotics lab, a method of starting interaction had to be considered. It was ultimately decided that a gesture-based system of initializing individual interactions with DIM would be the most ideal method to peruse given that it would likely produce less false positive results from students walking by the Robotics window that could result in distractions to students working in the robotics lab.

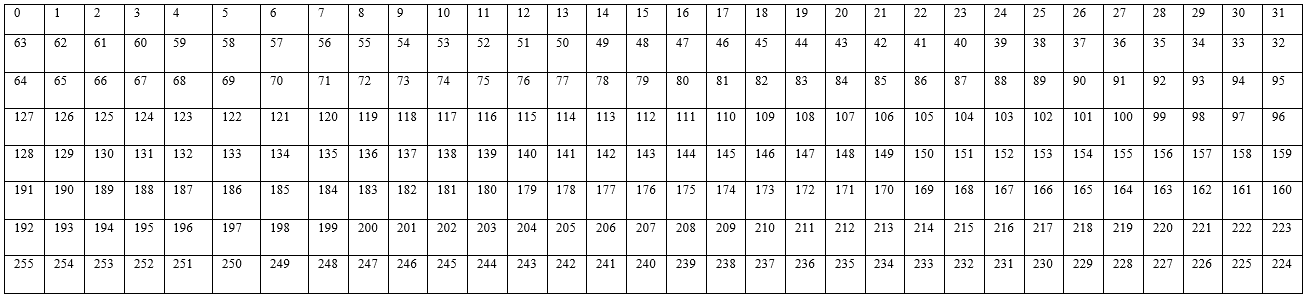
**Hand Gesture Control using OpenCV**

A python script for vision-based accounting of digits was used as the “main” body of the DIM’s code. The code explores a similar method of masking and image adjustments within a defined ROI as the face detection script used in Robotics 1. An important distinction is that the digit detection method uses trigonometric equations to calculate and verify the existence of a triangle shape that occurs when

**LED ARRAY**

Longruner 8x32 256 Pixels Digital Flexible LED Panel, Built-in WS2812B IC Individually Addressable LED Light with Full Dream Color Lighting DC5V LWS03



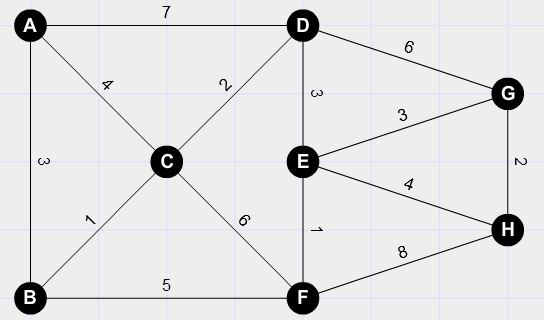


**Kinnect Integrations**

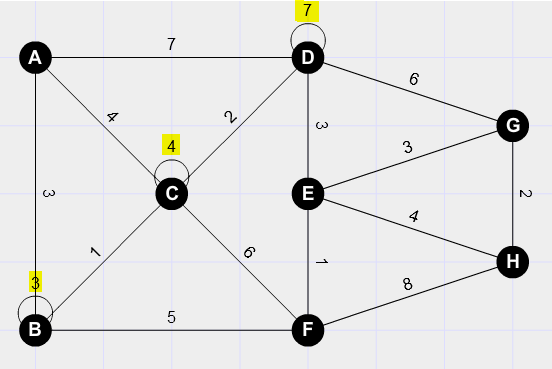
**Prolog**

**Shortest Path: Dijkstra’s Algorithm**

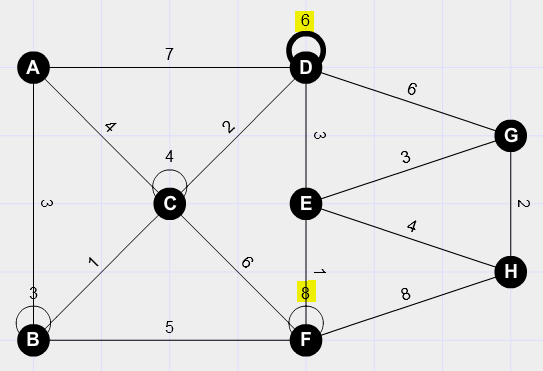
Given the following edge structure with weighted distances we would like to obtain the shortest path from A to H.



From starting edge A, list all distances between the connected edges:

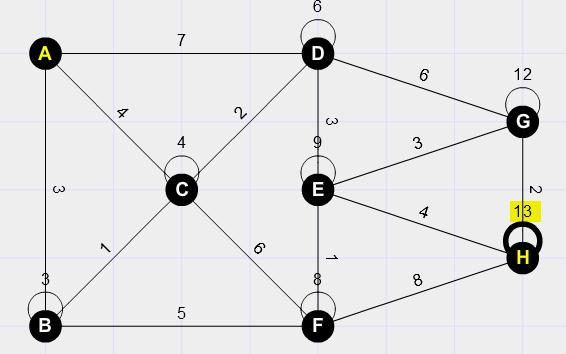


Repeat the process from B to connected edges. In the example of moving from B to C, the total distance would = 3+1. Since this value is the same/not less than the value already present on C, the value of 4 is kept on C. The distance from B to F is then determined to be 3+5 which is added to the F edge. This process is then repeated with the following results:

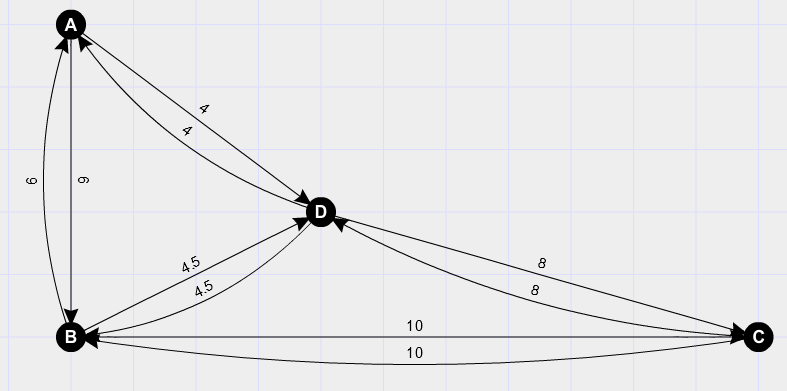


*Note: The value of six on edge D replaced the previous value of seven, since the path from A to C to D yields a shorter distance.*

If this process is repeated until we arrive at our desired destination H which indicates the shortest distance required to travel from A to H is 13.



**Shortest Path: Abstraction of Prolog Implementation**



The example above is used to illustrate the hierarchy of predicates for the shortest path prolog program.

“**Edges**” which are a natural prolog predicates are shown in the image above. An edge by itself does not yield any useful results, but when connected to other edges, they can be used to determine viable connections that can be traversed.

The predicate in our program called “linked” is used to connect edges and output a length/distance which exists between them. This predicate can be thought of as the foundation that our prolog program uses to determine interconnection between edges.

A predicate called “**walk**” is then used to build on the “linked” predicate by combining connected edges which exist from a start point to a desired end point. This predicate allows movement from the starting edge to intermediate edges, which ultimately lead to the desired destination edge. As it traverses these edges, they are recorded within a list that is then reversed upon completion. This reversal is necessary given that subsequent edges are appended to the list and by the end of the walk predicate the first edge is located at the end of the tail of the list. This predicate is also responsible for adding of total length between linked edges.

The next predicate in this hierarchy is “**path**”. This predicate is responsible for combining the information obtained thorough the “walk” predicate. In a sense this predicate is develop[ping permutations of lists of edges, that are linked together by the “walk” predicate. This predicate is also responsible for ensuring that paths do not result in situations that are infinite, such as walking back and forth from two edges continuously. This is done by comparing existing elements within the list to ensure they are not repeated, and no backtracking occurs.

The final predicate is the called “**shortes**t”. This predicate takes the list of paths obtained in the “path” predicate and selects a path from the list which has the shortest distance.

A final list which has the shortest path is then returned and outputted by the predicate, “**minimal**”.

**Hand Detection – Python Implementation**

import cv2

import numpy as np

import math

cap = cv2.VideoCapture(0)

while(1):

try: #an error comes if it does not find anything in window as it cannot find contour of max area

#therefore this try error statement

ret, frame = cap.read()

frame=cv2.flip(frame,1)

kernel = np.ones((3,3),np.uint8)

#define region of interest

roi=frame[100:300, 100:300]

cv2.rectangle(frame,(100,100),(300,300),(0,255,0),0)

hsv = cv2.cvtColor(roi, cv2.COLOR\_BGR2HSV)

# define range of skin color in HSV

lower\_skin = np.array([0,20,70], dtype=np.uint8)

upper\_skin = np.array([20,255,255], dtype=np.uint8)

#extract skin colur imagw

mask = cv2.inRange(hsv, lower\_skin, upper\_skin)

#extrapolate the hand to fill dark spots within

mask = cv2.dilate(mask,kernel,iterations = 4)

#blur the image

mask = cv2.GaussianBlur(mask,(5,5),100)

#find contours

\_,contours,hierarchy= cv2.findContours(mask,cv2.RETR\_TREE,cv2.CHAIN\_APPROX\_SIMPLE)

#find contour of max area(hand)

cnt = max(contours, key = lambda x: cv2.contourArea(x))

#approx the contour a little

epsilon = 0.0005\*cv2.arcLength(cnt,True)

approx= cv2.approxPolyDP(cnt,epsilon,True)

#make convex hull around hand

hull = cv2.convexHull(cnt)

#define area of hull and area of hand

areahull = cv2.contourArea(hull)

areacnt = cv2.contourArea(cnt)

#find the percentage of area not covered by hand in convex hull

arearatio=((areahull-areacnt)/areacnt)\*100

#find the defects in convex hull with respect to hand

hull = cv2.convexHull(approx, returnPoints=False)

defects = cv2.convexityDefects(approx, hull)

# l = no. of defects

l=0

#code for finding no. of defects due to fingers

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

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

start = tuple(approx[s][0])

end = tuple(approx[e][0])

far = tuple(approx[f][0])

pt= (100,180)

# find length of all sides of triangle

a = math.sqrt((end[0] - start[0])\*\*2 + (end[1] - start[1])\*\*2)

b = math.sqrt((far[0] - start[0])\*\*2 + (far[1] - start[1])\*\*2)

c = math.sqrt((end[0] - far[0])\*\*2 + (end[1] - far[1])\*\*2)

s = (a+b+c)/2

ar = math.sqrt(s\*(s-a)\*(s-b)\*(s-c))

#distance between point and convex hull

d=(2\*ar)/a

# apply cosine rule here

angle = math.acos((b\*\*2 + c\*\*2 - a\*\*2)/(2\*b\*c)) \* 57

# ignore angles > 90 and ignore points very close to convex hull(they generally come due to noise)

if angle <= 90 and d>30:

l += 1

cv2.circle(roi, far, 3, [255,0,0], -1)

#draw lines around hand

cv2.line(roi,start, end, [0,255,0], 2)

l+=1

#print corresponding gestures which are in their ranges

font = cv2.FONT\_HERSHEY\_SIMPLEX

if l==1:

if areacnt<2000:

cv2.putText(frame,'Use hand signal in box',(0,50), font, 1, (0,0,255), 3, cv2.LINE\_AA)

else:

if arearatio<12:

cv2.putText(frame,'0',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

else:

cv2.putText(frame,'1',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

elif l==2:

cv2.putText(frame,'2',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

elif l==3:

if arearatio<27:

cv2.putText(frame,'3',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

elif l==4:

cv2.putText(frame,'4',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

elif l==5:

cv2.putText(frame,'5',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

elif l==6:

cv2.putText(frame,'reposition',(0,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

else :

cv2.putText(frame,'reposition',(10,50), font, 2, (0,0,255), 3, cv2.LINE\_AA)

#show the windows

cv2.imshow('mask',mask)

cv2.imshow('frame',frame)

except:

pass

k = cv2.waitKey(5) & 0xFF

if k == 27:

break

cv2.destroyAllWindows()

cap.release()