In [1]:
```
## --------------------------------------------------------------------------------------
#                                   Dijkstra Algorithm
## --------------------------------------------------------------------------------------

# Author: Jai Sharma
# Course: ENPM661 - Planning for Autonomous Robots
# Assignment: Project  2 --> Dijkstra Path Planning

# Important Checks
#       -->  if start node and goal node inputs are integers
#       -->  if start node and goal node is within map
#       -->  if start node and goal node is within obstacle
```

In [2]:
```
## --------------------------------------------------------------------------------------
#                                  Step 0 --> Import Libraries
## --------------------------------------------------------------------------------------
import numpy as np
import heapq
import time
import cv2
import pygame
```

```
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html (https://www.pygame.org/contribute.html)
```

In [3]:

```python
## ------------------------------------------------------------------------------------------
#                                   Step 1 --> Inputs and Initialization
## ------------------------------------------------------------------------------------------

# Map Size is known
Map_width = 400
Map_height = 250

# Request Input from User
print("Note: All User Inputs Must be Integers")
print("x coordinates must be Smaller than ", Map_width)
print("y coordinates must be Smaller than ", Map_height, "\n")

Xi = int(input("x coordinate of START node -->  "))
Yi = int(input("y coordinate of START node -->  "))
Xg = int(input("x coordinate of GOAL node -->  "))
Yg = int(input("y coordinate of GOAL node -->  "))

# Initialize Nodes
start = (Xi,Yi)
goal =  (Xg,Yg)

# List of all Coordinates on the Map
map_cord = []
for x in range(Map_width + 1):
    for y in range(Map_height + 1):
        map_cord.append((x,y)) #appending to the list

# Check if Input Nodes are within Map Limits
if (start in map_cord) and (goal in map_cord):
    print("\n START and GOAL node inputs are within Map Limits")
else:
    print("\n ERROR: START and/or GOAL node inputs are Outside Map Limits")
    print("\n Please enter new node values")
```

```
Note: All User Inputs Must be Integers
x coordinates must be Smaller than  400
y coordinates must be Smaller than  250

x coordinate of START node -->  386
y coordinate of START node -->  63
x coordinate of GOAL node -->  9
y coordinate of GOAL node -->  223

 START and GOAL node inputs are within Map Limits
```

In [4]:

```
## ------------------------------------------------------------------------------------------
#                                     Step 2 -->  Obstacle Mapping
## ------------------------------------------------------------------------------------------

def Obstacles(map_pts,start,goal):
    obstacle_cord = []
    for pts in map_pts:
        x, y = pts[0], pts[1]

        # Circle Obstacle
        if (x > 250):
            if (y > 125):
                if((x-300)**2 + (y-175)**2 <= (40)**2):
                    obstacle_cord.append((x,y))
        # Hexagon Obstacle
        if x > 165 and x < 235:
            if y > 50 and y < 150:
                if (y - 0.577*x < 24.97):
                    if (y - 0.55*x > -50):
                        if (y + 0.577*x < 255.829):
                            if (y + 0.548*x > 169.314):
                                obstacle_cord.append((x,y))
        # Weird Shape Obstacle
        if y > 90 and x < 120:
            if y + 1.232*x > 229.348:
                if y - 0.316*x < 173.608:
                    if (y - 0.857*x > 114.29) or (y + 3.2*x < 436):
                        obstacle_cord.append((x,y))

    if start in obstacle_cord:
        print('Error!! START Node is within the Obstacle Space')

    if goal in obstacle_cord:
        print('Error!! GOAL Node is within the Obstacle Space')

    return(obstacle_cord)
```

In [5]:
```python
## -----------------------------------------------------------------------------------------
#                                        Step 3 -->  Define Actions
## -----------------------------------------------------------------------------------------

# 8 actions, make 8 functions or sub-functions

def Up(curr_node):    # Action Step to move UP
    new_node_y = curr_node[0] + 1
    new_node = (new_node_y,curr_node[1])
    if new_node[0]>= 0 and new_node[1]>=0:
        return(new_node,True)
    else:
        return(curr_node,False)

def Down(curr_node):  # Action Step to move DOWN
    new_node_y = curr_node[0] - 1
    new_node = (new_node_y,curr_node[1])
    if new_node[0]>=0 and new_node[1]>=0:
        return(new_node,True)
    else:
        return(curr_node,False)

def Left(curr_node):   # Action Step to move LEFT
    new_node_x = curr_node[1] - 1
    new_node = (curr_node[0],new_node_x)
    if new_node[0]>=0 and new_node[1]>=0:
        return(new_node,True)
    else:
        return(curr_node,False)

def Right(curr_node): # Action Step to move RIGHT
    new_node_x = curr_node[1] + 1
    new_node = (curr_node[0],new_node_x)
    if new_node[0]>=0 and new_node[1]>=0:
        return(new_node,True)
    else:
        return(curr_node,False)

def UpLeft(curr_node): # Action Step to move UP LEFT
    new_node_y = curr_node[0] + 1
    new_node_x = curr_node[1] - 1
    new_node = (new_node_y,new_node_x)
    if new_node[0]>=0 and new_node[1]>=0:
        return(new_node,True)
    else:
        return(curr_node,False)

def UpRight(curr_node): # Action Step to move UP Right
    new_node_y = curr_node[0] + 1
    new_node_x = curr_node[1] + 1
    new_node = (new_node_y,new_node_x)
    if new_node[0]>=0 and new_node[1]>=0:
        return(new_node,True)
    else:
        return(curr_node,False)

def DownLeft(curr_node): # Action Step to move DOWN LEFT
    new_node_y = curr_node[0] - 1
    new_node_x = curr_node[1] - 1
    new_node = (new_node_y,new_node_x)
    if new_node[0]>=0 and new_node[1]>=0:
        return(new_node,True)
    else:
```

```
        return(curr_node,False)

def DownRight(curr_node): # Action Step to move DOWN RIGHT
    new_node_y = curr_node[0] - 1
    new_node_x = curr_node[1] + 1
    new_node = (new_node_y,new_node_x)
    if new_node[0]>=0 and new_node[1]>=0:
        return(new_node,True)
    else:
        return(curr_node,False)

# print('No errors in Action Functions')
```

In [6]:
```
## ---------------------------------------------------------------------------------------
#                                 Step 4 -->  Represent Map as Graph
## ---------------------------------------------------------------------------------------

# building a graph on which the Dijkstra Algorithm will be implemented
# use graph data structure

def map2graph(start,Map_height,Map_width):
    i, j = start[0], start[1]
    graph_map={}
    if i < Map_height and j < Map_width:

        # if the Node is at a Map corner
        if i == 0 and j == 0: # Bottom Left Corner (0,0)
            graph_map[(i,j)]={(i+1,j+1),(i+1,j),(i,j+1)}
        elif i == Map_height-1 and j ==0: # Bottom Right Corner (400,0)
            graph_map[(i,j)]={(i-1,j),(i-1,j+1),(i,j+1)}
        elif i == Map_height-1 and j==Map_width-1: # Top Right Corner (400,250)
            graph_map[(i,j)]={(i-1,j),(i-1,j-1),(i,j-1)}
        elif j == Map_width-1 and i ==0: # Top Left Corner (0,250)
            graph_map[(i,j)]={(i,j-1),(i+1,j-1),(i+1,j)}

        # if the Node is along a Map border
        elif i == Map_height-1 and j!=0 and j!=Map_width-1: # Left Border
            graph_map[(i,j)]={(i,j-1),(i,j+1),(i-1,j-1),(i-1,j),(i-1,j+1)}
        elif j == 0 and i!=0 and i!=Map_height-1: # Bottom Border
            graph_map[(i,j)]={(i-1,j),(i+1,j),(i+1,j+1),(i,j+1),(i-1,j+1)}
        elif i == 0 and j!=0 and j!=Map_width-1: # Right Border
            graph_map[(i,j)]={(i,j-1),(i,j+1),(i+1,j-1),(i+1,j),(i+1,j+1)}
        elif j == Map_width-1 and i!=0 and i!=Map_height-1: # Top Border
            graph_map[(i,j)]={(i-1,j),(i+1,j),(i+1,j-1),(i,j-1),(i-1,j-1)}

        # for any other Node in map
        else:
            graph_map[(i,j)]={(i-1,j),(i-1,j+1),(i-1,j-1),(i+1,j-1),(i+1,j),(i+1,j+1),(i,j-

        return(graph_map)

    else:
        return('The Start Node is outside the bounds of the Map')
```

In [7]:
```python
## ------------------------------------------------------------------------------------------------
#                                     Step 5 --> Cost of Action Steps
## ------------------------------------------------------------------------------------------------

def cost(graph,start):
    cost_dic= {} # let key: node, value: cost
    for key,value in graph.items():
        cost_dic[key]={}
        U,D,R,L = Right(key),Left(key), Up(key), Down(key)
        UpL, UpR, DwL, DwR = UpLeft(key),UpRight(key), DownLeft(key), DownRight(key)
        for n in value:
            # Assign Costs
            if (n == U[0]) or (n == L[0]) or ( n == U[0]) or (n == D[0]):
                cost_dic[key][n] = 1
            elif (n == UpL[0]) or (n == UpR[0]) or (n == DwL[0]) or (n == DwR[0]):
                cost_dic[key][n]= 1.4
    return(cost_dic)
```

In [8]:
```python
## ------------------------------------------------------------------------------------------------
#                                     Step 6 --> Dijkstra Algorithm
## ------------------------------------------------------------------------------------------------

OpenList = {}
ClosedList = []
BackTrackList = {}

def Dijkstra(graph,start):
    OpenList[start]=0
    ClosedList.append(start)
    priority_queue = [(0,start)]
    for node,edge in graph.items(): # retrieve nodes and edges from graph
        OpenList[node]= float('inf')
    Goal_Reached = False
    while len(priority_queue)>0 and Goal_Reached == False:
        dist,node = heapq.heappop(priority_queue)
        for nghbr_node,cost in graph[node].items():
            Cost2Come = dist + cost
            if Cost2Come < OpenList[nghbr_node]:
                BackTrackList[nghbr_node] = {}
                BackTrackList[nghbr_node][Cost2Come] = node
                OpenList[nghbr_node]= Cost2Come
                heapq.heappush(priority_queue, (Cost2Come, nghbr_node))
                if nghbr_node not in ClosedList:
                    ClosedList.append(nghbr_node)
                    if nghbr_node == goal:
                        print('GOAL Node Reached !!')
                        Goal_Reached = True
                        break

    print("Total Cost to Reach Goal: ",Cost2Come)

    return(OpenList,ClosedList,BackTrackList)
```

In [9]:
```python
## ---------------------------------------------------------------------------------
#                                    Step 7 -->  Back-Tracking
## ---------------------------------------------------------------------------------

def BackTrack(BackTrack_Dict,goal,start):#goal is the starting point now and start is the g
    BackTrackList_new = []
    BackTrackList_new.append(start)
    break_loop = False
    while break_loop == False:
        for key,value in BackTrackList.items():
            for _,val in value.items():
                if key==start:
                    if val not in BackTrackList_new:
                        BackTrackList_new.append(start)
                    start = val
                    if val == goal:
                        break_loop = True
                        break
    #returns the backtracked list
    return(BackTrackList_new)
```

In [10]:
```python
## ---------------------------------------------------------------------------------
#                                    Step 8 -->  Build Main Function
## ---------------------------------------------------------------------------------

def main(Map_width,Map_height,start,goal):
    map_pts = []
    OpenList = {}
    BackTrack = {}
    ClosedList = []
    for i in range(Map_width + 1):
        for j in range(Map_height + 1):
            map_pts.append((i,j)) #appending to the list
    ObstacleList = Obstacles(map_pts,start,goal)
    graph_base = {}
    for i in range(Map_width-1,-1,-1):
        for j in range(Map_height-1,-1,-1):
            graph_map = map2graph((i,j),Map_width,Map_height)
            graph_base[(i,j)] = graph_map[(i,j)]
    for key,value in graph_base.items():
        value_copy = value.copy()
        for cord in value_copy:
            if cord in ObstacleList:
                value.remove(cord)
    graph_base_copy=graph_base.copy()
    for key,value in graph_base_copy.items():
        if key in cord:
            del graph_base[key]
    graph = cost(graph_base,start)
    OpenList,ClosedList,BackTrack= Dijkstra(graph,start)
    OpenList_copy = OpenList.copy()
    for k,v in OpenList_copy.items():
        if OpenList_copy[k] == float('inf'):
            del OpenList[k]
    return(OpenList,ClosedList,BackTrack,ObstacleList)
```

In [11]:

```python
## ----------------------------------------------------------------------------------------
#                              Step 9 -->  Call Main Functions
## ----------------------------------------------------------------------------------------

print('Running Main Function')

start_time = time.time()

Map_width = 400
Map_height = 250
OpenList,ClosedList,BackTrack_Dict,ObstacleList= main(Map_width,Map_height,start,goal)
```

```
Running Main Function
GOAL Node Reached !!
Total Cost to Reach Goal:  440.9999999999985
```

In [12]:

```
## -----------------------------------------------------------------------------
#                            Step 10 -->  Call BackTracking Functions
## -----------------------------------------------------------------------------

BackTrackList = BackTrack(BackTrack_Dict,start,goal)
print(BackTrackList)

print("Time to Run Algorithm: ",time.time() - start_time, "seconds")
```

```
[(9, 223), (9, 223), (10, 222), (11, 221), (12, 220), (13, 219), (14, 218), (15, 217), (1
6, 216), (17, 215), (18, 214), (19, 213), (20, 212), (21, 211), (22, 210), (23, 209), (24,
208), (25, 208), (26, 208), (27, 208), (28, 208), (29, 208), (30, 208), (31, 208), (32, 20
8), (33, 208), (34, 208), (35, 208), (36, 208), (37, 208), (38, 208), (39, 208), (40, 20
8), (41, 208), (42, 208), (43, 208), (44, 208), (45, 208), (46, 208), (47, 208), (48, 20
8), (49, 208), (50, 208), (51, 208), (52, 208), (53, 208), (54, 208), (55, 208), (56, 20
8), (57, 208), (58, 208), (59, 208), (60, 208), (61, 208), (62, 208), (63, 208), (64, 20
8), (65, 208), (66, 208), (67, 208), (68, 208), (69, 208), (70, 208), (71, 208), (72, 20
8), (73, 208), (74, 208), (75, 208), (76, 208), (77, 208), (78, 208), (79, 208), (80, 20
8), (81, 208), (82, 208), (83, 208), (84, 208), (85, 208), (86, 208), (87, 208), (88, 20
8), (89, 208), (90, 208), (91, 208), (92, 208), (93, 208), (94, 208), (95, 208), (96, 20
8), (97, 208), (98, 208), (99, 208), (100, 208), (101, 208), (102, 208), (103, 208), (104,
208), (105, 208), (106, 208), (107, 208), (108, 208), (109, 207), (110, 206), (111, 205),
(112, 204), (113, 203), (114, 202), (115, 201), (116, 200), (117, 199), (118, 198), (119,
197), (120, 196), (121, 195), (122, 194), (123, 193), (124, 192), (125, 191), (126, 190),
(127, 189), (128, 188), (129, 187), (130, 186), (131, 185), (132, 184), (133, 183), (134,
182), (135, 181), (136, 180), (137, 179), (138, 178), (139, 177), (140, 176), (141, 175),
(142, 174), (143, 173), (144, 172), (145, 171), (146, 170), (147, 169), (148, 168), (149,
167), (150, 166), (151, 165), (152, 164), (153, 163), (154, 162), (155, 161), (156, 160),
(157, 159), (158, 158), (159, 157), (160, 156), (161, 155), (162, 154), (163, 153), (164,
152), (165, 151), (166, 150), (167, 149), (168, 148), (169, 147), (170, 146), (171, 145),
(172, 144), (173, 143), (174, 142), (175, 141), (176, 141), (177, 141), (178, 141), (179,
141), (180, 141), (181, 141), (182, 141), (183, 141), (184, 141), (185, 141), (186, 141),
(187, 141), (188, 141), (189, 141), (190, 141), (191, 141), (192, 141), (193, 141), (194,
141), (195, 141), (196, 141), (197, 141), (198, 141), (199, 141), (200, 141), (201, 140),
(202, 140), (203, 139), (204, 139), (205, 138), (206, 137), (207, 137), (208, 136), (209,
136), (210, 135), (211, 135), (212, 134), (213, 133), (214, 133), (215, 132), (216, 132),
(217, 131), (218, 131), (219, 130), (220, 129), (221, 129), (222, 128), (223, 128), (224,
127), (225, 127), (226, 126), (227, 125), (228, 125), (229, 124), (230, 124), (231, 123),
(232, 122), (233, 122), (234, 121), (235, 120), (236, 119), (237, 118), (238, 117), (239,
116), (240, 115), (241, 114), (242, 113), (243, 112), (244, 111), (245, 110), (246, 109),
(247, 108), (248, 108), (249, 108), (250, 108), (251, 108), (252, 108), (253, 108), (254,
108), (255, 108), (256, 108), (257, 108), (258, 108), (259, 108), (260, 108), (261, 108),
(262, 108), (263, 108), (264, 108), (265, 108), (266, 108), (267, 108), (268, 108), (269,
108), (270, 108), (271, 108), (272, 108), (273, 108), (274, 108), (275, 108), (276, 107),
(277, 107), (278, 107), (279, 107), (280, 107), (281, 107), (282, 107), (283, 107), (284,
107), (285, 107), (286, 107), (287, 107), (288, 107), (289, 107), (290, 107), (291, 107),
(292, 107), (293, 107), (294, 107), (295, 107), (296, 107), (297, 107), (298, 107), (299,
107), (300, 107), (301, 107), (302, 107), (303, 107), (304, 107), (305, 107), (306, 107),
(307, 107), (308, 107), (309, 107), (310, 107), (311, 107), (312, 107), (313, 107), (314,
107), (315, 107), (316, 107), (317, 107), (318, 107), (319, 107), (320, 107), (321, 107),
(322, 107), (323, 107), (324, 107), (325, 107), (326, 107), (327, 107), (328, 107), (329,
107), (330, 107), (331, 107), (332, 107), (333, 107), (334, 107), (335, 107), (336, 107),
(337, 107), (338, 107), (339, 107), (340, 106), (341, 105), (342, 104), (343, 103), (344,
102), (345, 101), (346, 100), (347, 99), (348, 98), (349, 97), (350, 96), (351, 95), (352,
94), (353, 93), (354, 92), (355, 91), (356, 90), (357, 89), (358, 88), (359, 87), (360, 8
6), (361, 85), (362, 84), (363, 83), (364, 82), (365, 81), (366, 80), (367, 79), (368, 7
8), (369, 77), (370, 76), (371, 75), (372, 74), (373, 73), (374, 72), (375, 71), (376, 7
0), (377, 69), (378, 68), (379, 67), (380, 66), (381, 65), (382, 65), (383, 64), (384, 6
3), (385, 63), (386, 63)]
Time to Run Algorithm:  319.5883388519287 seconds
```

In [13]:

```
## -------------------------------------------------------------------------------------------
#                                     Step 11 -->  Visualization
## -------------------------------------------------------------------------------------------

# display obstacle map using cv2

blank_map = np.zeros((251,401,3),np.uint8) # 251 rows, 401 col

for pts in ObstacleList:
    x,y = pts[1], pts[0]
    blank_map[(x,y)]=[0,0,255]

main_map = np.flipud(blank_map)
cv2.imshow('The Map with Obstacles',main_map)
cv2.waitKey(0)
cv2.destroyAllWindows()

# make copy of map for other plots
backtracked_plot = main_map.copy()
visited_plot = main_map.copy()
visited_plot = cv2.resize(visited_plot,(Map_width,Map_height))
```

In [14]:
```
# Display Animation of Dijkstra Path Planning using Pygame Library

pygame.init()
display_width = 400
display_height = 250
gameDisplay = pygame.display.set_mode((display_width,display_height),pygame.FULLSCREEN)
pygame.display.set_caption('Dijkstra Path Planning Algorithm')
surf = pygame.surfarray.make_surface(visited_plot)
clock = pygame.time.Clock()
loop_break = False

while not loop_break:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            loop_break = True

    gameDisplay.fill((0,0,0))
    for path in ClosedList:
        if path not in visited_plot:
            x = path[0]
            y = abs(250-path[1])
            pygame.draw.rect(gameDisplay, (70,70,245), [x,y,1,1])
            pygame.display.flip()

    for path in BackTrackList:
        pygame.time.wait(5)
        x = path[0]
        y = abs(250-path[1])
        pygame.draw.rect(gameDisplay, (255,255,0), [x,y,1,1])
        pygame.display.flip()

        loop_break = True

pygame.quit()
```

```
C:\Users\jai10\Anaconda3\envs\pyjai\lib\site-packages\ipykernel_launcher.py:19: Deprecatio
nWarning: elementwise comparison failed; this will raise an error in the future.
```

In [*]:
```
for path in ClosedList: # Display map with all Visited Nodes using cv2
    x,y  = path[0], path[1]
    backtracked_plot[(display_height-y,x)]=[245,70,70]
cv2.imshow('Map of all Visited Nodes',backtracked_plot)
cv2.waitKey(0)
cv2.destroyAllWindows()

for path in BackTrackList: # Display plot with Dijkstra Path using cv2
    x,y  = path[0], path[1]
    backtracked_plot[(display_height-y,x)]=[0,255,255]

cv2.imshow('Map of Dijkstra Algorithm Generated Path', backtracked_plot)
cv2.waitKey(0)
cv2.destroyAllWindows()

print('Program Fully Executed')
```

In [ ]: