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 -->  5
y coordinate of START node -->  5
x coordinate of GOAL node -->  120
y coordinate of GOAL node -->  223

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

In [4]:

```python
## ------------------------------------------------------------------------------------
#                                     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]:

```
## ----------------------------------------------------------------------------------------
#                                       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]:

```
## ---------------------------------------------------------------------------------
#                                   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]:

```
## ---------------------------------------------------------------------------------
#                                   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]:
```
## ------------------------------------------------------------------------------------
#                           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:  268.0000000000001
```

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")
```

```
[(120, 223), (120, 223), (119, 222), (118, 221), (117, 220), (116, 219), (115, 218), (114,
217), (113, 216), (112, 215), (111, 214), (110, 213), (110, 212), (110, 211), (110, 210),
(110, 209), (110, 208), (109, 207), (108, 206), (107, 205), (106, 204), (105, 203), (105,
202), (105, 201), (105, 200), (105, 199), (105, 198), (105, 197), (105, 196), (105, 195),
(105, 194), (105, 193), (105, 192), (105, 191), (105, 190), (105, 189), (105, 188), (105,
187), (105, 186), (105, 185), (105, 184), (105, 183), (105, 182), (105, 181), (105, 180),
(105, 179), (105, 178), (105, 177), (105, 176), (105, 175), (105, 174), (105, 173), (105,
172), (105, 171), (105, 170), (105, 169), (105, 168), (105, 167), (105, 166), (105, 165),
(105, 164), (105, 163), (105, 162), (105, 161), (105, 160), (105, 159), (105, 158), (105,
157), (105, 156), (105, 155), (105, 154), (105, 153), (105, 152), (105, 151), (105, 150),
(105, 149), (105, 148), (105, 147), (105, 146), (105, 145), (105, 144), (105, 143), (105,
142), (105, 141), (105, 140), (105, 139), (105, 138), (105, 137), (105, 136), (105, 135),
(105, 134), (105, 133), (105, 132), (105, 131), (105, 130), (105, 129), (105, 128), (105,
127), (105, 126), (105, 125), (105, 124), (105, 123), (105, 122), (105, 121), (105, 120),
(105, 119), (105, 118), (105, 117), (105, 116), (105, 115), (105, 114), (105, 113), (105,
112), (105, 111), (105, 110), (105, 109), (105, 108), (105, 107), (105, 106), (105, 105),
(105, 104), (105, 103), (105, 102), (104, 101), (103, 100), (102, 99), (101, 98), (100, 9
7), (99, 96), (98, 95), (97, 94), (96, 93), (95, 92), (94, 91), (93, 90), (92, 89), (91, 8
8), (90, 87), (89, 86), (88, 85), (87, 84), (86, 83), (85, 82), (84, 81), (83, 80), (82, 7
9), (81, 78), (80, 77), (79, 76), (78, 75), (77, 74), (76, 73), (75, 72), (74, 71), (73, 7
0), (72, 69), (71, 68), (70, 67), (69, 66), (68, 65), (67, 64), (66, 63), (65, 62), (64, 6
1), (63, 60), (62, 59), (61, 58), (60, 57), (59, 56), (58, 55), (57, 54), (56, 53), (55, 5
2), (54, 51), (53, 50), (52, 49), (51, 48), (50, 47), (49, 46), (48, 45), (47, 44), (46, 4
3), (45, 42), (44, 41), (43, 40), (42, 39), (41, 38), (40, 37), (39, 36), (38, 35), (37, 3
4), (36, 33), (35, 32), (34, 31), (33, 30), (32, 29), (31, 28), (30, 27), (29, 26), (28, 2
5), (27, 24), (26, 23), (25, 22), (24, 21), (23, 20), (22, 19), (21, 18), (20, 17), (19, 1
6), (18, 15), (17, 14), (16, 13), (15, 12), (14, 11), (13, 10), (12, 9), (11, 8), (10, 7),
(9, 6), (8, 5), (7, 4), (6, 3), (5, 4)]
Time to Run Algorithm:  203.461256980896 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]:

```python
# 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 [*]:

```python
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 [ ]: