```python
import numpy as np
import time
import cv2
from queue import PriorityQueue

# Define the move functions (up, down, left, right, up-left, up-right, down-left, down-right)
move_up = lambda node: ((node[0] - 1, node[1]), 1)
move_down = lambda node: ((node[0] + 1, node[1]), 1)
move_left = lambda node: ((node[0], node[1] - 1), 1)
move_right = lambda node: ((node[0], node[1] + 1), 1)
move_up_left = lambda node: ((node[0] - 1, node[1] - 1), np.sqrt(2))
move_up_right = lambda node: ((node[0] - 1, node[1] + 1), np.sqrt(2))
move_down_left = lambda node: ((node[0] + 1, node[1] - 1), np.sqrt(2))
move_down_right = lambda node: ((node[0] + 1, node[1] + 1), np.sqrt(2))

# Define obstacles (rectangles, hexagons, triangles)
def rectangle1(x, y):                                   # Define the rectangle function
    return 0 <= y <= 100 and 100 <= x <= 150

def rectangle2(x, y):
    return 150 <= y <= 250 and 100 <= x <= 150

def hexagon(x, y):                              # Define the hexagon function
    return (75/2) * abs(x-300)/75 + 50 <= y <= 250 - (75/2) * abs(x-300)/75 - 50 and 225 <= x <= 375

def triangle(x, y):                   # Define the triangle function
    return (200/100) * (x-460) + 25 <= y <= (-200/100) * (x-460) + 225 and 460 <= x <= 510

# Define the dictionary of obstacles (rectangles, hexagons, triangles) and their functions
```

```
equation = {
    "Rectangle1": rectangle1,
    "Rectangle2": rectangle2,
    "Hexagon": hexagon,
    "Triangle": triangle}

plot_width, plot_height, clearance = 600, 250, 5                        # Define the plot size and clearance
pixels = np.full((plot_height, plot_width, 3), 255, dtype=np.uint8)        # Create a plot (white background)

# Draw obstacles
for i in range(plot_height):
    for j in range(plot_width):
        for eqn in equation.values():
            if eqn(j, i):
                pixels[i, j] = [0, 0, 0]  # obstacle
                break
        else:
            for eqn in equation.values():
                if eqn(j, i-clearance) or eqn(j, i+clearance) or eqn(j-clearance, i) or eqn(j+clearance, i):
                    pixels[i, j] = [192, 192, 192]  # clearance

def is_valid_node(node):                                         # Check if the node is valid
    x, y = node                                                  # Get the x and y coordinates of the node
    return 0 <= x < plot_width and 0 <= y < plot_height and (pixels[y, x] == [255, 255, 255]).all()

def is_goal(current_node, goal_node):
    return current_node == goal_node
```

```
def backtrack_path(parents, start_node, goal_node, animation):
    height, _, _ = animation.shape                                          # Get the height of the plot
    path, current_node = [goal_node], goal_node
    while current_node != start_node:
        path.append(current_node)
        current_node = parents[current_node]
        animation[height - 1 - current_node[1], current_node[0]] = (255, 0, 0)  # Mark path (in Red)
        cv2.imshow('Animation', animation)
        cv2.waitKey(1)
    path.append(start_node)
    return path[::-1]


def dijkstra(start_node, goal_node):
    open_list = PriorityQueue()
    closed_list = set()
    cost_to_come = {start_node: 0}
    cost = {start_node: 0}
    parent = {start_node: None}
    open_list.put((0, start_node))
    animation = pixels.copy()
    visited = set([start_node])
    height, _, _ = animation.shape
    while not open_list.empty():
        _, current_node = open_list.get()


        closed_list.add(current_node)
        animation[height - 1 - current_node[1], current_node[0]] = (0, 255, 0)  # Mark current node in green
as visisted
        cv2.imshow('Animation', animation)
```

```
        cv2.waitKey(1)

        if is_goal(current_node, goal_node):
            path = backtrack_path(parent, start_node, goal_node, animation)
            # Mark start and goal nodes as green
            animation[height - 1 - start_node[1], start_node[0]] = (0, 255, 0)
            animation[height - 1 - goal_node[1], goal_node[0]] = (0, 255, 0)
            cv2.imshow('Animation', animation)
            cv2.waitKey(0)
            cv2.destroyAllWindows()
            # Print the final cost
            print("Final Cost: ", cost[goal_node])
            return path

        for move_func in [move_up, move_down, move_left, move_right, move_up_left, move_up_right,
move_down_left, move_down_right]:
            new_node, move_cost = move_func(current_node)

            if is_valid_node(new_node) and new_node not in closed_list:
                new_cost_to_come = cost_to_come[current_node] + move_cost

                if new_node not in cost_to_come or new_cost_to_come < cost_to_come[new_node]:
                    cost_to_come[new_node] = new_cost_to_come
                    cost[new_node] = new_cost_to_come
                    parent[new_node] = current_node
                    open_list.put((new_cost_to_come, new_node))
                    visited.add(new_node)
    cv2.imshow('Animation', animation)
    cv2.waitKey(0)
```

```python
    cv2.destroyAllWindows()
    return None

# Get start and goal nodes from user
while True:
    # Get start and goal nodes from user input
    start_str = input("Enter the start node (format 'x y'): ")
    start_node = tuple(map(int, start_str.split()))
    goal_str = input("\nEnter the goal node (format 'x y'): ")
    goal_node = tuple(map(int, goal_str.split()))

    # Check if start and goal nodes are valid
    if not is_valid_node(start_node):
        print(f"Invalid {start_str} start node. Give a valid node i.e. not in the obstacle space.")
        continue
    if not is_valid_node(goal_node):
        print(f"Invalid {goal_str} goal node. Give a valid node i.e. not in the obstacle space.")
        continue
    # If both nodes are valid, break out of the loop
    break

# Run Dijkstra's Algorithm and print the time taken to find the path (if found)
start_time = time.time()
path = dijkstra(start_node, goal_node)
if path is None:
    print("Path Not Found")
else:
    print("Goal Node Achieved")
end_time = time.time()
```

```
print("Time Taken:", end_time - start_time, "seconds\n")
```