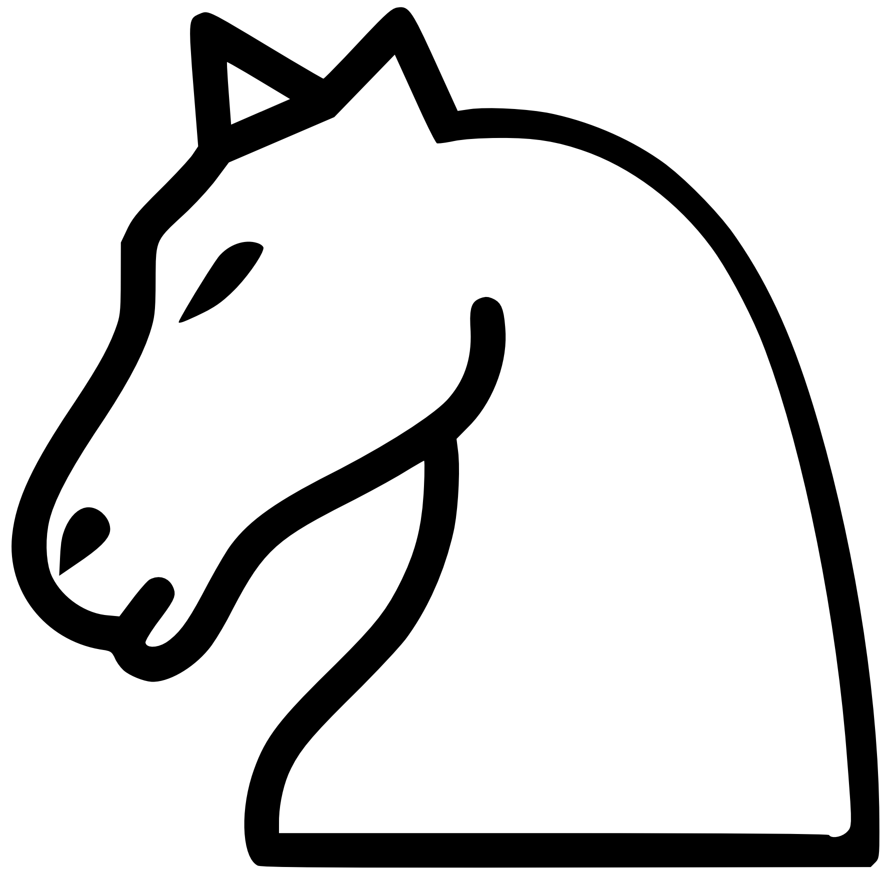


Knights Travails

a DSA project
by

Ahtisham Uddin and Mubashir Anees

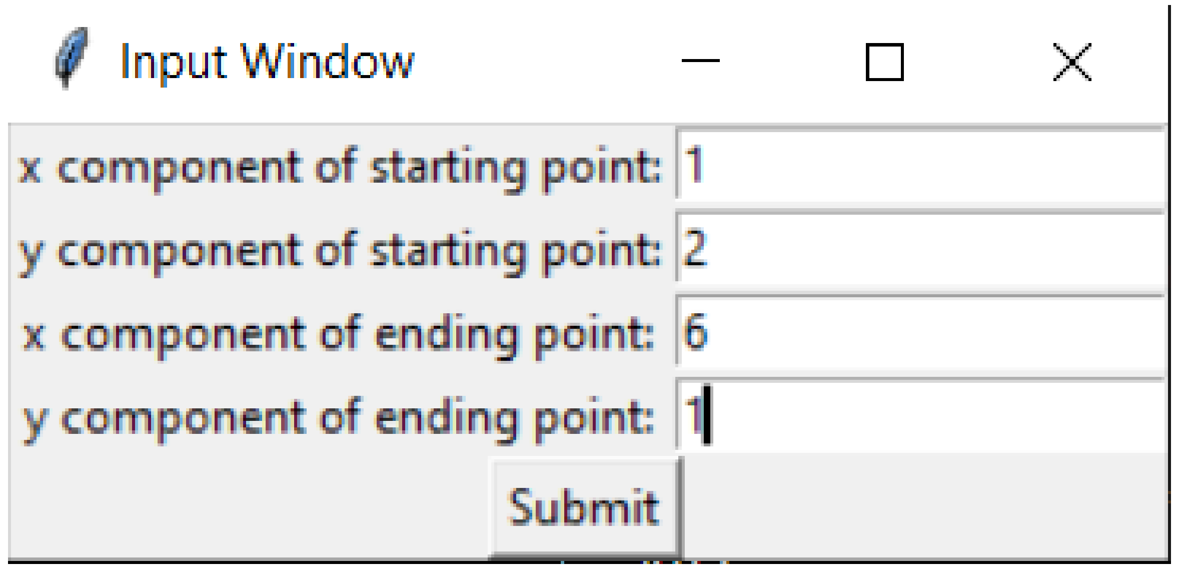
Objective

The Idea is that a knight can travel from any tile to any other tile on a chessboard and in this project we are interested in finding the least moves the knight has to make to reach its destination tile from a starting tile

Our Approach

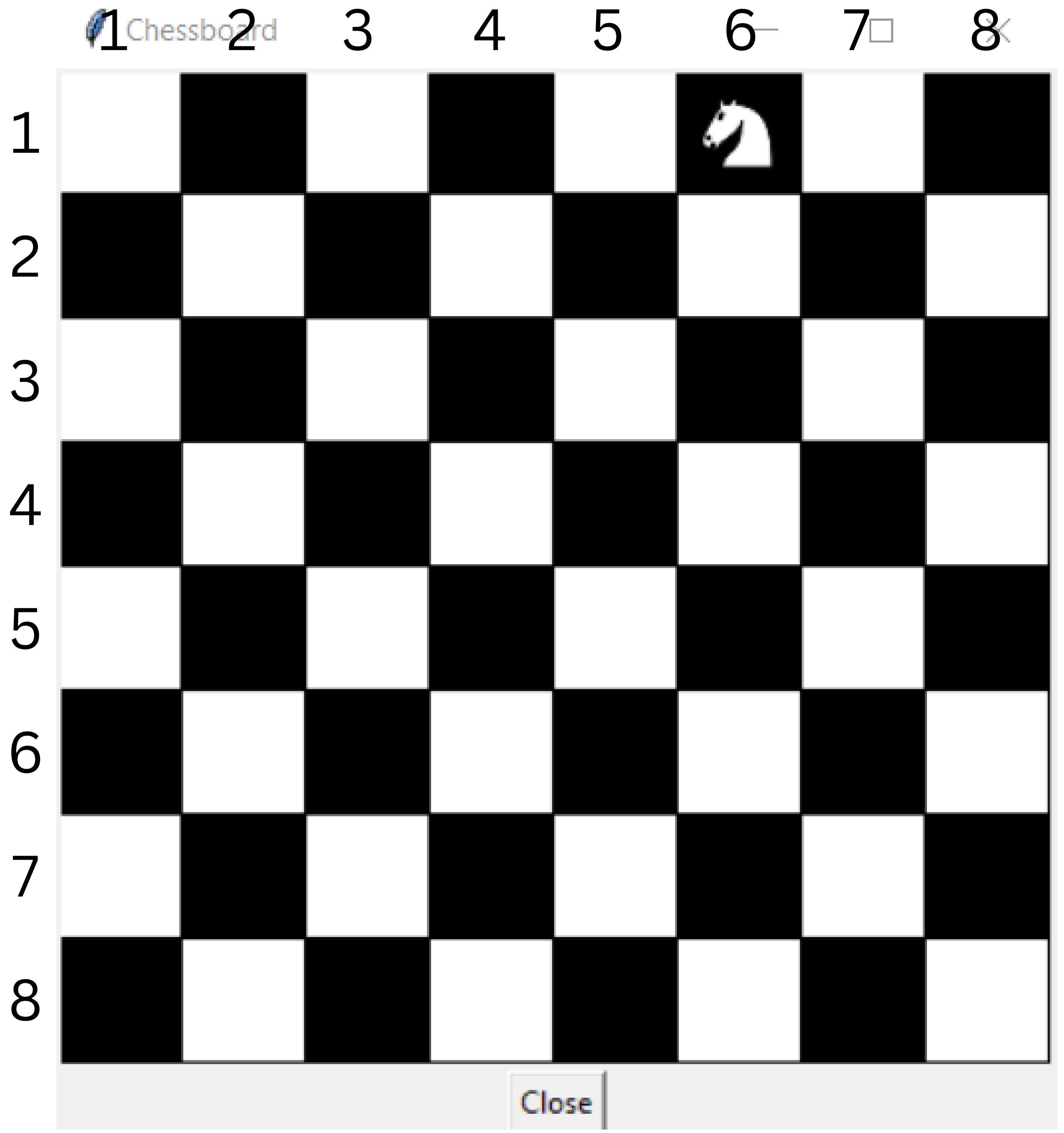
We decided to incorporate the chessboard using a graph and find the least steps the knight makes using Dijkstra's algorithm. We also created a GUI using pythons Tkinter library

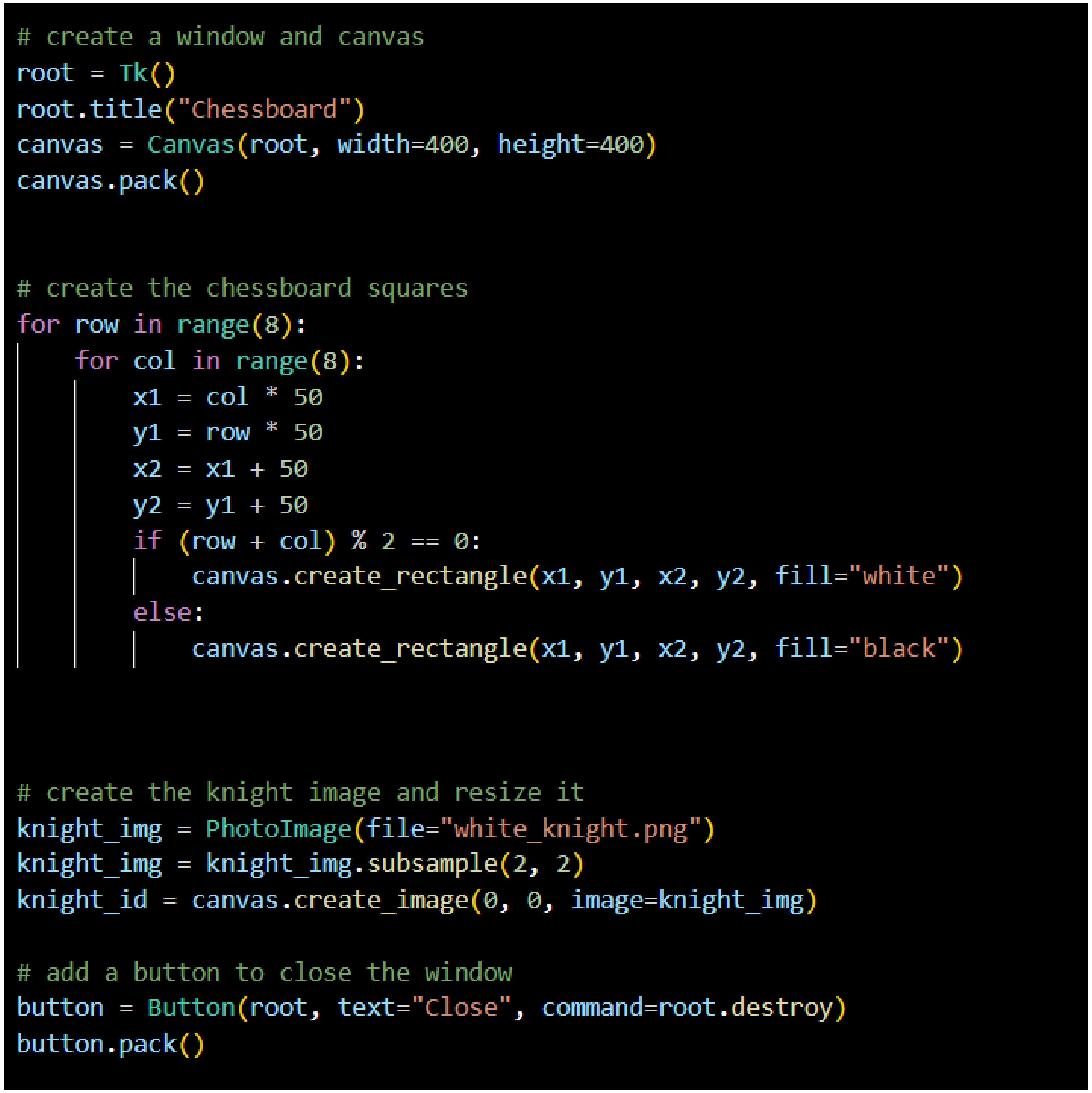
- This is the Window that visualizes the movement of the Knight
- We decided to use the x and y coordinates that are shown in the image ⟶
- The code will take input in the window below and find the shortest path. The number of steps in this path would be the output



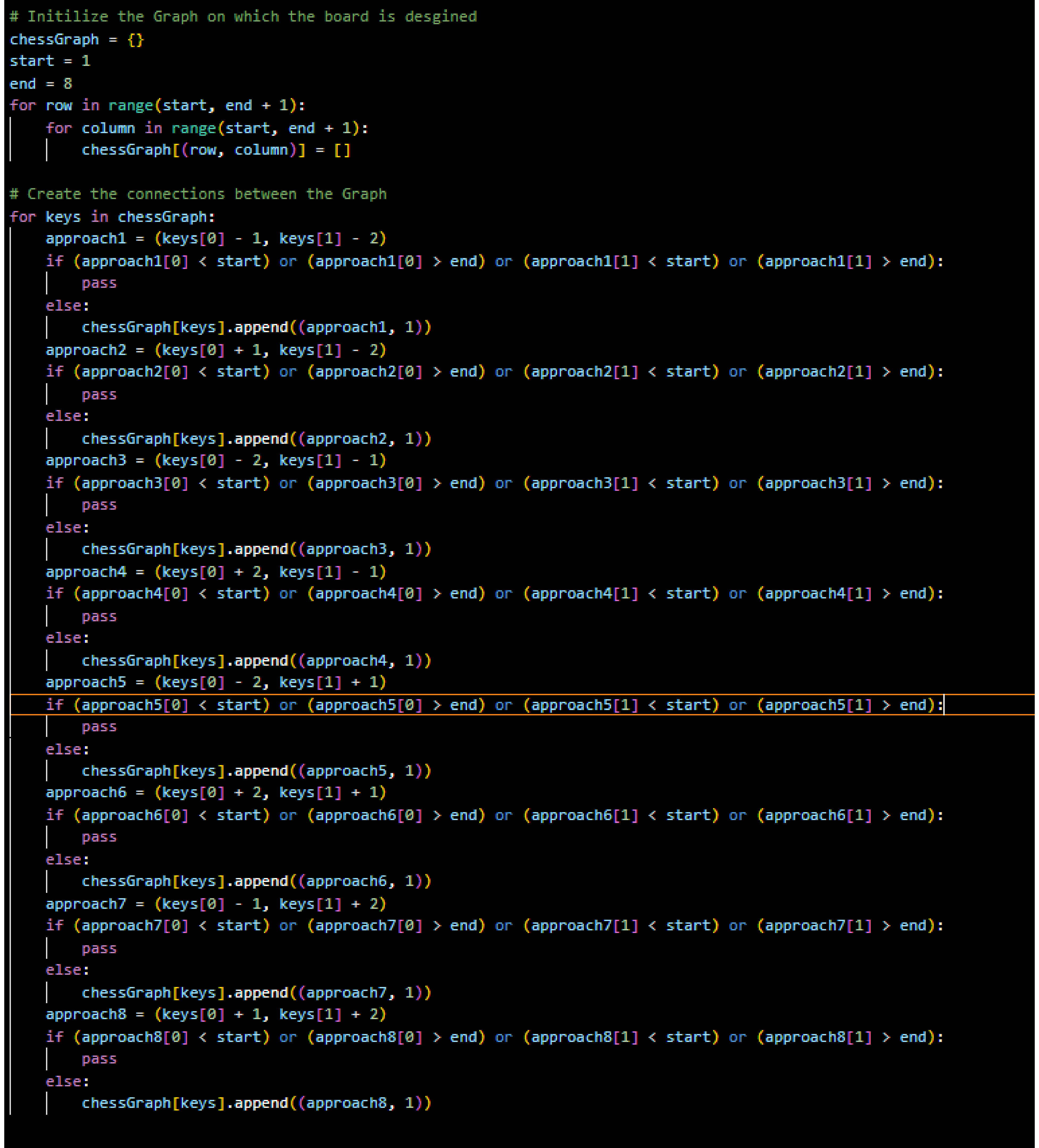
Input Window

x component of starting point:	1
y component of starting point:	2
x component of ending point:	6
y component of ending point:	1

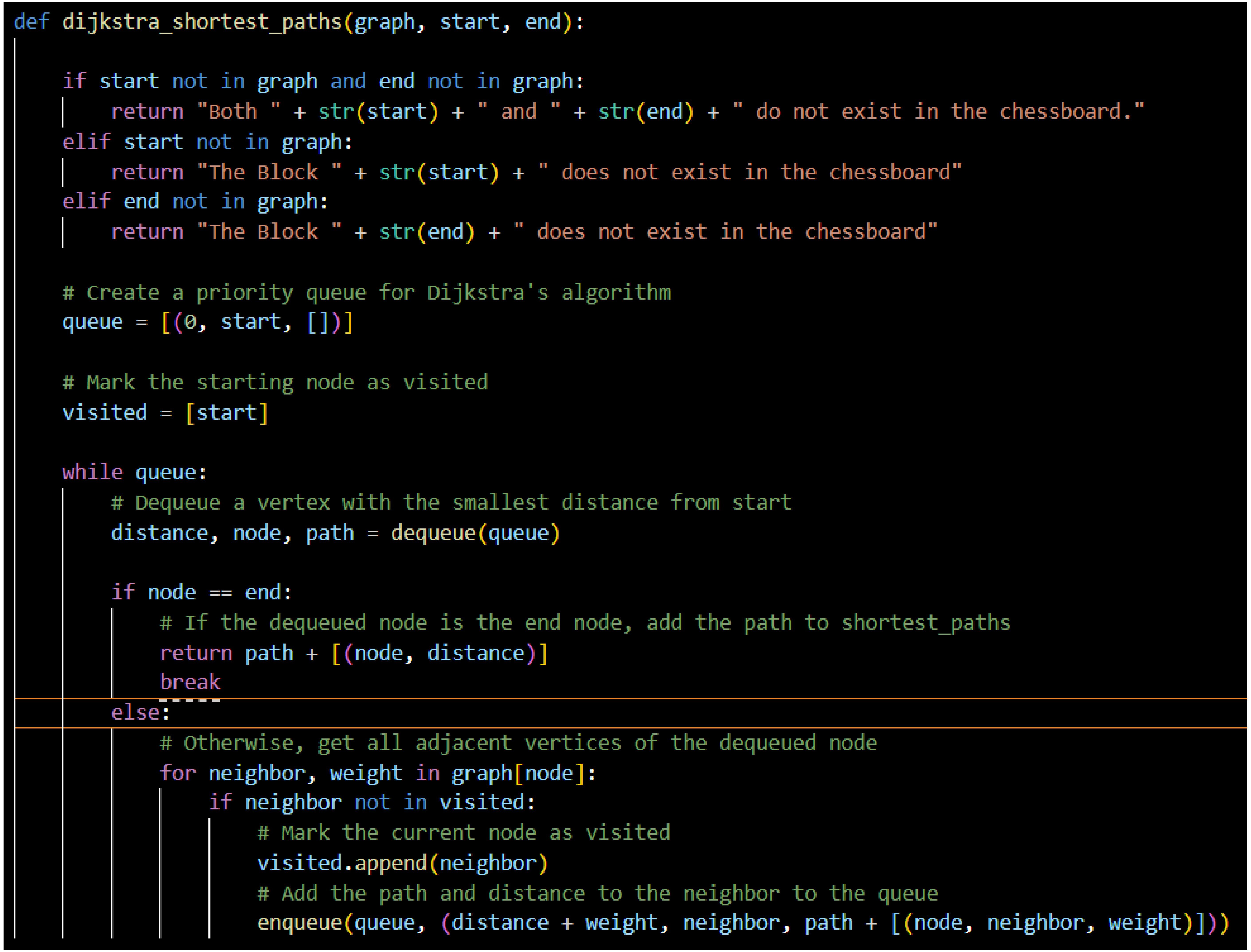




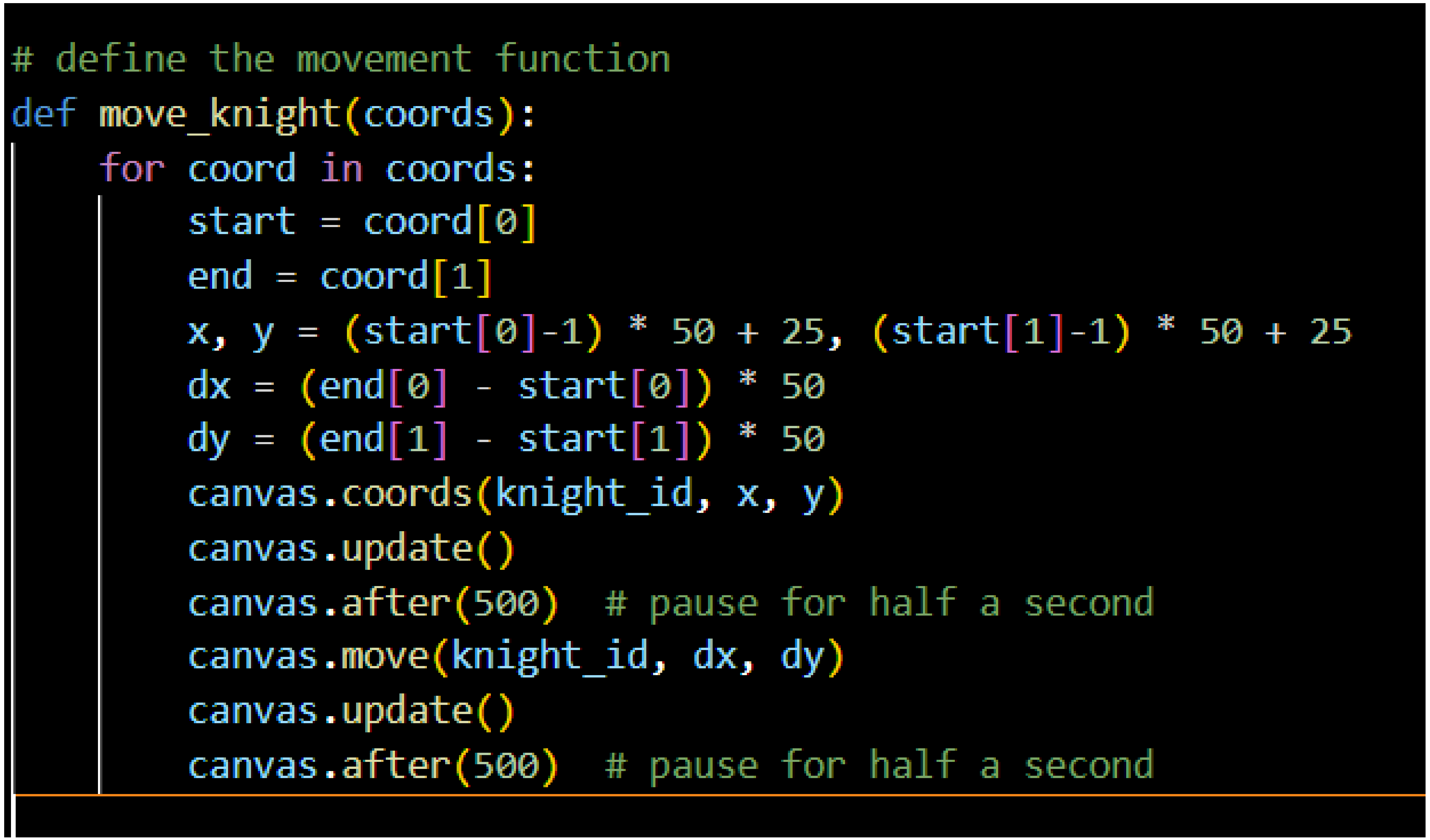
```python
# create a window and canvas
root = Tk()
root.title("Chessboard")
canvas = Canvas(root, width=400, height=400)
canvas.pack()


# create the chessboard squares
for row in range(8):
    for col in range(8):
        x1 = col * 50
        y1 = row * 50
        x2 = x1 + 50
        y2 = y1 + 50
        if (row + col) % 2 == 0:
            canvas.create_rectangle(x1, y1, x2, y2, fill="white")
        else:
            canvas.create_rectangle(x1, y1, x2, y2, fill="black")



# create the knight image and resize it
knight_img = PhotoImage(file="white_knight.png")
knight_img = knight_img.subsample(2, 2)
knight_id = canvas.create_image(0, 0, image=knight_img)

# add a button to close the window
button = Button(root, text="Close", command=root.destroy)
button.pack()
```



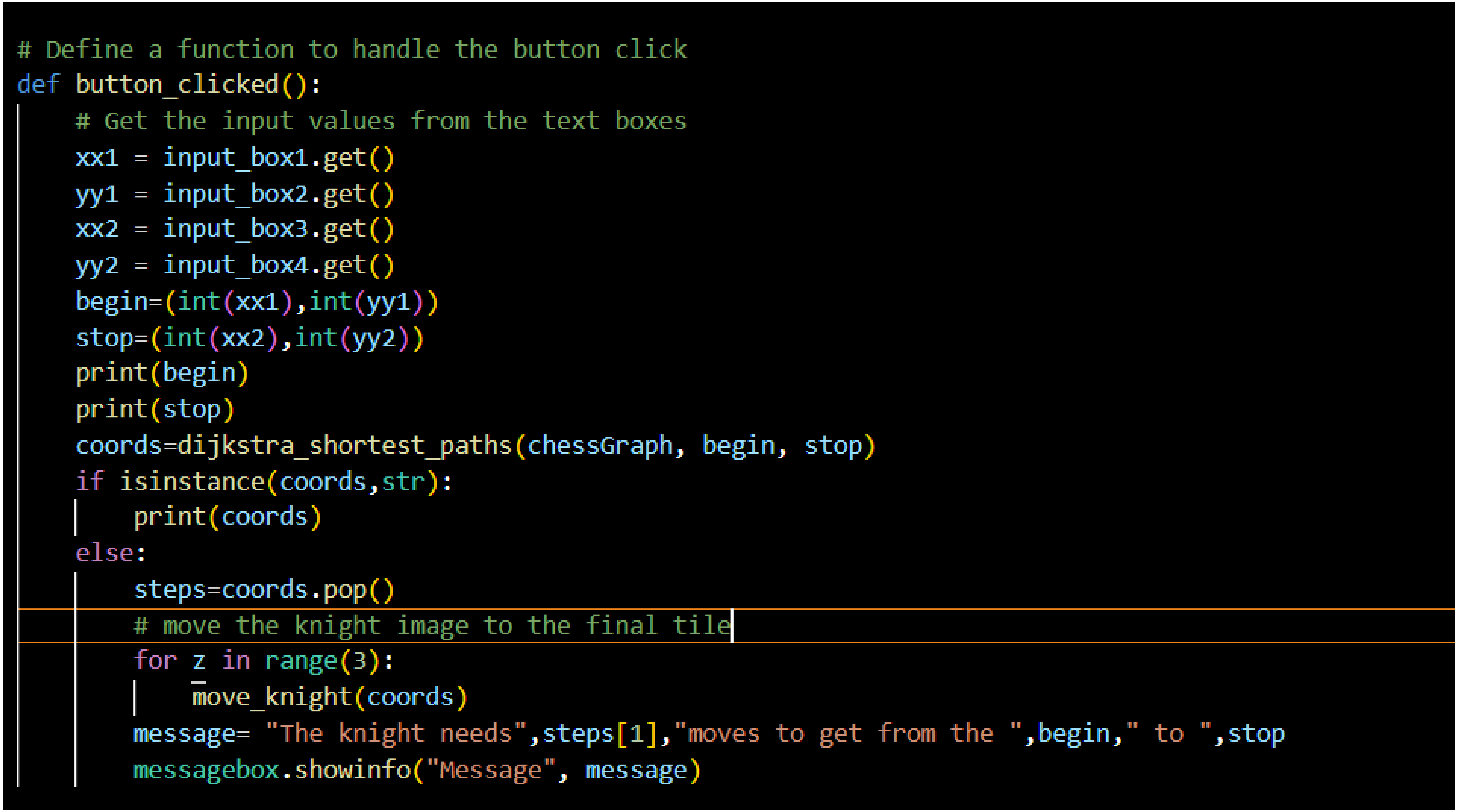
```python
# Initilize the Graph on which the board is desgined
chessGraph = {}
start = 1
end = 8
for row in range(start, end + 1):
    for column in range(start, end + 1):
        chessGraph[(row, column)] = []

# Create the connections between the Graph
for keys in chessGraph:
    approach1 = (keys[0] - 1, keys[1] - 2)
    if (approach1[0] < start) or (approach1[0] > end) or (approach1[1] < start) or (approach1[1] > end):
        pass
    else:
        chessGraph[keys].append((approach1, 1))
    approach2 = (keys[0] + 1, keys[1] - 2)
    if (approach2[0] < start) or (approach2[0] > end) or (approach2[1] < start) or (approach2[1] > end):
        pass
    else:
        chessGraph[keys].append((approach2, 1))
    approach3 = (keys[0] - 2, keys[1] - 1)
    if (approach3[0] < start) or (approach3[0] > end) or (approach3[1] < start) or (approach3[1] > end):
        pass
    else:
        chessGraph[keys].append((approach3, 1))
    approach4 = (keys[0] + 2, keys[1] - 1)
    if (approach4[0] < start) or (approach4[0] > end) or (approach4[1] < start) or (approach4[1] > end):
        pass
    else:
        chessGraph[keys].append((approach4, 1))
    approach5 = (keys[0] - 2, keys[1] + 1)
    if (approach5[0] < start) or (approach5[0] > end) or (approach5[1] < start) or (approach5[1] > end):
        pass
    else:
        chessGraph[keys].append((approach5, 1))
    approach6 = (keys[0] + 2, keys[1] + 1)
    if (approach6[0] < start) or (approach6[0] > end) or (approach6[1] < start) or (approach6[1] > end):
        pass
    else:
        chessGraph[keys].append((approach6, 1))
    approach7 = (keys[0] - 1, keys[1] + 2)
    if (approach7[0] < start) or (approach7[0] > end) or (approach7[1] < start) or (approach7[1] > end):
        pass
    else:
        chessGraph[keys].append((approach7, 1))
    approach8 = (keys[0] + 1, keys[1] + 2)
    if (approach8[0] < start) or (approach8[0] > end) or (approach8[1] < start) or (approach8[1] > end):
        pass
    else:
        chessGraph[keys].append((approach8, 1))
```



```python
def dijkstra_shortest_paths(graph, start, end):

    if start not in graph and end not in graph:
        return "Both " + str(start) + " and " + str(end) + " do not exist in the chessboard."
    elif start not in graph:
        return "The Block " + str(start) + " does not exist in the chessboard"
    elif end not in graph:
        return "The Block " + str(end) + " does not exist in the chessboard"

    # Create a priority queue for Dijkstra's algorithm
    queue = [(0, start, [])]

    # Mark the starting node as visited
    visited = [start]

    while queue:
        # Dequeue a vertex with the smallest distance from start
        distance, node, path = dequeue(queue)

        if node == end:
            # If the dequeued node is the end node, add the path to shortest_paths
            return path + [(node, distance)]
            break
        else:
            # Otherwise, get all adjacent vertices of the dequeued node
            for neighbor, weight in graph[node]:
                if neighbor not in visited:
                    # Mark the current node as visited
                    visited.append(neighbor)
                    # Add the path and distance to the neighbor to the queue
                    enqueue(queue, (distance + weight, neighbor, path + [(node, neighbor, weight)]))
```



```python
# define the movement function
def move_knight(coords):
    for coord in coords:
        start = coord[0]
        end = coord[1]
        x, y = (start[0]-1) * 50 + 25, (start[1]-1) * 50 + 25
        dx = (end[0] - start[0]) * 50
        dy = (end[1] - start[1]) * 50
        canvas.coords(knight_id, x, y)
        canvas.update()
        canvas.after(500)  # pause for half a second
        canvas.move(knight_id, dx, dy)
        canvas.update()
        canvas.after(500)  # pause for half a second
```



```python
# Define a function to handle the button click
def button_clicked():
    # Get the input values from the text boxes
    xx1 = input_box1.get()
    yy1 = input_box2.get()
    xx2 = input_box3.get()
    yy2 = input_box4.get()
    begin=(int(xx1),int(yy1))
    stop=(int(xx2),int(yy2))
    print(begin)
    print(stop)
    coords=dijkstra_shortest_paths(chessGraph, begin, stop)
    if isinstance(coords,str):
        print(coords)
    else:
        steps=coords.pop()
        # move the knight image to the final tile
        for z in range(3):
            move_knight(coords)
        message= "The knight needs",steps[1],"moves to get from the ",begin," to ",stop
        messagebox.showinfo("Message", message)
```

Time complexity and memory requirements?

Creating the chessboard and initializing the graph: This part of the code iterates over the chessboard and creates connections between nodes in the graph. The chessboard has dimensions of 8x8, so the number of nodes in the graph is constant (64 nodes). Therefore, the time complexity for this operation is O(1).

Dijkstra's algorithm for finding the shortest path: The implementation of Dijkstra's algorithm in the dijkstra_shortest_paths function is based on a priority queue. In the worst-case scenario, where all nodes are connected to each other, the number of edges in the graph is 8 times the number of nodes (each node can have up to 8 neighbors). Therefore, the time complexity of Dijkstra's algorithm is O(V + E * log(V)), where V is the number of nodes (constant) and E is the number of edges (constant). Hence, the time complexity for this operation is also O(1).

Moving the knight on the canvas: The move_knight function moves the knight image on the canvas based on the calculated coordinates. It iterates over the coordinates and performs canvas operations such as updating, moving, and pausing. The number of coordinates in the list is determined by the shortest path length, which can vary depending on the start and end positions. In the worst case, the number of coordinates would be equal to the number of cells on the chessboard (64). Therefore, the time complexity for moving the knight is O(N), where N is the number of coordinates in the shortest path.

Overall, the time complexity of the code can be considered as O(1) since the major operations have constant time complexities. The only operation that can vary is moving the knight on the canvas, which has a time complexity of O(N) in the worst case.