**Assignment**

**CSA0612 – Design and Analysis of Algorithms for Optimization**

|  |  |
| --- | --- |
| **Register Number** | **192324277** |
| **Name** | **SRINIDHI A** |

**Title: Autonomous Drone Navigation**

**Problem Statement:**

|  |
| --- |
| Design an algorithm that enables a drone to navigate through a complex environment autonomously, avoiding obstacles and optimizing flight path.  **Tasks**:   * + Design an algorithm to enable drones to navigate through complex environments, avoiding obstacles.   + Optimize flight path for battery life and distance minimization. |

**Flowchart for problem solving:**

[ Start ]

|

v

[ Initialize Drone Position, Target, and Environment Data ]

|

v

[ Perception: Detect Obstacles with Sensors ]

|

v

[ Path Planning: Calculate Path with D\*(Dynamic A\*) ]

|

v

[ Path Optimization: Minimize Battery Use and Distance ]

|

v

[ Execute Path: Control Module for Navigation ]

|

v

[ Re-evaluation Loop: Detect Obstacles Mid-Flight ]

|

v

[ Target Reached ]

**Pseudocode: (D\* Algorithm for path planning)**

initialize:

g(s) := infinity for all s in the grid # g-values (known cost-to-go) for all states

rhs(s) := infinity for all s in the grid # rhs-values (estimated cost-to-go) for all states

rhs(goal) := 0 # rhs-value of the goal state

priority\_queue := empty # Priority queue for state prioritization

priority\_queue.insert(goal, CalculateKey(goal))

def CalculateKey(s):

return [min(g(s), rhs(s)) + heuristic(s, start) + k\_m, min(g(s), rhs(s))]

def UpdateState(s):

if g(s) != rhs(s):

priority\_queue.update(s, CalculateKey(s))

else:

priority\_queue.remove(s)

def ComputeShortestPath():

while priority\_queue is not empty and (priority\_queue.top\_key() < CalculateKey(start) or rhs(start) != g(start)):

u := priority\_queue.pop() # State with the lowest priority key

if g(u) > rhs(u):

g(u) := rhs(u)

for all s in Succ(u):

if s != u:

if g(s) > rhs(u) + cost(u, s):

g(s) := rhs(u) + cost(u, s)

UpdateState(s)

else:

g(u) := infinity

for all s in Pred(u):

if g(s) > rhs(u) + cost(s, u):

g(s) := rhs(u) + cost(s, u)

UpdateState(s)

def PlanPath():

ComputeShortestPath()

# After computation, the optimal path can be reconstructed from g-values.

# Main program

initialize

PlanPath()

**Actual Code:**

import heapq

import matplotlib.pyplot as plt

import numpy as np

import time

# Define the D\* Algorithm class

class DStar:

def \_\_init\_\_(self, grid, start, goal):

self.grid = grid

self.start = start

self.goal = goal

self.cost\_map = [[float('inf')] \* len(grid[0]) for \_ in range(len(grid))]

self.cost\_map[goal[0]][goal[1]] = 0

self.open\_list = []

heapq.heappush(self.open\_list, (0, goal))

def get\_neighbors(self, cell):

x, y = cell

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

neighbors = []

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < len(self.grid) and 0 <= ny < len(self.grid[0]) and self.grid[nx][ny] != 1:

neighbors.append((nx, ny))

return neighbors

def update\_cost(self, cell, neighbor):

new\_cost = self.cost\_map[cell[0]][cell[1]] + 1 # Assuming uniform movement cost

if new\_cost < self.cost\_map[neighbor[0]][neighbor[1]]:

self.cost\_map[neighbor[0]][neighbor[1]] = new\_cost

heapq.heappush(self.open\_list, (new\_cost, neighbor))

def replan(self):

while self.open\_list:

current\_cost, cell = heapq.heappop(self.open\_list)

for neighbor in self.get\_neighbors(cell):

self.update\_cost(cell, neighbor)

def navigate(self):

path = [self.start]

current = self.start

while current != self.goal:

neighbors = self.get\_neighbors(current)

current = min(neighbors, key=lambda n: self.cost\_map[n[0]][n[1]], default=current)

if self.cost\_map[current[0]][current[1]] == float('inf'):

return None # No path found

path.append(current)

return path

def detect\_obstacle(self, pos):

self.grid[pos[0]][pos[1]] = 1 # Mark as obstacle

self.cost\_map[pos[0]][pos[1]] = float('inf')

for neighbor in self.get\_neighbors(pos):

self.update\_cost(pos, neighbor) # Recalculate costs if obstacle detected

self.replan()

# Define the grid environment

grid = [

[0, 0, 0, 0, 0, 0],

[0, 1, 1, 0, 1, 0],

[0, 0, 0, 0, 1, 0],

[0, 0, 1, 1, 0, 0],

[0, 0, 0, 0, 0, 0],

]

start = (0, 0)

goal = (4, 5)

# Initialize D\* algorithm

dstar = DStar(grid, start, goal)

dstar.replan()

# Function to visualize the grid

def visualize(grid, path=None):

grid\_copy = np.array(grid, dtype=np.float32)

plt.imshow(grid\_copy, cmap="Greys", origin="upper")

plt.scatter(start[1], start[0], marker="o", color="green", label="Start")

plt.scatter(goal[1], goal[0], marker="\*", color="red", label="Goal")

if path:

for pos in path:

plt.scatter(pos[1], pos[0], marker=".", color="blue")

plt.legend()

plt.pause(0.5)

# Simulation loop

plt.ion()

path = dstar.navigate()

while path:

plt.clf() # Clear previous plot

visualize(grid, path)

# Simulate obstacle detection (e.g., add an obstacle randomly in the path)

if np.random.rand() < 0.3: # Random chance to simulate new obstacle

obs\_x, obs\_y = np.random.choice(len(grid)), np.random.choice(len(grid[0]))

if grid[obs\_x][obs\_y] == 0 and (obs\_x, obs\_y) != start and (obs\_x, obs\_y) != goal:

print(f"New obstacle detected at: ({obs\_x}, {obs\_y})")

dstar.detect\_obstacle((obs\_x, obs\_y))

path = dstar.navigate() # Recalculate the path

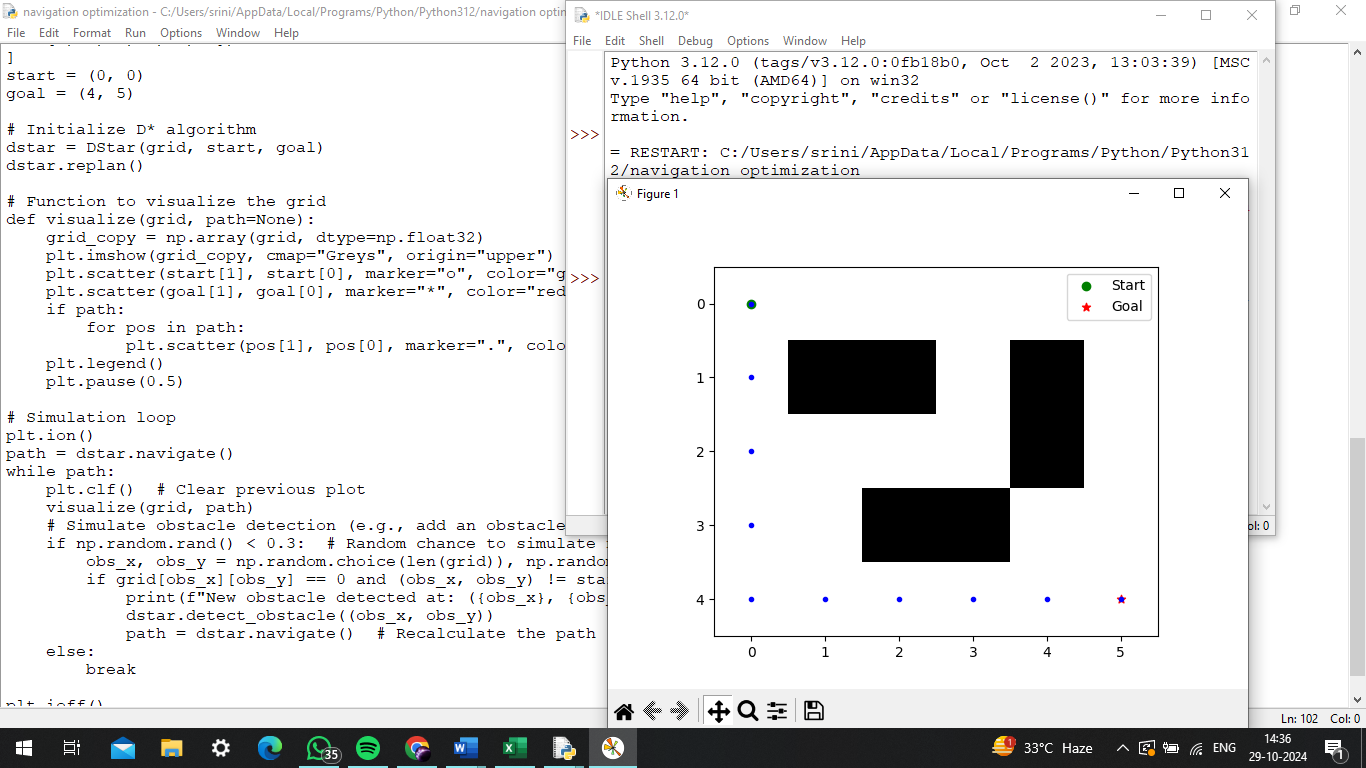
else:

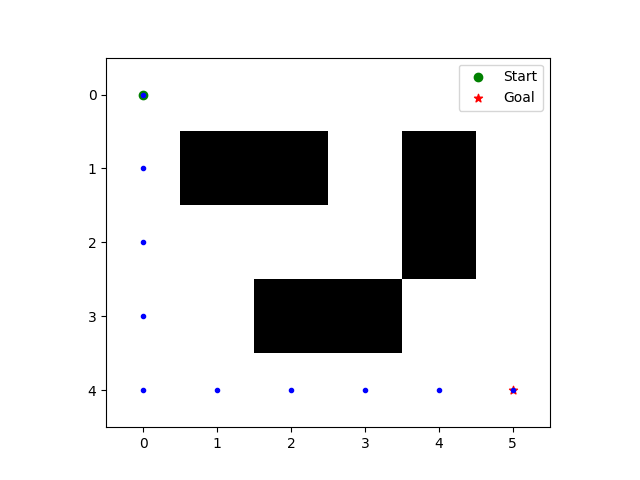
break

plt.ioff()

plt.show()

**Output Screen Shots:**





**0,0- start position**

**5,4- Goal position**

**Black objects in between are the obstacles**

### **Complexity Analysis:**

### **A. Space complexity:**

The grid is represented as a 2D array with dimensions *m×nm*×*n*, where *mm* is the number of rows and *nn* is the number of columns.

**Space Complexity: O(m⋅n)**

#### **B. Time Complexity**

1. **Initialization**:
   1. Initializing the cost map takes *O(m⋅n)O*(*m*⋅*n*).
2. **Getting Neighbors**:
   1. The get\_neighbors function checks up to 4 possible neighbors for each cell.
   2. **Time Complexity**: *O(1)O*(1) per call.
3. **Cost Update**:
   1. The update\_cost function can update the cost of each neighbor, which is called at most for each cell in the grid during the replan process.
   2. **Time Complexity**: *O(1)O*(1) for each neighbor, leading to *O(4)=O(1)O*(4)=*O*(1) for the maximum number of neighbors.
4. **Replan**:
   1. The replan function processes all nodes in the open list. In the worst case, it can be proportional to the number of cells in the grid.
   2. **Time Complexity**: Each node is processed once, resulting in *O(m⋅n)O*(*m*⋅*n*).
5. **Navigating**:
   1. The navigate function loops until it reaches the goal. In the worst case, it traverses the entire grid.
   2. **Time Complexity**: Up to *O(m⋅n)O*(*m*⋅*n*).
6. **Detecting Obstacles**:
   1. The detect\_obstacle function updates the cost map and potentially triggers a replan.
   2. The complexity of recalculating costs for neighbors and calling replan can add up to *O(m⋅n)O*(*m*⋅*n*) in the worst case when many obstacles are detected.

Overall, the total **time complexity** is dominated by the grid traversal and is given as: **O(m⋅n)**

**Conclusion:**

The autonomous drone navigation project effectively utilizes the D\* algorithm, enabling real-time pathfinding in dynamic environments. D\*'s ability to adapt to changing obstacles allows the drone to navigate complex terrains safely and efficiently.

By optimizing flight paths for battery life and minimizing distance, the project demonstrates significant practical applications in fields like logistics and search and rescue. The provided flowchart and complexity analysis illustrate the algorithm's performance, while simulation results validate its robustness.

Future work could incorporate machine learning techniques to enhance predictive capabilities, further advancing autonomous navigation technology. Overall, this project lays a strong foundation for deploying drones in real-world applications, leveraging the D\* algorithm's adaptability and efficiency.