Question 1

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
```

In [ ]:
```
vertices = ['A', "B", "C", "D", "E", "F", 'S', 'T']
graph = defaultdict(dict)
graph['S']['B'] = 18
graph['S']['C'] = 10
graph['S']['A'] = 12
graph['C']['F'] = 15
graph['F']['E'] = 14
graph['B']['F'] = 8
graph['B']['D'] = 21
graph['A']['D'] = 21
graph['A']['E'] = 40
graph['E']['T'] = 16
graph['D']['T'] = 27
T = len(vertices) - 1
value = defaultdict(dict)
policy = defaultdict(dict)
```

In [ ]:
```
# initialize
for i in range(T+1):
    value[i]["T"] = 0

for v in vertices:
    if v == "T":
        continue
    value[T][v] = np.inf
    cost = graph[v].get('T', float('inf'))
    value[T-1][v] = cost
    policy[T-1][v] = "T"

for t in range(T-2, -1, -1):
    for i in vertices:
        best_value = np.inf
        best_policy = None
        if i == "T":
            continue
        for j in vertices:
            cost = graph[i].get(j, float('inf'))
            val = value[t+1].get(j, float('inf'))
            if val + cost < best_value:
                best_value = val + cost
                best_policy = j
        value[t][i] = best_value
        policy[t][i] = best_policy

# reconstruct path
for v in vertices:
    if v == "T":
        continue
    current_node = v
    t = 0
    path = []
    while current_node != "T":
        current_node = policy[t][current_node]
        path.append(current_node)
        t += 1
    print(f"Start from {v}: {path}, cost:{value[0][v]}")
```

Question 2

In [ ]:
```
from shapely.geometry import LineString, Point
import matplotlib.pyplot as plt
import numpy as np
from shapely.geometry import LineString, Point
from mpl_toolkits.mplot3d.art3d import Poly3DCollection


def rotate_line_segment(angle_rad,length=1.0):
    """Rotate a line segment of given length by angle around origin"""
    x1, y1 = -length / 2, 0
    x2, y2 =  length / 2, 0
    R = np.array([[np.cos(angle_rad), -np.sin(angle_rad)],
                  [np.sin(angle_rad),  np.cos(angle_rad)]])
    p1 = R @ np.array([x1, y1])
    p2 = R @ np.array([x2, y2])
    return LineString([tuple(p1), tuple(p2)])

degree = 30
rad = degree * np.pi / 180
robot_line = rotate_line_segment(rad)

# Obstacle
obstacle_center = Point(0, 0)
obstacle_radius = 1

# Minkowski sum
cspace_obstacle = robot_line.buffer(obstacle_radius, cap_style=1)

fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.set_xlim(-5, 5)
ax.set_ylim(-5, 5)

circle = plt.Circle((0, 0), obstacle_radius, color='gray', alpha=0.3, label='Ori
ax.add_patch(circle)
x, y = cspace_obstacle.exterior.xy
ax.fill(x, y, alpha=0.7, fc='red', ec='black', label='C-space Obstacle')

plt.title("C-space Obstacle using Minkowski Sum (theta = 30)")
plt.legend()
plt.grid(True)
plt.savefig('cspace_30.png')
plt.show()
```

In [ ]:
```
# Parameters
obstacle_radius = 1.0
line_length = 1.0
n_theta = 30
```

```
thetas = np.linspace(-np.pi, np.pi, n_theta)

def rotate_line_segment(length, angle_rad):
    """Rotate a line segment of given length by angle around origin"""
    x1, y1 = -length / 2, 0
    x2, y2 =  length / 2, 0
    R = np.array([[np.cos(angle_rad), -np.sin(angle_rad)],
                  [np.sin(angle_rad),  np.cos(angle_rad)]])
    p1 = R @ np.array([x1, y1])
    p2 = R @ np.array([x2, y2])
    return LineString([tuple(p1), tuple(p2)])

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("theta")
ax.set_xlim(-5, 5)
ax.set_ylim(-5, 5)
ax.set_zlim(-np.pi, np.pi)

for theta in thetas:
    # compute line segment under certain rotation
    line = rotate_line_segment(line_length, theta)

    # inflate circle
    cspace_slice = line.buffer(obstacle_radius, cap_style=1)
    x, y = cspace_slice.exterior.xy
    z = np.full_like(x, theta)
    verts = [list(zip(x, y, z))]
    poly = Poly3DCollection(verts, alpha=0.15, facecolor='red', edgecolor='k')
    ax.add_collection3d(poly)
plt.title("C-space Obstacle Volume (x, y, theta)")
plt.tight_layout()
plt.savefig('cspace.png')
plt.show()
```

Question 4

In [ ]:
```
import numpy as np
from collections import defaultdict
import heapq
from pqdict import pqdict
```

In [ ]:
```
vertices = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# graph
edges = defaultdict(dict)
edges[1][2] = 1
edges[1][6] = 1
edges[2][1] = 1
edges[2][5] = 1
edges[2][3] = 1
edges[3][2] = 1
edges[3][4] = 1
edges[4][3] = 1
edges[4][5] = 1
edges[5][2] = 1
```

```
edges[5][4] = 1
edges[5][6] = 1
edges[6][1] = 1
edges[6][5] = 1
edges[6][7] = 1
edges[7][6] = 1
edges[7][8] = 1
edges[8][7] = 1
edges[8][9] = 1
edges[9][8] = 1
edges[9][10] = 1
edges[10][9] = 1

# heuristic
heuristic = {}
heuristic[1] = 5
heuristic[2] = 4
heuristic[3] = 3
heuristic[4] = 2
heuristic[5] = 3
heuristic[6] = 4
heuristic[7] = 3
heuristic[8] = 2
heuristic[9] = 1
heuristic[10] = 0
```

In [ ]:

```
class Node(object):
    def __init__(self, key):
        self.key = key
        self.parent = None
        self.f = float('inf')
        self.g = float('inf')
        self.h = 0.0
        self.is_open = False
        self.is_closed = False
        self.children = []

    def setParent(self, parent):
        self.parent = parent

    def setChildren(self, children):
        self.children = children

    def setG(self, g):
        self.g = g
        self.f = self.g + self.h

    def setHeuristic(self, h):
        self.h = h
        self.f = self.g + self.h

    def getHeuristic(self):
        return self.h

    def isOpen(self):
        return self.is_open

    def isClosed(self):
        return self.is_closed
    def getParent(self):
        return self.parent

    def getG(self):
        return self.g

    def getF(self):
        self.f = self.g + self.h
        return self.f
```

In [ ]:
```
def tie_breaking_precedes(a, b):
    # a, b are (priority, key) tuples
    # Prefer smaller priority first; if equal, prefer smaller key
    return a[0] < b[0] or (a[0] == b[0] and a[1] < b[1])

class RTAA():
    def __init__(self, start, goal, heuristic, edges, vertices, step=4):
        self.start = start
        self.goal = goal
        self.heuristic = heuristic
        self.step = step
        self.edges = edges
        self.vertices = vertices

    def init_nodes(self, start):
        open_heap = pqdict(precedes=tie_breaking_precedes).minpq()
        nodes = {}
        for v in self.vertices:
            node = Node(v)
            node.setHeuristic(self.heuristic[v])
            node.setChildren(list(self.edges[v].keys()))
            if v == start:
                node.setG(0)
                node.is_open = True
                f = node.getF()
                open_heap[v] = (f, v)
            nodes[v] = node
        return open_heap, nodes

    def find_node(self, nodes, key):
        return nodes.get(key)

    def find_optimal(self, open_set):
        if not open_set:
            return None, float('inf')
        return min(open_set.items(), key=lambda item: (item[1], item[0]))

    def update_heuristic(self, closed_list, node_dict, f):
        for v in closed_list:
            node = node_dict[v]
            self.heuristic[v] = f - node.getG()
            #print(f"Node: {v}, f: {f}, g: {node.getG()}")

    def get_path(self, optimal_key, node_dict):
        path = []
        current_node = node_dict[optimal_key]

        while current_node is not None:
            path.append(current_node.key)
            current_node = current_node.getParent()
        return list(reversed(path))

    def a_star(self, start):
        open_heap, node_dict = self.init_nodes(start)
        closed_list = []

        # Reset the nodes for a fresh search
        for key, node in node_dict.items():
            if key != start:
                node.setG(float('inf'))
            node.parent = None
            node.is_open = False
            node.is_closed = False

        start_node = node_dict[start]
        start_node.setG(0)
        start_node.is_open = True

        expanded_count = 0

        while open_heap and expanded_count < self.step:
            # Get the node with the lowest f-score
            item = open_heap.popitem()
            current_key = item[0]
            current_f = item[1][0]

            current_node = node_dict[current_key]
            current_node.is_open = False
            current_node.is_closed = True
            closed_list.append(current_key)

            # Check if goal reached
            if current_key == self.goal:
                return open_heap, closed_list, node_dict, True

            # expand node
            for child_key, edge_cost in self.edges[current_key].items():
                child_node = node_dict[child_key]
                if child_node.is_closed:
                    continue

                tentative_g = current_node.getG() + edge_cost
                if tentative_g < child_node.getG():
                    child_node.setParent(current_node)
                    child_node.setG(tentative_g)

                    # update
                    f = child_node.getF()
                    open_heap[child_key] = (f, child_key)
                    child_node.is_open = True

            expanded_count += 1
        return open_heap, closed_list, node_dict, False


    def print_open_and_heuristic(self, open_set):
        keys = sorted(open_set.keys())
        open_keys_str = "OPEN:     [" + "  ".join(f"{k:2}" for k in keys) + "]"
        f_values_str = "OPEN f:   [" + "  ".join(f"{open_set[k]:2}" for k in key
        print(open_keys_str)
        print(f_values_str)

    def print_heuristic(self):
        keys = sorted(self.heuristic.keys())
        open_keys_str = "i:     [" + "  ".join(f"{k:2}" for k in keys) + "]"
        h_str = "hi:    [" + "  ".join(f"{self.heuristic[k]:2}" for k in keys) +
        print(open_keys_str)
        print(h_str)

    def process_openheap(self, open_heap):
        open_set = {}
        for node, min_dist in open_heap.popitems():
            open_set[node] = min_dist[0]
        return open_set


    def run(self):
        path = [self.start]
        current = self.start
        iteration = 0

        while current != self.goal:
            print("================================================")
            print(f"Iteration {iteration + 1}: Current position = {current}")

            # expand by A* for a limited number of steps
            open_heap, closed_list, node_dict, goal_reached = self.a_star(curren
            open_set = self.process_openheap(open_heap)

            if goal_reached:
                # complete the path to the goal
                remaining_path = self.get_path(self.goal, node_dict)[1:]
                path.extend(remaining_path)
                print(f"Goal reached Final path: {path}")
                return path

            if not open_set:
                print("Failed to find a path. No nodes in open set.")
                return path

            # find the best next node to move to
            optimal_key, optimal_cost = self.find_optimal(open_set)
            if optimal_key is None:
                print("No optimal node found. Path finding failed.")
                return path

            # update heuristics for closed nodes
            self.update_heuristic(closed_list, node_dict, optimal_cost)
            print(f"CLOSED list: {closed_list}")
            self.print_open_and_heuristic(open_set)
            self.print_heuristic()

            # move to the next best node
            next_node = optimal_key
            next_path = self.get_path(next_node, node_dict)
            if len(next_path) > 1:
                move_segment = next_path[1:]
                path.extend(move_segment)
                current = next_node
                print(f"Moving to node {current}, path segment: {move_segment}")
            else:
                print("No valid move found. Path finding failed.")
                return path
            iteration += 1
        return path


print("Starting RTAA* algorithm")
rtaa = RTAA(start=1, goal=10, heuristic=heuristic, edges=edges, vertices=vertice
path = rtaa.run()
```

Question 5

In [ ]:

```
vertices = [1,2,3,4,5,6,7]
edges = defaultdict(dict)
edges[1][3] = 5
edges[1][5] = 2
edges[1][6] = 5
edges[2][5] = 9
edges[2][6] = 1
edges[3][4] = 1
edges[3][5] = 1
edges[5][6] = 4
edges[5][3] = 1
edges[6][7] = 5
edges[6][1] = 5
edges[7][4] = 5


# heuristic
heuristic = {}
heuristic[1] = 1
heuristic[2] = 10
heuristic[3] = 3
heuristic[4] = 0
heuristic[5] = 2
heuristic[6] = 7
heuristic[7] = 5

start = 2
goal = 4
```

In [ ]:

```
class A_Star():
    def __init__(self, start, goal, heuristic, edges, vertices, step=4, epsilon=
        self.start = start
        self.goal = goal
        self.heuristic = heuristic
        self.step = step
        self.edges = edges
        self.vertices = vertices
        self.eps = epsilon

    def initialize(self, start):
        open_heap = pqdict(precedes=tie_breaking_precedes).minpq()
        nodes = {}
        for v in self.vertices:
            node = Node(v)
            node.setHeuristic(self.heuristic[v] * self.eps)
            node.setChildren(list(self.edges[v].keys()))
            if v == start:
                node.setG(0)
                node.is_open = True
                f = node.getF()
                open_heap[v] = (f, v)
            nodes[v] = node
        return open_heap, nodes

    def find_node(self, nodes, key):
        return nodes.get(key)

    def run(self):
        self.open_heap, self.node_dict = self.initialize(self.start)
        self.closed_list = []
        expanded_count = 0
        self.inconsist = []

        while self.open_heap and expanded_count < self.step:
            print("================================================")
            print(f"Iteration {expanded_count}")
            # Get the node with the lowest f-score
            item = self.open_heap.popitem()
            current_key = item[0]
            current_f = item[1][0]
            current_node = self.node_dict[current_key]
            current_node.is_open = False
            current_node.is_closed = True
            self.closed_list.append(current_key)
            current_node.setV(current_node.getG())
            print(f"Node exiting OPEN: {current_key}")

            # Check if goal reached
            if current_key == self.goal:
                return self.open_heap, self.closed_list, self.node_dict, self.in

            # expand node
            for child_key, edge_cost in self.edges[current_key].items():
                child_node = self.node_dict[child_key]

                tentative_g = current_node.getG() + edge_cost
                if tentative_g < child_node.getG():
                    child_node.setParent(current_node)
                    child_node.setG(tentative_g)

                    # update
                    if child_key in self.closed_list:
                        self.inconsist.append(child_key)
                    else:
                        f = child_node.getF()
                        self.open_heap[child_key] = (f, child_key)
                        child_node.is_open = True
            print(f"OPEN: {list(self.open_heap.keys())}")
            for key, node in self.node_dict.items():
                print(f"Node {key}: {node.getG()}")

            expanded_count += 1
        return self.open_heap, self.closed_list, self.node_dict, self.inconsist,
```

In [ ]:
```
a_start = A_Star(start, goal, heuristic.copy(), edges, vertices, step=5, epsilon
open_heap, close_list, node_dict, inconsist, is_goal = a_start.run()
```