```python
from os import close
import numpy as np
from heapq import heappop, heappush
import matplotlib.pyplot as plt

class Node(object):
    """
    Class Node: a data structure that help process calculation of AStar
    """
    def __init__(self, pose):
        """
        param self.pose: [x, y] index position of node
        """
        self.pose = np.array(pose)
        self.x = pose[0]
        self.y = pose[1]
        self.g_value = 0
        self.h_value = 0
        self.f_value = 0
        self.parent = None

    def __lt__(self, other):
        """
        less than function for heap comparison
        """
        return self.f_value < other.f_value

    def __eq__(self, other):
        return (self.pose == other.pose).all()

class AStar(object):
    def __init__(self, map_path):
        self.map_path = map_path
        self.map = self.load_map(self.map_path).astype(int)
        print(self.map)
        self.resolution = 0.05
        self.y_dim = self.map.shape[0]
        self.x_dim =self.map.shape[1]
```

```python
        print(f'map size ({self.x_dim}, {self.y_dim})')

    def load_map(self, path):
        return np.load(path)

    def reset_map(self):
        self.map = self.load_map(self.map_path)

    def heuristic(self, current, goal):
        """
        TODO:
        Euclidean distance
        """
        # Euclidean distance calculation
        dx = current.x - goal.x
        dy = current.y - goal.y
        Euclidean_distance = np.sqrt(dx**2 + dy**2)

        return Euclidean_distance

    def get_successor(self, node):
        """
        :param node: A Node data structure
        :return: a list of Nodes containing successors of current Node
        """
        successor_list = []
        x,y = node.pose  # Get x, y coordinates of the current node
        pose_list = [[x+1, y+1], [x, y+1], [x-1, y+1], [x-1, y],
                        [x-1, y-1], [x, y-1], [x+1, y-1], [x+1, y]]  # Pose list contains 8 neighbors of the current node

        for pose_ in pose_list:
            x_, y_ = pose_
            if 0 <= x_ < self.y_dim and 0 <= y_ < self.x_dim and self.map[x_, y_] == 0: # Eliminate nodes that are out of bound, and nodes that are
```

```python
obstacles
                self.map[x_, y_] = -1
                successor_list.append(Node(pose_))

        return successor_list

    def calculate_path(self, node):
        """
        :param node: A Node data structure
        :return: a list with shape (n, 2) containing n path point
        """
        path_ind = []
        path_ind.append(node.pose.tolist())
        current = node
        while current.parent:
            current = current.parent
            path_ind.append(current.pose.tolist())
        path_ind.reverse()
        print(f'path length {len(path_ind)}')
        path = list(path_ind)

        return path

    def plan(self, start_ind, goal_ind):
        """
        TODO:
        Fill in the missing lines in the plan function
        @param start_ind : [x, y] represents coordinates in webots world
        @param goal_ind : [x, y] represents coordinates in webots world
        @return path : a list with shape (n, 2) containing n path point
        """

        # initialize start node and goal node class
        start_node = Node(start_ind)
        goal_node = Node(goal_ind)
```

```
        """
        TODO:
        calculate h and f value of start_node
        (1) h can be computed by calling the heuristic method
        (2) f = g + h
        """
        # calculate h and f value of start_node
        start_node.g_value = 0
        start_node.h_value = self.heuristic(start_node, goal_node)
        start_node.f_value = start_node.g_value + start_node.h_value

        """
        END TODO
        """

        # Reset map
        self.reset_map()

        # Initially, only the start node is known.
        # This is usually implemented as a min-heap or priority queue rather than a hash-set.
        # Please refer to https://docs.python.org/3/library/heapq.html for more details about heap data structure
        open_list = []
        closed_list = np.array([])
        heappush(open_list, start_node)

        # while open_list is not empty
        while len(open_list):

            """
            TODO:
            get the current node and add it to the closed list
            """
            # Current is the node in open_list that has the lowest f value
```

```
            # This operation can occur in O(1) time if open_list is a min-heap or a priority queue

            # get and add current node to the closed list
            current = heappop(open_list)
            """
            END TODO
            """
            closed_list = np.append(closed_list, current)

            self.map[current.x, current.y] = -1

            # if current is goal_node: calculate the path by passing through the current node
            # exit the loop by returning the path
            if current == goal_node:
                print('reach goal')
                return self.calculate_path(current)

            for successor in self.get_successor(current):
                """
                TODO:
                1. pass current node as parent of successor node
                2. calculate g, h, and f value of successor node
                    (1) d(current, successor) is the weight of the edge from current to successor
                    (2) g(successor) = g(current) + d(current, successor)
                    (3) h(successor) can be computed by calling the heuristic method
                    (4) f(successor) = g(successor) + h(successor)
                """
                successor.parent = current
```

```
                successor.g_value = current.g_value + 1
                successor.h_value = self.heuristic(successor, goal_node)
                successor.f_value = successor.g_value + successor.h_value

                if tuple(successor.pose) in closed_list:
                    continue

                in_open_list = any(successor == node for node in open_list)

                if not in_open_list or successor.g_value < current.g_value:
                    if in_open_list:
                        open_list.remove(successor)
                    heappush(open_list, successor)
                """
                END TODO
                """

        # If the loop is exited without return any path
        # Path is not found
        print('path not found')
        return None

    def run(self, cost_map, start_ind, goal_ind):
        if cost_map[start_ind[0], start_ind[1]] == 0 and cost_map[goal_ind[0], goal_ind[1]] == 0:
            return self.plan(start_ind, goal_ind)

        else:
            print('already occupied')

```