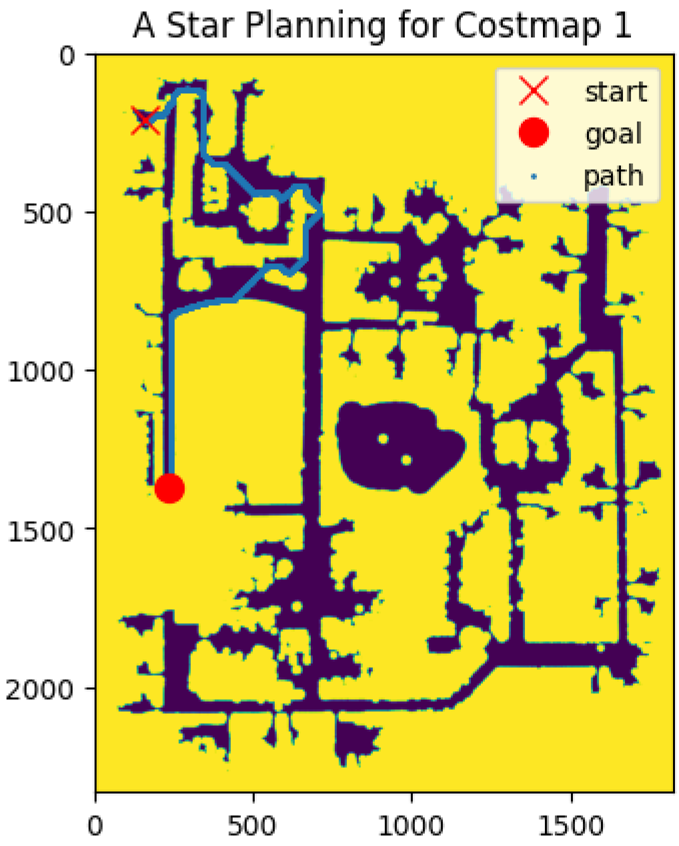
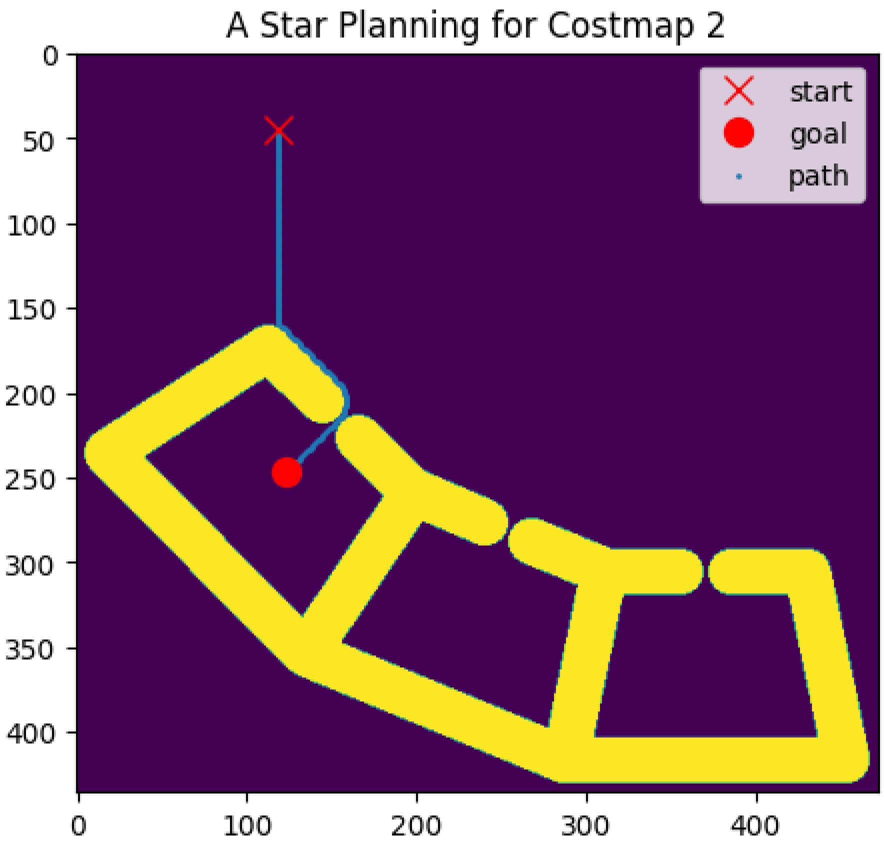
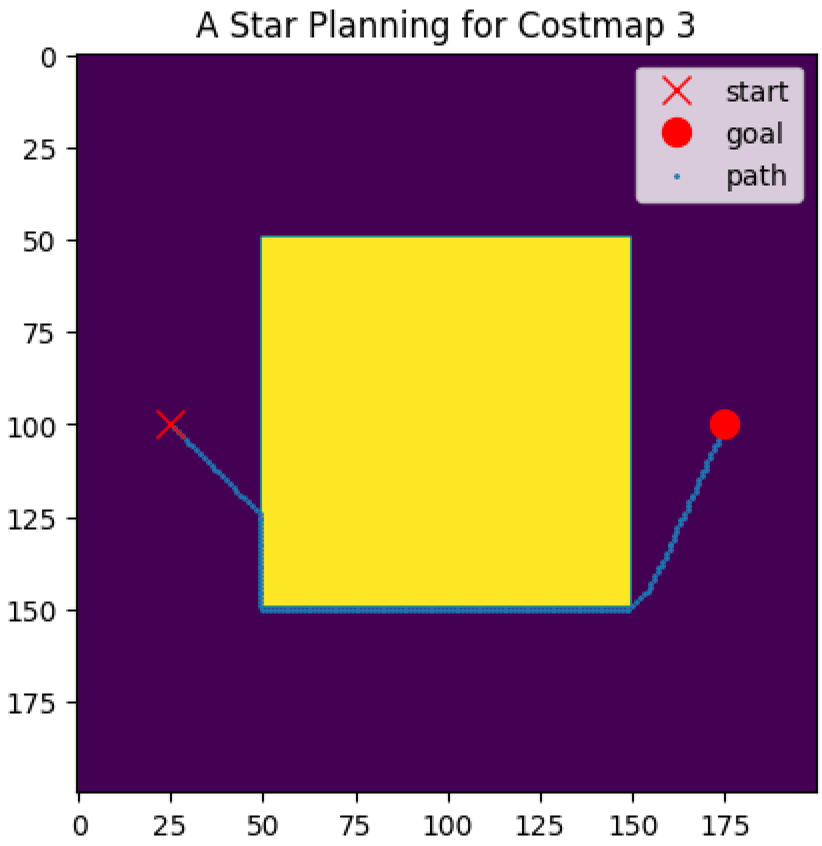
In [2]:

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

class Node(object):
    def __init__(self, pose):
        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):
        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]
        print(f'map size ({self.x_dim}, {self.y_dim})')

    def load_map(self, path):
        #return np.load(path)
        return np.genfromtxt(path, delimiter = ",")

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

    def heuristic(self, current_node, next_node):
        """
        TODO:
        Euclidean distance
        """
        h=np.sqrt((current_node.x-next_node.x)**2+(current_node.y-next_node.y)**2)
        return h

    def get_successor(self, node):
        successor_list = []
        x,y = node.pose
        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]]

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

        return successor_list

    def calculate_path(self, node):
        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 #staring point & end point in map
        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
        """
        #start_node.h_value = None
        #start_node.f_value = None
        start_node.h_value=self.heuristic(start_node, goal_node)
        start_node.g_value=0
        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
        #flist=[]
        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
            current = heappop(open_list)
            #for value in open_list:
             #   flist.append(value.f_value)
            #current = open_list[open_list.index(min(flist))]
            #current = open_list[open_list.index(min(open_list))]
            #current = open_list[0]
            #current_index=0
            #for index, item in enumerate(open_list):
              #  if item.f_value <current.f_value:
               #     current=item
                #    current_index=index

            """
            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+self.heuristic(current, successor) #?? #g+d
                successor.h_value=self.heuristic(successor, goal_node)
                successor.f_value = successor.h_value+successor.g_value
                """
                END TODO
                """
                heappush(open_list, successor)

        # 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):
        '''
        Change the original main function to a method "run" inside the AStar class
        '''

        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')


def visualize_path(cost_map, path, title):
    x = [item[0] for item in path]
    x = x[1:-1]
    y = [item[1] for item in path]
    y = y[1:-1]

    plt.imshow(np.transpose(cost_map))
    plt.plot(path[0][0], path[0][1], 'x', color = 'r', label = 'start', markersize = 10)
    plt.plot(path[-1][0], path[-1][1], 'o', color = 'r', label = 'goal', markersize = 10)
    plt.scatter(x, y, label = 'path', s = 1)
    plt.legend()
    plt.title(title)
    plt.show()

if __name__ == "__main__":
    costmap1 = np.genfromtxt('map1.csv', delimiter = ',')
    costmap2 = np.genfromtxt('map2.csv', delimiter = ',')
    costmap3 = np.genfromtxt('map3.csv', delimiter = ',')

    # plt.imshow(np.transpose(costmap3))
    # plt.show()

    start_ind1 = [159, 208]
    goal_ind1 = [231, 1369]
    start_ind2 = [119, 45]
    goal_ind2 = [123, 247]
    start_ind3 = [25, 100]
    goal_ind3 = [175, 100]

    Planner1 = AStar('map1.csv')
    Planner2 = AStar('map2.csv')
    Planner3 = AStar('map3.csv')

    path_ind1 = Planner1.run(costmap1, start_ind1, goal_ind1)
    path_ind2 = Planner2.run(costmap2, start_ind2, goal_ind2)
    path_ind3 = Planner3.run(costmap3, start_ind3, goal_ind3)

    visualize_path(costmap1, path_ind1, 'A Star Planning for Costmap 1')
    visualize_path(costmap2, path_ind2, 'A Star Planning for Costmap 2')
    visualize_path(costmap3, path_ind3, 'A Star Planning for Costmap 3')
```

```
map size (2332, 1825)
map size (436, 473)
map size (200, 200)
reach goal
path length 1904
reach goal
path length 203
reach goal
path length 200
```


A Star Planning for Costmap 1
0
500
1000
1500
2000
0
500
1000
1500
start
goal
path


A Star Planning for Costmap 2
0
50
100
150
200
250
300
350
400
0
100
200
300
400
start
goal
path


A Star Planning for Costmap 3
0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175
start
goal
path