Iby and Aladar Fleischman
Faculty of Engineering
Tel Aviv University
הפקולטה להנדסה
ע"ש איבי ואלדר פליישמן
אוניברסיטת תל-אביב

מעבדה ברובוטיקה ובקרה של מערכות

Robotics & systems control Laboratory

0542-4624

Final project – A* path planning simulation.

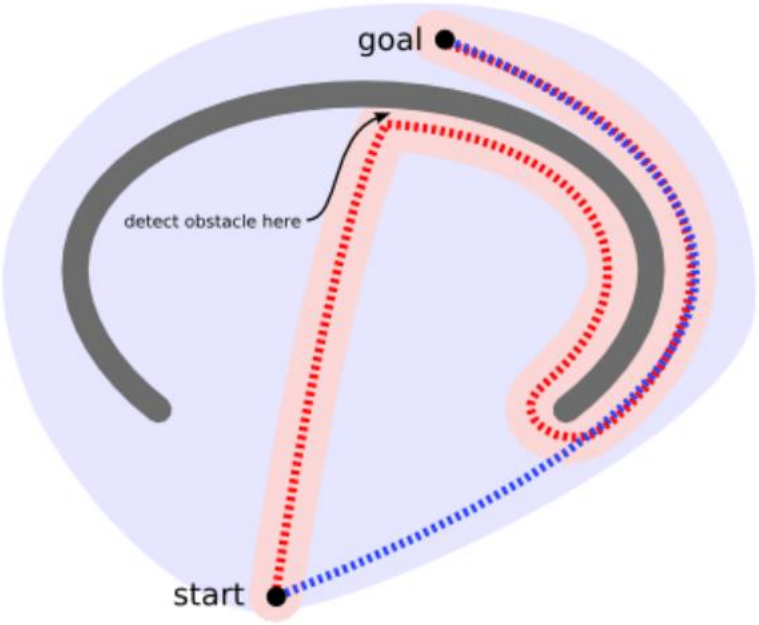
Group 6

מס'	שמות הסטודנטים ומספר ת"ז
1	תומר הראל 313135741
2	עידן שוורץ 316122092
3	אורון בנימין 208306274

תאריך הגשה : 14/08/2023

מטרת הפרויקט

במסגרת פרויקט זה התבקשנו לכתוב קוד לתכנון מסלול, אשר בהינתן מרחב עבודה קיים וממספר מכשולים סופי, האלגוריתם יידע לנווט את הרכב הרובוטי מנקודת התחלה לנקודת היעד תוך התחמקות מהמכשולים הקיימים ובזמן הקצר ביותר. האלגוריתם אותו מימשנו הינו אלגוריתם לתכנון מסלול **A***.



תפקידי החברים בקבוצה:

- אורון בנימין – כתיבת האלגוריתם ויצירת סימולציה נוספת.
- תומר הראל – אינטגרציה בין האלגוריתם שנכתב לבין האובייקטים שבקלאס **wheeled car**.
- עידן שוורץ – דיבאגר ראשי, עזרה בחיפוש אלגוריתמים נוספים ובדיקת יעילותם.

כל חברי הקבוצה כתבו את הדו"ח הסופי יחדיו וכלל העבודה הייתה משותפת.

האלגוריתם שמומש - Path planning A* Algorithm

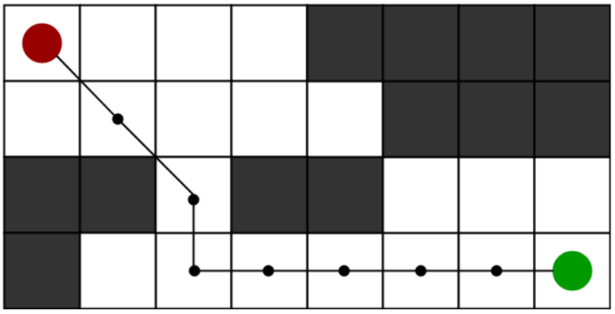
בחרנו להשתמש באלגוריתם מסוג A*, נסביר את עקרון פעולת האלגוריתם ולאחר מכן את הסיבות לבחירתו. אלגוריתם A* הינו אחד האלגוריתמים הנפוצים ביותר לתכנון מסלול השימושי בעיקר בתחומי תכנון התנועה ובעולם הרובוטיקה. אלגוריתם זה נועד לחקור ולבנות ביעילות את הנתיב הקצר ביותר מנקודת התחלה לנקודת יעד בסביבה נתונה תוך התחמקות ממכשולים אפשריים, ובזמן הקצר ביותר. להלן הסבר שלב אחר שלב של אלגוריתם זה:

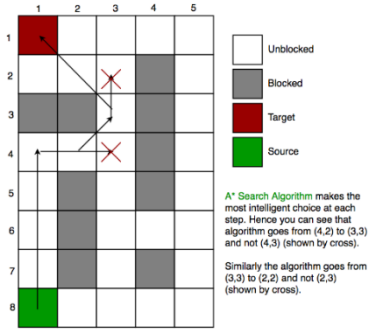
- **אתחול**: הגדרת רשימה פתוחה שלמעשה מאתחלת צמתים במרחב שיש לחקור. הצבת הצומת ההתחלתי ברשימה הפתוחה עם עלות התחלתית של 0 ואתחול רשימה ריקה של הצמתים שנבדקו.
- **לולאה מרכזית:** הלולאה המרכזית מתרחשת כל עוד הרשימה הפתוחה שאתחלנו אינה ריקה. כל עוד זה מתקיים, נבחר את הצומת בעלת הערך הנמוך ביותר מהרשימה, צומת זה נקרא הצומת הנוכחי. במידה והצומת הנוכחי הוא צומת היעד, אזי התקבל מסלול שהוא למעשה כל הנקודות שאוחסנו בצומת זה. אחרת, נעביר את הצומת ברשימה הפתוחה לרשימה הסגורה של צמתים שנחקרו ואת ערכם אנו יודעים. לאחר מכן, נרחיב את הצומת הנוכחי על ידי חקירה של הצמתים השכנים, כך שעבור כל שכן - אם השכן כבר נמצא ברשימה הסגורה, נדלג עליו. אחרת, נחשב את העלות להגיע מהצומת הנוכחי לצומת השכן. אם הערך של הצומת השכן קטנה מהערך של הצומת הנוכחי, נעדכן ערך זה ונוסיף אותו לרשימה הפתוחה. במידה והרשימה הפתוחה בסיום הלולאה עדיין ריקה, אזי צומת היעד לא הושגה ולא יימצא נתיב אפשרי להגיע מנקודת ההתחלה לנקודת היעד. הפונקציה בה באלגוריתם משתמש על מנת להעריך ערך של צומת מסוים אל היעד נקראת פונקציית יוריסטיקה, שלמעשה היא זאת שמנחה את רשת החיפוש על ידי העדפת צמתים שנראים מבטיחים יותר להגעה אל היעד בצורה מהירה יותר. איכות פונקציה זו משפיע בצורה משמעותית על יעילות האלגוריתם. בדרך כלל פונקציה זו מחשבת את הערך של צומת בעזרת מרחק אוקלידי או שיטת חישוב מרחק מנהטן בין הצמתים. להלן פונקציית יוריסטיקה בשיטת האוקלידית:

```
h = sqrt ( (current_cell.x - goal.x)2 +
           (current_cell.y - goal.y)2 )
```

ובשיטת מנהטן:

```
h = abs (current_cell.x - goal.x) +
    abs (current_cell.y - goal.y)
```





להלן פסאודו-קוד קצר המתאר את ההסבר שכתבנו לעיל בצורה ברורה:

```
1  Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
2  while the OPEN list is not empty {
3    Take from the open list the node node_current with the lowest
4         f(node_current) = g(node_current) + h(node_current)
5    if node_current is node_goal we have found the solution; break
6    Generate each state node_successor that come after node_current
7    for each node_successor of node_current {
8      Set successor_current_cost = g(node_current) + w(node_current, node_successor)
9      if node_successor is in the OPEN list {
10       if g(node_successor) ≤ successor_current_cost continue (to line 20)
11     } else if node_successor is in the CLOSED list {
12       if g(node_successor) ≤ successor_current_cost continue (to line 20)
13       Move node_successor from the CLOSED list to the OPEN list
14     } else {
15       Add node_successor to the OPEN list
16       Set h(node_successor) to be the heuristic distance to node_goal
17     }
18     Set g(node_successor) = successor_current_cost
19     Set the parent of node_successor to node_current
20   }
21   Add node_current to the CLOSED list
22 }
23 if(node_current != node_goal) exit with error (the OPEN list is empty)
```

סקירת הקוד:

על מנת שנוכל לבצע את המשימה קיבלנו מהמדריך קובץ פייתון המגדיר class של הרובוט הקיים, הכולל את נקודת ההתחלה, נקודת היעד, וארבעת המכשולים. על כן יכלנו למשוך מקוד זה את ה-state הקיים, כלומר את נקודת ההתחלה והסיום של הרובוט, נתוניו הגיאוטריים, גודל השגיאה הרצוי בהגעה אל היעד והמכשולים הנתונים. בנוסף, ישנן פונקציות נוספות בקוד אשר בדקו התנגשות במכשול, ובעזרת הפונקציה הראשית אשר מקבלת בסוף את הנתיב הסופי מהאלגוריתם ומריצה סימולציה המחשבת את הזמן להגעה אל היעד בנתיב היעודי ובנוסף גם את מספר ההתנגשויות במכשול. כמו כן, התבקשנו להוסיף שני מכשולים נוספים, המהווים קירות שמחברים בין מכשול 1 ל-2, ובין מכשול 3 ל-4, תוך כדי שהאלגוריתם שלנו אמור להתחשב במכשולים אלו בשלב יצירת הנתיב האופטימלי להגעה אל היעד. זוהי למעשה אחת הסיבות העיקריות שבחרנו להשתמש באלגוריתם זה, משום שהוא יודע להתמודד בצורה איכותית עם מכושלים רציפים דמויי קירות או קווים שתוחמים את המרחב. המשחק העיקרי באלגוריתם היה ליצור מצד אחד חיפוש רשתי (Grid) אשר יבטיח מסלול אופטימלי אל היעד אל מול זמן חישוב וגודל צעד לא גדול מדי על מנת לא לפספס נקודות במסלול, ובנוסף שלא יתנגש במכשול קיים.

להלן תיאור הפונקציות העיקריות בקוד:

1. ייבוא ומשיכה של class wheeled car:

```
import math
import numpy as np
import time
from WheeledCar import WheeledCar
import matplotlib.pyplot as plt

show_animation = True

p_g = np.array([0.40073523, 0.70651546])# Set <x_g, y_g> goal position
x_r = np.array([2, 2.2, np.deg2rad(110)])# Set initial pose of the robot: <x_r, y_r, phi_r>
O = np.array([[2, 0.8], [1.2, 1.9], [0.6, 1.6], [0.8, 0.3]])# Set positions of obstacles [p_1, p
W = WheeledCar(x = x_r, goal = p_g, Obs = O)# Initiate simulation with x0, goak and Obs
```

2. בניית פונקציה המייצרת מעגלים המדמים את המכשולים לאלגוריתם:

```
def generate_circle_points(center_x, center_y, radius, num_points):
    circle_points = []
    safety_co = 0.7
    for i in range(num_points):
        angle = 2 * math.pi * i / num_points
        x = center_x + radius * math.cos(angle)
        y = center_y + radius * math.sin(angle)
        circle_points.append((x, y))
    return circle_points
```

```python
def bresenham_line(x0, y0, x1, y1, num_points):
    points = []
    dx = x1 - x0
    dy = y1 - y0
    increment = 1.0 / num_points

    for i in range(num_points + 1):
        x = x0 + dx * increment * i
        y = y0 + dy * increment * i
        points.append((x, y))

    return points
```

4. האלגוריתם עצמו – בניית האלגוריתם והפונקציות בו (יופיע בנספחים).
5. פונקציית main- הפונקציה הראשית שמאתחלת את class של האלגוריתם, מגדירה את המכשולים המיועדים ובסופו של דבר מחזירה הן סימולציה בפני עצמה שמראה את אלגוריתם החיפוש והנתיב האופטימלי, ובנוסף רשימה המכילה את הנקודות של הנתיב האופטימלי.

נתיב זה הוא האינפוט לפונקציית run של קובץ wheeled car. לאחר שנתיב זה מוכנס לפונקציה זו, מתבצעת הסימולציה המקורית המחשבת את הזמן להגעה אל היעד.
הערה: בסימולציה המקורית איננו רואים את המכשולים הנוספים (הקווים המחברים בין המכשולים 1 ו-2 ובין 3 ו-4, אך כמובן שהם מוגדרים כמכשולים והנתיב מתחשב בהם. (אם היינו רוצים לצייר אותם גם בסימולציה המקורית היינו צריכים לשנות את הפונקציה בקלאס (לא אלגנטי).

נספחים:

1. קוד פיתון A* שכתבנו על מנת לממש את הפתרון.

```
import math
import numpy as np
import time
from WheeledCar import WheeledCar
import matplotlib.pyplot as plt

show_animation = True

p_g = np.array([0.40073523, 0.70651546]) # Set <x_g, y_g> goal position
x_r = np.array([2, 2.2 , np.deg2rad(110)]) # Set initial pose of the robot: <x_r, y_r, phi_r>
O = np.array([[2, 0.8], [1.2, 1.9], [0.6, 1.6], [0.8, 0.3]]) # Set positions of obstacles [p_1, p_2, p_3, p_4]
W = WheeledCar(x = x_r, goal = p_g, Obs = O) # Initiate simulation with x0, goak and Obs

# # Get status of car - use these parameters in your planning
x_r, p_g, p_1, p_2, p_3, p_4 = W.field_status()

def generate_circle_points(center_x, center_y, radius, num_points):
    circle_points = []
    safety_co = 0.7
    for i in range(num_points):
        angle = 2 * math.pi * i / num_points
        x = center_x + radius * math.cos(angle)
        y = center_y + radius * math.sin(angle)
        circle_points.append((x, y))
    return circle_points


def bresenham_line(x0, y0, x1, y1, num_points):
    points = []
    dx = x1 - x0
    dy = y1 - y0
    increment = 1.0 / num_points

    for i in range(num_points + 1):
        x = x0 + dx * increment * i
        y = y0 + dy * increment * i
        points.append((x, y))

    return points
# ... (rest of your code)


class AStarPlanner:

    def __init__(self, ox, oy, resolution, rr):

        self.resolution = resolution
        self.rr = rr
        self.min_x, self.min_y = 0, 0
        self.max_x, self.max_y = 0, 0
        self.obstacle_map = None
        self.x_width, self.y_width = 0, 0
        self.motion = self.get_motion_model()
        self.calc_obstacle_map(ox, oy)

    class Node:
        def __init__(self, x, y, cost, parent_index):
            self.x = x  # index of grid
            self.y = y  # index of grid
            self.cost = cost
```

```
        self.parent_index = parent_index

    def __str__(self):
        return str(self.x) + ",," + str(self.y) + ",," + str(
            self.cost) + ",," + str(self.parent_index)

def planning(self, sx, sy, gx, gy):

    start_node = self.Node(self.calc_xy_index(sx, self.min_x),
                           self.calc_xy_index(sy, self.min_y), 0.0, -1)
    goal_node = self.Node(self.calc_xy_index(gx, self.min_x),
                          self.calc_xy_index(gy, self.min_y), 0.0, -1)

    open_set, closed_set = dict(), dict()
    open_set[self.calc_grid_index(start_node)] = start_node

    while True:
        if len(open_set) == 0:
            print("Open set is empty..")
            break

        c_id = min(
            open_set,
            key=lambda o: open_set[o].cost + self.calc_heuristic(goal_node,
                                                                 open_set[
                                                                     o]))
        current = open_set[c_id]

        # show graph
        if show_animation:  # pragma: no cover
            plt.plot(self.calc_grid_position(current.x, self.min_x),
                     self.calc_grid_position(current.y, self.min_y), "xc")
            # for stopping simulation with the esc key.
            plt.gcf().canvas.mpl_connect('key_release_event',
                                         lambda event: [exit(
                                             0) if event.key == 'escape' else None])
            if len(closed_set.keys()) % 10 == 0:
                plt.pause(0.001)

        if current.x == goal_node.x and current.y == goal_node.y:
            print("Find goal")
            goal_node.parent_index = current.parent_index
            goal_node.cost = current.cost
            break

        # Remove the item from the open set
        del open_set[c_id]

        # Add it to the closed set
        closed_set[c_id] = current

        # expand_grid search grid based on motion model
        for i, _ in enumerate(self.motion):
            node = self.Node(current.x + self.motion[i][0],
                             current.y + self.motion[i][1],
                             current.cost + self.motion[i][2], c_id)
            n_id = self.calc_grid_index(node)

            # If the node is not safe, do nothing
            if not self.verify_node(node):
                continue

            if n_id in closed_set:
                continue
```

```
            if n_id not in open_set:
                open_set[n_id] = node  # discovered a new node
            else:
                if open_set[n_id].cost > node.cost:
                    # This path is the best until now. record it
                    open_set[n_id] = node

    rx, ry = self.calc_final_path(goal_node, closed_set)

    return rx, ry

def calc_final_path(self, goal_node, closed_set):
    # generate final course
    rx, ry = [self.calc_grid_position(goal_node.x, self.min_x)], [
        self.calc_grid_position(goal_node.y, self.min_y)]
    parent_index = goal_node.parent_index
    while parent_index != -1:
        n = closed_set[parent_index]
        rx.append(self.calc_grid_position(n.x, self.min_x))
        ry.append(self.calc_grid_position(n.y, self.min_y))
        parent_index = n.parent_index

    return rx, ry

@staticmethod
def calc_heuristic(n1, n2):
    w = 1.0  # weight of heuristic
    d = w * math.hypot(n1.x - n2.x, n1.y - n2.y)
    return d

def calc_grid_position(self, index, min_position):

    pos = index * self.resolution + min_position
    return pos

def calc_xy_index(self, position, min_pos):
    return round((position - min_pos) / self.resolution)

def calc_grid_index(self, node):
    return (node.y - self.min_y) * self.x_width + (node.x - self.min_x)

def verify_node(self, node):
    px = self.calc_grid_position(node.x, self.min_x)
    py = self.calc_grid_position(node.y, self.min_y)

    if px < self.min_x:
        return False
    elif py < self.min_y:
        return False
    elif px >= self.max_x:
        return False
    elif py >= self.max_y:
        return False

    # collision check
    if self.obstacle_map[node.x][node.y]:
        return False

    return True

def calc_obstacle_map(self, ox, oy):

    self.min_x = round(min(ox))
    self.min_y = round(min(oy))
    self.max_x = round(max(ox))
```

```
        self.max_y = round(max(oy))
        print("min_x:", self.min_x)
        print("min_y:", self.min_y)
        print("max_x:", self.max_x)
        print("max_y:", self.max_y)

        self.x_width = round((self.max_x - self.min_x) / self.resolution )
        self.y_width = round((self.max_y - self.min_y) / self.resolution )
        print("x_width:", self.x_width)
        print("y_width:", self.y_width)

        # obstacle map generation
        self.obstacle_map = [[False for _ in range(self.y_width)]
                             for _ in range(self.x_width)]
        for ix in range(self.x_width):
            x = self.calc_grid_position(ix, self.min_x)
            for iy in range(self.y_width):
                y = self.calc_grid_position(iy, self.min_y)
                for iox, ioy in zip(ox, oy):
                    d = math.hypot(iox - x, ioy - y)
                    if d <= self.rr:
                        self.obstacle_map[ix][iy] = True
                        break

    @staticmethod
    def get_motion_model():
        # dx, dy, cost
        motion = [[1, 0, 1],
                  [0, 1, 1],
                  [-1, 0, 1],
                  [0, -1, 1],
                  [-1, -1, math.sqrt(2)],
                  [-1, 1, math.sqrt(2)],
                  [1, -1, math.sqrt(2)],
                  [1, 1, math.sqrt(2)]]

        return motion

def main():
    print(__file__ + " start!!")

    # start and goal position
    sx = x_r[0]*10  # [m]
    sy = x_r[1]*10 # [m]
    gx = p_g[0]*10  # [m]
    gy = p_g[1]*10  # [m]
    grid_size = 0.5 # [m]
    robot_radius = W.r*10 + 1.2   # [m]

    num_points = 50  # Adjust this value as needed
    line_points1 = bresenham_line(p_1[0] * 10, p_1[1] * 10, p_2[0] * 10, p_2[1] * 10, num_points)
    line_points2 = bresenham_line(p_3[0] * 10, p_3[1] * 10, p_4[0] * 10, p_4[1] * 10, num_points)

    obs_points = []
    for obstacle in O:
        center_x, center_y = obstacle
        radius = W.r_obs  # Use the radius of the obstacle (W.r_obs)
        num_circle_points = 20  # Adjust this value to control the circle resolution
        circle_points = generate_circle_points(center_x * 10, center_y * 10, radius * 11, num_circle_points)
        obs_points.extend(circle_points)

    # set obstacle positions
    ox, oy = [], []

    for i in range(0, 24):
```

```
        ox.append(i)
        oy.append(0)
        ox.append(i)
        oy.append(24)
        ox.append(0)
        oy.append(i)
        ox.append(24)
        oy.append(i)

    for point in line_points1:
        ox.append(point[0])
        oy.append(point[1])

    for point in line_points2:
        ox.append(point[0])
        oy.append(point[1])

    ox.extend([point[0]/10 for point in obs_points])
    oy.extend([point[1]/10 for point in obs_points])

    if show_animation:  # pragma: no cover
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "og")
        plt.plot(gx, gy, "xb")
        plt.grid(True)
        plt.axis("equal")


    print(ox[-1])
    a_star = AStarPlanner(ox, oy, grid_size, robot_radius)
    rx, ry = a_star.planning(sx, sy, gx, gy)
    # path= [rx[i],ry[i]] for i in range(len(rx))

    print(rx)
    print(ry)
    path = np.array([[x/10, y/10] for x, y in zip(rx, ry)])[::-1]

    print('this',path)
    if show_animation:  # pragma: no cover
        plt.plot(rx, ry, "-r")
        plt.pause(0.001)
        plt.show()


    W.run(path)

if __name__ == '__main__':
    main()
```