המחלקה להנדסה מכנית

עבודת סוף סמסטר (2021-01) – קורס מבוא לאלגוריתמיקה ברובוטיקה 30242

ניווט רובוטים בסביבה לא ידועה למטרת מיפוי מכשולים סטטיים ודינאמיים – קוד קונספטואלי

מרצה: ד"ר ילוב הנדזל שרון

מגישים: גיא יעקב – 311320485

גיא צור – 203298310

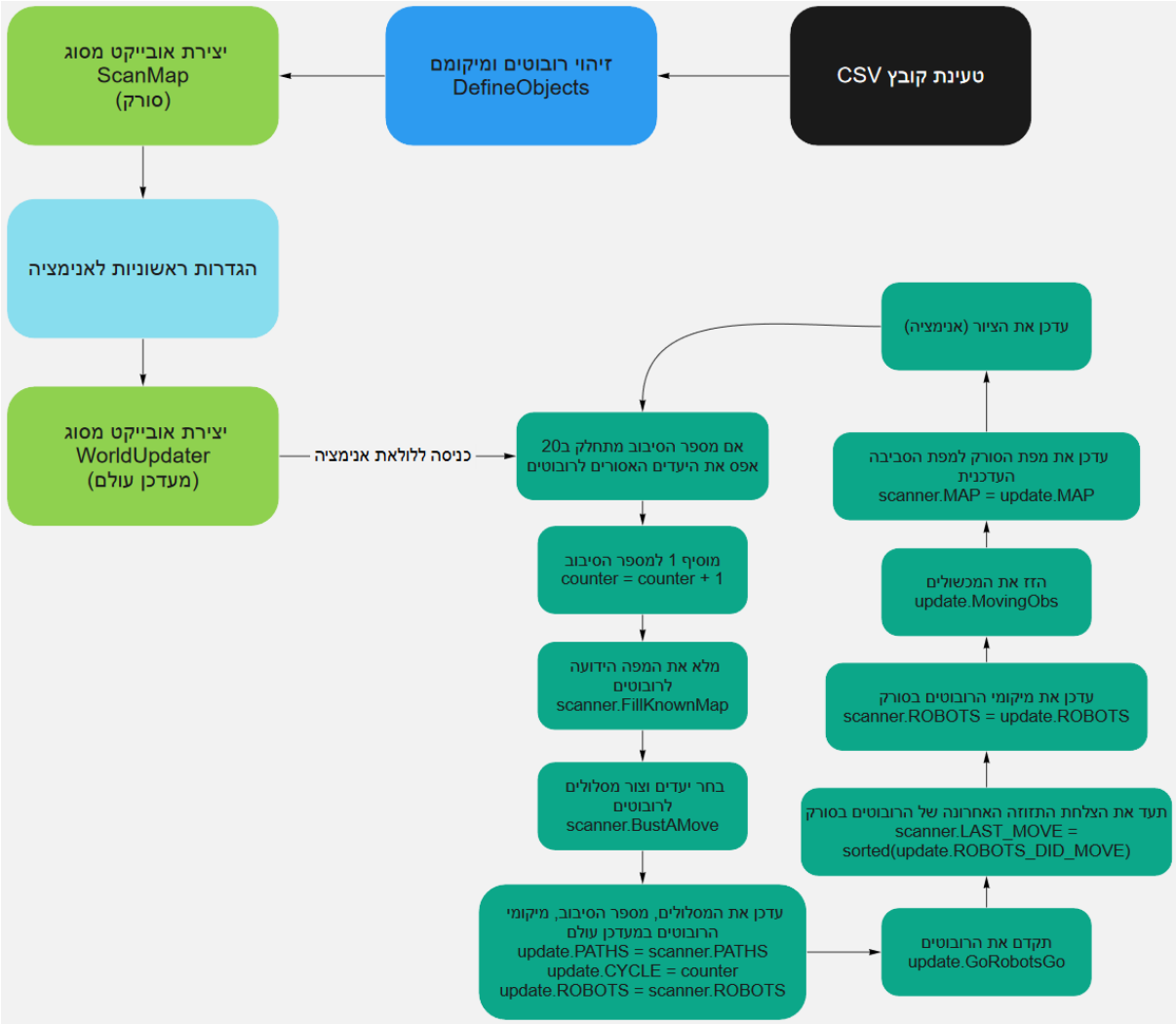
טל פומרנץ – 200047546

תוכן עניינים

רשימת איורים

1 הצגת מרכיבי הקוד

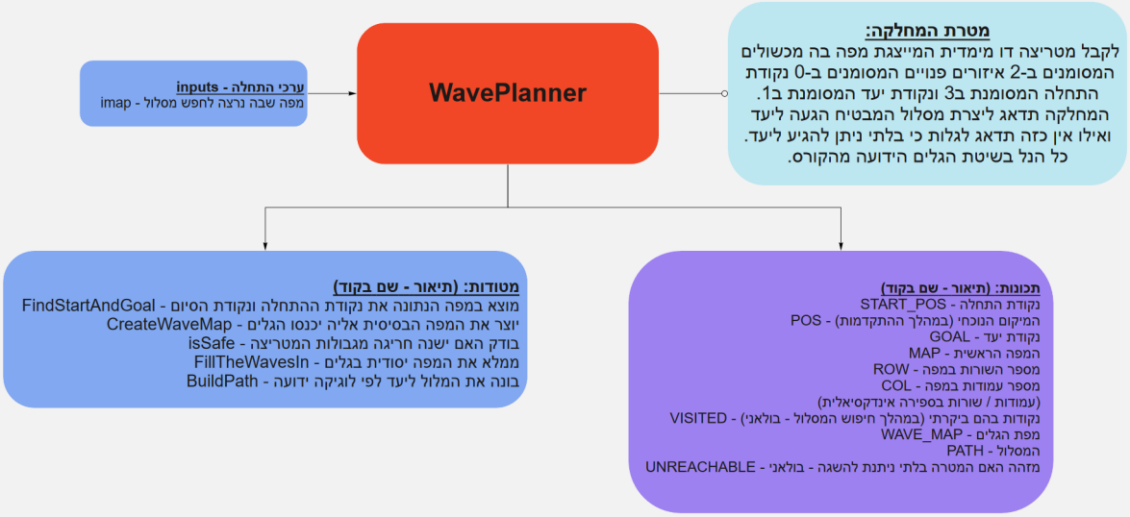
1.1 תרשים כללי לפעולת הקוד



איור 1-1 תרשים זרימה של המתודולוגיה של הקוד

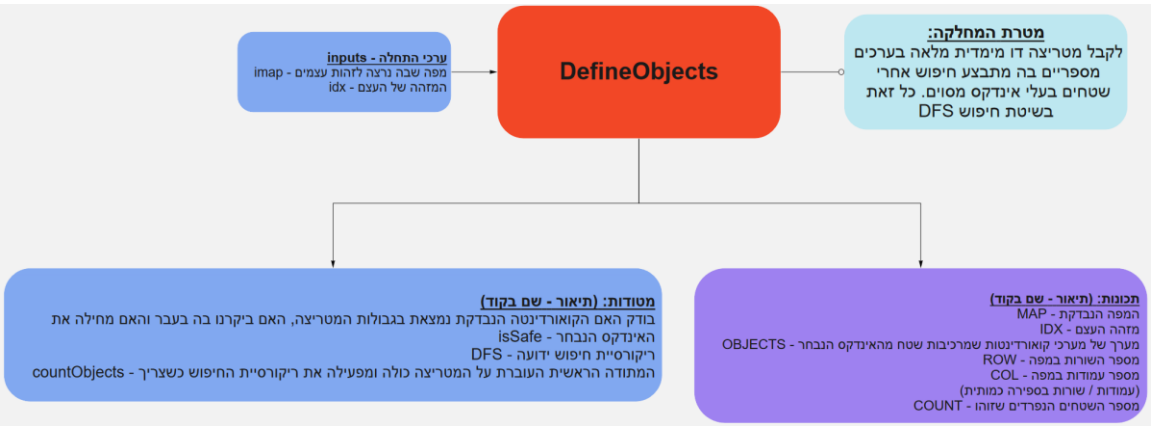
1.2 הצגה ככלית למחלקות – תכונות ומטודות

1.2.1 WavePlanner



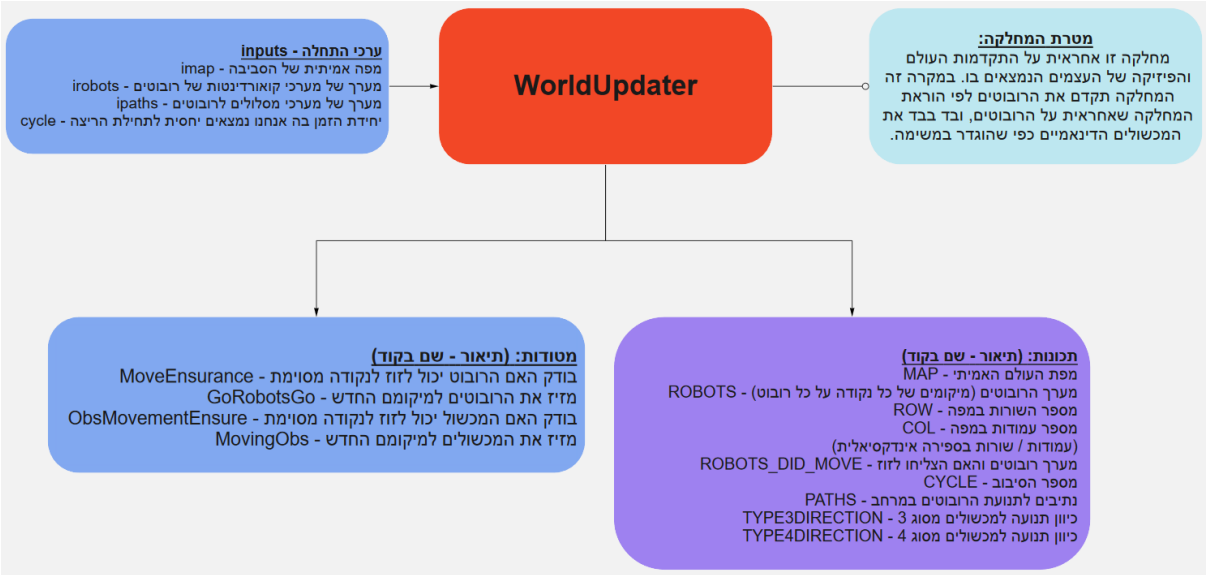
איור 1-2 WavePlanner

1.2.2 DefineObjects



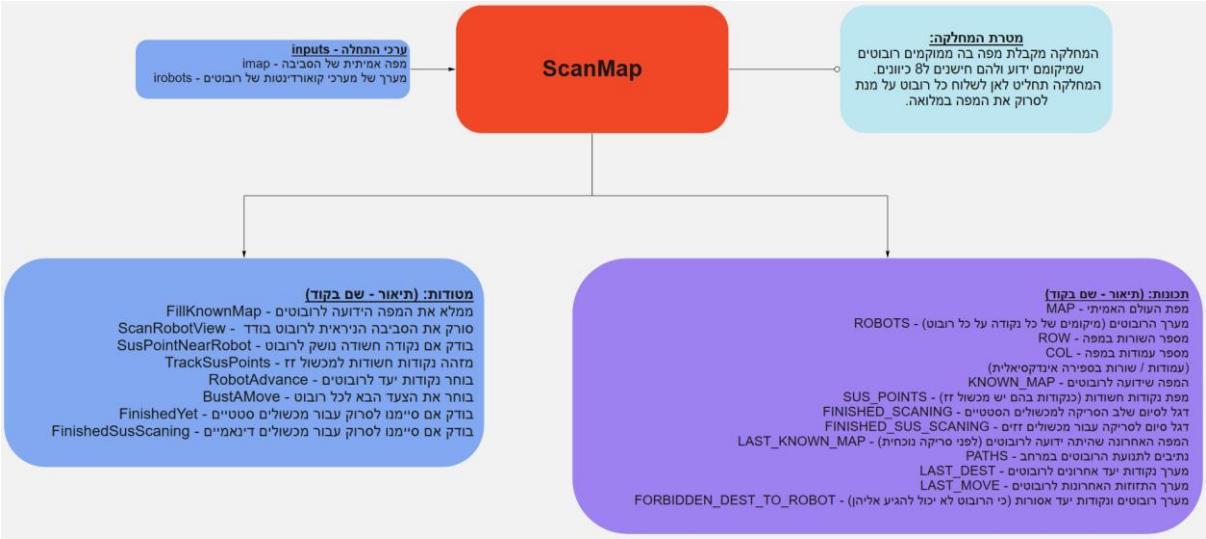
איור 1-3 Define Objects

1.2.3 WorldUpdater



איור 1-4 World Updater

1.2.4 ScanMap



איור 1-5 ScanMap

2 פירוט המטודות במחלקות המרכזיות

המחלקות Waveplanner ו- DefineObjects אינן מרשימות במיוחד מכיוון שהן גנריות לחלוטין. על כן אנו נפרט על המטודות הנמצאות במחלקות האחרות שם קיימת הסיבוכיות האמיתית בקוד.

פירוט המטודות יעשה לפי סדר לוגי ולא לפי סדר ההופעה בקוד.

מומלץ לקרוא את פרק זה תוך מעקב אחרי התרשים הכללי של הקוד בפרק הראשון.

2.1 מטודות במחלקת ScanMap

2.1.1 FillKnownMap

שורות בקוד 185 עד 205:

```
185     def FillKnownMap(self):
186         self.LAST_KNOWN_MAP.clear()
187         self.LAST_KNOWN_MAP = copy.deepcopy(self.KNOWN_MAP)
188
189         for y in range(self.ROW+1):
190             for x in range(self.COL+1):
191              if self.KNOWN_MAP[y][x] == 1:
192                  self.KNOWN_MAP[y][x] = -1
193
194         for robot in self.ROBOTS:
195             for coordonate in robot:
196                 self.KNOWN_MAP[coordonate[1]][coordonate[0]] = 1
197
198         for robot in self.ROBOTS:
199             rowNbr = [-1, -1, -1, 0, 0, 1, 1, 1]
200             colNbr = [-1, 0, 1, -1, 1, -1, 0, 1]
201             for a in range(8):
202                 direction = (colNbr[a],rowNbr[a])
203                 self.ScanRobotView(robot, direction)
204
205         self.TrackSusPoints()
```

איור 2-1 מטודת FillKnownMap

המטודה פותחת בניקוי המפה האחרונה שהייתה ידועה לרובוטים והעתקת המפה הנוכחית לתוך המפה האחרונה שהייתה ידועה לרובוטים. לאחר מכן מתבצעת הסריקה שלמעשה משנה את המפה הידועה לרובוטים, תחילה אילו הרובוטים זזו, המיקום שלהם במפה הידועה לא נכון ולכן נסמן במקום האחרון של הרובוטים אינדקס של משהו לא ידוע (בחרנו שאותו אינדקס יהיה מינוס 1). לאחר מכן נעדכן במפה את המיקום הנכון של הרובוטים. לאחר מכן, לכל נקודה על כל רובוט נשלח קרניים ל8 כיוונים אותם קרניים יסרקו את אשר הן פוגעות בו עד אשר יפגעו בגבול או במכשול או ברובוט אחר. לאחר הסריקה נפעיל מטודה לזיהוי נקודות חשודות על בסיס שינויים בין המפה הידועה עכשיו לבין המפה האחרונה שהייתה ידועה לנו (יפורט בהמשך).

2.1.2 ScanRobotView

שורות בקוד 208 עד 222:

```
208     def ScanRobotView(self, robot, direction):
209         for coordinate in robot:
210             x = coordinate[0] + direction[0]
211             y = coordinate[1] + direction[1]
212             while (y >= 0 and y < self.ROW+1 and x >= 0 and x < self.COL+1):
213                 if self.MAP[y][x] == 0:
214                     self.KNOWN_MAP[y][x] = 0
215                 elif self.MAP[y][x] == 1:
216                     self.KNOWN_MAP[y][x] = 1
217                     break
218                 else:
219                     self.KNOWN_MAP[y][x] = 2
220                     break
221                 x = x + direction[0]
222                 y = y + direction[1]
```

איור 2-2 מטודת ScanRobotView

המטודה מקבלת סט קואורדינטות של רובוט: [(x1,y1),(x2,y2),...,(xn,yn)]

וכיוון התקדמות (x,y) כאשר כיוון ההתקדמות יכול להיות:

(-1,-1)	(0,-1)	(1,-1)
(-1,0)	---	(1,0)
(-1,1)	(0,1)	(1,1)

(ראשית הצירים מלמעלה צד שמאל)

הלולאה במטודה סורקת את המטריצה מכל נקודה על הרובוט אל עבר כיוון ההתקדמות עד היתקלות במכשול או רובוט או בהגעה לגבולות המטריצה. הערכים הסרוקים נשמרים ישירות במפה הידועה.

2.1.3 TrackSusPoints

שורות בקוד 237 עד 251:

```
237     def TrackSusPoints(self):
238         for y in range(self.ROW + 1):
239             for x in range(self.COL + 1):
240                 if self.KNOWN_MAP[y][x] == 0 and self.LAST_KNOWN_MAP[y][x] == 2:
241                     add_it_to_sus = not self.SusPointNearRobot(x , y)
242                     if add_it_to_sus:
243                         self.SUS_POINTS[y][x] = not self.SUS_POINTS[y][x]
244                 elif self.KNOWN_MAP[y][x] == 2 and self.LAST_KNOWN_MAP[y][x] == 0:
245                     add_it_to_sus = not self.SusPointNearRobot(x, y)
246                     if add_it_to_sus:
247                         self.SUS_POINTS[y][x] = not self.SUS_POINTS[y][x]
248                 if self.SUS_POINTS[y][x] and self.KNOWN_MAP[y][x] == 1:
249                     self.SUS_POINTS[y][x] = not self.SUS_POINTS[y][x]
250                 elif self.SUS_POINTS[y][x] and self.SusPointNearRobot(x , y):
251                     self.SUS_POINTS[y][x] = not self.SUS_POINTS[y][x]
```

איור 2-3 מטודת TrackSusPoints

ראשית נציין כי מטריצת SUS_POINTS הינה מטריצה בוליאנית המאותחלת לFALSE בכל תא. המטודה עוברת על כל תא במפה הידועה עכשיו, ומשווה אותו למפה שהייתה ידועה לפני הסריקה הנוכחית. היגיון המטודה פשוט למדי, אם היה שם קודם מכשול ועכשיו אין שם מכשול, המקום הזה חשוד, ואם לא היה שם קודם מכשול ועכשיו יש שם מכשול, המקום חשוד. המקום כבר לא חשוד אם היה שם מכשול ועכשיו יש שם רובוט או אם המקום היה חשוד אבל רובוט נמצא ממש ליד* המקום הזה (* שורה 250 מפורט במטודה הבאה).

2.1.4 SusPointNearRobot

שורות בקוד 224 עד 235:

```
224     def SusPointNearRobot(self, x , y):
225         for robot in self.ROBOTS:
226             for coo in robot:
227                 x_r = coo[0]
228                 y_r = coo[1]
229                 rowNbr = [-1, -1, -1, 0, 0, 1, 1, 1]
230                 colNbr = [-1, 0, 1, -1, 1, -1, 0, 1]
231                 for a in range(8):
232                     direction = (colNbr[a], rowNbr[a])
233                     if x == x_r + direction[0] and y == y_r + direction[1]:
234                         return True
235         return False
```

איור 2-4 מטודת SusPointNearRobot

המטודה מקבלת ערכי X ו- Y של נקודה ומחזירה True או False אם הנקודה נמצאת ליד רובוט או לא, בהתאמה.

2.1.5 BustAMove

שורות בקוד 324 עד 368:

```
324     def BustAMove(self):  # Infected Mushroom - Classical Mushroom - Song #1
325         if not self.FINISHED_SCANING:
326             self.FINISHED_SCANING = self.FinishedYet()
327         dest_arr = self.RobotAdvance()
328         self.LAST_DEST.clear()
329         self.LAST_DEST = sorted(dest_arr)
330         self.PATHS.clear()
331         for robot_dest in dest_arr:
332             # Here you may choose any map.csv file wich contains:
333             # 0's for empty space, 1 for target point, 2's for obstacles
334             # and 3 for starting point
335             map_for_waveplanner = []
336             for i in range(self.ROW + 1):
337                 line = []
338                 for j in range(self.COL + 1):
339                     line.append(0)
340                 map_for_waveplanner.append(line)
341             for y in range(self.ROW + 1):
342                 for x in range(self.COL + 1):
343                     if self.KNOWN_MAP[y][x] == 2 or self.KNOWN_MAP[y][x] == 1:
344                         map_for_waveplanner[y][x] = 2
345             for coo in self.ROBOTS[robot_dest[0]]:
346                 map_for_waveplanner[coo[1]][coo[0]] = 0
347             coo = self.ROBOTS[robot_dest[0]]
348             x = coo[0][0]
349             y = coo[0][1]
350             map_for_waveplanner[y][x] = 3
351             if robot_dest[1][0] != x:
352                 map_for_waveplanner[robot_dest[1][1]][robot_dest[1][0]] = 1
353             else:
354                 if robot_dest[1][0] + 1 <= self.COL:
355                     map_for_waveplanner[robot_dest[1][1]][robot_dest[1][0] + 1] = 1
356                 else:
357                     map_for_waveplanner[robot_dest[1][1]][robot_dest[1][0] - 1] = 1
358             b2 = WavePlanner(map_for_waveplanner)
359             if not b2.UNREACHABLE:
360                 path_arr = []
361                 last_pos = (x, y)
362                 for pos in b2.PATH:
363                     tmoora = (pos[0] - last_pos[0], pos[1] - last_pos[1])
364                     last_pos = pos
365                     path_arr.append(tmoora)
366                 self.PATHS.append([robot_dest[0], path_arr])
367             else:
368                 self.KNOWN_MAP[robot_dest[1][1]][robot_dest[1][0]] = 2
```

איור 2-5 מטודת BustAMove

תחילה המטודה בודקת האם הסריקה עבור מכשולים סטטיים הסתיימה (שורה 326 יפורט בהמשך). לאחר מכן מופעלת המטודה המסובכת ביותר בקוד והיא המטודה של בחירת נקודות יעד לרובוטים (שורה 327 יפורט בהמשך), אותה מטודה תחזיר מערך של נקודות יעד לכל רובוט. לאחר מכן ננקה את מערך נקודות היעד האחרונות ונשמור את נקודות היעד הנוכחיות למערך האחרונות, וננקה את מערך הנתיבים.

לאחר מכן, עבור כל יעד לרובוט ניצור מפה שאותה נשלח ל- WavePlanner (המפה צריכה לעמוד בדרישות המפורטות בהערה בשורות 332 עד 334). המפה הנ"ל נבנית על בסיס המפה שידועה לרובוטים כאשר אנו מניחים שהנקודות הלא ידועות לרובוטים פנויות, ועבור כל רובוט, הרובוטים האחרים נחשבים למכשול. לאחר בניית המפה, נשלח את המפה למחלקה המטפלת בבניית המסלול אל היעד – שורה 358. אותה מחלקה יודעת לזהות האם ניתן להגיע

לנקודת היעד, אם היא זיהתה שניתן, אזי קיים גם מסלול. אם זיהתה שלא ניתן להגיע אזי נסמן במפה שידוע לרובוטים – מכשול. אם יש מסלול אנו נחשב את התמורות שיש על כל נקודה ברובוט לבצע כדי להתקדם, למעשה נרכיב ממערך המסלול שמציין נקודות במטריצה שאותם הרובוט צריך לעבור, מערך תמורות שיש על הרובוט לבצע.

2.1.6 FinishedYet

שורות בקוד 370 עד 375:

```
370    def FinishedYet(self):
371        for y in range(self.ROW + 1):
372            for x in range(self.COL + 1):
373                if self.KNOWN_MAP[y][x] == -1:
374                    return False
375        return True
```

איור 6-2 מטודת FinishedYet

המטודה עוברת על כל המפה הידועה לרובוטים ואם יש שם מינוס 1 איפשהו, עדיין לא סיימנו לסרוק עבור מכשולים סטטיים. אחרת, סיימנו.

2.1.7 RobotAdvance

שורות בקוד 253 עד 322:

חלק ראשון:

```
253     def RobotAdvance(self):
254         if len(self.LAST_DEST) != 0:
255             for move in self.LAST_MOVE:
256                 if not move[1]:
257                     for dest in self.LAST_DEST:
258                         if dest[0] == move[0]:
259                             self.FORBIDDEN_DEST_TO_ROBOT.append((move[0],dest[1]))
260                             break
261         if not self.FINISHED_SCANING: # static obs
262             occ_robots = []
263             dest_to_robot = []
264             idx_and_distance_arr = []
265             for y in range(self.ROW + 1):
266                 for x in range(self.COL + 1):
267                     if self.KNOWN_MAP[y][x] == -1:
268                         dist = []
269                         for robot in self.ROBOTS:
270                             dist.append(math.dist([x,y],[robot[0][0],robot[0][1]]))
271                         i=0
272                         for distance in dist:
273                             idx_and_distance_arr.append((i , distance))
274                             i = i + 1
275                         idx_and_distance_sorted = sorted(idx_and_distance_arr, key=lambda x: x[1])
276                         pass_this_dest = False
277                         min_dist_between_dests = (max([self.ROW,self.COL]) / (max([self.ROW,self.COL])/10)) * 2
278                         for dest in dest_to_robot:
279                             if math.dist([x,y],[dest[1][0],dest[1][1]]) < min_dist_between_dests:
280                                 pass_this_dest = True
281                         if not pass_this_dest:
282                             for robot_and_distance in idx_and_distance_sorted:
283                                 if robot_and_distance[0] not in occ_robots:
284                                     if (robot_and_distance[0],(x,y)) not in self.FORBIDDEN_DEST_TO_ROBOT:
285                                         occ_robots.append(robot_and_distance[0])
286                                         dest_to_robot.append((robot_and_distance[0],(x,y)))
287                                         break
288                     if len(occ_robots) == len(self.ROBOTS):
289                         return dest_to_robot
290             return dest_to_robot
```

איור 2-7 מטודת RobotAdvance חלק 1

המטודה יודעת לבחור נקודות יעד לכל רובוט ל2 שלבים של הסריקה – שלב סריקה ראשונית (מכשולים סטטיים) ושלב סריקה מתקדם עבור מכשולים דינמיים (באם קיימים). ראשית, לפני התחלת תהליך קבלת ההחלטה, המטודה מתעדת האם נקודת יעד לרובוט מסוים ברת השגה עבור אותו רובוט כאשר ההיגיון הוא פשוט: אילו בתזוזה האחרונה הרובוט לא הצליח לזוז ממקומו, סימן שנקודת היעד שנבחרה לאותו הרובוט באותו שלב, אינה ברת השגה מבחינתו. ולכן ישנו מערך שמכיל צמדים של אינדקס רובוט ונקודת יעד אסורה: (robot_index , (x , y)).

תהליך קבלת ההחלטות עבור נקודות יעד לרובוטים בשלב הסריקה עבור מכשולים סטטיים נעשית בשורות 262 עד 290:

ניצור מערך שיכיל את האינדקסים של הרובוטים שכבר נתנו להם נקודות יעד, מערך שיכיל את נקודות היעד, ומערך שיכיל את אינדקס הרובוט ומרחקו מנקודת יעד מועמדת. לאחר מכן נרוץ בלולאה על כל המטריצה עד שנפגוש נקודה לא ידועה. כאשר נפגוש בנקודה כזו, עבור כל רובוט נחשב את המרחק מנקודה שרירותית על הרובוט (הנחה של רובוטים קטנים יחסית

למפה וקטנים אחד יחסית לשני) אל הנקודה המדוברת ונאכן את התוצאות במערך המיועד ולאחר מכן גם נמיין את המערך לפי המרחקים. לאחר מכן נבדוק האם הנקודה רחוקה מספיק מנקודת היעד האחרונה שבחרנו – שורות 276 עד 280. אילו היא עומדת בתנאים נשייך את הנקודה עבור רובוט פנוי לפי סדר עדיפות לרובוט הקרוב ביותר לנקודה, אם ורק אם נקודת היעד מותרת לרובוט. נסיים את הלולאה כאשר לכל רובוט יש נקודת יעד, אם לא אז הלולאה תמשיך לרוץ עד הסוף.

חלק שני:

```
291
292         else: # Go and track what is going on the sus points
293             self.FINISHED_SUS_SCANING = self.FinishedSusScaning()
294             occ_robots = []
295             dest_to_robot = []
296             idx_and_distance_arr = []
297             for y in range(self.ROW + 1):
298                 for x in range(self.COL + 1):
299                     if self.SUS_POINTS[y][x]:
300                         dist = []
301                         for robot in self.ROBOTS:
302                             dist.append(math.dist([x, y], [robot[0][0], robot[0][1]]))
303                         i = 0
304                         for distance in dist:
305                             idx_and_distance_arr.append((i, distance))
306                             i = i + 1
307                         idx_and_distance_sorted = sorted(idx_and_distance_arr, key=lambda x: x[1])
308                         pass_this_dest = False
309                         min_dist_between_dests = 5 #(max([self.ROW, self.COL]) / (max([self.ROW, self.COL]) / 10))
310                         for dest in dest_to_robot:
311                             if math.dist([x, y], [dest[1][0], dest[1][1]]) < min_dist_between_dests:
312                                 pass_this_dest = True
313                         if not pass_this_dest:
314                             for robot_and_distance in idx_and_distance_sorted:
315                                 if robot_and_distance[0] not in occ_robots:
316                                     if (robot_and_distance[0], (x, y)) not in self.FORBIDDEN_DEST_TO_ROBOT:
317                                         occ_robots.append(robot_and_distance[0])
318                                         dest_to_robot.append((robot_and_distance[0], (x, y)))
319                                         break
320                     if len(occ_robots) == len(self.ROBOTS):
321                         return dest_to_robot
322             return dest_to_robot
```

איור 2-8 מטודת RobotAdvance חלק 2

החלק שני מדבר על קבלת ההחלטות עבור נקודת יעד לרובוט בשלב החיפוש אחר נקודות דינאמיות. שלב זה מאוד דומה לשלב הקודם, השינויים היחידים בו הם:

- את הנקודה המועמדת נבחר על בסיס המשה של הנקודות החשודות – שורה 299
- המרחק המינימלי בין נקודות יעד קטן יותר – שורה 309

2.2 מטודות במחלקת WorldUpdater

2.2.1 GoRobotsGo

שורות בקוד 410 עד 430:

```
410     def GoRobotsGo(self):
411         self.ROBOTS_DID_MOVE.clear()
412         for path in self.PATHS:
413             robot = self.ROBOTS[path[0]]
414             y_move = path[1][0][1]
415             x_move = path[1][0][0]
416             if self.MoveEnsurance(x_move, y_move, robot):
417                 self.ROBOTS_DID_MOVE.append((path[0], True))
418                 prv_coo = []
419                 for coo in robot:
420                     self.MAP[coo[1]][coo[0]] = 0
421                     prv_coo.append(coo)
422                 for coo in prv_coo:
423                     self.ROBOTS[path[0]].remove(coo)
424                     new_x = coo[0] + x_move
425                     new_y = coo[1] + y_move
426                     self.ROBOTS[path[0]].append((new_x,new_y))
427                     self.MAP[new_y][new_x] = 1
428
429             else:
430                 self.ROBOTS_DID_MOVE.append((path[0], False))
```

איור 2-9 מטודת GoRobotsGo

המטודה מעדכנת את המיקום של כל נקודה על כל רובוט, במפה האמיתית בהתאם לתזוזה שעל הרובוט לבצע. המטודה מפעילה שיקול דעת בעזרת מטודת עזר אשר אומרת לה האם הרובוט יכול לזוז לכיוון הנדרש (שורה 416). אם הוא יכול לזוז אז המטודה תעדכן את המיקום של כל נקודה בו ותתעד תזוזה מוצלחת, באם הוא לא יכול לזוז, המטודה תתעד תזוזה לא מוצלחת.

2.2.2 MoveEnsurance

שורות בקוד 397 עד 408

```
397     def MoveEnsurance(self, x_move, y_move, robot_coo):
398         for coo in robot_coo:
399             if coo[1] + y_move < 0 or coo[1] + y_move > self.ROW:
400                 return False
401             elif coo[0] + x_move < 0 or coo[0] + x_move > self.COL:
402                 return False
403             elif self.MAP[coo[1] + y_move][coo[0] + x_move] == 2 or self.MAP[coo[1] + y_move][coo[0] + x_move] == 3 or self.MAP[coo[1] + y_move][coo[0] + x_move] == 4:
404                 return False
405             elif self.MAP[coo[1] + y_move][coo[0] + x_move] == 1 and (coo[0] + x_move , coo[1] + y_move) not in robot_coo:
406                 return False
407
408         return True
```

איור 2-10 מטודת MoveEnsurance

המטודה מקבלת סט קואורדינטות של רובוט ותזוזה שהרובוט צריך לבצע ומוודאת שבתזוזה הרובוט לא חורג מגבולות המפה, לא עולה על מכשול, או לא עולה על רובוט אחר. באם כל התנאים לכל נקודה על הרובוט מתקיימים אז המטודה תחזיר אמת, ולא – תחזיר שקר.

2.2.3 MovingObs

שורות בקוד 448 עד 485:

```
448     def MovingObs(self):
449         type3 = DefineObjects(self.MAP, 3)
450         type4 = DefineObjects(self.MAP, 4)
451         dont_move = False
452         if self.CYCLE % 2 == 0:
453             for type3obj in type3.OBJECTS:
454                 for coo in type3obj:
455                     if coo[0] + self.TYPE3DIRECTION < 0 or coo[0] + self.TYPE3DIRECTION > self.COL \
456                             or not (self.MAP[coo[1]][coo[0] + self.TYPE3DIRECTION] == 0
457                                     or self.MAP[coo[1]][coo[0] + self.TYPE3DIRECTION] == 3):
458                         self.TYPE3DIRECTION = self.TYPE3DIRECTION * -1
459                         break
460                 for coo in type3obj:
461                     if not self.ObsMovementEnsure(coo,3):
462                         dont_move = True
463                         break
464                 if not dont_move:
465                     for coo in type3obj:
466                         self.MAP[coo[1]][coo[0]] = 0
467                         self.MAP[coo[1]][coo[0] + self.TYPE3DIRECTION] = 3
468
469         dont_move = False
470         if self.CYCLE % 4 == 0:
471             for type4obj in type4.OBJECTS:
472                 for coo in type4obj:
473                     if coo[1] + self.TYPE4DIRECTION < 0 or coo[1] + self.TYPE4DIRECTION > self.ROW \
474                             or not (self.MAP[coo[1] + self.TYPE4DIRECTION][coo[0]] == 0
475                                     or self.MAP[coo[1] + self.TYPE4DIRECTION][coo[0]] == 4):
476                         self.TYPE4DIRECTION = self.TYPE4DIRECTION * -1
477                         break
478                 for coo in type4obj:
479                     if not self.ObsMovementEnsure(coo, 4):
480                         dont_move = True
481                         break
482                 if not dont_move:
483                     for coo in type4obj:
484                         self.MAP[coo[1]][coo[0]] = 0
485                         self.MAP[coo[1] + self.TYPE4DIRECTION][coo[0]] = 4
```

איור 11-2 מטודת MovingObs

המטודה אחראית להזיז את המכשולים הדינמיים בכיוון אליהן יש להזיזם. ראשית המטודה מזהה את המכשולים הזזים באמצעות מחלקת DefineObjects ממנה נקבל את סט הקואורדינטות עבור כל מכשול שנבחר. לאחר מכן נזיז את כל מה שמסומן ב3 כל 2 פעימות זמן (שורות 452 עד 467) וכל מה שמסומן ב4 כל 4 פעימות זמן (שורות 469 עד 485). פעולת ההזזה דומה לוגית לשני המקרים:

עבור כל קואורדינטה של כל אובייקט נבדוק האם האובייקט יכול לזוז לכיוון אליו רוצה לזוז, אם לא אז נשנה את הכיוון. אם כן נקדם את הקואורדינטות של האובייקט בכיוון המדובר אם ורק אם הוא יכול להתקדם לשם (נדע בעזרת מטודת עזר, יפורט בהמשך).

2.2.4 ObsMovementEnsure

שורות בקוד 432 עד 446:

```
432     def ObsMovementEnsure(self, obs_coo, type):
433         x = obs_coo[0]
434         y = obs_coo[1]
435         if type == 3:
436             if x + self.TYPE3DIRECTION > self.COL or x + self.TYPE3DIRECTION < 0:
437                 return False
438             if self.MAP[y][x+self.TYPE3DIRECTION] == 1 or self.MAP[y][x+self.TYPE3DIRECTION] == 2 or self.MAP[y][x+self.TYPE3DIRECTION] == 4:
439                 return False
440             return True
441         elif type ==4:
442             if y+self.TYPE4DIRECTION > self.ROW or y+self.TYPE4DIRECTION<0:
443                 return False
444             if self.MAP[y+self.TYPE4DIRECTION][x] == 1 or self.MAP[y+self.TYPE4DIRECTION][x] == 2 or self.MAP[y+self.TYPE4DIRECTION][x] == 3:
445                 return False
446             return True
```

איור 2-12 מטודת ObsMovementEnsure

המטודה מקבלת סט של נקודות של מכשול ואת סוגו ובודקת שהוא יכול לזוז לכיוון אליו רוצה לזוז. המטודה תחזיר אמת אם ורק אם המכשול יכול לזוז לאותו כיוון, ולא תחזיר שקר.