Δεκέμβριος 2016

Εργασία στο μάθημα:

Ευφυή Συστήματα Ρομπότ

Κωνσταντίνος Μαυροδής

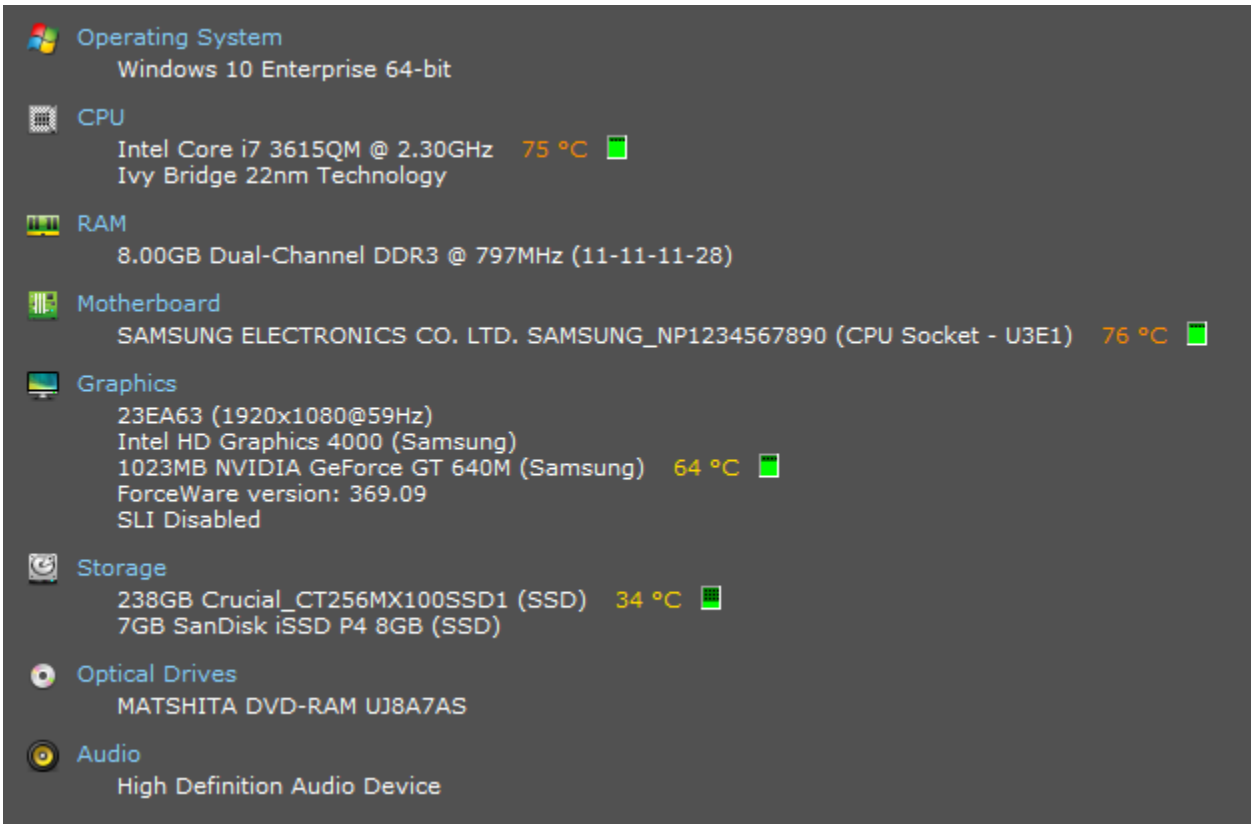
mavrkons@auth.gr

ΑΕΜ: 7922

Εισαγωγή

Η παρούσα αναφορά περιγράφει την εργασία που εκπονήθηκε κατά τη διάρκεια του μαθήματος Ευφυή Συστήματα Ρομπότ του έτους 2016-2017. Η εργασία βασίστηκε στην υποδομή και των κώδικα που δόθηκε από τον διδάσκοντα και απαντήθηκαν ένα προς ένα τα ερωτήματα που ζητήθηκαν από αυτόν. Σε γενικές γραμμές η εργασία ασχολείται με την δημιουργία του λογισμικού ενός ρομπότ, ώστε αυτό να εκτελεί αυτόνομη πλοήγηση σε ένα μη γνωστό εκ των προτέρων περιβάλλον. Σκοπός της είναι η συμπλήρωση του υπάρχοντα κώδικα ώστε να έχουμε ένα πιο ολοκληρωμένο σύστημα με τις λειτουργίες που απαιτούνται.

Η εκτέλεση της εργασίας έγινε με τη χρήση της τεχνολογίας των Virtual Machines. Ως περιβάλλον Host χρησιμοποιήθηκε το λειτουργικό σύστημα Windows 10 σε ένα μηχάνημα με τις παρακάτω προδιαγραφές:



Επάνω σε αυτό χρησιμοποιήθηκε το λογισμικό VirtualBox και ως Guest OS το image (Ubuntu 14.04) που δόθηκε από τον διδάσκοντα χωρίς να τροποποιηθεί κάποια παράμετρός του. Για τη συγγραφή/τροποποίηση του κώδικα χρησιμοποιήθηκε το πρόγραμμα Sublime Text και για την εκτέλεσή του το τερματικό Terminator.

Ερώτημα 1ο

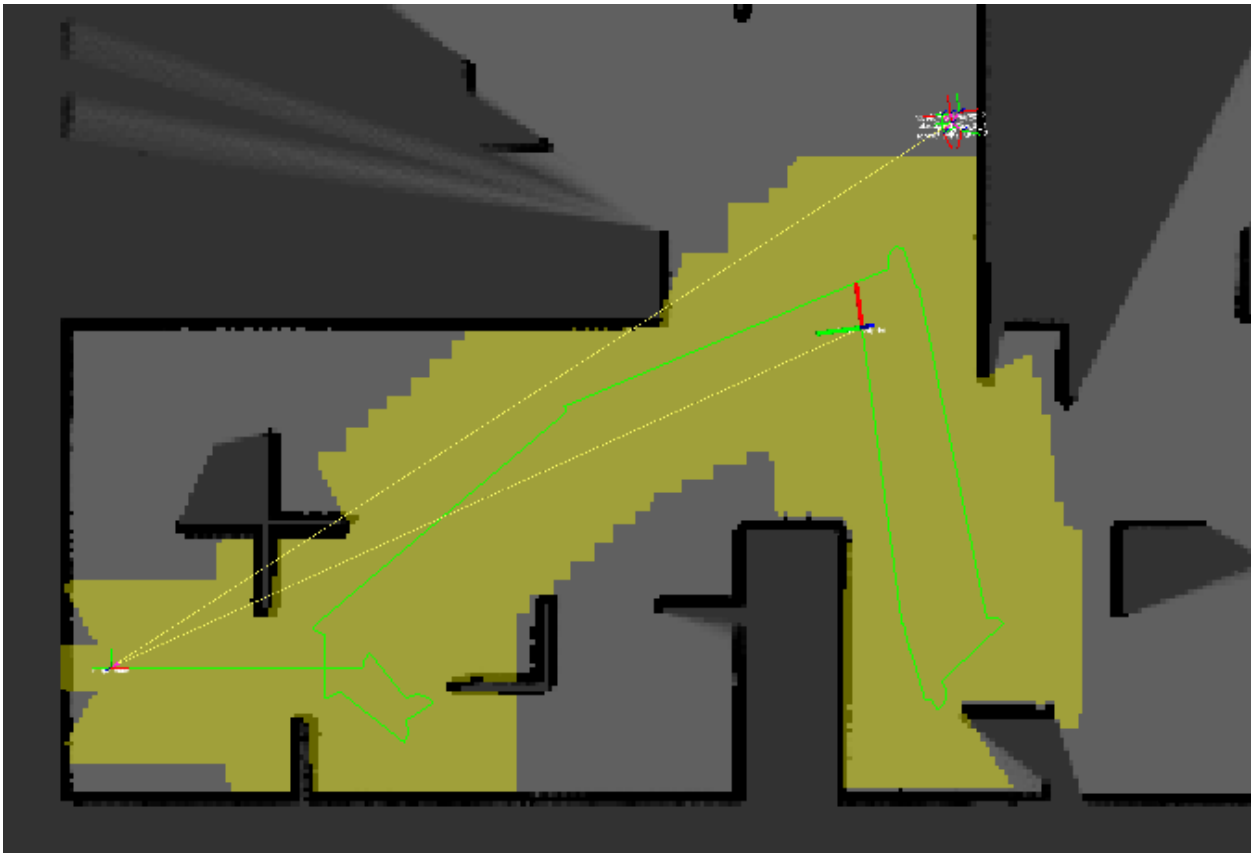
Στο πρώτο ερώτημα ζητήθηκε η υλοποίηση του module που επιτρέπει στο ρομπότ να κινείται στον χώρο χωρίς να χτυπά σε εμπόδια, χρησιμοποιώντας τις τιμές που διαβάζονται από το λέιζερ. Ο αλγόριθμος που υλοποιήθηκε φαίνεται στα παρακάτω screenshot.

```
# Produces speeds from the laser
def produceSpeedsLaser(self):
  scan = self.laser_aggregation.laser_scan
  linear  = 0
  angular = 0
  ############################### NOTE QUESTION ############################
  # Check what laser_scan contains and create linear and angular speeds
  # for obstacle avoidance

  # Checks if there is an obstacle near the front of the robot
  if min(scan[(len(scan)/2)-80:(len(scan)/2)+80]) > 1:
    linear  = 1
    angular = 0
  else:
    linear  = 0
    angular = 1

  ##########################################################################

  return [linear, angular]
```

```
else:
  ############################### NOTE QUESTION ############################
  # Implement obstacle avoidance here using the laser speeds.
  # Hint: Subtract them from something constant

  self.linear_velocity = 0.3*l_laser
  self.angular_velocity = -0.3*a_laser

  ##########################################################################
```

Έτσι, λοιπόν, διαβάζονται οι ενδείξεις του λέιζερ και εντοπίζεται αν βάσει αυτών υπάρχει κάποιο κοντινό εμπόδιο σε έναν μικρότερο κώνο ακριβώς μπροστά από το ρομπότ. Αν πράγματι βρεθεί εμπόδιο εντός αυτού του κώνου δίνεται εντολή στο ρομπότ να στρίψει, διαφορετικά συνεχίζει να πηγαίνει ευθύγραμμα.

Αυτή η υλοποίηση δοκιμάστηκε εκτενώς και βρέθηκε ότι το ρομπότ δεν προσκρούει σε κάποιον τοίχο, εκτός από πολύ σπάνιες περιπτώσεις. Ο αλγόριθμος, επομένως είναι καλός αλλά όχι τέλειος. Ένα παράδειγμα εκτέλεσής του φαίνεται παρακάτω.



Ερώτημα 2ο

Στο δεύτερο ερώτημα το ζητούμενο ήταν να γίνει ορατό το path που σχηματίζεται από το target και τα subtargets όταν υπολογίζεται μία διαδρομή. Για να γίνει αυτό προστέθηκε ο κώδικας που φαίνεται παρακάτω. Χρησιμοποιήθηκαν προφανώς τα self.robot_perception.resolution και self.robot_perception.origin όπως άλλωστε και σε μερικές γραμμές παρακάτω ήταν ήδη κάτι παρόμοιο υλοποιημένο.

```
######################### NOTE: QUESTION  ##############################
# Fill the ps.pose.position values to show the path in RViz
# You must understand what self.robot_perception.resolution
# and self.robot_perception.origin are.

ps.pose.position.x = p[0] * self.robot_perception.resolution + self.robot_perception.origin.get('x')
ps.pose.position.y = p[1] * self.robot_perception.resolution + self.robot_perception.origin.get('y')

#########################################################################
```

Ερώτημα 3ο

Το επόμενο ερώτημα είχε να κάνει με την ακολούθηση του μονοπατιού που σχηματίζεται βάσει των subtargets. Ο αλγόριθμος που υλοποιήθηκε χρησιμοποιεί τις συντεταγένες του ρομπότ και του επόμενου υπό-στόχου ώστε να βρει την γωνία που σχηματίζει η μεταξύ τους γωνία. Στη συνέχεια συγκρίνει αυτήν

με την γωνία θήτα που δίνει το robot pose και τελικά στρίβει έως ότου η διαφορά ανάμεσα στις δύο γωνίες γίνει επαρκώς μικρή. Αφού ολοκληρωθεί αυτή η διαδικασία το ρομπότ πλέον κοιτάει στον στόχο. Τελευταίο βήμα είναι να αποκτήσει το ρομπότ γραμμική ταχύτητα μέχρι να φτάσει στον στόχο. Φυσικά, σε περίπτωση που χρειαστούν οποιεσδήποτε διορθωτικές κινήσεις κατά αυτή τη διαδικασία αυτές εκτελούνται. Παρακάτω γίνεται φανερή η μεθοδολογία που χρησιμοποιήθηκε.

```
########################## NOTE: QUESTION  ##############################
# The velocities of the robot regarding the next subtarget should be
# computed. The known parameters are the robot pose [x,y,th] from
# robot_perception and the next_subtarget [x,y]. From these, you can
# compute the robot velocities for the vehicle to approach the target.
# Hint: Trigonometry is required

if self.subtargets and self.next_subtarget <= len(self.subtargets) - 1:
    st_x = self.subtargets[self.next_subtarget][0]
    st_y = self.subtargets[self.next_subtarget][1]

    fi = math.atan2(st_y-ry, st_x-rx)

    if abs(fi-theta) >= 0.1 and fi >= theta:
      linear = 0
      angular = 0.3
    elif abs(fi-theta) >= 0.1 and fi < theta:
      linear = 0
      angular = -0.3
    else:
      linear = 0.3
      angular = 0

########################## NOTE: QUESTION  ##############################
```

Ερώτημα 4ο

Στο 4ο ερώτημα ζητήθηκε να συνδυαστούν οι ταχύτητες που παίρνουμε από τον απλό obstacle avoidance αλγόριθμο και τον path planning που περιεγράφηκε προηγουμένως ώστε να δημιουργηθεί ένας καθολικότερος αλγόριθμος. Η μεθοδολογία που αναπτύχθηκε είναι αρκετά απλή αλλά και αποτελεσματική. Όπως γίνεται φανερό και παρακάτω σε περίπτωση που η γραμμική ταχύτητα του λέιζερ είναι διαφορετική από το μηδέν, πράγμα που σημαίνει ότι δεν υπάρχει εμπόδιο μπροστά, τότε παίρνουμε κανονικά τις ταχύτητες του path planning. Αντίθετα αν υπάρχει εμπόδιο, το ρομπότ κάνει πίσω.

```
    ############################## NOTE QUESTION ##############################
    # You must combine the two sets of speeds. You can use motor schema,
    # subsumption of whatever suits your better.

    if l_laser != 0:
      self.linear_velocity  = l_goal
      self.angular_velocity = a_goal
    else:
      self.linear_velocity  = -0.3*l_laser
      self.angular_velocity = 0*a_laser

    ###########################################################################
```

Σημαντικό είναι, επίσης, να σημειωθεί ότι για να καλύτερα αποτελέσματα έγινε τροποποίηση του αλγορίθμου που διαβάζει τις μετρήσεις του λέιζερ ώστε να πηγαίνει πιο κοντά σε εμπόδια.

```
if self.move_with_target == True:
  maxDistanceToObject = 0.1
else:
  maxDistanceToObject = 1

############################## NOTE QUESTION ##############################
# Check what laser_scan contains and create linear and angular speeds
# for obstacle avoidance

# Checks if there is an obstacle near the front of the robot
if min(scan[(len(scan)/2)-80:(len(scan)/2)+80]) > maxDistanceToObject:
  linear  = 1
  angular = 0
else:
  linear  = 0
  angular = 1

###########################################################################
```

Ερώτημα 5ο

Στο 5ο ερώτημα ζητήθηκε ένας αλγόριθμος για να ελέγχεται να το ρομπότ φτάσει σε κάποιο επόμενο υπο-στόχο χωρίς να περάσει από τον προηγούμενο και να συνεχίζεται η πλοήγηση από αυτό. Αυτό που υλοποιήθηκε φαίνεται παρακάτω. Ο αλγόριθμος είναι παρόμοιος με τον αρχικό μόνο που δεν τρέχει μόνο για τον επόμενο υπο-στόχο αλλά για όλους.

```
######################### NOTE: QUESTION  ##############################
# What if a later subtarget or the end has been reached before the
# next subtarget? Alter the code accordingly.
# Check if distance is less than 7 px (14 cm)

for index in range(self.next_subtarget, len(self.subtargets)):
  # Find the distance between the robot pose and the next subtarget
    dist = math.hypot(\
        rx - self.subtargets[index][0], \
        ry - self.subtargets[index][1])
    if dist < 5:
        self.next_subtarget = index+1
        self.counter_to_next_sub = self.count_limit
        # Check if the final subtarget has been approached
        if self.next_subtarget == len(self.subtargets):
            self.target_exists = False

########################################################################
```

Ερώτημα 6ο

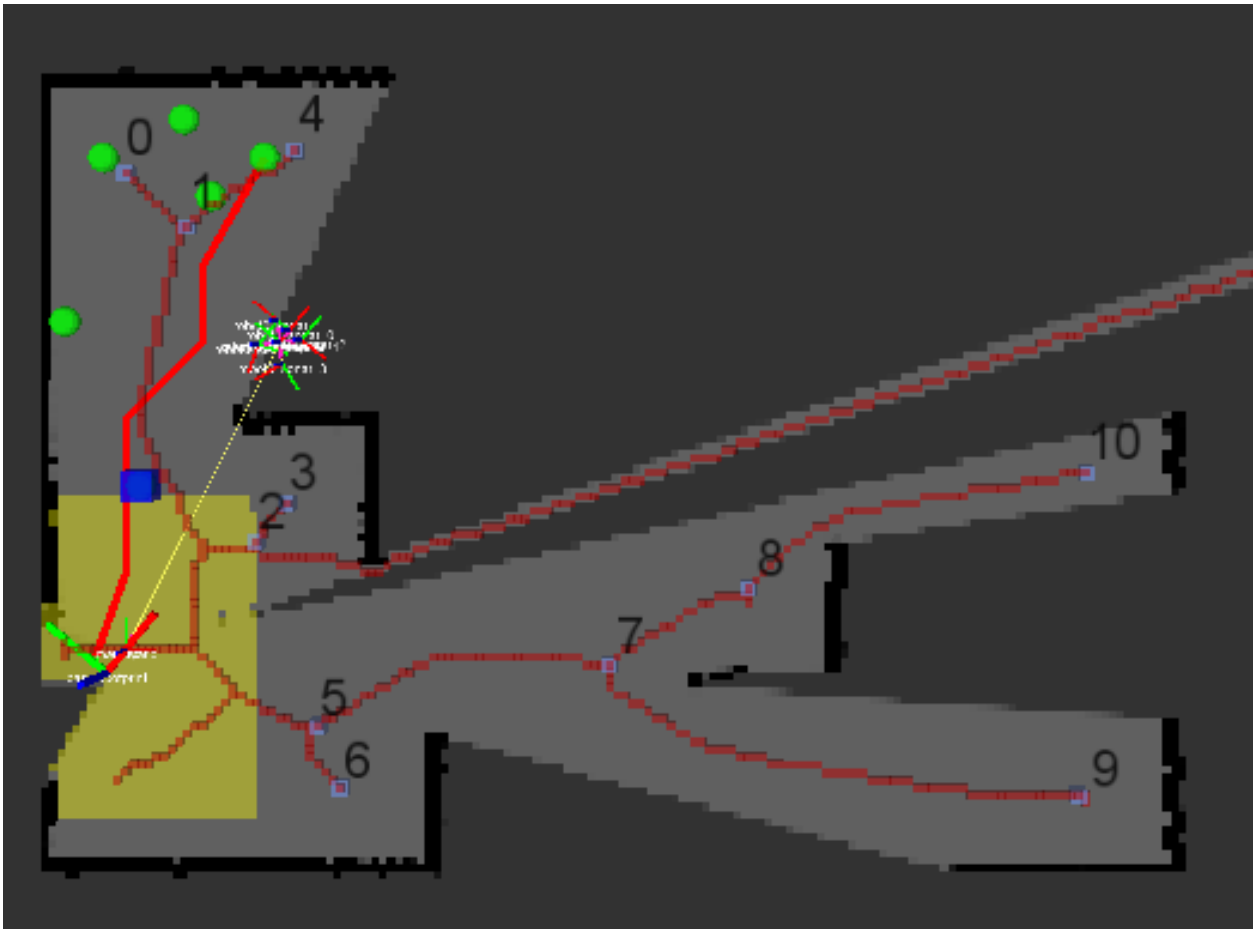
Στο επόμενο ερώτημα ζητήθηκε ένας εξυπνότερος αλγόριθμος για επιλογή στόχου. Η μεθοδολογία που αναπτύχθηκε εκμεταλλεύεται τον σκελετό που ήδη δημιουργείται και τα nodes που βρίσκονται στα άκρα του. Από αυτά διαλέγει αυτό με το μεγαλύτερο label, στο οποίο βέβαια το ρομπότ να μπορεί να πάει και το καθορίζει σαν στόχο. Για να εκτελεσθεί αυτός ο αλγόριθμος πρέπει στο yaml αρχείο να γίνει η αλλαγή στη μεταβλητή target_selector από 'random' σε 'furthest'. Ο κώδικας που υλοποιεί αυτό βρίσκεται παρακάτω.

```
    # Random point
    if self.method == 'random' or force_random == True:
      target = self.selectRandomTarget(ogm, coverage, brush, ogm_limits)

    # Furhest node-point
    if self.method == 'furthest':
      target = self.selectFurthestTarget(ogm, coverage, brush, ogm_limits, nodes)

    return target

def selectFurthestTarget(self, ogm, coverage, brushogm, ogm_limits, nodes):
  # The next target in pixels
    tinit = time.time()
    next_target = [0,0]
    for i in range(len(nodes)-1, 0, -1):
        x = nodes[i][0]
        y = nodes[i][1]
        if  ogm[x][y] < 50 and coverage[x][y] < 50 and \
          brushogm[x][y] > 10:
            next_target = [x,y]
            Print.art_print("Select furthest target time: " + str(time.time() - tinit), \
                Print.ORANGE)
            return next_target
    return self.selectRandomTarget(ogm, coverage, brushogm, ogm_limits)

########################################################################
```

Με αυτόν τον τρόπο έχουμε ένα σημαντικά καλύτερο αλγόριθμο από τον τυχαίο και οδηγεί το ρομπότ να εξερευνήσει τα πιο απομακρυσμένα σημεία της πίστας. Στο επόμενο παράδειγμα παρατίθεται ένα παράδειγμα εξερεύνησης χρησιμοποιώντας αυτόν τον αλγόριθμο.

Έξτρα Ερώτημα 1ο

Σε αυτό το ερώτημα ζητήθηκε ένας εναλλακτικός τρόπος path planning. Ο αλγόριθμος που υλοποιήθηκε είναι εντελώς custom made και φτιαγμένος ώστε να βελτιώσει το coverage ενώ να παραμείνει σχετικά smooth. Παίρνει σαν είσοδο το στίγμα του ρομπότ και αυτό του τελικού στόχου και τοποθετεί υποστόχους όπως φαίνεται στο ακόλουθο σχήμα.



Για να γίνει πιο ξεκάθαρο το πώς λειτουργεί, αυτό που κάνει είναι σπάει τη διαδρομή σε δύο τμήματα τοποθετώντας έναν υπο-στόχο με τετμημένη αυτή του ρομπότ και τεταμένη αυτή του στόχου (είναι η πράσινη βούλα κοντά στο 0). Τοποθετεί επίσης και εκατέρωθεν των δύο κάθετων ευθύγραμμων τμημάτων που σχηματίζει ο υπο-στόχος αυτός με την θέση του ρομπότ και με τη θέση του στόχου, έξτρα υποστόχους ώστε να έχουμε περισσότερα σημεία και άρα μεγαλύτερο coverage. Ο αλγόριθμος ακολουθεί.

```
######################### NOTE: QUESTION  ##############################
# The path is produced by an A* algorithm. This means that it is
# optimal in length but 1) not smooth and 2) length optimality
# may not be desired for coverage-based exploration

step = 1
n_subgoals = (int) (len(self.path)/step)
self.subtargets = []

[rx, ry] = [\
    self.robot_perception.robot_pose['x_px'] - \
            self.robot_perception.origin['x'] / self.robot_perception.resolution,\
    self.robot_perception.robot_pose['y_px'] - \
            self.robot_perception.origin['y'] / self.robot_perception.resolution\
            ]

st_x = self.path[-1][0]
st_y = self.path[-1][1]

offset = 3.5
if (abs(st_y - ry) >= 20):
    self.subtargets.append([rx - offset, ry + (st_y - ry)/3])
    self.subtargets.append([rx + offset, ry + 2*((st_y - ry)/3)])

if abs(st_y - ry) >= 5 and abs(st_x - rx) >= 5:
    self.subtargets.append([rx, st_y])

if (abs(st_x - rx) >= 20):
    self.subtargets.append([rx + (st_x - rx)/2, st_y - offset])
    self.subtargets.append([rx + 2*((st_x - rx)/3), st_y + offset])

self.subtargets.append(self.path[-1])

self.next_subtarget = 0
print "The path produced " + str(len(self.subtargets)) + " subgoals"

#########################################################################
```

Ένα παράδειγμα δημιουργίας χάρτη χρησιμοποιώντας για target selection τον αλγόριθμο του ερωτήματος 6 και για path planning τον αλγόριθμο του 1ο έξτρα ερωτήματος φαίνεται στην ακόλουθη εικόνα. Παρατηρούμε ότι πράγματι το ρομπότ κινείται στρατηγικά καλύπτοντας μεγάλο μέρος της πίστας και ταξιδεύοντας σε μέρη που δεν έχει πάει. Είναι όμως σημαντικό να σημειωθεί ότι δεν λαμβάνει υπόψιν του την ύπαρξη εμποδίων για την εύρεση του path planning και για αυτό μπορεί να δημιουργήσει μονοπάτι μέσα από εμπόδιο. Σε μια μελλοντική έκδοση αυτό θα έπρεπε να διορθωθεί.

Έξτρα Ερώτημα 2ο

Στο δεδομένο ερώτημα απαιτείται βελτιστοποίηση των αλγορίθμων ώστε αυτοί να τρέχουν σε λιγότερο χρόνο. Πράγματι για τον υπολογισμό του επόμενου στόχου και των υπο-στόχων η υπάρχουσα μεθοδολογία απαιτεί αρκετά δευτερόλεπτα καθώς εκτελεί διαδικασίες όπως το skeletonization. Αυτή είναι και η διαδικασία με την οποία ασχολείται η παρούσα εργασία. Το skeletonization, λοιπόν, απαιτεί σε μία τυπική εκτέλεση περίπου 3 με 4 δευτερόλεπτα. Πιο συγκεκριμένα το παρακάτω κομμάτι κώδικα, που βρίσκεται στο αρχείο topology.py απαιτεί 1.17 δευτερόλεπτα και εκτελείται για να υπολογιστεί ο σκελετός του χάρτη.

```
38      for i in range(0, width):
39        for j in range(0, height):
40          if ogm[i][j] < 49:
41            local[i][j] = 1
```

Καθώς στην ουσία είναι μία διπλή for-λούπα όπου ο υπολογισμός του κάθε στοιχείου της δεν απαιτεί τον υπολογισμό κάποιου άλλου, η πρώτη και ξεκάθαρη λύση είναι να παραλληλοποιηθεί. Με τη χρήση των σύγχρονων CPU θεωρητικά μπορούμε να έχουμε βελτίωση τουλάχιστον κατά 4 φορές αν μάλιστα είχαμε διαθέσιμη και GPU τότε αυτό το block θα μπορούσε να τρέξει σχεδόν στιγμιαία. Με αυτή τη σκέψη έγινε προσπάθεια παραλληλισμού με χρήση της βιβλιοθήκης της python joblib. Το αποτέλεσμα όμως δεν ήταν θετικό καθώς όπως αποδείχθηκε το thread που σηκώνει το topology.py δεν είναι το MainThread και για αυτό δεν έχει δικαίωμα (;) να σηκώσει και άλλα threads, ή τουλάχιστον όχι με τη συγκεκριμένη βιβλιοθήκη.

Η εναλλακτική μέθοδος που τελικά υλοποιήθηκε απαιτεί μεγαλύτερη κατανόηση της φυσικής σημασίας των δομών που επηρεάζονται από το μπλοκ αυτό του κώδικα. Ο πίνακας ogm[][] μπορεί να είναι μικρότερος από το 49 μόνο εντός της περιοχής που είναι κοντά στο ρομπότ και που εν πάση περίπτωσή αποτελεί την πίστα. Κατά τα άλλα ο ogm[][] έχει και αρκετά «σκουπίδια» πριν και μετά από αυτό το κομμάτι, τα οποία δεν υπάρχει λόγος να διαβάζουμε ένα προς ένα. Τελικά, η μέθοδος που χρησιμοποιήθηκε διαβάζει ανά ένα step τα στοιχεία του ogm[][] προσπαθώντας να βρει ένα από τα πρώτα σημεία στα οποία παρατηρείται τιμή μικρότερη του 49 και ένα από τα τελευταία σημεία. Στη συνέχεια διαβάζει αναλυτικά όλα τα σημεία εντός της περιοχής που αυτά τα δύο ορίζουν. Ο κώδικας ακολουθεί.

```
############################ Added code ###################################
  step = 50
  first = [0,0]
  last = [0,0]
  # Find the area in which there is propability of coverage
  for i in range(0, width, step):
    for j in range(0, height, step):
      if ogm[i][j] < 49:
        if first == [0,0]:
          first = [i,j]
        last = [i,j]

  # Search only in the area found
  for i in range (first[0]-step, last[0]+step):
    for j in range (first[1]-step, last[1]+step):
      if ogm[i][j] < 49:
        local[i][j] = 1

###########################################################################
```

Το αποτέλεσμα είναι να έχουμε εκτέλεση σε **0.12s** του συγκεκριμένου block κώδικα (ενώ παλιότερα έπαιρνε **1.17s**) και εκτέλεση σε **2.83s** ολόκληρου του skeletonization (ενώ με την προηγούμενη μέθοδο έπαιρνε **3.79s**). Οι χρόνοι αυτοί είναι μέσοι όροι πολλών διαδοχικών μετρήσεων. Είναι σημαντικό,

επίσης, να σημειωθεί ότι δεν υπάρχει καμία απώλεια πληροφορίας και ότι η λογική αυτή μπορεί να χρησιμοποιηθεί σε πολλές άλλες περιπτώσεις εντός του κώδικα καθώς υπάρχουν αρκετοί τέτοιοι πίνακες.

Έξτρα Ερώτημα 3ο

Έχει συμπεριληφθεί ένα βίντεο με το όνομα **Demo.mp4** στο εμφανίζονται τα σημαντικότερα βήματα εν ώρα εκτέλεσης. Επίσης, θα αφήσω εδώ μία πατάτα!