



# Ενσωματωμένα Συστήματα Μικροεπεξεργαστών

MILESTONE 2: AUTONOMOUS MAP EXPLORING CAR

Καλογεράκης Στέφανος | Ζερβάκης Αρης

## Εισαγωγή

Σε συνέχεια της υλοποίησης του πρότζεκτ εξαμήνου Autonomous Map Exploring Car με την χρήση της τεχνολογίας Lego Mindstorms στο δεύτερο milestone είχαμε να υλοποιήσουμε ακόμα ορισμένες λειτουργίες που μας οδηγούν ένα βήμα πιο κοντά στο τρίτο και τελευταίο milestone που οφείλουμε να παραδώσουμε όλες τις λειτουργίες που καθορίστηκαν στις αρχικές προδιαγραφές. Πιο συγκεκριμένα, στο δεύτερο milestone υλοποιήθηκε αρχικά μια προσέγγιση του αλγόριθμου που θα χρησιμοποιηθεί με χρήση της γλώσσας προγραμματισμού C. Ακόμα, υλοποιήθηκε και μια εφαρμογή σε Java που θα χρησιμοποιηθεί για την τελική απεικόνιση του λαβύρινθου που διέτρεξε το αυτοκίνητό μας. Τέλος, εξελίξαμε την λειτουργία του αυτόνομου αυτοκινήτου πραγματοποιώντας ελεύθερη πορεία πάνω σε μονοπάτι.

## Περιγραφή

Παρακάτω περιγράφονται αναλυτικά οι υλοποιήσεις που περιεγράφηκαν παραπάνω

### **Εξαντλητικός αλγόριθμος ανάλυσης λαβυρίνθου σε C**

Σε ένα πρώτο στάδιο, οφείλαμε να βρούμε ποιόν αλγόριθμο θα χρησιμοποιήσουμε προκειμένου να βεβαιωθούμε ότι το αυτοκίνητο μας θα διατρέξει εξαντλητικά όλο τον λαβύρινθο. Ο κατάλληλος αλγόριθμος για την συγκεκριμένη λειτουργία είναι ο **Depth-First Search** αλγόριθμο (θα αναγράφεται ως **DFS** στην συνέχεια της αναφοράς).

Στην υλοποίηση μας, προκειμένου να απλοποιηθεί το πρόβλημα ορίζουμε στατικά τον «λαβύρινθο» που υλοποιείται ο αλγόριθμος όπως φαίνεται παρακάτω σε ένα τυχαία ορισμένο λαβύρινθο.

```
int sampleMatrix[5][5] = {
    {3,2,2,2,2},
    {1,1,1,2,2},
    {2,1,2,2,2},
    {2,1,1,2,2},
    {2,2,2,2,1}
};
```

Οι διαφορετικοί αριθμοί που χρησιμοποιούνται συμβολίζουν διαφορετικές καταστάσεις ο καθένας. Πιο συγκεκριμένα:

Αριθμός	Χρήση-Συμβολισμός
1	Τοίχος
2	Επισκέψιμα σημεία-κόμβοι
3	Σημείο εκκίνησης

Κατά την εκκίνηση του κώδικα αυτό που εντοπίζεται αρχικά είναι το σημείο εκκίνησης βάσει του συμβολισμού που δόθηκε παραπάνω. Σε κάθε περίπτωση για την αποτελεσματική λειτουργία του προγράμματος οφείλουμε να έχουμε ορίσει σημείο εκκίνησης.

Στην συνέχεια, ακολουθεί η εκτέλεση του αναδρομικού DFS αλγόριθμου με τον κώδικα της εικόνας και ενημερώνεται με τις κατάλληλες τιμές ο αρχικός μας πίνακας.

```
if (sampleMatrix[row][col] == 2 || sampleMatrix[row][col] == 3) {
    sampleMatrix[row][col] = 7;

    if (dfs(row - 1, col)){
        sampleMatrix[row][col] = 7;
        return 1;
    }

    if (dfs(row, col - 1)){
        sampleMatrix[row][col] = 7;
        return 1;
    }

    if (dfs(row + 1, col)){
        sampleMatrix[row][col] = 7;
        return 1;
    }

    if (dfs(row, col + 1)){
        sampleMatrix[row][col] = 7;
        return 1;
    }
}
```

Στην δική μας υλοποίηση η προτεραιότητα κόμβων προς επίσκεψη κατά σειρά είναι **ΠΑΝΩ, ΑΡΙΣΤΕΡΑ, ΚΑΤΩ, ΔΕΞΙΑ**. Το πρόγραμμα επομένως εξετάζει τους γειτονικούς κόμβους και εφόσον κάποιος είναι επισκέψιμος (αν κάποιος κόμβος έχει ήδη επισκεφτεί δεν θεωρείται επισκέψιμος) ανάλογα με την παραπάνω αλληλουχία προτεραιοτήτων αποφασίζει ποιον κόμβο θα επισκεφτεί στην συνέχεια. Αν βρεθούμε σε σημείο τερματισμού δεν προχωράμε σε επόμενο κόμβο και σταματάει ο αλγόριθμος.

Παρακάτω επιδεικνύουμε τις λειτουργίες όπως παρουσιάστηκαν με ένα παράδειγμα εκτέλεσης του κώδικα μας.

3	2	2	2	2
1	1	1	2	2
2	1	2	2	2
2	1	1	9	2
2	2	2	2	1

7	7	7	7	2
1	1	1	7	2
2	1	7	7	2
2	1	1	9	2
2	2	2	2	1

Στην αριστερή εικόνα φαίνεται ο αρχικός μας λαβύρινθος, ενώ δεξιά είναι ο λαβύρινθος μετά από την εκτέλεση και τις αλλαγές. Αποδεικνύουμε με αυτόν τον τρόπο, ότι όλοι οι παραπάνω ισχυρισμοί ισχύουν.

Τέλος, σε ένα ακόμα παράδειγμα ορθής λειτουργίας αυτή την φορά χωρίς την χρήση κάπου τερματικού κόμβου, παρατηρούμε ότι ο αλγόριθμος λειτουργεί εξαντλητικά όπως ήταν το ζητούμενο.

3	2	2	2	2
1	1	1	2	2
2	1	2	2	2
2	1	1	9	2
2	2	2	2	1

7	7	7	7	7
1	1	1	7	7
7	1	7	7	7
7	1	1	7	7
7	7	7	7	1

Το συγκεκριμένο πρόγραμμα σε C είναι κυρίως για σκοπούς δοκιμαστικούς πριν μεταβούμε στην ROBOTC. Για αυτό τον λόγο, **δεν υπάρχουν πολλοί έλεγχοι εγκυρότητας και συνεπώς ανοχή σε σφάλματα του χρήστη**. Με την καθοδήγηση όμως, από τα σχόλια που βρίσκονται στον πηγαίο κώδικα μπορεί ο χρήστης να πειραματιστεί σε διαφορετικούς και τυχαίους λαβυρίνθους αυξομειώνοντας και το μέγεθος του στατικού πίνακα


### **Εφαρμογή απεικόνισης λαβύρινθου σε Java**

Στα πλαίσια του πρότζεκτ, κυρίως για λόγους πληρότητας και σωστότερης απεικόνισης του τελικού αποτελέσματος δημιουργήσαμε μια εφαρμογή σε γλώσσα προγραμματισμού **Java** όπου απεικονίζεται γραφικά η διαδρομή που πραγματοποίησε το αυτοκίνητο.

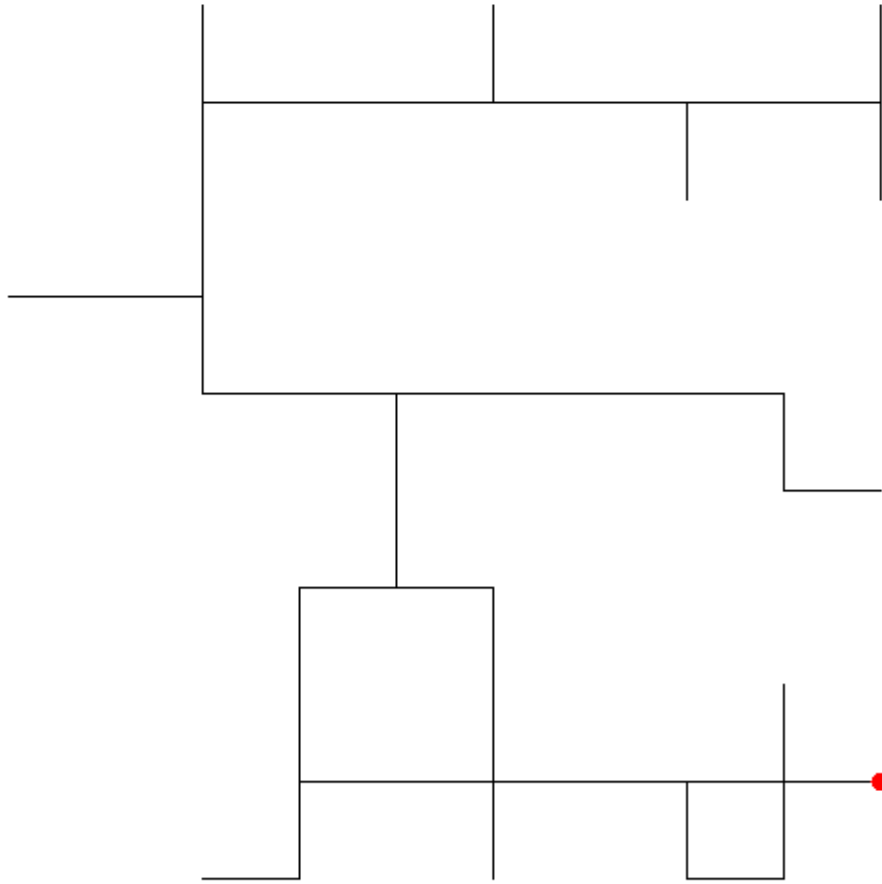
Το πρόγραμμα δέχεται ως παράμετρο ένα αρχείο (το έτοιμο αρχείο που δίνεται για δοκιμές είναι το data.txt) το οποίο περιέχει αριθμητικές τιμές ανάλογα με την διαδρομή που έχει πραγματοποιήσει το αυτοκίνητο. Συγκεκριμένα, θεωρούμε ότι οι τιμή 1 δηλώνουν ότι το αυτοκίνητο πέρασε από την συγκεκριμένη διαδρομή και ζωγραφίζονται ενώ η τιμή 0 δηλώνει ότι ένας κόμβος δεν επισκέφτηκε και δεν ζωγραφίζονται. Τέλος, η τιμή 2 υποδηλώνει τερματικό σημείο και συμβολίζεται με κόκκινη κουκίδα.

Αξίζει να επισημάνουμε ότι στο τελευταίο milestone, που θα γίνουν οι πραγματικές δοκιμές στην εφαρμογή ενδέχεται να υπάρξουν κάποιες μικρές αλλαγές στα δεδομένα που δέχεται η εφαρμογή σαν είσοδο χωρίς να επηρεαστεί η λογική που περιγράφεται.

Στο αρχείο data.txt έχει δοθεί ένας ενδεικτικός λαβύρινθος με το παράθυρο που προκύπτει να εμφανίζεται στην συνέχεια

 Maze Visualizer

— □ ×



Η υλοποίηση βασίστηκε σε λειτουργίες και βιβλιοθήκες της Java κατάλληλες για ζωγραφική και απεικόνιση δεδομένων ενώ αναλυτική υλοποίηση με σχόλια υπάρχει στον πηγαίο κώδικα που παραδίδεται με την αναφορά.

Το αρχείο που θα δίνεται σαν είσοδος στο πρόγραμμα πρέπει να ακολουθεί ορισμένους **κανόνες για την αποτελεσματική λειτουργία**. Τέτοιοι είναι:

- Τα δεδομένα πρέπει να είναι όπως το format που δίνεται στο ενδεικτικό αρχείο παραδείγματος, δηλαδή σε μορφή ενός δισδιάστατου πίνακα
- Τα δεδομένα σε μορφή πίνακα πρέπει να είναι συμμετρική. Δηλαδή οι πίνακες που υποστηρίζονται σωστά είναι  $n \times n$  όπου  $n$  ο αριθμός κόμβων

- Οι τιμές που επιτρέπονται είναι μόνο θετικές ακέραιες, ειδάλλως εμφανίζεται μήνυμα σφάλματος
- Μεταξύ των τιμών πρέπει να υπάρχει διαχωριστικό(delimiter) ο χαρακτήρας κόμμα(,)
- **Οι συμμετρικοί πίνακες που υποστηρίζονται πρέπει να είναι στην μεγαλύτερη περίπτωση μέχρι 20\*20.** Η συγκεκριμένη ρύθμιση μπορεί να τροποποιηθεί μέσα από περιβάλλον εκτέλεσης κώδικας java. Σε περίπτωση πίνακα μεγαλύτερου πίνακα από 20\*20 εμφανίζεται απλώς ένα κενό παράθυρο
- Σε περίπτωση που δεν δοθεί αρχείο εισόδου δεν πραγματοποιείται εκτέλεση του προγράμματος και εμφανίζεται κατάλληλο μήνυμα λάθους

Ο κώδικας δοκιμάζεται με την εκτέλεση ενός .jar αρχείου με όνομα MazeVisualizer.jar. Προκειμένου να δοκιμαστεί το πρόγραμμα πρέπει να είναι εγκατεστημένη η java στον υπολογιστή του χρήστη σε έκδοση πιο καινούργια από αυτή που έγινε το build του jar αρχείου. Στο αρχείο **requirements.txt** υπάρχουν λεπτομέρειες αναφορικά με την έκδοση java που πρέπει να είναι εγκατεστημένη στον εκάστοτε υπολογιστή.

Εκτός από την εντολή java -jar που απαιτείται για την εκτέλεση τέτοιου τύπου αρχείων απαιτείται και ένα όρισμα το οποίο θα απευθύνεται στο **full path** που βρίσκεται το αρχείο εισόδου. Ακολουθεί παράδειγμα εκτέλεσης του προγράμματος από την γραμμή εντολών των Windows. Ο κώδικας εκτελείται με αντίστοιχο τρόπο και σε εκδόσεις των Linux

```
java -jar MazeVisualizer.jar
```

```
D:\TUC_PROJECT\TUC_Autonomous_Map_Exploring_Car\MazeVisualizer\data.txt
```

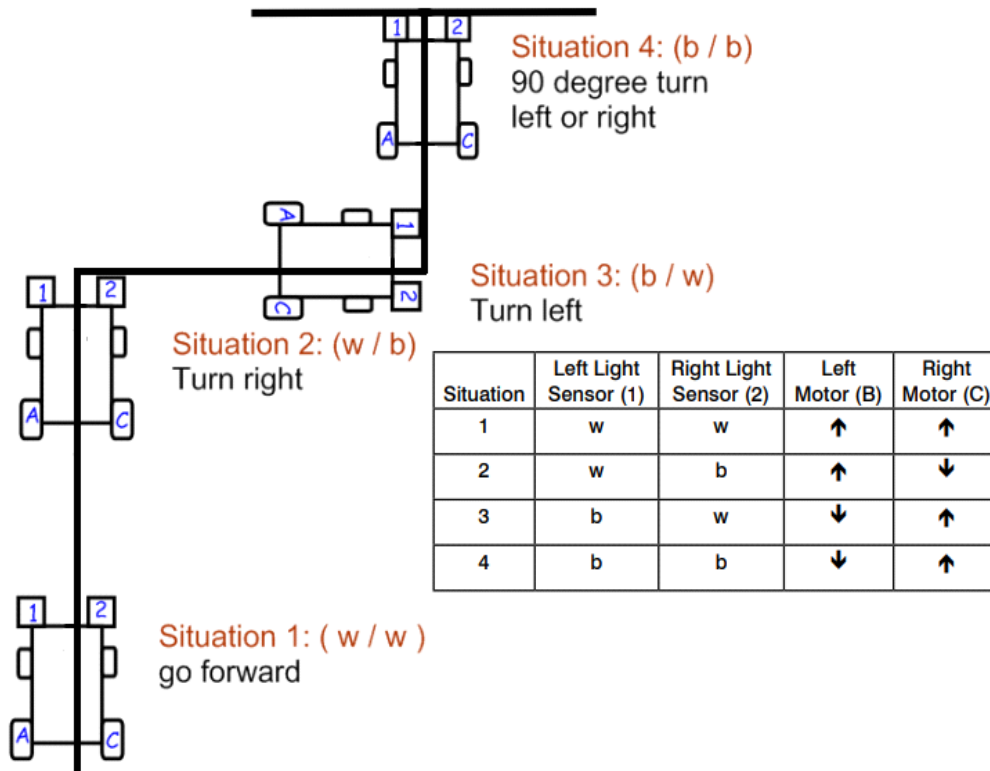
Συνοψίζοντας σε μια γενική μορφή θα πρέπει να ισχύει

```
java -jar <JAR NAME> <FULL PATH INPUT FILE>
```

## Κώδικας RobotC για περιφορά αυτοκινήτου σε τυχαίο λαβύρινθο

Για την περιφορά στο λαβύρινθο είχαμε στη διάθεσή μας δεδομένα τα οποία λαμβάναμε από 2 Light Sensors και ένα RGB Sensor.

Η κύρια ιδέα πίσω από το line following είναι να γίνεται contain η μαύρη γραμμή ανάμεσα στους δύο Light Sensors ενώ ο RGB Sensor μας δίνει απαραίτητη πληροφορία για το επόμενο μας βήμα, αν δηλαδή το επόμενο block είναι μαύρο (διάδρομος), άσπρο (κενό) ή κόκκινο (συνθήκη τερματισμού).



W: Ένδειξη Sensor για λευκό

B: Ένδειξη Sensor για μαύρο

Με πειραματισμό παρατηρήσαμε ότι οι 2 Light Sensors δεν βγάζουν τις ίδιες μετρήσεις, αναγκάζοντας μας να βγάλουμε το ανάλογο threshold για τον καθένα, δηλαδή το όριο της μέτρησης κάτω από το οποίο βλέπει το μαύρο χρώμα του διαδρόμου και πάνω από αυτό το άσπρο της πίστας, με δοκιμές.

Για το milestone επιλέξαμε την ακόλουθη συμπεριφορά:

- Αν το αυτοκίνητο συναντήσει κάποια διακλάδωση αλλά μπορεί να συνεχίσει ευθεία, θα συνεχίζει πάντα ευθεία.
- Αν το αυτοκίνητο δεν έχει την επιλογή να κινηθεί ευθεία αλλά είναι δυνατές και η αριστερή και η δεξιά στροφή, το αυτοκίνητο την πρώτη φορά στρίβει δεξιά και στην επόμενη ανάλογη επιλογή αριστερά συνεχίζοντας εναλλάξ.

- Αν η επιλογή είναι μοναδική, η γραμμή στρίβει δηλαδή μόνο δεξιά, μόνο αριστερά ή συνεχίζει ευθεία, το αυτοκίνητο συνεχίζει κανονικά την κίνηση πάνω στο μονοπάτι.
- Αν το αυτοκίνητο συναντήσει κάποιο κόκκινο σημείο τερματισμού, σταματάει.

Προφανώς οι δύο πρώτες επιλογές συμπεριφοράς είναι τυχαίες, και στη θέση τους θα μπορούσε να πραγματοποιηθεί οποιαδήποτε απόφαση.

**Για τη λειτουργία του αυτοκινήτου υπάρχει και ανάλογο Demo Video το οποίο υπάρχει στα παραδοτέα.**

### **Προβλήματα που αντιμετωπίσαμε**

Αρχικά η διαδικασία του line following, είχε υλοποιηθεί με διαφορετικό τρόπο. Πιο συγκεκριμένα χρησιμοποιήσαμε τον RGB Sensor για να την εντοπίσουμε και σε περίπτωση που την χάναμε, ορίζαμε ένα τόξο ώστε να κινηθούμε δεξιά και αριστερά ώστε να την επανεντοπίσουμε. Έτσι η μόνη χρήση των Light Sensors ήταν για τον εντοπισμό των διακλαδώσεων.

Τελευταία στιγμή ωστόσο, αντιμετωπίσαμε σοβαρό πρόβλημα με τον RGB Sensor, καθώς δεν μας έδινε συνεχόμενα τη μέτρηση την οποία λάμβανε με αποτέλεσμα ο υλοποιημένος τρόπος να αποτυγχάνει. Σε επικοινωνία και συζήτηση με το διδάσκοντα καταλήξαμε ότι το πρόβλημα αυτό μπορεί να λυθεί με τη λογική του polling, ζητώντας από τον RGB Sensor μια μέτρηση ανά τακτά χρονικά διαστήματα αντί για τον interrupt-driven τρόπο. Παρόλα αυτά επιλέξαμε και πάλι τον τρόπο του line following με τους 2 Light Sensors λόγω απόδοσης και ευκολίας παραμετροποίησης. Τρόπο φυσικά που σκεφτήκαμε και υλοποιήσαμε αναγκαστικά λόγω του προβλήματος που συναντήσαμε.



## Βιβλιογραφία

Forum με καθοδήγηση για προβλήματα που συναντήσαμε σε ROBOTC:

<http://www.robotc.net/forums/viewforum.php?f=1&sid=6decc57fb332bd1c7039d8a0840df4bb>

DFS: [https://en.wikipedia.org/wiki/Depth-first\\_search](https://en.wikipedia.org/wiki/Depth-first_search)

DFS maze search: [https://www.algosome.com/articles/maze-generation-depth-first.html?fbclid=IwAR0t53lVX8ygkHFH\\_KYkIxzBFZfNnL0po1lYU6OQIy-0GnZrLu4y6wVPa2k](https://www.algosome.com/articles/maze-generation-depth-first.html?fbclid=IwAR0t53lVX8ygkHFH_KYkIxzBFZfNnL0po1lYU6OQIy-0GnZrLu4y6wVPa2k)