ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Πλοήγηση Πράκτορα Σε Δισδιάστατο Χώρο

Διπλωματική Εργασία

Φωτιάδης-Κρικέλης Αθανάσιος

Επιβλέπων: Δασκαλοπούλου Ασπασία

Βόλος έτος 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Πλοήγηση Πράκτορα Σε Δισδιάστατο Χώρο

Διπλωματική Εργασία

Φωτιάδης-Κρικέλης Αθανάσιος

Επιβλέπων: Δασκαλοπούλου Ασπασία

Βόλος έτος 2020

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

AGENT NAVIGATION IN TWO DIMENSIONAL GRID

Diploma Thesis

Fotiadis-Krikelis Athanasios

Supervisor: Daskalopulu Aspassia

Volos year 2020

Ευχαριστίες

Με αυτήν εργασία ολοκληρώνω τον κύκλο σπουδών μου κατά την διάρκεια των οποίων έμαθα πολλά πράγματα και θα ήθελα να ευχαριστήσω τους ανθρώπους που ήταν κοντά μου όλα αυτά τα χρόνια.

Αρχικά θα ήθελα να ευχαριστήσω τους καθηγητές του τμήματος αυτού για τις πολύτιμες γνώσεις καθώς και συμβουλές που μου προσέφεραν.

Ευχαριστώ ιδιαίτερα την κυρία Δασκαλοπούλου που χωρίς την βοήθεια της και καθοδήγηση δεν θα ήταν δυνατόν να ολοκληρωθεί αυτήν η εργασία.

Ευχαριστώ όλους τους φίλους και συναδέλφους για την συμπαράσταση τους στα δύσκολα και τέλος ευχαριστώ τους δικούς μου ανθρώπους και την οικογένεια μου για την απεριόριστη στήριξη, αγάπη και υπομονή που έδειξαν όλα αυτά τα χρόνια.

Σας ευχαριστώ αυτή εργασία είναι για εσάς.

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

«Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής».

Ο Δηλών

(Υπογραφή)
Φωτιάδης-Κρικέλης Αθανάσιος
Ημερομηνία 3/3/2020

Περίληψη

Το πιο σημαντικό πρόβλημα που καλείται να αντιμετωπίσει ένας πράκτορας, προκειμένου να κινηθεί σε ένα χώρο, είναι αυτό της εύρεσης του μονοπατιού με χαμηλότερο κόστος. Εάν τα κόστη κατά τη διάρκεια της διάσχισης του μονοπατιού αλλάξουν, τότε ίσως χρειαστεί το υπόλοιπο μονοπάτι να ξανασχεδιαστεί. Αυτό συμβαίνει στην περίπτωση ενός πράκτορα , ο οποίος διαθέτει έναν αισθητήρα και περιορισμένη(μη απόλυτη) γνώση του περιβάλλοντός του.

Όσο ο πράκτορας λαμβάνει πρόσθετη πληροφορία μέσω του αισθητήρα του μπορεί να αναθεωρήσει το σχέδιό του, να μειώσει το συνολικό κόστος της διαδρομής. Αυτή η εργασία, παρουσιάζει μια συνολική εικόνα των αλγορίθμων που χρησιμοποιούνται συχνότερα(Dijkstra ,A* , D*,LPA* , D* Lite) για την εύρεση μονοπατιού με χαμηλότερο κόστος , τόσο σε μερικώς γνωστό περιβάλλον , όσο και σε εντελώς άγνωστο.

Επιπλέον ,στη θέση αυτή παρουσιάζεται μία υλοποίηση ενός πράκτορα – ρομπότ, ο οποίος χρησιμοποιεί έναν από τους παραπάνω αλγορίθμους γραμμένο στην γλώσσα προγραμματισμού Python και την πλατφόρμα Raspberry Pi.

Abstract

The most important problem that an agent needs to solve in order to navigate through space is that of planning a path with the lowest cost. Sometimes the arc costs can change while traversing through a path and the remaining course must be recalculated.

That's usually the case of an agent who is equipped with a sensor and limited knowledge of the environment. This thesis, presents and overview of the most used algorithms such as (Dijkstra, A*, D*, LPA*, D* Lite) in order to find a path at the lowest cost, in a known, partially known, or completely unknown environment.

Moreover, this thesis presents an implementation of a robot-agent, who uses one of the above algorithms, written in Python and uses the Raspberry Pi microcontroller.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Η πλοήγηση ενός αυτόνομου συστήματος αποτελεί σημαντικό κομμάτι της επιστήμης της Ρομποτικής και έχει μελετηθεί πολύ και με εκτενή τρόπο. Στην πράξη το πρόβλημα που καλούνται να λύσουν τόσο οι μελετητές, όσο και οι μεγάλες βιομηχανίες, είναι αυτό της εύρεσης του συντομότερου μονοπατιού μεταξύ ενός σημείου εκκίνησης και ενός σημείου τερματισμού (στόχου). Έχουν αναπτυχθεί κατά καιρούς διάφοροι αλγόριθμοι που επιλύουν αυτό το ζήτημα, ανάμεσά τους βρίσκονται οι αλγόριθμοι Dijkstra [1], A*[2] , D*[3] , LPA*[4] και D* Lite[5]. Όλοι αυτοί ανήκουν στην κατηγορία των αλγόριθμων Path Planning.

1.1 Αλγόριθμοι Path Planning

Οι αλγόριθμοι Path Planning αποτελούνται από τον σχεδιαστή(planner) ο οποίος δεν κάνει τίποτε άλλο παρά να δημιουργεί ένα πλάνο , δηλαδή μια ακολουθία από ενέργειες που πρέπει να εκτελεστούν. Μπορεί να είναι άνθρωπος ή μηχανή. Εάν είναι μηχανή, τότε είναι αλγόριθμος σύμφωνα με το μοντέλο Church-Turing.

1.2 Σχετικά με τα πλάνα ενεργειών.

Μόλις καταστρωθεί ένα πλάνο υπάρχουν τρεις τρόποι με τους μπορεί κάποιος να το χρησιμοποιήσει.

1. **Εκτέλεση:** Να το εκτελέσει είτε σε προσομοιωτή είτε σε κάποια μηχανική συσκευή(ρομπότ) συνδεδεμένο με τον φυσικό κόσμο.
2. **Βελτιστοποίηση:** Να το βελτιώσει σε ένα καλύτερο πλάνο.
3. **Ενσωμάτωση:** Να το ενσωματώσει ως ενέργεια σε ένα πλάνο υψηλότερου επιπέδου.

1.3 Μαθηματική περιγραφή του προβλήματος.

Για να περιγράψουμε το πρόβλημά μας ,το οποίο είναι η μετακίνηση του ρομπότ σε ένα δισδιάστατο πλέγμα θα χρησιμοποιήσουμε το μοντέλο του διακριτού εφικτού σχεδιασμού (Discrete feasible planning)[1]. Η βασική ιδέα είναι ότι κάθε ξεχωριστή περίπτωση στον κόσμο ονομάζεται κατάσταση , αντιπροσωπεύεται με το γράμμα χ, και

το σύνολο όλων αυτών των πιθανών καταστάσεων ονομάζεται χώρος καταστάσεων τον οποίο συμβολίζουμε με το γράμμα X. Επιπλέον είναι απαραίτητο το σύνολο των καταστάσεων να είναι μετρήσιμο καθώς και να είναι αρκετά μεγάλο ώστε να παρέχει όλες τις σχετικές πληροφορίες που αφορούν την λύση του προβλήματος.

Ο κόσμος μπορεί να φτάσει στην επιθυμητή κατάσταση μέσω της εφαρμογής μιας σειράς ενεργειών που επιλέγονται από τον σχεδιαστή. Κάθε ενέργεια όταν εφαρμόζεται σε μια δεδομένη κατάσταση x , παράγει μια άλλη κατάσταση x' όπως αυτή ορίζεται μέσω μιας συνάρτησης μετάβασης f. Η εξίσωση αυτή ορίζεται x' = f(x,u). Έστω U(x) ο συμβολισμός για τον χώρο ενεργειών για κάθε κατάσταση x , ο οποίος χώρος περιλαμβάνει κάθε πιθανή ενέργεια που μπορεί να πραγματοποιηθεί από την κατάσταση x. Επειδή τα x,x' $\in X$, $\tau\alpha\ U(x)\ \kappa\alpha\iota\ U(x')$ δεν είναι απαραίτητα ξένα μεταξύ τους , καθώς η ίδια ενέργεια μπορεί να εφαρμοστεί σε περισσότερες από μια καταστάσεις, επομένως ως χώρος όλων των πιθανών ενεργειών που μπορούν να συμβούν από την κατάσταση x ορίζεται ως η ένωση $U = \bigcup_{x\in X} U(x)$.Τέλος, για να ολοκληρωθεί η διατύπωση τους προβλήματος πρέπει να οριστεί ένα σύνολο $X_G \subset X$ το οποίο περιλαμβάνει όλες τις καταστάσεις–στόχους. Αυτό που πρέπει να κάνει τώρα ένας αλγόριθμος planning είναι να βρει μια ακολουθία από ενέργειες , οι οποίες εάν εφαρμοστούν , θα μετατρέψουν την αρχική κατάσταση x_I σε μια κατάσταση του X_G.

Έτσι το μοντέλο συνοψίζεται ως εξής:

1. Ένας μη κενός χώρος καταστάσεων X που όμως είναι πεπερασμένος ή διαθέτει μετρήσιμο αριθμό καταστάσεων.
2. Για κάθε κατάσταση $x \in X$ υπάρχει ένα σύνολο ενεργειών U(x).
3. Μία συνάρτηση μετάβασης f , η οποία παράγει μια κατάσταση x' $f(x,u) \in X$ για κάθε $x \in X\ \kappa\alpha\iota\ u \in U(x)$. Η εξίσωση μετάβασης προκύπτει από τη f ως $x' = f(x,u)$.
4. Μια αρχική κατάσταση $x_I \in X$
5. Ένα σύνολο στόχων $X_G \subset X$.

Συνήθως , είναι βολικό να εκφράζεται το μοντέλο ως ένα κατευθυνόμενο γράφημα μετάβασης καταστάσεων. Το σύνολο των κορυφών αποτελούν τον χώρο καταστάσεων X, ενώ μια ακμή από το $x \in X$ στο $x' \in X$ υπάρχει στο γράφημα εάν και μόνον αν υπάρχει

ενέργεια $u \in U(x)$ τέτοια ώστε να ισχύει $x' = f(x,u)$. Η αρχική κατάσταση καθώς και η κατάσταση – στόχος θεωρούνται ειδικές κορυφές . Πιο συγκεκριμένα , στο δικό μας πρόβλημα αρχικά υποθέτουμε ότι ένας πράκτορας-ρομπότ κινείται σε ένα δισδιάστατο πλέγμα το οποίο έχει συντεταγμένες της μορφής (i,j). Το ρομπότ κάνει διακεκριμένα βήματα σε μια από τις οκτώ κατευθύνσεις (επάνω , κάτω, αριστερά, δεξιά, επάνω διαγώνια δεξιά, επάνω διαγώνια αριστερά, κάτω διαγώνια δεξιά, κάτω διαγώνια αριστερά). Η μοντελοποίηση αυτού μπορεί να γίνει ως εξής:

Έστω X το σύνολο των καταστάσεων της μορφής (i,j) όπου τα $i,j \in Z$ με Z το σύνολο των ακεραίων. Έστω

$U = \{(0,1),(0,-1),(1,0),(-1,0),(1,1),(-1,1),(1,-1),(-1,-1)\}$. Το σύνολο των διακριτών βημάτων. Έστω $U(x) = U\ για\ κάθε\ x \in X\ το\ σύνολο\ των\ εργειών$. Η εξίσωση μετάβασης είναι $f(x,u) = x + u$ όπου $x \in X\ και\ u \in U$ θεωρούνται δισδιάστατα διανύσματα. Παραδείγματος χάριν, εάν $x = (3,4)\ και\ u = (0,1), τότε\ f(x,u) = (3,5)$. Υποθέτοντας ότι η αρχική κατάσταση X_I = (0,0) και $X_G = \{(100,100)\}$, τότε είναι πολύ εύκολο να βρούμε μια σειρά ενεργειών που αν εφαρμοστούν θα αλλάξουν την κατάσταση από (0,0) σε (100,100). Το πρόβλημα γίνεται ποιο ενδιαφέρον ένα θεωρήσουμε ότι σε κάποια από τα τετράγωνα του πλέγματος υπάρχουν εμπόδια που πρέπει να αποφύγει το ρομπότ.

Η εργασία αυτή , επικεντρώνεται κυρίως σε χώρους καταστάσεων οι οποίοι περιέχουν εμπόδια καθώς και στους συνήθεις αλγορίθμους επίλυσης τέτοιων προβλημάτων.

Κεφάλαιο 2

Αναζητώντας εφικτά σύνολα ενεργειών

2.1 Εισαγωγή

Σε αυτό το κεφάλαιο παρουσιάζονται κάποιες μέθοδοι , που δεν είναι τίποτε άλλο από αναζητήσεις γράφων, οι οποίες όμως φανερώνουν τις μεταβάσεις καταστάσεων με έναν σταδιακά αυξανόμενο τρόπο εφαρμογής ενεργειών αντί να είναι προκαθορισμένες. Σημαντική προϋπόθεση για τις μεθόδους αυτές είναι να είναι συστηματικές. Εάν το γράφημα είναι πεπερασμένο, αυτό σημαίνει ότι ο αλγόριθμος θα επισκεφθεί όλες τις απαραίτητες για την λύση καταστάσεις και έτσι θα μπορέσει να δηλώσει σε πεπερασμένο χρόνο ένα υπάρχει όντως κάποια λύση ή όχι. Για να είναι συστηματικός ένας αλγόριθμος θα πρέπει να παρακολουθεί ποιες καταστάσεις έχει επισκεφθεί , αλλιώς υπάρχει πιθανότητα να εγκλωβιστεί και να μην τερματίσει ποτέ. Σε περίπτωση που το γράφημα είναι άπειρο , τότε για να θεωρηθεί ο αλγόριθμος συστηματικός θα πρέπει να βρίσκει την λύση σε πεπερασμένο χρόνο.

2.2 Αλγόριθμος του Dijkstra

Ένας από τους πρώτους αλγορίθμους που χρησιμοποιήθηκαν για την εύρεση βέλτιστης ακολουθίας ενεργειών είναι ο αλγόριθμος του Dijkstra[2], ο οποίος βρίσκει το συντομότερο μονοπάτι μιας πηγής. Αποτελεί μια ειδική περίπτωση δυναμικού προγραμματισμού καθώς και συστηματική αναζήτηση.

Εάν υποθέσουμε ότι κάθε ακμή $e \in E$, στην γραφική αναπαράσταση διακριτού προβλήματος σχεδιασμού, διαθέτει ένα μη αρνητικό κόστος $l(e)$, το οποίο είναι το κόστος εφαρμογής ενέργειας. Χρησιμοποιώντας τη σημειογραφία για τον χώρο καταστάσεων το κόστος μπορεί να γραφεί στην μορφή $l(x,u)$, δηλώνοντας έτσι ότι κοστίζει $l(x,u)$ η μετάβαση από την κατάσταση x χρησιμοποιώντας την ενέργεια u. Το συνολικό κόστος ενός σχεδίου ανέρχεται στην πρόσθεση των κοστών , όλων των ακμών που βρίσκονται στο μονοπάτι. Η ουρά προτεραιότητας, Q , θα ταξινομείται σύμφωνα με μια συνάρτηση C : $X \to (0,\infty)$ η οποία είναι το κόστος για να έρθει ο κόσμος από την αρχική κατάσταση x_i. Για κάθε κατάσταση x , η τιμή $C^*(x)$ δηλώνει το βέλτιστο κόστος, το οποίο το παίρνουμε προσθέτοντας όλα τα κόστη των ακμών, $l(e)$, από όλα τα δυνατά μονοπάτια μεταξύ της αρχικής κατάστασης x_i και της x, και χρησιμοποιώντας το μονοπάτι με το μικρότερο

συνολικό κόστος. Σε περίπτωση που το κόστος δεν είναι το βέλτιστο το γράφουμε απλά $C(x)$. Σε κάθε περίπτωση το κόστος υπολογίζεται αυξητικά , κατά την εκτέλεση του αλγορίθμου. Αρχικά $C^*(x_i) = 0$. Κάθε φορά που μια νέα κατάσταση x' δημιουργείται , υπολογίζεται ένα κόστος $C(x') = C^*(x) + l(e)$, όπου e , είναι η ακμή που συνδέει το x με το x' η αλλιώς μπορούμε να γράψουμε $C(x') = C^*(x) + l(x, u)$. Δεν το καταχωρούμε ως βέλτιστο επειδή δεν το γνωρίζουμε στην παρούσα φάση, διότι εάν η x' υπάρχει στην ουρά Q, τότε είναι πολύ πιθανόν να βρούμε ένα νέο μονοπάτι προς την x' που να είναι πιο αποδοτικό. Αν ισχύει κάτι τέτοιο, τότε το κόστος πρέπει να μειωθεί για την x' και να γίνει κατάλληλη αναδιοργάνωση στην ουρά Q.

Η πολυπλοκότητα του χρόνου εκτέλεσης για τον αλγόριθμο αυτό είναι $O(|V|\log|V| + |E|)$, όπου |V| και |E|είναι ο αριθμός των κορυφών και των ακμών αντίστοιχα, στο γράφημα αναπαράστασης του διακριτού προβλήματος[2].

2.2 Αλγόριθμος A*

Ο αλγόριθμος A *[3], είναι αλγόριθμος αναζήτησης και αποτελεί μια επέκταση του αλγορίθμου του Dijkstra [2]. Κύριο χαρακτηριστικό του είναι το ότι προσπαθεί να ελαττώσει τον συνολικό αριθμό των καταστάσεων που ο αλγόριθμος επισκέπτεται χρησιμοποιώντας μια ευρετική εκτίμηση του κόστους μετάβασης από μια κατάσταση στην κατάσταση- στόχο.

Έστω $C(x)$ δηλώνει το κόστος που απαιτείται για την μετάβαση από την αρχική κατάσταση X_I στην κατάσταση X , και έστω $G(x)$ δηλώνει το κόστος που απαιτείται για την μετάβαση από την κατάσταση X στην κατάσταση X_G. Παρόλο που το βέλτιστο κόστος $C^*(x)$ μπορεί να υπολογιστεί μέσω δυναμικού προγραμματισμού , το $G^*(x)$ όμως δεν υπάρχει τρόπος να ξέρουμε ότι είναι το βέλτιστο από πριν. Αυτό που μπορούμε όμως να κάνουμε είναι μια υποτιμημένη εκτίμηση του κόστους αυτού. Ο στόχος είναι να υπολογίσουμε μια προσέγγιση του κόστους πολύ κοντά στην πραγματική τιμή χωρίς όμως να τη ξεπερνάει. Ας θεωρήσουμε G'^*(x), την προσέγγιση αυτήν. Ο αλγόριθμος A*, λειτουργεί με τον ίδιο τρόπο όπως ο αλγόριθμος του Dijkstra. Η μόνη διαφορά βρίσκεται στην συνάρτηση που αποσκοπεί στην ταξινόμηση της ουράς Q. Στον αλγόριθμο A-Star , χρησιμοποιείται το άθροισμα $C^*(x') + G'^*(x')$ υπονοώντας έτσι ότι η ουρά

προτεραιότητας Q , ταξινομείται με βάση τις προσεγγίσεις του βέλτιστου κόστους για την μετάβαση από την αρχική κατάσταση X_i προς την X_G.

Εάν το G'^*(x), αποτελεί όντως μια υποτίμηση του αληθινού βέλτιστου κόστους για όλα τα $x \in X$, τότε ο αλγόριθμος A-Star εγγυάται ότι θα βρει την βέλτιστη ακολουθία ενεργειών [3]. Όσο η εκτιμώμενη τιμή του κόστους πλησιάζει την πραγματική τιμή, τόσο ο αλγόριθμος δύναται να απαιτεί λιγότερες κορυφές για εξερεύνηση σε σύγκριση με τον αλγόριθμο του Dijkstra[3]. Τέλος είναι σημαντικό να τονίσουμε ότι εάν $G'^*(x) = 0$ για όλα τα $x \in X$, τότε ο αλγόριθμος συμπεριφέρεται σαν τον αλγόριθμο του Dijkstra[1]. Σε κάθε περίπτωση όμως είναι συστηματικός.

2.3 Αλγόριθμος D*

Εμπνευσμένος από τον A * ο αλγόριθμος D * είναι και αυτός ένας από τους δημοφιλέστερους αλγορίθμους που χρησιμοποιήθηκαν για την λύση τέτοιων προβλημάτων. Η ονομασία του προέρχεται από την ιδιαιτερότητα του ότι είναι δυναμικός , με την έννοια ότι τα κόστη μετάβασης μπορούν να αλλάξουν κατά την διάνυση του μονοπατιού επίλυσης.[4] Ο αλγόριθμος βρίσκει εγγυημένα βέλτιστο μονοπάτι εάν η διαδικασία αναθεώρησης τροφοδοτηθεί με σωστό τρόπο[5].

2.3.1 Μαθηματική διατύπωση

Ο χώρος του προβλήματος μπορεί να μοντελοποιηθεί ως ένα σύνολο καταστάσεων X που αντιπροσωπεύουν πιθανές θέσεις του ρομπότ , συνδεδεμένες με κατευθυνόμενες ακμές. Σε κάθε ακμή έχει ανατεθεί ένα σχετικό κόστος. Το ρομπότ ξεκινώντας από μία αρχική κατάσταση X_i , προχωράει επάνω στις ακμές όπως του υποδεικνύουν τα κόστη του μονοπατιού , μέχρι να φτάσει στην κατάσταση στόχο X_G. Κάθε κατάσταση χ εκτός από την X_G διαθέτει έναν δείκτη οπισθοδρόμησης προς μια επόμενη κατάσταση $y \in X$, τον οποίο σημειώνουμε $b(x) = y$.

Ο D* χρησιμοποιεί τους δείκτες οπισθοδρόμησης για να αναπαραστήσει μονοπάτια προς το στόχο. Το κόστος για να διανύσει το ρομπότ μια ακμή από την κατάσταση $y \in X$ στην κατάσταση $x \in X$ είναι ένας θετικός αριθμός που δίνεται από την συνάρτηση κόστους (cost function) $c(x, y)$. Εάν δεν υπάρχει ακμή από την y προς την x τότε το $c(x, y)$ δεν ορίζεται. Δυο καταστάσεις x και y θεωρούνται γειτονικές εάν έχει οριστεί το κόστος $c(x, y)$ ή το $c(y, x)$.

Σαν τον A*, ο D* διατηρεί μια ανοικτή λίστα (OPEN) καταστάσεων η οποία χρησιμοποιείται για την μετάδοση πληροφορίας σχετικά με αλλαγές στη συνάρτηση κόστους καθώς και στον υπολογισμό κοστών των καταστάσεων[4]. Κάθε κατάσταση x, έχει μια ετικέτα $t(x)$ τέτοια ώστε $t(x) = NEW$ εάν δεν έχει βρεθεί ποτέ στην ανοικτή λίστα στο παρελθόν , $t(x) = OPEN$ εάν βρίσκεται στην λίστα και $t(x) = CLOSED$ σε περίπτωση που δεν βρίσκεται πια σε αυτήν.

Για κάθε μια από τις καταστάσεις x, ο D* αποθηκεύει μια προσέγγιση της τιμής του αθροίσματος των κοστών μονοπατιού από την x στην X_G μέσω της συνάρτησης κόστους $h(x_G$,x). Υπό κατάλληλες συνθήκες η προσέγγιση θεωρείται βέλτιστη. Για κάθε κατάσταση που βρίσκεται στην ανοικτή λίστα (OPEN), ορίζεται μια συνάρτηση κλειδί $k(x_{G,}x)$ η οποία ισοδυναμεί με την ελάχιστη τιμή της $h(x_G$,x) από την ώρα που η x έχει τοποθετηθεί στην λίστα. Η συνάρτηση αυτή, κατηγοριοποιεί μια κατάσταση x σε μια από τις δυο κατηγορίες υψηλή(RAISE) εάν $k(x_G, x) < h(x_G, x)$ και χαμηλή(LOWER) εάν $k(x_G, x) = h(x_G, x)$ αντίστοιχα. Κάθε φορά που μια κατάσταση διαγράφεται από την λίστα, διευρύνεται ώστε να περάσει τυχόν αλλαγές κοστών στις γειτονικές καταστάσεις και αυτές με τις σειρά τους εισέρχονται στην λίστα για διεύρυνση. Οι καταστάσεις ταξινομούνται στην ανοικτή λίστα με βάση την παράμετρο k_{min}. Η παράμετρος αυτή ορίζεται ως $\min(k(x))$ για όλες τις καταστάσεις που έχουν $t(x) = OPEN$. Η παράμετρος k_{min} είναι πολύ σημαντική καθώς κόστη που είναι μικρότερα ή ίσα από αυτήν θεωρούνται βέλτιστα, ενώ αυτά που έχουν τιμή μεγαλύτερη μπορεί να μην είναι βέλτιστα. Η παράμετρος k_{old} ορίζεται να είναι ίση με την k_{min} αμέσως πριν την πιο πρόσφατη απομάκρυνση κάποιας κατάστασης από την λίστα. Εάν δεν έχει απομακρυνθεί καμία κατάσταση τότε η παράμετρος δεν ορίζεται.

Ένα σύνολο καταστάσεων $\{X_l , X_N\}$ ορίζεται σαν μια ακολουθία εάν $b(X_{i+1}) = X_i$ για όλα τα i έτσι ώστε $l \leq i < N$ και εάν $X_i \neq X_j$ για όλα τα (I ,j) τέτοια ώστε $l \leq i < j \leq N$.Έτσι η ακολουθία ορίζει ένα μονοπάτι από δείκτες οπισθοδρόμησης από την X_N στην X_l. Μια ακολουθία $\{X_i, X_N\}$ θεωρείται μονότονη εάν ($t(X_i) = CLOSED$) και $h(X_G, X_i) < h(X_G, X_{i+1})$) ή ($t(X_i) = OPEN$ και ($k(X_G, X_i) < h(X_G, X_{i+1})$) για όλα τα I που ισχύει $l \leq i < N$[5].Τέλος ο D* δημιουργεί και συντηρεί μια μονότονη ακολουθία $\{X_G, x\}$, η οποία αναπαριστά μειώσεις τρέχοντος κόστους για κάθε κατάσταση x που βρίσκεται ή βρισκόταν κάποια στιγμή στην ανοικτή λίστα(OPEN). Σε μια ακολουθία $\{X_l, X_N\}$ μια

κατάσταση X_i λέγεται πρόγονος μιας κατάστασης X_j εάν $l \leq i < j \leq N$ και απόγονος της X_j εάν ισχύει $l \leq j < i \leq N$[5].

Παρακάτω περιγράφεται ο αλγόριθμος D* όπως τον εισήγαγε ο Stentz[6].

2.3.2 Περιγραφή D* με κώδικα

Η βασική εκδοχή του αλγορίθμου D* περιέχει δυο βασικές συναρτήσεις : PROCESS-STATE και MODIFY-COST. Η PROCESS-STATE χρησιμοποιείται για τον υπολογισμό βέλτιστων κοστών προς τον στόχο, και η MODIFY-COST για την αλλαγή τιμών της συνάρτησης κόστους C(X) καθώς και να προσθέσει στην ανοικτή λίστα όλες τις καταστάσεις που επηρεάζονται. Οι ενσωματωμένες συναρτήσεις είναι $MIN(a, b)$, η οποία επιστρέφει τον ελάχιστο αριθμό από τους δύο, $LESS(a, b)$ η οποία επιστρέφει TRUE εάν $a < b$ και FALSE σε αντίθετη περίπτωση. $COST(X)$ η οποία επιστρέφει το $h(X)$ για την κατάσταση X, $MIN - STATE$ που επιστρέφει την κατάσταση στην ανοικτή λίστα με την μικρότερη τιμή k , (NULL αν η λίστα είναι άδεια). Η συνάρτηση $MIN - VAL$, η οποία επιστρέφει την ελάχιστη τιμή k_{min} για την λίστα OPEN ($NO - VAL$ εάν η λίστα είναι άδεια) , ενώ η $DELETE(X)$ διαγράφει μια κατάσταση X από την λίστα OPEN και θέτει το $t(X) = CLOSED$ και τέλος η συνάρτηση $INSERT(X, h_{new})$, υπολογίζει το $k(X) = h_{new}$ $\varepsilon\acute{\alpha}\nu$ $t(X) = NEW$, $k(X) = MIN(k(X), h_{new})$ $\varepsilon\acute{\alpha}\nu$ $t(X) = OPEN$, $k(X) = MIN(h(X), h_{new})$ θέτει $h(X) = h_{new}$ $\kappa\alpha\iota$ $t(X) = OPEN$ και ανακατατάσσει την X στην λίστα με βάση την τιμή k(x).

Συνάρτηση PROCESS-STATE:

1. $x = MIN - STATE( )$
2. $if\ x = NULL\ then\ return\ NO - VAL$
3. $k_{old} = k(x); DELETE(x)$
4. $if\ k_{old} < h(x)\ then$
5. for each neighbor y of x:
6. $if\ t(y) \neq NEW\ and\ h(y) \leq k_{old}\ and\ h(x) > h(y) + c(y, x)\ then$
7. $b(x) = y\ ; h(x) = h(y) + c(y, x)$
8. $if\ k_{old} = h(x)\ then$

9. for each neighbor y of x:

10. $if\ t(y) = NEW\ or$

11. $(b(y) = x\ and\ h(y) \neq h(x) + c(x,y))\ or$

12. $\big(b(y) \neq x\ and\ h(y) > h(x) + c(x,y)\big)\ then$

13. $b(y) = x\ ; INSERT\big(y, h(x) + c(x,y)\big)$

14. Else

15. for each neighbor y of x:

16. $if\ t(y) = NEW\ or$

17. $\big(b(y) = x\ and\ h(y) \neq h(x) + c(x,y)\big)\ then$

18. $b(y) = x\ ; INSERT\big(y, h(x) + c(x,y)\big)$

19. else

20. $if\ b(y) \neq x\ and\ h(y) > h(x) + c(x,y)$

21. $and\ t(x) = CLOSED\ then$

22. $INSERT\big(x, h(x)\big)$

23. else

24. $if\ b(y) \neq x\ and\ h(x) > h(y) + c(y,x)\ and$

25. $t(y) = CLOSED\ and\ h(y) > k_{old}\ then$

26. $INSERT\big(y, h(y)\big)$

27. $return\ MIN - VAL(\ )$

Εάν η κατάσταση X είναι τύπου RAISE, τότε το κόστος μονοπατιού της μπορεί να μην είναι βέλτιστο. Πριν η X μεταδώσει τυχόν αλλαγές στα κόστη των γειτονικών καταστάσεων, στις γραμμές 4 έως 7 οι γειτονικές καταστάσεις ελέγχονται για να δούμε αν το $h(x)$ μπορεί να ελαττωθεί. Στις γραμμές 15 ως 18, οι αλλαγές στα κόστη μεταδίδονται στις νέες καταστάσεις(NEW) καθώς και στους άμεσους απογόνους αυτών με τον ίδιο τρόπο όπως

για τις χαμηλές(LOWER) καταστάσεις. Εάν η X έχει την δυνατότητα να μειώσει το κόστος από μια κατάσταση που δεν είναι άμεσος απόγονός της(γραμμές 20 ως 21), τότε η X τοποθετείται πάλι στην λίστα για μελλοντική διερεύνηση. Αυτή η ενέργεια απαιτείται για μην δημιουργηθεί κλειστός κύκλος στις ακμές οπισθοδρόμησης. Αν το κόστος της X δύναται να μειωθεί από μια μη βέλτιστη γειτονική κατάσταση (γραμμές 23 με 25), τότε η γειτονική αυτή κατάσταση τοποθετείται πίσω στην λίστα. Έτσι η ενημέρωση του κόστους καθυστερεί μέχρις ότου η γειτονική κατάσταση έχει βέλτιστο κόστος.

Συνάρτηση MODIFY-COST$(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{cval})$**:**

1. $c(x, y) = cval$
2. $if\ t(x) = CLOSED\ then\ INSERT(x, h(x))$
3. return $MIN - VAL(\ )$

Στην παραπάνω συνάρτηση ,η συνάρτηση κόστους ακμής ενημερώνεται με την νέα τιμή. Επειδή το κόστος μονοπατιού για την Y θα αλλάξει , η κατάσταση X τοποθετείται στην λίστα OPEN. Όταν η κατάσταση X διευρυνθεί μέσω της $PROCESS - STATE$, υπολογίζει ένα νέο $h(y) = h(x) + c(x, y)$ και τοποθετεί την κατάσταση Y στην λίστα OPEN για διεύρυνση. Επιπρόσθετες διευρύνσεις της κατάστασης μεταδίδουν τυχόν αλλαγές κόστους στους απογόνους της Y.

Συνάρτηση MOVE-ROBOT $(\boldsymbol{S}, \boldsymbol{G})$**:**

1. $for\ each\ state\ x\ in\ graph$:
2. $t(x) = NEW$
3. $INSERT(G, 0)$
4. $val = 0$
5. $while\ t(S) \neq CLOSED\ and\ val \neq NO - VAL$
6. $val = PROCESS - STATE(\ )$
7. $if\ t(S) = NEW\ then\ return\ NO - PATH$
8. $R = S$

9. $while\ S \neq G$:

10. $for\ each\ (x, y)\ such\ that\ s(x, y) \neq c(x, y)$:

11. $val = MODIFY - COST(x, y, s(x, y)$

12. $while\ LESS\big(val, COST(R)\big)\ and\ val \neq NO - VAL$

13. $val = PROCESS - STATE(\ )$

14. $R = b(R)$

15. $return\ GOAL - REACHED$

Η παραπάνω συνάρτηση $MOVE - ROBOT$ δείχνει πως χρησιμοποιούνται οι $PROCESS - STATE$ και $MODIFY - COST$ για να κινηθεί το ρομπότ από την αρχική κατάσταση S στην κατάσταση στόχου G μέσω μιας βέλτιστης διαδρομής στο περιβάλλον. Στις γραμμές 1 ως 3 το $t(x)$ ορίζεται σε NEW για όλες τις καταστάσεις, το $h(G)$ ορίζεται σε μηδέν και η κατάσταση G τοποθετείται στην λίστα $OPEN$.Μεταξύ των γραμμών 5 και 6 καλείται η $PROCESS - STATE$ μέχρις ότου υπολογιστεί κάποιο αρχικό μονοπάτι για την κατάσταση που βρίσκεται το ρομπότ, (για παράδειγμα $t(S) = CLOSED$) ή έχει αποφασιστεί ότι δεν υπάρχει διαθέσιμο μονοπάτι οπότε($val = NO - VAL\ και\ t(S) = NEW$). Το ρομπότ τότε, ακολουθεί τους δείκτες οπισθοδρόμησης της ακολουθίας {R} μέχρι να φτάσει στον στόχο ή ανακαλύψει κάποια αλλαγή κόστους λόγω εμποδίου. (γραμμές 10 ως 11). Η συνάρτηση $MODIFY - COST$ καλείται να διορθώσει τις αλλαγές κόστους και να τοποθετήσει τις καταστάσεις που επηρεάζονται στην λίστα $OPEN$, ενώ η συνάρτηση $PROCESS - STATE$ που καλείται επανειλημμένα στην γραμμή 13 μέχρι να ισχύσει $val \geq h(R)$ έχει σκοπό να μεταδώσει τα κόστη και να υπολογίσει νέα ακολουθία {R} για τον στόχο. Το ρομπότ συνεχίζει να ακολουθεί τους δείκτες οπισθοδρόμησης της ακολουθίας προς τον στόχο. Η συνάρτηση επιστρέφει $GOAL - REACHED$ αν έφτασε στον στόχο και $NO - PATH$ αν ο στόχος είναι απρόσιτος.

Θα πρέπει να επισημάνουμε ότι στην γραμμή 7 , η συνάρτηση $MOVE - ROBOT$ ανιχνεύει μόνο ότι δεν υπάρχει ακολουθία από την θέση του ρομπότ προς τον στόχο, (διακοπτόμενο γράφημα)[5]. Δεν αναγνωρίζει την περίπτωση ότι όλα τα μονοπάτια προς τον στόχο είναι απροσπέλαστα λόγω εμποδίων.

Για να δώσουμε αυτήν την δυνατότητα στον αλγόριθμο μπορούμε να θέσουμε στις ακμές που εμποδίζονται μια πολύ μεγάλη θετική τιμή $OBSTACLE$ και στις ακμές που δεν εμποδίζονται μια πολύ μικρή θετική τιμή $EMPTY$. Δεν υπάρχει μη εμποδιζόμενο μονοπάτι από την S στον στόχο αν $h(s) \geq OBSTACLE$ κατά την έξοδο της επανάληψης στην γραμμή 5. Όμοια, δεν υπάρχει μη εμποδιζόμενο μονοπάτι από μια κατάσταση R προς τον στόχο κατά την διάνυση αν $h(R) \geq OBSTACLE$ κατά την έξοδο της επανάληψης στην γραμμή 12.

2.4 Αλγόριθμος Lifelong Planning A*

Ο αλγόριθμος αυτός (LPA^*) είναι μια σταδιακή(incremental) εκδοχή του A*[6]. Έχει εφαρμογή στα προβλήματα αναζήτησης σε πεπερασμένους Γράφους , πιο συγκεκριμένα σε γραφήματα στα οποία τα κόστη ακμών αυξάνονται ή μειώνονται με την πάροδο του χρόνου.(Μπορεί να χρησιμοποιηθεί και για την μοντελοποίηση κορυφών και ακμών που προστίθενται ή αφαιρούνται.) με το γράμμα S ορίζεται το πεπερασμένο σύνολο των κορυφών s του γραφήματος.

Ο όρος $Succ(s) \subseteq S$ δηλώνει το σύνολο των απογόνων της κορυφής $s \in S$. Με όμοιο τρόπο, ο όρος $Pred(s) \subseteq S$ δηλώνει το σύνολο των προγόνων της κορυφής $s \in S$, ενώ με τον όρο $0 < c(s, s') \leq \infty$ δηλώνεται το κόστος μετακίνησης από την κορυφή s στην κορυφή $s' \in S$. Ο αλγόριθμος LPA* , πάντα αποφασίζει το συντομότερο μονοπάτι μεταξύ μιας δοσμένης κορυφής $s_{start} \in S$ και μια δοσμένης κορυφής στόχου $s_{goal} \in S$ γνωρίζοντας την τοπολογία του γραφήματος καθώς και τα τρέχοντα κόστη μετακίνησης. Χρησιμοποιούμε το $g^*(s)$ για να ορίσουμε την αρχική απόσταση της κορυφής $s \in S$ που είναι το συντομότερο μονοπάτι από την s_{start} προς την s.

Όπως ο A* , έτσι και ο LPA*, χρησιμοποιεί ευρετικές συναρτήσεις $h(s, s_{goal})$ που προσεγγίζουν τις αποστάσεις των κορυφών s από τον στόχο s_{goal}. Οι ευρετικές συναρτήσεις πρέπει να είναι μη αρνητικές και συνεπείς, δηλαδή να ικανοποιούν την τριγωνική ανισότητα $h(s_{goal}, s_{goal}) = 0 \; και \; h(s, s_{goal}) \leq c(s, s') + h(s', s_{goal})$ για κάθε κορυφή $s \in S$ και $s' \in Succ(s) \; με \; s \neq s_{goal}$.[6]

2.4.1 Μαθηματικές μεταβλητές

Ο LPA* διατηρεί μια προσέγγιση $g(s)$ της αρχικής απόστασης $g'(s)$ για κάθε κορυφή s.Τις τιμές αυτές τις κουβαλάει από αναζήτηση σε αναζήτηση. Επιπλέον, διατηρεί και μια ακόμη προσέγγιση της αρχικής απόστασης. Οι τιμές rhs, είναι ένα βήμα μπροστά αναζήτησης βασισμένες στις τιμές g και έτσι πιθανότερα περισσότερο ενημερωμένες. Οι τιμές rhs πάντα ικανοποιούν την παρακάτω μαθηματική εξίσωση:

$$rhs = \begin{cases} 0, & s = s_{start} \\ min_{s' \in Pred(s)}(g(s') + c(s', s)), & s \neq s_{start} \end{cases}$$

Μια κορυφή καλείται τοπικά συνεπείς αν και μόνο αν η g-τιμή της ισούται με την rhs-τιμή, ($g = rhs$) και μη συνεπείς σε αντίθετη περίπτωση. Οι g-τιμές από όλες τις κορυφές είναι ίσες με τις αρχικές αποστάσεις τους αν και μόνον αν όλες οι κορυφές είναι τοπικά συνεπείς. Σε αυτήν την περίπτωση κάποιος μπορεί να εντοπίσει πίσω το συντομότερο μονοπάτι από τη s_{start}στην οποιαδήποτε κορυφή u μεταβαίνοντας πάντα από την τρέχουσα κορυφή s , ξεκινώντας από την κορυφή u, προς οποιοδήποτε πρόγονο s' που ελαχιστοποιεί την $g(s') + c(s', s)$ μέχρι την αρχική κορυφή s_{start}. Ο αλγόριθμος LPA* δεν καθιστά όλες τις κορυφές συνεπείς μετά την αλλαγή κάποιων κοστών ακμών αλλά αντιθέτως χρησιμοποιεί τις ευρετικές για να εστιάσει την αναζήτηση και ενημερώνει μόνο τις g-τιμές που σχετίζονται άμεσα με τον υπολογισμό συντομότερου μονοπατιού. Έτσι ο LPA* διατηρεί μια ουρά ή λίστα προτεραιότητας. Η ουρά αυτή πάντα περιέχει τις κορυφές που δεν είναι τοπικά συνεπείς δηλαδή περιέχει εκείνες τις κορυφές των οποίων οι g-τιμές πιθανώς χρειάζονται ενημέρωση ώστε να γίνουν συνεπείς. Η προτεραιότητα μια κορυφής είναι ίδια με το κλειδί της που είναι ένα δισδιάστατο διάνυσμα :

$$k(s) = [k_1(s), k_2(s)]$$

Όπου $k_1(s) = \min\big(g(s), rhs(s)\big) + h(s, s_{goal})$ και $k_2(s) = \min\,(g(s), rhs(s))$ {γραμμή 1}. Ένα κλειδί $k(s)$ είναι μικρότερο ή ίσο με ένα κλειδί $k'(s)$, δηλαδή $k(s) \leq k'(s)$ αν και μόνον αν $k_1(s) < k'_1(s)$ ή $(k_1(s) = k'_1(s)\ \kappa\alpha\iota\ k_2(s) \leq k'_2(s))$. Ο LPA* διευρύνει πάντα την κορυφή με το μικρότερο κλειδί όμοια με τον A*.

Παρακάτω παρουσιάζεται ο ψευδο-κώδικας όπως τον παρουσίασαν οι Koenig και Likhachev:[7].

Συνάρτηση CalculateKey(s):

1. $retun\left[\min\left(g(s), rhs(s)\right) + h(s, s_{start}); \min\left(g(s), rhs(s)\right)\right];$

Συνάρτηση Initialize( ):

2. $U = \emptyset;$
3. $for\ all\ s \in S\ rhs = g(s) = \infty;$
4. $rhs(s_{start}) = 0;$
5. $U.Insert\left(s_{start}, CalculateKey(s_{start})\right);$

Συνάρτηση UpdateVertex(u):

6. $if\ (u \neq s_{start})\ then\ rhs(u) = min_{s' \in Pred(u)}\left(g(s') + c(s', u)\right);$
7. $if(u \in U)\ then\ U.Remove(u);$
8. $if\left(g(u) \neq rhs(u)\right)\ then\ U.Insert\left(u, Calculatekey(u)\right);$

Συνάρτηση ComputeShortestPath( ):

9. $while\ (U.TopKey(\ ) < CalculateKey\left(s_{goal}\right)\ OR\ rhs\left(s_{goal}\right) \neq g(s_{goal})$
10. $u = U.Pop(\ );$
11. $if\left(g(u) > rhs(u)\right)$
12. $g(u) = rhs(u);$
13. $for\ all\ s \in Succ(u)\ UpdateVertex(s);$
14. $else$
15. $g(u) = \infty;$
16. $for\ all\ s \in Succ(u)\ \cup \{u\}\ UpdateVertex(s);$

Συνάρτηση Main( ):

17. $Ininitalize(\ );$
18. $forever$
19. $ComputeShortestPath(\ );$
20. $wait\ for\ changes\ in\ edge\ costs;$
21. $for\ all\ directed\ edges\ (u, v)\ with\ changed\ edge\ costs$
22. $Update\ the\ edge\ cost\ c(u, v);$
23. $UpdateVertex(v);$

Ο παραπάνω ψευδοκώδικας χρησιμοποιεί τις ακόλουθες συναρτήσεις για να διαχειριστεί την ουρά προτεραιότητας: Η συνάρτηση $U.Top(\,)$, επιστρέφει την κορυφή με τον μικρότερο αριθμό προτεραιότητας. Η $U.TopKey(\,)$, επιστρέφει την μικρότερη προτεραιότητα από όλες τις κορυφές, σε περίπτωση που η ουρά είναι άδεια επιστρέφει $[\infty;\infty]$. Σειρά έχει η $U.Pop(\,)$ η οποία διαγράφει την κορυφή που βρίσκεται πάνω πάνω , αυτήν δηλαδή με τον μικρότερο αριθμό προτεραιότητας και επιστρέφει την κορυφή αυτήν. Η $U.Insert(s,k)$ τοποθετεί την κορυφή s στην λίστα U με προτεραιότητα k, ενώ η $U.Update(s,k)$ αλλάζει τον αριθμό προτεραιότητας της κορυφής s σε k.(Σε περίπτωση που η προτεραιότητα ήδη ισούται με k δεν γίνεται τίποτα). Τέλος, η συνάρτηση $U.Remove(s)$ διαγράφει την κορυφή s από την λίστα U.

2.4.2 Επεξήγηση του αλγορίθμου LPA*

Η κύρια συνάρτηση $Main(\,)$ του αλγορίθμου καλεί πρώτα την συνάρτηση $Initialize(\,)$ {γραμμή 17} ,η οποία αρχικοποιεί το πρόβλημα αναζήτησης θέτοντας τις τιμές g σε όλες τις κορυφές καθώς και τις τιμές rhs σύμφωνα με τις εξισώσεις στις γραμμές {3 και 4}. Έτσι η s_{start} είναι η μόνη μη συνεπείς κορυφή και γι' αυτό εισέρχεται στην ουρά προτεραιότητας για διεύρυνση {γραμμή 5}.Η αρχικοποίηση αυτή, εγγυάται ότι στην πρώτη εκτέλεση της συνάρτησης $ComputeShortestPath(\,)$ κάνει μια αναζήτηση ακριβώς ίδια με τον αλγόριθμο A*,αφού διευρύνει τις ίδιες κορυφές με την ίδια ακριβώς σειρά. Εδώ πρέπει να σημειωθεί ότι σε μια πραγματική υλοποίηση η $Initialize(\,)$ πρέπει να αρχικοποιεί μια κορυφή την στιγμή που την ανακαλύπτει στην αναζήτηση και έτσι δεν χρειάζεται να αρχικοποιήσει όλες τις κορυφές εξ 'αρχής.[7] Αυτό είναι πολύ σημαντικό καθώς το πλήθος των κορυφών μπορεί να είναι μεγάλο και το ρομπότ να φτάσει μόνο μερικές κατά την αναζήτηση. Αμέσως μετά τα παραπάνω ο αλγόριθμος περιμένει για αλλαγές κόστους στις ακμές {γραμμή 20}. Εάν όντως υπάρξουν αλλαγές στα κόστη των ακμών τότε καλεί την $UpdateVertex(\,)$ με σκοπό να ενημερώσει τις rhs-τιμές και τα κλειδιά των κορυφών που πιθανώς επηρεάζονται από τις αλλαγές αυτές , επιπλέον τις προσθέτει ή τις αφαιρεί από την ουρά προτεραιότητας αντίστοιχα αν μετά τις αλλαγές γίνουν ασυνεπείς ή συνεπείς. Τέλος υπολογίζει ξανά ένα συντομότερο μονοπάτι καλώντας την $ComputeShortestPath()$, που διευρύνει επανειλημμένα τις τοπικά ασυνεπείς κορυφές ανάλογα με την προτεραιότητα τους.

Μια τοπικά ασυνεπής κορυφή ονομάζεται τοπικά υπερσυνεπής αν και μόνον αν $g(s) > rhs(s)$[7]. Όταν η $ComputeShortestPath()$ διευρύνει μια τοπικά υπέρ συνεπή κορυφή, τότε θέτει την g-τιμή της ίση με την rhs-τιμή κάτι το οποίο κάνει την κορυφή τοπικά συνεπή {γραμμές 12-13}. Μια τοπικά ασυνεπής κορυφή ονομάζεται τοπικά υπό συνεπής αν και μόνον αν $g(s) < rhs(s)$[7]. Όταν η $ComputeShortestPath()$ διευρύνει μια τοπικά υπό συνεπή κορυφή , τότε θέτει την g-τιμή του ίση με άπειρο {γραμμές 15-16}.Αυτό έχει ως αποτέλεσμα η κορυφή να γίνει τοπικά είτε συνεπής είτε υπέρ συνεπής. Εάν η διευρυμένη κορυφή ήταν υπερσυνεπής, τότε η αλλαγή της g-τιμής μπορεί να επηρεάσει την τοπική συνέπεια και των απογόνων της {γραμμή 13}. Αντίστοιχα εάν η διευρυμένη κορυφή ήταν υπό συνεπής τότε θα επηρεαζόταν και αυτή και οι απόγονοί της{γραμμή 16}.Έτσι λοιπόν η συνάρτηση $ComputeShortestPath()$ ενημερώνει τις rhs-τιμές των κορυφών αυτών, ελέγχει την τοπική συνέπεια τους ώστε να τις προσθέσει στην λίστα ή να τις απομακρύνει από αυτήν. {γραμμές 6-8}. Επιπλέον, διευρύνει κορυφές μέχρις ότου η s_{goal} να γίνει τοπικά συνεπή και το κλειδί της επόμενης κορυφής προς διεύρυνση να μην είναι μικρότερο από το κλειδί της s_{goal}.Μετά το τέλος της αναζήτησης αν $g(s_{goal}) = \infty$ τότε δεν υπάρχει κανένα μονοπάτι πεπερασμένου κόστους από την s_{start} $στην$ S_{goal}.Σε αντίθετη περίπτωση κάποιος μπορεί να εντοπίσει το συντομότερο μονοπάτι από την s_{start} $στην$ s_{goal} ξεκινώντας από την s_{goal} και πηγαίνοντας προς τα πίσω σε οποιαδήποτε πρόγονο s' που ελαχιστοποιεί την τιμή: $g(s') + c(s', s)$.

Στην παρακάτω ενότητα παρουσιάζονται κάποια θεωρήματα τα οποία σύμφωνα με τους Likhachev και Koenig[7] δείχνουν ότι ο LPA* τερματίζει, είναι σωστός, παρόμοιος με τον A* και αποτελεσματικός.

2.4.3 Αναλυτικά αποτελέσματα

Το πρώτο θεώρημα δείχνει ότι ο LPA* τερματίζει καθώς και ότι είναι σωστός.

Θεώρημα 1: η συνάρτηση $ComputeShortestPath()$ διευρύνει κάθε κορυφή το πολύ δυο φορές, το πολύ μια αν είναι υπερσυνεπής ή υποσυνεπής και άρα τερματίζει. Μόλις τερματίσει η συνάρτηση αυτή, μπορούμε να ανακτήσουμε ένα συντομότερο μονοπάτι από την s_{start} στην s_{goal} όπως ειπώθηκε λίγο παραπάνω.

Θεώρημα 2: Τα κλειδιά των κορυφών που η $ComputeShortestPath()$ επιλέγει για διεύρυνση {γραμμή 10}, δεν μειώνονται με τον χρόνο μέχρι τον τερματισμό της συνάρτησης. Αυτό είναι όμοιο με τις f-τιμές των κορυφών που διευρύνονται από τον A*.

Θεώρημα 3: Όταν η $ComputeShortestPath()$ επιλέγει μια υπερσυνεπή κορυφή s για διεύρυνση τότε το κλειδί της είναι $k(s) = [f(s); g^*(s)]$. Ο A* διαθέτει την ίδια ιδιότητα δεδομένου ότι όταν υπάρχει ισοπαλία μεταξύ δύο κορυφών, επιλέγει αυτή με την μικρότερη αρχική απόσταση.

Θεώρημα 4: Η συνάρτηση $ComputeShortestPath()$ δεν διευρύνει κορυφές των οποίων οι g-τιμές ήταν ίσες με τις αρχικές αποστάσεις τους πριν την κλήση της συνάρτησης. Το Θεώρημα αυτό δείχνει ότι ο LPA* είναι αποτελεσματικός, επειδή κάνει σταδιακές αναζητήσεις και έτσι υπολογίζει μόνο τις g-τιμές που έχουν επηρεαστεί από τυχόν αλλαγές κόστους ή δεν έχουν υπολογιστεί σε προηγούμενες αναζητήσεις.

Θεώρημα 5: Τα κλειδιά των κορυφών που η επιλέγει $ComputeShortestPath()$ για διεύρυνση {γραμμή 10}, δεν υπερβαίνουν ποτέ το $[f(s_{goal}); g^*(s_{goal})]$. Το θεώρημα αυτό επίσης δείχνει ότι ο LPA* είναι αποτελεσματικός επειδή κάνει ευρετικές αναζητήσεις και έτσι υπολογίζει μόνο τις g-τιμές των κορυφών που είναι απαραίτητες για την εκλογή συντομότερου μονοπατιού. Μπορούμε να κατανοήσουμε καλύτερα το θεώρημα αυτό αν αναλογιστούμε ότι το κλειδί $k(s)$ μιας κορυφής s είναι $k(s) = [\min(g(s), rhs(s) + h(s, s_{goal}); \min(g(s), rhs(s))]$. Έτσι , όσο καλύτερες και μεγαλύτερες είναι οι ευρετικές τιμές , τόσο λιγότερες κορυφές ικανοποιούν την συνθήκη $k(s) \leq [f(s_{goal}), g^*(s_{goal})]$ και άρα διευρύνονται.

2.5 Αλγόριθμος D*Lite

Ο τελευταίος αλγόριθμος που ασχολείται αυτή η εργασία είναι ο D*Lite. Οι Likhachev και Koenig[8] χρησιμοποιώντας τον LPA* ως βάση ανέπτυξαν τον αλγόριθμο αυτό , ο οποίος βρίσκει επανειλημμένα συντομότερα μονοπάτια μεταξύ της τρέχουσας κορυφής και της κορυφής στόχου, καθώς τα κόστη των ακμών αλλάζουν ενώ το ρομπότ κινείται προς την κορυφή στόχο. Ο D*Lite μπορεί να χρησιμοποιηθεί επιτυχώς για την επίλυση προβλημάτων πλοήγησης κατευθυνόμενου στόχου σε άγνωστο περιβάλλον. Το περιβάλλον αυτό μοντελοποιείται ως ένα γράφημα συνδεδεμένο σε οκτώ κατευθύνσεις. Το αρχικό κόστος για κάθε ακμή είναι ένα. Όταν το ρομπότ ανακαλύπτει πως μια ακμή δεν

είναι προσβάσιμη τότε το κόστος αυτό αλλάζει σε άπειρο. Θεωρούμε s_{start} την κορυφή εκκίνησης και s_{goal} την κορυφή στόχο.

2.5.1 Ψεύδο-κώδικας του D*Lite

Συνάρτηση $\boldsymbol{CalculateKey}(\boldsymbol{s})$:

1. $return\ [\min\big(g(s), rhs(s) + h(s_{start}, s) + k_m; \min\big(g(s), rhs(s)\big)\big]$;

Συνάρτηση $\boldsymbol{Initialize}(\ )$:

2. $U = \emptyset$;
3. $k_m = 0$;
4. $for\ all\ s \in S\ rhs(s) = g(s) = \infty$;
5. $rhs\big(s_{goal}\big) = 0$;
6. $U.Insert(s_{goal}, CalculateKey\big(s_{goal}\big)$;

Συνάρτηση $\boldsymbol{UpdateVertex}(\boldsymbol{u})$:

7. $if\ \big(u \neq s_{goal}\big)\ rhs(u) = min_{s' \in\ Succ(u)} c(u, s') + g(s')$;
8. $if\ (u \in U)\ U.Remove(u)$;
9. $if\ \big(g(u) \neq rsh(u)\big)\ U.Insert\big(u, CalculateKey(u)\big)$;

Συνάρτηση $\boldsymbol{ComputeShortestPath}(\ )$:

10. $while\ U.TopKey(\ ) < CalculateKey(s_{start}) OR\ rhs(s_{start}) \neq g(s_{start}))$
11. $k_{old} = U.TopKey(\ )$;
12. $u = U.Pop()$;
13. $if\ \big(k_{old} < CalculateKey(u)\big)$
14. $U.Insert\big(u, CalculateKey(u)\big)$;
15. else $if\big(g(u) > rhs(u)\big)$
16. $g(u) = rhs(u)$;
17. $for\ all\ s \in Pred(u)\ UpdateVertex(s)$;
18. else
19. $g(u) = \infty$;
20. $for\ all\ s \in Pred(u) \cup \{u\}\ UpdateVertex(s)$;

Συνάρτηση $\boldsymbol{Main}()$:

21. $s_{last} = s_{start}$;
22. $Initialize()$;
23. $ComputeShortestPath()$;
24. $while\ (s_{start} \neq s_{goal})$
25. $if\ (g(s_{start}) = \infty)$ /*Τότε δεν υπάρχει γνωστό μονοπάτι*/
26. $s_{start} = \arg min_{s' \in Succ(s_{start})}\ c(s_{start}, s') + g(s')$;
27. Move to s_{start};
28. Scan graph for changed edge costs;
29. If any costs changed
30. $k_m = k_m + h(s_{start}, s_{goal})$;
31. $s_{last} = s_{start}$;
32. $for\ all\ directed\ edges\ (u, v)\ with\ costs$
33. $Update\ the\ edge\ cost\ c(u, v)$;
34. $UpdateVertex(u)$;
35. $ComputeShortestPath()$;

2.5.2 Κατεύθυνση αναζήτησης:

Αρχικά, πρέπει αλλάξουμε την κατεύθυνση αναζήτησης του LPA* εφόσον χρησιμοποιείται ως βάση. Η εκδοχή του LPA* που αναφέρθηκε στην ενότητα 2.4 αναζητεί το συντομότερο μονοπάτι από την αρχική κορυφή s_{start} στην κορυφή στόχο s_{goal} και έτσι οι g-τιμές υπολογίζονται με βάση τις αρχικές αποστάσεις[7]. Αντιθέτως, ο D*Lite αναζητεί το συντομότερο μονοπάτι από την κορυφή στόχο προς την αρχική κορυφή και έτσι οι g-τιμές υπολογίζονται με βάση τις αποστάσεις από την κορυφή στόχο[8]. Αυτό προκύπτει από τον LPA* αλλάζοντας την αρχική κορυφή με τον στόχο και αντιστρέφοντας όλες τις ακμές στον ψευδο-κώδικα. Έτσι ο D*Lite το μόνο που έχει να κάνει είναι να αποφασίσει τους προγόνους και απογόνους των κορυφών. Μόλις η $ComputeShortestPath()$ επιστρέψει, μπορεί κάποιος να ακολουθήσει το συντομότερο μονοπάτι από την s_{start} στην s_{goal} κινούμενος πάντα από την τρέχουσα κορυφή s, ξεκινώντας από την s_{start}, προς οποιαδήποτε κορυφή s’ που ελαχιστοποιεί το $c(s, s') + g(s')$ μέχρι να φτάσει την s_{goal}.

2.5.3 Ανακατάταξη της ουράς προτεραιότητας.

Για την επίλυση προβλημάτων πλοήγησης σε άγνωστο περιβάλλον, η συνάρτηση $Main()$ πρέπει να μετακινήσει το ρομπότ από το μονοπάτι που έχει υπολογίσει η συνάρτηση $ComputeShortestPath()$. Για να το πετύχει αυτό, η συνάρτηση $Main()$ θα μπορούσε να επανυπολογίσει τις προτεραιότητες των κορυφών της ουράς προτεραιότητας κάθε φορά που το ρομπότ εντοπίζει μια αλλαγή στα κόστη των ακμών αφού έχει μετακινηθεί. Επειδή οι προτεραιότητες των κορυφών είναι βασισμένες σε ευρετικές τιμές που υπολογίστηκαν έχοντας λάβει υπόψιν την παλιά θέση του ρομπότ πρέπει να ξανά υπολογιστούν. Όμως οι πολλαπλές ανακατατάξεις της ουράς προτεραιότητας μπορούν να κάνουν τον αλγόριθμο μη αποδοτικό διότι είναι ακριβές, ειδικά όταν η ουρά διαθέτει μεγάλο αριθμό κορυφών. Έτσι για να αποφύγει την συχνή ανακατάταξη της ουράς προτεραιότητας ο D*Lite χρησιμοποιεί μια μέθοδο από τον D*[6] και ποιο συγκεκριμένα χρησιμοποιεί προτεραιότητες που αποτελούν κάτω όρια των προτεραιοτήτων που χρησιμοποιεί ο LPA*[7]. Οι ευρετικές τιμές που επιστρέφει η $h(s, s')$ πρέπει να είναι μη αρνητικές και να ικανοποιούν την εξίσωση $h(s, s') \leq c^*(s, s')\ \kappa\alpha\iota\ h(s, s'') \leq h(s, s') + h(s', s'')$ για όλες τις κορυφές $s, s', s'' \in S$,όπου $c^*(s, s')$ δηλώνει το κόστος συντομότερου μονοπατιού από την κορυφή s στην s'. Μετά την μετακίνηση του ρομπότ από την κορυφή s στην κορυφή s' όπου έχει εντοπίσει αλλαγή σε κάποιο κόστος ακμής, το πρώτο μέρος των προτεραιοτήτων που είχαν επηρεαστεί θα μπορούσαν να είχαν μειωθεί το πολύ $h(s, s')$. Έτσι για να διατηρηθούν τα κάτω όρια ο D*Lite πρέπει να αφαιρέσει από το πρώτο μέρος των προτεραιοτήτων όλων των κορυφών της ουράς, την τιμή $h(s, s')$. Επιπλέον επειδή η τιμή αυτή είναι ίδια για όλες τις κορυφές η σειρά τους δεν αλλάζει εάν δεν γίνει η αφαίρεση. Όταν υπολογίζονται νέες προτεραιότητες , το πρώτο μέρος τους είναι κατά $h(s, s')$ μικρότερο σε σχέση με τις υπάρχουσες στην ουρά. Έτσι πρέπει να προστεθεί η τιμή $h(s, s')$ κάθε φορά που εντοπίζεται αλλαγή σε κόστος ακμής. Εάν το ρομπότ μετακινηθεί ξανά , οι σταθερές αυτές πρέπει να προστεθούν ξανά. Αυτό γίνεται με την μεταβλητή k_m {γραμμή 30}. Μόλις υπολογιστούν νέες προτεραιότητες, η μεταβλητή k_m προστίθεται στα πρώτα μέρη τους {γραμμή 1}, η σειρά των κορυφών δεν αλλάζει στην ουρά αφού μετακινηθεί το ρομπότ και η ουρά προτεραιότητας δεν χρειάζεται ανακατανομή. Από την άλλη μεριά , οι προτεραιότητες αποτελούν πάντα κάτω όρια των προτεραιοτήτων που χρησιμοποιούνται στον LPA*[7] αφού το πρώτο μέρος τους αυξηθεί κατά την τρέχουσα τιμή της μεταβλητής k_m.Η ιδιότητα αυτή αξιοποιείται τροποποιώντας

λίγο την συνάρτηση $ComputeShortestPath()$. Πιο συγκεκριμένα, αφού έχει αφαιρέσει από την ουρά την κορυφή u με την μικρότερη προτεραιότητα $k_{old} = U.TopKey()$ {γραμμή 12}, χρησιμοποιεί την $CalculateKey()$ για να υπολογίσει την προτεραιότητα που θα έπρεπε η κορυφή αυτή να έχει. Εάν $k_{old} < CalculateKey(u)$, τότε ξανά τοποθετεί την κορυφή u πίσω στην ουρά με την νέα υπολογισμένη προτεραιότητα {γραμμές 13-14}. Έτσι παραμένει το αληθές ότι οι προτεραιότητες των κορυφών που βρίσκονται στην ουρά, αποτελούν κάτω όρια των αντίστοιχων προτεραιοτήτων που χρησιμοποιεί ο LPA*, αφού προηγουμένως έχει προστεθεί στο πρώτο μέρος αυτών η τρέχουσα τιμή της μεταβλητής k_m. Εάν $k_{old} \geq CalculateKey(u)$, τότε η $ComputeShortestPath()$ κρατάει ότι $k_{old} = CalculateKey(u)$ επειδή k_{old} ήταν κάτω όριο της τιμής που επέστρεψε η $CalculateKey()$, και διευρύνει την κορυφή u. {γραμμές 15-20}.

2.5.4 Βελτιστοποιημένος αλγόριθμος D*Lite

Συνάρτηση $\boldsymbol{CalculateKey(s)}$:

1. $return\ [\min(\mathrm{g}(s)\ , rhs(s)) + h(s_{start}, s) + k_m; \min\ (g(s), rhs(s))];$

Συνάρτηση $\boldsymbol{Initialize()}$:

2. $U = \emptyset;$
3. $k_m = 0;$
4. $for\ all\ s \in S\ rhs(s) = g(s) = \infty;$
5. $rhs(s_{goal}) = 0;$
6. $U.Insert(s_{goal}, [h(s_{start}, s_{goal}); 0]);$

Συνάρτηση $\boldsymbol{UpdateVertex(u)}$:

7. $if(g(u) \neq rhs(u)\ AND\ u \in U)\ U.Update(u, CalculateKey(u));$
8. $else\ if(g(u) \neq rhs(u)\ AND\ u \notin U)\ U.Insert(u, CalculateKey(u));$
9. $else\ if(g(u) = rhs(u)\ AND\ u \in U)\ U.Remove(u);$

Συνάρτηση $\boldsymbol{ComputeShortestPath()}$:

10. $while\ (U.TopKey(\ ) < CalculateKey(s_{start})\ OR\ rhs(s_{start}) > g(s_{start}))$
11. $\quad u = U.Top();$
12. $\quad k_{old} = U.TopKey(\ );$

13. $k_{new} = CalculateKey(u)$;
14. $if(k_{old} < k_{new})$
15. $\quad U.Update(u, k_{new})$;
16. $else\ if\big(g(u) > rhs(u)\big)$
17. $\quad g(u) = rhs(u)$;
18. $\quad U.Remove(u)$;
19. $\quad for\ all\ s \in Pred(u)$
20. $\quad\quad if\big(s \neq s_{goal}\big)\ rhs(s) = \min\big(rhs(s), c(s,u) + g(u)\big)$;
21. $\quad\quad UpdateVertex(s)$;
22. $else$
23. $\quad g_{old} = g(u)$;
24. $\quad g(u) = \infty$;
25. $\quad for\ all\ s \in Pred(u) \cup \{u\}$
26. $\quad\quad if(rhs(s) = c(s,u) + g_{old})$
27. $\quad\quad\quad if\big(s \neq s_{goal}\big) rhs(s) = min_{s' \in Succ(s)}\big(c(s,s') + g(s')\big)$;
28. $\quad\quad UpdateVertex(s)$;

Συνάρτηση $Main()$:

29. $s_{last} = s_{start}$;
30. $Initialize()$;
31. $ComputeShortestPath()$;
32. $while\big(s_{start} \neq s_{goal}\big)$
33. $if(rhs(s_{start}) = \infty)$ $/*$ τοτε δεν υπάρχει γνωστό μονοπάτι $*/$
34. $s_{start} = \arg min_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$;
35. Move to s_{start};
36. Scan graph for changed edge costs ;
37. If any edge costs changed
38. $\quad k_m = k_m + h(s_{last}, s_{start})$;
39. $\quad s_{last} = s_{start}$;
40. $\quad for\ all\ directed\ edges\ (u,v)\ with\ changed\ edge\ costs$

41. $c_{old} = c(u, v)$;
42. $Update\ edge\ cost\ c(u, v)$;
43. $if\big(c_{old} > c(u, v)\big)$
44. $if\big(u \neq s_{goal}\big)\ rhs(u) = \min\big(rhs(u), c(u, v) + g(v)\big)$;
45. $else\ if\big(rhs(u) = c_{old} + g(v)\big)$
46. $if\big(u \neq s_{goal}\big)\ rhs(u) = min_{s' \in Succ(u)}\big(c(u, s') + g(s')\big)$;
47. $UpdateVertex(u)$;
48. $ComputeShortestPath()$;

2.5.5 Βελτιστοποιήσεις

Στην ενότητα 2.5.4 παρουσιάζεται ο αλγόριθμος D*Lite με μερικές βελτιστοποιήσεις που ανεβάζουν την απόδοση στα μεγάλα και πυκνά γραφήματα[8]. Μία από αυτές τις βελτιστοποιήσεις είναι η συνθήκη τερματισμού της $ComputeShortestPath()$. Όπως έχει ειπωθεί αρχικά στην ενότητα 2.5.1 , η $ComputeShortestPath()$ τερματίζει όταν η αρχική κορυφή s_{start} είναι τοπικά συνεπής και το κλειδί προτεραιότητάς της είναι μικρότερο ή ίσο με το $U.TopKey()$ {γραμμή 10}. Όμως, η $ComputeShortestPath()$ τερματίζει ήδη όταν η αρχική κορυφή s_{start} δεν είναι τοπικά υπό συνεπής και το κλειδί της είναι μικρότερο ή ίσο με $U.TopKey()$. Για να το κατανοήσουμε αυτό καλύτερα ας θεωρήσουμε ότι η s_{start} είναι τοπικά υπέρ συνεπής και το κλειδί της είναι μικρότερο ή ίσο του $U.TopKey()$. Επειδή η $U.TopKey()$ ούτως ή άλλως επιστρέφει το κλειδί με την μικρότερη τιμή, τότε το κλειδί της s_{start} θα πρέπει να είναι ίσο με $U.TopKey()$. Έτσι η $ComputeShortestPath()$ θα μπορούσε να διευρύνει την s_{start} μετά, στην περίπτωση αυτή θα έθετε την g-τιμή της ίση με την rhs-τιμή. Η s_{start}τότε γίνεται τοπικά συνεπής , το κλειδί της είναι μικρότερο του $U.TopKey()$ και η συνάρτηση τερματίζει. Στο σημείο αυτό, η g-τιμή της κορυφής s_{start} είναι ίση με την απόσταση της από την κορυφή στόχο s_{goal}. Έτσι ισχύει ό,τι ειπώθηκε παραπάνω πως η $ComputeShortestPath()$ ήδη τερματίζει αν η s_{start} είναι τοπικά υπό συνεπής και το κλειδί προτεραιότητας είναι μικρότερο ή ίσο με $U.Topkey()$ {γραμμή 10}. Στην περίπτωση αυτήν, η s_{start} μπορεί να παραμείνει τοπικά μη συνεπής αφού τερματίσει η $ComputeShortestPath()$ και η g-τιμή της να μην ισούται με τη απόσταση από την s_{goal}. (η rhs μπορεί να είναι όμως) Αυτό το ενδεχόμενο δεν αποτελεί πρόβλημα καθώς η g-τιμή δεν χρησιμοποιείται για τις αποφάσεις κίνησης του ρομπότ.

2.5.6 Αναλυτικά αποτελέσματα του D*Lite

Η συνάρτηση $ComputeShortestPath()$ του D*Lite είναι παρόμοια με την αντίστοιχη του LPA* και έχει αρκετές όμοιες ιδιότητες για παράδειγμα και οι δύο διευρύνουν κάθε κορυφή το πολύ δυο φορές μέχρι να επιστρέψουν. Παρακάτω δίνεται ένα θεώρημα που σύμφωνα με τους Likhachev και Koenig[8] δείχνει ότι η $ComputeShortestPath()$ τερματίζει και είναι σωστή.

Θεώρημα 6:

Η $ComputeShortestPath()$, πάντα τερματίζει και μπορεί κάποιος να ακολουθήσει ένα συντομότερο μονοπάτι από την s_{start} στην s_{goal} ,κινούμενος από την τρέχουσα κορυφή s , ξεκινώντας από την s_{start} προς οποιοδήποτε απόγονο s' που ελαχιστοποιεί το $c(s, s') + g(s')$ μέχρι να φτάσει στην s_{goal}.

Ο αλγόριθμος D*Lite είναι τουλάχιστον αποδοτικός όσο ο D*[8]. Σύμφωνα με τους Likhachev και Koenig[8] ,ο D*Lite όταν χρησιμοποιεί συνδυασμό σταδιακής και ευρετικής αναζήτησης είναι πιο αποδοτικός από όταν χρησιμοποιεί είτε μόνο την μια είτε την άλλη μέθοδο. Επιπλέον να επισημάνουμε ότι η διαφορά απόδοσης D*Lite χωρίς ευρετική αναζήτηση, σε μικρά περιβάλλοντα της τάξεως $10\ X\ 10$ ή $15\ X\ 15$ είναι αμελητέα.

Κεφάλαιο 3

Υλικά/ Μέρη του Ρομπότ

Σε αυτό το κεφάλαιο παρουσιάζονται τα υλικά που χρησιμοποιήθηκαν για την κατασκευή του πράκτορα ρομπότ με τα χαρακτηριστικά τους.

Ως βασικός μικροϋπολογιστής επιλέχτηκε το Raspberry Pi [10], ένας πλήρες υπολογιστής σε μέγεθος πιστωτικής κάρτας με τα παρακάτω τεχνικά χαρακτηριστικά:

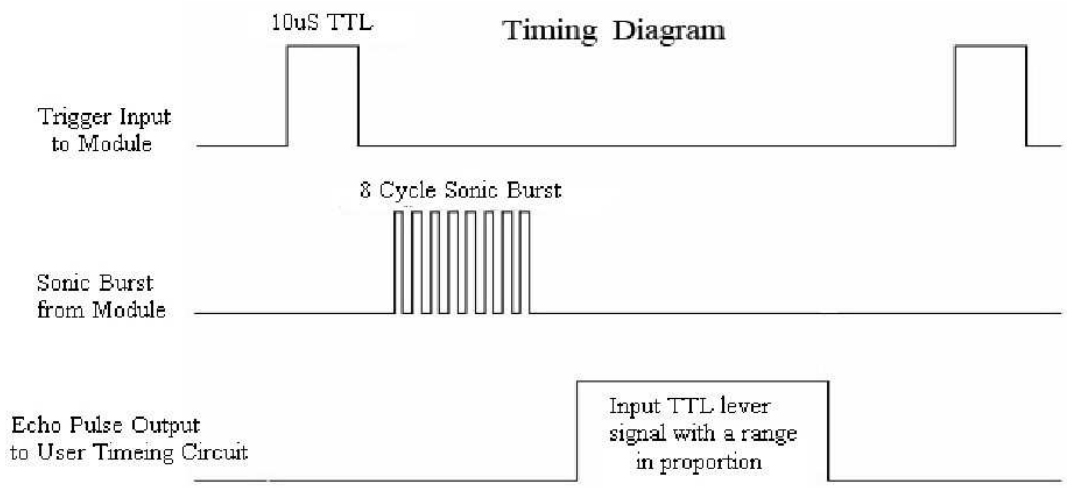
Εικόνα 1

- Chip Broadcom BCM2835
- SoC Core architecture ARM11
- CPU 700 MHz Low Power ARM1176JZFS Applications Processor
- GPU Dual Core Video Core IV® Multimedia Co-Processor Provides Open GL ES 2.0, hardware-accelerated Open VG, and 1080p30 H.264 high-profile decode Capable of 1Gpixel/s, 1.5Gtexel/s or 24GFLOPs with texture filtering and DMA infrastructure
- Memory 512MB SDRAM
- Operating System Boots from Micro SD card, running a version of the Linux operating system
- Dimensions 85 x 56 x 17mm
- Power Micro USB socket 5V, 2A
- 40 GPIO pins

Για την διαπίστωση ύπαρξης εμποδίου , καθώς και την εύρεση της απόστασης του από τον πράκτορα ρομπότ μας, χρησιμοποιήθηκε ένα HC-SR04. Το τσιπ αυτό, με την βοήθεια ενός υπέρηχου σήματος 40KHz μπορεί να μετρήσει την απόσταση που διάνυσε ο υπέρηχος με βάση τον χρόνο που έκανε το σήμα να πάει και να γυρίσει. Εικόνα-2.

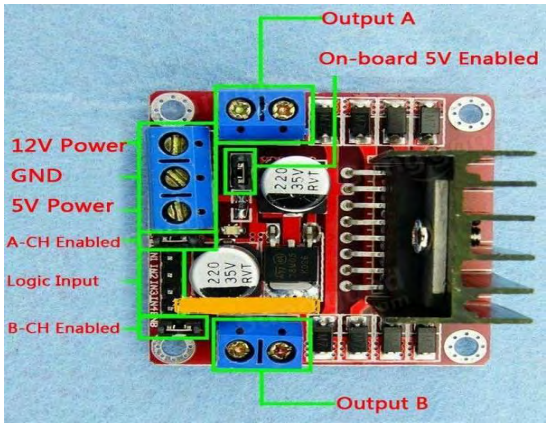
Εικόνα 2

Παρακάτω παρουσιάζεται και το διάγραμμα χρόνου του τσιπ. Εικόνα-3



Εικόνα 3

Για την κίνηση των 2 κινητήρων χρησιμοποιήθηκε ένα L298n Motor Driver Module όπως φαίνεται στην παρακάτω εικόνα-4



Εικόνα 4

Στην εικόνα-5 που ακολουθεί φαίνεται ο πίνακας αληθείας της πλακέτας L298n και μας δείχνει την λογική λειτουργία των κινητήρων.

Motor A Truth Table

ENA	IN1	IN2	Description
0	X	X	Motor A is off
1	0	0	Motor A is stopped (brakes)
1	0	1	Motor A is on and turning backwards
1	1	0	Motor A is on and turning forwards
1	1	1	Motor A is stopped (brakes)

Motor B Truth Table

ENB	IN1	IN2	Description
0	X	X	Motor B is off
1	0	0	Motor B is stopped (brakes)
1	0	1	Motor B is on and turning backwards
1	1	0	Motor B is on and turning forwards
1	1	1	Motor B is stopped (brakes)

www.MechatronicsWorkshop.com

Εικόνα 5

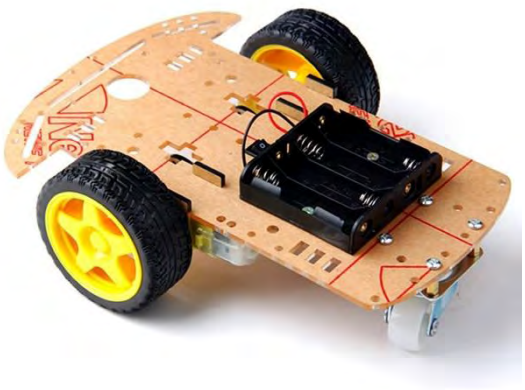
Παράλληλα, για να μπορεί ο αισθητήρας HC-SR04 να περιστρέφεται χρησιμοποιήθηκε ένας μικρός κινητήρας τύπου servo SG90 στον οποίο έχει τοποθετηθεί ο αισθητήρας. Η εικόνα-6 δείχνει έναν τέτοιο μικρό κινητήρα.

Εικόνα 6

Τέλος ως βάση επιλέχτηκε ένα Smart Car Kit με 2 ρόδες το οποίο περιείχε τα εξής:

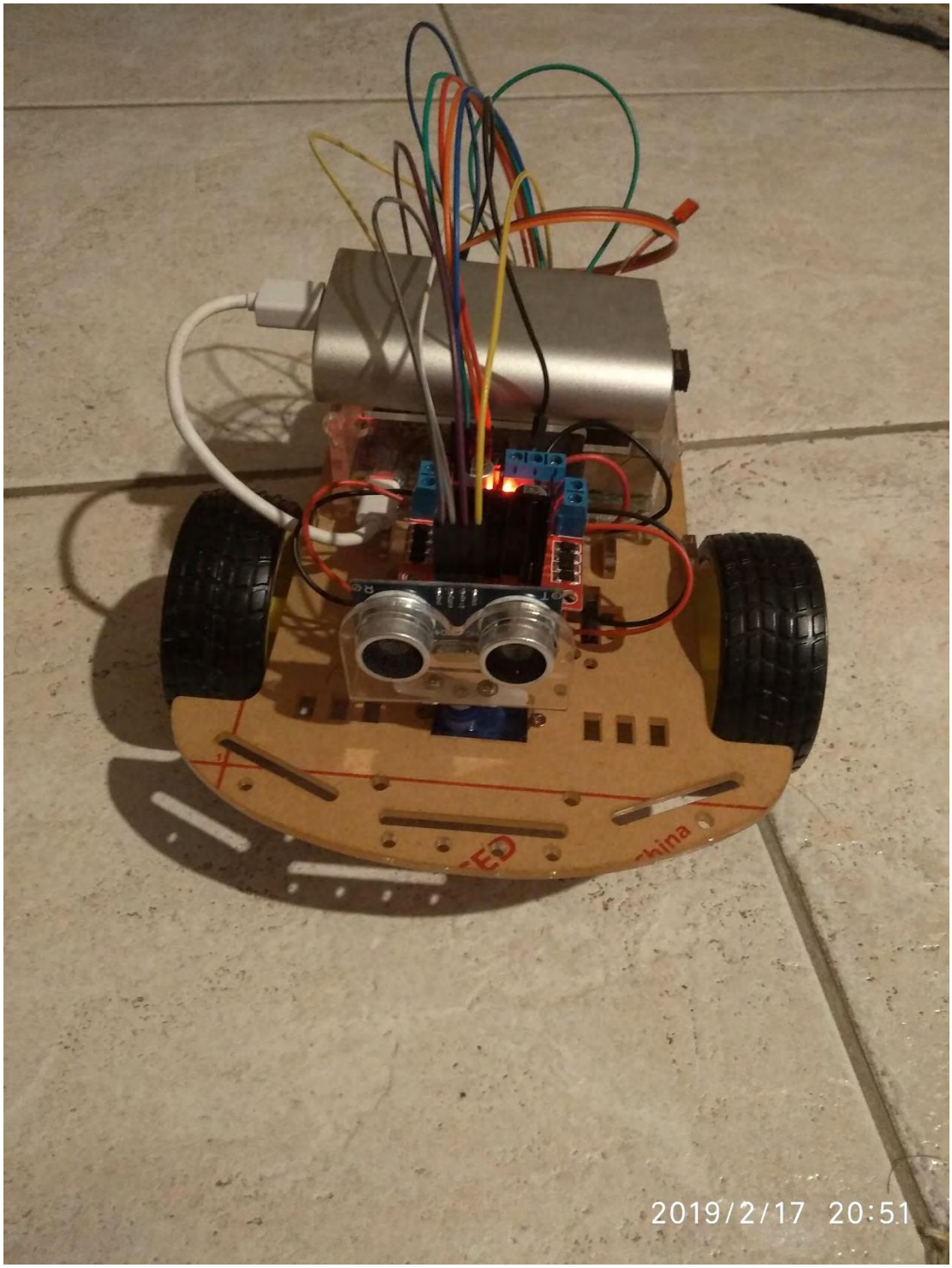
- 1 σασί
- 2 ρόδες
- 2 μικρούς κινητήρες συνεχόμενης τάσεως

Στην εικόνα 7 που ακολουθεί εμφανίζεται το Car Kit.



Εικόνα 7 σασί δίκυκλο.

Στην εικόνα 8 που ακολουθεί , φαίνεται ολοκληρωμένη η κατασκευή μας.



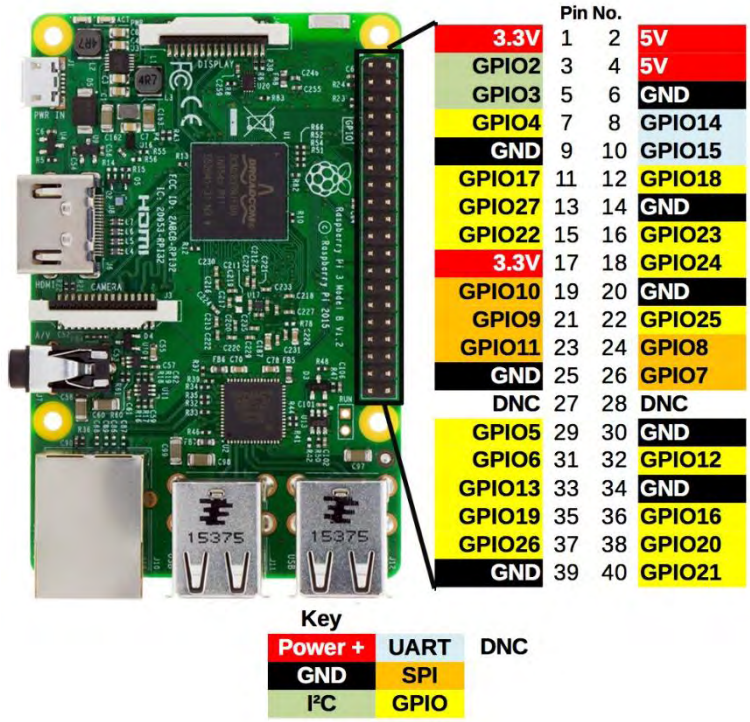
Εικόνα 8 Ολοκληρωμένο Ρομπότ

Κεφάλαιο 4

Συνδεσμολογία

Κατά την συναρμολόγηση του πράκτορα–ρομπότ μας , ακολουθήθηκε η εξής συνδεσμολογία.

Το Raspberry Pi διαθέτει 40 pin από τα οποία τα δύο είναι έξοδοι τάσεως 5V, δύο έξοδοι τάσεως 3.3 V , οκτώ pinγείωσης και τα υπόλοιπα 28 pin(GPIO) γενικού σκοπού. Μπορούν να χρησιμοποιηθούν τόσο ως έξοδοι σήματος όσο και είσοδοι, HIGH (3.3 V) η LOW(0V).



Εικόνα 9

4.1 Σύνδεση του Servomotor SG 90, Εικόνα-6, με το Raspberry Pi:

Το SG 90, διαθέτει τρία καλώδια που συνδέονται ως εξής. Συνδέουμε το καλώδιο ρεύματος (κόκκινο καλώδιο) στην έξοδο 5 Volt του Raspberry Pi(PIN 4), το καλώδιο γείωσης (γκρι καλώδιο) στο PIN γείωσης(PIN 6) και το καλώδιο σήματος (πορτοκαλί καλώδιο) στο GPIO14 (PIN 8).

4.2 Σύνδεση του Ultrasonic Sensor HC-SR 04,Εικόνα-2, με το Raspberry Pi:

Το HC-SR 04 διαθέτει 4 PIN(VCC,TRIGGER,ECHO,GND)όπως φαίνεται στην εικόνα-2. Για να το συνδέσουμε χρησιμοποιούμε μία μικρή πλακέτα breadboard, στην οποία έχουμε υλοποιήσει ένα μικρό κύκλωμα διαιρέτη τάσεως ,χρησιμοποιώντας δύο αντιστάσεις 1ΚΩ και 2ΚΩ με σκοπό την μείωση της τάσης από 5 σε 3.3 Volt ώστε να προφυλάξουμε το Raspberry Pi. Συνδέουμε το PIN Trigger στο GPIO 04 και το VCC στο PIN 02 έξοδος 5 Volt. Ενώ τα PIN Echo και GND συνδέονται μέσω του breadboard στα PIN 11 και 39 αντίστοιχα.(GPIO 17, GND)

4.3 Σύνδεση του L298N ,εικόνα-4 , με το Raspberry Pi:

Το L298Νδιαθέτει 4 PIN εισόδου (IN1,IN2,IN3,IN4) τα οποία συνδέονται σε 4 GPIO PINS του Raspberry PI. Πιο συγκεκριμένα τα GPIO 16, GPIO 19, GPIO 20 και GPIO 26 (PIN 36,35,38 και 37)αντίστοιχα.

Η είσοδος της γείωσης (GND) συνδέεται με το PIN 9(GND) του Raspberry PI ενώ η τροφοδοσία του τσιπ γίνεται από τέσσερις μπαταρίες τύπου ΑΑ των 1,5 Volt συνολική ισχύς 6 Volt.

Επίσης στις υποδοχές εξόδου Α και εξόδου Β που φαίνονται στην εικόνα-4 συνδέονται τα μοτέρ Α και μοτέρ Β τύπου συνεχόμενης τάσεως αντίστοιχα. Τέλος έχει συνδεθεί και ένας απλός διακόπτης σε σειρά με τις μπαταρίες για ευκολότερο έλεγχο της τροφοδοσίας.

Κεφάλαιο 5

Προγραμματισμός Του Ρομπότ

Ο προγραμματισμός του πράκτορα έγινε με κώδικα γραμμένο στην γλώσσα Python[9]. Η Python είναι μια πολύ υψηλού επιπέδου γλώσσα προγραμματισμού με πολλές χρήσεις στην σύγχρονη εποχή των υπολογιστών. Επιλέχτηκε, επειδή το λειτουργικό σύστημα Raspbian OS της πλακέτας Raspberry Pi [10]διαθέτει προ-εγκατεστημένη την Python καθώς και όλες της βιβλιοθήκες της.

5.1 Η δική μου εκδοχή του D*Lite

Ο αλγόριθμος που υλοποιείται στην εργασία αυτήν, είναι ο DstarLite[8] και ανήκει στην οικογένεια των αλγορίθμων εύρεσης συντομότερου μονοπατιού μεταξύ δύο σημείων. Επιλέχτηκε επειδή ικανοποιεί ιδανικά τις απαιτήσεις για λίγους υπολογισμούς καθώς το Raspberry Pi διαθέτει μικρή μνήμη και μικρή υπολογιστική ισχύ.

Ακολουθεί Ψεύδο-κώδικας Αλγορίθμου DstarLite:

Συνάρτηση $\boldsymbol{Main}()$:

1. $Initialize()$;
2. $ComputeShortestPath()$;
3. /*$if(g(s_{start}) = \infty$ then there is no Known Path*/
4. $CreatePath()$;
5. $MoveAcrossPath()$;
6. Move to s_{start};
7. Scan for changed edge costs due to new objects;
8. If any edge costs changed
9. For all directed edges (u, v) with changed costs
10. Update edge cost $c(u, v)$;
11. $UpdateVertex(u)$;
12. $ComputeShortestPath()$;

13. $MoveAcrossNewPath()$;

Συνάρτηση $\boldsymbol{Initialize}()$:

14. $U = \emptyset$;
15. $for\ all\ s \in S\ rhs(s) = g(s) = \infty$;
16. $rhs(s_{goal}) = 0$;
17. $U.Insert\left(s_{goal}, CalculateKey(s_{goal})\right)$;

Συνάρτηση $\boldsymbol{ComputeShortestPath(U, s_{start}, s_{goal})}$:

18. $while(U.TopKey(\ ) < CalculateKey(s_{start})\ OR\ rhs(s_{start})\ !=\ g(s_{srart}))$
19. $u = U.Pop()$;
20. $if\ (g(u) > rhs(u))\ then$
21. $\quad g(u) = rhs(u)$;
22. $\quad for\ all\ s \in\ Pred(u)$
23. $\quad\quad UpdateVertex(s)$;
24. $else$
25. $\quad g(u) = \infty$;
26. $\quad for\ all\ s \in Pred(u)$
27. $\quad\quad UpdateVertex(s)$;
28. $\quad UpdateVertex(u)$;

Συνάρτηση $\boldsymbol{CalculateKey(s)}$:

29. $return\ \min\left(g(s), rhs(s)\right)$;

Συνάρτηση $\boldsymbol{UpdateVertex(s)}$:

30. $if\ (s \neq s_{goal})\ then$
31. $\quad for\ all\ s' \in Succ(s)$
32. $\quad\quad min_{rhs} = \min\left(\infty, g(s) + C(s', s)\right)$;
33. $rhs(s) = min_{rhs}$;
34. $if\ (s \in U)\ then$
35. $\quad U.Remove(s)$;
36. $else\ if\ \left(g(s) \neq rhs(s)\right)\ then$
37. $\quad U.Insert\left(s, CalculateKey(s)\right)$;

5.2 Επεξήγηση του αλγορίθμου

Αρχικά, ο αλγόριθμος καλεί την συνάρτηση $Initialize()$ και αρχικοποιεί όλες τις g-τιμές και rhs-τιμές ίσες με ∞ {γραμμή 15}. Στη συνέχεια θέτει το rhs της κορυφής στόχου ίσο με $rhs(s_{goal}) = 0$ καθώς η αναζήτηση ξεκινάει από τον τέλος προς την αρχική κορυφή s_{start} {γραμμή 16}.

Έπειτα τοποθετείται η κορυφή στόχος s_{goal} στην ουρά για διεύρυνση με την προτεραιότητα που έχει υπολογιστεί. Επειδή ο σκοπός αυτής της εκδοχής είναι η χρησιμοποίηση του ρομπότ σε κάποιο δωμάτιο ή περιορισμένο χώρο δεν απαιτείται να είναι βέλτιστος ο αλγόριθμος. Έτσι για λόγους απλότητας έχει επιλεγεί για το κλειδί της κάθε κορυφής s, η ελάχιστη τιμή g ή $rhs(s) : \min(rhs(s), g(s))$.

Επιπλέον για την ουρά προτεραιότητας έχει χρησιμοποιηθεί μια απλή ουρά και όχι μια τύπου Fibonacci[11] πάλι για λόγους απλότητας.

Στη συνέχεια, αφού έχει επιστρέψει η $ComputeShortestPath()$, εάν η τιμή g της αρχικής κορυφής είναι ίση με $g = 0$, τότε δεν υπάρχει διαθέσιμο μονοπάτι για μετάβαση από την s_{start} στην s_{goal}. Σε αντίθετη περίπτωση καλείται η $CreatePath()$ η οποία φτιάχνει το υπολογισμένο μονοπάτι σε βήματα.

Έπειτα, καλείται η $MoveAcrossPath()$ η οποία δίνει στο ρομπότ την εντολή να επιχειρήσει να κινηθεί στο μονοπάτι. Εάν κατά την διάρκεια της κίνησης εντοπιστεί μέσω του αισθητήρα άγνωστο εμπόδιο, τότε γίνεται ενημέρωση μέσω της $UpdateVertex()$ και ο αλγόριθμος ξανά καλεί την διαδικασία $ComputeShortestPath()$. Υπολογίζεται νέο μονοπάτι με τα νέα δεδομένα του περιβάλλοντος και το ρομπότ ξανά επιχειρεί μετάβαση στον στόχο.

Ο αλγόριθμος είναι σωστός και τερματίζει επιτυχώς βρίσκοντας το συντομότερο μονοπάτι προς την κορυφή στόχο, ή τερματίζει αποφασίζοντας πως δεν υπάρχει μονοπάτι.

Κεφάλαιο 6

Μελλοντική εργασία

Υπάρχουν πολλά πράγματα που μπορούμε να κάνουμε στο μέλλον με τον πράκτορα ρομπότ μας για να του δώσουμε περισσότερες δυνατότητες ,όπως φωνητικές εντολές και ανταποκρίσεις.

Παράλληλα, θα είναι εξαιρετικά ενδιαφέρον η προσάρτηση μίας κάμερας ώστε να μπορεί ο πράκτορας χρησιμοποιώντας βιβλιοθήκες Computer Vision να εντοπίζει διάφορα αντικείμενα σε φωτογραφίες ή βίντεο και να τα κατηγοριοποιεί ή να αποφασίζει αυτόνομα ποια κίνηση θα κάνει. Παραδείγματος Χάριν, αυτόνομη οδική συμπεριφορά και λήψη αποφάσεων , αναγνωρίζοντας τυχόν ταμπέλες προορισμού σε διασταυρώσεις.

Κεφάλαιο 7

Συμπεράσματα

Από την μελέτη των παραπάνω αλγορίθμων, (Dijkstra, A* , LPA*, D* και D*Lite) που στοχεύουν στην επίλυση του προβλήματος της εύρεσης συντομότερου μονοπατιού καθώς και εν τέλη στην επίτευξη ασφαλούς πλοήγησης σε γνωστό, μερικώς γνωστό ή εντελώς άγνωστο περιβάλλον , αποφεύγοντας τυχόν εμπόδια προκύπτουν τα εξής συμπεράσματα.

Ο αλγόριθμος Dijkstra που είναι από τους πρώτους αλγορίθμους που χρησιμοποιήθηκαν για εύρεση συντομότερου μονοπατιού, είναι συστηματικός όμως αρκετά αργός και απαιτεί μεγάλο κόστος, ιδίως όταν εφαρμόζεται σε μεγάλο περιβάλλον.

Στη συνέχεια ασχοληθήκαμε με τον αλγόριθμο A*, έναν επίσης συστηματικό αλγόριθμο ο οποίος χρησιμοποιείται ευρέως. Επιδιώκει την μείωση του αριθμού των απαιτούμενων κορυφών χρησιμοποιώντας ευρετική αναζήτηση. Είναι περισσότερο αποδοτικός από τον Dijkstra όμως υστερεί και αυτός σε μεγάλο περιβάλλον αφού κρατάει στην μνήμη όλους τους κόμβους προκειμένου να τους επεξεργαστεί. Ο D*, είναι ένας αποδοτικός αλγόριθμος βασισμένος στον A*[3], χρησιμοποιεί σταδιακή ευρετική αναζήτηση που του

επιτρέπει να είναι πολύ αποδοτικός και σε μεγάλο περιβάλλον, είτε γνωστό είτε άγνωστο. Χρησιμοποιεί δείκτες οπισθοδρόμησης ως έξτρα πληροφορία στην μνήμη.

Ο αλγόριθμος LPA* ,ενώ εμφανίζει πολλές ομοιότητες με τον A*[7] αποτελεί μια σταδιακή εκδοχή του, επεξεργάζεται μόνο τους απαραίτητους κόμβους για την εύρεση ελαχίστου μονοπατιού. Είναι πολύ αποτελεσματικός αλλά ξεπερασμένος.

Τέλος για τον αλγόριθμο D*Lite συμπεραίνουμε ότι είναι ο πιο αποδοτικός ,ειδικά εάν η ουρά προτεραιότητας υλοποιηθεί με σωρό, ο οποίος χρησιμοποιεί για προτεραιότητες αριθμούς της ακολουθίας Fibonacci[11], βρίσκει πάντα λύση ενώ παράλληλα απαιτεί και λιγότερη μνήμη. Ικανός να επιλύσει κάθε πρόβλημα πλοήγησης ακόμη και σε άγνωστο περιβάλλον. Χρησιμοποιείται σε όλα τα ρομπότ της σύγχρονης εποχής όπως παραδείγματος χάριν τα ρομπότ εξερεύνησης που στέλνει η N.A.S.A στον Άρη.

Ακόμη , πρέπει να σημειωθεί πως ο D*Lite επειδή έχει χαμηλές απαιτήσεις σε μνήμη και υπολογιστική ισχύ συμπεραίνουμε ότι αποτελεί τον ιδανικό αλγόριθμο για υλοποιήσεις πραγματικών ρομπότ και αυτό αποδείχτηκε από την υλοποίηση μας χρησιμοποιώντας την πλακέτα Raspberry pi[10].

Βιβλιογραφία

[1] S. LaValle, *Planning algorithms*, 8th ed. New York (NY): Cambridge University Press, 2014, pp. 28-29.

[2] S. LaValle, *Planning algorithms*, 8th ed. New York (NY): Cambridge University Press, 2014, pp. 32-37.

[3] S. LaValle, *Planning algorithms*, 8th ed. New York (NY): Cambridge University Press, 2014, pp. 37-38.

[4] N. J. Nilsson, “Principles of Artificial Intelligence”, Tioga Publishing Company, 1980.

[5] A. Stentz, “Optimal and Efficient Path Planning for Partially-Known Environments”. Proc. of the IEEE International Conference on Robotics and Automation, May. 1994.

[6] A. Stentz, “The d* algorithm for real-time planning of optimal traverses, Tech. Rep. CMU-RI-TR-94-37, Oct 1994.

[7] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," in IEEE Transactions on Robotics, vol. 21, no. 3, pp. 354-363, June 2005.

[8] S. Koenig and M. Likhachev, "D* lite", in *Eighteenth National Conference on Artificial Intelligence*, Edmonton, Alberta, Canada, 2002, pp. 476-483.

[9] https://www.python.org

[10] https://www.raspberrypi.org

[11] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596-615, 1987. Available: 10.1145/28869.28874.

ΠΑΡΑΡΤΗΜΑ Α

Κώδικας αρχείου Main.py

Στο παράρτημα αυτό δίνεται ο κώδικας του κυρίου προγράμματος που βρίσκεται στο αρχείο main.py:

```
#!/usr/bin/python

from gpiozero import Robot

from dstar import transitionCost, initDstarLite, topKey, computeShortestPath, calculateKey, c

from Distance import getDistance

import heapq

import time

start_time = time.time()

class State:

 'Common base class for all states'

 pathable = True

 previous = None

 neighborList = []

 parents = []

 children = []


 def __init__(self,x,y):

 self.x = x

 self.y = y

 self.g = float('inf')

 self.rhs = float('inf')
```

$$\textbf{def assign\_neighbors}(\textbf{self}):$$

$$'neighbors\ init'$$

$$if\ (self.x\ ==\ 0\ and\ self.y\ ==\ 0):$$

$$self.neighborList\ =\ [(self\ +\ a[0][1]), (self\ +\ a[1][1]), (self\ +a[1][0])]$$

$$elif\ (self.x\ ==0\ and\ self.y\ ==\ 4):$$

$$self.neighborList$$
$$=\ [(self\ +\ State(0,-1)), (self\ +\ State(1,-1)), (self\ +a[1][0])]$$

$$elif\ (self.x\ ==\ 4\ and\ self.y\ ==\ 0):$$

$$self.neighborList$$
$$=\ [(self\ +\ a[0][1]), (self\ +\ State(-1,1)), (self\ +State(-1,0))]$$

$$elif\ (self.x\ ==\ 4\ and\ self.y\ ==\ 4):$$

$$self.neighborList$$
$$=\ [(self\ +\ State(0,-1)), (self\ +\ State(-1,-1)), (self$$
$$+State(-1,0))]$$

$$elif\ ((self.x\ ==\ 0\ and\ self.y\ !=\ 0)\ or\ (self.x == 0\ and\ self.y\ != 4)):$$

$$self.neighborList$$
$$=\ [(self\ +\ a[0][1]), (self\ +\ a[1][1]), (self\ +a[1][0]), (self$$
$$+\ State(0,-1)), (self\ +\ State(1,-1))]$$

$$elif\ ((self.x\ ==\ 4\ and\ self.y\ != 0)\ or\ (self.x\ ==\ 4\ and\ self.y\ != 4)):$$

$$self.neighborList$$
$$=\ [(self\ +\ a[0][1]), (self\ +\ State(-1,1)), (self$$
$$+State(-1,0)), (self\ +\ State(-1,-1)), (self\ +State(0,-1))]$$

$$elif\ ((self.y\ ==\ 0\ and\ self.x\ !=\ 0)\ or\ (self.y == 0\ and\ self.x\ != 4)):$$

$$self.neighborList$$
$$=\ [(self\ +State(-1,0)), (self\ +\ State(-1,1)), (self$$
$$+\ a[0][1]), (self\ +\ a[1][1]), (self\ +a[1][0])]$$

```
elif ((self.y == 4 and self.x != 0) or (self.y == 4 and self.x != 4)):
    self.neighborList
            = [(self + State(-1,0)), (self + State(-1,-1)), (self
            + State(0,-1)), (self + State(1,-1)), (self + a[1][0])]
else :
    self.neighborList
            = [(self + a[0][1]), (self + a[1][1]), (self + a[1][0]), (self
            + State(1,-1)), (self + State(0,-1)), (self
            + State(-1,-1)), (self + State(-1,0)), (self + State(-1,1))]
```

```
def displayState(self):
print ("State ({},{})" .format(self.x ,self.y))
```

```
def __add__(self,other):
    total_x = self.x + other.x
    total_y = self.y + other.y
    if total_x > 4 or total_y > 4:
        #print ("Out of the grid")
        return None
    if total_x < -1 or total_y < -1:
        #print ("Out of the grid")
        return None
    return a[total_x][total_y]
```

```
def __radd__(self, other):

    if other == 0:
      return self
    else:
      return self.__add__(other)


def __str__(self):
return 'State (%d, %d)' % (self.x, self.y)
def move_up (self):
next = self + a[0][1]
next.previous = self
#print ("up", str(next))
return next

def move_down (self):
next = self + State(0, −1)
next.previous = self
#print ("up", str(next))
return next

def move_left (self):
next = self + State(−1,0)
```

```
next.previous = self
#print ("up", str(next))
return next


def move_right(self):
next = self + a[1][0]
next.previous = self
#print ("right", str(next))
return next


def move_diag(self):
next = self + a[1][1]
next.previous = self
#print ("diag", str(next))
return next


def move_diag2(self):
next = self + State(-1, -1)
next.previous = self
#print ("diag", str(next))
return next
```

```
def move_diag3(self):
    next = self + State(-1,1)
    next.previous = self
    #print ("diag", str(next))
    return next

def move_diag4(self):
    next = self + State(1,-1)
    next.previous = self
    #print ("diag", str(next))
    return next

#----code for creating te world and assign neighbohrs for each cell----
a = [[None for x in range(5)] for y in range(5)]
for i in range (5):
    for j in range(5):
        a[i][j] = State(i,j)
for i in range (5):
    for j in range(5):
        a[i][j].assign_neighbors()

#set up the barriers
```

```
a[1][1].pathable = False
a[2][0].pathable = False
a[2][1].pathable = False
#foreach Neighbor assign children and parent so they are all connected
for i in range (5):
  for j in range(5):
    a[i][j].children = a[i][j].neighborList
    a[i][j].parents = a[i][j].children

# - - - - - code to initialize the robot - - - - - - - - - - - - - - - - - - -#
robot = Robot((16,19), (20,26))

# - - - - - code for implementing the replan logic - - - - - - - - #
def replan(path, queue, goal, item):
  print" - - - - - -please wait while replanin - - - - - - - - -"
  heapq.heappush(queue, (calculateKey(path[item]), path[item]))
  computeShortestPath(queue, path[item - 1], goal)
  print" \nall table values after replan calculations"
  for i in range (5):
    for j in range(5):
      print "Node: {} Rhs: {} G: {}".format(str(a[i][j]), a[i][j].rhs, a[i][j].g)
  current = path[item - 1]
  del path [:]
  print"\nReplaning has finished"
```

```
createPath(path, current, goal)
if len(path) > 1:
  print"New path contains these items: "
  for item in range (len(path)):
    print "Item (x, y) is ({}, {}) ". format(path[item]. x, path[item]. y)
    print "{}. pathable = {}". format(path[item], path[item]. pathable)
moveAcrossPath(path, queue, goal)


# − − − − − code for following the path and scan for new obstacles
def moveAcrossPath(path, queue, goal):
  start = path[0]
  for item in range (1, len(path)):
    print "Trying to move to {}". format(path[item])
    if path[item]. pathable == False:
      path[item]. rhs = float('inf')
      print"I found a new obstacle i need to replan following the path is terminated"
      replan(path, queue, goal, item)
      break
    (x, y) = (path[item]. x − start. x , path[item]. y − start. y)
    if (x, y) == (−1, −1):
      start = start. move_diag2()
      print "I moved to {}". format(start)
    elif (x, y) == (−1,0):
      start = start. move_left()
```

```
        print "I moved to {}".format(start)
    elif (x,y) == (0,1):
        start = start.move_up()
        print "I moved to {}".format(start)
    elif (x,y) == (0,-1):
        start = start.move_down()
        print "I moved to {}".format(start)
    elif (x,y) == (-1,1):
        start = start.move_diag3()
        print "I moved to {}".format(start)
    elif (x,y) == (1,-1):
        start = start.move_diag4()
        print "I moved to {}".format(start)
    if (start == path[len(path) - 1]):
        print "I have reached the goal! at {}".format(path[len(path) - 1])


# ----- code main procedure ----- #
def main():
    queue = []
    start = a[4][2]
    goal = a[0][0]
    path = []
    queue = initDstarLite(queue, start, goal)
```

```
computeShortestPath(queue, start, goal)
print"-----------please wait while searching for optimal path----
                    ------- "
print "\nall table values after first calculations"
for i in range (5):
  for j in range(5):
    print "Node: {} Rhs: {} G: {}".format(str(a[i][j]), a[i][j].rhs, a[i][j].g)
createPath(path, start, goal)
for item in range (len(path)):
  print "Item (x, y) is ({},{}) ".format(path[item].x, path[item].y)
path[3].pathable = False
if len(path) > 1:
  print "following the path .....\n"
  moveAcrossPath(path, queue, goal)
    print("--- %s seconds ---" % (time.time() - start_time))
# ----- code to call main ----- #
if __name__ == "__main__":
  main()
```

ΠΑΡΑΡΤΗΜΑ Β

Κώδικας αρχείου dstar.py

Το παράρτημα β περιέχει τον κώδικα του αλγορίθμου DstarLite όπως βρίσκεται στο αρχείο dstar.py :

```
# −∗− coding: cp1253 −∗−

import heapq

def transitionCost(self, other):

  if other.pathable == False:

    return float('inf')

  tc = (other.x − self.x, other.y − self.y)

  if (tc == (1,0)) or (tc == (0,1)) or (tc == (−1,0)) or (tc == (0, −1)):

    return 1

  elif (tc == (1,1)) or (tc == (1, −1)) or (tc == (−1,1)) or (tc == (−1, −1)):

    return 1.4


def topKey(queue):

  #queue.sort()

  # print(queue)

  if len(queue) > 0:

    return queue[0][: 2]

  else:

    # Empty queue

    return (float('inf'), float('inf'))
```

```
def calculateKey(self):
  return (min(self.g, self.rhs))

def initDstarLite(queue, start, goal):
  goal.rhs = 0
  heapq.heappush(queue, (calculateKey(goal), goal))
  return queue

def computeShortestPath(queue, start, goal):
  while(topKey(queue)[0] < calculateKey(start) or start.rhs != start.g):
    u = heapq.heappop(queue)
    #if u[1] == start:
      #print "I just popped start node {}\n".format(start)
      #break
    if (u[1].g > u[1].rhs):
      u[1].g = u[1].rhs
      #print u[1].g
      #Διατρέχω για κάθε πρόγονο του κόμβου που κάνω pop τυπώνω τον πρόγονο
      for parent in range(len(u[1].parents)):
        #print"now i am in parent {} of {}".format(u[1].parents[parent], u[1])
        updateVertex(queue, u[1].parents[parent], goal)
    else:
      u[1].g = float('inf')
      for parent in range(len(u[1].parents)):
```

```
        #print"now i am in parent {} of {}".format(u[1].parents[parent],u[1])
        updateVertex(queue,u[1].parents[parent],goal)
      updateVertex(queue,u[1],goal)

def updateVertex(queue,current,goal):
  if (current != goal) :
    min_rhs = float('inf')
    for child in range(len(current.children)):
      min_rhs
            = min(min_rhs,current.children[child].g
            + transitionCost(current.children[child],current))
    current.rhs = min_rhs
    #print "Length of queue is {}".format(len(queue))
    id_in_queue = [item for item in queue if current in item]
    if id_in_queue != []:
      #print "Length of id_in_queue list is : {}".format(len(id_in_queue))
    #found = False
    #for item in range(len(queue)):
    #  if current == queue[item][1]:
    #    found = True
    #if found == True:
      queue.remove(id_in_queue[0])
    if (current.g != current.rhs):
      heapq.heappush(queue,(calculateKey(current),current))
```

```
def createPath(path, start, goal):
    current = start
    nextToGo = None
    path.append(start)
    while current != goal:
        min_g = float('inf')
        for child in range (len(current.children)):
            if current.children[child].g < min_g :
                min_g = current.children[child].g
                nextToGo = current.children[child]
        if nextToGo == None:
            print("\nThere is No path towards goal")
            break
        path.append(nextToGo)
        current = nextToGo
    if len(path) > 1:
            print("\nOptimal path to target is found!")
```

ΠΑΡΑΡΤΗΜΑ Γ

Κώδικας αρχείου Distance.py

Στο παράρτημα Γ, βρίσκεται ο κώδικας που υπολογίζει την απόσταση με χρήση του αισθητήρα υπερήχων HCSR-04 όπως αναγράφονται στο αρχείο Distance.py

```
import time
import RPi.GPIO as GPIO
def getDistance():
# Use BCM GPIO references
# instead of physical pin numbers
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# Define GPIO to use on Pi
GPIO_TRIGGER = 4
GPIO_ECHO    = 17

# Set pins as output and input
GPIO.setup(GPIO_TRIGGER,GPIO.OUT)  # Trigger
GPIO.setup(GPIO_ECHO,GPIO.IN)      # Echo

# Set trigger to False (Low)
GPIO.output(GPIO_TRIGGER, False)
print("Ultrasonic Measurement")
```

```
# Allow module to settle
time.sleep(0.5)

# Send 10us pulse to trigger
GPIO.output(GPIO_TRIGGER, True)
time.sleep(0.00001)
GPIO.output(GPIO_TRIGGER, False)
start = time.time()

while GPIO.input(GPIO_ECHO) == 0:
 start = time.time()

while GPIO.input(GPIO_ECHO) == 1:
stop = time.time()

# Calculate pulse length
elapsed = stop - start

# Distance pulse travelled in that time is time
# multiplied by the speed of sound (cm/s)
distancet = elapsed * 34300

# That was the distance there and back so halve the value
distance = distancet / 2
```

$print\ ("Distance:", distance)$

$print\ ("Elaspsed\ time:", elapsed)$

$print\ ("Total\ distance:", distancet)$

$\#\ Reset\ GPIO\ settings$

$\#GPIO.cleanup()$

$return(distance)$