ΠΛΗ417 Τεχνητή Νοημοσύνη

1η Προγραμματιστική Εργασία

Ομάδα Εργασίας LAB41732909

Μερσινιάς Μιχαήλ - 2013030057

Τρουλλινός Δημήτριος - 2013030032

1. Ζητούμενα και δοκιμές

1.1 Ζητούμενα

Στο project αυτό μας ζητήθηκε να υλοποιήσουμε αλγορίθμους, σκοπός των οποίων ήταν η εύρεση βέλτισης διαδρομής μεταξύ 2 κόμβων ενός αρχείου γράφου.

Οι αλγόριθμοι που υλοποιήσαμε ήταν οι εξής:

- Uniform Cost Search (UCS) ήταν η επιλογή μας σχετικά με τον αλγόριθμο απληροφόρητης αναζήτησης του ερωτήματος Α, αφού τον θεωρήσαμε την βέλτιστη επιλογή τόσο σε θέμα ταχύτητας και πολυπλοκότητας όσο και σε θέμα πληρότητας.

-IDA* ως αλγόριθμο πληροφορημένης αναζήτησης αναζήτησης του ερωτήματος Α, με τα ευριστικά που θα περιγραφούν εκτενέστερα στις παρακάτω ενότητες της αναφοράς.

-LRTA* ως αλγόριθμο online search του ερωτήματος Β.

Το project υλοποιήθηκε σε γλώσσα προγραμματισμού Java και σε περιβάλλον Eclipse IDE(παρέχεται το exported archive file).

1.2 Δοκιμές

Τα κομμάτια του project στα οποία χρειαστήκαμε δοκιμές ήταν βασικά δύο: Η απληροφόρητη αναζήτηση και τα ευριστικά του IDA*.

Αρχικά, για να καταλήξουμε στον αλγόριθμο Uniform Cost Search δοκιμάσαμε (είτε με έτοιμο κώδικα είτε με δικό μας) και τον συγκρίναμε (είτε θεωρητικά είτε πρακτικά) με τους εξής αλγορίθμους:

-*Breath-First Search:*

Γνωρίζοντας ότι είναι πλήρης και βέλτιστος, δηλαδή μας εγγυάται την βέλτιστη διαδρομή (αν και αυτό συμβαίνει μόνο όταν το path-cost είναι non-decreasing function του node depth, σύμφωνα με την θεωρία), ο αλγόριθμος αυτός ήταν ο πρώτος που σκεφτήκαμε. Η εύκολη υλοποίησή του τον έκανε να φαίνεται μια ιδανική επιλογή, παρόλα αυτά απορρίφθηκε στο τελικό πρόγραμμα διότι η πολυπλοκότητά του ήταν από τις χειρότερες των επιλογών μας (χρονική και χωρική worst-case πολυπλοκότητα O(b^(d+1))).

-*Depth-First Search:*

Αν και η worst-case πολυπλοκότητά του είναι καλύτερη του Breadth-First Search, και πάλι υστερεί έναντι του Uniform Cost Search. Επιπλέον, ο αλγόριθμος αυτός δεν είναι πλήρης και δεν μας εγγυάται την βέλτιστη διαδρομή, επομένως κρίθηκε ακατάλληλος.

-*Depth-Limited Search:*

Ο αλγόριθμος αυτός είναι μια παραλλαγή του προηγούμενου έτσι ώστε να αποφεύγονται οι άπειρες διαδρομές. Παρόλα αυτά, η παρόμοια πολυπλοκότητα αλλά κυρίως το γεγονός ότι δεν είναι ούτε πλήρης ούτε βέλτιστος, δηλαδή δεν μας εγγυάται την βέλτιστη διαδρομή, τον καθορίζει επίσης ακατάλληλο.

-*Iterative Deepening Depth-First:*

Αρχικά είχαμε επιλέξει αυτόν τον αλγόριθμο απληροφόρητης αναζήτησης, θεωρώντας τον ιδανικό αφού είναι πλήρης και βέλτιστος, δηλαδή μας εγγυάται την βέλτιστη διαδρομή. Επιπλέον, τόσο η χρονική όσο και η χωρική πολυπλοκότητά του είναι αρκετά καλές και σίγουρα καλύτερες από αυτές των παραπάνω αλγορίθμων (worst-case χρονική: O(b^d), χωρική: O(b*d)).

-*Dijkstra:*

Ο αλγόριθμος αυτός είναι μια παραλλαγή του UCS, με χειρότερη όμως χρονική και χωρική πολυπλοκότητα που χρησιμεύει στο να βρούμε την βέλτιστη διαδρομή προς κάθε κόμβο, ζητούμενο που δεν επιδιώκουμε στο project αυτό. Για τον λόγο αυτόν, λοιπόν, απορρίφθηκε.

-*Τελικά:* Καταλήξαμε να επιλέξουμε ως αλγόριθμο απληροφόρητης αναζήτησης τον Uniform Cost Search (UCS), διότι έχει πολύ καλή χρονική και χωρική πολυπλοκότητα και ταιριάζει καλύτερα στις ανάγκες του project αφού μας ενδιαφέρει το συνολικό κόστος και όχι ο συνολικός αριθμός βημάτων (γεγονός που μας έκανε να τον θεωρήσουμε προτιμότερο από τον Iterative Deepening Depth-First).

Όσον αφορά το κομμάτι των ευριστικών του IDA* δοκιμάσαμε τα εξής:

-*Pessimistic:*

Ο συγκεκριμένος αλγόριθμος κάνει την υπόθεση ότι για να φτάσουμε στον στόχο(goal) θα χρειαστεί να διασχίσουμε όλους τους κόμβους. Κρατώντας πληροφορία για το βάθος του δέντρου αναζήτησης κάνουμε αυτή την εκτίμηση(για παράδειγμα αν έχουμε βάθος 3 σημαίνει πως έχουμε ήδη διασχίσει 3 κόμβους και άρα θα διασχίσουμε l-3(l:αριθμός κόμβων) κόμβους μέχρι να φτάσουμε στον στόχο). Πολλαπλασιάζουμε την τιμή αυτή με τον μέσο όρο των κόστων της κάθε οδού που έχουμε προϋπολογίσει και το αποτέλεσμα αυτό είναι το ευριστικό h(n). Προφανώς, επειδή μπορεί να υπερεκτιμήσει το υπολοιπόμενο κόστος για τον στόχο, δεν είναι admissible ευριστικό, και άρα δεν θα δώσει απαραίτητα το καλύτερο(optimal) μονοπάτι. Όμως, ένα τέτοιο ευριστικό είναι αρκετά γρήγορο. Στα αρχεία εισόδου που μας δώθηκαν, πάντα γινόταν επιλογή του βέλτιστου μονοπατιού, κάτι που όμως δεν μπορούμε να εγγυηθούμε ότι θα συμβαίνει σε κάθε περίπτωση.

-*Optimistic:*

Για το "αισιόδοξο" ευριστικό, κάνουμε την υπόθεση ότι ο επόμενος κόμβος που θα επιλέξουμε θα είναι ο στόχος μας. Βρίσκοντας τον πιο οικονομικό δρόμο, και επιστρέφοντας το κόστος αυτού, ποτέ δεν θα υπερεκτιμήσουμε το κόστος για τον στόχο (άρα το ευριστικό αυτό θα είναι admissible), όμως το optimistic ευριστικό χρειάζεται ελαφρώς παραπάνω χρόνο. Δηλαδή ενώ θα μας δίνει πάντα το καλύτερο μονοπάτι, θα είναι και λίγο πιο αργό σε σχέση με τον Pessimistic.

-*Dijkstra:*

Ο αλγόριθμος του Dijkstra μοιάζει αρκετά σε λογική με τον UCS αλγόριθμο, με την διαφορά ότι επιστρέφει τα κόστη για κάθε κόμβο. Γι αυτό το λόγο η υλοποίηση του Dijkstra είναι διαφορετική από αυτή του UCS. Πριν τρέξουμε τον IDA*, προϋπολογίζουμε τα ευριστικά για κάθε κόμβο και αποθηκεύουμε τα αποτελέσματα σε ένα Array, το οποίο μετά ανατρέχουμε για να βρούμε την τιμή ενός h(n), όπου n:κόμβος του γράφου. Οι τιμές της συνάρτησης h(n) θα είναι πάντα το ακριβές κόστος για να φτάσουμε στον στόχο, άρα θα έχουμε ιδανική ευριστική συνάρτηση. Το πρόβλημα αυτής όμως είναι ότι χρειάζεται περισσότερο χρόνο και μνήμη σε σχέση με τις παραπάνω προσεγγίσεις.

2. Περιγραφή του Κώδικα

2.1 Διάβασμα αρχείου

Αρχικά, πριν κάνουμε οτιδήποτε, χρειάζεται να διαβάσουμε το αρχείο γράφου που μας δίνεται και να εισάγουμε τα δεδομένα στο πρόγραμμά μας. Αυτό έγινε ως εξής:

Διαβάσαμε αρχικά όλο το περιεχόμενο του αρχείου με όνομα που έχει ήδη καταχωρηθεί στην μεταβλητή Filename ως είσοδο από τον χρήστη (το αρχείο πρέπει να βρίσκεται στον φάκελο Pathfinder_AI).

Έπειτα, μετατρέψαμε όλα τα new line characters (\n) σε ελληνικό ερωτηματικό (;) έτσι ώστε να έχουμε ένα κοινό delimeter παντού.

Τέλος, ακολουθήθηκε μια συγκεκριμένη διαδικασία για να πάρουμε κάθε φορά το κομμάτι περιεχομένου που θέλουμε, χρησιμοποιώντας regular expressions. Πιο συγκεκριμένα, μπορέσαμε να πάρουμε το περιεχόμενο ανάμεσα σε δύο συγκεκριμένα όρια. Η διαδικασία αυτή ακολουθήθηκε και στα source, destination, roads, predictions, actual_traffic.

Για τα predictions και το actual_traffic, όμως, χρειαζόμαστε την πληροφορία αυτή για 80 μέρες. Οπότε επαναλάβαμε εμφωλευμένα την διαδικασία, αλλά την δεύτερη φορά θεωρούμε το περιεχόμενο του prediction, actual_traffic ως το συνολικό περιεχόμενο και το εκάστοτε Day ως το εντός ορίων κομμάτι.

Τέλος, χρησιμοποιήσαμε την βιβλιοθήκη java.util.scanner η οποία μας προσφέρει δύο πολύ σημαντικά εργαλεία.

Πρώτον, την μέθοδο .hasNext() η οποία επιστρέφει true όταν υπάρχει επόμενο στοιχείο και false όταν δεν υπάρχει. Αυτήν χρησιμοποιήσαμε σε μια while loop ως όριο, έτσι ώστε να προσπελάσουμε όλα τα στοιχεία του prediction_per_day[j], για παράδειγμα, αφού πρώτα το περάσουμε στο Scanner.

Δεύτερον, την μέθοδο .useDelimeter η οποία χωρίζει τα στοιχεία σύμφωνα με ένα delimeter (το ελληνικό ερωτηματικό στην περίπτωσή μας) που χρησιμοποιεί regular expressions, έτσι ώστε να είναι μετά εύκολη η αποθήκευσή τους σε πίνακες.

Όσον αφορά το predictions και το actual_traffic, χρησιμοποιήσαμε μια επιπλέον εξωτερική while(j<num_of_days) έτσι ώστε να μπορέσουμε να κάνουμε την ακόλουθη διαδικασία για κάθε μέρα. Το num_of_days είναι μεταβλητή στο πρόγραμμά μας με τιμή 80.

Τέλος, αξίζει να σημειώσουμε ότι για να βρούμε το μέγεθος των πινάκων Road, NodeA, NodeB, cost για τα roads έπρεπε να ξέρουμε τον αριθμό των δρόμων. Αυτό έγινε με κώδικα, ο οποίος βασίστηκε στο γεγονός ότι ένας δρόμος εμφανίζεται κάθε 4 στοιχεία (0, 4, 8, etc).

2.2 Uniform Cost Search

Επιλέξαμε για αλγόριθμο απληροφόρητης αναζήτησης τον Uniform Cost Search επειδή ο γράφος μας είναι βεβαρυμένος, για να βρίσκει πάντα το βέλτιστο αποτέλεσμα. Για την υλοποίηση αυτού δημιουργήσαμε πρώτα τις κλάσεις Node και Vertex. Η κλάση Vertex κρατάει πληροφορία για τον κάθε κόμβο του δέντρου αναζήτησης και η κλάση Node(που περιέχει αντικείμενο Vertex) υλοποιεί ένα δέντρο αναζήτησης.

Στην συνάρτηση του Uniform Cost Search δίνουμε ως είσοδο τον κόμβο του σημείου εκκίνησης, τα δεδομένα εισόδου, την ημέρα που θέλουμε να τρέξουμε, το τρέχων βάθος του αλγορίθμου και δύο λίστες Explored και Expanded. Η λίστα Explored περιέχει όλους τους κόμβους που έχουν ανακαλυφτεί και η λίστα Expanded όλους όσους έχουν ήδη επεκταθεί. Οι λίστες αυτές αξιοποιούνται για να κάνουμε κλάδεμα των κόμβων.

Η λογική που ακολουθούμε είναι να μην επεκτείνουμε κόμβους που έχουν ήδη επεκταθεί, μιας και το μονοπάτι μας θα εμφανίσει κύκλους και σίγουρα δεν θα είναι βέλτιστο. Κόμβοι που είναι υποψήφιοι να επεκταθούν βρίσκονται στην Explored. Κοιτώντας τα περιεχόμενα της Explored και αποκλείοντας τους κόμβους που εμφανίζονται και στην Expanded πετυχαίνουμε το κλάδεμα αυτό.

Έπειτα διαλέγουμε να επεκτείνουμε τον κόμβο με το μικρότερο κόστος g(n) (δηλαδή το κόστος από τον κόμβο εκκίνησης εώς τον τρέχον κόμβο) από αυτούς που ικανοποιούν την προηγούμενη συνθήκη (δηλαδή οι κόμβοι που είναι στην λίστα Explored, αλλά όχι στην λίστα Expanded). Πρακτικά ελέγχουμε όλα τα φύλλα του δέντρου για να βρούμε αυτό με το μικρότερο κόστος και καλούμε αναδρομικά την συνάρτηση υπολογισμού Uniform Cost Search μέχρι να φτάσουμε στον προορισμό μας. Η συνάρτηση επιστρέφει τον κόμβο προορισμού. Ο κάθε κόμβος κρατάει πληροφορία για το κόστος g(n), οπότε μπορούμε άμεσα να εμφανίσουμε το συνολικό κόστος του μονοπατιού.

Το ακριβές δρομολόγιο υπολογίζεται ανατρέχοντας στο δέντρο από τον κόμβο προορισμού προς τον κόμβο εκκίνησης. Η μέθοδος που υλοποιεί τον αλγόριθμο Uniform Cost Search ονομάζεται Uniform_Cost_Search και βρίσκεται στην κλάση Pathfinder.

2.3 IDA*

Η λογική του Iterative Deepening A* είναι παρόμοια με την λογική του UCS. Οι βασικές διαφορές είναι ότι χρησιμοποιούμε μία από τις ευριστικές συναρτήσεις που έχουμε καταστευάσει και ότι κάνουμε χρήση ενός ορίου(limit) κόστους το οποίο ανανεώνεται όταν φτάσουμε το όριο αυτό χωρίς να έχουμε φτάσει στον προορισμό. Τότε ο IDA* θα τρέξει πάλι από τον κόμβο εκκίνησης με το νέο όριο.

Η διαδικασία αυτή επαναλαμβάνεται μέχρι να φτάσουμε στον κόμβο προορισμού. Μια βασική διαφορά με τον αλγόριθμο απληροφόρητης αναζήτησης είναι πως ως κόστος δεν θεωρούμε το g(n) αλλά το f(n)=g(n)+h(n). Η μέθοδος που υλοποιεί τον αλγόριθμο IDA* ονομάζεται IDAstar και βρίσκεται στην κλάση Pathfinder.

2.4 Heuristics

Δοκιμάσαμε στους αλγόριθμους αναζήτησης μας όλα τα ευριστικά που δημιουργήσαμε(Optimistic,Pessimistic,Dijkstra) και αναφέραμε παραπάνω. Παρατηρήσαμε ότι το Pessimistic ευριστικό παρόλο που δεν είναι admissible έβρισκε το optimal path κάθε φορά(το path που πρότεινε ο UCS), όμως αυτό είναι κάτι που σε μία διαφορετική είσοδο δεν θα ίσχυε απαραίτητα. Ο Optimistic αλγόριθμος και ο Dijkstra (ως admissible), προφανώς

έδιναν και αυτοί optimal path στον IDA*, όμως χρειάζονται λίγο περισσότερο χρόνο για να τρέξουν. Στο δικό μας παράδειγμα, επειδή ο γράφος μας δεν είναι πολύ μεγάλος, ήταν αρκετά γρήγορο να χρησιμοποιούμε τον Dijkstra για τον υπολογισμό του κόστους. Όμως σε ένα πολύ μεγαλύτερο αρχείο εισόδου, που θα μας έδινε απαγορευτικό χρόνο εκτέλεσης, το Optimistic ευριστικό θα ήταν προτιμότερο.

Αρκετά έντονες διαφορές βλέπουμε στον LRTA*, καθώς το κόστος αυξάνεται αρκετά σε περίπτωση που δεν έχουμε το ακριβές υπολοιπόμενο κόστος (Dijkstra) αφού σε κάθε άλλη περίπτωση χρειάζεται να εξερευνήσουμε σε μεγαλύτερο βαθμό το δέντρο για να βρούμε μονοπάτι. Όλες οι μέθοδοι που αφορούν τα ευριστικά μας βρίσκονται στην κλάση Heuristic.

2.5 Predictions

Για τον υπολογισμό των εκτιμήσεων, ορίζουμε μία νέα κλάση Speculation, και την πληροφορία για τις εκτιμήσεις μας εμπεριέχουμε σε αντικείμενο τύπου Speculation στην κλάση FileData. Κρατάμε πληροφορία p1(πιθανότητα η εκτίμηση να ήταν σωστή), p2,p3(πιθανότητες η εκτίμηση να μην ήταν ορθή) για κάθε μία από τις τρεις περιπτώσεις κίνησης(heavy, normal, low). Αρχικοποιούμε τα p1,p2,p3 και για τα τρία ενδεχόμενα στις τιμές 0.6, 0.2, 0.2 αντίστοιχα.

Στο τέλος κάθε ημέρας κάνουμε αναθεώρηση των πιθανοτήτων αυτών συγκρίνοντας τις εκτιμήσεις που μας δώθηκαν με τα πραγματικά δεδομένα. Επιπλέον, υπάρχει ένας μετρητής για κάθε περίπτωση κίνησης, έτσι ώστε σε επόμενες μέρες να μην χρειάζεται να ανατρέχουμε ξανά στα δεδομένα των προηγούμενων ημερών.

Πράγματι, η πιθανότητα οι εκτιμήσεις να είναι σωστές προσέγγιζε τα δεδομένα που δώθηκαν(p1=0.6,p2=0.2,p3=0.2). Βέβαια, ακόμη και αν ακολουθούσαν διαφορετική πιθανοτική κατανομή, ο αλγόριθμός μας θα προσαρμοστεί στα δεδομένα με το πέρας των ημερών(όταν δηλαδή θα έχει ένα αντιπροσωπευτικό δείγμα).

2.7 LRTA*

Ο αλγόριθμος online search, LRTA*, επίσης υλοποιήθηκε ως αναδρομικός. Ξεκινάμε από τον αρχικό κόμβο, βρίσκουμε όλους τους γείτονες (με την χρήση της συνάρτησης findNodeIdxs) και από αυτούς διαλέγουμε τον κόμβο-γείτονα με το μικρότερο μικτό κόστος f=cost[n,j] + h[j]. Αυτό σημαίνει ότι θα πρέπει να πάμε στο node με το μικρότερο μικτό κόστος το οποίο ορίζεται ως το κόστος να μεταβούμε απο το current node στο target node προστιθέμενο με το ευριστικό του ίδιου του target node.

Επιπλέον, κρατάμε πίνακα κόστους και πίνακα ευριστικών (h). Ο πίνακας κόστους αρχικοποιείται με βάση τα predictions, αλλά αφού διαλέξουμε τον δρόμο, τον ανανεώνουμε με το actual cost. Ο πίνακας ευριστικών (h) αρχικοποιείται στο μηδέν και όταν χρειάζεται να βρούμε το ευριστικό κάποιου κόμβου, καλούμε την μέθοδο υπολογισμού ευριστικού και αποθηκεύουμε το αποτέλεσμα για μελλοντική χρήση.

Αφού βρούμε το μικρότερο μικτό κόστος f, που είναι αυτό που περιγράφηκε παραπάνω, ορίζουμε έπειτα το ευριστικό h(n) του current node ίσο με το συγκεκριμένο αυτό μικρότερο μικτό κόστος του κόμβου-γείτονα που επιλέξαμε να μεταβούμε. Τέλος, καλούμε αναδρομικά την μέθοδο του LRTA* μέχρι τελικά να φτάσουμε στον κόμβο προορισμού, άρα και τερματίζει ο αλγόριθμος με επιτυχία. Η μέθοδος που υλοποιεί τον αλγόριθμο LRTA* ονομάζεται runLRTA και βρίσκεται στην κλάση Pathfinder.

Αξίζει να σημειώσουμε ότι κάναμε μια βελτιστοποίηση σε σχέση με την “by the book” υλοποίηση του αλγόριθμου LRTA*. Εάν ο αλγόριθμός μας παρατηρήσει ότι ο κόμβος προορισμού (στόχος) είναι γείτονας του current node, τότε μεταβαίνει σε αυτόν και τερματίζει το πρόγραμμα με επιτυχία, ακόμα και αν αυτός δεν έχει το ελάχιστο μικτό κόστος μεταξύ των γειτόνων. Με την βελτιστοποίηση αυτήν παρατηρήσαμε μια βελτίωση του συνολικού μέσου κόστους καθώς και του χρόνου, περίπου της τάξης του 5 - 10%.

2.8 Εκτύπωση και εξαγωγή αποτελεσμάτων σε αρχείο

Τέλος, εκτυπώσαμε και αποθηκεύσαμε σε αρχείο τα στατιστικά στοιχεία που ζητούσε η εκφώνηση, δηλαδή για κάθε ημέρα και για τον κάθε αλγόριθμο:

*Τον αριθμό των κόμβων που επεκτείνονται.

*Τον πραγματικό χρόνο εκτέλεσης κάθε αλγορίθμου.

*Το ακριβές δρομολόγιο (ακολουθία οδών) που προτείνει ο κάθε αλγόριθμος να ακολουθήσει ο εργαζόμενος μαζί με το εκτιμώμενο από τον αλγόριθμο χρονικό κόστος διάτρεξής τους.

*Το εκτιμόμενο από τον αλγόριθμο συνολικό κόστος της προτεινόμενης διαδρομής.

*Το πραγματικό κόστος της προτεινόμενης διαδρομής εκείνη την ημέρα.

*Το συνολικό πραγματικό κόστος μέχρι την συγκεκριμένη μέρα.

*Τον μέσο ημερήσιο χρόνο των προτεινόμενων διαδρομών του κάθε αλγορίθμου μετά απο 3 μήνες (=συγκεντρωτικός χρόνος / 80).

*Το μέσο ημερήσιο πραγματικό κόστος των προτεινόμενων διαδρομών του κάθε αλγορίθμου μετά απο 3 μήνες (=συγκεντρωτικό πραγματικό κόστος / 80).

Αξίζει να σημειώσουμε ότι οι χρονικοί μας υπολογισμοί έγιναν χρησιμοποιώντας την μέθοδο της java ".currentTimeMillis()" και πολλαπλασιάζοντας το αποτέλεσμα επί 0.001 για να το έχουμε σε seconds.

Επιπλέον, η εγγραφή σε αρχείο πραγματοποιήθηκε με την χρήση της βιβλιοθήκης BufferedWriter και της αντίστοιχης μεθόδου write(string), αφού συγκρίνοντας την μέθοδο αυτήν με άλλες (όπως memory map, write paths / text.getBytes) καταλήξαμε στο συμπέρασμα ότι είναι η πιο γρήγορη όσον αφορά το γράψιμο σειρά-σειρά, ειδικά για μεγάλα δεδομένα.

3. Αποτελέσματα

Από τα αρχεία OnlineResults και OfflineResults, αφού τρέξαμε το πρόγραμμα αρκετές φορές και για τα 3 διαφορετικά sample graphs, πήραμε τα εξής αποτελέσματα (παραθέτουμε ένα δείγμα των αποτελεσμάτων):

Τελικός χρόνος εκτέλεσης (Dijkstra Heuristic):

Run 1 (SampleGraph1):
UCS: 0.0050375 sec
IDA:* 0.0703 sec
LRTA:* 0.014174 sec

Run 2 (SampleGraph2):
UCS: 0.0032 sec
IDA:* 0.06039 sec
LRTA:* 0.0147125 sec

Run 3 (SampleGraph3):
UCS: 0.0025875 sec
IDA:* 0.033774 sec
LRTA:* 0.012225 sec

Τελικό μέσο πραγματικό κόστος (Dijkstra Heuristic):

Run 1 (SampleGraph1):
UCS: 115.818125
IDA:* 115.818125
LRTA:* 124.29

Run 2 (SampleGraph2):
UCS: 194.6931249
IDA:* 194.6931249
LRTA:* 173.11

Run 3 (SampleGraph3):
UCS: 101.323125
IDA:* 101.323125
LRTA:* 119.375

Τελικός χρόνος εκτέλεσης (Optimistic Heuristic):

Run 1 (SampleGraph1):
UCS: 0.0029625 sec
IDA:* 0.0418124 sec
LRTA:* 0.0248124 sec

Run 2 (SampleGraph2):
UCS: 0.002625 sec
IDA:* 0.0471374 sec
LRTA:* 0.014175 sec

Run 3 (SampleGraph3):
UCS: 0.002375 sec
IDA:* 0.0429624 sec
LRTA:* 0.0097 sec

Τελικό μέσο πραγματικό κόστος (Optimistic Heuristic):

Run 1 (SampleGraph1):
UCS: 115.818125
IDA:* 115.818125
LRTA:* 2182.8225

Run 2 (SampleGraph2):
UCS: 194.693124
IDA:* 194.693124
LRTA:* 642.3625

Run 3 (SampleGraph3):
UCS: 101.323125
IDA:* 101.323125
LRTA:* 749.22

Τελικός χρόνος εκτέλεσης (Pessimistic Heuristic):

Run 1 (SampleGraph1):
UCS: 0.0026125 sec
IDA:* 0.0409874 sec
LRTA:* 0.006975 sec

Run 2 (SampleGraph2):
UCS: 0.0029125 sec
IDA:* 0.049474 sec
LRTA:* 0.011675 sec

Run 3 (SampleGraph3):
UCS: 0.0026375 sec
IDA:* 0.0347 sec
LRTA:* 0.0075125 sec

Τελικό μέσο πραγματικό κόστος (Pessimistic Heuristic):

Run 1 (SampleGraph1):
UCS: 115.818125
IDA:* 115.818125
LRTA:* 413.0825

Run 2 (SampleGraph2):
UCS: 194.693124
IDA:* 194.693124
LRTA:* 991.5824

Run 3 (SampleGraph3):
UCS: 101.323125
IDA:* 101.323125
LRTA:* 445.6124

4. Συμπεράσματα

Συνοψίζοντας, από την παραπάνω ενότητα της αναφοράς, μπορούμε να βγάλουμε τα εξής συμπεράσματα όσον αφορά τους αλγορίθμους (η σύγκριση των ευριστικών έγινε στις ενότητες 1.2 και 2.4):

* *Uniform Cost Search (UCS):* Ο αλγόριθμος αυτός, ως αλγόριθμος απληροφόρητης αναζήτησης, έχει υψηλό κόστος σε μνήμη. Παρόλα αυτά, έχει πάρα πολύ καλή ταχύτητα και είναι πλήρης και βέλτιστος, δηλαδή μας εγγυάται πάντα ότι θα βρει την βέλτιστη διαδρομή.

* *IDA*:* Ο αλγόριθμος αυτός, αν και πιο αργός, χρειάζεται λιγότερη μνήμη από τον UCS. Επίσης, είναι πλήρης και βέλτιστος υπό μια συνθήκη. Για να έχουμε εγγυημένα βέλτιση διαδρομή, το ευριστικό που χρησιμοποιούμε πρέπει να είναι admissible (admissible=the heuristic function never overestimates the cost of reaching the goal). Αλλιώς, υπάρχει πιθανότητα η διαδρομή να είναι ή να μην είναι βέλτιση. Από τα ευριστικά που χρησιμοποιήσαμε το Optimistic Heuristic και το Dijkstra Heuristic είναι admissible, ενώ το Pessimistic Heuristic δεν είναι.

**LRTA*:* Ο αλγόριθμος αυτός χρειάζεται περισσότερο χρόνο από τον UCS (σχεδόν παρόμοιο με τον IDA*), ενώ οι απαιτήσεις του σε μνήμη είναι αρκετά μικρότερες. Όπως και στον IDA*, χρειαζόμαστε admissible heuristics για τον ίδιο λόγο που αναφέραμε προηγουμένως. Επιπλέον, όσο καλύτερα είναι τα initial heuristics, τόσο μικρότερο μέσο κόστος και τόσο πιο γρήγορος είναι ο αλγόριθμος όσον αφορά τον χρόνο μέχρι να βρεθεί η βέλτιστη διαδρομή. Το μέσο κόστος μπορεί να είναι αρκετά μεγάλο εάν τα αρχικά ευριστικά μας δεν είναι καλά. Για τον λόγο αυτό, προτιμούμε το Dijkstra Heuristic στον LRTA* διότι τόσο το Optimistic όσο και το Pessimistic παράγουν αρκετά μεγάλα τελικά μέσα κόστη.

Σε έναν πολύ μεγάλο γράφο, όμως, στον οποίον η χρήση του Dijkstra θα είναι απαγορευτική, θα προτιμήσουμε Optimistic Heuristic (ως admissible heuristic). Σε κάθε περίπτωση, η βέλτιστη διαδρομή θα βρεθεί εφόσον το ευριστικό είναι admissible.

Ο LRTA*, ως αλγόριθμος online αναζήτησης, μας δίνει το πλεονέκτημα να γνωρίζουμε το πραγματικό κόστος (actual cost) μιας διαδρομής εφόσον την έχουμε διασχίσει έστω και μία φορά. Το γεγονός αυτό ίσως να τον καθιστά πολύ καλό όταν υπάρχουν μεγάλες αποκλίσεις μεταξύ predictions και actual costs, αφού ενώ οι offline αλγόριθμοι θα πέσουν έξω, ο LRTA* θα μας δώσει καλύτερη βέλτιστη διαδρομή.

**Online vs Offline:* Επομένως, συγκρίνοντας τον LRTA* (online) με τους UCS και IDA* (offline), συμπεραίνουμε ότι υπάρχει ένα trade-off ανάμεσα σε ταχύτητα/ελάχιστο μέσο κόστος, όπου υπερτερούν οι offline αλγόριθμοι UCS και IDA*, και σε μνήμη/μικρότερη τυχαιότητα όσον αφορά τις εκτιμήσεις (βασιζόμαστε μονάχα εν μέρη σε αυτές, αφού γνωρίζουμε τα πραγματικά κόστη των διαδρομών που έχουμε διασχίσει), όπου υπερτερεί ο online αλγόριθμος LRTA*.