Πολυτεχνείο Κρήτης

Σχολή ΗΜΜΥ

Τεχνητή Νοημοσύνη

1η Προγραμματιστική Εργασία

Συνοδευτική Αναφορά

Στρατάκης Ανδρέας – 2018030179

Πέτρου Δημήτριος – 2018030070

1. ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ

Η υλοποίηση της άσκησης επιλέχθηκε να γίνει σε Java κυρίως για λόγους ταχύτητας, για οργανωμένη συγγραφή κώδικα αλλά ταυτόχρονα και εξαιτίας του πλήθους εργαλείων που παρέχει όπως δομές δεδομένων και αλγόριθμοι ταξινόμησης.

Τρόπος εκτέλεσης - Παραγόμενα αρχεία - Προϋποθέσεις εκτέλεσης

Για την εκτέλεση του προγράμματος σε JRE είναι απαραίτητο να έχει προηγηθεί compilation των αρχείων `.java` που βρίσκονται εντός του `/src` στα εκάστοτε packages (dirs).

Compilation:
```
cd src
javac core/Console.java
```
Run:
```
cd src
java core.Console %arg%
```

όπου `%arg%` το όνομα κάποιου δοκιμαστικού αρχείου που βρίσκεται εντός του /src/input.

Παράδειγμα εκτέλεσης μετά από επιτυχημένο compilation:
`java core.Console sampleGraph1.txt`

Ανάγνωση και αποθήκευση δεδομένων

Πολύ σημαντική ήταν η σωστή αποθήκευση δεδομένων στη μνήμη ώστε όταν γίνεται αναφορά σε ένα αντικείμενο να μην υπάρχουν άλλα αντίγραφα αυτού. Είναι προφανές ότι 2 δρόμοι δε μπορούν να έχουν το ίδιο όνομα. Ωστόσο πολλοί δρόμοι μπορεί να έχουν κοινούς αρχικούς ή τελικούς κόμβους. Χρησιμοποιήθηκαν λοιπόν ευρετήρια με κλειδιά τα ονόματα των δρόμων και των κόμβων `HashMap<String, Node> nodes` και `HashMap<String, Road> roads`. Κατα την επαναληπτική ανάγνωση της κάθε γραμμής από το αρχείο εισόδου ελέγχεται κάθε φορά εάν οι εκάστοτε κόμβοι που διαβάστηκαν σε εκείνη τη γραμμή υπάρχουν ήδη καταχωρημένοι στη λίστα `HashMap<String, Node> nodes`. Σε περίπτωση που δεν υπάρχουν καταχωρούνται και στην συνέχεια ο κάθε ένας απόκτα reference για τον άλλο ως γείτονα στο `HashMap<Road, Node> neighbors`. Ο έλεγχος πραγματοποιείται με `try – catch` της μεθόδου `get(name: String) : Node` της διεπαφής `java.util.Map`. Αφού το πρόγραμμα έχει λάβει γνώση της πληροφορίας που του δίνεται και έχει «μεταφράσει στη δική του γλώσσα» το αρχείο εισόδου δημιουργώντας ένα ιδεατό map, συνεχίζει στην προσομοίωση με τους επιλεγμένους αλγορίθμους.

Οργάνωση δεδομένων

Εντός της κλάσης `Node.java` εντοπίζονται ορισμένες δομές οι οποίες έχουν οργανωμένες τις πληροφορίες εισόδου σε διάφορες μορφές σύμφωνα με τις ανάγκες που παρουσιάστηκαν κατά τη συγγραφή του κώδικα. Παρατηρήθηκε ότι μπορεί να υπάρχουν παραπάνω από 1 δρόμος που συνδέει 2 κόμβους γεγονός που οδήγησε στη δημιουργία του `HashMap<Road, Node> neighbors` για κάθε κόμβο. Προκειμένου να απαλλαχθούν οι αλγόριθμοι από υπολογισμούς δρόμων με το ελάχιστο κόστος ανά 2 κόμβους, κάθε μέρα υπολογίζονται για κάθε κόμβο οι φθηνότεροι δρόμοι που τον συνδέουν με τους γείτονες του και τα δεδομένα αυτά διατηρούνται στο `HashMap<Node, Road> cheapestRoads.`

2. ΠΛΗΡΟΦΟΡΗΜΕΝΗ & ΑΠΛΗΡΟΦΟΡΗΤΗ ΑΝΑΖΗΤΗΣΗ

- Dijkstra - Απληροφόρητη αναζήτηση

Ο αλγόριθμος του Dijkstra χρησιμοποιείται για τον υπολογισμού του κόστους της συντομότερης διαδρομής από τον κόμβο αφετηρίας προς όλους τους άλλους κόμβους του γράφου. Για να είναι εφικτή η υλοποίηση ο κάθε κόμβος κρατάει πληροφορία για τον φθηνότερο predecessor του και για και το κόστος διαδρομής από τον κόμβο έναρξης μέχρι και τον εαυτό του . Ο αλγόριθμος επίσης διατηρεί δομή αποθήκευσης για τους κόμβους που έχει επισκεφθεί. Ξεκινώντας από τον κόμβο αφετηρίας, με μια iterative δομή «επισκέπτεται» όλους τους γειτονικούς του κόμβους. Αφού υπολογίσει την απόσταση (προβλεπόμενο κόστος διάσχισης ανά ημέρα) μεταξύ του parent και του child κόμβου την αναθέτει υπό την συνθήκη ότι είναι η μικρότερη ή η μόνη υπάρχουσα μέχρι εκείνη τη στιγμή (≠∞), που έχει προκύψει για τον εκάστοτε child κόμβο. Αντίστοιχα ενημερώνει και τον predecessor. Στη συνέχεια επιλέγει να επισκεφθεί τον κόμβο με την επόμενη μικρότερη απόσταση από την αφετηρία που δεν έχει ήδη επισκεφθεί. Η διαδικασία συνεχίζεται μέχρις ότου ο αλγόριθμος να έχει επισκεφθεί όλους τους κόμβους του γράφου. Για την κατασκευή του path είναι απαραίτητη η δημιουργία μιας αλληλουχίας διαδοχικών κόμβων. Ξεκινώντας από τον κόμβο προορισμού με επανάληψη επιλέγεται να συμμετάσχει στο μονοπάτι ο φθηνότερος προηγούμενος κόμβος, δηλ. ο predecessor, μέχρι να προκύψει ο κόμβος αφετηρίας. Το path κόμβων μετατρέπεται μέσω της συνάρτησης `findRoadPath(LinkedList<Node>: path) : LinkedList<Road>` σε path δρόμων και υπολογίζεται το συνολικό κόστος διαδρομής σύμφωνα με τις προβλέψεις της ημέρας καθώς και με βάση τα πραγματικά κόστη διάσχισης των δρόμων.

- Heuristics - Εργαλείο πληροφορημένης αναζήτηση

Οι αλγόριθμοι που υλοποιήθηκαν στην συνέχεια χρειάστηκε να γνωρίζουν μια εκτίμηση για την απόσταση ενός κόμβου του γράφου μέχρι τον τελικό προορισμό. Ένας αλγόριθμος Dijkstra επιλέχθηκε για να κάνει τον υπολογισμό των ευρυστικών τιμών που χρειάζονταν. Πιο συγκεκριμένα κλήθηκε η συνάρτηση Dijkstra με αρχικό κόμβο το destination. Η κλάση `IterativeDeepeningAstar.java` διατηρεί `HashMap<Node, Double>: heuristics.` Οι προβλέψεις για τα κόστη διάσχισης των δρόμων αλλάζουν ανά τις ημέρες, οπότε κάποιος θα σκεφτόταν ότι τα heuristic values θα πρέπει να αλλάζουν από μέρα σε μέρα. Κάτι τέτοιο ωστόσο δεν θα ήταν τόσο cost-efficient καθώς και μη απαραίτητο εφόσον χρειάζεται μια προσεγγιστική τιμή στο κόστος διαδρομής και όχι το βέλτιστο αφού ο αλγόριθμος έχει απλά ευρυστικό χαρακτήρα. Επίσης οι μεταβολές μεταξύ προβλέψεων και πραγματικών τιμών είναι σχετικά μικρές. Ο Dijkstra επιλέχθηκε με την προοπτική ότι η λύση που παρέχει είναι πάντα η βέλτιστη και τα heuristic values θα ήταν όσο το δυνατόν πιο ακριβή, γεγονός το οποίο δικαιώθηκε από τα αποτελέσματα του IDA* τα οποία ήταν αμιγώς καλύτερα από αυτά, κάποιων πειραματισμών με χρήση heuristic τιμών που είχαν προκύψει από έναν BFS αλγόριθμο (ο BFS χρησιμοποιήθηκε στο πλαίσιο πειραματισμού, ωστόσο αργότερα απορρίφθηκε εντελώς).

- Iterative Deepening A* (IDA*) - Πληροφορημένη αναζήτηση

Ο αλγόριθμος IDA* χρησιμοποιήθηκε για την κατασκευή ενός μονοπατιού με το μικρότερο κόστος διαδρομής από τον αρχικό προς τον τελικό κόμβο χωρίς να δοκιμάζει όλες τις επιλογές γειτονικών κόμβων. Η ιδιαιτερότητα του είναι ότι χρησιμοποιεί μια heuristic function ώστε να αξιολογήσει το κόστος της διαδρομής που υπολείπεται από έναν κόμβο προς τον κόμβο προορισμού με στόχο να κάνει τη βέλτιστη επιλογή. Με κατάλληλες δομές διατηρείται πληροφορία για το path κόμβων που βρίσκει ο αλγόριθμος (`path : LinkedList<Node>`), για τον κόμβο ο οποίος επισκέπτεται εκείνη τη στιγμή και για το εκτιμώμενο κόστος διαδρομής f (root...curNode...goal). Ο αλγόριθμος επιλέχθηκε να χωρίζεται σε 2 μεθόδους, μια συνάρτηση εκτέλεσης τού `execute(Node source, Node destination, Day day)` και μια συνάρτηση αναδρομικής αναζήτησης `search(double cost, double bound, Node destination): double` . Πιο συγκεκριμένα ο αλγόριθμος ξεκινώντας εισάγει την αφετηρία ως τον μοναδικό κόμβο στο path και θέτει ένα bound κόστους εντός του οποίου αναμένεται να βρίσκεται το κόστος διαδρομής το οποίο θα υπολογίζεται. Είναι αυτονόητο ότι το κόστος διαδρομής αρχικά είναι μηδενικό. Η ροή εκτέλεσης περνάει στην αναδρομική μέθοδο. Ως επισκεπτόμενος κόμβος λογίζεται ο τελευταίος κόμβος του path. Υπολογίζεται αρχικά το κόστος με βάση το heuristic του κόμβου και βάσει του κόστους της έως τωρα διαδρομής. Πραγματοποιείται έλεγχος αν το εκτιμώμενο κόστος έχει ξεπεράσει το bound ώστε να αποκλειστούν διαδρομές οι οποίες είναι ιδιαίτερα κοστοβόρες. Αν βρεθεί ο προορισμός (`FOUND = NEGATIVE_INFINITY`) επιστρέφει αναδρομικά προς τα πίσω και η ροή εκτέλεσης μεταβιβάζεται στην 1η μέθοδο. Διαφορετικά καλείται ξανά η `search` για όλους τους γειτονικούς κόμβους. Αν δεν προκύψει ικανοποιητικό αποτέλεσμα, ο κόμβος ή οι κόμβοι αυτοί δεν μπορούν να χρησιμοποιηθούν, αφαιρούνται από το path και η διαδικασία συνεχίζεται. Όταν βρεθεί ο προορισμός, μετατρέπεται το path κόμβων σε path δρόμων μέσω της `findRoadPath(LinkedList<Node>: path) : LinkedList<Road>.`

3. ONLINE ΑΝΑΖΗΤΗΣΗ

Learning Real Time A* (LRTA*)

Ο αλγόριθμος LRTA υλοποιήθηκε για την κατασκευή ενός μονοπατιού χρησιμοποιώντας παρατηρήσεις που γίνονται “online” τη στιγμή εκτέλεσης του αλγορίθμου. Ο αλγόριθμος έχει τη δυνατότητα να κάνει τοπικές κινήσεις, να γυρίζει πίσω, να ελέγχει, να επανεξετάζει και στη συνέχεια να δοκιμάζει ξανά προς μια επιλεγμένη κατεύθυνση. Αυτό συμβαίνει καθώς η συνέπεια/το αποτέλεσμα μιας πράξης γίνεται γνωστό μόνο μετά την εκτέλεση της. Τα αποτελέσματα των πράξεων καταγράφονται με στόχο να υπάρξει βελτίωση με το πέρασμα του χρόνου (π.χ. να μην επιλέγονται διαδρομές που αποδείχθηκαν μη cost efficient). Επιλέχθηκε να υλοποιηθεί η κλάση `LearningRealTimeAstar.java` η οποία διατηρεί πληροφορία για τις heuristic values στο `Map<Node, Double> heuristics` των κόμβων τις οποίες βρίσκει η online search “μαθαίνοντας”. Ο αλγόριθμος διατηρεί πληροφορία για το μονοπάτι που ακολουθεί (`path: LinkedList<Road>`) καθώς και για το κόστος των δρόμων που έχει επιλέξει (`weights: ArrayList<Double>`). Για κάθε δρόμο επιλέγει κατάλληλα τον άλλο κόμβο του δηλαδή εκείνον με τον οποίο συνδεέται ο τωρινός κόμβος. Σύμφωνα με τις heuristic values των γειτονικών κόμβων εξετάζει ψάχνει τον κοντινότερο στον

τερματισμό. Αφού επιλέξει τον επόμενο δρόμο που θα ακολουθήσει, ενημερώνεται το heuristic value στον πίνακα των heuristics του LRTA*.

4. SIMULATION & ΑΠΟΤΕΛΕΣΜΑΤΑ

Για κάθε ημέρα εκτελούνται 3 αλγόριθμοι :

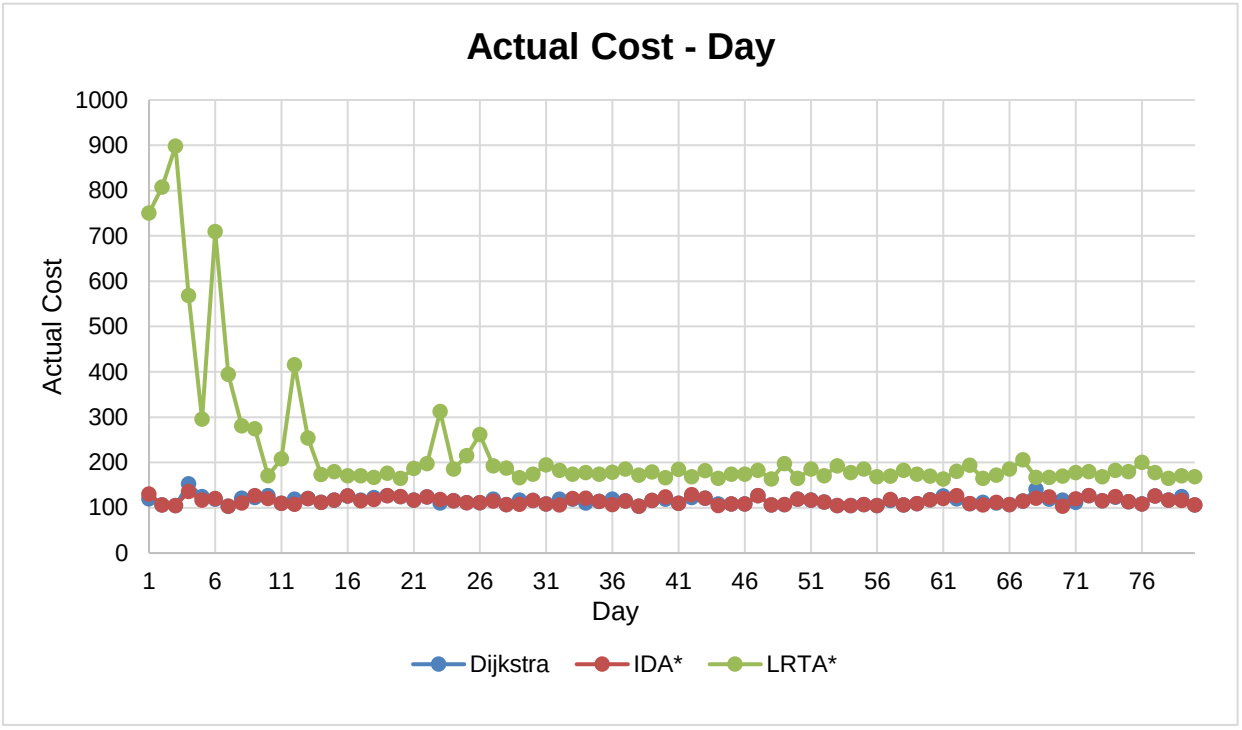
- Dijkstra
- IDA* (Iterative Deepening A*)
- LRTA* (Learning Real Time A*)

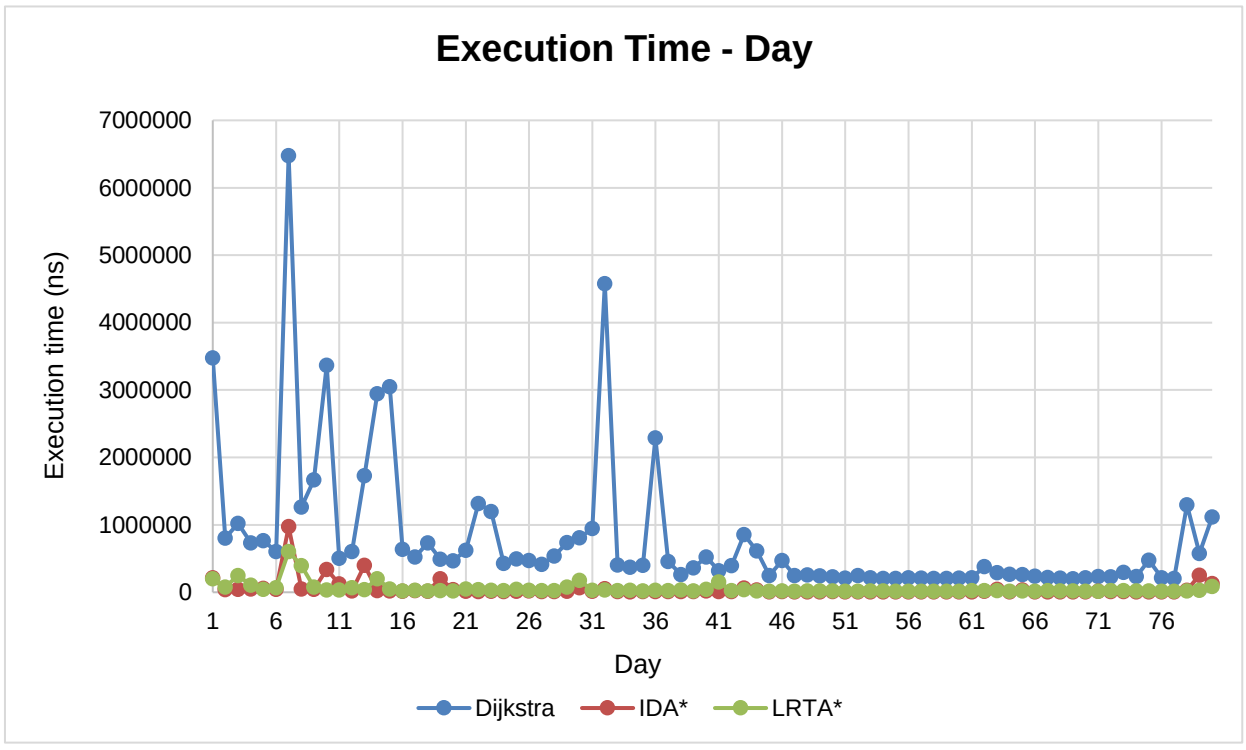
Για καθέναν από τους 3 αυτούς αλγορίθμους καταγράφηκαν για κάθε μία από τις 80 μέρες μέσω της κλάσης `Result.java` τα ακόλουθα:

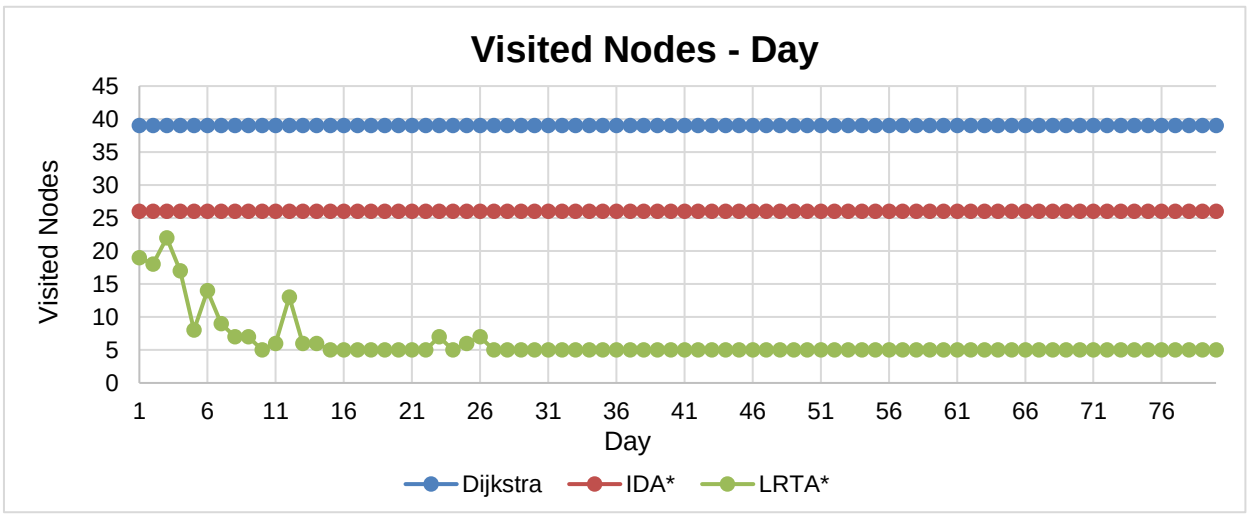
- Αριθμός κόμβων που επισκέφθηκαν εώς την παραγωγή αποτελέσματος
- Χρόνος εκτέλεσης αλγορίθμου
- Προτεινόμενο μονοπάτι δρόμων (ή μονοπάτι που ακολουθήθηκε: LRTA*)
- Κόστος προτεινόμενης διαδρομής σύμφωνα με προβλέψεις ημέρας
- Πραγματικό κόστος διαδρομής μετά το πέρας της ημέρας
- Το όνομα του αλγορίθμου

Κάθε μέρα κρατάει την πληροφορία των Results σε ένα `List<Result> results.` Τα δεδομένα πλέον είναι προσβάσιμα για την κάθε μέρα μέσω της λίστας των ημερών `ProblemInstance.days .`

Παρακάτω παρατίθενται διαγράμματα που απεικονίζουν την εξέλιξη των παραμέτρων που κατάγραφονται για κάθε αλγόριθμο για κάθε μέρα, σε διάστημα 80 ημερών. Τα αποτελέσματα έχουν προκύψει από simulation στο αρχείο εισόδου `sampleGraph1.txt.`


Actual Cost - Day
Actual Cost
1000
900
800
700
600
500
400
300
200
100
0
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
Day
Dijkstra
IDA*
LRTA*


Execution Time - Day
Execution time (ns)
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
Day
Dijkstra
IDA*
LRTA*


Visited Nodes - Day
Visited Nodes
45
40
35
30
25
20
15
10
5
0
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
Day
Dijkstra
IDA*
LRTA*

Στον παρακάτω πίνακα φαίνεται το μέσο κόστος διαδρομής στις 80 μέρες που υπολογίστηκε από κάθεναν από τους 3 αλγορίθμους με το αντίστοιχο αρχείο εισόδου:

Αρχείο	Dijkstra	IDA*	LRTA*
sampleGraph1.txt	115,8	114,7	226,9
sampleGraph2.txt	194,6	191,1	248,4
sampleGraph3.txt	101,3	102,4	135,2

5. ΣΧΟΛΙΑΣΜΟΣ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

Σύμφωνα με το διάγραμμα κόστους διαδρομής ανά ημέρα παρατηρείται μια σχετική σταθερότητα με μικρές αποκλίσεις για τους αλγορίθμους Dijkstra και IDA*. Οι αποκλίσεις αυτές οφείλονται στο γεγονός ότι οι heuristic τιμές που χρησιμοποιεί ο IDA* διαφέρουν από τις προβλέψεις που λαμβάνονται υπόψη από τον Dijkstra, καθώς και σε τυχαίες αλλαγές στις προβλέψεις της κίνησης στους δρόμους. Οι δύο αλγόριθμοι στο 80% των περιπτώσεων τυχαίνει να παράγουν ίδιο path δρόμων επειδή η πιθανότητα του να αλλάξει η κίνηση σε ένα δρόμο είναι σχετικά μικρή και το περιθώριο αλλαγής αυτής επίσης μικρό (+25% -10%). Ακόμη όπως έχει αναφερθεί παραπάνω οι heuristic τιμές που χρησιμοποιεί ο IDA* παράγονται από τον αλγόριθμο του Dijkstra με αρχικό κόμβο τον τερματισμό. (`dijkstra(destination)`).

Όσον αφορά τον LRTA* φαίνεται ότι απέχει αρκετά απο το βέλτιστο μονοπάτι. Αυτό ίσως είναι λογικό δεδομένου του ότι χρειάζεται περίπου 15 μέρες για να ενημερώσει τις heuristic τιμές και να μην πραγματοποιεί «πισωγυρίσματα», όπως φαίνεται και από το διάγραμμα επισκεψιμότητας κόμβων. Αν οι heuristic τιμές που τελικά παράγει είναι ιδανικές δεν μπορούμε να το ξέρουμε εφόσον το διάστημα ημερών είναι σχετικά μικρό. Ακόμη άλλο ένα αρνητικό της υλοποίησης αυτού του αλγορίθμου είναι το γεγονός ότι φτάνει σχετικά γρήγορα σε τοπικό ελάχιστο από το οποίο μετά δεν μπορεί να βρει καλύτερες διαδρομές και να ενημερώσει τις heuristic τιμές του.

Αναφορικά με τον χρόνο εκτέλεσης δεν είναι εφικτό να προκύψει ενά ακριβές και αντικειμενικό συμπέρασμα για το ποιος αλγόριθμος είναι πιο γρήγορος. Προκειμένου να γίνει μια ακριβής μέτρηση μιας τέτοιας παραμέτρου χρειάζεται simulation πολλών περισσότερων κόμβων και δρόμων. Σε χρόνο εκτέλεσης της κλίμακας των ns και 1-10ms οι εξωτερικοί παράγοντες (άλλες διεργασίες παρασκηνίου) δεν μπορούν να θεωρηθούν αμελητέοι. Κατ΄εξαίρεση ο Dijkstra φαίνεται να έχει χρόνο εκτέλεσης 10πλάσιο έως και 20πλάσιο από αυτό των IDA* και LRTA* και αυτό οφείλεται στο γεγονός ότι ο Dijkstra υπολογίζει προς *όλους* τους κόμβους από έναν αρχικό κόμβο σε αντίθεση με τους άλλους αλγορίθμους που «κοιτούν» σε μικρότερο φάσμα κόμβων.