Πανεπιστήμιο Πειραιώς
Τμήμα Ψηφιακών Συστημάτων

Συστήματα Ευφυών Πρακτόρων

2η Εργασία

Επιβλέπων Καθηγητής:
Γεώργιος Βούρος

06 Φεβρουαρίου, 2022

Κωνσταντίνος Παππάς, Ε18129 – kostaspappas2000@hotmail.com
Παναγιώτα Βίτσα, Ε18025 – peggyvitsa@gmail.com

Εισαγωγή στο Πολυπρακτορικό Περιβάλλον

Η εργασία αυτή αφορά ένα 5x5 περιβάλλον που περιέχει τρεις πράκτορες ταξί, οι οποίοι έχουν σκοπό με τη βοήθεια της A-Star να εξυπηρετήσουν όσο το δυνατόν πιο γρήγορα τους πελάτες του περιβάλλοντος. Αποτελείται από δύο βασικά αρχεία Grid.java και MyAgent.java, καθώς και ένα υποστηρικτικό Node.java για τις απαραίτητες υλοποιήσεις των ζητουμένων.

Τα πρώτα δύο αφορούν τις λειτουργίες του περιβάλλοντος και του πράκτορα αντίστοιχα. Χρησιμοποιούνται οι βιβλιοθήκες της jade για την άμεση επικοινωνία και ανταλλαγή μηνυμάτων μεταξύ των δύο προαναφερθέντων, συνεπώς αυτός είναι και ο λόγος που πραγματοποιείται το extend Agent.

Εδώ, εφ' όσον έχουμε πολλούς πράκτορες θα υπάρχουν και συγκρούσεις. Πιο συγκεκριμένα, δύο. Η μία προκύπτει όταν πολλοί πράκτορες θέλουν να μπουν στο ίδιο κελί ενώ απαγορεύεται, ενώ η άλλη προκύπτει όταν ένας ή δύο πράκτορες οι οποίοι δεν έχουν πελάτη για εξυπηρέτηση επιλέγουν να πάνε στον ίδιο κόμβο, στον οποίο υπάρχει μόνο ένας πελάτης.

Grid.java

Το αρχείο αυτό περιέχει πολλές μεθόδους ξεκινώντας από την βασικότερη, την setup().

setup()

Η συγκεκριμένη μέθοδος συμβάλλει στην επικοινωνία του περιβάλλοντος με τον πράκτορα, συμπεριλαμβάνει τον εντοπισμό του μέσω του DF registry, την προσθήκη των services του agent και την ανταλλαγή μηνυμάτων για ανταλλαγή πληροφοριών. Καλεί την ***findStartingLocations()*** για να φτιάξει τον 5x5 πίνακα locations του περιβάλλοντος με τις θέσεις των πελατών και το πλήθος τους ανά τοποθεσία. Επιπλέον δείχνει τη θέση των agent (συμβολισμοί: A, B, C) και οποιαδήποτε άλλη θέση με X.

Έπειτα, ξεκινά κυκλικό Behaviour στο οποίο συμπεριλαμβάνεται μία μέθοδος ***action()***.

action()

Για κάθε agent ετοιμάζει ένα μήνυμα. Το περιεχόμενο του θα σταλεί στον receiver agent, ζητώντας του να επιλέξει μία κατεύθυνση.

Στη συνέχεια, θα λάβει την απάντηση μέσω της blockingReceive() μεθόδου που είναι τύπου ACLMessage και συμπεριλαμβάνεται στη βιβλιοθήκη της. Το μήνυμα που περιέχεται αποτελείται από δύο στοιχεία χωρισμένα με κόμμα. Το πρώτο αφορά το λεκτικό της επόμενης κίνησης (up/down/left/right/Client target and current node are the same staying here/There are no clients left) και το δεύτερο την απόσταση του κάθε πράκτορα από το στόχο του. Αυτά αποθηκεύονται σε δύο πίνακες τριών θέσεων messageQueue[] και distanceToDestination[] αντίστοιχα. Αφού ελέγξει τα conflict, εκτυπώνεται το αντίστοιχο ενημερωτικό μήνυμα. Έχοντας αυτό το λεκτικό καλεί την ***makeMove()*** για να υλοποιηθεί η σωστή κίνηση του

πράκτορα. Όταν τελειώσουν οι απαραίτητες επαναλήψεις, τερματίζει το πρόγραμμα κάνοντας terminate όλους τους agents.

ΕΠΙΛΥΣΗ ΠΡΩΤΟΥ ΤΥΠΟΥ CONFLICT

checkConflicts(String[] messageQueue)

Μέσω αυτής της μεθόδου ελέγχεται αν υπάρχει ο πρώτος τύπος conflict ο οποίος είναι όταν δύο ή και οι τρεις πράκτορες θέλουν να μεταβούν στο ίδιο κελί. Εφ' όσον αυτό δεν είναι εφικτό, καλείται αυτή η μέθοδος και το περιβάλλον επιλύει αυτή τη σύγκρουση μέσω της διαπραγμάτευσης που εκτελεί.

Αρχικά, εντοπίζονται οι συντεταγμένες κάθε πράκτορα και αποθηκεύονται σε έναν δισδιάστατο πίνακα. Έπειτα, με βάση αυτές και την επιθυμία του κάθε πράκτορα (πίνακας messageQueue που περιέχει το επόμενο βήμα του πράκτορα up, down, left, right) για την επόμενη μετακίνηση του υπολογίζεται το επόμενο κελί του καθενός. Αν αυτό το κελί συμπίπτει για κάποιους και δεν είναι ένα από τα R, G, B, Y – μέθοδος ***isOnRGBY()*** (καθώς εκεί επιτρέπεται η είσοδος σε πολλούς πράκτορες) τότε υπάρχει σύγκρουση και πρέπει να επιλυθεί.

Γι' αυτό το λόγο, δημιουργήθηκε ένας πίνακας Boolean rightOfWay τριών θέσεων, με την κάθε θέση να αναφέρεται στο δικαίωμα του κάθε πράκτορα να κινηθεί ή να μείνει στάσιμος. Αρχικοποιείται με true, θεωρώντας ότι εξαρχής δεν υπάρχουν συγκρούσεις και όλοι επιτρέπεται να μετακινούνται.

Αν η σύγκρουση υπάρχει και για τους τρεις, πραγματοποιείται αναζήτηση του πράκτορα με την ελάχιστη απόσταση σε μία μέθοδο ***minDestination()*** και επιστρέφεται ο δείκτης του πράκτορα, συνεπώς προχωράει μόνο εκείνος που είναι πιο κοντά στο στόχο του. Για τους υπόλοιπους δύο, το rightOfWay ισούται με false, ώστε να μη προχωρήσουν.

Στη περίπτωση που υπάρχει σύγκρουση μεταξύ δύο μόνο πρακτόρων ελέγχεται η απόσταση τους. Ο πράκτορας με τη μεγαλύτερη προς το στόχο απόσταση σταματάει (τιμή false) και προηγείται ο άλλος. Σε αυτή τη περίπτωση πραγματοποιείται ένας δεύτερος έλεγχος με τη μέθοδο ***reCheckConflicts(int [][] coordinates)***.

reCheckConflicts(int[][] coordinates)

Μέσω αυτής ελέγχεται η περίπτωση όπου η συντεταγμένη του ενός ταυτίζεται πάλι με του άλλου, άρα επιστέφει την αληθοτιμή true, διαφορετικά false. Αν η τιμή είναι true, στη μέθοδο ***checkConflicts(String[] messageQueue)*** αλλάζει το rightOfWay, ώστε τελικά να προηγηθεί ο δεύτερος έναντι του πρώτου.

isOnRGBY(int i, int j)

Οι παράμετροι που δέχεται αποτελούν τη συντεταγμένη του κελιού που μας ενδιαφέρει. Αν αυτή ισούται με κάποια από τις συντεταγμένες των R, G, B, Y επιστρέφεται η αληθοτιμή true, διαφορετικά η false.

minDestination()

Αυτή η μέθοδος βρίσκει τον πράκτορα με τη μικρότερη απόσταση προς το στόχο του με βάση τον πίνακα τριών θέσεων distanceToDestination που περιέχει την απόσταση αυτή του κάθε πράκτορα αντίστοιχα. Όποιος έχει τη μικρότερη απόσταση, επιστρέφεται ο δείκτης του.

ΕΠΙΛΥΣΗ ΔΕΥΤΕΡΟΥ ΤΥΠΟΥ CONFLICT

Σε περίπτωση που 2 ή 3 agent πάνε να εξυπηρετήσουν σε node που έχει μόνο ένα πελάτη δημιουργείται conflict.

Για να μεριμνά το σύστημα για αυτή τη περίπτωση υλοποιήσαμε το εξής:

Συγκρατεί σε πίνακα *isInterested[]* εάν κάποιος πράκτορας ενδιαφέρεται να πάει σε κόμβο που έχει μόνο ένα πελάτη. Σε περίπτωση που πάνω από ένας ενδιαφέρεται (άρα έχουμε conflict) γίνεται δημοπρασία, υπολογίζεται ο πράκτορας με την λιγότερη απόσταση προς το ζητούμενο κόμβο και αυτός νικάει τη δημοπρασία. Οι χαμένοι της δημοπρασίας χάνουν το δικαίωμα να πάνε στο κόμβο (αυτό υλοποιείται κάνοντας το κόστος για να πάει εκεί πολύ μεγάλο) με ένα πελάτη και πρέπει να βρούνε νέο προορισμό.

Στην παραπάνω διαδικασία βοηθάνε οι συναρτήσεις:

checkSecondTypeOfConflict()

Ελέγχει πότε εμφανίζονται τα conflict κοιτώντας τον πίνακα *isInterested*.

secondConflictSolution()

Στέλνει μήνυμα στον πράκτορα πως πρέπει να επιλέξει νέο προορισμό καλώντας την pickdirection().

findStartingLocations()

Εδώ αρχικοποιείται ο δυσδιάστατος – τύπου String – πίνακας locations. Σε όλες τις θέσεις εκχωρείται το X, ενώ στις θέσεις με τις αντίστοιχες συντεταγμένες από τις οποίες εμφανίζονται οι πελάτες υπάρχουν νούμερα. Αυτά τα νούμερα συμβολίζουν το πλήθος των πελατών που υπάρχουν σε κάθε μία από αυτές τις τέσσερις τοποθεσίες, ξεκινώντας όλες τις θέσεις από το

0. Μέσω μίας random number generator κάνουμε spawn τους πράκτορες σε τυχαία θέση, πάντα βέβαιοι ότι δε θα εμφανιστούν στο ίδιο κελί.

Στη συνέχεια, δημιουργείται ένας 5x5 πίνακας myNodes τύπου Node (βλ. αρχείο Node.java) και για κάθε node δημιουργούνται οι γείτονες που έχει γύρω του, προσθέτοντας έτσι το κόστος κάθε κίνησης. Για παράδειγμα, εκχωρείται στο weight η τιμή 100 αν ανάμεσα με τον διπλανό γείτονα υπάρχει τοίχος που οδηγεί στη πρόσκρουση. Αυτές οι προσθήκες γειτόνων υλοποιούνται με τις συναρτήσεις ***addNeighbor()*** και ***removeNeighbor()*** που υλοποιούνται στο αρχείο Node.java και το οποίο θα αναλυθεί στη συνέχεια.

Τέλος, κάνουμε spawn μερικούς πελάτες για την εκκίνηση του προγράμματος με τη βοήθεια της μεθόδου ***spawnClient()***.

spawnClient()

Με τη βοήθεια του random number generator επιλέγει μία τυχαία θέση στην οποία είναι δυνατό να εμφανιστεί πελάτης, ώστε να την αυξήσει κατά ένα και συνεπώς να δείξει με αυτό το τρόπο ότι νέος πελάτης εμφανίστηκε στο περιβάλλον μας. Έπειτα, προσθέτει στον πίνακα myNodes αυτόν τον client, με δύο παραμέτρους: όνομα (id) και τελικός στόχος πελάτη (target). Ο τελικός προορισμός του πελάτη υπολογίζεται μέσω της ***chooseClientTarget()***.

chooseClientTarget()

Σε αυτή τη μέθοδο πάλι με τη βοήθεια του random number generator βρίσκουμε έναν τυχαίο προορισμό που αντιστοιχεί στον πελάτη, δημιουργώντας με αυτό το τρόπο νέο target node που επιστρέφεται και ορίζεται ως ο target του συγκεκριμένου πελάτη.

makeMove(String content)

Η συγκεκριμένη μέθοδος δέχεται ως όρισμα το λεκτικό της κίνησης που έχει λάβει από τον agent, δηλαδή up/down/left/right/Client target and current node are the same staying here/ There are no available nodes left. Εφ' όσον δέχεται τέτοια πληροφορία, πρέπει να αλλάξει και τη θέση του agent – γι' αυτό το λόγο, πρώτα θα την ψάξει με τη βοήθεια της ***search()*** τις συντεταγμένες του πράκτορα. Θα αποθηκευτούν σε έναν πίνακα δύο θέσεων, στην θέση 0 θα είναι το i και στη θέση 1 το j.

Αν ο πράκτορας δε βρίσκεται σε μία από τις τέσσερις θέσεις από τις οποίες μπορεί να υπάρξουν πελάτες, τότε απλά αντί για τον συμβολισμό A θα βάλουμε τον αδιάφορο συμβολισμό X που υποδεικνύει ότι πλέον ο πράκτορας έχει φύγει από εκεί. Στη περίπτωση που ο agent είναι σε μία από τις τέσσερις δυνατές θέσεις στις οποίες υπάρχει το πλήθος των πελατών και θέλει να φύγει, αφαιρείται ο συμβολισμός A και αντικαθίσταται με το κενό, αφού αλλάζει θέση.

Συνεπώς, εκτελώντας μία switch για κάθε περίπτωση του string content αλλάζουμε την αντίστοιχη συντεταγμένη του πράκτορα. Για παράδειγμα, αν η κίνηση που πρέπει να κάνει ο πράκτορας είναι προς τα πάνω, θα αλλάξει το i του πράκτορα κατά *μείον ένα*. Στη περίπτωση που δεν υπάρχουν πράκτορες στο grid, ο πράκτορας μένει στη θέση του.

Αφού αλλάξει η συντεταγμένη του πράκτορα, αλλάζει μόνο στην τοπική αυτή μεταβλητή. Για να αλλάξει επίσημα πάνω στον πίνακα locations, καλείται η ***changeCoordinates()*** η οποία αλλάζει τη συντεταγμένη για τη θέση που προορίζεται να πάει ο πράκτορας.

Search(String AgentName)

Αυτή η μέθοδος λαμβάνει ένα String για παράμετρο το οποίο αναφέρεται στο όνομα του πράκτορα (A, B, C) και εκτελεί μία διπλή for στην locations για να δει που περιέχεται η παράμετρος αυτή. Όταν τη βρει, την επιστρέφει σε πίνακα δύο θέσεων, όπου στη θέση 0 είναι το i και στη θέση 1 το j.

changeCoordinates(int[] coordinates)

Εδώ αλλάζουν οι συντεταγμένες της θέσης της οποίας προορίζεται να πάει ο πράκτορας, όταν δηλαδή θελήσει να κάνει κίνηση προς κάποιο κόμβο. Αν αυτός ο κόμβος, λοιπόν, είναι X, τότε αλλάζει στο όνομα του πράκτορα ώστε να υπάρχει εκεί πλέον. Αν, από την άλλη, ο κόμβος που θέλει να πάει αποτελείται από νούμερο σημαίνει ότι ο κόμβος είναι θέση στην οποία πιθανόν να υπάρχουν πελάτες. Σε αυτή τη περίπτωση, αν η θέση περιέχει τουλάχιστον έναν πελάτη, ο πράκτορας τον κάνει κάποιον pick up, μειώνει το πλήθος αυτού του κόμβου πελατών κατά ένα και επιπλέον προσθέτει και τον συμβολισμό A για να δείξει τη θέση του. Ο τρόπος με τον οποίο αποφασίζει ποιον πελάτη θα επιλέξει – στη περίπτωση που είναι δύο ή παραπάνω – αποφασίζεται μέσω της μεθόδου ***chooseClientWithClosestDestination()***.

chooseClientWithClosestDestination(int[] coordinates)

Αυτή η μέθοδος καλείται στη περίπτωση που ο πράκτορας έχει συναντήσει δύο ή παραπάνω πελάτες στην ίδια θέση και πρέπει αποφασίσει ποιον θα επιλέξει. Για το συγκεκριμένο node του δισδιάστατου πίνακα myNodes, υπάρχει μία λίστα με πελάτες. Για κάθε πελάτη εκτελείται η επανάληψη και υπολογίζεται με τη βοήθεια της A-Star το ελάχιστο μονοπάτι που θέλει ο καθένας για να φτάσει στον προορισμό του. Κατά συνέπεια, εντοπίζεται και επιστρέφεται ως αποτέλεσμα ο πελάτης με τη μικρότερη διαδρομή, άρα αυτόν επιλέγει ο πράκτορας.

Extra Methods Included:

- **getLocations()**, βασική getter μέθοδος για τον πίνακα locations
- **setLocations(String[][] locations)**, βασική setter μέθοδος για τον πίνακα locations
- **getMyNodes()**, βασική getter μέθοδος για τον πίνακα myNodes
- **setMyNodes(Node[][] myNodes)**, βασική setter μέθοδος για τον πίνακα myNodes

MyAgent.java

Το αρχείο αυτό αφορά τις λειτουργίες που εκτελούν οι πράκτορες, όπως την επιλογή της κατεύθυνσης.

setup()

Η συγκεκριμένη μέθοδος συμβάλλει στην επικοινωνία του πράκτορα με το περιβάλλον. Για αρχή κάνει τον agent register στο DH, καταχωρώντας του το όνομα καθώς και τον τύπο του πράκτορα που είναι απλή κατηγορία agent. Προσθέτει behaviour μέσα στο οποίο δημιουργεί μία ***action()*** μέθοδο, στην οποία περιγράφει όλα τα actions του πράκτορα που αφορούν την αποστολή και λήψη μηνυμάτων προς και από το περιβάλλον.

action()

Αν το μήνυμα που έλαβε από το περιβάλλον Grid είναι ίσο με το «Pick a Direction» τότε καλεί την ***pickDirection()*** και την αποθηκεύει σε μία μεταβλητή String. Φτιάχνει ένα μήνυμα τύπου ACLMessage και ορίζει τον παραλήπτη. Έπειτα, στέλνει το μήνυμα. Αν το μήνυμα που έλαβε από το περιβάλλον είναι «End» τότε τερματίζεται ο πράκτορας, αφαιρώντας τον και από το registry.

pickDirection()

Η μέθοδος αυτή για αρχή ψάχνει τον agent μέσω της προαναφερθέντας μεθόδου ***search()*** και αποθηκεύει τις συντεταγμένες του πράκτορα σε έναν πίνακα agentCoordinates δύο θέσεων.

Κατηγοριοποιεί την επιλογή κατεύθυνσης σε δύο περιπτώσεις:

α. ο πράκτορας έχει κάνει pick up κάποιον πελάτη και πρέπει να επιλέξει διαδρομή για να τον πάει στον προορισμό του

β. ο πράκτορας είναι ελεύθερος και πρέπει να βρει τον κοντινότερο κόμβο στο grid που έχει διαθέσιμους πελάτες

Στην πρώτη περίπτωση, ελέγχει αν το node του agent ισούται με το target του πελάτη. Αν όντως είναι ίσα αυτό σημαίνει ότι δε πρέπει να γίνει κάποια έξτρα κίνηση, αφού βρίσκονται ήδη στον προορισμό του πελάτη. Από την άλλη, καλείται ο A-Star ο οποίος επιστρέφει το node του agent, και το parent αυτού του node είναι το επόμενο βήμα. Αν αυτό το βήμα δεν υπάρχει, θέτει τον client που είχε ο agent κενό αφού υποδηλώνεται ότι ο agent έφτασε στον προορισμό του πελάτη και τον κάνει ofload. Αλλιώς, αν έχει ακόμα βήματα να κάνει εκτυπώνει ενημερωτικό μήνυμα ότι εκτελεί ακόμα τη διαδρομή.

Στην δεύτερη περίπτωση, ορίζει αποστάσεις για κάθε έναν από τους τέσσερις διαθέσιμους κόμβους πελατών. Επιπλέον ελέγχει αν οι κόμβοι έχουν πελάτες, ώστε στη περίπτωση που κάποιος κόμβος είναι ο κοντινότερος αλλά δεν έχει πελάτες να μην επισκεφθεί ο πράκτορας. Βρίσκει τα length καθενός από τους τέσσερις κόμβους με τη βοήθεια της ***printPath()*** και κάνοντας τις απαραίτητες συγκρίσεις βρίσκει τον κοντινότερο. Στη περίπτωση που υπάρχει κάποια θέση με διαθέσιμο πελάτη, καλεί την A-Star για να βρει το συντομότερο path προς αυτόν και να επιστραφεί η επόμενη θέση node. Τέλος, καλείται η ***decidePath()*** για να επιστραφεί το ακριβές λεκτικό up, down, left ή right ως αποτέλεσμα της μεθόδου pickDirection().

decidePath()

Λαμβάνοντας ως παραμέτρους τον πίνακα συντεταγμένων του πράκτορα και την επόμενη κίνηση που πρέπει να κάνει αυτός, δηλαδή την επόμενη θέση, υπολογίζεται η διαφορά. Για παράδειγμα, αν αυτή η διαφορά για τον άξονα i (των γραμμών) είναι 1, τότε σημαίνει ότι ο πράκτορας πρέπει να κινηθεί προς τα πάνω, ενώ αν είναι -1 τότε πρέπει να κινηθεί προς τα κάτω. Αντίστοιχοι υπολογισμοί γίνονται και για τον άξονα j (των στηλών). Σε διαφορετική κατάσταση ο πράκτορας έχει φτάσει στη τελική του θέση.

Extra Methods Included:

- **isPickedUp()**, βασική getter μέθοδος για τη μεταβλητή client

είτε θα έχει κάνει ο πράκτορας pick up τον client και, συνεπώς, αυτή η συνάρτηση θα επιστρέφει τα στοιχεία του – είτε δεν θα έχει πελάτη pick up και θα επιστρέφει null

- **setPickedUp(Client client)**, βασική setter μέθοδος για τη μεταβλητή client

γίνεται set στην μέθοδο ***changeCoordinates()*** της κλάσης Grid

- **getDirection()**, βασική getter μέθοδος για τη μεταβλητή direction

- **setDirection(String direction)**, βασική setter μέθοδος για τη μεταβλητή direction

Node.java

Η κλάση Node αποτελεί το βασικό στοιχείο για την δημιουργία του Grid. Κάθε τετραγωνάκι είναι ένα node και όλα μαζί βρίσκονται σε ένα πίνακα 5x5. Κάθε node έχει τη λίστα με πελάτες (εάν υπάρχουν πελάτες στο συγκεκριμένο node) και τη λίστα με τα γειτονικά node.

Οι γείτονες αναπαρίστανται με δικιά τους κλάση που έχει στο constructor της ένα node (το γειτονικό) και το κόστος για να πάμε από το τωρινό node σε αυτό το γείτονα. Ως αυτό το κόστος έχουν χρησιμοποιηθεί οι κανόνες από την εκφώνηση της άσκησης. Δηλαδή, από το κόμβο (0,0) στο κόμβο (0,1) έχουμε κόστος 1. Ενώ από το (0,1) στο (0,2) αφού υπάρχει τοίχος έχουμε κόστος 100.

Οι πελάτες έχουν και αυτοί δικιά τους κλάση και στο δικό τους constructor έχουν το όνομά τους και το node που θέλουν να πάνε. Η κλάση node έχει τα κατάλληλα constructor για τις κλάσεις που αναφέρθηκαν παραπάνω.

CalculateHeuristic()

Σε αυτή τη κλάση βρίσκεται και η ευριστική συνάρτηση. Ως ευριστική χρησιμοποιούμε την απόσταση Chebyshev. Η απόσταση Chebyshev μεταξύ δύο κόμβων είναι το μέγιστο από την απόλυτη τιμή της διαφοράς των x,y. Για παράδειγμα, στο grid μας η απόσταση Chebyshev μεταξύ του κόμβου στη θέση (2,3) και του κόμβου στη θέση (0,0) είναι 3.

$$D_{\text{Chebyshev}}(x, y) := \max_i(|x_i - y_i|)$$

aStar()

Σε αυτή τη κλάση υπάρχει και η μέθοδος για την A*. Δέχεται ένα starting node και ένα target node και επιστρέφει το καλύτερο μονοπάτι που πρέπει να ακολουθήσει το ταξί μας.

Για να το κάνει αυτό ξεκινάει δημιουργώντας 2 priority queue. Τα priority queue είναι λίστες που γίνονται αυτόματα sort σε κάθε εκχώρηση νέου στοιχείου στη λίστα σύμφωνα με το natural ordering των στοιχείων. Σε αυτή τη περίπτωση για να αποφασιστεί το natural ordering γίνεται χρήση της μεθόδου ***compareTo()*** που συγκρίνει τα f δύο κόμβων, τα f είναι το ολικό κόστος που χρειάζεται για να πάμε σε ένα κόμβο (περισσότερα για αυτό παρακάτω).

Χρειαζόμαστε δύο priority queue: ένα για τους κόμβους που έχει τελειώσει η ανάλυση τους και ένα για τους οποίους δεν έχει τελειώσει. Θα τα ονομάσουμε closedlist και openlist αντίστοιχα. Οι κόμβοι στο closedlist έχουν υπολογισμένο το μικρότερο μονοπάτι και έχουν όλους τους γειτονικούς κόμβους υποψήφιους για ανάλυση αφού τους βάζουμε στο openlist. Κλειστά nodes μπορούν να γίνουν πάλι ανοιχτά αν τα συναντήσουμε μέσω νέου μονοπατιού.

Ξεκινάμε θέτοντας τον αρχικό κόμβο ως ανοιχτό. Εισάγουμε στην openlist τους γείτονες του αρχικού κόμβου και υπολογίζουμε για το κάθε γείτονα τα:

- g του που είναι το σταθερό κόστος για να πάμε σε αυτό το γείτονα
- το h του που είναι η ευριστική μας
- και στην τελική το άθροισμα των 2 που είναι το f

Ο γείτονας με το μικρότερο f εκχωρείται ως το επόμενο βήμα στο μονοπάτι μας.

Κάθε φορά αναλύουμε το node που είναι στη πρώτη θέση της λίστας openlist δηλαδή έχει και το μικρότερο f (λόγω του sorting που κάνει αυτόματα το priority queue)

Μόλις το πρώτο στοιχείο του openlist είναι το target node μας η μέθοδος γυρνάει ένα linked list με πρώτο στοιχείο τον αρχικό κόμβο και τελευταίο στοιχείο το κόμβο target.

Η σύνδεση του linked list γίνεται μέσω της μεταβλητής parent που είναι τύπου node.

PrintPath()

Χρησιμοποιείται για να γίνει η εκτύπωση του linked list που αναφέρθηκε παραπάνω, δηλαδή, το αποτέλεσμα του A*.

Extra Methods Included:

- **addNeighbor()**, βασική μέθοδος προσθήκης γειτόνων ανά κόμβο
- **removeNeighbor()**, μέθοδος αφαίρεσης γειτόνων ανά κόμβο
- **addClient()**, μέθοδος προσθήκης πελατών ανά κόμβο
- **removeClient()**, μέθοδος αφαίρεσης πελατών ανά κόμβο

Υπάρχει μία μοναδική ιδιαιτερότητα στις μεθόδους για την διαγραφή πελάτη και γείτονα. Αφού επιδιώκουμε να αφαιρέσουμε από arraylist, στη συνθήκη του remove πρέπει να χρησιμοποιηθεί η *lambda* συνάρτηση που παίρνει το ρόλο του φίλτρου ώστε να αφαιρεθεί ο κατάλληλος πελάτης/γείτονας.