ΙΩΑΝΝΗΣ ΚΑΠΕΤΑΝΓΕΩΡΓΗΣ **Α.Μ. 1115201800061**

PROJECT 0
Τεχνητή Νοημοσύνη
Θεωρητικές Ασκήσεις (Προβλήματα 2-5)

Πρόβλημα 2

Δεδομένα:
Έχουμε ένα πρόβλημα Π το οποίο θα λύσουμε με τον αλγόριθμο επαναληπτικής εκβάθυνσης.
d : βάθος του δένδρου
b : παράγοντας διακλάδωσης του δένδρου
g : το βάθος του goal state το οποίο έχει το μικρότερο βάθος από όλα τα υπόλοιπα goal states

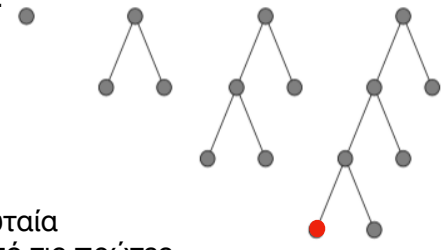
Αρχικά θα αναλύσω τον μεγαλύτερο αριθμό κόμβων που μπορούν να δημιουργηθούν από τον αλγόριθμο.
Η χειρότερη περίπτωση είναι αν το goal state βρίσκεται στον τελευταίο κόμβο του επιπέδου g δηλαδή σχηματικά να είναι ο δεξιότερος κόμβος στο επίπεδο g. Αφού το goal state βρίσκεται στο depth g ο αλγόριθμος θα εκτελέσει g+1 επαναλήψεις (ξεκινώντας από depth=0), και για κάθε μία επανάληψη πέραν της τελευταίας θα "εξερευνήσει" όλους τους κόμβους μέχρι το αντίστοιχο επίπεδο. Επίσης αφού η παράγοντας διακλάδωσης είναι b κάθε επίπεδο θα έχει $depth^b$ κόμβους. Άρα το επίπεδο 0 έχει 1 κόμβο και θα εξερευνηθεί g+1 φορές (δηλαδή σε κάθε επανάληψη), το επίπεδο 1 έχει d κόμβους και θα εξερευνηθεί g φορές (δηλαδή σε όλες τις επαναλήψεις εκτός από την πρώτη). Συνεχίζουμε την ίδια λογική για κάθε επίπεδο και καταλήγουμε στο τελευταίο επίπεδο που θα εξερευνηθεί, δηλαδή στο επίπεδο g, το οποίο έχει g^b κόμβους και ο καθένας θα εξερευνηθεί μόνο 1 φορά (δηλαδή μόνο στην τελευταία επανάληψη). Έτσι ο μέγιστος αριθμός των κόμβων που θα εξερευνηθούν προκύπτει από τον τύπο:

$$(g+1) + gb + (g-1)b^2 + \ldots + 2b^{g-1} + b^g = \sum_{d=0}^{g} (g-(d-1))b^d$$

Όσον αφορά τον μικρότερο αριθμό κόμβων που μπορούν να δημιουργηθούν από τον αλγόριθμο:
Μέχρι την προτελευταία επανάληψη (δηλαδή όσο ο αλγόριθμος δεν έχει βρει λύση) θα εξερευνά όλους τους κόμβους. Υποθέτοντας ότι το αρχικό δένδρο είναι πλήρες (δηλαδή είναι γεμάτο (full) μέχρι και το επίπεδο g-1), ο αλγόριθμος μέχρι και την επανάληψη για βάθος g-1 θα κάνει τον μέγιστο αριθμό εξερευνήσεων.
Σύμφωνα με τον τύπο που αποδείξαμε παραπάνω, ο μέγιστος αριθμός εξερευνήσεων για βάθος g-1 είναι:

$$g + (g-1)b + (g-2)b^2 + \ldots + 2b^{g-2} + b^{g-1} = \sum_{d=0}^{g-1} ((g-1)-(d-1))b^d = \sum_{d=0}^{g-1} (g-d)b^d$$

Στην τελευταία επανάληψη (δηλαδή για βάθος g) η καλύτερη περίπτωση είναι το goal state να είναι ο πρώτος κόμβος που θα εξερευνηθεί από τον αλγόριθμο στο επίπεδο g. Στην περίπτωση αυτή ο αλγόριθμος έχει κάνει expand μόνο έναν κόμβο σε κάθε επίπεδο όποτε σε κάθε επίπεδο (πλην του επιπέδου 0) έχουν δημιουργηθεί b κόμβοι. Αφού αυτό έχει γίνει σε g επίπεδα (από το επίπεδο 1 μέχρι και το επίπεδο g) έχουν δημιουργηθεί gb κόμβοι. Προσθέτουμε και την ρίζα άρα έχουμε συνολικά gb+1 κόμβους.
Ένα σχηματικό παράδειγμα που δείχνει αυτήν την περίπτωση παρουσιάζεται στην διπλανή εικόνα.
Στο παράδειγμα αυτό τρέχουμε τον αλγόριθμο για βάθος 3 και υποθέτουμε ότι g=3 (δηλαδή εκτελείται η τελευταία επανάληψη) και ο goal node είναι ο κόκκινος κόμβος.



Προσθέτοντας τον αριθμό τον κόμβων που δημιουργήθηκαν στην τελευταία επανάληψη (δηλαδή gb+1) στον αριθμό κόμβων που δημιουργούνται από τις πρώτες g-1 επαναλήψεις (το οποίο υπολογίσαμε παραπάνω) καταλήγουμε στον συνολικό ελάχιστο αριθμό κόμβων που μπορούν να δημιουργηθούν από τον αλγόριθμο. Ο αριθμός αυτός είναι:

$$g + (g-1)b + (g-2)b^2 + \ldots + 2b^{g-2} + b^{g-1} + gb + 1 = \sum_{d=0}^{g-1} (g-d)b^d \quad + (gb+1)$$

Πρόβλημα 3

Για τα ερωτήματα του προβλήματος έχω υποθέσει ότι ο γράφος είναι κατευθυνόμενος, δηλαδή ισχύουν τα βελάκια που φαίνονται στο σχήμα της εκφώνησης.

Ερώτημα α

Αρχικά θα ελέγξω αν η ευρετική συνάρτηση είναι συνεπής. Για να είναι συνεπής μια ευρετική συνάρτηση πρέπει για κάθε action με κόστος c η τιμή της ευρετικής συνάρτησης να μειώνεται το πολύ κατά c. Δηλαδή αν ένα action είναι η μετακίνηση από τον κόμβο n στον κόμβο n' πρέπει να ισχύει: $h(n) \leq c(n,n') + h(n')$.
Θα ελέγξω αν ισχύει η ανισότητα αυτή για κάθε κόμβο του γράφου και κάθε πιθανή κίνηση από τον κόμβο αυτό.

Κόμβος που ελέγχεται	Action	Ανισότητα που πρέπει να ισχύει	Τιμές ανισότητας	Συνεπής
o103	o103 -> ts	$h(o103) \leq c(o103,ts) + h(ts)$	$21 \leq 8 + 23$	Ναι
o103	o103 -> b3	$h(o103) \leq c(o103,b3) + h(b3)$	$21 \leq 4 + 17$	Ναι
o103	o103 -> o109	$h(o103) \leq c(o103,o109) + h(o109)$	$21 \leq 12 + 24$	Ναι
ts	ts -> mail	$h(ts) \leq c(ts,mail) + h(mail)$	$23 \leq 6 + 26$	Ναι
b3	b3 -> b1	$h(b3) \leq c(b3,b1) + h(b1)$	$17 \leq 4 + 13$	Ναι
b3	b3 -> b4	$h(b3) \leq c(b3,b4) + h(b4)$	$17 \leq 7 + 18$	Ναι
b1	b1 -> c2	$h(b1) \leq c(b1,c2) + h(c2)$	$13 \leq 3 + 10$	Ναι
b1	b1 -> b2	$h(b1) \leq c(b1,b2) + h(b2)$	$13 \leq 6 + 15$	Ναι
c2	c2 -> c1	$h(c2) \leq c(c2,c1) + h(c1)$	$10 \leq 4 + 6$	Ναι
c2	c2 -> c3	$h(c2) \leq c(c2,c3) + h(c3)$	$10 \leq 6 + 12$	Ναι
c1	c1 -> c3	$h(c1) \leq c(c1,c3) + h(c3)$	$6 \leq 8 + 12$	Ναι
b2	b2 -> b4	$h(b2) \leq c(b2,b4) + h(b4)$	$15 \leq 3 + 18$	Ναι
b4	b4 -> o109	$h(b4) \leq c(b4,o109) + h(o109)$	$18 \leq 7 + 24$	Ναι
o109	o109 -> o111	$h(o109) \leq c(o109,o111) + h(o111)$	$24 \leq 4 + 27$	Ναι
o109	o109 -> o119	$h(o109) \leq c(o109,o119) + h(o119)$	$24 \leq 16 + 11$	Ναι
o119	o119 -> storage	$h(o119) \leq c(o119,storage) + h(storage)$	$11 \leq 7 + 12$	Ναι
o119	o119 -> o123	$h(o119) \leq c(o119,o123) + h(o123)$	$11 \leq 9 + 4$	Ναι
o123	o123 -> o125	$h(o123) \leq c(o123,o125) + h(o125)$	$4 \leq 4 + 6$	Ναι
o123	o123 -> r123	$h(o123) \leq c(o123,r123) + h(r123)$	$4 \leq 4 + 0$	Ναι

Ελέγξαμε αν η ευρετική συνάρτηση είναι συνεπής για κάθε δυνατό action στον γράφο. Για τον κόμβο στόχου ισχύει ότι h(r123)=0. Επίσης υποθέτουμε ότι για κόμβους όπου δεν υπάρχει κάποιο μονοπάτι προς την κατάσταση στόχου (όπως ο mail) το πραγματικό κόστος εύρεσης λύσης είναι άπειρο. Έτσι η ευρετική συνάρτηση δεν υπερεκτιμά το πραγματικό κόστος.
Άρα σύμφωνα με όλα τα παραπάνω η ευρετική συνάρτηση h είναι συνεπής για το συγκεκριμένο πρόβλημα. Επίσης, όπως γνωρίζουμε από την θεωρία, αφού η h είναι συνεπής (και επίσης αφού δεν υπερεκτιμά για κόμβος όπως ο mail) είναι και παραδεκτή.

Ερώτημα β

• Αναζήτηση πρώτα σε πλάτος (BFS)
Τα αποτελέσματα μου είναι βασισμένα στον αλγόριθμο που περιγράφεται στις διαφάνειες “Αναζήτηση σε Γράφους” του μαθήματος. Δηλαδή το αν ένας κόμβος είναι κατάσταση στόχου ελέγχεται πριν προσθέσουμε έναν κόμβο στο σύνορο (και όχι όταν τον κάνουμε pop από το σύνορο). Επομένως ο κόμβος στόχου δεν φαίνεται στα αποτελέσματα καθώς δεν μπαίνει ποτέ στο σύνορο.
Οι σειρά των κόμβων που βγαίνουν από το σύνορο είναι:

o103, b3, o109, ts, b1, b4, o111, o119, mail, b2, c2, o123

• Αναζήτηση πρώτα σε βάθος (DFS)
Τα αποτελέσματα είναι βασισμένα στο αλγόριθμο που περιγράφεται στις ίδιες διαφάνειες ο οποίος είναι βασισμένος στον αλγόριθμο GraphSearch. Επειδή ο αλγόριθμος αυτός είναι επαναληπτικός, κάθε φορά που γίνεται expand ένας κόμβος, μπαίνουν όλα τα παιδιά του στη στοίβα, γίνεται pop το “πάνω-πάνω” στοιχείο της στοίβας και τα υπόλοιπα παραμένουν στην στοίβα για να “εξερευνηθούν” αργότερα. Τα παιδιά μπαίνουν στην στοίβα με αλφαβητική σειρά επομένως είναι προφανές (και φαίνεται στα αποτελέσματα) ότι θα βγουν από το σύνορο-στοίβα με την αντίθετη σειρά. Οι σειρά των κόμβων που βγαίνουν από το σύνορο είναι:

o103, ts, mail, o109, o119, storage, o123, r123

• Αναζήτηση πρώτα σε βάθος με επαναληπτική εκβάθυνση
Ο IDS βασίζεται και αυτός στον GraphSearch, επομένως ισχύει το ίδιο με τον DFS για την αλφαβητική σειρά που βγαίνουν οι κόμβοι από το σύνορο. Επίσης ο IDS δεν αποθηκεύει τους κόμβους που επισκέπτεται, άρα μπορεί σε μια επανάληψη να εξετάσουμε τον ίδιο κόμβο παραπάνω από μια φορά (π.χ. 4η επανάληψη κόμβος o109).
Οι σειρά των κόμβων που βγαίνουν από το σύνορο για κάθε επανάληψη είναι:

Για μέγιστο_βάθος=0 (1η επανάληψη):
o123
Για μέγιστο_βάθος=1 (2η επανάληψη):
o103, ts, o109, b3
Για μέγιστο_βάθος=2 (3η επανάληψη):
o103, ts, mail, o109, o119, o111, b3, b4, b1
Για μέγιστο_βάθος=3 (4η επανάληψη):
o103, ts, mail, o109, o119, storage, o123, o111, b3, b4, o109, b1, c2, b2
Για μέγιστο_βάθος=4 (5η επανάληψη):
o103, ts, mail, o109, o119, storage, o123, r123

• Άπληστη αναζήτηση πρώτα στον καλύτερο με ευρετική συνάρτηση την h
Η αναζήτηση αυτή βασίζεται στον GraphSearch και το σύνορο υλοποιείται με μια ουρά προτεραιότητας.
Οι σειρά των κόμβων που βγαίνουν από το σύνορο για κάθε επανάληψη είναι:

o103, b3, b1, c2, c1, c3, b2, b4, ts, o109, o119, o123, r123

• A* με ευρετική συνάρτηση την h

o103, b3, b1, c2, c1, b2, b4, c3, ts, o109, o119, mail, o123, r123

Πρόβλημα 4

Ερώτημα α

Για να οριστεί το πρόβλημα πρέπει να ορίσουμε: έναν χώρο καταστάσεων, την αρχική κατάσταση, την κατάσταση στόχου, την συνάρτηση διαδόχων και την συνάρτηση κόστους.
Για το πρόβλημα της εκφώνησης έχουμε:

Χώρος καταστάσεων:
Μια τρέχουσα κατάσταση χαρακτηρίζεται από: την τρέχουσα θέση του ρομπότ, το πακέτο που κουβαλάει την τρέχουσα στιγμή, και μία λίστα με όλα τα πακέτα καθώς και την πληροφορία για το αν έχει παραδοθεί το κάθε ένα ή όχι. Πιο συγκεκριμένα:
τρέχουσα_θέση=(συντεταγμένες της θέσης του ρομπότ)
πακέτο=(σημείο_παραλαβής, σημείο_παράδοσης)
λίστα_πακέτων=[(πακέτο_1, έχει_παραδοθεί), (πακέτο_2, έχει_παραδοθεί), … , (πακέτο_N, έχει_παραδοθεί)]
* η μεταβλητή έχει παραδοθεί είναι μια boolean μεταβλητή και μας δείχνει αν το πακέτο έχει παραδοθεί ή όχι
* οι μεταβλητές πακέτο_1, πακέτο_2, … είναι μεταβλητές τύπου πακέτο (που ορίστηκε παραπάνω)

Άρα μια κατάσταση ορίζεται ως:
κατάσταση=(τρέχουσα_θέση, πακέτο_που_κουβαλάει, λίστα_πακέτων)
* οι μεταβλητή πακέτο_που_κουβαλάει περιέχει είτε το πακέτο που κουβαλάει το ρομπότ (δηλαδή μια μεταβλητή τύπου πακέτο) είτε είναι κενή αν δεν κουβαλάει κάποιο πακέτο (π.χ. περιέχει NULL)

Αρχική Κατάσταση:
Το ρομπότ αρχικά ξεκινάει από το mail, δεν κουβαλάει κανένα πακέτο και κανένα πακέτο δεν έχει παραδοθεί. Άρα η αρχική κατάσταση είναι:
αρχική_κατάσταση=(αρχική_θέση, πακέτο_που_κουβαλάει, λίστα_πακέτων) όπου
αρχική_θέση = mail
πακέτο_που_κουβαλάει = NULL
λίστα_πακέτων=[(πακέτο_1, False), (πακέτο_2, False), … , (πακέτο_N, False)] (δηλαδή δεν έχει παραδοθεί κανένα πακέτο)

Κατάσταση Στόχου:
Το ρομπότ πρέπει να έχει παραδώσει όλα τα πακέτα και να επιστρέψει στο mail. Άρα πρέπει:
κατάσταση_στόχου=(τελική_θέση, πακέτο_που_κουβαλάει, λίστα_πακέτων) όπου
τελική_θέση = mail
πακέτο_που_κουβαλάει = NULL
λίστα_πακέτων=[(πακέτο_1, True), (πακέτο_2, True), … , (πακέτο_N, True)] (δηλαδή να έχει παραδώσει όλα τα πακέτα)

Συνάρτηση διαδόχων:
Επιστρέφει μια λίστα με όλες τις κινήσεις που μπορεί να κάνει το ρομπότ από την τρέχουσα θέση.
Πιο συγκεκριμένα μια κίνηση χαρακτηρίζεται από τις συντεταγμένες της επόμενης θέσης αλλά και την κίνηση που πρέπει να κάνει για να πάει στην επόμενη θέση. Δηλαδή:
επόμενη_κίνηση=(νέα_κατάσταση, action)
Όπου νέα_κατάσταση είναι μια κατάσταση, δηλαδή περιέχει την νέα θέση, πληροφορία για το αν κουβαλάει πακέτο καθώς και την ανανεωμένη λίστα πακέτων (και της πληροφορίας παράδοσης για το κάθε ένα) για την νέα κίνηση.
Η συνάρτηση διαδόχων επιστρέφει μια λίστα από επόμενες καταστάσεις δηλαδή:
[επόμενη_κίνηση_1, επόμενη_κίνηση_2, … , επόμενη_κίνηση_N].
Το ποιες πραγματικά είναι οι αποδεκτές επόμενες καταστάσεις ορίζεται από τους κανόνες του προβλήματος, ο οποίοι δεν αναφέρονται στην εκφώνηση. Μπορούμε να υποθέσουμε ότι είναι ίδιες με του pacman, δηλαδή ότι από μια τρέχουσα θέση μπορείς να κινηθείς δεξιά, αριστερά, πάνω και κάτω με την προϋπόθεση ότι δεν χτυπάει σε τοίχο.

Συνάρτηση κόστους:
Με την υπόθεση ότι το ρομπότ μπορεί να κάνει κάθε φορά ένα βήμα σε μία από τις κατευθύνσεις: πάνω, κάτω, αριστερά και δεξιά, ορίζουμε το κόστος κάθε κίνησης να είναι ίσο με 1 (αφού κάνει ένα βήμα την φορά).
Άρα ορίζουμε την συνάρτηση κόστους ως εξής:
κόστος(κατάσταση_1, κατάσταση_2, action) = 1
* με την προϋπόθεση ότι η κατάσταση 2 είναι μία από τις καταστάσεις που επιστρέφει η συνάρτηση διαδόχων όταν κληθεί για την κατάσταση_1

Ερώτημα β

Αρχικά χρειαζόμαστε μία συνάρτηση η οποία να εκτιμά την απόσταση μεταξύ 2 σημείων χωρίς αυτή να υπερβαίνει ποτέ την πραγματική απόσταση. Μπορούμε να χρησιμοποιήσουμε την ευκλείδεια απόσταση η οποία σίγουρα δεν υπερεκτιμά την απόσταση. Μια άλλη επιλογή είναι να χρησιμοποιήσουμε την απόσταση manhattan η οποία όμως, ανάλογα το πρόβλημα ίσως και να υπερεκτιμά την απόσταση (π.χ. αν το ρομπότ μπορεί να κινηθεί και διαγώνια). Εφόσον δεν αναφέρονται αυτές οι λεπτομέρειες στην εκφώνηση, επιλέγω την ευκλείδεια απόσταση. Σε όλο το υπόλοιπο ερώτημα, όπου αναφέρω απόσταση εννοώ ευκλείδεια απόσταση.

Η αρχική σκέψη για την ευρετική συνάρτηση είναι ότι ασχέτως με ποια σειρά θα επιλέξει το ρομπότ να παραδώσει τα πακέτα, σίγουρα θα διανύσει την απόσταση από το σημείο παραλαβής στο σημείο παράδοσης για κάθε πακέτο (προφανώς μιλάμε για τα πακέτα τα οποία δεν έχουν παραδοθεί σε μία δοθείσα κατάσταση). Αυτό συμβαίνει διότι το ρομπότ μπορεί να κουβαλάει μόνο ένα πακέτο την φορά. Δηλαδή αφού πρέπει να παραδώσει όλα τα πακέτα για να φτάσει στον τελικό στόχο για κάθε πακέτο που απομένει σίγουρα αφού το παραλάβει θα πάει κατευθείαν στο σημείο παράδοσης του, ασχέτως με το ποια πακέτα έχει παραδώσει πριν ή θα παραδώσει μετά. Άρα για να φτάσει το ρομπότ από μία τρέχουσα κατάσταση σε μία κατάσταση στόχου σίγουρα θα διανύσει την εξής απόσταση:

totalCost = 0
Για κάθε πακέτο που πρέπει να παραδοθεί:
totalCost = totalCost + απόσταση(σημείο_παραλαβής, σημείο_παράδοσης)

Για να βελτιώσω ακόμα περισσότερο την ευρετική συνάρτηση στηρίζομαι στο ότι το ρομπότ ξεκινάει πάντα από το δωμάτιο mail. Άρα σίγουρα θα πρέπει να διανύσει κάποια απόσταση για να μεταβεί από το mail στο πρώτο σημείο παραλαβής δέματος. Προφανώς δεν ξέρουμε ποιό πακέτο θα παραλάβει πρώτο, αλλά μπορούμε να υπολογίσουμε την καλύτερη περίπτωση. Πιο συγκεκριμένα, υπολογίζουμε την απόσταση μεταξύ του mail και όλων των σημείων παραλαβής και επιλέγουμε την μικρότερη, δηλαδή το κοντινότερο σημείο παραλαβής από το mail. Μπορούμε να προσθέσουμε την απόσταση αυτήν καθώς δεν υπερεκτιμά ποτέ την απόσταση που θα διανύσει το ρομπότ, αφού αυτή είναι η ελάχιστη απόσταση που θα διανύσει το ρομπότ για να πάει στο πρώτο πακέτο. Άρα έχουμε:

κοντινότερη_παραλαβή = ∞
Για κάθε πακέτο που πρέπει να παραδοθεί:
    κοντινότερη_παραλαβή = min{ κοντινότερη_παραλαβή, απόσταση(mail, σημείο_παραλαβής) }
totalCost = totalCost + κοντινότερη_παραλαβή

Τέλος, ευρετική συνάρτηση μπορεί να βελτιωθεί λίγο ακόμα αν λάβουμε υπόψιν ότι το ρομπότ αφού παραδώσει το τελευταίο πακέτο πρέπει να επιστέψει στο mail και ακολουθήσουμε την ίδια λογική με την αμέσως παραπάνω παράγραφο. Πιο συγκεκριμένα μόλις το ρομπότ ολοκληρώσει την τελευταία παράδοση θα βρίσκεται στο σημείο παράδοσης του πακέτου αυτού. Προφανώς δεν γνωρίζουμε ποιο θα είναι το πακέτο αυτό. Μπορούμε όμως να υπολογίσουμε το κοντινότερο σημείο παράδοσης στο mail. Έτσι ξέρουμε ότι το ρομπότ σε κάθε περίπτωση θα διανύσει τουλάχιστον αυτήν την απόσταση για να γυρίσει στο mail. Δηλαδή:

κοντινότερη_παράδοση = ∞
Για κάθε πακέτο που πρέπει να παραδοθεί:
    κοντινότερη_παράδοση = min{ κοντινότερη_παράδοση, απόσταση(mail, σημείο_παράδοσης) }
totalCost = totalCost + κοντινότερη_παράδοση

Προσθέτουμε και την απόσταση αυτή στις προηγούμενες και έτσι έχουμε μια πολύ καλή προσέγγιση του πραγματικού κόστους. Άρα συνολικά έχουμε:

totalCost = 0
κοντινότερη_παραλαβή = ∞ ← αρχικοποίηση των μεταβλητών
κοντινότερη_παράδοση = ∞
Για κάθε πακέτο που πρέπει να παραδοθεί:
    totalCost = totalCost + απόσταση(σημείο_παραλαβής, σημείο_παράδοσης)
    κοντινότερη_παραλαβή = min{ κοντινότερη_παραλαβή, απόσταση(mail, σημείο_παραλαβής) }
    κοντινότερη_παράδοση = min{ κοντινότερη_παράδοση, απόσταση(mail, σημείο_παράδοσης) }
totalCost = totalCost + κοντινότερη_παραλαβή
totalCost = totalCost + κοντινότερη_παράδοση

Το totalCost είναι το αποτέλεσμα που μας επιστρέφει η ευρετική για την δοθείσα κατάσταση.
Η ευρετική συνάρτηση είναι παραδεκτή καθώς η όπως αναφέρθηκε και στην αρχή η ευκλείδεια απόσταση δεν υπερεκτιμά ποτέ την απόσταση 2 σημείων και επίσης όπως εξηγήθηκε σε κάθε μία παράγραφο οι αποστάσεις που προσθέτουμε θα γίνουν κάθε φόρα ανεξαρτήτως της σειράς που το ρομπότ θα παραδώσει τα πακέτα. Έτσι υ ευρετική δεν υπερεκτιμά ποτέ ρο πραγματικό κόστος, δηλαδή είναι παραδεκτή.

Πρόβλημα 5

Τυπικά για να είναι πλήρης ένας αλγόριθμος αμφίδρομης αναζήτησης, δηλαδή ότι η λύση (εάν υπάρχει) θα βρεθεί σίγουρα, αρκεί μόνο ένας από τους 2 αλγορίθμους να είναι πλήρης καθώς ο αλγόριθμος αυτός θα βρει σίγουρα την λύση ακόμη και αν δεν “συναντηθεί” ποτέ με τον άλλον αλγόριθμο. Στην περίπτωση αυτή όμως δεν έχει κανένα νόημα η αμφίδρομη αναζήτηση καθώς θα ήταν προτιμότερο να “τρέχαμε” μόνο του τον αλγόριθμο αυτό. Έτσι, για κάθε έναν από τους αλγορίθμους (α)-(δ) θα θεωρήσω ότι για να είναι πλήρης πρέπει να είναι πλήρεις και οι δύο αλγόριθμοι που χρησιμοποιεί. Την ίδια λογική ακολουθούν και οι διαφάνειες και το βιβλίο του μαθήματος καθώς για την πληρότητα ενός αλγορίθμου αμφίδρομης αναζήτησης απαιτείται να ισχύουν οι προϋποθέσεις πληρότητας και των 2 αλγορίθμων.
Επίσης για να είναι βέλτιστος ο αλγόριθμος αμφίδρομης αναζήτησης πρέπει να ισχύουν οι προϋποθέσεις βέλτιστης συμπεριφοράς και για τους 2 αλγορίθμους που χρησιμοποιούνται.

- (α) Αναζήτηση πρώτα σε πλάτος και αναζήτηση περιορισμένου βάθους

Για να είναι πλήρης αυτή η αμφίδρομη αναζήτηση πρέπει:
- Ο παράγοντας διακλάδωσης να είναι πεπερασμένος (προϋπόθεση πληρότητας BFS).
- Το όριο βάθους (έστω L) να είναι μεγαλύτερο ή ίσο από το βάθος της λύσης (έστω D), δηλαδή να ισχύει $L \geq D$ (προϋπόθεση πληρότητας του DLS).

Ο αλγόριθμος αυτός δεν είναι βέλτιστος καθώς χρησιμοποιεί τον αλγόριθμο αναζήτησης περιορισμένου βάθους (DLS) ο οποίος δεν είναι βέλτιστος. Ο DLS δεν είναι βέλτιστος καθώς είναι βασισμένος στον DFS (ο οποίος επίσης δεν είναι βέλτιστος), δηλαδή παράγει πολύ περισσότερους κόμβους από ότι χρειάζεται για να βρει τον στόχο. Αυτό συμβαίνει για οποιοδήποτε $L \geq D$, δηλαδή ανεξαρτήτως του ορίου βάθους ο DLS δεν είναι βέλτιστος.

(β) Αναζήτηση με επαναληπτική εκβάθυνση και αναζήτηση περιορισμένου βάθους

Για να είναι πλήρης αυτή η αμφίδρομη αναζήτηση πρέπει:

- Ο παράγοντας διακλάδωσης να είναι πεπερασμένος καθώς η αναζήτηση με επαναληπτικής εκβάθυνσης (IDS) είναι βασισμένη στον DFS. Επομένως προϋπόθεση πληρότητας του IDS είναι να ισχύει η προϋπόθεση πληρότητας του DFS.
- Το όριο βάθους (έστω L) να είναι μεγαλύτερο ή ίσο από το βάθος της λύσης (έστω D), δηλαδή να ισχύει (προϋπόθεση πληρότητας του DLS).

Όπως αναφέρθηκε και παραπάνω, για να είναι βέλτιστος ο αλγόριθμος αμφίδρομης αναζήτησης πρέπει να είναι βέλτιστοι και οι δυο αλγόριθμοι που χρησιμοποιεί. Αφού χρησιμοποιείται και σε αυτήν την περίπτωση ο DLS, ο οποίος δεν είναι βέλτιστος (το οποίο εξηγήθηκε στο (α) και επίσης αναφέρεται στις διαφάνειες και το βιβλίο), ο αλγόριθμος αμφίδρομης αναζήτησης δεν είναι βέλτιστος.

(γ) A* και αναζήτηση περιορισμένου βάθους

Για να είναι πλήρης αυτή η αμφίδρομη αναζήτηση πρέπει:

- Ο παράγοντας διακλάδωσης να είναι πεπερασμένος, το κόστος κάθε ενέργειας να είναι θετικό και η ευρετική συνάρτηση που χρησιμοποιεί ο A* να μην υπερεκτιμά το πραγματικό κόστος (δηλαδή να είναι παραδεκτή). (προϋποθέσεις πληρότητας του A*)
- Το όριο βάθους (έστω L) να είναι μεγαλύτερο ή ίσο από το βάθος της λύσης (έστω D), δηλαδή να ισχύει (προϋπόθεση πληρότητας του DLS).

Ο αλγόριθμος αυτός δεν είναι βέλτιστος αφού και αυτός χρησιμοποιεί αναζήτηση περιορισμένου βάθους. Άρα για τους λόγους που εξηγήθηκαν στα (α) και (β) η αναζήτηση αυτή δεν είναι βέλτιστη.

(γ) A* και A*

Για να είναι πλήρης αυτή η αμφίδρομη αναζήτηση πρέπει:

- Ο παράγοντας διακλάδωσης να είναι πεπερασμένος, το κόστος κάθε ενέργειας να είναι θετικό και η ευρετική συνάρτηση που χρησιμοποιεί ο A* να μην υπερεκτιμά το πραγματικό κόστος (δηλαδή να είναι παραδεκτή). Δηλαδή να ισχύουν οι προϋποθέσεις πληρότητας του A* (αφού χρησιμοποιεί A* και για τις 2 αναζητήσεις).

Εφόσον χρησιμοποιείται ο ίδιος αλγόριθμος και για τις 2 αναζητήσεις (A*) για να είναι βέλτιστη η αμφίδρομη αναζήτηση αρκεί να είναι βέλτιστος ο A* (δηλαδή να ισχύουν οι προϋποθέσεις βέλτιστης συμπεριφοράς του A*). Για να είναι βέλτιστος ο A* έχουμε της εξής 2 επιλογές:

- Να επεκταθεί ο αλγόριθμος GraphSearch (ο standard αλγόριθμος για αναζήτηση σε γράφους που παρουσιάζεται στις διαφάνειες του μαθήματος) έτσι ώστε να απορρίπτει την πιο δαπανηρή από δύο διαδρομές που βρίσκει προς τον ίδιο κόμβο.
- Η ευρετική συνάρτηση που χρησιμοποιεί ο A* να είναι συνεπής.

Αν ισχύει τουλάχιστον μία από τις 2 παραπάνω προϋποθέσεις τότε ο αλγόριθμος A* είναι βέλτιστος και κατά συνέπεια ο αλγόριθμος αμφίδρομης αναζήτησης (που χρησιμοποιεί A* και για τις δύο αναζητήσεις) είναι και αυτός βέλτιστος.

Έλεγχος για το αν οι δύο αναζητήσεις συναντιούνται

Ψάχνουμε μια αποδοτική λύση για να γίνεται ο έλεγχος για το αν οι δύο αναζητήσεις συναντιούνται. Αυτό σημαίνει ότι ο έλεγχος πρέπει να γίνεται την ώρα που τρέχουν οι 2 αναζητήσεις έτσι ώστε μόλις συναντηθούν να σταματήσουν οι αναζητήσεις καθώς έχουμε βρει ένα μονοπάτι. Μια λύση είναι οι αλγόριθμοι να εκτελούνται διαδοχικά κάνοντας ένα βήμα ο καθένας. Για παράδειγμα, θα κάνει ένα βήμα ο BFS, μετά ένα βήμα ο DLS, μετά το επόμενο βήμα ο BDS κ.ο.κ. . Ο έλεγχος για το αν συναντήθηκαν θα γίνεται μετά απο την εκτέλεση ενός σετ βημάτων, δηλαδή η σειρά των λειτουργιών στο προηγούμενο παράδειγμα είναι: βήμα BFS, βήμα DLS, έλεγχος συνάντησης, βήμα BFS, κ.ο.κ. . Το ποιό θα είναι το βήμα που θα κάνει ο κάθε αλγόριθμος ορίζεται στο κάθε πρόβλημα ξεχωριστά.

Αλγόριθμος α

Ακολουθώντας την παραπάνω λογική ορίζουμε ως ένα βήμα του BFS το standard βήμα του αλγορίθμου, δηλαδή pop ενός κόμβου από το σύνορο, expand αυτού του κόμβοι και προσθήκη των παιδιών του στο σύνορο. Ομοίως ορίζουμε και το βήμα του DLS. Επίσης υποθέτουμε ότι πρώτα γίνεται το βήμα του BFS και μετά του DLS. Όπως γνωρίζουμε ο BFS αποθηκεύει τους κόμβους που επισκέπτεται, ενώ ο DLS όχι. Έτσι, δεν μπορούμε να κάνουμε τον έλεγχο αφού ολοκληρωθούν και τα 2 βήματα. Για αυτό, επεκτείνουμε τον αλγόριθμο του DLS έτσι ώστε μόλις κάνει pop έναν κόμβο από τον σύνορο να ελέγχει αν ο κόμβος αυτός υπάρχει στους κόμβους που έχει επισκεφτεί ο BFS. Δηλαδή ο έλεγχος για τον αν οι αναζητήσεις συναντιούνται γίνεται εσωτερικά σε κάθε βήμα του DLS. Αν υπάρχει σημαίνει ότι τον συγκεκριμένο κόμβο τον έχουν επισκεφτεί και οι 2 αναζητήσεις, δηλαδή συναντήθηκαν και έτσι μπορούμε να σταματήσουμε τον αλγόριθμο αμφίδρομης αναζήτησης καθώς έχει βρεθεί ένα μονοπάτι. Τέλος, μπορούμε να υποθέσουμε ότι ο BFS αποθηκεύει τους κόμβους που επισκέπτεται σε κάποια δομή όπως set, hash table, dictionary στις οποίες μπορούμε να κάνουμε αναζήτηση σε χρόνο O(1). Δηλαδή ο έλεγχος χρειάζεται χρόνο O(1) και καθόλου επιπλέον χώρο, άρα είναι πολύ αποδοτικός.

Αλγόριθμος β

Στον αλγόριθμο αυτόν καμία από τις 2 αναζητήσεις δεν αποθηκεύει τους κόμβους που επισκέπτεται. Ορίζουμε ως βήμα της αναζήτησης επαναληπτικής εκβάθυνσης (IDS) μια ολόκληρη αναζήτηση για ένα συγκεκριμένο depth limit, δηλαδή μια εκτέλεση του DLS για το τρέχον depth limit (το οποίο είναι 0 στο πρώτο βήμα και αυξάνεται κατά 1 σε κάθε βήμα). Για DLS ορίζουμε ως ένα βήμα το standard βήμα του αλγορίθμου, δηλαδή pop ενός κόμβου από το σύνορο, expand αυτού του κόμβοι και προσθήκη των παιδιών του στο σύνορο. Για να μπορούμε να κάνουμε τον έλεγχο, σε κάθε βήμα του IDS αποθηκεύουμε τους κόμβους τους οποίους βρίσκονται στο τελευταίο επίπεδο που έψαξε ο αλγόριθμος, δηλαδή το depth limit του βήματος αυτού. Σε κάθε βήμα διαγράφονται οι προηγούμενοι αποθηκευμένοι κόμβοι και αποθηκεύονται οι καινούργιοι οι οποίοι βρίσκονται στο επόμενο επίπεδο από τους προηγούμενους. Για τον DLS, ομοίως με το ερώτημα α, σε κάθε βήμα του όταν γίνεται pop ένας κόμβος ελέγχουμε αν ο κόμβος αυτός βρίσκεται στους αποθηκευμένους κόμβους του IDS, δηλαδή αν οι 2 αναζητήσεις συναντήθηκαν.
Η λύση αυτή μας κοστίζει λίγο σε χώρο καθώς πρέπει να αποθηκεύουμε κάθε φορά κάποιους κόμβους. Ο χώρος αυτός, όμως, είναι πολύ μικρότερος από ότι χρησιμοποιούν άλλοι αλγόριθμοι όπως ο BFS καθώς κάθε φορά είναι αποθηκευμένοι οι κόμβοι που βρίσκονται στο τρέχον βαθύτερο επίπεδο (δηλαδή τα φύλλα του έως την τρέχουσα στιγμή εξερευνημένου δένδρου) και όχι όλους τους κόμβους που έχει επισκεφτεί η αναζήτηση. Επίσης η αναζήτηση στο explored set γίνεται και πάλι σε χρόνο O(1).
Άρα η λύση αυτή είναι αρκετά αποδοτική και βρίσκει πάντα το αν συναντήθηκαν οι αναζητήσεις.

Αλγόριθμος γ

Για τον αλγόριθμο αυτόν ακολουθούμε ακριβώς την ίδια λογική με τον αλγόριθμο α. Πιο συγκεκριμένα, στην αναζήτηση A* αποθηκεύονται οι κόμβοι που επισκέπτονται ενώ στον DLS όχι. Ορίζουμε ως βήμα των A* και DLS τα standard βήματα των αλγορίθμων, δηλαδή pop ενός κόμβου από το σύνορο, expand του κόμβου αυτού και προσθήκη των “παιδιών” του στο σύνορο (υπό τις προϋποθέσεις του καθενός). Πάντα εκτελείται πρώτα το βήμα του A* και μετά του DLS. Πάλι, επεκτείνουμε τον αλγόριθμο του DLS έτσι ώστε με το που κάνει pop έναν κόμβο να ελέγχει το explored set του A*. Έτσι, ομοίως με το ερώτημα α έχουμε έναν πλήρως αποδοτικό έλεγχο για το αν οι 2 αναζητήσεις συναντιούνται.

Αλγόριθμος δ

Ο αλγόριθμος αυτός χρησιμοποιεί A* και για τις δύο αναζητήσεις, επομένως και οι 2 αναζητήσεις αποθηκεύουν τους κόμβους τους οποίους έχουν επισκεφτεί. Ορίζουμε σαν βήμα και των δύο αναζητήσεων το standard βήμα (όπως και στον αλγόριθμο γ). Στον αλγόριθμο αυτόν δεν χρειάζεται να παρέμβουμε στην αναζήτηση, όπως κάναμε στον DLS. Ο έλεγχος μπορεί να γίνει μετά από κάθε σετ βημάτων, δηλαδή: κάνει ένα βήμα ο πρώτος A*, κάνει ένα βήμα ο δεύτερος A* και μετά την ολοκλήρωση του βήματος του δεύτερου A* ελέγχουμε για τον αν συναντήθηκαν. Ο έλεγχος αυτός γίνεται ελέγχοντας αν υπάρχει κάποιος κοινός κόμβος στα explored set των δύο αναζητήσεων A*. Αν υπάρχει σημαίνει ότι συναντήθηκαν.
Αυτή είναι μια απλή και αποδοτική λύση για να ελέγξουμε αν οι δύο αναζητήσεις συναντιούνται.
Διαφορετικά , θα μπορούσαμε να ακολουθήσουμε την ίδια λογική με τα παραπάνω ερωτήματα, δηλαδή να τροποποιήσουμε τον δεύτερο A* έτσι ώστε να ελέγχει εσωτερικά το explored set του πρώτου A*.