email at: sdi2000055@di.uoa.gr

Περιεχομενα

1. Ασκηση 1

Ο αλγόριθμος DFS έχει υλοποιηθεί όπως και στις διαφάνειες του μαθήματος. Για το σύνορο έχει χρησιμοποιηθεί στοίβα η οποία ήταν ήδη υλοποιημένη από το util.py, ενώ στην στοίβα βάζουμε το ζεύγος του κόμβου, και του μονοπατιού που έχει ακολουθηθεί μέχρι εκείνη την στιγμή προς αυτόν (path). Τέλος για το αν έχουμε φτάσει σε κατάσταση στόχου ελέγχουμε όταν βγάζουμε έναν κόμβο από την στοίβα, ενώ ελέγχουμε προτού βάλουμε κόμβο να μην έχει ήδη εξερευνηθεί.

2. Ασκηση 2

Όπως και με τον DFS, η υλοποίηση του BFS είναι πολύ παρόμοια. Διαφορά προφανώς συναντάται στην δομή για το σύνορο, η οποία εδώ είναι ουρά, ενώ για να είμαστε πιστοί με την υλοποίηση στις διαφάνειες ελέγχουμε, πριν εισάγουμε επόμενο κόμβο στο σύνορο, να μην είναι ήδη εξερευνημένος και να μην υπάρχει ήδη στο σύνορο (αυτό συμβαίνει με το (states[0] for states in frontier.list) το οποίο μετατρέπει το σύνορο σε μία λίστα η οποία να αποτελείται από το πρώτο στοιχείο των ζεύγων (tuples) που εισάγονται εκεί, δηλαδή των κόμβων.

3. Ασκηση 3

Για τον UCS ακολούθησα μια πιο περίπλοκη λογική για να κρατάω το μονοπάτι.
Πρακτικά κάθε κόμβος αποθηκεύει σε ένα δικό του dictionary κάποιες σημαντικές πληροφορίες για το ίδιο, όπως το ποιος είναι ο γονιός του (parent), ποιά η κατάστασή του (state), το κόστος (cost) και η κίνηση για να φτάσουμε προς αυτόν (action). Προφανώς ο αρχικός κόμβος δεν έχει πατέρα, προηγούμενη κίνηση που να οδηγεί προς αυτόν, μηδενικό κόστος και η κατάσταση του είναι η αρχική. Ολη αυτήν την πληροφορία να την προσθέτω στο σύνορο ,το οποίο είναι ουρά προτεραιότητας, μαζί με το αντίστοιχο ΄βάρος' που χρειάζεται η ουρά προτεραιότητας για να κρατάει την σειρά, με το βάρος αυτό να είναι το κόστος.
Έπειτα σε κάθε επανάληψη (μέχρι να είναι άδειο το σύνορο), δέχομαι τον κόμβο από την ουρά προτεραιότητας. Αν έχουμε ήδη επισκεπτεί τον κόμβο, συνεχίζουμε ενώ

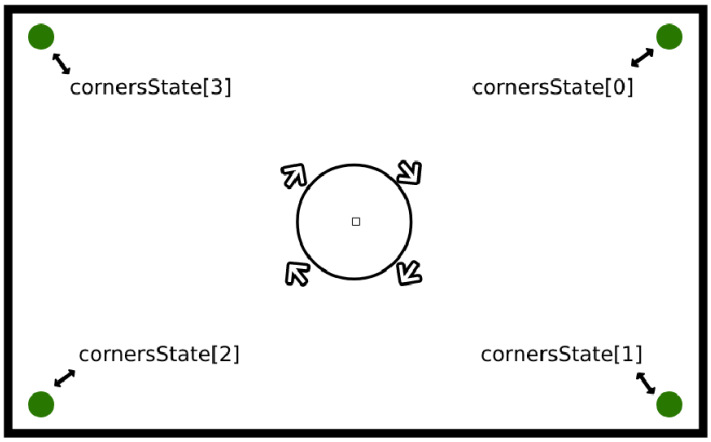
αν φτάσαμε σε κατάσταση στόχου σπάμε εκτός της επανάληψης για να λάβουμε το μονοπάτι. Αν δεν ισχύει τίποτα αυτών, βάζω τον κόμβο στους εξερευνημένους και για κάθε επόμενο κόμβο ss → successor state, αφού ελέγξω για το οτι δεν τον έχω εξερευνήσει, του δημιουργώ το δικό του dictionary το οποίο συνδέω κατάλληλα με τον πρόγονο από τον οποίο ήρθε (δηλαδή ως γονιό ορίζω τον παραπάνω κόμβο που έκανα expand, ως κίνηση για να φτάσω εκεί ορίζω την κίνηση που μου έδωσε η getSuccessors, και προσθέτω και το κατάλληλο cost στο ήδη υπάρχων.) Τέλος ξανά όλη αυτην την πληφορία την προσθέτω στο σύνορο μαζί με το νέο κόστος.
Όταν θα βγούμε από όλη αυτήν την λούπα επανάληψης θα έχουμε στο node αποθηκευμένο τον κόμβο που είναι στην κατάσταση στόχου. Άρα λόγω την συνδεσης των κόμβων, αν ακολουθήσω τον γονιό μπορώ να φτάσω τελικώς στον αρχικό κόμβο και να έχω ακολουθήσει το σωστό μονοπάτι προς τα πίσω.

4. Ασκηση 4

Ο αλγόριθμος για το A* είναι σχεδόν ολόιδιος με τον UCS της (3) με μόνη διαφορά ότι πλέον για τον κάθε κόμβο αποθηκέυω και την τιμή της ευριστικής συνάρτησης στο σημείο αυτό, και ως βάρος για το σύνορο προσθέτω το κόστος + την τιμή της ευριστικής.

5. Ασκηση 5

__init__
Για να κρατήσουμε πληροφορία σχετικά με τις γωνίες που έχουμε επισκεφτεί προφανώς χρειαζόμαστε μια δομή. Αποφάσισα η δομή αυτη να είναι μια απλή λίστα η οποία σώζει πληροφορίες για τις επισκεφθείσες γωνίες με βάσει την ροή του ρολογιού. Δηλαδή η πρώτη θέση μιλάει για την πάνω δεξιά γωνία, η δεύτερη θέση για την κάτω δεξιά, η τρίτη θέση για την κάτω αριστερά και η τελευταία θέση για την πάνω αριστερά.



(1)

isGoalState

Σκοπός μας είναι να έχουμε επισκεφθεί όλες τις γωνίες, άρα κοιτάζουμε η λίστα έχει μόνο τιμές True.

getSuccessors

Για κάθε μία από τις επιτρεπτές κινήσεις, υπολογίζουμε την επόμενη θέση και αν αυτή δεν χτυπάει τοίχο (γιατί αν χτυπάει δεν επιτρέπεται να είναι απόγονος), τοτε προσθέτουμε στην λίστα successors το επόμενο state, την δράση, και το κόστος της κίνησης που είναι 1. Για το επόμενο state καλούμε την συνάρτηση γετΣυςςεσσορ-Στατε που επιστρέφει τις πληροφορίες για την επόμενη κατάσταση.

getSuccessorState

Υπολογίζει ξανά με βάσει την θέση μας, την επόμενη θέση και έπειτα ελέγχει αν αυτή χτυπάει κάποια γωνία (επίσης ελέγχει αν χτυπάει τοίχο, αλλά αυτός ο έλεγχος γίνεται και από την getSuccessors άρα πρακτικά δεν χρειάζεται ξανά έλεγχος, αλλά εγώ τον κάνω ως sanity-check). Τέλος επιστρέφει την επόμενη θέση, και φυσικά την ανανεωμένη λίστα για την κατάσταση των γωνιών.

6. Ασκηση 6

Αρχικά για την ευρετική χρησιμοποίησα την συνάρτηση απόστασης Manhattan η οποία είναι η ιδανική για χρήση όταν έχουμε 4 επιλογές κίνησης, αφού δεν υπερκτιμά ποτέ την απόσταση με βάσει τις κινήσεις που θα πρέπει να κάνει ο pacman.
Έπειτα, για να είναι παραδεκτή ,αρα και συνεπής, πρέπει να είναι μια ευρετική η οποία δεν θα υπερεκτιμά την απόσταση που θα πρέπει να κάνει ο pacman. Αφού έχουμε τις γωνίες που έχει χτυπήσει, είναι προφανές ότι θα πρέπει να χρησιμοποιηθούν για την ευρετική.
Η ιδέα που σκέφτηκα, αν και απλή, φαίνεται να δουλεύει καλά. Συγκεκριμένα, κάνει expand 692 κόμβους. Η ιδέα είναι:
Αρχικά λαμβάνουμε τις γωνίες που δεν έχουμε χτυπήσει ακόμα (μέσω της συνάρτησης που όρισα εγώ getCornersNotHit). Έπειτα, υπολογίζω τις αποστάσεις της θέσης μας τώρα στην πιο κοντινή γωνία, και θέτω την θέση του pacman να είναι σε εκείνη την θέση για την επόμενη επανάληψη. Επαναλαμβάνω μέχρι να τελειώσουν οι γωνίες.
Η λύση είναι παραδεκτή αφού σίγουρα δεν υπερεκτιμά την απόσταση που θα χρειαστεί να κάνει ο πάκμαν, αφενώς λόγω της συνάρτησης απόστασης που επιλέξαμε, αλλά αφετέρου και λόγω της τακτικής που ορίσαμε να ακολουθήσουμε για την επίσκεψη των γωνιών (πήραμε πρακτικά την ελάχιστη απόσταση που θα πρέπει να ακολυθηθεί με χρήση Manhattan απόστασης).

7. Ασκηση 7

Η ευριστική αυτή μου φάνηκε αρκετά απαιτητική, για αυτό και έβαλα πολλη σκέψη στο να βγει καλή.
Αρχικά περνάμε 2 φορές από τα φαγητά για να υπολογίσουμε τις μεταξύ τους αποστάσεις, τις οποίες σώζω σε μία λίστα (distances) με tuples της μορφής (point

A, point B, distance between A and B), και επιλέγουμε το ζεύγος που έχει την μεγαλύτερη απόσταση μεταξύ τους, και προσθέτουμε την μικρότερη απόσταση του πάκμαν μεταξύ των σημείων A ή B.
Αυτό σημαίνει ότι:
Αν ο πάκμαν πάει στο πιο κοντινό από τα δύο φαγητά:
· Αν υπάρχουν μόνο δύο φαγητά, τότε η απόσταση μεταξύ τους είναι όση η ευρετική υπολόγισε.
· Αν έχει περισσότερα φαγητά θα κάνει σίγουρα μεγαλύτερη ή ίση απόσταση.
Αν δεν πάει στο πιο κοντινό από τα δύο φαγητά: Θα κάνει μια παραπάνω απόσταση που σίγουρα θα καταστήσει το τελικό αποτέλεσμα μεγαλύτερο από την τιμή της ευρετικής, κάποια στιγμή θα φτάσει σε ένα από τα δύο φαγητά:
· Αν υπάρχουν μόνο 2 φαγητά, τότε θα κάνει την μέγιστη απόσταση, άρα πάλι η απόσταση που διένυσε ηταν μεγαλύτερη (λόγω της αρχικής επιλογής)
· Αν υπάρχουν παραπάνω από 2 φαγητά και δεν επιλέξει να πάει στο τελικό, σίγουρα θα κάνει πάλι παραπάνω απόσταση, λόγω της τριγωνικής ανισότητας (η απόσταση μεταξύ των σημείων A και B είναι σίγουρα μικρότερη του αθροίσματος των αποστάσεων αν σπάσει και πάει απτό σημείο A στο Γ και απτό Γ στο B.
Άρα η ευρετική δεν υπερεκτιμά τις αποστάσεις άρα είναι παραδεκτή και λόγω χρήσης του mazeDistance είναι και συνεπής.

8. Ασκηση 8

Συμπλήρωσα το AnyFoodSearchProblem με το goal state του, το οποίο είναι προφανώς ότι είμαστε σε κατάσταση στόχου αν στην θέση μας υπάρχει φαγητό. Τέλος για τια την αναζήτηση στο findPathToClosestPath απλά χρησιμοποίησα την ήδη υλοποιημένη UCS. Όπως αναφέρεται και στην σελίδα άλλωστε η λύση πρέπει να είναι αρκετά σύντομη.
Γενική υποσημείωση: Στο αρχείο searchAgents.py έχω υλοποίησει και μια άλλη συνάρτηση για τον υπολογισμό απόστασης καθώς έψαχνα, την Chebyshev Distance η οποία εν τελει δεν μου χρειάστηκε κάπου, αλλά απόφασισα απλά να την αφήσω εκεί.