Project 1 Pacman

Μαραντίδης Θεοφάνης

1115201800106

Η χρήση του set() αντί της λίστας σε κάποια ερωτήματα γίνεται προκειμένου να μειώσουμε την πολυπλοκότητα του προγράμματος μας, αφού το set σε αντίθεση με την λίστα χρειάζεται O(1) και όχι O(n) για τις βασικές πράξεις.

Q1.

Το ερώτημα αυτό αφορά την υλοποίηση του αλγορίθμου αναζήτησης

πρώτα κατά βάθος (DFS).Ο αλγόριθμος έχει υλοποιηθεί ακριβώς όπως στις διαφάνειες.

Επεκτείνουμε πάντα τον βαθύτερο κόμβο του συνόρου αν δεν τον έχουμε ήδη επισκεφτεί(visited set).

Χρησιμοποιούμε σαν δομή μία στοίβα Stack που σημαίνει ότι ο πιο πρόσφατος κόμβος που παράχθηκε επιλέγεται για επέκταση.

Μέχρι να αδειάσει η στοίβα ή να φτάσουμε σε κατάσταση στόχου επεκτείνουμε τους κόμβους και κάνουμε push στην στοίβα κάθε φορά τον κόμβο και το κόστος του μονοπατιού που μας οδηγεί εκεί.

Q2.

Το ερώτημα αυτό αφορά την υλοποίηση του αλγορίθμου αναζήτησης

πρώτα κατά πλάτος (BFS). Ο αλγόριθμος έχει υλοποιηθεί ακριβώς όπως στις διαφάνειες.

Στην αναζήτηση πρώτα κατά πλάτος ο κόμβος ρίζα επεκτείνεται πρώτος μετά επεκτείνονται οι διάδοχοί του, μετά επεκτείνονται οι διάδοχοι των διαδόχων της ρίζας και ούτω κάθε εξής.

Χρησιμοποιούμε σαν δομή μία ουρά Queue (για το σύνορο = frontier)

στην οποία εισάγονται οι κόμβοι του κάθε επιπέδου διαδοχικά αν δεν τους έχουμε επισκεφτεί και δεν υπάρχουν και στην ουρά.

Αντίστοιχα με τον DFS κάνουμε push τον κόμβο και το κόστος του μονοπατιού που μας οδηγεί εκεί.

Q3.

Το ερώτημα αυτό αφορά την υλοποίηση του αλγορίθμου αναζήτησης ομοιόμορφου κόστους (UCS). Ο αλγόριθμος έχει υλοποιηθεί ακριβώς όπως στις διαφάνειες. Είναι σχεδόν ίδιος με τον BFS με την μόνη διαφορά ότι αντί να επεκτείνουμε τον πιο ρηχό κόμβο επεκτείνουμε τον κόμβο με το μικρότερο κόστος μοναπατιού.

Χρησιμοποιούμε σαν δομή μία ουρά προτεραιότητας Priority Queue (για το σύνορο = frontier) στην οποία κάνουμε push τον κόμβο και το μονοπάτι που μας οδηγεί σε αυτόν και ως priority το αντίστοιχο κόστος του μονοπατιού.

Κάνουμε pop() κάθε φορά τον κόμβο με το μικρότερο κόστος γεγονός που μπορεί να μας οδηγήσει στην βέλτιστη λύση εάν χρησιμοποιήσουμε το κόστος από την αρχική κατάσταση (state) στην τρέχουσα κατάσταση ως συνάρτηση κόστους.

Q4.

Το ερώτημα αυτό αφορά την υλοποίηση του αλγορίθμου A*.

Ο αλγόριθμος αυτός γνωρίζουμε ότι συνδυάζει τον αλγόριθμο UCS (f(x) = g(x)) και τον αλγόριθμο greedy-best-first-search(f(x) = h(x)) από τα οποία προκύπτει ότι f(x) = g(x) + h(x).

[g = κόστος από την αρχή στον κόμβο]

[h = εκτιμώμενο κόστος για το φθηνότερο μονοπάτι από τον κόμβο στην κατάσταση στόχου]

Ο αλγόριθμος A* είναι όμοιος με τον UCS με την διαφορά ότι στον A* χρησιμοποιούμε g+h αντί για g.

Χρησιμοποιούμε λοιπόν και εδώ μία ουρά προτεραιότητας ως δομή για το σύνορο, με προτεραιότητα το αποτέλεσμα της συνάρτησης f(n)=g(n)+h(n) [όπου n ο τρέχων κόμβος].

Κάνουμε λοιπόν pop() από το σύνορο τον κόμβο με το μικρότερο f(n).

Για κάθε παιδί του κόμβου που δεν έχουμε ήδη επισκεφτεί(visited set) υπολογίζουμε το f(n) και το κάνουμε push στην στοίβα μαζί με το path που χρειάζεται για να φτάσουμε εκεί.

Q5.

Το ερώτημα αυτό αφορά την υλοποίηση της κλάσης CornersProblem.

Στον constructor της κλάσης θέτουμε την μεταβλητή startingGameState = (self.startingPosition,())

Ένα tuple με την αρχική θέση του pacman και ένα κενό set που αναπαριστά τα corner που έχουμε επισκεφτεί.

Για την μέθοδο getStartState(self)

Επιστρέφουμε το startingGameState που ορίσαμε παραπάνω.

Για την μέθοδο isGoalState(self, state)

Παίρνουμε τις τρέχουσες συντεταγμένες του pacman και το set των corner που έχουμε επισκεφτεί.

Στη συνέχεια ελέγχουμε αν οι συντεταγμένες μας αντιστοιχούν σε κάποιο corner.Αν αντιστοιχούν και δεν υπάρχουν οι συντεταγμένες μέσα στο visited τις προσθέτουμε αφού επισκεφτήκαμε τον κόμβο.

Ελέγχουμε τέλος αν το πλήθος των corner που έχουμε επισκεφτεί είναι ίσο με 4. Αν είναι σημαίνει ότι βρισκόμαστε σε κατάσταση στόχου άρα και επιστρέφουμε True.

Σε κάθε αντίθετη περίπτωση επιστρέφουμε False.

Για την μέθοδο getSuccessors(self, state)

Επιστρέφουμε τα states των successors μαζί με τις ενέργειες που απαιτούνται για να φτάσουμε εκεί αλλά και το σταθερό κόστος = 1.

Δημιουργούμε μία κενή λίστα successors = []

Όπου θα βάζουμε τα tuples με τις πληροφορίες των successors του τρέχοντα κόμβου. [πχ successor = ((suc_coords, suc_corners),action,1)

Σε περίπτωση που από την κατάσταση που βρισκόμαστε μπορούμε να κάνουμε κάποια επιτρεπτή ενέργεια χωρίς να χτυπήσουμε σε τοίχο και να πάμε σε κάποια άλλη κατάσταση (next_state) κρατάμε τις συντεταγμένες του state αυτού.Ακόμα δημιουργούμε μία λίστα με τα visited corners των successors και ελέγχουμε αν ο successor από τον τρέχον κόμβο βρίσκεται σε κάποιο corner.Αν βρίσκεται και δεν υπάρχει μέσα στην λίστα με τα visited corners των successors τον προσθέτουμε.

Βάζουμε τέλος στην λίστα των διαδόχων το tuple για τον κάθε διάδοχο [((next_coords, sucs_corners),action,1)]

Q6.

Το ερώτημα αυτό αφορά την υλοποίηση μίας ευρετικής συνάρτησης

για το CornersProblem.

Η ευρετική συνάρτηση αυτή πρέπει να είναι συνεπής άρα και παραδεκτή.

Η γενική ιδέα είναι να θεωρήσουμε ένα πιο «χαλαρό» πρόβλημα κατά το οποίο δεν λαμβάνουμε υπόψη τους τοίχους.

Έτσι το κόστος της μετάβασης από μία θέση στην άλλη είναι η απόσταση Manhattan μεταξύ των δύο σημείων.

Επιστρέφουμε ένα κάτω φράγμα για την συντομότερη διαδρομή από το state που βρισκόμαστε στο στόχο(παραδεκτή).

Και συνεπής διότι για κάθε εκτέλεση μίας ενέργειας με σταθερό κόστος c μειώνεται η ευρετική μας συνάρτηση το πολύ κατά c.

Πιο συγκεκριμένα,

Δημιουργούμε μία λίστα με όλα τα corner που δεν έχουμε επισκεφτεί.

Υπολογίζουμε την απόσταση από την τρέχουσα θέση σε κάθε ένα από τα unvisted corners με την βοήθεια της μεθόδου manhattanDistance.

Επισκεπτόμαστε το κόρνερ με την μικρότερη απόσταση, το αφαιρούμε από την λίστα με τα unvisited corner και ανανεώνουμε την τρέχουσα θέση στο corner αυτό.

Συνεχίζουμε την ίδια διαδικασία μέχρι να επισκεφτούμε όλα τα corner.

Κρατάμε σε μία μεταβλητή το άθροισμα των κόστων των φθηνότερων διαδρομών κάθε φορά και το επιστρέφουμε.

Q7.

Το ερώτημα αυτό αφορά την υλοποίηση μίας ευρετικής συνάρτησης

για το FoodSearchProblem όπου πρέπει να βρούμε ένα μονοπάτι το οποίο να συλλέγει όλα τα φαγητά.

Σε αυτό το πρόβλημα δεν μπορούμε να παραλείψουμε την ύπαρξη τοίχων για αυτό τον λόγο χρησιμοποιούμε την ήδη υλοποιημένη συνάρτηση για την εύρεση αποστάσεων mazeDistance.

Υπολογίζουμε λοιπόν όλες τις αποστάσεις από την κατάσταση που βρισκόμαστε προς όλα τα φαγητά και επιστρέφουμε την μεγαλύτερη

(ο Pacman πρέπει να φάει όλα τα φαγητά).

Υπολογίζουμε με την βοήθεια της mazeDistance την απόσταση του κάθε φαγητού από την τρέχουσα θέση.

Αποθήκευουμε τις υπολογισμένες αποστάσεις σε περίπτωση που τις χρειαστούμε αργότερα για να μειώσουμε τον χρόνο.

Η ευρετική συνάρτηση είναι αποδεκτή αφού επιστρέφει ένα θετικό κάτω φράγμα στο κόστος και είναι και συνεπής διότι η εκτέλεση μίας ενέργειας c μειώνει την ευρετική μας το πολύ κατά c.

Q8.

Το πρόβλημα μας υποδεικνύει ότι ακόμα και μία πολύ καλά ορισμένη ευρετική συνάρτηση μπορεί να αποτύχει στην εύρεση του βέλτιστου μονοπατιού.

Συνεπώς για την υλοποίηση ενός πράκτορα που τρώει το πλησιέστερο φαγητό με απληστία χρησιμοποιούμε ένα από τους αλγόριθμους (BFS,UCS,A*) που υλοποιήσαμε προηγουμένως.

Ο αλγόριθμος χαρακτηρίζεται suboptimal αφού γίνεται χρήση άπληστης αναζήτησης.

Η υλοποίηση έγινε με τον A*.