Τεχνητή Νοημοσύνη
(Project1)

Ονοματεπώνυμο: Δημήτριος Σιταράς
Αριθμός μητρώου: 1115201800178
Εξάμηνο: 5ο

Πρόβλημα 1

search.py
Σε ολες τις συναρτήσεις αναζήτησης χρησιμοποιώ set για να αναπαστίσω το εξερευνημένο σύνολο. Η χρήση του set αντι για μιας list είναι θέμα αποδοτικότητας και χρόνου. Συγκεκριμένα ένα set υλοποιείται με hashtable που σημαίνει ότι η αναζήτηση, η εισαγωγή και η διαγραφή ενός στοιχείου μου κοστίζει O(1), σε σύγκριση με την λίστα η οποία για τις ίδιες “πραξεις” μου κοστίζει χρόνο ίσο με O(n).

(q1)
depthFirstSearch:
Στην αναζήτηση πρώτα κατά βάθος υλοποιώ το σύνορο χρησιμοποιώντας μια στοίβα (Stack). Ουσιαστικά, βασίζεται στον Graph-Search αλγοριθμο των διαφάνειων.
Στο συνορο “βαζω” τους απογόνους του κόμβου μόνο αν δεν υπάρχουν ήδη στο εξερευνημένο σύνολο.
Στην Stack κάνω push καθε φορα ενα tuple που περιέχει τον κόμβο και το μονοπάτι που μας οδηγει στον κόμβο αυτον.

(q2)
breadthFirstSearch:
Στην αναζήτηση πρώτα κατά πλάτος υλοποιώ το σύνορο χρησιμοποιώντας μια ουρά (Queue). Ουσιαστικά, βασίζεται στον Graph-Search αλγοριθμο των διαφάνειων (μαλιστα υπάρχει και ο αλγόριθμος για το BFS στις διαφάνειες).
Στο συνορο “βαζω” τους απογόνους του κόμβου μόνο αν δεν υπάρχουν ήδη στην ουρα (Queue) και στο εξερευνημένο σύνολο.
Στην Queue κάνω push καθε φορα ενα tuple που περιέχει τον κόμβο και το μονοπάτι που μας οδηγει στον κόμβο αυτον.

(q3)
uniformCostSearch:
Στην ομοιόμορφη αναζήτηση κόστους υλοποιώ το σύνορο χρησιμοποιώντας μια ουρά προτεραιότητας (Priority Queue). Ουσιαστικά, είναι ο αλγόριθμος Uniform-cost-Search όπως υπαρχει στις διαφάνειες (δηλαδη ο Graph-Search εμπλουτισμένος με μια ακομα συνθήκη ώστε ένας κομβος να επεκτείνεται καθε φορά με το μικροτερο δυνατο κόστος μονοπατιου).
Στην Priority Queue:

- κάνω push καθε φορα ενα list που περιέχει τον κόμβο και το μονοπάτι που μας οδηγει στον κόμβο αυτον και ως προτεραιότητα για το list αυτο, έχω το κόστος του συγκεκριμένου μονοπατιου.
- κάνω update το list που υπάρχει ήδη μέσα στην Priority Queue ανανεωνοντας το μονοπατι με ενα συντομότερο που μας οδηγει στον ίδιο κόμβο. Προφανώς, ανανεώνεται και η προτεραιότητα του list με το νέο μικροτερο κοστος του συγκεκριμένου μονοπατιου.

(q4)
aStarSearch:
Στην αναζήτηση A* υλοποιώ το σύνορο χρησιμοποιώντας μια ουρά προτεραιότητας (Priority Queue). Ουσιαστικά, βασίζεται στον Graph-Search αλγοριθμο των διαφανειών.
Στο συνορο “βαζω” τους απογόνους του κόμβου μόνο αν δεν υπάρχουν ήδη στο εξερευνημένο σύνολο.
Στην Priority Queue κάνω push καθε φορα ενα tuple που περιέχει τον κόμβο και το μονοπάτι που μας οδηγει στον κόμβο αυτον και ως προτεραιότητα για το tuple αυτο έχω το κόστος του συγκεκριμένου μονοπατιου συν την τιμή της ευρετικής συνάρτησης για τον συγκεκριμένο κόμβο.

searchAgents.py

(q5 & q6)
class CornersProblem:

- Στον constructor ορίζω ως startState ενα tuple που περιέχει την αρχική θέση του pacman όπως αυτή ορίζεται και ένα κενό tuple με το ονομα visited (αυτό θα περιέχει tuples με τις συντεταγμένες των γωνιών που θα επισκέπτεται ο pacman).

- Η συνάρτηση-μέλος getStartState() επιστρέφει το startState που όρισα παραπάνω.

- Η συνάρτηση-μέλος isGoalState() επιστρέφει True αν η τρεχων κατασταση ειναι κατάσταση στόχου δηλαδη αν ο pacman εχει επισκεφτεί και τις 4 γωνίες, διαφορετικα επιστρέφει False.

- Η συνάρτηση-μέλος getSuccessors() επιστρεφει σε μια λίστα με ένα tuple που περιέχει την διαδοχο κατασταση της τρεχουσας καταστασης του pacman ,την ενεργεια (πανω,κατω,δεξια,αριστερα) που απαιτειται για να φτάσει στην διαδοχο κατασταση καθώς και το κόστος της ενεργειας αυτης το οποίο είναι σταθερό και ίσο με 1. Επισής, εξετάζει αν η διαδοχος κατασταση ειναι μια απο τις γωνίες που δεν έχει επισκεφτεί ο pacman.

- Η συνάρτηση-μέλος cornersHeuristic() ειναι μια ευρετική συνάρτηση που επιστρέφει ενα νουμερο το οποιο αποτελει το κάτω φραγμα για το συντομότερο μονοπατι απο την αρχικη κατασταση στην κατασταση στοχου του προβλήματος. Συγκεκριμένα, η ευρετική που υλοποίησα επιστρέφει ένα άθροισμα με τις συντομότερες αποστάσεις μιας τρεχων καταστασης προς ολες τις γωνίες που δεν εχουν εξερευνηθει. Μετα απο κάθε υπολογισμό, η τρεχων κατασταση αλλαζει καθε φορα και γινεται η γωνια που “επισκέπτομαι”, (την πρωτη φορα η τρεχων κατασταση μπορει να ειναι οποιαδηποτε συντεταγμενη) συνεχίζοντας έτσι μεχρι να μην εχω αλλες γωνιες προς εξερεύνηση.
  Οι αποστάσεις υπολογίζονται με βάση της απόσταση Manhattan (δεν λαμβάνονται υπόψη οι τοίχοι). Η ευρετικη ειναι αποδεκτη διοτι επιστρέφει ένα θετικο κάτω φράγμα στο πραγματικο κοστος και ειναι συνεπής γιατι η εκτελεση μιας ενεργειας με σταθερο κόστος c έχει σαν αποτέλεσμα την μείωση της ευρετικης μου κατα το πολυ το c.

(q7)
class AStarFoodSearchAgent:

- Η συνάρτηση-μέλος foodHeuristic() ειναι μια ευρετική συνάρτηση η οποία υπολογίζει όλες τις αποστάσεις απο την τρέχων κατάσταση του pacman προς όλες τις τελείες (φαγητά) και επιστρέφει την μεγαλύτερη απόσταση.
  Οι αποστάσεις υπολογίζονται με βάση της απόσταση Λαβύρινθου (maze distance, στην οποια λαμβανονται υπόψη και οι τοίχοι). Η ευρετικη ειναι αποδεκτη διοτι επιστρέφει ένα θετικο κάτω φράγμα στο πραγματικο κοστος και ειναι συνεπής γιατι η εκτελεση μιας ενεργειας με σταθερο κόστος c έχει σαν αποτέλεσμα την μείωση της ευρετικης μου κατα το πολυ το c. Επίσης, επεκτείνει έναν καλό αριθμό κόμβων (4137) σε χρονο ~1.2 seconds. Μάλιστα, χρησιμοποιώ dictionary προκειμένου να αποθηκεύω τις αποστάσεις και να μην σπαταλώ χρονο υπολογίζοντας τις ξανα και ξανα.

(q8)
class ClosestDotSearchAgent:

- Η συνάρτηση-μέλος findPathToClosestDot() απλα επιστρεφει ενα μονοπάτι (μια λιστα με ενέργειες) προς την κοντινότερη τελεία χρησιμοποιώντας την αναζήτηση ομοιομορφου κοστους (UCS) την οποία έχω υλοποιήσει στο αρχείο search.py.

(q8)
class AnyFoodSearchProblem:

- Η συνάρτηση-μέλος isGoalState επιστρέφει True αν η τρέχων κατάσταση είναι κατάσταση στόχου, δηλαδή αν οι τρεχων συντεταγμένες του pacman είναι συντεταγμενες φαγητού (τελείας), διαφορετικά επιστρέφει False.