Pacman Project 1 : Search

Question 1(DFS):

Έχοντας το ψευδοκώδικα που περιγράφεται στο **Graph Search Pseudocode** το μόνο που έπρεπε να κάνω ήταν να ορίσω το node ως οντότητα και τι θα περιέχει και να βρω πώς θα γίνεται η επιστροφή του μονοπατιού όταν έχουμε συναντήσει κατάσταση στόχου. Το σύνορο(frontier) το φυλάω σε ένα Stack οπότε όταν γίνεται pop κάθε node ,εφαρμόζει την τεχνική αναζήτηση κατά βάθος. Αν ο κόμβος δεν υπάρχει στο set με τους κόμβους που έχουμε ήδη επισκεφτεί τότε τον προσθέτουμε στο set .Στη συνέχεια για κάθε παιδί του κόμβου εξετάζουμε αν υπάρχει στο set,αν όχι τότε το προσθέτουμε στη στοίβα .Το αρχικό node ορίζεται ως dictionary με τα κλειδιά 'place','cost' .Τα παιδιά ορίζονται επίσης έτσι με την μόνη διαφορά ότι έχουν δύο επιπλέον κλειδιά, το action'(δηλαδή την ενέργεια για να πάμε από το parent στο child) και το 'parent'.Όταν βρίσκει έναν κόμβο στόχου τότε ξεκινάμε μία αναζήτηση προς τα πίσω, όσο υπάρχει γονέας στο κόμβο κάνοντας insert το κόμβο στο πρώτο σημείο μίας λίστας path.Η επαναληπτική διαδικασία «while 'parent' in node» που πραγματοποιεί αυτό το πισωγύρισμα ουσιαστικά λέει «όσο ο κόμβος έχει κάποιο γονέα φύλαξε αυτό το κόμβο στο path και ξανακάνε το ερώτημα στο γονέα του» επομένως ξεκινάμε από τη κατάσταση στόχου στο γονέα του, επαναλαμβάνουμε τη διαδικασία και φτάνουμε στον αρχικό κόμβο που δεν έχει γονέα. Αν δεν βρει κατάσταση στόχου επιστρέφει κενή λίστα.

Question 2(BFS):

Ίδια λογική με την (1).Μοναδική διαφορά ότι το σύνορο είναι Queue αντί για Stack ,εφαρμόζοντας την τεχνική αναζήτησης κατά πλάτος.

Question 3(Astar):

Ίδια λογική με (1) και (2) ωστόσο εδώ έχουμε περισσότερες αλλαγές. Αρχικά το σύνορο κρατιέται ως PriorityQueue αφού η A* εφαρμόζει την αναζήτηση ουράς βάση προτεραιότητας. Σε κάθε παιδί του node βρίσκουμε το fn ως άθροισμα του κόστους που έχει να πάει μέχρι εκεί'cost to go'-*καινούριο κλειδί στο Child & node το οποίο είναι το 'cost to go' του γονέα + το κόστος του παιδιού*- + του heuristic του παιδιού μέχρι το στόχο. Το fn αν είναι μικρότερο του 'cost to go' του παιδιού τότε εκχωρείται σε αυτό και γίνεται προώθηση στη στοίβα με βάση αυτή τη τιμή οπότε το pop θα βγάλει μετά το κόμβο που πηγαίνει στο στόχο με το μικρότερο για τη δεδομένη στιγμή κόστος.

Question 4(Finding All the Corners):

Σε αυτό το ερώτημα έπρεπε να επεξεργαστώ τις παρακάτω συναρτήσεις της κλάσης **CornersProblem**

getStartState: Επιστρέφω ένα tuple με το StartingPosition και ένα κενό tuple που θα αναφέρεται στις γωνίες που έχουμε επισκεφτεί.

isGoalState: Επιστρέφει True αν το μήκος του tuple που έχουμε βάλει τις γωνίες που έχουμε επισκεφτεί είναι ίσο με τέσσερα, διαφορετικά False

expand:Για κάθε ενέργεια από τις πιθανές valid ενέργειες που μας δίνονται από την GetActions αποθηκεύουμε την επόμενη κατάσταση-με την getNextState- στη μεταβλητή new και το κόστος της ενέργειας-με την getActionCost- στη μεταβλητή Cost .Έπειτα προσθέτουμε στην λίστα children που ορίζεται πριν την επαναληπτική διαδικασία το tuple που περιέχει τη νέα κατάσταση, την ενέργεια και το κόστος της ενέργειας (new,action,Cost)

getNextState:Έχουμε μία Visited που είναι το δεύτερο κομμάτι του state που φυλάσσονται οι γωνίες που έχουμε επισκεφτεί αλλά ως λίστα-αρχικά το state[1] είναι tuple.Στη συνέχεια εξετάζω αν το (nextx,nexty) που έχει οριστεί πιο πάνω στη συνάρτηση ανήκει στο πίνακα με τις γωνίες και αν δεν είναι στο tuple με τις γωνίες που ανέφερα προηγουμένως. Εφόσον ισχύουν αυτά τότε προσθέτω στο Visited τη γωνία που μόλις έχω βρει. Έπειτα εξετάζω αν το (nextx,nexty)δεν ανήκει στους τοίχους(αν ανήκει τότε δεν μπορώ να επιστρέψω τις συντεταγμένες αφού είναι μη έγκαιρο state, αν όχι τότε στη TVisited μεταφέρω τη Visited ως tuple και επιστρέφω το((nextx,nexty),TVisited) που θα είναι η νέα κατάσταση.

Question 5(Corners Problem Heuristic):

Σε αυτό το ερώτημα φτιάχνω μία συνάρτηση heuristic για το Corners Problem .Για κάθε corner ελέγχω αν υπάρχει στην state[1], δημιουργώντας ένα πίνακα με τις γωνίες που δεν έχω επισκεφτεί .Για κάθε γωνία από το unvisited_corners βρίσκω τη τιμή από το κόμβο που είμαι μέχρι την γωνία. Παράλληλα στην Min έχοντας αρχικά έναν μεγάλο αριθμό ελέγχω αν η manhattanDistance με τη συγκεκριμένη γωνία είναι μικρότερη,αν ναι τότε κρατάω ποια γωνία είναι και θέτω ως Min την απόσταση. Έπειτα το pacman πάει στη μικρότερη γωνία και την αφαιρώ από το unvisited_corners.Αυτή η διαδικασία γίνεται επαναληπτικά μέχρι να μην υπάρχουν γωνίες που δεν έχουμε επισκεφτεί.Η συνάρτηση είναι consistent γιατί κάθε απόγονος ρίχνει το heuristic το πολύ κατά πολύ c ,όπου c το κόστος της ενέργειας.

H(n)<=C(n,action,n')+h(n')

Αφού είναι consistent είναι και admissible.

Question 6(Eating all dots):

Σε αυτό το ερώτημα αρχικά δημιουργώ μία λίστα με τις συντεταγμένες των dots ,έπειτα για κάθε dot υπολογίζω την manhattanDistance όπως και στο (5) και την αποθηκεύω σε έναν αρχικά μηδενικό πίνακα. Τέλος καλώ την max για να βρω την μεγαλύτερη τιμή-δηλαδή το πιο μακρινό dot- και την επιστρέφω.

Question 7(Suboptimal Search):

Σε αυτό το ερώτημα έπρεπε να επεξεργαστώ τη συνάρτηση findPathToClosestDot της AStarFoodSearchAgent και την isGoalState στην κλάση AnyFoodSearchProblem.

findPathToClosestDot:Για να βρω το κοντινότερο dot αρκεί να χρησιμοποιήσω μία αρκετά αποδοτική συνάρτηση αναζήτησης ,και μιας και δεν με ενδιαφέρει το κόστος αλλά να είναι η κοντινότερη, η BFS είναι η κατάλληλη συνάρτηση .Κάνω import την bfs από το αρχείο search και την καλώ με όρισμα το problem.

isGoalState: στη συνάρτηση currFood εκχωρώ την self.food η οποία περιέχει τις συντεταγμένες x,y των κόμβων που έχουν food ,επομένως αρκεί να επιστρέφω το currFood[x][y] το οποίο αν τα x,y δεν έχουν food θα στέλνει False,διαφορετικά True.