Νικόλαος Ηλιόπουλος 1115201800332

Γενικά ο κώδικας είναι γεμάτος με σχόλια.

Αν δεν αναφέρω κάτι εδώ σίγουρα θα υπάρχει με σχόλια στον κώδικα μου.

QUESTION 1

Τον αλγόριθμο DFS τον υλοποίησα με stack αφού θέλουμε να πάμε πρώτα σε βάθος, διότι η λογική του stack είναι LIFO (LAST-IN FIRST-OUT).

Χρησιμοποιώ από το util.py την έτοιμη stack και βάζω το αρχικο state του προβλήματος μέσα. Επίσης φτιάχνω και δύο λίστες μία για να κρατάω του κόμβους που έχω επισκευτείκ και μία για να κρατάω το path.

Η λογική στην while είναι ότι βγάζω το πρώτο στοιχείο από το stack δηλαδή αυτό που μπήκε τελευταίο, τσεκάρω να δω αν φτάσαμε στον στόχο, αποθηκεύω ότι επισκέφτηκα αυτόν τον κόμβο, βρίσκω τους επόμενους κόμβους από αυτόν τον κόμβο και αν δεν τους έχω ή\δη επισκεφτεί τους βάζω στο stack.

QUESTION 2

Η λογική εδώ είναι ίδια με την DFS αλλά ο BFS είναι πρώτα σε πλάτος δηλαδή FIFO (FIRST-IN FIRST-OUT). FIFO δηλαδή υλοποιείτε με queue. Ο κώδικας είναι ίδιος με του DFS απλά αλλάζει η δομή.

QUESTION 3

Ο BFS στην ουσία είναι UCS με g(n) = depth(n).

Δηλαδή ο BFS ψάχνει πρώτα τους κόμβους με το μικρότερο depth.

Άρα πάλι έχουμε queue αλλά με priority το κόστος και όχι το depth.
Ο κώδικας είναι πάλι ο ίδιος απλά αλλάζει η δομή.

Και έχει την μικρή ιδιαιτερότητα ότι το κόστος ανάλογα πότε θα συναντήσουμε τον κόμβο και από ποιόν θα ερχόμαστε μπορεί να διαφέρει ενώ το depth (πάντα παραμένει το ίδιο) άρα προσθέτουμε μία if που ελέγχει να δεί στους κόμβους που έχουμε μέσα αν το νέο

κόστος που βρήκαμε για αυτούς αν είναι μικρότερο για να το αλλάξουμε αλλιώς το αφήνουμε ως έχει.

QUESTION 4

Ο A* είναι και αυτός παρόμοιος με τον UCS αλλά είναι ποιο ‘’έξυπνος’’.

Ο A* χρησιμοποιεί μία συνάρτηση για να υπολογίσει καλύτερα το priority. Με τον τύπο f(n) = g(n) + h(n).

f(n) το priority που θα χρησιμοποιήσουμε.

g(n) = το κόστος που είχαμε πριν

Η h(n) λέγεται heuristic (ευρευτική) συνάρτηση. Και μπορεί να είναι οποιαδήποτε συνάρτηση μπορεί να μας βοηθήσει να κάνουμε το Priority Ποιο κοντά στην πραγματικότητα.

Άρα πάλι έχουμε queue με priority το κόστος + το αποτέλεσμα της ευρετικής.
Ο κώδικας είναι παρόμοιος. Αλλά δεν ασχολιόμαστε με κανένα κόμβο που έχουμε ξαναεπισκευτεί γιατί σημαίνει ότι για να τον έχουμε επισκευτεί σημαίνει ότι ποιο πριν ήμασταν στο ίδιο σημείο με μικρότερο συνολικό κόστος. Άρα άσκοπα θα προσωρήσουμε. Οπότε πάμε στην επόμενη επανάληψη.

QUESTION 5

Στο initialize φτιάχνουμε μία λίστα με tuples για να κρατήσουμε τις μια μη visited γωνίες.

Στο getStartState απλά γυρνάμε το την αρχική θέση μαζί με τις γωνίες που δεν έχουμε επισκεφτεί.

Για να έχουμε φτάσει στον στόχο μας σε ένα state θα πρέπει αυτό το state να μην έχει δίπλα του στο tuple γωνίες(σημαίνει ότι τις επισκέφτηκε όλες)

Και για να πάρουμε τους succesors θα κάνουμε έναν έλεγχο για κάθε πιθανή επιτρεπόμενη κίνηση και θα της επιστρεψουμε και βέβαια θα κοιτάξουμε αν κάποιες από αυτές τις κινήσεις καταλήγει σε γωνία για να την αφαιρέσουμε.

QUESTION 6

Επιλέγω σαν heuristic την manhattanDistance στο util.py.
Αν πάω στην ποιο κοντινή μου γωνία μετά απλά μπορώ να ακολουθήσω την μικρότερη πλευρά για να πάω στην άλλη γωνία και μετά απλά ακολουθώ τις πλευρές αν δεν υπήρχαν οι τοίχοι να με εμποδίσουν(αυτή η λογική είναι η καλύτερη περίπτωση).

QUESTION 7

Χρησιμοποιώ την mazeDistance στο searchAgents.py.

Θα βρούμε απλά την απόσταση μας από κάθε φαγητό και θα επιστρέψουμε το μέγιστο αυτών τον αποστάσεων. Οπότε κάθε φορά αποθηκεύονται οι κόμβοι στην ουρά με Priority το μεγαλύτερο που είναι δυνατόν. Η ουρά θα βγάλει τον κόμβο με το μικρότερο αυτών τον μεγαλύτερων άρα κάθε φορά κάνουμε το καλύτερο από τις χειρότερες επιλογές.

QUESTION 8

Απλά καλούμε τον astar(με το πρόβλημα) για να βρεί το μονοπάτι για το ποιο κοντινό φαγητό. Δεν έχει σημασία ποια συνάρτηση θα καλέσουμε A* ή UCS ή BFS.