PACMAN PROJECT 1

Βάιος Λύτρας

Q1: Για τον DFS χρησιμοποίησα Stack και όποια γραμμή κώδικα είναι λίγο πιο δυσνόητη έχω βάλει τα κατάλληλα σχόλια.Για να βρω το μονοπάτι στο τέλος έχω ένα λεξικό που ονομάζεται father{} το οποίο κρατάει από που προήλθε κάποιος κόμβος. Με backtrack σε αυτό το λεξικό βρίσκεται έυκολα το τελικό μονοπάτι

Q2: Ο BFS έχει σχεδόν ίδιο κώδικα με τον DFS απλά αλλάζει η δομή δεδομένων απο stack σε queue (ο DFS έκανε το push σε stack και τώρα ο BFS το κάνει σε queue)

Q3: Ο UCS πάλι μοιάζει με τον BFS απλά έχει κάποια κομμάτια κώδικα παραπάνω τα οποία τσεκάρουν κάθε φορά που βρίσκουμε έναν κόμβο αν υπάρχει ήδη μέσα στο priorityQueue με μεγαλύτερο priority τότε το κάνει update και επίσης κάνει update τον κόμβο στο father{} επείδη βρήκαμε ένα μικρότερου κόστους μονοπάτι που καταλήγει σε αυτόν τον κόμβο.

Q4: Η A* μέχρι ένα σημείο είναι ίδια με τις παραπάνω.Όταν η A* παίρνει τους succesors ενός κόμβου πρώτα ελέγχει αν κάποιος υπάρχει ήδη στο PQ με χαμηλότερο κόστος και όταν λέω κόστος δεν εννοώ το priority γιατί το priority είναι priority = κόστος + heuristic(κόμβου) αν τον βρει με ψηλότερο κόστος μονοπατιού τότε τον κάνει Update. Αυτό το έχω κάνει για να δουλεύει η συνάρτηση και σε περιπτώσεις που η ευριστική επιστρέφει 0 σε όλους τους κόμβους.Δηλαδή αν η ευριστική κάνει 0 σε όλους τους κόμβους τότε η A* είναι σαν την UCS αφού priority = συνεχόμενο_κόστος + ευριστική(κόμβου) (=0).

Q5: Για να λύσω το CornersProblem χρησιμοποίησα την αυθαίρετη state η οποία είναι η τρέχων θέση συνοδευόμενη απο ένα tuple από tuples Που έχει μέσα ποιες γωνίες έχω ήδη επισκεφτεί.Άρα όταν αυτό το tuple έχει μέγεθος 4 (όσες οι γωνίες) σημαίνει ότι φτάσαμε στο goal state. Κάθε φορά που στην συνάρτηση getSuccesors μία από τις επόμενες θέσεις είναι γωνία τότε επιστρέφουμε την επόμενη θέση και την λίστα με τις γωνίες που έχουμε επισκεφθεί ανανεωμένη με την καινούρια γωνία μέσα.Έτσι όταν ο BFS θα πάρει αυτο το state να το εξετάζει θα είναι σαν να εξετάζει το πρόβλημα από την αρχή με αρχική θέση την τελευταία γωνία που επισκεφθήκαμε και τελική την απόμενη γωνία (περίπου).

Q6: Η ευριστική δουλεύει ως εξής: Έχουμε μία λίστα με όλες τις γωνίες που δεν έχουμε επισκεφθεί ακόμα (η οποία έχει προκύψει απο το state[1] δηλαδή το tuple of tuples που συνοδεύει κάθε κόμβο με τις γωνίες που έχουμε επισκεφθεί). Μετά παίρνει την απόσταση manhattan για κάθε μία από αυτές και τις προσθέτει όλες μαζί αυτή είναι η και η τιμή που επιστρέφει η συνάρτηση. Προφανώς είναι παραδεκτή γιατι ποτέ δεν υπερεκτιμά το κόστος (λόγω manhattanDistance) και είναι και συνεπής.

Q7: Η ευριστική αυτή επιστρέφει απλά την ΠΡΑΓΜΑΤΙΚΗ απόσταση του pacman απο το πιο μακρινό φαγητό.Είναι μία πολυ καλή ευρετική και πολύ απλή απλά το μόνο κακό είναι ότι είανι λίγο χρονοβόρα (για το trickySearch κάνει ~17 δευτερόλεπτα). Είναι χρονοβόρα γιατί χρησιμοποιεί την βοηθητική συνάρτηση mazeDistance() η οποία επιστρέφει την πραγματικη απόσταση μεταξύ 2 κόμβων χρησιμοποιώντας συναρτήσεις που έχω ήδη υλοποιήσει (πχ. BFS).Η πρώτη ευρετική που σκέφτηκα ήταν να πάρω την απόσταση manhattan του πιο μακρινού φαγητού αλλά αυτή έκανε expand λίγο πάνω από 9400 κόμβους και η ερώτηση έπαιρνε 3/4. Με την πραγματική απόσταση κάνω expand κάτω απο 4500 κόμβους (!!!) σε κόστος 13 δευτερολέπτων παραπάνω (με την manhttan έβρισκε τη λυση σε ~7 δευτερόλεπτα)

Q8: Αυτή η ερώτηση ήταν πολύ απλή, έγραψα την isGoalState() να επιστρέφει True όταν θέση είναι φαγητό και χρησιμοποίησα την ήδη υλοποιημένη BFS μέσα στην findPathToClosestDot(). Ο συνδυασμός αυτών των 2 βρίσκει το κοντινότερο φαγητό μέχρι να τερματίσει το πρόγραμμα (για να τελειώσει το πρόγραμμα όταν ο pacman τρώει όλα τα φάγητα είναι ήδη υλοποιημένο μέσα στην συνάρτηση def registerInitialState()οποία τρέχει την findPathToClosestDot() τόσες φορές όσες και τα φαγητά που έχει ο χάρτης