***ΥΣ02 Τεχνητή Νοημοσύνη – Χειμερινό Εξάμηνο 2019-2020***

***Εργασία Πρώτη***

***Berkeley Project 1: Search***

***Γιάννης Δαλιάνης***

***1115201700027***

Question 1: Η συνάρτηση depthFirstSearch καλεί την DFS\_BFS δίνοντάς της μία στοίβα ως δομή υλοποίησης του DFS. Στην αρχή στη στοίβα τοποθετώ τον πρώτο κόμβο του προβλήματος, μαζί με μία κενή λίστα η οποία στο τέλος θα έχει όλες τις κινήσεις και ένα τυχαίο κόστος, έστω 1. Για κάθε κόμβο που δεν έχει επεκταθεί, δηλαδή δε τον έχουμε ήδη επισκεφτεί, βάζουμε όλους τους κληρονόμους του στη στοίβα, εφόσον και αυτοί δεν έχουν ήδη επεκταθεί. Όταν βάζουμε έναν κληρονόμο στη στοίβα, του δίνουμε το σύνολο κινήσεων που πραγματοποιούνται μέχρι να φτάσουμε στον πατέρα του συν την κίνηση από τον πατέρα μέχρι τον συγκεκριμένο κληρονόμο. Αυτή η διαδικασία επαναλαμβάνεται για κάθε κληρονόμο κάθε κόμβου μέχρι να βρούμε τον κόμβο-στόχο και να γυρίσουμε το σύνολο των κινήσεων.

Question 2: Η συνάρτηση breadthFirstSearch καλεί την DFS\_BFS δίνοντάς της μία ουρά ως δομή υλοποίησης του BFS. Η λογική είναι ίδια με την υλοποίηση του DFS, απλά χρησιμοποιείται ουρά αντί για στοίβα.

Question 3: Η συνάρτηση unifrormCostSearh καλεί την Implementation\_withPQ, η οποία χρησιμοποιεί σα δομή υλοποίησης μια ουρά προτεραιότητας που επιστρέφει το αντικείμενο με την ελάχιστη κάθε φορά προτεραιότητα-κόστος (ουρά προτεραιότητας ελαχίστου). Το κόστος του αρχικού κόμβου είναι 0. Η λογική εισαγωγής των κληρονόμων στους επισκεπτόμενους είναι ίδια με την αντίστοιχη λογική της DFS και της BFS, με τη διαφορά ότι κάθε κόμβος-κληρονόμος που δεν έχουμε ήδη επισκεφτεί και εισάγεται στην ουρά, πέρα από ένα σύνολο κινήσεων κρατάει και ένα κόστος, το κόστος μέχρι να φτάσουμε στον πατέρα του συν το κόστος από τον πατέρα μέχρι αυτόν. Η μεταβλητή new\_h χρησιμοποιείται στο ερώτημα 4 και για τον UCS ισούται με 0.

Question 4: Ακολουθώντας την περιγραφή του αλγορίθμου όπως δίνεται στο σύνδεσμο <https://www.geeksforgeeks.org/a-search-algorithm/>, το closed list είναι το dictionary των visited και το open\_list είναι η δομή Priority Queue. Η συνάρτηση aStarSearch καλεί τη συνάρτηση Implementation\_withPQ με ορίσματα το problem και τη heuristic που χρησιμοποιεί η A\*. Ουσιαστικά η υλοποίηση της A\* είναι ίδια με την υλοποίηση της UCS, με τη διαφορά ότι το κόστος-προτεραιότητα στους διάφορους κόμβους της Priority Queue είναι το άθροισμα της απόστασής τους από την αρχή συν το αποτέλεσμα της heuristic.

Question 5: Χρησιμοποιώ ένα tuple corners\_to\_pop το οποίο περιέχει τις θέσεις όλων των γωνιών του χάρτη. Όσο υπάρχουν γωνίες μέσα σε αυτό το tuple, συνεχίζεται η αναζήτηση. Από το tuple αφαιρείται μια γωνία όταν τη βρίσκω. Συνεπώς το Goal State είναι η περίπτωση όπου το corners\_to\_pop είναι άδειο. Στη getSuccessors χρησιμοποιώ μια λίστα have\_not\_visited η οποία περιέχει τις γωνίες που δεν έχουμε ακόμα βρει. Αν σε έναν από τους κληρονόμους του τρέχοντος state βρεθεί μια γωνία την οποία δεν έχω ξαναεπισκεφτεί, αυτή αφαιρείται από το have\_not\_visited. Σε κάθε κληρονόμο που επιστρέφεται από τη συνάρτηση δίνεται και το have\_not\_visited. Έχουν προστεθεί οι κατάλληλες εντολές για να φαίνεται στο λαβύρινθο το χρώμα των κόμβων που επισκέπτεται το Pacman.

Question 6: Στην cornersHeuristic, η αρχικοποίηση του corners και του walls, είναι αχρείαστη στη συγκεκριμένη υλοποίηση. Η συνάρτηση επιστρέφει 0 όταν από το state[1], δηλαδή από το corners\_to\_pop έχουν αφαιρεθεί όλες οι γωνίες. Σε αντίθετη περίπτωση με χρήση και της manhattan\_general, επιστρέφει το κόστος προς την πιο μακρινή γωνία με τη τεχνική της manhattan heuristic.

Question 7: Η foodHeuristic επιστρέφει χρησιμοποιώντας την έτοιμη συνάρτηση mazeDistance, την μέγιστη τιμή της mazeDistane από το τρέχον position προς κάθε σημείο του grid. Στη mazeDistance δίνω ως ορίσματα την τρέχουσα θέση, ένα σημείο του χάρτη και την αρχική κατάσταση του προβλήματος. Με αυτό τον τρόπο γίνονται expand μόλις 4137 κόμβοι, ωστόσο παίρνει αρκετό χρόνο. Στο σταθερό μου υπολογιστή το path για να βρεθεί χρειάζονται 17 με 18 δευτερόλεπτα