# Τεχνητή Νοημοσύνη 1 - Χειμερινό 2020-2021

## Εργασια πρώτη - README

Ροβιθάκης Ιωάννης | sdi1800164

**Q1 - Q4:**

Κατά την υλοποίηση των ερωτημάτων 1 με 4, προσπάθησα να γράψω κάτι κοντα στον ψευδοκώδικα που μας δώθηκε.

1) Στον **BFS** χρησιμοποίησα **stack**, καθώς ο BFS χρειάζεται LIFO λειτουργικότητα με βάση την λογική και την θεωρία.

Είχα τον έλεγχο στόχου μέσα στο εσωτερικό for loop όπως στον ψευδοκώδικα, αλλα αναγκάστηκα να τον μετακινήσω έξω για να το δεχτεί ο autograder.

2) Στον **DFS** χρησιμοποίησα **queue**, καθώς απαιτείται FIFO λειτουργικότητα

Εδώ προσθεσα ένα ακόμα set, το frontierBuf, το οποίο αποτρέπει κόμβους που βρίσκονται μέσα στο σύνορο, να μπούν δεύτερη φορά σε αυτό.

**3)** Στον **UCS** χρησιμοποίησα **priority queue**, ώστε να μπορώ να βγάζω από το σύνορο τους κόβους με αύξουσα σειρά προτεραιότητας, και κατα συνέπεια να επεκτείνω πρώτα τις πιο οικονομικές διαδρομες.

Εδώ αντικατέστησα τον τρόπο που μέχρι τώρα υπολόγιζα το path προς κάθε κόμβο (οι καταστάσεις περιείχαν μια λίστα με τις κινήσεις για να φτάσεις σε αυτές από την αρχική), με ένα dictionary το οποίο δεδομένου ενός κόμβου, επιστρέφει τον γονέα του για την βέλτιστη διαδρομή, την κίνηση που έκανε ο γονέας για να φτάστει σε αυτόν, καθώς και το κόστος της κίνησης αυτής. Το παραπάνω μου επιτρέπει πολύ εύκολα ( σε O(1)) να ελέγχω άν ένας κόμβος έχει ήδη εξερευνηθεί, και να αλλάζω εύκολα τον πατέρα του, και συνεπώς και την βέλτιστη διαδρομή)

4) Ο **A*** είναι πολύ παρόμοιος με τον UCS (Για ίδιο λόγο χρησιμοποιώ priority queue), αλλα πλέον το priority στο queue καθορίζεται και με βάση μια ευρετική συνάρτηση, η οποία δίνει στον αλγόριθμο επιπλέον δεδομένα για τις πιθανες εκβάσεις των κινήσεων του και κατα συνέπεια βοηθάει να λαμβάνονται πιο γρήγορα "σωστες" αποφάσεις.

Σε όποια περίπτωση είχα δομή στην οποία έλεγχα τακτικά την ύπαρξη συγκεκριμένων στοιχείων, χρησιμοποίησα sets ώστε να έχω πολυπλοκότητα O(1). (πχ το explored set )

**Q5:**

**getStartState():** Έθεσα το start state ως ένα tuple που περιέχει το position και ένα tuple το οποίο περιέχει τις γωνίες τις οποίες έχει επισκεφθεί μέχρι το state αυτό ο πακμαν. Αρα το αρχικό state, έχει την αρχική τοποθεσία και κενό το tuple καθώς δεν έχει επισκεφθεί καμία γωνια ακόμα. Θεωρώ ότι η αρχική τοποθεσία δεν είναι μια απο τις γωνίες. (Δεν είναι δυσκολο να προσθέσεις έναν επιπλέον έλεγχο και αν αρχίζει από γωνία, να ξεκινάς με το tuple να έχει τη γωνία αυτη). Χρησιμοποίησα tuple, καθώς το που επιστρέφει η συνάρτηση, χρησιμοποιείται-εισάγεται στο set explored στην BFS και κατα συνέπεια πρεπει το state να είναι hashable.

**isGoalState():** Αν η λίστα με τις γωνίες φτάσει μήκος 4, δηλαδη ο πακμαν επισκεφθηκε όλες τις γωνίες, επιτυχια.

**getSuccessors():** Οι successors απλά είναι οι κόμβοι γύρω από τον πάκμαν που δεν έχει επισκεφθεί (Αν επισκεφθεί μια γωνία στην πράξη αλλαζει το 2ο μέρος του tuple, οπότε έχουμε "νέα" states, και ο BFS μπορεί να συνεχίσει την αναζήτηση) και δέν είναι τοιχοι. Αν ενας από τους successors είναι γωνια, τον προσθέτουμε στο tuple του state ως σημάδι ότι περνώντας από το state αυτό, επισκεφτόμαστε την γωνία. Σημαντικό είναι ότι δεν προσθέτουμε την ίδια γωνία παραπάνω από μια φορά στο tuple, γεγονός που μας επιτρέπει να επιβεβαιώσουμε το goal state με χρήση της len().

**Q6:**

Διαδικασία σκέψης:

Το πρόβλημα θέλει να βρούμε μία ευρετική η οποία να κατευθύνει τον πακμαν σε μια διαδρομή που να περνάει από όλες τις γωνίες. Συνεπώς πρέπει να βρούμε έναν τρόπο να συνδέσουμε τις γωνίες με το πρόβλημα αλλά και μεταξύ τους, για να πετύχουμε το επιθυμητό αποτέλεσμα.

Πριν ξεκινήσω την εργασία, χάλασα λίγη ώρα να διαβάζω τα αρχεία τα οποία πρότεινε η εκφώνηση ως "χρήσιμα". Στο στάδιο αυτό σημείωσα τις συναρτήσεις στις οποίες είχα πρόσβαση, που πίστεψα ότι μπορεί να φανουν χρήσιμες αργότερα. Συνεπώς, ξεκινώντας να προσεγγίσω το πρόβλημα της ευρετικής, είχα στον νου μου τις συναρτήσεις euclidean, manhattan και maze distance ως εν δυνάμη εργαλεία (Την euclidean την έγραψα εγώ αλλα τελικά την άφησα λόγω απογοητευτικών αποτελεσμάτων, λογικό καθώς η manhattan καιη maze αναπαριστούν καλυτερα τον τρόπο με τον οποίο κινείται ο πακμαν στον λαβύρινθο) (fun fact: Με μια αναζήτηση στα αρχεία του project 1, της συναρτήσης maze distance, είδα ότι δεν χρησιμοποιείται πουθενα, οπότε το πήρα σαν hint ότι είμαι σε σωστο δρόμο)

Ξεκίνησα δοκιμάζοντας τυχαίες ιδέες για να δώ την επίδραση τους στο πρόβλημα και να πάρω ιδέες. (π.χ. Απόσταση από κοντινότερη ή/και μακρύτερη γωνία - Δεν συνέδεε με κάποιο τρόπο τις γωνίες μεταξύ τους

οπότε δεν ήταν αποτελεσματικό, Μέση απόσταση από τις υπολειπόμενες γωνίες - συνέδεε τις γωνίες αλλά όχι με ιδιαίτερα εξυπνο/αποτελεσματικο τροπο)

Προσπαθώντας παράλληλα να σκεφτώ έναν τρόπο να κάνω την ευρετική συνεπή και συνεπώς και παραδεκτή (Βασιζόμουν σε αποστάσεις για να πετύχω μη-υπερεκτίμηση, αλλα όχι με τον σωστο τροπο), προσπάθησα να κάνω την maze distance να υπολογίζει το κόστος για να περάσει ο πακμαν από όλους τους στοχους.

Η ιδέα στην οποία τελικά κατέληξα και χρησιμοποίησα είναι η εξής:

i) Βρισκω την απόσταση του πάκμαν από την κοντινότερη του γωνία για ένα state.

ii) Αλλάζω το node από το οποίο υπολογίζω την απόσταση και πλέον υπολογίζω την απόσταση από την γωνία που βρήκα στην κοντινότερη της γωνία.

iii) Αλλάζω πάλι γωνία και πάει λέγοντας, αθροίζοντας τις αποστάσεις μέχρις ότου να καλυφθουν όλες οι γωνίες = διαδρομή που τις συνδέει όλες

Δοκίμασα την παραπάνω ιδέα με τις euclidean, manhattan και maze distances, και κατέληξα ότι η manhattan είναι η καλύτερη και ταχύτερη λύση καθώς ο υπολογισμός της είναι οικονομικός και ταυτόχρονα δεν μπορεί να υπερεκτιμήσει στην περίπτωση αυτή, καθώς δεδομένου του χώρου του παιχνιδιου και του τρόπου κίνησης του πακμαν, η τιμή που υπολογίζει αντιστοιχεί στην βέλτιστη δυνατή απόσταση αν δεν υπήρχαν τοίχοι (που υπάρχουν, άρα στην καλύτερη περίπτωση το πετυχαίνει ακριβώς, όχι παραπάνω)

**Q7:**

Η πρώτη μου ιδέα ήταν ότι πρέπει να βρώ κάποιον τρόπο να προσδιορίζω αν ενα state είναι καλό ή όχι σε μια δεδομένη στιγμη, με βάση τους γειτονές του, και αν αυτοί ήταν φαγητό ή τοίχοι ( Να βρώ ένα μέτρο πυκνότητας φαγητού γύρω από κάθε state). Το παραπάνω όμως ήταν υπερβολικά τοπική λύση για να είναι αποτελεσματική. Προσπάθησα να υπολογίσω πυκνότητα φαγητού σε όλο τον χάρτη αλλα βρέθηκα σε αδιέξοδο λόγω προβλημάτων δικής μου υλοποίησης (+ οι περιοχές με πολύ φαγητο δεν είναι απαραίτητα στη βελτιστη διαδρομή ). Η γενική λογική που είχα στο μυαλό μου απο την αρχή ήταν ότι όσο πιο κοντά βρίσκεται ανα πάσα στιγμή ο πακμαν σε όλο το φαγητό, τόσο πιο γρήγορα μπορεί να το καταναλώσει. Δοκίμασα να υπολογίσω την απόσταση κάθε φαγητού από τον πάκμαν, αλλα αν και στην θεωρία ήταν καλή ιδέα, υπερεκτιμούσε και ταυτόχρονα ήταν υπερβολικά αργο. Τελικά σκέφτηκα να δοκιμάσω να υπολογίσω την απόσταση από το μακρύτερο από τον πάκμαν φαγητο την οποία μέθοδο και χρησιμοποίησα. Η μέθοδος δεν υπερεκτιμά, καθώς είναι απλά η απόσταση από ένα φαγητό, συνεπώς ακόμα και μόνο αυτό να έχει μείνει τελευταίο, δεν θα γίνει υπερεκτίμηση. Στην περίπτωση αυτή, χρησιμοποίησα τον maze distance, καθώς δίνει την ακριβή απόσταση μέσα στον λαβύριθνο, γεγονός που έκανε την ευρετική αυτή συνεπή (Η manhattan δεν λαμβάνει υπ όψην τους τοιχους οπότε υπήρχε τελικά πρόβλημα συνέπειας )

Το μόνο πρόβλημα της παραπάνω υλοποίησης ήταν η απόδοσή της, καθώς έπαιρνε 20 sec για να εκτελεστει ο autograder. Το πρόβλημα αυτό το έλυσα υπολογίζοντας μία φορα, την πρώτη φορά που καλείται η συνάρτηση, ένα dictionary το οποίο περιέχει την maze distance από κάθε κόμβο σε κάθε άλλο (εκτώς απο κόμβους τοιχους). Συνεπώς στην εκτέλεση πλέον απλά χρησιμοποιείται η έτοιμε πληροφορία με αποτέλεσμα ικανοποιητικους χρόνους ~3-4 δευτερόλεπτα. (Σημείωση, καλό είναι μετά την λύση του προβλήματος το dictionary να αδειάζει ωστε να μπορεί να χρησιμοποιηθεί και σε άλλο map )

**Q8:**

**findPathToClosestDot()**: Έχουμε διαθέσιμο το problem και επιθυμούμε την συντομότερη διαδρομή για οποιοδήποτε φαγητο (=την διαδρομή για το κοντινότερο φαγητό). Συνεπώς, μπορουμε να χρησιμοποιήσουμε οποιονδήποτε από τους αλγορίθμους Q2-Q4 καθώς όλοι τους επιστρέφουν την βέλτιστη απάντηση. Αφού οι κινήσεις έχουν μοναδιαίο κόστος, ο A* και ο UCS δεν έχουν ιδιαιτερο νόημα, οπότε αποφάσισα απλά να καλέσω τον BFS και να επιστρέψω την διαδρομή που μου δίνει. ( Και οι αλλοι 2 δουλευουν εξίσου καλα)

**isGoalState():** Θέλουμε να βρούμε διαδρομή για οποιοδήποτε φαγητό, συνεπώς όλα τα φαγητά αποτελούν goal states. Αρκεί να ελέγξουμε αν ένα state έχει φαγητό η όχι.

**Υ.Γ.** : Ρίξτε μια ματιά και στα comments στα αρχεία πηγαίου κώδικα. Για οποιαδήποτε απορία/παρατηρηση/διευκρίνηση στείλτε μου email στο sdi1800164@di.uoa.gr