# Τεχνητή Νοημοσύνη: Εργασία 1 (Pacman Project)
# Μυλωνόπουλος Δημήτριος – Α.Μ 1115201600112

Στην εργασία που παραδόθηκε, υλοποιήθηκαν όλα τα ζητούμενα ερωτήματα χωρίς να εντοπισθεί κάποιο πρόβλημα. Παρακάτω θα αναλυθούν οι υλοποιήσεις στα ερωτήματα Q1-Q8 του Pacman Project του Berkley. Όλες οι υλοποιήσεις των αλγορίθμων κινήθηκαν με βάση τους αλγορίθμους που διδαχτήκαμε στο μάθημα. (Γι' αυτό και τα q1, q2, q5 βγάζουν 0/3 στον autograder.py του Project Pacman. Θα εξηγήσουμε τους λόγους παρακάτω.).

**Question 1: Findind a Fixed Food Dot using Depth First Search**

Η **depthFirstSearch(problem)** υλοποιήθηκε με βάση τον αλγόριθμο Graph-Search που μπορούμε να βρούμε στις διαφάνειες του μαθήματος. Για τη δομή του frontier χρησιμοποίηθηκε η δομή της στοίβας που υπήρχε έτοιμη στο util.py ως util.Stack. Για τη λίστα τον κόμβων που έχουμε εξερευνήσει χρησιμοποιήθηκε η λίστα explored, ενώ για να αποθηκευθούν οι γονείς τον κόμβων χρησιμοποιήθηκε ένα λεξικό με το όνομα parents. Ως κλειδί παίρνει ένα state και αποθηκευονται σε μια λίστα: [parent, action, Boolean] με τον Boolean να εκφράζει αν ο κόμβος κλειδί είναι στο Explored ή στο Frontier.

Ο τρόπος με τον οποίο ο αλγόριθμος μας επιστρέφει τη σειρά των βημάτων απ το starting state στο goalstate γίνεται μέσω backtracing με τη βοήθεια του λεξικού που δημιουργήθηκε. Δηλαδή μόλις βρούμε την κατάσταση στόχο βρίσκων των πατέρα της και τον πατέρα του πατέρα της κ.ο.κ μέχρι να φτάσω στο starting state μου. Ο αλγόριθμος ακολουθεί τη δομή το Graph-Search με μοναδική προσθήκη η κατασκευή του λεξικού για το path. Ο λόγος που ο συγκεκριμένος αλγόριθμος βγάζει fail στον autograder.py είναι επειδή ο graph-search περνάει όλα τα παιδιά ενός κόμβου που κάνουμε expand στο frontier σε αντίθεση με τον συμβατικό dfs που γνωρίζουμε με αποτέλεσμα να υπάρχουν περιπτώσεις με διαφορές τόσο στα expansion όσο και στη διαδρομή που θα επιλεγεί τελικά. Προφανώς εφόσον ο DFS δεν βρίσκει βέλτιστη λύση η διαφορά στο μονοπάτι δεν αποτελεί πρόβλημα.

**Question 2: Breadth First Search**

Η **breadthFirstSearch(problem)** υλοποιήθηκε με βάση τον αλγόριθμο BFS που υπάρχει στις διαφάνειες του μαθήματος. Δηλαδή γίνεται ξανά χρήση frontier και explored δομών για την υλοποίηση του αλγορίθμου όπως στην dfs με τη διαφορά ότι εδώ το frontier είναι queue που υπάρχει έτοιμο στο util.py ως util.Queue. Το λεξικό παρουσιάζει ίδια μορφή με τον dfs.

Ο τρόπος με τον οποίο επιστρέφεται η σειρά των βημάτων είναι και αυτός παρόμοιος με εκείνη που χρησιμοποιήθηκε στο dfs. Ο λόγος που ο autograder.py βγάζει fail στο συγκεκριμένο ερώτημα είναι επειδή ο bfs των διαφανειών δεν κάνει σπάταλα expansions όπως ο συμβατικός bfs δηλαδή όταν εντοπίσει στους απόγονους ενός κόμβου goal-state επιστρέφει τη λύση και δεν το βάζει στο frontier για να του γίνει expansion. Έτσι γλιτόνουμε αρκετά περίττά expansions απ τους υπόλοιπους απογόνους του state στο οποίο βρισκόμαστε.

**Question 3: Varying the Cost Function**

Η **uniformCostSearch(problem)** υλοποιήθηκε με βάση τον αλγόριθμο ucs που υπάρχει στις διαφάνειες του μαθήματος. Χρησιμοποιήθηκαν και εδώ οι δομές frontier και explored για την υλοποίηση του αλγορίθμου, μόνο που εδώ η frontier υλοποιείται ως priority queue με τη χρήση του util.PriorityQueue απ' το αρχείο util.py. Επίσης η δομή του λεξικού είναι διαφορετική σε σχέση με τα προηγούμενα ερωτήματα εφόσον έχει άλλο ένα στοιχείο. Συγκεκριμένα έχουμε πάλι ως κλειδί το state και ως τιμή μια λιστα [Parent, Action, CostOfPath, Boolean] με Boolean να είναι True αν το

κλειδί βρίσκεται στο explored. Η εύρεση του μονοπατιού απ’ το goal-State γίνεται με όμοιο τρόπο με τα προηγούμενα ερωτήματα. Εδώ ο autograder.py τον αξιολογεί τον αλγόριθμο ως ορθό (3/3).

**Question 4: A* Search**

Η **aStartSearch(problem, heuristic=nullHeuristic)** υλοποιήθηκε με βάση τον αλγόριθμο A* που υπάρχει στις διαφάνειες του μαθήματος. Ως κύριως άξονας για τη συγγραφή του κώδικα του αλγορίθμου ήταν ο UCS του προηγούμενου ερωτήματος με ορισμένες αλλαγές, συγκεκριμένα στην συνάρτηση f στην οποία πλέον εκτός απ το g προστίθεται και το heuristic. Δηλαδή όταν κάνουμε push ένα στοιχείο στο priorityqueue στο priority εκτός απ’ το κόστος για να φτάσουμε σε αυτό τον κόμβο προσθέτουμε και το εκτιμώμενο κόστος που βρίσκουμε μέσω της ευρετικής για να φτάσουμε απ’ αυτόν στον κόμβο στόχο.

Χρησιμοποιήθηκαν και εδώ οι δομές frontier και explored για την υλοποίηση του αλγορίθμου, μόνο που εδώ η frontier υλοποιείται ως priority queue με τη χρήση του util.PriorityQueue απ’ το αρχείο util.py. Επίσης η δομή του λεξικού είναι διαφορετική σε σχέση με τα προηγούμενα ερωτήματα εφόσον έχει άλλο ένα στοιχείο. Συγκεκριμένα έχουμε πάλι ως κλειδί το state και ως τιμή μια λιστα [Parent, Action, CostOfPath, Boolean] με Boolean να είναι True αν το κλειδί βρίσκεται στο explored. Η εύρεση του μονοπατιού απ’ το goal-State γίνεται με όμοιο τρόπο με τα προηγούμενα ερωτήματα. Εδώ ο autograder.py τον αξιολογεί τον αλγόριθμο ως ορθό (3/3).

**Question 5: Finding All the Corners**

Στο συγκεκριμένο ερώτημα συμπληρώθηκα ορισμένες μέθοδοι της κλάσης CornersProblem στο seachAgents.py. Για να λυθεί το πρόβλημα δημιούργηθηκε ένα καινούργιο state απ’ αυτό που χρησιμοποιούσαμε μέχρι στιγμής. Αυτό είναι ενα tuple που περιέχει στο πρώτο μέρος του τις συντεταγμένες του state μας επάνω στο grid και στο δεύτερο μέρος του ένα tuple με 4 booleans τα οποία είναι True αν έχουμε επισκεφτεί τη συγκεκριμένη γωνία, σύμφωνα με το self.corners της τάξης. State = (coordinates, (Boolean, Boolean, Boolean, Boolean))

Η **getStartState(self)** επιστρέφει την αρχική κατασταση που έχει ως συντεταγμένες το startingPosition και για Boolean tuple όλα τα πεδία false, εφόσον τότε δεν έχουμε εξερευνήσει καμια γωνία.

Η **isGoalState(self)** επιστρέφει True εάν σε αυτό το state έχουμε επισκεφτεί όλες τις γωνίες του grid δηλαδή το boolean tuple έχει όλα τα παιδία του True.

Η **getSuccessors(self, state)** βρίσκει τους απογόνους ενός κόμβου (οι οποίοι δεν είναι έχουν συντεταγμένες τοίχου). Κατά τη δημιουργία της τριπλέτας (state, action, cost) που θα μπει στη λίστα με τους successors γίνεται το εξής: Αν ο απόγονος είναι γωνία που δεν έχουμε επισκεφτεί τότε κάνοντας τις απαρραίτητες μετατροπές του tuple με τα booleans ώστε να είναι mutable (δηλαδή μετατρέποντας το σε λίστα) αλλάζουμε το πεδίο που λέει πως η επίσκεψη αυτής της γωνίας είναι False σε True και προσθέτουμε την νέα τουπλέτα στη λίστα. Αλλιώς αν ο απόγονος δεν είναι unvisited corner το boolean tuple είναι το ίδιο με του πατέρα του. Επίσης το κόστος των βημάτων είναι 1.

Ο autograder.py σε αυτό το ερώτημα βγάζει fail δεδομένου ότι δεν έχει υλοποιήθει για τον autograder σωστά η q2 (δηλαδή 3/3) άρα δεν τρέχει το πρόγραμμα όμως οι εντολές που δώθηκαν και τα επόμενα ερωτήματα που βασίζονται στο q5 λειτουργούν κανονικά.

**Question 6: Corners Problem: Heuristic**

Στο συγκεκριμένο ερώτημα υλοποιήθηκε η **cornersHeuristic(state, problem)** στο searchAgents.py. Η φιλοσοφία του αλγορίθμου είναι η εξής. Απ' τα unvisited corners δημιουργεί όλα τα δυνατά permutations που μπορούν να παραχθούν. Βασιζόμενος τώρα στον Manhattan distance βρίσκει τον συνδιασμό curr_state + permutation που θα δώσει την μικρότερη απόσταση αθροίζοντας όλες τις αποστάσεις ManHattan. Δηλαδή αν εχουμε curr_state = A και corner_permutation= C, D θα επιστρέφαμε το manhattan_dist(A,C) + manhattan_dist(C, D). Άρα Επιστρέφουμε μικρότερη τιμή που μπορούμε να βρούμε συνδιαζοντας το curr_state με τις διατάξεις των unvisited_corners. Αυτό το πετυχαίνουμε υλοποιόντας μια λίστα που κρατάει αυτές τις συνολικές αποστάσεις και στο τέλος επιστρέφει την μικρότερη. Αν δεν υπάρχουν unvisited corners τότε η ευρετική θα επιστρέψει 0.

Για να βρεθούν όλα τα δυνατά permutations χρησιμοποιήθηκε η βιβλιοθήκη της python "itertools" και συγκεκριμένα η συναρτηση itertools.permutations.

Η ευρετική συνάρτηση είναι παραδεκτή επειδή δεν υπερεκτιμά ποτέ το κόστος προς την κατάσταση στόχο. Αυτό συμβαίνει γιατί το ManHattan distance αποτελεί μια χαλαρωμένη εκδοχή του προβλήματος η οποία δεν λαμβάνει υπόψιν τους τοίχους μέσα στο grid. Έτσι η απόσταση που επιστρέφει θα είναι πάντα μικρότερη ή ίση απ' την ελαχιστή πραγματική απόσταση που πρέπει να διανύσουμε λαμβάνοντας υπόψιν τους τοίχους για να φτάσουμε απ το παρούσα κατάσταση στην κατάσταση στόχου.

Γενικά εφόσον η ευρετική μας είναι παραδεκτή και βασίζεται σε χαλαρωμένη εκδοχή του προβλήματος είναι και πιθανότατα συνεπής (σύμφωνα με τις σημειώσεις του Berkley). Πιο αναλυτικά η ευρετική μας γενικά αποφεύγει τις διαγωνίους αν μπορεί να βρει ενα μονοπάτι προς τη διαγώνιο στην οποία καλύπτει και άλλη μια γωνία εξαιτίας των permutations και του manhattan. Οι γείτονες του curr_state μας είναι οι κομβοι που δεν ειναι τοίχοι και έχουν αυξημένη ή μειωμένη τη συντεταγμένη τους σε έναν απ τους δύο άξονες μόνο κατα 1 (άρα κόστος 1). Άρα η heuristic του γειτονα μαζί με το κόστος να πάει στο state δεν μπορούν να είναι μικρότερα στο χαλαρωμένο πρόβλημα απ την ευρετική του state στο οποίο βρισκόμαστε.

Η ευρετική σύμφωνα με τον autograder είναι ορθη και το q6 παίρνει 3/3.

**Question 7: Eating All the Dots**

Στο συγκεκριμένο ερώτημα υλοποιήθηκε η **foodHeuristic(state, problem)** που βρίσκεται στο searchAgents.py. Η λογική της ευρετικής είναι απλή. Επιστρέφει την πραγματική απόσταση του φαγητού που απέχει περισσότερο απ' τον παρόν κόμβο. Για αυτό χρησιμοποιεί τη συνάρτηση mazeDistance του searchAgents.py.

Η συνάρτηση είναι παραδεκτή εφόσον δεν υπερεκτιμά το κόστος απ' τον κόμβο που βρισκόμαστε στον κόμβο στόχο. Αυτό συμβαίνει γιατί για να "φαει" το πακμαν όλα τα φαγητά πρέπει να διανύσει τουλάχιστον την πραγματική απόσταση προς το μακρυνότερο φαγητό. Δηλαδή αποτελεί κάτω φράγμα της λύσης του προβλήματος.

Για να δείξουμε πως είναι συνεπής σκεφτόμαστε ως εξής. Για να πάμε σε οποιονδήποτε γείτονα του current state μας χρειάζεται κόστος 1. Αν ο γείτονας του έχει το ίδιο φαγητό ως το πραγματικά μακρυνότερο του τότε, είτε αυτό θα είναι το ίδιο μακρυά, ή πιο μακρυά ή πιο κοντά κατά 1. Συνεπώς και στις 3 περιπτώσεις ισχύει η εξίσωση της συνέπειας $h(n) <= 1 + h(n')$ όπου n είναι ο

παρόν κόμβος και n’ ο γείτονας. Στην άλλη περίπτωση το μακρυνότερο φαγητό του γείτονα βρίσκεται σε άλλο κόμβο. Τότε αυτός ο νέος κόμβος φαγητού δεν μπορεί να απέχει περισσότερο απ το γειτονικο state ότι απέχει ο παλιός κόμβος φαγητού απ το παρόν state γιατί και το παρον state θα επέλεγε αυτό. Άρα θα ισχύει h(n’) = h(n) συνεπώς h(n) <= 1 + h(n’) => h(n) <= h(n) +1 => 0 <= 1 που ισχύει. Άρα η συνάρτηση μας είναι παραδεκτη.

Ο autograder σε αυτό το ερώτημα δίνει 5/4 και τα expanded nodes είναι 4137.

**Question 8: Suboptimal Search**

Συμπληρώσαμε το isGoalState της κλάσης AnyFoodSearchProblem του searchAgents.py, στην οποία ουσιαστικά επιστρέφουμε αν το συγκεκριμένο state έχει συντεταγμένες που ανήκουν σε κάποια απ τις συντεταγμένες τον εναπομείναντων φαγητών.

Επίσης συμπληρώσαμε το findPathToClosestDot της κλάσης ClosestDotSearchAgent. Εδώ επιστρέφει το μονοπάτι που δίνει το bfs για το κοντινότερο φαγητό απ το current state μας.

Ο autograder σε αυτό το ερώτημα δίνει 3/3.

**Για περισσότερες πληροφορίες κοιτάξτε τα σχόλια του κώδικα.**