ΙΩΑΝΝΗΣ ΚΑΠΕΤΑΝΓΕΩΡΓΗΣ **Α.Μ. 1115201800061**

PROJECT 0
Τεχνητή Νοημοσύνη
Pacman Project (Πρόβλημα 1)

Πρόβλημα 1 - Pacman Project

Q1 - DepthFirstSearch

- Ο αλγόριθμος αναζήτησης κατά βάθος, όπως και οι υπόλοιποι, έχουν υλοποιηθεί με βάση τους αλγορίθμους που διδαχθήκαμε στο μάθημα και υπάρχουν στις διαφάνειες του μαθήματος.
- Για το σύνορο (frontier) χρησιμοποιείται μια στοίβα (Stack) η οποία είναι ήδη υλοποιημένη στο αρχείο util.py του project.
- Για να αποθηκεύονται τους κόμβους που επισκεπτόμαστε (explored) χρησιμοποιείται ένα set έτσι ώστε να μπορεί να ελέγχεται αν ένας κόμβος υπάρχει στο set σε χρόνο O(1).
- Πριν εισαχθεί ένας successor node στο σύνορο ελέγχεται εάν έχει ήδη προσπελαστεί στο παρελθόν ο κόμβος αυτός (ελέγχοντας εάν υπάρχει στο explored set) και εισάγετε στο σύνορο μόνο εάν δεν έχει ήδη προσπελαστεί.
- Ο έλεγχος για το goal state γίνεται κάθε φορά που κάνουμε pop έναν κόμβο από την στοίβα.
- Στην στοίβα και στο explored set εισάγουμε tuples όπου το καθένα περιέχει τον κόμβο και την διαδρομή (path) που έχουμε ακολουθήσει για να φτάσουμε από τον αρχικό κόμβο στο κόμβο αυτό. Το path είναι μια λίστα από κόμβους ξεκινώντας από τον αρχικό κόμβο και καταλήγοντας στον “πατέρα” του συγκεκριμένου κόμβου.

Q2 - BreadthFirstSearch

Η υλοποίηση είναι σχεδόν ίδια με αυτήν του DFS. Οι μόνες διαφορές είναι:
- Για την υλοποίηση του συνόρου χρησιμοποιείται μια ουρά (Queue) η οποία είναι υλοποιημένη στο αρχείο util.py
- Πριν εισάγουμε έναν successor node στο σύνορο εκτός από το να ελέγξουμε αν υπάρχει ο κόμβος αυτός στο explored set, ελέγχουμε και αν ήδη υπάρχει μέσα στο σύνορο. Αν δεν υπάρχει σε κανένα από τα δύο, τότε εισάγουμε τον κόμβο στο σύνορο.

Q3 - UniformCostSearch

Για την αναζήτηση ομοιόμορφου κόστους έχω επιλέξει μια υλοποίηση η οποία είναι λίγο πιο περίπλοκη και ίσως λίγο πιο σπάταλη από ότι θα χρειαζόταν για να βαθμολογηθεί σωστά από τον autograder.
Αυτό καθώς, όπως παρατήρησα, εισάγοντας στην ήδη υλοποιημένη PriorityQueue tuples τα οποία εκτός από τον κόμβο και το κόστος (προτεραιότητα) περιέχουν και το path (όπως και στους bfs, dfs) έχει ως αποτέλεσμα οι συναρτήσεις διαχείρισης της PriorityQueue να μην λειτουργούν σωστά. Πιο συγκεκριμένα οι συγκρίσεις που γίνονται στην update είναι πάντα αληθείς καθώς εκτός από τους κόμβους συγκρίνονται και τα paths τα οποία προφανώς είναι διαφορετικά αφού σκοπός να είναι να βρούμε ένα καλύτερο μονοπάτι. Έτσι καλώντας την update στην πραγματικότητα δεν γίνεται ποτέ update, αλλά απλά push, γεμίζοντας έτσι την ουρά με διπλότυπα κάτι που είναι λάθος. Τα παραπάνω, φαίνεται να μην τα εντοπίζει ο autograder. Στα σχόλια του κώδικα έχω αφήσει μία απλή υλοποίηση της UniformCostSearch η οποία κάνει όσα εξηγήθηκαν παραπάνω (δηλαδή λανθασμένα) η οποία όμως βαθμολογείται κανονικά από τον autograder.
Για μία πιο σωστή υλοποίηση, στην οποία δουλεύει σωστά η συνάρτηση update της PriorityQueue, επέλεξα να κάνω τα εξής:
- Προφανώς το σύνορο έχει υλοποιηθεί με μια PriorityQueue
- Τα paths αντί να τα βάζουμε την priority queue μαζί με τους κόμβους τα αποθηκεύουμε σε ένα dictionary (keepPaths).
- Κάθε φορά που εισάγουμε έναν κόμβο στο σύνορο, υπολογίζουμε το path (με τον ίδιο τρόπο με τις bfd, dfs) και το αποθηκεύουμε στο dictionary που έχουμε στο σώμα της συνάρτησης UniformCostSearch. Το dictionary έχει σαν κλειδί τον κόμβο και σαν item του κάθε κόμβου ένα tuple με το path του και το κόστος του path αυτού.
- Στην περίπτωση που βρούμε έναν κόμβο ο οποίος πρέπει να γίνει update στην PQ, πρώτα παίρνουμε το παλιό κόστος τους path από το dictionary και το συγκρίνουμε με το κόστος του path του νέου κόμβου. Εάν το κόστος του νέου κόμβου είναι μικρότερο, καλούμε την update για να ανανεώσουμε τον κόμβο με την νέα προτεραιότητα και επίσης ενημερώνουμε το dictionary με το νέο μονοπάτι και το νέο κόστος.
- Κάθε φορά που κάνουμε pop έναν κόμβο από το σύνορο, παίρνουμε το μονοπάτι του από το dictionary χρησιμοποιώντας για κλειδί τον κόμβο αυτό.
- Το κόστος του κάθε μονοπατιού υπολογίζεται με την χρήση της ήδη υλοποιημένης συνάρτησης getCostOfActions() δίνοντας σαν όρισμα το path.

Όλα τα παραπάνω είναι πιο κατανοητά βλέποντας τον κώδικα και τα αντίστοιχα σχόλια.

Q4 - A* search

Στον αλγόριθμο A* για την υλοποίηση του συνόρου χρησιμοποιείται μια PriorityQueue.
Όπως αναφέρεται και στις διαφάνειες του μαθήματος ο A* βασίζεται στον απλό αλγόριθμο GraphSearch. Δηλαδή δεν χρησιμοποιούμε την update για ανανέωση των κόμβων της ουράς, αλλά κάθε φορά απλά κάνουμε push. Έτσι, δεν αντιμετωπίζουμε το πρόβλημα που αναφέρθηκε παραπάνω. Ακολουθεί ίδια λογική με τους bfs και dfs με τις εξής διαφορές:

- Η priority queue ως γνωστών δέχεται σαν ορίσματα το item και την προτεραιότητα του. Σαν item εισάγουμε ένα tuple το οποίο περιέχει τον κόμβο και το path από το οποίο “καταλήξαμε” στον κόμβο αυτό. Σαν προτεραιότητα εισάγουμε το κόστος του path αυτού (χρησιμοποιώντας την getCostOfActions()) συν την τιμή της ευρετικής συνάρτησης για τον κόμβο αυτό.
- Πριν εισάγουμε έναν κόμβο στο σύνορο ελέγχουμε ένα υπάρχει στο explored set (όπως δηλαδή και στην dfs)

Q5 - CornersProblem

__init__:
Το επιπλέον δεδομένο που χρειαζόμαστε είναι μια δομή η οποία θα μας παρέχει πληροφορία για το ποιες γωνίες έχουμε επισκεφθεί. Για την αναπαράσταση της πληροφορίας αυτής χρησιμοποιώ ένα dictionary (cornersVisited). Σαν κλειδί έχει την κάθε γωνία (δηλαδή τις συντεταγμένες της) και σαν value έχει μια boolean μεταβλητή η οποία είναι True αν έχουμε επισκεφθεί την γωνία αυτή και False αν όχι.
Για να είναι διαχειρίσιμη και η πληροφορία αυτή και από τις υπόλοιπες συναρτήσεις, χρειάζεται να μετατρέπουμε το dictionary σε ένα ενιαίο tuple. Έτσι το μετατρέπουμε σε ένα tuple το οποίο περιέχει tuples της μορφής (corner, True/False).
Εσωτερικά των συναρτήσεων που υλοποιούμε, κάθε φορά μετατρέπουμε αυτό το tuple σε dictionary, κάνουμε τις αντίστοιχες ενέργειες πάνω στο dictionary και τελικά το επαναφέρουμε στην μορφή του tuple

getStartState:
Ένα state χαρακτηρίζεται από την τρέχουσα θέση του pacman και από την πληροφορία για το πόσες γωνίες έχει επισκεφθεί (cornersVisited). Έτσι η getStartState επιστρέφει ένα tuple με την τρέχουσα θέση και το cornersVisited τα οποία έχουν αρχικοποιηθεί στην __init__ .

isGoalState:
Αρχικά μετατρέπουμε σε dictionary το cornersVisited. Έπειτα, επιστρέφουμε False αν υπάρχει τουλάχιστον ένα False στα values του dictionary (δηλαδή αν δεν έχουμε επισκεφθεί τουλάχιστον μια γωνία) και True αν όλα τα values του dictionary είναι True (δηλαδή έχουμε επισκεφθεί όλες τις γωνίες).

getSuccessors:
Βρίσκουμε τις συντεταγμένες για κάθε μία από τις πιθανές επόμενες κινήσεις όπως μα υποδεικνύεται από την άσκηση. Για κάθε μία κίνηση:
Ελέγχουμε αν η κίνηση αυτή είναι αποδεκτή, δηλαδή αν χτυπάει κάποιον τοίχο. Αν είναι αποδεκτή, δημιουργούμε την κατάσταση γωνιών (cornersVisited) για την νέα θέση, η οποία βασίζεται στην προηγούμενη κατάσταση με έναν επιπλέον έλεγχο για το αν η νέα θέση είναι μια από τις γωνίες. Αν είναι, στην αντίστοιχη θέση του dictionary βάζουμε True. Έπειτα, όπως εξηγήθηκε, μετατρέπουμε το dictionary σε ένα tuple. Τέλος προσθέτουμε το νέο state, την κίνηση αυτή και το κόστος της κίνησης ως ένα tuple στην λίστα με τους successors.
Το κόστος της κάθε κίνησης έχουμε υποθέσει ότι είναι 1.

Q6 - cornersHeuristic

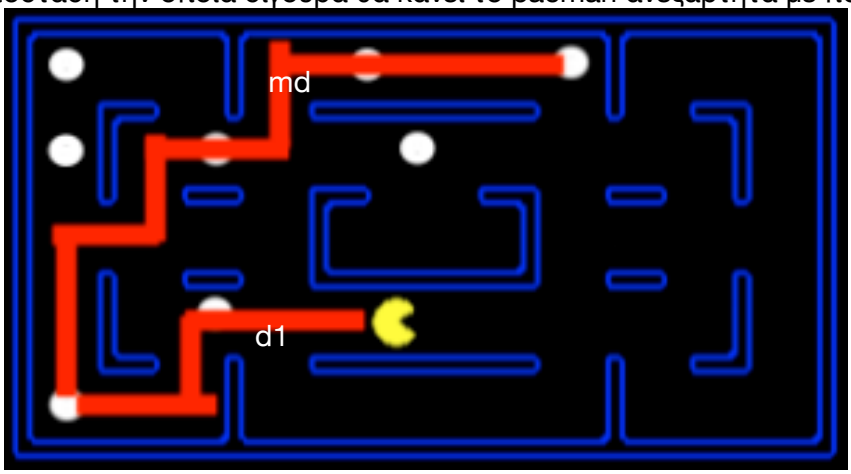
Η ευρετική συνάρτηση αυτή ξεκινάει από την τρέχουσα θέση. Υπολογίζει την απόσταση manhattan από την αρχική θέση στην κοντινότερη γωνία. Έπειτα θέτει ως τρέχουσα θέση την γωνία αυτή και υπολογίζει την απόσταση manhattan για την κοντινότερη γωνία από την νέα τρέχουσα θέση. Ακολουθείται αυτή η διαδικασία επαναληπτικά μέχρι να έχουμε επιστεφθεί όλες τις γωνίες. Επίσης λαμβάνουμε υπόψιν και ότι μπορεί κάποιες γωνίες να έχουν τις έχουμε ήδη επισκεφθεί παλαιότερα (δηλαδή στο cornersVisited state που μας δόθηκε να κάποιες γωνίες να έχουν τιμή True).
Η ευρετική αυτή είναι παραδεκτή καθώς αγνοούμε τελείως τους τοίχους του maze και χρησιμοποιούμε απόσταση manhattan η οποία είναι πάντα μικρότερη από την απόσταση του pacman στο maze με τοίχους. Επίσης με τους “νέους κανόνες” που υποθέτουμε η λύση που βρίσουμε είναι κάθε φορά η βέλτιστη. Άρα ποτέ δεν υπερεκτιμάται το αποτέλεσμα. Με το ίδιο επιχείρημα αποδεικνύεται ότι η ευρετική είναι και συνεπής.
Η ακριβής υλοποίηση της συνάρτησης εξηγείται πλήρως στα σχόλια.
Χρησιμοποιώντας αυτήν την ευρετική συνάρτηση κάνουμε expand 692 κόμβους, δηλαδή βαθμολογείται με 3/3

Q7 - FoodSearchProblem (foodHeuristic)

Αρχικά ο υπολογισμός αποστάσεων στην συνάρτηση αυτήν γίνεται μέσω της συνάρτησης mazeDistance η οποία είναι ήδη υλοποιημένη στο project. Η συνάρτηση αυτή υπολογίζει την ακριβή απόσταση μεταξύ 2 σημείων το maze του pacman (δηλαδή της απόσταση λαμβάνοντας υπόψιν τους τοίχους). Χρησιμοποιώντας της συνάρτηση αυτή πετυχαίνουμε μία πολύ καλύτερη προσέγγιση από ότι θα πετυχαίναμε με συναρτήσεις όπως η απόσταση manhattan. Έτσι έχοντας προσεγγίσει καλύτερα το πραγματικό κόστος η συνάρτηση αναζήτησης κάνει expand πολύ λιγότερους κόμβους. Το πρόβλημα της συνάρτησης mazeDistance είναι ότι είναι χρονοβόρα (όταν την καλούμε πάρα πολλές φορές). Έτσι, έχω χρησιμοποιήσει ένα dictionary το οποίο είναι αποθηκευμένο στα δεδομένα του προβλήματος και έτσι είναι διαθέσιμο κάθε φορά που καλούμε την συνάρτηση για το ίδιο πρόβλημα. Το dictionary είναι το heuristicInfo το οποίο μας προτείνεται και στα σχόλια της άσκησης να χρησιμοποιήσουμε. Έτσι κάθε φορά που υπολογίζεται μια απόσταση μεταξύ 2 σημείων αποθηκεύεται το dictionary και κάθε επόμενη φορά που θα το χρειαστούμε το έχουμε σε χρόνο O(1).

Πιο συγκεκριμένα, όσον αφορά την ουσία της ευρετικής συνάρτησης, συγκρίνουμε κάθε πιθανό συνδυασμό 2 σημείων που υπάρχει φαγητό. Για κάθε έναν συνδυασμό βρίσκουμε:
- Την απόσταση (mazeDistance) των 2 σημείων φαγητού (μεταβλητή: md)
- Την απόσταση από την τρέχουσα θέση του pacman και το σημείου φαγητού 1 (μεταβλητή: d1)
- Την απόσταση από την τρέχουσα θέση του pacman και το σημείου φαγητού 2 (μεταβλητή: d2)

Βρισκούμε την μικρότερη απόσταση από τις d1 και d2, δηλαδή ποιό σημείο φαγητού είναι πιο κοντά στην τρέχουσα θέση του pacman και την προσθέτουμε στην απόσταση md (Έστω distance το αποτέλεσμα).
Έτσι για κάθε συνδυασμό έχουμε υπολογίσει ένα distance.
Επιλέγουμε το μεγαλύτερο distance από όλους του συνδυασμούς.
Το μεγαλύτερο αυτό distance είναι μία απόσταση την οποία σίγουρα θα κάνει το pacman ανεξάρτητα με ποια σειρά θα φάει τα φαγητά, δηλαδή δεν υπερεκτιμά ποτέ το πραγματικό κόστος.
Παράδειγμα:
Η το κόστος της κόκκινης γραμμής είναι το αποτέλεσμα της ευρετικής συνάρτησης.



Η συνάρτηση αυτή φαίνεται να είναι αρκετά αποδοτική καθώς προσεγγίζει καλά το κόστος της πραγματικής λύση και επίσης δεν είναι αργή καθώς εκτός από τους πρώτους υπολογισμούς των αποστάσεων όλες οι άλλες αποστάσεις βρίσκονται σε χρόνο O(1) μέσω του dictionary.
Επίσης η ευρετική αυτή είναι παραδεκτή καθώς το pacman θα πάει σίγουρα κάποια στιγμή στα 2 αυτά φαγητά από τα οποία προέκυψε το μέγιστο distance και η γρηγορότερος τρόπος να πάει και στα δυο είναι αυτός που βρίσκει η ευρετική συνάρτηση. Άρα σίγουρα το κόστος να φάει όλα τα φαγητά είναι μεγαλύτερο η ίσο του αποτελέσματος της ευρετικής. Επίσης αφού χρησιμοποιούμε το mazeDistance είναι εύκολο να αποδειχθεί ότι η ευρετική αυτή είναι και συνεπής.

Απόδειξη ότι η ευρετική συνάρτηση αυτή είναι πολύ αποδοτική είναι ότι για το tricky search χρειάζεται 0.3 seconds για να βρει την λύση ενώ κάνει expand μόνο 353 κόμβους (ενώ για την βαθμολόγηση με 5/4 από τον autograder αρκεί να κάνει expand το πολύ 7000 κόμβους).

Q8 - ClosestDotSearchAgent

Αρχικά πρέπει να υλοποιήσουμε το GoalState του AnyFoodSearchProblem. Στο πρόβλημα μας ενδιαφέρει απλά να πάμε σε ένα σημείο όπου υπάρχει φαγητό. Επομένως για το goal state αρκεί απλά να ελέγξουμε αν στην τρέχουσα θέση του pacman (δηλαδή το state που δίνεται σαν όρισμα) υπάρχει φαγητό (ελέγχοντας αν το state είναι στην λίστα συντεταγμένων των φαγητών).
Όσον αφορά την findPathToClosestDot του ClosestDotSearchAgent αρκεί απλά οριστεί το πρόβλημα με την AnyFoodSearchProblem (το οποίο είναι ήδη έτοιμο) και να αναθέσουμε το πρόβλημα αυτό στον αλγόριθμο αναζήτησης breadthFirstSearch που έχουμε υλοποιήσει. Εναλλακτικά μπορούμε να χρησιμοποιήσουμε και τον αλγόριθμο ομοιόμορφου κόστους.
Τέλος απλά επιστρέφουμε το path που μας δίνει ο αλγόριθμος αναζήτησης.