Σε αυτήν την εργασία ολοκλήρωσα το Pacman Project 1 του Πανεπιστημίου του Berkeley, με βάση τις προδιαγραφές της εκφώνησης.

Στα πρώτα **4 questions** υλοποιώ τους αλγορίθμους **Αναζήτηση Πρώτα σε Βάθος (DFS), Αναζήτηση Πρώτα σε Πλάτος (BFS), Αναζήτηση Ομοιόμορφου Κόστους (UCS) και Αναζήτηση Α***. Σε όλους τους αλγόριθμους ακολουθώ την ίδια διαδικασία, δηλαδή αρχικά δημιούργησα μια δομή για το σύνορο: **στοίβα** για τον *DFS*, **ουρά** για τον *BFS* και **ουρά προτεραιότητας** για *UCS* και *A**, και ένα **set** για τους ήδη επισκεφθέντες κόμβους. Ξεκινώντας από την ***αρχική κατάσταση***, αφού ελέγξω αν είναι κόμβος στόχου, την τοποθετώ στο σύνορο και ύστερα τρέχω μια λούπα, η οποία τερματίζει όταν το σύνορο αδειάσει. Μέσα στη ***λούπα***, βγάζω τον κόμβο που πρέπει με τη συνάρτηση pop, δηλαδή στην ουρά αυτόν που μπήκε πρώτος, στη στοίβα αυτόν που μπήκε τελευταίος και στην ουρά προτεραιότητας αυτόν με την υψηλότερη προτεραιότητα (αυτόν με την μικρότερη τιμή). Τον κόμβο που έβγαλα, ελέγχω αν ανήκει στους ***visited*** κι αν όχι τον βάζω στο *set* των *visited* και ελέγχω αν είναι ***goal state***, αν είναι goal επιστρέφω το path με τα actions που έχω ακολουθήσει για να φτάσω στον κόμβο αυτόν. Αν δεν είναι *goal*, τον κάνω *expand* και παίρνω τους *successors*, τους οποίους τους βάζω στο σύνορο, αν δεν ανήκουν στους *visited,* και επιστρέφω στην αρχή της λούπας. Το **popped** είναι ένα *tuple*, με πρώτο στοιχείο τις συντεταγμένες του κόμβου, δεύτερο στοιχείο τη λίστα με τα *actions* και τρίτο στοιχείο το κόστος. Στον *UCS* στην ουρά προτεραιότητας, ο αριθμός προτεραιότητας είναι το κόστος για να πάω στον κόμβο αυτό, ενώ στον A* είναι το κόστος + την τιμή της ευρετικής για τον κόμβο αυτό. Επειδή η *getSuccessors* στο δεύτερο στοιχείο του *tuple* έχει μόνο το action που χρειάζεται για να φτάσεις στον κόμβο αυτόν από τον προκάτοχο του, σε κάθε κόμβο κάνω *append* το action που χρειάζεται για να πας στον προκάτοχο και έτσι δημιουργείται το συνολικό ***path***.

Στο **question 5** συμπλήρωσα το **cornersProblem**, αρχικά προσθέτοντας στην **init** μια λίστα visitedCorners αρχικοποιημένη σε 0, όπου κάθε στοιχείο δείχνει αν έχει επισκεφθεί ο *pacman* την αντίστοιχη γωνία στο *self.corners* (0 = *False*, 1 = *True*). Επίσης ελέγχω αν το *startingPosition* είναι κάποιο corner και αν είναι, ενημερώνω τη λίστα και κάνω *cast* τη λίστα αυτή σε *tuple*. Στην **getStartState** επιστρέφω ένα tuple με πρώτο στοιχείο το *startingPosition* και δεύτερο στοιχείο τη λίστα *visitedCorners*, που είναι σε μορφή *tuple*, για να είναι εύκολο κάθε στιγμή να έχω πρόσβαση στην πιο πρόσφατη μορφή της δομής αυτής αλλά έτσι κιόλας προσπερνάω τον έλεγχο για το αν ο κόμβος ανήκει στους visited, ο οποίος καλείται στους αλγορίθμους αναζήτησης όπως περιγράφτηκε παραπάνω. Αν επέστρεφα μόνο συντεταγμένες τότε άμα ο pacman έφτανε σε μία γωνία και η μόνη κίνηση που μπορούσε να κάνει ήταν να γυρίσει προς τα πίσω, τότε θα έμενε κολλημένος εκεί γιατί ο προηγούμενος κόμβος θα ήταν στα *visited* και δε θα έμπαινε στο σύνορο. Στην **isGoalState** παίρνω τη λίστα αυτή από το *state[1*] αφού το κάνω cast σε

λίστα, και ελέγχω αν έχω επισκεφθεί όλες τις γωνίες, αν όχι επιστρέφει *False*, αν ναι *True*. Στην **getSuccessors** κάνω ότι ακριβώς κάνει και η αντίστοιχη συνάρτηση στο *PositionSearchProblem*, δηλαδή κάνω *expand* τον κόμβο βρίσκοντας του επόμενους κόμβους στους οποίους μπορώ να πάω με τις επιτρεπτές κινήσεις *(NORTH, SOUTH, EAST, WEST)* με τη διαφορά ότι παίρνω πάλι τη λίστα *visitedCorners* από το *state[1]* και αφού βρω το nextState που να μην είναι τοίχος ελέγχω αν αυτό το *nextStat*e είναι *corner* και ενημερώνω τη λίστα αν χρειαστεί. Κάνω cast τη λίστα πάλι σε *tuple* και τη βάζω σε *tuple* μαζί με το ***nextState***, τα οποία θα αποτελούν το πρώτο στοιχείο του μεγάλου *tuple* που γίνεται *append* στη λίστα *successors* και επιστρέφω τη λίστα αυτή.

Στο **question 6**, υλοποίησα την **cornersHeuristic**, η οποία υπολογίζει την απόσταση (με τρόπο παρόμοιο της *manhattan*) από το *currentState* του pacman μέχρι την κοντινότερη γωνία που δεν έχει επισκεφθεί ακόμα (αν υπάρχει) και μετά αν μένουν κι άλλες γωνίες που δεν έχει επισκεφθεί τότε προσθέτει και την απόσταση μέχρι τη μακρινότερη από αυτές. Με αυτόν τον τρόπο η συνάρτηση αυτή δεν υπερεκτιμά ποτέ την απόσταση από τον στόχο, ο οποίος είναι να επισκεφθεί ο pacman όλες τις γωνίες με αποτέλεσμα να είναι **παραδεκτή (admissible)** και **συνεπής (consistent)**. Επίσης διαλέγοντας την απόσταση της μακρινότερης επιτυγχάνω την παραγωγή λιγότερων κόμβων, αφού το νούμερο που βρίσκω είναι αρκετά κοντά στην πραγματική απόσταση. Αν το *currentState* είναι **goalState** (όλες οι γωνίες να έιναι visited), τότε η συνάρτηση επιστρέφει 0.

Στο **question 7**, υλοποίησα την **foodHeuristic**, η οποία υπολογίζει την απόσταση της μακρινότερης τροφής από το pacman και την μεγαλύτερη απόσταση μεταξύ των τροφών και επιστρέφει τη μεγαλύτερη τιμή από τις δύο αυτές. Έτσι όπως και προηγουμένως, δεν υπερεκτιμά την απόσταση αλλά ταυτόχρονα βρίσκει κάτι πολύ κοντινό στο πραγματικό, με αποτέλεσμα την παραγωγή λίγων κόμβων και η συνάρτηση να είναι **παραδεκτή (admissible)** και **συνεπής (consistent)**. Αν το *currentState* είναι **goalState** (αν έχουν φαγωθεί όλες οι τροφές), τότε η συνάρτηση επιστρέφει 0.

Στο **question 8**, υλοποίησα την **findPathToClosestDot**, αφού πρώτα συμπλήρωσα τη συνάρτηση *isGoalState*, που βρίσκεται παρακάτω στην κλάση *AnyFoodSearchProblem*. Στην **isGoalState**, απλά επιστρέφω το **self.food[x][y]**, το οποίο θα επιστρέψει *True* αν υπάρχει φαγητό στον κόμβο με συντεταγμένες x, y και *False* αν δεν υπάρχει. Στην **findPathToClosestDot**, απλά επιστρέφω το **search.breadthFirstSearch(problem)**, στην ουσία δηλαδή καλείται μία *BFS* αναζήτηση στο πρόβλημα αυτό. Με αυτόν τον τρόπο βρίσκω το **path στην κοντινότερη τροφή** (κουκίδα), εφ' όσον η *BFS* πάντα ψάχνει πρώτα στους κοντινότερους κόμβους και έχοντας έλεγχο για *goalState* αν υπάρχει τροφή στον κόμβο αυτό. Έτσι, όπως βλέπουμε στην **registerInitialState**, μέσα σε μια λούπα, η οποία τελείωνει μόλις φαγωθούν όλες οι τροφές, καλείται στην ουσία κάθε φορά μια *BFS* η οποία βρίσκει την κοντινότερη τροφή.