Στον κώδικα υπάρχουν επεξηγηματικά σχόλια (γραμμένα στα Αγγλικα) στα περισσότερα σημεία. Στα σημεία, τα οποία είναι άμεση μεταφορά του ψευδοκώδικα ή της θεωρίας σε python, δεν υπάρχουν και τόσο αναλυτικά σχόλια, αφού είναι προφανές το τι κάνω. Παρακάτω, παραθέτω περιληπτικά τι κάνω σε κάθε question του Project.

Question 1:

Σε αυτό το ερώτημα, χρειαζόμαστε μια συνάρτηση η οποία αξιολογεί την κατάσταση στην οποία θα βρεθεί ο Pacman αφού κάνει μία συγκεκριμένη κίνηση. Η συνάρτηση επιστρέφει μία μεταβλητή **score**, η οποία ξεκινάει αρχικά με τιμή 0 και ανάλογα με κάποιους παράγοντες, της προσθέτω και τις αφαιρώ πόντους. Κάποιοι παράγοντες έχουν μεγαλύτερη βαρύτητα από άλλους, γιατί είναι πιο σημαντικοί. Στο παιχνίδι του Pacman για να νικήσουμε πρέπει να φάμε όλες τις τροφές και να αποφύγουμε τα φαντάσματα. Αναλυτικότερα, τα νούμερα φαίνονται πιο συγκεκριμένα στον κώδικα και οι εξηγήσεις για κάθε παράγοντα υπάρχουν σε σχόλια αναλυτικά. Τέτοιοι παράγοντες είναι οι εξής: αν η επόμενη κίνηση του Pacman προσφέρει **τροφή** προσθέτω πολλούς πόντους, αν η επόμενη κίνηση τον φέρνει πάνω σε **φάντασμα** αφαιρώ πολλούς πόντους, αν με την επόμενη κίνηση πλησιάζει στην **κοντινότερη τροφή** προσθέτω λίγους πόντους αλλιώς αφαιρώ ελάχιστους και τέλος αν, βρίσκοντας τις **νέες θέσεις των φαντασμάτων**, ο Pacman απομακρύνεται από αυτά με τη νέα του κίνηση ή η απόσταση μένει ίδια, τότε προσθέτω ελάχιστους πόντους.

Question 2:

Σε αυτό το ερώτημα, υλοποίησα τον αλγόριθμο του minimax βάσει της θεωρίας και του ψευδοκώδικα που βρήκα στις διαφάνειες, με κάποιες μικρές αλλαγές. Πιο συγκεκριμένα, έχω φτιάξει 3 συναρτήσεις: minimaxDecision, maxValue και minValue, οι οποίες παίρνουν ως ορίσματα το gameState δηλαδή η τωρινή κατάσταση του παιχνιδιού, ένα index που μας λέει ποιος agent είναι (Pacman ή φάντασμα) και ένα depth που μας λέει σε τι βάθος βρισκόμαστε στην αναζήτηση. Εδώ ο Pacman λειτουργεί ως παίκτης MAX και τα φαντάσματα ως παίκτες MIN. Στην ουσία, παίζει πρώτα ο Pacman και για κάθε επιτρεπτή κίνηση υπολογίζει τις τιμές τον κόμβων των επόμενων καταστάσεων για να βρει ποια κίνηση αποφέρει τους περισσότερους πόντους, δηλαδή ποια κίνηση είναι καλύτερη για αυτόν. Αυτό επιτυγχάνεται, βρίσκοντας τα successorStates που προκύπτουν από τις επιτρεπτές κινήσεις. Σε αυτά τα states παίζει το πρώτο φάντασμα, το οποίο με τη σειρά του ακολουθεί την ίδια διαδικασία, μετά παίρνει θέση και το επόμενο φάντασμα ακολουθώντας πάλι την ίδια διαδικασία και ύστερα το επόμενο κ.ο.κ. . Μόλις παίξουν από μία φορά όλοι οι agents, επαναλαμβάνεται η προηγούμενη διαδικασία μέχρι να φτάσουμε σε τελική κατάσταση (νίκης ή ήττας για τον Pacman) ή σε

μέγιστο βάθος, το οποίο ισούται με αριθμό κινήσεων (ο οποίος ορίζεται ως self.depth από το πρόγραμμα και αρχικοποιείται με την κλήση του προγράμματος μέσα στην κλάση των συναρτήσεων) επί αριθμό παικτών. Όταν φτάσουμε σε αυτές τις καταστάσεις, υπολογίζεται η τιμή τους από την **evaluationFunction** και γυρνώντας προς τα πίσω ο αλγόριθμος (επειδή οι συναρτήσεις καλούν η μία την άλλη) υπολογίζει τις τιμές των ενδιάμεσων κόμβων και έτσι αποκτά και η ρίζα τιμή και τότε μπορεί ο Pacman να διαλέξει την καλύτερη κίνηση.

Στην **minimaxDecision**, αρχικά ελέγχω αν φτάσαμε σε κατάσταση νίκης ή ήττας ή σε μέγιστο βάθος το οποίο είναι ίσο με το βάθος που έχει οριστεί από το πρόγραμμα στην αρχική δημιουργία της κλάσης επί τον αριθμό των agents. Αυτό απαιτείται και από την εκφώνηση "*Important: A single search ply is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times.* ". Αν ισχύει ένα από τα παραπάνω επιστρέφω το αποτέλεσμα της evaluationFunction, η οποία αξιολογεί την κατάσταση και επιστρέφει έναν αριθμό βάσει του πόσο καλή είναι για τον Pacman αυτή η κατάσταση. Αν δεν ισχύει κανένα από αυτά, τότε ελέγχω για ποιον παίκτη ψάχνουμε τη minimax τιμή του και καλώ την maxValue για τον Pacman και την minValue για τα φαντασματάκια.

Στην **maxValue**, αρχικά δημιουργώ ένα tuple ονόματι **v** με πρώτο στοιχείο την τιμή **πλην άπειρο** και δεύτερο στοιχεία μια συμβολοσειρά **"none"**. Καλώ την getLegalActions για να βρω τις πιθανές κινήσεις που μπορεί να κάνει ο agent από εκεί που είναι. Για κάθε τέτοια κίνηση, αναπτύσσω τους successors με την generateSuccessors και έτσι έχω την κατάσταση του παιχνιδιού μετά από αυτήν την κίνηση. Ύστερα, καλώ τη minimaxDecision για κάθε έναν από αυτούς, αυξάνοντας το agentIndex και το depth κατά 1 και αποθηκεύω την τιμή που επιστρέφει στο successorValue. Στην ουσία, δηλαδή, για κάθε κίνηση του Pacman, βρίσκω τις τιμές των κόμβων-παιδιών του, οι οποίοι αντιστοιχούν σε καταστάσεις που κάνει κίνηση το επόμενο φάντασμα στη σειρά. Για κάθε τιμή που βρίσκω, κάνω v = max(v, successorValue), έτσι αν η κίνηση προς αυτόν το successor προσφέρει περισσότερους πόντους, το v θα πάρει την τιμή αυτή και θα αποθηκευτεί και το action στη δεύτερη θέση του tuple. Στο τέλος επιστρέφω το **v**.

Στην **minValue**, αρχικά δημιουργώ ένα tuple ονόματι **v** με πρώτο στοιχείο την τιμή **συν άπειρο** και δεύτερο στοιχεία μια συμβολοσειρά **"none"**. Καλώ την getLegalActions για να βρω τις πιθανές κινήσεις που μπορεί να κάνει ο agent από εκεί που είναι. Για κάθε τέτοια κίνηση, αναπτύσσω τους successors με την generateSuccessors και έτσι έχω την κατάσταση του παιχνιδιού μετά από αυτήν την κίνηση. Ύστερα, καλώ τη minimaxDecision για κάθε έναν από αυτούς, αυξάνοντας το agentIndex και το depth κατά 1 και αποθηκεύω την τιμή που επιστρέφει στο successorValue. Εδώ όμως κάνω και έναν παραπάνω έλεγχο. Αν το επόμενο agentIndex είναι ίσο με getNumAgents (η συνάρτηση αυτή επιστρέφει πόσοι agents παίζουν), τότε σημαίνει πως ελέγξαμε όλους τους agents από μία φορά και έτσι κάνω το index ίσο με 0 για να ελέγξουμε πάλι τον Pacman. Στην ουσία, δηλαδή, για κάθε κίνηση ενός φαντάσματος, βρίσκω τις τιμές των κόμβων-παιδιών του, οι οποίοι αντιστοιχούν σε καταστάσεις που κάνει κίνηση το επόμενο φάντασμα στη σειρά ή ο Pacman αν φτάσαμε στο τελευταίο. Για κάθε τιμή που βρίσκω, κάνω v = min(v, successorValue), έτσι αν η κίνηση προς αυτόν το successor προσφέρει

λιγότερους πόντους, το v θα πάρει την τιμή αυτή και θα αποθηκευτεί και το action στη δεύτερη θέση του tuple. Στο τέλος επιστρέφω το **v**.

Στην **getAction**, καλώ τη maxValue με ορίσματα: gameState, index = 0 γιατί παίζει ο Pacman πρώτος και depth = 0 γιατί ξεκινάμε την αναζήτηση από βάθος 0 και επιστρέφω μόνο το 2ο στοιχείο του tuple, επειδή αυτό αντιστοιχεί στο ζητούμενο Action. Ουσιαστικά, η μία συνάρτηση καλεί την άλλα μέχρι να φτάσουμε σε μέγιστο βάθος ή σε τελική κατάσταση και γυρνώντας προς τα πίσω συμπληρώνονται οι τιμές των κόμβων MIN και MAX και εν τέλει επιστρέφεται το action το οποίο θεωρείται καλύτερο για τον Pacman.

Question 3:

Σε αυτό το ερώτημα, υλοποίησα τον αλγόριθμο του κλαδέματος άλφα βήτα, βάσει της θεωρίας και του ψευδοκώδικα που βρήκα στις διαφάνειες, με κάποιες μικρές αλλαγές. Πιο συγκεκριμένα, έχω φτιάξει 3 συναρτήσεις: **alphaBetaDecision**, maxValue και minValue, οι οποίες παίρνουν ως ορίσματα το gameState δηλαδή η τωρινή κατάσταση του παιχνιδιού, ένα index που μας λέει ποιος agent είναι (Pacman ή φάντασμα) ,ένα depth που μας λέει σε τι βάθος βρισκόμαστε στην αναζήτηση και τα α και β. Εδώ ο Pacman λειτουργεί ως παίκτης MAX και τα φαντάσματα ως παίκτες MIN.

Η διαδικασία που ακολουθείται σε αυτόν τον αλγόριθμο, είναι ακριβώς ίδια με του minimax, όπως αναλύθηκε προηγουμένως (για αυτόν το λόγο δε θα επαναλάβω τα κοινά σημεία), με εξαίρεση των παρακάτω. Στην **getAction** καλώ πάλι τη maxValue, με μόνη διαφορά ότι περνάω και ως ορίσματα τα α και β αρχικοποιημένα σε πλην άπειρο και συν άπειρο αντίστοιχα. Επίσης στην **maxValue** ελέγχω αν η τιμή του successorValue είναι μεγαλύτερη του β, δηλαδή της καλύτερης τιμής μέχρι στιγμής για τον MIN, τότε επιστρέφω κατευθείαν το v γιατί δεν έχει νόημα να συνεχίσουμε παρακάτω, αφού ο MIN προτιμάει μικρότερες τιμές. Αλλιώς θέτω το α = v αν το v είναι μεγαλύτερο του α. Αντίστοιχα, στην **minValue** ελέγχω αν η τιμή του successorValue είναι μικρότερη του α, δηλαδή της καλύτερης τιμής μέχρι στιγμής για τον MAX, τότε επιστρέφω κατευθείαν το v γιατί δεν έχει νόημα να συνεχίσουμε παρακάτω, αφού ο MAX προτιμάει μεγαλύτερες τιμές.

Έτσι, έχουμε τα ίδια αποτελέσματα με τον αλγόριθμο του minimax αλλά γλυτώνουμε κιόλας πολύ χρόνο, διότι ο αλγόριθμος δεν ψάχνει μονοπάτια τα οποία οι παίκτες δε θα τα διάλεγαν ποτέ βάσει των τιμών τους. Οι συναρτήσεις, όπως και στον minimax καλούν η μία την άλλη και έτσι υπολογίζονται οι τιμές των κόμβων των καταστάσεων από το τέλος προς την αρχή.

Question 4:

Σε αυτό το ερώτημα, υλοποίησα τον αλγόριθμο του expectimax, βάσει της θεωρίας και του ψευδοκώδικα που βρήκα στις διαφάνειες, με κάποιες μικρές αλλαγές. Πιο συγκεκριμένα, έχω φτιάξει 3 συναρτήσεις: **expectimaxDecision**, maxValue και expectedValue, οι οποίες παίρνουν ως ορίσματα το gameState

δηλαδή την τωρινή κατάσταση του παιχνιδιού, ένα index που μας λέει ποιος agent είναι (Pacman ή φάντασμα) ,ένα depth που μας λέει σε τι βάθος βρισκόμαστε στην αναζήτηση. Η διαδικασία εδώ, είναι ίδια με του minimax (για αυτόν το λόγο δε θα επαναλάβω τα κοινά σημεία) αλλά δεν υπάρχουν στην ουσία κόμβοι MIN. Αντ' αυτού, τα φαντασματάκια λειτουργούν ως ενδιάμεσοι κόμβοι τύχης από τους οποίους παίρνει τιμές ο MAX (Pacman) και οι ίδιοι παίρνουν τιμές από το άθροισμα των τιμών που επιστρέφουν οι κινήσεις των φαντασμάτων επί της πιθανότητας της αντίστοιχης κίνησης.

Η **expectimaxDecision** λειτουργεί όπως η minimaxDecision, απλά όταν παίζουν τα φαντασματάκια καλείται η expectedValue για να υπολογιστεί η τιμή των κόμβων τύχης.

Στην **expectedValue** έχω ορίσει τις κινήσεις ως ισοπίθανες, για αυτό και βάζω probability = 1.0 / length, όπου length είναι το πόσες είναι οι πιθανές κινήσεις σε κάθε κατάσταση. Αρχικά, θέτω τη μεταβλητή chance ίση με 0 και μετά για κάθε κίνηση βρίσκω την τιμή της successorValue, την πολλαπλασιάζω με το probability και προσθέτω το αποτέλεσμα στο chance. Στο τέλος της συνάρτησης επιστρέφω ένα tuple με 1ο στοιχείο τη μεταβλητή chance και 2ο στοιχείο το action το οποίο επιλέχθηκε τυχαία τελευταίο, εφ' όσον δε μας νοιάζει στην ουσία ποια κίνηση κάναν τα φαντάσματα, αφού κινούνται τυχαία. Εδώ μας ενδιαφέρει μόνο η αναμενόμενη τιμή, δηλαδή ο μέσος όρος των τιμών για όλες τις δυνατές κινήσεις που προκύπτει από το άθροισμα των τιμών που επιστρέφουν οι κινήσεις των φαντασμάτων επί της πιθανότητας της αντίστοιχης κίνησης.

Question 5:

Σε αυτό το ερώτημα, δημιούργησα μια συνάρτηση αξιολόγησης, όπως στο 1ο ερώτημα αλλά αυτή τη φορά πολύ πιο λεπτομερής. Αυτήν τη φορά δεν έχουμε κάποια ιδέα για το τι κίνηση πρόκειται να κάνει ο Pacman, παρά μόνο για την τωρινή κατάσταση. Πάλι εδώ προσθέτω και αφαιρώ πόντους στη μεταβλητή score ανάλογα με κάποιους παράγοντες. Σε αυτό το σημείο να σημειώσω ότι η μεταβλητή score που χρησιμοποιώ είναι διαφορετική από τη getScore και όταν λέω ότι προσθέτω ή αφαιρώ πόντους, αλλάζω την τιμή της δικιάς μου μεταβλητής score. Η getScore μας επιστρέφει την τιμή του score που φαίνεται κάτω αριστερά όταν τρέχει το παιχνίδι. Αρχικά ελέγχω αν βρισκόμαστε σε κατάσταση νίκης και επιστρέφω κατευθείαν πολλούς πόντους, αλλιώς αν βρισκόμαστε σε κατάσταση ήττας επιστρέφω κατευθείαν μείον πολλούς πόντους. Μετά, παίρνω το score του παιχνιδιού αυτή τη στιγμή από τη getScore και το προσθέτω πολλαπλασιασμένο επί 10 γιατί αυτή η τιμή είναι η πιο αντιπροσωπευτική για το πόσο καλά πάει το παιχνίδι. Γενικά, παρατήρησα ότι η evaluationFunction μου αργούσε πολύ και δε λειτουργούσε καλά όταν χρησιμοποιούσα μεγάλες τιμές. Έτσι αποφάσισα να χρησιμοποιήσω αρνητικά βάρη στους παράγοντες. Παράγοντες είναι: το πόσες τροφές μένουν, το πόσες κάψουλες (pellets) μένουν και οι αποστάσεις από την κοντινότερη τροφή και το κοντινότερο τρομαγμένο φάντασμα. Έτσι για παράδειγμα

όσο περισσότερες τροφές του μένουν του Pacman τόσο περισσότερους πόντους του αφαιρώ (μιας και ο βασικός στόχος του παιχνιδιού είναι το να φάει όλες τις τροφές). Όσο περισσότερες κάψουλες μένουν τόσο περισσότερους πόντους του αφαιρώ, αφού είναι πολύ σημαντικό να τρώει τις κάψουλες γιατί κάνουν τα φαντάσματα τρομαγμένα και ευάλωτα σε φάγωμα, τα οποία είναι ο βασικός εχθρός του. Όσο πιο μακριά είναι η κοντινότερη τροφή τόσο περισσότερους πόντους του αφαιρώ, διότι βασικός στόχος όπως είπαμε είναι να φάει όλες τις τροφές, άρα αν απομακρύνεται από αυτές είναι κακό. Όσο πιο μακριά είναι το κοντινότερο τρομαγμένο φάντασμα, τόσο πιο πολλούς πόντους του αφαιρώ, επειδή τα τρομαγμένα φαντάσματα πρέπει να τα κυνηγάει για να τα φάει. Ένας ακόμη παράγοντας είναι τα κανονικά φαντάσματα, για τα οποία ελέγχω την απόσταση κι αν είναι πάρα πολύ κοντά στον Pacman (manhattan απόσταση < 2) τότε αφαιρώ πάρα πολλούς πόντους και τέλος αν η κοντινότερη τροφή είναι πιο κοντά από το κοντινότερο κανονικό φάντασμα τότε προσθέτω ελάχιστους πόντους, έτσι ώστε ο Pacman να προτιμήσει να πάει στην τροφή και πιο συγκεκριμένα αν μένει μόνο μία τροφή τότε πρέπει να το βάλει ως προτεραιότητα. Γενικά, έχω δώσει μεγαλύτερη βαρύτητα στις κάψουλες και στην τιμή που επιστρέφει η getScore γιατί οι κάψουλες βοηθάνε τον Pacman να αντιμετωπίσει τους εχθρούς του και το η getScore μας δίνει μια καλή εικόνα για το πόσο καλά πάμε. Αμέσως μετά στην προτεραιότητα έρχονται τα τρομαγμένα φαντασματάκια, γιατί άμα υπάρχουν είναι σημαντικό να τα φάει άμεσα για να πάρει πολλούς πόντους στο παιχνίδι και να γλυτώσει από το να πεθάνει από αυτά. Επόμενο στην προτεραιότητα, έχουμε το πόσες τροφές μένουν γιατί αποτελούν το βασικό στόχο του παιχνιδιού και τέλος έχουμε το πόσο κοντά είναι η κοντινότερη τροφή. Τα κανονικά φαντάσματα έχουν την μικρότερη βαρύτητα στο αποτέλεσμα της συνάρτησης εκτός αν έρθουν πάρα πολύ κοντά, που τότε θα αφαιρεθούν πάρα πολλοί πόντοι. Στον κώδικα φαίνεται αναλυτικότερα όλη η διαδικασία μαζί με τα νούμερα, τα αρνητικά βάρη που αντιστοιχούν σε κάποιους παράγοντες και οι πράξεις. Επίσης υπάρχουν σχόλια, όπου επεξηγώ τα παραπάνω δίπλα από κάθε περίπτωση.