PACMAN PROJECT 1

q1, q2

Σε αυτές τις ασκήσεις υλοποιήθηκε ο αλγόριθμος όπως περιγράφεται στις διαφάνειες:

Αναζήτηση Πρώτα κατά Πλάτος

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution or failure

node ← a node with STATE=*problem*.INITIAL-STATE, PATH-COST=0
if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

frontier ← a FIFO queue with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*)
  add *node*.STATE to *explored*
  for each *action* in *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      if *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)
      frontier ← INSERT(*child*, *frontier*)

Υπάρχουν μερικές διαφορές (optimizations) στην ακριβής υλοποίηση όσον αφορά στο πότε ελέγχουμε για goal state διότι δεν αρέσει στον autograder να κάνουμε περιττά expansions.

Το "explored" υλοποιήθηκε με set() στην python, ενώ το frontier με ουρά (στο DFS, q1) και αντίστοιχα με στοίβα (στο BFS, q2). Η στοίβα και η ουρά διαθέτονται έτοιμες από το `util.py`. Η διαφορά αυτή του frontier είναι η μόνη διαφορά μεταξύ του BFS και του DFS.

Στο frontier εισάγεται ένα tuple που περιέχει το node (state) και το path μέχρι τώρα (για το επιστρέψουμε στο τέλος), αποθηκευμένο σε list.

q3, q4

Εδώ πάλι υλοποιήθηκαν οι αλγόριθμοι UCS και Astar όπως περιγράφονται στις διαφάνειες.

Αναζήτηση Ομοιόμορφου Κόστους

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution or failure

node ← a node with STATE=*problem*.INITIAL-STATE, PATH-COST=0
frontier ← a priority queue ordered by PATH-COST, with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*)
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  add *node*.STATE to *explored*
  for each *action* in *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      frontier ← INSERT(*child*, *frontier*)
    else if *child*.STATE is in *frontier* with higher PATH-COST **then**
      replace that *frontier* node with *child*

Για το frontier εδώ χρησιμοποίησα το priority queue που ορίζεται στο `util.py`.

Επιπλέον, χρησιμοποίησα και ένα dictionary (`costs`) για να κάνω track το cost του κάθε node στο prioq.

Αξίζει να σημειωθεί ότι αυτή η συνθήκη:

if *child*.STATE is not in *explored* or *frontier* **then**
  frontier ← INSERT(*child*, *frontier*)
else if *child*.STATE is in *frontier* with higher PATH-COST **then**
  replace that *frontier* node with *child*

Διαχειρίζεται πλήρως από το Priority Queue:

```
def update(self, item, priority):
    # If item already in priority queue with higher priority, update its priority and rebuild the hea
    # If item already in priority queue with equal or lower priority, do nothing.
    # If item not in priority queue, do the same thing as self.push.
    for index, (p, c, i) in enumerate(self.heap):
        if i == item:
            if p <= priority:
                break
```

οπότε είναι περιττή.

Στο prioq εισάγεται ένα tuple που κρατάει το node (state), το path και το cost.

Η υλοποίηση του astar είναι ίδια με του UCS απλά στο priority queue εισάγουμε τα nodes προσθέτοντας την ευρετική στο κόστος.

q5

Για το cornersproblem υλοποίσα ένα abstract state που περιέχει το position του pacman καθώς και ένα tuple που κωδικοποιεί ποια corners έχουμε εξερυνήσει (`(0, 0, 0, 0)` για κανένα node explored και `(1, 1, 1, 1)` για όλα (goal state)) . Ο λόγος που δεν χρησιμοποίησα λίστα ήταν επειδή δεν είναι hashable και δημιουργούσε πρόβλημε με το visited στο bfs implementation.

Αντίστοιχα στο getSuccessors ενημερώνεται το tuple ανάλογα με το αν το node που κάνουμε expanded είναι corner. (πρέπει να το μετατρέψουμε σε list και μετα πάλι σε tuple για να κανουμε edit ενα entry επειδη δεν επιτρέπεται το modification των tuples).

q6

Για heuristic στο q6 χρησιμοποιείται το εξής:
heuristic = manhattan distance μεταξύ του state και του furthest corner

Είναι εύκολο να δει κανείς γιατί αυτό είναι admissable. Όποια και να είναι η λύση, σίγουρα θα πρέπει να φτάνει στο πιο μακρινό corner. Και εφ'όσον το manhattan distance πάντα κάνει understimate την απόσταση σε αυτό το corner, είναι admissible.

q7

Για heuristic στο q7 είχα την εξής ιδέα. Για κάθε ζευγάρι κόμβων υπολογίζω την μεταξύ τους απόσταση με το mazeDistance και θα το αποθηκευσω στο heuristicInfo, για να το κάνω μόνο μία φορά καθώς η mazeDistance είναι χρονοβόρα.

Έπειτα, θα χρησιμοποιήσω ως heuristic την απόσταση μεταξώ των δύο dots με την μεγαλύτερη μεταξύ τους απόσταση. Όποιο και αν είναι το μονοπάτι, θα περιέχει αυτά τα δύο dots, επομένως θα πρέπει να καλύψει αυτήν την απόσταση. Άρα είναι admissable. Αν υπάρχει ένα dot, απλά επιστρέφω το manhattan distance από το pacman position σε αυτό το dot.

q8

Η λύση του q8 ήταν εξαιρετικά απλή. Πρώτα συμπληρώθηκε το goal test του AnyFoodSearchProblem (που ήταν απλά `return self.food[x][y]`) και έπειτα στο ClosestDotSearchAgent λύνουμε το πρόβλημα με bfs (`return search.bfs(problem)`).