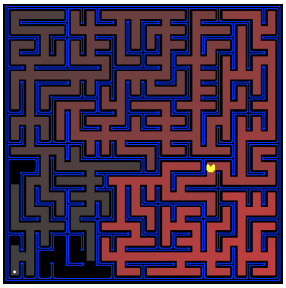
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837



Τεχνητή Νοημοσύνη Ι

Βασιλική Χριστοφιλοπούλου
1115202000216

Οκτώβριος 2024

Χειμερινό Εξάμηνο 2024-2025

Project 1

Table of Contents

1 Ερώτημα 1 : Finding a Fixed Food Dot using Depth First Search

Για την δημιουργία του αλγορίθμου αναζήτησης κατά βάθος, χρησιμοποιήθηκε ο αλγόριθμος που παρουσιάζεται στις διαφάνεις της δέυτερης ενότητας blind1spp.pdf στη σελίδα 53 αλλά και η δομή που παρουσιάζεται στο graph_search.pdf σελίδα 19.

Πιο αναλυτικα, η δομή σύνορο (fringe), υλοποιείται ως μια στοίβα LIFO, όπου αποθηκεύονται κόμβοι προς εξέταση και το μονοπάτι που πραγματοποιούμε για να φτάσουμε εκεί.
Κάθε κόμβος αποτελείται απο τέσσερα στοιχεία

1. state
2. parent node
3. action
4. path cost

Επείτα, όσο η στοίβα δεν είναι άδεια, αφαιρείται ο τελευτάιο κόμβος και αν δεν έχει ήδη εξερευνηθεί, προστίθεται στο explored_set και ανακτώνται οι διάδοχοι του κόμβου. Για κάθε διάδοχο, ελέγχεται αν δεν έχει ήδη εξερευνηθεί και αν ισχύει, προστίθεται στη στοίβα για μελλοντική επεξεργασία. Στο τέλος, επιστρέφεται το καλύτερο μονοπάτι.

Ο κώδικας δοκιμάστηκε όπως αναφέρεται και απο την εκφώνηση για τις ακόλουθες εντολές:

```
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
python autograder.py -q q1
```

Σημείωση: Για λόγους testing πειράχθηκε η δομή Stack ώστε να μπορώ να εκτυπώνω τα περιέχομενα του fringe.

```
def __repr__(self):
    return f"Stack({self.list})"

def __str__(self):
    return "Stack:" + str(self.list)
```

2 Ερώτημα 2 : Breadth First Search

Για την δημιουργία του αλγορίθμου αναζήτησης κατά πλάτος, χρησιμοποιήκε ακριβώς ο ίδιος κώδικας της DFS με μόνη διαφορά ότι το fringe υλοποιείται ως μια ουρά FIFO.
Ο κώδικας δοκιμάστηκε όπως αναφέρεται και απο την εκφώνηση για τις ακόλουθες εντολές:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
python eightpuzzle.py
python autograder.py -q q2
```

3 Ερώτημα 3 : Varying the Cost Function

Για την δημιουργία του αλγορίθμου αναζήτησης ομοιόμορφου κόστους, χρησιμοποιήκε παρόμοιος κώδικα με αυτόν της DFS, με μόνη διαφορά ότι το fringe υλοποιείται ως ουρά προτεραιότητας (Priority Queue). Η ουρά προτεραιότητας χρειεάζεται ενα επιπλέον δεδομένο, με βάση την τιμή του οποίου γινεται η επιλογή του επόμενου κόμβου. Ετσι, καταχωρούμε το κόμβο, το μονοπάτι και το κόστος που απαιτείται για να φτάσουμε απο την αφετηρία στον κόμβο που βρισκόμαστε.

```
fringe.push((node, newpath), cost)
```

Μια διαφορά σε σχέση με τους παραπάνω αλγορίθμους είναι ότι για κάθε successor ελέγχεται αν είναι ήδη στην ουρά με υψηλότερο κόστος, και τότε η ουρά ενημερώνεται ώστε να έχει τώρα το χαμηλότερο κόστος. Προκειμένου να υλοποιηθεί αυτή η διαφορά προγραμματιστικά, διατηρείται ένα λεξικό με την προτεραιότητα κάθε κόμβου.

```
priority_dic[i[0]] = cost
```

Ο κώδικας δοκιμάστηκε όπως αναφέρεται και απο την εκφώνηση για τις ακόλουθες εντολές:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
python autograder.py -q q3
```

4 Ερώτημα 4 : A* search

Ο αλγόρθισμος A* είναι ίδιος με τον αλγόριθμο αναζήτησης ομοιόμορφου κόστους, με μόνη διαφορά την προσθήκη της ευριστικής (heuristic) για τον υπολογισμό του κόστους άρα και της προτεραιότητας. Οπώς και στην UCS, διατηρείται και εδώ ένα λαξικό για να υπάρχει η αντιστοιχία κόμβου και προτεραιότητας.

```
priority = cost + heuristic(node[0], problem)
priority_dic[i[0]] = priority
```

Ο κώδικας δοκιμάστηκε όπως αναφέρεται και απο την εκφώνηση για τις ακόλουθες εντολές:

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
python autograder.py -q q4
```

5 Ερώτημα 5 : Finding All the Corners

Για την υλοποίηση του ερωτήματος ακολουθήσα την εξης λογικη:
Αρχικά, δημιουργούμε μια λίστα με όλες τις γωνίες του packman και κάθε φορά που επισκεπτόμαστε μία, αφαιρείται απο την λίστα. Το πρόγραμμα τερματίζει, όταν έχουμε επισκεφτεί όλες τις γωνίες.
Πιο συγκεκριμένα,

- στην getStartState δημιουργούμε μια λίστα με όλες τις γωνίες του packman, και επιστρέφουμε τόσο την αρχική κατάσταση του packman όσο και την λίσα με τις γωνίες.

```
notvisited = list(self.corners)
return (self.startingPosition, tuple(notvisited))
```

- στην isGoalState, ελέγχεται αν έχουμε επισκεφτεί όλες τις γωνίες και αν ναι βρσικόμαστε στην τελική κατάσταση.

```
notvisited = state[1]
if len(notvisited) == 0:
    return True
return False
```

- στην getSuccessors, αρχικά ακολουθούνται τα ίδια βήματα που δώθηκαν σε σχόλιο και επείτα ελέγχεται αν ο δίαδοχος χτυπάει σε τοίχο. Αν ναι, τότε είναι λάθος κίνηση και παραλείπεται, διαφορετικά αν είναι γωνία, αφαιρείται αρχικά απο την λίστα και μετά προστείθεται στους successors, αλλίως απλά προστείθεται στην λίστα με τους successors.

```
if not hitsWall:
    newposition = (nextx, nexty)
    if newposition in notvisited:
        new_notvisited = list(notvisited)
        new_notvisited.remove(newposition)
        nextstate = (newposition, tuple(new_notvisited))
    else:
        nextstate = (newposition, notvisited)
    successors.append((nextstate, action, 1))
```

- Οι __init__ και getCostOfActions δεν τροποποιήθηκαν.

Ο κώδικας δοκιμάστηκε όπως αναφέρεται και απο την εκφώνηση για τις ακόλουθες εντολές:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python autograder.py -q q5
```

6 Ερώτημα 6 : Corners Problem: Heuristic

Στο ερώτημα αυτό ζητείται η δημιοργία μιας ευρετικής η οποία θα χρησιμοποιηθεί απο τον αλγόριθμο A* προκειμένου το pacman να επισκευτεί και τις τέσσερις γωνίες, προκειμένου να φάει τις τελείες που βρίσκονται εκεί. Όπως υλοποιήθηκε και getStartState του προηγούμενου ερωτήματος, η τρέχουσα κατάσταση (state) αποτελείται απο τις συντεταγμένες (position) και μια λίστα με τις γωνίες δεν έχουν επισκεφτεί ακόμα (notvisited). Η λογική που ακολούθησα για την υλοποίηση είναι ότι:

1. Αρχικά ελέγχεται αν η τρέχουσα κατάσταση είναι και η τελική.
2. Αν δεν είναι, υπολογίζονται οι αποστάσεις απο το τρέχον σημείο ως προς όλες τις μη επισκέψιμες γωνίες, χρησιμοποιώντας την Manhattan απόσταση που μας έχει δωθεί απο util.py. Οι αποστάσεις αποθηκεύονται στη λίστα distances.

3. Η συνάρτηση τελικά επιστρέφει τη μέγιστη απόσταση από την τρέχουσα θέση προς οποιαδήποτε από τις μη επισκέψιμες γωνίες.

Επιλέχθηκε η απόσταση Manhattan ως ευρετική αφού είναι πάντα ένα κατώφλι για την πραγματική απόσταση, καθώς δεν μπορεί να υπερεκτιμά την απόσταση που απαιτείται για να φτάσει ο Pacman σε μια γωνία. Ο υπολογισμός της Manhattan απόστασης είναι απλός και αποδοτικός, κάτι που είναι σημαντικό για την ταχύτητα εκτέλεσης του αλγορίθμου A*.

Ο κώδικας δοκιμάστηκε όπως αναφέρεται και απο την εκφώνηση για τις ακόλουθες εντολές:

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
python autograder.py -q q6
```

7 Ερώτημα 7 : Eating All The Dots

Σε αυτό το ερώτημα μας χρειάζεται να βρούμε ένα path προκειμένου το pacman να καταφέρει να "φάει" όλες τις τελείες. Συγκεκριμένα πρέπει να βρούμε μια ευρετική μέσω της οποίας ο αλγόριθμος A* θα βρίσκει όλες τις τελείες και το pacman θα τις τρώει.

Για αυτή την λύση, χρειάζονται μόλις 3.7 sec και επεκτάθηκαν 8447 κόμβοι. Πιο αναλύτικά, βρίσκεται αρχικά το πιο κοντινό σημείο με φαγητό χρησιμοποιώντας την Manhattan Distance, και έπειτα, για τον σημείο αυτό, υπολογίζεται και επιστρέφεται η mazeDistance. Αφού οι υπολογισμοί για να βρεθούν οι Manhattan αποστάσεις, είναι σημαντικά λιγότεροι, η λύση αυτή κερδίζει σε χρόνο αλλά επεκτείνει περισσότερους κόμβους.

```
closestFood = closest_point(position, foodList)
distance = mazeDistance(position, closestFood, problem.startingGameState)
return distance
```

Ο κώδικας δοκιμάστηκε όπως αναφέρεται και απο την εκφώνηση για τις ακόλουθες εντολές:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
python autograder.py -q q7
```

8 Ερώτημα 8 : Suboptimal Search

Στο συγκεκριμένο ερώτημα μας ζητείτε να συμπληρώσουμε την συνάρτηση findPathToClosestDot έτσι ώστε να βρίσκει ένα μονοπάτι προς την κοντινότερη τελεία. Για την υλοποιήση αυτή υπάρχουν δύο αλγόριθμοι που μπορουν να αξιοποιηθούν οι οποίοι και οι δύο είναι πλήρεις και βέλτιστοι αλγόριθμοι:

1. Αναζήτηση πρώτα κατά πλάτος (BFS)
2. A*

Επιπλέον, συμπληρώσαμε την συνάρτηση isGoalState της κλάσης AnyFoodSearchProblem έτσι ώστε αν η τωρινή κατάσταση υπάρχει και στην λίστα με τα σημεία που υπάρχει τελεία, να επιστέφει True καθώς βρίσκεται σε κατάσταση στόχου.

```
if state in self.food.asList():
    return True
else:
    return False
```

Ο κώδικας και των δύο αναζητήσεων δοκιμάστηκε όπως αναφέρεται και απο την εκφώνηση για τις ακόλουθες εντολές:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
python autograder.py -q q8
```