ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Τεχνητή Νοημοσύνη Ι

Πρώτη Εργασία
Project1 - Pacman

Συγγραφή : Χρήστος Νίκου
ΑΜ: 1115201800330

20 Οκτωβρίου 2022

Περιεχόμενα

1 Αρχείο search.py

1.1 Συνάρτηση depthFirstSearch

Η υλοποίηση που χρησιμοποιήθηκε φαίνεται και στο Σχήμα 1, στην οποία έχουν παραλειφθεί τα σχόλια του Berkley. Για την υλοποίηση των συναρτήσεων χρησιμοποιήθηκαν βοηθητικές συναρτήσεις και κλάσεις που ορίζονται στο βοηθητικό αρχείο custom_functions.py. Η λειτουργία τους δίνεται στα παρακάτω.

Στην αναζήτηση που γίνεται πρώτα κατά βάθος διαλέγεται για επέκταση πάντα ο πρώτος κόμβος που συναντάμε στο κάθε επίπεδο του δένδρου. Έτσι, το «σύνορο» σε αυτή την περίπτωση υλοποιείται μέσω μιας στοίβας. Για την υλοποίηση της στοίβας χρησιμοποιείται η έτοιμη υλοποίηση στοίβας *Stack* που βρίσκεται στο βοηθητικό αρχείο *util.py*. Αρχικά, αρχικοποιείται η στοίβα σε μια μεταβλητή εν ονόματι *fringe* στη γραμμή 4 του Σχήματος 1. Η μεταβλητή *closed* ορίζει ένα σύνολο στο οποίο αποθηκεύονται όλοι οι κόμβοι που έχουν επισκεφθεί απ' την αναζήτηση. Με αυτό τον τρόπο αποφεύγουμε την περίπτωση να επεκταθεί δύο φορές ο ίδιος κόμβος. Η προσέγγιση αυτή αντιστοιχεί στην υλοποίηση του graphSearch. Στη γραμμή 6 ορίζεται ο αρχικός κόμβος του γράφου χρησιμοποιώντας την κλάση *tree_node* που βρίσκεται στο βοηθητικό αρχείο συναρτήσεων custom_functions.py. Ένα αντικείμενο tree_node περιέχει πληροφορία για την τοποθεσία του pacman στον χάρτη μέσω της μεταβλητής *state*. Τον κόμβο από τον οποίο πρόηλθε ο δεδομένος κόμβος μέσω της μεταβλητής *parentnode*, την ενέργεια μέσω της οποία φτάσαμε στον δεδομένο κόμβο μέσω της μεταβλητής *action*, όπου οι δυνατές τιμές είναι μια απ' τις West, East, South, North. Τέλος, το αντικείμενο tree_node περιέχει πληροφορίες για το συνολικό κόστος του μονοπατιού αλλά και το βάθος στο οποίο βρίσκεται ο δεδομένος κόμβος μέσω των μεταβλητών *pathcost* και *depth* αντιστοίχως. Ο αλγόριθμος συνεχίζει να επεκτείνει κόμβους μέχρις ότου η στοίβα *fringe* να αδειάσει. Κάθε φορά που εξάγεται ένας κόμβος απ' τη στοίβα εξετάζεται αν αυτός είναι ο κόμβος στόχος του προβλήματος. Αυτό γίνεται μέσω της συνθήκης στη γραμμή 12 του Σχήματος 1. Στη περίπτωση που ο δεδομένος κόμβος είναι ο κόμβος στόχου η συνάρτηση επιστρέφει μια λίστα με τις ενέργειες που πραγματοποιήθηκαν μέχρι τον στόχο, αυτό γίνεται στη γραμμή 13 μέσω της βοηθητικής συνάρτησης *get_path* που βρίσκεται στο αρχείο *custom_functions.py*. Σε αντίθετη περίπτωση, εξετάζεται αν ο κόμβος αυτός έχει ήδη επεκταθεί μέσω της συνθήκης στη γραμμή 14. Εάν δεν έχει επεκταθεί τότε αυτός προστίθεται στο σύνολο *closed* και επεκτείνεται μέσω της βοηθητικής συνάρτησης *expand_tree*. Η συνάρτηση *expand_tree* δέχεται ως ορίσματα τον κόμβο, το σύνορο, το πρόβλημα και μια κατηγορική μεταβλητή που υποδηλώνει σε ποια περίπτωση αλγορίθμου βρισκόματε. Π.χ. για την περίπτωση της αναζήτησης πρώτα σε βάθος η μεταβλητή αυτή έχει την τιμή "DFS". Η συνάρτηση προσθέτει όλους τους διαδόχους του κόμβου στη στοίβα επεκτείνοντας με αυτόν τον τρόπο το δένδρο.

```python
def depthFirstSearch(problem: SearchProblem):
    # My implementation of GraphSearch with DFS
    # DFS is implemented using a stack
    fringe = util.Stack() # Initialize an empty stack
    closed = []
    starting_node = tree_node(state = problem.getStartState(),
                              ParentNode=None, Action=None, PathCost=0,
                              Depth=0) # Initialize the Starting node
    fringe.push(starting_node) # Append the starting node in stack
    while not fringe.isEmpty():
        node = fringe.pop()
        if problem.isGoalState(node.state): # Then we have found the goal
    state
            return get_path(node) # Return the path that leads to the goal
        elif node.state not in closed:
            closed.append(node.state)
            fringe = expand_tree(node, fringe, problem, mode = "DFS")
    if fringe.isEmpty():
        print(f"- Search algorithm finished without reaching to a solution.
    ")
    util.raiseNotDefined()
```

Listing 1: Implementation of DFS

1.2 Συνάρτηση breadthFirstSearch

Η υλοποίηση της αναζήτησης πρώτα κατά πλάτος είναι ακριβώς η ίδια με την υλοποίηση της αναζήτησης πρώτα σε βάθος με τη μόνη διαφορά ότι τώρα το σύνορο *fringe* υλοποιείται μέσω μιας ουράς ουράς που ακολουθεί τη λογική FIFO (first in first out). Η ουρά χρησιμοποιεί την έτοιμη υλοποίηση *Queue* που βρίσκεται στο αρχείο *util.py*. Παρακάτω βλέπουμε και τον κώδικα της υλοποίησης.

```python
def breadthFirstSearch(problem: SearchProblem):
    # My BFS Iplementation of GraphSearch with BFS
    # BFS is implemented using a queue
    fringe = util.Queue() # Initialize an empty queue
    closed = []
    starting_node = tree_node(state = problem.getStartState(),
                              ParentNode=None, Action=None, PathCost=0,
                              Depth=0) # Initialize the Starting node
    fringe.push(starting_node) # Append the starting node in queue
    while not fringe.isEmpty():
        node = fringe.pop()
        if problem.isGoalState(node.state): # Then we have found the goal
    state
            return get_path(node) # Return the path that leads to the goal
```

```
        elif node.state not in closed:
            closed.append(node.state)
            fringe = expand_tree(node, fringe, problem, mode = "BFS")
    if fringe.isEmpty():
        print(f"- Search algorithm finished without reaching to a solution.
    ")

    util.raiseNotDefined()
```

Listing 2: Implementation of BFS

1.3 Συνάρτηση uniformCostSearch

Παρομοίως η υλοποίηση της αναζήτησης ομοιόρφου κόστους διαφέρει μόνο στη συνθήκη με την οποία επιλέγονται οι κόμβοι απ' το σύνορο για να επεκταθούν. Σε αυτή την περίπτωση διαλέγεται ο κόμβος με τη μικρότερη απόσταση από τον αρχικό κόμβο. Η δομή δεδομένων που χρησιμοποιείται σε αυτή την περίπτωση για το σύνορο είναι η ουρά προτεραιότητας όπου υψηλότερη προτεραιτότητα έχει ο κόμβος με το μικρότερο κόστος μονοπατιού απ' την αρχική κατάσταση. Η υλοποίηση του συνόρου γίνεται μέσω της συνάρτησης *PriorityQueue* που βρίσκεται στο αρχείο *util.py*. Παρακάτω βλέπουμε και τον κώδικα της υλοποίησης.

```
def uniformCostSearch(problem: SearchProblem):
    # My Iplementation of uniform-cost graph
    # UCS is implemented using a priorityQueue where the
    # node with the minimum cost path has the highest priority

    fringe = util.PriorityQueue() # Initialize an empty priority Queue
    closed = []
    starting_node = tree_node(state = problem.getStartState(),
                              ParentNode=None, Action=None, PathCost=0,
                              Depth=0) # Initialize the Starting node
    fringe.push(starting_node, starting_node.pathcost) # Append the
    starting node in queue
    while not fringe.isEmpty():
        node = fringe.pop()
        if problem.isGoalState(node.state): # Then we have found the goal
    state
            return get_path(node) # Return the path that leads to the goal
        elif node.state not in closed:
            closed.append(node.state)
            priority = node.pathcost
            fringe = expand_tree(node = node, fringe = fringe, problem =
    problem, mode = "UCS")
    if fringe.isEmpty():
```

```
        print(f"- Search algorithm finished without reaching to a solution.
    ")

    util.raiseNotDefined()
```

Listing 3: Implementation of UCS

1.4 Συνάρτηση aStarSearch

Στη περίπτωση του αλγορίθμου A^* ο κόμβος n που επιλέγεται για να εξαχθεί από το σύνορο βασίζεται στην ποσότητα $f(n) = g(n) + h(n)$, όπου $g(n)$ είναι το κόστος της διαδρομής από την αρχική κατάσταση μέχρι τον κόμβο n και $h(n)$ είναι μια ευρετική συνάρτηση που εκφράζει την εκτιμώμενη απόσταση μέχρι τον κόμβο στόχου. Γνωρίζουμε ότι στην περίπτωση που η h είναι παραδεκτή και ο παράγοντας διακλάδωσης b είναι πεπερασμένος τότε ο αλγόριθμος αναζήτησης A^* είναι πλήρης και βέλτιστος. Στη συγκεκριμένη υλοποίηση χρησιμοποιείται η απόσταση Manhattan ως ευρετική συνάρτηση όπως είναι υλοποιημένη στο αρχείο *util.py*. Η δομή δεδομένων που περιγράφει το σύνολο είναι μια ουρά προτεραίτητας *PriorityQueue* από το αρχείο *util.py* η οποία δίνει τη μεγαλύτερη προτεραιότητα στον κόμβο n με την ελάχιστη τιμή $f(n) = g(n) + h(n)$. Παρακάτω βλέπουμε και την αντίστοιχη υλοποίηση αυτής της αναζήτησης.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    # My implementation of aStar algorithm
    # We use a queue with function as a data structure
    fringe = util.PriorityQueue() # Initialize an empty priority Queue/L1
   distance
    closed = []
    starting_node = tree_node(state = problem.getStartState(),
                              ParentNode=None, Action=None, PathCost=0,
                              Depth=0) # Initialize the Starting node
    priority = starting_node.pathcost+heuristic(starting_node.state,
   problem)
    fringe.push(starting_node, priority) # Append the starting node in
   queue
    while not fringe.isEmpty():
        node = fringe.pop()
        if problem.isGoalState(node.state): # Then we have found the goal
   state
            return get_path(node) # Return the path that leads to the goal
        elif node.state not in closed:
            priority = node.pathcost + heuristic(node.state, problem) # The
    priority is the sum of pathcost + heuristic
            closed.append(node.state)
            fringe = expand_tree(node = node, fringe = fringe, problem =
   problem,
```

```
                                    mode = "aStar", heuristic=heuristic)
    if fringe.isEmpty():
        print(f"- Search algorithm finished without reaching to a solution.
    ")

    util.raiseNotDefined()
```

Listing 4: Implementation of A^*

2 Αρχείο searchAgents.py

2.1 Βρίσκοντας όλες τις γωνίες

Σε αυτή την υποπαράγραφο παρουσιάζουμε τη προσέγγιση για την επίλυση της 5ης και 6ης ερώτησης στο project1 του Pacman με τίτλο «Finding All the Corners» και «Corners Problem: Heuristic». Σκοπός σε αυτές τις ερωτήσεις είναι να μπορέσει ο Pacman να συλλέξει το φαγητό στις 4 γωνίες του εκάστοτε λαβίρυνθου. Στην ερώτηση 5 συμπληρώνοντας την κλάση *CornersProblem* στο αρχείο *searchAgents.py* ορίζουμε το πρόβλημα της εύρεσης των γωνιών ενώ συμπληρώνοντας τη συνάρτηση *cornersHeuristic* ορίζουμε μια συνεπή (και άρα παραδεκτή) συνάρτηση για την επίλυση του προβλήματος με χρήση του αλγορίθμου αναζήτησης A^*.

2.1.1 Υλοποίηση του CornersProblem

Οι ουσιαστικές διαφορές στην υλοποίηση του CornersProblem σε σχέση με το PositionSearchProblem, όπου σκόπος ήταν ο Pacman να βρεθεί σε ένα συγκεκριμένο σημείο του χάρτη, είναι ότι τώρα αλλάζει ο τελικός στόχος τον οποίο θέλουμε να επιβάλλουμε στον agent για να τερματίσει. Ξεκινάμε με την υλοποίηση του τελικού στόχου που είναι και το σημαντικότερο κομμάτι της υλοποίησης. Στο Σχήμα 5, έχοντας παραλείψει τα σχόλια και κάποιες εντολές της ομάδας του Berkley, βλέπουμε την υλοποίηση της μεθόδου *isGoalState* της κλάσης *CornersProblem*.

```
def isGoalState(self, state: Any):
    point = state[0] # The coordinates of current point state
    visited_corners = state[1] # List of visited corners
    if point in self.corners:
        if not point in visited_corners:
            visited_corners.append(point)
        return len(visited_corners) == 4 # Have we visited all corners?
    return False
```

Listing 5: Goal State of CornersProblem

Σε αντίθεση με την υλοποίηση του PositionSearchProblem, σε αυτή την περίπτωση το state αποτελείται από ένα tuple όπου στην 1η συνιστώσα περιέχονται οι συντεταγμένες του σημείου που βρίσκεται ο Pacman στον χάρτη και η 2η συνιστώσα αποτελείται από μια λίστα η οποία περιέχει τις γωνίες που έχει επισκεφθεί ο Pacman μέχρι εκείνο το σημείο. Η κατασκευή του tuple γίνεται μέσω της μεθόδου *getSuccessors* της κλάσης *PositionSearchProblem* της οποίας η υλοποίηση φαίνεται στο επόμενο σχήμα.

```
1  def getSuccessors(self, state: Any):
2      successors = []
3      for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
       Directions.WEST]:
4          x,y = state[0] # Point coordinates
5          visited_corners = state[1] # List of visited corners
6          dx, dy = Actions.directionToVector(action)
7          nextx, nexty = int(x + dx), int(y + dy)
8          if not self.walls[nextx][nexty]: # If not wall
9              successors_visited_corners = list(visited_corners) # Successor'
       s visited corner list
10             nextState = (nextx, nexty) # Point of nextState
11             """Check if nextState is corner and update"""
12             if nextState in self.corners and nextState not in
       visited_corners:
13                 successors_visited_corners.append(nextState)
14             cost = self.costFn(nextState)
15             successors.append(((nextState, successors_visited_corners),
       action, cost))
16
17     self._expanded += 1 # DO NOT CHANGE
18     return successors
```

Listing 6: getSuccessors method in CornersProblem

Έχοντας τη λίστα των γωνιών που έχει επισκεφθεί ο Pacman μέχρι εκείνο το σημείο είναι εύκολο τώρα να διατυπώσουμε τη συνθήκη τερματισμού της αναζήτησης, το μόνο που έχουμε να κάνουμε είναι να ελέγξουμε αν η λίστα αυτή έχει μήκος ίσο με 4. Η συνθήκη αυτή περιγράφεται μέσω της εντολής στη γραμμή 7 του Σχήματος 5. Όσον αφορά τον κώδικα του Σχήματος 6, η λειτουργία του είναι παρόμοια με την περίπτωση του PositionSearchProblem με μόνη διαφορά ότι κάθε φορά που υπολογίζουμε έναν νέο διάδοχο ελέγχουμε αν αυτός βρίσκεται σε κάποια απ' τις 4 γωνίες του λαβύρινθου, αν ναι, τότε τον προσθέτουμε στη λίστα των κόμβων που έχουμε επισκεφθεί μέχρι και τον κόμβο διάδοχο. Τέλος, η τελευταία ύπο-ρουτίνα της κλάσης CornersProblem που συμπληρώθηκε είναι η ρουτίνα που επιστρέφει την αρχική κατάσταση απ' την οποία ξεκινάει ο Pacman. Η υλοποίησή της γίνεται μέσω της μεθόδου *getStartState* που δίνεται στο επόμενο σχήμα.

```python
def getStartState(self):
    starting_visited_corners = []
    point = self.startingPosition
    if point in self.corners:
        starting_visited_corners.append(point)
    return point, starting_visited_corners
```

Listing 7: getStartState method in CornersProblem

Η μέθοδος *getStartState* αποθηκεύει τις συντεταγμένες του σημείου που βρίσκεται ο Pacman κατά την εκκίνηση και ελέγχει αν αυτό το σημείο ανήκει σε μια απ' τις 4 γωνίες του λαβυρίνθου, αν ναι, τότε το προσθέτει στη λίστα των γωνιών που έχουμε επισκεφθεί διαφορετικά επιστρέφεται η κενή λίστα. Έχοντας διατυπώσει και υλοποιήσει το πρόβλημα της εύρεσης των 4 γωνιών μπορούμε να το λύσουμε με χρήση της BFS αναζήτησης για παράδειγμα. Το γεγονός ότι όλες οι ακμές του δέντρου έχουν όλες κόστος ίσο με 1 μας εγγυάται ότι ο αλγόριθμος αναζήτησης πρώτα κατά πλάτος θα βρει τη βέλτιστη λύση. Παρακάτω βλέπουμε τα αποτελέσματα που παίρνουμε.

```
\$ python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 2448
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Όπως βλέπουμε με χρήση του BFS ο αλγόριθμος βρίσκει τη λύση επεκτείνοντας συνολικά 2448 κόμβους, με κόστος μονοπατιού 106 και σκορ ίσο με 434. Μπορούμε να επιταχύνουμε τη διαδικασία εύρεσης της λύσης με χρήση του DFS έχοντας χάνοντας τη βέλτιστη λύση, όπως βλέπουμε και παρακάτω.

```
\$ python pacman.py -l mediumCorners -p SearchAgent -a fn=dfs,prob=CornersProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 221 in 0.0 seconds
Search nodes expanded: 371
Pacman emerges victorious! Score: 319
Average Score: 319.0
Scores:        319.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2.1.2 Βρίσκοντας τις γωνίες με χρήση του A^*

Όπως βλέπουμε και απ' τα παραπάνω αποτελέσματα η αναζήτηση πρώτα σε πλάτος βρίσκει το βέλτιστο μονοπάτι επεκτείνοντας 2448 κόμβους συνολικά, σε αυτό το σημείο χρησιμοποιούμε τον αλγόριθμο A^* με κατάλληλη ευρετική συνάρτηση για να μειώσουμε τον αριθμό αυτών των κόμβων. Πριν προχωρήσουμε στην υλοποίηση της συνάρτησης *cornersHeuristic* δίνουμε τον ορισμό της ευρετικής συνάρτησης που χρησιμοποίηθηκε και αποδεικνύουμε ότι είναι συνεπής. Προς τούτο, συμβολίζουμε με $C = \{c_1, c_2, c_3, c_4\}$ τις 4 γωνίες του λαβυρίνθου. Έστω ότι κάποια στιγμή βρισκόμαστε στον κόμβο n. Ας υποθέσουμε χωρίς βλάβη της γενικότητας ότι οι γωνίες που μας απομένουν να μας επισκεφθούμε είναι 3 και ότι αυτές, πάλι χωρίς βλάβη της γενικότητας, είναι οι c_2, c_3, c_4 (ομοίως ορίζεται και η συνάρτηση στις άλλες περιπτώσεις). Τότε, για να ορίσουμε την h κοιτάμε ποια γωνία είναι πλησιέστερα στον κόμβο n ως προς την απόσταση Manhattan. Δηλαδή, θεωρούμε την γωνία $n_1 = \arg\min_{x \in \{c_2, c_3, c_4\}} d(x, n)$, όπου d η απόσταση Manhattan. Ύστερα θεωρούμε τη γωνία που απομένει να επισκεφθούμε και βρίσκεται πλησιέστερα στον κόμβο n_1, δηλαδή $n_2 = \arg\min_{x \in \{c_2, c_3, c_4\} \setminus \{n_1\}} d(n_1, x)$. Τέλος, θεωρούμε και $n_3 = \arg\min_{x \in \{c_2, c_3, c_4\} \setminus \{n_1, n_2\}} d(x, n_2)$. Τότε, ορίζουμε την h μέσω της

$$h(n) = d(n, n_1) + d(n_1, n_2) + d(n_2, n_3). \tag{2.1}$$

Περιγραφικά, η h είναι η απόσταση Manhattan που πρέπει να διανύσουμε για να πάμε από τον κόμβο στον οποίο βρισκόμαστε μέχρι την πλησιέστερη γωνία, ύστερα από αυτή τη γωνία μέχρι την επόμενη πλησιέστερη γωνία κ.ο.κ. μέχρις ώτου να περάσουμε από όλες τις εναπομείνασες γωνίες.

Η h είναι συνεπής. Το ότι η h είναι συνεπής προκύπτει απ'την απλή παρατήρηση ότι για κάθε διάδοχο s ενός κόμβου n ισχύει ότι $d(n, s) = 1$. Δηλαδή, η απόσταση Manhattan είναι 1 όταν οι δύο εμπλεκόμενοι κόμβοι είναι γειτονικοί. Πράγματι, για να το δούμε αυτο θεωρούμε ότι ο κόμβος n είναι το σημείο (x, y) του λαβύρινθου όπου $x, y \in \mathbb{Z}$. Τότε, η επόμενη κίνηση του Pacman είναι μια απ' τις North, South, West, East. Οι κινήσεις αυτές μας μεταφέρουν στα σημεία $(x, y+1), (x, y-1), (x-1, y), (x+1, y)$ αντιστοίχως. Εύκολα βλέπουμε ότι η απόσταση Manhattan μεταξύ του (x, y) και καθενός απ' τα παραπάνω 4 σημεία είναι 1. Πράγματι, ενδεικτικά υπολογίζοντας ένα ζευγάρι από αυτά βρίσκουμε ότι

$$d\left((x, y), (x, y+1)\right) = |x - x| + |y - (y+1)| = 1.$$

Τώρα, για να δείξουμε ότι η h είναι συνεπής θα πρέπει να δείξουμε ότι αν n' είναι ένας διάδοχος του n τότε θα πρέπει να ισχύει $h(n) \leq c(n, \alpha, n') + h(n')$, όπου με α συμβολίζουμε την ενέργεια που μεταφέρει τον Pacman στον κόμβο n', δηλαδή $\alpha \in \{\text{North, South, West, East}\}$. Τώρα, εφόσον όλες οι ακμές έχουν κόστος ίσο με 1 θα έχουμε ότι $c(n, \alpha, n') = 1$. Τώρα, εφόσον ο κόμβος n' είναι γειτονικός με τον n θα έχουμε $d(n, n') = 1$. Έτσι, υπάρχουν δύο περιπτώσεις: 1) είτε ο κόμβος n' θα απομακρυνθεί κατά απόσταση Manhattan 1 από

την πλησιέστερη γωνία είτε 2) θα πλησιάσει κατά 1 προς την πλησιέστερη γωνία. Στη μια περίπτωση θα έχουμε ότι $h(n') = h(n) + 1$ το οποίο μας δίνει ότι

$$h(n) = h(n') - 1 \leq h(n') \leq h(n') + c(n, \alpha, n'),$$

ενώ στην άλλη περίπτωση θα έχουμε ότι $h(n) = h(n') + 1$ το μας δίνει ότι

$$h(n) = h(n') + 1 = h(n') + c(n, \alpha, n').$$

Σε κάθε περίπτωση βλέπουμε ότι ικανοποιείται η (2.1), πράγμα το οποίο δείχνει ότι η h είναι συνεπής. Τέλος, αφού η h είναι συνεπής θα είναι και παραδεκτή. □

Παρακάτω βλέπουμε την υλοποίηση της συνάρτησης *cornersHeuristic* και τα αποτελέσματα που παίρνουμε επιλύοντας τον λαβύρινθο μεσαίου μεγέθους χρησιμοποιώντας τον A^* με ευρετική συνάρτηση την h.

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    current_point = state[0] # Coordinates of pacman's current state
    visited_corners = state[1] # The corners that pacman has visited
    total_cost = 0
    left_to_visit = [] # Which corners are left to visit
    for corner in corners:
        if not corner in visited_corners:
            left_to_visit.append(corner)
    while left_to_visit:
        """ Get the manhattan distance of the whole path cost
                until all left has been visited. """
        pair = min([(util.manhattanDistance(current_point,next_corner),
   next_corner) for next_corner in left_to_visit])
        current_point = pair[1]
        total_cost += pair[0]
        left_to_visit.remove(current_point)
    # return 0 # Default to trivial solution
    return total_cost
```

Listing 8: Heuristic function for CornersProblem

```
\$ python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 901
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Όπως βλέπουμε σε αυτή την περίπτωση ο αλγόριθμος A^* βρίσκει τη βέλτιστη λύση επεκτείνοντας 901 κόμβους σε αντίθεση με τον BFS που επεκτείνει 2448.

2.2 Συλλέγοντας όλο το φαγητό

2.2.1 Η ευρετική συνάρτηση foodHeuristic

Σε αυτό το σημείο υλοποιούμε μια συνεπή ευρετική συνάρτηση για το πρόβλημα εύρεσης όλου του φαγητού το οποίο ορίζεται μέσω της κλάσης *FoodSearchProblem* στο αρχείο *searchAgents.py*. Η ευρετική συνάρτηση ορίζεται στη μέθοδο *foodHeuristic* στο προαναφερθέν αρχείο. Όπως και στην περίπτωση της ευρετικής συνάρτησης για το πρόβλημα των 4 γωνιών πρώτα δίνουμε τον μαθηματικό ορισμό της συνάρτησης και αποδεικνύουμε ότι είναι συνεπής και εν συνεχεία παρουσιάζουμε την υλοποίηση και τα αποτελέσματα. Η ευρετική συνάρτηση ορίζεται ως εξής: Θεωρούμε ότι μια δεδομένη χρονική στιγμή βρισκόμαστε στον κόμβο n του οποίου γνωρίζουμε τις συντεταγμένες $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ στον λαβύρινθο καθώς και τη λίστα L_n των σημείων που περιέχουν το φαγητό που απομένει να συλλεχθεί. Ορίζουμε την ευρετική συνάρτηση h μέσω της

$$h(n) = \begin{cases} \max\limits_{x \in L_n \setminus \{n\}} \text{MazeDistance}(n, x), & L_n \setminus \{n\} \neq \emptyset \\ 0 \qquad\qquad\qquad\qquad\quad , & \text{διαφορετικά} \end{cases} . \tag{2.2}$$

Η h είναι συνεπής. Όπως και στην περίπτωση της ευρετικής συναρτησης για πρόβλημα των γωνιών για να δείξουμε ότι η h είναι συνεπής πρέπει να δείξουμε ότι για κάθε κόμβο n και κάθε διάδοχό του n' στον οποίο καταλήγουμε μέσω της ενέργειας α ισχύει ότι

$$h(n) \leq c(n, \alpha, n') + h(n'). \tag{2.3}$$

Αφού το κόστος κάθε ακμής του δένδρου είναι 1 θα έχουμε ότι $c(n, \alpha, n') = 1$ για κάθε επιτρεπτή τριάδα (n, α, n'). Θεωρούμε έναν κόμβο n' διάδοχο του n. Τότε, εφόσον απ' τον έναν στον άλλο κόμβο πάμε σε ένα βήμα θα ισχύει ότι η Maze απόστασή τους από ένα σταθερό σημείο k στον λαβύρινθο θα διαφέρει κατά 1. Δηλαδή,

$$|\text{MazeDistance}(n, k) - \text{MazeDistance}(n', k)| = 1. \tag{2.4}$$

Υποθέτουμε πρώτα ότι $L_n \setminus \{n\} \neq \emptyset$ και $L_{n'} \setminus \{n'\} \neq \emptyset$. Θεωρούμε τα σημεία

$$n_f = \arg\max_{x \in L_n \setminus \{n\}} \text{MazeDistance}(x, n)$$

και

$$n'_f = \arg\max_{x \in L_{n'} \setminus \{n'\}} \text{MazeDistance}(x, n').$$

Τότε, εξ' ορισμού της h στη σχέση (2.2) θα έχουμε ότι $h(n) = \text{MazeDistance}(n, n_f)$ και $h(n') = \text{MazeDistance}(n', n'_f)$. Αν $n_f = n'_f$ τότε απ' τη (2.4) θα έχουμε ότι

$$\begin{aligned} h(n) &= \text{MazeDistance}(n, n_f) \\ &\leq \text{MazeDistance}(n', n'_f) + 1 \\ &= h(n') + c(n, \alpha, n'). \end{aligned}$$

Δηλαδή, η (2.3) ισχύει σε αυτή την περίπτωση. Αν τώρα απ' την άλλη ισχύει $n_f \neq n'_f$ τότε αυτό σημαίνει ότι το n'_f απέχει περισσότερο απ' το n' απ'ότι το n_f απ'το n'. Δηλαδή, ισχύει ότι $\text{MazeDistance}(n'_f, n') > \text{MazeDistance}(n_f, n')$. Χρησιμοποιώντας αυτή την ανισότητα και την (2.4) για τα ζευγάρια $(n', n_f), (n', n'_f)$ καταλήγουμε στο ότι

$$\text{MazeDistance}(n', n_f) = \text{MazeDistance}(n', n'_f) - 1. \tag{2.5}$$

Τώρα, πάλι απ' την (2.4) θα έχουμε ότι

$$\begin{aligned}\text{MazeDistance}(n, n_f) &= \text{MazeDistance}(n, n_f) - \text{MazeDistance}(n', n_f) + \text{MazeDistance}(n', n_f) \\ &\leq 1 + \text{MazeDistance}(n', n_f).\end{aligned}$$

Επομένως, απ' την (2.5) θα έχουμε

$$\begin{aligned}h(n) &= \text{MazeDistance}(n, n_f) \leq 1 + \text{MazeDistance}(n', n_f) \\ &= 1 + \text{MazeDistance}(n', n'_f) - 1 = \text{MazeDistance}(n', n'_f) = h(n') \\ &\leq h(n') + c(n, \alpha, n'),\end{aligned}$$

και έτσι η (2.3) ισχύει και σε αυτή την περίπτωση. Τέλος, στην περίπτωση που ισχύει $L_n = \{n\}$ είτε $L_{n'} = \{n'\}$ τότε είναι εύκολο να δούμε ότι η (2.3) ικανοποιείται πάλι. Πράγματι, ας υποθέσουμε χ.β.γ. ότι $L_n = \{n\}$. Τότε, θα ισχύει ότι $h(n) = 0$ το οποίο μας δίνει τετριμμένα ότι $h(n) \leq c(n, \alpha, n') + h(n')$, αφού $c(n, \alpha, n') = 1$. Αν απ' την άλλη ισχύει ότι $L_{n'} = \{n'\}$ τότε θα έχουμε ότι $h(n') = 0$. Αυτό σημαίνει ότι στον κόμβο n' βρίσκεται το τελευταίο σημείο στον λαβύρινθο που περιέχει φαγητό. Έτσι, θα έχουμε ότι $h(n) = 1$ το οποίο μας δίνει πάλι ότι $h(n) \leq c(n, \alpha, n') + h(n')$. Άρα, η (2.3) ισχύει σε όλες τις περιπτώσεις απ' όπου έπεται ότι είναι συνεπής. □

Παρακάτω βλέπουμε και την υλοποίηση της συνάρτησης *foodHeuristic* στο αρχείο *searchAgents.py* και τα αποτελέσματα που παίρνουμε στον autograder για το 7ο ερώτημα.

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem:
    FoodSearchProblem):
    position, foodGrid = state
    food_left = foodGrid.asList()
    gameState = problem.startingGameState
    if position in food_left:
        food_left.remove(position)
    if food_left:
        d = max([mazeDistance(position,point,gameState) for point in
    food_left])
        return d
    else:
        return 0
```

Listing 9: Heuristic function for FoodProblem

```
python autograder.py -q q7
Note: due to dependencies, the following tests will be run: q4 q7
Starting on 10-20 at 13:04:38

Question q4
===========
*** PASS: test_cases/q4/astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###


Question q7
===========
*** PASS: test_cases/q7/food_heuristic_1.test
*** PASS: test_cases/q7/food_heuristic_10.test
*** PASS: test_cases/q7/food_heuristic_11.test
*** PASS: test_cases/q7/food_heuristic_12.test
*** PASS: test_cases/q7/food_heuristic_13.test
*** PASS: test_cases/q7/food_heuristic_14.test
*** PASS: test_cases/q7/food_heuristic_15.test
*** PASS: test_cases/q7/food_heuristic_16.test
*** PASS: test_cases/q7/food_heuristic_17.test
*** PASS: test_cases/q7/food_heuristic_2.test
*** PASS: test_cases/q7/food_heuristic_3.test
*** PASS: test_cases/q7/food_heuristic_4.test
*** PASS: test_cases/q7/food_heuristic_5.test
*** PASS: test_cases/q7/food_heuristic_6.test
*** PASS: test_cases/q7/food_heuristic_7.test
*** PASS: test_cases/q7/food_heuristic_8.test
```

```
*** PASS: test_cases/q7/food_heuristic_9.test
*** PASS: test_cases/q7/food_heuristic_grade_tricky.test
***     expanded nodes: 4137
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 5/4 ###


Finished at 13:05:31

Provisional grades
==================
Question q4: 3/3
Question q7: 5/4
------------------
Total: 8/7
```

2.3 Suboptimal αναζήτηση

Για το 8ο ερώτημα του project1 συμπληρώνεται η μέθοδος *isGoalState* της κλάσης *AnyFoodSearchProblem* και η μέθοδος *findPathToClosestDot* της κλάσης *ClosestDotSearchAgent* στο αρχείο *searchAgents.py*. Σκοπός σε αυτό το ερώτημα είναι κατασκευαστεί ένα μονοπάτι το οποίο επιτυγχάνει τον στόχο που είναι περάσει απ' όλα τα σημεία που περιέχουν φαγητό χωρίς απαραίτητα να είναι το βέλτιστο δυνατό. Ο λόγος γιάυτό είναι για να μειωθεί ο χρόνος εκτέλεσης της αναζήτησης. Η προσέγγιση που ακολουθείται είναι μια greedy προσέγγιση κατά την οποία ο Pacman επιλέγει να κινηθεί πάντα προς το πλησιέστερο σημείο που περιέχει φαγητό κατά την απόσταση Manhattan. Έτσι, για τη μέθοδο *isGoalState* ο στόχος είναι ο Pacman να βρεθεί στο πλησιέστερο σημείο που υπάρχει φαγητό. Αυτό γίνεται διατρέχοντας τα στοιχεία της λίστας των σημειών που περιέχουν φαγητό και βρίσκουμε αυτό με την ελάχιστη απόσταση, αν αυτό συμπίπτει με το current state επιστρέφεται η τιμή True αλλιώς επιστρέφεται η τιμή False. Αυτά περιγράφονται απ' τις γραμμές 8-9 του παρακάτω κώδικα.

```
1 def isGoalState(self, state: Tuple[int, int]):
2     """
3     The state is Pacman's position. Fill this in with a goal test that will
4     complete the problem definition.
5     """
6     x,y = state
7     "*** YOUR CODE HERE ***"
8     _,goal = min([(util.manhattanDistance(state,goal), goal) for goal in
      self.food.asList()])
9     return True if state == goal else False
```

```
10      util.raiseNotDefined()
```

Listing 10: AnyFoodSearchProblem Goal

Τώρα, είναι εύκολο να βρούμε το μονοπάτι προς το πλησιέστερο σημείο φαγητού μέσω της συνάρτησης *findPathToClosestDot*. Αυτό που έχουμε να κάνουμε είναι να ορίσουμε το πρόβλημα προς την το σημείο αυτό και να εκτελέσουμε τον αλγόριθμο BFS. Αυτό φαίνεται και στο παρακάτω block κώδικα.

```
1 def findPathToClosestDot(self, gameState: pacman.GameState):
2     # Here are some useful elements of the startState
3     startPosition = gameState.getPacmanPosition()
4     food = gameState.getFood()
5     walls = gameState.getWalls()
6     problem = AnyFoodSearchProblem(gameState)
7     return search.breadthFirstSearch(problem=problem)
8     util.raiseNotDefined()
```

Listing 11: findPathToClosestDot