Pacman: Project 1

Search

Κωνσταντίνος Χούσος

Περίληψη

Το συγκεκριμένο report αναλύει τις λύσεις μου στο project 1 των Pac-Man Projects του Berkeley [1], στα πλαίσια του προπτυχιακού μαθήματος "Τεχνητή Νοημοσύνη". Τα ερωτήματα 1–4 υλοποιήθηκαν με τον ίδιο γενικής μορφής αλγόριθμο για αναζήτηση σε γράφους, το καθένα με το σύνορο σε διαφορετική δομή δεδομένων. Στο ερώτημα 5 το state αποτελείται από ένα tuple με τις συντεταγμένες του pacman και τις γωνίες που δεν έχει επισκεφθεί ακόμα. Στο ερώτημα 6 η ευρετική επιστρέφει το συνολικό κόστος για να πάμε σε κάθε γωνία πηγαίνοντας κάθε φορά στην πιο κοντινή. Στο ερώτημα 7 επιστρέφεται η απόσταση για το πιο κοντινό φαγητό συν την μέγιστη απόσταση για κάποιο άλλο από την θέση του πρώτου. Η υλοποίηση εκμεταλλεύεται το **heuristicinfo** dictionary και την συνάρτηση **mazeDistance**—κάνοντας expand 719 nodes και με βαθμολογία 5/4. Το ερώτημα 8 υλοποιείται με βάση τα hint της εκφώνησης, προσθέτοντας μόνο μια γραμμή κώδικα στη συνάρτηση **isGoalState** της κλάσης **AnyFoodSearchProblem** και στη συνάρτηση **findPathToClosestDot** της κλάσης **ClosestDotSearchAgent**.

1–4: BFS, DFS, UCS, A*

Τα πρώτα τέσσερα ερωτήματα έχουν υλοποιηθεί με βάση τον ίδιο αλγόριθμο **GRAPH_SEARCH**, εμπνευσμένο από την παρακάτω διαφάνεια του φροντιστηρίου, που φαίνεται στο σχήμα 1.

```
Algorithm: GRAPH_SEARCH:
 frontier = {startNode}
 expanded = {}
 while frontier is not empty:
     node = frontier.pop()
    if isGoal(node):
          return path_to_node
    if node not in expanded:
          expanded.add(node)
          for each child of node's children:
               frontier.push(child)
 return failure
```

Σχήμα 1: Αλγόριθμος Graph Search

Σε κάθε ερώτημα το μόνο που αλλάζει είναι η δομή δεδομένων που χρησιμοποιείται για την υλοποίηση του συνόρου (fringe). Πιο αναλυτικά έχουμε: **Stack** για τον DFS, **Queue** για τον BFS και **Priority Queue** για τους UCS και A*. Οι συγκεκριμένες δομές υπήρχαν ήδη υλοποιημένες στο αρχείο **util.py**.

Με λίγα λόγια, η λειτουργία του αλγορίθμου είναι η εξής: Ξεκινώντας από το **StartNode**, ελέγχουμε αν το σύνορο είναι άδειο. Αν δεν είναι, κάνουμε **pop** από το σύνορο (το ποιο node θα μας δώσει εξαρτάται από το τι δομή έχουμε χρησιμοποιήσει για την υλοποίησή της και είναι αυτό που διαφέρει ανάμεσα στους αλγορίθμους). Αν το node που πήραμε είναι ο στόχος, τότε επιστρέφουμε το **path**.

Η διαφοροποίηση ανάμεσα στις υλοποιήσεις των DFS και BFS με αυτές των UCS και A* είναι ότι στις τελευταίες δύο παίρνουμε υπόψιν και το κόστος κάθε κόμβου(στον A* για να υπολογίσουμε την τιμή της ευρετικής). Παρά όλα αυτά, ο γενικός αλγόριθμος παραμένει ίδιος.

Στον πίνακα 1 φαίνεται η σύγκριση κι η επίδοση των τεσσάρων αλγορίθμων για το **mediumMaze**. Ο αλγόριθμος A* εξετάστηκε με την **manhattanHeuristic** ευρετική.

Πίνακας 1: Σύγκριση DFS, BFS, UCS, A*

Αλγόριθμος	Κόστος	Exp. Nodes	Score
DFS	130	146	380
BFS	68	269	442
UCS	68	269	442
A*	68	221	442

Άρα διαπιστώνουμε ότι από τους τέσσερις αλγορίθμους ο DFS έχει την χειρότερη απόδοση, ενώ οι άλλοι τρεις έχουν ίδια αποτελέσματα, με τον A* να είναι πιο αποδοτικός λόγω της ευρετικής. Αυτό είναι αναμενόμενο, καθώς επαληθεύεται κι από την θεωρία.

5: CORNERS PROBLEM

Στη συγκεκριμένη υλοποίηση ένα **state** μοντελοποιείται από ένα tuple όπου το πρώτο στοιχείο είναι οι συντεταγμένες του pacman και το δεύτερο ένα tuple με τις γωνίες του λαβυρίνθου που ο pacman δεν έχει επισκεφθεί ακόμα. Οι γωνίες αναπαρίστανται με την ίδια σειρά του **self.walls**. Έτσι, αρκεί να αφαιρούμε κάθε φορά από το tuple **state[1]** την γωνία που επισκεπτόμαστε, και να ελέγχουμε αν παραμένουν γωνίες που δεν έχουμε επισκεφθεί για το αν φτάσαμε σε goal state.

Στην υλοποίηση της συνάρτησης **getSuccessors** ακολουθούμε κάνουμε τους εξής ελέγχους/βήματα για κάθε επόμενη συντεταγμένη **x, y**:

1. Πρόκειται για τοίχο; Αν όχι, συνέχισε.

2. Αντιγράφουμε το **state**—συγκεκριμένα τους τοίχους που απομένουν—σε ένα καινούργιο **cornerState**. Στο συγκεκριμένο βήμα είναι αναγκαίο να κάνουμε cast σε λίστα το tuple του **state** έτσι ώστε να μπορούμε αργότερα να το αλλάξουμε.

3. Αν οι συντεταγμένες πρόκειται για γωνία που δεν έχουμε επισκεφθεί, σημαίνει ότι πλέον την επισκεφθήκαμε. Άρα, την βγάζουμε από την λίστα **cornerState**.

4. Ανεξάρτητα από το αν την αλλάξαμε πρέπει να κάνουμε την **cornerState** cast σε tuple, έτσι ώστε να μπορούμε να δημιουργήσουμε με αυτήν ένα καινούργιο successor state.

5. Προσθέτουμε στα successor states το τωρινό state.

Στον πίνακα 2 φαίνεται η επίδοση του BFS στο πρόβλημα cornersProblem για καθέναν από τους τρεις λαβυρίνθους **tinyCorners**, **mediumCorners** και **bigCorners**.

Πίνακας 2: Απόδοση BFS στο corners

Λαβύρινθος	Κόστος	Exp. Nodes	Score
tinyCorners	28	252	512
mediumCorners	106	1966	434
bigCorners	162	7949	378

6: CORNERS PROBLEM: HEURISTICS

Ο αλγόριθμος της ευρετικής συνάρτησης στο ερώτημα 6 μπορεί να περιγραφεί με τα ακόλουθα βήματα.

- Όσο υπάρχουν γωνίες που δεν έχουμε επισκεφθεί:
  - Βρες την πιο κοντινή γωνία
  - Πρόσθεσε την απόστασή της στο αποτέλεσμα
  - Πλέον είσαι σε αυτή τη γωνία
- Επέστρεψε το αποτέλεσμα

Για το συγκεκριμένο ερώτημα, αγνοούμε την ύπαρξη των τοίχων, ως χαλάρωση του προβλήματος. Άρα η απόσταση προς μία γωνία είναι απλά το **manhattanDistance** προς

αυτή. Στην ουσία, η συνάρτηση επιστρέφει το ελάχιστο κόστος που μπορούμε να έχουμε για να φτάσουμε σε ένα κόμβο στόχου—δηλαδή στην τελευταία γωνία έχοντας επισκεφθεί όλες τις άλλες—για το εκάστοτε state. Έτσι, ο **AStarCornersAgent** κάνει expand μόλις 692 nodes.

Στον πίνακα 3 φαίνεται η επίδοση του A* στο πρόβλημα cornersProblem για καθέναν από τους τρεις λαβυρίνθους **tinyCorners**, **mediumCorners** και **bigCorners**.

Πίνακας 3: Απόδοση A* στο corners

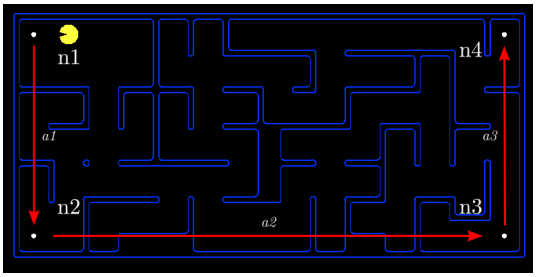
Λαβύρινθος	Κόστος	Exp. Nodes	Score
tinyCorners	28	154	512
mediumCorners	106	692	434
bigCorners	162	1725	378

Βλέποντας τα expanded nodes επαληθεύεται η καλύτερη αποδοτικότητα του A*.

Απόδειξη συνέπειας

Έστω η συνάρτηση **cornersHeuristic** = $h(n)$, όπου n κόμβος του προβλήματος. θέτουμε τις γωνίες ως n_1, n_2, n_3, n_4 και τις ενέργειες για να πάμε από έναν κόμβο στον κοντινότερο a_1, a_2, a_3 αντίστοιχα. θέτουμε και μια συνάρτηση $c(n, a, n')$ όπου περιγραφεί το κόστος της ενέργειας a για να πάμε από τον n στον n'.

Έστω ότι ξεκινάμε στον n_1. $h(n_1) = c(n_1, a_1, n_2) + c(n_2, a_2, n_3) + c(n_3, a_3, n_4)$. $h(n_2) = c(n_2, a_2, n_3) + c(n_3, a_3, n_4)$. Από τα παραπάνω έχουμε: $h(n_1) = c(n_1, a_1, n_2) + h(n_2)$, το οποίο επεκτείνεται για οποιοδήποτε ζευγάρι n, n'. Άρα η συνάρτηση **cornersHeuristic** είναι συνεπής, άρα και παραδεκτή. □



7: EATING ALL THE DOTS

Η ευρετική του ερωτήματος 7 ακολουθεί παρόμοια λογική με αυτή του ερωτήματος 6, με διαφορά ότι τώρα έχουμε αντί για γωνίες φαγητό. Μια άλλη σημαντική διαφορά είναι ότι αυτήν τη φορά εκμεταλλευόμαστε το **heuristicInfo** dictionary που προτείνεται από τα σχόλια. Αντί για την **manhattanDistance**, χρησιμοποιείται μια καινούργια συνάρτηση, η **heuristicInfoDistance**. Η δεύτερη ελέγχει πρώτα αν η απόσταση ανάμεσα σε δύο σημεία έχει ήδη υπολογιστεί (κι άρα αποθηκευτεί στο **heuristicInfo**). Αν ναι, την επιστρέφει. Αν όχι, τότε την υπολογίζει με την βοήθεια της **mazeDistance**, την αποθηκεύει στο **heuristicInfo** κι ύστερα την επιστρέφει.

Σε αυτήν την ευρετική, βρίσκουμε την απόσταση μεταξύ του pacman και του κοντινότερου φαγητού συν την απόσταση μεταξύ αυτού του κοντινότερου φαγητού και το μακρινότερο φαγητό από αυτό. Η συνέπειά της αποδεικνύεται με παρόμοιο τρόπο με την **cornersHeuristic**.

Στον πίνακα 4 φαίνεται η επίδοση του A* στο πρόβλημα foodSearchProblem για καθέναν από τους δύο λαβυρίνθους **testSearch** και **trickySearch**.

Πίνακας 4: Απόδοση A* στο foodSearch

Λαβύρινθος	Κόστος	Exp. Nodes	Score
testSearch	7	7	513
trickySearch	60	719	570

Λόγω του ότι η **foodHeuristic** κάνει expand 719 nodes στον **trickySearch** λαβύρινθο, η βαθμολογία του ερωτήματος 7 είναι 5/4.

8: SUBOPTIMAL SEARCH

Το goal state του anyFoodSearchProblem είναι απλά το να βρισκόμαστε σε θέση φαγητού. Η εύρεση της κοντινότερης τελείας γίνεται τετριμμένη εάν χρησιμοποιήσουμε έναν από τους αλγορίθμους αναζήτησης που

υλοποιήσαμε στα προηγούμενα ερωτήματα. Στην υλοποίηση επιλέχθηκε ο A*.

Στον πίνακα 5 φαίνεται η σύγκριση κι η επίδοση των τεσσάρων αλγορίθμων για το **bigSearch**. Ο αλγόριθμος A* εξετάστηκε με την **nullHeuristic** ευρετική.

Πίνακας 5: Σύγκριση DFS, BFS, UCS, A*

Αλγόριθμος	Κόστος	Score
DFS	5324	-2614
BFS	350	2360
UCS	350	2360
A*	350	2360

Πάλι, όπως και στην σύγκριση των αλγορίθμων για το **mediumMaze**, ο DFS βγαίνει τελευταίος με πολύ χειρότερη επίδοση από τους υπόλοιπους. Οι BFS, UCS και A*, έχουν τα ίδια αποτελέσματα, κάτι που είναι αναμενόμενο. Προφανώς μια αναζήτηση τέτοιου είδους δεν είναι βέλτιστη, όπως παρατηρείται κι από την εκφώνηση, λόγω του ότι ο λαβύρινθος στην συγκεκριμένη περίπτωση είναι πυκνός όσον αφορά τις τελείες.

Αναφορές

[1] John DeNero, Dan Klein και Pieter Abbeel. *Projects - CS 188: Introduction to Artificial Intelligence, Spring 2022*. 2022. URL: https://inst.eecs.berkeley.edu/~cs188/sp22/projects/.