Τεχνητή Νοημοσύνη

3η Άσκηση

Δεμερτζόγλου Ευστράτιος | TH20580

Table of Contents

Περιγραφή του Προβλήματος 3
Προσέγγιση Επίλυσης 3
Επεξήση Κώδικα 4
Παράδειγμα Εκτέλεσης 5
Κώδικας σε Python που εκτελέστηκε 5
Συμπέρασμα 8

Περιγραφή του Προβλήματος

Το πρόβλημα αφορά την εύρεση βέλτιστης διαδρομής σε έναν σταθμισμένο γράφο με 21 κόμβους (A έως U). Ζητείται η εύρεση διαδρομών από τις κορυφές A, B και C προς την κορυφή F, χρησιμοποιώντας δύο διαφορετικούς αλγορίθμους αναζήτησης: τον Greedy Best First Search και τον A*. Κάθε κόμβος του γράφου έχει μια ευρετική τιμή (εντός αγκυλών) που αντιπροσωπεύει την εκτιμώμενη απόσταση από τον τελικό προορισμό F. Οι ακμές του γράφου έχουν βάρη που αντιπροσωπεύουν το πραγματικό κόστος μετάβασης μεταξύ των κόμβων.

Προσέγγιση Επίλυσης

Η επίλυση του προβλήματος πραγματοποιήθηκε με την υλοποίηση δύο διαφορετικών αλγορίθμων:

1. Greedy Best First Search:

- Χρησιμοποιεί μόνο την ευρετική τιμή h(n) για την επιλογή του επόμενου κόμβου
- Επιλέγει πάντα τον κόμβο με τη μικρότερη ευρετική τιμή
- Είναι πιο γρήγορος αλλά δεν εγγυάται τη βέλτιστη λύση

2. A* (A-Star):

- Συνδυάζει το πραγματικό κόστος διαδρομής g(n) με την ευρετική τιμή h(n)
- Χρησιμοποιεί τη συνάρτηση f(n) = g(n) + h(n)
- Εγγυάται την εύρεση της βέλτιστης διαδρομής εφόσον η ευρετική είναι παραδεκτή

Επεξήση Κώδικα

Ο κώδικας υλοποιήθηκε σε Python χρησιμοποιώντας τις εξής βασικές δομές:

1. Αναπαράσταση Γράφου:

- Dictionary για τον γράφο όπου κάθε κλειδί είναι ένας κόμβος και η τιμή του είναι ένα dictionary με τους γείτονές του και τα αντίστοιχα βάρη
- Dictionary για τις ευρετικές τιμές κάθε κόμβου

2. Υλοποίηση Αλγορίθμων:

- Χρήση PriorityQueue για την επιλογή του επόμενου κόμβου προς εξερεύνηση
- Σύνολο visited για την αποφυγή κύκλων
- Παρακολούθηση κόστους και μονοπατιού για κάθε διαδρομή

Βασικές Συναρτήσεις:

- has_on_top(state, x): Ελέγχει αν υπάρχει άλλο μπλοκ πάνω στο x.
- successors(state): Επιστρέφει όλες τις έγκυρες μετακινήσεις από την παρούσα κατάσταση.
- is_goal(state): Ελέγχει αν η κατάσταση είναι η τελική.
- bfs(start): Υλοποίηση του BFS για εύρεση της συντομότερης λύσης.
- dfs(start): Υλοποίηση DFS για εύρεση κάποιας λύσης.

Παράδειγμα Εκτέλεσης

```
Greedy Best First Search:
Path from A to F: A -> R -> K -> F
Cost: 175

Path from B to F: B -> S -> H -> F
Cost: 135

Path from C to F: C -> I -> E -> N -> F
Cost: 165

A* Search:
Path from A to F: A -> D -> G -> F
Cost: 120

Path from B to F: B -> P -> S -> N -> F
Cost: 85

Path from C to F: C -> Q -> K -> O -> F
Cost: 100
```

Κώδικας σε Python που εκτελέστηκε

```python
from queue import PriorityQueue

# Ορισμός του γράφου ως λεξικό γειτνίασης με βάρη
# Κάθε κλειδί είναι ένας κόμβος και η τιμή του είναι ένα λεξικό με τους γείτονές του και τα αντίστοιχα βάρη
graph = {
    'A': {'D': 20, 'R': 55},
    'B': {'S': 50, 'P': 20, 'L': 10},
    'C': {'J': 15,'Q': 30, 'I' : 80},
    'D': {'G': 45, 'R': 40, 'A': 20},
    'E': {'I': 30, 'N': 30, 'O' : 30 },
    'F': {'T': 35, 'N': 25, 'O': 30},
    'G': {'T': 20, 'D': 45, 'F': 55},
    'H': {'F': 40, 'S': 45, 'U': 40, 'T': 20},
    'I': {'P': 15, 'C': 80, 'E': 30},
    'J': {'C': 15, 'M': 20},
    'K': {'M': 40, 'Q': 30, 'O': 5, 'F': 45},
    'L': {'T': 40, 'B': 10},
    'M': {'Q': 85, 'J': 20, 'K': 40},
    'N': {'F': 25, 'E': 30, 'S': 30},
    'O': {'F': 35, 'K': 5, 'E' : 30},
    'P': {'I': 15, 'S': 10, 'B': 20},
```

```
    'Q': {'C': 30, 'M': 85, 'K': 30},
    'R': {'D': 40, 'K': 75, 'A': 55},
    'S': {'B': 50, 'N': 30, 'P': 10, 'H': 45},
    'T': {'G': 20, 'F': 35, 'L': 40, 'H': 20},
    'U': {'H': 40}
}

# Ευρετικές τιμές για κάθε κόμβο (εκτιμώμενη απόσταση από τον στόχο F)
# Η τιμή 0 για τον κόμβο F υποδεικνύει ότι είναι ο στόχος
heuristic = {
    'A': 85, 'B': 80, 'C': 80, 'D': 85, 'E': 35, 'F': 0,
    'G': 50, 'H': 20, 'I': 60, 'J': 85, 'K': 40, 'L': 70,
    'M': 80, 'N': 20, 'O': 35, 'P': 60, 'Q': 70, 'R': 60,
    'S': 50, 'T': 35, 'U': 55
}

def greedy_best_first_search(graph, start, goal, heuristic):
    """
    Υλοποίηση του αλγορίθμου Greedy Best First Search
    Παράμετροι:
        graph: το λεξικό του γράφου
        start: ο αρχικός κόμβος
        goal: ο κόμβος-στόχος
        heuristic: το λεξικό με τις ευρετικές τιμές
    """

    # Σύνολο για την παρακολούθηση των επισκεπτόμενων κόμβων
    visited = set()

    # Ουρά προτεραιότητας για την επιλογή του επόμενου κόμβου
    pq = PriorityQueue()

    # Εισαγωγή του αρχικού κόμβου (ευρετική τιμή, κόμβος, μονοπάτι, κόστος)
    pq.put((heuristic[start], start, [start], 0))

    while not pq.empty():

        # Εξαγωγή του κόμβου με τη μικρότερη ευρετική τιμή
        (h, current, path, cost) = pq.get()

        # Έλεγχος αν φτάσαμε στον στόχο
        # Αν ναι, επιστρέφουμε το μονοπάτι και το κόστος
        if current == goal:
            return path, cost
```

```
        # Αν όχι, συνεχίζουμε την αναζήτηση
        if current not in visited:
            visited.add(current)

            # Εξερεύνηση των γειτόνων του τρέχοντος κόμβου
            for neighbor in graph[current]:
                if neighbor not in visited:
                    new_path = path + [neighbor]
                    new_cost = cost + graph[current][neighbor]
                    # Χρήση μόνο της ευρετικής τιμής για την προτεραιότητα
                    pq.put((heuristic[neighbor], neighbor, new_path, new_cost))

    return None, None

def a_star(graph, start, goal, heuristic):
    """
    Υλοποίηση του αλγορίθμου A*
    Παράμετροι:
        graph: το λεξικό του γράφου
        start: ο αρχικός κόμβος
        goal: ο κόμβος-στόχος
        heuristic: το λεξικό με τις ευρετικές τιμές
    """
    visited = set()
    pq = PriorityQueue()
    pq.put((heuristic[start], start, [start], 0))

    while not pq.empty():
        (f, current, path, cost) = pq.get()

        if current == goal:
            return path, cost

        if current not in visited:
            visited.add(current)

            for neighbor in graph[current]:
                if neighbor not in visited:
                    new_path = path + [neighbor]
                    new_cost = cost + graph[current][neighbor]

                    # Υπολογισμός της συνάρτησης f(n) = g(n) + h(n)
                    f = new_cost + heuristic[neighbor]
                    pq.put((f, neighbor, new_path, new_cost))
```

```
    return None, None

# Ορισμός των αρχικών κόμβων και του στόχου
start_vertices = ['A', 'B', 'C']
goal = 'F'

# Εκτέλεση και εκτύπωση αποτελεσμάτων για Greedy Best First Search
print("Greedy Best First Search:")
for start in start_vertices:
    path, cost = greedy_best_first_search(graph, start, goal, heuristic)
    if path:
        print(f"Path from {start} to F: {' -> '.join(path)}")
        print(f"Cost: {cost}")
    else:
        print(f"No path found from {start} to F")
    print()

# Εκτέλεση και εκτύπωση αποτελεσμάτων για A*
print("A* Search:")
for start in start_vertices:
    path, cost = a_star(graph, start, goal, heuristic)
    if path:
        print(f"Path from {start} to F: {' -> '.join(path)}")
        print(f"Cost: {cost}")
    else:
        print(f"No path found from {start} to F")
    print()
```

Συμπέρασμα

Και οι δύο αλγόριθμοι επιτυγχάνουν τον στόχο τους, αλλά ο A* προσφέρει βέλτιστες λύσεις με το κόστος της αυξημένης υπολογιστικής πολυπλοκότητας.