Assignment Comments

Επιμέλεια: Α.Δρακόπουλος

Πίνακας περιεχομένων

```
#robot_mid.py – συμπληρώστε τα TODO
# Το αρχείο αυτό περιέχει τον σκελετό κώδικα για την εργασία του ρομπότ συλλέκτη.
# Οι σημειώσεις #TODO υποδεικνύουν τα σημεία που πρέπει να συμπληρωθούν για την πλήρη υλοποίηση.

from collections import deque
# Εισάγει τη δομή δεδομένων 'deque' από τη βιβλιοθήκη 'collections'.
# Το 'deque' (double-ended queue) χρησιμοποιείται για την υλοποίηση ουρών , απαραίτητο για τον αλγόριθμο BFS (Breadth-First Search).

ROWS, COLS = 10, 10
# Ορίζει τις σταθερές για τις διαστάσεις του κόσμου: 10 γραμμές (ROWS) και 10 στήλες (COLS).

EMPTY, OBST, ITEM = ".", "#", "*"
# Ορίζει σταθερές για τα σύμβολα που χρησιμοποιούνται στο πλέγμα (grid):
# EMPTY: Κενό κελί ('.')
# OBST: Εμπόδιο ('#')
# ITEM: Αντικείμενο προς συλλογή ('*')

def load_world(path: str) -> list[list[str]]:
# Ορισμός συνάρτησης για τη φόρτωση (ανάγνωση) του κόσμου από αρχείο.
# Δέχεται ως όρισμα τη διαδρομή (path) του αρχείου κειμένου (π.χ., "world.txt").
# Επιστρέφει μια δι-διάστατη λίστα (2d list)  από συμβολοσειρές (list[list[str]) που αναπαριστά τον κόσμο.

    world = []
    # Αρχικοποιεί μια κενή λίστα 'world' που θα αποθηκεύσει τον κόσμο ως πλέγμα.

    with open(path, "r", encoding="utf-8") as f:
    # Ανοίγει το αρχείο στη διαδρομή 'path' για ανάγνωση ("r"), χρησιμοποιώντας κωδικοποίηση UTF-8.
    # Η εντολή 'with open' εξασφαλίζει ότι το αρχείο θα κλείσει αυτόματα, ακόμη και αν υπάρξουν λάθη.

        for line in f:
        # Επανάληψη (loop) ανά γραμμή του αρχείου.

            row = line.strip().split()
            # line.strip(): Αφαιρεί τυχόν κενά ή αλλαγές γραμμής από την αρχή και το τέλος της γραμμής.
            # .split(): Διαχωρίζει τη γραμμή σε μια λίστα συμβολοσειρών, χρησιμοποιώντας τα κενά ως διαχωριστή (βάσει της μορφής του αρχείου).

            if row:
            # Έλεγχος: Εάν η λίστα 'row' δεν είναι κενή (δηλαδή, η γραμμή δεν ήταν κενή).

                world.append(row)
                # Προσθέτει τη λίστα των συμβόλων (τη γραμμή) στη λίστα 'world'.

    assert len(world) == ROWS and all(len(r)==COLS for r in world), "Λάθος διαστάσεις world.txt"
    # Έλεγχος ορθότητας (Assertion): Εξασφαλίζει ότι ο φορτωμένος κόσμος έχει τις σωστές διαστάσεις 10x10.
```

```
    # len(world) == ROWS: Ο κόσμος πρέπει να έχει 10 γραμμές.
    # all(len(r)==COLS for r in world): Κάθε γραμμή (r) πρέπει να έχει 10 στήλες.
    # Εάν η συνθήκη είναι ψευδής, εκτυπώνεται το μήνυμα λάθους.

    valid = {EMPTY, OBST, ITEM}
    # Ορίζει ένα σύνολο (set) με τα έγκυρα σύμβολα κελιών ('.', '#', '*').

    for r in world:
    # Επανάληψη (loop) για κάθε γραμμή (r) του κόσμου.
        for s in r:
        # Επανάληψη (loop) για κάθε σύμβολο (s) στην τρέχουσα γραμμή.
            assert s in valid, f"Άκυρο σύμβολο: {s}"
            # Έλεγχος ορθότητας: Εξασφαλίζει ότι κάθε σύμβολο ανήκει στα έγκυρα.
            # Εάν βρεθεί άκυρο σύμβολο, εκτυπώνεται μήνυμα λάθους με το σύμβολο.

    return world
    # Επιστρέφει την αναπαράσταση του κόσμου ως δι-διάστατη λίστα.

def print_world(world: list[list[str]], r: int, c: int) -> None:
# Ορισμός συνάρτησης για την εμφάνιση της τρέχουσας κατάστασης του κόσμου.
# Δέχεται τον κόσμο (world) και την τρέχουσα θέση του ρομπότ: γραμμή (r) και στήλη (c).
# Δεν επιστρέφει τιμή (None), απλώς εκτυπώνει στην οθόνη.

    for i in range(ROWS):
    # Επανάληψη (loop) για κάθε γραμμή (i) από 0 έως ROWS-1.

        line = []
        # Αρχικοποιεί μια κενή λίστα 'line' για να αποθηκεύσει τα σύμβολα της τρέχουσας γραμμής προς εκτύπωση.

        for j in range(COLS):
        # Επανάληψη (loop) για κάθε στήλη (j) από 0 έως COLS-1.

            line.append("R" if (i,j)==(r,c) else world[i][j])
            # Προσθέτει το σύμβολο στο 'line'.
            # Χρησιμοποιεί τριαδικό τελεστή:
            # Εάν η τρέχουσα θέση (i, j) είναι η θέση του ρομπότ (r, c), προσθέτει το 'R' (robot).
            # Διαφορετικά, προσθέτει το σύμβολο του κελιού από τον κόσμο (world[i][j]).

        print(" ".join(line))
        # Ενώνει τα στοιχεία της λίστας 'line' με κενό (' ') ως διαχωριστή και εκτυπώνει την τελική συμβολοσειρά (τη γραμμή).

    print()
    # Εκτυπώνει μια κενή γραμμή για οπτικό διαχωρισμό.

def in_bounds(r: int, c: int) -> bool:
# Ορισμός συνάρτησης για τον έλεγχο ορίων.
# Δέχεται συντεταγμένες (r, c) και επιστρέφει True αν είναι εντός των ορίων του πλέγματος, αλλιώς False.

    return 0 <= r < ROWS and 0 <= c < COLS
```

```
    # Επιστρέφει True αν η γραμμή (r) είναι μεταξύ 0 και 9, και η στήλη (c) είναι μεταξύ 0 και 9 (ROWS και COLS είναι 10).

def neighbors(world, r, c):
# Ορισμός συνάρτησης για την εύρεση των έγκυρων γειτονικών κελιών.
# Δέχεται τον κόσμο (world) και την τρέχουσα θέση (r, c).
# Είναι μια γεννήτρια (generator) που επιστρέφει γειτονικές θέσεις (nr, nc).

    for dr, dc in [(-1,0), (1,0), (0,-1), (0,1)]:
    # Επανάληψη (loop) στις 4 δυνατές κινήσεις (πάνω, κάτω, αριστερά, δεξιά), εκφρασμένες ως μεταβολές στις συντεταγμένες (dr, dc).

        nr, nc = r+dr, c+dc
        # Υπολογίζει τις νέες συντεταγμένες (nr, nc) του γειτονικού κελιού.

        if in_bounds(nr, nc) and world[nr][nc] != OBST:
        # Έλεγχος: Εάν το γειτονικό κελί:
        # 1. Είναι εντός των ορίων του πλέγματος (in_bounds(nr, nc)).
        # 2. Δεν είναι εμπόδιο (OBST).

            yield nr, nc
            # Επιστρέφει (yield) τις συντεταγμένες του έγκυρου γειτονικού κελιού.
            # Η χρήση του 'yield' κάνει τη συνάρτηση γεννήτρια, δίνοντας μία-μία τις έγκυρες γειτονικές θέσεις.

def bfs_path_to_nearest_item(world, start):
# Ορισμός συνάρτησης για τον υπολογισμό της συντομότερης διαδρομής προς το πλησιέστερο αντικείμενο με BFS.
# Δέχεται τον κόσμο (world) και τη θέση εκκίνησης (start).

    # TODO: Υλοποιήστε BFS. Επιστρέψτε λίστα θέσεων (χωρίς το start) μέχρι ένα ITEM ή [] αν δεν υπάρχει.
    # Αυτό το σημείο πρέπει να συμπληρωθεί με τον αλγόριθμο BFS.

    pass
    # Προσωρινή εντολή, δεν κάνει τίποτα.

def next_step_bfs(world, pos):
# Ορισμός συνάρτησης για τον **υπολογισμό του επόμενου βήματος** χρησιμοποιώντας τη στρατηγική BFS.
# Δέχεται τον κόσμο (world) και την τρέχουσα θέση (pos).

    path = bfs_path_to_nearest_item(world, pos)
    # Καλεί τη συνάρτηση BFS για να βρει τη συντομότερη διαδρομή (path) προς το πλησιέστερο αντικείμενο.

    return path[0] if path else pos
    # Επιστρέφει:
    # - Το πρώτο βήμα της διαδρομής (path[0]), αν βρέθηκε διαδρομή (path είναι μη-κενή λίστα).
    # - Την τρέχουσα θέση (pos), αν δεν βρέθηκε αντικείμενο (path είναι κενή λίστα).

def next_step_sweep(world, r, c, direction):
```

Ορισμός συνάρτησης για τον υπολογισμό του επόμενου βήματος χρησιμοποιώντας τη στρατηγική Sweep.
Δέχεται τον κόσμο (world), την τρέχουσα θέση (r, c) και την τρέχουσα κατεύθυνση κίνησης (direction: 1 για δεξιά, -1 για αριστερά).

    # υλοποιημένη για να επικεντρωθείτε στο BFS
    # Σχόλιο: Η υλοποίηση της sweep δίνεται έτοιμη, ώστε ο χρήστης να επικεντρωθεί στον BFS.

    if direction == 1 and c == COLS-1:
    # Έλεγχος: Εάν η κίνηση είναι προς τα δεξιά (direction == 1) ΚΑΙ το ρομπότ είναι στο δεξιό άκρο (c == COLS-1, δηλαδή 9η στήλη).

        return (r+1 if r+1 < ROWS else r, c, -1)
        # Επιστρέφει τη νέα κατάσταση:
        # - Νέα γραμμή: r+1 (κατεβαίνει μία γραμμή), εκτός αν είναι ήδη στην τελευταία (τότε μένει στην r).
        # - Στήλη: παραμένει στην c.
        # - Νέα κατεύθυνση: -1 (αριστερά).

    if direction == -1 and c == 0:
    # Έλεγχος: Εάν η κίνηση είναι προς τα αριστερά (direction == -1) ΚΑΙ το ρομπότ είναι στο αριστερό άκρο (c == 0).

        return (r+1 if r+1 < ROWS else r, c, 1)
        # Επιστρέφει τη νέα κατάσταση:
        # - Νέα γραμμή: r+1 (κατεβαίνει μία γραμμή), εκτός αν είναι ήδη στην τελευταία (τότε μένει στην r).
        # - Στήλη: παραμένει στην c.
        # - Νέα κατεύθυνση: 1 (δεξιά).

    return (r, c+direction, direction)
    # Επιστρέφει τη νέα κατάσταση (τυπική κίνηση, όχι άκρο):
    # - Γραμμή: παραμένει η r.
    # - Νέα στήλη: c + direction (κινείται μία θέση δεξιά ή αριστερά).
    # - Κατεύθυνση: παραμένει η direction.

def run(strategy="BFS", verbose_every=10):
# Ορισμός της κύριας συνάρτησης προσομοίωσης.
# Δέχεται την επιθυμητή στρατηγική ("BFS" ή "Sweep", με default "BFS") και την συχνότητα εμφάνισης της κατάστασης (verbose_every, default 10).

    world = load_world("world.txt")
    # Φορτώνει τον κόσμο από το αρχείο "world.txt".

    r, c = 0, 0
    # Αρχικοποιεί τη θέση του ρομπότ: γραμμή r=0, στήλη c=0 (πάνω αριστερά).

    direction = 1
    # Αρχικοποιεί την κατεύθυνση κίνησης για τη στρατηγική Sweep (1 = δεξιά).

    score, steps = 0, 0
    # Αρχικοποιεί το σκορ (συλλεγμένα αντικείμενα) και τον αριθμό βημάτων.

```
    while steps < 80:
    # Κύριος βρόχος προσομοίωσης: Συνεχίζει όσο ο αριθμός βημάτων είναι μικρότερος από 80.

        if world[r][c] == ITEM:
        # Έλεγχος: Εάν το τρέχον κελί περιέχει αντικείμενο.

            world[r][c] = EMPTY
            # Το ρομπότ συλλέγει το αντικείμενο: Το κελί γίνεται κενό (.).

            score += 1
            # Αυξάνει το σκορ κατά 1.

            # Εμφάνιση μηνύματος συλλογής αντικειμένου (ΑΠΑΙΤΗΣΗ)
            # print(f"Συλλέχθηκε αντικείμενο στη θέση: ({r}, {c})")

            if not any(ITEM in row for row in world):
            # Έλεγχος: Εάν δεν υπάρχουν άλλα αντικείμενα (ITEM) σε καμία γραμμή του κόσμου.
            # Η συνάρτηση any() ελέγχει αν υπάρχει έστω και ένα True (δηλαδή ITEM) σε όλες τις γραμμές.

                break
                # Εάν τελείωσαν τα αντικείμενα, η προσομοίωση σταματά (κανόνας τερματισμού 1).

        if strategy.upper() == "BFS":
        # Έλεγχος: Εάν η επιλεγμένη στρατηγική είναι BFS.

            nr, nc = next_step_bfs(world, (r, c))
            # Υπολογίζει τις συντεταγμένες του επόμενου βήματος (nr, nc) με τη στρατηγική BFS.

            r, c = (nr, nc)
            # Ενημερώνει την τρέχουσα θέση του ρομπότ.

        else:
        # Εάν η στρατηγική δεν είναι BFS (άρα είναι Sweep).

            r, c, direction = next_step_sweep(world, r, c, direction)
            # Υπολογίζει τη νέα θέση (r, c) και την πιθανώς νέα κατεύθυνση (direction) με τη στρατηγική Sweep.

        steps += 1
        # Αυξάνει τον αριθμό των βημάτων κατά 1.

        if verbose_every and steps % verbose_every == 0:
        # Έλεγχος για εμφάνιση λεπτομερειών: Εάν το verbose_every είναι μη-μηδενικό ΚΑΙ τα βήματα είναι πολλαπλάσιο του verbose_every.

            print(f"Step {steps} | Pos=({r},{c}) | Score={score} | Strat={strategy}")
            # Εκτυπώνει την τρέχουσα κατάσταση (βήμα, θέση, σκορ, στρατηγική).

            print_world(world, r, c)
            # Εμφανίζει την τρέχουσα μορφή του κόσμου.
```

```
    # Εκτύπωση αποτελεσμάτων ολοκλήρωσης
    # Εδώ θα πρέπει να προστεθεί και το μήνυμα ολοκλήρωσης (λόγος τερματισμού: 80 βήματα ή 0 αντικείμενα).
    # Επίσης, η εμφάνιση της τελικής μορφής του κόσμου.

    print(f"Τελικό σκορ: {score}, Βήματα: {steps}, Στρατηγική: {strategy}")
    # Εκτυπώνει τα συνολικά αποτελέσματα της προσομοίωσης.

# (Η κλήση της συνάρτησης run() για την εκτέλεση του προγράμματος λείπει από τον σκελετό, π.χ. if __name__ == "__main__": run("BFS") ή run("Sweep"))
```

```
def print_world(world: list[list[str]], r: int, c: int) -> None:
# Ορισμός συνάρτησης για την εμφάνιση της τρέχουσας κατάστασης του κόσμου.
# Δέχεται τον κόσμο (world) ως 2D λίστα συμβολοσειρών και την τρέχουσα θέση του ρομπότ (r, c).
# Δεν επιστρέφει τιμή (None).

    for i in range(ROWS):
    # Επανάληψη (loop) για κάθε γραμμή (i) του πλέγματος.

        line = []
        # Αρχικοποιεί μια κενή λίστα 'line' για τα σύμβολα της τρέχουσας γραμμής.

        for j in range(COLS):
        # Επανάληψη (loop) για κάθε στήλη (j) στην τρέχουσα γραμμή.

            line.append("R" if (i,j)==(r,c) else world[i][j])
            # Προσθέτει το σύμβολο: "R" αν η θέση (i, j) είναι η θέση του ρομπότ (r, c),
            # αλλιώς προσθέτει το σύμβολο του κελιού από τον κόσμο (world[i][j]).

        print(" ".join(line))
        # Ενώνει τα σύμβολα της λίστας 'line' με ένα κενό ως διαχωριστή και εκτυπώνει τη γραμμή.

    print()
    # Εκτυπώνει μια κενή γραμμή για οπτικό διαχωρισμό.
```