Πρώτο μέρος εργαστηριακής εργασίας στο μάθημα Τεχνήτη Νοημοσύνη

Ιάκωβος Μαστρογιαννόπουλος - cse242017102

2020-11-22

Contents

Listings

1 Πρόλογος

Στη συγκεκρίμενη εργασία του μαθήματος «Τεχνητής Νοημοσύνης» είχαμε να μελετήσουμε το προβλήμα του parking. Βέβαια, για να μπορέσουμε να φτάσουμε στο σημείο της υλοποίησης του κωδικά, πρώτα πρέπει να εξηγηθούν μερικά πραγμάτα με το ποιο είναι το πρόβλημα, πώς μπορούμε να το υλοποιήσουμε και ποια θα είναι τα πιθανά αποτελέσματα που θα μπορούσαμε να πάρουμε πίσω. Όπως έχει αναφερθεί και σε email, την εργασία την κάνει ένα ατόμο μόνο του. Η γλώσσα προγραμματισμού που υλοποιήθηκαν οι αλγόριθμοι είναι η Python, στον IDE Pycharm και συγκεκριμένα ήταν η έκδοση 3.8.6.

2 Κατανόηση προβλήματος - Προβλήμα του Parking

Το πρόβλημα που έχουμε να λύσουμε είναι το πρόβλημα του Parking. Θεωρετηκά, υπάρχει ένας αυτόματος οδηγός ο οποίος προσπαθεί να βρει ελευθέρο χώρο για να γεμίσει τα αμάξια που θέλουν να μπουν μέσα στο Parking. Υπάρχουν N spaces από τα οποία θα μπορούσαν μερικά από αυτά να ήταν τελείως άδεια, ενώ κάποια αλλά να είχαν μια πλατφόρμα.

Σκοπός είναι να βρίσκει τις πλατφόρμες και από εκεί να καταλαβαίνει ποιες από αυτές έχουν ελεύθερο χρόνο και να τις γεμίζει. Τα βήματα που πρέπει να ακολουθήσουμε είναι τα εξής:

- Να βρίσκει σε ποιο node με πλατφόρμα υπάρχει ελεύθερη θέση
- Να τρέχει έναν αλγόριθμο αναζήτησης και να βρίσκει το πιο γρήγορο path
- Να ανταλλάζει θέση τα nodes μεταξύ τους, έτσι ώστε το node με την άδεια πλατφόρμα να πηγαίνει στην θέση 1
- Τέλος να έχει κάποιον στόχο που όταν τον εκπληρώσει να λήγει το πρόγραμμα

3 Μοντελοποίηση του προβλήματος

3.1 Χώρος καταστάσεων

Χώρο καταστάσεων ενός προβλήματος ονομάζουμε το σύνολο των πιθανών καταστάσεων, στις οποίες μπορούν να βρεθούν οι καταστάσεις του προβλήματος. Στο πρόβλημα του parking τα αντικείμενα είναι τα αυτοκίνητα, τα spaces και οι πλατφόρμες. Μια κατάσταση είναι εάν το space έχει πλατφόρμα ή εάν η πλατφόρμα έχει ελεύθερο χώρο για να χωρέσει αυτοκίνητα.

3.2 Αρχική κατάσταση

Η αρχική κατάσταση του Parking απαιτεί να υπάρχουν 4 spaces, όπου μόνο 3 από αυτά έχουν ελεύθερες θέσεις, και 3 αυτοκίνητα που περιμένουν απέξω για να μπουν μέσα. Στην προέκταση, μας ζητήθηκε να αυξήσουμε τον αριθμό των spaces.

3.3 Τελική κατάσταση

Δεν υπάρχει κάποια συγκεκριμένη τελική κατάσταση για το συγκεκριμένο πρόβλημα. Ομώς, εμείς μπορούμε να θεωρήσουμε ως τελική κατάσταση όταν όλες οι πλατφόρμες έχουν γεμίσει με αυτοκίνητα.

3.4 Τελεστές προβλήματος

Στο πρόβλημα μας έχουμε δύο τελεστές:

- Τελεστή IN: ο Τελεστής όπου βάζει τα αυτοκίνητα μέσα στις πλατφόρμες.
- Τελεστή Neighbour: ο Τελεστής όπου διαβάζει τους γειτονές του κάθε node.

Στην κωδικοποίηση, βέβαια, η υλοποιήση των τελεστών ήταν λίγο διαφορετική, αλλά η ίδεα παραμένει ίδια.

4 Κωδικοποίηση

4.1 Ορισμός κατάστασης

Για την κωδικοποιήση του προβλήματος, φτιάξαμε 2 dictionaries όπου στο ένα έχουμε αποθηκευμένο τις γειτονικές σχεσείς των nodes, ενώ στο αλλό την κατάσταστη του κάθε node (πχ εάν έχει πλατφόρμα ή εάν είναι άδειο).

Δηλαδή, η αρχική μας κατάσταση θα είναι η εξής:

```
        neighbours = {
            '1': set(['2', '4']),
            '2': set(['1', '3']),
            '3': set(['2', '4', '6']),
            '4': set(['1', '3', '5']),
            '5': set(['4', '6']),
            '6': set(['3', '5'])
        }

        spaces = {
            '1': ["Empty", "NO"],
            '2': ["Platform 1", "NO"],
            '3': ["Platform 2", "NO"],
            '4': ["Platform 3", "NO"],
            '5': ["Platform 4", "NO"],
            '6': ["Platform 5", "NO"]
        }

```

Ως τελική κατάσταση, μπορούμε να ορίσουμε ότι όλα τα spaces είναι γεμάτα.

```
        spaces = {
            '1': ["Empty", "NO"],
            '2': ["Platform 1", "YES"],
            '3': ["Platform 2", "YES"],
            '4': ["Platform 3", "YES"],
            '5': ["Platform 4", "YES"],
            '6': ["Platform 5", "YES"]
        }

```

4.2 Κωδικοποιήμενοι Τελεστές Ελέγχου

Έχουμε δύο τελεστές ελεγχού

Τελεστής ελέγχου	Λειτουργία
is empty	Ψαχνεί να βρει σε ποιο node έχει αδεία πλατφόρμα.
is goal	Κοιτάει κάθε στοιχείο του γράφου εάν η καταστάση του είναι ίδια με του goal state.

Ακολουθεί αναλυτικά το πως δουλευουν αναλυτικά

4.3 is empty

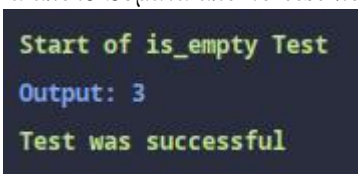
```
from Parking.colors import *


def is_empty(spaces, goal_state):
    for i in range(1, len(spaces) + 1):
        if spaces[str(i)][0][:1] == 'E' or spaces[str(i)][1] != "NO
    " or goal_state[spaces[str(i)][0]] == "NO":
            continue

        return str(i)


if __name__ == '__main__':
    print(GREEN + "Start of is_empty Test" + DEFAULT)

    test_spaces = {
        '1': ["Empty"],
        '2': ["Platform 1", "YES"],
        '3': ["Platform 2", "NO"],
        '4': ["Platform 3", "YES"]
    }

    test_goal_state = {
        "Empty": "NO",
        "Platform 1": "YES",
        "Platform 2": "YES",
        "Platform 3": "YES"
    }

    print(BLUE + f"Output: {is_empty(test_spaces, test_goal_state)}
    " + DEFAULT)

    print(GREEN + "Test was successful" + DEFAULT)
```

Listing 1: Is Empty Algorithm

Μπορούμε να παρατηρήσουμε ότι κοιτάει τρία πραγμάτα:

- Εάν το space που είμαστε τώρα δεν είναι empty
- Εάν το space που είμαστε τώρα έχει πλατφόρμα και είναι άδεια
- Εάν το goal της πλατφόρμα που έχει το space, την θέλουμε άδεια.

Τα αποτελέσματα από το test που έχω γράψει μας επιστρέφουν το εξής:



Μπορούμε να παρατηρήσουμε ότι βρήκε σωστά ότι στο 3ο index έχει άδεια θέση.

4.4 is goal

```
from Parking.colors import *


def is_goal(state, goal_state):
    for node in state:
        if state[node] != goal_state[node]:
            return False

    return True


if __name__ == '__main__':
    print(GREEN + "Start of is_goal Test" + DEFAULT)

    test_state1 = {
        "Platform1": "Yes",
        "Platform2": "Yes",
        "Platform3": "Yes",
        "Empty": "No"
    }

    test_state2 = {
        "Platform1": "No",
        "Platform2": "Yes",
        "Platform3": "Yes",
        "Empty": "No"
    }

    test_goal = {
        "Platform1": "Yes",
        "Platform2": "Yes",
        "Platform3": "Yes",
        "Empty": "No"
    }

    if is_goal(test_state1, test_goal):
        print(BLUE + "We have reached the goal for test_state1" +
    DEFAULT)

    else:
        print(BLUE + "We haven't reached the goal for test_state1"
    + DEFAULT)

```

```
42      if is_goal(test_state2, test_goal):
43          print(BLUE + "We have reached the goal for test_state2" +
        DEFAULT)
44
45      else:
46          print(BLUE + "We haven't reached the goal for test_state2"
        + DEFAULT)
47
48      print(GREEN + "Test was successful" + DEFAULT)
```

Listing 2: Is Goal Algorithm

Εδώ παρατηρούμε ότι ο αλγόριθμος κοιτάει όλα τα nodes του state, και τα συγκρίνει με τα ίδια nodes του goal state έαν έχουν την ίδια τιμή. Εάν βρει ένα ζευγάρι που δεν έχει την ίδια τιμή, επιστρέφει False, ενώ εάν ολοκληρωθεί το loop και δεν βρει κάτι, σημαίνει ότι έχει βρεθεί το goal και επιστρέφει True.

Πάμε να δούμε τα αποτελέσματα του test που έχει γραφτεί:

```
Start of is_goal Test
We have reached the goal for test_state1
We haven't reached the goal for test_state2
Test was successful
```

Για το πρώτο test state, μπορούμε να δούμε ότι βρήκε ότι έχει ολοκληρωθεί και εκπληρώσει τον σκοπό του, ενώ για το δεύτερο δεν τον έχει εκπληρώσει.

4.5 Τελεστές Μετάβασης

Η κωδικοποιήση δυστυχώς δεν έχει ξεχωρίστους τελεστές μετάβασης, όλα βρίσκονται μέσα στο find solution, λόγω ελείψης χρόνου. Όποτε πάμε να δούμε αναλυτικά τι κάνει.

4.6 find solution

```
from Parking.colors import *
from Parking.DFS import *
from Parking.BFS import *
from Parking.BestFS import *
from Parking.is_empty import *
from Parking.swap import *


def find_solution(neighbour_graph, space_graph, goal_state, method):
    goal = is_empty(space_graph, goal_state)
    if goal is None:
        print(RED + "NO SOLUTION COULD BE FOUND" + DEFAULT)
        return None

    visited = []
    if method == "DFS":
        visited = list(dfs(neighbour_graph, '1', goal))

    elif method == "BFS":
        visited = list(bfs(neighbour_graph, '1', goal))

    elif method == "BestFS":
        visited = list(best_fs(neighbour_graph, '1', goal))

    else:
        print(RED + "METHOD NOT FOUND" + DEFAULT)
        return None

    print(GREEN + "FRONT: " + BLUE + f"{visited}")
    visited = min(visited, key=len)
    print(GREEN + "FASTEST SOLUTION: " + BLUE + f"{visited}")
    swap(space_graph, visited)
    space_graph['1'][1] = "YES"

    return space_graph


if __name__ == '__main__':
    print(GREEN + "Start of find_solution Test" + DEFAULT)

    test_neighbours = {
        '1': set(['2', '4']),
        '2': set(['1', '3']),
        '3': set(['2', '4', '6']),
        '4': set(['1', '3', '5']),
        '5': set(['4', '6']),
        '6': set(['3', '5'])
    }

    test_spaces = {
        '1': ["Empty", "NO"],
        '2': ["Platform 1", "NO"],
        '3': ["Platform 2", "NO"],
        '4': ["Platform 3", "NO"],
```

```
        '5': ["Platform 4", "NO"],
        '6': ["Platform 5", "NO"]
    }

    test_goal_state = {
        "Empty": "NO",
        "Platform 1": "NO",
        "Platform 2": "YES",
        "Platform 3": "YES",
        "Platform 4": "YES",
        "Platform 5": "YES"
    }

    find_solution(test_neighbours, test_spaces, test_goal_state, "
    DFS")
    print(test_spaces)

    print(GREEN + "Test ended successfully" + DEFAULT)
```

Listing 3: Find Solution Algorithm

Μπορούμε να δούμε ότι το πρώτο πράγμα που κάνει είναι να βρει σε ποιο index βρίσκεται η άδεια πλατφόρμα με την χρήση του is empty που μελετήσαμε προηγουμένος. Μετά, τρέχει τον αλγόριθμο αναζήτησης που έχουμε επιλέξει να τρέξει για να πάρουμε πίσω έναν πίνακα που έχει μέσα το Front της μεθόδου, όπου μετά τον τρέχουμε στην συνάρτηση min με key το len για να βρούμε το γρηγορότερο path. Πιο αναλυτικά για το πως δουλεύει κάθε αλγόριθμος αναζήτησης θα το δούμε στο επόμενο κεφάλαιο. Στην συνέχεια τρέχει την swap. Παμέ να δούμε και τον αλγόριθμο swap.

```python
from Parking.colors import *


def swap(spaces, visited):
    for i in range(len(visited) - 1, 0, -1):
        spaces[visited[i]], spaces[visited[i - 1]] = spaces[visited
    [i - 1]], spaces[visited[i]]
    return spaces


if __name__ == '__main__':
    print(GREEN + "Start of Swap Test" + DEFAULT)

    test_spaces = {
        '1': ["Empty"],
        '2': ["Platform 1", "YES"],
        '3': ["Platform 2", "NO"],
        '4': ["Platform 3", "YES"]
    }

    test_visited = ['1', '4', '3']

    print(BLUE + f"BEFORE: {test_spaces}" + DEFAULT)
    test_spaces = swap(test_spaces, test_visited)
    print(BLUE + f"AFTER: {test_spaces}" + DEFAULT)

    print(GREEN + "Test was successful" + DEFAULT)
```

Listing 4: Swap Algorithm

Η swap είναι μία βοηθητική συνάρτηση μετάβασης που την λίστα από τον αλγόριθμο αναζήτησης και ανταλλάζει τα στοιχεία μεταξύ τους επιστρέφοντας πίσω μία νέα λίστα.

```
Start of Swap Test
BEFORE: {'1': ['Empty'], '2': ['Platform 1', 'YES'], '3': ['Platform 2', 'NO'], '4': ['Platform 3', 'YES']}
AFTER: {'1': ['Platform 2', 'NO'], '2': ['Platform 1', 'YES'], '3': ['Platform 3', 'YES'], '4': ['Empty']}
Test was successful
```

Στην συνέχεια, το find solution αλλάζει την τιμή στην θέση 1, αφού εκεί θέλουμε να γινόνται όλες οι αλλάγες και το κάνει YES, ότι είναι γεμάτο, και επιστρέφει πίσω ένα τελικό αποτέλεσμα. Παμέ να δούμε τα αποτελέσματα του find solution.

```
Start of find_solution Test
FRONT: [['1', '4', '3'], ['1', '4', '5', '6', '3'], ['1', '2', '3']]
FASTEST SOLUTION: ['1', '4', '3']
{'1': ['Platform 2', 'YES'], '2': ['Platform 1', 'NO'], '3': ['Platform 3', 'NO'], '4': ['Empty', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
Test ended successfully
```

Μπορούμε να δούμε ότι βρίσκει ένα Front από τον αλγόριθμο αναζήτησης (σε αυτή την περιπτώση ήταν ο DFS) και επιτυχημένα βρίσκει το πιο γρήγορο μονοπάτι, κάνει το swap και παίρνει σωστά αποτελέσματα

5 Αλγόριθμοι Αναζήτησης

Οι αλγόριθμοι αναζήτησης που υποστιρίζει η κωδικοποιήση μας είναι τρεις:

- Ο Αλγόριθμος DFS
- Ο Αλγόριθμος BFS
- Ο Αλγόριθμος BestFS

Στην συνέχεια θα δούμε αναλυτικά πως δουλεύει ο καθένας του ξεχωριστά.

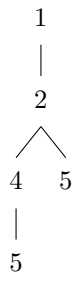
5.1 DFS

```python
from Parking.colors import *


def dfs(graph, node, goal, path=None):
    if path is None:
        path = [node]

    if node == goal:
        yield path

    for next_node in graph[node] - set(path):
        yield from dfs(graph, next_node, goal, path + [next_node])


if __name__ == '__main__':
    print(GREEN + "Start of DFS Test" + DEFAULT)

    test_graph = {
        '1': set(['2', '3']),
        '2': set(['1', '4', '5']),
        '3': set(['1']),
        '4': set(['2', '5']),
        '5': set(['2', '4'])
    }

    print(BLUE + f"Starting point: {test_graph}" + DEFAULT)

    for i in range(1, 6):
        visited = list(dfs(test_graph, str(i), '5'))
        print(BLUE + f"{i}) Visited: {visited}" + DEFAULT)

        if len(visited) != 0:
            visited = min(visited, key=len)
            print(BLUE + f"Fastest path: {visited}" + DEFAULT)

        else:
            print(BLUE + "There is no path" + DEFAULT)

    print(GREEN + "Test was successful" + DEFAULT)
```

Listing 5: DFS Algorithm

Ο αλγορίθμος βασίζετε στην αναδρομική διαδικάσια η οποία δέχεται σαν είσοδο μία κορυφή u και επισκέπτεται αναδρομικά το αριστερό και μετά το δεξίο του γειτονική κορυφή. Στην πραγματικότητα βεβαία υλοποιεί μία στοίβα, με ιεραρχία LIFO. Στο συγκεκριμένο παράδειγμα, έχουμε μια παραλλαγή του DFS όπου παρακολούθει παραλλήλα κάθε πιθανή πορεία που θα μπορούσε να πάρει το node για να φτάσει στον σκοπό του. Στον κώδικα του DFS παρατηρούμε ότι δεν επιστρέφει πίσω μεταβλήτες. Στην πραγματικότητα, επιστρέφει πίσω generators τιμών και ετσί μειώνεται η πολυπλοκότητα του αλγοριθμού, αφού δεν χρειάζεται να αποθηκεύει την ίδια λίστα ξανά και ξανά.

Παμέ να δούμε θεωρετικά το γράφο που χρησιμοποιούμε στο test case. Από το 1 εώς το 5, θα μας επιστρέψει το εξής tree:



Όποτε, εάν ο αλγόριθμος είναι σωστός πρέπει να δούμε το εξής αποτέλεσμα: [[1,2,4,5],[1,2,5]]. Ας τον τρέξουμε τον αλγόριθμο.

```
1) Visited: [['1', '2', '4', '5'], ['1', '2', '5']]
```

Βλεπούμε ότι όντως παίρνουμε το σωστό αποτέλεσμα, όποτε είναι σωστή η υλοποιήση του DFS.

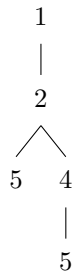
5.2 BFS

```python
from Parking.colors import *


def bfs(graph, node, goal):
    queue = [(node, [node])]

    while queue:
        (vertex, path) = queue.pop(0)

        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]

            else:
                queue.append((next, path + [next]))


if __name__ == '__main__':
    print(GREEN + "Start of BFS Test" + DEFAULT)

    test_graph = {
        '1': set(['2', '3']),
        '2': set(['1', '4', '5']),
        '3': set(['1']),
        '4': set(['2', '5']),
        '5': set(['2', '4'])
    }

    print(BLUE + f"Starting point: {test_graph}" + DEFAULT)

    for i in range(1, 6):
        visited = list(bfs(test_graph, str(i), '5'))
        print(BLUE + f"{i}) Visited: {visited}" + DEFAULT)

        if len(visited) != 0:
            visited = min(visited, key=len)
            print(BLUE + f"Fastest path: {visited}" + DEFAULT)

        else:
            print(BLUE + "There is no path" + DEFAULT)

    print(GREEN + "Test was successful" + DEFAULT)
```

Listing 6: BFS Algorithm

Η διάσχιση κατά πλάτος (BFS) πρώτα επισκέπτεται όλους τους κόμβους που βρίσκονται στο ίδιο επείπεδο και μέτα προχωράει στους επόμενους κόμβους. Με άλλα λόγια, αντί να υλοποιείται LIFO, έχουμε FIFO. Αλλή μια διάφορα μεταξύ DFS και BFS είναι ότι ο BFS δεν είναι αναδρομίκος. Όπως στον DFS, έτσι και στον BFS έχουμε μία παραλλαγή του αλγοριθμού όπου παρακολοθεί παράλληλα κάθε πιθανή πορία που θα μπορούσαμε να παρούμε.

Πάμε να δούμε το ίδιο πάραδειγμα που είχαμε δει στον BFS:



Οπότε εάν τρέξουμε τον κώδικα, περιμένουμε να δούμε το εξής αποτέλεσμα: [[1,2,5],[1,2,4,5]]. Ας τρέξουμε και τον αλγόριθμο.

```
1) Visited: [['1', '2', '5'], ['1', '2', '4', '5']]
```

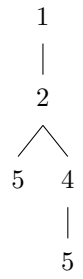
Τα αποτελέσματα για άλλη μια φορά συμφωνούν μαζί μας, όποτε είναι σωστή και αυτή η υλοποιήση.

5.3 BestFS

```python
from Parking.colors import *


def best_fs(graph, node, goal):
    queue = [(node, [node])]

    while queue:
        (vertex, path) = queue.pop(0)

        if vertex == goal:
            yield path

        else:
            for next in graph[vertex] - set(path):
                queue.append((next, path + [next]))


if __name__ == '__main__':
    print(GREEN + "Start of BestFS Test" + DEFAULT)

    test_graph = {
        '1': set(['2', '3']),
        '2': set(['1', '4', '5']),
        '3': set(['1']),
        '4': set(['2', '5']),
        '5': set(['2', '4'])
    }

    print(BLUE + f"Starting point: {test_graph}" + DEFAULT)

    for i in range(1, 6):
        visited = list(best_fs(test_graph, str(i), '5'))
        print(BLUE + f"{i}) Visited: {visited}" + DEFAULT)

        if len(visited) != 0:
            visited = min(visited, key=len)
            print(BLUE + f"Fastest path: {visited}" + DEFAULT)

        else:
            print(BLUE + "There is no path" + DEFAULT)

    print(GREEN + "Test was successful" + DEFAULT)
```

Listing 7: BestFS Algorithm

Ο BestFS είναι μια παραλλαγή του BFS. Βεβαία, πιστεύω ότι η υλοποιήση που έφτιαξα ότι δεν είναι σωστή και θα μπορούσε να βελτιωθεί. Περιμένουμε να δούμε ένα παρόμοιο tree, με αυτό του BFS, δηλαδή:



Οπότε εάν τρέξουμε τον κώδικα, περιμένουμε να δούμε το εξής αποτέλεσμα: [[1,2,5],[1,2,4,5]]. Ας τρέξουμε και τον αλγόριθμο.

```
1) Visited: [['1', '2', '5'], ['1', '2', '4', '5']]
```

Παρόλο που πήραμε ότι περιμέναμε, ακόμα δεν περιμένω να είναι σωστή η υλοποιήση.

6 Ενδεικτικά Τρεξίματα

```
from Parking.find_solution import find_solution
from Parking.is_goal import is_goal
from Parking.colors import *

if __name__ == '__main__':
    print(GREEN + "Start of main program" + DEFAULT)

    neighbours = {
        '1': set(['2', '4']),
        '2': set(['1', '3']),
        '3': set(['2', '4', '6']),
        '4': set(['1', '3', '5']),
        '5': set(['4', '6']),
        '6': set(['3', '5'])
    }

    spaces = {
        '1': ["Empty", "NO"],
        '2': ["Platform 1", "NO"],
        '3': ["Platform 2", "NO"],
        '4': ["Platform 3", "NO"],
        '5': ["Platform 4", "NO"],
        '6': ["Platform 5", "NO"]
    }

    goal_state = {
        "Empty": "NO",
        "Platform 1": "YES",
        "Platform 2": "YES",
        "Platform 3": "YES",
        "Platform 4": "YES",
        "Platform 5": "YES"
    }

    print(GREEN + "INITIAL VALUES")
    print("NEIGHBOURS: " + BLUE + f"{neighbours}")
    print(GREEN + "SPACES: " + BLUE + f"{spaces}")
    print(GREEN + "GOAL STATE: " + BLUE + f"{goal_state}")

    method = input("Choose method you want to use.\nAvailable
    Methods DFS, BFS, BestFS: " + GREEN)

    while True:
        spaces = find_solution(neighbours, spaces, goal_state,
    method)
        if spaces is None:
            break

        print(GREEN + "NEW STATE: " + BLUE + f"{spaces}")

        state = dict()
        for node in spaces:
            state[spaces[node][0]] = spaces[node][1]

        if is_goal(state, goal_state):
```

```
54              print(BLUE + "Found goal state..." + GREEN + "\nExiting
    " + DEFAULT)
55              break
```

Αυτή είναι η main του προγράμματος. Πάμε να δούμε την εκτέλεση του με DFS, BFS και BestFS.

```
Start of main program
INITIAL VALUES
NEIGHBOURS: {'1': {'2', '4'}, '2': {'1', '3'}, '3': {'2', '4', '6'}, '4': {'5', '1', '3'}, '5': {'4', '6'}, '6': {'5', '3'}}
SPACES: {'1': ['Empty', 'NO'], '2': ['Platform 1', 'NO'], '3': ['Platform 2', 'NO'], '4': ['Platform 3', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
GOAL STATE: {'Empty': 'NO', 'Platform 1': 'YES', 'Platform 2': 'YES', 'Platform 3': 'YES', 'Platform 4': 'YES', 'Platform 5': 'YES'}
Choose method you want to use.
Available Methods DFS, BFS, BestFS: DFS
FRONT: [['1', '2'], ['1', '4', '5', '6', '3', '2'], ['1', '4', '3', '2']]
FASTEST SOLUTION: ['1', '2']
NEW STATE: {'1': ['Platform 1', 'YES'], '2': ['Empty', 'NO'], '3': ['Platform 2', 'NO'], '4': ['Platform 3', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '2', '3'], ['1', '4', '5', '6', '3'], ['1', '4', '3']]
FASTEST SOLUTION: ['1', '2', '3']
NEW STATE: {'1': ['Platform 2', 'YES'], '2': ['Platform 1', 'YES'], '3': ['Empty', 'NO'], '4': ['Platform 3', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '2', '3', '4'], ['1', '2', '3', '6', '5', '4'], ['1', '4']]
FASTEST SOLUTION: ['1', '4']
NEW STATE: {'1': ['Platform 3', 'YES'], '2': ['Platform 1', 'YES'], '3': ['Empty', 'NO'], '4': ['Platform 2', 'YES'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '2', '3', '4', '5'], ['1', '2', '3', '6', '5'], ['1', '4', '5'], ['1', '4', '3', '6', '5']]
FASTEST SOLUTION: ['1', '4', '5']
NEW STATE: {'1': ['Platform 4', 'YES'], '2': ['Platform 1', 'YES'], '3': ['Empty', 'NO'], '4': ['Platform 3', 'YES'], '5': ['Platform 2', 'YES'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '2', '3', '4', '5', '6'], ['1', '2', '3', '6'], ['1', '4', '5', '6'], ['1', '4', '3', '6']]
FASTEST SOLUTION: ['1', '2', '3', '6']
NEW STATE: {'1': ['Platform 5', 'YES'], '2': ['Platform 4', 'YES'], '3': ['Platform 1', 'YES'], '4': ['Platform 3', 'YES'], '5': ['Platform 2', 'YES'], '6': ['Empty', 'NO']}
Found goal state...
Exiting
```

```
Start of main program
INITIAL VALUES
NEIGHBOURS: {'1': {'4', '2'}, '2': {'1', '3'}, '3': {'4', '6', '2'}, '4': {'1', '3', '5'}, '5': {'4', '6'}, '6': {'5', '3'}}
SPACES: {'1': ['Empty', 'NO'], '2': ['Platform 1', 'NO'], '3': ['Platform 2', 'NO'], '4': ['Platform 3', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
GOAL STATE: {'Empty': 'NO', 'Platform 1': 'YES', 'Platform 2': 'YES', 'Platform 3': 'YES', 'Platform 4': 'YES', 'Platform 5': 'YES'}
Choose method you want to use.
Available Methods DFS, BFS, BestFS: BFS
FRONT: [['1', '2'], ['1', '4', '3', '2'], ['1', '4', '5', '6', '3', '2']]
FASTEST SOLUTION: ['1', '2']
NEW STATE: {'1': ['Platform 1', 'YES'], '2': ['Empty', 'NO'], '3': ['Platform 2', 'NO'], '4': ['Platform 3', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '4', '3'], ['1', '2', '3'], ['1', '4', '5', '6', '3']]
FASTEST SOLUTION: ['1', '4', '3']
NEW STATE: {'1': ['Platform 2', 'YES'], '2': ['Empty', 'NO'], '3': ['Platform 3', 'NO'], '4': ['Platform 1', 'YES'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '4', '3'], ['1', '2', '3'], ['1', '4', '5', '6', '3']]
FASTEST SOLUTION: ['1', '4', '3']
NEW STATE: {'1': ['Platform 3', 'YES'], '2': ['Empty', 'NO'], '3': ['Platform 1', 'YES'], '4': ['Platform 2', 'YES'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '4', '5'], ['1', '4', '3', '6', '5'], ['1', '2', '3', '4', '5'], ['1', '2', '3', '6', '5']]
FASTEST SOLUTION: ['1', '4', '5']
NEW STATE: {'1': ['Platform 4', 'YES'], '2': ['Empty', 'NO'], '3': ['Platform 1', 'YES'], '4': ['Platform 3', 'YES'], '5': ['Platform 2', 'YES'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '4', '5', '6'], ['1', '4', '3', '6'], ['1', '2', '3', '6'], ['1', '2', '3', '4', '5', '6']]
FASTEST SOLUTION: ['1', '4', '5', '6']
NEW STATE: {'1': ['Platform 5', 'YES'], '2': ['Empty', 'NO'], '3': ['Platform 1', 'YES'], '4': ['Platform 4', 'YES'], '5': ['Platform 3', 'YES'], '6': ['Platform 2', 'YES']}
Found goal state...
Exiting
```

```
Start of main program
INITIAL VALUES
NEIGHBOURS: {'1': {'2', '4'}, '2': {'3', '1'}, '3': {'2', '6', '4'}, '4': {'5', '3', '1'}, '5': {'6', '4'}, '6': {'5', '3'}}
SPACES: {'1': ['Empty', 'NO'], '2': ['Platform 1', 'NO'], '3': ['Platform 2', 'NO'], '4': ['Platform 3', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
GOAL STATE: {'Empty': 'NO', 'Platform 1': 'YES', 'Platform 2': 'YES', 'Platform 3': 'YES', 'Platform 4': 'YES', 'Platform 5': 'YES'}
Choose method you want to use.
Available Methods DFS, BFS, BestFS: BestFS
FRONT: [['1', '2'], ['1', '4', '3', '2'], ['1', '4', '5', '6', '3', '2']]
FASTEST SOLUTION: ['1', '2']
NEW STATE: {'1': ['Platform 1', 'YES'], '2': ['Empty', 'NO'], '3': ['Platform 2', 'NO'], '4': ['Platform 3', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '2', '3'], ['1', '4', '3'], ['1', '4', '5', '6', '3']]
FASTEST SOLUTION: ['1', '2', '3']
NEW STATE: {'1': ['Platform 2', 'YES'], '2': ['Platform 1', 'YES'], '3': ['Empty', 'NO'], '4': ['Platform 3', 'NO'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '4'], ['1', '2', '3', '4'], ['1', '2', '3', '6', '5', '4']]
FASTEST SOLUTION: ['1', '4']
NEW STATE: {'1': ['Platform 3', 'YES'], '2': ['Platform 1', 'YES'], '3': ['Empty', 'NO'], '4': ['Platform 2', 'YES'], '5': ['Platform 4', 'NO'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '4', '5'], ['1', '2', '3', '6', '5'], ['1', '2', '3', '4', '5'], ['1', '4', '3', '6', '5']]
FASTEST SOLUTION: ['1', '4', '5']
NEW STATE: {'1': ['Platform 4', 'YES'], '2': ['Platform 1', 'YES'], '3': ['Empty', 'NO'], '4': ['Platform 3', 'YES'], '5': ['Platform 2', 'YES'], '6': ['Platform 5', 'NO']}
FRONT: [['1', '2', '3', '6'], ['1', '4', '3', '6'], ['1', '4', '5', '6'], ['1', '2', '3', '4', '5', '6']]
FASTEST SOLUTION: ['1', '2', '3', '6']
NEW STATE: {'1': ['Platform 5', 'YES'], '2': ['Platform 4', 'YES'], '3': ['Platform 1', 'YES'], '4': ['Platform 3', 'YES'], '5': ['Platform 2', 'YES'], '6': ['Empty', 'NO']}
Found goal state...
Exiting
```

7 Συμπεράσματα - Βελτιώσεις

Μπορούμε να παρατηρήσουμε ότι ο BFS είναι πιο αποδοτικός απο τον DFS, αλλά ο DFS είναι πιο γρήγορος. Εάν υπήρχε σωστή υλοποιήση του BestFS, βεβαία, αυτός θα ήταν πιο γρήγορος. Μια βελτιώση που θα ήθελα να κάνω στην εργασία είναι η προσθήκη του Find Children, όμως δεν βρήκα κάποια χρησιμότητα στον τρόπο προσέγγισης που ακολούθησα ο ίδιος. Κατά τα αλλά, είδα μία πιο πρακτική χρήση των αλγορίθμων που έμαθα στους Αλγοριθμούς και Πολυπλοκότητα.