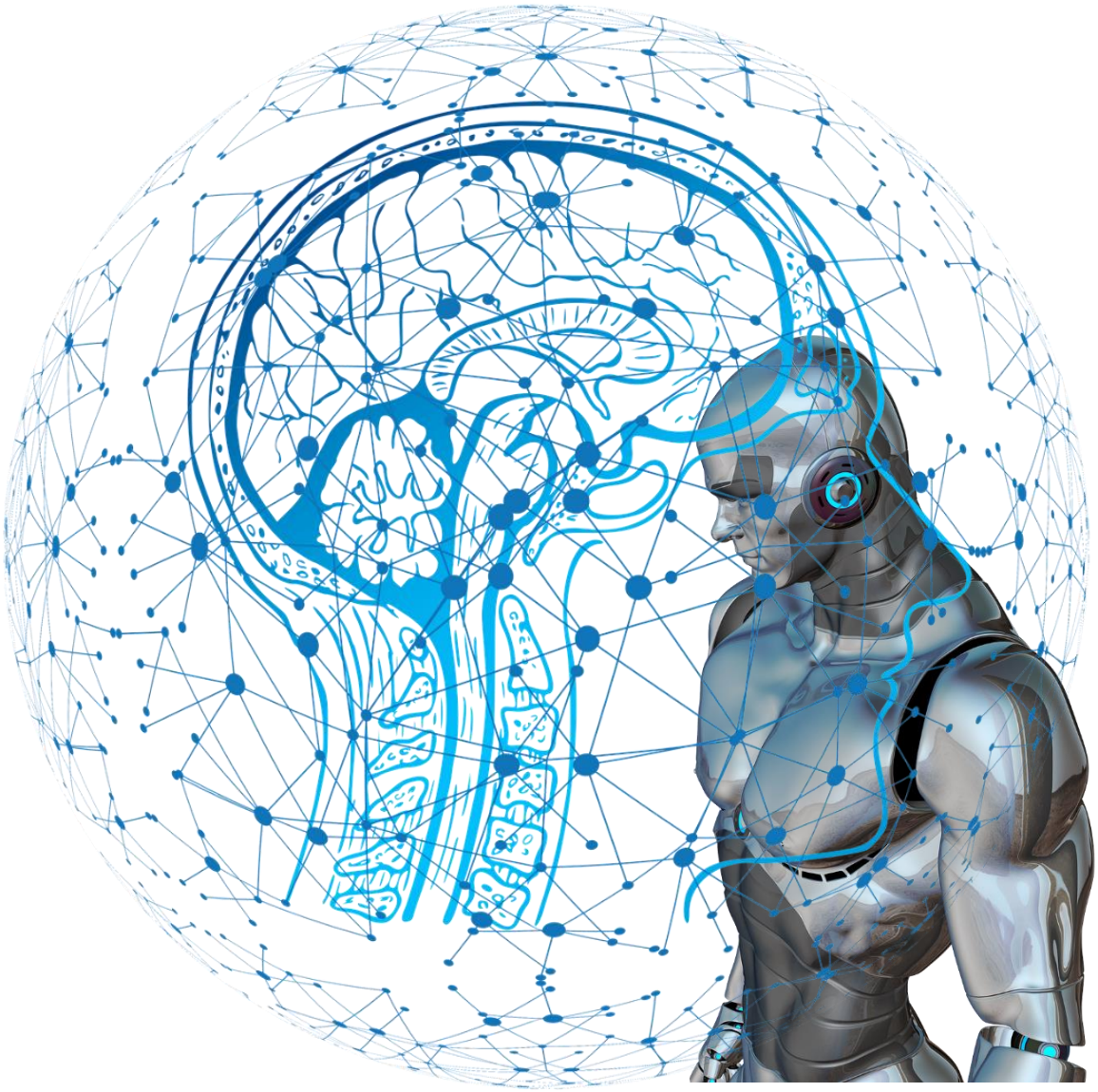
ΚΑΤΑΘΕΣΗ ΝΟΕΜΒΡΙΟΣ 24, 2020



ΤΟ ΠΡΟΒΛΗΜΑ ΤΟΥ PARKING

ΕΡΓΑΣΤΗΡΙΟ

ΒΕΛΑΣΚΟ ΠΑΟΛΑ cs161020
ΜΙΧΑ ΕΥΑΓΓΕΛΙΑ cs171102

ΤΜΗΜΑ 3 (ΝΕΟΙ) ΤΡΙΤΗ 15:00-17:00

Πίνακας Περιεχομένων

Περιγραφή του προβλήματος

Το πρόβλημα του Parking

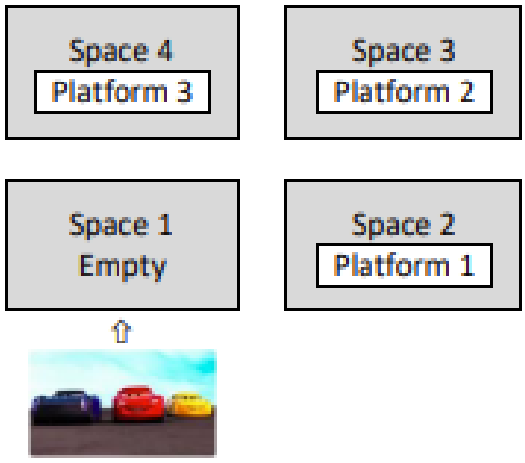
Το πρόβλημα του parking αποτελείται από ένα χώρο στάθμευσης με αυτόματη λειτουργία. Ο χώρος στάθμευσης στο πρόβλημα μας διαθέτει ένα επίπεδο λειτουργίας με n χώρους στάθμευσης (spaces), αριθμημένους από S_1 έως S_n. Ο πρώτος από τους χώρους, ο S1, βρίσκεται στην πρόσοψη και χαρακτηρίζεται ως χώρος υποδοχής. Επίσης, υπάρχουν (n-1) πλατφόρμες εναπόθεσης αυτοκινήτων (platforms), αριθμημένες από P_1 έως P_(n-1) και είναι τοποθετημένες ανά μια σε n-1 χώρους στάθμευσης.

Στην αρχική κατάσταση του προβλήματος, το parking είναι άδειο από αυτοκίνητα δηλαδή όλες οι πλατφόρμες είναι κενές και στον χώρο υποδοχής δεν υπάρχει πλατφόρμα εναπόθεσης.

Το parking διαθέτει αυτόματο σύστημα πλοηγού για την ρύθμιση της διαδικασίας εισόδου. Για να μπορέσει ένα αυτοκίνητο να μπει στο parking, θα πρέπει ο αυτόματος πλοηγός να εξασφαλίσει ότι υπάρχει μια ελεύθερη πλατφόρμα στο χώρο υποδοχής, και να εναποθέσει ένα αυτοκίνητο που είναι σε αναμονή εισόδου πάνω στην πλατφόρμα αυτή.

Κάθε πλατφόρμα μπορεί να φέρει το πολύ ένα αυτοκίνητο. Ο πλοηγός μπορεί να μετακινήσει μια πλατφόρμα από το χώρο που βρίσκεται σε έναν γειτονικό χώρο αρκεί αυτός να είναι κενός.

Πιο συγκεκριμένα, στο πρόβλημα που μας δόθηκε για λόγους απλούστευσης υπάρχουν 4 χώροι στάθμευσης (spaces), εκ των οποίων 3 φέρουν πλατφόρμα εναπόθεσης αυτοκινήτων καθώς και 3 αυτοκίνητα βρίσκονται σε αναμονή εισόδου στο parking.



Μοντελοποίηση του προβλήματος

Κόσμος του προβλήματος

Ο κόσμος του προβλήματος αποτελείται από 4 χώρους στάθμευσης (spaces) αριθμημένους από S_1 έως S_4. Ο πρώτος από τους χώρους, ο S1, βρίσκεται στην πρόσοψη και χαρακτηρίζεται ως χώρος υποδοχής. Επίσης, υπάρχουν 3 πλατφόρμες εναπόθεσης αυτοκινήτων (platforms), αριθμημένες από P_1 έως P_3 και είναι τοποθετημένες ανά μια σε 3 χώρους στάθμευσης.

Αντικείμενα

Τα αντικείμενα στον κόσμο του προβλήματός μας είναι:

- Αυτοκίνητα
- Πλατφόρμες
- Χώροι στάθμευσης

Ιδιότητες

- Κάθε χώρος στάθμευσης μπορεί να καταληφθεί από μια πλατφόρμα ή μπορεί να μείνει άδεια. Δεν μπορούν να μετακινηθούν οι χώροι.
- Αυτοκίνητα μπορούν να εισαχθούν στο χώρο στάθμευσης και να μετακινηθούν ταυτόχρονα μαζί με τις πλατφόρμες.
- Οι πλατφόρμες μπορούν να μετακινηθούν μεταξύ των χώρων στάθμευσης

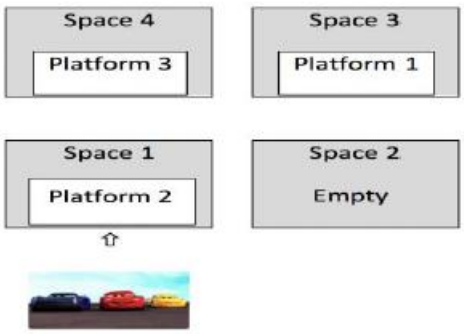
Μεταξύ τους σχέσεις

- Ένα αυτοκίνητο μπορεί να εισαχθεί μόνο από το χώρο στάθμευσης εισόδου και μόνο αν αυτός περιέχει μια άδεια πλατφόρμα.
- Οι πλατφόρμες μπορούν να καταληφθούν από ένα μόνο αυτοκίνητο στο καθένα.

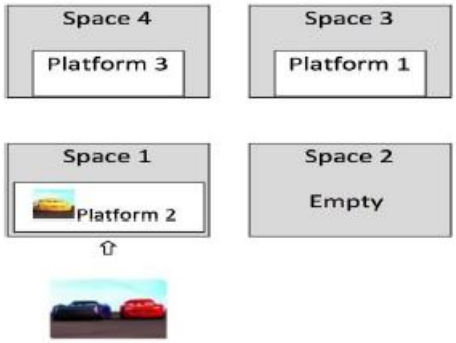
Χώρος καταστάσεων

Ως χώρος καταστάσεων (state space) ορίζεται το σύνολο των έγκυρων καταστάσεων που μπορεί να βρεθεί ένα πρόβλημα κατά την εξέλιξη του κόσμου του. Για παράδειγμα, ως έγκυρη κατάσταση μπορεί να είναι η είσοδος αυτοκινήτου σε άδεια πλατφόρμα που βρίσκεται στο χώρο υποδοχής. Αντίστοιχα, ως μη έγκυρη κατάσταση μπορούμε να ορίσουμε την αντιμετάθεση πλατφορμών που βρίσκονται διαγώνια η μια στην άλλη.

Ενδεικτικές έγκυρες καταστάσεις:



Εικόνα 1: Παράδειγμα κατάστασης 1



Εικόνα 2: Παράδειγμα κατάστασης 2

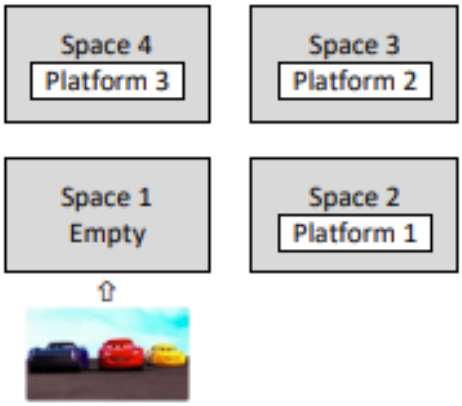
Αρχική κατάσταση

Στην αρχική κατάσταση του προβλήματος, το parking είναι άδειο από αυτοκίνητα δηλαδή όλες οι πλατφόρμες είναι κενές και στον χώρο υποδοχής δεν υπάρχει πλατφόρμα εναπόθεσης.

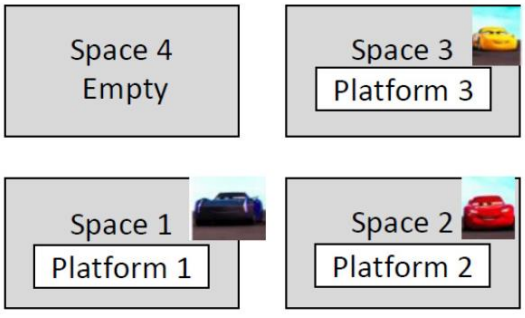
```
[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]
```

Στόχος – Τελική Κατάσταση

Η τελική κατάσταση του προβλήματος είναι η εισαγωγή όλων των αυτοκινήτων που βρίσκονται σε αναμονή στο parking.



Εικόνα 4: Αρχική Κατάσταση όπως δίνεται στο πρόβλημα



Εικόνα 4: Μια κατάσταση που μπορεί να είναι και κατάσταση στόχου

Τελεστές Μετάβασης

Οι τελεστές μετάβασης αποτελούν τις ενέργειες που μπορούν να πραγματοποιηθούν κατά τη διάρκεια επίλυσης του προβλήματος με σκοπό η τρέχουσα κατάσταση του προβλήματος να εξελιχθεί σε μία νέα κατάσταση του κόσμου του προβλήματος.
Στο πρόβλημα μας έχουμε τρεις τελεστές μετάβασης:

> Είσοδος αυτοκινήτου σε πλατφόρμα στο χώρο υποδοχής – **enter**
>
> Κίνηση 1ης πλατφόρμας στην γειτονική κενή της θέσης – **neighbour1**
>
> Κίνηση 2ης πλατφόρμας στην γειτονική κενή της θέσης – **neighbour2**

Τελεστής enter

Η ενέργεια «enter» είναι πραγματοποιήσιμη μόνο

- αν υπάρχει αυτοκίνητο προς εισαγωγή
- και αν υπάρχει άδεια πλατφόρμα στο χώρο υποδοχής (space 1)

Αποτελέσματα ενέργειας του τελεστή:

- Μείωση κατά ενός αριθμού αυτοκινήτων που περιμένουν να εισαχθούν.
- Η πλατφόρμα που βρίσκεται στο space 1 θα καταληφθεί από ένα αυτοκίνητο

Τελεστής neighbour 1

Αρχικά θεωρούμε ως γειτονικές πλατφόρμες της «empty», τις πλατφόρμες που βρίσκονται σε χώρο στάθμευσης είτε πάνω, είτε κάτω, είτε αριστερά, είτε δεξιά (δηλαδή ένας διαγώνιος χώρος στάθμευσης, δε θεωρείται γειτονικός).

Η ενέργεια neighbour1 είναι επιτεύξιμη μόνο αν έχει τουλάχιστον 1 γείτονα η «empty». Αν υπάρχει γείτονας τότε γίνεται αντιμετάθεση θέσεων του «empty» και της πλατφόρμας που βρίσκεται στο γειτονικό του χώρο στάθμευσης.

Αποτελέσματα ενέργειας του τελεστή:

- Μετακίνηση του empty σε γειτονικό χώρο στάθμευσης
- Μετακίνηση της πλατφόρμας που βρίσκεται στο γειτονικό χώρο του empty, στη θέση του empty που βρισκόταν

Τελεστής neighbour 2

Όπως και ο τελεστής neighbour 1, ο neighbour 2 λειτουργεί μόνο αν υπάρχει 2ος γείτονας.

Αποτελέσματα ενέργειας του τελεστή:

- Μετακίνηση του empty στο 2ο γειτονικό χώρο στάθμευσης
- Μετακίνηση της πλατφόρμας που βρίσκεται στο γειτονικό χώρο του empty, στη θέση του empty που βρισκόταν

Κωδικοποίηση του προβλήματος

Στις παρακάτω ενότητες θα παρατεθεί ο κώδικας του προβλήματος με τις απαραίτητες επεξηγήσεις.

Λίστα γειτνίασης

Αρχικά, στο πρόγραμμα μας βλέπουμε ότι δημιουργούμε μια λίστα γειτνίασης που ουσιαστικά μας δείχνει κάθε χώρος στάθμευσης με ποιους γειτονεύει.

```
# **** The Parking Spaces Diagram
# **** Διάγραμμα των Χώρων του Πάρκινγκ
#
#   +-------+-------+
#   |   4   |   3   |
#   +-------+-------+
#   |   1   |   2   |
#   +-------+-------+
#       ^
#    entrance

spaces = {
    1: [2,4],
    2: [1,3],
    3: [2,4],
    4: [1,3],
}

# Ο space 1 έχει πρώτο γείτονα τον space 2 και ως δεύτερο γείτονα τον space 2
# Ο space 2 έχει πρώτο γείτονα τον space 1 και ως δεύτερο γείτονα τον space 3
# Ο space 3 έχει πρώτο γείτονα τον space 2 και ως δεύτερο γείτονα τον space 4
# Ο space 4 έχει πρώτο γείτονα τον space 1 και ως δεύτερο γείτονα τον space 3
```

Ορισμός κατάστασης

Μια κατάσταση αναπαρίσταται με μια λίστα, η οποία περιέχει ως πρώτο στοιχείο το πλήθος των αυτοκινήτων που περιμένουν να εισέλθουν στο parking. Τα υπόλοιπα στοιχεία της είναι λίστες οι οποίες αποτελούν χώρους στάθμευσης με τη μόνη εξαίρεση ότι το 2ο στοιχείο της λίστας, δηλαδή ο χώρος στάθμευσης 1 αποτελεί και τον χώρο εισόδου. Οι εσωτερικές λίστες περιέχουν ως πρώτο στοιχείο είτε "Ε" ο οποίος συμβολίζει ότι ο χώρος είναι κενός (δηλαδή χωρίς πλατφόρμα), είτε "P1","P2","P3" οι οποίοι αντιπροσωπεύουν ποια πλατφόρμα περιέχεται σε κάποιο χώρο. Ως δεύτερο στοιχείο των εσωτερικών λιστών μπορούν να έχουν "NO" που σημαίνει ότι δεν υπάρχει αυτοκίνητο σε αυτό το χώρο στάθμευσης. Αντίστοιχα μπορούν να έχουν "YES" σε καταστάσεις όπου περιέχεται αυτοκίνητο

Για παράδειγμα:

```
state= [1, ['P2', 'YES'], ['P1', 'YES'], ['E', 'NO'], ['P3', 'NO']]
```

- 1 αυτοκίνητο περιμένει να μπει μέσα στο parking
- Ο space 1 περιέχει την πλατφόρμα P2 και είναι κατειλημμένος με ένα αυτοκίνητο
- Ο space 2 περιέχει την πλατφόρμα P1 και είναι κατειλημμένος με ένα αυτοκίνητο
- Ο space 3 είναι άδειος και δεν περιέχει αυτοκίνητα
- Ο space 4 περιέχει την πλατφόρμα P3 και δεν περιέχει κάποιο αυτοκίνητο

Ορισμός αρχικής κατάστασης

Έπειτα ορίζεται η αρχική κατάσταση (state) του προβλήματος.

```
# **** The Parking Initial State Diagram
# **** Διάγραμμα Αρχικής Κατάστασης του Πάρκινγκ
#
#   +--------+--------+
#   |  P3 NO |  P2 NO |
#   +--------+--------+
#   |   E    |  P1 NO |
#   +--------+--------+
#       ^
#   3 vehicles waiting

# **** The problem's initial state
state= [3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]

#Περιμένουν 3 αυτοκίνητα να μπουν στο πάρκινγ
#Ο space 1 είναι άδειος και δεν περιέχει αυτοκίνητα
#Ο space 2 περιέχει την πλατφόρμα P1 και δεν περιέχει κάποιο αυτοκίνητο
#Ο space 3 περιέχει την πλατφόρμα P2 και δεν περιέχει κάποιο αυτοκίνητο
#Ο space 4 περιέχει την πλατφόρμα P3 και δεν περιέχει κάποιο αυτοκίνητο
```

Τελεστές μετάβασης

Τελεστής enter

Αρχικά, θα δούμε τον τελεστή enter που επιτρέπει την είσοδο ενός αυτοκινήτου και το τοποθετεί σε άδεια πλατφόρμα στον χώρο εισόδου εφόσον ο αριθμός των αυτοκινήτων δεν είναι μηδενικός, υπάρχει πλατφόρμα στο 1ο χώρο στάθμευσης και αυτός ο χώρος δεν περιέχει άλλο αυτοκίνητο.

```
def enter(state):
    if state[0]!=0 and state[1][0][0]=='P' and state[1][1]=='NO': # υπάρχει πλατφόρμα στο χώρο εισόδου χωρίς αυτοκίνητο (NO)
        new_state=[state[0]-1] + [[state[1][0], 'YES']] + state[2:] # είσοδος αυτοκινήτου στο parking
        return new_state
```

Επεξήγηση κώδικα:

1. Η συνάρτηση παίρνει ως όρισμα μια λίστα που αντιπροσωπεύει μια κατάσταση

2. **Αν** το πρώτο στοιχείο της λίστας είναι διάφορο του μηδενός ΚΑΙ
   η πρώτη εσωτερική λίστα της έχει ως πρώτο στοιχείο ως πρώτο γράμμα P ΚΑΙ
   ως δεύτερο στοιχείο της το αλφαριθμητικό NO, **ΤΟΤΕ**

3. Να δημιουργηθεί μια νέα λίστα η οποία έχει ως
   - πρώτο στοιχείο τον αριθμό των αυτοκινήτων που περίμεναν έξω από το πάρκινγκ μειωμένος κατά έναν αριθμό.
   - δεύτερο στοιχείο θα έχει μια λίστα η οποία έχει ως πρώτο στοιχείο τα ίδια περιεχόμενα της προηγούμενης λίστας και ως δεύτερο στοιχείο της "YES"
   - Τα υπόλοιπα στοιχεία της νέας λίστας θα είναι ίδια με αυτά τα υπόλοιπα στοιχεία της προηγούμενης λίστας

4. Τέλος επιστρέφει τη νέα κατάσταση στο σημείο του κώδικα όπου έγινε η κλήση της συνάρτησης enter(state).

Πίνακας με διάφορες καταστάσεις και πότε ο τελεστής "enter" μπορεί να εφαρμοστεί

Καταστάσεις	Έγκυρη
[2, ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO']]	ΟΧΙ
[1, ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES'], ['E', 'NO']]	ΟΧΙ
[[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']]	ΝΑΙ
[0, ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES']]	ΟΧΙ

Τελεστής neighbour 1

Ο τελεστής neighbour1() εξυπηρετεί την μετακίνηση 1ης πλατφόρμας που συνορεύει με κενό χώρο προς το γειτονικό της κενό χώρο.

```
def neighbour1(state):

    elem=['E','NO']
    i=state.index(elem) if elem in state else -1
    if i >=0:
        swap(state, i, spaces[i][0])
        return state
```

Επεξήγηση κώδικα:

1. Η συνάρτηση παίρνει ως όρισμα μια λίστα που αντιπροσωπεύει μια κατάσταση
2. Θέτει μια μεταβλητή λίστα elem με στοιχεία ['E', 'NO']
3. Αν βρει στη λίστα-state το στοιχείο elem, αποθηκεύει σε μια μεταβλητή i τη θέση όπου εντοπίστηκε και
4. ΤΟΤΕ αν το i είναι μεγαλύτερο από το 0 να γίνει αντιμετάθεση του empty με την πλατφόρμα που βρίσκεται στον πρώτο γείτονα/χώρο στάθμευσης
5. Τέλος επιστρέφει τη νέα κατάσταση στο σημείο του κώδικα όπου έγινε η κλήση της συνάρτησης neighbour1(state).

Πίνακας με διάφορες καταστάσεις και πότε ο τελεστής "neighbour1" μπορεί να εφαρμοστεί

Καταστάσεις	Έγκυρη
`[1, ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES'], ['P5', 'NO']]`	**ΟΧΙ**
`[[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']]`	**ΝΑΙ**

Τελεστής neighbour 2

Αντιστοίχως, ο τελεστής neighbour2() εξυπηρετεί την μετακίνηση 2ης πλατφόρμας που συνορεύει με κενό χώρο προς το γειτονικό της κενό χώρο.

```
def neighbour2(state):

    elem=['E','NO']
    i=state.index(elem) if elem in state else -1
    if i >=0:
        swap(state, i, spaces[i][1])
        return state
```

Επεξήγηση κώδικα:

Ο κώδικας λειτουργεί ακριβώς με τον ίδιο τρόπο όπως η συνάρτηση neighbour1(state), με τη μόνη διαφορά στο βήμα 4, όπου η αντιμετάθεση γίνεται μεταξύ του δεύτερο γείτονά του.

Πίνακας με διάφορες καταστάσεις και πότε ο τελεστής "neighbour2" μπορεί να εφαρμοστεί

Καταστάσεις	Έγκυρη

[1, ['P2', 'YES'], ['E', 'NO']
 	
Και
 	
spaces {
1: [2]
2: [1]
}
	ΟΧΙ

[[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']]	ΝΑΙ
[2, ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES'], ['P4', 'NO']]	ΟΧΙ

Κωδικοποίηση της συνάρτησης find_children()

Η συνάρτηση find_children() σκοπό έχει την εύρεση απογόνων της τρέχουσας κατάστασης.

```
def find_children(state):

    children=[]

    enter_state=copy.deepcopy(state)
    enter_child=enter(enter_state)

    tr1_state=copy.deepcopy(state)
    tr1_child=neighbour1(tr1_state)

    tr2_state=copy.deepcopy(state)
    tr2_child=neighbour2(tr2_state)

    if tr1_child is not None:
        children.append(tr1_child)

    if tr2_child is not None:
        children.append(tr2_child)

    if enter_child is not None:
        children.append(enter_child)

    return children
```

Επεξήγηση κώδικα:

1. Η συνάρτηση παίρνει ως όρισμα μια λίστα (state).
2. Δημιουργεί μια κενή λίστα με όνομα children[] στην οποία θα αποθηκευτούν τα παιδιά της τρέχουσας κατάστασης.
3. Αντιγράφει τη λίστα με όνομα enter_state
   - Ύστερα καλεί τη συνάρτηση enter και της περνάει ως παράμετρο το αντίγραφο της λίστας. Σε αυτό το σημείο θα της επιστραφεί μια τιμή σύμφωνα με αυτά που επιστρέφει η ανίστοιχη καλούσα συνάρτηση.
4. Αντιγράφει τη λίστα με όνομα tr1_state
   - Ύστερα καλεί τη συνάρτηση neighbour1 και της περνάει ως παράμετρο το αντίγραφο της λίστας. Σε αυτό το σημείο θα της επιστραφεί μια τιμή.
5. Αντιγράφει τη λίστα με όνομα tr2_state
   - Ύστερα καλεί τη συνάρτηση neighbour2 και της περνάει ως παράμετρο το αντίγραφο της λίστας. Σε αυτό το σημείο θα της επιστραφεί μια τιμή.
6. Μετά ακολουθούν εντολές ελέγχου
   - Αν η tr1_child δεν είναι κενή τότε πρόσθεσε στη λίστα children[] τα περιεχόμενά της.
   - Αν η tr2_child δεν είναι κενή τότε πρόσθεσε στη λίστα children[] τα περιεχόμενά της.
   - Αν η enter_child δεν είναι κενή τότε πρόσθεσε στη λίστα children[] τα περιεχόμενά της.
7. Τέλος επιστρέφει τη λίστα children[] στο σημείο όπου έγινε η κλήση της.

Διαχείριση Μετώπου

Αρχικοποίηση Μετώπου

Το μέτωπο αρχικοποιείται με την αρχική κατάσταση του προβλήματος.

```
def make_front(state):
    return [state]
```

Επέκταση Μετώπου

Γίνεται επέκταση μετώπου με βάση τον αλγόριθμο αναζήτησης που έχουμε επιλέξει, στη προκειμένη περίπτωση τον αλγόριθμο DFS – Πρώτα σε Βάθος. Βρίσκουμε τους απογόνους της δοθείσας κατάστασης καλώντας τη συνάρτηση find_children() και τους προσθέτουμε στο μέτωπο αναζήτησης.

```
# παίρνει ως όρισμα όλο το τρέχον μέτωπο και τη μέθοδο που έχουμε διαλέξει
από τη main()
def expand_front(front, method):
    if method=='DFS':
        if front:
            print("Front:")
            print(front) # τρέχουσα κατάσταση
            node=front.pop(0) # αφαίρεσε το πρώτο στοιχείο του μετώπου
            for child in find_children(node):
# Βάλε κάθε παιδί που περιέχει η λίστα children[] στο μέτωπο στην πρώτη θέση
                front.insert(0,child)

    return front
```

Παράδειγμα επέκτασης ουράς με τη μέθοδο DFS :

1. Έστω **queue** = `[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]`
2. **node** = `[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]`
   queue = []
   children = `[[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],`
   `[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]`
   path = **node**

3. ***1η επανάληψη***

   node = `[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]`
   path = `[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],`
   `[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]`
   queue = `[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],`
   `[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]`

2η επανάληψη

node = `[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]`
path = `[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],`
`[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]`
queue = `[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],`
`[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E','NO']]],`
`[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],`
`[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]`

Συνάρτηση find_solution()

Η find_solution() είναι αναδρομική συνάρτηση που δημιουργεί το δένδρο αναζήτησης, γίνεται δηλαδή αναδρομική επέκταση του δένδρου με βάση τον αλγόριθμο DFS – Πρώτα σε Βάθος.

```
def find_solution(front, closed, goal, method):  # Δέχεται ως όρισμα το μια λίστα - τρεχόν μέτωπο, μια λίστα - closed [], μια λίστα - goal κατάσταση, τη μέθοδο

    if not front: # Αν είναι άδειο το μέτωπο
        print('_NO_SOLUTION_FOUND_') # δε βρέθηκε λύση

    elif front[0] in closed: # αν η πρώτη κατάσταση στο τρέχον μέτωπο υπάρχει στη λίστα closed[]
        new_front=copy.deepcopy(front) # αντιγραφή front στο new_front
        new_front.pop(0) #αφαίρεση του πρώτου στοιχείου από το new_front
        find_solution(new_front, closed, goal, method) # αναδρομική κλήση

    elif front[0]== goal: # Αν το η πρώτη κατάσταση στο τρέχον μέτωπο είναι η τελική κατάσταση
        print("Front:")
        print(front)
        print("\n")
        print('_GOAL_FOUND_')
        print(front[0]) #Εκτύπωσε την πρώτη κατάσταση η οποία ταυτίζεται με την τελική κατάσταση

    else: # Αν δεν ισχύει τίποτα από τα παραπάνω
        closed.append(front[0]) # Πρόσθεσε στο τέλος της λίστας-closed, την πρώτη κατάσταση (πρώτο στοιχείο) στο τρέχον μέτωπο
        front_copy=copy.deepcopy(front)  # Αντιγραφή του front στο front_copy
        front_children=expand_front(front_copy, method) # Κλήση της συνάρτησης expand_front και επιστροφή τιμή της στην front children
        closed_copy=copy.deepcopy(closed)
        find_solution(front_children, closed_copy, goal, method) # Αναδρομική κλήση με καινούρια δεδομένα
```

Συνάρτηση find_solution() με χρήση is_goal_state()

Βελτιστοποίηση της συνάρτησης find_solution() με την χρήση της συνάρτησης is_goal_state, η οποία βρίσκει αν το πρώτο στοιχείο της τρέχουσα κατάστασης ισούται με μηδέν, δηλαδή αν δεν έχουν μείνει αυτοκίνητα που περιμένουν να παρκάρουν, αντί να γίνεται χειροκίνητη ανάθεση τελικής κατάστασης σε μια μεταβλητή goal.

```
def is_goal_state(front):

    if front[0] == 0:
        return 1
    else:
        return 0


def find_solution(front, closed, method):

    if not front:
        print('_NO_SOLUTION_FOUND_')

    elif front[0] in closed:
        new_front=copy.deepcopy(front)
        new_front.pop(0)
        find_solution(new_front, closed, method)
    elif is_goal_state(front[0]):
        print('_GOAL_FOUND_')
        print(front[0])

    else:
        closed.append(front[0])
        front_copy=copy.deepcopy(front)
        front_children=expand_front(front_copy, method)
        closed_copy=copy.deepcopy(closed)
        find_solution(front_children, closed_copy, method)
```

Επεξήγηση κώδικα:

Η συνάρτηση λειτουργεί όπως και **find_solution**(front, closed, goal, method). Η διαφορά παρατηρείται στο ότι δε δέχεται ως όρισμα τη λίστα goal. Γίνεται η κλήση της συνάρτησης **is_goal_state**(front).

Η **is_goal_state**(front) συνάρτηση δέχεται ως όρισμα μια λίστα-κατάσταση, η οποία κατάσταση είναι το πρώτο στοιχείο του μετώπου. Η λειτουργεία της είναι να ελέγξει αν υπάρχουν αυτοκίνητα που περιμένουν να μπουν μέσα στο πάρκινγκ. Ελέγχει δηλαδή αν το πρώτο στοιχείο της λίστας-κατάστασης είναι ίσο με το 0. Αν αυτός ο αριθμός ισούται με το μηδέν, τότε το πρόβλημά μας λύθηκε με την κατάσταση ως κατάσταση στόχου.

Κλήση εκτέλεσης κώδικα

Στη main() δίνουμε την αρχική κατάσταση, τον τελικό μας στόχο καθώς και τον τρόπο αναζήτησης που θέλουμε να ακολουθήσουμε και καλούμε τη συνάρτηση find_solution() ώστε να ξεκινήσει η αναζήτηση.

```
def main():

    initial_state = [3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3',
'NO']]
    goal = [0,['P1','YES'],['E','NO'],['P2','YES'],['P3','YES']]
    method='DFS'

    """ ----------------------------------------------------------------
    ** starting search
    ** έναρξη αναζήτησης
    """

    print('_____BEGIN__SEARCHING_____')
    find_solution(make_front(initial_state), [], goal, method)

if __name__ == "__main__":
    main()
```

Κλήση εκτέλεσης κώδικα όταν γίνεται χρήση της is_goal_state()

Στη προκειμένη περίπτωση δεν χρειάζεται να ορίσουμε τον τελικό μας στόχο και έτσι δεν υπάρχει και σαν όρισμα στη συνάρτηση find_solution().

```
def main():

    initial_state = [3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3',
'NO']]
    method='DFS'

    """ ----------------------------------------------------------------
    ** starting search
    ** έναρξη αναζήτησης
    """

    print('_____BEGIN__SEARCHING_____')
    find_solution(make_front(initial_state), [], method)

if __name__ == "__main__":
    main()
```

Αποτελέσματα αλγορίθμου για διαφορετικές αρχικές καταστάσεις

Παρακάτω βλέπουμε ενδεικτικά αποτελέσματα του αλγόριθμου (DFS) ***χωρίς*** και ***με χρήση*** της συνάρτησης is_goal_state().

Κατάσταση 1 – χωρίς is_goal_state

```
initial_state = [3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]
goal = [0,['P1','YES'],['E','NO'],['P2','YES'],['P3','YES']]

_GOAL_FOUND_
[0, ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES']]
```

Κατάσταση 1 – με is_goal_state

```
initial_state = [3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]

_GOAL_FOUND_
[0, ['P1', 'YES'], ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO']]
```

Κατάσταση 2 – χωρίς is_goal_state

```
initial_state = [3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]
goal = [0,['P1','YES'],['E','NO'],['P2','YES'],['P3','YES']]

_GOAL_FOUND_
[0, ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES']]
```

Κατάσταση 2 – με is_goal_state

```
initial_state = [3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]

_GOAL_FOUND_
[0, ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'YES']]
```

Κατάσταση 3 – χωρίς is_goal_state

```
initial_state = [3, ['E', 'YES'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]
goal = [0,['P1','NO'],['E','YES'],['P2','YES'],['P3','YES']]

_NO_SOLUTION_FOUND_
```

Κατάσταση 3 – με is_goal_state

```
initial_state = [3, ['E', 'YES'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]

_NO_SOLUTION_FOUND_
```

Εναλλαγή της σειράς των τελεστών στη find_children()

Με σκοπό την καλύτερη διερεύνηση του προβλήματος καθώς και τη σύγκριση αποτελεσμάτων, έγινε εναλλαγή της σειρά των τελεστών στη συνάρτηση find_children(). Αντί να γίνεται πρώτα χρήση του τελεστή enter και μετά του neighbour1 και neighbour2 όπως στον αρχικό μας κώδικα, πλέον η σειρά των τελεστών είναι neighbour2, enter και neighbour1.

Παρακάτω βλέπουμε τις αλλαγές που πραγματοποιήθηκαν στο κώδικα.

```
def find_children(state):

    children=[]

    tr2_state=copy.deepcopy(state)
    tr2_child=neighbour2(tr2_state)

    enter_state=copy.deepcopy(state)
    enter_child=enter(enter_state)

    tr1_state=copy.deepcopy(state)
    tr1_child=neighbour1(tr1_state)

    if tr2_child is not None:
        children.append(tr2_child)

    if enter_child is not None:
        children.append(enter_child)

    if tr1_child is not None:
        children.append(tr1_child)


    return children
```

Ενδεικτικά τρεξίματα και παρατήρηση αλλαγών

Εναλλάσσοντας τη σειρά των τελεστών παρατηρούμε ότι η τελική κατάσταση διαφέρει συγκριτικά με τις τελικές καταστάσεις που προέκυψαν από τον αρχικό κώδικα. Επιπρόσθετα, παρατηρούμε ότι οι αλλαγές προκύπτουν όταν υπάρχει χρήση της συνάρτησης is_goal_state() καθώς δεν είναι προκαθορισμένη η τελική κατάσταση από τον χρήστη.

Κατάσταση 1 – χωρίς is_goal_state

```
initial_state = [3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]
goal = [0,['P1','YES'],['E','NO'],['P2','YES'],['P3','YES']]

_GOAL_FOUND_
[0, ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES']]
```

Κατάσταση 1 – με is_goal_state

```
initial_state = [3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]

_GOAL_FOUND_
[0, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P1', 'YES']]
```

Διαφορές

```
_GOAL_FOUND_
[0, ['P1', 'YES'], ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO']]
```

Κατάσταση 2 – χωρίς is_goal_state

```
initial_state = [3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]
goal = [0,['P1','YES'],['E','NO'],['P2','YES'],['P3','YES']]

_GOAL_FOUND_
[0, ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES']]
```

Κατάσταση 2 – με is_goal_state

```
initial_state = [3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]

_GOAL_FOUND_
[0, ['P2', 'YES'], ['P3', 'YES'], ['P1', 'YES'], ['E', 'NO']]
```

Διαφορές

```
_GOAL_FOUND_
[0, ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'YES']]
```

Κατάσταση 3 – χωρίς is_goal_state

```
initial_state = [3, ['E', 'YES'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]
goal = [0,['P1','NO'],['E','YES'],['P2','YES'],['P3','YES']]

_NO_SOLUTION_FOUND_
```

Κατάσταση 3 – με is_goal_state

```
initial_state = [3, ['E', 'YES'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]

_NO_SOLUTION_FOUND_
```

Επέκταση προβλήματος

Παρακάτω παρατίθεται ο κώδικας του προβλήματος για 6 θέσεις στάθμευσης με 5 πλατφόρμες εναπόθεσης αυτοκινήτων. Ώστε να μπορέσει να γίνει η επέκταση του προβλήματος.
Έγιναν οι εξής αλλαγές στον κώδικα:

Αρχικά έγινε αλλαγή στη λίστα γειτνίασης, προστέθηκαν οι καινούριοι γείτονες και ανανεώθηκαν οι παλιοί. Μπορούμε να παρατηρήσουμε ότι πλέον οι θέσεις 3 και 6 έχουν τρεις γείτονες ο καθένας.

```
spaces = {
    1: [2, 6],
    2: [1, 3],
    3: [2, 4, 6],
    4: [3, 5],
    5: [4, 6],
    6: [1, 3, 5]
}
```

Έπειτα έγινε αλλαγή της αρχικής κατάστασης του προβλήματος και προστέθηκαν οι καινούριες θέσεις καθώς και μεγάλωσε ο αριθμός αυτοκινήτων που περιμένουν να εισέλθουν στο parking.

```
# **** The Parking Initial State Diagram
# **** Διάγραμμα Αρχικής Κατάστασης του Πάρκινγκ
#   +--------+--------+
#   |  P4 NO |  P3 NO |
#   +--------+--------+
#   |  P5 NO |  P2 NO |
#   +--------+--------+
#   |   E    |  P1 NO |
#   +--------+--------+
#       ^
#   5 vehicles waiting

initial_state = [5, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO'],
['P4','NO'],['P5','NO']]
```

Έγινε προσθήκη ενός καινούριου τελεστή, ο neighbour3 εξυπηρετεί την μετακίνηση 3ης πλατφόρμας που συνορεύει με κενό χώρο προς το γειτονικό της κενό χώρο. Προϋπόθεση για να τρέξει ο τελεστής μετάβασης είναι το μήκος της λίστας των γειτόνων του στοιχείου να είναι ίσο με 3, δηλαδή να έχει τρεις γείτονες.

```
def neighbour3(state):

    elem=['E','NO']
    i=state.index(elem) if elem in state else -1
    if i >=0 and len(spaces[i]) == 3:
        swap(state, i, spaces[i][2])
        return state
```

Τέλος, έγινε αλλαγή στη συνάρτηση εύρεσης απογόνων, find_children(), προστέθηκε δηλαδή ο τελεστής neighbour3 στον αντίστοιχο κώδικα.

```
def find_children(state):

    children=[]

    enter_state=copy.deepcopy(state)
    enter_child=enter(enter_state)

    tr1_state=copy.deepcopy(state)
    tr1_child=neighbour1(tr1_state)

    tr2_state=copy.deepcopy(state)
    tr2_child=neighbour2(tr2_state)

    tr3_state=copy.deepcopy(state)
    tr3_child=neighbour3(tr3_state)

    if tr1_child is not None:
        children.append(tr1_child)

    if tr2_child is not None:
        children.append(tr2_child)

    if tr3_child is not None:
        children.append(tr3_child)

    if enter_child is not None:
        children.append(enter_child)

    return children
```

Αποτελέσματα

Τρέχοντας τον αλγόριθμο με την αρχική κατάσταση που θέσαμε προηγουμένως, βλέπουμε ότι υπάρχει λύση και ο αλγόριθμος λειτουργεί σωστά.

```
_GOAL_FOUND_
[0, ['P4', 'YES'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'YES'], ['P5', 'YES'],
['P3', 'YES']]
```

Αλγόριθμοι αναζήτησης και μελέτη αποτελεσμάτων

Τυφλοί Αλγόριθμοι Αναζήτησης

Οι τυφλές μέθοδοι αναζήτησης (blind search methods) εφαρμόζονται σε προβλήματα στα οποία είτε δεν μας ενδιαφέρει να βρεθεί μια ποιοτική λύση είτε δε διαθέτουν πληροφορίες που να επιτρέπουν στον αλγόριθμο αναζήτησης την επιλογή του ποιοτικά καλύτερου κόμβου μέσα από το μέτωπο αναζήτησης, όταν αυτός επιχειρεί να αναπτύξει το δένδρο αναζήτησης.

Στις παρακάτω ενότητες θα αναλύσουμε τους τυφλούς αλγόριθμους αναζήτησης, DFS (Depth-First Search) και BFS (Breadth-First Search).

Μελέτη DFS

Η Πρώτα σε Βάθος Αναζήτηση (Depth-First Search - DFS) επεκτείνει κάθε φορά το αριστερότερο από τα υπάρχοντα μονοπάτια του δένδρου, αρχίζοντας από το μονοπάτι που περιέχει τη ρίζα του δένδρου. Ουσιαστικά, επεκτείνει το μέτωπο αναζήτησης προς αριστερά και προς το βάθος του δένδρου αναζήτησης. Η διαδικασία ολοκληρώνεται, όταν εντοπιστεί ένας κόμβος που αντιστοιχεί σε μια επιθυμητή τελική κατάσταση ή όταν εξαντληθεί η αναζήτηση, δηλαδή όταν όλα τα μονοπάτια καταλήξουν σε κόμβους που δεν μπορούν να επιλεγούν ως γονικοί κόμβοι, γιατί οδηγούν σε κύκλους.

Τα πλεονεκτήματα της DFS είναι ότι το μέτωπο της αναζήτησης δε μεγαλώνει πάρα πολύ καθώς και ότι εγγυάται πάντα την εύρεση λύσης, αν αυτή υπάρχει, με την προϋπόθεση ότι ο χώρος αναζήτησης είναι πεπερασμένος.

Τα μειονεκτήματα της DFS είναι ότι δεν υπάρχει εγγύηση ότι η πρώτη λύση που θα βρεθεί είναι η βέλτιστη ούτε εγγυάται την εύρεση λύσης, αν ο χώρος αναζήτησης δεν είναι πεπερασμένος.

Παρακολούθηση μετώπου

Η κωδικοποίηση του προβλήματος με τη χρήση του αλγόριθμου DFS με παρακολούθηση μετώπου έχει παρατεθεί σε προηγούμενη ενότητα.

Παρακάτω βλέπουμε τα μέτωπα που προκύπτουν με αρχική κατάσταση

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

Παρατηρούμε ότι, η τελική κατάσταση και λύση του προβλήματος είναι η πρώτη κατάσταση του τελευταίου μετώπου.

1ο Μέτωπο

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

2ο Μέτωπο

```
[[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']],
[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

3ο Μέτωπο

```
[[2, ['P3', 'YES'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']],
[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

4ο Μέτωπο

```
[[2, ['P3', 'YES'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']],
[2, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'YES']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']],
[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

•
•
•

Τελευταίο Μέτωπο - 2

```
[[1, ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES'], ['P1', 'NO']],
[1, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'YES']],
[2, ['P2', 'NO'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO']],
[2, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'NO']],
[2, ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO']],
[2, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'YES']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']],
[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

Τελευταίο Μέτωπο - 1

```
[[1, ['P1', 'NO'], ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO']],
[1, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO']],
[1, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'YES']],
[2, ['P2', 'NO'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO']],
[2, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'NO']],
[2, ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO']],
[2, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'YES']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']],
[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

Τελευταίο Μέτωπο

```
[[0, ['P1', 'YES'], ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO']],
[1, ['P1', 'NO'], ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES']],
[1, ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES'], ['P1', 'NO']],
[1, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO']],
[1, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'YES']],
[2, ['P2', 'NO'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO']],
[2, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'NO']],
[2, ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO']],
[2, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'YES']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']],
[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

DFS με δυνατότητα παράλληλης παρακολούθησης ουράς μονοπατιών

Παρακάτω βλέπουμε τον κώδικα που αναπτύχθηκε με σκοπό ο αλγόριθμος να έχει τη δυνατότητα παράλληλης παρακολούθησης της ουράς μονοπατιών. Προστέθηκαν οι συναρτήσεις make_queue(), η οποία αρχικοποιεί την ουρά με την αρχική κατάσταση, και η extend_queue() που επεκτείνει την ουρά και τέλος έγιναν αλλαγές στην ήδη υπάρχουσα συνάρτηση find_children().

```
def make_queue(state):
    return [[state]]

def extend_queue(queue, method):
    if method=='DFS':
        print("Queue:")
        print(queue)
        node=queue.pop(0)
        queue_copy=copy.deepcopy(queue)
        children=find_children(node[-1])
        for child in children:
            path=copy.deepcopy(node)
            path.append(child)
            queue_copy.insert(0,path)

    #elif method=='BFS':
    #elif method=='BestFS':
    #else: "other methods to be added"

    return queue_copy

def find_solution(front, queue, closed, method):

    if not front:
        print('_NO_SOLUTION_FOUND_')

    elif front[0] in closed:
        new_front=copy.deepcopy(front)
        new_front.pop(0)
        new_queue=copy.deepcopy(queue)
        new_queue.pop(0)
        find_solution(new_front, new_queue, closed, method)

    elif is_goal_state(front[0]):
        print('_GOAL_FOUND_')
        print(queue[0])

    else:
        closed.append(front[0])
        front_copy=copy.deepcopy(front)
        front_children=expand_front(front_copy, method)
        queue_copy=copy.deepcopy(queue)
        queue_children=extend_queue(queue_copy, method)
        closed_copy=copy.deepcopy(closed)
        find_solution(front_children, queue_children, closed_copy, method)
```

Παρακολούθηση DFS ουράς μονοπατιών

Παρακάτω βλέπουμε τις ουρές μονοπατιών που προκύπτουν με αρχική κατάσταση.

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

Παρατηρούμε ότι, η τελική κατάσταση και λύση του προβλήματος είναι η τελευταία κατάσταση της τελικής ουράς.

1ο Μονοπάτι ουράς

```
[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]
```

2ο Μονοπάτι ουράς

```
[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3', 'NO'],
['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]],
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P1', 'NO'],
['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]
```

3ο Μονοπάτι ουράς

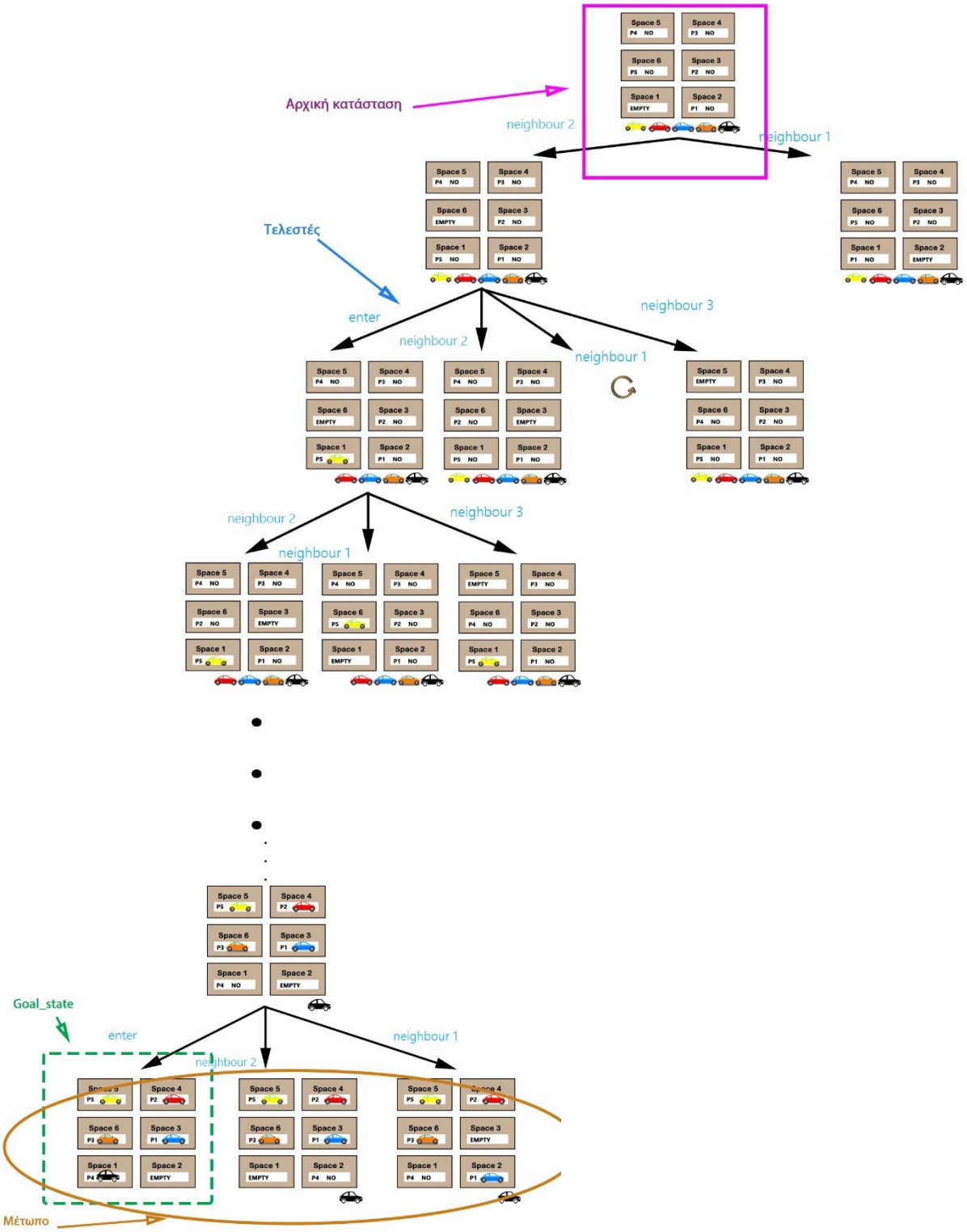
```
[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3', 'NO'],
['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']], [2, ['P3', 'YES'], ['P1', 'NO'],
['P2', 'NO'], ['E', 'NO']]], [[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'],
['P3', 'NO']], [3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']], [3,
['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']]],
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3', 'NO'],
['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']], [3, ['E', 'NO'], ['P1', 'NO'], ['P2',
'NO'], ['P3', 'NO']]], [[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3',
'NO']], [3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]
```

•
•
•

Ουρά Goal State

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']],
[2, ['P3', 'YES'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']],
[2, ['P3', 'YES'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']],
[2, ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO']],
[2, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'NO']],
[2, ['P2', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['E', 'NO']],
[1, ['P2', 'YES'], ['P3', 'YES'], ['P1', 'NO'], ['E', 'NO']],
[1, ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO']],
[1, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO']],
[1, ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES'], ['P1', 'NO']],
[1, ['P1', 'NO'], ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO']],
[0, ['P1', 'YES'], ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO']]]
```

Ανάπτυξη δένδρου - DFS



Μελέτη BFS

Η Πρώτα σε Πλάτος Αναζήτηση (Breadth-First Search - BFS) εξετάζει πρώτα τα μονοπάτια που βρίσκονται στο ίδιο βάθος του δένδρου αναζήτησης. Μόνο όταν τα εξετάσει όλα, αρχίζει την επέκτασή τους στο αμέσως επόμενο επίπεδο. Ουσιαστικά, επεκτείνει το μέτωπο αναζήτησης ανά επίπεδο του δένδρου αναζήτησης. Η διαδικασία ολοκληρώνεται, όταν εντοπιστεί ένας κόμβος που αντιστοιχεί σε μια επιθυμητή τελική κατάσταση ή όταν εξαντληθεί η αναζήτηση, δηλαδή όταν όλα τα μονοπάτια καταλήξουν σε κόμβους που δεν μπορούν να επιλεγούν ως γονικοί κόμβοι, γιατί οδηγούν σε κύκλους.

Τα Πλεονεκτήματα της BFS είναι ότι εγγυάται πάντα την εύρεση λύσης, αν αυτή υπάρχει, με την προϋπόθεση ότι ο χώρος αναζήτησης είναι πεπερασμένος καθώς και ότι βρίσκει πάντα τη συντομότερη λύση (μικρότερη σε μήκος μονοπατιού).

Τα Μειονεκτήματα της BFS είναι ότι το μέτωπο αναζήτησης μπορεί να γίνεται προοδευτικά πολύ μεγάλο. Αντίστοιχα, οι απαιτήσεις σε μνήμη για την αποθήκευση των μονοπατιών της ουράς μεγαλώνει εκθετικά, όσο βαθαίνει το μέτωπο αναζήτησης.

Το να κρίνουμε ποια από τις 2 τυφλές μεθόδους, DFS και BFS, είναι προσφορότερη εξαρτάται άμεσα από το πρόβλημα προς επίλυση. Γενικά, πρέπει να αποφεύγεται η BFS, όταν είναι γνωστό ότι οι υπάρχουσες λύσεις βρίσκονται σε μεγάλο βάθος. Αντίστοιχα, η χρήση της DFS είναι προτιμότερη, όταν ο στόχος βρίσκεται σε μονοπάτια μεγάλου μήκους.

Αναζήτηση BFS με μέτωπο

Η κωδικοποίηση του προβλήματος με χρήση BFS και παρακολούθηση μετώπου είναι η εξής:

```
def expand_front(front, method):
    if method=='DFS':
        if front:
            print("Front:")
            print(front)
            node=front.pop(0)
            for child in find_children(node):
                front.insert(0,child)

    elif method=='BFS':
        if front:
            print("Front:")
            print(front)
            node=front.pop(0)
            for child in find_children(node):
                front.append(child)
```

Παρατηρούμε ότι, η μόνη διαφορά της αναζήτησης BFS με τη DFS είναι ότι οι διάδοχοι κόμβοι προστίθενται στο τέλος του μετώπου, σε αντίθεση με τη μέθοδο αναζήτησης DFS που προστίθενται στην αρχή.

Παρακολούθηση μετώπου

Παρακάτω βλέπουμε τα μέτωπα που προκύπτουν με αρχική κατάσταση

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

Παρατηρούμε ότι, η τελική κατάσταση και λύση του προβλήματος είναι η πρώτη κατάσταση του τελευταίου μετώπου.

1ο Μέτωπο

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

2ο Μέτωπο

```
[[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]
```

3ο Μέτωπο

```
[[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']],
[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'NO']],
[2, ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

4ο Μέτωπο

```
[[3, ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'NO']],
[2, ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO']],
[2, ['P3', 'YES'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]
```

•
•
•

Τελευταίο Μέτωπο - 1

```
[[1, ['P2', 'NO'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES']],
[0, ['P2', 'YES'], ['P3', 'YES'], ['P1', 'YES'], ['E', 'NO']],
[2, ['P3', 'NO'], ['E', 'NO'], ['P1', 'NO'], ['P2', 'YES']],
[2, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'YES'], ['E', 'NO']],
[1, ['P3', 'YES'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'YES']],
[1, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'YES']],
[1, ['P3', 'YES'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'YES']],
[1, ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES'], ['P1', 'NO']],
[1, ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO']],
[1, ['E', 'NO'], ['P2', 'NO'], ['P3', 'YES'], ['P1', 'YES']],
[1, ['P2', 'NO'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES']],
[0, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P1', 'YES']],
[2, ['E', 'NO'], ['P1', 'NO'], ['P2', 'YES'], ['P3', 'NO']],
```

```
[2, ['P1', 'NO'], ['P2', 'YES'], ['E', 'NO'], ['P3', 'NO']],
[1, ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES'], ['P3', 'NO']],
[1, ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES'], ['P3', 'NO']],
[1, ['P1', 'YES'], ['P2', 'YES'], ['P3', 'NO'], ['E', 'NO']],
[1, ['P3', 'NO'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'YES']],
[1, ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES'], ['E', 'NO']]]
```

Τελευταίο Μέτωπο

```
[[0, ['P2', 'YES'], ['P3', 'YES'], ['P1', 'YES'], ['E', 'NO']],
[2, ['P3', 'NO'], ['E', 'NO'], ['P1', 'NO'], ['P2', 'YES']],
[2, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'YES'], ['E', 'NO']],
[1, ['P3', 'YES'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'YES']],
[1, ['E', 'NO'], ['P3', 'YES'], ['P1', 'NO'], ['P2', 'YES']],
[1, ['P3', 'YES'], ['P1', 'NO'], ['E', 'NO'], ['P2', 'YES']],
[1, ['E', 'NO'], ['P2', 'YES'], ['P3', 'YES'], ['P1', 'NO']],
[1, ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'NO']],
[1, ['E', 'NO'], ['P2', 'NO'], ['P3', 'YES'], ['P1', 'YES']],
[1, ['P2', 'NO'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES']],
[0, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P1', 'YES']],
[2, ['E', 'NO'], ['P1', 'NO'], ['P2', 'YES'], ['P3', 'NO']],
[2, ['P1', 'NO'], ['P2', 'YES'], ['E', 'NO'], ['P3', 'NO']],
[1, ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES'], ['P3', 'NO']],
[1, ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES'], ['P3', 'NO']],
[1, ['P1', 'YES'], ['P2', 'YES'], ['P3', 'NO'], ['E', 'NO']],
[1, ['P3', 'NO'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'YES']],
[1, ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES'], ['E', 'NO']],
[1, ['P2', 'NO'], ['E', 'NO'], ['P3', 'YES'], ['P1', 'YES']],
[1, ['P2', 'NO'], ['P3', 'YES'], ['P1', 'YES'], ['E', 'NO']],
[0, ['P2', 'YES'], ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES']]]
```

BFS με δυνατότητα παράλληλης παρακολούθησης ουράς μονοπατιών

Παρακάτω βλέπουμε τον κώδικα που αναπτύχθηκε με σκοπό ο αλγόριθμος να έχει τη δυνατότητα παράλληλης παρακολούθησης ουράς μονοπατιών. Παρατηρούμε ότι προστέθηκε στην extend_queue() κομμάτι κώδικα με σκοπό να μπορεί να γίνει παρακολούθηση ουράς και για τον αλγόριθμο BFS. Αντίστοιχα, όπως είδαμε και με τον κώδικα παρακολούθησης μετώπου, η μόνη διαφορά της αναζήτησης BFS με τη DFS είναι ότι οι διάδοχοι κόμβοι προστίθενται στο τέλος της ουράς, σε αντίθεση με τη μέθοδο αναζήτησης DFS που προστίθενται στην αρχή.

```
def extend_queue(queue, method):
    if method=='DFS':
        print("Queue:")
        print(queue)
        node=queue.pop(0)
        queue_copy=copy.deepcopy(queue)
        children=find_children(node[-1])
        for child in children:
            path=copy.deepcopy(node)
            path.append(child)
            queue_copy.insert(0,path)

    elif method=='BFS':
        print("Queue:")
        print(queue)
        node=queue.pop(0)
        queue_copy=copy.deepcopy(queue)
        children=find_children(node[-1])
        for child in children:
            path=copy.deepcopy(node)
            path.append(child)
            queue_copy.append(path)
```

Παρακολούθηση BFS ουράς μονοπατιών

Παρακάτω βλέπουμε τις ουρές μονοπατιών που προκύπτουν με αρχική κατάσταση

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

Παρατηρούμε ότι, η τελική κατάσταση και λύση του προβλήματος είναι η τελευταία κατάσταση της τελικής ουράς.

1ο Μονοπάτι ουράς

```
[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]
```

2ο Μονοπάτι ουράς

```
[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P1', 'NO'],
['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]],

[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3', 'NO'],
['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]]
```

3ο Μονοπάτι ουράς

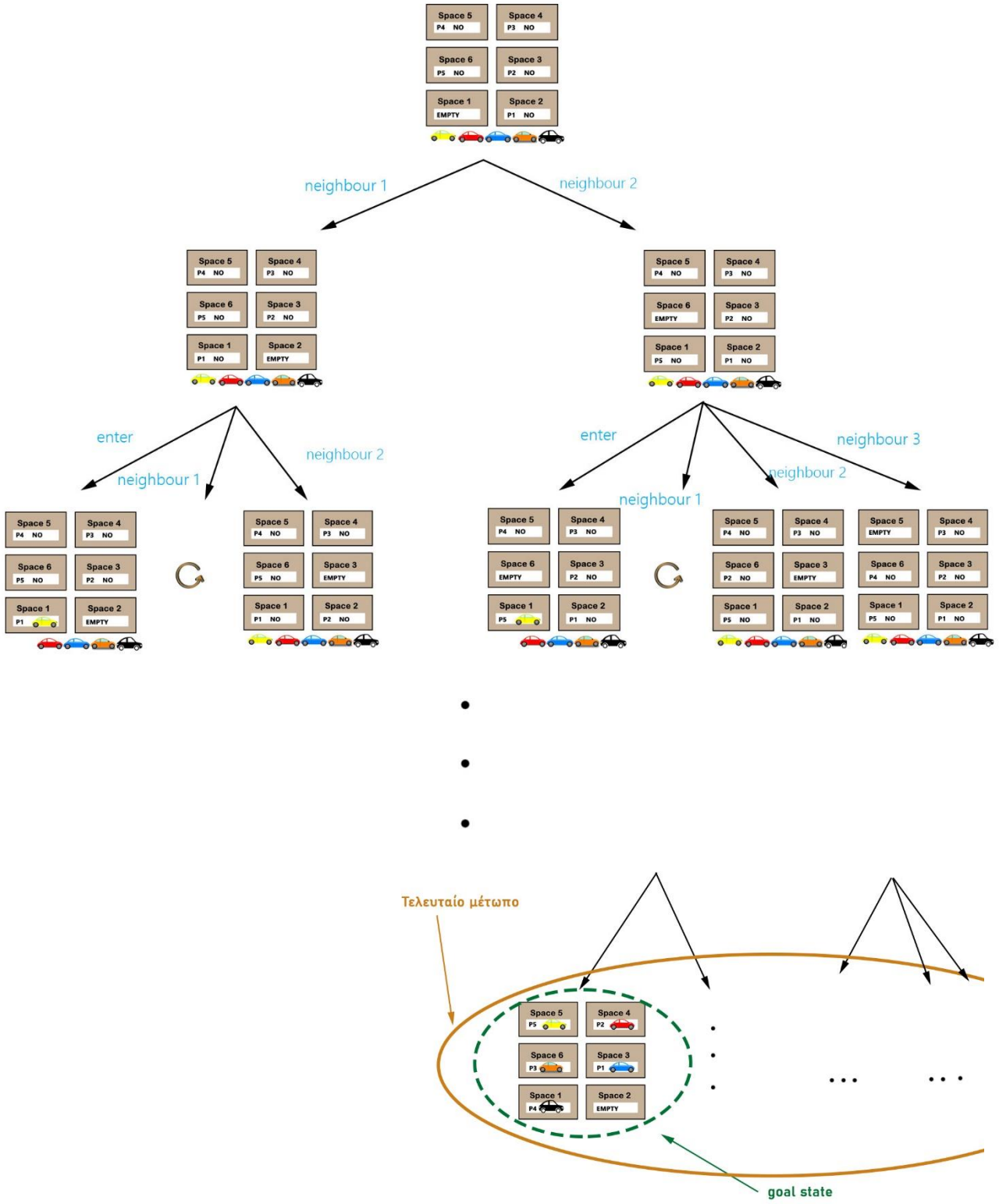
```
[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3', 'NO'],
['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]],

[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P1', 'NO'],
['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['E', 'NO'], ['P1', 'NO'], ['P2',
'NO'], ['P3', 'NO']]],

[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P1', 'NO'],
['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P1', 'NO'], ['P2', 'NO'], ['E',
'NO'], ['P3', 'NO']]],

[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P1', 'NO'],
['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [2, ['P1', 'YES'], ['E', 'NO'], ['P2',
'NO'], ['P3', 'NO']]]]
```

•
•
•

Ουρά Goal State

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[2, ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[2, ['E', 'NO'], ['P1', 'YES'], ['P2', 'NO'], ['P3', 'NO']],
[2, ['P3', 'NO'], ['P1', 'YES'], ['P2', 'NO'], ['E', 'NO']],
[2, ['P3', 'NO'], ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO']],
[2, ['P3', 'NO'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'NO']],
[1, ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'NO']],
[1, ['E', 'NO'], ['P3', 'YES'], ['P1', 'YES'], ['P2', 'NO']],
[1, ['P2', 'NO'], ['P3', 'YES'], ['P1', 'YES'], ['E', 'NO']],
[0, ['P2', 'YES'], ['P3', 'YES'], ['P1', 'YES'], ['E', 'NO']]]
```

Ανάπτυξη δένδρου - BFS



Ευριστικοί Αλγόριθμοι Αναζήτησης

Η ευρετική αναζήτηση (heuristic search) είναι η μέθοδος αναζήτησης που υποστηρίζεται από κάποιον ευρετικό μηχανισμό. Βασίζεται στη γνώση για το συγκεκριμένο πρόβλημα προς επίλυση και στην κοινή λογική· συνήθως χρησιμοποιείται για εύρεση μίας «καλής» λύσης και όχι απαραίτητα της άριστης. Στην πράξη, σχηματίζει το δένδρο αναζήτησης αναδιατάσσοντας σε κάθε κύκλο λειτουργίας το μέτωπο αναζήτησης σύμφωνα με το κριτήριο που χρησιμοποιεί, χωρίς να ερευνά αν είναι ορθό ή λανθασμένο. Το κριτήριο προκύπτει μέσω μιας ευρετικής συνάρτησης. Μια ευρετική αναζήτηση δεν εγγυάται πάντα την εύρεση λύσης, αν αυτή υπάρχει.

Μελέτη BestFs

Η μέθοδος αναζήτησης Πρώτα στο Καλύτερο ή Καλύτερη-Πρώτη (Best First Search - BestFS) σε κάθε κύκλο λειτουργίας της επισκέπτεται την κατάσταση του μετώπου αναζήτησης την οποία το κριτήριο θεωρεί «καλύτερη» βάσει ενός ευρετικού κριτηρίου άμεσα εξαρτώμενου από το πρόβλημα προς επίλυση. Χαρακτηριστικό της μεθόδου είναι ότι σε κάθε βήμα αναζήτησης η μέθοδος επιλέγει προς επέκταση το πιο πολλά υποσχόμενο μονοπάτι μεταξύ των μονοπατιών που βρίσκονται στην ουρά.

Ένα από τα μειονεκτήματα αυτής της μεθόδου είναι η δυσκολία εύρεσης ευριστικής συνάρτησης. Αν μια ευριστική συνάρτηση είναι αρκετά περίπλοκη και όχι σωστά ορισμένη, η αναζήτηση μπορεί να κοστίσει αρκετά, το οποίο φέρει ως αποτέλεσμα να γίνονται πιο αργά οι συγκρίσεις κόστους.

Ευριστικός μηχανισμός

Ο ευριστικός μηχανισμός που επιλέχθηκε για το πρόβλημα του parking, είναι η εύρεση της απόστασης της πιο κοντινής κενής πλατφόρμας στο κενό χώρο και εκφράζεται από το παρακάτω τύπο:

$$|(j\%2+ j//2) - (j\%2 + j//2)|$$

Με τη χρήση του πηλίκο βρίσκουμε την οριζόντια θέση των αντικειμένων μας ενώ με το ακέραιο υπόλοιπο βρίσκουμε αντίστοιχα τη κάθετη θέση.

Αναζήτηση BestFS με μέτωπο

Η κωδικοποίηση του προβλήματος με χρήση BestFS και παρακολούθηση μετώπου είναι η εξής:

```
def expand_front(front, method):
    if method=='DFS':
        if front:
            print("Front:")
            print(front)
            node=front.pop(0)
            for child in find_children(node):
                front.insert(0,child)

    elif method=='BFS':
        if front:
            print("Front:")
            print(front)
            node=front.pop(0)
            for child in find_children(node):
                front.append(child)
    elif method=='BestFS':
        if front:
            print("Front:")
            print(front)
            node=front.pop(0)
            for child in find_children(node):
                front.insert(0,child)
            front =sort_front(front)
    #else: "other methods to be added"

    return front
```

Παρατηρούμε ότι, η αναζήτηση BestFS με τη DFS είναι ότι ακολουθούν την ίδια λογική, η μόνη διαφορά είναι ότι στη BestFS μετά τη προσθήκη του παιδιού στο μέτωπο ακολουθεί ταξινόμηση του μετώπου.

Ταξινόμηση μετώπου

Παρακάτω βλέπουμε σχολιασμένο το κώδικα ταξινόμησης μετώπου.

```
#Συνάρτηση για ταξινόμηση του μετώπου
#Η ταξινόμηση γίνεται με βάση την απόσταση κενής πλατφόρμας με κενό χώρο
def sort_front(front):
    if front:
        temp_front = front
        distances=[]
        #Για κάθε κατάσταση μέσα στο μέτωπο
        for i in range(len(front)):
            #Αρχικόποιηση τιμών
            platform_pos = -1
            empty_pos = 0
            #Για κάθε λίστα κάθε κατάστασης
            for j in range(1,len(front[i])):
                #Ελέγχουμε αν υπάρχει κενή πλατφόρμα και καταγράφουμε τη θέση
της
                if front[i][j][0] == 'P' and front[i][j][1] == 'NO' and
platform_pos == -1:
                     # Με το πηλίκο βρίσκουμε τη θέση οριζόντια και με το
υπόλοιπο κάθετα
                    platform_pos = j//2 + j%2 #Επέκταση - j//3 + j%2
                   # break
                #Καταγράφουμε τη θέση του κενού χώρου
                if front[i][j][0] == 'E':
                    # Με το πηλίκο βρίσκουμε τη θέση οριζόντια και με το
υπόλοιπο κάθετα
                    empty_pos = j//2 + j%2 #Επέκταση - j//3 + j%2

            if platform_pos != -1:
                #Υπολογίζουμε απόσταση
                distance = abs(platform_pos - empty_pos)
            else:
                #Αν δεν υπάρχει βάζουμε αρνητική τιμή
                distance = -1

            distances.append(distance)

        #Bubble sort για τις καταστάσεις του μετώπου
        for i in range(len(front)):
            for j in range(0, len(front) -i -1):
                if distances[j] > distances[j+1]:
                    temp_front[j], temp_front[j+1] = temp_front[j+1],
temp_front[j]
                    distances[j], distances[j+1] = distances[j+1],
distances[j]

        return temp_front #επιστροφή ταξινομημένου μετώπου
    else:
        return front
```

Παρακολούθηση Μετώπου

1ο Μέτωπο

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

2ο Μέτωπο

```
[[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]
```

3ο Μέτωπο

```
[[2, ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]
```

Τελευταίο Μέτωπο - 1

```
[[1, ['P3', 'NO'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'YES']],
[1, ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES'], ['E', 'NO']],
[1, ['P2', 'YES'], ['P3', 'NO'], ['E', 'NO'], ['P1', 'YES']],
[1, ['P2', 'YES'], ['E', 'NO'], ['P3', 'NO'], ['P1', 'YES']],
[1, ['E', 'NO'], ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES']],
[2, ['P2', 'NO'], ['P3', 'NO'], ['E', 'NO'], ['P1', 'YES']],
[2, ['P1', 'YES'], ['P2', 'NO'], ['P3', 'NO'], ['E', 'NO']],
[2, ['P1', 'YES'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'NO']],
[2, ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[2, ['E', 'NO'], ['P1', 'YES'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]
```

Τελευταίο Μέτωπο

```
[[0, ['P3', 'YES'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'YES']],
[1, ['E', 'NO'], ['P3', 'NO'], ['P1', 'YES'], ['P2', 'YES']],
[1, ['P3', 'NO'], ['P1', 'YES'], ['E', 'NO'], ['P2', 'YES']],
[1, ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES'], ['E', 'NO']],
[1, ['P2', 'YES'], ['P3', 'NO'], ['E', 'NO'], ['P1', 'YES']],
[1, ['P2', 'YES'], ['E', 'NO'], ['P3', 'NO'], ['P1', 'YES']],
[1, ['E', 'NO'], ['P2', 'YES'], ['P3', 'NO'], ['P1', 'YES']],
[2, ['P2', 'NO'], ['P3', 'NO'], ['E', 'NO'], ['P1', 'YES']],
[2, ['P1', 'YES'], ['P2', 'NO'], ['P3', 'NO'], ['E', 'NO']],
[2, ['P1', 'YES'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'NO']],
[2, ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[2, ['E', 'NO'], ['P1', 'YES'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]]
```

Παρακολούθηση Μετώπου στην Επέκταση

1ο Μέτωπο

```
[[5, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['P5', 'NO']]]
```

2ο Μέτωπο

```
[[5, ['P5', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['E', 'NO']],
[5, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['P5', 'NO']]]
```

3ο Μέτωπο

```
[[4, ['P5', 'YES'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['E', 'NO']],
[5, ['P5', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO'], ['E', 'NO'], ['P4', 'NO']],
[5, ['P5', 'NO'], ['P1', 'NO'], ['E', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['P2', 'NO']],
[5, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['P5', 'NO']],
[5, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['P5', 'NO']]]
```

•
•
•

Εύρεση τελικής κατάστασης

```
_GOAL_FOUND_
```

```
[0, ['P4', 'YES'], ['E', 'NO'], ['P1', 'YES'], ['P2', 'YES'], ['P5', 'YES'],
['P3', 'YES']]
```

BestFS με δυνατότητα παράλληλης παρακολούθησης ουράς μονοπατιών

Παρακάτω βλέπουμε τον κώδικα που αναπτύχθηκε με σκοπό ο αλγόριθμος να έχει τη δυνατότητα παράλληλης παρακολούθησης ουράς μονοπατιών. Παρατηρούμε ότι προστέθηκε στην extend_queue() κομμάτι κώδικα με σκοπό να μπορεί να γίνει παρακολούθηση ουράς και για τον αλγόριθμο BestFS. Αντίστοιχα, όπως είδαμε και με τον κώδικα παρακολούθησης μετώπου, η μόνη διαφορά της αναζήτησης BestFS με τη DFS είναι ότι αφού προστεθούν οι διάδοχοι κόμβοι καλείται συνάρτηση για ταξινόμηση της ουράς.

```
def extend_queue(queue, method):
    if method=='DFS':
        print("Queue:")
        print(queue)
        node=queue.pop(0)
        queue_copy=copy.deepcopy(queue)
        children=find_children(node[-1])
        for child in children:
            path=copy.deepcopy(node)
            path.append(child)
            queue_copy.insert(0,path)

    elif method=='BFS':
        print("Queue:")
        print(queue)
        node=queue.pop(0)
        queue_copy=copy.deepcopy(queue)
        children=find_children(node[-1])
        for child in children:
            path=copy.deepcopy(node)
            path.append(child)
            queue_copy.append(path)

    elif method=='BestFS':
        #print("Queue:")
        #print(queue)
        node=queue.pop(0)
        queue_copy=copy.deepcopy(queue)
        children=find_children(node[-1])
        for child in children:
            path=copy.deepcopy(node)
            path.append(child)
            queue_copy.insert(0,path)
        queue_copy = sort_queue(queue_copy)
    #else: "other methods to be added"

    return queue_copy
```

Ταξινόμηση ουράς

Παρακάτω βλέπουμε σχολιασμένο το κώδικα ταξινόμησης ουράς.

```
#Συνάρτηση για ταξινόμηση της ουράς
#Η ταξινόμηση γίνεται με βάση την απόσταση κενής πλατφόρμας με κενό χώρο όπως
και στη sort_front
def sort_queue(queue):
    if queue:
        distances=[]

        for i in range(len(queue)):

            temp_queue = queue
            #Παίρνουμε τη τελευταία κατάσταση για κάθε στοιχείο
            base = queue[i][-1]
            #Αρχικόποιηση τιμών
            platform_pos = -1
            empty_pos = 0
            #Για κάθε λίστα κάθε κατάστασης
            for j in range(1,len(base)):
                #Ελέγχουμε αν υπάρχει κενή πλατφόρμα και καταγράφουμε τη θέση
της
                if base[j][0] == 'P' and base[j][1] == 'NO' and platform_pos
== -1:
                    # Με το πηλίκο βρίσκουμε τη θέση οριζόντια και με το
υπόλοιπο κάθετα
                    platform_pos = j//2 + j%2 #Επέκταση - j//3 + j%2
                #Καταγράφουμε τη θέση του κενού χώρου
                if base[j][0] == 'E':
                    empty_pos = j//2 + j%2 #Επέκταση - j//3 + j%2

            if platform_pos != -1:
                #Υπολογίζουμε απόσταση
                distance = abs(platform_pos - empty_pos)
            else:
                distance = -1

            distances.append(distance)

        #Bubble sort για τις καταστάσεις του μετώπου
        for i in range(len(queue)):
            for j in range(0, len(queue) -i -1):
                if distances[j] > distances[j+1]:
                    temp_queue[j], temp_queue[j+1] = temp_queue[j+1],
temp_queue[j]
                    distances[j], distances[j+1] = distances[j+1],
distances[j]

        return temp_queue #επιστροφή ταξινομημένης ουράς
    else:
        return queue
```

Παρακολούθηση Ουράς

1ο Μονοπάτι ουράς

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]
```

2ο Μονοπάτι ουράς

```
[[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P3', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']]],

[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']],
[3, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]
```

3ο Μονοπάτι ουράς

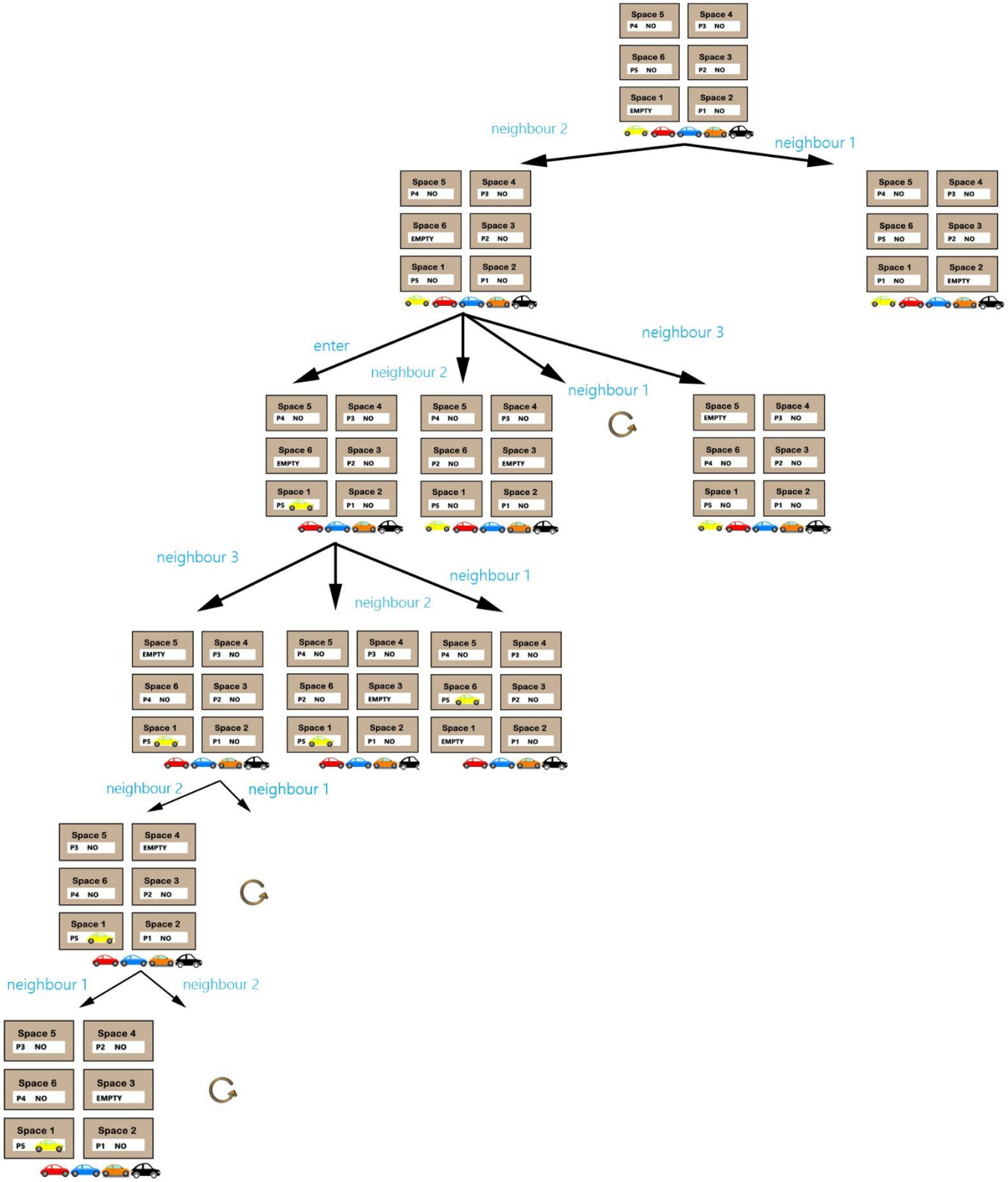
```
 [[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3',
'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']], [2, ['P3', 'YES'], ['P1',
'NO'], ['P2', 'NO'], ['E', 'NO']]],
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3',
'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']], [3, ['P3', 'NO'], ['P1',
'NO'], ['E', 'NO'], ['P2', 'NO']]],
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3',
'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']], [3, ['E', 'NO'], ['P1',
'NO'], ['P2', 'NO'], ['P3', 'NO']]],
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P1',
'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3', 'NO']]]]
```

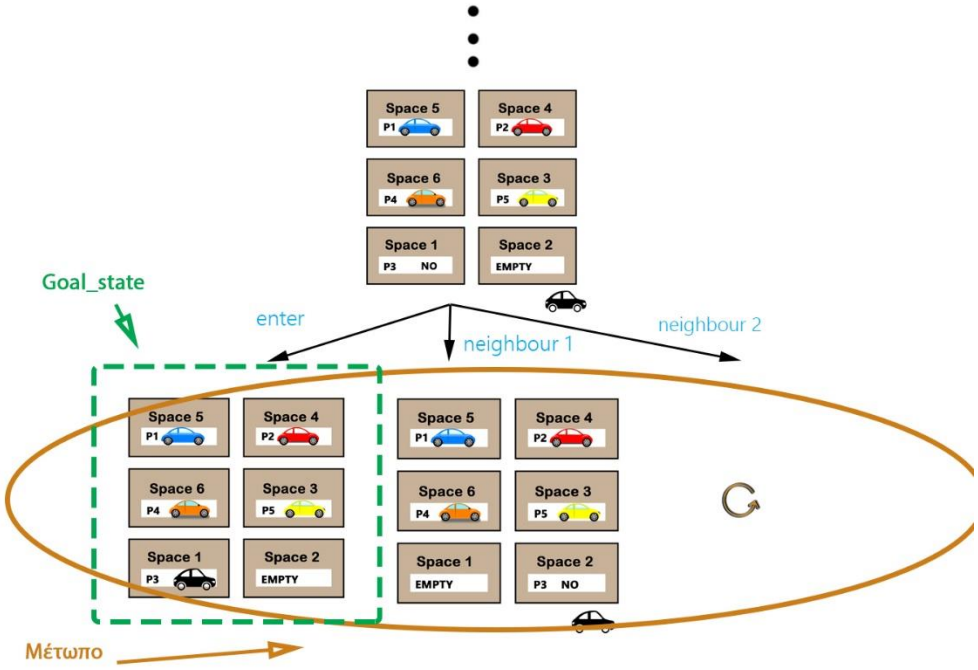
•
•
•

Ουρά Goal State

```
[[3, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO']], [3, ['P3',
'NO'], ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO']], [2, ['P3', 'YES'], ['P1',
'NO'], ['P2', 'NO'], ['E', 'NO']], [2, ['E', 'NO'], ['P1', 'NO'], ['P2',
'NO'], ['P3', 'YES']], [2, ['P1', 'NO'], ['E', 'NO'], ['P2', 'NO'], ['P3',
'YES']], [2, ['P1', 'NO'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'YES']], [1,
['P1', 'YES'], ['P2', 'NO'], ['E', 'NO'], ['P3', 'YES']], [1, ['P1', 'YES'],
['P2', 'NO'], ['P3', 'YES'], ['E', 'NO']], [1, ['E', 'NO'], ['P2', 'NO'],
['P3', 'YES'], ['P1', 'YES']], [1, ['P2', 'NO'], ['E', 'NO'], ['P3', 'YES'],
['P1', 'YES']], [0, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P1',
'YES']]]
```

Ανάπτυξη δένδρου – BestFS





Σύγκριση Μεθόδων Αναζήτησης Προβλήματος

Εξαντλητικοί Έλεγχοι (DFS, BFS, BestFS)

```
Αρχική Κατάσταση:  [2, ['E', 'NO'], ['P1', 'NO'], ['P2', 'NO'], ['P3', 'NO'],
['P4','NO'],['P5','NO']]
```

Τελική Κατάσταση	
DFS	[0, ['P2', 'YES'], ['P5', 'YES'], ['P1', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['E', 'NO']]
BFS	[0, ['P5', 'YES'], ['P1', 'YES'], ['P2', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['E', 'NO']]
BestFs	[0, ['P2', 'YES'], ['P5', 'YES'], ['P1', 'NO'], ['P3', 'NO'], ['P4', 'NO'], ['E', 'NO']]

```
Αρχική Κατάσταση:  [2, ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO'], ['P3',
'NO'], ['P4','NO'],['P5','NO']]
```

Τελική Κατάσταση	
DFS	[0, ['P3', 'YES'], ['E', 'NO'], ['P4', 'NO'], ['P5', 'NO'], ['P1', 'YES'], ['P2', 'YES']]
BFS	[0, ['P2', 'YES'], ['P5', 'YES'], ['P1', 'YES'], ['P3', 'NO'], ['P4', 'NO'], ['E', 'NO']]
BestFs	[0, ['P4', 'YES'], ['P5', 'YES'], ['P3', 'NO'], ['P1', 'YES'], ['P2', 'NO'], ['E', 'NO']]

```
Αρχική Κατάσταση: [1, ['P1', 'YES'], ['E', 'NO'], ['P2', 'NO'], ['P3',
'YES'], ['P4','YES'],['P5','NO']]
```

Τελική Κατάσταση	
DFS	**[0, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P4', 'YES'], ['P5', 'NO'], ['P1', 'YES']]**
BFS	**[0, ['P5', 'YES'], ['P1', 'YES'], ['P2', 'NO'], ['P3', 'YES'], ['P4', 'YES'], ['E', 'NO']]**
BestFs	**[0, ['P2', 'YES'], ['E', 'NO'], ['P3', 'YES'], ['P4', 'YES'], ['P5', 'NO'], ['P1', 'YES']]**

```
Αρχική Κατάσταση: [5, ['P1', 'NO'], ['E', 'NO'], ['P2', 'YES'], ['P3',
'YES'], ['P4','YES'],['P5','YES']]
```

Τελική Κατάσταση	
DFS	_NO_SOLUTION_FOUND_
BFS	_NO_SOLUTION_FOUND_
BestFs	_NO_SOLUTION_FOUND_

Συμπεράσματα

Η υλοποίηση αυτής της εργασίας μας έφερε σε πρώτη επαφή με το αντικείμενο «Τεχνητή Νοημοσύνη». Μελετήσαμε τους τρόπους επίλυσης ενός προβλήματος και πώς αυτό μπορούμε να τον αναπαραστήσουμε σε κώδικα. Υπάρχουν διάφορα είδη αλγορίθμων αναζήτησης. Από αυτούς μελετήσαμε συγκεκριμένα τη DFS, BFS και BestFS.

Αν και όλοι οι αλγόριθμοι δίνουν κάποια λύση, ανάλογα με τις απαιτήσεις μας διαλέγουμε τον πιο κατάλληλο. Για παράδειγμα, παρατηρήσαμε ότι ο DFS είναι πολύ πιο γρήγορος αλγόριθμος, ωστόσο δε δίνει τη βέλτιστη λύση. Η βέλτιστη λύση την εγγυάται ο BFS, ο οποίος φροντίζει να βρει τη μικρότερη διαδρομή προς την κατάσταση στόχου, αλλά η αναζήτησή της είναι αρκετά χρονοβόρα διότι εξετάζει όλο το δέντρο.

Τέλος, αναπτύξαμε και την BestFS, ορίζοντας κάποιο ευριστικό κριτήριο. Παρατηρούμε ότι η εκτέλεση του είναι σημαντικά πιο γρήγορη από αυτή των άλλων δυο αλγορίθμων αναζήτησης που δοκιμάσαμε. Σημαντικό να αναφερθεί είναι ότι χρειάζεται ιδιαίτερη προσοχή στην επιλογή ευριστικού κριτηρίου καθώς μπορεί ο αλγόριθμος να γίνει ιδιαίτερα χρονοβόρος.

Βιβλιογραφία

Γεωργούλη, Α. (2015). *Τεχνητή Νοημοσύνη - Μια εισαγωγική προσέγγιση.*

https://repository.kallipos.gr/pdfviewer/web/viewer.html?file=/bitstream/11419/3381/1/Τεχνητή%20Νοημοσύνη.pdf