Πανεπιστήμιο Δυτικής Αττικής
Σχολή Μηχανικών
Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών

Εργαστήριο
Τεχνητής Νοημοσύνης

ΝΙΚΟΛΑΟΣ ΘΩΜΑΣ - 21390068
ΦΙΛΙΠΠΟΣ ΠΑΠΑΓΕΩΡΓΙΟΥ - 21390174

ΑΘΗΝΑ
Παρασκευή , 12 Ιανουαρίου 2024

1. Κοσμος του προβληματος

Αντικείμενα	Ιδιότητες	Σχέσεις
1ος Όροφος	Παρουσία Ενοίκων	Ασανσέρ -> Μέχρι 8 άτομα Ασανσέρ -> Αφήνει άτομα στην ταράτσα Ασανσέρ -> Άδειο Ένοικοι -> Στους ορόφους
2ος Όροφος		
3ος Όροφος		
4ος Όροφος		
Ένοικοι		
Ασανσέρ	Χωρητικότητα ανά όροφο	
Ισόγειο	Παρουσία Ασανσέρ	
Ταράτσα		

Αρχική κατάσταση

0	9	12	4	7	0

Τελική κατάσταση

5	0	0	0	0	0

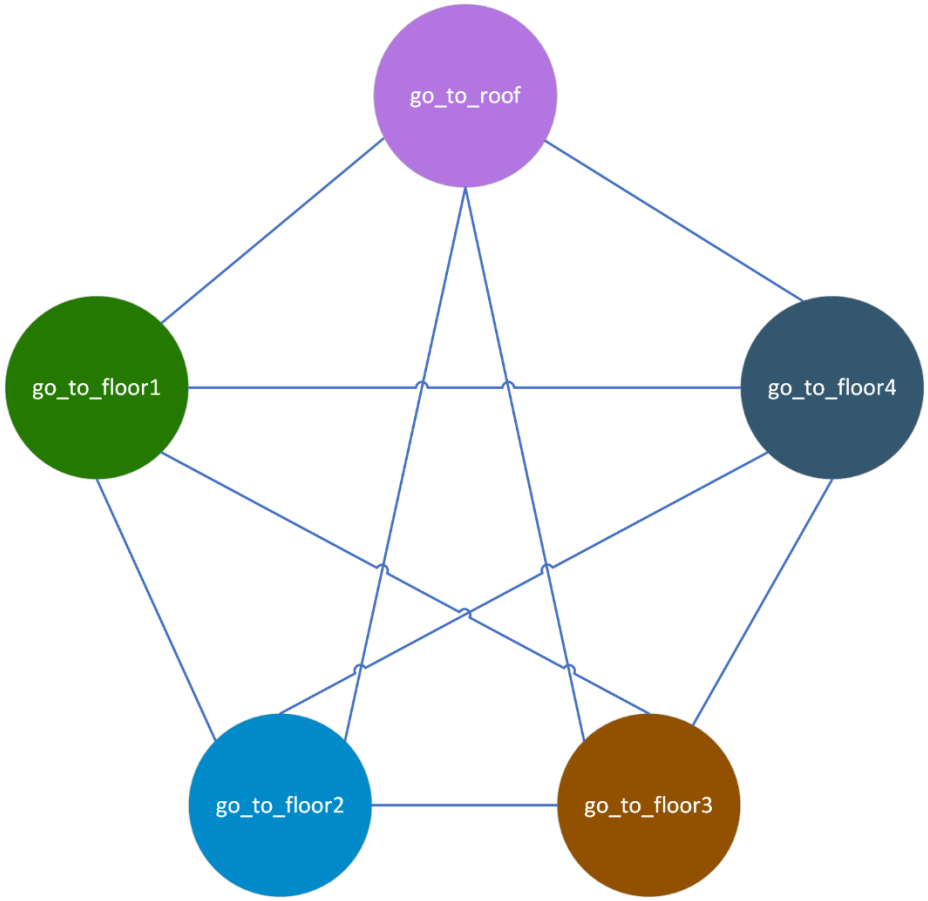
2. ΤΕΛΕΣΤΕΣ ΜΕΤΑΒΑΣΗΣ

Όνομα	go_to_floor_1	go_to_floor_2
Προϋποθέσεις	• Άτομα στο ασανσέρ < 8 • Κάτοικοι 1ου ορόφου >= 1 • Όροφος ασανσέρ <> 1	• Άτομα στο ασανσέρ < 8 • Κάτοικοι 2ου ορόφου >= 1 • Όροφος ασανσέρ <> 2
Αποτέλεσμα	• Όροφος ασανσέρ -> 1 • Άτομα ασανσέρ -> 8 • Μειώνοντας τα άτομα στο ασανσέρ	• Όροφος ασανσέρ -> 2 • Άτομα ασανσέρ -> 8 • Μειώνοντας τα άτομα στο ασανσέρ

Όνομα	go_to_floor_3	go_to_floor_4
Προϋποθέσεις	• Άτομα στο ασανσέρ < 8 • Κάτοικοι 3ου ορόφου >= 1 • Όροφος ασανσέρ <> 3	• Άτομα στο ασανσέρ < 8 • Κάτοικοι 4ου ορόφου >= 1 • Όροφος ασανσέρ <> 4
Αποτέλεσμα	• Όροφος ασανσέρ -> 3 • Άτομα ασανσέρ -> 8 • Μειώνοντας τα άτομα στο ασανσέρ	• Όροφος ασανσέρ -> 4 • Άτομα ασανσέρ -> 8 • Μειώνοντας τα άτομα στο ασανσέρ

Όνομα	go_to_roof
Προϋποθέσεις	• Άτομα στο ασανσέρ = 8 ή Κάτοικοι 1ου , 2ου , 3ου , 4ου ορόφου = 0 • Όροφος ασανσέρ <> 5
Αποτέλεσμα	• Όροφος ασανσέρ -> 5 • Άτομα ασανσέρ -> 0 • Μειώνοντας τα άτομα στους ορόφους

3. ΧΩΡΟΣ ΚΑΤΑΣΤΑΣΕΩΝ



Εικόνα 1: Χώρος Καταστάσεων

P = (I, G, T, S)

I = Αρχική Κατάσταση

G = Τελική Κατάσταση

T = Σύνολο τελεστών Μετάβασης

S = Χώρος Καταστάσεων

P = ([0, 9, 12, 4, 7, 0], [5, 0, 0, 0, 0], [5], [*])

Χώρος καταστάσεων του προβλήματος, είναι το σύνολο των έγκυρων καταστάσεών που μπορεί να βρεθεί ένα πρόβλημα κατά την εξέλιξη του κόσμου του. Στο πρόβλημα της εκκένωσης κτηρίου προς την ταράτσα, έχουμε κλειστό χώρο αναζήτησης, δηλαδή ο αλγόριθμος αναζήτησης εξετάζει ένα πεπερασμένο αριθμό πιθανόν καταστάσεών η καταστάσεων του προβλήματος.

Ο χώρος καταστάσεων, που αναφέρεται στο σύνολο όλων των πιθανών οροφών, που μπορεί να ακολουθήσει το ασανσέρ αντιπροσωπεύει μια δυνατή κατάσταση του προβλήματος.

Στόχος: Είναι η δημιουργία ενός κλειστού μονοπατιού που θα επισκεφτεί όλες τις πιθανές καταστάσεις, καθώς το ασανσέρ μετακινείται στην πολυκατοικία περνώντας τους ένοικούς, με απώτερο σκοπό την μεταφορά τους στην ταράτσα έως ότου να είναι όλοι στην ταράτσα.

4. ΚΩΔΙΚΟΠΟΙΗΣΗ ΤΟΥ ΚΟΣΜΟΥ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ

Ο κόσμος του προβλήματος αναφέρεται σε ένα κτήριο με ορόφους, κατοίκους, τον ανελκυστήρα και την ταράτσα. Όλοι έχουν διάφορες ιδιότητες που θα μας βοηθήσουν να κωδικοποιήσουμε το πρόβλημα. Καθώς έχουμε την αρχική κατάστασή, [0, 9, 12, 4, 7, 0] (0 κάτοικοι στο ασανσέρ, 9 στον 1^{ο}, 12 στον 2^{ο}, 4 στον 3^{ο}, 7 στον 4^{ο} όροφο, 0 στην ταράτσα) και το ασανσέρ στο ισόγειο με 0 άτομα. Την τελική κατάσταση, [5, 0, 0, 0, 0, 0] (Το ασανσέρ είναι στον 5^{ο}, 0 κάτοικοι σε άλλους ορόφους, δηλαδή όλοι οι κάτοικοί έχουν μεταφερθεί στην ταράτσα).

Στην συνέχεια, εισήγαμε τους τελεστές μετάβασης με τους οποίους θα υλοποιήσουμε τις σχέσεις μεταξύ των αντικειμένων μέσα στον κόσμο του προβλήματος όπως φαίνεται στην 2^{η} εικόνα.

```
def go_to_floor1(state):
    if state[-1]<8 and state[1]>0:
        if state[1]>8-state[-1]:
            new_state = [1] + [state[1] + state[-1] - 8] + [state[2]] + [state[3]] + [state[4]] + [8]
        else:
            new_state = [1] + [0] + [state[2]] + [state[3]] + [state[4]] + [state[1] + state[-1]]
        return new_state

def go_to_floor2(state):
    if state[5]<8 and state[2]>0:
        if state[2]>8-state[5]:
            new_state = [2] + [state[1]] + [state[2] + state[5] - 8] + [state[3]] + [state[4]] + [8]
        else:
            new_state = [2] + [state[1]] + [0] + [state[3]] + [state[4]] + [state[2] + state[5]]
        return new_state

def go_to_floor3(state):
    if state[5]<8 and state[3]>0:
        if state[3]>8-state[5]:
            new_state = [3] + [state[1]] + [state[2]] + [state[3]+state[5]-8] + [state[4]] + [8]
        else:
            new_state = [3] +  [state[1]] + [state[2]] + [0] + [state[4]] + [state[3] + state[5]]
        return new_state

def go_to_floor4(state):
    if state[5]<8 and state[4]>0:
        if state[4]>8-state[5]:
            new_state = [4] + [state[1]] + [state[2]] + [state[3]] + [state[4]+state[5] - 8] + [8]
        else:
            new_state = [4] +  [state[1]] + [state[2]] + [state[3]] + [0] + [state[4] + state[5]]
        return new_state


def go_to_roof(state):
    if state[5] == 0:
       return None
    else:
       new_state = [5] + [state[1]] + [state[2]] + [state[3]] + [state[4]] + [0]
       return new_state
```

Εικόνα 2: Τελεστές Μετάβασης Κώδικας

5. ΚΩΔΙΚΟΠΟΙΗΣΗ ΤΩΝ ΤΕΛΕΣΤΩΝ ΜΕΤΑΒΑΣΗΣ ΚΑΙ Η ΣΥΝΑΡΤΗΣΗ ΕΥΡΕΣΗΣ ΑΠΟΓΟΝΩΝ

Ο κώδικας αποτελείται από συναρτήσεις που υλοποιούν τους τελεστές μετάβασης. Κάθε συνάρτηση ελέγχει τις καθορισμένες συνθήκες πριν από την ενημέρωση της κατάστασης με βάση τις μεταβάσεις. Ο κώδικας βοηθά στη μοντελοποίηση της κίνησης του ανελκυστήρα και των αλλαγών στην κατάσταση των κατοίκων στους διάφορους ορόφους. Συγκεκριμένα οι συναρτήσεις go_to_floor[*]:

Προϋποθέσεις:

- state[-1] < 8: Ελέγχει αν υπάρχει χώρος στο ασανσέρ (το τελευταίο στοιχείο της κατάστασης αντιπροσωπεύει τον αριθμό των ατόμων στο ασανσέρ).
- state[1] > 0: Ελέγχει εάν υπάρχουν κάτοικοι στον 1^{ο} όροφο.
- state[1] > 8 - state[-1]: Αυτή η κατάσταση ελέγχει αν υπάρχουν περισσότεροι κάτοικοι στον 1^{ο} όροφο από την εναπομένουσα χωρητικότητα του ανελκυστήρα. Με άλλα λόγια, ελέγχει αν υπάρχουν περισσότεροι κάτοικοι στον 1^{ο} όροφο από τον διαθέσιμο χώρο στον ανελκυστήρα.

Εάν η συνθήκη είναι αληθής:

- new_state = [1] + [state[1] + state[-1] - 8] + [state[2]] + [state[3]] + [state[4]] + [8]: Το ασανσέρ μετακινείται στον 1ο όροφο ([1]). Ο αριθμός των ατόμων στο ασανσέρ γίνεται η εναπομένουσα χωρητικότητα μετά τη λήψη των κατοίκων από τον 1ο όροφο (κατάσταση[1] + κατάσταση[-1] - 8). Η υπόλοιπη κατάσταση παραμένει αμετάβλητη, συμπεριλαμβανομένου του αριθμού των κατοίκων στους άλλους ορόφους. Ο ανελκυστήρας τίθεται στην πλήρη χωρητικότητά του, των 8 ατόμων ([8]).

Εάν η κατάσταση είναι ψευδής:

- new_state = [1] + [0] + [state[2]] + [state[3]] + [state[4]] + [state[1] + state[-1]]: Το ασανσέρ μετακινείται στον 1ο όροφο ([1]). Ο αριθμός των ατόμων στο ασανσέρ γίνεται 0 ([0]) επειδή υπάρχουν αρκετές διαθέσιμες θέσεις για όλους τους κατοίκους στον 1^{ο} όροφο. Η υπόλοιπη κατάσταση παραμένει αμετάβλητη, συμπεριλαμβανομένου του αριθμού των κατοίκων στους άλλους ορόφους.

Go_to_roof:

Προϋποθέσεις:

- state[5] == 0: Ελέγχει αν δεν υπάρχουν άτομα στο ασανσέρ.

Αποτέλεσμα:

Εάν πληρούνται οι συνθήκες, η συνάρτηση δημιουργεί μια νέα κατάσταση (new_state) όπου το ασανσέρ μετακινείται στην ταράτσα (5ος όροφος) και ο αριθμός των ατόμων στο ασανσέρ ορίζεται σε 0. Επιστρέφεται η νέα κατάσταση.

Στην συνέχεια η κωδικοποίηση της συνάρτησης find_children.

```
def find_children(state):

    children=[]

    floor1_state=copy.deepcopy(state)
    floor1_child=go_to_floor1(floor1_state)
    floor2_child=go_to_floor2(floor1_state)
    floor3_child=go_to_floor3(floor1_state)
    floor4_child=go_to_floor4(floor1_state)
    roof_child=go_to_roof(floor1_state)


    if roof_child!=None:
        children.append(roof_child)
    if floor4_child!=None:
        children.append(floor4_child)
    if floor3_child!=None:
        children.append(floor3_child)
    if floor2_child!=None:
        children.append(floor2_child)
    if floor1_child!=None:
        children.append(floor1_child)

    return children
```

Εικόνα 3: Κωδικοποίηση της συνάρτησης findchildren

Η συνάρτηση find_children παράγει πιθανές διαδοχικές καταστάσεις (παιδιά) από τη δεδομένη κατάσταση εφαρμόζοντας πιθανές ενέργειες (που ορίζονται στις συναρτήσεις go_to_floor και go_to_roof). Δημιουργεί αντίγραφα της τρέχουσας κατάστασης για κάθε πιθανή ενέργεια και ελέγχει αν οι καταστάσεις-παιδιά που προκύπτουν είναι έγκυρες. Οι έγκυρες παιδικές καταστάσεις συλλέγονται στη συνέχεια σε μια λίστα και επιστρέφονται.

Συγκεκριμένα φτιάχνουμε την λίστα children = []: για την αποθήκευση πιθανών παιδιών. Στην συνέχεια η floor1_state = copy.deepcopy(state): Δημιουργεί ένα αντίγραφο της τρέχουσας κατάστασης για κάθε όροφο. Δημιουργεί πιθανα παιδιά καλώντας τις αντίστοιχες συναρτήσεις go_to_floor και go_to_roof, με τις αντιγραμμένες καταστάσεις. Ελέγχει αν κάθε κατάσταση-παιδί δεν είναι None και την προστίθει στη λίστα παιδιών. Τέλος επιστρέφεται η λίστα με τις πιθανές καταστάσεις-παιδιά.

6. ΜΕΘΟΔΟ ΑΝΑΖΗΤΗΣΗΣ

Οι μέθοδοι αναζήτησης που χρησιμοποιήσαμε στο πρόβλημα της πολυκατοικίας με το ασανσέρ, είναι ο αλγόριθμος της αναζήτησης κατά βάθος (DFS), ο αλγόριθμος αναζήτησης κατά πλάτος (BFS) και ο αλγόριθμος A*.

Τυφλοί αλγόριθμοι αναζήτησης:

Οι αλγόριθμοι τυφλής αναζήτησης, όπως οι DFS και BFS, είναι γενικές μέθοδοι που εξερευνούν έναν χώρο αναζήτησης χωρίς να λαμβάνουν υπόψη τα ειδικά χαρακτηριστικά του προβλήματος. Λειτουργούν με βάση τη συστηματική εξερεύνηση, είτε εμβαθύνοντας στο χώρο αναζήτησης (DFS) είτε εξερευνώντας όλους τους γείτονες στο τρέχον επίπεδο βάθους (BFS)".

Ευριστικοί αλγόριθμοι αναζήτησης:

Αντίθετα, οι ευριστικοί αλγόριθμοι, όπως ο Alpha-Star A*, ενσωματώνουν ειδικές γνώσεις για τον τομέα για να καθοδηγήσουν την αναζήτηση πιο έξυπνα. Ο A* συνδυάζει τα πλεονεκτήματα τόσο των άπληστων αλγορίθμων όσο και της συστηματικής αναζήτησης, χρησιμοποιώντας μια ευριστική συνάρτηση για την εκτίμηση του κόστους επίτευξης του στόχου από μια συγκεκριμένη κατάσταση. Αυτή η τεκμηριωμένη προσέγγιση επιτρέπει στον A* να δίνει προτεραιότητα στα μονοπάτια που πιθανόν να οδηγούν στη βέλτιστη λύση, με αποτέλεσμα πιο αποδοτικές και αποτελεσματικές διαδικασίες αναζήτησης. Η συμπερίληψη ευριστικών στοιχείων επιτρέπει στο A* να επιτύχει μια ισορροπία μεταξύ εξερεύνησης και εκμετάλλευσης, καθιστώντας το μια ισχυρή επιλογή για προβλήματα όπου η τεκμηριωμένη λήψη αποφάσεων είναι ζωτικής σημασίας.

➢ DFS ΑΛΓΟΡΙΘΜΟΣ

```
def extend_queue(queue, method):
    if method=='DFS':
        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)
```

```
def find_solution(front, queue, closed, goal, method,
    if not front:
        print('_NO_SOLUTION_FOUND_')
        return None

    elif front[0] in closed:
        front=deque(front)
        new_front=copy.deepcopy(front)
        new_front.popleft()
        new_queue=copy.deepcopy(queue)
        new_queue.popleft()
        find_solution(new_front, new_queue, closed, goal, method,steps+1)
    elif front[0]==goal:
        print('_GOAL_FOUND_')
        print(queue[0])
        print(len(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, goal, method,steps+1)
```

Εικόνα 4: Χρήση του DFS

• Για αρχή, η συνάρτηση extend_queue όπου κάνει extend την ουρά βασεί την συκγεκριμένη μέθοδο αναζήτησης. Στην περίπτωση του DFS κάνει pop το front της ουράς δηλαδή “queue.pop(0)” για να εξερευνήσει τους βαθύτερους κόμβους με την find_children (το ποιό αρίστερο παιδί). Για κάθε κόμβο-παιδί δημιουργεί καινούργιο μονοπάτι και το προσθέτει στην αρχή της ουράς .

• Στην συνέχεια έχουμε την find_solution. Η find_solution λαμβάνει παραμέτρους όπως το front(τρέχουσα κατάσταση), queue(η ουρά εξερεύνησης) , closed(επισκέψεις κατάστασης), goal και method. Εξερευνά αναδρομικά το χώρο αναζήτησης μέχρι να βρεθεί μια λύση ή να εξαντληθούν όλες οι δυνατότητες.

• Επίσης η Ούρα(queue) αναπαριστά το σύνορο εξερεύνησης, αποθηκεύοντας προς εξερεύνηση μονοπατιών με συγκεκριμένη μέθοδο αναζήτησης. Στην extend_queue τροποποιείται με την προσθήκη νεών μονοπατιών στο μπροστινό η το πίσω μέρος ανάλογα με την μέθοδο αναζήτησης (DFS ή BFS).

• Ενώ η front αντιπροσωπεύει την τρέχουσα κατάσταση που διερευνάται στη διαδικασία αναζήτησης. Είναι ένα υποσύνολο της ουράς, που περιέχει μονοπάτια προς περαιτέρω επέκταση. Η συνάρτηση find_solution χειρίζεται και εξερευνά διαφορετικές καταστάσεις ενημερώνοντας τη front κατά τη διάρκεια της διαδικασίας της αναζήτησης.

➢ BFS ΑΛΓΟΡΙΘΜΟΣ

```
elif method == 'BFS':
    print("Queue:")
    print(queue)
    queue=deque(queue)
    node =queue.popleft()
    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)
```

Εικόνα 5: Χρήση του BFS

• Η ουρά εκτυπώνεται για να δείξει τα περιεχόμενά της πριν από την τροποποίηση. Στη συνέχεια μετατρέπεται σε deque (deque(queue)) για την αποτελεσματική χρήση της popleft() για BFS.

• Διαδικασία BFS: Ο αριστερότερος κόμβος αφαιρείται (node = queue.popleft()) για να εξερευνηθούν πρώτα οι πρώτοι κόμβοι. Τα παιδιά του τρέχοντος κόμβου βρίσκονται με τη χρήση της find_children. Για κάθε παιδί, δημιουργείται ένα νέο μονοπάτι και προστίθεται στο τέλος της ουράς (queue_copy.append(path)). Επιστρέφεται η τροποποιημένη queue_copy.

• Στην BFS, η ουρά λειτουργεί ως δομή FIFO (first-in-first-out) και ο αλγόριθμος διερευνά συστηματικά όλους τους κόμβους στο τρέχον επίπεδο βάθους πριν προχωρήσει σε βαθύτερα επίπεδα. Αυτό εξασφαλίζει ότι ο αλγόριθμος εξετάζει μονοπάτια με σειρά αυξανόμενου βάθους, καθιστώντας τον κατάλληλο για την εύρεση της συντομότερης διαδρομής προς μια λύση. Η συνάρτηση find_solution, όπως αναφέρθηκε προηγουμένως, μπορεί στη συνέχεια να χρησιμοποιηθεί για να συνεχιστεί η διαδικασία αναζήτησης και να βρεθεί μια λύση χρησιμοποιώντας την ενημερωμένη ουρά BFS.

➢ EXPAND_FRONT • MAKE_FRONT • MAKE_QUEUE

```
def expand_front(front, method):
    if method=='DFS':
        if front:
            node=front.pop(0)
            for child in find_children(node):
                front.insert(0,child)
    elif method == 'BFS':
        if front:
            front=deque(front)
            node = front.popleft()
            for child in find_children(node):
                front.append(child)
```

Η συνάρτηση expand_front στο DFS:

- ελέγχει ποια είναι η μέθοδος αναζήτησης
- αν η front δεν είναι κενή
- αφαιρεί και επιστρέφει το πρώτο στοιχείο της front με την pop(0)
- ξεκινάει ένα loop για την εύρεση των παιδιών στον κόμβο
- εισάγει κάθε παιδί στην αρχή της front

Η συνάρτηση expand_front στο BFS:

- αν η front δεν είναι κενή
- μετατρέπει την front σε deque (επεξεργάζεται και από τις δύο μεριές)
- επιστρέφει το πιο αριστερό στοιχείο της front με την popleft
- ξεκινάει ένα loop για την εύρεση των παιδιών στον κόμβο
- εισάγει κάθε παιδί στο τέλος της front

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

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

Η συνάρτηση make_front:

- δέχεται μία μοναδική παράμετρο state
- επιστρέφει μια λίστα με μοναδικό στοιχείο

Η συνάρτηση make_queue:

- δέχεται μία μοναδική παράμετρο state
- επιστρέφει μία λίστα που περιέχει λίστα ενός στοιχείου

➢ A* ΑΛΓΟΡΙΘΜΟΣ

```python
def heuristic(state, goal):
    return sum(abs(state[i] - goal[i]) for i in range(len(state)))

def a_star_search(initial_state, goal):
    open_list = [(0, 0, initial_state, [])]
    closed_set = set()

    while open_list:
        total_cost, actual_cost, current_state, path = heapq.heappop(open_list)

        if current_state == goal:
            print("Goal Found!")
            print("Goal State:", current_state)
            print("Total Cost:", total_cost)
            print("Path:", [(actual_cost, state) for state in path + [current_state]])
            return

        if tuple(current_state) in closed_set:
            continue

        closed_set.add(tuple(current_state))

        for child in find_children(current_state):
            child_actual_cost = actual_cost + 1
            heuristic_cost = heuristic(child, goal)
            child_total_cost = child_actual_cost + heuristic_cost
            heapq.heappush(open_list, (child_total_cost, child_actual_cost, child, path + [current_state]))

    print("Goal not found")
```

*Εικόνα 6: Η χρήση του A**

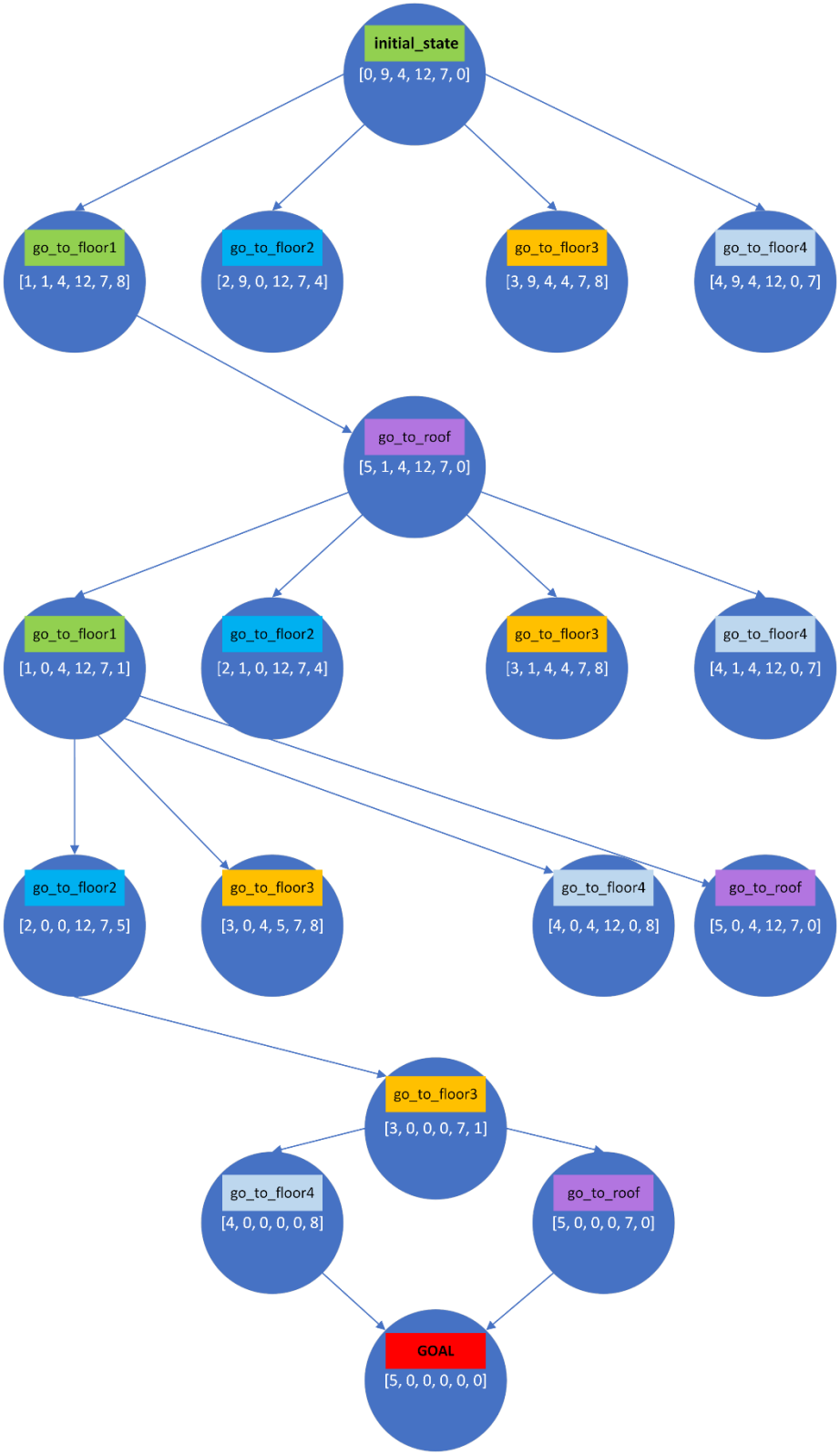
Ευριστική λειτουργία:

- Εκτιμά το κόστος από την τρέχουσα κατάσταση έως τον στόχο. Αυτή η συνάρτηση, που ονομάστηκε ευριστική, μετρά την απόσταση Μανχάταν μεταξύ των αντίστοιχων στοιχείων της κατάστασης και του στόχου.

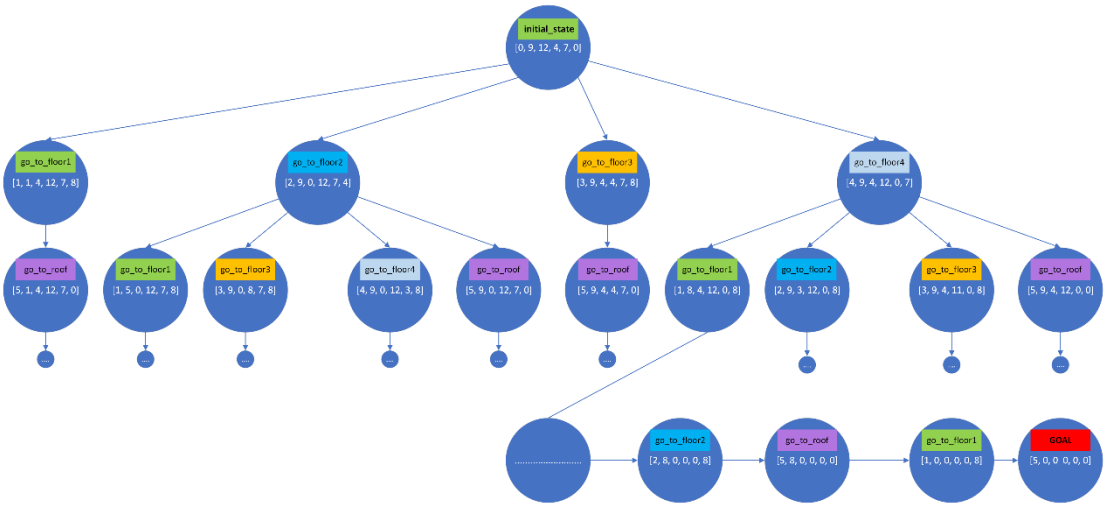
- Η ευριστική συνάρτηση καθοδηγεί τον αλγόριθμο παρέχοντας μια ένδειξη για το πόσο κοντά βρίσκεται μια κατάσταση στον επιθυμητό στόχο. Συμβάλλει στην αποτελεσματικότητα του A* επιτρέποντάς του να εστιάζει σε πολλά υποσχόμενα μονοπάτια.

- Αρχικά ορίσαμε μια ευριστική συνάρτηση που υπολογίζει την απόσταση Μανχάταν μεταξύ της τρέχουσας κατάστασης και του στόχου δηλαδή την “heusteric(state, goal)”. Στην συνέχεια αρχικοποιήσαμε μια ουρά προτεραιότητας “open_list” με την αρχική κατάσταση και ένα κενό μονοπάτι, καθώς και ένα κλειστό σύνολο για τις καταστάσεις που έχει επισκεφτεί “closed_set”. Μετά εισήγαμε έναν κύριο βρόχο αναζήτησης που συνεχίζει μέχρι να αδειάσει η ουρά προτεραιότητας. Στην συνέχεια, αφαιρείται η κατάσταση με το χαμηλότερο συνολικό κόστος από την ουρά προτεραιότητας. Αν είναι ο στόχος ή αν η κατάσταση έχει ήδη εξερευνηθεί την παραλείπουμε. Τέλος, προστίθεται η τρέχουσα κατάσταση στο κλειστό σύνολο. Εξερευνά τις καταστάσεις, υπολογίζοντας το κόστος και προστίθενται στην ουρά προτεραιότητας. Εάν ο βρόχος ολοκληρωθεί χωρίς να βρεθεί ο στόχος, εκτυπώνει το αντίστοιχο μήνυμα.

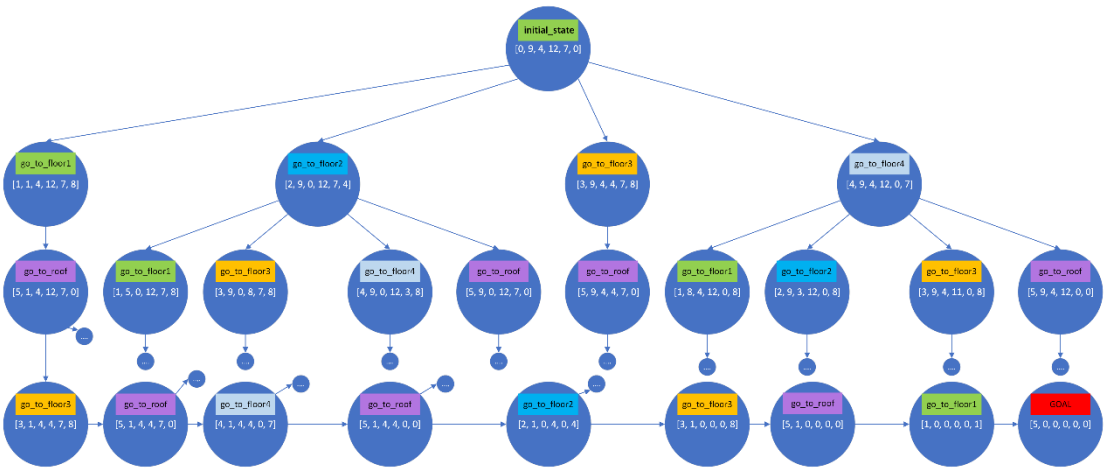
7. ΔΕΝΔΡΑ ΑΝΑΖΗΤΗΣΗΣ



Εικόνα 7: DFS TREE



Εικόνα 8: BFS TREE



Εικόνα 9: A TREE*

8. ΕΞΑΝΤΛΗΤΙΚΟΣ ΕΛΕΓΧΟΣ

➢ DFS ΑΛΓΟΡΙΘΜΟΣ

Εκτελώντας το πρόγραμμα, με μέθοδο τον **DFS**, παρατηρείται ότι:

- εκτελείται πιο γρήγορα
- δεν βρίσκει την βέλτιστη λύση
- είναι αποδοτικός σε χρόνο και μνήμη
- το μέτωπο της αναζήτησης δε μεγαλώνει πάρα πολύ

➢ BFS ΑΛΓΟΡΙΘΜΟΣ

Εκτελώντας το πρόγραμμα, με μέθοδο τον **BFS**, παρατηρείται ότι:

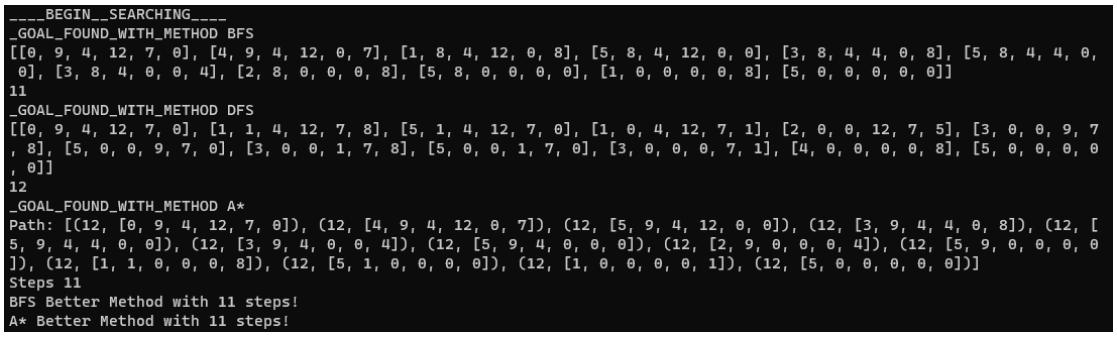
- βρίσκει βέλτιστη λύση
- το μέτωπο αναζήτησης είναι πλήρες
- χρονοβόρο και απαιτητικό σε μνήμη

➢ A* ΑΛΓΟΡΙΘΜΟΣ

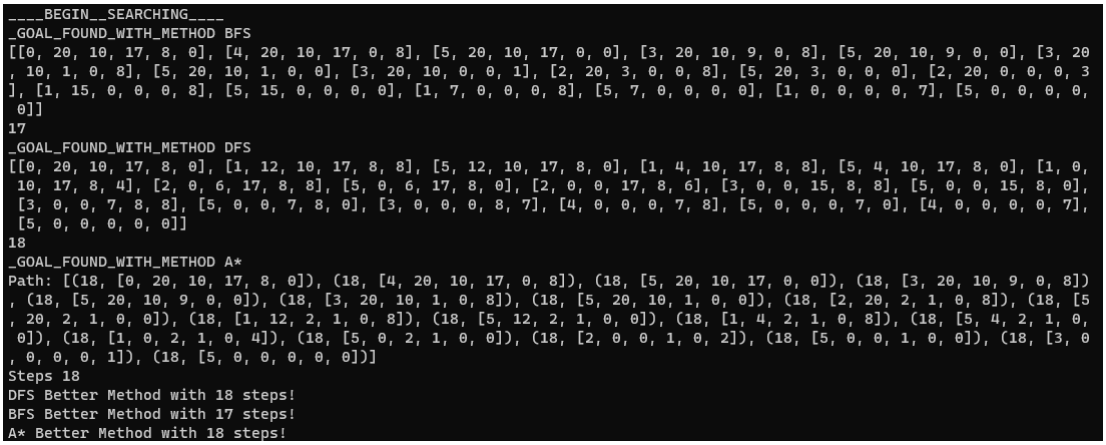
Εκτελώντας το πρόγραμμα, με μέθοδο τον **A***, παρατηρείται ότι:

- βρίσκει βέλτιστη λύση
- το μέτωπο αναζήτησης είναι πλήρες
- χρονοβόρο και απαιτητικό σε μνήμη
- εξαρτάται από τον ευριστικό μηχανισμό

9. ΠΑΡΑΔΕΙΓΜΑΤΑ ΕΚΤΕΛΕΣΗΣ



Εικόνα 10: Εκτέλεση με το initial_state του προβλήματος



Εικόνα 11: Εκτέλεση με εξαντλητικό initial_state