Πανεπιστήμιο Πειραιώς

Σχολή Τεχνολογιών Πληροφορικής και επικοινωνιών

Τμήμα πληροφορικής

Ακαδημαϊκό έτος 2020 – 2021

Τεχνητή νοημοσύνη και έμπειρα συστήματα

1η εργασία

Ομάδα φοιτητών:

Καγκλή Αικατερίνη Π16037

Καπόλος Ιωάννης Π16042

Καθηγητής: Δημήτριος Αποστόλου

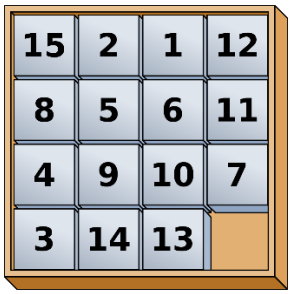
Περιεχόμενα

1. Εισαγωγή

1.1 Στόχος εργασίας

Στόχος της εργασίας είναι η επίλυση του κλασικού γρίφου n-puzzle με την χρήση δύο αλγορίθμων αναζήτησης. Στην παρούσα εργασία επιλέχθηκαν οι αλγόριθμοι αναζήτησης κατά βάθος (depth first search) και κατά πλάτος (breadth first search). Η εκπόνηση του προγράμματος έγινε με τη γλώσσα προγραμματισμού Python.

1.2 Περιγραφή N-puzzle



Παράδειγμα άλυτου 15-puzzle

Το n-puzzle είναι ένα συρόμενο παζλ, με n αριθμημένα τετραγωνικά πλακάκια, όπου (n) ο αριθμός των πλακακιών. Τα πλακάκια είναι τοποθετημένα σε μια κορνίζα, αφήνοντας αρκετό χώρο για να ολισθήσει ένα πλακάκι. Στόχος του παζλ είναι η αναδιάταξη των πλακακιών με τρόπο τέτοιο ώστε οι αριθμοί τους να βρίσκονται σε μια σειρά.

Παράδειγμα λυμένου 15-puzzle

1.3 Σύντομη περιγραφή depth first search και breadth first search αλγορίθμων

1.3.1 Depth first search algorithm

Στην αναζήτηση κατά βάθος ο αλγόριθμος ξεκινάει από μια αρχική κατάσταση του προβλήματος και αναζητά τις πιθανές επόμενες καταστάσεις (καταστάσεις παιδιά). Μόλις βρεθούν οι καταστάσεις παιδιά, εισάγονται σε μια ***στοίβα*** αναζήτησης (μέτωπο αναζήτησης). Στη συνέχεια, αναζητούνται οι καταστάσεις παιδιά του αντικειμένου που βρίσκεται πρώτο στο μέτωπο αναζήτησης. Αφού βρεθούν οι καινούργιες καταστάσεις παιδιά, **εισάγονται πρώτα στο μέτωπο αναζήτησης (last in first out),** αφαιρείται το πρώτο αντικείμενο του μετώπου αναζήτησης, δηλαδή, το αντικείμενο που μόλις εξετάσθηκε και εισάγεται στο κλειστό σύνολο(το σύνολο των καταστάσεων που έχουν ήδη εξετασθεί). Η διαδικασία επαναλαμβάνεται μέχρις ότου να βρεθεί η επιθυμητή κατάσταση ή μέχρι να αδειάσει το μέτωπο αναζήτησης. Αν βρεθεί η επιθυμητή κατάσταση, τα βήματα της λύσης του προβλήματος είναι οι καταστάσεις που βρίσκονται στο κλειστό σύνολο.

1.3.2 Breadth first search algorithm

Στην αναζήτηση κατά πλάτος ο αλγόριθμος ξεκινάει από μια αρχική κατάσταση του προβλήματος και αναζητά τις πιθανές επόμενες καταστάσεις (καταστάσεις παιδιά). Μόλις βρεθούν οι καταστάσεις παιδιά, εισάγονται σε μια ***ουρά*** αναζήτησης (μέτωπο αναζήτησης). Στη συνέχεια, αναζητούνται οι καταστάσεις παιδιά του αντικειμένου που βρίσκεται πρώτο στο μέτωπο αναζήτησης. Αφού βρεθούν οι καινούργιες καταστάσεις παιδιά, **εισάγονται τελευταία στο μέτωπο αναζήτησης (first in first out),** αφαιρείται το πρώτο αντικείμενο του μετώπου αναζήτησης, δηλαδή, το αντικείμενο που μόλις εξετάσθηκε και εισάγεται στο κλειστό σύνολο(το σύνολο των καταστάσεων που έχουν ήδη εξετασθεί). Η διαδικασία επαναλαμβάνεται μέχρις ότου να βρεθεί η επιθυμητή κατάσταση ή μέχρι να αδειάσει το μέτωπο αναζήτησης. Αν βρεθεί η επιθυμητή κατάσταση, τα βήματα της λύσης του προβλήματος είναι οι καταστάσεις που βρίσκονται στο κλειστό σύνολο.

1.4 Αναδρομή vs επανάληψη (Recursion vs iteration)

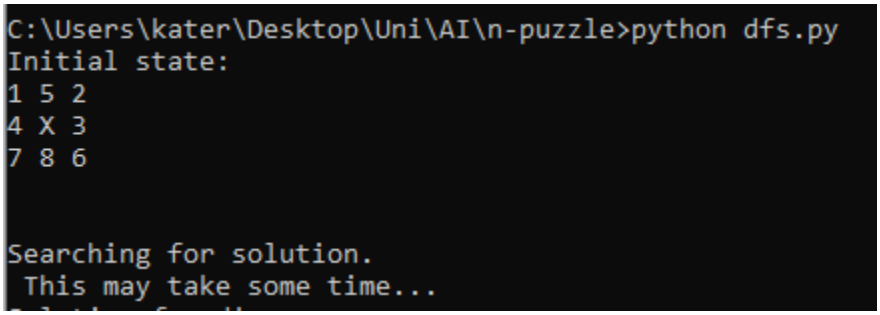
Τα βήματα των αλγορίθμων που αναλύθηκαν παραπάνω υλοποιούνται με την χρήση συναρτήσεων. Η φύση του προβλήματος είναι τέτοια που η αναδρομική κλήση των συναρτήσεων είναι μια λογική προσέγγιση. Ωστόσο, η Python, παρά το ότι είναι μια πολύ ευέλικτη γλώσσα προγραμματισμού, περιορίζει τις αναδρομές μεγάλου βάθους. Επομένως, μια καλύτερη προσέγγιση, είναι η επαναληπτική κλήση των συναρτήσεων, με τη χρήση δομών επανάληψης, των οποίων η περάτωση θα γίνεται με κάποια συνθήκη.

2. Εκτέλεση προγραμμάτων

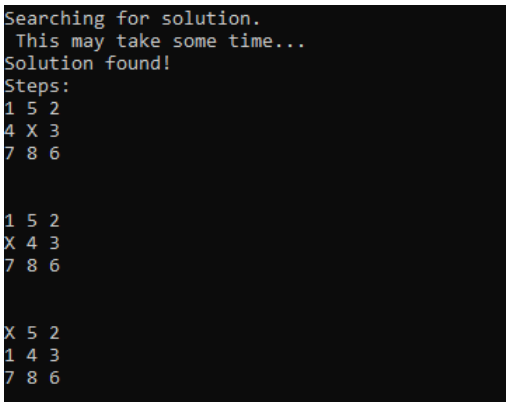
Ακολουθούν screenshots από την εκτέλεση των προγραμμάτων.

Εκτέλεση αλγορίθμου dfs:

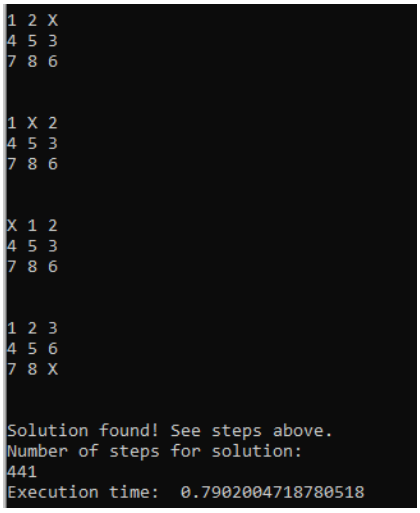
Αρχικά εμφανίζεται η αρχική κατάσταση του προβλήματος και μερικά μηνύματα προς τον χρήστη.



Εάν ο αλγόριθμος βρει λύση, εμφανίζει σχετικό μήνυμα και στην συνέχεια εμφανίζει τις καταστάσεις που οδηγούν στην λύση:



(μεσολαβούν πολλά βήματα)



Εάν ο αλγόριθμος δεν βρει λύσει, εμφανίζει σχετικό μήνυμα και το πρόγραμμα σταματά:

```
C:\Users\kater\Desktop\Uni\AI\n-puzzle>python dfs.py
Initial state:
7 6 2
5 X 1
3 4 8


Searching for solution.
 This may take some time...
Searching frontier is empty.
Could not find solution.
Exiting program...
Execution time:  6376.37236905098
```

Εκτέλεση αλγορίθμου bfs:

Όμοια με τον αλγόριθμο dfs, ο bfs εμφανίζει την αρχική κατάσταση και εφόσον υπάρχει λύση, εμφανίζει τα βήματα της.

```
C:\Users\kater\Desktop\Uni\AI\n-puzzle>python bfs.py
Initial state:
1 8 2
X 4 3
7 6 5


Searching for solution.
 This may take some time...
Solution found!
Steps:
1 8 2
X 4 3
7 6 5


X 8 2
1 4 3
7 6 5


1 8 2
4 X 3
7 6 5
```

(μεσολαβούν πολλά βήματα)

```
X 1 3
4 2 5
7 8 6


1 2 3
4 5 6
7 8 X


Solution found! See steps above.
Number of steps for solution:
316
Execution time:  0.5939078330993652

C:\Users\kater\Desktop\Uni\AI\n-puzzle>
```

Παρατήρηση: Όταν ο αλγόριθμος δεν βρίσκει λύση, η αναζήτηση μπορεί να πάρει μέχρι και 2 ώρες.

```
Initial state:
2 4 8
7 X 1
5 3 6


Searching for solution.
 This may take some time...
Searching frontier is empty.
Could not find solution.
Exiting program...
Execution time:  7137.8914268016815
```

Χρόνος σε δευτερόλεπτα

3. Ανάλυση κώδικα

Ακολουθούν screenshot του κώδικα και ανάλυσή του.

3.1 Αλγόριθμος dfs:

Συνάρτηση main()

```
def main():
    # Setting initial state
    # initial_state = creating_initial_state() # For random initial state
    initial_state = [[1, 5, 2], [4, None, 3], [7, 8, 6]]
    print("Initial state: ")
    print_array(initial_state)
    # Setting puzzle goal
    goal = [[1, 2, 3], [4, 5, 6], [7, 8, None]]
    # Initializing frontier and setting initial state as frontier
    frontier = []
    frontier.insert(0, initial_state)
    # Initializing close set
    close_set = []
    # Starting solution
    solution(frontier, goal, close_set)
```

Στη συνάρτηση main γίνονται οι απαραίτητες προεργασίες για να ξεκινήσει η επίλυση του προβλήματος.

Αρχικά, θέτεται η αρχική κατάσταση:

```
initial_state = [[1, 5, 2], [4, None, 3], [7, 8, 6]]
```

Η αρχική κατάσταση μπορεί να δημιουργηθεί και τυχαία βγάζοντας από σχόλιο την εντολή:

```
# initial_state = creating_initial_state() # For random initial state
```
Για την ευκολία του παραδείγματος έχει δημιουργηθεί μια αρχική κατάσταση η οποία είναι επιλύσιμη.

Στη συνέχεια, εκχωρείται στην μεταβλητή goal η επιθυμητή κατάσταση την οποία καλείται ο αλγόριθμος να βρει.

```
goal = [[1, 2, 3], [4, 5, 6], [7, 8, None]]
```
Επόμενο βήμα είναι η αρχικοποίηση του μετώπου αναζήτησης σε μορφή λίστας και η εισαγωγή της αρχικής κατάστασης (initial_state) στην πρώτη θέση της λίστας (0) με την εντολή insert.

```
frontier = []
frontier.insert(0, initial_state)
```
Τέλος, αρχικοποιείται το κλειστό σύνολο σε μορφή λίστας και καλείται η συνάρτηση solution με ορίσματα τα frontier, goal και close set που μόλις δημιουργήθηκαν.

Συνάρτηση solution()

```
def solution(frontier, goal, close_set):
    print("Searching for solution.\n This may take some time...")
    # Setting current state as the initial state for first iteration
    current_state = set_current_state(frontier)
    while current_state != goal:
        # Popping first element of frontier (the one that is about to hbe examined)
        frontier.pop(0)
        # Todo: Debug code
        # print("Frontier: ", len(frontier), " \t", "Close set: ", len(close_set))
        # If current state has not been examined yet,
        # find its children, put them in frontier and insert state in close set
        if current_state not in close_set:
            children = find_children(current_state)
            frontier = set_frontier(frontier, children)
            close_set = set_close_set(current_state, close_set)
        # If frontier is empty, a solution cannot be found
        if not frontier:
            print("Searching frontier is empty. \nCould not find solution. \nExiting program...")
            print("Execution time: ", time.time() - start_time)
            sys.exit()
        current_state = set_current_state(frontier)

    # This code will run only if the above while loop is escaped, meaning, a solution has been found
    print("Solution found! \nSteps: ")
    for i in close_set:
        print_array(i)
    print_array(goal)
    print("Solution found! See steps above.")
    print("Number of steps for solution: ")
    print(len(close_set))
    print("Execution time: ", time.time() - start_time)
    sys.exit()
```

Η συνάρτηση solution είναι ο κορμός του προγράμματος. Σε αυτή την συνάρτηση καλούνται επαναληπτικά οι υπόλοιπες συναρτήσεις που αποτελούν τα βήματα των αλγορίθμων. Δηλαδή, τα βήματα των δύο αλγορίθμων που αναλύθηκαν σύντομα στην εισαγωγή, καλούνται επαναληπτικά στην συνάρτηση solution. Αρχικά, θα αναλυθεί ολόκληρη η συνάρτηση solution και ύστερα οι συναρτήσεις τις οποίες καλεί.

Για την εισαγωγή στον βρόχο, εκχωρείται η μεταβλητή current_state η οποία καθορίζει την κατάσταση η οποία εξετάζεται σε κάθε επανάληψη.

```
current_state = set_current_state(frontier)
```

Πλέον, μπορεί να ξεκινήσει η επανάληψη των βημάτων. Η συνθήκη στην κορυφή του βρόχου εξετάζει εάν η εξεταζόμενη κατάσταση (current_state) είναι η επιθυμητή κατάσταση.

```
while current_state != goal:
```

Εάν αυτή η συνθήκη γίνει αληθής, τότε ο βρόχος τερματίζει και το πρόγραμμα εμφανίζει τα βήματα που ακολούθησε για την εύρεση της λύσης, δηλαδή, εμφανίζει της καταστάσεις που υπάρχουν στο κλειστό σύνολο και στην συνέχεια τερματίζεται. Επίσης, εμφανίζει τον χρόνο που χρειάστηκε για την περάτωση του αλγορίθμου.

```
# This code will run only if the above while loop is escaped, meaning, a
solution has been found
print("Solution found! \nSteps: ")
for i in close_set:
    print_array(i)
print_array(goal)
print("Solution found! See steps above.")
print("Number of steps for solution: ")
print(len(close_set))
print("Execution time: ", time.time() - start_time)
sys.exit()
```

Εάν το while current_state != goal: δεν είναι αληθές, τότε γίνονται τα εξής βήματα:

Αφαιρείται το πρώτο στοιχείο του frontier με την εντολή pop(). Αφαιρείται δηλαδή η κατάσταση που εξετάζεται σε αυτή την επανάληψη.

```
# Popping first element of frontier (the one that is about to be examined)
frontier.pop(0)
```

Στην συνέχεια, γίνεται ένας ακόμα έλεγχος. Ελέγχεται αν η current_state έχει εξετασθεί ξανά, δηλαδή, εάν υπάρχει στο close_set. Εάν δεν υπάρχει, βρίσκει τις καταστάσεις παιδιά της current_state, θέτει καινούργιο frontier βάζοντας του τις καταστάσεις παιδιά και εισάγει το current_state στο close_set.

```
if current_state not in close_set:
    children = find_children(current_state)
    frontier = set_frontier(frontier, children)
    close_set = set_close_set(current_state, close_set)
```

Τελευταίος έλεγχος που γίνεται είναι εάν το μέτωπο αναζήτησης είναι άδειο. Εάν είναι, σημαίνει ότι έχουν εξετασθεί όλες οι πιθανές καταστάσεις και δεν έχει βρεθεί λύση. Στην περίπτωση αυτή, εμφανίζεται σχετικό μήνυμα και το πρόγραμμα τερματίζεται.

```
# If frontier is empty, a solution cannot be found
if not frontier:
    print("Searching frontier is empty. \nCould not find solution. \nExiting program...")
    print("Execution time: ", time.time() - start_time)
    sys.exit()
```

Τέλος, ανανεώνεται το current_state και ο βρόχος συνεχίζεται είτε μέχρι να βρεθεί λύση, είτε μέχρι να αδειάσει το μέτωπο αναζήτησης.

```
current_state = set_current_state(frontier)
```

Συνάρτηση set_current_state()

```
# Setting new current state that is to be examined
def set_current_state(frontier):
    new_current_state = frontier[0]
    return new_current_state
```

Η συνάρτηση set_current_state χρησιμοποιείται για να ανανεωθεί η κατάσταση που πρόκειται να εξετασθεί. Παίρνει ως όρισμα το frontier διότι το new_current_state είναι το πρώτο στοιχείο του μετώπου αναζήτησης (frontier[0]). Επιστρέφει πίσω μια νέα κατάσταση.

Συνάρτηση find_children()

```
# Finding children of current state
def find_children(current_state):
    children = []
    # Calling neighbors function to locate the neighbors of null, thus locating the children
    neighbors = find_neighbors(current_state)

    for z in range(len(neighbors)):
        # Resetting child with current state information after each iteration.
        # This way, after finding each child we can reset the variable to look for the rest
        child = copy.deepcopy(current_state)
        # Searching for the null and neighbors to swap them
        for i in range(3):
            for j in range(3):
                if current_state[i][j] is None:
                    break
            else:
                continue
            break
        for x in range(3):
            for y in range(3):
                if current_state[x][y] == neighbors[z]:
                    child[x][y], child[i][j] = child[i][j], child[x][y]
                    break
            else:
                continue
            break
        children.append(child)
    return children
```

Η συνάρτηση find_children() βρίσκει τις καταστάσεις παιδιά της current_state. Παίρνει ως όρισμα το current_state και επιστρέφει μια λίστα από καταστάσεις παιδιά.

Αρχικά, αρχικοποιείται μια λίστα children.

```
children = []
```

Έπειτα, καλείται η συνάρτηση find_neighbors η οποία εντοπίζει το κενό πλακάκι (None) και βρίσκει τους γείτονες του, δηλαδή, τα σημεία που μπορεί να μετατοπιστεί. Κάθε μετατόπιση αντιπροσωπεύει μία κατάσταση παιδί.

Διευκρίνηση: Η λίστα neighbors είναι μία λίστα με αριθμούς. Πχ [1,4,6,7] Δηλαδή, το κενό πλακάκι συνορεύει με αυτούς τους αριθμούς.

```
neighbors = find_neighbors(current_state)
```

Στη συνέχεια, πρέπει να γίνουν οι αντιμεταθέσεις για κάθε γείτονα και να αποθηκευτούν στην λίστα children ώστε να ολοκληρωθεί οι λίστα των καταστάσεων παιδιών. Αυτό, επιτυγχάνεται με πέντε βρόχους.

```
for z in range(len(neighbors)):
    # Resetting child with current state information after each iteration.
    # This way, after finding each child we can reset the variable to look for the rest
    child = copy.deepcopy(current_state)
    # Searching for the null and neighbors to swap them
    for i in range(3):
        for j in range(3):
            if current_state[i][j] is None:
                break
        else:
            continue
        break
    for x in range(3):
        for y in range(3):
            if current_state[x][y] == neighbors[z]:
                child[x][y], child[i][j] = child[i][j], child[x][y]
                break
        else:
            continue
        break
    children.append(child)
return children
```

Ο πρώτος βρόχος προσπελαύνει την λίστα με τους γείτονες. Για κάθε γείτονα, αντιγράφει την τωρινή κατάσταση σε μια μεταβλητή child διότι κάθε γείτονας είναι μία εν δυνάμει κατάσταση παιδί. Στην μεταβλητή child θα γίνει η αντιμετάθεση ώστε να προκύψει το παιδί.

```
for z in range(len(neighbors)):
child = copy.deepcopy(current_state)
```

Με δύο νέους βρόχους, αναζητά το κενό πλακάκι(None). Μόλις το εντοπίσει, κάνει break από τους βρόχους. Έτσι αποθηκεύονται οι συντεταγμένες του None στο i και το j.

```
for i in range(3):
    for j in range(3):
        if current_state[i][j] is None:
            break
    else:
        continue
    break
```

Στην συνέχεια, με δύο τελευταίους βρόχους, το πρόγραμμα ψάχνει να βρει τον εκάστοτε γείτονα. Μόλις τον εντοπίσει, αντικαθιστά στην μεταβλητή child το None με τον γείτονα και βγαίνει από τον βρόχο.

```
for x in range(3):
    for y in range(3):
        if current_state[x][y] == neighbors[z]:
            child[x][y], child[i][j] = child[i][j], child[x][y]
```

```
            break
    else:
        continue
    break
```

Τέλος, εισάγει το child στο τέλος της λίστας children και είτε συνεχίζει με τον επόμενο γείτονα, ή επιστρέφει τη λίστα με τα παιδιά.

```
    children.append(child)
return children
```

Συνάρτηση find_neighbors()

```
# Finding neighbors of null cell of current state
def find_neighbors(current_state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if current_state[i][j] is None:
                if j == 0:
                    if i == 0:
                        neighbors.append(current_state[i + 1][j])
                        neighbors.append(current_state[i][j + 1])
                        return neighbors
                    elif i == 1:
                        neighbors.append(current_state[i + 1][j])
                        neighbors.append(current_state[i - 1][j])
                        neighbors.append(current_state[i][j + 1])
                        return neighbors
                    else:
                        neighbors.append(current_state[i - 1][j])
                        neighbors.append(current_state[i][j + 1])
                        return neighbors
                elif j == 1:
                    if i == 0:
                        neighbors.append(current_state[i + 1][j])
                        neighbors.append(current_state[i][j + 1])
                        neighbors.append(current_state[i][j - 1])
                        return neighbors
                    elif i == 1:
                        neighbors.append(current_state[i + 1][j])
                        neighbors.append(current_state[i - 1][j])
                        neighbors.append(current_state[i][j + 1])
                        neighbors.append(current_state[i][j - 1])
                        return neighbors
```

```
                    return neighbors
                else:
                    neighbors.append(current_state[i][j + 1])
                    neighbors.append(current_state[i - 1][j])
                    neighbors.append(current_state[i][j - 1])
                    return neighbors
            else:
                if i == 0:
                    neighbors.append(current_state[i + 1][j])
                    neighbors.append(current_state[i][j - 1])
                    return neighbors
                elif i == 1:
                    neighbors.append(current_state[i + 1][j])
                    neighbors.append(current_state[i - 1][j])
                    neighbors.append(current_state[i][j - 1])
                    return neighbors
                else:
                    neighbors.append(current_state[i - 1][j])
                    neighbors.append(current_state[i][j - 1])
                    return neighbors
```

Η συνάρτηση find_neighbors παίρνει ως όρισμα το current_state και βρίσκει τους γείτονες του κενού πλακακιού(None)

Αρχικοποιείται μια λίστα neighbors στην οποία εισαχθούν οι γείτονες που θα εντοπιστούν.

```
neighbors = []
```

Αρχικά, η κατάσταση προσπελαύνεται για να βρεθεί το None.

```
for i in range(3):
    for j in range(3):
        if current_state[i][j] is None:
```

Μόλις το βρεθεί, διακρίνονται περιπτώσεις για τα διαφορετικά i και j αφού υποδεικνύουν τις διαφορετικές πιθανές θέσεις του None μέσα στην κορνίζα (γωνία, μέση, κέντρο).

Ανάλογα την θέση, εισάγονται στην λίστα neighbors οι διαθέσιμοι γείτονες.

```
if j == 0:
    if i == 0:
        neighbors.append(current_state[i + 1][j])
        neighbors.append(current_state[i][j + 1])
        return neighbors
    elif i == 1:
```

```
        neighbors.append(current_state[i + 1][j])
        neighbors.append(current_state[i - 1][j])
        neighbors.append(current_state[i][j + 1])
        return neighbors
    else:
        neighbors.append(current_state[i - 1][j])
        neighbors.append(current_state[i][j + 1])
        return neighbors
```

Σε κάθε περίπτωση, επιστρέφεται η λίστα neighbors για να μην γίνονται αχρείαστες επαναλήψεις.

Συνάρτηση set_close_set()

```
# Setting close set
def set_close_set(state, close_set):
    close_set.append(state)
    return close_set
```

Η συνάρτηση set_close_set δέχεται ως ορίσματα το current_state και το υπάρχον close_set.

Εισάγει στην λίστα close_set την κατάσταση που μόλις εξετάσθηκε και την επιστρέφει.

Συνάρτηση set_frontier()

```
# Setting frontier
def set_frontier(frontier, children):
    # Inserting children in frontier. Dfs algorithm prioritizes children, thus, children will be inserted first
    for i in children:
        frontier.insert(0, i)
    return frontier
```

Η συνάρτηση set_frontier δέχεται ως ορίσματα το υπάρχον frontier και τα children του state που μόλις ελέγχθηκε. Εισάγει τις καταστάσεις παιδιά στο μέτωπο αναζήτησης ανάλογα με τον αλγόριθμο που χρησιμοποιείται. Στην περίπτωση του dfs τα εισάγει στην αρχή της λίστας(θέση 0), γιατί δίνεται προτεραιότητα στις καινούργιες καταστάσεις που “ανακαλύφθηκαν” .

```
frontier.insert(0, i)
```

Συναρτήσεις creating_inital_state και print_array

```python
# Creating random initial state
def creating_initial_state():
    # Available numbers that can be used in puzzle
    possible_numbers = [1, 2, 3, 4, 5, 6, 7, 8]
    # Initializing the list
    init_state = [[0, 0, 0], [0, None, 0], [0, 0, 0]]
    for i in range(3):
        for j in range(3):
            if init_state[i][j] is not None:
                # Choosing random number from possible numbers
                x = random.choice(possible_numbers)
                # Removing the number that was chosen from possible numbers
                possible_numbers.remove(x)
                # Assigning number to puzzle's cell
                init_state[i][j] = x
    return init_state
```

```python
# Print list as array
def print_array(e):
    for i in range(3):
        for j in range(3):
            if e[i][j] is None:
                print("X", end=' ')
            else:
                print(e[i][j], end=' ')
        print()
    print("\n")
```

Στο πρόγραμμα υπάρχουν και δυο δευτερεύουσες συναρτήσεις, για την παραγωγή τυχαίων καταστάσεων, και την εκτύπωση το εμφολευμένων λιστών.

3.2 Αλγόριθμος bfs:

Ο αλγόριθμος breadth first search στα βήματα του είναι σχεδόν ίδιος με τον depth first search. Η διαφορά τους βρίσκεται στον τρόπο που γίνεται η εισαγωγή των καταστάσεων παιδιών στο μέτωπο αναζήτησης. Πρακτικά επομένως, η διαφορά τους βρίσκεται στην συνάρτηση set_frontier().

```
# Setting frontier
def set_frontier(frontier, children):
    # Inserting children in frontier. Bfs algorithm prioritizes parents, thus, children will be inserted last
    for i in children:
        frontier.insert(-1, i)
    return frontier
```

Στην περίπτωση του bfs η εισαγωγή των children γίνεται στο τέλος της λίστας(θέση -1) διότι δίνεται προτεραιότητα στις προ υπάρχουσες καταστάσεις.

```
frontier.insert(-1, i)
```