ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΤΕΛΙΚΗ ΕΡΓΑΣΙΑ ΜΑΘΗΜΑΤΟΣ

ΕΥΦΥΕΙΣ ΠΡΑΚΤΟΡΕΣ

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: ΘΕΜΗΣ ΠΑΝΑΓΙΩΤΟΠΟΥΛΟΣ

Ονοματεπώνυμο: Γκολέμι Κριστιάν

Αριθμός μητρώου: Π18029

3) ΑΝΑΠΤΥΞΗ GENERIC PLANNER

Επίλυση προβλήματος Water Jug χρησιμοποιώντας δύο διαφορετικούς αλγορίθμους αναζήτησης, υλοποιημένο σε Python.

Παρακάτω παρατίθεται ο κώδικας επίλυσης του συγκεκριμένου προβλήματος, ο οποίος θα αναλυθεί στη συνέχεια:

```
"""
WATER JUG PROBLEM
2 water jugs with an initial amount of water: jugA and jugB (the initial amount can be defined by the user)
jugA's capacity is 70ml and jugB's capacity is 50ml (the jugs do not have markings on them)
The goal is to fill one of the jugs with 40ml of water
The actions we can take are the following:
    1) Fill the first jug
    2) Fill the second jug
    3) Empty the first jug
    4) Empty the second jug
    5) Pour the first jug into the second jug
    6) Pour the second jug into the first jug
"""


def choose():
    """
    Function that gives the user the ability to choose the algorithm he wants to implement in order to solve the problem
    """

    press = input("-----CHOOSE THE ALGORITHM YOU WANT TO USE. ENTER 'B' FOR BFS, 'D' FOR DFS: \n")
    # take only the first letter from the user's input
    press = press[0].lower()

    # check for invalid input
    while press != 'b' and press != 'd':
        press = input("INVALID INPUT! CHOOSE THE ALGORITHM YOU WANT TO USE. ENTER 'B' FOR BFS, 'D' FOR DFS: ")
```

```
        press = press[0].lower()

    if press == 'd':
        return True
    else:
        return False

def alreadyVisited(state, visited):
    """
    Function to check whether a state has been visited or not
    """

    # if the state has been visited returns True, otherwise returns False
    return visited.get(str(state), False)

def goalReached(path, goal_amount):
    """
    Function to check whether the goal has been reached or no
    """

    if path[-1][0] == goal_amount or path[-1][1] == goal_amount:
        return True
    else:
        return False

def createChildren(jugs_capacities, path, visited):
    """
    Function that creates the possible next states based on the current state
    """

    possible_paths = []     # list that stores the possible paths followed based on the current state
    next_states = []        # list that stores all the possible next states based on the current state
```

```
state = []

a_max = 70
b_max = 50

a = path[-1][0]     # initial amount of water in the first jug
b = path[-1][1]     # initial amount of water in the second jug

# 1) fill in the first jug
state.append(a_max)
state.append(b)
if not alreadyVisited(state, visited):
    next_states.append(state)
state = []

# 2) fill in the second jug
state.append(a)
state.append(b_max)
if not alreadyVisited(state, visited):
    next_states.append(state)
state = []

# 3) pour water from the second jug to the first jug
state.append(min(a_max, a + b))
state.append(b - (state[0] - a))
if not alreadyVisited(state, visited):
    next_states.append(state)
state = []

# 4) pour water from the first jug to the second jug
state.append(min(a + b, b_max))
state.insert(0, a - (state[0] - b))
```

```
    if not alreadyVisited(state, visited):
        next_states.append(state)
    state = []

    # 5) empty the first jug
    state.append(0)
    state.append(b)
    if not alreadyVisited(state, visited):
        next_states.append(state)
    state = []

    # 6) empty the second jug
    state.append(a)
    state.append(0)
    if not alreadyVisited(state, visited):
        next_states.append(state)

    for i in range(0, len(next_states)):
        temp = list(path)
        temp.append(next_states[i])
        possible_paths.append(temp)

    return possible_paths

def search(initial_state, jugs_capacities, goal_amount, visited, choice):
    """
    Function to search using either BFS or DFS for the wanted state and return the followed path
    """

    if choice:
        print("-----IMPLEMENTING DFS")
    else:
```

```
    print("-----IMPLEMENTING BFS")


found = False


# search_front represents metwpo anazhthshs
search_front = collections.deque()
search_front.appendleft(initial_state)


while len(search_front) != 0:
    # path represents mikroskopio
    path = search_front.popleft()


    # mark the current state as visited
    visited[str(path[-1])] = True


    if goalReached(path, goal_amount):
        found = True
        goal = path
        break


    next_moves = createChildren(jugs_capacities, path, visited)
    for i in next_moves:
        if choice:
            # implementing DFS
            search_front.appendleft(i)
        else:
            # implementing BFS
            search_front.append(i)


if found:
    print("-----THE GOAL HAS BEEN ACHIEVED, PRINTING THE PATH FOLLOWED TO THE GOAL STATE\n")
    for i in range(0, len(goal)):
```

```
        print(i, ". ", goal[i])
    else:
        print("-----THE PROBLEM CANNOT BE SOLVED, SORRY\n")


def waterJugs():
    jug1_initial = int(input("-----ENTER INITIAL AMOUNT OF WATER IN JUG 1: "))
    jug2_initial = int(input("-----ENTER INITIAL AMOUNT OF WATER IN JUG 2: "))

    initial_state = [[jug1_initial, jug2_initial]]        # initially both jugs are empty
    jugs_capacities = [70, 50]                            # list to store the max capacities of the jugs
    goal_amount = 40                                      # the wanted amount of water in a jug
    visited = {}                                          # dictionary to store the visited states
    choice = choose()                                     # choice calls def choose, if the user chooses DFS choice = True, otherwise choice = False
    search(initial_state, jugs_capacities, goal_amount, visited, choice)
```

Ανάλυση των παραπάνω συναρτήσεων αναλυτικά:

def choose():

Η παραπάνω συνάρτηση δίνει τη δυνατότητα στον χρήστη του προγράμματος να επιλέξει με ποιόν τρόπο θα επιλυθεί το πρόβλημα. Η συνάρτηση δέχεται είσοδο από το πληκτρολόγιο και ανάλογα με το πλήκτρο που θα πατήσει ο χρήστης ορίζεται αν θα χρησιμοποιηθεί BFS ή DFS. Σε περίπτωση που ο χρήστης πατήσει d ή D, η συνάρτηση επιστρέφει True και το πρόβλημα επιλύεται με DFS, ενώ διαφορετικά αν ο χρήστης πατήσει b ή B, η συνάρτηση επιστρέφει False και το πρόβλημα επιλύεται με BFS.

def alreadyVisited(state, visited):

Η συνάρτηση alreadyVisited(), ελέγχει εάν έχουμε επισκεφθεί ήδη μια κατάσταση ή όχι.

```
# if the state has been visited returns True, otherwise returns False
return visited.get(str(state), False)
```

Μέσω του παραπάνω return statement, η συνάρτηση επιστρέφει True σε περίπτωση που έχουμε επισκεφθεί ήδη την κατάσταση που δίνεται ως όρισμα, ή διαφορετικά False σε περίπτωση που είναι η πρώτη φορά που ελέγχουμε την κατάσταση. Η συνάρτηση get() που καλείται πάνω στο dictionary visited{}, ελέγχει αν υπάρχει το στοιχείο με key str(state) μέσα στο dictionary. Η τρέχουσα κατάσταση state μετατρέπεται σε string μέσω της συνάρτησης

str(), καθώς το state είναι λίστα και οι λίστες στην python είναι unhashable, ενώ τα strings είναι hashable.

def goalReached(path, goal_amount):

Η παραπάνω συνάρτηση ελέγχει εάν επιτεύχθηκε ο στόχος του προγράμματος, ο οποίος στη δική μας περίπτωση είναι να φτάσουμε ένα από τα δύο ποτήρια να περιέχουν 40ml νερό.

```
if path[-1][0] == goal_amount or path[-1][1] == goal_amount:
    return True
else:
    return False
```

Η λίστα path περιέχει το μονοπάτι που έχει ακολουθηθεί μέχρι την τρέχουσα κατάσταση, επομένως στο τέλος της λίστας path υπάρχει η τρέχουσα κατάσταση η οποία θα είναι π.χ. της μορφής [[…], […], …, [20, 30]]. Εάν ένα από τα δύο στοιχεία της τελευταίας υπολίστας που είναι και η τρέχουσα κατάσταση είναι ίσο με goal_amount = 40, τότε επετεύχθει ο αρχικός μας στόχος και η συνάρτηση επιστρέφει True, ενώ διαφορετικά επιστρέφει False.

def createChildren(jugs_capacities, path, visited):

Η συνάρτηση createChildren() επιστρέφει μια λίστα από τα πιθανά μονοπάτια που μπορούν να ακολουθηθούν με βάση την τρέχουσα κατάσταση στην οποία βρισκόμαστε. Ανάλογα με την κατάσταση στην οποία βρισκόμαστε, μπορούμε να κάνουμε μια από τις παρακάτω κινήσεις:

1) γέμισε το 1ο ποτήρι

2) γέμισε το 2ο ποτήρι

3) άδειασε το 1ο ποτήρι

4) άδειασε το 2ο ποτήρι

5) ρίξε νερό στο 1ο ποτήρι από το 2ο ποτήρι

6) ρίξε νερό στο 2ο ποτήρι από το 1ο ποτήρι

```
a = path[-1][0]     # initial amount of water in the first jug
b = path[-1][1]     # initial amount of water in the second jug
```

Ορίζονται οι μεταβλητές a, b οι οποίες περιέχουν την τρέχουσα κατάσταση τον δύο ποτηριών τη στιγμή που καλείται η συνάρτηση createChildren(). Το a είναι το 1ο στοιχείο της τελευταίας υπολίστας της λίστας path, ενώ το b είναι αντίστοιχα το 2ο.

Στη συνέχεια δημιουργείται η λίστα next_states[], η οποία περιέχει τις επόμενες καταστάσεις που μπορούμε να μεταβούμε με βάση την τρέχουσα κατάσταση.

```
# 1) fill in the first jug
state.append(a_max)
state.append(b)
if not alreadyVisited(state, visited):
    next_states.append(state)
state = []
```

Για παράδειγμα μια από τις κινήσεις που μπορούμε να κάνουμε είναι να γεμίσουμε το 1ο ποτήρι. Η λίστα state[] αποθηκέυει μια προσωρινή κατάσταση παιδί, η οποία αν δεν έχει την έχουμε ξαναεπισκεφθεί νωρίτερα αποθηκέυεται στη λίστα next_states[], η οποία περιέχει όλες τις καταστάσεις παιδία της τρέχουσας κατάστασης. Αρχικά, εκτελώντας την κίνηση αυτή η λίστα state[] γίνεται state[a_max, b] που σημαίνει ότι το πρώτο ποτήρι γεμίζει πέρνοντας την τιμή a_max = 70, ενώ το 2ο ποτήρι μένει ως είχε στην προηγούμενη κατάσταση καθώς προς το παρόν δεν μας ενδιαφέρει. Έπειτα γίνεται έλεγχος αν έχουμε επισκεφθεί νωρίτερα τη συγκεκριμένη κατάσταση και αν είναι η πρώτη φορά που την επισκεπτόμαστε, προστίθεται στην λίστα next_states[]. Όμοια λειτουργεί η συνάρτηση και για τις υπόλοιπες κινήσεις που μπορούμε να κάνουμε.

```
for i in range(0, len(next_states)):
    temp = list(path)
    temp.append(next_states[i])
    possible_paths.append(temp)

return possible_paths
```

Τέλος, αφού γίνουν όλες οι κινήσεις που μπορούν να γίνουν με βάση την τρέχουσα κατάσταση, δημιουργείται η λίστα possible_paths η οποία περιέχει όλα τα δυνατά μονοπάτια που μπορούμε να πάρουμε ξεκινώντας από την προηγούμενη κατάσταση, με τον παραπάνω κώδικα. Για παράδειγμα αν η αρχική κατάσταση είναι η [0, 0], η λίστα next_states

θα είναι η [[70, 0], [0, 50]] καθώς όταν τα ποτήρια είναι άδεια το μόνο που μπορούμε να κάνουμε είναι να γεμίσουμε ένα από τα δύο ποτήρια. Έχοντας τη λίστα next_states[], δημιουργείται τέλος η λίστα possible_paths[], η οποία στο συγκεκριμένο παράδειγμα θα είναι η εξής: [[[0, 0], [70, 0]], [[0, 0], [0, 50]]].

def search(initial_state, jugs_capacities, goal_amount, visited, choice):

Η συνάρτηση search(), είναι ουσιαστικά η συνάρτηση στην οποία υλοποιείται ο αλγόριθμος BFS ή DFS (ανάλογα με την επιλογή του χρήστη) ώστε να φτάσουμε από την αρχική κατάσταση [[0, 0]] στην επιθυμητή κατάσταση όπου ένα από τα δύο ποτήρια περιέχουν 40 ml νερό.

```
if choice:
    print("-----IMPLEMENTING DFS")
else:
    print("-----IMPLEMENTING BFS")
```

Ξεκινώντας, αν ο χρήστης έχει πληκτρολογήσει ‘d’, η συνάρτηση choose() επιστρέφει την τιμή True και αφού choice = choose(), η μεταβλητή choice παίρνει και αυτή την τιμή True και η συνάρτηση τυπώνει ‘’-----IMPLEMENTING DFS’’. Όμοια γίνεται στην περίπτωση που ο χρήστης επιλέξει BFS.

```
# search_front represents metwpo anazhthshs
search_front = collections.deque()
search_front.appendleft(initial_state)

while len(search_front) != 0:
    # path represents mikroskopio
    path = search_front.popleft()

    # mark the current state as visited
    visited[str(path[-1])] = True

    if goalReached(path, goal_amount):
        found = True
        goal = path
        break
```

Ξεκινώντας, ορίζουμε το μέτωπο αναζήτησης (search_front), το οποίο είναι μια μορφή λίστας από τη βιβλιοθήκη collections ονόματι deque, για ευκολότερο χειρισμό της λίστας και αφαίρεση ή προσθήκη στοιχείων στο δεξί ή στο αριστερό της μέρος. Έπειτα, προσθέτουμε

στο αριστερό μέρος της λίστας την αρχική κατάσταση η οποία είναι η initial_state = [[0, 0]]. Όσο το μέγεθος του μετώπου αναζήτησης δεν είναι 0, προσθέτουμε στο μονοπάτι path (μικροσκόπιο) το πρώτο στοιχείο του μετώπου αναζήτησης και ταυτόχρονα το αφαιρούμε από το ίδιο (χρησιμοποιώντας τη συνάρτηση popleft()). Ταυτόχρονα ορίζουμε ότι έχουμε επισκεφθεί την τρέχουσα κατάσταση μέσα στο dictionary visited ώστε να μην την ξαναεπισκεφθούμε και καταλήγουμε σε ατέρμονα βρόγχο. Στη συνέχεια, γίνεται έλεγχος αν η τρέχουσα κατάσταση η οποία είναι η τελευταία υπολίστα της λίστας path, είναι goal state. Σε περίπτωση που είναι, η μεταβλητή found γίνεται True, η αναζήτηση τερματίζει και τελικά εμφανίζεται το μονοπάτι το οποίο ακολουθήσαμε μέχρι να φτάσουμε στην τελική κατάσταση. Διαφορετικά σε περίπτωση που δεν έχουμε φτάσει ακόμα στο goal state, εκτελείται ο παρακάτω κώδικας:

```
next_moves = createChildren(jugs_capacities, path, visited)
for i in next_moves:
    if choice:
        # implementing DFS
        search_front.appendleft(i)
    else:
        # implementing BFS
        search_front.append(i)
```

Στο παραπάνω κομμάτι κώδικα γίνονται τα εξής βήματα: Αρχικά ορίζουμε ότι next_moves = createChildren(jugs_capacities, path, visited), δηλαδή η next_moves είναι πλέον μια λίστα από τα πιθανά μονοπάτια που μπορούμε να ακολουθήσουμε ξεκινώντας από την τρέχουσα κατάσταση (η λίστα possible_paths) που επιστρέφει η συνάρτηση createChildren(). Για κάθε στοιχείο που είναι μέσα σε αυτή τη λίστα (για κάθε πιθανό μονοπάτι δηλαδή), προσθέτουμε το μονοπάτι αυτό είτε στο αριστερό μέρος του μετώπου αναζήτησης searchFront είτε στο δεξί κομμάτι ανάλογα με την επιλογή που έχει κάνει ο χρήστης για τον αλγόριθμο αναζήτησης. Αφού προστεθούν όλα τα πιθανά μονοπάτια, επαναλαμβάνεται η παραπάνω διαδικασία (path = searchFront.popleft()……) εώς ότου φτάσουμε στην τελική επιθυμητή κατάσταση.

```
if found:
    print("-----THE GOAL HAS BEEN ACHIEVED, PRINTING THE PATH FOLLOWED TO THE GOAL STATE\n")
    for i in range(0, len(goal)):
        print(i, ". ", goal[i])
else:
    print("-----THE PROBLEM CANNOT BE SOLVED, SORRY\n")
```

Φτάνοντας στην τελική κατάσταση, τυπώνεται το μονοπάτι που ακολουθήσαμε μέχρι να φτάσουμε στην τελική κατάσταση. Σε περίπτωση που δεν μπορεί να βρεθεί λύση στο πρόβλημα τυπώνεται ‘’-----THE PROBLEM CANNOT BE SOLVED, SORRY’’

Όλες οι παραπάνω συναρτήσεις και ο αλγόριθμος ξεκινούν να εκτελούνται αφού η συνάρτηση waterJugs() του κώδικα καλέσει αρχικά τη συνάρτηση search() δίνοντας της τα κατάλληλα και επιθυμητά ορίσματα, τα οποία έχουν οριστεί με βάση τις ανάγκες του προβλήματος, ωστόσο μπορούν να αλλαχθούν ώστε να επιλύεται το πρόβλημα και με διαφορετικές τιμές.

```
def waterJugs():
    jug1_initial = int(input("-----ENTER INITIAL AMOUNT OF WATER IN JUG 1: "))
    jug2_initial = int(input("-----ENTER INITIAL AMOUNT OF WATER IN JUG 2: "))

    initial_state = [[jug1_initial, jug2_initial]]        # initially both jugs are empty
    jugs_capacities = [70, 50]                            # list to store the max capacities of the jugs
    goal_amount = 40                                      # the wanted amount of water in a jug
    visited = {}                                          # dictionary to store the visited states
    choice = choose()                                     # choice calls def choose, if the user chooses DFS choice = True, otherwise choice = False
    search(initial_state, jugs_capacities, goal_amount, visited, choice)
```

Παρακάτω παρατίθονται παραδείγματα εκτέλεσης του κώδικα:

BFS, με αρχικές ποσότητες ενδεικτικά 0 ml σε κάθε κανάτα

```
/opt/homebrew/bin/python3 "/Users/kristiangolemi/Desktop/Intelligent Agents/generic_planner.py"
kristiangolemi@Kristians-MBP Intelligent Agents % /opt/homebrew/bin/python3 "/Users/kristiangolemi/Desktop/Intelligent Agents/generic_planner.py"

CHOOSE ONE OF THE BELOW PROBLEMS TO SOLVE:
1. WATER JUG
2. BLOCKS WORLD
ENTER YOUR CHOICE [1-2]: 1
-----ENTER INITIAL AMOUNT OF WATER IN JUG 1: 0
-----ENTER INITIAL AMOUNT OF WATER IN JUG 2: 0
-----CHOOSE THE ALGORITHM YOU WANT TO USE. ENTER 'B' FOR BFS, 'D' FOR DFS:
B
-----IMPLEMENTING BFS
-----THE GOAL HAS BEEN ACHIEVED, PRINTING THE PATH FOLLOWED TO THE GOAL STATE

0 .  [0, 0]
1 .  [70, 0]
2 .  [20, 50]
3 .  [20, 0]
4 .  [0, 20]
5 .  [70, 20]
6 .  [40, 50]
```

BFS, με τυχαίες αρχικές ποσότητες σε κάθε κανάτα

```
/opt/homebrew/bin/python3 "/Users/kristiangolemi/Desktop/Intelligent Agents/generic_planner.py"
kristiangolemi@Kristians-MBP Intelligent Agents % /opt/homebrew/bin/python3 "/Users/kristiangolemi/Desktop/Intelligent Agents/generic_planner.py"

CHOOSE ONE OF THE BELOW PROBLEMS TO SOLVE:
1. WATER JUG
2. BLOCKS WORLD
ENTER YOUR CHOICE [1-2]: 1
-----ENTER INITIAL AMOUNT OF WATER IN JUG 1: 30
-----ENTER INITIAL AMOUNT OF WATER IN JUG 2: 50
-----CHOOSE THE ALGORITHM YOU WANT TO USE. ENTER 'B' FOR BFS, 'D' FOR DFS:
b
-----IMPLEMENTING BFS
-----THE GOAL HAS BEEN ACHIEVED, PRINTING THE PATH FOLLOWED TO THE GOAL STATE

0 .  [30, 50]
1 .  [70, 50]
2 .  [70, 0]
3 .  [20, 50]
4 .  [20, 0]
5 .  [0, 20]
6 .  [70, 20]
7 .  [40, 50]
```

DFS, με αρχικές ποσότητες ενδεικτικά 0 ml σε κάθε κανάτα

```
/opt/homebrew/bin/python3 "/Users/kristiangolemi/Desktop/Intelligent Agents/generic_planner.py"
kristiangolemi@Kristians-MBP Intelligent Agents % /opt/homebrew/bin/python3 "/Users/kristiangolemi/Desktop/Intelligent Agents/generic_planner.py"

CHOOSE ONE OF THE BELOW PROBLEMS TO SOLVE:
1. WATER JUG
2. BLOCKS WORLD
ENTER YOUR CHOICE [1-2]: 1
-----ENTER INITIAL AMOUNT OF WATER IN JUG 1: 0
-----ENTER INITIAL AMOUNT OF WATER IN JUG 2: 0
-----CHOOSE THE ALGORITHM YOU WANT TO USE. ENTER 'B' FOR BFS, 'D' FOR DFS:
d
-----IMPLEMENTING DFS
-----THE GOAL HAS BEEN ACHIEVED, PRINTING THE PATH FOLLOWED TO THE GOAL STATE

0 .  [0, 0]
1 .  [0, 50]
2 .  [50, 0]
3 .  [50, 50]
4 .  [70, 30]
5 .  [70, 0]
6 .  [20, 50]
7 .  [20, 0]
8 .  [0, 20]
9 .  [70, 20]
10 .  [40, 50]
```

DFS, με τυχαίες αρχικές ποσότητες σε κάθε κανάτα

```
/opt/homebrew/bin/python3 "/Users/kristiangolemi/Desktop/Intelligent Agents/generic_planner.py"
kristiangolemi@Kristians-MBP Intelligent Agents % /opt/homebrew/bin/python3 "/Users/kristiangolemi/Desktop/Intelligent Agents/generic_planner.py"

CHOOSE ONE OF THE BELOW PROBLEMS TO SOLVE:
1. WATER JUG
2. BLOCKS WORLD
ENTER YOUR CHOICE [1-2]: 1
-----ENTER INITIAL AMOUNT OF WATER IN JUG 1: 50
-----ENTER INITIAL AMOUNT OF WATER IN JUG 2: 70
-----CHOOSE THE ALGORITHM YOU WANT TO USE. ENTER 'B' FOR BFS, 'D' FOR DFS:
d
-----IMPLEMENTING DFS
-----THE GOAL HAS BEEN ACHIEVED, PRINTING THE PATH FOLLOWED TO THE GOAL STATE

0 .  [50, 70]
1 .  [50, 0]
2 .  [0, 0]
3 .  [0, 50]
4 .  [70, 50]
5 .  [70, 0]
6 .  [20, 50]
7 .  [20, 0]
8 .  [0, 20]
9 .  [70, 20]
10 .  [40, 50]
```

Επίλυση προβλήματος Blocks World χρησιμοποιώντας τον αλγόριθμο αναζήτησης A Star, υλοποιημένο σε Python.

Παρακάτω παρατίθεται ο κώδικας επίλυσης του συγκεκριμένου προβλήματος, ο οποίος θα αναλυθεί στη συνέχεια:

```
'''
BLOCKS WORLD PROBLEM
The blocks world is a planning domain in artificial intelligence.
The algorithm is similar to a set of wooden blocks of various shapes and colors sitting on a table.
The goal is to build one or more vertical stacks of blocks. Only one block may be moved at a time:
it may either be placed on the table or placed atop another block. Because of this, any blocks that are under another block
cannot be moved. Moreover, some kinds of blocks cannot have other blocks stacked on top of them.
'''


class PriorityQueue:
    '''
    Priotity queue is used to store all the possible children of a state in it,
    based on distance to goal, closest first.
    '''
    def __init__(self):
        self.heap = []
        self.count = 0

    # function to push an item to the priority queue
    def push(self, item, priority):
        heapq.heappush(self.heap, (priority, self.count, item))
        self.count += 1

    # function to remove or retrieve an item from the priority queue
    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
```

```
        return item


    def aStar(node):
        '''
        Implementation of the a star search algorithm. A star algorithm, is an algorithm
        used to find the shortest possible path from start to end states.
        We search the node that has the lowest combined cost and heuristic first.
        '''
        closed = []
        queue = PriorityQueue()                    # initialize a priority queue object
        queue.push(node, node.calculateCost())     # insert it in the priority queue along with its cost

        while True:
            if len(queue.heap) == 0:
                print ('-----FINAL STATE IS NOT ACHIEVABLE FROM GIVEN START STATE!')

            node = queue.pop()
            # check if the goal state has been achieved
            if node.goal():
                return

            if node.state not in closed:
                closed.append(node.state)
                for childNode in node.createChildren():
                    queue.push(childNode, childNode.calculateCost())

class State:
    def __init__(self, start_state, final_state, parent = None) :
        self.state = start_state
        self.finalSt = final_state
        self.parent = parent
        self.cost = 0
```

```
    if parent:
        self.cost = parent.cost + 1


# check if the goal state has been reached
def goal(self) :
    if self.state == self.finalSt:
        self.findPathFollowed()
        return True
    else:
        return False


def calculateHeuristic(self):
    '''
    Heuristic function to help decide which path to follow next.
    The heuristic function provides an estimate of the minimum cost required
    to reach from a given node to a target node.
    '''
    arg1 = arg2 = arg3 = 0

    arg1 = len(self.finalSt[0]) - len(self.state[0])

    for i in range(len(self.state[0])) :
        if self.state[0][i] != self.finalSt[0][i]:
            arg2 += 2

    for i in range(1, len(self.state)):
        for val in range(len(self.state[i]) - 1):
            if self.state[i][val] > self.state[i][val + 1]:
                arg3 += 1

    heuristic_value = arg1 + 4 * arg2 - arg3
```

```
    return heuristic_value


# function which generates all possible children of the current state
def createChildren(self):
    children = []
    for i, stack in enumerate(self.state):
        for j, stack1 in enumerate(self.state):
            if i != j and len(stack1):
                temp = copy.deepcopy(stack)
                child = copy.deepcopy(self)
                temp1 = copy.deepcopy(stack1)
                temp.append(temp1[-1])
                del temp1[-1]
                child.state[i] = temp
                child.state[j] = temp1
                child.parent = copy.deepcopy(self)
                children.append(child)

    return children


# calculate and display the path followed to reach from the start to the goal state
def findPathFollowed(self):
    node, path = self, []
    while node:
        path.append(node.state)
        node = node.parent

    print("\n-----SOLUTION FOUND, PRINTING THE PATH FOLLOWED TO THE GOAL STATE: ")
    move_nr = 0;
    for i in list(reversed(path)) :
        print(str(move_nr) + ". " + str(i))
        move_nr += 1
```

```
    def calculateCost(self) :
        return self.calculateHeuristic() + self.cost


def calculateStates(stacks, blocks):
    '''
    Function which calculates the starting and final states of the problem,
    given the stacks and blocks which are retrieved from the users input.
    '''
    # CALCULATE THE STARTING STATE
    stacks_nr = stacks
    digits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
    block_nrs = digits[:blocks]     # if blocks = 5, blocks_nrs = ['0', '1', '2', '3', '4']
    random.shuffle(block_nrs)

    startState = []
    while blocks:
        if stacks == 1:
            startState.append(block_nrs)
            break
        else:
            r = random.randint(1, blocks)
            s = block_nrs[:r]
            startState.append(s)

        blocks -= r
        stacks -= 1
        block_nrs = block_nrs[r:]

    while len(startState) < stacks_nr:
        startState += [[]]
```

```
    # randomize the stacks positions
    random.shuffle(startState)

    # CALCULATE THE FINAL STATE
    finalState = []
    for stack in startState:
        finalState += stack

    finalState.sort()
    finalState = [finalState]

    # complete the final states stacks with empty stacks
    for _ in range(len(startState) - 1):
        finalState += [[]]

    return startState, finalState

def blocksWorld():
    stacks = int(input("-----ENTER NUMBER OF STACKS: "))
    blocks = int(input("-----ENTER NUMBER OF BLOCKS: "))

    startState, finalState = calculateStates(stacks, blocks)
    print("\n-----START STATE: " + str(startState))
    print("-----FINAL STATE: " + str(finalState))

    PriorityQueue.aStar(State(startState, finalState))
```

Για την επίλυση του συγκεκριμένου προβλήματος, έχουμε δημιουργήσει δύο βασικές κλάσεις τις οποίες θα αναλύσουμε παρακάτω: PriorityQueue, State.

Η κλάση PriorityQueue χρησιμοποιείται για την αποθήκευση όλων των πιθανών καταστάσεων που μπορούμε να μεταβούμε από μια κατάσταση, λειτουργώντας ως μια δομή δεδομένων που αποθηκεύει πιθανές καταστάσεις μετάβασης με βάση την απόσταση από την κατάσταση στόχος, με την κοντινότερη να αποθηκεύεται πρώτα.

Η συγκεκριμένη κλάση αποτελείται από δύο βασικές συναρτήσεις που χρησιμοποιούνται για την είσοδο και έξοδο καταστάσεων ''παιδιών'' στη δομή δεδομένων, τις push() και pop(), οι οποίες φαίνονται στα παρακάτω στιγμιότυπα:

```
# function to push an item to the priority queue
def push(self, item, priority):
    heapq.heappush(self.heap, (priority, self.count, item))
    self.count += 1

# function to remove or retrieve an item from the priority queue
def pop(self):
    (_, _, item) = heapq.heappop(self.heap)

    return item
```

Η βασικότερη συνάρτηση της κλάσης PriorityQueue είναι η aStar(), μέσω της οποίας υλοποιείται ο αλγόριθμος αναζήτησης A Star. Ο συγκεκριμένος αλγόριθμος χρησιμοποιείται για την εύρεση του συντομότερου μονοπατιού ώστε να μεταβούμε από μια αρχική κατάσταση σε μια κατάσταση στόχος. Η συγκεκριμένη συνάρτηση αναζητά πρώτα την κατάσταση με το χαμηλότερο κόστος και heuristic πρώτα. Παρακάτω φαίνεται η υλοποίηση της χρησιμοποιώντας επεξηγηματικά σχόλια.

```
def aStar(node):
    '''
    Implementation of the a star search algorithm. A star algorithm, is an algorithm
    used to find the shortest possible path from start to end states.
    We search the node that has the lowest combined cost and heuristic first.
    '''
    closed = []
    queue = PriorityQueue()                    # initialize a priority queue object
    queue.push(node, node.calculateCost())     # insert it in the priority queue along with its cost

    while True:
        if len(queue.heap) == 0:
            print ('-----FINAL STATE IS NOT ACHIEVABLE FROM GIVEN START STATE!')

        node = queue.pop()
        # check if the goal state has been achieved
        if node.goal():
            return

        if node.state not in closed:
            closed.append(node.state)
            for childNode in node.createChildren():
                queue.push(childNode, childNode.calculateCost())
```

Η δεύτερη κλάση που έχει υλοποιηθεί για την επίλυση του προβλήματος είναι όπως αναφέραμε και προηγουμένως, η State, η οποία περιέχει συναρτήσεις για τον χειρισμό των καταστάσεων και λειτουργούν συνδυαστικά με τις συναρτήσεις της κλάσης PriorityQueue.

Παρακάτω επεξηγούνται η συναρτήσεις της συγκεκριμένης κλάσης ξεχωριστά:

def goal(self):

Η συγκεκριμένη συνάρτηση χρησιμοποιείται για τον έλεγχο αν η κατάσταση που της δίνεται ως όρισμα είναι η κατάσταση στόχος. Σε περίπτωση που έχουμε φτάσει στην κατάσταση στόχος, καλείται η συνάρτηση findPathFollowed() η οποία εμφανίζει στο χρήστη το μονοπάτι που ακολουθήσαμε για να καταλήξουμε στην τελική κατάσταση στόχος. Παρακάτω φαίνεται η συγκεριμένη συνάρτηση.

```
# check if the goal state has been reached
def goal(self) :
    if self.state == self.finalSt:
        self.findPathFollowed()
        return True
    else:
        return False
```

def calculateHeuristic(self):

Συνάρτηση η οποία χρησιμοποιείται για να μπορέσουμε να λάβουμε την απόφαση σχετικά με το ποία από τις πιθανές μεταβατικές καταστάσεις θα επιλέξουμε ώστε να φτάσουμε συντομότερα και με μικρότερο κόστος στην κατάσταση στόχος. Η calculateHeuristic() επιστρέφει έναν αριθμό για κάθε πιθανή κατάσταση παιδί, ο οποίος είναι μια προσέγγιση του ελάχιστου κόστους που απαιτείται για να φτάσουμε από τη συγκεκριμένη κατάσταση, στην κατάσταση στόχος.

```
def calculateHeuristic(self):
    '''
    Heuristic function to help decide which path to follow next.
    The heuristic function provides an estimate of the minimum cost required
    to reach from a given node to a target node.
    '''
    arg1 = arg2 = arg3 = 0

    arg1 = len(self.finalSt[0]) - len(self.state[0])

    for i in range(len(self.state[0])) :
        if self.state[0][i] != self.finalSt[0][i]:
            arg2 += 2

    for i in range(1, len(self.state)):
        for val in range(len(self.state[i]) - 1):
            if self.state[i][val] > self.state[i][val + 1]:
                arg3 += 1

    heuristic_value = arg1 + 4 * arg2 - arg3

    return heuristic_value
```

def createChildren(self):

Η συγκεκριμένη συνάρτηση χρησιμοποιείται για τον υπολογισμό όλων των καταστάσεων παιδιών που μπορούμε να μεταβούμε από την τρέχουσα κατάσταση. Για κάθε κατάσταση που της δίνουμε ως όρισμα, επιστρέφει μια λίστα με τις καταστάσεις που μπορούμε να μεταβούμε από αυτήν.

```
# function which generates all possible children of the current state
def createChildren(self):
    children = []
    for i, stack in enumerate(self.state):
        for j, stack1 in enumerate(self.state):
            if i != j and len(stack1):
                temp = copy.deepcopy(stack)
                child = copy.deepcopy(self)
                temp1 = copy.deepcopy(stack1)
                temp.append(temp1[-1])
                del temp1[-1]
                child.state[i] = temp
                child.state[j] = temp1
                child.parent = copy.deepcopy(self)
                children.append(child)

    return children
```

def findPathFollowed(self):

Συνάρτηση που όπως αναφέραμε και νωρίτερα, καλείται όταν φτάνουμε στην κατάσταση στόχος και εμφανίζει στο χρήστη το μονοπάτι με τις καταστάσεις που ακολουθήσαμε ώστε να φτάσουμε εκεί.

```
# calculate and display the path followed to reach from the start to the goal state
def findPathFollowed(self):
    node, path = self, []
    while node:
        path.append(node.state)
        node = node.parent

    print("\n-----SOLUTION FOUND, PRINTING THE PATH FOLLOWED TO THE GOAL STATE: ")
    move_nr = 0;
    for i in list(reversed(path)) :
        print(str(move_nr) + ". " + str(i))
        move_nr += 1
```

def calculateCost(self):

Τελευταία συνάρτηση της συγκεκριμένης κλάσης, καλείται για την εύρεση του συνολικού κόστους που απαιτείται για να μεταβούμε στην τελική κατάσταση στόχο, από μια άλλη κατάσταση που περνάμε ως όρισμα.

```
def calculateCost(self):
    return self.calculateHeuristic() + self.cost
```

def calculateStates(stacks, blocks:

Τελευταία συνάρτηση που έχει υλοποιηθεί για την επίλυση του προβλήματος Blocks World, χρησιμοποιείται για τον υπολογισμό της αρχικής και της τελικής κατάστασης που θέλουμε να μεταβούμε, χρησιμοποιώντας τα δεδομένα που εισάγει ο χρήστης.

```
def calculateStates(stacks, blocks):
    '''
    Function which calculates the starting and final states of the problem,
    given the stacks and blocks which are retrieved from the users input.
    '''
    # CALCULATE THE STARTING STATE
    stacks_nr = stacks
    digits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
    block_nrs = digits[:blocks]     # if blocks = 5, blocks_nrs = ['0', '1', '2', '3', '4']
    random.shuffle(block_nrs)

    startState = []
    while blocks:
        if stacks == 1:
            startState.append(block_nrs)
            break
        else:
            r = random.randint(1, blocks)
            s = block_nrs[:r]
            startState.append(s)

        blocks -= r
        stacks -= 1
        block_nrs = block_nrs[r:]

    while len(startState) < stacks_nr:
        startState += [[]]

    # randomize the stacks positions
    random.shuffle(startState)

    # CALCULATE THE FINAL STATE
    finalState = []
    for stack in startState:
        finalState += stack

    finalState.sort()
    finalState = [finalState]

    # complete the final states stacks with empty stacks
    for _ in range(len(startState) - 1):
        finalState += [[]]

    return startState, finalState
```

Ο κώδικας που αναλύθηκε παραπάνω, για την επίλυση των προβλημάτων Water Jug, Blocks World βρίσκεται στο αρχείο generic_planner.py, το οποίο μπορεί να εκτελεστεί σε οποιονδήποτε python editor.

Εκτελώντας το αρχείο, μέσω της συνάρτησης print_menu() δίνεται η δυνατότητα στο χρήστη να επιλέξει ποιο από τα δύο προβλήματα θέλει να επιλύσει, εισάγοντας στη συνέχεια τις μεταβλητές που θα του ζητηθούν.

```python
def print_menu():
    print("\nCHOOSE ONE OF THE BELOW PROBLEMS TO SOLVE: ")
    print("1. WATER JUG")
    print("2. BLOCKS WORLD")

while True:
    print_menu()
    choice = int(input("ENTER YOUR CHOICE [1-2]: "))

    if choice == 1:
        waterJugs()
    elif choice == 2:
        blocksWorld()
```

Παρακάτω παρατίθονται κάποια παραδείγματα εκτέλεσης του προβλήματος Blocks World.

```
CHOOSE ONE OF THE BELOW PROBLEMS TO SOLVE:
1. WATER JUG
2. BLOCKS WORLD
ENTER YOUR CHOICE [1-2]: 2
-----ENTER NUMBER OF STACKS: 5
-----ENTER NUMBER OF BLOCKS: 4

-----START STATE: [['0', '2', '1'], [], [], ['3'], []]
-----FINAL STATE: [['0', '1', '2', '3'], [], [], [], []]

-----SOLUTION FOUND, PRINTING THE PATH FOLLOWED TO THE GOAL STATE:
0. [['0', '2', '1'], [], [], ['3'], []]
1. [['0', '2'], [], [], ['3', '1'], []]
2. [['0'], ['2'], [], ['3', '1'], []]
3. [['0', '1'], ['2'], [], ['3'], []]
4. [['0', '1', '2'], [], [], ['3'], []]
5. [['0', '1', '2', '3'], [], [], [], []]
```

```
CHOOSE ONE OF THE BELOW PROBLEMS TO SOLVE:
1. WATER JUG
2. BLOCKS WORLD
ENTER YOUR CHOICE [1-2]: 2
-----ENTER NUMBER OF STACKS: 6
-----ENTER NUMBER OF BLOCKS: 9

-----START STATE: [['4', '5'], [], [], ['7', '1', '2', '8', '3', '0'], [], ['6']]
-----FINAL STATE: [['0', '1', '2', '3', '4', '5', '6', '7', '8'], [], [], [], [], []]

-----SOLUTION FOUND, PRINTING THE PATH FOLLOWED TO THE GOAL STATE:
0. [['4', '5'], [], [], ['7', '1', '2', '8', '3', '0'], [], ['6']]
1. [['4'], [], [], ['7', '1', '2', '8', '3', '0'], [], ['6', '5']]
2. [[], [], [], ['7', '1', '2', '8', '3', '0'], [], ['6', '5', '4']]
3. [['0'], [], [], ['7', '1', '2', '8', '3'], [], ['6', '5', '4']]
4. [['0'], [], [], ['7', '1', '2', '8'], [], ['6', '5', '4', '3']]
5. [['0'], ['8'], [], ['7', '1', '2'], [], ['6', '5', '4', '3']]
6. [['0'], ['8'], [], ['7', '1'], [], ['6', '5', '4', '3', '2']]
7. [['0', '1'], ['8'], [], ['7'], [], ['6', '5', '4', '3', '2']]
8. [['0', '1'], ['8', '7'], [], [], [], ['6', '5', '4', '3', '2']]
9. [['0', '1', '2'], ['8', '7'], [], [], [], ['6', '5', '4', '3']]
10. [['0', '1', '2', '3'], ['8', '7'], [], [], [], ['6', '5', '4']]
11. [['0', '1', '2', '3', '4'], ['8', '7'], [], [], [], ['6', '5']]
12. [['0', '1', '2', '3', '4', '5'], ['8', '7'], [], [], [], ['6']]
13. [['0', '1', '2', '3', '4', '5', '6'], ['8', '7'], [], [], [], []]
14. [['0', '1', '2', '3', '4', '5', '6', '7'], ['8'], [], [], [], []]
15. [['0', '1', '2', '3', '4', '5', '6', '7', '8'], [], [], [], [], []]
```

```
ENTER YOUR CHOICE [1-2]: 2
-----ENTER NUMBER OF STACKS: 3
-----ENTER NUMBER OF BLOCKS: 8

-----START STATE: [['0', '7', '3', '2', '6', '5', '1'], ['4'], []]
-----FINAL STATE: [['0', '1', '2', '3', '4', '5', '6', '7'], [], []]

-----SOLUTION FOUND, PRINTING THE PATH FOLLOWED TO THE GOAL STATE:
0. [['0', '7', '3', '2', '6', '5', '1'], ['4'], []]
1. [['0', '7', '3', '2', '6', '5'], ['4', '1'], []]
2. [['0', '7', '3', '2', '6'], ['4', '1', '5'], []]
3. [['0', '7', '3', '2'], ['4', '1', '5', '6'], []]
4. [['0', '7', '3'], ['4', '1', '5', '6', '2'], []]
5. [['0', '7'], ['4', '1', '5', '6', '2', '3'], []]
6. [['0'], ['4', '1', '5', '6', '2', '3'], ['7']]
7. [['0'], ['4', '1', '5', '6', '2'], ['7', '3']]
8. [['0'], ['4', '1', '5', '6'], ['7', '3', '2']]
9. [['0'], ['4', '1', '5'], ['7', '3', '2', '6']]
10. [['0'], ['4', '1'], ['7', '3', '2', '6', '5']]
11. [['0', '1'], ['4'], ['7', '3', '2', '6', '5']]
12. [['0', '1'], ['4', '5'], ['7', '3', '2', '6']]
13. [['0', '1'], ['4', '5', '6'], ['7', '3', '2']]
14. [['0', '1', '2'], ['4', '5', '6'], ['7', '3']]
15. [['0', '1', '2', '3'], ['4', '5', '6'], ['7']]
16. [['0', '1', '2', '3'], ['4', '5'], ['7', '6']]
17. [['0', '1', '2', '3'], ['4'], ['7', '6', '5']]
18. [['0', '1', '2', '3', '4'], [], ['7', '6', '5']]
19. [['0', '1', '2', '3', '4', '5'], [], ['7', '6']]
20. [['0', '1', '2', '3', '4', '5', '6'], [], ['7']]
21. [['0', '1', '2', '3', '4', '5', '6', '7'], [], []]
```