Kralici

```
from collections import deque

for n in range(8):
    print(n)
```

```
0
1
2
3
4
5
6
7
```

```
def is_valid(state, i, j):
    #za da e validna ne smee da izlezi od granicite na tablata
    if 0 > i > 7 or 0 > j > 7:
        return False
    #proverka dali na poleto kade sto sme vekje ima kralica
    if state[i][j] == 1:
        return False
    #sledna proverka za validnost ni e toa sto ne smeat da se
napagjaat, t.e da se vo ist red i kolona,
    for n in range(8):
        #1 oznacuva deka imame kralica na toa pole
        if state[i][n] == 1 or state[n][j] == 1:
            return False
    #proverka za dijagonalno preklopuvanje
    # prvo ni treba za glavna dijagonala razlikata da e ista a za
sporedna zbirot
    temp_zbir = i + j
    temp_razlika = i - j
    for n,row in enumerate(state):
        for m,column in enumerate(row):
            if state[n][m] == 1:
                new_zbir = n + m
                new_razlika = n - m
                # proveruvame ako zbirot im e ist togas ima dve na
sporedna dijagonala
                if new_zbir == temp_zbir or new_razlika==temp_razlika:
                    return False

    return True

def end_check(state):
    state_matrix = [list(row) for row in state]
```

```
    c = 0
    for i, row in enumerate(state_matrix):
        for j, column in enumerate(row):
            if state_matrix[i][j] == 1:
                c+=1
    if c == 8:
        return True

def expand_state(state):
    states = []
    state_matrix = [list(row) for row in state]
    # barame na koj koordinati ima slobodno mesto za da ja smestime
kralicata

    for i in range(8):
        for j in range(8):
            if is_valid(state_matrix,i,j):
                state_matrix = [list(row) for row in state]
                state_matrix[i][j] = 1
                states.append(tuple(map(tuple,state_matrix)))
    return states

def search_path(initial_state, alg):
    visited = {initial_state}
    states_queue = deque([initial_state])
    while states_queue:
        state_to_expand = states_queue.popleft()
        for next_state in expand_state(state_to_expand):
            if next_state not in visited:
                #posle ova obicno pravime proverka dali next_state e
ista so goal state ama oti tuka nemame goal state
                #a celta ni e vo listate da gi popolnime site kralici
t.e da nema None zato ako brojot na none e 0 vrati ja next state
                if end_check(next_state):
                    return next_state
                visited.add(next_state)
                if alg == 'dfs':
                    states_queue.appendleft(next_state)
                elif alg == 'bfs':
                    states_queue.append(next_state)


start_state = ((0,0,0,0,0,0,0,0),(0,0,0,0,0,0,0,0),(0,0,0,0,0,0,0,0),
(0,0,0,0,0,0,0,0),(0,0,0,0,0,0,0,0),(0,0,0,0,0,0,0,0),
(0,0,0,0,0,0,0,0),(0,0,0,0,0,0,0,0))
search_path(start_state, alg='dfs')
```

```
((0, 0, 0, 0, 0, 1, 0, 0),
 (0, 0, 0, 1, 0, 0, 0, 0),
 (0, 1, 0, 0, 0, 0, 0, 0),
```

```
 (0, 0, 0, 0, 0, 0, 0, 1),
 (0, 0, 0, 0, 1, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 1, 0),
 (1, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 1, 0, 0, 0, 0, 0))
```

sudoku

```
sudoku = [1,2,3,4,5,6,7,8,9]

def is_valid_s(s,state,x,y): #s ke mi bidat vrednostite vo listata
sudoku
    if 0 > x > 8 or 0 > y > 8:
        return False
    #treba da proverime dali se vo ist red ili ista kolona
    for i in range(9):
        if state[x][i] == s or state[i][y] == s:
            return False
    #proverka za dali se vo isto kvadratce
    if is_number_in_square(state,x,y,s):
        return False

    return True


def find_zero_s(state):
    for i,row in enumerate(state):
        for j,column in enumerate(state):
            if state[i][j] == 0:
                return i,j

def is_number_in_square(state, row, col, s):
    # Calculate the starting row and column of the 3x3 square
    #ovde prajme plostina na kvadrat 3*3
    # gi deli na redovi i koloni 3 na 3
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)

    # Iterate through the square
    for i in range(start_row, start_row + 3):
        for j in range(start_col, start_col + 3):
            # Check if the number already exists in the square
            if state[i][j] == s:
                return True

def expand_state_s(state):
    states = []
    state_matrix = [list(row) for row in state]
    x,y = find_zero_s(state_matrix)
```

```
    for i in sudoku:
        if is_valid_s(i,state_matrix,x,y):
            state_matrix = [list(row) for row in state]
            state_matrix[x][y] = i
            states.append(tuple(map(tuple,state_matrix)))
    return states

def end_check_sudoku(state):
    matrix_state = [list(row) for row in state]
    c = 0
    for i,row in enumerate(matrix_state):
        for The sum of these conditions gives an estimate of how many
entities are not in the correct position relative to the farmer. Since
the farmer can only take one entity at a time in the boat, this
heuristic provides a reasonable estimate of the number of moves needed
to correct the positions of the entities.j,column in enumerate(row):
            if matrix_state[i][j] == 0:
                c+=1
    if c == 0:
        return True
    else:
        return False

def search_sudoku(initial_state, alg):
    visited = {initial_state}
    states_queue = deque([initial_state])
    while states_queue:
        state_to_expand = states_queue.popleft()
        for next_state in expand_state_s(state_to_expand):
            if next_state not in visited:
                if end_check_sudoku(next_state):
                    return next_state
                visited.add(next_state)
                if alg == 'dfs':
                    states_queue.appendleft(next_state)
                elif alg == 'bfs':
                    states_queue.append(next_state)

g = (
    (5, 3, 0, 0, 7, 0, 0, 0, 0),
    (6, 0, 0, 1, 9, 5, 0, 0, 0),
    (0, 9, 8, 0, 0, 0, 0, 6, 0),
    (8, 0, 0, 0, 6, 0, 0, 0, 3),
    (4, 0, 0, 8, 0, 3, 0, 0, 1),
    (7, 0, 0, 0, 2, 0, 0, 0, 6),
    (0, 6, 0, 0, 0, 0, 2, 8, 0),
    (0, 0, 0, 4, 1, 9, 0, 0, 5),
    (0, 0, 0, 0, 8, 0, 0, 7, 9),
)
```

```
search_sudoku(g,alg = 'dfs')
```

```
((5, 3, 4, 6, 7, 8, 9, 1, 2),
 (6, 7, 2, 1, 9, 5, 3, 4, 8),
 (1, 9, 8, 3, 4, 2, 5, 6, 7),
 (8, 5, 9, 7, 6, 1, 4, 2, 3),
 (4, 2, 6, 8, 5, 3, 7, 9, 1),
 (7, 1, 3, 9, 2, 4, 8, 5, 6),
 (9, 6, 1, 5, 3, 7, 2, 8, 4),
 (2, 8, 7, 4, 1, 9, 6, 3, 5),
 (3, 4, 5, 2, 8, 6, 1, 7, 9))
```

Истражувач

Предложете соодветна репрезентација и напишете ги потребните функции во Python за да се реши следниот проблем за кој една можна почетна состојба е прикажана на сликата Потребно е човечето безбедно да дојде до куќичката. Човечето може да се придвижува на кое било соседно поле хоризонтално или вертикално. Пречките 1 и 2 се подвижни, при што и двете пречки се движат вертикално. Секоја од пречките се придвижува за едно поле во соодветниот правец и насока со секое придвижување на човечето. Притоа, пречката 1 на почетокот се движи надолу, додека пречката 2 на почетокот се движи нагоре. Пример за положбата на пречките после едно придвижување на човечето надесно е прикажан на десната слика. Кога некоја пречка ќе дојде до крајот на таблата при што повеќе не може да се движи во насоката во која се движела, го менува движењето во спротивната насока. Доколку човечето и која било од пречките се најдат на исто поле човечето ќе биде уништено.

За сите тест примери изгледот и големината на таблата се исти како на примерот даден на сликите. За сите тест примери почетните положби, правец и насока на движење за препреките се исти. За секој тест пример почетната позиција на човечето се менува, а исто така се менува и позицијата на куќичката.

Во рамки на почетниот код даден за задачата се вчитуваат влезните аргументи за секој тест пример.

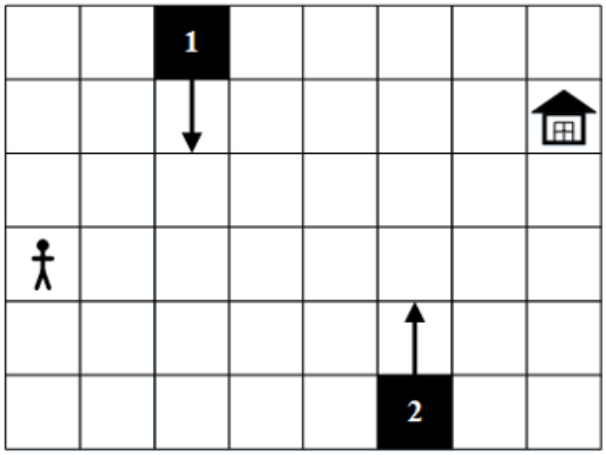
Движењата на човечето потребно е да ги именувате на следниот начин:

Right - за придвижување на човечето за едно поле надесно Left - за придвижување на човечето за едно поле налево Up - за придвижување на човечето за едно поле нагоре Down - за придвижување на човечето за едно поле надолу

Вашиот код треба да има само еден повик на функција за приказ на стандарден излез (print) со кој ќе ја вратите секвенцата на движења која човечето треба да ја направи за да може од својата почетна позиција да стигне до позицијата на куќичката.

Треба да примените неинформирано пребарување. Врз основа на тест примерите треба самите да определите кое пребарување ќе го користите.

```
from IPython import display
display.Image('explorer1.png', width=300)
```




```
def limits(i,j):
    if 0 <= i <= 5 and 0 <= j <= 7:
        return True

def find_obstacle_1(matrix_state):
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if matrix_state[i][j] == 2:
                return(i,j)
    return(0,0)

def find_obstacle_2(matrix_state):
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if column == 3:
                return (i,j)
    return (0,0)

def find_person(matrix_state):
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if column == 1:
                return (i,j)
    return (0,0)

def same_position(i_p,j_p,state):
    i_o1, j_o1 = find_obstacle_1(state)
    i_o2, j_o2 = find_obstacle_2(state)

    if (i_p == i_o1 and j_p == j_o1) or (i_p == i_o2 and j_p == j_o2):
        return False
    else:
        return True
```

```
def end_check(state):
    state_matrix = [list(row) for row in state]
    i,j = find_person(state_matrix)
    if i == 1 and j == 7:
        return True

def expand_state(state):
    states = []
    new_state_matrix = [list(row) for row in state]

    moves_p = [(0,-1),(-1,0),(0,1),(1,0)]
    moves_o1 = [(1,0),(1,0),(1,0),(1,0)]
    moves_o2 = [(-1,0),(-1,0),(-1,0),(-1,0)]

    i_p,j_p = find_person(new_state_matrix)
    i_o1, j_o1 = find_obstacle_1(new_state_matrix)
    i_o2, j_o2 = find_obstacle_2(new_state_matrix)

    for (d_ip,d_jp),(d_io1,d_jo1),(d_io2,d_jo2) in
zip(moves_p,moves_o1,moves_o2):
        new_state_matrix = [list(row) for row in state]
        new_ip = i_p + d_ip
        new_jp = j_p + d_jp

        new_io1 = i_o1 + d_io1
        new_jo1 = j_o1 + d_jo1

        new_io2 = i_o2 + d_io2
        new_jo2 = j_o2 + d_jo2

        if limits(new_ip,new_jp) and
same_position(new_ip,new_jp,new_state_matrix):
            new_state_matrix[i_p][j_p] = 0
            new_state_matrix[new_ip][new_jp] = 1
        if limits(new_io1,new_jo1):
            new_state_matrix[i_o1][j_o1] = 0
            new_state_matrix[new_io1][new_jo1] = 2
        if limits(new_io2,new_jo2):
            new_state_matrix[i_o2][j_o2] = 0
            new_state_matrix[new_io2][new_jo2] = 3


        states.append(tuple(map(tuple,new_state_matrix)))

    return states

def search_path(initial_state):
    visited = {initial_state}
    states_queue = deque([[initial_state]])
    while states_queue:
```

```
        states_list = states_queue.popleft()
        state_to_expand = states_list[-1]
        for next_state in expand_state(state_to_expand):
            if next_state not in visited:
                if end_check(next_state):
                    return states_list + [next_state]
                visited.add(next_state)
                states_queue.append(states_list + [next_state])
    return []

def visualise_path(path):
    for states in zip(path, path[1:]):
        old_state, new_state = states
        for row in new_state:
            print(' '.join(map(str, row)))

        print()

def a_star_search(starting_vertex, heuristic_function, alpha=1):
    expanded = set()
    queue = [((0, 0), [starting_vertex])]
    heapq.heapify(queue)
    c = 0
    while queue:
        c += 1
        weight_tuple, vertex_list = heapq.heappop(queue)
        current_a_star_weight, current_path_weight = weight_tuple
        vertex_to_expand = vertex_list[-1]
        if end_check(vertex_to_expand):
            return current_path_weight, vertex_list, c
        if vertex_to_expand in expanded:
            continue
        for neighbour in expand_state(vertex_to_expand):
            if neighbour not in expanded:
                heuristic = heuristic_function(neighbour)
                path_weight = current_path_weight + 1
                a_star_weight = path_weight + alpha * heuristic
                heapq.heappush(queue, ((a_star_weight, path_weight),
vertex_list + [neighbour]))
        expanded.add(vertex_to_expand)

def manhattan_distance(state):
    matrix_explorer = [list(row) for row in state]
    i_p, j_p = find_person(matrix_explorer)
    return abs(i_p - 1) + abs(j_p - 7)

def h(state):
    distance = manhattan_distance(state)
    return distance
```

Преку Реката

Предложете соодветна репрезентација и напишете ги потребните функции во Python за да се реши следниот проблем за кој почетната состојба е прикажана на сликата. Нотребно е да се пренесат зелката, јарето, волкот и фармерот од источната страна на западната страна на реката. Само фармерот го вози чамецот. Во чамецот има простор за двајца патници: фармерот и уште еден патник.

Ограничувања: Доколку останат сами (без присуство на фармерот):

Јарето ја јаде зелката Волкот го јаде јарето

Вашиот код треба да има само еден повик на функција за приказ на стандарден излез (print) со кој ќе ја вратите секвенцата од позиции на актерите која одговара на секвенцата на движења со која сите актери ќе бидат пренесени на западната страна на реката.

Треба да примените информирано пребарување. Дефинирајте соодветна хевристика која ќе биде прифатлива за проблемот.

```
def is_valid(state): #farmer, volk, jare, zelka
    farmer, wolf, goat, cabbage, boat = state
    # ako volkot e vo ista pozicija so jagneto i ako faremrot ne e
tamu da ni vrati deka s-jbata e nevalidna
    if wolf == goat and farmer != wolf:
        return False
    if goat == cabbage and farmer != goat:
        return False
    return True

def expand_state(state):
    # 0 1 2 3 4 farmer, volk, jagne, zelka brod
    states = []
    #sega vrvime niz site mozni dvizenja
    for i in range(len(state)):
        #za vooopsto da mozi da prenesime nekoj preku rekata mora toj
da e vo ista pozicija so camecot!!!
        if state[4] == state[i]:
            new_state = list(state)
            # 1- ednata strana na bregot 0- za dr str na bregot
            # novata sostojba na farmerot ja promenuvame na bregot
kade sto se premestil so camecot
            #istoto go pravime i so sostojbata na camecot
            new_state[0] = 1 - state[0]
            new_state[4] = 1 - state[4]
            # pa sostojbata na i-tiot element koj ke se prenesuvsa
preku rekata ke e ista so sostojbata na camecot
            new_state[i] = new_state[4]
            new_state = tuple(new_state)
            #otkako izvrsivme premestuvanje na dr strana treba da
proverime dali e toa validno
```

```
            if is_valid(new_state):
                states.append(new_state)

    return states


def search_path(initial_state, goal_state):
    visited = {initial_state}
    states_queue = deque([[initial_state]])
    while states_queue:
        states_list = states_queue.popleft()
        state_to_expand = states_list[-1]
        for next_state in expand_state(state_to_expand):
            if next_state not in visited:
                if next_state == goal_state:
                    return states_list + [next_state]
                visited.add(next_state)
                states_queue.append(states_list + [next_state])
    return []
```

def h(state): # Heuristic - number of mismatched pairs return sum([state[0] != state[1], state[0] != state[2], state[0] != state[3]]) #The heuristic considers how many entities are on the opposite side of the river compared to the farmer. #The sum of these conditions gives an estimate of how many entities are not in the correct position relative to the farmer. Since the farmer can only take one entity at a time in the boat, #this heuristic provides a reasonable estimate of the number of moves needed to correct the positions of the entities.

```
def a_star_search(starting_vertex, goal_vertex, heuristic_function,
alpha=1):
    expanded = set()
    queue = [((0, 0), [starting_vertex])]
    heapq.heapify(queue)
    c = 0
    while queue:
        c += 1
        weight_tuple, vertex_list = heapq.heappop(queue)
        current_a_star_weight, current_path_weight = weight_tuple
        vertex_to_expand = vertex_list[-1]
        if goal_vertex == vertex_to_expand:
            return current_path_weight, vertex_list, c
        if vertex_to_expand in expanded:
            continue
        for neighbour in expand_state(vertex_to_expand):
            if neighbour not in expanded:
                heuristic = heuristic_function(neighbour)
                path_weight = current_path_weight + 1
                a_star_weight = path_weight + alpha * heuristic
                heapq.heappush(queue, ((a_star_weight, path_weight),
```

```
vertex_list + [neighbour]))
        expanded.add(vertex_to_expand)
```

Koli

На квадратна табла со димензии5 5 , слика 2.16, во првата колона се поставени5 колички BURAGO. Количките треба да се преместат во последната колона, но во обратен редослед. Тоа значи дека количката i што се наоѓа на почетна позиција( ),1i мора да се најде на крајната позиција( )5 1 ,5i− − . При секој чекор на поместување, секоја од количките може да се помести за едно поле во лево, десно, горе, долу или да остане на истата позиција. Доколку во тековниот чекор некоја количка не се помести од својата тековна позиција, една од соседните колички (но само една) може да ја прескокне. Две колички не можат едновремено да се најдат на исто поле.

```
def limits(x,y):
    if 0 <= x <= 3 and 0 <= y <= 3:
        return True
    else:
        return False

def is_valid(x,y,state):
    if limits(x,y):
        if state[x][y] == 0:
            return True

def find_car_1(state):
    matrix_state = [list(row) for row in state]
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if matrix_state[i][j] == 1:
                return i,j
    return (0,0)

def find_car_2(state):
    matrix_state = [list(row) for row in state]
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if matrix_state[i][j] == 2:
                return i,j
    return (0,0)


def find_car_3(state):
    matrix_state = [list(row) for row in state]
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if matrix_state[i][j] == 3:
```

```
                return i,j
    return (0,0)

def find_car_4(state):
    matrix_state = [list(row) for row in state]
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if matrix_state[i][j] == 4:
                return i,j
    return (0,0)

def expand_state(state):
    states = []
    x,y = find_car_1(state)
    w,z = find_car_2(state)
    m,n = find_car_3(state)
    p,q = find_car_4(state)
    moves_1 = [(1,0),(0,1),(-1,0),(0,-1),(0,0)]
    moves_2 = [(0,1),(0,-1),(0,0),(1,0),(-1,0)]
    moves_3 = [(0,-1),(-1,0),(1,0),(0,0),(0,1)]
    moves_4 = [(0,0),(1,0),(0,1),(-1,0),(0,-1)]
    for (dx,dy),(dw,dz),(dm,dn),(dp,dq) in
zip(moves_1,moves_2,moves_3,moves_4):
        matrix_state = [list(row) for row in state]
        new_x = x + dx
        new_y = y + dy
        new_w = w + dw
        new_z = z + dz
        new_m = m + dm
        new_n = n + dn
        new_p = p + dp
        new_q = q + dq

        if is_valid(new_x,new_y,matrix_state):
            matrix_state = [list(row) for row in state]
            matrix_state[x][y] = 0
            matrix_state[new_x][new_y] = 1
            states.append(tuple(map(tuple,matrix_state)))
        if is_valid(new_w,new_z,matrix_state):
            matrix_state = [list(row) for row in state]
            matrix_state[w][z] = 0
            matrix_state[new_w][new_z] = 2
            states.append(tuple(map(tuple,matrix_state)))
        if is_valid(new_m,new_n,matrix_state):
            matrix_state = [list(row) for row in state]
            matrix_state[m][n] = 0
            matrix_state[new_m][new_n] = 3
            states.append(tuple(map(tuple,matrix_state)))
        if is_valid(new_p,new_q,matrix_state):
            matrix_state = [list(row) for row in state]
```

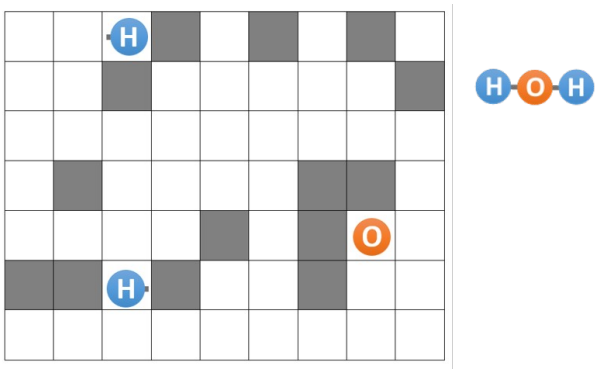
```
            matrix_state[p][q] = 0
            matrix_state[new_p][new_q] = 4
            states.append(tuple(map(tuple,matrix_state)))
    return states

def search_path(initial_state,goal_state):
    visited = {initial_state}
    states_queue = deque([[initial_state]])
    while states_queue:
        states_list = states_queue.popleft()
        state_to_expand = states_list[-1]
        for next_state in expand_state(state_to_expand):
            if next_state not in visited:
                if next_state == goal_state:
                    return states_list + [next_state]
                visited.add(next_state)
                states_queue.append(states_list + [next_state])
    return states_list + [next_state]
```

Молекули

Предложете соодветна репрезентација и напишете ги потребните функции во Python за да се реши следниот проблем за кој една можна почетна состојба е прикажана на сликата на следниот слајд. На табла 7x9 поставени се три атоми (внимавајте, двата H-атоми се различни: едниот има линк во десно, а другиот има линк во лево). Полињата обоени во сива боја претставуваат препреки. Играчот може да ја започне играта со избирање на кој било од трите атоми. Играчот во секој момент произволно избира точно еден од трите атоми и го „турнува“ тој атом во една од четирите насоки: горе, долу, лево или десно. Движењето на „турнатиот“ атом продолжува во избраната насока се' додека атомот не „удри“ во препрека или во некој друг атом (атомот секогаш застанува на првото поле што е соседно на препрека или на друг атом во соодветната насока). Не е возможно ротирање на атомите (линковите на атомите секогаш ќе бидат поставени како што се на почетокот на играта). Исто така, не е дозволено атомите да излегуваат од таблата. Целта на играта е атомите да се доведат во позиција во која ја формираат „молекулата“ прикажана десно од таблата. Играта завршува во моментот кога трите атоми ќе бидат поставени во бараната позиција, во произволни три соседни полиња од таблата. Потребно е проблемот да се реши во најмал број на потези. За сите тест примери изгледот и големината на таблата се исти како на примерот даден на сликата. За сите тест примери положбите на препреките се исти. За секој тест пример се менуваат почетните позиции на сите три атоми, соодветно. Во рамки на почетниот код даден за задачата се вчитуваат влезните аргументи за секој тест пример. Движењата на атомите потребно е да ги именувате на следниот начин: RightX - за придвижување на атомот X надесно (X може да биде H1, O или H2) LeftX - за придвижување на атомот X налево (X може да биде H1, O или H2) UpX - за придвижување на атомот X нагоре (X може да биде H1, O или H2) DownX - за придвижување на атомот X надолу (X може да биде H1, O или H2) Вашиот код треба да има само еден повик на функција за приказ на стандарден излез (print) со кој ќе ја вратите секвенцата на движења која треба да се направи за да може атомите од почетната позиција да се доведат до бараната позиција.

Треба да примените неинформирано пребарување. Врз основа на тест примерите треба самите да определите кое пребарување ќе го користите.

```
from IPython import display
display.Image('molecule1.png', width=300)
```



```
def table_limits(x, y):
    if 0 <= x <= 6 and 0 <= y <= 8:
        return True

def obstacles(x, y):
    obstacle_positions = {(0, 5), (1, 3), (1, 5), (2, 1), (3, 0), (3,
5), (4, 4), (5, 0), (6, 3), (6, 4), (6, 5), (7, 0), (7, 3), (8, 1)}
    return (y, x) not in obstacle_positions #vrakja tocno ako x,y ne e
elem od m-vo obstacle_possitions

def find_pos_H1(matrix_state):
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if matrix_state[i][j] == 1:
                return (i,j)
    raise Exception("Go izgubivme atomot")

def find_pos_H2(matrix_state):
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if column == 2:
                return (i,j)
    return (0,0)

def find_pos_O(matrix_state):
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if column == 3:
                return (i,j)

def mismatch(x1,y1,x2,y2):
    if x1 == x2 and y1 == y2:
        return False
```

```
    else:
        return True

def is_valid(x1,y1,x2,y2):
    if table_limits(x1,y1) and obstacles(x1,y1) and
mismatch(x1,y1,x2,y2):
        return True
    else:
        return False

def end_check(new_state):

    xo,yo = find_pos_O(new_state)
    xh_1,yh_1 = find_pos_H1(new_state)
    xh_2,yh_2 = find_pos_H2(new_state)
    if yh_1 == yo + 1 and xh_1 == xo and yh_2 == yo - 1 and xh_2 ==
xo:
        return True

def expand_check(state):
    states = []
    new_state_matrix = [list(row) for row in state]

    xo,yo = find_pos_O(new_state_matrix)
    xh_1,yh_1 = find_pos_H1(new_state_matrix)
    xh_2,yh_2 = find_pos_H2(new_state_matrix)

    moves = [(0,-1),(-1,0),(0,1),(1,0)]

    for (dxo,dyo),(dxh_1,dyh_1),(dxh_2,dyh_2) in
zip(moves,moves,moves):
        new_state_matrix = [list(row) for row in state]
        new_xo = xo + dxo
        new_yo = yo + dyo
        new_xh_1 = xh_1 + dxh_1
        new_yh_1 = yh_1 + dyh_1
        new_xh_2 = xh_2 + dxh_2
        new_yh_2 = yh_2 + dyh_2

        if is_valid(new_xo,new_yo,new_xh_1,new_yh_1) and
is_valid(new_xo,new_yo,new_xh_2,new_yh_2):
            new_state_matrix[xo][yo] = 0 # predhodnoto klajgo na nula
            new_state_matrix[new_xo][new_yo] = 3
        if is_valid(new_xh_1,new_yh_1,new_xo,new_yo) and
is_valid(new_xh_1,new_yh_1,new_xh_2,new_yh_2):
            new_state_matrix[xh_1][yh_1] = 0 # predhodnoto klajgo na
nula
            new_state_matrix[new_xh_1][new_yh_1] = 1
        if is_valid(new_xh_2,new_yh_2,new_xh_1,new_yh_1) and
is_valid(new_xh_2,new_yh_2,new_xo,new_yo):
```

```
            new_state_matrix[xh_2][yh_2] = 0 # predhodnoto klajgo na
nula
            new_state_matrix[new_xh_2][new_yh_2] = 2

        states.append(tuple(map(tuple,new_state_matrix)))


    return states

def search_path(initial_state):
    visited = {initial_state}
    states_queue = deque([[initial_state]])
    while states_queue:
        states_list = states_queue.popleft()
        state_to_expand = states_list[-1]
        for next_state in expand_check(state_to_expand):
            if next_state not in visited:
                if end_check(next_state):
                    return states_list + [next_state]
                visited.add(next_state)
                states_queue.append(states_list + [next_state])
    return []

def visualise_path(path):
    for states in zip(path, path[1:]):
        old_state, new_state = states
        for row in new_state:
            print(' '.join(map(str, row)))

        print()
```

Свезди

Предложете соодветна репрезентација и напишете ги потребните функции во Python за да се реши следниот проблем за кој една можна почетна состојба е прикажана на сликата. На шаховска табла 8x8 поставени се еден коњ, еден ловец и три свезди. Движењето на коњите на шаховската табла е во облик на буквата Г: притоа, од дадена позиција можни се 8 позиции до кои даден коњ може да се придвижи, како што е прикажано на сликата (1 = горе + горе + лево, 2 = горе + горе + десно, 3 = десно + десно + горе, 4 = десно + десно + долу, 5 = долу + + долу + десно, 6 = долу + долу + лево, 7 = лево + лево + долу, 8 = лево + лево + горе) Движењето на ловците на таблата е по дијагонала. Ловецот прикажан на сликата може да се придвижи на кое било од полињата означени со X. Целта на играта е да се соберат сите три свезди. Една свезда се собира доколку некоја од фигурите застане на истото поле каде што се наоѓа и свездата.

Притоа, не е дозволено двете фигури да бидат позиционирани на истото поле и не е дозволено фигурите да излегуваат од таблата. Фигурите меѓусебно не се напаѓаат.

Движењето на фигурите е произволно, т.е. во кој било момент може да се придвижи која било од двете фигури. Потребно е проблемот да се реши во најмал број на потези.

За сите тест примери изгледот и големината на таблата се исти како на примерот даден на сликата. За секој тест пример положбите на ѕвездите се различни. Исто така, за секој тест пример се менуваат и почетните позиции на коњот и ловецот, соодветно. Во рамки на почетниот код даден за задачата се вчитуваат влезните аргументи за секој тест пример.

Движењата на коњот потребно е да ги именувате на следниот начин:

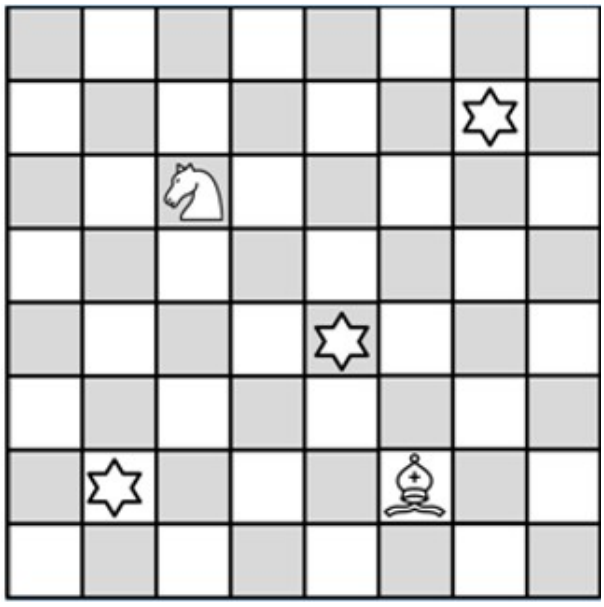
K1 - за придвижување од тип 1 (горе + лево) K2 - за придвижување од тип 2 (горе + десно) K3 - за придвижување од тип 3 (десно + горе) K4 - за придвижување од тип 4 (десно + долу) K5 - за придвижување од тип 5 (долу + десно) K6 - за придвижување од тип 6 (долу + лево) K7 - за придвижување од тип 7 (лево + долу) K8 - за придвижување од тип 8 (лево + горе)

Движењата на ловецот потребно е да ги именувате на следниот начин:

B1 - за придвижување од тип 1 (движење за едно поле во насока горе-лево) B2 - за придвижување од тип 2 (движење за едно поле во насока горе-десно) B3 - за придвижување од тип 3 (движење за едно поле во насока долу-лево) B4 - за придвижување од тип 4 (движење за едно поле во насока долу-десно)

Вашиот код треба да има само еден повик на функција за приказ на стандарден излез (print) со кој ќе ја вратите секвенцата на движења која треба да се направи за да може фигурите да ги соберат сите три ѕвезди. Треба да примените неинформирано пребарување. Врз основа на тест примерите треба самите да определите кое пребарување ќе го користите.

```
from IPython import display
display.Image('stars1.png', width=300)
```



```
def end_check(state):
    table_check_matrix = [list(row) for row in state]
```

```
    count_stars = sum(row.count(3) for row in table_check_matrix)
#ovde broj kolku trojki ima na matricata ako nema trojki togas ja
resivme zad
    #print(count_1)
    if count_stars == 0:
        return True
    else:
        return False

def limits_of_table(state):
    x,y = state
    if 0 <= x and x <= 7 and 0 <= y and y <= 7:
        return True
    else:
        return False


def same_position(state_knight, state_bishop):
    x,y = state_knight
    w,z = state_bishop

    if x == w and y == z:
        return False
    #else:
     #   return True

def find_pos_knight(matrix_state):
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if matrix_state[i][j] == 1:
                return (i,j)
    return(0,0)

def find_pos_bishop(matrix_state):
    for i, row in enumerate(matrix_state):
        for j, column in enumerate(row):
            if matrix_state[i][j] == 2:
                return (i,j)
    return (0,0)

def expand_state(state):
    states = []
    new_state_matrix = [list(row) for row in state]

    possible_moves_bishop = [(1, 1),(1, -1),(-1, 1),(-1, -1),(1, 1),
(1, -1),(-1, 1),(-1, -1)]
    possible_moves_knight = [(1,2), (2,1), (-1,-2), (-2,-1), (-1,2),
(2,-1), (-2,1), (1,-2)]

    x,y = find_pos_knight(new_state_matrix)
    w,z = find_pos_bishop(new_state_matrix)
```

```
    for (dx,dy),(dw,dz) in zip(possible_moves_knight,
possible_moves_bishop):
        new_state_matrix = [list(row) for row in state]
        new_x = x + dx
        new_y = y + dy
        new_w = w + dw
        new_z = z + dz
        #print(new_x,new_y,new_w,new_z)
        if same_position((new_x,new_y),(new_w,new_z)):
            if limits_of_table((new_x,new_y)):
                new_state_matrix[x][y] = 0
                new_state_matrix[new_x][new_y] = 1
            if limits_of_table((new_w,new_z)):
                new_state_matrix[w][z] = 0
                new_state_matrix[new_w][new_z] = 2
            states.append(tuple(map(tuple,new_state_matrix)))
    return states

def search_path(initial_state):
    visited = {initial_state}
    states_queue = deque([[initial_state]])
    while states_queue:
        states_list = states_queue.popleft()
        state_to_expand = states_list[-1]
        for next_state in expand_state(state_to_expand):
            if next_state not in visited:
                if end_check(next_state):
                    return states_list + [next_state]
                visited.add(next_state)
                states_queue.append(states_list + [next_state])
    return []

def visualise_path(path):
    for states in zip(path, path[1:]):
        old_state, new_state = states
        for row in new_state:
            print(' '.join(map(str, row)))

        print()
```

Сензори

Распореди сензори на различни производители така што нема да има производител во ист ред и иста колона

```
start_state = ((0,0,0,0),(0,0,0,0),(0,0,0,0),(0,0,0,0))
manufacturer = [1,2,3,4]
```

```
def end_check(state):
    matrix_state = [list(row) for row in state]
    c = 0
    for i,row in enumerate(matrix_state):
        for j,column in enumerate(row):
            if matrix_state[i][j] == 0:
                c+=1
    if c == 0:
        return True
    else:
        return False

def is_valid(ma_value,state,x,y):
    if 0 > x > 3 and 0 > y > 3: # ako e nadvor od granicite vrati
false
        return False
    #prajme ciklus za da proverime dali e vo ist red ili ista kolona
    # we check if the ma_value appears on the same coords as x y,
which are available coords
    for i in range(4):
        if state[x][i] == ma_value or state[i][y] == ma_value:
            return False

    return True


def find_next_zero(state):
    for i, row in enumerate(state):
        for j,column in enumerate(row):
            if state[i][j] == 0:
                return i,j

def expand_state(state):
    states = []
    state_matrix = [list(row) for row in state]
    #finding the next zero coordinates
    x, y = find_next_zero(state_matrix)

    for m in manufacturer:
        if is_valid(m,state_matrix,x,y):
            state_matrix = [list(row) for row in state]
            state_matrix[x][y] = m
            states.append(tuple(map(tuple,state_matrix)))

    return states

def search(initial_state, alg):
    visited = {initial_state}
    states_queue = deque([initial_state])
    while states_queue:
```

```
        state_to_expand = states_queue.popleft()
        for next_state in expand_state(state_to_expand):
            if next_state not in visited:
                if end_check(next_state):
                    return next_state
                visited.add(next_state)
                if alg == 'dfs':
                    states_queue.appendleft(next_state)
                elif alg == 'bfs':
                    states_queue.append(next_state)
search(start_state,alg='dfs')
```

```
((4, 3, 2, 1), (3, 4, 1, 2), (2, 1, 4, 3), (1, 2, 3, 4))
```

Коњи шах

Да се постават што е можно повеќе коњи на шаховска табла така што нема да се напаѓаат меѓу себе.

```
def is_valid(state_matrix,x,y):
    if 0 > x >= N or 0 > y >= N:
        return False
    if state_matrix[x][y] == 1:
        return False
    possible_moves = [(1,2),(1,-2),(2,1),(-2,1),(-1,2),(2,-1),(-1,-2),
(-2,-1)]
    for m,n in possible_moves:
        new_m = m + x
        new_n = n + y
        #so eden oznacuvame deka na taa poz ima konnj
        if 0 <= new_m <= N-1 and 0 <= new_n <= N-1:
            if state_matrix[new_m][new_n] == 1:
                return False

    return True

def expand_state(state):
    states = []
    state_matrix = [list(row) for row in state]
    for i in range(N):
        for j in range(N):
            if is_valid(state_matrix,i,j):
                state_matrix = [list(row) for row in state]
                state_matrix[i][j] = 1
                states.append(tuple(map(tuple,state_matrix)))
    return states
```

```
def end_check(state):
    state_matrix = [list(row) for row in state]
    c = 0
    for i, row in enumerate(state_matrix):
        for j, column in enumerate(row):
            if state_matrix[i][j] == 1:
                c+=1
    #ovde ispituvame da ni raboti za bilo kolkava matrica
    if c % 2 == 0:
        if c == (N**2/2):
            return True

    elif c % 2 != 0:
        if c == ((N**2 + 1)/2):
            return True

def search(initial_state, alg):
    visited = {initial_state}
    states_queue = deque([initial_state])
    while states_queue:
        state_to_expand = states_queue.popleft()
        for next_state in expand_state(state_to_expand):
            if next_state not in visited:
                if end_check(next_state):
                    return next_state
                visited.add(next_state)
                if alg == 'dfs':
                    states_queue.appendleft(next_state)
                elif alg == 'bfs':
                    states_queue.append(next_state)
```

sijalicki lab-3

Разгледуваме игра која се игра самостојно. Правилата на играта се:

Се игра на табла со димензии N по N. Има празни и полни кругчиња, по едно кругче за секое поле од таблата. Се игра исклучиво со притискање на кругчињата. Ако се притисне некое кругче: Се менува од празно во полно, или пак обратно, од полно во празно. Неговите соседи се менуваат од празно во полно, или пак обратно, од полно во празно. Соседи на едно кругче се соседните кругчиња кои се наоѓаат над, под, лево и десно. Целта на играта е сите кругчиња да се празни.

Следи интерактивна апликација за оваа игра.

```
from plotly import graph_objects as go
from collections import deque
#import matplotlib.pyplot as plt
from pprint import pprint
```

```
import heapq
import random

def expand_square(square):
    neighbour_squares = []
    x, y = square
    for x, y in [(x, y), (x+1, y), (x-1, y), (x, y+1), (x, y-1)]:
        if 0 <= x < N and 0 <= y < N:
            neighbour_squares.append((x, y))
    return neighbour_squares

def toggle_state(x, y, TABLE):
    matrix_table = [list(row) for row in TABLE]
    for dx, dy in [(0, 0), (1, 0), (-1, 0), (0, 1), (0, -1)]:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < N and 0 <= new_y < N:
            matrix_table[new_x][new_y] = 1 - matrix_table[new_x]
[new_y]
    return tuple(map(tuple, matrix_table))

random.choice(states_test)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
Cell In[4], line 1
----> 1 random.choice(states_test)

NameError: name 'states_test' is not defined
```

```
states_test = expand_state(table1)
elements = [(x,y) for x in range(N) for y in range(N)]
#prajme torka od nuli
state = ((0,)*N,)*N
for _ in range(7):
    states_test += expand_state(random.choice(states_test))
states_test
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
Cell In[5], line 1
----> 1 states_test = expand_state(table1)
      2 elements = [(x,y) for x in range(N) for y in range(N)]
      3 #prajme torka od nuli

NameError: name 'expand_state' is not defined
```

```
def expand_state(TABLE):
    states = []
    for i,row in enumerate(TABLE):
        for j,element in enumerate(row):
            #if element == 1: go komentirame oti problemot mozi da se
popraj i iskluceno da go vklucime
                #print(j,element)
            NEW_TABLE = toggle_state(i,j,TABLE) #ovde mesto i+1 samo i
            states.append(NEW_TABLE)

    return states
    #return tuple(map(tuple, NEW_TABLE))


def end_check(table_to_check):
    table_check_matrix = [list(row) for row in table_to_check]
    count_1 = sum(row.count(1) for row in table_check_matrix)
    #print(count_1)
    if count_1 == 0:
        return True
    else:
        return False

def print_game_board(TABLE):
    step = 0
    print_game_board(TABLE)
    for row in TABLE:
        step+=1
        print(' '.join(map(str, row)))

def search_path(initial_state):
    visited = {initial_state}
    states_queue = deque([[initial_state]])
    while states_queue:
        states_list = states_queue.popleft()
        state_to_expand = states_list[-1]
        for next_state in expand_state(state_to_expand):
            if next_state not in visited:
                if end_check(next_state):
                    return states_list + [next_state]
                visited.add(next_state)
                states_queue.append(states_list + [next_state])
    return []

def visualise_path(path):
    for states in zip(path, path[1:]):
        old_state, new_state = states
        for row in new_state:
            print(' '.join(map(str, row)))
```

```
        print()

def h(state):
    #state_matrix = [list(row) for row in state]
    count_1 = sum(row.count(1) for row in state)
    answer = count_1/5
    return answer

def a_star_search(starting_vertex, heuristic_function, alpha=1):
    expanded = set()
    queue = [((0, 0), [starting_vertex])]
    heapq.heapify(queue)
    c = 0
    while queue:
        c += 1
        weight_tuple, vertex_list = heapq.heappop(queue)
        current_a_star_weight, current_path_weight = weight_tuple
        vertex_to_expand = vertex_list[-1]
        if end_check(vertex_to_expand):
            return current_path_weight, vertex_list, c
        if vertex_to_expand in expanded:
            continue
        for neighbour in expand_state(vertex_to_expand):
            if neighbour not in expanded:
                heuristic = heuristic_function(neighbour)
                path_weight = current_path_weight + 1
                a_star_weight = path_weight + alpha * heuristic
                heapq.heappush(queue, ((a_star_weight, path_weight),
vertex_list + [neighbour]))
        expanded.add(vertex_to_expand)
```

Градови во РМ

да се најдее најкраткиот пат мегју два града

```
class Graph():
    def __init__(self):
        """
        Initialises an empty dict as the graph data structure.
        """
        self.graph_dict = {}

    def add_vertex(self, vertex):
        """
        Adds a vertex to the graph.

        Args:
```

```
            vertex: vertex to be added in the graph
        """
        if vertex not in self.graph_dict:
            self.graph_dict[vertex] = []

    def vertices(self):
        """
        Returns the graph's vertices.
        """
        return list(self.graph_dict.keys())

    def add_edge(self, edge, add_reversed=True):
        """
        Adds an edge to the graph.

        Args:
            edge: a tupple of two vertices, (first_vertex,
second_vertex)
            add_reversed: whether to add the edge in reversed
direction, (second_vertex, first_vertex)
        """
        vertex1, vertex2 = edge
        self.graph_dict[vertex1].append(vertex2)
        if add_reversed:
            self.graph_dict[vertex2].append(vertex1)

    def edges(self):
        """
        Returns a list of all edges in the graph.
        """
        edges = []
        for vertex in self.graph_dict:
            for neighbour in self.graph_dict[vertex]:
                edges.append((vertex, neighbour))
        return edges

    def neighbours(self, vertex):
        """
        Returns all neighbours of the given vertex.
        """
        return self.graph_dict[vertex]

    def remove_vertex(self, vertex_to_remove):
        """
        Removes a vertex from the graph.

        First, the vertex's list is removed.
        Then, we remove all the occurances of the vertex in another
vertex's list.
```

```
        Args:
            vertex_to_remove: the vertex to be removed.
        """
        del self.graph_dict[vertex_to_remove]
        for vertex in self.vertices():
            if vertex_to_remove in self.graph_dict[vertex]:
                self.graph_dict[vertex].remove(vertex_to_remove)

    def remove_edge(self, edge_to_remove, remove_reversed=True):
        """
        Removes an edge from the graph.

        Args:
            edge_to_remove: the edge to be removed
            remove_reversed: whether to remove the edge in reversed
direction
        """
        vertex1, vertex2 = edge_to_remove
        if vertex2 in self.graph_dict[vertex1]:
            self.graph_dict[vertex1].remove(vertex2)
        if remove_reversed:
            if vertex1 in self.graph_dict[vertex2]:
                self.graph_dict[vertex2].remove(vertex1)

    def isolated_vertices(self):
        """
        Returns a list of all isolated vertices.
        """
        isolated_vertices = []
        for vertex in self.graph_dict:
            if not self.graph_dict[vertex]:
                isolated_vertices.append(vertex)
        return isolated_vertices

#go oznacuvame grafot
path = Graph()

path.add_vertex('S')
path.add_vertex('T')
path.add_vertex('G')
path.add_vertex('D')
path.add_vertex('K')
path.add_vertex('M')
path.add_vertex('U')
path.add_vertex('O')
path.add_vertex('R')
path.add_vertex('B')
path.add_vertex('V')
path.add_vertex('P')
```

```
path.add_edge(('S','T'))
path.add_edge(('S','V'))
path.add_edge(('T','G'))
path.add_edge(('G','K'))
path.add_edge(('K','M'))
path.add_edge(('K','D'))
path.add_edge(('K','U'))
path.add_edge(('K','O'))
path.add_edge(('O','U'))
path.add_edge(('O','R'))
path.add_edge(('D','U'))
path.add_edge(('R','B'))
path.add_edge(('B','P'))
path.add_edge(('V','P'))

path.neighbours('K')
```

```
['G', 'M', 'D', 'U', 'O']
```

```
def breadth_first_search_find_path(graph, starting_vertex,
goal_vertex, verbose=False):
    """
    Returns the path from starting_vertex to goal_vertex using the DFS
algorithm.
    """
    # Ако почетниот јазол е еднаков на целниот, тогаш нема логика да
пребаруваме воопшто
    if starting_vertex == goal_vertex:
        if verbose:
            print('Почетниот и бараниот јазол се исти')
        return []
    # Користиме листа на посетени јазли која всушност е податочна
структура множество.
    # За посетен јазол го сметаме оној јазол кој ќе го истражиме како
сосед на јазолот кој го разгрануваме.
    visited = {starting_vertex}
    # Користиме двојно поврзана листа која ни е редицата од која ќе го
земаме следниот јазол за разгранување.
    # Тука ја памтиме и моменталната патека за секој јазол од
почетниот.
    queue = deque([[starting_vertex]])
    # Пребаруваме сè додека има јазли за разгранување во редицата.
    while queue:
        if verbose:
            print('Ред за разгранување:')
            for element in queue:
                print(element, end=' ')
            print()
            print()
        # Членови на редицата јазли се патеките од почетниот јазол до
```

```
некој јазол кој треба да се разграни.
        # За да го земаме наредниот јазол за разгранување,
        #    ние треба од редицата да ја извадиме патеката на тој
јазол.
        vertex_list = queue.popleft()
        # Јазолот за разгранување е последниот во оваа листа.
        vertex_to_expand = vertex_list[-1]
        if verbose:
            print('Го разгрануваме јазолот
{}'.format(vertex_to_expand))
        # Го разгрануваме така што пребаруваме низ сите негови соседи.
        for neighbour in graph.neighbours(vertex_to_expand):
            if neighbour in visited:
                if verbose:
                    print('{} е веќе посетен'.format(neighbour))
            else:
                # Ако некој сосед не е посетен, тогаш го додаваме во
листата на посетени,
                #     и во редицата на јазли за разгранување.
                if verbose:
                    print('{}, кој е соседен јазол на {} го немаме
посетено до сега, затоа го додаваме во редот '
                          'за разгранување и го означуваме како
посетен'.format(neighbour, vertex_to_expand))
                # Тука ја вршиме проверката дали сме стигнале до
целниот јазол
                if neighbour == goal_vertex:
                    if verbose:
                        print('Го пронајдовме посакуваниот јазол {}.
Патеката да стигнеме до тука е {}'
                              .format(neighbour, vertex_list +
[neighbour]))
                    return vertex_list + [neighbour]
                visited.add(neighbour)
                # Бидејќи ова е пребарување прво по широчина,
                #     соседот го додаваме на крајот од редицата јазли
за разгранување.
                # Соседот го врзуваме со патеката од почетниот јазол
до моменталниот кој го разгрануваме.
                queue.append(vertex_list + [neighbour])
        if verbose:
            print()

breadth_first_search_find_path(path, 'S', 'M')
```

```
['S', 'T', 'G', 'K', 'M']
```