Домашна задача Група - 4

Задача 1:

(а)

-Табелата (Grid) кодирана ми е како кординатен систем, каде што сите пекмени почињаат во 0,0

-Во торка ми се ставени информациите за пекменот (x, y, bag_size),

x, y кординатата на пекменот и колкав е капацитетот на торбата, а сите пекмени ги статив во посебна торка ((x1, y1, bag_size), (x2, y2, bag_size), (x3, y3, bag_size), ...)

-Почетната состојба кодирана ми е во самите пекмени (x,y се 0 на почеток)

```
pacmans = tuple((0, 0, x) for i in range(k))
```

-Пребарувањето завршува кога или сите пелети ќе бидат собрани или кога торбите на сите пекмени ќе биде полна (во мојот код торбата е полна ако има вредност 0)

Во општ случај кога има к-пекмени, само 8 пекмени можат да се придвижат или горе или десно (максимум 4 во секое поле) а останатите пекмени ќе ја имаат одбрано акцијата стоп

Можни акции кои може да ги превземе било кој пекмен зависи од тоа:

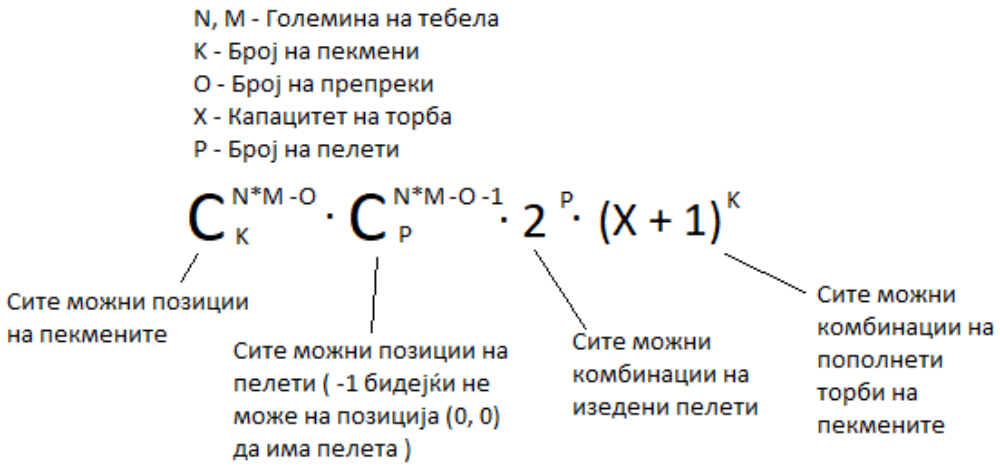
1) Дали полето кое што сака да се предвижи пекменот искача од табелата

2) Дали на полето кое што сака да се предвижи содржи препрека

3) Дали на полето кое што сака да се предвижи содржи 4 пекмени

-Во класата Problem, променливата inital (Се што е менливо) ги содржи позициите и капацитетот на торбите на сите пекмени и сите пелети на табелата.

(б)



(в)

-Максималниот вредност на фактор на разгранување на еден потег за еден пекмен е 5 (горе, доле, лево, десно, стоп).

-пр. [‘Pacman 1: right’, ‘Pacman 1: left,‘Pacman 1: up, ‘Pacman 1: down, ‘Pacman 1: stop]

-А во општ случај за k - пекмени, фактор на разгранување е 5 ^ k (Ако сите пекмени имаат можност да се движат во сите насоки)

-пр. за k = 2: ['Pacman 1: right Pacman 2: right', 'Pacman 1: right Pacman 2: left', 'Pacman 1: right Pacman 2: up', 'Pacman 1: right Pacman 2: down', 'Pacman 1: right Pacman 2: stop', ... , 'Pacman 1: stop Pacman 2: stop']

-Има вкупно 25 можни опции.

(г)

-Почетна состојба е кога сите пекмени имаат вредност (0, 0, x), односно се наоѓаат во координнатен систем на позиција 0,0 и имаат капацитет на торба x.

```
pacmans = tuple((0, 0, x) for i in range(k))
```

```
pacman = Pacman((pacmans, dots), obstacle, (m, n))
```

-(pacmans, dots) е initial состојба

-Целна состојба е кога нема нема повеќе пелети за собирање или торбата на сите пекмени е исполнета (односно има вредност 0).

```
def goal_test(self, state):
    boolean = True
    for pacman in state[0]:
        if (pacman[2] != 0):
            boolean = False
    return boolean or state[1].__len__() == 0
```

(д)

Акцијата е легелна ако ги задоволува следниве услови:

1)Ако на полето кое што сака да се предвижи пекменот не искача од табелата

2) Ако на полето кое што сака да се предвижи пекменот не содржи препрека

3) Ако на полето кое што сака да се предвижи содржи помалку од 4 пекмени

Пр. за вртење десно:

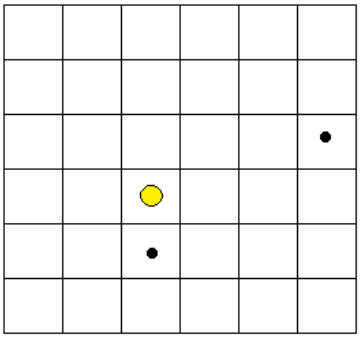
```
 if x_pacman + 1 < m and not obstacles.__contains__((x_pacman + 1, y_pacman)) and
number_of_pacmans_in_postition(x_pacman + 1, y_pacman, pacmans) < 4:
```

(ѓ)

Ha) Бројот на активни пелети поделени со бројот на пекемни

0 <= h(start) <= h*(start)

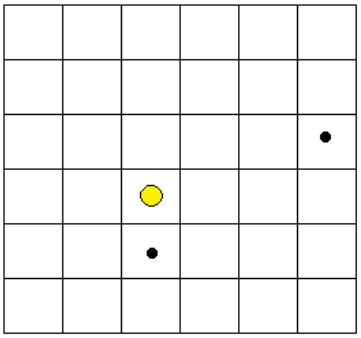
0 <= 2/1 <= 1



Следи дека не е допустлива, бидејќи се најде еден случај кога вредноста на хевристиката е поголема од вистинската (има многу повеќе)

Hb) Бројот на активни пелети

Не е допустлива во овој случај и во многу други:



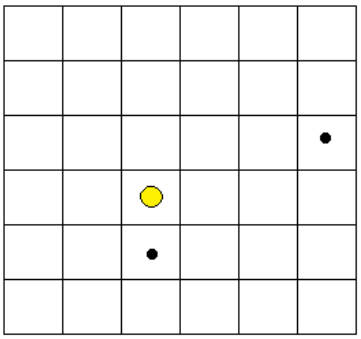
0 <= h(start) <= h*(start)

0 <= 2 <= 1

Бидејќи се најде барем еден случај кога хевристичката функција е поголема од вистинската (overestimate) тоа значи дека не е допустлива

Hc) Максималната менхетен одалеченост од пекмен и пелета од сите

пекмени и пелети



0 <= h(start) <= h*(start)

0 <= 4 <= 1

Со логиката од предходните хевристики истото важи и овде, што значи хевристикава не е допустлива

Hd) Макслимална вредност од минималните менхетен растојаниа помеѓу пекмените и пелетите

-Следи дека е допустлива и најдобра од сите останати 5 хевристики, бидејќи го одбира најлошиот случај на оддалеченост на пелети која најдобро ја апроксимира вистинската хевристика.

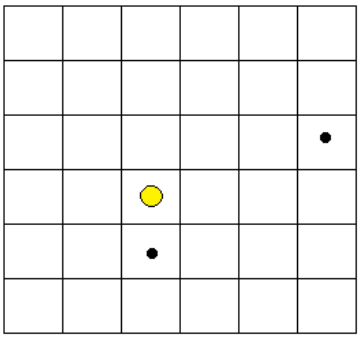
He) Минимална менхетен одалеченост од пекмен и пелета од сите пекмени и пелети

0 <= h(start) <= h*(start)

0 <= 2 <= 2

Следи дека е допустлива, бидејќи неможе никогаш да ја надмине вистинската хевристика, но полоша од предходната (Hd)

Hf) Минимална вредност од максималите менхетен растојаниа помеѓу пекмените и пелетите



0 <= h(start) <= h*(start)

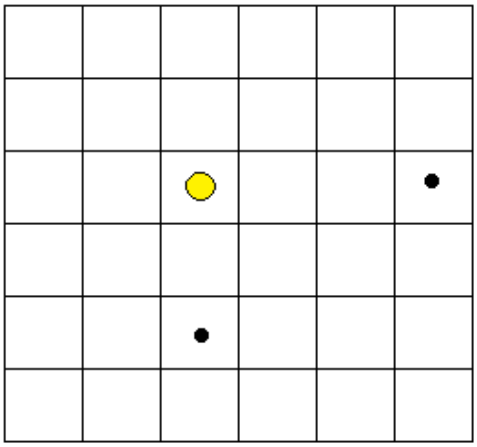
0 <= 4 <= 1

Бидејќи се најде барем еден случај кога хевристичката функција е поголема од вистинската (overestimate) тоа значи дека не е допустлива.

(e)

```
def h(self, node):
        pacmans = node.state[0]
        dots = node.state[1]

        bag_size = 0
        for pacman in pacmans:
            bag_size += pacman[2]


        if self.max_bag_size * len(pacmans) <= self.initial_dot_lenght:
            return bag_size/len(pacmans)
        else:
            return len(dots)/len(pacmans)
```

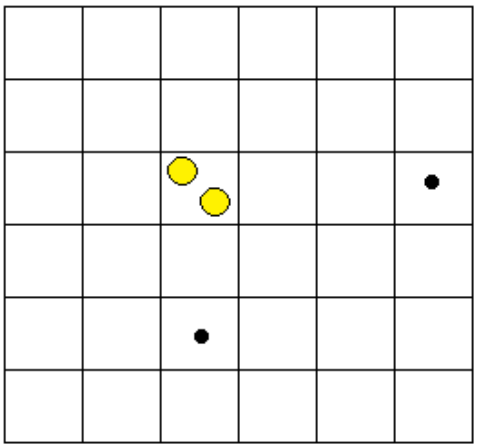
Се користи: `return len(dots)/len(pacmans)`



0 <= h(start) <= h*(start)

0 <= 2/1 <= 2

Се користи: `return bag_size/len(pacmans)`



0 <= h(start) <= h*(start)

0 <= 1/2 <= 3

Хевристика:

Прв случај: Ако X * K <= P (X-капацитет на торба, K- број на пекмени, P -број на пелети) тогаш goal_state е постигнат кога капацитетот на сите торби ќе биде 0, а тоа секогаш се случува дека бројот на пелети е поголем од сите торби.

Втор Случај: Ако X * K > P тогаш goal_state е постигнат кога ќе се соберат сите пелети, а тоа секогаш се случува бидејќи бројот на пелети е помал од капацитетот на тобрите на сите пекемни.

(Немав доволно време да смислам добра хевристика без да употребам менхетен растојание)

(ж)

-DFS: Не дава оптимално решение

-BFS: Дава оптимално решение, но е не ефикасен (долго време на извршување)

-UCS: Дава оптимално решение, но е не ефикасен

-A*: Даваат оптиманлно решение под услов да има допустлива хевристика

-Бидејќи факторот на разгранување експоненцијално расте со бројот на пекмени, неинформираното пребарување (BFS, DFS и UCS) ќе страдаат од долго време на компутација, односно ќе има голема временска комплексност. Од друга страна информираното пребарување (A*) со помош на допустлива хевристика драстично ќе го намали времето на компутација (Временската комплексност). Според овие причини ја би користел A* search, бидејќи поважна ми е временската комплексност да биде помала.

Целосен код:

```
def number_of_pacmans_in_postition(x, y, pacmans):
    num = 0
    for i in range(pacmans.__len__()):
        if pacmans.__contains__((x, y)):
            num += 1
    return num


def update_pacmans(pacmans, x_pacman, y_pacman, bag_size, i):
    pacmans = list(pacmans)

    pacmans[i] = list(pacmans[i])
    pacmans[i][0] = x_pacman
    pacmans[i][1] = y_pacman
    pacmans[i][2] = bag_size
    pacmans[i] = tuple(pacmans[i])

    pacmans = tuple(pacmans)
    return pacmans


def update_dots(x_pacman, y_pacman, dots):
```

```
    if dots.__contains__((x_pacman, y_pacman)):
        dots = list(dots)
        dots.remove((x_pacman, y_pacman))
        dots = tuple(dots)
    return dots


def right(pacmans, pacman_number_i, i, dots, obstacles, board_size):
    x_pacman = pacman_number_i[0]
    y_pacman = pacman_number_i[1]
    bag_size = pacman_number_i[2]
    m = board_size[0]
    n = board_size[1]
    if bag_size == 0:
        return None
    if x_pacman + 1 < m and not obstacles.__contains__((x_pacman + 1, y_pacman))
and number_of_pacmans_in_postition(
            x_pacman + 1, y_pacman, pacmans) < 4:
        x_pacman = x_pacman + 1
        if dots.__contains__((x_pacman, y_pacman)):
            bag_size -= 1
        dots = update_dots(x_pacman, y_pacman, dots)
        pacmans = update_pacmans(pacmans, x_pacman, y_pacman, bag_size, i)
    else:
        return None
    return pacmans, dots


def left(pacmans, pacman_number_i, i, dots, obstacles, board_size):
    x_pacman = pacman_number_i[0]
    y_pacman = pacman_number_i[1]
    bag_size = pacman_number_i[2]
    m = board_size[0]
    n = board_size[1]
    if bag_size == 0:
        return None
    if x_pacman - 1 >= 0 and not obstacles.__contains__((x_pacman - 1, y_pacman))
and number_of_pacmans_in_postition(
            x_pacman - 1, y_pacman, pacmans) < 4:
        x_pacman = x_pacman - 1
        if dots.__contains__((x_pacman, y_pacman)):
            bag_size -= 1
        dots = update_dots(x_pacman, y_pacman, dots)
        pacmans = update_pacmans(pacmans, x_pacman, y_pacman, bag_size, i)
    else:
        return None
    return pacmans, dots


def down(pacmans, pacman_number_i, i, dots, obstacles, board_size):
    x_pacman = pacman_number_i[0]
    y_pacman = pacman_number_i[1]
    bag_size = pacman_number_i[2]
    m = board_size[0]
    n = board_size[1]
    if bag_size == 0:
        return None
    if y_pacman - 1 >= 0 and not obstacles.__contains__((x_pacman, y_pacman - 1))
```

```
and number_of_pacmans_in_postition(
            x_pacman, y_pacman - 1, pacmans) < 4:
        y_pacman = y_pacman - 1
        if dots.__contains__((x_pacman, y_pacman)):
            bag_size -= 1
        dots = update_dots(x_pacman, y_pacman, dots)
        pacmans = update_pacmans(pacmans, x_pacman, y_pacman, bag_size, i)
    else:
        return None
    return pacmans, dots


def up(pacmans, pacman_number_i, i, dots, obstacles, board_size):
    x_pacman = pacman_number_i[0]
    y_pacman = pacman_number_i[1]
    bag_size = pacman_number_i[2]
    m = board_size[0]
    n = board_size[1]
    if bag_size == 0:
        return None
    if y_pacman + 1 < n and not obstacles.__contains__((x_pacman, y_pacman + 1))
and number_of_pacmans_in_postition(
            x_pacman, y_pacman + 1, pacmans) < 4:
        y_pacman = y_pacman + 1
        if dots.__contains__((x_pacman, y_pacman)):
            bag_size -= 1
        dots = update_dots(x_pacman, y_pacman, dots)
        pacmans = update_pacmans(pacmans, x_pacman, y_pacman, bag_size, i)
    else:
        return None
    return pacmans, dots


def stop(pacmans, dots):
    return pacmans, dots


def menheten(p1, p2):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])



class Pacman(Problem):
    def __init__(self, initial, obstacles, board_size, goal = None):
        super().__init__(initial, goal)
        self.obstacles = obstacles
        self.board_size = board_size
        self.number_of_initial_dots = len(initial[1])
        self.number_of_pacmans = len(initial[0])
        self.max_bag_size = initial[0][1][2]
        self.initial_dot_lenght = len(initial[1])

    def goal_test(self, state):
        boolean = True
        for pacman in state[0]:
            if (pacman[2] != 0):
                boolean = False
        return boolean or state[1].__len__() == 0
```

```
    def successor(self, state):
        successor = dict()
        dots = state[1]
        pacmans = state[0]

        for i in range(self.number_of_pacmans):
            pacman_number_i = pacmans[i]

            if i > 0:
                successor_tmp = dict()

                for key in successor:
                    dots = successor.get(key)[1]
                    pacmans = successor.get(key)[0]

                    if pacman_number_i[2] > 0:
                        tmp = right(pacmans, pacman_number_i, i, dots,
self.obstacles, self.board_size)
                        if tmp is not None:
                            successor_tmp[key + ' Pacman ' + str(i+1) + ': right']
= tmp

                        tmp = left(pacmans, pacman_number_i, i, dots,
self.obstacles, self.board_size)
                        if tmp is not None:
                            successor_tmp[key + ' Pacman ' + str(i+1) + ': left']
= tmp

                        tmp = up(pacmans, pacman_number_i, i, dots, self.obstacles,
self.board_size)
                        if tmp is not None:
                            successor_tmp[key + ' Pacman ' + str(i+1) + ': up'] =
tmp

                        tmp = down(pacmans, pacman_number_i, i, dots,
self.obstacles, self.board_size)
                        if tmp is not None:
                            successor_tmp[key + ' Pacman ' + str(i+1) + ': down']
= tmp
                    else:
                        successor_tmp[key + ' Pacman ' + str(i+1) + ': stop'] =
stop(pacmans, dots)
                successor = successor_tmp

            else:
                if pacman_number_i[2] > 0:
                    tmp = right(pacmans, pacman_number_i, i, dots, self.obstacles,
self.board_size)
                    if tmp is not None:
                        successor['Pacman ' + str(i+1) + ': right'] = tmp

                    tmp = left(pacmans, pacman_number_i, i, dots, self.obstacles,
self.board_size)
                    if tmp is not None:
                        successor['Pacman ' + str(i+1) + ': left'] = tmp

                    tmp = up(pacmans, pacman_number_i, i, dots, self.obstacles,
```

```
self.board_size)
                    if tmp is not None:
                        successor['Pacman ' + str(i+1) + ': up'] = tmp

                    tmp = down(pacmans, pacman_number_i, i, dots, self.obstacles,
self.board_size)
                    if tmp is not None:
                        successor['Pacman ' + str(i+1) + ': down'] = tmp
                else:
                    successor['Pacman ' + str(i+1) + ': stop'] = stop(pacmans, dots)
        return successor

    def actions(self, state):
        return self.successor(state).keys()

    def result(self, state, action):
        return self.successor(state)[action]

    def h(self, node):
        pacmans = node.state[0]
        dots = node.state[1]

        bag_size = 0
        for pacman in pacmans:
            bag_size += pacman[2]

        if self.max_bag_size * len(pacmans) <= self.initial_dot_lenght:
            return bag_size/len(pacmans)
        else:
            return len(dots)/len(pacmans)



if __name__ == '__main__':
    m = int(input())
    n = int(input())
    k = int(input())
    x = int(input())
    p = int(input())

    inputs = [input().split(",") for i in range(p)]
    dots = tuple((int(dot[0]), int(dot[1])) for dot in inputs)

    pacmans = tuple((0, 0, x) for i in range(k))

    o = int(input())
    inputs = [input().split(",") for i in range(o)]
    obstacle = tuple((int(ob[0]), int(ob[1])) for ob in inputs)

    pacman = Pacman((pacmans, dots), obstacle, (m, n))

    start = time.time()
    result = astar_search(pacman)
    end = time.time()

    print("Default test_case:")
    print(result.solution())
```

```
    print(str(result.solution().__len__()) + " Moves")
    print(str(end - start) +" Seconds")
```

Задача 2:

(а)

- Променливите се сите полиња кои немаат почетна вредност, нумерирани од 0 до 15 (почнувајќи од доле лево, без променливите 8 и 15 бидејќи имаат почетна вредност)

- Доменот (можните вредности кои можат да бидат доделени на променливите) ќе биде од 1 до 4

- Имам напревено два вида на ограничувања (две функции за **addConstraint**):

constraint_function1 функцијата ја користам кога променливите се во иста редица или колона и ако се поврзани преку знакот ><

constraint_function2 ја користам само кога променливите се наоѓаат во ист ред или колона

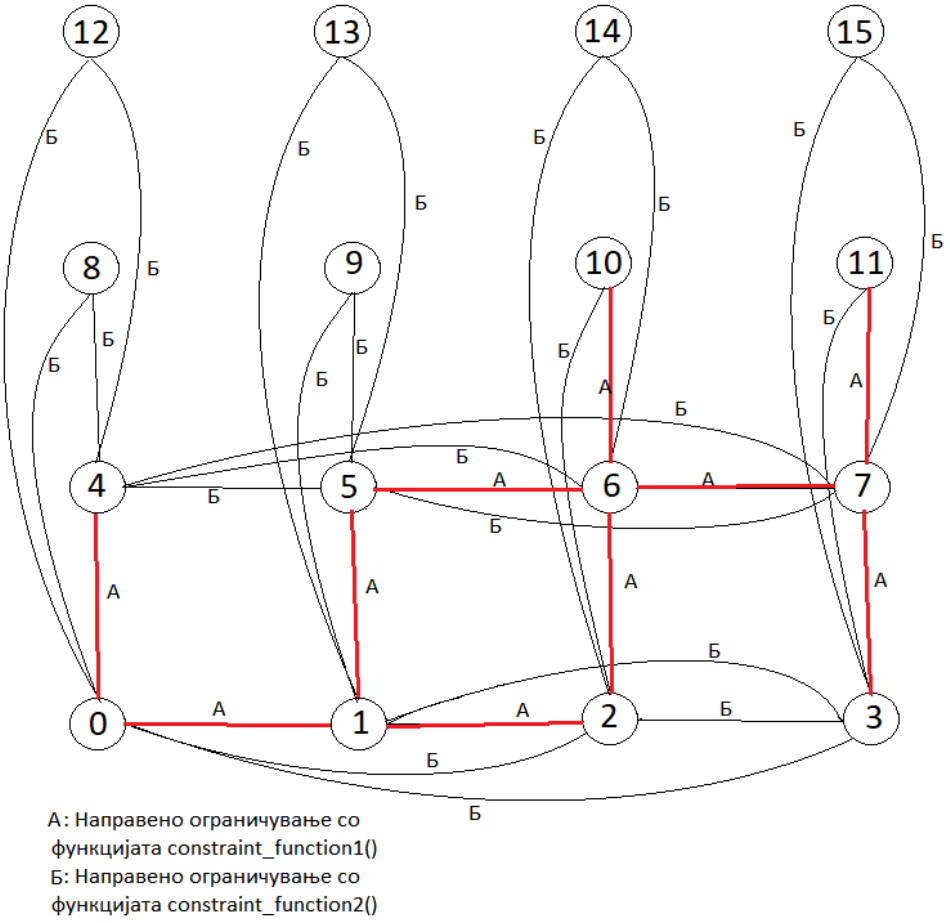
```
def constraint_function1(var1, var2):
    print(problem._variables)
    if var1 != var2 and abs(var1 - var2) == 1:
        return True
    return False


def constraint_function2(var1, var2):
    print(problem._variables)
```

```
    if var1 != var2:
        return True
    return False
```

Функцијата **get_list_of_variables(v)** ги враќа сите променливи кои се нои се наоѓаат во ист ред и иста колона со променливата v

```
def get_list_of_variables(v):
    num_row = int(v / 4)
    num_col = v % 4
    list_row = [x + num_row * 4 for x in range(4) if v != x + num_row * 4]
    list_col = [num_col + 4 * x for x in range(4) if
                not list_row.__contains__(num_col + 4 * x) and num_col + 4 * x !=
v]
    return list_row + list_col
```

(б)



-Со црвена линија е прикажан знакот ><

-Во графот се направени сите ограничувања за промеливите 0, 1, 2, 3, 4, 5, 6 и 7 (Лошо изгледаше кога ги нацрав сите ограничувања помеѓу сите променливи)

(в)

-Хевристика за избор на променлива: MVR (Minimum Remaining Values)

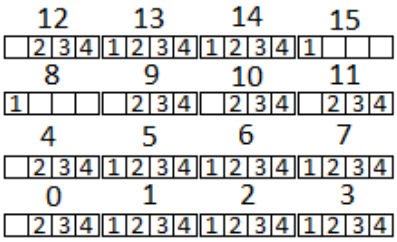
-Хевристика за избор на вредност на променлива: Least Constraining Value

Чекор: 1

Променлива: 8 е избрана

Вредност: 1

Forward checking:

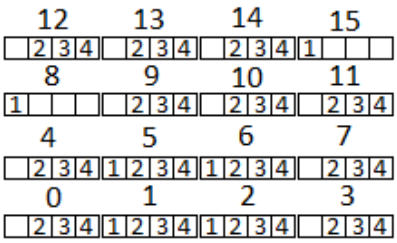


Чекор: 2

Променлива: 15 е избрана

Вредност: 1

Forward checking:

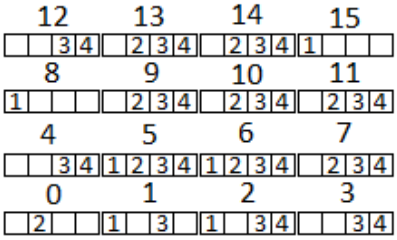


Чекор: 3

Променлива: 0 е избрана

Вредност: 2

Forward checking:

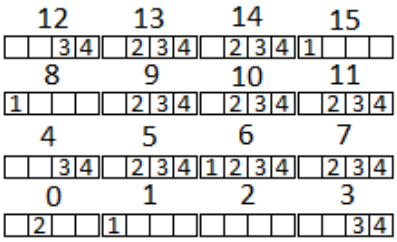


Чекор: 4

Променлива: 1 е избрана

Вредност: 1

Forward checking:

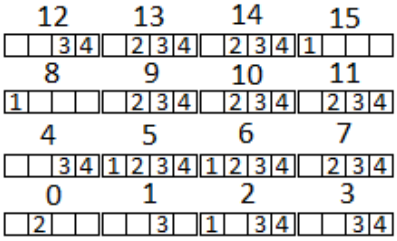


Чекор: 5

Променлива: 2 е избрана

Вредност: Непостоечка

Backtracking:

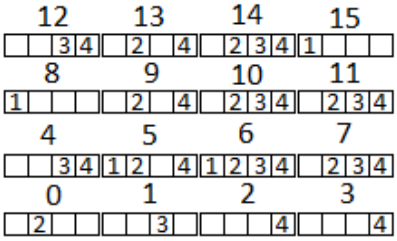


Чекор: 6

Променлива: 1 е избрана

Вредност: 3

Forward checking:

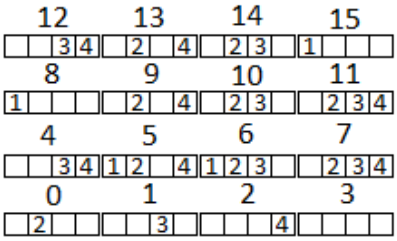


Чекор: 7

Променлива: 2 е избрана

Вредност: 4

Forward checking:



Чекор: 8

Променлива: 3 е избрана

Вредност: Непостоечка

Backtracking:

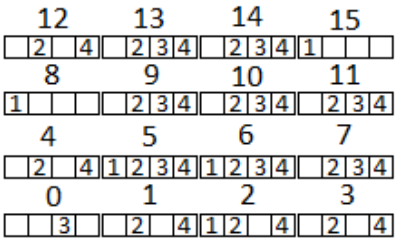
...

Чекор: x

Променлива: 0 е избрана

Вредност: 3

Forward checking:

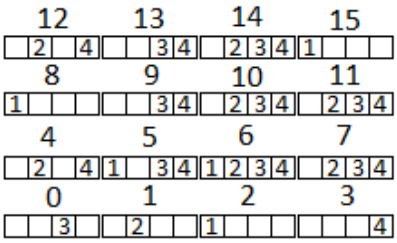


Чекор: x+1

Променлива: 1 е избрана

Вредност: 2

Forward checking:



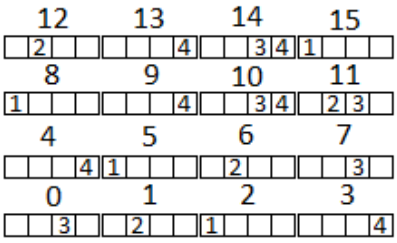
...

Чекор: y

Променлива: 6 е избрана

Вредност: 2

Forward checking:



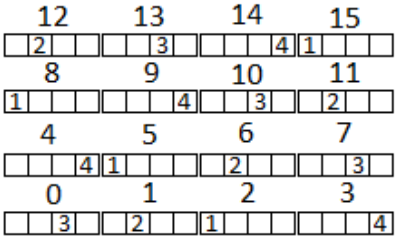
...

Чекор: Последен

Променлива: 14 е избрана

Вредност: 4

Forward checking:



Целосен код:

```
from constraint import *

connected_variables = ((0, 1), (0, 4), (1, 5), (1, 2), (2, 6),
                       (3, 7), (7, 6), (6, 5), (6, 10), (8, 12),
                       (9, 13), (9, 10), (10, 11), (10, 14), (11, 15),
                       (12, 13), (13, 14))


def get_list_of_variables(v):
    num_row = int(v / 4)
    num_col = v % 4
    list_row = [x + num_row * 4 for x in range(4) if v != x + num_row * 4]
    list_col = [num_col + 4 * x for x in range(4) if
                not list_row.__contains__(num_col + 4 * x) and num_col + 4 * x !=
v]
    return list_row + list_col


def constraint_function1(var1, var2):
    print(problem._variables)
    if var1 != var2 and abs(var1 - var2) == 1:
        return True
    return False


def constraint_function2(var1, var2):
    print(problem._variables)
    if var1 != var2:
        return True
    return False
```

```
if __name__ == '__main__':
    domains = range(1, 5)
    variables = [i for i in range(16) if i != 8 and i != 15]
    problem = Problem()

    problem.addVariables(variables, domains)
    problem.addVariable(15, [1])
    problem.addVariable(8, [1])

    for variable1 in variables:
        list = get_list_of_variables(variable1)
        for variable2 in list:
            if connected_variables.__contains__((variable1, variable2)) or
connected_variables.__contains__(
                    (variable2, variable1)):
                problem.addConstraint(constraint_function1, (variable1,
variable2))
            else:
                problem.addConstraint(constraint_function2, (variable1,
variable2))

    solution = problem.getSolution()
    print(solution)
```

Изработил: Бојан Робев, 185028 (КИ)