ЛАБОРАТОРНАЯ РАБОТА 2
"ИЗУЧЕНИЕ АЛГОРИТМОВ ПОИСКА РЕШЕНИЙ: DFS, BFS, A*SEARCH"

Цель работы: знакомство с задачей поиска пути на дереве решений, реализация и исследование алгоритмов поиска пути: поиск в глубину, поиск в ширину и поиск с использованием оценочной функцией.

Задачи:

– выбрать игру-головоломку в соответствии с вариантом задания;

– описать состояние игры, ход игры в терминах структур данных выбранного языка реализации алгоритмов;

– придумать и реализовать следующие алгоритмы:

1) инициализация начального состояния игры;

2) получение списка всех возможных ходов для заданного состояния;

3) выполнение хода;

4) отмена хода;

5) проверка, что достигнута цель игры;

– придумать и реализовать как минимум одну оценочную функцию;

– изучить учебный пример – реализацию игры 3x3;

– реализовать алгоритмы *dfs*, *bfs*, *asearach* (или использовать их реализацию из учебного примера);

– сравнить количество вершин дерева игры для трех алгоритмов и время поиска хода в зависимости от заданной глубины дерева.

Отчет по лабораторной работе должен содержать:

– описание игры-головоломки и ее правил;

– описание состояние игры, ход игры в терминах структур данных выбранного языка реализации алгоритмов;

– описание алгоритма расчета оценочной функции;

– исходный код программы (код модулей учебного примера, которые использовались, но не изменялись, в отчет не включать).

ОПИСАНИЕ УЧЕБНОГО ПРИМЕРА

Учебный пример реализован в виде следующего набора модулей:

– *base.py* – модуль базового класса для описания состояния игры;

– *dfs.py* – модуль для поиска в глубину;

– *bfs.py* – модуль для поиска в ширину;

– *asearch.py* – модуль для поиска с оценочной функцией.

Модуль базового класса *base.py* определяет набор методов, которые необходимо реализовать в классе-наследнике (для реализации конкретной игры), чтобы работали методы поиска в модулях *dfs, bfs, asearch*.

– *eight.py* – модуль класса для описания состояния игры 3x3;

– *eight_test.py* – модуль для проверки работоспособности модуля *eight.py*;

– *dfs_test.py* – модуль для проверки поиска в глубину для игры 3x3;

– *bfs_test.py* – модуль для проверки поиска в ширину для игры 3x3;

– *asearch_test.py* – модуль для проверки поиска с оценочными функциями для игры 3x3.

При выполнении лабораторных работ можно использовать программный код из учебного примера.

ВАРИАНТЫ ЗАДАНИЙ

1) Поиск пути в лабиринте. Лабиринт представляет собой двумерную сеть ячеек, часть из которых – «стены», через которые нельзя пройти. Игрок находится в некоторой начальной ячейке и может совершать ходы в соседние свободные ячейки. Цель – найти путь – последовательность ходов, ведущую от начальной ячейки до конечной. (Дополнительное задание: генерация случайных лабиринтов).
2) Три миссионера и три людоеда находятся на западном берегу реки. У них есть каноэ, в котором помещаются два человека, и все они должны переехать на восточный берег реки. Нельзя, чтобы на любой стороне реки в какой-то момент оказалось больше людоедов, чем миссионеров, иначе каннибалы съедят миссионеров. Кроме того, в каноэ всегда должен находиться хотя бы один человек, чтобы пересечь реку. Какая последовательность переправ позволит успешно перевезти всех через реку?
3) Имеется карта, на которой необходимо разными цветами обозначить территории. Никакие две соседние области не должны быть окрашены в одинаковый цвет. Определить, какое минимальное количество красок может быть использовано?
4) Шахматная доска - это сетка размером 8 x 8 клеток. Ферзь - шахматная фигура, которая может перемещаться по шахматной доске на любое количество клеток по любой горизонтали, вертикали или диагонали. Если за один ход ферзь может переместиться на клетку, на которой стоит другая фигура, не перепрыгивая ни через какую другую фигуру, то ферзь атакует эту фигуру. Задача восьми ферзей состоит в том, как разместить восемь ферзей на шахматной доске таким образом, чтобы ни один из них не атаковал другого.
5) **Мосты** — это логическая головоломка (https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D1%81%D1%82%D1%8B_(%D0%

B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BE%D0%BB%D0%BE%D0%BC%D0%BA%D0%B0). Задача игрока заключается в том, чтобы соединить линиями острова, и при этом число мостов должно соответствовать указанному на острове числу. Острова соединяются при помощи прямых линий по следующим правилам:
    a. Число в кружке соответствует количеству мостов у данного острова.
    b. Между любыми двумя островами не может быть более двух мостов.
    c. Мосты должны быть горизонтальными или вертикальными, и не могут пересекать другие мосты и острова.
    d. Острова должны быть соединены так, чтобы с любого острова можно было попасть на любой другой.
6) SEND + MORE = MONEY - это криптоарифметическая головоломка, где нужно найти такие цифры, которые, будучи подставленными вместо букв, сделают математическое утверждение верным. Каждая буква в задаче представляет одну цифру от 0 до 9. Никакие две разные буквы не могут представлять одну и ту же цифру. Если буква повторяется, это означает, что цифра в решении также повторяется.
7) Производитель должен установить прямоугольные микросхемы на прямоугольную плату. По сути, эта задача сводится к вопросу: «Как разместить все прямоугольники разных размеров внутри другого прямоугольника таким образом, чтобы они плотно прилегали друг к другу?
8) Игра в 15 (пятнашки) – представляет собой набор одинаковых квадратных костяшек с нанесёнными числами (от 1 до 15, одно поле - пустое), заключённых в квадратную коробку 4x4. Цель игры — перемещая костяшки по коробке, добиться упорядочивания их по номерам, желательно сделав как можно меньше перемещений.
9) Судоку. Игровое поле представляет собой квадрат размером 9×9, разделённый на меньшие квадраты со стороной в 3 клетки. Таким образом, всё игровое поле состоит из 81 клетки. В них уже в начале игры стоят некоторые числа (от 1 до 9), называемые подсказками. От игрока требуется заполнить свободные клетки цифрами от 1 до 9 так, чтобы в каждой строке, в каждом столбце и в каждом малом квадрате 3×3 каждая цифра встречалась бы только один раз.
10) Ханойская башня. Даны три стержня, на один из которых нанизаны восемь колец, причём кольца отличаются размером и лежат меньшее на большем. Задача состоит в том, чтобы перенести пирамиду из восьми колец за наименьшее число ходов на другой стержень. За один раз разрешается переносить только одно кольцо, причём нельзя класть большее кольцо на меньшее.
11) **Каку́ро** (https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BA%D1%83%D1%80%D0%BE). Поле состоит из клеток чёрного и белого цвета. Несколько белых клеток, идущих подряд по горизонтали или по вертикали, называются блоком. Про каждый блок известна сумма цифр, которые должны стоять в этом блоке. Для горизонтальных блоков эта сумма обычно записывается непосредственно слева от блока, а для вертикальных — непосредственно сверху.
12) Dominosa Omnibus – представляет собой разложенный в виде определённой фигуры комплект из 28 костей домино, границы между которыми стёрты. Задача состоит в том, чтобы отметить, где какая кость лежит (https://ru.wikipedia.org/wiki/Dominosa_Omnibus).

ТЕКСТЫ ПРОГРАММНЫХ МОДУЛЕЙ

1. Модуль base.py

```
class base_state:
    def __hash__(self):
        '''Метод должен возвращать значение хэш-функции -
           числа, которое будет уникальным образом
           задавать состояние игры
           (используется для того, чтобы можно было
           использовать тип множество для объектов класса)
        '''
        raise NotImplementedError

    def __eq__(self, other):
        '''Перекрытие операции сравнения.
           Метод сравнивает два объекта,
           используя их хэш-функции
        '''
        return self.__hash__()==other.__hash__()

    def get_moves(self):
        '''Метод должен возвращать список состояний,
           в которые можно перейти из текущего
        '''
        raise NotImplementedError
```

2. Модуль dfs.py

```
def dfs(initial, goal, max_depth):
    """Deep search function

    Keyword arguments:
        initial -- inital state, start of the search tree
        goal -- final state, we want to find path
        max_depth -- maximum of search tree level

    Return: a tuple (is_find, path, open_state_count, close_state_count)
        is_find -- does path has been found
        path -- a path to final state, a solution
        open_state_count -- count of elements in open states list
        close_state_count -- count of elements in close states list
    """
    if initial == goal: # начальное состояние равно целевому
        return (True, [], 0, 0)

    # список открытых состояний
    initial.depth = 0
    open_states = [initial]

    # список закрытых состояний
    closed_states = []

    while open_states != []:
        # извлекаем последний элемент из списка открытых состояний
        current = open_states.pop()
        # добавляем его в закрытые
        closed_states.append(current)
        # генерируем список возможных ходов
        for m in current.get_moves():
            # если этот ход не в списке закрытых состояний
            if not m in closed_states:
                if m == goal:
                    # сформировать список ходов до текущего
                    path = [m]
                    s = m.parent
```

```
                    while s:
                        path.append(s)
                        s = s.parent
                    path.reverse()
                    return (True, path, len(open_states), len(closed_states))
                if current.depth <= max_depth:
                    open_states.append(m)
    return (False, [], -1, -1) # нет решения, возвращаем пустой список
```

3. Модуль bfs.py

```
def bfs(initial, goal):
    """breadth search function

    Keyword arguments:
    initial -- inital state, start of the search tree
    goal -- final state, we want to find path

    Return: a tuple (is_find, path, open_state_count, close_state_count)
    is_find -- does path has been found
    path -- a path to final state, a solution
    open_state_count -- count of elements in open states list
    close_state_count -- count of elements in close states list
    """
    if initial == goal: # начальное состояние равно целевому
        return (True, [], 0, 0)

    # список открытых состояний
    initial._depth = 0
    open_states = [initial]

    # список закрытых состояний
    closed_states = []

    while open_states != []:
        # извлекаем первый элемент из (очереди) списка открытых состояний
        current = open_states.pop(0)
        # добавляем его в закрытые
        closed_states.append(current)

        # генерируем список возможных ходов
        for m in current.get_moves():
            # get_moves() это делает, но вдруг забыли реализовать
            m._depth = current._depth + 1

            # если этот ход не в списке закрытых состояний
            if not m in closed_states:
                if m == goal:
```

```
                    # сформировать список ходов до текущего
                    path = [m]
                    s = m.parent
                    while s:
                        path.append(s)
                        s = s.parent
                    path.reverse()
                    return (True, path, len(open_states), len(closed_states))
                open_states.append(m)
    return (False, [], -1, -1) # нет решения, возвращаем пустой список
```

4. Модуль asearch.py

```
from queue import PriorityQueue

def a_search(initial, goal, evaluator):
    """A*search algorythm function

    Keyword arguments:
        initial -- inital state, start of the search tree
        goal -- final state, we want to find path
        evaluator -- a function to calculate priority for queue

    Return: a dict (solved, path, open state count, close_state_count)
        is_find -- does path has been found
        path -- a path to final state, a solution
        open_state_count -- count of elements in open states list
        close_state_count -- count of elements in close states list
    """
    if initial == goal: # начальное состояние равно целевому
        return {'solved':True, 'path': [initial], 'openstates':1, 'closedstat
es':0}

    initial._depth = 0

    # список открытых состояний
    open_states = PriorityQueue()
    open_states.put((evaluator(initial, goal), initial))

    # список закрытых состояний
    closed_states = set()

    while not open_states.empty():
        # извлекаем первый элемент из (очереди) списка открытых состояний
        _, current = open_states.get()

        # добавляем его в закрытые
        closed_states.add(current)

        if current == goal:
            # сформировать список ходов до текущего
```

```
            item = current
            path = [item]
            while item._parent != None:
                item = item._parent
                path.append(item)
            path.reverse()
            return {'solved':True,
                'path': path,
                'openstates':len(open_states.queue),
                'closedstates':len(closed_states)
                }

        # генерируем список возможных ходов
        for move in current.get_moves():
            # если этот ход не в списке закрытых состояний
            if not move in closed_states:
                move._depth = current._depth + 1
                # по алгоритму надо проверить, есть ли move
                # в очереди открытых состояний
                # извлечь его и сравнить с приоритетом
                # текущего состояния move
                # если приоритет move меньше,
                # то заменить в очереди состояние на move
                # но если не искать и не удалять, что мы просто потом
                # извлекая очередное состояние из очереди будем видеть,
                # что оно уже есть в закрытых
                open_states.put((evaluator(move, goal) + move.depth, move))

    # нет решения, возвращаем пустой список
    return {'solved':False,
        'path': [],
        'openstates':len(open_states.queue),
        'closedstates':len(closed_states)
        }
```

5. Модуль eight.py

```
from base import base_state

class state(base_state):
    space = 0 # код для обозначения пустого поля
    size = 3

    # соответствие номеров ячеек и их координат
    places = {
        1:(0,0), 2:(0,1), 3:(0,2),
        4:(1,0), 5:(1,1), 6:(1,2),
        7:(2,0), 8:(2,1), 9:(2,2)
        }

    # список допустимых ходов для ячейки с заданным номером
    moves = {
        1: [2, 4],
        2: [1,3,5],
        3: [2, 6],
        4: [1, 5, 7],
        5: [2, 4, 6, 8],
        6: [3, 5, 9],
        7: [4, 8],
        8: [5, 7, 9],
        9: [6, 8]
        }

    @staticmethod
    def get_key(items, elem):
        for key, item in items.items():
            if item == elem:
                return key
        return None

    def __init__(self, parent, data, depth=0):
        if isinstance(data, dict):
            self.data = data
        else:
            if isinstance(data, list):
```

```
                self.data = {}
                for idx, cell in enumerate(data):
                    self.data[cell] = idx+1
        self.parent = parent
        self.depth = depth

    def __hash__(self):
        k = 0
        d = 1
        for key in sorted(self.data.keys()):
            k += d * self.data[key]
            d *= 10
        return k

    def __str__(self):
        field = [[0,0,0],[0,0,0],[0,0,0]]
        for key, item in self.data.items():
            row, col = self.places[item]
            field[row][col] = key
        return str(field)

    def __eq__(self, other):
        return self.data == other.data

    def get_moves(self):
        d = self.data
        new_moves = []
        # получаем список ходов для пустого поля:
        for position in self.moves[d[0]]:
            key = state.get_key(d, position)
            new_state = d.copy()
            new_state[0] = d[key]
            new_state[key] = d[0]
            new_moves.append(state(self, new_state, self.depth+1))
        return new_moves

# Fair Evaluator ::= P(h)
#   P(h) - сумма манхеттенских расстояний между ячейками текущего состояния и
 целевым
def fair_evaluator(state, goal):
```

```
    sum = 0
    for dice in range(1, state.size**2):
        x1, y1 = state.places[state.data[dice]]
        x2, y2 = state.places[goal.data[dice]]
        sum += abs(x2-x1) + abs(y2-y1)
    return sum

# Good Evaluator ::= P(h)+3*S(h)
#   P(h) - сумма манхеттенских расстояний между всеми ячейками
#   S(h) - для каждой плитки,
#            0 - если за ней идет корректный приемник
#            2 - в противном случае
#            1 - если это плитка в центре
def good_evaluator(state, goal):
    cells = [(0,1), (1, 2), (2, 5), (5, 8), (8, 7), (7, 6), (6, 3), (3, 0)]
    anc = lambda x: 1 if x == 8 else x+1

    s = 0
    # для каждой кости
    for dice in range(1, state.size**2):
        if not (state.data[dice], state.data[anc(dice)]) in cells:
            s += 2
    if state.data[state.space] != 4:
        s += 1
    return s*3 + fair_evaluator(state, goal)

# Weak Evaluator ::= N(h)
#   N(h) - количество плиток не на своем месте
def weak_evaluator(state, goal):
    # count = 0
    # for dice in range(1, state.size**2):
    #     if state.data[dice] != goal.data[dice]:
    #         count += 1
    # return count
    return sum([1 for dice in range(1, state.size**2) if state.data[dice] !=
goal.data[dice]])

# Bad Evaluator ::= |D(h) - 16|
#   D(h) - сумма разностей значений противоположных (относительно центра) пли
ток
```

```
#          для текущего состояния
#   G(h) - сумма разностей значений противоположных (относительно центра) пли
ток
#          для целевого состояния
def bad_evaluator(state, goal):
    f = lambda d: d[3]+d[6]+d[7]+d[8]-d[5]-d[2]-d[1]-d[0]
    d = [0 for _ in range(state.size**2)]
    for dice in range(1, state.size**2):
        d[state.data[dice]] = dice
    print(d)
    g = [goal.data[dice]  for dice in range(0, state.size**2)]
    print(g)
    return abs(f(d)-f(g))
```

6. Модуль dfs_test.py

```
from eight import state
from dfs import *
from timeit import Timer

initial = state(None, [[8,1,3],[2,4,5],[state.space,7,6]], 0)
goal = state(None, [[1,2,3],[8,state.space,4],[7,6,5]])

print(f"Initial state: {initial}")
print(f"Goal state: {goal}")

f = lambda: dfs(initial, goal, 20)

res = f()
print('has decision  :', res[0])
print('open states   :', res[2])
print('closed states :', res[3])
print('result path   :')
for m in res[1]:
    print(m)

t = Timer(f)
print("Time = ", t.timeit(number=1))
```

7. Модуль bfs_test.py

```
from eight import state
from bfs import *
from timeit import Timer

initial = state(None, [[8,1,3],[2,4,5],[state.space,7,6]], 0)
goal = state(None, [[1,2,3],[8,state.space,4],[7,6,5]])

print(initial)

f = lambda: bfs(initial, goal)

res = f()
print('has decision  :', res[0])
print('open states   :', res[2])
print('closed states :', res[3])
print('result path   :')
for m in res[1]:
    print(m)

t = Timer(f)
print("Time = ", t.timeit(number=100))
```

8. Модуль asearch_test.py

```
from eight import state, fair_evaluator, good_evaluator,\
     weak_evaluator, bad_evaluator
from asearch import a_search
from timeit import Timer

initial = state(None, [8,1,3,0,4,5,2,7,6])
goal = state(None, [1,2,3,8,0,4,7,6,5])

print('fair =', fair_evaluator(initial, goal))
print('good =', good_evaluator(initial, goal))
print('weak =', weak_evaluator(initial, goal))
print('bad  =', bad_evaluator(initial, goal))

f = lambda: a_search(initial, goal, fair_evaluator)
# f = lambda: a_search(initial, goal, good_evaluator)
# f = lambda: a_search(initial, goal, weak_evaluator)
# f = lambda: a_search(initial, goal, bad_evaluator)

res = f()
print('has decision  :', res['solved'])
print('open states   :', res['openstates'])
print('closed states :', res['closedstates'])
print('result path   :')
for m in res['path']:
    print(m)

t = Timer(f)
print("Time = ", t.timeit(number=1))
```