МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ВТ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Искусственный интеллект»

Тема: «Неинформированный поиск (императивный ЯП)»

Студенты гр. 9308 _____ Дементьев Д.П.
Хамитов А.К.
Яловега Н.В.

Преподаватель _____ Родионов С.В.

Санкт-Петербург
2022

Цель работы

Практическое закрепление понимания общих идей поиска в пространстве состояний и стратегий слепого поиска.

Постановка задачи

Рассматривается задача «Головоломка 8-ка».

Задана доска с 8 пронумерованными фишками и с одним пустым участком.

Фишка, смежная с пустым участком, может быть передвинута на этот участок. Требуется достичь указанного целевого состояния.

Нашей задачей является написание на языке Python алгоритмов поиска пути из начального состояния в конечное, которые изображены на рисунках 1 и 2 соответственно.

6		8
5	2	1
4	3	7

Рисунок 1. Начальное состояние

1	2	3
8		4
7	6	5

Рисунок 2. Конечное состояние

Написаны алгоритмы:

1. Поиск в глубину
2. Двунаправленный поиск

Описание выбранных структур данных

Структура	Поля	Описание
class **Node**	**State: list**	Состояние в пространстве остояний, которому соответствует данный узел
	Parent: Node	Указатель на родительский узел
	Previous_action: Action	Действие, которое было применено к родительскому узлу для формирования данного узла
	Path_cost: int	Стоимость пути от начального состояния до данного узла
	Depth: int	Количество этапов пути от начального состояния (глубина)
	Node_id: int	Уникальный идентификатор узла
class **Tree**	**hashset: dict**	Словарь с хэшами (уникальные идентификаторы) каждого узла

Описание методов класса Tree

Метод	Описание
get_root(self)->list	Получение корня дерева
get_node(self, node_id: int)->int	Получение узла по его идентификатору
add_node(self, node_id: int)->void	Добавление узла в дерево
is_int_tree(self, new_node)->bool	Проверка на наличие узла в дереве
get_path(self, node)->list	Получение пути до узла, указанного в параметре

Описание алгоритмов

Поиск в глубину:

Поиск в глубину раскрывает одну из вершин на самом глубоком уровне дерева. Останавливается, когда поиск достигает цели или заходит в тупик (ни одна вершина не может быть раскрыта). В последнем случае выполняется возврат назад и раскрываются вершины на более верхних уровнях.

Поиск в глубину эффективнее, когда задача имеет много решений, так как в этом случае высока вероятность найти решение, исследовав малую часть пространства поиска. Недостаток поиска в глубину – способность углубляться в неверном направлении. Многие задачи имеют очень глубокие или даже бесконечные деревья поиска, в этом случае поиск в глубину никогда не может восстановиться после неудачного выбранного вблизи корня направления.

Пусть N - глубина результата, C - какая-то константа, тогда:

$O(C*N) = O(N)$

Получается порядок временной сложности поиска – O(N). Ёмкостная сложность равна временной O(N).

Алгоритм обхода в глубину:

Обход в глубину реализуется с помощью стека.

Инициализация:

Добавляем исходную вершину в стек.

Множество посещенных вершин visited.

Очередной шаг алгоритма:

Пока стек не пуст выполнять:

1. Достаем вершину из стека и добавляем ее в множество посещенных состояний.

2. Проверяем вершину на конечность, если конечна, то заканчиваем алгоритм, иначе продолжаем

3. Пробегаем по всем вершинам соседей, проверяем их на наличие во множестве посещенных состояний, если нет, то добавляем их в стек.

Иначе путь не найден.

Двунаправленный поиск:

Двунаправленный поиск - усложнённый алгоритм поиска в ширину (или глубину), идея которого заключается в формировании процесса поиска от начальной (прямой поиск) и от конечной вершины (обратный поиск) графа.

Решено было использовать за основу поиск в глубину, так как он наиболее выгоден для задач, с большими деревьями решений.

Пусть N - глубина результата, C - какая-то константа, тогда:

$$O(C*N/2) + O(C*N/2) = O(N/2) + O(N/2)$$

Получается порядок временной сложности поиска – O(N). Ёмкостная сложность равна временной O(2N).

Алгоритм двунаправленного поиска:

Инициализация:

Записываем начальное и конечное состояния в собственные стеки fringe1 и fringe 2 (поиск с двух направлений).

Два множества посещенных состояний visited1 и visited2 (поиск с двух направлений).

Очередной шаг алгоритма:

Пока очереди не пусты выполнять:

1. Добавляем вершину в очередь fringe 1.
2. Если вершина в множестве visited2, то:

   возвращаем путь от начальной вершины к конечной

   Иначе:

   Пробегаемся по соседям вершины и, если их нет в visited 1, то добавляем их в visited1 и в fringe1.

3. Добавляем вершину в очередь fringe 2.
4. Если вершина в множестве visited1, то:

   возвращаем путь от начальной вершины к конечной

Иначе:

Пробегаемся по соседям вершины и, если их нет в visited 2, то добавляем их в visited2 и в fringe2.

Иначе путь не найден.

Результаты работы программы:

Программа на рисунке 3 говорит о том, что мы должны передавать аргументы в командной строке.

```
# kivyfreakt @ hpomen in ~/documents/code/etu_ai/lab1_2 on git:main x [23:13:41]
$ python main.py
usage: main.py [-h] [-v] [-m] [-r] algorithm heuristic
main.py: error: the following arguments are required: algorithm, heuristic
```

Рисунок 3. Попытка запустить программу

Выведем справку с описанием каждого аргумента. Результат представлен на рисунке 4.

```
# kivyfreakt @ hpomen in ~/documents/code/etu_ai/lab1_2 on git:main x [23:14:22] C:2
$ python main.py -h
usage: main.py [-h] [-v] [-m] [-r] algorithm heuristic

Solve 8-puzzle game

positional arguments:
  algorithm        name of the algorithm used (list: dfs, bds)
  heuristic        type of the heuristic (list: position[1], manhatten[2], none)

options:
  -h, --help       show this help message and exit
  -v, --visualize  gui visualisation of puzzle solution
  -m, --manual     step-by-step mode of operation of the program
  -r, --random     generate random states
```

Рисунок 4. Справка аргументов

Запустим поиск в глубину с ключом визуализации. Запуститься окно с визуализацией смены каждого состояния с начального до конечного. Результат представлен на рисунках 5, 6.

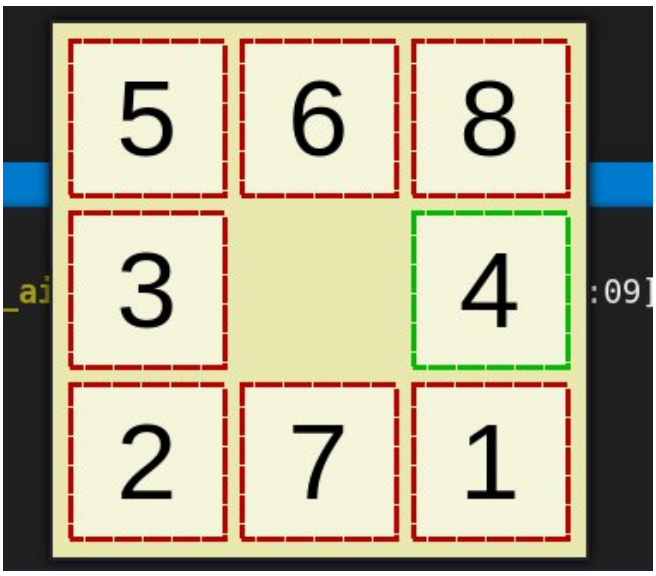


Рисунок 5. Поиск в глубину с ключом визуализации

```
# kivyfreakt @ hpomen in ~/documents/code/etu_ai/lab1_2 on git:main x [23:15:14]
$ python main.py -v dfs 0
 Iteration count: 20939
Nodes: 36635
Time: 172.57552800000002 ms
```

Рисунок 6. Количество итераций, созданных узлов и потраченного времени

Запустим двунаправленный поиск с ключом визуализации. Результат представлен на рисунках 7, 8.

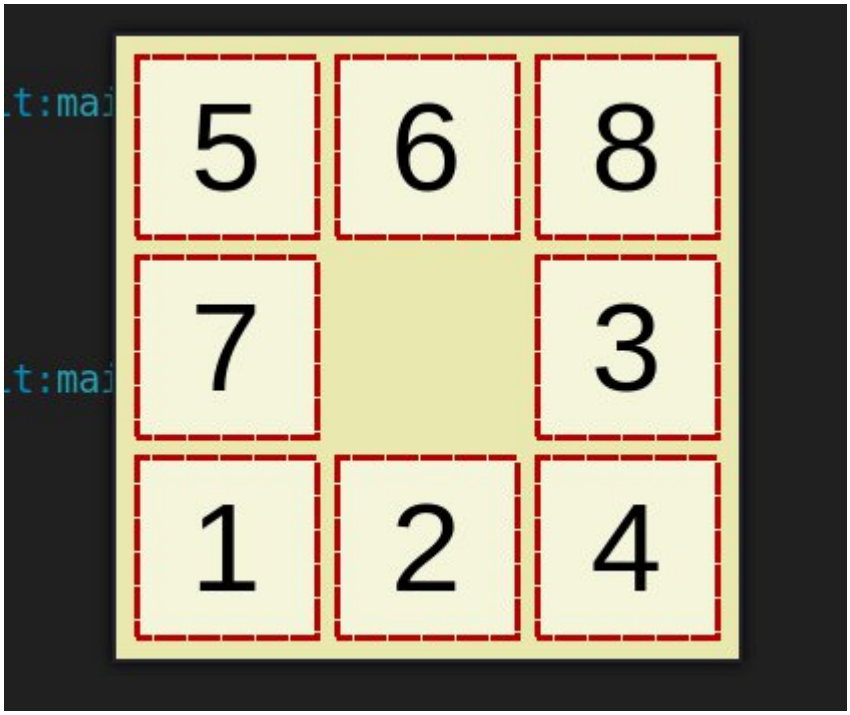
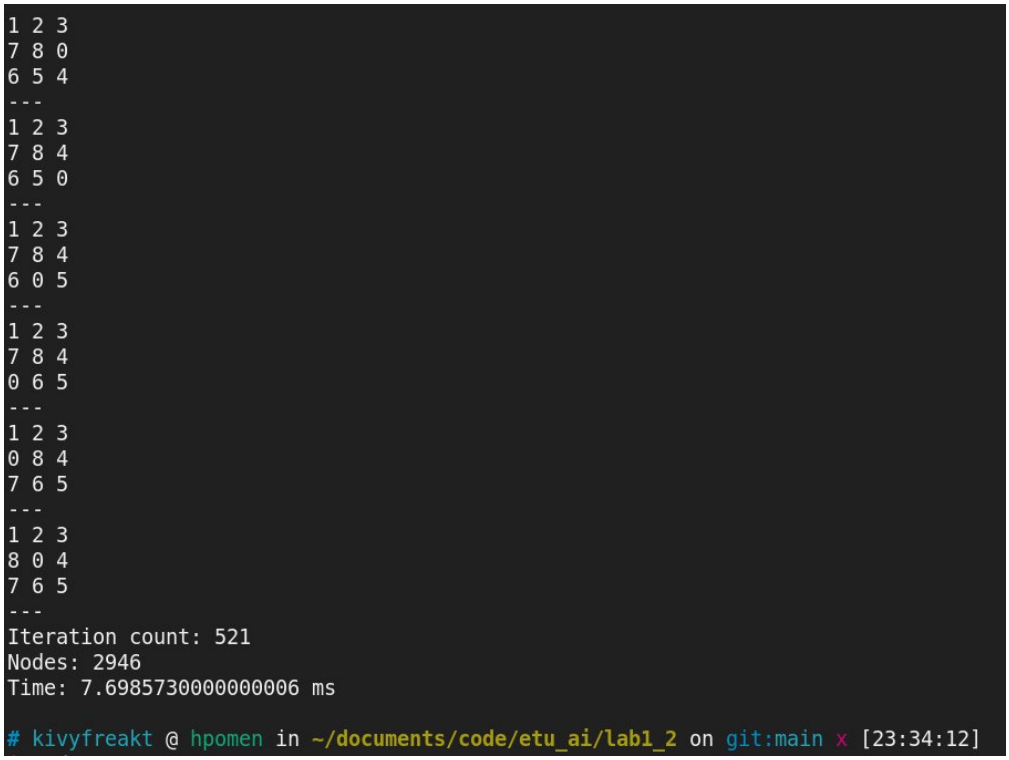


Рисунок 7. Двунаправленный поиск с ключом визуализации

```
# kivyfreakt @ hpomen in ~/documents/code/etu_ai/lab1_2 on git:main x [23:31:55]
$ python main.py -v bds 0
Iteration count: 521
Nodes: 2946
Time: 8.728171999999992 ms
```

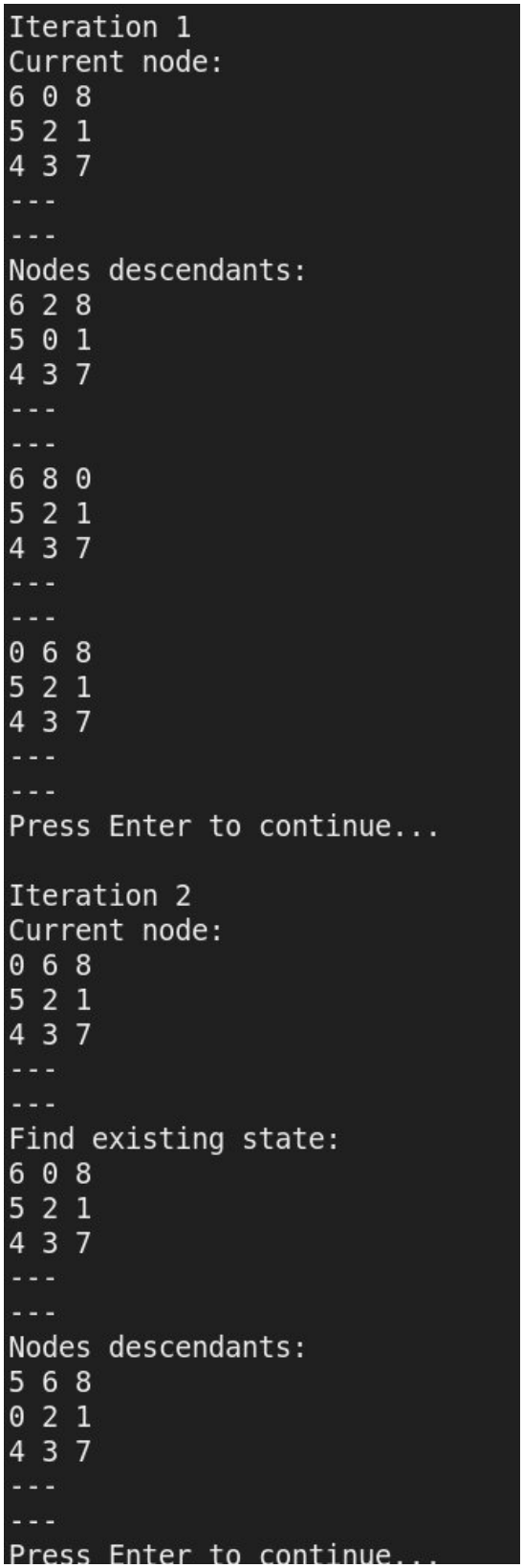
Рисунок 8. Количество итераций, созданных узлов и потраченного времени

Программа поддерживает режим без визуализации игры. Для этого запустим ее без ключа.



В этом случае программа выводит путь в терминале.

Кроме того, программа поддерживает ручной режим. При этом режиме можно отследить каждый шаг работы алгоритма. Проверим это.



Видим, что алгоритм выводит текущее состояние, детей этого узла, отловленные повторные узлы.

После запуска алгоритмов, можем составить сравнительную оценку их работы.

Сравнительные оценки сложности алгоритмов поиска

	Неинформированный поиск	
	Стратегия 1	Стратегия 2
Временная сложность (кол-во шагов)	20939	521
Емкостная сложность (кол-во вершин в дереве поиска)	36635	2946

Вывод

В лабораторной работе изучены основные стратегии неинформированного поиска. Реализованы стратегии: Поиск в глубину и Двунаправленный поиск (реализация через стек), обоснованы теоретические оценки сложности. Проведен эксперимент с получением реальных значений затраченного времени и памяти.

Двунаправленный поиск сработал быстрее и был менее ресурсоёмким, чем поиск в глубину, так как поиск с одной из сторон может найти вершину, которую уже посетил второй и алгоритм на этом уже заканчивается, существенно ускоряя работу алгоритма.

Распределение обязанностей:

Участник	Задачи
Дементьев Д.П.	Оценка порядков сложностей, проведение эксперемента по поиску эксперементальной оценки временной и ёмкостной сложности, составление отчёта
Хамитов А.	Формализация задания, выбор структур данных, выбор пользовательских структур данных, реализация части алгоритма.
Яловега Н.В.	Распределение обязанностей, руководство при реализации стратегий, составление алгоритмов стратегий неинформированного поиска, поиск решения возникших при разработке вопросов, согласование реализации и формализации, реализация визуализации графов

Листинг

main.py

```
import argparse
from time import process_time
from random import shuffle
from solver import common
from solver.search import dfs, bidirectional_search, a_star, heuristic1, heuristic2
from solver.visualizer import visualizer
from solver.tree import Tree, Node


def main():
    ''' Главная функция программы '''
    # парсинг входных значений
    parser = argparse.ArgumentParser(description="Solve 8-puzzle game")
    parser.add_argument("algorithm", type=str,
                        help="name of the algorithm used (list: dfs, bds)")
    parser.add_argument("heuristic", type=str,
                        help="type of the heuristic (list: position[1], manhatten[2], none)")
    parser.add_argument("-v", "--visualize", action='store_true',
                        help="gui visualisation of puzzle solution")
    parser.add_argument("-m", "--manual", action='store_true',
                        help="step-by-step mode of operation of the program")
    parser.add_argument("-r", "--random", action='store_true',
                        help="generate random states")

    args = parser.parse_args()

    # создать
    common.TREE = Tree()
    common.TREE2 = Tree(False)

    # начать отсчет времени
    start_time = process_time()
```

```
# поиск решения
solution = []
iterations = 0

if args.manual:
    common.MANUAL = True

if args.random:
    shuffle(common.INITIAL_STATE)
    shuffle(common.FINISH_STATE)
    print(common.INITIAL_STATE)
    print(common.FINISH_STATE)

if args.algorithm == "dfs":
    if args.heuristic in ('1', 'position'):
        solution, iterations = a_star("dfs", heuristic1)
    elif args.heuristic in ('2', 'manhatten'):
        solution, iterations = a_star("dfs", heuristic2)
    else:
        solution, iterations = dfs(0)
else:
    if args.heuristic in ('1', 'position'):
        solution, iterations = a_star("bds", heuristic1)
    elif args.heuristic in ('2', 'manhatten'):
        solution, iterations = a_star("bds", heuristic2)
    else:
        solution, iterations = bidirectional_search(0)

if solution and iterations:
    solution.reverse()

    # завершить отсчет времени
    finish_time = process_time()

    # визуализация
```

```
        if args.visualize:
            visualizer(solution)
        else:
            for state in solution:
                common.print_state(state)

        # вывод результатов
        print(f"Iteration count: {iterations}")
        print(f"Nodes: {Node.get_nodes_count()}")
        print(f"Time: {(finish_time-start_time)*1000} ms")
    else:
        print("No solution")


if __name__ == '__main__':
    main()
```

solver/common.py

```
MANUAL = None
TREE = None
TREE2 = None
SIZE = 3
INITIAL_STATE = [6, 0, 8,
                 5, 2, 1,
                 4, 3, 7, ]

FINISH_STATE = [1, 2, 3,
                8, 0, 4,
                7, 6, 5, ]



def print_state(state: list):
    ''' Вывод состояния на экран '''
    for i, num in enumerate(state):
        print(num, end=" ")
        if (i + 1) % 3 == 0:
            print("")

    print("---")



def get_initial_state() -> list:
    ''' Получение начального состояния игры (Вариант 4) '''
    return INITIAL_STATE



def get_finish_state() -> list:
    ''' Получение конечного состояния игры (Вариант 4) '''
    return FINISH_STATE
```

solver/tree.py

```
from enum import Enum
from . import common

class Tree:
    ''' Класс представления дерева '''

    def __init__(self, _flag=True):
        self.__hashset = {}  # хеши всех узлов

        if _flag:
            node = Node(common.get_initial_state(), None, None, 0, 0)
        else:
            node = Node(common.get_finish_state(), None, None, 0, 0)

        self.__hashset[node.node_id] = node

    def get_root(self) -> list:
        ''' Получить корень дерева '''
        return [list(self.__hashset.values())[0]]

    def get_node(self, node_id: int):
        ''' Получить узел по идентификатору (хэшу)'''
        return self.__hashset[node_id]

    def add_node(self, new_node):
        ''' Добавить узел в дерево '''
        if not self.is_in_tree(new_node.node_id):
            self.__hashset[new_node.node_id] = new_node

    def is_in_tree(self, node_id: int) -> bool:
        ''' Проверка, есть ли состояние в дереве '''
        return node_id in self.__hashset
```

```
    def get_path(self, node) -> list:
        ''' Получение пути'''
        path = []
        current_node = node

        while current_node.parent_node:
            path.append(current_node.current_state)
            current_node = current_node.parent_node

        path.append(current_node.current_state)

        return path

class Action(Enum):
    ''' Перечисление действий над полем '''
    UP = 1
    DOWN = 2
    RIGHT = 3
    LEFT = 4

class Node:
    ''' Класс представления узла '''
    __nodes_count = 0

    def __init__(self, state, parent, action, cost, depth):
        # Состояние в пространстве состояний, которому соответствует данный узел
        self.current_state = state
        self.parent_node = parent  # Указатель на родительский узел
      # Действие, которое было применено к родительскому узлу для формирования данного
узла
        self.previous_action = action
        # Стоимость пути от начального состояния до данного узла g(n)
        self.path_cost = cost
        # Количество этапов пути от начального состояния (глубина)
        self.depth = depth
```

```
    self.node_id = hash(tuple(state))  # уникальный идентификатор узла
    Node.__nodes_count += 1


@classmethod
def get_nodes_count(cls) -> int:
    ''' Статический метод класса, возвращающий количество узлов '''
    return cls.__nodes_count + 1
```

solver/search.py

```
from . import common
from .tree import Node, Action


def check_final(current_state: list) -> bool:
    ''' Проверка, является ли данное состояние конечным '''
    return current_state == common.get_finish_state()


def clone_and_swap(state, i, j):
    ''' Клонирование состояния и перемещение пустой ячейки '''
    clone = list(state)
    clone[i], clone[j] = clone[j], clone[i]
    return clone


def get_new_states(current_state: list) -> dict:
    ''' Получение новых состояний поля '''
    new_states = {}
    pos = current_state.index(0)

    # up
    if pos - 3 >= 0:
        new_states[Action.UP] = clone_and_swap(current_state, pos, pos-3)

    # down
    if pos + 3 < 9:
        new_states[Action.DOWN] = clone_and_swap(current_state, pos, pos+3)

    # right
    if pos % 3 + 1 < 3:
        new_states[Action.RIGHT] = clone_and_swap(current_state, pos, pos+1)
```

```
    # left
    if pos % 3 > 0:
        new_states[Action.LEFT] = clone_and_swap(current_state, pos, pos-1)

    return new_states


def heuristic1(state):
    ''' не на своих местах '''
    count = 0
    for i, num in enumerate(state):
        if common.get_finish_state()[i] == num:
            count += 1

    return count


def heuristic2(state):
    ''' Manhatten '''
    res = 0
    for i in range(common.SIZE**2):
        if state[i] != 0 and state[i] != common.get_finish_state()[i]:
            c_i = common.get_finish_state().index(state[i])
            diff_y = (i // common.SIZE) - (c_i // common.SIZE)
            diff_x = (i % common.SIZE) - (c_i % common.SIZE)
            res += abs(diff_y) + abs(diff_x)
    return res


def dfs(heuristic):
    ''' Поиск в глубину '''
    visited = set()
    stack = []
    iterations = 0
```

```
stack += common.TREE.get_root()

while stack:
    current_node = stack.pop()
    visited.add(current_node.node_id)

    iterations += 1

    if common.MANUAL:
        print(f"Iteration {iterations}")
        print("Current node: ")
        common.print_state(current_node.current_state)
        print("---")

    if check_final(current_node.current_state):
        if common.MANUAL:
            print("SOLUTION FIND")
            common.print_state(current_node.current_state)
            print("---")

        return common.TREE.get_path(current_node), iterations

    new_states = get_new_states(current_node.current_state)
    if not isinstance(heuristic, int):
        new_states = dict(
            sorted(new_states.items(), key=lambda item: heuristic(item[1])))

    neighbors = []
    level = current_node.depth
    for action, new_state in new_states.items():
        new_state_hash = hash(tuple(new_state))
        if common.TREE.is_in_tree(new_state_hash):
            if common.MANUAL:
                print("Find existing state: ")
                common.print_state(new_state)
```

```
                print("---")
            continue

        new_node = Node(new_state, current_node,
                        action, level + 1, level + 1)
        neighbors.append(new_node)
        common.TREE.add_node(new_node)

    if common.MANUAL:
        print("Nodes descendants: ")
    for next_node in neighbors:
        if common.MANUAL:
            common.print_state(next_node.current_state)
            print("---")
        if next_node.node_id not in visited:
            stack.append(next_node)

    if common.MANUAL:
        print("Press Enter to continue... ")
        input()

return None, None


def bidirectional_search(heuristic):
    ''' Двунаправленный поиск '''

    iterations = 0

    start = common.TREE.get_root()
    goal = common.TREE2.get_root()

    fringe1 = [] + start
    fringe2 = [] + goal
```

```
visited1 = set(start)
visited2 = set(goal)

while fringe1 or fringe2:
    iterations += 1
    if fringe1:
        current1 = fringe1.pop()

        if common.MANUAL:
            print(f"Iteration {iterations}")
            print("First subtree current node: ")
            common.print_state(current1.current_state)
            print("---")

        if current1.node_id in visited2:
            # print("VARIANT 1")
            if common.MANUAL:
                print("MEET POINT FIND")
                common.print_state(current1.current_state)
                print("---")

            meet = common.TREE2.get_node(current1.node_id)
            path2 = common.TREE2.get_path(meet)
            del path2[0]
            path = list(reversed(path2)) + common.TREE.get_path(current1)
            return path, iterations

        new_states = get_new_states(current1.current_state)
        if not isinstance(heuristic, int):
            new_states = dict(
                sorted(new_states.items(), key=lambda item: heuristic(item[1])))

        neighbors1 = []
        for action, node_state in new_states.items():
            node = Node(node_state, current1, action,
```

```
                current1.depth + 1, current1.depth + 1)

            neighbors1.append(node)

        if common.MANUAL:
            print("Nodes descendants: ")
        for next_node in neighbors1:
            if common.MANUAL:
                common.print_state(next_node.current_state)
                print("---")
            if next_node.node_id not in visited1:
                visited1.add(next_node.node_id)
                fringe1.append(next_node)
                common.TREE.add_node(next_node)

    if fringe2:
        current2 = fringe2.pop()

        if common.MANUAL:
            print(f"Iteration {iterations}")
            print("Second subtree current node: ")
            common.print_state(current2.current_state)
            print("---")

        if current2.node_id in visited1:
            if common.MANUAL:
                print("MEET POINT FIND")
                common.print_state(current2.current_state)
                print("---")
            # print("VARIANT 2")
            meet = common.TREE.get_node(current2.node_id)
            path2 = common.TREE2.get_path(meet)
            del path2[0]
            path = list(reversed(path2)) + common.TREE.get_path(current2)
            return list(reversed(path)), iterations
```

```
        new_states = get_new_states(current2.current_state)
        if not isinstance(heuristic, int):
            new_states = dict(
                sorted(new_states.items(), key=lambda item: heuristic(item[1])))

        neighbors2 = []
        for action, node_state in new_states.items():
            node = Node(node_state, current2, action,
                        current2.depth + 1, current2.depth + 1)
            neighbors2.append(node)

        if common.MANUAL:
            print("Nodes descendants: ")
        for next_node in neighbors2:
            if common.MANUAL:
                common.print_state(next_node.current_state)
                print("---")
            if next_node.node_id not in visited2:
                visited2.add(next_node.node_id)
                fringe2.append(next_node)
                common.TREE2.add_node(next_node)

    if common.MANUAL:
        print("Press Enter to continue... ")
        input()

return None, None
```

solver/visualization.py

```
import sys
from tkinter import Tk, Canvas

GUI_FONT = ("Arial", 32)
GUI_BOX_SIZE = 100
GUI_BOX_SPACING = 10
GUI_BOX_BORDER_WIDTH = 3
GUI_DELAY = 200
PUZZLE_SIZE = 3

GUI_COLOR_1 = "#f5f5dc"
GUI_COLOR_2 = "#e9e9af"
GUI_COLOR_3 = "#dddd88"

GUI_OUTLINE_1 = "#ff0000"
GUI_OUTLINE_2 = "#00ff00"
GUI_OUTLINE_3 = "#0000ff"

GUI_COLOR_GREEN = "#00bb00"
GUI_COLOR_RED = "#bb0000"
GUI_COLOR_BLACK = "#000000"

GUI_DASH = (5, 4, 5, 3)

# MANUAL_MODE = False


def gui_replay(master, frame, canvas, item_matrix, solution):
    ''' Проигрывание анимации'''
    numbers = solution[frame]
    next_zero = None
    color_this = None
    if frame + 1 < len(solution):
        next_zero = solution[frame + 1].index(0)
```

```
    color_this = solution[frame][next_zero]

for row in range(PUZZLE_SIZE):
    for col in range(PUZZLE_SIZE):
        num = numbers[row + PUZZLE_SIZE * col]
        border_color = None
        if num == solution[-1][row + PUZZLE_SIZE * col]:
            border_color = GUI_COLOR_GREEN
        else:
            border_color = GUI_COLOR_RED

        if num == 0:
            canvas.itemconfig(
                item_matrix[row][col][0],
                fill=GUI_COLOR_2,
                outline=GUI_COLOR_2,
                width=GUI_BOX_BORDER_WIDTH,
            )
        elif num == color_this:
            canvas.itemconfig(
                item_matrix[row][col][0],
                fill=GUI_COLOR_1,
                outline=border_color,
                width=GUI_BOX_BORDER_WIDTH,
            )
        else:
            canvas.itemconfig(
                item_matrix[row][col][0],
                fill=GUI_COLOR_1,
                outline=border_color,
                width=GUI_BOX_BORDER_WIDTH,
            )

        num_string = str(num)
        if not num:
```

```
            num_string = ""
        canvas.itemconfig(item_matrix[row][col][1], text=num_string)

    frame += 1
    if frame >= len(solution):
        frame = 0
    canvas.update()

    if frame != 0:
        master.after(
            GUI_DELAY, gui_replay, master, frame, canvas, item_matrix, solution)


def gui_close(_event):
    ''' Закрытие приложения '''
    sys.exit(0)


def gui_item_matrix(canvas):
    ''' Составление матрицы '''
    item_matrix = [
        [[None, None] for col in range(PUZZLE_SIZE)] for row in range(PUZZLE_SIZE)
    ]
    for row in range(PUZZLE_SIZE):
        for col in range(PUZZLE_SIZE):

            row0 = row * GUI_BOX_SIZE + GUI_BOX_SPACING
            col0 = col * GUI_BOX_SIZE + GUI_BOX_SPACING
            row1 = row0 + GUI_BOX_SIZE - GUI_BOX_SPACING
            col1 = col0 + GUI_BOX_SIZE - GUI_BOX_SPACING
            item_matrix[row][col][0] = canvas.create_rectangle(
                row0, col0, row1, col1, dash=GUI_DASH, fill=GUI_COLOR_1
            )

            row_t = row0 + ((GUI_BOX_SIZE - GUI_BOX_SPACING) / 2)
```

```
            col_t = col0 + ((GUI_BOX_SIZE - GUI_BOX_SPACING) / 2)
            item_matrix[row][col][1] = canvas.create_text(
                (row_t, col_t), font=GUI_FONT, text="")

    return item_matrix


def visualizer(solution):
    ''' Функция визуализации решения '''
    master = Tk()
    canvas_width = (GUI_BOX_SIZE * PUZZLE_SIZE) + GUI_BOX_SPACING
    canvas_height = (GUI_BOX_SIZE * PUZZLE_SIZE) + GUI_BOX_SPACING
    canvas = Canvas(
        master,
        width=canvas_width + 1,
        height=canvas_height + 1,
        bg=GUI_COLOR_2,
        borderwidth=0,
        highlightthickness=0,
    )
    canvas.pack()
    # MANUAL_MODE = manual_mode
    item_matrix = gui_item_matrix(canvas)
    master.bind("<Escape>", gui_close)
    master.after(0, gui_replay, master, 0, canvas,
                 item_matrix, solution)
    master.mainloop()
```