**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.

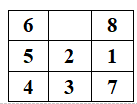


Рисунок 1. Начальное состояние

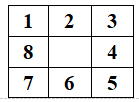


Рисунок 2. Конечное состояние

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.

1. Добавляем вершину в очередь fringe 2.
2. 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.

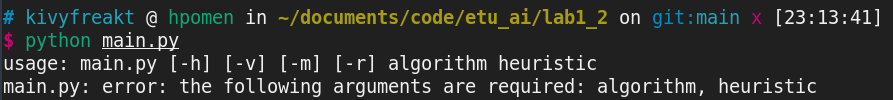


Рисунок 3. Попытка запустить программу

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

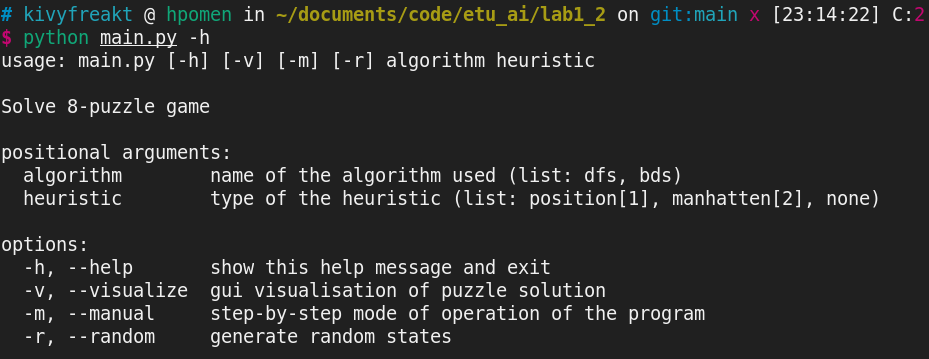


Рисунок 4. Справка аргументов

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

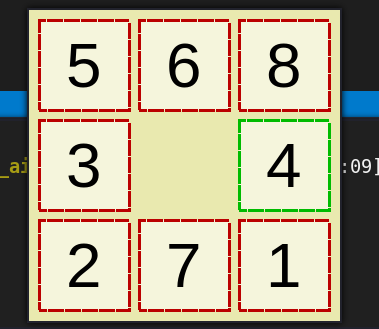


Рисунок 5. Поиск в глубину с ключом визуализации

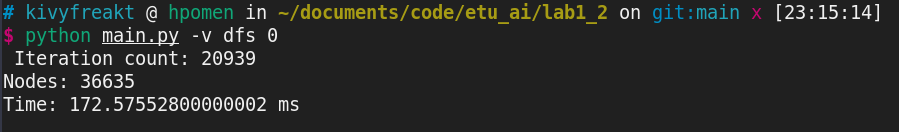


Рисунок 6. Количество итераций, созданных узлов и потраченного времени

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

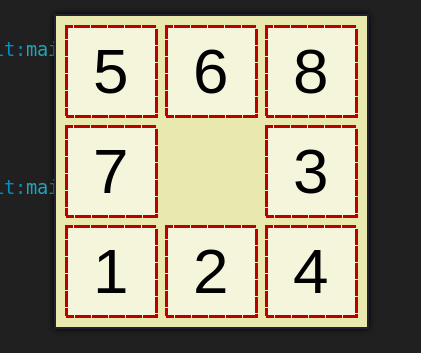


Рисунок 7. Двунаправленный поиск с ключом визуализации

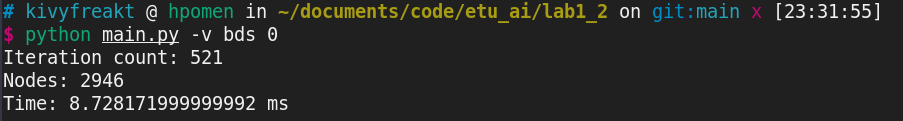
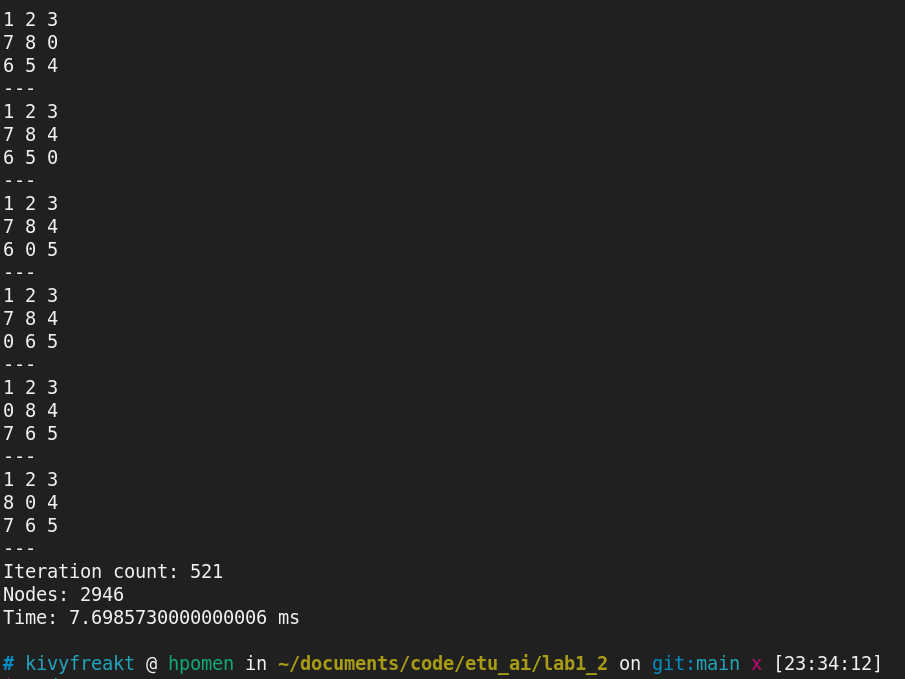


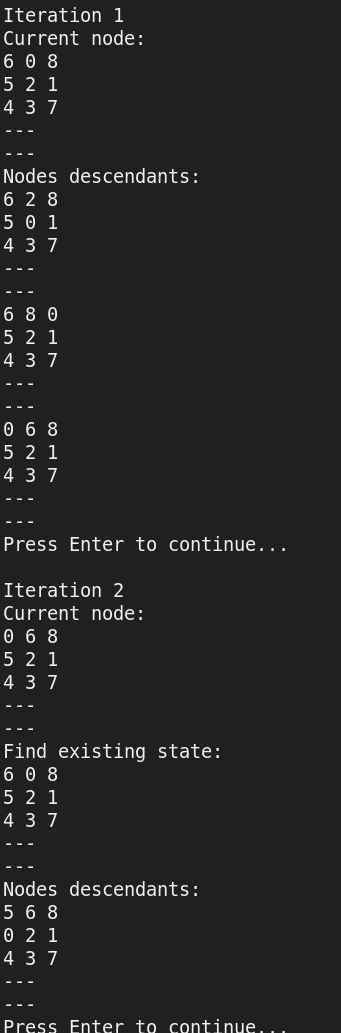
Рисунок 8. Количество итераций, созданных узлов и потраченного времени

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()