|  |  |
| --- | --- |
| Picture 1 | **МИНОБРНАУКИ РОССИИ**  **федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  «Санкт-Петербургский государственный морской технический университет» (СПбГМТУ) |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Факультет цифровых промышленных технологий

Направление подготовки 09.03.01.03 "Интеллектуальные технологии киберфизических систем"

**«Контрольный Проект»**

Студент 1 курса группы 20121

Очного отделения

Руденко В. С.

Проверил:

2024

# Оглавление

[Оглавление 2](#_Toc170786446)

[Цель работы 3](#_Toc170786447)

[Результаты работы 3](#_Toc170786448)

[Реализация программы с использованием функционального программирования (прошлый семестр) 3](#_Toc170786449)

[Ход работы 3](#_Toc170786450)

[Демонстрация работы программы 12](#_Toc170786451)

[Листинг 13](#_Toc170786452)

[Реализация программы с использованием ООП языка Python 19](#_Toc170786453)

[Ход работы 19](#_Toc170786454)

[Демонстрация работы программы 34](#_Toc170786455)

[Листинг 37](#_Toc170786456)

[Заключение 60](#_Toc170786457)

[Список используемых источников 60](#_Toc170786458)

# Цель работы

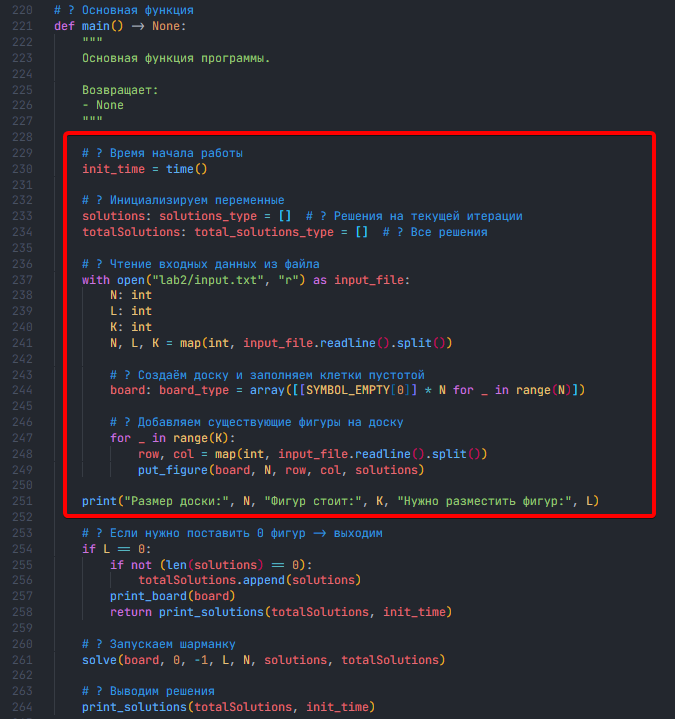
Целью данной лабораторной работы является разработка и оптимизация алгоритма для размещения дополнительных фигур на шахматной доске таким образом, чтобы никакая фигура не находилась под боем другой. А также создание GUI приложения.

# Результаты работы

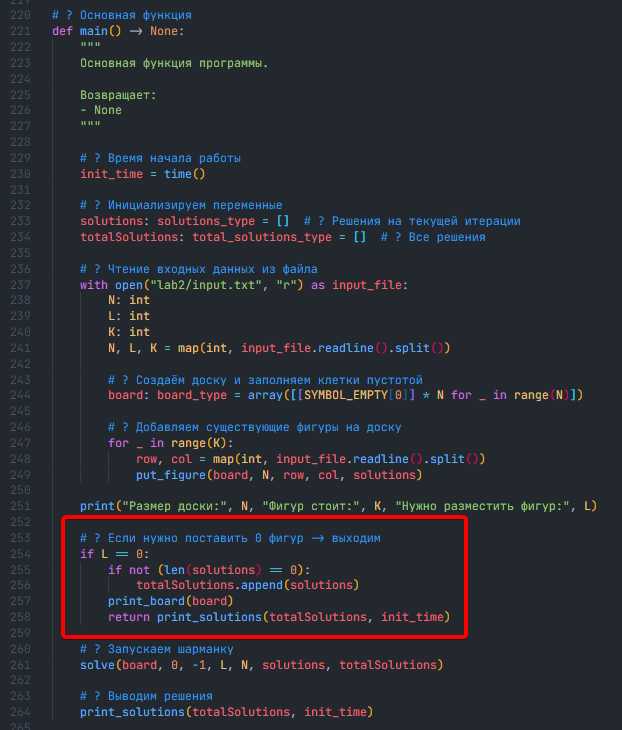
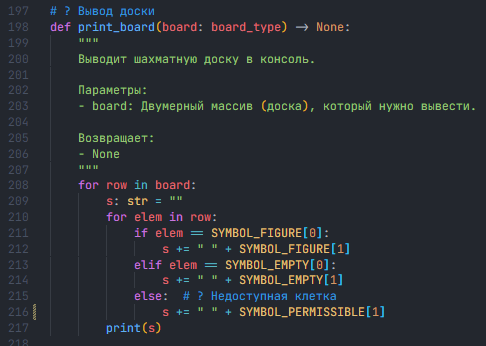
## Реализация программы с использованием функционального программирования (прошлый семестр)

### Ход работы

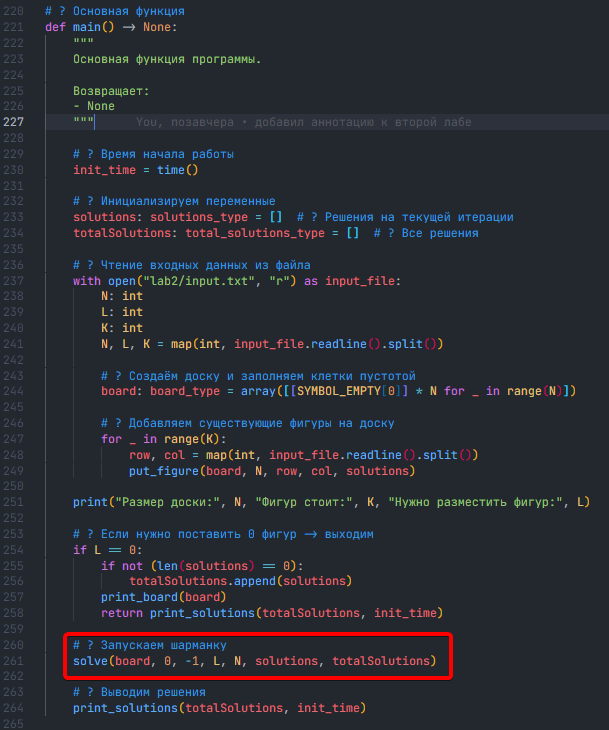
* 1. Функция main инициализирует начальное значение времени, считывает данные с файла и располагает на доске начальные фигуры. Затем выводит полученные данные в консоль



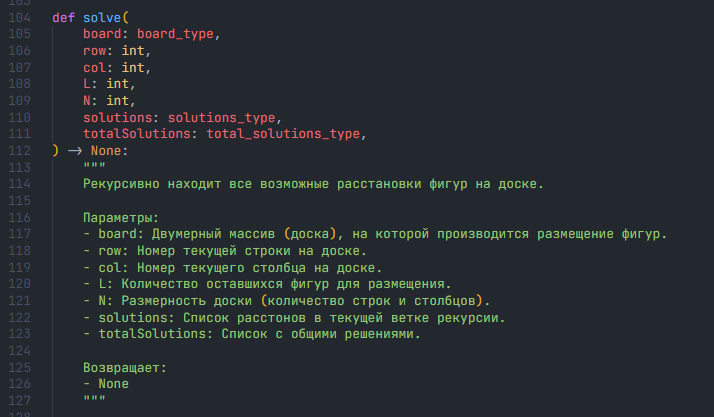
* 1. Если нужно расставить 0 фигур, то выводим данные в файл, а также выводим состояние доски с начальными фигурами

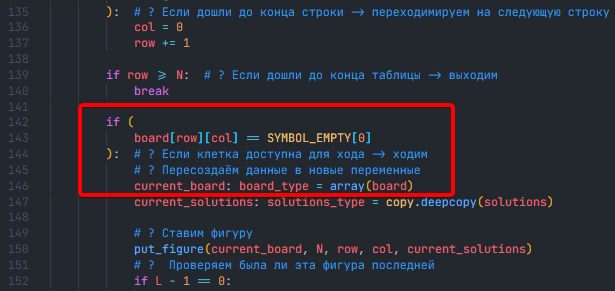
 

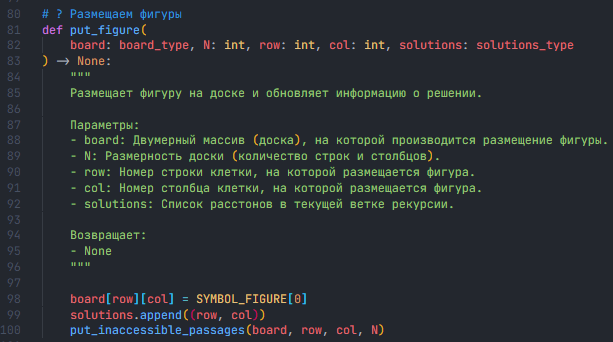
3. Запускаем рекурсивную функцию solve



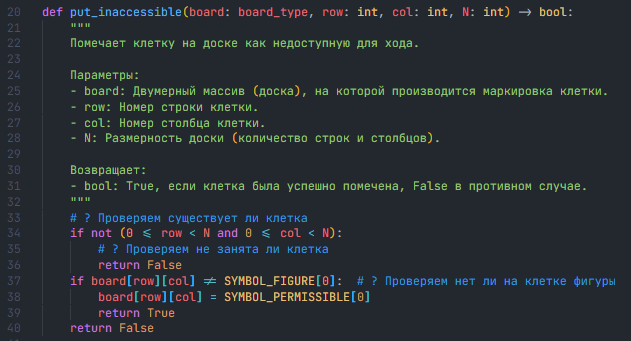
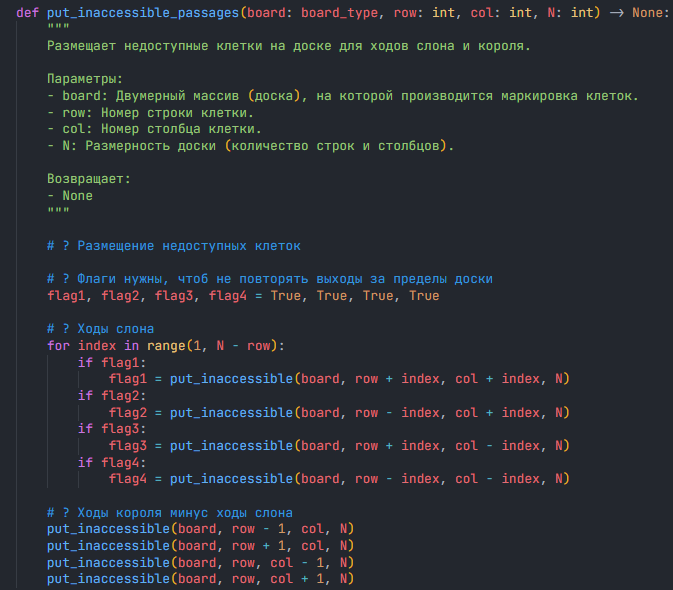
1. Функция solve принимает доску, номер ряда, номер столбца, количество оставшихся фигур для размещения, размерность доски, Список расстонов в текущей ветке рекурсии, список с общими решениями. Возвращает None.

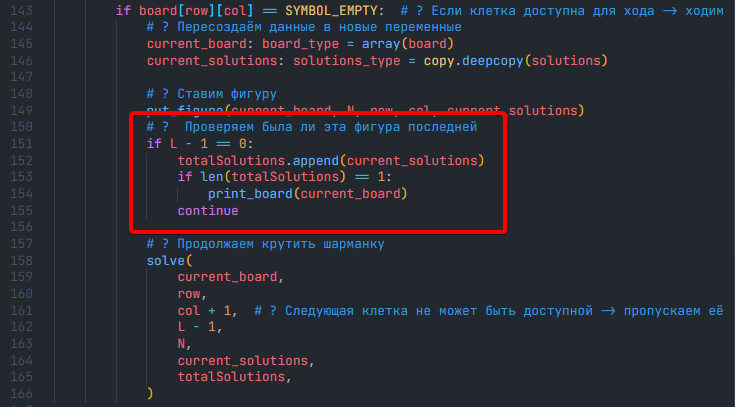


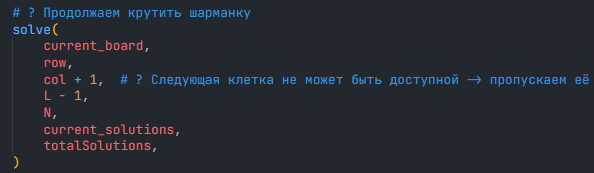
1. Внутри функции через цикл while который, пробегает по каждой клетке доски, и, если можем на эту клетку поставить нашу фигуру, то ставим. Проверка, можем ли мы походить на текущую клетку есть проверка: 
2. В случае, если мы можем поставить фигуру, копируем доску и список решений и вызываем функцию put\_figure, которая принимает: ссылку на список доски, размерность доски, номер ряда и столбик, на который нам нужно поставить фигуру и список решений, чтоб впоследствии добавить в список решений координаты



1. Функция put\_figure вызывает функцию put\_inaccessible\_passages, которая помечает клетки недоступными к ходу



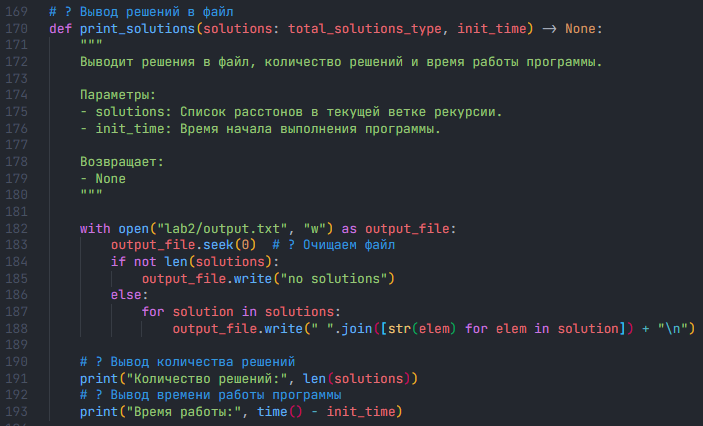
9. Проверяем, поставили ли мы последнюю фигуру, если да, то заносим данные в список с общими решениями, который мы изначально передавали ссылкой. В случае, если решение первое, то выводим его 

10. Если фигура была не последней, то вызываем функцию solve, ументшив количество оставшихся фигур.

11. После того, как все рекурсии отработают мы получим список всех решений в totalSolutions. Нам остаётся только вывести данные. Запускаем функцию print\_solutions в main



Функция делает вывод всех решений в файл output.txt, а также выводит некоторые данные в консоль:



### Демонстрация работы программы

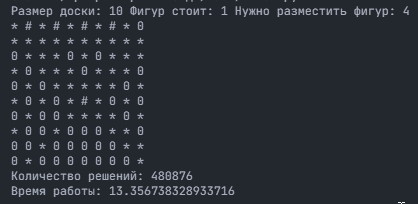
Набор данных №1

Файл input.txt

10 4 1

5 5

Консоль:



Набор данных №2

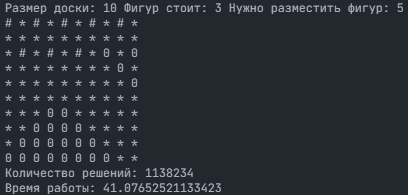
Файл input.txt

10 5 3

0 0

0 2

0 4

Консоль:  


### Листинг

index.py

# ? Импорт пакетов

from numpy import **array**

from time import **time**

import copy

# ? Импорт типов

from my\_types import (

    symbol\_type,

    board\_type,

    solutions\_type,

    total\_solutions\_type,

)

# ? Константы

SYMBOL\_FIGURE: symbol\_type = [1, "#"]

SYMBOL\_PERMISSIBLE: symbol\_type = [2, "\*"]

SYMBOL\_EMPTY: symbol\_type = [0, "0"]

def **put\_inaccessible**(board: board\_type, row: int, col: int, N: int) -> bool:

    """

    Помечает клетку на доске как недоступную для хода.

    Параметры:

    - board: Двумерный массив (доска), на которой производится маркировка клетки.

    - row: Номер строки клетки.

    - col: Номер столбца клетки.

    - N: Размерность доски (количество строк и столбцов).

    Возвращает:

    - bool: True, если клетка была успешно помечена, False в противном случае.

    """

    # ? Проверяем существует ли клетка

    if not (0 <= row < N and 0 <= col < N):

        # ? Проверяем не занята ли клетка

        return False

    if board[row][col] != SYMBOL\_FIGURE[0]:  # ? Проверяем нет ли на клетке фигуры

        board[row][col] = SYMBOL\_PERMISSIBLE[0]

        return True

    return False

def **put\_inaccessible\_passages**(board: board\_type, row: int, col: int, N: int) -> None:

    """

    Размещает недоступные клетки на доске для ходов слона и короля.

    Параметры:

    - board: Двумерный массив (доска), на которой производится маркировка клеток.

    - row: Номер строки клетки.

    - col: Номер столбца клетки.

    - N: Размерность доски (количество строк и столбцов).

    Возвращает:

    - None

    """

    # ? Размещение недоступных клеток

    # ? Флаги нужны, чтоб не повторять выходы за пределы доски

    flag1, flag2, flag3, flag4 = True, True, True, True

    # ? Ходы слона

    for index in range(1, N):

        if flag1:

            flag1 = **put\_inaccessible**(board, row + index, col + index, N)

        if flag2:

            flag2 = **put\_inaccessible**(board, row - index, col + index, N)

        if flag3:

            flag3 = **put\_inaccessible**(board, row + index, col - index, N)

        if flag4:

            flag4 = **put\_inaccessible**(board, row - index, col - index, N)

    # ? Ходы короля минус ходы слона

**put\_inaccessible**(board, row - 1, col, N)

**put\_inaccessible**(board, row + 1, col, N)

**put\_inaccessible**(board, row, col - 1, N)

**put\_inaccessible**(board, row, col + 1, N)

# ? Размещаем фигуры

def **put\_figure**(

    board: board\_type, N: int, row: int, col: int, solutions: solutions\_type

) -> None:

    """

    Размещает фигуру на доске и обновляет информацию о решении.

    Параметры:

    - board: Двумерный массив (доска), на которой производится размещение фигуры.

    - N: Размерность доски (количество строк и столбцов).

    - row: Номер строки клетки, на которой размещается фигура.

    - col: Номер столбца клетки, на которой размещается фигура.

    - solutions: Список расстонов в текущей ветке рекурсии.

    Возвращает:

    - None

    """

    board[row][col] = SYMBOL\_FIGURE[0]

    solutions.**append**((row, col))

**put\_inaccessible\_passages**(board, row, col, N)

def **solve**(

    board: board\_type,

    row: int,

    col: int,

    L: int,

    N: int,

    solutions: solutions\_type,

    totalSolutions: total\_solutions\_type,

) -> None:

    """

    Рекурсивно находит все возможные расстановки фигур на доске.

    Параметры:

    - board: Двумерный массив (доска), на которой производится размещение фигур.

    - row: Номер текущей строки на доске.

    - col: Номер текущего столбца на доске.

    - L: Количество оставшихся фигур для размещения.

    - N: Размерность доски (количество строк и столбцов).

    - solutions: Список расстановки в текущей ветке рекурсии.

    - totalSolutions: Список с общими решениями.

    Возвращает:

    - None

    """

    # ? Перебираем возможные ходы с текущей точки

    while True:

        # ? Идём на следующую клетку

        col += 1

        if col >= N:  # ? Если дошли до конца строки -> переходим на на следующую строку

            col = 0

            row += 1

        if row >= N:  # ? Если дошли до конца таблицы -> выходим

            break

        if (

            board[row][col] == SYMBOL\_EMPTY[0]

        ):  # ? Если клетка доступна для хода -> ходим

            # ? Пересоздаём данные в новые переменные

            current\_board: board\_type = **array**(board)

            current\_solutions: solutions\_type = copy.**deepcopy**(solutions)

            # ? Ставим фигуру

**put\_figure**(current\_board, N, row, col, current\_solutions)

            # ?  Проверяем была ли эта фигура последней

            if L - 1 == 0:

                totalSolutions.**append**(current\_solutions)

                if **len**(totalSolutions) == 1:

**print\_board**(current\_board)

                continue

            # ? Продолжаем крутить шарманку

**solve**(

                current\_board,

                row,

                col + 1,  # ? Cледующая клетка не может быть доступной -> пропускаем её

                L - 1,

                N,

                current\_solutions,

                totalSolutions,

            )

# ? Вывод решений в файл

def **print\_solutions**(solutions: total\_solutions\_type, init\_time) -> None:

    """

    Выводит решения в файл, количество решений и время работы программы.

    Параметры:

    - solutions: Список расстонов в текущей ветке рекурсии.

    - init\_time: Время начала выполнения программы.

    Возвращает:

    - None

    """

    with **open**("lab2/output.txt", "w") as output\_file:

        output\_file.**seek**(0)  # ? Очищаем файл

        if not **len**(solutions):

            output\_file.**write**("no solutions")

        else:

            for solution in solutions:

                output\_file.**write**(" ".**join**([str(elem) for elem in solution]) + "\n")

    # ? Вывод количества решений

**print**("Количество решений:", **len**(solutions))

    # ? Вывод времени работы программы

**print**("Время работы:", **time**() - init\_time)

# ? Вывод доски

def **print\_board**(board: board\_type) -> None:

    """

    Выводит шахматную доску в консоль.

    Параметры:

    - board: Двумерный массив (доска), который нужно вывести.

    Возвращает:

    - None

    """

    for row in board:

        s: str = ""

        for elem in row:

            if elem == SYMBOL\_FIGURE[0]:

                s += " " + SYMBOL\_FIGURE[1]

            elif elem == SYMBOL\_EMPTY[0]:

                s += " " + SYMBOL\_EMPTY[1]

            else:  # ? Недоступная клетка

                s += " " + SYMBOL\_PERMISSIBLE[1]

**print**(s)

# ? Основная функция

def **main**() -> None:

    """

    Основная функция программы.

    Возвращает:

    - None

    """

    # ? Время начала работы

    init\_time = **time**()

    # ? Инициализируем переменные

    solutions: solutions\_type = []  # ? Решения на текущей итерации

    totalSolutions: total\_solutions\_type = []  # ? Все решения

    # ? Чтение входных данных из файла

    with **open**("lab2/input.txt", "r") as input\_file:

        N: int

        L: int

        K: int

        N, L, K = map(int, input\_file.**readline**().**split**())

        # ? Создаём доску и заполняем клетки пустотой

        board: board\_type = **array**([[SYMBOL\_EMPTY[0]] \* N for \_ in range(N)])

        # ? Добавляем существующие фигуры на доску

        for \_ in range(K):

            row, col = map(int, input\_file.**readline**().**split**())

**put\_figure**(board, N, row, col, solutions)

**print**("Размер доски:", N, "Фигур стоит:", K, "Нужно разместить фигур:", L)

    # ? Если нужно поставить 0 фигур -> выходим

    if L == 0:

        if not (**len**(solutions) == 0):

            totalSolutions.**append**(solutions)

**print\_board**(board)

        return **print\_solutions**(totalSolutions, init\_time)

    # ? Запускаем шарманку

**solve**(board, 0, -1, L, N, solutions, totalSolutions)

    # ? Выводим решения

**print\_solutions**(totalSolutions, init\_time)

if \_\_name\_\_ == "\_\_main\_\_":

**main**()

my\_types.py

from typing import Tuple

from numpy import ndarray

# ? Создаём типы

symbol\_type = list

board\_type = ndarray

solutions\_type = list[Tuple[int, int]]

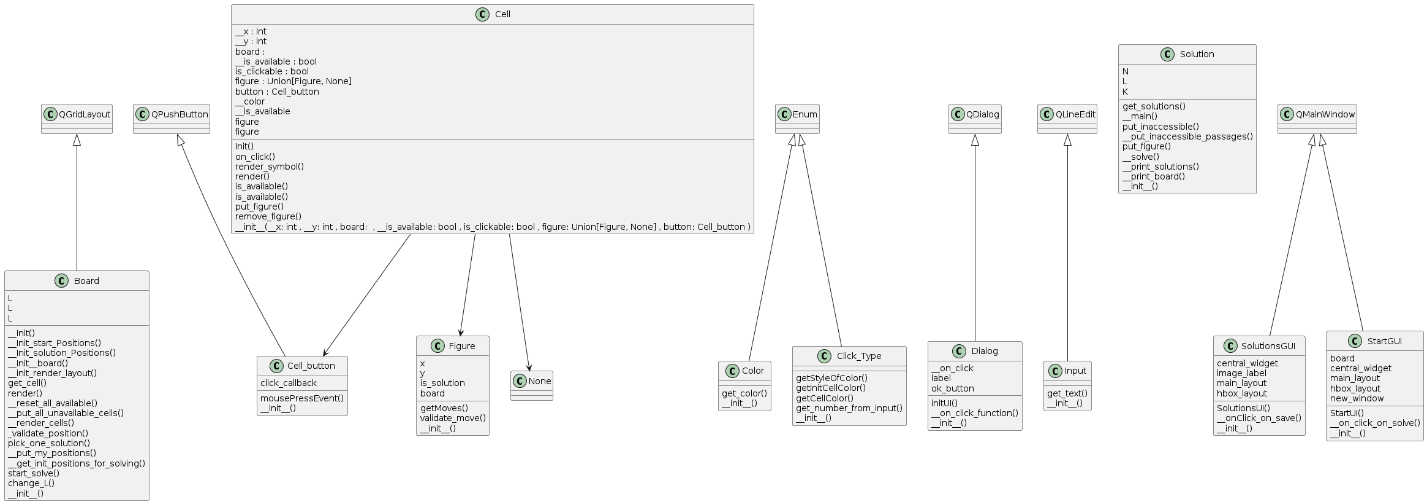
total\_solutions\_type = list[solutions\_type]

input.txt

8 4 0

## Реализация программы с использованием ООП языка Python

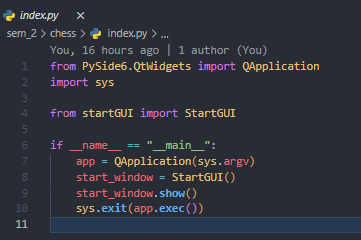
### Ход работы

UML диаграмма:

#### index.py

Создаём наше приложение в index.py

В нём вы показываем стартовое окно



В стартовом окне происходит инициализация UI, который содержит:

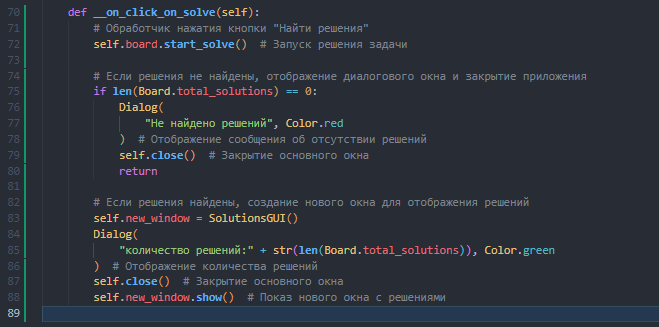
Доску с фигурами, кнопку “найти решения”, поле ввода необходимого количества фигур к расстановке при подсчёте

#### startGUI.py



При нажатии на кнопку “найти решение”, начинается анализ решения (вызывается метод `start\_solve` класса `Board`)

Далее следуют проверки количества найденных решений и оповещение. Если решений не было найдено, то программа закрывается



#### Board.py

Инициализация экземпляра класса происходит в несколько этапов:



1. Сначала идёт инициализация доски
2. Затем заполнение стартовыми позициями, если они есть
3. Заполнение позиций решения, если оно есть
4. Рендеринг всех клеток
5. Добавление этих клеток в grid сетку

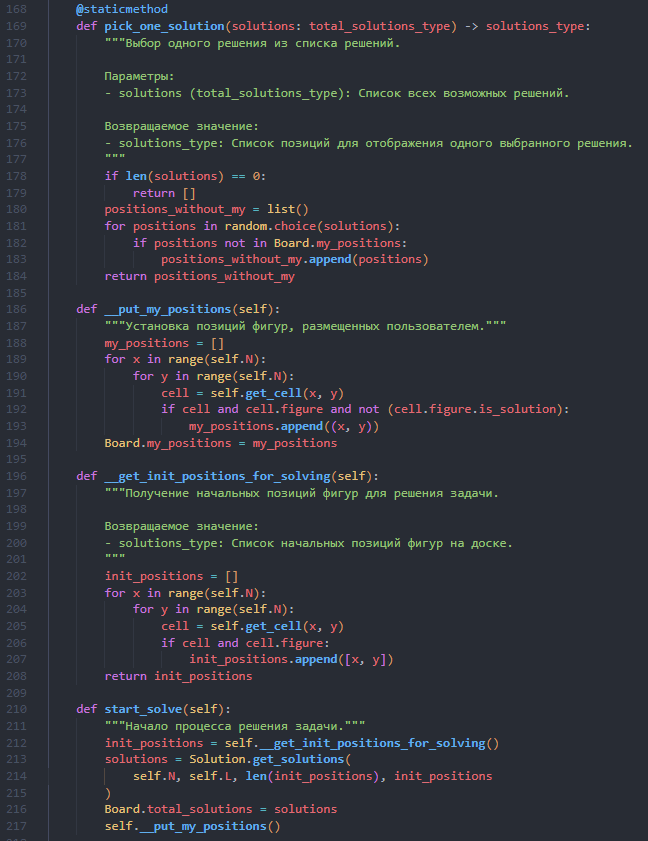
Рендеринг всех клеток так же происходит в несколько этапов:



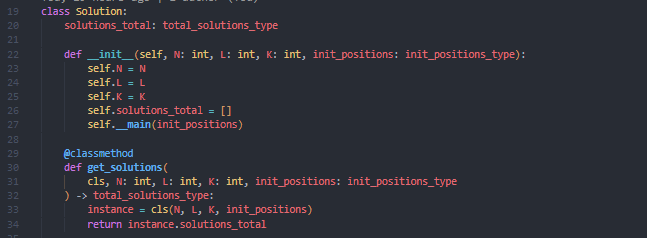
1. Сначала все клетки сбрасывают значение is\_available, которое отвечает за то, доступна ли клетка к ходу
2. Затем просчитывает все недопустимые клетки исходя из стоящих фигур на доске
3. В конце вызывает метод `render` у каждой клетки.

Этот метод изменяет внешний вид клетки

Функция, которая вычисляет решения, вызывает метод get\_solutions у класса Solution, куда аргументами передаёт данные о доске и о позициях фигур на ней

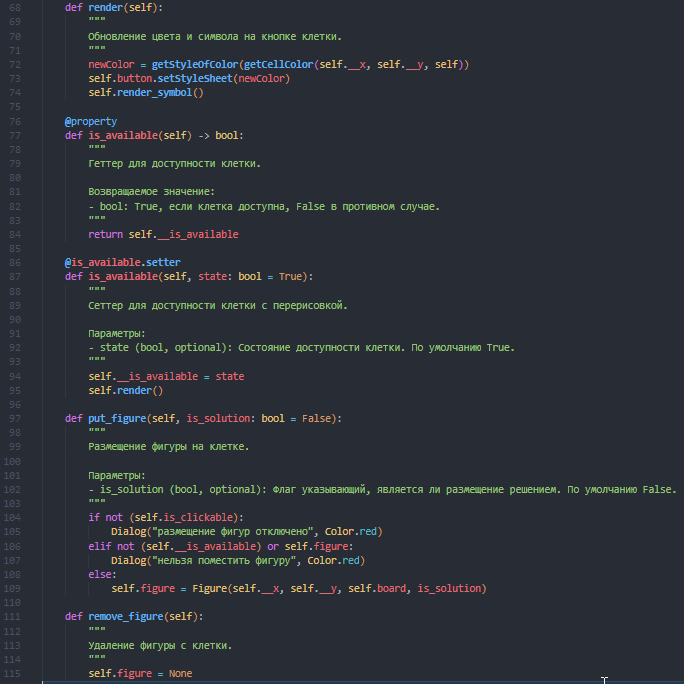


#### Solution.py

Метод ` get\_solutions` вызывает `\_\_init\_\_`, а тот в свою очередь вызывает метод main, в котором происходят вычисления на более примитивной доске

#### Cell.py

Класс Cell содержит в себе ссылку на кнопку и на фигуру



При инициализации экземпляра класса, происходит дефолтное заполнение полей, инициализация цвета и символа кнопки

Метод `render` обновляет цвет и символ на кнопке, так же при каждом изменении атрибута is\_available происходит вызов метода render

#### Figure.py

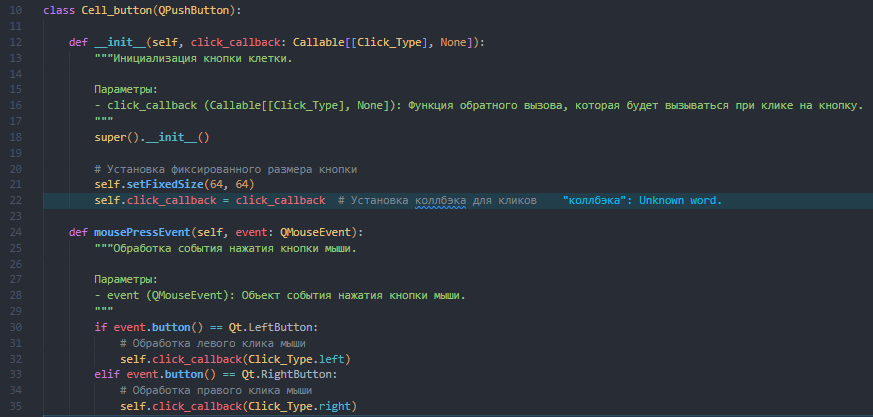
При инициализации фигура получает ссылку на доску, чтоб валидировать ходы, которая она представляет

Метод getMoves выдаёт ходы фигуры, расположенной на клетке с координатами x, y, а так же фильтрует их в случае выхода хода за границу



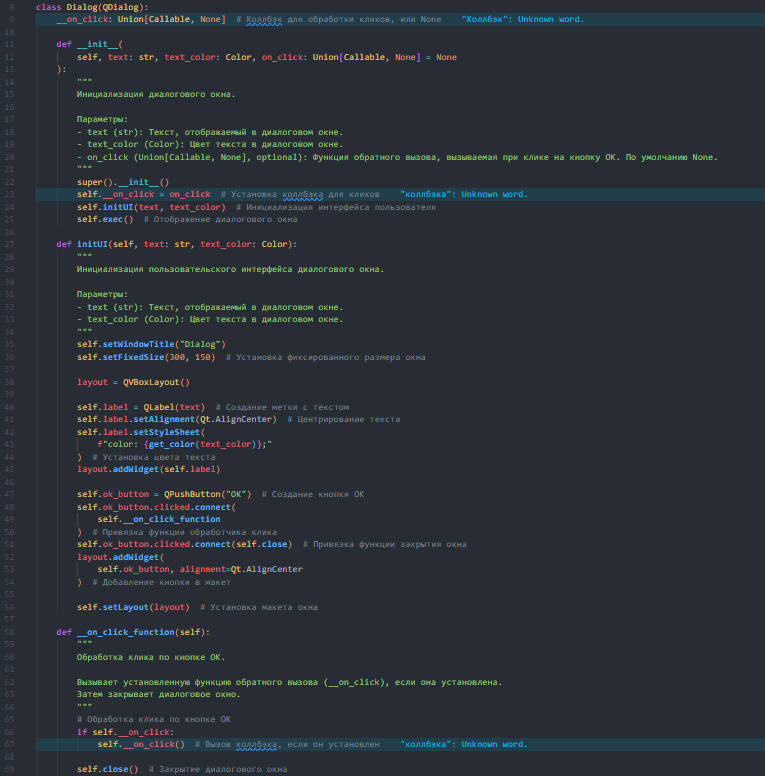
#### cell\_button.py

cell\_button.py – обёртка над кнопкной из PySide6.QtWidgets, которая способна передавать в функцию нажатия кнопки дополнительную информацию о том, правый или левый был клик мыши



#### dialog.py

dialog.py – обёртка над `QDialog` из ` PySide6.QtWidgets`, которая принимает текст, цвет текста и ссылку на функцию, которую необходимо выполнил при нажатии на кнопку `ок`



#### Color.py

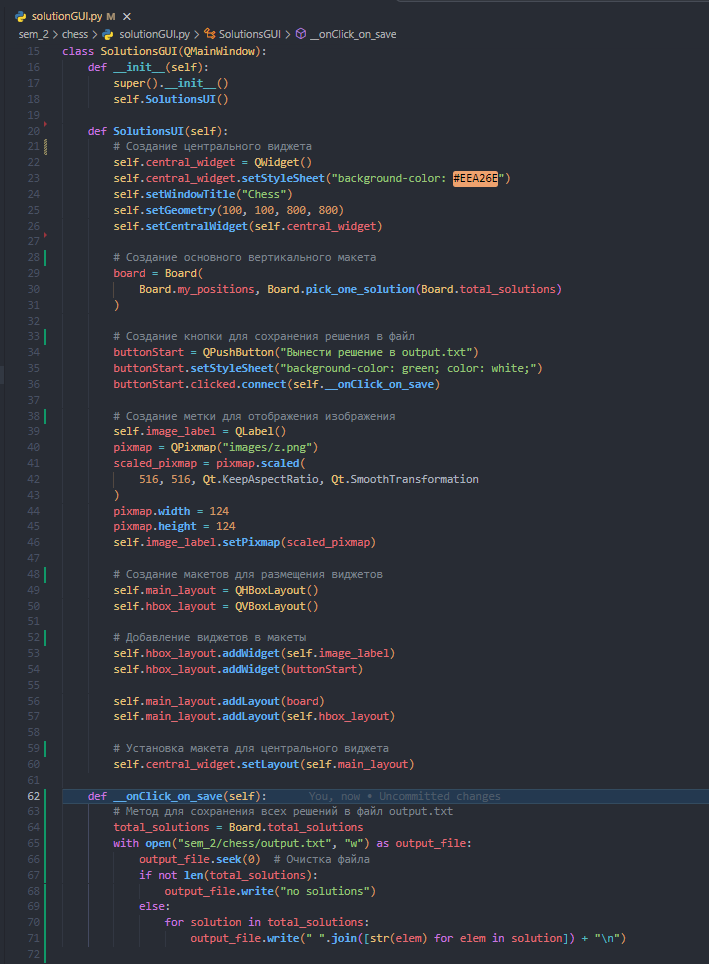
Color.py содержит `enum` с цветами, а так же функцию получения этого цвета для стилизации



#### SolutionsGUI.py

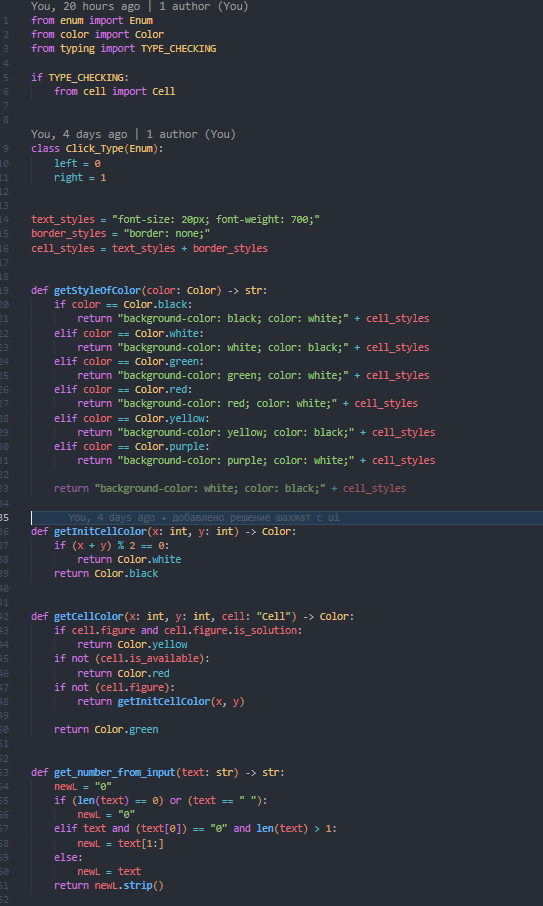
SolutionsGUI содержит второе окно, в котором отображается, а так же записывается в output.txt результат вычислений

Тут создаётся новая доска, в параметры которой добавляется найденное решение, от чего доска меняет поведение – блокирует ручную расстановку фигур.



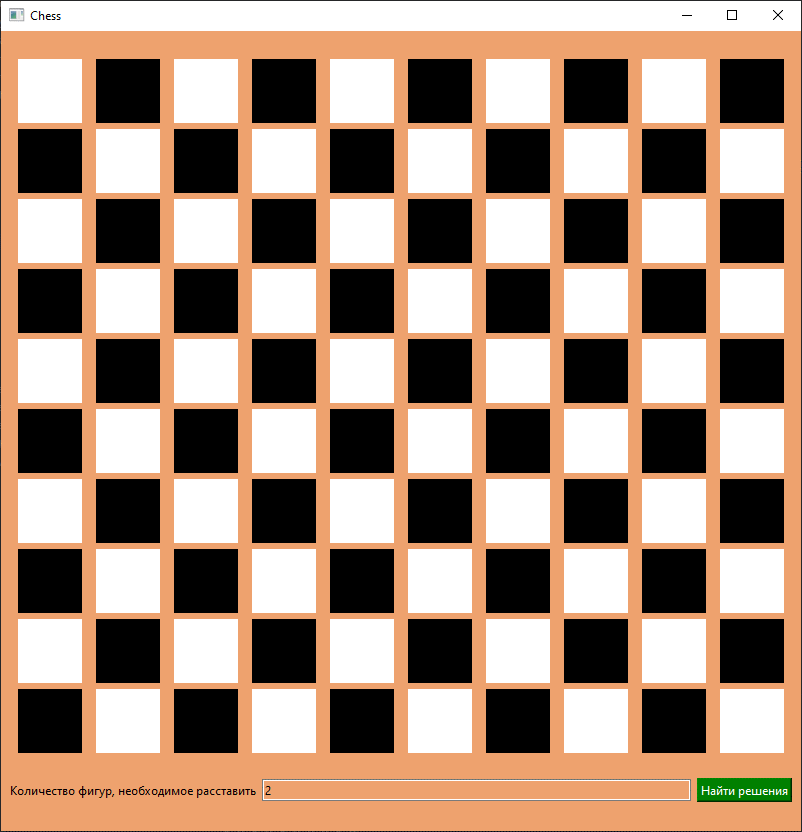
Так же здесь добавляется кнопка “Вынести решение в output.txt”, по нажатии на которую вызывается метод ` \_\_onClick\_on\_save`, который уже записывает решения в output.txt

#### Utils.py

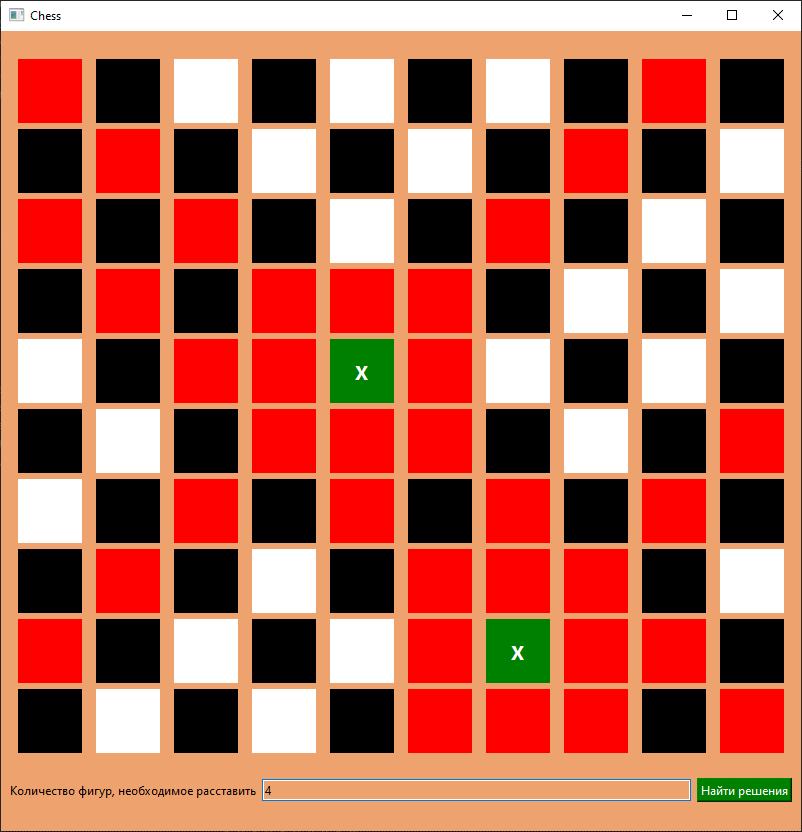
В utils.py содержатся вспомогательные функции и enum типа клика (ПКМ/ЛКМ), а так же константы стилей и вспомогательные функции стиллизации 

### Демонстрация работы программы

При запуске `index.py` появляется окно, в котором предлагается задать начальные значения расстановки фигур, а так же внести число – количества фигур, которые необходимо расставить

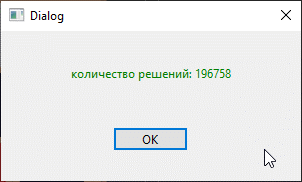


После выбора расстановки нажимаем на кнопку “Найти решения”

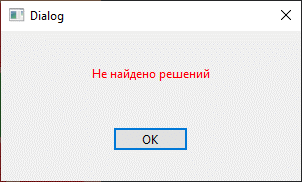


После выполнения алгоритма появляется окно с количеством найденных решений

Если решения есть:

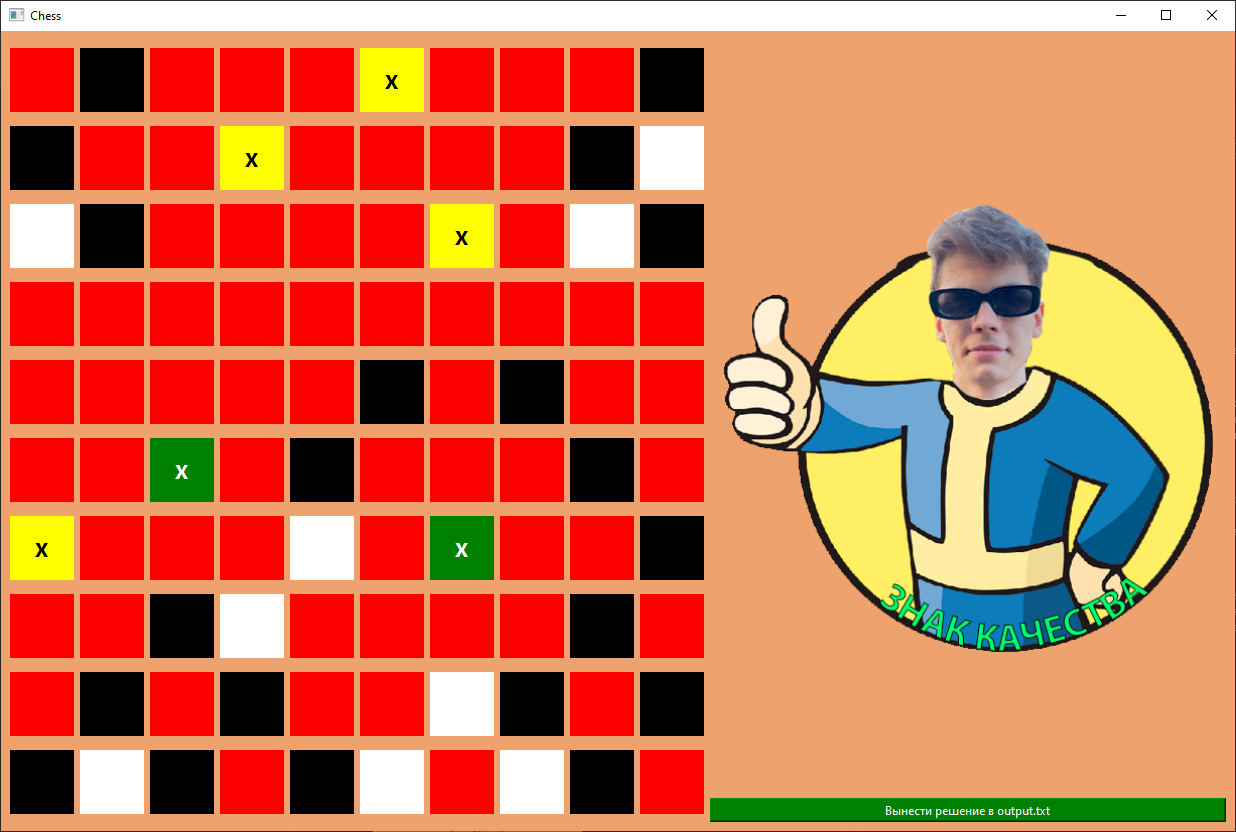


Если решений нет:

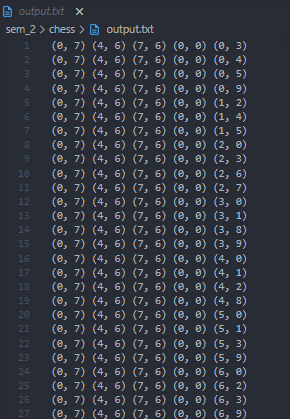


Если решения найдены, то появляется второе окно с выводом найденного решения

Зеленые фигуры – те, что мы поставили ручками, жёлтые – найденные алгоритмом

Красные клетки – недоступные клетки к ходу или клетки, которые является ходом какой-либо фигуры

По нажатии на кнопку “Вывести решения в output.txt” решения выводятся в файл `output.txt`



### Листинг

Board.py

# Импорт необходимых классов и модулей

from cell import Cell

from typing import Union

from solution import Solution

from dialog import Dialog

from color import Color

from my\_types import solutions\_type, total\_solutions\_type

from PySide6.QtWidgets import (

    QGridLayout,  # Сетка для размещения виджетов

    QLineEdit,  # Текстовое поле ввода

)

from utils import get\_number\_from\_input  # Утилита для получения числа из ввода

import random  # Для случайного выбора решений

# Типы для аннотаций

Board\_type = list[list[Cell]]

Solutions\_type = list[int]

Total\_solutions\_type = list[Solutions\_type]

# Класс доски, наследующий QGridLayout

class Board(QGridLayout):

    N: int = 10  # Размер доски

    my\_positions: solutions\_type  # Позиции фигур, размещенных пользователем

    total\_solutions: Total\_solutions\_type = list()  # Все возможные решения

    def \_\_init\_\_(

        self,

        init\_positions: Union[solutions\_type, None] = None,

        solution: Union[solutions\_type, None] = None,

    ):

        super().\_\_init\_\_()

        self.\_\_Init(init\_positions, solution)

    def **\_\_Init**(

        self,

        init\_positions: Union[solutions\_type, None] = None,

        solution: Union[solutions\_type, None] = None,

    ):

        """Инициализация доски с начальными позициями и решением."""

        self.\_\_Init\_\_board()

        if init\_positions is not None:

            self.\_\_Init\_start\_Positions(init\_positions)

        if solution is not None:

            self.\_\_Init\_solution\_Positions(solution)

        self.render()

        self.\_\_init\_render\_layout()

    def **\_\_Init\_start\_Positions**(self, init\_positions: solutions\_type):

        """Установка начальных позиций фигур.

        Параметры:

        - init\_positions (solutions\_type): Список начальных позиций фигур на доске.

        """

        for position in init\_positions:

            x, y = position

            cell = self.get\_cell(x, y)

            if cell:

                cell.put\_figure()

    def **\_\_Init\_solution\_Positions**(self, solution: solutions\_type):

        """Установка позиций для отображения решения.

        Параметры:

        - solution (solutions\_type): Список позиций для отображения решения на доске.

        """

        for position in solution:

            x, y = position

            cell = self.get\_cell(x, y)

            if cell:

                cell.put\_figure(True)

        # Отключение возможности клика на клетках после отображения решения

        for x in range(self.N):

            for y in range(self.N):

                cell = self.get\_cell(x, y)

                if cell:

                    cell.is\_clickable = False

    def **\_\_Init\_\_board**(self):

        """Инициализация шахматной доски."""

        self.L = 2

        self.board = [[0 for \_ in range(self.N)] for \_\_ in range(self.N)]

        for x in range(self.N):

            for y in range(self.N):

                cell = Cell(x, y, self)

                self.board[x][y] = cell

    def **\_\_init\_render\_layout**(self):

        """Инициализация и отрисовка раскладки доски."""

        for x in range(self.N):

            for y in range(self.N):

                cell = self.get\_cell(x, y)

                if cell:

                    self.addWidget(cell.button, x, y)

    def **get\_cell**(

        self, x: int, y: int, board: Union[Board\_type, None] = None

    ) -> Union[Cell, None]:

        """Получение клетки по координатам.

        Параметры:

        - x (int): Координата по горизонтали.

        - y (int): Координата по вертикали.

        - board (Union[Board\_type, None]): Доска, на которой производится поиск клетки.

        Возвращаемое значение:

        - Union[Cell, None]: Объект клетки (Cell) или None, если клетка не найдена.

        """

        current\_board = board if board is not None else self.board

        try:

            return current\_board[x][y]

        except:

            return None

    def **render**(self):

        """Полная отрисовка доски. Капитальная такая"""

        self.\_\_reset\_all\_available()

        self.\_\_put\_all\_unavailable\_cells()

        self.\_\_render\_cells()

    def **\_\_reset\_all\_available**(self):

        """Сброс доступности всех клеток."""

        for x in range(self.N):

            for y in range(self.N):

                cell = self.get\_cell(x, y)

                if cell:

                    cell.is\_available = True

    def **\_\_put\_all\_unavailable\_cells**(self):

        """Обновление состояния всех клеток."""

        N = self.N

        for x in range(N):

            for y in range(N):

                cell = self.get\_cell(x, y)

                # Если клетка занята фигурой, обновляем ее ходы

                if cell and cell.figure:

                    moves = cell.figure.getMoves()

                    for move in moves:

                        move\_x, move\_y = move

                        move\_cell = self.get\_cell(move\_x, move\_y)

                        if move\_cell:

                            move\_cell.is\_available = False

                else:

                    cell.Init()

    def **\_\_render\_cells**(self) -> None:

        """Отрисовка всех клеток."""

        N = self.N

        for x in range(N):

            for y in range(N):

                cell = self.get\_cell(x, y)

                if cell:

                    cell.render()

    def **\_validate\_position**(self, x: int, y: int) -> bool:

        """Проверка валидности координат клетки.

        Параметры:

        - x (int): Координата по горизонтали.

        - y (int): Координата по вертикали.

        Возвращаемое значение:

        - bool: True, если координаты являются валидными для клетки на доске, False в противном случае.

        """

        N: int = self.N

        return 0 <= x < N and 0 <= y < N

**@**staticmethod

    def **pick\_one\_solution**(solutions: total\_solutions\_type) -> solutions\_type:

        """Выбор одного решения из списка решений.

        Параметры:

        - solutions (total\_solutions\_type): Список всех возможных решений.

        Возвращаемое значение:

        - solutions\_type: Список позиций для отображения одного выбранного решения.

        """

        if len(solutions) == 0:

            return []

        positions\_without\_my = list()

        for positions in random.choice(solutions):

            if positions not in Board.my\_positions:

                positions\_without\_my.append(positions)

        return positions\_without\_my

    def **\_\_put\_my\_positions**(self):

        """Установка позиций фигур, размещенных пользователем."""

        my\_positions = []

        for x in range(self.N):

            for y in range(self.N):

                cell = self.get\_cell(x, y)

                if cell and cell.figure and not (cell.figure.is\_solution):

                    my\_positions.append((x, y))

        Board.my\_positions = my\_positions

    def **\_\_get\_init\_positions\_for\_solving**(self):

        """Получение начальных позиций фигур для решения задачи.

        Возвращаемое значение:

        - solutions\_type: Список начальных позиций фигур на доске.

        """

        init\_positions = []

        for x in range(self.N):

            for y in range(self.N):

                cell = self.get\_cell(x, y)

                if cell and cell.figure:

                    init\_positions.append([x, y])

        return init\_positions

    def **start\_solve**(self):

        """Начало процесса решения задачи."""

        init\_positions = self.\_\_get\_init\_positions\_for\_solving()

        solutions = Solution.get\_solutions(

            self.N, self.L, len(init\_positions), init\_positions

        )

        Board.total\_solutions = solutions

        self.\_\_put\_my\_positions()

    def **change\_L**(self, L: str, input: QLineEdit):

        """Изменение значения L (количество фигур).

        Параметры:

        - L (str): Значение L, вводимое пользователем.

        - input (QLineEdit): Виджет текстового поля для ввода значения L.

        """

        newL = get\_number\_from\_input(L)

        try:

            self.L = int(newL)

            input.setText(str(newL))

        except:

            self.L = 4

            input.setText(str(self.L))

            Dialog("некорректное значение L,\nL должно быть целым числом", Color.red)

Cell\_button.py

from typing import Callable

from PySide6.QtWidgets import QPushButton

from PySide6.QtGui import QMouseEvent

from PySide6.QtCore import Qt

from utils import Click\_Type

# Класс кнопки клетки

class Cell\_button(QPushButton):

    def **\_\_init\_\_**(self, click\_callback: Callable[[Click\_Type], None]):

        """Инициализация кнопки клетки.

        Параметры:

        - click\_callback (Callable[[Click\_Type], None]): Функция обратного вызова, которая будет вызываться при клике на кнопку.

        """

        super().**\_\_init\_\_**()

        # Установка фиксированного размера кнопки

        self.**setFixedSize**(64, 64)

        self.click\_callback = click\_callback  # Установка коллбэка для кликов

    def **mousePressEvent**(self, event: QMouseEvent):

        """Обработка события нажатия кнопки мыши.

        Параметры:

        - event (QMouseEvent): Объект события нажатия кнопки мыши.

        """

        if event.**button**() == Qt.LeftButton:

            # Обработка левого клика мыши

            self.click\_callback(Click\_Type.left)

        elif event.**button**() == Qt.RightButton:

            # Обработка правого клика мыши

            self.click\_callback(Click\_Type.right)

cell.py

# Импорт необходимых классов и модулей

from typing import Union, TYPE\_CHECKING

from color import Color

from utils import **getCellColor**, **getStyleOfColor**, Click\_Type

from figure import Figure

from cell\_button import Cell\_button

from dialog import Dialog

# Импортирование класса Board, если он доступен (для аннотаций типов)

if TYPE\_CHECKING:

    from board import Board

# Класс клетки доски

class Cell:

    def **\_\_init\_\_**(self, x: int, y: int, board: "Board"):

        """

        Инициализация клетки на доске.

        Параметры:

        - x (int): Горизонтальная координата клетки на доске.

        - y (int): Вертикальная координата клетки на доске.

        - board (Board): Ссылка на объект доски, к которой принадлежит клетка.

        """

        self.\_\_x: int = x

        self.\_\_y: int = y

        self.board: "Board" = board

        self.\_\_is\_available: bool = True

        self.is\_clickable: bool = True

        self.figure: Union[Figure, None] = None

        self.button: Cell\_button = Cell\_button(

            self.**on\_click**

        )  # Инициализация кнопки с обработчиком кликов

        self.**Init**()  # Инициализация клетки

    def **Init**(self):

        """

        Инициализация цвета клетки и символа на кнопке.

        """

        self.\_\_color = **getStyleOfColor**(**getCellColor**(self.\_\_x, self.\_\_y, self))

        self.button.**setStyleSheet**(self.\_\_color)

        self.**render\_symbol**()

    def **on\_click**(self, button\_type: Click\_Type):

        """

        Обработчик кликов по кнопке клетки.

        Параметры:

        - button\_type (Click\_Type): Тип клика (левый или правый).

        """

        if button\_type == Click\_Type.left:

            self.**put\_figure**()  # Левый клик - размещение фигуры

        else:

            self.**remove\_figure**()  # Правый клик - удаление фигуры

        self.board.**render**()  # Перерисовка доски

    def **render\_symbol**(self):

        """

        Отображение символа на кнопке в зависимости от наличия фигуры.

        """

        if not (self.figure):

            self.button.**setText**(" ")

        else:

            self.button.**setText**("X")

    def **render**(self):

        """

        Обновление цвета и символа на кнопке клетки.

        """

        newColor = **getStyleOfColor**(**getCellColor**(self.\_\_x, self.\_\_y, self))

        self.button.**setStyleSheet**(newColor)

        self.**render\_symbol**()

**@**property

    def **is\_available**(self) -> bool:

        """

        Геттер для доступности клетки.

        Возвращаемое значение:

        - bool: True, если клетка доступна, False в противном случае.

        """

        return self.\_\_is\_available

**@is\_available.setter**

    def **is\_available**(self, state: bool = True):

        """

        Сеттер для доступности клетки с перерисовкой.

        Параметры:

        - state (bool, optional): Состояние доступности клетки. По умолчанию True.

        """

        self.\_\_is\_available = state

        self.**render**()

    def **put\_figure**(self, is\_solution: bool = False):

        """

        Размещение фигуры на клетке.

        Параметры:

        - is\_solution (bool, optional): Флаг указывающий, является ли размещение решением. По умолчанию False.

        """

        if not (self.is\_clickable):

            Dialog("размещение фигур отключено", Color.red)

        elif not (self.\_\_is\_available) or self.figure:

            Dialog("нельзя поместить фигуру", Color.red)

        else:

            self.figure = Figure(self.\_\_x, self.\_\_y, self.board, is\_solution)

    def **remove\_figure**(self):

        """

        Удаление фигуры с клетки.

        """

        self.figure = None

color.py

from enum import Enum

class Color(Enum):

    white = "white"

    black = "black"

    green = "green"

    red = "red"

    yellow = "yellow"

    purple = "purple"

def **get\_color**(color: Color) -> str:

    if color == Color.black:

        return "black"

    elif color == Color.white:

        return "white"

    elif color == Color.green:

        return "green"

    elif color == Color.red:

        return "red"

    elif color == Color.yellow:

        return "yellow"

    elif color == Color.purple:

        return "purple"

    return "white"

dialog.py

from typing import Callable, Union

from color import Color, **get\_color**

from PySide6.QtWidgets import QDialog, QVBoxLayout, QLabel, QPushButton

from PySide6.QtCore import Qt

# Класс диалогового окна

class Dialog(QDialog):

    \_\_on\_click: Union[Callable, None]  # Коллбэк для обработки кликов, или None

    def **\_\_init\_\_**(

        self, text: str, text\_color: Color, on\_click: Union[Callable, None] = None

    ):

        """

        Инициализация диалогового окна.

        Параметры:

        - text (str): Текст, отображаемый в диалоговом окне.

        - text\_color (Color): Цвет текста в диалоговом окне.

        - on\_click (Union[Callable, None], optional): Функция обратного вызова, вызываемая при клике на кнопку OK. По умолчанию None.

        """

        super().**\_\_init\_\_**()

        self.\_\_on\_click = on\_click  # Установка коллбэка для кликов

        self.**initUI**(text, text\_color)  # Инициализация интерфейса пользователя

        self.**exec**()  # Отображение диалогового окна

    def **initUI**(self, text: str, text\_color: Color):

        """

        Инициализация пользовательского интерфейса диалогового окна.

        Параметры:

        - text (str): Текст, отображаемый в диалоговом окне.

        - text\_color (Color): Цвет текста в диалоговом окне.

        """

        self.**setWindowTitle**("Dialog")

        self.**setFixedSize**(300, 150)  # Установка фиксированного размера окна

        layout = QVBoxLayout()

        self.label = QLabel(text)  # Создание метки с текстом

        self.label.**setAlignment**(Qt.AlignCenter)  # Центрирование текста

        self.label.**setStyleSheet**(

            f"color: {**get\_color**(text\_color)};"

        )  # Установка цвета текста

        layout.**addWidget**(self.label)

        self.ok\_button = QPushButton("OK")  # Создание кнопки OK

        self.ok\_button.clicked.**connect**(

            self.**\_\_on\_click\_function**

        )  # Привязка функции обработчика клика

        self.ok\_button.clicked.**connect**(self.**close**)  # Привязка функции закрытия окна

        layout.**addWidget**(

            self.ok\_button, alignment=Qt.AlignCenter

        )  # Добавление кнопки в макет

        self.**setLayout**(layout)  # Установка макета окна

    def **\_\_on\_click\_function**(self):

        """

        Обработка клика по кнопке OK.

        Вызывает установленную функцию обратного вызова (\_\_on\_click), если она установлена.

        Затем закрывает диалоговое окно.

        """

        # Обработка клика по кнопке OK

        if self.\_\_on\_click:

            self.\_\_on\_click()  # Вызов коллбэка, если он установлен

        self.**close**()  # Закрытие диалогового окна

figure.py

from typing import Union, TYPE\_CHECKING

# Импортирование класса Board, если он доступен (для аннотаций типов)

if TYPE\_CHECKING:

    from board import Board

# Класс фигуры

class Figure:

    def **\_\_init\_\_**(self, x: int, y: int, board: "Board", is\_solution: bool = False):

        """

        Инициализация фигуры.

        Параметры:

        - x (int): Горизонтальная координата фигуры на доске.

        - y (int): Вертикальная координата фигуры на доске.

        - board (Board): Ссылка на объект доски, к которой принадлежит фигура.

        - is\_solution (bool, optional): Флаг, указывающий, является ли фигура решением. По умолчанию False.

        """

        super().**\_\_init\_\_**()

        self.x = x  # Установка координаты X

        self.y = y  # Установка координаты Y

        self.is\_solution = is\_solution  # Установка флага решения

        self.board = board  # Установка ссылки на доску

    def **getMoves**(self) -> list[list[int]]:

        """

        Получение возможных ходов фигуры.

        Возвращаемое значение:

        - list[list[int]]: Список позиций, представляющих возможные ходы фигуры.

        """

        moves: list[Union[list[int], None]] = []

        x, y = self.x, self.y

        N: int = self.board.N  # Размер доски

        # Ходы слона (диагональные)

        for index in range(1, N):

            moves.**append**(

                self.**validate\_move**(x + index, y + index)

            )  # Диагональ вправо-вниз

            moves.**append**(

                self.**validate\_move**(x - index, y + index)

            )  # Диагональ влево-вниз

            moves.**append**(

                self.**validate\_move**(x + index, y - index)

            )  # Диагональ вправо-вверх

            moves.**append**(

                self.**validate\_move**(x - index, y - index)

            )  # Диагональ влево-вверх

        # Ходы короля (горизонтальные и вертикальные)

        moves.**append**(self.**validate\_move**(x - 1, y))  # Влево

        moves.**append**(self.**validate\_move**(x + 1, y))  # Вправо

        moves.**append**(self.**validate\_move**(x, y - 1))  # Вверх

        moves.**append**(self.**validate\_move**(x, y + 1))  # Вниз

        # Фильтрация валидных ходов

        filtered\_moves: list[list[int]] = list(filter(None, moves))

        return filtered\_moves

    def **validate\_move**(self, x: int, y: int) -> Union[list[int], None]:

        """

        Проверка валидности хода для фигуры.

        Параметры:

        - x (int): Предполагаемая горизонтальная координата хода.

        - y (int): Предполагаемая вертикальная координата хода.

        Возвращаемое значение:

        - Union[list[int], None]: Список с координатами хода, если он валиден, или None, если ход невалиден.

        """

        if self.board.**\_validate\_position**(

            x, y

        ):  # Проверка, находится ли позиция в пределах доски

            return [x, y]  # Возвращение валидной позиции

        return None  # Возвращение None для невалидной позиции

index.py

from PySide6.QtWidgets import QApplication

import sys

from startGUI import StartGUI

if \_\_name\_\_ == "\_\_main\_\_":

    app = QApplication(sys.argv)

    start\_window = StartGUI()

    start\_window.**show**()

    sys.**exit**(app.**exec**())

input.py

from PySide6.QtWidgets import QLineEdit

class Input(QLineEdit):

    def **\_\_init\_\_**(self, parent=None):

        """

        Получение текста из поля ввода.

        Возвращаемое значение:

        - str: Текущий текст, введенный в поле ввода.

        """

        super().**\_\_init\_\_**(parent)

    def **get\_text**(self) -> str:

        return self.**text**()

my\_types.py

# ? Импорт пакетов

from numpy import **array**, ndarray

from typing import Tuple, List, Tuple

# ? Создаём типы

symbol\_type = list

board\_type = ndarray

solutions\_type = list[Tuple[int, int]]

total\_solutions\_type = list[solutions\_type]

init\_positions\_type = List[Tuple[int, int]]

solution.py

from time import **time**

import copy

from numpy import **array**

from my\_types import (

    symbol\_type,

    board\_type,

    solutions\_type,

    total\_solutions\_type,

    init\_positions\_type,

)

# ? Константы

SYMBOL\_FIGURE: symbol\_type = [1, "#"]

SYMBOL\_PERMISSIBLE: symbol\_type = [2, "\*"]

SYMBOL\_EMPTY: symbol\_type = [0, "0"]

class Solution:

    solutions\_total: total\_solutions\_type

    def **\_\_init\_\_**(self, N: int, L: int, K: int, init\_positions: init\_positions\_type):

        self.N = N

        self.L = L

        self.K = K

        self.solutions\_total = []

        self.**\_\_main**(init\_positions)

**@**classmethod

    def **get\_solutions**(

        cls, N: int, L: int, K: int, init\_positions: init\_positions\_type

    ) -> total\_solutions\_type:

        instance = cls(N, L, K, init\_positions)

        return instance.solutions\_total

    def **\_\_main**(self, init\_positions: init\_positions\_type) -> None:

        """

        Основная функция программы.

        Возвращает:

        - None

        """

        # ? Время начала работы

        init\_time = **time**()

        # ? Инициализируем переменные

        solutions: solutions\_type = []  # ? Решения на текущей итерации

        # ? Чтение входных данных из файла

        N = self.N

        K = self.K

        # ? Создаём доску и заполняем клетки пустотой

        board: board\_type = **array**([[SYMBOL\_EMPTY[0]] \* self.N for \_ in range(N)])

        # ? Добавляем существующие фигуры на доску

        for pos in init\_positions:

            row, col = pos

            self.**put\_figure**(board, N, row, col, solutions)

**print**("Размер доски:", N, "Фигур стоит:", K, "Нужно разместить фигур:", self.L)

        # ? Если нужно поставить 0 фигур -> выходим

        if self.L == 0:

            if not (**len**(solutions) == 0):

                self.solutions\_total.**append**(solutions)

            return self.**\_\_print\_solutions**(self.solutions\_total, init\_time)

        # ? Запускаем шарманку

        self.**\_\_print\_board**(board)

        self.**\_\_solve**(board, 0, -1, self.L, N, solutions)

        # ? Выводим решения

        self.**\_\_print\_solutions**(self.solutions\_total, init\_time)

**@**staticmethod

    def **put\_inaccessible**(board: board\_type, row: int, col: int, N: int) -> bool:

        """

        Помечает клетку на доске как недоступную для хода.

        Параметры:

        - board: Двумерный массив (доска), на которой производится маркировка клетки.

        - row: Номер строки клетки.

        - col: Номер столбца клетки.

        - N: Размерность доски (количество строк и столбцов).

        Возвращает:

        - bool: True, если клетка была успешно помечена, False в противном случае.

        """

        # ? Проверяем существует ли клетка

        if not (0 <= row < N and 0 <= col < N):

            # ? Проверяем не занята ли клетка

            return False

        if board[row][col] != SYMBOL\_FIGURE[0]:  # ? Проверяем нет ли на клетке фигуры

            board[row][col] = SYMBOL\_PERMISSIBLE[0]

            return True

        return False

    def **\_\_put\_inaccessible\_passages**(

        self, board: board\_type, row: int, col: int, N: int

    ) -> None:

        """

        Размещает недоступные клетки на доске для ходов слона и короля.

        Параметры:

        - board: Двумерный массив (доска), на которой производится маркировка клеток.

        - row: Номер строки клетки.

        - col: Номер столбца клетки.

        - N: Размерность доски (количество строк и столбцов).

        Возвращает:

        - None

        """

        # ? Размещение недоступных клеток

        # ? Флаги нужны, чтоб не повторять выходы за пределы доски

        flag1, flag2, flag3, flag4 = True, True, True, True

        # ? Ходы слона

        for index in range(1, N):

            if flag1:

                flag1 = self.**put\_inaccessible**(board, row + index, col + index, N)

            if flag2:

                flag2 = self.**put\_inaccessible**(board, row - index, col + index, N)

            if flag3:

                flag3 = self.**put\_inaccessible**(board, row + index, col - index, N)

            if flag4:

                flag4 = self.**put\_inaccessible**(board, row - index, col - index, N)

        # ? Ходы короля минус ходы слона

        self.**put\_inaccessible**(board, row - 1, col, N)

        self.**put\_inaccessible**(board, row + 1, col, N)

        self.**put\_inaccessible**(board, row, col - 1, N)

        self.**put\_inaccessible**(board, row, col + 1, N)

    # ? Размещаем фигуры

    def **put\_figure**(

        self, board: board\_type, N: int, row: int, col: int, solutions: solutions\_type

    ) -> None:

        """

        Размещает фигуру на доске и обновляет информацию о решении.

        Параметры:

        - board: Двумерный массив (доска), на которой производится размещение фигуры.

        - N: Размерность доски (количество строк и столбцов).

        - row: Номер строки клетки, на которой размещается фигура.

        - col: Номер столбца клетки, на которой размещается фигура.

        - solutions: Список расстановки в текущей ветке рекурсии.

        Возвращает:

        - None

        """

        board[row][col] = SYMBOL\_FIGURE[0]

        solutions.**append**((row, col))

        self.**\_\_put\_inaccessible\_passages**(board, row, col, N)

    def **\_\_solve**(

        self,

        board: board\_type,

        row: int,

        col: int,

        L: int,

        N: int,

        solutions: solutions\_type,

    ) -> None:

        """

        Рекурсивно находит все возможные расстановки фигур на доске.

        Параметры:

        - board: Двумерный массив (доска), на которой производится размещение фигур.

        - row: Номер текущей строки на доске.

        - col: Номер текущего столбца на доске.

        - L: Количество оставшихся фигур для размещения.

        - N: Размерность доски (количество строк и столбцов).

        - solutions: Список расстановки в текущей ветке рекурсии.

        - totalSolutions: Список с общими решениями.

        Возвращает:

        - None

        """

        # ? Перебираем возможные ходы с текущей точки

        while True:

            # ? Идём на следующую клетку

            col += 1

            if (

                col >= N

            ):  # ? Если дошли до конца строки -> переходим на на следующую строку

                col = 0

                row += 1

            if row >= N:  # ? Если дошли до конца таблицы -> выходим

                break

            if (

                board[row][col] == SYMBOL\_EMPTY[0]

            ):  # ? Если клетка доступна для хода -> ходим

                # ? Пересоздаём данные в новые переменные

                current\_board: board\_type = **array**(board)

                current\_solutions: solutions\_type = copy.**deepcopy**(solutions)

                # ? Ставим фигуру

                self.**put\_figure**(current\_board, N, row, col, current\_solutions)

                # ?  Проверяем была ли эта фигура последней

                if L - 1 == 0:

                    self.solutions\_total.**append**(current\_solutions)

                    continue

                # ? Продолжаем крутить шарманку

                self.**\_\_solve**(

                    current\_board,

                    row,

                    col

                    + 1,  # ? Cледующая клетка не может быть доступной -> пропускаем её

                    L - 1,

                    N,

                    current\_solutions,

                )

    # ? Вывод решений в файл

**@**staticmethod

    def **\_\_print\_solutions**(solutions: total\_solutions\_type, init\_time) -> None:

        """

        Выводит решения в файл, количество решений и время работы программы.

        Параметры:

        - solutions: Список расстановки в текущей ветке рекурсии.

        - init\_time: Время начала выполнения программы.

        Возвращает:

        - None

        """

        # ? Вывод количества решений

**print**("Количество решений:", **len**(solutions))

        # ? Вывод времени работы программы

**print**("Время работы:", **time**() - init\_time)

    # ? Вывод доски

**@**staticmethod

    def **\_\_print\_board**(board: board\_type) -> None:

        """

        Выводит шахматную доску в консоль.

        Параметры:

        - board: Двумерный массив (доска), который нужно вывести.

        Возвращает:

        - None

        """

        for row in board:

            s: str = ""

            for elem in row:

                if elem == SYMBOL\_FIGURE[0]:

                    s += " " + SYMBOL\_FIGURE[1]

                elif elem == SYMBOL\_EMPTY[0]:

                    s += " " + SYMBOL\_EMPTY[1]

                else:  # ? Недоступная клетка

                    s += " " + SYMBOL\_PERMISSIBLE[1]

**print**(s)

solutionGUI.py

from PySide6.QtWidgets import (

    QMainWindow,

    QWidget,

    QVBoxLayout,

    QHBoxLayout,

    QPushButton,

    QLabel,

)

from PySide6.QtGui import QPixmap

from PySide6.QtCore import Qt

from board import Board

# Класс для отображения GUI решений

class SolutionsGUI(QMainWindow):

    def **\_\_init\_\_**(self):

        super().**\_\_init\_\_**()

        self.**SolutionsUI**()

    def **SolutionsUI**(self):

        # Создание центрального виджета

        self.central\_widget = QWidget()

        self.central\_widget.**setStyleSheet**("background-color: #EEA26E")

        self.**setWindowTitle**("Chess")

        self.**setGeometry**(100, 100, 800, 800)

        self.**setCentralWidget**(self.central\_widget)

        # Создание основного вертикального макета

        board = Board(

            Board.my\_positions, Board.**pick\_one\_solution**(Board.total\_solutions)

        )

        # Создание кнопки для сохранения решения в файл

        buttonStart = QPushButton("Вынести решение в output.txt")

        buttonStart.**setStyleSheet**("background-color: green; color: white;")

        buttonStart.clicked.**connect**(self.**\_\_onClick\_on\_save**)

        # Создание метки для отображения изображения

        self.image\_label = QLabel()

        pixmap = QPixmap("images/z.png")

        scaled\_pixmap = pixmap.**scaled**(

            516, 516, Qt.KeepAspectRatio, Qt.SmoothTransformation

        )

        pixmap.**width** = 124

        pixmap.**height** = 124

        self.image\_label.**setPixmap**(scaled\_pixmap)

        # Создание макетов для размещения виджетов

        self.main\_layout = QHBoxLayout()

        self.hbox\_layout = QVBoxLayout()

        # Добавление виджетов в макеты

        self.hbox\_layout.**addWidget**(self.image\_label)

        self.hbox\_layout.**addWidget**(buttonStart)

        self.main\_layout.**addLayout**(board)

        self.main\_layout.**addLayout**(self.hbox\_layout)

        # Установка макета для центрального виджета

        self.central\_widget.**setLayout**(self.main\_layout)

    def **\_\_onClick\_on\_save**(self):

        # Метод для сохранения всех решений в файл output.txt

        total\_solutions = Board.total\_solutions

        with **open**("sem\_2/chess/output.txt", "w") as output\_file:

            output\_file.**seek**(0)  # Очистка файла

            if not **len**(total\_solutions):

                output\_file.**write**("no solutions")

            else:

                for solution in total\_solutions:

                    output\_file.**write**(" ".**join**([str(elem) for elem in solution]) + "\n")

startGUI.py

# Импорт необходимых классов и виджетов из библиотеки PySide6

from PySide6.QtWidgets import (

    QMainWindow,  # Основное окно приложения

    QWidget,  # Базовый виджет

    QPushButton,  # Кнопка

    QLabel,  # Метка текста

    QVBoxLayout,  # Вертикальный макет

    QHBoxLayout,  # Горизонтальный макет

)

# Импорт пользовательских модулей и классов

from board import Board  # Класс для представления доски (вероятно, шахматной)

from solutionGUI import SolutionsGUI  # Класс для отображения решений

from dialog import Dialog  # Класс для создания диалоговых окон

from color import Color  # Класс для определения цветов

from input import Input  # Класс для пользовательского ввода

# Определение класса основного графического интерфейса приложения

class StartGUI(QMainWindow):

    def **\_\_init\_\_**(self):

        super().**\_\_init\_\_**()  # Вызов конструктора базового класса QMainWindow

        self.**StartUI**()  # Инициализация пользовательского интерфейса

    def **StartUI**(self):

        # Создание центрального виджета

        self.board = Board()  # Создание экземпляра доски

        self.central\_widget = QWidget()  # Создание центрального виджета

        self.central\_widget.**setStyleSheet**("background-color: #EEA26E")  # Установка фона

        self.**setWindowTitle**("Chess")  # Установка заголовка окна

        self.**setGeometry**(100, 100, 800, 800)  # Установка размеров и позиции окна

        self.**setCentralWidget**(self.central\_widget)  # Установка центрального виджета

        # Создание основного вертикального макета

        self.main\_layout = QVBoxLayout()  # Основной вертикальный макет

        self.hbox\_layout = QHBoxLayout()  # Горизонтальный макет для кнопки и ввода

        # Создание кнопки для начала поиска решений

        buttonStart = QPushButton("Найти решения")

        buttonStart.**setStyleSheet**(

            "background-color: green; color: white;"

        )  # Стилизация кнопки

        buttonStart.clicked.**connect**(

            self.**\_\_on\_click\_on\_solve**

        )  # Привязка обработчика нажатия

        # Создание метки и поля ввода для количества фигур

        L\_Label = QLabel("Количество фигур, необходимое расставить")

        L\_input = Input()  # Поле ввода

        L\_input.**setPlaceholderText**(

            "Количество фигур, необходимое расставить"

        )  # Заполнитель текста

        L\_input.textChanged.**connect**(

            lambda: self.board.**change\_L**(L\_input.**text**(), L\_input)

        )  # Обновление количества фигур при изменении текста

        L\_input.**setText**(str(self.board.L))  # Установка начального текста

        # Добавление метки, поля ввода и кнопки в горизонтальный макет

        self.hbox\_layout.**addWidget**(L\_Label)

        self.hbox\_layout.**addWidget**(L\_input)

        self.hbox\_layout.**addWidget**(buttonStart)

        # Добавление доски и горизонтального макета в основной вертикальный макет

        self.main\_layout.**addLayout**(self.board)

        self.main\_layout.**addLayout**(self.hbox\_layout)

        # Установка основного макета на центральный виджет

        self.central\_widget.**setLayout**(self.main\_layout)

    def **\_\_on\_click\_on\_solve**(self):

        # Обработчик нажатия кнопки "Найти решения"

        self.board.**start\_solve**()  # Запуск решения задачи

        # Если решения не найдены, отображение диалогового окна и закрытие приложения

        if **len**(Board.total\_solutions) == 0:

            Dialog(

                "Не найдено решений", Color.red

            )  # Отображение сообщения об отсутствии решений

            self.**close**()  # Закрытие основного окна

            return

        # Если решения найдены, создание нового окна для отображения решений

        self.new\_window = SolutionsGUI()

        Dialog(

            "количество решений: " + str(**len**(Board.total\_solutions)), Color.green

        )  # Отображение количества решений

        self.**close**()  # Закрытие основного окна

        self.new\_window.**show**()  # Показ нового окна с решениями

utils.py

from enum import Enum

from color import Color

from typing import TYPE\_CHECKING

if TYPE\_CHECKING:

    from cell import Cell

class Click\_Type(Enum):

    left = 0

    right = 1

text\_styles = "font-size: 20px; font-weight: 700;"

border\_styles = "border: none;"

cell\_styles = text\_styles + border\_styles

def **getStyleOfColor**(color: Color) -> str:

    if color == Color.black:

        return "background-color: black; color: white;" + cell\_styles

    elif color == Color.white:

        return "background-color: white; color: black;" + cell\_styles

    elif color == Color.green:

        return "background-color: green; color: white;" + cell\_styles

    elif color == Color.red:

        return "background-color: red; color: white;" + cell\_styles

    elif color == Color.yellow:

        return "background-color: yellow; color: black;" + cell\_styles

    elif color == Color.purple:

        return "background-color: purple; color: white;" + cell\_styles

    return "background-color: white; color: black;" + cell\_styles

def **getInitCellColor**(x: int, y: int) -> Color:

    if (x + y) % 2 == 0:

        return Color.white

    return Color.black

def **getCellColor**(x: int, y: int, cell: "Cell") -> Color:

    if cell.figure and cell.figure.is\_solution:

        return Color.yellow

    if not (cell.is\_available):

        return Color.red

    if not (cell.figure):

        return **getInitCellColor**(x, y)

    return Color.green

def **get\_number\_from\_input**(text: str) -> str:

    newL = "0"

    if (**len**(text) == 0) or (text == " "):

        newL = "0"

    elif text and (text[0]) == "0" and **len**(text) > 1:

        newL = text[1:]

    else:

        newL = text

    return newL.**strip**()

# Заключение

В ходе выполнения лабораторной работы был разработан и оптимизирован алгоритм размещения дополнительных фигур на шахматной доске. Полученный алгоритм работает эффективно и находит все возможные решения задачи.

Были проведены тесты с различными входными данными, включая доски размером до 20х20. Программа успешно справляется с поставленной задачей, обеспечивая оптимальное распределение фигур и выводя доску. Количество найденных решений выводится под доской.

Таким образом, цель работы достигнута, и разработанный алгоритм может быть использован для эффективного решения аналогичных задач размещения фигур на шахматной доске, а также разработано приложение на основе этого алгоритма.

# Список используемых источников

1) Документация Qt для Python https://doc.qt.io/qtforpython-6/