| 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 " Интеллектуальные технологии киберфизических систем"

**«Курсовая работа»**

Студент 1 курса группы 20121

Очного отделения

Лаптев И. А.

Проверил:

Поделенюк П.П.

2024**Цели работы**

Целью работы является разработка программы для решения задачи размещения фигур на шахматной доске с графическим интерфейсом, обеспечивающей визуализацию найденных решений и взаимодействие с пользователем.

**Ход работы**

1. Инициализация интерфейса: Создание графического интерфейса с использованием библиотеки PySide6, включая размещение виджетов для ввода данных (размер доски, количество исходно размещенных фигур, координаты фигур), кнопок для управления и вывода результатов.
2. Управление шахматной доской и фигурами: Реализация класса BoardManager для управления состоянием шахматной доски (матрица), размещением фигур и проверкой их корректного расположения с использованием класса Moves, определяющего возможные ходы фигур.
3. Алгоритм нахождения решений: Разработка класса Solver для рекурсивного поиска всех возможных комбинаций размещения фигур на доске с учетом правил шахматной игры (фигуры не должны находиться под боем друг друга).
4. Визуализация результатов: После нахождения решений, результаты визуализируются на шахматной доске в графическом интерфейсе. Фигуры и их ходы отображаются с различной графической обозначенностью (например, цветом или символами).
5. Обработка пользовательского ввода: Реализация функций для добавления фигур на доску через интерфейс, импорта данных из файла, вывода координат уже размещенных фигур.
6. Управление файлами и вывод результатов: Обработка данных, введенных пользователем или импортированных из файла input.txt. По завершении поиска решений результаты записываются в файл output.txt или выводятся сообщения об отсутствии решений.
7. Очистка и обновление интерфейса: Реализация функций для очистки состояния доски, сброса введенных данных и обновления интерфейса для новых расчетов.

### **Классы и их взаимодействие:**

1. **Moves**:

Методы и атрибуты:

possibleMoves(x: int, y: int) -> Set[Tuple[int, int]]: статический метод для определения возможных ходов фигуры в заданной позиции (x, y).

1. **BoardManager**:

Атрибуты:

placedFigure: Set[Tuple[int, int]]: множество для хранения координат установленных фигур.

Методы:

putFigure(x: int, y: int) -> None: добавление новой фигуры на доску.

figureDislocation(x: int, y: int, matrix: List[List[str]]) -> List[List[str]]: модификация матрицы для установки фигуры в заданной позиции и обозначения её ходов.

otherFiguresDislocation(matrix: List[List[str]], figures: List[Tuple[int, int]]) -> List[List[str]]: модификация матрицы для установки нескольких фигур на доске.

boardInitializer(N: int) -> List[List[str]]: инициализация пустой матрицы размера NxN.

boardPrinter(matrix: List[List[str]]) -> None: печать матрицы в консоль для отладки.

1. **Solver**:

Атрибуты:

boardManager: BoardManager: экземпляр класса для управления доской и фигурами на ней.

Методы:

recursia(N: int, L: int, solutions: Set[Tuple[Tuple[int, int], ...]], currentSolution: List[Tuple[int, int]], figureCount: int, startRow: int = 0, startCol: int = 0) -> None: рекурсивный метод для нахождения всех комбинаций установки фигур на доске.

1. **BigRombWindow**:

Атрибуты:

Все атрибуты, связанные с интерфейсом: виджеты для ввода данных, кнопки, таблица для вывода результатов и т.д.

boardManager: BoardManager: экземпляр класса для управления доской и фигурами.

Методы:

Все методы, связанные с обработкой событий пользовательского взаимодействия, такие как добавление фигур, импорт данных, вывод результатов, очистка интерфейса и т.д.

**UML диаграмма**

+----------------+ +----------------+ +----------------+

| Moves | | BoardManager | | Solver |

+----------------+ +----------------+ +----------------+

| + possibleMoves(x, y) | | - placedFigure: Set[(x, y)] | | + recursia(N, L, solutions, currentSolution, figureCount, startRow, startCol) |

+----------------+ | + putFigure(x, y) | +----------------+

| + figureDislocation(x, y, matrix) |

| + otherFiguresDislocation(matrix, figures) |

| + boardInitializer(N) |

| + boardPrinter(matrix) |

+----------------+ +----------------+

|

|

v

+----------------+

| BigRombWindow |

+----------------+

| + initUI() |

| + putFigure() |

| + dataInput() |

| + answer() |

| + clear() |

| + coords() |

+----------------+

**Результаты работы**

Разработанная программа и интерфейс позволяют эффективно управлять процессом размещения фигур на доске и анализировать все возможные комбинации размещения. Это полезный инструмент как для игровых ситуаций, так и для обучения основам алгоритмического мышления и визуализации данных.

**Реализация программы с использованием функционального программирования языка Python**

#### **Ход работы**

1. **Определение задачи**: Необходимо разработать программу для размещения фигур на квадратной шахматной доске размером N x N таким образом, чтобы фигуры не находились под боем друг друга. Программа должна находить все возможные решения и визуализировать их.
2. **Проектирование решения**:
   * **Классы и функции**: Используем функциональный подход, разделяя основные функции на чистые функции и функции высшего порядка.
   * **Работа с доской**: Создаем функции для инициализации доски, размещения фигур и проверки нахождения под угрозой.
   * **Рекурсивный поиск решений**: Используем рекурсию для перебора всех возможных комбинаций размещения фигур на доске.
   * **Визуализация**: Функции для вывода результатов в консоль и сохранения в файл.
3. **Реализация программы**:
   * Определяем чистые функции для работы с доской и фигурами.
   * Написание рекурсивной функции для поиска всех решений.
   * Функции для визуализации решений в консоли и записи в файл.
4. **Тестирование и отладка**:
   * Проверяем работу программы на различных входных данных.
   * Убеждаемся, что все решения находятся корректно и визуализируются правильно

#### **Демонстрация работы программы**

Программа демонстрирует следующие возможности:

1. Ввод размера доски и количества фигур.
2. Ввод начального расположения фигур, как через интерфейс, так и через файл.
3. Поиск всех возможных комбинаций размещения фигур на доске.
4. Визуализация найденных решений в консоли и сохранение в файл.

**Листинг кода:**

import sys

import time

from PySide6.QtWidgets import (QApplication, QWidget, QLabel, QLineEdit, QPushButton, QGridLayout, QSpinBox, QTableWidget, QTableWidgetItem, QMessageBox)

from PySide6.QtGui import QColor

from typing import Set, Tuple, List

# Класс, определяющий возможные ходы фигуры

class Moves:

@staticmethod

def possibleMoves(x: int, y: int) -> Set[Tuple[int, int]]:

"""Возвращает возможные ходы для фигуры в заданной позиции"""

moves = {

(x - 3, y),

(x - 2, y - 1), (x - 2, y + 1),

(x - 1, y - 2), (x - 1, y), (x - 1, y + 2),

(x, y - 3), (x, y - 1), (x, y + 1), (x, y + 3),

(x + 1, y - 2), (x + 1, y), (x + 1, y + 2),

(x + 2, y - 1), (x + 2, y + 1),

(x + 3, y)

}

return moves

# Класс для управления доской и фигурами на ней

class BoardManager:

def \_\_init\_\_(self):

# Множество для хранения установленных фигур

self.placedFigure: Set[Tuple[int, int]] = set()

def putFigure(self, x: int, y: int) -> None:

"""Добавление новой фигуры на доску"""

self.placedFigure.add((x, y))

def figureDislocation(self, x: int, y: int, matrix: List[List[str]]) -> List[List[str]]:

"""Модифицирует матрицу для установки фигуры в заданной позиции"""

matrix[x][y] = "#"

# Применяем все возможные ходы для данной фигуры

for i, j in Moves.possibleMoves(x, y):

if 0 <= i < len(matrix) and 0 <= j < len(matrix):

matrix[i][j] = "\*"

return matrix

def otherFiguresDislocation(self, matrix: List[List[str]], figures: List[Tuple[int, int]]) -> List[List[str]]:

"""Модифицирует матрицу для установки нескольких фигур"""

for x, y in figures:

self.figureDislocation(x, y, matrix)

return matrix

@staticmethod

def boardInitializer(N: int) -> List[List[str]]:

"""Инициализирует пустую матрицу NxN"""

return [["0" for \_ in range(N)] for \_ in range(N)]

@staticmethod

def boardPrinter(matrix: List[List[str]]) -> None:

"""Печатает матрицу в консоль (для отладки)"""

for row in matrix:

print(" ".join(row))

# Класс для решения задачи установки фигур на доске

class Solver:

def \_\_init\_\_(self, boardManager: BoardManager):

self.boardManager = boardManager

def recursia(self, N: int, L: int, solutions: Set[Tuple[Tuple[int, int], ...]], currentSolution: List[Tuple[int, int]], figureCount: int, startRow: int = 0, startCol: int = 0) -> None:

"""Рекурсивная функция для нахождения всех комбинаций установки фигур"""

if figureCount == L:

if tuple(sorted(currentSolution)) not in solutions:

solutions.add(tuple(sorted(currentSolution)))

if len(solutions) == 1:

self.boardManager.boardPrinter(self.boardManager.otherFiguresDislocation(self.boardManager.boardInitializer(N), tuple(sorted(currentSolution))))

return

for i in range(startRow, N):

for j in range(startCol if i == startRow else 0, N):

if (i, j) not in currentSolution and not Moves.possibleMoves(i, j).intersection(currentSolution):

currentSolution.append((i, j))

nextStartRow = i if j == N - 1 else i

nextStartCol = j + 1 if j < N - 1 else 0

self.recursia(N, L, solutions, currentSolution, figureCount + 1, nextStartRow, nextStartCol)

currentSolution.pop()

# Класс для графического интерфейса пользователя

class BigRombWindow(QWidget):

def \_\_init\_\_(self) -> None:

super().\_\_init\_\_()

self.setWindowTitle("BigRombWindow")

self.setGeometry(400, 150, 600, 600)

self.grid = QGridLayout()

self.setLayout(self.grid)

self.boardManager = BoardManager()

self.solver = Solver(self.boardManager)

# Инициализация виджетов интерфейса

self.initUI()

def initUI(self):

"""Инициализация элементов интерфейса"""

self.boardName = QLabel("NxN:")

self.boardSize = QSpinBox()

self.boardSize.setMinimum(1)

self.boardSize.setMaximum(20)

self.labelOldFigures = QLabel("Сколько фигур поставленно:")

self.oldFigureCount = QLineEdit()

self.labelCoordinates = QLabel("Координаты фигуры:")

self.lineCoordinates = QLineEdit()

self.newFigureButton = QPushButton("Добавить фигуру")

self.newFigureButton.clicked.connect(self.putFigure)

self.inputButton = QPushButton("Импортировать данные из файла")

self.inputButton.clicked.connect(self.dataInput)

self.figureButton = QPushButton("Показать добавленные фигуры")

self.figureButton.clicked.connect(self.coords)

self.figureButton.setEnabled(False)

self.newFigures = QLabel("Фигуры которые необходимо поставить:")

self.rullerFigures = QSpinBox()

self.rullerFigures.setMinimum(1)

self.rullerFigures.setMaximum(399)

self.numOfSolution = QLabel("Колличество решений:")

self.time = QLabel("Время выполнения:")

self.totalAnswer = QTableWidget()

self.totalAnswer.horizontalHeader().setVisible(False)

self.totalAnswer.verticalHeader().setVisible(False)

self.totalAnswer.setColumnCount(0)

self.totalAnswer.setRowCount(0)

self.totalAnswer.verticalHeader().setDefaultSectionSize(50)

self.totalAnswer.horizontalHeader().setDefaultSectionSize(50)

self.readyButton = QPushButton("Готово")

self.readyButton.clicked.connect(self.answer)

self.clearButton = QPushButton("Очистить")

self.clearButton.clicked.connect(self.clear)

# Размещение виджетов на сетке

self.grid.addWidget(self.boardName, 0, 0)

self.grid.addWidget(self.boardSize, 0, 1)

self.grid.addWidget(self.labelOldFigures, 1, 0)

self.grid.addWidget(self.oldFigureCount, 1, 1)

self.grid.addWidget(self.labelCoordinates, 2, 0)

self.grid.addWidget(self.lineCoordinates, 2, 1)

self.grid.addWidget(self.inputButton, 2, 2)

self.grid.addWidget(self.newFigureButton, 3, 0)

self.grid.addWidget(self.figureButton, 3, 1)

self.grid.addWidget(self.newFigures, 4, 0)

self.grid.addWidget(self.rullerFigures, 4, 1)

self.grid.addWidget(self.numOfSolution, 5, 0)

self.grid.addWidget(self.time, 6, 0)

self.grid.addWidget(self.totalAnswer, 7, 0, 1, 2)

self.grid.addWidget(self.readyButton, 8, 0)

self.grid.addWidget(self.clearButton, 9, 0)

def putFigure(self) -> None:

"""Добавление новой фигуры"""

try:

k = int(self.oldFigureCount.text())

self.oldFigureCount.setReadOnly(True)

x, y = map(int, self.lineCoordinates.text().split(","))

self.boardManager.putFigure(x, y)

if len(self.boardManager.placedFigure) > 0:

self.figureButton.setEnabled(True)

if len(self.boardManager.placedFigure) == k:

self.oldFigureCount.setReadOnly(True)

self.lineCoordinates.clear()

self.lineCoordinates.setReadOnly(True)

except ValueError:

QMessageBox.warning(self, "Error", "Данные введены неверно")

return

self.lineCoordinates.clear()

def dataInput(self) -> None:

"""Импорт данных из файла"""

try:

with open("input.txt", "r") as file:

A, B, C = map(int, file.readline().split())

self.boardSize.setValue(A)

self.rullerFigures.setValue(B)

self.oldFigureCount.setText(str(C))

for line in file:

x, y = map(int, line.split())

self.boardManager.putFigure(x, y)

if len(self.boardManager.placedFigure) > 0:

self.figureButton.setEnabled(True)

if len(self.boardManager.placedFigure) == C:

self.lineCoordinates.clear()

self.oldFigureCount.setReadOnly(True)

self.lineCoordinates.setReadOnly(True)

except Exception as e:

QMessageBox.warning(self, "Error", str(e))

def answer(self) -> None:

"""Вызывается для нахождения решения и отображения результатов"""

try:

A = self.boardSize.value()

B = self.rullerFigures.value()

placedFigure = self.boardManager.placedFigure

start\_time = time.time()

solutions: Set[Tuple[Tuple[int, int], ...]] = set()

first\_solution = self.boardManager.boardInitializer(A)

self.solver.recursia(A, B, solutions, list(placedFigure), len(placedFigure))

end\_time = time.time()

elapsed\_time = end\_time - start\_time

if solutions:

self.numOfSolution.setText(f"Колличество решений: {len(solutions)}")

self.time.setText(f"Время выполнения:\n{elapsed\_time:.2f} сек")

first\_solution = self.boardManager.otherFiguresDislocation(first\_solution, list(solutions)[0])

self.totalAnswer.setColumnCount(A)

self.totalAnswer.setRowCount(A)

for i in range(A):

for j in range(A):

item = QTableWidgetItem()

if first\_solution[i][j] == "#":

item.setBackground(QColor("blue"))

elif first\_solution[i][j] == "\*":

item.setBackground(QColor("green"))

self.totalAnswer.setItem(i, j, item)

else:

QMessageBox.information(self, "Ответ", "Нет решений")

with open("output.txt", "w") as file:

file.write("No solution")

except Exception as e:

QMessageBox.warning(self, "Error", str(e))

def clear(self) -> None:

"""Очистка интерфейса"""

self.boardManager = BoardManager()

self.oldFigureCount.setReadOnly(False)

self.oldFigureCount.clear()

self.lineCoordinates.setReadOnly(False)

self.numOfSolution.setText("Колличество решений:")

self.time.setText("Время выполнения:")

self.totalAnswer.clear()

def coords(self) -> None:

"""Показывает координаты установленных фигур"""

QMessageBox.information(self, "Координаты", str(self.boardManager.placedFigure))

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

app = QApplication([])

window = BigRombWindow()

window.show()

sys.exit(app.exec())

**Заключение**

Разработанная программа на Python успешно решает задачу размещения фигур на шахматной доске без конфликтов с использованием объектно-ориентированного подхода. Программа способна находить и выводить все возможные решения размещения фигур на доске заданного размера, обеспечивая надежность и эффективность алгоритма.

**Список литературы**

Официальная документация Python:<https://docs.python.org/>

Документация библиотеки Pygame: https://www.pygame.org/docs/

Документация библиотеки Tkinter:<https://docs.python.org/3/library/tkinter.html>

Книга "Python Crash Course" by Eric Matthes

Книга "Automate the Boring Stuff with Python" by Al Sweigart

Статьи и учебники по алгоритмам размещения фигур на шахматной доске