|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

|  |  |
| --- | --- |
| ФАКУЛЬТЕТ | Робототехника и комплексная автоматизация (РК) |
| КАФЕДРА | Системы автоматизированного проектирования (РК6) |

**Отчет по лабораторной работе по курсу*:***

***«проектированиЕ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ»***

|  |  |  |
| --- | --- | --- |
| Студент РК6-21М | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Антонов А. С.** |
|  | (Подпись, дата) | И.О. Фамилия |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Божко А. Н.** |
|  | (Подпись, дата) | И.О. Фамилия |

Москва 2024 г*.*

# Постановка задачи

Решить судоку на рисунке 1 при помощи локального поиска с перезапусками.

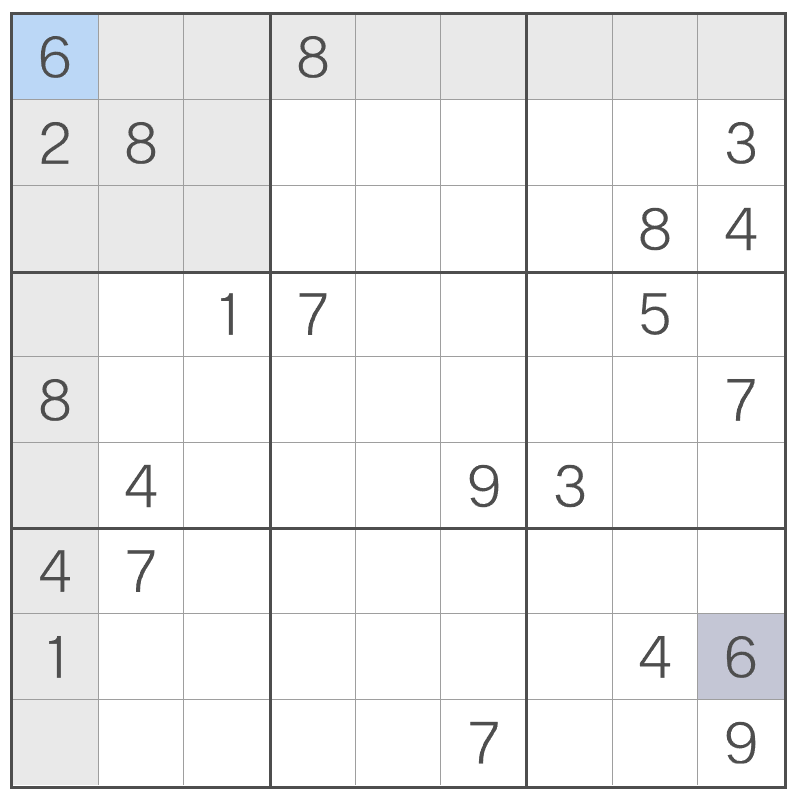


Рис. 1 – Исходное судоку

**ЛОКАЛЬНЫЙ ПОИСК С ПЕРЕЗАПУСКАМИ**

Является модификацией простого локального поиска, позволяющей выбираться из локальных минимумов или плато. Для этого начальное состояние задаётся случайным образом, а когда локальный поиск начинает стагнировать, происходит перезапуск из другой точки.

Как пример, рассмотрим график функции

У этой функции есть два локальный минимума в точках (-2; 0) и (2; 0), а также глобальный минимум в точке (0; -4).

Пусть алгоритм стартует в точке (1). Тогда он застрянет в точке локального минимума (2). В этом случае будет перезапуск и локальный поиск начнётся из случайной точки, обозначенной (3). Стартовав из неё, алгоритм дойдёт до глобального минимума и завершится.

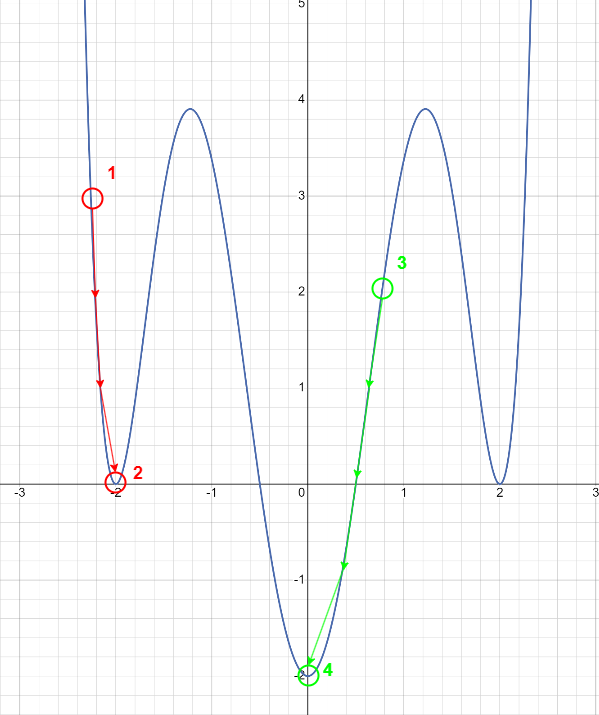


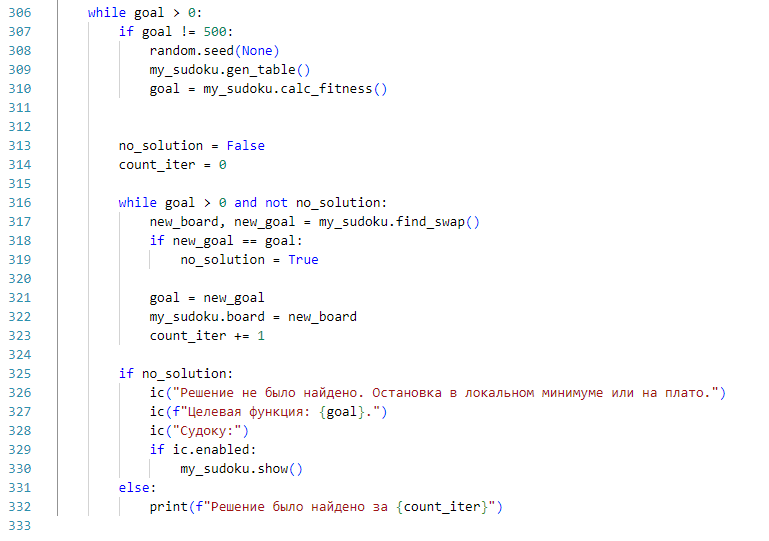
Рис. 2 – График функции с локальными минимумами

**ОПИСАНИЕ РЕАЛИЗАЦИИ**

Для решения задачи был выбран язык программирования Python. Как референс, использовалась библиотека py-sudoku, откуда были взяты структуры хранения данных и некоторые функции. Например: для вывода судоку в терминал методом show().

Основой решения является класс Puzzle. В его конструктор передаётся судоку в виде матрицы 3 на 3. После инициализации требуется вызвать метод gen\_table(), чтобы таблицу числами, сохранив решаемость.

Сам алгоритм локального поиска с перезапусками вынесен в функцию main() для сохранения наглядности.



Листинг 1 – Локальный поиск в функции main()

Метод find\_swap() ищет лучшую на данной итерации перестановку. Перезапуск осуществляется за счёт флага no\_solution.

**РЕЗУЛЬТАТЫ**

Решение было найдено за ~65 минут. Для проработки одной стартовой точки требуется от двух до четырёх секунд.

На рисунке 3 пример вывода программы.

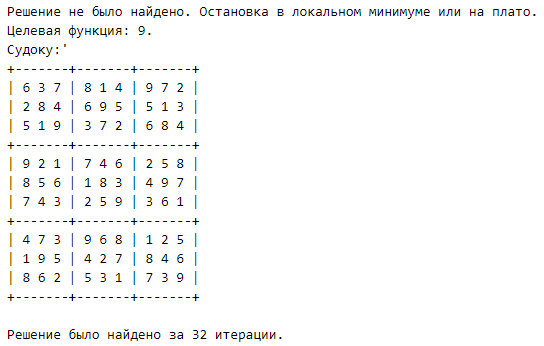


Рис. 3 – Вывод программы

**ЛИСТИНГ ПРОГРАММЫ**

import random

import copy

from sudoku import Sudoku

from icecream import ic

random.seed(42)

class Puzzle:

    """A class that represents sudoku."""

    def \_\_init\_\_(self, board: list[int]) -> None:

        """

        Initialization of a sudoku object.

        :param board: Sudoku 3x3.

        :type board: list[int]

        """

        self.board = copy.deepcopy(board)

        self.init\_board = copy.deepcopy(board)

        self.blanks = self.\_get\_blanks()

        self.blank\_fillers = self.\_calculate\_blank\_cell\_fillers()

    def \_get\_blanks(self) -> list[tuple[int, int]]:

        """

        Method to find cells whose values can be changed.

        :return: List with indexes of found cells.

        :rtype: list[tuple[int, int]]

        """

        blanks = []

        for i, row in enumerate(self.board):

            for j, cell in enumerate(row):

                if cell == 0:

                    blanks += [(i, j)]

        return blanks

    def \_calculate\_blank\_cell\_fillers(self) -> list[list[list[bool]]]:

        """

        Method for determining available values in empty cells.

        :return: Boolean map of available values for each cell. But only applies to empty cells.

        :rtype: list[list[list[bool]]]

        """

        valid\_fillers = [[[True for \_ in range(9)] for \_ in range(9)] for \_ in range(9)]

        for row, col in self.blanks:

            for i in range(9):

                same\_row = self.board[row][i]

                same\_col = self.board[i][col]

                if same\_row and i != col:

                    valid\_fillers[row][col][same\_row - 1] = False

                if same\_col and i != row:

                    valid\_fillers[row][col][same\_col - 1] = False

            grid\_row, grid\_col = row // 3, col // 3

            grid\_row\_start = grid\_row \* 3

            grid\_col\_start = grid\_col \* 3

            for y\_offset in range(3):

                for x\_offset in range(3):

                    if grid\_row\_start + y\_offset == row and grid\_col\_start + x\_offset == col:

                        continue

                    cell = self.board[grid\_row\_start +

                                        y\_offset][grid\_col\_start + x\_offset]

                    if cell:

                        valid\_fillers[row][col][cell - 1] = False

        return valid\_fillers

    def \_row\_rule(self, i: int, j: int)-> int:

        """

        Method of counting the number of identical values in a row. For cell with specified indices.

        :param i: Row index.

        :type i: int

        :param j: Column index.

        :type j: int

        :return: Number of identical values in a row.

        :rtype int

        """

        row\_collisions\_counter = -1

        el\_to\_check = self.board[i][j]

        for el in self.board[i]:

            if el == el\_to\_check:

                row\_collisions\_counter += 1

        return row\_collisions\_counter

    def \_col\_rule(self, i: int, j: int):

        """

        Method of counting the number of identical values in a column. For cell with specified indices.

        :param i: Row index.

        :type i: int

        :param j: Column index.

        :type j: int

        :return: Number of identical values in a column.

        :rtype int

        """

        col\_collisions\_counter = -1

        el\_to\_check = self.board[i][j]

        for el in self.board[:][j]:

            if el == el\_to\_check:

                col\_collisions\_counter += 1

        return col\_collisions\_counter

    def \_square\_rule(self, i: int, j: int):

        """

        Method of counting the number of identical values in a square, which contains cell with specified indices.

        :param i: Row index.

        :type i: int

        :param j: Column index.

        :type j: int

        :return: Number of identical values in a square.

        :rtype int

        """

        square\_collisions\_counter = -1

        el\_to\_check = self.board[i][j]

        sq\_i = i // 3

        sq\_j = j // 3

        sq = self.board[sq\_i: sq\_i + 3][sq\_j: sq\_j + 3]

        sq = [row[sq\_j: sq\_j + 3] for row in self.board[sq\_i: sq\_i + 3]]

        for row in sq:

            for el in row:

                if el == el\_to\_check:

                        square\_collisions\_counter += 1

        return square\_collisions\_counter

    # @timing\_decorator

    def calc\_fitness(self, board\_in=None)-> int:

        """

        Method to calculate goal function. Number of conflicts in this case.

        """

        if board\_in == None:

            board = self.board

        else:

            board = copy.deepcopy(board\_in)

        collisions = 0

        # Подсчет коллизий в строках

        for row in board:

            collisions += len(row) - len(set(row))

        # Подсчет коллизий в столбцах

        for col in range(len(board)):

            column\_values = [board[row][col] for row in range(len(board))]

            collisions += len(column\_values) - len(set(column\_values))

        # Подсчет коллизий в квадратах 3x3

        for i in range(0, 9, 3):

            for j in range(0, 9, 3):

                square\_values = []

                for row in range(3):

                    for col in range(3):

                        square\_values.append(board[i + row][j + col])

                collisions += len(square\_values) - len(set(square\_values))

        return collisions

    def gen\_table(self) -> list[int]:

        """

        A method of filling a table with values that preserves its solvability.

        """

        valid\_values = []

        for i in range(1, 10):

            count = 0

            for row in self.init\_board:

                for cell in row:

                    if cell == i:

                        count += 1

            valid\_values +=  [i] \* (9 - count)

        random.shuffle(valid\_values)

        for x, row in enumerate(board):

            for y, cell in enumerate(row):

                if cell == 0:

                    cell = random.randint(1, 9)

                    self.board[x][y] = valid\_values[0]

                    valid\_values.remove(valid\_values[0])

        return self.board

    def show(self) -> str:

        """Method to print and return table in ASCII format.

        :rtype: str

        """

        table = ''

        cell\_length = len(str(9))

        format\_int = '{0:0' + str(cell\_length) + 'd}'

        for i, row in enumerate(self.board):

            if i == 0:

                table += ('+-' + '-' \* (cell\_length + 1) \*

                        3) \* 3 + '+' + '\n'

            table += (('| ' + '{} ' \* 3) \* 3 + '|').format(\*[format\_int.format(

                x) if x != 0 else ' ' \* cell\_length for x in row]) + '\n'

            if i == 9 - 1 or i % 3 == 3 - 1:

                table += ('+-' + '-' \* (cell\_length + 1) \*

                        3) \* 3 + '+' + '\n'

        print(table)

        return table

    def get\_cell(self, cell: tuple[int, int])-> int:

        return self.board[cell[0]][cell[1]]

    def \_possible\_to\_swap(self, cell\_1: tuple[int, int], cell\_2: tuple[int, int])-> bool:

        """

        An internal method to evaluate the ability to rearrange.

        :param cell\_1: First cell.

        :type cell\_1: tuple[int, int]

        :param cell\_2: Second cell.

        :type cell\_2: tuple[int, int]

        :return: If the permutation is valid, it returns True. Otherwise False.

        :rtype: bool

        """

        cell\_1\_values = self.blank\_fillers[cell\_1[0]][cell\_1[1]]

        cell\_2\_values = self.blank\_fillers[cell\_2[0]][cell\_2[1]]

        get\_cell\_1 = self.get\_cell(cell\_1)

        get\_cell\_2 = self.get\_cell(cell\_2)

        num\_1 = [i for i in range(1, 10) if cell\_1\_values[i - 1]]

        num\_2 = [i for i in range(1, 10) if cell\_2\_values[i - 1]]

        left = cell\_2\_values[self.get\_cell(cell\_1) - 1]

        right = cell\_1\_values[self.get\_cell(cell\_2) - 1]

        if cell\_2\_values[self.get\_cell(cell\_1) - 1] and cell\_1\_values[self.get\_cell(cell\_2) - 1]:

            return True

        return False

    def find\_swap(self)-> list[tuple[int, int], int]:

        """

        A method for finding the best permutation.

        :return: List with new board and new goal.

        :rtype: list[list[list[int]], int]

        """

        def swap\_cells(board: list[int], first: tuple[int, int], second: tuple[int, int])-> list[int]:

            new\_board = copy.deepcopy(board)

            new\_board[first[0]][first[1]] = board[second[0]][second[1]]

            new\_board[second[0]][second[1]] = board[first[0]][first[1]]

            return new\_board

        best\_board = self.board

        best\_fitness = self.calc\_fitness()

        for anchor in self.blanks:

            for stone in self.blanks:

                new\_fitness = best\_fitness

                if self.\_possible\_to\_swap(anchor, stone):

                    new\_board = swap\_cells(self.board, anchor, stone)

                    new\_fitness = self.calc\_fitness(board\_in=new\_board)

                if new\_fitness < best\_fitness:

                    best\_fitness = new\_fitness

                    best\_board = new\_board

        return [best\_board, best\_fitness]

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

    no\_solution = False

    count\_iter  = 0

    board = [

        [6, 0, 0, 8, 0, 0, 0, 0, 0],

        [2, 8, 0, 0, 0, 0, 0, 0, 3],

        [0, 0, 0, 0, 0, 0, 0, 8, 4],

        [0, 0, 1, 7, 0, 0, 0, 5, 0],

        [8, 0, 0, 0, 0, 0, 0, 0, 7],

        [0, 4, 0, 0, 0, 9, 3, 0, 0],

        [4, 7, 0, 0, 0, 0, 0, 0, 0],

        [1, 0, 0, 0, 0, 0, 0, 4, 6],

        [0, 0, 0, 0, 0, 0, 7, 0, 9],

    ]

    reference = Sudoku(3, 3, board=board)

    reference.show\_full()

    reference.solve().show\_full()

    ic.disable()

    my\_sudoku = Puzzle(board)

    my\_sudoku.show()

    ic(my\_sudoku.blanks)

    ic(my\_sudoku.blank\_fillers)

    ic.enable()

    my\_sudoku.gen\_table()

    my\_sudoku.show()

    my\_sudoku.gen\_table()

    goal = 500

    while goal > 0:

        if goal != 500:

            random.seed(None)

            my\_sudoku.gen\_table()

            goal = my\_sudoku.calc\_fitness()

        no\_solution = False

        count\_iter = 0

        while goal > 0 and not no\_solution:

            new\_board, new\_goal = my\_sudoku.find\_swap()

            if new\_goal == goal:

                no\_solution = True

            goal = new\_goal

            my\_sudoku.board = new\_board

            count\_iter += 1

        if no\_solution:

            ic("Решение не было найдено. Остановка в локальном минимуме или на плато.")

            ic(f"Целевая функция: {goal}.")

            ic("Судоку:")

            if ic.enabled:

                my\_sudoku.show()

        else:

            print(f"Решение было найдено за {count\_iter}")