**دانشگاه صنعتی امیرکبیر**

**(پلی تکنیک تهران)**

**دانشکده ریاضی و علوم کامپیوتر**

# گزارش سوم – سودوکو

**نگار ابوالحسنی – 9833002**

**استاد قطعی – دکتر یوسفی‌مهر**

**فروردین هزاروچهارصدودو**

# چکیده

سودوکو یک بازی پازلی است که در آن یک جدول n*n داریم که هر خانه آن باید توسط یک عدد بین 1 تا n پر شود، به صورتی که از هر عدد در هر سطر، ستون و زیرجدول مرتبطش دقیقا یکی وجود داشته باشد.

در ادامه، ما با استفاده از الگوریتم backtracking، و با بهبود آن با MRV,LCV , forward checking به حل سودوکو پرداختیم.

برنامه ما قادر به حل بسیار سریع جدول‌های سودوکو در چند صدم ثانیه است. در ادامه به توضیح الگوریتم‌ها و کد می‌پردازیم.

# فهرست مطالب

صفحه

# فصل اول

## مقدمه

# فصل دوم

# منطق بازی و اجرای آن در پایتون

## منطق بازی و اجرای آن در پایتون

برنامه از فایل‌های solver.py, solver.optimized.py, main.py, utils.py تشکیل شده. همچنین یک فایل تست کیس برای حل سودوکو 9*9 به همراه آن آپلود شده است.

### solver.py -1-2

این فایل شامل تابع solve_sudoku است. تابع solve_sudoku بورد سودوکو را به عنوان ورودی می‌پذیرد و وقتی معمای سودوکو حل شد، True را برمی‌گرداند. ابتدا این تابع با فراخوانی تابع find_empty_cell بررسی می‌کند که آیا هنوز هیچ سلول خالی در بورد وجود دارد یا نه. اگر سلول خالی وجود ندارد، True برگردانده می‌شود. اگر هر سلولی خالی باشد، از عدد 1 شروع کرده و به تعداد سایز بورد، مقدارهای مختلف را در آن سلول امتحان می‌کند. اگر یک مقدار معتبر برای سلول فعلی پیدا شد، آن مقدار را به سلول اختصاص داده و تابع solve_sudoku را به صورت بازگشتی فراخوانی می‌کند. اگر solve_sudoku True را برگرداند، True برگردانده می‌شود. اگر solve_sudoku False را برگرداند، مقدار سلول فعلی را صفر قرار داده و به مقدار بعدی ادامه می‌دهد.

```python
import numpy as np
from utils import *

#solve sudoku
def solve_sudoku(board):
    #if there is no empty cell left, return true
    if not find_empty_cell(board):
        return True

    #for empty cells try possible variables
    row, col = find_empty_cell(board)
    for value in range(1, len(board)+1):
        if is_valid_move(board, row, col, value):
            board[row][col] = value

            if solve_sudoku(board):
                return True

            board[row][col] = 0

    return False

```

## Solver_optimized.py -2-2

تابع mrv با استفاده از تابع get_valid_moves (که در utils به توضیح آن می‌پردازیم)، سلول خالی با کمترین حرکت معتبر باقیمانده را پیدا می‌کند و یک تاپل از ردیف و ستون سلول انتخاب شده، برمیگرداند.

```python
def mrv(board):
    empty_cells = []
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] == 0:
                empty_cells.append((i, j))

    return min(empty_cells, key=lambda x: len(get_valid_moves(board, x[0], x[1])))
```

تابع lcv با دریافت بورد و ردیف و ستون یک سلول به عنوان ورودی، لیستی از حرکت‌های معتبر برای آن سلول را برمیگرداند که بر اساس تعداد حرکت‌های معتبری که با حذف هر حرکت انجام می‌شود، مرتب شده است. این تابع از تابع get_valid_moves برای پیدا کردن حرکت‌های معتبر استفاده می‌کند.

```python
def lcv(board, row, col):
    moves = list(get_valid_moves(board, row, col))
    return sorted(moves, key=lambda x: len(get_valid_moves(board, row, col) - set([x])))
```

تابع forward_check با گرفتن بورد، ردیف و ستون یک سلول و یک مقدار به عنوان ورودی، بررسی می‌کند که قرار دادن مقدار در سلول با محدودیت‌های پازل سازگار است یا خیر، با بررسی زیرشبکه، سطر و ستون برای تداخل با مقدار، اگر تداخلی یافت شود، False را برمیگرداند. اگر یک سلول پس از قرار دادن مقدار، هیچ حرکت معتبری باقی نمانده باشد، همچنین False را برمیگرداند. در غیر این صورت، True را برمیگرداند.

```python
def forward_check(board, row, col, value):
    subgrid_row = (row // int(np.sqrt(len(board)))) * int(np.sqrt(len(board)))
    subgrid_col = (col // int(np.sqrt(len(board)))) * int(np.sqrt(len(board)))

    for i in range(subgrid_row, subgrid_row + int(np.sqrt(len(board)))):
        for j in range(subgrid_col, subgrid_col + int(np.sqrt(len(board)))):
            if board[i][j] == value:
                return False

            if board[i][j] == 0 and not get_valid_moves(board, i, j).difference({value}):
                return False

    return True
```

تابع solve_sudoku_optimized با استفاده از یک جستجوی عمقی بازگشتی، به دنبال حرکت‌های معتبر مختلف می‌گردد تا تا یافتن یک راه حل برسد. در ابتدا با فراخوانی تابع find_empty_cell (که در utils به توضیح آن می‌پردازیم )، بررسی می‌کند که آیا تخته قبلاً حل شده است یا خیر. اگر حل شده باشد، True را برمیگرداند. در غیر این صورت، یک سلول خالی برای پرکردن انتخاب می‌کند، با استف

```python
def solve_sudoku_optimized(board):
    if not find_empty_cell(board):
        return True

    row, col = mrv(board)
    for value in lcv(board, row, col):
        if is_valid_move(board, row, col, value):
            board[row][col] = value

            if forward_check(board, row, col, value) and solve_sudoku_optimized(board):
                return True

            board[row][col] = 0

    return False
```

## Util.py -3-2

تابع "print_board" یک تخته سودوکو n*n را به عنوان ورودی دریافت می‌کند و آن را به صورت قابل خواندن در کنسول چاپ می‌کند.

```python
def print_board(grid):
    for row in grid:
        print(*row)
```

تابع "find_empty_cell" یک تخته سودوکو را به عنوان ورودی دریافت می‌کند و ردیف و ستون اولین خانه خالی (خانه با مقدار صفر) را که پیدا می‌کند، برمی‌گرداند. اگر هیچ خانه‌ی خالی وجود نداشته باشد، تابع None را برمی‌گرداند.

```python
#find emty cell
def find_empty_cell(board):
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] == 0:
                return (i, j)
    return None
```

تابع "is_valid_move" یک تخته سودوکو، یک شاخص ردیف و ستون و یک مقدار را به عنوان ورودی دریافت می‌کند و اگر مقدار را بتوان در خانه‌ی مشخص شده با شاخص ردیف و ستون قرار داد بدون نقض هیچ یک از قوانین سودوکو (یعنی مقدار در همان ردیف، ستون و یا زیرشبکه‌ی (n*1/2)*(n*1/2) خود قرار ندارد)، True را برمی‌گرداند. در غیر این صورت، تابع False را برمی‌گرداند.

```python
#check the validation of value
def is_valid_move(board, row, col, value):
    # check if value is already in the row
    if value in board[row]:
        return False

    # check if value is already in the column
    if value in [board[i][col] for i in range(len(board))]:
        return False

    # check if value is already in the sub-grid
    subgrid_row = (row // int(np.sqrt(len(board)))) * int(np.sqrt(len(board)))
    subgrid_col = (col // int(np.sqrt(len(board)))) * int(np.sqrt(len(board)))

    for i in range(subgrid_row, subgrid_row + int(np.sqrt(len(board)))):
        for j in range(subgrid_col, subgrid_col + int(np.sqrt(len(board)))):
            if board[i][j] == value:
                return False

    return True
```

تابع "get_valid_moves" یک تخته سودوکو و یک شاخص ردیف و ستون را به عنوان ورودی دریافت می‌کند و مجموعه‌ای از تمام حرکت‌های معتبر (یعنی تمام مقادیری که می‌توانند در خانه‌ای با شاخص ردیف و ستون مشخص شده بدون نقض هیچ یک از قوانین سودوکو قرار داد)

```python
#give all possible values
def get_valid_moves(board, row, col):
    moves = set(range(1, len(board)+1))

    # remove values already in the row
    moves -= set(board[row])

    # remove values already in the column
    moves -= set([board[i][col] for i in range(len(board))])

    # remove values already in the sub-grid
    subgrid_row = (row // int(np.sqrt(len(board)))) * int(np.sqrt(len(board)))
    subgrid_col = (col // int(np.sqrt(len(board)))) * int(np.sqrt(len(board)))

    for i in range(subgrid_row, subgrid_row + int(np.sqrt(len(board)))):
        for j in range(subgrid_col, subgrid_col + int(np.sqrt(len(board)))):
            moves.discard(board[i][j])

    return moves
```

## main.py -4-2

در این فایل با استفاده از فانکشن‌هایی که توضیح دادیم، به اجرای سودوکو می‌پردازیم. می‌توانیم فایل تست کیس را مستقیم بخوانیم یا آن را به عنوان ورودی از کاربر بگیریم. در خط اول اندازه سودوکو، در خط دوم تعداد خانه‌های پر شده و در خط‌های بعدی مختصات و عدد آن خانه‌ها را می‌گیریم. با هردو فانکشن سولور و سولور آپتیمایز شده که پیش‌تر توضیح دادیم سودوکو را حل می‌کنیم و زمان استفاده شده برای حل آن را در کنسول نمایش می‌دهیم.

به عنوان مثال یک تست کیس از سودوکو 9*9 به عنوان ورودی به آن دادم:

خط اول 9 سایز سودوکو، خط دوم 38، تعداد خانه‌های پر شده و در 38 خط بعد مختصات و عدد خانه‌های پر شده را وارد کردیم.

```
solvable
2 9 6 3 4 5 8 7 1
5 8 1 7 2 6 3 4 9
7 4 3 8 9 1 2 5 6
6 5 8 9 1 3 4 2 7
4 3 2 6 8 7 9 1 5
9 1 7 2 5 4 6 8 3
3 7 4 1 6 2 5 9 8
1 2 9 5 3 8 7 6 4
8 6 5 4 7 9 1 3 2
Elapsed time: 0.05086469650268555 seconds
solvable
2 9 6 3 4 5 8 7 1
5 8 1 7 2 6 3 4 9
7 4 3 8 9 1 2 5 6
6 5 8 9 1 3 4 2 7
4 3 2 6 8 7 9 1 5
9 1 7 2 5 4 6 8 3
3 7 4 1 6 2 5 9 8
1 2 9 5 3 8 7 6 4
8 6 5 4 7 9 1 3 2
Elapsed time optimized: 0.025936365127563477 seconds
PS D:\daneshga\8\AI\Sudoku>
```

# منابع

در حل این پروژه از chatgpt استفاده کردم.