Python Implementation

Sioros Vasileios
Andrinopoulou Christina

January 2020

1 Modeling Sudoku Variations in Python

The implementation was based on the paper *"An Integer Programming Model for the Sudoku Problem"*.

We are going to be using the **PuLP** python library, in order to model the given problems in terms of Linear Programming.

1.1 The Classic Sudoku Solver class

```
self.matrix = matrix
self.n = len(matrix)
self.m = int(sqrt(self.n))

super().__init__(
    name=f"{type(self).__name__}_solver_{self.n}_x_{self.n}".lower(),
    sense=LpMinimize
)
```

Listing 1: Initializing the **LpProblem** super class as a minimization problem

```
self.x = [
    [
        [
            LpVariable(
                f"x_{i + 1:02d}_{j + 1:02d}_{k + 1:02d}", cat=LpBinary)
            for k in range(self.n)
        ] for j in range(self.n)
    ] for i in range(self.n)
]
```

Listing 2: Declaring our variables

```
self += 0
```

Listing 3: Declaring the objective function

```
for j in range(self.n):                                                    1
    for k in range(self.n):                                                2
        self += lpSum([self.x[i][j][k]                                     3
                       for i in range(self.n)]) == 1, f"in column {j +     4
1:02d} only one {k + 1:02d}"
```

Listing 4: Declaring that there should only be one k in each column

```
for i in range(self.n):                                                    1
    for k in range(self.n):                                                2
        self += lpSum([self.x[i][j][k]                                     3
                       for j in range(self.n)]) == 1, f"in row {i +        4
1:02d} only one {k + 1:02d}"
```

Listing 5: Declaring that there should only be one k in each row

```
for k in range(self.n):                                                    1
    for p in range(self.m):                                                2
        for q in range(self.m):                                            3
            self += lpSum([                                                4
                [                                                          5
                    lpSum([                                                6
                        self.x[i][j][k]                                    7
                        for i in range(self.m * p, self.m * (p + 1))       8
                    ])                                                     9
                ]                                                          10
                for j in range(self.m * q, self.m * (q + 1))               11
            ]) == 1, f"in submatrix {p + 1:02d} {q + 1:02d} only one {k    12
+ 1:02d}"
```

Listing 6: Declaring that there should only be one k in each submatrix

```
for i in range(self.n):                                                    1
    for j in range(self.n):                                                2
        self += lpSum([self.x[i][j][k]                                     3
                       for k in range(self.n)]) == 1, f"cell {i +          4
1:02d} {j + 1:02d} must be assigned exactly one value"
```

Listing 7: Declaring that there should only be one k in each cell

```
for i in range(self.n):
    for j in range(self.n):
        if self.matrix[i][j]:
            self += self.x[i][j][self.matrix[i][j] -
                                 1] == 1, f"cell {i + 1:02d} {j +
1:02d} has an initial value of {self.matrix[i][j]:02d}"
```

Listing 8: Declaring that some cells have already received some initial values

```
for i in range(self.n):
    for j in range(self.n):
        if not self.matrix[i][j]:
            for value in self.illegal_values(i, j):
                self += (
                    self.x[i][j][value - 1] == 0,
                    f"cell {i + 1:02d} {j + 1:02d} cannot be assigned a
value of {value:02d}"
                )
```

Listing 9: Performing some preprocessing and invalidating some of the candidate values of each cell

```
def solve(self, solver=None, **kwargs):

    super().solve(solver=solver, **kwargs)

    if LpStatus[self.status] != "Optimal":
        raise ValueError(
            f"Solver failed with status '{LpStatus[self.status]}'")

    for i in range(self.n):
        for j in range(self.n):
            if not self.matrix[i][j]:
                self.matrix[i][j] = [
                    self.x[i][j][k].varValue for k in range(self.n)
                ].index(1) + 1
```

Listing 10: Calling the LpProblem solve method and reflecting the result on the two-dimensional array ″matrix″

```
    def illegal_values(self, row, col):

        values = set()

        for j in range(self.n):
            if self.matrix[row][j] is not None:
                values.add(self.matrix[row][j])

        for i in range(self.n):
            if self.matrix[i][col] is not None:
                values.add(self.matrix[i][col])

        p, q = row // self.m, col // self.m

        for i in range(self.m * p, self.m * (p + 1)):
            for j in range(self.m * q, self.m * (q + 1)):
                if self.matrix[i][j] is not None:
                    values.add(self.matrix[i][j])

        return values
```

Listing 11: Checking for invalid candidate values before even calling LpProblem.solve

If we now print an instance of the previously defined class we get the following output.

```
sudokulp_solver_9_x_9:

MINIMIZE
0*__dummy + 0

SUBJECT TO
in_column_01_only_one_01: x_01_01_01 + x_02_01_01 + x_03_01_01 +
x_04_01_01
+ x_05_01_01 + x_06_01_01 + x_07_01_01 + x_08_01_01 + x_09_01_01 = 1
...
in_column_09_only_one_09: x_01_09_09 + x_02_09_09 + x_03_09_09 +
x_04_09_09
+ x_05_09_09 + x_06_09_09 + x_07_09_09 + x_08_09_09 + x_09_09_09 = 1

in_row_01_only_one_01: x_01_01_01 + x_01_02_01 + x_01_03_01 + x_01_04_01
+ x_01_05_01 + x_01_06_01 + x_01_07_01 + x_01_08_01 + x_01_09_01 = 1
...
in_row_09_only_one_09: x_09_01_09 + x_09_02_09 + x_09_03_09 + x_09_04_09
+ x_09_05_09 + x_09_06_09 + x_09_07_09 + x_09_08_09 + x_09_09_09 = 1

in_submatrix_01_01_only_one_01: x_01_01_01 + x_01_02_01 + x_01_03_01
+ x_02_01_01 + x_02_02_01 + x_02_03_01 + x_03_01_01 + x_03_02_01 +
x_03_03_01
= 1
...
in_submatrix_03_03_only_one_09: x_07_07_09 + x_07_08_09 + x_07_09_09
+ x_08_07_09 + x_08_08_09 + x_08_09_09 + x_09_07_09 + x_09_08_09 +
x_09_09_09
= 1

cell_01_01_must_be_assigned_exactly_one_value: x_01_01_01 + x_01_01_02
+ x_01_01_03 + x_01_01_04 + x_01_01_05 + x_01_01_06 + x_01_01_07 +
x_01_01_08
+ x_01_01_09 = 1
...
cell_09_09_must_be_assigned_exactly_one_value: x_09_09_01 + x_09_09_02
+ x_09_09_03 + x_09_09_04 + x_09_09_05 + x_09_09_06 + x_09_09_07 +
x_09_09_08
+ x_09_09_09 = 1

cell_01_08_has_an_initial_value_of_02: x_01_08_02 = 1
...
cell_09_02_has_an_initial_value_of_01: x_09_02_01 = 1

cell_01_01_cannot_be_assigned_a_value_of_02: x_01_01_02 = 0
...
cell_09_09_cannot_be_assigned_a_value_of_09: x_09_09_09 = 0
```

Listing 12: The objective function and constraints of SudokuLP

```
VARIABLES                                                                1
__dummy = 0 Continuous                                                   2
0 <= x_01_01_01 <= 1 Integer                                             3
...                                                                      4
0 <= x_09_09_09 <= 1 Integer                                             5
```

Listing 13: The variables of SudokuLP

1.2 The Sudoku X Solver class

```
super().__init__(matrix)
```

Listing 14: Initializing the **SudokuLP** super class

```
for k in range(self.n):

    self += lpSum([
        self.x[r][r][k] for r in range(self.n)
    ]) == 1, f"in the diagonal only one {k + 1}"
```

Listing 15: Declaring that there should only be one k in the positive diagonal

```
for k in range(self.n):

    self += lpSum([
        self.x[r][self.n - 1 - r][k] for r in range(self.n)
    ]) == 1, f"in the anti diagonal only one {k + 1}"
```

Listing 16: Declaring that there should only be one k in the negative diagonal

1.3 The Four Square Sudoku Solver class

```
super().__init__(matrix)
```

Listing 17: Initializing the **SudokuLP** super class

```
for i in [1, self.n - self.m - 1]:
    for j in [1, self.n - self.m - 1]:
        for k in range(self.n):
            self += lpSum([
                [
                    lpSum([
                        self.x[r][c][k]
                        for c in range(j, j + self.m)
                    ])
                ]
                for r in range(i, i + self.m)
            ]) == 1, f"in square {i + 1:02d} {j + 1:02d} {i +
self.m:02d} {j + self.m:02d} only one {k + 1:02d}"
```

Listing 18: Declaring that there should only be one k in each shaded square

1.4 The Four Pyramid Sudoku Solver class

```
super().__init__(matrix)
```

Listing 19: Initializing the **SudokuLP** super class

```
for k in range(1, self.n + 1):
    self += lpSum([
        lpSum([
            self.x[r - 1][c - 1][k - 1]
            for c in range(self.m + r, self.n - r + 1)
        ]) for r in range(1, self.m + 1)
    ]) == 1, f"in pyramid 1 only one {k + 1}"
```

Listing 20: Declaring that there should only be one k in the first pyramid-shaped pyramid region

```
for k in range(1, self.n + 1):
    self += lpSum([
        lpSum([
            self.x[r - 1][c - 1][k - 1]
            for r in range(1 + c, self.n - self.m + 1 - c + 1)
        ]) for c in range(1, self.m + 1)
    ]) == 1, f"in pyramid 2 only one {k + 1}"
```

Listing 21: Declaring that there should only be one k in the second pyramid-shaped pyramid region

```
for k in range(1, self.n + 1):
    self += lpSum([
        lpSum([
            self.x[r - 1][c - 1][k - 1]
            for c in range(self.n + self.m - 1 - r, r - self.m + 1)
        ]) for r in range(self.n - self.m + 1, self.n + 1)
    ]) == 1, f"in pyramid 3 only one {k + 1}"
```

Listing 22: Declaring that there should only be one k in the third pyramid-shaped pyramid region

```
for k in range(1, self.n + 1):
    self += lpSum([
        lpSum([
            self.x[r - 1][c - 1][k - 1]
            for r in range(self.n + self.m + 1 - c, c - 1 + 1)
        ]) for c in range(self.n - self.m + 1, self.n + 1)
    ]) == 1, f"in pyramid 4 only one {k + 1}"
```

Listing 23: Declaring that there should only be one k in the fourth pyramid-shaped pyramid region

```
options = {
    "sdk": SudokuLP,
    "sdkx": SudokuXLP,
    "sdkfs": FourSquareSudokuLP,
    "sdkfp": FourPyramidSudokuLP
}

extension = path.splitext(args.load)[1][1:]

matrix = load(args.load)
problem = options[extension](matrix)
```

Listing 24: Example Usage

```
def transpose(matrix):
    return list(map(list, [*zip(*matrix)]))
```

Listing 25: Generating a new Sudoku Puzzle by transposing the supplied one

```
def relabel(matrix):
    replacements = {
        original: replacement
        for original, replacement in zip(
            range(1, len(matrix) + 1),
            np.random.permutation(range(1, len(matrix) + 1))
        )
    }

    for i in range(len(matrix)):
        for j in range(len(matrix)):
            if matrix[i][j] is not None:
                matrix[i][j] = replacements[matrix[i][j]]

    return matrix
```

Listing 26: Generating a new Sudoku Puzzle by relabeling the values in the supplied one

```
def reorder(matrix):
    def swap_rows(matrix):

        m = int(sqrt(len(matrix)))

        for p in range(m):
            for q in range(m):
                for r in range(m):
                    i1 = randint(m * p, m * (p + 1) - 1)
                    i2 = randint(m * p, m * (p + 1) - 1)

                    matrix[i1], matrix[i2] = matrix[i2], matrix[i1]

        return matrix

    return transpose(swap_rows(transpose(swap_rows(matrix))))
```

Listing 27: Generating a new Sudoku Puzzle by block reordering the rows and columns of the supplied one

```
methods = {
    "transpose": transpose,
    "relabel": relabel,
    "reorder": reorder
}

matrix = load(args.load)
matrix = methods[args.method](matrix)

dump(matrix, args.save)
```

Listing 28: Example Usage

3 The Sudoku (.sdk*) file format

For anyone interested in the details of our implementation, we now present how sudoku puzzles are being represented internally as well as how one could load or save a two-dimensional array in .sdk* format.

```
9
1, 1, 5
1, 2, 3
1, 5, 7
2, 1, 6
2, 4, 1
2, 5, 9
2, 6, 5
3, 2, 9
3, 3, 8
3, 8, 6
4, 1, 8
4, 5, 6
4, 9, 3
5, 1, 4
5, 4, 8
5, 6, 3
5, 9, 1
6, 1, 7
6, 5, 2
6, 9, 6
7, 2, 6
7, 7, 2
7, 8, 8
8, 4, 4
8, 5, 1
8, 6, 9
8, 9, 5
9, 5, 8
9, 8, 7
9, 9, 9
```

```
# +---+---+---+---+---+---+---+---+---+
# | 5 | 3 |   |   | 7 |   |   |   |   |
# +---+---+---+---+---+---+---+---+---+
# | 6 |   |   | 1 | 9 | 5 |   |   |   |
# +---+---+---+---+---+---+---+---+---+
# |   | 9 | 8 |   |   |   |   | 6 |   |
# +---+---+---+---+---+---+---+---+---+
# | 8 |   |   |   | 6 |   |   |   | 3 |
# +---+---+---+---+---+---+---+---+---+
# | 4 |   |   | 8 |   | 3 |   |   | 1 |
# +---+---+---+---+---+---+---+---+---+
# | 7 |   |   |   | 2 |   |   |   | 6 |
# +---+---+---+---+---+---+---+---+---+
# |   | 6 |   |   |   |   | 2 | 8 |   |
# +---+---+---+---+---+---+---+---+---+
# |   |   |   | 4 | 1 | 9 |   |   | 5 |
# +---+---+---+---+---+---+---+---+---+
# |   |   |   |   | 8 |   |   | 7 | 9 |
# +---+---+---+---+---+---+---+---+---+
```

Listing 29: A .sdk* file example

3.1 Loading Sudoku Puzzles

```
lines = file.readlines()
lines = map(lambda line: sub(r"#.*", "", line), lines)
lines = map(lambda line: sub(r"\s+", "", line), lines)
lines = enumerate(lines)
lines = filter(lambda data: len(data[1]) > 0, lines)
```

Listing 30: Removing any useless characters

```
try:
    index, line = next(lines)

    size = int(line)

    if size <= 0:
        raise ValueError

except ValueError:
    raise ParseError(
        index, line, "is not a valid size specifier")
```

Listing 31: Determining the size of the matrix

```
_sqrt = sqrt(size)

if _sqrt != int(_sqrt):
    raise ParseError(
        index, line, f"{size} is not a perfect square")
```

Listing 32: Performing some sanity checks corresponding to the size of the matrix

```
matrix = [[None for _ in range(size)] for _ in range(size)]

for index, line in lines:
    try:
        x, y, z = tuple(map(int, line.split(',')))

        if x < 0 or y < 0 or z < 0 or z > size:
            raise IndexError

        if matrix[x - 1][y - 1] is not None:
            raise ParseError(
                index, line,
                f"the cell has already been assigned")

        matrix[x - 1][y - 1] = z

    except IndexError:
        raise ParseError(
            index, line,
            f"is not a valid entry for a puzzle of size {size}")

    except ParseError as parse_error:
        raise parse_error

    except:
        raise ParseError(
            index, line, "Malformed entry")

return matrix
```

Listing 33: Creating and populating a two-dimensional matrix while performing some sanity checks corresponding to each matrix entry

3.2 Dumping Sudoku Puzzles

```
    file.write(f"{len(matrix)}\n")

    for i in range(len(matrix)):
        for j in range(len(matrix)):
            if matrix[i][j] is not None:
                file.write(f"{i + 1}, {j + 1}, {matrix[i][j]}\n")
```

Listing 34: Dumping a two-dimensional matrix corresponding to a Sudoku Puzzle to a file