```python
import copy

'''
Each Block object represents one block in the sudoku table
    Each Block object consists of:
        data which represents the value of itself,
        domain that represents the possible values for the block
        three list of references to the other Block objects (excluding
itself) within the three groups of constraint
        (row, column, square)
        Block will also store its own x,y position on the board with a flag
for assignment
'''


class Block:
    def __init__(self, data=0, x=None, y=None):
        self.data = data
        self.domain = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        self.constraint = []
        self.assigned = False
        self.x = x
        self.y = y

    # check given value with all constraint for repeats
    # return F for conflict, else T
    def check_constraint(self, var):
        for group in self.constraint:
            for refer in group:
                if refer.data == var:
                    return False
        return True

    # count the number of unassigned blocks that are constraint of current
block
    def num_unassigned_reference(self):
        count = 0
        for group in self.constraint:
            for refer in group:
                if not refer.assigned:
                    count += 1
        return count

    # change self value from 0 to a number and toggle assignment flag
    def assign(self, num):
        self.data = num
        self.assigned = True

    # change self value back to 0 and un-toggle assignment flag
    def clear(self):
        self.data = 0
        self.assigned = False


'''
Sudoku class consists one 2d table with int value representing what the table
looks like
```

```
also consist of a 2d matrix of references to blocks each a square in the
table
'''


class Sudoku:
    def __init__(self, data):
        self.data = data  # int table
        self.var_list = []  # block table

        # initialize variable + horizontal domain + reference
        for i in range(9):
            new = []  # reference list of each group
            for j in range(9):
                new_block = Block(data[i][j], i, j)
                if data[i][j] != 0:  # given numbers don't need domain and
are already assigned
                    new_block.domain = []
                    new_block.assigned = True
                new.append(new_block)
            # each block will have constraint reference of the horizontal
group
            for var in new:
                if var.data == 0:
                    for num in data[i]:
                        if num != 0:  # If see a number, remove that number
from domain of all unassigned blocks
                            if num in var.domain:
                                var.domain.remove(num)
                    copy = new.copy()
                    copy.remove(var)
                    var.constraint.append(copy)  # each unassigned block will
have reference of all other block in
                    # the same group
            self.var_list.append(new)

        # vertical domain + reference
        for i in range(9):
            constraint_reference = []
            for j in range(9):  # appending reference to list
                constraint_reference.append(self.var_list[j][i])
            for k in range(9):  # update domain that reduce domain size when
some nums are given
                check = self.var_list[k][i]
                if check.data != 0:
                    for refer in constraint_reference:
                        if refer.data == 0:
                            if check.data in refer.domain:
                                refer.domain.remove(check.data)
                else:
                    copy = constraint_reference.copy()
                    copy.remove(self.var_list[k][i])
                    check.constraint.append(copy)

        # square domain + reference
        for i in range(3):
            for j in range(3):
                constraint_reference = []
                for x in range(3):  # first find all reference in the square
groups
                    for y in range(3):
                        constraint_reference.append(self.var_list[i * 3 +
x][j * 3 + y])
                for p in range(3):  # then update domain
                    for q in range(3):
                        check = self.var_list[i * 3 + p][j * 3 + q]
                        if check.data != 0:
                            for refer in constraint_reference:
                                if refer.data == 0:
                                    if check in refer.domain:
                                        refer.domain.remove(check.data)
                        else:
                            copy = constraint_reference.copy()
                            copy.remove(check)
                            check.constraint.append(copy)

    def assign(self, block, var):
        block.assign(var)
        self.data[block.x][block.y] = var

    def unassign(self, block):
        block.clear()
        self.data[block.x][block.y] = 0

    # check if the sudoku is filled
    def is_done(self):
        for i in range(9):
            for j in range(9):
                if self.data[i][j] == 0:
                    return False
        return True

# backtracking algorithm
def BackTrack(sudoku):
    if sudoku.is_done():
        return sudoku
    target_block = select_next_block(sudoku)
    for value in target_block.domain:
        if target_block.check_constraint(value):
            sudoku.assign(target_block, value)
            if BackTrack(sudoku):
                return sudoku
    sudoku.unassign(target_block)
    return False

# pick the optimal block to expand
def select_next_block(sudoku):
    next = mrv(sudoku)
    return degree_heuristics(next)

# minimum remaining value
# look for block with smallest domain
def mrv(sudoku):
    candidate = []
    for i in range(9):
        for j in range(9):
            check = sudoku.var_list[i][j]
            if not check.assigned:
                if not candidate:
                    candidate.append(check)
                elif len(candidate[0].domain) > len(check.domain):
                    candidate = [check]
                elif len(candidate[0].domain) == len(check.domain):
                    candidate.append(check)
    return candidate

# degree heuristics take in output of mrv function
# pick max num of affected reference as the final pick
def degree_heuristics(mrv):
    if len(mrv) == 1:
        return mrv[0]
    else:
        candidate = mrv[0]
        cur_max = mrv[0].num_unassigned_reference()
        for i in range(1, len(mrv)):
            if mrv[i].num_unassigned_reference() > cur_max:
                candidate = mrv[i]
                cur_max = mrv[i].num_unassigned_reference()
        return candidate


if __name__ == '__main__':
    filename = input("Enter file name: ")
    file = open(filename, 'r')
    lines = file.readlines()
    matrix = []

    for line in lines:
        num_list = line.split(' ')
        row = []
        for i in range(9):
            row.append(int(num_list[i]))
        matrix.append(row)
    file.close()

    my_sudoku = Sudoku(matrix)
    solution = BackTrack(my_sudoku)

    # for i in my_sudoku.data:
    #     for j in i:
    #         print(j, end=" ")
    #     print()

    outName = input("Enter output file name: ")
    end = open(outName + '.txt', 'w')
    for i in my_sudoku.data:
        for j in i:
            end.write(str(j) + ' ')
        end.write('\n')
    end.close()
```

Output1.txt:

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

6 8 5 2 7 4 1 9 3

4 9 7 8 3 1 2 6 5

8 5 6 4 9 2 3 1 7

3 7 1 6 8 5 9 4 2

9 2 4 7 1 3 6 5 8

2 4 9 3 5 6 8 7 1

5 1 8 9 2 7 4 3 6

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

Output2.txt:

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

2 8 1 5 3 9 7 6 4

9 6 7 4 1 2 3 5 8

3 7 5 1 6 4 2 8 9

6 9 4 2 8 3 5 7 1

1 2 8 7 9 5 6 4 3

8 3 6 9 5 1 4 2 7

5 4 9 8 2 7 1 3 6

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

Output3.txt:

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

8 2 1 9 6 5 7 4 3

9 4 3 8 7 2 5 6 1

1 6 8 4 5 7 3 9 2

2 9 7 1 3 8 6 5 4

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

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

6 1 4 5 2 3 8 7 9

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

Instructions:

The program run right away. After clicking run, the first input the code will ask is the name of the input file. Make sure the text file input is in the same directory as the program. After inputting the text file, wait for some time for the program to solve the sudoku. After the problem is solved, the code will ask for the name of the output file to produce the output. The output will be a text file. Make sure to type the name without .txt.

Description:

Here is my setup of Sudoku as a constraint satisfaction problem. Each sudoku will consist of 9x9 number of variables called Block that represent each individual square where the numbers will be filled to solve the problem. Even the given numbers from the Sudoku will be a variable as well, except they will be flagged as already assigned with no domain for possible values and constraints. The other blank Blocks will consist of domain of possible values ranging of [1,9] inclusive. The domain will be reduced during the setup of the sudoku because the program will check each vertical, horizontal, and squared groups that each Block belongs to and remove the given numbers from the domain. The constraint of each block is setup as 3 lists of references to other blocks that belong to the same group as the current block. Since the check constraint function checks the domain of possible values with the values of the other block in the constraint references, the set of constraints is essentially 3 * 8 of comparisons between cur_block.value and refer_block.value.