```python
import numpy as np

from typing import List, Tuple, Set, Dict, Union
from collections import deque
from itertools import chain
from copy import deepcopy

EMPTY_ENTRY = 0
CHAR_WIDTH_OF_BOARD = 21
SIZE_OF_BOARD = 9
BLOCK_SIZE = SIZE_OF_BOARD // 3
ALL_CELLS = [(i, j) for i in range(SIZE_OF_BOARD) for j in range(SIZE_OF_BOARD)]


def pretty_print_board(board: "np.ndarray[np.int8]", empty_value: str = " ") -> None:
    """
    Takes a in a board and pretty prints it

    Args:
        board: board to be printed
        empty_value: value to be printed if cell is empty
    Print:
        board with empty_value if the cell is empty
    """

    print("-"*CHAR_WIDTH_OF_BOARD)
    for row in range(SIZE_OF_BOARD):
        print("|", end="")
        print(("{} {} {}| "*3).format(*
                                        map(lambda x: x if x != 0 else empty_value, board[row])))
        if row % 3 == 2:
            print("-"*CHAR_WIDTH_OF_BOARD)


def pretty_print_domain(domains: Dict[Tuple[int, int], Set[int]], empty_value: str = " ") ->
None:
    """
    Takes in domains, converts into a board and pretty prints it. If there are multiple values
    in the domain of a cell, it prints empty_value

    Args:
        domains: A dictionary with key: ALL_CELLS val: domain set of cell
    Prints:
        The board with empty_value if the domain of a cell has multiple values
    """
    pretty_print_board(domains2Board(domains), empty_value)


def domains2Board(domains: Dict[Tuple[int, int], Set[int]]) -> "np.ndarray[np.int8]":
    """
    Takes in domains and returns a board. If there are multiple values in the domain of a cell,
    it sets the value to 0.
```

```
    Args:
        domains: A dictionary with key: ALL_CELLS val: domain set of cell
    Returns:
        board: the domain as a board
    """
    # Unpacks one element sets and sets multi element sets to 0
    domains = {cell: 0 if len(domain) != 1 else next(iter(domain)) for cell, domain in domains.
    items()}

    # convert dictionary to ndarray
    return np.reshape(np.asarray(list(map(lambda x: domains[x[0]],
                                          sorted(list(domains.items())))), np.int8),
                      (-1, SIZE_OF_BOARD))


def validSolve(pre: "np.ndarray[np.int8]", post: "np.ndarray[np.int8]") -> bool:
    """
    Takes in a board before and after an algorithm has been applied and checks if the board
    is still an valid sudoku. Assumes that the pre algo board was valid.

    Args:
        pre: board before change
        post: board with new cell value for testing
    Returns:
            True if new boards are valid, else False
    """
    # Check if there are duplicates or invalid characters in each row
    for row in post:
        domain = set(range(1, SIZE_OF_BOARD+1))
        for val in row:
            if val not in domain and val != 0:
                return False
            elif val != 0:
                domain.remove(val)
    # Check if there are duplicates or invalid characters in each col
    for col in post.transpose():
        domain = set(range(1, SIZE_OF_BOARD+1))
        for val in col:
            if val not in domain and val != 0:
                return False
            elif val != 0:
                domain.remove(val)
    BLOCK_TOP_LEFT = [(x, y) for x in range(0, 9, 3) for y in range(0, 9, 3)]

    # Check if there are duplicates or invalid characters in each block
    for block_cell in BLOCK_TOP_LEFT:
        domain = set(range(1, SIZE_OF_BOARD+1))
        x_coord, y_coord = block_cell
        for val in map(
            lambda x: post[x],
            {(x_coord // BLOCK_SIZE * BLOCK_SIZE + x_val, y_coord // BLOCK_SIZE * BLOCK_SIZE +
            y_val)
             for x_val in range(BLOCK_SIZE) for y_val in range(BLOCK_SIZE)} - {block_cell}):
```

```
            if val not in domain and val != 0:
                return False
            elif val != 0:
                domain.remove(val)

    # Check if any of filled cells in pre are changed
    return np.all(np.any((pre == post, pre == np.zeros((SIZE_OF_BOARD, SIZE_OF_BOARD))), axis=0))


def validDomains(domains: Dict[Tuple[int, int], Set[int]]) -> bool:
    """
    Takes in a board before and after an algorithm has been applied and checks if the board
    is still an valid sudoku. Assumes that the pre algo board was valid.

    Takes in domains and checks if it's valid domain
    Args:
        domains: A dictionary with key: ALL_CELLS val: domain set of cell
    Returns:
            True if new boards are valid, else False
    """
    # Check if any cell has domain length 0
    if any(map(lambda x: not len(x), domains.values())):
        return False

    # Check if the constraining cells of any cell (with domain length one) has the same value
    for cell in ALL_CELLS:
        if len(
            domains[cell]) == 1 and any(
            map(lambda x: domains[x] == set(domains[cell]),
                constrained_variables(cell))):
            return False

    return True


def solved(domains: Dict[Tuple[int, int], Set[int]]) -> bool:
    """
    Takes in a domain and checks if the board has been solved

    Args:
        domains: A dictionary with key: ALL_CELLS val: domain set of cell
    Returns:
            True if board was been solved, else False
    """
    return all(map(lambda x: len(x) == 1, domains.values()))


def load_file(path: str = r"A2\sudoku_small.csv", n: int = 1) -> List["np.ndarray[np.int8]"]:
    """
    Reads file at path and constructs board

    Args:
        path: location of the file with sudoku boards. The first row is the column titles. All
```

```
        others of the
                form " 'partial sudoku', 'solved sudoku' " . A partial sudoku is 81 character
                string, containing
                intgers between 0 and 9 inclusive. The integer 0 implies that the cell is
                blank.
        n: integer number of how many boards to create
    Returns:
        boards: a list of boards
    """
    fv = open(path, "r")
    _ = fv.readline()

    boards = []
    for i in range(n):
        ln = fv.readline()
        if "," in ln:
            partial, solved = ln.split(",")
        else:
            partial = ln.strip("\n")
        boards.append(np.reshape(np.asarray(list(partial), np.int8), (-1, SIZE_OF_BOARD)))

    return boards


def constrained_variables(coord: Tuple[int, int]) -> Set[Tuple[int, int]]:
    """
    When given coordinates for a cell, returns all of the coordinates of cells which constrain
    that cell

    Args:
        coord: A 2-tuple with integers between 0 and 8 inclusive, representing the coordinate
        of the cell

    Returns:
        A list of coordinates, which act as constraints for the coord cell,
        i.e. the coordinates of all of the cells in the same row, column, or block as the given
        cell coord
    """
    x_coord = coord[0]
    y_coord = coord[1]
    variables = {
        (x_coord, val) for val in range(SIZE_OF_BOARD)} | {
        (val, y_coord) for val in range(SIZE_OF_BOARD)} | {
        (x_coord // BLOCK_SIZE * BLOCK_SIZE + x_val, y_coord // BLOCK_SIZE * BLOCK_SIZE + y_val)
        for x_val in range(BLOCK_SIZE) for y_val in range(BLOCK_SIZE)}
    # cell does not constrain itself
    return variables - {coord}


def create_constraint_set() -> Set[Tuple[Tuple[int, int], Tuple[int, int]]]:
    """
    Generates initialized constraint_set for blank sudoku board. A cell's value can not be same
    as the
```

```
    value of any cell in it's row, column or block.

    Returns:
        constraints: a set of constraints. A constraint is a 2-tuple of cell coords. Constraint
        (Xi,Xj) implies that Xi != Xj.
    """
    return set(chain(*map(lambda cell: {(cell, diff) for diff in constrained_variables(cell)},
    ALL_CELLS)))


def create_domain_set(board: "np.ndarray[np.int8]") -> Dict[Tuple[int, int], Set[int]]:
    """
    Generates a domains from a board. If cell has a set value, it's domain is just that value,
    else the domain is all integers from 1 to 9 inclusive.

    Args:
        board: a board with either a value for each cell or 0 if it has no value
    Returns:
        domains: A dictionary with key: ALL_CELLS val: domain set of cell
    """
    return {cell: set(range(1, 10)) if not board[cell] else {board[cell]} for cell in ALL_CELLS}


def AC3(constraints: Set[Tuple[int, int]],
        domains: Dict[Tuple[int, int],
                      Set[int]], returnQueueLength = False) -> Union[Union[Dict[Tuple[int, int],
                      Set[int]], bool],Tuple[Union[Dict[Tuple[int, int],Set[int]], bool],List[
                      int]]]:
    """
    Takes in a Constraint Satisfaction Problem (CSP) and makes it arc-constraint

    Args:
        constraints: a set of all relationship between variables. Constraint (x,y) => x != y
        domains: a dictionary with key: ALL_CELLS val: domain set of cell
        returnQueueLength: if true, this function returns a list of the length of the queue at
        each step
    Returns:
        domains: A dictionary with key: ALL_CELLS val: domain set of cell
        queue_lengths: a list of the length of the queue at each step (if returnQueueLength ==
        True)
    """
    queue = deque(constraints)
    qlen = []
    while queue:
        qlen.append(len(queue))
        Xi, Xj = queue.popleft()
        revised, domains = revise(Xi, Xj, domains)
        if revised:
            if not domains[Xi]:
                return False
            queue.extend(set((Xk, Xi) for Xk in constrained_variables(Xi)))

    if returnQueueLength:
```

```
        return domains, qlen
    else:
        return domains


def revise(Xi: Tuple[int, int],
           Xj: Tuple[int, int],
           domains: Dict[Tuple[int, int],
                         Set[int]]) -> Tuple[bool, Dict[Tuple[int, int],
                                                        Set[int]]]:
    """
    Takes in an arc and makes it arc consistent

    Args:
        Xi: variable's whose domain is to be adjusted
        Xj: variables's whose being checked against Xi
        domains: A dictionary with key: ALL_CELLS val: domain set of cell
    Returns:
        revised: was Xi's domain changed
        domains: A dictionary with key: ALL_CELLS val: domain set of cell
    """
    revised = False
    removed = set()
    for val in domains[Xi]:
        if not (domains[Xj] - set([val])):
            revised = True
            removed.add(val)
    domains[Xi] -= removed
    return revised, domains

def select_unassigned_variable(domains: Dict[Tuple[int, int],
                      Set[int]])-> Tuple[int,int]:
    """
    Takes in domains and selects the domain with lowest number of options which hasn't be
    assigned
    Args:
        domains: a dictionary with key: ALL_CELLS val: domain set of cell
    Returns:
        The coord of the domain with lowest number of options which hasn't be assigned
    """
    return min(filter(lambda cell: len(domains[cell])>1,domains.keys()), key = lambda cell: len(
    domains[cell]))


def backtracking_search(
        domains: Dict[Tuple[int, int],
                      Set[int]]):
    """
    Takes in a sudoku Constraint Satisfaction Problem (CSP) and searches solution using
    backtracking,
    starting from the last arc-consistent step of an unsuccessful AC-3 attempt.

    Args:
```

```python
        constraints: a set of all relationship between variables. Constraint (x,y) => x != y
        domains: a dictionary with key: ALL_CELLS val: domain set of cell
    Returns:
        A solution to the board, else False.
    """
    return backtracking_search_aux(select_unassigned_variable(domains), domains)


def backtracking_search_aux(currCell: Tuple[int, int], domains: Dict[Tuple[int, int], Set[int]]):
    """
    Recursive auxilary assist for  backtracking_search

    Args:
        currCell: a tuple containing coordinates of current cell we're working with
        constraints: a set of all relationship between variables. Constraint (x,y) => x != y
        domains: a dictionary with key: ALL_CELLS val: domain set of cell
       board: array containing board values. If there are multiple values in the domain of a
       cell, value is 0.
    Returns:
        A solution domain as board, else False.
    """
    if solved(domains):
        return domains

    for testVal in domains[currCell]:
        newDomains = deepcopy(domains)
        newDomains[currCell] = set([testVal])

        for cell in constrained_variables(currCell):
            newDomains[cell] -= set([testVal])

        if validDomains(newDomains):
            ret = backtracking_search_aux(select_unassigned_variable(domains), newDomains)
            if ret:
                return ret
    else:
        return False
```