VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

ARTIFICIAL INTELLIGENCE

IT159IU

REPORT LAB 4-5

Instructor:

Dr. Nguyen Trung Ky

Dr. Ly Tu Nga

Nguyen Huynh Ngan Anh - ITDSIU23003

1. Exercise 1:

```
class csp:

    # INITIALIZING THE CSP      (variable) digits: Literal['123456789']
    def __init__(self, domain=digits, grid=""):
        """
        Unitlist consists of the 27 lists of peers
        Units is a dictionary consisting of the keys and the corresponding lists of peers
        Peers is a dictionary consisting of the 81 keys and the corresponding set of 27 peers
        Constraints denote the various all-different constraints between the variables
        """
        "***YOUR CODE HERE ***"
        self.variables = squares
        self.domain = self.getDict(grid)
        self.values = self.getDict(grid)

        self.unitList = (
                [cross(rows, c) for c in cols]
                + [cross(r, cols) for r in rows]
                + [
                    cross(rs, cs)
                    for rs in ("ABC", "DEF", "GHI")
                    for cs in ("123", "456", "789")
                ]
        )

        self.units = dict((s, [u for u in self.unitList if s in u]) for s in squares)
        self.peers = dict((s, set(sum(self.units[s], [])) - set([s])) for s in squares)
        self.constraints = {
            (variable, peer)
            for variable in self.variables
            for peer in self.peers[variable]
        }
```

Key Components of the CSP Representation

- Variables:
  - Represented by self.variables, which is a list of all 81 cells in the Sudoku grid (e.g., "A1", "A2", ..., "I9").
  - These variables correspond to the rows (A-I) and columns (1-9) of the grid.
- Domains:
  - Represented by self.domain and self.values, which are dictionaries mapping each variable to its possible values.
  - Initially, each variable's domain is "123456789" unless the grid specifies a pre-filled value.
- Units:
  - Represented by self.unitList, which contains 27 lists:
    - 9 rows
    - 9 columns
    - 9 sub-grids (3x3 blocks)
  - Each unit is a group of variables that must satisfy the "all-different" constraint.
- Peers:
  - Represented by self.peers, which is a dictionary mapping each variable to the set of other variables in the same row, column, or sub-grid.
  - Peers are used to enforce constraints during the solving process.
- Constraints: represented by self.constraints, which is a set of all variable-peer pairs that must satisfy the "all-different" constraint.

Design Decisions

- Grid Representation:
  - The grid is represented as a single string of 81 characters, where each character corresponds to a cell in the Sudoku grid.
  - The getDict method converts this string into a dictionary mapping variables to their values or possible domains.
- Units and Peers:
  - Units are precomputed to simplify constraint checking.
  - Peers are derived from units, ensuring that each variable has a direct reference to its related variables.
- Constraints: constraints are implicitly enforced using the peers dictionary, which avoids the need to explicitly store all constraints.

How the Implementation Works

- Initialization (__init__):
  - The __init__ method initializes the CSP by defining variables, domains, units, peers, and constraints.
  - It uses helper functions like cross to generate units and peers.
- Domain Initialization (getDict): the getDict method processes the input grid string and assigns values to variables:
    - If a cell is pre-filled (non-zero), its domain is restricted to that value.
    - Otherwise, the domain is set to "123456789".
- Units and Peers:
  - Units are generated for rows, columns, and sub-grids using the cross function.
  - Peers are computed by summing all units containing a variable and removing the variable itself.

I choose this design because:

- **Efficiency**: precomputing units and peers reduces the computational overhead during constraint checking.
- **Modularity**: the design separates variables, domains, units, peers, and constraints, making the code easier to understand and extend.
- **Scalability**: the representation can be adapted to other CSP problems by modifying the variables, domains, and constraints.
- **Alignment with CSP Principles**: the implementation adheres to the standard CSP framework, making it compatible with generic CSP-solving algorithms like backtracking and forward checking.

2. **Exercise 2:**

Implement backtracking search algorithm

```
def Backtracking_Search(csp):
    """
    Backtracking search initialize the initial assignment
    and calls the recursive backtrack function
    """

    assignment = {}
    return Recursive_Backtracking(assignment, csp)

    util.raiseNotDefined()
```

```
def Recursive_Backtracking(assignment, csp):
    """
    The recursive function which assigns value using backtracking
    """

    # util.raiseNotDefined()
    if isComplete(assignment):
        return assignment

    var = Select_Unassigned_Variables(assignment, csp)
    domain = deepcopy(csp.values)

    for value in csp.values[var]:
        if isConsistent(var, value, assignment, csp):
            assignment[var] = value
            inferences = {}
            inferences = Inference(assignment, inferences, csp, var, value)
            if inferences != "FAILURE":
                result = Recursive_Backtracking(assignment, csp)
                if result != "FAILURE":
                    return result

            del assignment[var]
            csp.values.update(domain)

    return "FAILURE"
```

Test the backtracking search for euler puzzles

```
The board -  50  takes  0.005178213119506836  seconds
After solving:
3  5  1  | 2  8  6  | 4  9  7
4  9  2  | 1  5  7  | 6  3  8
7  8  6  | 9  3  4  | 5  1  2
-------------------------------------------
2  7  5  | 4  6  9  | 1  8  3
9  3  8  | 5  2  1  | 7  6  4
6  1  4  | 8  7  3  | 2  5  9
-------------------------------------------
8  2  9  | 6  4  5  | 3  7  1
1  6  3  | 7  9  2  | 8  4  5
5  4  7  | 3  1  8  | 9  2  6
Number of problems solved is:  50
Time taken to solve the puzzles is:  0.47748851776123047
```

Test the backtracking search for magictour puzzles

```
The board -  95  takes  0.11361980438232422  seconds
After solving:
3  5  4  | 1  8  6  | 9  2  7
2  9  8  | 7  4  3  | 6  1  5
1  6  7  | 9  5  2  | 4  8  3
-------------------------------------------
4  8  1  | 5  2  7  | 3  6  9
9  3  2  | 6  1  4  | 5  7  8
5  7  6  | 3  9  8  | 2  4  1
-------------------------------------------
7  2  9  | 8  6  5  | 1  3  4
8  4  5  | 2  3  1  | 7  9  6
6  1  3  | 4  7  9  | 8  5  2
Number of problems solved is:  95
Time taken to solve the puzzles is:  33.15889263153076
```

3. Exercise 3:

Implement the AC3 algorithm

```
def ac3(csp):
    """
    AC-3 algorithm to enforce arc consistency.
    """
    # Initialize the queue with all arcs (constraints)
    arc_queue = [(variable, peer) for variable in csp.variables for peer in csp.peers[variable]]

    while arc_queue:
        (variable, peer) = arc_queue.pop(0)

        # If the domain of the variable is revised
        if revise(csp, variable, peer):
            # If the domain of the variable becomes empty, the CSP is unsolvable
            if len(csp.values[variable]) == 0:
                return False

            # Add all arcs (neighbor, variable) back to the queue for further checking
            for neighbor in csp.peers[variable] - {peer}:
                arc_queue.append((neighbor, variable))
    assignment = {var: csp.values[var] for var in csp.variables if len(csp.values[var]) == 1}
    return assignment
```

```
def revise(csp, variable, peer):
    """
    Revise the domain of the variable to ensure consistency with the peer.
    """
    revised = False
    for value in csp.values[variable]:
        # Check if there is no value in the peer's domain that satisfies the constraint
        if not any(value != other_value for other_value in csp.values[peer]):
            csp.values[variable] = csp.values[variable].replace(value, "")
            revised = True

    return revised

def Backtracking_Search_With_AC3(csp):
    """
    Perform AC-3 preprocessing and then solve the CSP using backtracking search.
    """
    # Perform AC-3 preprocessing
    assignment = ac3(csp)
    if assignment is False:
        return "FAILURE"  # If AC-3 fails, the CSP is unsolvable

    # Continue solving with backtracking search
    return Recursive_Backtracking(assignment, csp)
```

Change the algorithm used in sudoku.py

```
sudoku.py > ...
            array.append(line)
    ins.close()
    i = 0
    boardno = 0
    start = time.time()
    f = open("output.txt", "w")
    for grid in array:
        startpuzle = time.time()
        boardno = boardno + 1
        sudoku = csp(grid=grid)
        solved = Backtracking_Search_With_AC3(sudoku)
        print("The board - ", boardno, " takes ", time.time() - startpuzle, " seconds")
        if solved != "FAILURE":
            print("After solving: ")
            display(solved)
            f.write(write(solved)+"\n")
            i = i + 1

    f.close()
    print ("Number of problems solved is: ", i)
    print ("Time taken to solve the puzzles is: ", time.time() - start)
```

Test the backtracking search with AC3 algortihm for preprocessing for euler puzzles

```
PS D:\Documents\IU - VNUHCM\ARTIFICIAL INTELLIGENCE\ArtificialIntelligence\Lab45\Lab45-Sudoku> python3 sudoku.py --inputFile data/euler.txt
The board -  1  takes  0.051653146743774414  seconds
After solving:
4  8  3   | 9  2  1   | 6  5  7
9  6  7   | 3  4  5   | 8  2  1
2  5  1   | 8  7  6   | 4  9  3
---------------------------------------
5  4  8   | 1  3  2   | 9  7  6
7  2  9   | 5  6  4   | 1  3  8
1  3  6   | 7  9  8   | 2  4  5
---------------------------------------
3  7  2   | 6  8  9   | 5  1  4
8  1  4   | 2  5  3   | 7  6  9
6  9  5   | 4  1  7   | 3  8  2
The board -  50  takes  0.05199766159057617  seconds
After solving:
3  5  1   | 2  8  6   | 4  9  7
4  9  2   | 1  5  7   | 6  3  8
7  8  6   | 9  3  4   | 5  1  2
---------------------------------------
2  7  5   | 4  6  9   | 1  8  3
9  3  8   | 5  2  1   | 7  6  4
6  1  4   | 8  7  3   | 2  5  9
---------------------------------------
8  2  9   | 6  4  5   | 3  7  1
1  6  3   | 7  9  2   | 8  4  5
5  4  7   | 3  1  8   | 9  2  6
Number of problems solved is:  50
Time taken to solve the puzzles is:  2.9530835151672363
```

Test the backtracking search with AC3 algortihm for preprocessing for magictour puzzles

```
The board -  95  takes  0.12813711166381836  seconds
After solving:
3  5  4   | 1  8  6   | 9  2  7
2  9  8   | 7  4  3   | 6  1  5
1  6  7   | 9  5  2   | 4  8  3
---------------------------------------
4  8  1   | 5  2  7   | 3  6  9
9  3  2   | 6  1  4   | 5  7  8
5  7  6   | 3  9  8   | 2  4  1
---------------------------------------
7  2  9   | 8  6  5   | 1  3  4
8  4  5   | 2  3  1   | 7  9  6
6  1  3   | 4  7  9   | 8  5  2
Number of problems solved is:  95
Time taken to solve the puzzles is:  32.01500630378723
```

4. Experience description:

It is a fascinating application of AI techniques. I am very interesting to see how the design options of variables, domains, and constraints can be used to represent the Sudoku puzzle as a CSP, and how the backtracking algorithm can be used to efficiently search for solutions. I find it a difficulty that can arise is choosing the right heuristics for variable and value selection, which can significantly affect the efficiency of the solution process. Overall, it was a fun and challenging assignment that showcases the power of AI in solving complex problems. In addition, the Assignment is difficult. When I initially looked at the specifications and the source code offered, I was confused. Just to get the gist of what to accomplish, I have to browse through theoretical presentations and web resources. I finally finish it after working with the lab together and using several examples

5. **Time spend:** it took me around 4 hours to complete the lab