HU22CSEN0100999
Set-2

INDEX

Name : Eshwar Deshmukh Chavan Roll No. HU22CSEN0100999

Class & Branch : AI APPLICATIONS

Assn. No.	Date	Unit Subject Name	Page	Date of Submission	Marks	Sign. of Faculty
1.		8 Puzzle Problem				
2.		WUMPUS WORLD Problem				
3.		VACUUM Cleaner Problem				
4.		SODUKU				
5		CROSSWORD PUZZLE				

8 - Puzzle Problem

```
import copy
from heapq import heapdpush, heapdpop

n= 3
row = [1, 0, -1, 0]
col = [0, -1, 0, 1]

Class priorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, k):
        heapdpush(self.heap, k)

    def pop(self):
        return heappppp(self.heap)

    def empty(self):
        return not self.heap

Class node:
    def __init__(self, parent, mat, empty_tile_pos, cost, level):
        self.parent = parent
        Self.mat = mat
        Self.empty_title_pos = empty_title_pos
        self.cost = cost
        self.level = level
```

```
    def __lt__(self, nxt):
        return self.cost < nxt.cost

def calculateCost(mat, final) -> int:
    count = 0
    for i in range(n):
        for j in range(n):
            if mat[i][j] and mat[i][j] != final[i][j]:
                count += 1
    return count

def newNode(mat, empty_tile_pos, new_empty_tilepos, level, parent, final) -> node:
    new_mat = copy.deepcopy(mat)
    x1, y1 = empty_tile_pos
    x2, y2 = new_empty_tilepos
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
    cost = calculateCost(new_mat, final)
    return node(parent, new_mat, new_empty_tile_pos, cost, level)

def printMatrix(mat):
    for i in range(n):
        for j in range(n):
            print("%d " % mat[i][j], end=" ")
        print()

def isSafe(x, y):
    return 0 <= x < n and 0 <= y < n
```

```
def printPath(root):
    if root is None:
        return
    printPath(root.parent)
    printMatrix(root.mat)
    print()

def solve(initial, empty_tile_pos, final):
    pq = priorityQueue()
    cost = calculateCost(initial, final)
    root = node(none, initial, empty_tile_pos, cost, 0)
    pq.push(root)
    while not pq.empty():
        minimum = pq.pop()
        if minimum.cost == 0:
            printpath(minimum)
            return
        for i in range(4):
            new_tile_pos = [minimum.empty_tile_pos[0] + row[i],
minimum.empty_title_pos[1] + col[i]]
            if issafe(new_tile_pos[0], new_tile_pos[1]):
                child = newNode(minimum.mat, minimum.empty_tile_pos,
new_tile_pos, minimum.level+1, minimum, final)
                pq.push(child)
initial = [[1, 2, 3], [5, 6, 0], [7, 8, 4]]
final = [[1, 2, 3], [5, 8, 6], [0, 7, 4]]
empty_tile_pos = [1, 2]
```

Solve(initial, empty_tile_pos, final)

OUTPUT:

1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4

AIM: Algorithm for 8 puzzle problem.

ACTUATORS: Tile movers, state updaters

Sensors: Tile Position sensors, Goal state checker

In 8-puzzle probem, there are 8-tiles which are numbered from 1 to 8 placed on 9-tile capacity square frame.

The objective of 8-puzzle problem is to transform the arrangement of tiles from Initial arragement to a goal arrangement.

The Initial and Goal arrangement is show in below figure.

There is always an empty slot in the Initial arrangement

legal moves are the moves in which the tile adjacent to empty slot are moved to elements

↳ left, Right, Up, Down

The Initial arrangement is called as Initial stage and goal arrangement is called as goal state.

The state space Tree for 8-puzzle is large because, there can be 8! different arrangement a partial state space is shown in below fig.

In the state space Tree, the nodes are numbered as per the level

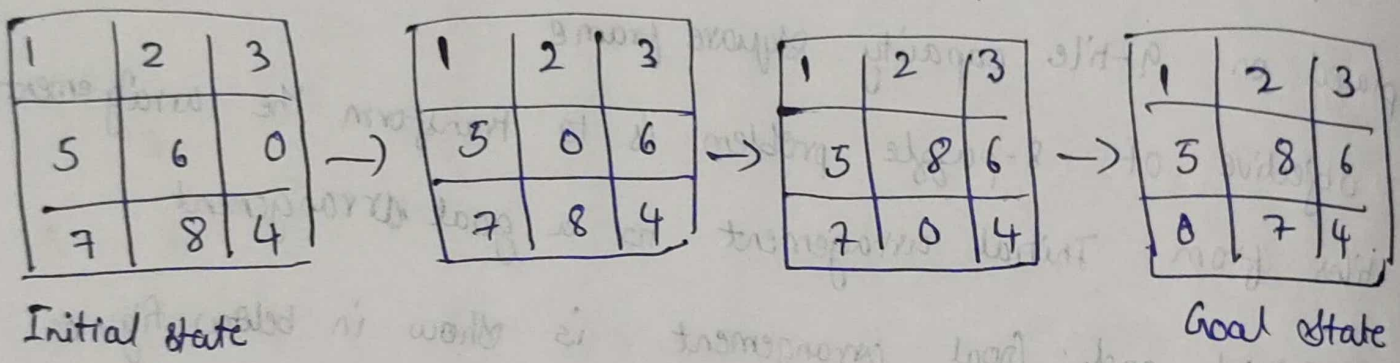
Each next move is generated based on empty slot positions

Edges are labelled according to the direction in which empty space moves

The least node becomes the E-node

we can decide which node to become an E-node based on estimation formula.

Example



Initial state

Goal state

OBSERVATIONAL ANALYSIS

Data structures:

1. Priority Queue : Manages nodes using a heap for efficient retrieval of the minimum cost node.
2. Node : Represents each state with attributes for the puzzle configuration, empty tile position, cost, and depth.

Methods:

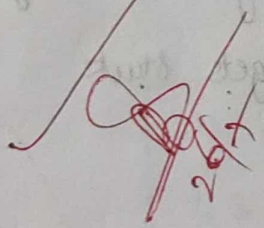
1. CalculateCosts : Counts misplaced tiles compared to the goal state
2. NewNodes : Creates a new node by swapping tiles and recalculating the cost.
3. printMatrix : Displays the puzzle matrix.
4. isSafe(x, y) : Checks if the tile position is within grid bounds.

5. printPath(root): Recursively prints the path from the root to the solution.

6. solve: Uses A search with a priority queue to find and print the solution path.

Implementation steps:

* Initialize the priority queue and root node
* Expand nodes by moving the empty tile and adding new nodes to the queue.
* Print the solution path when the goal state is reached.



WUMPUS WORLD PROBLEM

The Wumpus World is a grid-based environment used to illustrate the principles of knowledge-based agents and knowledge representation. Inspired by gregory Yob's 1973 game "Hunt the Wumpus", it consists of a 4×4 cave with 16 interconnected room.

Environment and Components:

Rooms: 16 rooms connected by passageways

Agent: Starts at room [1,1], initially facing right.

Wumpus: A beast in one room that eats the agent if they enter.

Pits: Bottomless rooms: the agent falls in and gets stuck

Gold: A single room contains gold.

Adjecent Indicators:

Stench: Indicate a nearby Wumpus.

Breeze: Indicate a nearby pit.

Glitter: Indicates gold in the current room.

Scream: Heard if the wumpus is killed.

Bump: Indicates walking into a wall.

Performance Measures:

Rewards: +1000 points for exiting with gold

Penalties: -1000 points for being eaten by the wumpus or falling into a pit

- 1 point per action ; 10 points for using an arrow.

Actuators:

Left Turn ; Right Turn ; Move forward ; Grab ; Release ; Shoot

Sensors:

Stench : Near Wumpus.

Breeze : Near pit

Glitter : In room with gold.

Bump : Hits wall

Scream : Wumpus is killed.

Example :

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
1,1 [A]	2,1	3,1	4,1

Symbols : A - Agent B - Breeze G - Glitter/Gold V - visited

P - pit W - Wumpus OK - Safe rooms

Agent's Exploration Strategies

① Initial Room [1, 1] : Safe, marked as OK

② Moves to [2, 1] : Perceives breeze, indicating nearby pit.

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2 P!	3,2	4,2
1,1 V OK	2,1 A OK B	3,1 P!	4,1

Pit maybe at [3,1] or [2,2]

(5) Move back to [1,1] : marks visited rooms.

(6) Move to [1,2] : Perceives stench, indicating nearby Wumpus.

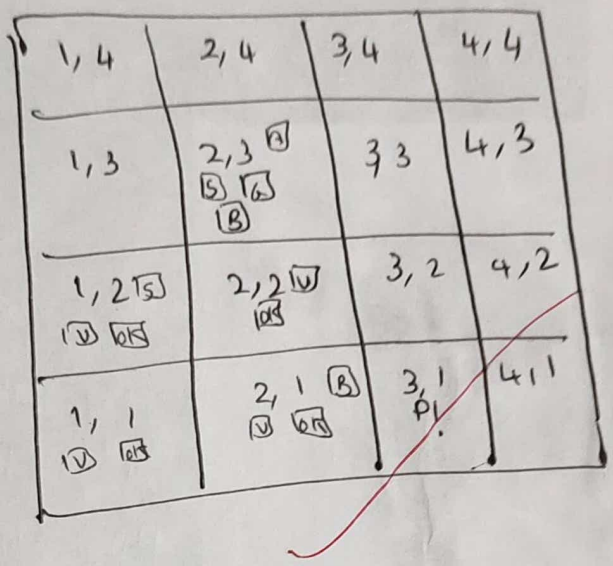
1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 A OK S	2,2 OK	3,2	4,2
1,1 V OK	B 2,1 OK V	3,1 P!	4,1

Wumpus cannot be in [2,2] as Agent had not selected any stench when he was at [2,1].

Therefore, agent enters that Wumpus is in the room [1,3] and in current state, there is no breeze which means in [2,2] there is no pit. So, it is marked OK.

(5) Move to [2,2] : Safe, no stench or breeze

(6) Move to [2,3] : finds gold, grabs it, and exits, the cave



Code:

```
import random

# Constants for the grid size
GRID_SIZE = 4

# Representations for different elements in the world
EMPTY = 0
PIT = 1
WUMPUS = 2
GOLD = 3
AGENT = 4

# Directions
UP, RIGHT, DOWN, LEFT = 0, 1, 2, 3

class WumpusWorld:
    def __init__(self):
        self.grid = [[EMPTY] * GRID_SIZE for _ in range(GRID_SIZE)]
        self.agent_position = [0, 0]
        self.agent_direction = RIGHT
        self.has_arrow = True
        self.has_gold = False

        # Place pits with a probability of 0.2
        for i in range(GRID_SIZE):
            for j in range(GRID_SIZE):
                if (i, j) != (0, 0) and random.random() < 0.2:
                    self.grid[i][j] = PIT

        # Place the Wumpus and gold
        self.grid[random.randint(1, GRID_SIZE-1)][random.randint(1, GRID_SIZE-
1)] = WUMPUS
        self.grid[random.randint(0, GRID_SIZE-1)][random.randint(0, GRID_SIZE-
1)] = GOLD

    def get_percepts(self):
        x, y = self.agent_position
        percepts = []

        # Check for stench (Wumpus nearby)
        if any(self.is_adjacent(x, y, WUMPUS)):
            percepts.append("Stench")

        # Check for breeze (Pit nearby)
        if any(self.is_adjacent(x, y, PIT)):
            percepts.append("Breeze")

        # Check for glitter (Gold in the same room)
```

```
        if self.grid[x][y] == GOLD:
            percepts.append("Glitter")

        return percepts

    def is_adjacent(self, x, y, element):
        adjacent = []
        if x > 0:
            adjacent.append(self.grid[x-1][y] == element)
        if x < GRID_SIZE-1:
            adjacent.append(self.grid[x+1][y] == element)
        if y > 0:
            adjacent.append(self.grid[x][y-1] == element)
        if y < GRID_SIZE-1:
            adjacent.append(self.grid[x][y+1] == element)
        return adjacent

    def move_forward(self):
        x, y = self.agent_position
        if self.agent_direction == UP and x > 0:
            self.agent_position[0] -= 1
        elif self.agent_direction == DOWN and x < GRID_SIZE-1:
            self.agent_position[0] += 1
        elif self.agent_direction == LEFT and y > 0:
            self.agent_position[1] -= 1
        elif self.agent_direction == RIGHT and y < GRID_SIZE-1:
            self.agent_position[1] += 1

    def turn_left(self):
        self.agent_direction = (self.agent_direction - 1) % 4

    def turn_right(self):
        self.agent_direction = (self.agent_direction + 1) % 4

    def grab_gold(self):
        x, y = self.agent_position
        if self.grid[x][y] == GOLD:
            self.has_gold = True
            self.grid[x][y] = EMPTY

    def shoot_arrow(self):
        if self.has_arrow:
            self.has_arrow = False
            # Simplified: Assume the Wumpus is in a direct line of sight and
is hit
            return "Scream"
        return None
```

```
# Simulation
def simulate():
    world = WumpusWorld()
    steps = 0
    actions = ["Move Forward", "Turn Left", "Turn Right", "Grab Gold", "Shoot
Arrow"]
    action_funcs = [world.move_forward, world.turn_left, world.turn_right,
world.grab_gold, world.shoot_arrow]

    while True:
        percepts = world.get_percepts()
        print(f"Step {steps}: Agent at {world.agent_position}, Facing
{world.agent_direction}")
        print("Percepts:", percepts)

        if "Glitter" in percepts:
            world.grab_gold()
            print("Action: Grab Gold")
            break

        # Simplified decision-making process
        if "Stench" in percepts and world.has_arrow:
            print("Action: Shoot Arrow")
            world.shoot_arrow()
        else:
            action = random.choice(action_funcs)
            action()
            print("Action:", actions[action_funcs.index(action)])

        steps += 1
        if steps > 100:  # Prevent infinite loop in case of unexpected issues
            break

imulate()
```

Output:

Step 0: Agent at [0, 0], Facing 1

Percepts: ['Breeze']

Action: Shoot Arrow

Step 1: Agent at [0, 0], Facing 1

Percepts: ['Breeze']

Action: Turn Right

Step 2: Agent at [0, 0], Facing 2

Percepts: ['Breeze']

Action: Move Forward

Step 3: Agent at [1, 0], Facing 2

Percepts: ['Stench', 'Breeze']

Action: Turn Left

Step 4: Agent at [1, 0], Facing 1

Percepts: ['Stench', 'Breeze']

Action: Grab Gold

Step 5: Agent at [1, 0], Facing 1

Percepts: ['Stench', 'Breeze']

Action: Move Forward

Step 6: Agent at [1, 1], Facing 1

Percepts: ['Breeze']

Action: Move Forward

Step 7: Agent at [1, 2], Facing 1

Percepts: ['Stench', 'Breeze']

Action: Move Forward

Step 8: Agent at [1, 3], Facing 1

Percepts: ['Glitter']

Action: Grab Gold

Observational Analysis

Constants and Representations

- **GRID_SIZE**: 4
- **Elements**: `EMPTY`, `PIT`, `WUMPUS`, `GOLD`, `AGENT`
- **Directions**: `UP`, `RIGHT`, `DOWN`, `LEFT`

Class: `WumpusWorld`

Initialization

- Creates a 4x4 grid.
- Agent starts at `(0, 0)` facing `RIGHT`.
- Randomly places pits with a 0.2 probability, and randomly places the Wumpus and gold.

Methods

- **`get_percepts`**: Returns percepts based on the agent's position ("Stench" for nearby Wumpus, "Breeze" for nearby pit, "Glitter" for gold in the same cell).
- **`move_forward`**: Moves the agent forward if within grid bounds.
- **`turn_left`** and **`turn_right`**: Rotate the agent.
- **`grab_gold`**: Picks up gold if in the current cell.
- **`shoot_arrow`**: Shoots an arrow if available.

Simulation (`simulate` function)

Initialization

- Creates a `WumpusWorld` instance.
- Defines actions and corresponding functions.

Main Loop

- Runs until the agent grabs the gold or exceeds 100 steps.
- Agent perceives surroundings, makes decisions, and acts:
  - Grabs gold if "Glitter" is perceived.
  - Shoots arrow if "Stench" is perceived and arrow is available.
  - Otherwise, performs a random action.

Methods and Data Structures

Methods

- Initialization, percept retrieval, movement, and actions.

Data Structures

- **Grid**: 4x4 2D list.
- **Agent State**: Position, direction, arrow, gold possession.
- **Percepts**: List of sensed environmental cues.

VACUUM CLEANER PROBLEM

CODE:-

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    loc = input("Enter Location of Vacuum: ")
    status = input("Enter status: ")
    otherstatus = input("Enter status of other room: ")

    if status == '1':
        print(f"Location {loc} is Dirty.")
        goal_state[loc] = '0'
        cost += 1
        print(f"Cost for CLEANING {loc} " + str(cost))

    if otherstatus == '1':
        otherloc = 'B' if loc == 'A' else 'A'
        print(f"Location {otherloc} is Dirty.")
        cost += 1
        print("Moving to other Location. Cost for moving " + str(cost))
        goal_state[otherloc] = '0'
        cost += 1
        print(f"Cost for CLEANING {otherloc}: " + str(cost))

    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

Output:

```
Enter Location of Vacuum: A
Enter status: 0
Enter status of other room: 1
Location B is Dirty.
Moving to other Location. Cost for moving 1
Cost for CLEANING B: 2
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

Aim: The aim is to simulate a vacuum cleaner agent that autonomously clears two rooms while minimizing the cost of actions.

Description:

The agent starts in either Room A or Room B, which may be dirty or clean. The goal is to clean both rooms efficiently.

Example Scenario:

Initial State: RoomA and RoomB are dirty, vacuum is RoomA,

Goal State: RoomA and RoomB are clean, vacuum can be in either room.

Actions:

Suck Dirt (if current room is dirty)

Move to the other room (if necessary)

Performance Evaluation

The performance of the vacuum cleaner agent is evaluated based on:

Search cost: The time taken by the agent to find the solution.

Path cost: The total cost of the actions taken by the agent to reach the goal state.

OBSERVATIONAL ANALYSIS

Methods Used:

Input Handling: The input fuction is used to taken the initial location of the vacuum cleaner and the status of each room fro

the user.

Conditional Statements: if statements are used to check the cleanliness status of each room and perform the necessary action (cleaning and moving).

Data Structures Used:

Dictionary: goal state is a dictionary used to store the cleanliness status of the rooms.

Integer: cost is an integer variable used to keep track of the total cost of actions.

STEP-BY-STEP EXECUTION:

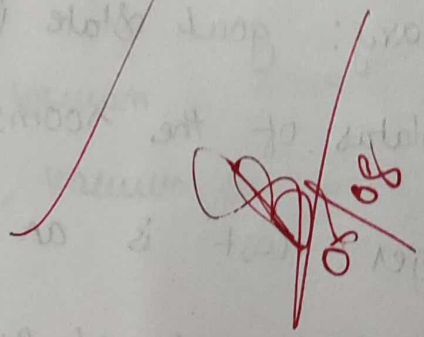
Initialization: The goal state dictionary is initialized with both rooms set to '0' (clean), and cost is initialized to 0.

User Input: The user is prompted to enter the initial location of vacuum cleaner and the status (clean or dirty) of both rooms.

Cleaning Current Room: If the current room is dirty (status == '1') the vacuum cleaner cleans it, updates the goal state, and increments the cost.

Moving and cleaning other Room: If the other room is dirty (otherStatus == '1'), the vacuum cleaner moves to the other room, cleans it, updates the goal state, and increments the cost for both moving and cleaning.

Output: The goal state and total performance cost are printed.



Sudoku

Aim: To solve a Sudoku puzzle by meeting all constraints, include unique numbers in rows, columns, 3×3 subgrids, and both main diagonals.

Description:
Sudoku is a 9×9 grid where digits 1 to 9 must appear once per row, column, 3×3 subgrid and both diagonals.

Actuator:
Algorithms such as backtracking or constraint propagation to fill the grid.

Sensors:
Grid Input: Initial values in the Sudoku grid.
Constraints checking: Mechanisms to verify if current values adhere to Sudoku rules.

Example:
Initial grid (Partial):

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

Solution Grid:

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

OBSERVATIONAL ANALYSIS

Methods:

- Backtracking: Place digits and recursively check constraints, backtrack if necessary.
- Constraint Propagation: Use techniques like naked and hidden singles to digit placement.
- Heuristics: choose the most constrained cell or least constraining digit

Data Structures:

- 2D Array: Represents the Sudoku grid.
- Sets: for checking used digits in rows, columns, and subgrids.
- Lists: Track possible values for cells.

STEP - BY - STEP Execution:

1. Parse Input: Convert the grid into a 2D-array.
2. Setup: Initialize structures for constraints.
3. Digit Placement: Select and place digits while checking constrain
4. Recursive filling: Continue filling cells or backtrack if needed
5. completion: Ensure the grid is fully and correctly filled.

Sudoku Code And Output

```python
def is_valid(board, row, col, num):
    # Check if num is not in the row
    if num in board[row]:
        return False
    # Check if num is not in the column
    if num in [board[i][col] for i in range(9)]:
        return False
    # Check if num is not in the 3x3 subgrid
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(start_row, start_row + 3):
        for j in range(start_col, start_col + 3):
            if board[i][j] == num:
                return False
    return True

def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:  # Find an empty cell
                for num in range(1, 10):  # Try numbers 1-9
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = 0  # Undo assignment (backtrack)
                return False  # Trigger backtracking
    return True  # Solution found

# Example Sudoku puzzle (0 represents empty cells)
sudoku_board = [
    [5, 0, 0, 0, 0, 0, 0, 0, 8],
    [0, 0, 0, 0, 6, 0, 0, 0, 0],
    [0, 4, 0, 0, 0, 0, 0, 2, 0],
    [0, 0, 8, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 3, 0, 9, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 4, 0, 0],
    [0, 7, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 5, 0, 0, 0, 0],
    [9, 0, 0, 0, 0, 0, 0, 0, 1]
]

if solve_sudoku(sudoku_board):
    for row in sudoku_board:
        print(row)
else:
    print("No solution exists")
```

Python

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

CROSSWORD PUZZLE CODE AND OUTPUT

```
def can_place_horizontally(grid, word, row, col):
    if col + len(word) > 10:
        return False
    for i in range(len(word)):
        if grid[row][col + i] not in ('-', word[i]):
            return False
    return True

def can_place_vertically(grid, word, row, col):
    if row + len(word) > 10:
        return False
    for i in range(len(word)):
        if grid[row + i][col] not in ('-', word[i]):
            return False
    return True

def place_word(grid, word, row, col, direction):
    positions = []
    for i in range(len(word)):
        if direction == 'H':
            grid[row][col + i] = word[i]
            positions.append((row, col + i))
        else:
            grid[row + i][col] = word[i]
            positions.append((row + i, col))
    return positions

def remove_word(grid, positions):
    for row, col in positions:
        grid[row][col] = '-'

def solve_crossword(grid, words, index):
    if index == len(words):
        return True

    word = words[index]
    for row in range(10):
        for col in range(10):
            if can_place_horizontally(grid, word, row, col):
                positions = place_word(grid, word, row, col, 'H')
                if solve_crossword(grid, words, index + 1):
                    return True
                remove_word(grid, positions)

            if can_place_vertically(grid, word, row, col):
                positions = place_word(grid, word, row, col, 'V')
                if solve_crossword(grid, words, index + 1):
                    return True
                remove_word(grid, positions)

    return False

def crossword_solver(grid, words):
    grid = [list(row) for row in grid]
    if solve_crossword(grid, words, 0):
        return [''.join(row) for row in grid]
    return []

# Example usage:
grid = [
    "----------",
    "----------",
    "----------",
    "----------",
    "----------",
    "----------",
    "----------",
    "----------",
    "----------",
    "----------"
]
words = ["CIVICS", "HISTORY", "MATHS", "GEOGRAPHY", "PHYSICS", "CHEMISTRY"]

solved_grid = crossword_solver(grid, words)
for row in solved_grid:
    print(row)
```

```
+++++++++C
P++++++++H
HISTORY++E
Y++++++++M
S++++++++I
I++++MATHS
CIVICS+++T
S++++++++R
+GEOGRAPHY
++++++++++
```

CROSSWORD PUZZLE

Aim: To solve a 10×10 crossword puzzle by filling in a list of provided words into the grid while respecting the constraints of fixed '+' cells and modifiable '-' cells.

Description:

A crossword grid and a list of words are provided. The task is to fill the grid with these words either horizontally or vertically, ensuring that words fit appropriately without modifying '+' cells.

Example:

Input:

```
+ + + + + + + + + +
- + + + + + + + + -
- - - - - - + + -
- + + + + + + + + -
- + + + + + + + + -
- + + + + - - - - -
- - - - - + + + -
- + + + + + + + + -
+ - - - - - - - - -
+ + + + + + + + + +
```

Output:

```
+ + + + + + + + + C
P + + + + + + + + H
H I S T O R Y + + E
Y + + + + + + + + M
S + + + + + + + + T
I + + + + M A T H S
C I V I C S + + + T
S + + + + + + + R
+ G E O G R A P H Y
+ + + + + + + + + +
```

Actuators: ① Grid Modification: Placing words into the grid

② Recursive Backtracking: Removing words from the gri
to alternative placements.

Sensors:

- Grid state checker: Verifying whether a word can be placed in a specific location without conflicts.
- word List Tracker: Monitoring which words have successfully placed and which remain to be placed.

OBSERVATIONAL ANALYSIS

Methods

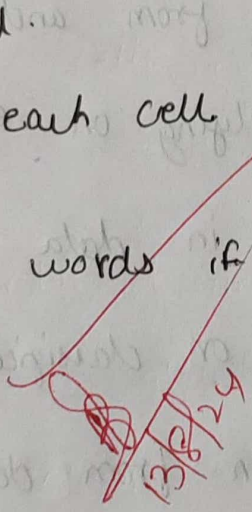
1. Can-place horizontally: Check if a word can be place horizontally at a given position
2. Can place vertically: Check if a word can be placed vertically at a given position.
3. place_word: place a word in the grid
4. remove word: Remove a word from the grid for backtracking
5. solve_crossword: Recursively attempt to place each word, backtracking as needed.

Data Structures

- Grid: 2D list representing the crossword grid.
- word List: Array of words to be placed in the grid.

Step - by - Step Execution

- Initialize the grid and word list.
- Use the recursive function solve crossword to place words in the grid.
- check each cell for horizontal and vertical placement of words.
- place words if they fit, and backtrack if they don't.



Conceptual Introduction To Machine Learning

Machine Learning(ML): is a branch of AI focused on developing algorithms that allow computers to learn from and make decisions based on data. Instead of relying on explicit programming. ML models identify patterns in data to improve their performance on tasks like prediction or classifications. The key objective is to enable machine to learn from data and make accurate decisions or predictions across various domains.

ML can be broadly classified into three types:

- Supervised Learning
- Unsupervised learning
- Semisupervised learning.

Supervised Learning

- use Labeled data to train models that predicts or classify outputs based on new inputs. The main types are:
  - → classification: Assign input to predefined categories (eg. spam detection in mails)

→ Regression: Predicts continuous values

Common Algorithms:

- linear Regression
- Decision Tree
- Neural Network

Applications:

- Image classification
- predictive Analytics
- Sentiment Analytics

UNSUPERVISED LEARNING

- Deals with unlabeled data, aiming to uncover hidden structures or patterns. The main tasks include:

→ clustering: grouping similar data points

→ Dimensionlity Reduction: Simplifying data while preserving important features.

Common Algorithms:

- K-means clustering
- Principal component Analysis (PCA)

Applications:

- Anomaly Detection
- Market Basket Analysis
- Data compression

SEMI SUPERVISED LEARNING

- Combines a small amount of labeled data with a large amount of unlabeled data, enhancing model performance when labeling data is costly or limited. It leverages the labeled data to guide the learning process on the unlabeled data.

Common Algorithms:

- Self Training
- Graph-Based Methods.

Applications:

- Text classification
- Image Recognition
- Medical Image Analysis.