|  |  |
| --- | --- |
| **PROJECT REPORT** | |
| **Project Title:** Design an Application that Solves Sudoku Puzzles Efficiently and Serves as a Teaching Aid for CSP Techniques using First-Order Logic and Backtracking | |
| **Problem Statement:**  Sudoku is a popular logic-based number placement puzzle. The challenge is to fill a 9x9 grid with digits from 1 to 9 so that each row, column, and 3x3 subgrid contains all the digits exactly once. While solving manually can be enjoyable, it becomes complex and time-consuming for difficult puzzles.  The goal of this project is to develop an intelligent application that:   * Solves Sudoku puzzles efficiently using artificial intelligence techniques. * Demonstrates the use of Constraint Satisfaction Problem (CSP) models and First-Order Logic (FOL) in solving real-world problems. * Provides interactive features such as hints and validation to help users understand the solving process, making it useful as a teaching aid. | |
| **Introduction:**  This project introduces a GUI-based Sudoku solver that combines Python (for the interface using PyQt6) and Prolog (for solving using logical rules). The solving strategy is built using CSP modeling and backtracking search in Prolog. The application allows users to interactively solve Sudoku puzzles, request hints, validate their input, and observe the AI solver in action.  The core idea is to model the puzzle in terms of variables (cells), domains (numbers 1 to 9), and constraints (distinct numbers in rows, columns, and subgrids). Using Prolog's constraint logic programming (clpfd), the application solves the puzzle by logically deducing valid number placements. | |
| **Flowchart & Concepts used in the Code:**  **Concepts Used**   1. **Constraint Satisfaction Problem (CSP):** The puzzle is framed as a CSP where each cell is a variable with a domain of values 1 through 9, and constraints ensure that each row, column, and block contains distinct values. 2. **First-Order Logic (FOL):** Constraints are declared using logical rules. For example, all rows must have unique values, expressed in Prolog as all\_distinct(Row). 3. **Backtracking Search:** Prolog systematically tries values for each cell and backtracks when it hits a constraint violation, ensuring it explores only valid possibilities. 4. **Constraint Logic Programming over Finite Domains (CLP(FD)):** The clpfd library in Prolog efficiently enforces constraints and supports domain reduction before search begins. 5. **Graphical User Interface (GUI):** Developed using PyQt6 in Python, the GUI allows users to view the puzzle, interact with cells, get hints, and observe solver behavior.   **Flowchart:** | |
| **Code:**  **sudoku\_solver.pl**  :- use\_module(library(clpfd)).  % Replace 0s with fresh variables  sudoku(Puzzle, Solution) :-      maplist(replace\_zeros, Puzzle, PuzzleVars),      Solution = PuzzleVars,      append(Solution, Vars), Vars ins 1..9,      maplist(all\_distinct, Solution),      transpose(Solution, Columns),      maplist(all\_distinct, Columns),      blocks(Solution),      maplist(label, Solution).  replace\_zeros([], []).  replace\_zeros([0|T1], [\_|T2]) :- replace\_zeros(T1, T2).  replace\_zeros([H|T1], [H|T2]) :- H \= 0, replace\_zeros(T1, T2).  blocks([]).  blocks([A,B,C|Rest]) :-      block3(A, B, C),      blocks(Rest).  block3([], [], []).  block3([A,B,C|R1], [D,E,F|R2], [G,H,I|R3]) :-      all\_distinct([A,B,C,D,E,F,G,H,I]),      block3(R1, R2, R3).  solve(Puzzle) :-      sudoku(Puzzle, Solution),      print\_solution(Solution).  print\_solution([]).  print\_solution([Row|Rest]) :-      format('~w~n', [Row]),      print\_solution(Rest).  **sudoku\_gui.py**  import sys  import subprocess  import ast  from PyQt6.QtWidgets import (      QApplication, QWidget, QGridLayout, QLineEdit, QPushButton, QLabel, QMessageBox, QVBoxLayout, QHBoxLayout  )  from PyQt6.QtGui import QFont  from PyQt6.QtCore import Qt  PUZZLES = [      [          [5,3,0, 0,7,0, 0,0,0],          [6,0,0, 1,9,5, 0,0,0],          [0,9,8, 0,0,0, 0,6,0],          [8,0,0, 0,6,0, 0,0,3],          [4,0,0, 8,0,3, 0,0,1],          [7,0,0, 0,2,0, 0,0,6],          [0,6,0, 0,0,0, 2,8,0],          [0,0,0, 4,1,9, 0,0,5],          [0,0,0, 0,8,0, 0,7,9]      ],      [          [0,2,0, 6,0,8, 0,0,0],          [5,8,0, 0,0,9, 7,0,0],          [0,0,0, 0,4,0, 0,0,0],          [3,7,0, 0,0,0, 5,0,0],          [6,0,0, 0,0,0, 0,0,4],          [0,0,8, 0,0,0, 0,1,3],          [0,0,0, 0,2,0, 0,0,0],          [0,0,9, 8,0,0, 0,3,6],          [0,0,0, 3,0,6, 0,9,0]      ]  ]  class SudokuGUI(QWidget):      def \_\_init\_\_(self):          super().\_\_init\_\_()          self.setWindowTitle("AI Sudoku Solver")          self.setFixedSize(700, 800)          self.grid = QGridLayout()          self.grid.setSpacing(0)          self.grid.setContentsMargins(0, 0, 0, 0)          self.puzzle\_index = 0          self.cells = [[QLineEdit() for \_ in range(9)] for \_ in range(9)]          self.solution = None          self.default\_cells = set()          self.setStyleSheet("background-color: white; color: black;")          self.init\_ui()          self.load\_puzzle(PUZZLES[self.puzzle\_index])      def init\_ui(self):          title = QLabel("Sudoku with AI Solver")          title.setFont(QFont("Arial", 20))          title.setAlignment(Qt.AlignmentFlag.AlignCenter)          title.setStyleSheet("color: black;")          for i in range(9):              for j in range(9):                  cell = self.cells[i][j]                  cell.setFixedSize(60, 60)                  cell.setAlignment(Qt.AlignmentFlag.AlignCenter)                  cell.setFont(QFont("Arial", 16))                  cell.setStyleSheet("""                      QLineEdit {                          background-color: white;                          color: black;                          border: 1px solid black;                      }                  """)                  cell.textChanged.connect(self.validate\_cell(i, j))                  self.grid.addWidget(cell, i, j)          btn\_solve = QPushButton("Solve with AI")          btn\_hint = QPushButton("Get Hint")          btn\_submit = QPushButton("Submit")          btn\_new = QPushButton("New Game")          for btn in [btn\_solve, btn\_hint, btn\_submit, btn\_new]:              btn.setStyleSheet("background-color: white; color: black; border: 1px solid black;")          btn\_solve.clicked.connect(self.solve)          btn\_hint.clicked.connect(self.get\_hint)          btn\_submit.clicked.connect(self.submit)          btn\_new.clicked.connect(self.new\_game)          btn\_layout = QHBoxLayout()          for btn in [btn\_solve, btn\_hint, btn\_submit, btn\_new]:              btn\_layout.addWidget(btn)          layout = QVBoxLayout()          layout.setSpacing(20)          layout.setContentsMargins(30, 30, 30, 30)          grid\_container = QWidget()          grid\_container.setLayout(self.grid)          layout.addWidget(title)          layout.addWidget(grid\_container, alignment=Qt.AlignmentFlag.AlignCenter)          layout.addLayout(btn\_layout)          self.setLayout(layout)      def validate\_cell(self, i, j):          def inner():              if (i, j) in self.default\_cells:                  return              text = self.cells[i][j].text()              if text == "":                  self.cells[i][j].setStyleSheet("background-color: white; color: black; border: 1px solid black;")              else:                  try:                      value = int(text)                      if 1 <= value <= 9:                          if self.solution and value != self.solution[i][j]:                              self.cells[i][j].setStyleSheet("background-color: red; color: black; border: 1px solid black;")                          else:                              self.cells[i][j].setStyleSheet("background-color: white; color: black; border: 1px solid black;")                      else:                          self.cells[i][j].setStyleSheet("background-color: red; color: black; border: 1px solid black;")                  except ValueError:                      self.cells[i][j].setStyleSheet("background-color: red; color: black; border: 1px solid black;")          return inner      def load\_puzzle(self, puzzle):          self.default\_cells.clear()          for i in range(9):              for j in range(9):                  val = puzzle[i][j]                  cell = self.cells[i][j]                  if val != 0:                      cell.setText(str(val))                      cell.setReadOnly(True)                      cell.setStyleSheet("background-color: white; color: black; border: 1px solid black;")                      self.default\_cells.add((i, j))                  else:                      cell.setText('')                      cell.setReadOnly(False)                      cell.setStyleSheet("background-color: white; color: black; border: 1px solid black;")          self.solution = self.run\_prolog\_solver(puzzle)      def get\_current\_puzzle(self):          puzzle = []          for i in range(9):              row = []              for j in range(9):                  text = self.cells[i][j].text()                  try:                      num = int(text)                      row.append(num if 1 <= num <= 9 else 0)                  except ValueError:                      row.append(0)              puzzle.append(row)          return puzzle      def solve(self):          if self.solution:              self.load\_puzzle(self.solution)          else:              QMessageBox.warning(self, "Error", "No solution found!")      def get\_hint(self):          if self.solution:              for i in range(9):                  for j in range(9):                      if self.cells[i][j].text() == '':                          self.cells[i][j].setText(str(self.solution[i][j]))                          return          else:              QMessageBox.warning(self, "Error", "No solution found!")      def submit(self):          if not self.solution:              QMessageBox.warning(self, "Error", "Solution not available for validation.")              return          correct = True          for i in range(9):              for j in range(9):                  if (i, j) in self.default\_cells:                      continue                  user\_val = self.cells[i][j].text()                  correct\_val = str(self.solution[i][j])                  if user\_val != correct\_val:                      self.cells[i][j].setStyleSheet("background-color: red; color: black; border: 1px solid black;")                      correct = False                  else:                      self.cells[i][j].setStyleSheet("background-color: white; color: black; border: 1px solid black;")          if correct:              QMessageBox.information(self, "Success", "Congratulations! You solved the puzzle!")          else:              QMessageBox.warning(self, "Incorrect", "Solution is not correct.")      def new\_game(self):          self.puzzle\_index = (self.puzzle\_index + 1) % len(PUZZLES)          self.load\_puzzle(PUZZLES[self.puzzle\_index])      def run\_prolog\_solver(self, puzzle):          prolog\_query = f"solve({puzzle}), halt."          try:              result = subprocess.run(                  ['swipl', '-s', 'sudoku\_solver.pl', '-g', prolog\_query],                  capture\_output=True,                  text=True              )              if result.returncode != 0:                  print("Prolog Error:\n", result.stderr)                  return None              lines = result.stdout.strip().split('\n')              solution = [ast.literal\_eval(line) for line in lines if line.startswith('[')]              return solution          except Exception as e:              print("Error running Prolog:", e)              return None  if \_\_name\_\_ == '\_\_main\_\_':      app = QApplication(sys.argv)      gui = SudokuGUI()      gui.show()      sys.exit(app.exec()) | |
| **Expected Output:** | |
| **Conclusion:**  The Sudoku Solver application successfully integrates AI concepts like CSP, FOL, and Backtracking to solve a complex puzzle interactively. By using Prolog for logical reasoning and Python for GUI interaction, the system provides a clear example of how declarative logic can be applied to practical problems.  The tool not only serves to solve puzzles efficiently but also acts as a learning platform for students and enthusiasts to understand the inner workings of AI problem-solving techniques, especially constraint-based approaches. With interactive features like hints and validations, users gain a better appreciation of the logical structure behind each solution. | |
| **Marks Awarded:** | **Faculty Signature with Date:** |