Figure 1: University of Arkansas Logo

CSCE 46103/56103: Introduction to Artificial Intelligence

Homework #2: ”CSP, Game, RL Map coloring, TreasureHunt, Sudoku”

Submission Deadline: 11:59 PM, September 29th, 2025

Instructions

- **Written Format & Template:** Students can use either Google Doc or LaTeXfor writing the report.
- Write your full name, email address, and student ID in the report.
- Submission through BlackBoard.
- **Submissions:** Your submission should be a zip file containing your report with the screenshots of your outputs, and the source code including your implementation.
- **Name the zip file as `lastname_studentID.zip`**
- submission should be made via black board.
- **Policy:** Review the late days policy and include the total number of “Late Days” used in your report.

Problem 1: Constraints Satisfaction Problem [20 pts]

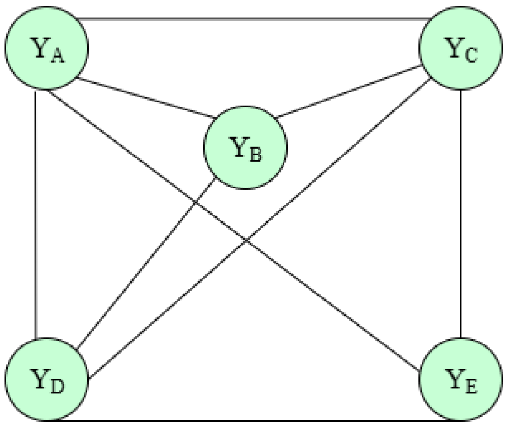


Figure 2: Constraint graph

We are given the constraint graph shown in **Figure 1**. There are five variables denoted as

$$\{Y_A, Y_B, Y_C, Y_D, Y_E\}.$$

The domains are:

$$D(Y_A) = \{10, 20, 30, 40, 50\},$$

$$D(Y_B) = \{10, 20, 30, 50\},$$

$$D(Y_C) = \{10, 20, 40, 50\},$$

$$D(Y_D) = \{10, 30, 40, 50\},$$

$$D(Y_E) = \{10, 30, 40, 50\}.$$

Constraints. Any two variables connected by an arc in the graph must take **different values**. All pairs of variables are connected **except** (Y_B, Y_E). For example, if $Y_B = 30$, then Y_A, Y_C, Y_D cannot take the value 30.

An assignment such as

$$\texttt{assign}(Y_A = 30, Y_E = 40)$$

means Y_A is assigned 30 first, followed by assigning 40 to Y_E, with other variables unassigned.

Questions

(a) What are the neighbors of Y_D in the constraint graph?

(b) Suppose we are performing **backtracking** with $\texttt{assign}(Y_A = 10, Y_B = 20)$. If the next chosen variable is Y_D, what are the possible values that could be assigned to Y_D?

(c) Suppose we are performing **backtracking with forward checking** and $\texttt{assign}(Y_A = 10, Y_B = 20)$. What are the reduced domains of all the variables?

(d) Suppose we are given reduced domains:

$$D(Y_C) = \{20\}, \qquad D(Y_E) = \{10\}$$

and we apply `assign`$(Y_C = 20, Y_E = 10)$. If we enforce **arc consistency** on all arcs, what are the resulting reduced domains of the other variables?

(e) Suppose we are doing backtracking, and the current assignment is `assign`$(Y_A = 10, Y_B = 20, Y_D = 40, Y_E = 50)$. The next variable to assign is Y_C. Should we continue or backtrack? Why?

Problem 2 Sudoku [pts 20]

You are given a partially completed Sudoku puzzle. Your task is to complete the puzzle by writing a backtracking-based solver.

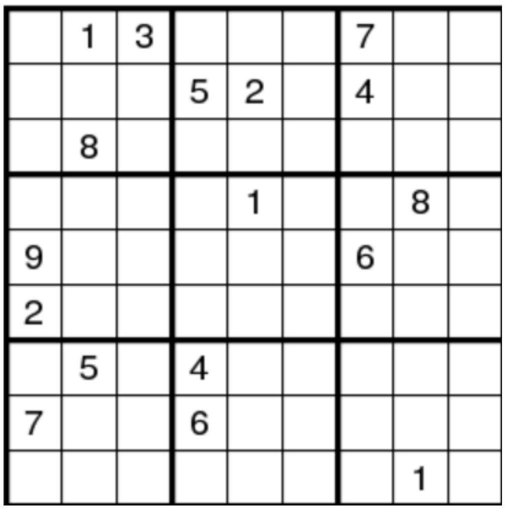
Sudoku is a 9×9 grid divided into 3×3 subgrids. The rules are:

- Each row must contain digits 1–9 without repetition.
- Each column must contain digits 1–9 without repetition.
- Each 3×3 box must contain digits 1–9 without repetition.

Implement the following helper functions:

- `find_empty(board)`: Finds the next empty cell (represented by 0). Returns the position as a tuple (row, col).
- `is_valid(board, num, pos)`: Checks whether placing `num` at position `pos` is valid under Sudoku rules.
- `solve_sudoku(board)`: Solves the puzzle using recursive backtracking.

The board is represented as a 2D list (9x9) of integers.



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

Figure 3: Sudoku puzzle

Use the image provided in the problem to fill in the `sudoku_board` variable.

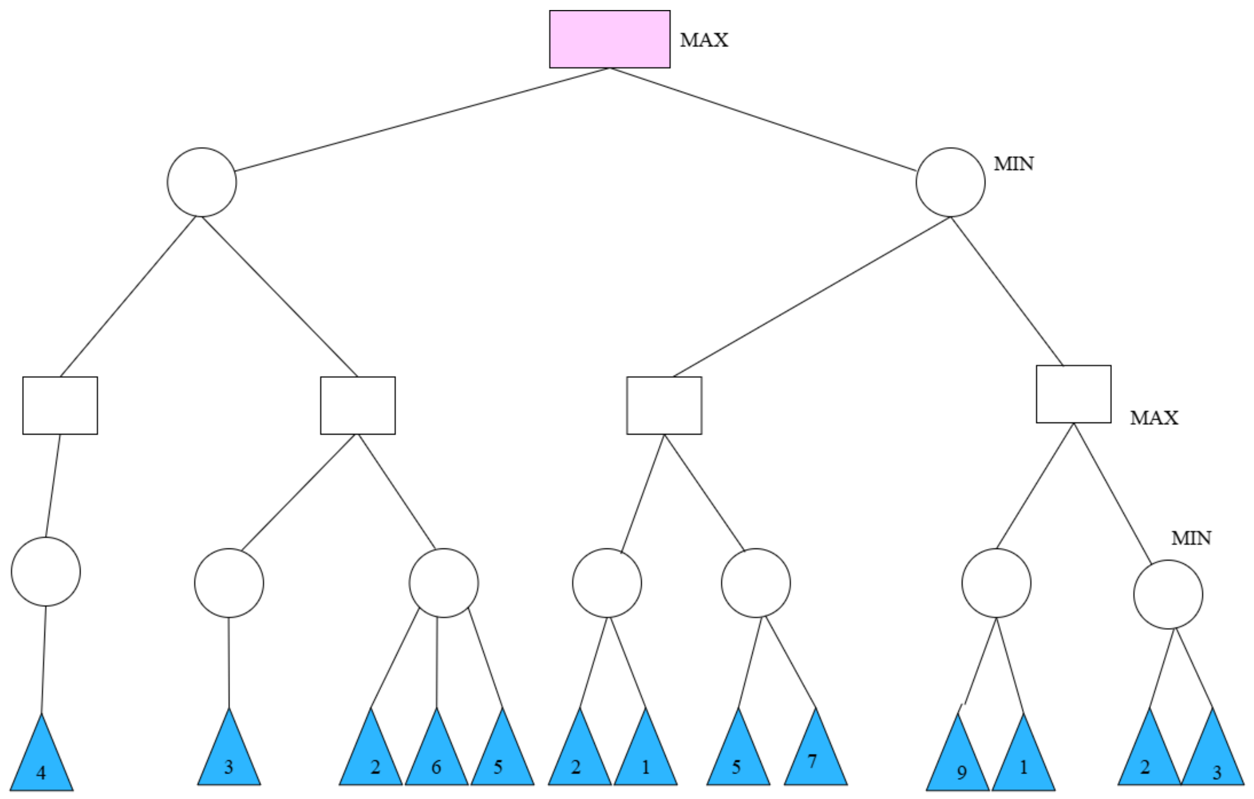
Your solution should solve the puzzle **in-place** and print the completed board.

The starter code is given below:

```
# Function to print the Sudoku board
def print_board(board):
    for i in range(len(board)):
        if i % 3 == 0 and i != 0:
            print("-" * 21)
        for j in range(len(board[0])):
            if j % 3 == 0 and j != 0:
                print("|", end="␣")
            if board[i][j] == 0:
                print(".", end="␣")
            else:
                print(board[i][j], end="␣")
        print()

# Find the next empty cell (denoted by 0)
def find_empty(board):
    # YOUR CODE HERE
    return None

# Check if the current value is valid at the given position
def is_valid(board, num, pos):
    # YOUR CODE HERE
    pass

# Main backtracking solver
def solve_sudoku(board):
    # YOUR CODE HERE
    pass

# Sudoku Puzzle (use the image provided to fill in the board)
sudoku_board = [
    # Fill in this board using the values from the image.
]

# Solve and print
print("Initial␣Sudoku␣Puzzle:")
print_board(sudoku_board)

if solve_sudoku(sudoku_board):
    print("\nSolved␣Sudoku␣Puzzle:")
    print_board(sudoku_board)
else:
    print('No␣solution␣exists'.)
```

Problem 3: MinMax and Alpha-Beta Pruning [20 pts]

b. (10pts) Fill in each blank trapezoid and cross out the branches pruned by minimax + alpha-beta pruning. Process the tree from left to right. What are values of α and β?



Problem [4] Pacman Collects All Food [40pts]

In Assignment 1, we developed search algorithms that guided Pacman to his food. In this task, we will extend that work so Pacman gathers *all* food items in the maze. This homework depends on Assignment 1, since it expands on that foundation.

To succeed, Pacman must bump into the walls at least once but no more than twice.

The starter code for this assignment consists of several Python scripts, with their roles summarized below:

Filename	**Purpose**
`search.py`	Provides the search algorithms from Assignment 2.
`searchAgents.py`	Contains the search agent problems (you will modify this).
`pacman.py`	The main driver for the Pacman game.
`util.py`	Helpful data structures and utilities.
`game.py`	Defines how Pacman runs; includes `Agent`, `AgentState`, etc.
`graphicsDisplay.py`, `graphicsUtils.py`, `textDisplay.py`, `ghostDisplay.py`, `keyboardAgents.py`, `layout.py`	Used for rendering the game. Can be ignored.

Acknowledgment: The base framework originates from the *Introduction to AI* course at UC Berkeley.

I. Format of the Search Problem Class

```python
# search.py
class SearchProblem:
    def getStartState(self):
        util.raiseNotDefined()
    def isGoalState(self, state):
        util.raiseNotDefined()
    def getSuccessors(self, state):
        util.raiseNotDefined()
    def getCostOfActions(self, actions):
        util.raiseNotDefined()
    def isWall(self, state):
        util.raiseNotDefined()
```

The `SearchProblem` class is abstract and defines several essential methods:

- `getStartState(self)`: returns the initial state.
- `isGoalState(self, state)`: evaluates whether a state is a goal.
- `getSuccessors(self, state)`: generates successor states (returns list of $(nextState, action, cost)$).
- `getCostOfActions(self, actions)`: gives the total cost of a sequence of actions.

- `isWall(self, state)`: returns `True` if the state is a wall.

Since this is abstract, you are not allowed to alter or implement it. You may, however, define what a "state" represents (position or more).

Review the `PositionSearchProblem` class in `searchAgents.py` to see a concrete example.

II. Finding All Corners (10 points)

Now that you already have implemented search algorithms, extend them to handle a new challenge: `CornersProblem`.

```python
class CornersProblem(search.SearchProblem):
    def __init__(self, startingGameState):
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right,1), (right,top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0
        "*** YOUR CODE HERE ***"
```

Implement the following methods:

- `getStartState`
- `isGoalState`
- `isWall`
- `getSuccessors`
- `getCostOfActions` (provided)

The goal state is reached when Pacman has collected food at all four corners.

Testing:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

III. Corners Heuristic (10 pts)

For the corners challenge, A* search is more effective when paired with a good heuristic. Implement `cornersHeuristic` in `searchAgents.py`.

```python
def cornersHeuristic(state, problem):
    """
    state: current search state
    problem: CornersProblem instance
    Return a number that is a lower bound on the
    shortest path to the goal.
    """
    corners = problem.corners
    walls = problem.walls
    "␣YOUR␣CODE␣HERE␣"
    return 0
```

Testing:

```
python pacman.py -l mediumCorners -p SearchAgent \
-a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

IV. Eating All Food (10 pts)

Next, extend the search so Pacman consumes *all* food, not just corners. Food may appear anywhere. Use the provided `FoodSearchProblem`.

Implement `foodHeuristic`:

```python
def foodHeuristic(state, problem):
    position, foodGrid = state
    "***␣YOUR␣CODE␣HERE␣***"
    return 0
```

Testing:

```
python pacman.py -l trickySearch -p SearchAgent \
-a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

V. Suboptimal Search (10 pts)

Sometimes finding an optimal path is too expensive. Instead, implement a greedy approach that directs Pacman to the *nearest* dot.

Modify `findPathToClosestDot` in `ClosestDotSearchAgent`:

```python
class ClosestDotSearchAgent(SearchAgent):
    def findPathToClosestDot(self, gameState):
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()
```

Testing:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```