KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY
Deemed to be University
BHUBANESWAR-751024

Spring Semester 2024

Course Handout

1. **Course code** : CS 39002
2. **Course Title** : **Artificial Intelligence Laboratory (AI Lab.)**
3. **LTP Structure** :

L	T	P	Total	Credit
2	0	0	2	1

4. **Course Faculty** :

Name : Dr. Hitesh Mohapatra
Email ID : hiteshmahapatra@kiit.ac.in
Faculty Room : F104 First Floor, Block-C, Campus-14

5. **Course offered to the school:** School Computer Engineering

6. **Course Objective:**
   - To provide skills for designing and analyzing AI based algorithms.
   - To enable students to work on various AI tools.
   - To provide skills to work towards solution of real-life problems

General Instructions:
Install Python and use the VS Code IDE to implement the assignments given for AI Lab. or,

Use Google Colab to implement the assignments given for the AI Lab. Students must use their KIIT mail ID to create Google Colab Notebooks. Naming convention can be "RollNo_AssignmentNo.ipynb."

Provide a brief overview of searching techniques, adversarial search in the context of artificial intelligence.

Assignment-1: Maze Solver using BFS and DFS

Objective: Implement BFS and DFS to solve a maze.

Problem Statement: Given a grid-based maze where 0 represents walls and 1 represents walkable paths, find the shortest path from a start cell to an end cell.

Tasks:
- Use BFS to find the shortest path.
- Use DFS to explore all possible paths and report one valid path (not necessarily the shortest).
- Compare the number of nodes explored by BFS and DFS.

Assignment 2: Route Finder Using Bi-Directional BFS/DFS

Objective: Use Bi-directional BFS/DFS to solve a navigation problem.

Problem Statement: Represent a city map as a graph where intersections are nodes and roads are edges. Find the shortest path between two locations.

Tasks:
- Implement Bi-directional BFS to minimize the number of nodes explored.
- Compare the performance of Bi-directional BFS with standard BFS and DFS.
- Visualize the search process (e.g., using a library like networkx in Python).

Assignment 3: Search for Treasure using the Best-First Search

Objective: Use Best-First Search to find a treasure in a grid.

Problem Statement: The treasure is hidden in a grid, and each cell has a heuristic value representing its "closeness" to the treasure. Implement Best-First Search to locate the treasure.

Tasks:
- Use Manhattan distance as a heuristic.
- Implement the algorithm to always move to the most promising cell first (minimum heuristic value).
- Analyze how heuristic choice affects performance.

Assignment 4: Uniform Cost Search for Optimal Path

Objective: Implement Uniform Cost Search for a weighted graph.

Problem Statement: Given a weighted graph (e.g., a transportation network with travel costs), find the minimum-cost path between two nodes.

Tasks:
- Represent the graph as an adjacency list.
- Implement Uniform Cost Search to find the optimal path.
- Compare it with BFS for unweighted graphs.

Assignment 5: A Search for a Puzzle Solver*

Objective: Solve the 8-puzzle using A* search.

Problem Statement: The 8-puzzle involves sliding tiles to achieve a goal state. Use A* to solve it.

Tasks:
Define heuristic functions:

- H1: Number of misplaced tiles.
- H2: Sum of Manhattan distances of all tiles from their goal positions.
- Implement A* with both heuristics.
- Compare the performance of the two heuristics in terms of the number of nodes explored and solution depth.

Assignment 6: Path Planning for a Robot

Objective: Use A* Search to find an optimal path for a robot navigating a 2D grid.

Problem Statement: A robot must move from a start point to a goal in a grid while avoiding obstacles.

Tasks:
Implement A* with:

- The Manhattan distance heuristic applies to grids without any diagonal movement.
- The Euclidean distance heuristic is applicable to grids that allow diagonal movement.
- Use a plotting library to visualize the found path.
- Compare A* with BFS and Uniform Cost Search.

Assignment 7: Game AI Using Search Algorithms

Objective: Implement AI to solve a simple turn-based game.

Problem Statement: Design an AI agent to play a game (e.g., Tic-Tac-Toe or Snake and Ladder) using search algorithms.

Tasks:

- Use BFS and DFS for exploring game states.
- Implement A* Search with a heuristic function to improve efficiency.
- Compare search strategies for different game board configurations.

Assignment 8: Navigation with Multiple Goals

Objective: Solve a problem where multiple goals exist using search algorithms.

Problem Statement: A robot in a grid needs to collect items (goals) before reaching an exit. Each goal has a different priority or cost.

Tasks:
- Use BFS/DFS for simpler scenarios (unweighted goals).
- Implement A* or Uniform Cost Search for weighted scenarios.
- Analyze the trade-offs between path length and goal priority.

Assignment 9: AI Planner Using A for Task Scheduling*

Objective: Use A* Search to optimize task scheduling.

Problem Statement: A set of tasks with dependencies and durations needs to be scheduled to minimize total time.

Tasks:
- Represent tasks and dependencies as a directed graph.
- Use A* Search when the heuristic estimates the remaining tasks' duration.
- Compare results with a greedy algorithm.

Assignment 10: Comparative Study

Objective: Evaluate and compare different search algorithms.

Problem Statement: Given a domain (e.g., pathfinding, puzzle solving), evaluate BFS, DFS, Bi-directional BFS, Uniform Cost Search, Best-First Search, and A* Search.

Tasks:
Analyze:
- Efficiency: Nodes explored, time taken.
- Optimality: Whether the solution is optimal.
- Create visualizations to compare algorithms

Assignment 11: Implementation of local optimization techniques, such as Hill Climbing, for solving AI-based search problems.

Assignment 12: Implementation of Genetic Algorithm (GA) for solving AI-based search problems.

Assignment 13: Solving the Water-Jugs Problem Using Adversarial Search

Develop a competitive version of the Water-Jugs problem where two players take turns making moves. Implement the Minimax algorithm to determine the optimal strategy for the AI player. Extend the solution by incorporating alpha-beta pruning for improved efficiency.

Problem Setup:
You are given two jugs of capacities A liters and B liters, and a target volume T liters. Two players alternately make moves, with the objective of the game being to reach

exactly T liters in any jug. The first player to achieve this wins. If no valid moves are left, the game ends in a draw.

Rules for Valid Moves:

1. Fill either jug completely.
2. Empty either jug completely.
3. Pour water from one jug into the other until one is empty or the other is full.

Game Design:

1. Implement the Water-Jugs problem as a two-player game.
2. Represent the game state as a tuple (x,y)(x, y)(x,y), where xxx and yyy are the current volumes in the two jugs.

Minimax Implementation:

1. Use the Minimax algorithm to evaluate all possible moves for both players.
2. Define a utility function:

    - Positive scores for states closer to the target volume TTT for the AI player.
    - Negative scores for states closer to the target for the opponent.
    - Zero for states with no valid moves left.

Alpha-Beta Pruning Integration:

1. Enhance the Minimax implementation by adding alpha-beta pruning to improve efficiency.

Player Interaction:

1. Allow a human player to compete against the AI agent.
2. Alternate turns between the human player and the AI.

Testing and Analysis:

1. Test the AI's performance against a human opponent and analyze its strategies.
2. Compare execution times for the Minimax algorithm with and without alpha-beta pruning.

Deliverables:

1. A Python program implementing the game with:

    - Minimax algorithm.
    - Alpha-beta pruning.

2. A brief report explaining:

    - The rules and structure of the game.
    - Design choices for the utility function.
    - Results of testing the AI's performance.

Assignment 14: Assignment: Solving the 4-Queens Problem Using Adversarial Search

Design a competitive version of the 4-Queens problem where two players alternately place queens on a 4x4 chessboard. Implement the Minimax algorithm to strategize queen placements and incorporate alpha-beta pruning for efficient decision-making.

Problem Setup:
The game involves two players (AI and Human) alternately placing queens on a 4x4 chessboard. The goal is to place queens such that no two queens threaten each other. The player unable to make a valid move loses the game.

Rules for Valid Moves:

1. A queen can be placed in any cell of the chessboard, provided it is not attacked by any previously placed queen.
2. A queen attacks cells in the same row, column, and diagonals.
3. Players take turns placing one queen at a time.
4. The game ends when no valid moves are left or all 4 queens are placed.

Game Design:

1. Represent the chessboard as a 4x4 grid.
2. Design functions to check valid placements and update the board.

Minimax Implementation:

1. Use the Minimax algorithm to evaluate all possible moves for both players.
2. Define a utility function:
    1. Positive scores for AI-favorable positions.
    2. Negative scores for opponent-favorable positions.
    3. Zero for neutral or draw positions.

Alpha-Beta Pruning Integration:

1. Optimize the Minimax implementation with alpha-beta pruning.

Player Interaction:

1. Alternate turns between the human player and the AI agent.
2. Display the board after each move for visualization.

Testing and Analysis:

1. Test the AI's performance against a human opponent.
2. Compare execution times for the Minimax algorithm with and without alpha-beta pruning.

Deliverables:

1. A Python program implementing the 4-Queens game with:
    1. Minimax algorithm.
    2. Alpha-beta pruning.
2. A brief report explaining:
    1. The rules and structure of the game.
    2. Utility function design.
    3. Results of testing the AI's performance.

Assignment 15: Solving the Tower of Hanoi Problem Using Adversarial Search

Transform the traditional Tower of Hanoi puzzle into a competitive two-player game. Implement adversarial search strategies using the Minimax algorithm to solve the problem. Enhance efficiency by incorporating alpha-beta pruning.

Problem Setup:

The Tower of Hanoi involves three rods and a set of N disks of decreasing size stacked on the first rod. The goal is to move all the disks to the last rod following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or an empty rod.
3. Moves alternate between two players (AI and Human).

The player who moves the largest disk to the target rod wins. If no valid moves are left and the game hasn't ended, it is considered a draw.

Game Design:

1. Represent the rods and disks using lists (e.g., rods = [[3, 2, 1], [], []] for N=3N=3N=3).
2. Create a function to validate moves and update the rods.

Adversarial Search:

1. Implement the Minimax algorithm to evaluate all possible moves for both players.
2. Define a utility function:
    - Assign positive scores for moves favorable to the AI (e.g., moving larger disks closer to the target rod).
    - Assign negative scores for moves favorable to the opponent.

Alpha-Beta Pruning:

1. Integrate alpha-beta pruning to optimize the search process by pruning unnecessary branches.

Player Interaction:

1. Alternate turns between the AI and the human player.
2. Display the state of the rods after each move.

End Condition:

1. The game ends when:
    - All disks are moved to the target rod.
    - No valid moves are left.
2. The winner is determined based on the size of the largest disk moved to the target rod by each player.

Deliverables:

1. A Python program implementing:
    - The Tower of Hanoi game mechanics.
    - Minimax algorithm.
    - Alpha-beta pruning optimization.
2. A brief report explaining:
    - The rules and structure of the game.
    - Utility function design.
    - Results of testing the AI's performance.

Assignment 16: Intelligent Solving of the 8-Puzzle Problem Using Heuristic Search

Implement and compare heuristic search strategies for solving the 8-Puzzle problem. Specifically, develop solutions using the **A*** algorithm and **Greedy Best-First Search (GBFS)**, leveraging heuristics like the Manhattan Distance or the number of misplaced tiles.

Problem Setup:

The **8-Puzzle** is a sliding puzzle consisting of a 3x3 grid with numbered tiles (1-8) and a blank space. The goal is to arrange the tiles in a specified order (usually numerical) by sliding tiles into the blank space, starting from a given configuration.

Representation of the Problem:

1. Model the puzzle as a 3x3 grid using a 2D list or a flat list of 9 elements.

2. Define the possible moves: Up, Down, Left, Right.
3. Represent the goal state as [1, 2, 3, 4, 5, 6, 7, 8, 0] (where 0 represents the blank space).

Heuristic Functions:

1. Implement heuristic functions to guide the search:
    1. **Manhattan Distance**: The sum of the absolute differences between the current and target positions of each tile.
    2. **Misplaced Tiles**: The number of tiles not in their correct positions.

Search Algorithms:

1. **A***: Combines the cost to reach a state (g(n)) with the heuristic estimate to the goal h(n).
2. **Greedy Best-First Search (GBFS)**: Considers only the heuristic value (h(n)) for each state.

Implementation Requirements:

1. Write a function to expand possible moves from a given state.
2. Implement a priority queue to explore states based on the chosen heuristic.
3. Ensure the solution handles unsolvable configurations gracefully.

Performance Evaluation:

1. Compare the performance of A* and GBFS in terms of:
    - Number of nodes expanded.
    - Solution optimality (path cost).
    - Execution time.
2. Test with multiple initial configurations.

Deliverables:

1. A Python program that:
    - Implements the 8-Puzzle problem with A* and GBFS.
    - Supports both heuristic functions.
2. A brief report discussing:
    - The implementation of heuristics.
    - Comparison of algorithms based on performance metrics.
    - Observations and conclusions.

Total Assignments: 16

Faculties are free to increase the number of assignments based on time and students' interest. However, the given assignments must not be reduced further (that is, less than 16).

Evaluation:

Aim is to examine student's understanding in problem solving using suitable AI concepts and implementation of the solution using Python.

Marks Distributions[$]:

1. Continuous Evaluation (daily basis) : 20 Marks
2. Lab Test-I (pre-midterm) and Lab Test-II (post-midterm) : 15 Marks
3. Quiz : 10 Marks
4. Viva : 05 Marks
5. Lab record hard-copy (hand written complete code) : 20 Marks
6. Sessional Test (External) : 30 Marks

[$]*There is a flexibility to change the marks ditribution based on requirements. However, theese components should be part of the overall 100 marks computation.*

References:

1. How to use Google Colab for Python: https://github.com/cserajdeep/Google-Colab-Helper
2. Python Coding Tutorial-1: https://www.youtube.com/watch?v=nLRL_NcnK-4
3. Python Coding Tutorial-1: https://www.youtube.com/watch?v=rfscVS0vtbw