Artificial Intelligence Practical File – Detailed Explanations

Experiment 1: Implement Tic■Tac■Toe using A* Algorithm

Aim: To implement intelligent move prediction in Tic■Tac■Toe using the A* search algorithm.

Theory:
Tic■Tac■Toe is a two■player zero■sum game. A* algorithm can be applied by representing each board configuration as a state and evaluating the best move using heuristic evaluation. A* uses the cost function: **f(n) = g(n) + h(n)** Where: • g(n): cost from start state to current state • h(n): heuristic estimating future winning chances For Tic■Tac■Toe, heuristic can be based on: • number of potential winning lines open • number of X or O in each row/column/diagonal • number of blocks preventing the opponent By expanding possible moves and selecting the minimum f(n), A* intelligently chooses the best move.

Algorithm:

1. Initialize an empty Tic■Tac■Toe board
2. Define a heuristic h(n) based on scoring patterns
3. For every possible move:
a. Generate successor board state
b. Compute g(n) as depth of move
c. Evaluate h(n)
d. Compute f(n) = g(n) + h(n)
4. Pick the move with lowest f(n)
5. Repeat until terminal state reached

Conclusion: A* successfully chooses optimal moves, making the Tic■Tac■Toe agent highly intelligent.

Experiment 2: 3 Missionaries and 3 Cannibals Problem using A*

Aim: To solve the classic state■space problem ensuring safety using A*.

Theory:
This problem involves transporting 3 missionaries and 3 cannibals across a river. Constraints: • Cannibals must never outnumber missionaries • Boat can carry max two persons • All moves must be legal A* explores combinations using heuristic: **h(n) = number_of_people_left_on_initial_side / boat_capacity** State representation: (M_left, C_left, boat_side) Illegal states are immediately discarded. A* guarantees the shortest valid sequence of crossings.

Algorithm:

1. Define the start state (3,3,left)
2. Define goal state (0,0,right)
3. Generate all legal successor states
4. For every successor compute:
g(n) = cost so far

h(n) = people remaining / 2
f(n) = g(n) + h(n)
5. Expand the lowest f(n)
6. Stop when reaching goal

Conclusion: A* provides the most efficient strategy while ensuring safety constraints are always met.

Experiment 3: 8■Puzzle Problem using A*

Aim: To solve the 8■puzzle using heuristic-based search.

Theory:
The 8■puzzle consists of 8 numbered tiles and one blank space. Heuristics used: **1. Misplaced Tiles Count 2. Manhattan Distance** (sum of distances of each tile from its goal location) A* is ideal because it finds the optimal least■cost path to solve the scrambled puzzle. Each move shifts the blank tile, generating new states.

Algorithm:

1. Input initial puzzle configuration
2. Represent blank moves as successor states
3. For every state calculate:
h1(n): misplaced tiles
OR
h2(n): Manhattan Distance
4. Compute f(n) = g(n) + h(n)
5. Expand the node with lowest f(n)
6. Stop when goal configuration is reached

Conclusion: A* efficiently solves the puzzle using powerful heuristics ensuring optimality.

Experiment 4: Implement Expert System

Aim: To design a rule■based expert system that mimics human reasoning.

Theory:
Expert systems use knowledge bases and inference engines. Components: • Knowledge base (rules, facts) • Inference engine (forward/backward chaining) • Working memory Process: IF condition THEN action rules define decisions. The expert system evaluates user input and matches rules to produce results.

Algorithm:

1. Define rules in IF■THEN format
2. Accept user input
3. Store input in working memory
4. Apply inference mechanism
5. Match rules and generate output

Conclusion: The expert system successfully demonstrates intelligent decision-making using rule-based reasoning.

Experiment 5: Implement Alpha■Beta Pruning

Aim: To optimize minimax search in adversarial games such as chess/tic■tac■toe.

Theory:
Minimax evaluates game states assuming two agents: MAX and MIN. Alpha■Beta pruning eliminates branches that cannot influence the final decision. • α = best score MAX can guarantee • β = best score MIN can guarantee If $\beta \le \alpha$, prune remaining branches.

Algorithm:

1. Apply minimax to root node
2. At each node track α and β
3. If $\alpha \ge \beta$ prune branch
4. Expand only useful branches
5. Return best scoring move

Conclusion: Alpha■Beta pruning reduces computation drastically while preserving optimality.

Experiment 6: Develop Elementary Chatbot

Aim: To create a basic conversational agent using pattern-matching.

Theory:
Chatbots rely on pattern■response rules: Example: User: “hi” Bot: “Hello! How can I help you?” Pattern-matching methods: • Keyword recognition • Template matching • Rule-based NLU System stores predefined input–output mappings.

Algorithm:

1. Define a set of patterns
2. Accept user input
3. Match with closest pattern
4. Generate predefined response
5. Continue till exit

Conclusion: The chatbot can carry simple conversations and demonstrates AI interaction basics.

Experiment 7: Goal■Stack Planning for Blocks World

Aim: To plan actions needed to achieve a final arrangement of blocks.

Theory:
Blocks World is a classic planning problem involving stacking blocks in a desired configuration. Goal■Stack Planning: • Push final goal onto stack • Decompose into subgoals • Apply operators (pickup, putdown, unstack, stack) • Maintain state integrity

Algorithm:

1. Define initial state and goal state
2. Push goal onto stack
3. Pop goal → create subgoals
4. Apply operators in correct sequence
5. Continue until stack empty

Conclusion: This method manages planning systematically, achieving efficient block rearrangement.

Experiment 8: Hill Climbing using Heuristic Search

Aim: To implement gradient■based local optimization.

Theory:
Hill Climbing selects the best immediate move based on heuristic scoring. Types: • Simple hill climbing • Steepest ascent • Stochastic hill climbing Limitation: May get stuck in local maxima or plateaus.

Algorithm:

1. Start at initial state
2. Evaluate neighbors
3. Move to the neighbor with best heuristic value
4. If no better neighbor exists → stop

Conclusion: Hill climbing is useful for optimization, although it may not always find global optimum.