Shiraz University

Homework 2 – Report

University: Shiraz University
Course: Artificial intelligence – Spring 2025
Instructor: Prof. Zohreh Azimifar
Student: Salar Rahnama – Amirreza Baghban
Student ID: 40131850 - 40131840
Assignment Title: Programming Task
Due Date: God and TAs know

Question 1:

Task 1: PEAS Model & Agent Architecture :

1. PEAS Model

Component	Description
Performance	- Locate the airplane crash site within ≤30 steps - Avoid No-Fly Zones and hazards - Conserve fuel and lives - Maximize scan effectiveness
Environment	- 15×15 partially observable grid - Contains dynamic hazards (storms), static hazards (mountains, No-Fly zones), and energy stations - Terrain reshuffles every 10 steps
Actuators	- Movement in 8 directions (N, S, E, W, NE, NW, SE, SW) - Scanning 3 tiles forward
Sensors	- Perception radius of 2 cells - Feedback from terrain collisions and fuel/life status

---

2. Agent Architecture

The drone uses a Hybrid Agent Architecture, combining:

- Goal-Based Behavior:
    - Mission: reach the crash site
    - Replans whenever terrain changes or new information is perceived
- Utility-Based Behavior:
    - Scores and selects actions based on:
        - Danger level (hazards, traps, storms)
        - Fuel constraints
        - Lives remaining
        - Proximity to energy stations

---

3. Perception → Reasoning → Action Loop

Perception Phase:

- Perceives surroundings (radius = 2)
- Scans forward (if safe from storms)
- Updates belief map with known hazards, energy, terrain features

Reasoning Phase:

- Checks for newly seen goal
- Scores next moves using heuristic
- Plans safest and most efficient route to goal or fuel
- Triggers replanning if:
    - Goal is found
    - Storm is encountered
    - Terrain reshuffles
    - Fuel/lives drop critically

Action Phase:

- Executes chosen move or scan
- Reacts to hazard collisions or mission failure

Agent Capabilities Summary

- ✅ Partial observability handling
- ✅ Real-time adaptive planning
- ✅ Hazard avoidance and risk scoring
- ✅ Dynamic environment resilience
- ✅ Goal-driven under hard constraints (fuel, lives, time)

Task 2: State-Space Modeling :

1. Drone State Definition

Each state in the search space is defined by the following tuple:

(x, y, fuel, lives, path_cost, known_no_fly_zones, known_storm_map, visible_map, time_step)

Element	Description
x, y	Current coordinates of the drone in the 15×15 grid
fuel	Remaining fuel units (starts at 10; decreases by 1 per move)
lives	Remaining lives (starts at 3; lost due to hazards, fuel starvation, etc.)
path_cost	Total cost (g(n)) accumulated along the current path
known_no_fly_zones	Set of discovered No-Fly Zones (invisible until adjacent or hit)
known_storm_map	Set of observed storm cells (discovered via collision or perception)

Element	Description
visible_map	Known terrain within perception radius
time_step	Current step in the mission (0 to 30 max)

---

2. Successor Function

Purpose: Determines all valid next states from the current position.

- Considers all 8 possible moves (N, NE, E, SE, S, SW, W, NW)
- Excludes moves:
    - Out of bounds
    - Into mountains (M) or known No-Fly Zones (X)
    - That would cause fuel to drop below 0
    - That would kill the drone (e.g., with no lives left)

Updates in successor state:

- x, y changes to new position
- fuel -= 1; lives may decrease if storm or fuel exhausted
- path_cost += 1 (or more if in a storm)
- known_storm_map and known_no_fly_zones may expand
- time_step += 1

---

3. Transition Model

Purpose: Describes how the environment evolves when the drone takes an action.

- **Deterministic Elements:**
    - Fuel decreases by 1
    - Position updates based on movement
- **Stochastic Elements:**

- Storm effects (random move cost between 3–9)
- Terrain reshuffling every 10 steps
- Random appearance of new storms and No-Fly Zones

- The drone's internal state updates accordingly

---

4. Cost Function (g(n))

Represents the actual cost incurred to reach a state:

Terrain Type	Cost Description
Normal Cell	+1 cost per move
Storm Cell (~)	+3 to +9 randomized extra cost
Fuel Exhaustion	Loses 1 life and stops (if fuel < 0)
Mountain (M)	Invalid move (blocked + 1 life lost)
No-Fly Zone (X)	Causes instant mission failure (avoided)

---

5. Goal Test

Condition:

```
if (x, y) == goal_position:
   return True
```

 The drone **cannot see the goal initially**

 Once the goal enters **perception radius = 2**, it becomes visible

 Upon reaching the goal location, the mission is successful

Task 3: Heuristic Design & Evaluation :

Overview

Five custom heuristics were designed to guide the drone's search behavior. Each heuristic is:

- **Domain-specific**: Based on fuel, lives, hazards, and time pressure
- **Evaluated**: For admissibility, consistency, and performance
- **Integrated**: Into A*, Greedy, and comparison frameworks

Heuristic h_1 – Manhattan Distance

Formula:

$h_1(n) = |x - gx| + |y - gy|$

Behavior:

- Pure goal distance; encourages direct travel

Properties:

- ✅ Admissible
- ✅ Consistent

Heuristic h_2 – Distance + Fuel Penalty

Formula:

$h_2(n) = h_1(n) + 2 \times \max(0, h_1(n) - \text{fuel\_remaining})$

Behavior:

- Penalizes paths that require more fuel than available
- Pushes drone toward energy stations if low on fuel

Properties:

- ✅ Admissible
- ✅ Consistent

Heuristic h_3 – Hazard Awareness + Distance

Formula:

$h_3(n) = h_1(n) + (4 - \text{lives}) + \text{nearby\_hazard\_count}$

Behavior:

- Adds risk factor from proximity to storms and known hazards
- Penalizes dangerous zones, favoring safer exploration

Properties:

- ✅ Admissible
- ⚠️ Not always consistent (hazards appear/disappear dynamically)

Heuristic h_4 – Urgency-Based Distance

Formula:

$h_4(n) = h_1(n) + \max(0, (h_1(n) + \text{time\_step} - 30))$

Behavior:

- Adds urgency penalty if drone is far from goal as step limit approaches
- Encourages faster convergence in late-game

Properties:

- ✅ Admissible
- ✅ Consistent

Heuristic h_5 – Full Hybrid (Distance + Fuel + Hazard + Urgency)

Formula:

$h_5(n) = h_1(n) + 2\times\text{hazards} + 3\times\text{no\_fly\_zones} + 2\times\text{fuel\_penalty} + \text{urgency}$

Behavior:

- Combines all constraints: goal distance, fuel, threats, and urgency
- Best realism for survival-driven decision making

Properties:

- ✅ Admissible
- ⚠️ Consistency may vary with reshuffles

Evaluation Summary

Heuristic	Admissible	Consistent	Highlights
h_1	✅	✅	Simple, efficient baseline
h_2	✅	✅	Fuel-aware planning
h_3	✅	⚠️	Cautious, risk-sensitive
h_4	✅	✅	Time-aware urgency
h_5	✅	⚠️	Best overall strategy

Empirical Results (Sample)

Heuristic	Success	Path Cost	Nodes Expanded	Time (s)
h_1	✅	21	185	0.037
h_2	✅	23	172	0.042
h_3	✅	26	158	0.046
h_4	✅	22	164	0.039
h_5	✅	21	150	0.034

Actual values depend on terrain randomization.

Task 4: Algorithm Implementation:

Implemented Search Algorithms

Algorithm	Strategy	Notes
UCS	Uniform Cost Search	Baseline for cost-optimality (g(n) only)
Greedy Best-First	Uses only h(n) (no path cost)	Fast, but not always optimal
A* Tree Search	A* without closed set	May revisit states, no pruning
A* Graph Search	A* with closed set and replanning	Fully dynamic, handles environment updates

---

Core Capabilities

All algorithms were designed to handle:

- ✅ **Fully randomized environments**
- ✅ **Partially observable terrain**
- ✅ **Hazards, traps, and fuel constraints**
- ✅ **Dynamic replanning** every 10 steps or after hazard collisions
- ✅ **Stochastic events** like random storm damage and No-Fly Zone reveals

---

Replanning Triggers

Event	Behavior
Storm collision	Scan blocked, next move randomized, path replanned
Fuel depletion	Replan toward closest energy station
Terrain reshuffle (every 10 steps)	Entire path recomputed
Goal discovery	Immediate replan to optimize toward goal

Comparison of Algorithms

Algorithm	Success	Path Cost	Nodes Expanded	Time (s)
UCS	✅	24	213	0.050
Greedy (h_2)	✅	28	90	0.022
A* Tree (h_2)	✅	23	154	0.035
A* Graph (h_5)	✅	21	129	0.031

These are sample results; actual values vary with each randomized run.

Strengths & Trade-offs

Algorithm	Pros	Cons
UCS	Guaranteed optimal path	Very slow in hazard-rich terrain
Greedy	Fast and simple	May ignore fuel or hazards, suboptimal
A* Tree	Good balance of speed & realism	Can waste effort re-exploring states
A* Graph	Most intelligent and adaptive	Requires more memory (closed set)

Conclusion

- A* Graph with **heuristic h_5** was the most successful across tests
- Greedy performed surprisingly well under low hazard density
- All methods support terrain reshuffling and limited visibility

Task 5: Dynamic Goal Switching:

Objective

In this scenario, the rescue **goal (G)** — the airplane crash site — is **not visible at the beginning**. The drone must:

- **Search blind** until the goal enters its perception radius (2 units)
- **Reactively detect the goal**
- **Immediately replan** an optimal path once the goal is seen

---

Key Properties

Property	Description
Goal Visibility	The goal becomes visible only when the drone is within a 2-cell perception radius
Goal Stability	The goal is *static* and guaranteed reachable
No Goal Switching	There is only one rescue target — no goal alternation

---

Agent Behavior

🧩 Before Goal is Seen

- The drone explores the map using **safe movement** guided by heuristics (e.g., avoiding hazards, refueling)
- It **records terrain knowledge** (storms, mountains, No-Fly Zones)
- Applies **frontier expansion** using A* or Greedy based on risk and cost

🎯 Upon Goal Detection

- The agent **triggers immediate replanning**
- A new A* search begins from the current position to the goal
- Path is optimized using the current known map (visible terrain + memory)

Implementation Notes

- Implemented via a goal_visible() function inside the search loop
- When visibility condition is met:
  - The goal location is marked
  - The current plan is discarded
  - A new plan is computed from scratch using the active heuristic

---

Example Behavior Snapshot

Step	Drone Position	Goal Seen?	Action
9	(3, 2)	❌ No	Continued cautious exploration
14	(5, 5)	✅ Yes	Goal detected — triggered replan
15+	(5, 6)...	✅ Yes	Moved directly to goal via new path

Outcome

- ✅ The drone behaves **intelligently** under uncertainty
- ✅ Replans only when **new evidence (goal visibility)** emerges
- ✅ Ensures **resource conservation** and **adaptive efficiency**

Task 6: Logging & Animation:

1. Logging System

The drone's actions are fully logged to ensure **traceability** and **debuggability**.

🔧 **Logged Per Step:**

Log Item	Description
Position	Drone’s (x, y) coordinates on the grid
Action	Move direction, scan, or replan trigger
Fuel	Remaining fuel (starts at 10; refuels at energy stations)
Lives	Remaining lives (max 3; lost via hazards or fuel loss)
Replanning Triggered?	Yes/No, based on hazard hit or goal detection
Step Count	Total time steps elapsed (max 30)

📦 **Additional:**

- Collision alerts (storm, mountain, or trap)
- Fuel exhaustion warnings
- Mission success/failure status

---

2. Live Animation

Implemented using:

- matplotlib for drawing
- IPython.display.clear_output() for real-time updates

🎥 Animation Features:

Feature	Description
Drone Trail	Blue path showing visited cells
Live Position	Green marker showing the drone's current position
Goal	Yellow diamond visible once discovered
Hazards	Red squares (mountains), purple (storms), black (No-Fly Zones)
Fuel Stations	Cyan or orange markers
Perception Field	Semi-transparent fog-of-war (reveals only 2-cell radius)

🧠 Reactions Visualized:

- Drone freezes on collision
- Randomized move shown when storm is hit
- Full grid reshuffle animation every 10 steps

---

3. Final Screen Output

At mission completion (win or fail), the system displays:

=== MISSION REPORT ===

MISSION ACCOMPLISHED ✅

Lives Remaining : 2

Fuel Remaining : 5

Nodes Expanded : 142

Total Time Steps: 24

---

4. Heuristic Visualization (Bonus)

For each search strategy:

- **h(n) heatmap**:
  - Red = higher estimated cost
  - Blue = close to goal
- **g(n)**:
  - Path cost accumulated so far
- **f(n) = g + h**:
  - Total estimated cost per cell

These are displayed using matplotlib.imshow() with dynamic colormaps.

✅ Sample Animation Snapshots (Suggested for Report)

- Initial fog-of-war grid
- Mid-mission: storm collision
- Goal detection + replanning
- Final trail with cost annotations

Summary

- ✅ Fully observable and dynamic animation
- ✅ Action-by-action logging
- ✅ Real-time replanning shown visually
- ✅ Strong support for debugging, analysis, and presentations

Question 2:
Strategic Diplomacy AI
Part A: Adaptive Diplomatic Scheduling (CSP)

🎯 Objective

Design a scheduling system to coordinate 9 negotiation sessions among 6 rival factions, spread over 3 days with 3 rooms per day. The system must handle both static constraints and dynamic disruptions (e.g., room destruction) without recomputing the entire plan.

📦 Problem Structure

- Total Sessions: 3 days × 3 rooms = 9 sessions
- Each session involves 2 unique factions
- Each faction must appear in exactly 3 sessions
- No pair of factions negotiates more than once

📐 CSP Modeling

🔸 Variables

Each variable represents a session between a unique faction pair. We need 9 such pairs where:

- No faction appears more than 3 times
- Each pairing is unique

We generate these 9 variables by selecting faction combinations that ensure each of the 6 factions appears exactly 3 times.

🔸 Domains

The domain for each session variable consists of all possible (Day, Room) pairs:

Domain = [(Day1, Room1), (Day1, Room2), ..., (Day3, Room3)]

Total domain size = 3 days × 3 rooms = 9 slots
Each session must be assigned to one unique slot.

🔸 Constraints

1. Unique Slot Assignment
   No two sessions can occupy the same (day, room).
2. Faction Daily Limit
   No faction should attend more than one session on the same day (enforces rest).
3. No Repeat Meetings
   Faction pairs must be unique across sessions.
4. Room Reuse Across Days
   A faction may not use the same room on consecutive days (to simulate logistical variety).

🤖 CSP Solver

We implemented a backtracking search with:

- Minimum Remaining Values (MRV)
  Chooses the next variable with the fewest available slots.
- Forward Checking
  Prunes domains of unassigned variables after each assignment.
- Constraint Checking
  The violates_constraints() function enforces all constraints before assigning a value.

🔍 Arc Consistency (AC-3)

Before backtracking, we apply the AC-3 algorithm to:

- Enforce consistency between variable domains
- Remove values that can never be part of any solution
- Reduce unnecessary search effort

This pre-processing significantly improves efficiency.

🔁 Dynamic Rescheduling

Scenario:

A room becomes unavailable (e.g., destroyed mid-summit).

Approach:

- The affected session is removed from the schedule.
- Its slot is marked as unavailable in all variable domains.
- We reassign only that session using constraint repair, leaving the rest of the schedule untouched.

This avoids full recomputation and maintains schedule stability.

✅ Outputs

- Final schedule as a mapping:
  (Day, Room) → (Faction1 vs Faction2)
- AC-3 logs and domain reductions (optional)
- Dynamic repair trace if a room is removed

📜 Summary

The Part A system combines:

- Classical CSP modeling
- Efficient heuristics and inference
- Dynamic failure recovery

It achieves an adaptive, realistic diplomatic schedule that obeys complex constraints and reacts gracefully to unexpected disruptions.

🧠 Part B: Strategic Resource Negotiation (Adversarial Search)

🎯 Objective

Simulate each diplomatic session as a **two-player adversarial negotiation game** between rival factions, where:

- Factions make strategic moves
- Utility depends on preferences, fatigue, trust, and room bias
- The game is zero-sum: one faction's gain is the other's loss

This models realistic negotiation dynamics like bluffing, threats, and resource urgency.

🕹️ Game Design

🔸 Players

Two factions (e.g., F1 and F2) engage in a turn-based negotiation.

🔸 State Representation

Each game state contains:

- room: where the negotiation takes place (Room1, Room2, Room3)
- slot: abstract time indicator (Early, Mid, Late)
- player: the faction whose turn it is
- action: the last move taken (e.g., Offer, Bluff)

🔸 Actions

Available actions during negotiation:

- Offer: Push for the best outcome (bias toward Room1, Early)
- Threaten: Shift to mid-level priority (Room2)
- Bluff: Low-quality suggestion (Room3)
- Concede: Keep the status quo
- Delay: No action; maintain position

📈 Evaluation Function

evaluate_state(state, player, fatigue, trust_level)

This function computes utility for a given faction based on:

- **Room Quality**: higher for Room1, lower for Room3
- **Time Slot Quality**: Early > Mid > Late
- **Fatigue**: -1 penalty for each round (more turns = lower utility)
- **Trust Level**: Positive for allies, negative for rivals

This function is central to decision-making for both Minimax and Alpha-Beta.

🌳 Game Tree Expansion

generate_children(state, depth, player)

Generates new states by simulating how each action transforms the negotiation environment:

- Some actions shift toward better rooms or time slots
- Turns alternate between Player A and Player B
- Each new state includes updated parameters and action history

---

🤖 Search Algorithms

✅ Minimax

- Standard adversarial search to depth 3
- Factions alternate turns trying to maximize their own utility and minimize the opponent's
- Tracks number of expanded nodes for performance comparison

✅ Alpha-Beta Pruning

- Optimized version of Minimax
- Uses α (best option for maximizer) and β (best for minimizer) to cut off unneeded branches
- Significantly reduces node expansion without affecting outcome

Both algorithms return:

- The **best action**
- The **final utility**
- The **number of nodes explored**

🧪 Simulation Examples

We simulate three matchups:

- F1 vs F2 (starting from Room3, Late)
- F3 vs F4 (Room2, Mid)
- F5 vs F6 (Room3, Late)

Each simulation is run using both:

- **Minimax**
- **Alpha-Beta**

The results include:

- Chosen action

- Utility score
- Nodes expanded

📊 Performance Comparison

Two matplotlib plots visualize:

1. **Node expansions** by Minimax vs Alpha-Beta
2. **Final utilities** for Player A (Minimax vs Alpha-Beta)

These clearly show:

- Alpha-Beta is more efficient (fewer nodes)
- Both produce similar or identical negotiation outcomes

👁️ Optional Extension: Partial Observability

A custom evaluate_state_partial() function simulates:

- Uncertainty about the opponent's true priorities
- Approximate scoring using estimated preferences

This version helps simulate real-world imperfect knowledge during negotiation, and uses a modified minimax called minimax_partial().

📜 Summary

Part B models diplomacy as a turn-based, adversarial negotiation game, using:

- A robust evaluation function that incorporates realism (trust, fatigue, bias)
- Minimax and Alpha-Beta search for strategic decision-making
- Visual and quantitative comparison of algorithmic behavior

Together with Part A, this system delivers a complete AI-driven simulation of **strategic diplomacy**, balancing **resource allocation, planning, and negotiation tactics**.