INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Group: 4

TASK ASSIGNMENT TABLE

MEMBER	ID - Email	MISSION	COMPLETE
NGUYỄN ĐÌNH VIỆT HOÀNG	522H0120 522H0120@student.tdtu.edu.vn	Task 4 Local Beam Search	100%
NGUYỄN NHẬT CHIÊU	522H0133 522H0133@student.tdtu.edu.vn	Task 3 Simulated Annealing Search	100%
ĐẶNG CÔNG MINH	522H0095 522H0095@student.tdtu.edu.vn	Task 2 Restart Hill-Climbing	100%
TRẦN THIÊN ÂN	522H0165 522H0165@student.tdtu.edu.vn	Task 1 Problem formulation	100%
VÕ MINH TÀI	522H0168 522H0168@student.tdtu.edu.vn	Pseudocode Presentation	100%

Task 1
Problem Formulation

Ways to solve problems of Task 1

1. __init__: Initializes the class instance and loads the state space from an image file.

2. load_state_space: Loads the state space by reading and processing an image file.

3. show: Displays a 3D plot of the state space surface.

4. draw_path: Displays a 3D plot of the state space surface with a path represented by a line.

5. evaluate_state: Evaluates the value of a given state in the state space.

6. get_random_state: Returns a random state within the state space.

7. get_highest_valued_successor: Returns the neighbor with the highest value (evaluation) among the successors of a given state.

8. get_successor: Returns the valid neighboring states (successors) of a given state.

9. is_edge_state: Checks if a given state is on the edge of the state space.

Task 2
Random Restart Hill-Climbing

NATURE

Random Restart Hill-Climbing

ADVANTAGES

- + Overcoming Local Optimal
- + Simplicity
- + Memory Efficiency
- + Exploitation Balance

DISADVANTAGES

- – Computational Cost
- – Repetitive Search
- – Lack of Guaranteed Global
- – Sensitivity to Initial States

Pseudocode For Task 2

```
Function random_restart_hill_climbing(problem, num_trials)
Returns a local maximum
Input problem, num_trials

For each trial in num_trials do:
    current_state <- get a random state from problem
    current_value <- evaluate the current_state using problem
    Initialize path with current_state

    Repeat:
        neighbor <- get the highest valued successor of current_state from problem
        neighbor_value <- evaluate the neighbor using problem
        If neighbor_value is less than or equal to current_value then
            Exit the loop

        Update current_state to neighbor
        Update current_value to neighbor_value
        Append current_state to path

    If current_value is greater than best_evaluation then
        Update best_evaluation to current_value
        Update best_path to path

Create xyz_list from best_path with each state transformed to (x, y, Z value)
Return xyz_list
```

Task 3
Simulated Annealing Search

NATURE

Simulated Annealing Search

ADVANTAGES

+ Global Optimization

+ Flexibility

+ Heuristic nature

+ Simplicity

DISADVANTAGES

- Convergence Speed

- Tuning Parameters

- Lack of Determinism

- Sensitivity to Initial Solutions

Pseudocode For Task 3

```
Function simulated_annealing_search(problem, schedule):
Returns xyz_list
Input problem, schedule

 current_state <- get a random state from problem
 Initialize path with current_state

 For t from 1 to 1,000,000 do:
     T <- schedule(t)
     If T equals 0 then:
        - Exit the loop

     neighbors <- get successors of current_state from problem
     If neighbors is empty then:
        - Exit the loop

     next_state <- choose a random state from neighbors
     ΔE <- evaluate next_state - evaluate current_state using problem
     If ΔE > 0 or random probability < exp(ΔE / T) then:
         Update current_state to next_state
         Append current_state to path

 Create xyz_list from path with each state transformed to (x, y, Z value)
 Return xyz_list
```

Task 4
Local Beam Search

NATURE

Local Beam Search

ADVANTAGES

+ Parallel Exploration
+ Diverse Exploration
+ Memory Efficiency
+ Easy Implementation

DISADVANTAGES

- Premature Convergence
- Beam Width Selection
- Lack of Diversity
- Lack of Global Optimality

Pseudocode For Task 4

```
Function local_beam_search(problem, k):
Returns path
Input problem, k

 start_state <- get a random state from problem
 Initialize beam with a deque containing a tuple of evaluation of start_state and start_state
 Initialize path with a tuple of start_state's coordinates and its evaluation

 While beam is not empty do:
     Initialize new_beam as an empty deque

     For each tuple of eval_value and state in beam do:
         successors <- get a list of tuples of evaluation and neighbor for each neighbor of state
         Extend new_beam with successors

     Sort new_beam in descending order by evaluation value and keep the top k elements
     If the first element's evaluation in new_beam is not less than any other's in new_beam then:
        - Exit the loop

     Update beam to new_beam
     Append a tuple of the first element's state coordinates and its evaluation to path

 Return path
```

THE END