

# INTRODUCTION TO ARTIFICIAL INTELLIGENCE

**Group: 4**

# TASK ASSIGNMENT TABLE

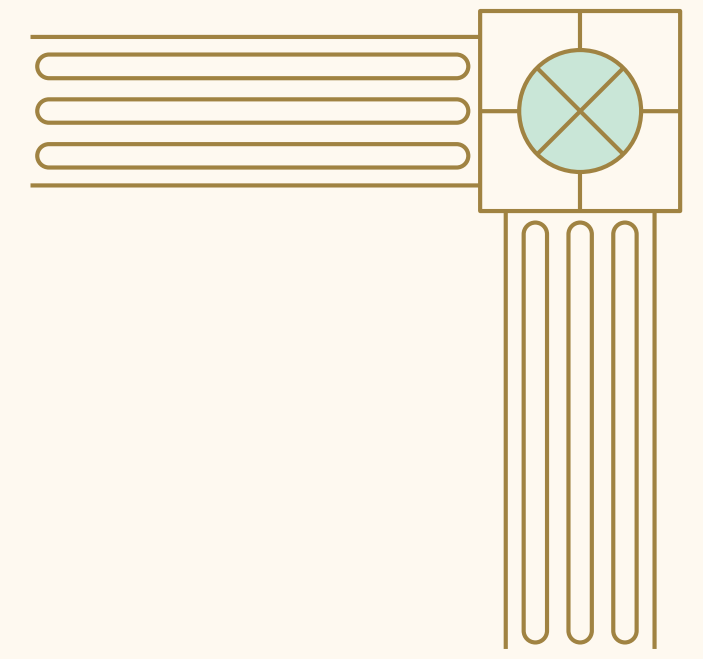
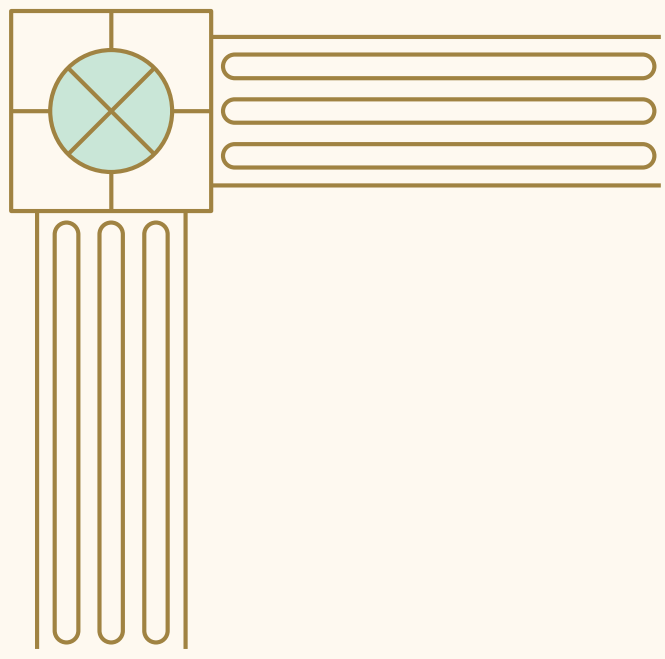
MEMBER	ID - Email	MISSION	COMPLETE
NGUYỄN ĐÌNH VIỆT HOÀNG	522H0120 522H0120@student.tdtu.edu.vn	Task 4 Local Beam Search	100%
NGUYỄN NHẬT CHIÊU	522H0133 522H0133@student.tdtu.edu.vn	Task 3 Simulated Annealing Search	100%
ĐẶNG CÔNG MINH	522H0095 522H0095@student.tdtu.edu.vn	Task 2 Restart Hill-Climbing	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

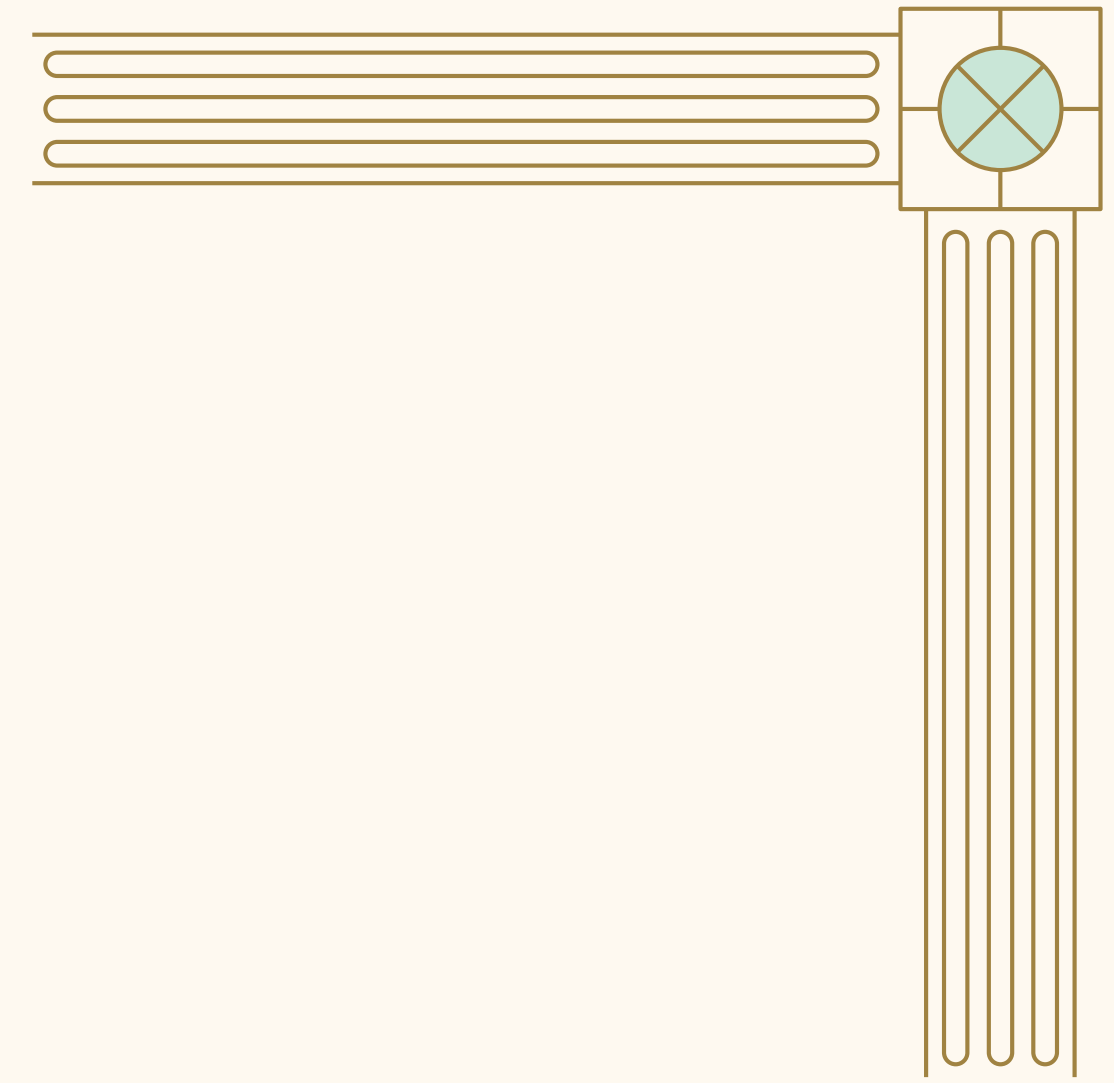
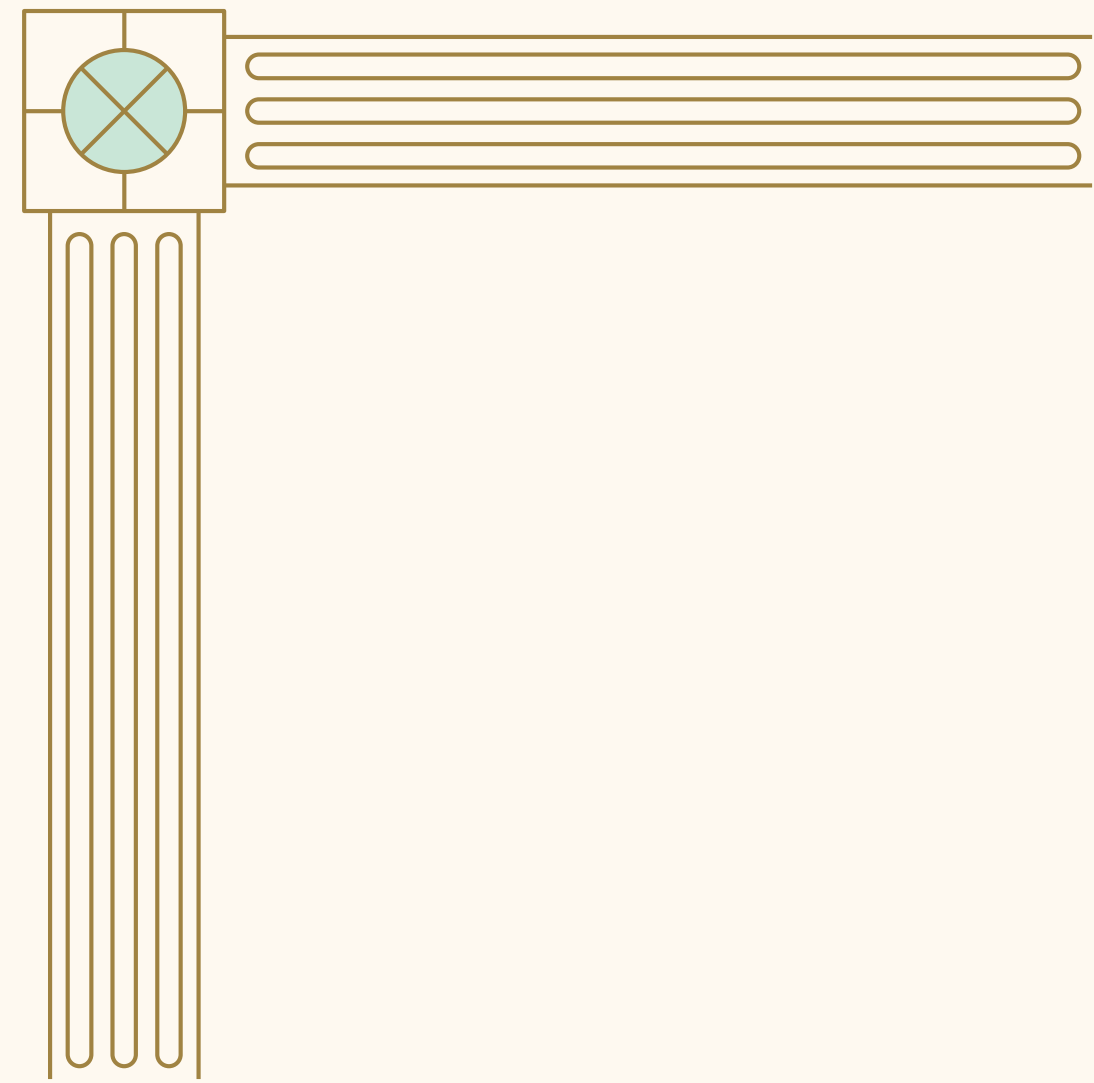




# Ways to solve problems of Task 1

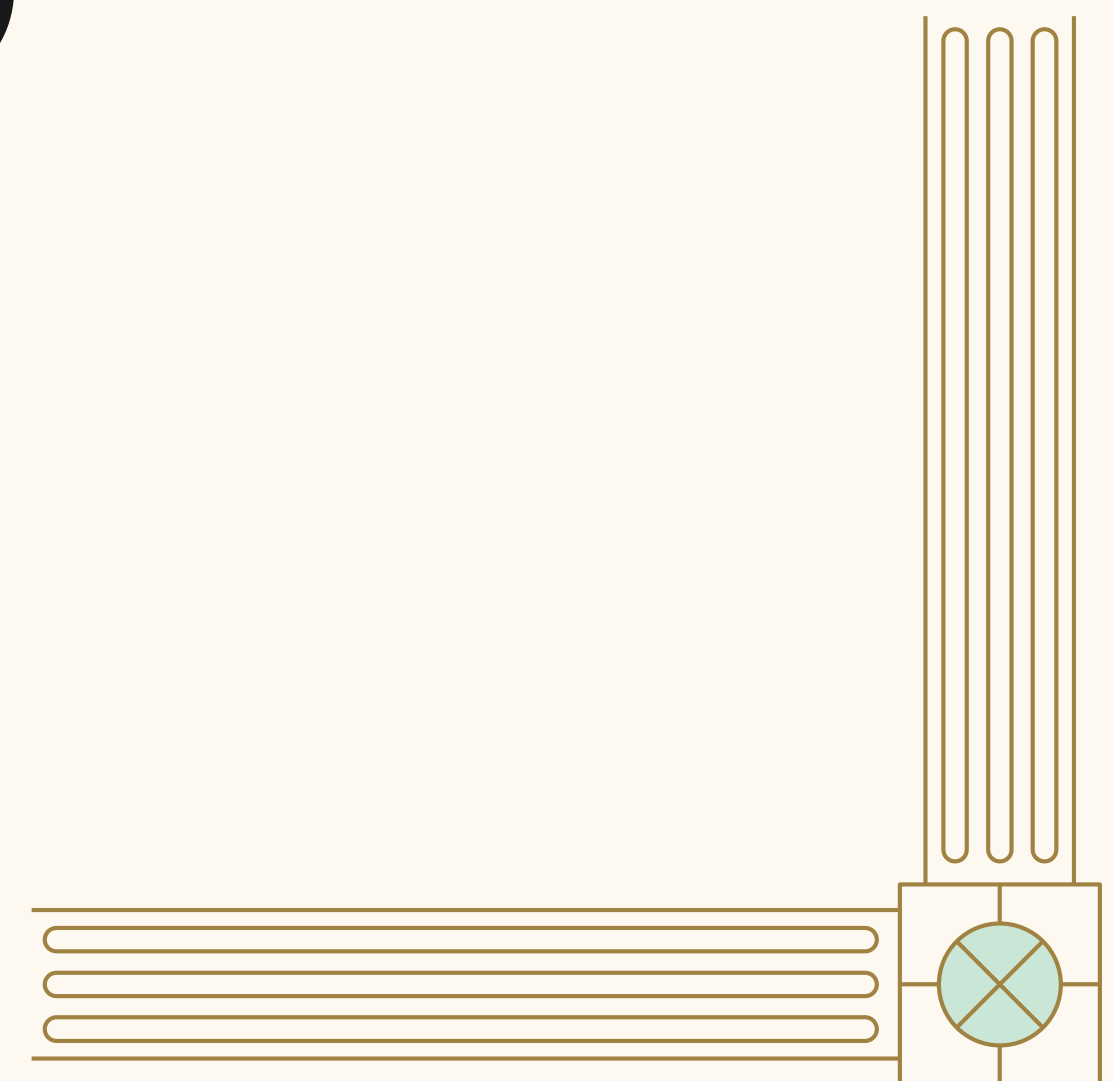
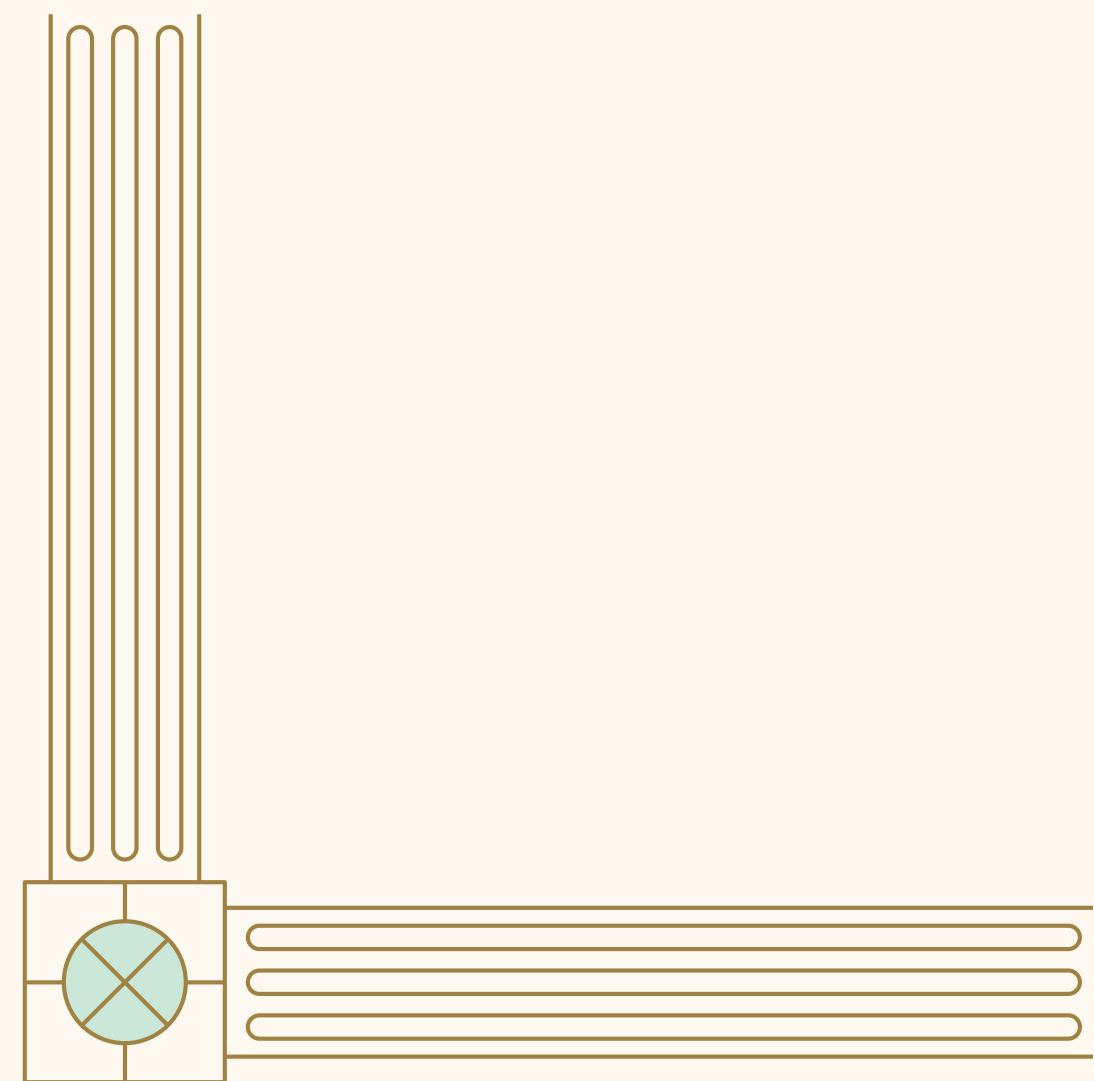
1. **\_\_init\_\_**: Initializes the class instance and loads the state space from an image file.
2. **load\_state\_space**: Loads the state space by reading and processing an image file.
3. **show**: Displays a 3D plot of the state space surface.
4. **draw\_path**: Displays a 3D plot of the state space 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.

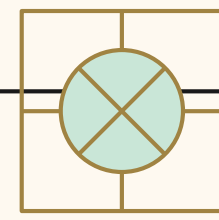




# Task 2

## Random Restart Hill-Climbing





# NATURE

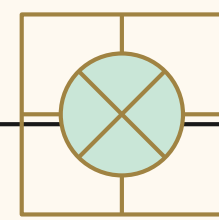
## Random Restart Hill-Climbing

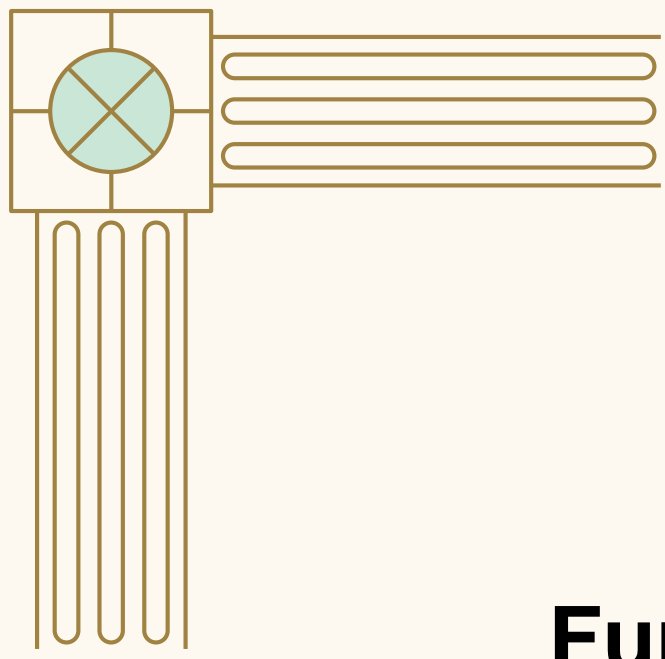
### ADVANTAGES

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

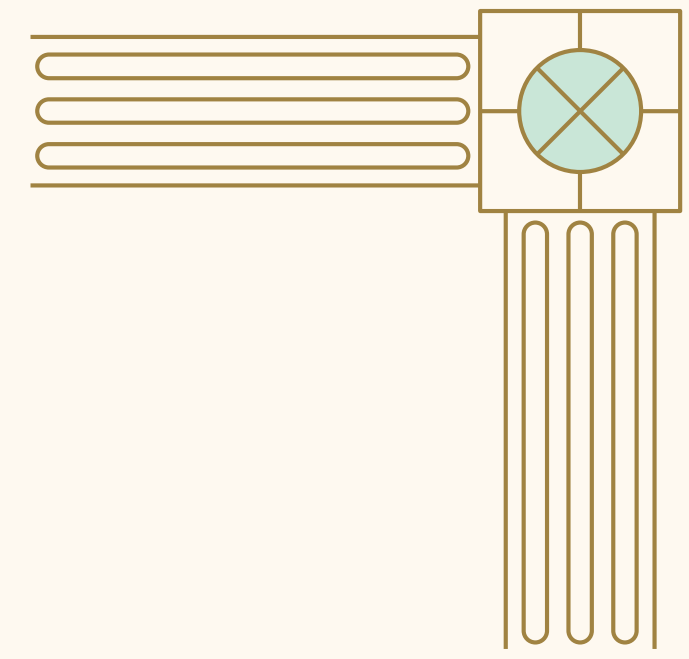
### DISADVANTAGES

- 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

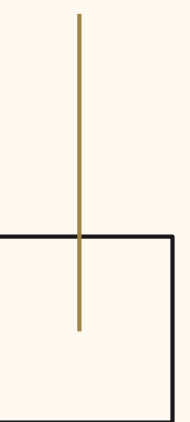
    If current\_value is greater than best\_evaluation then

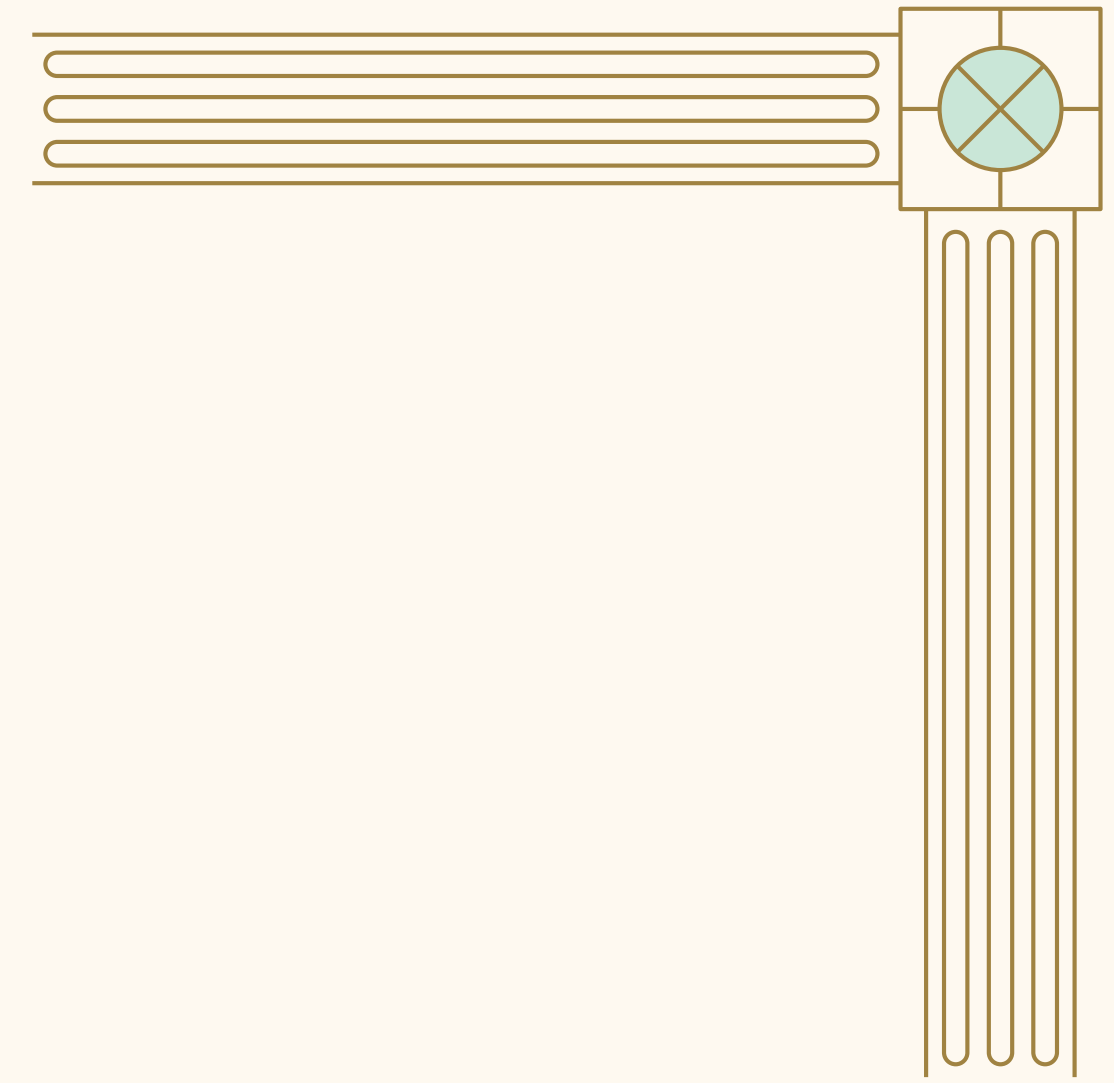
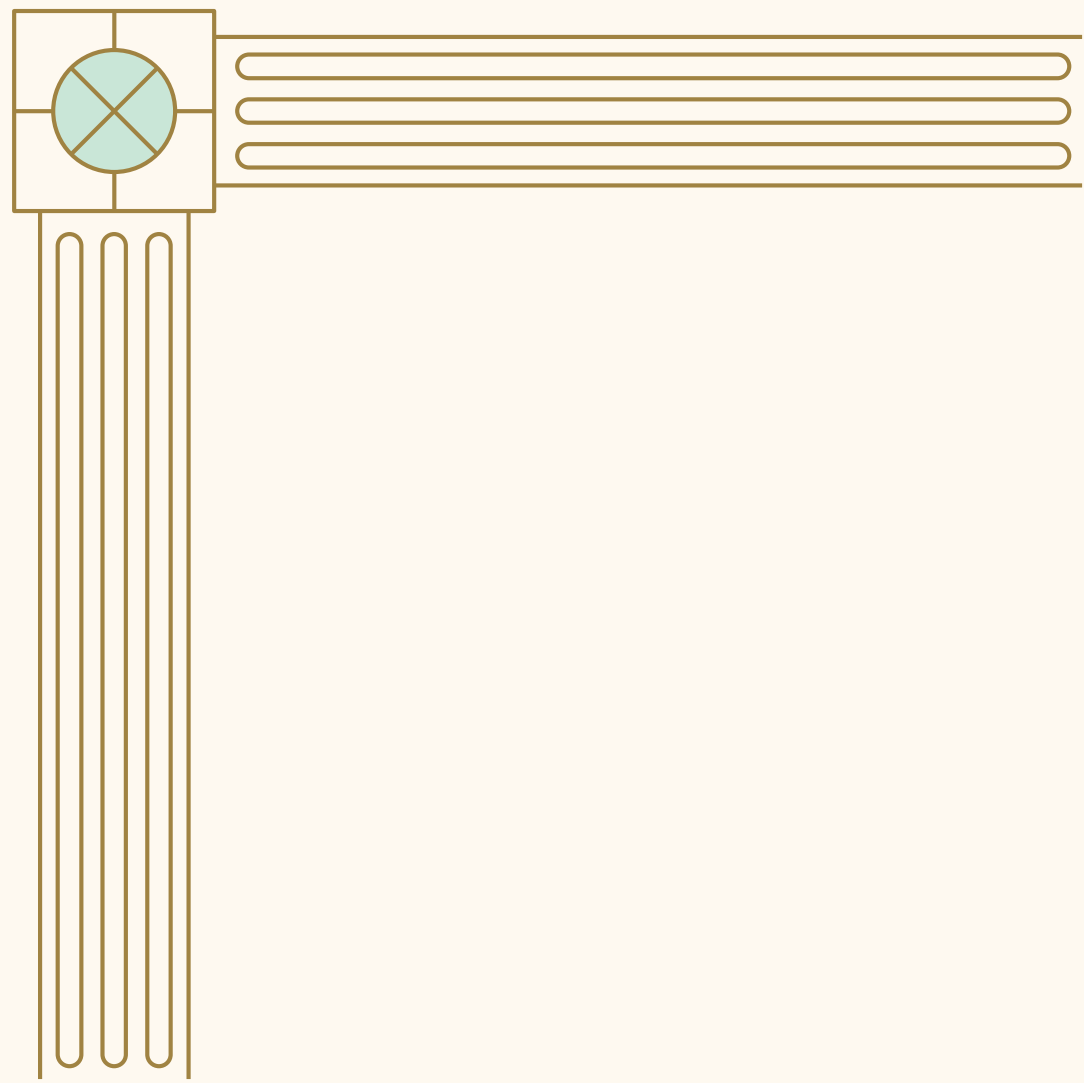
        Update best\_evaluation to current\_value

        Update best\_path to path

Create xyz\_list from best\_path with each state transformed to (x, y, Z value)

**Return** xyz\_list

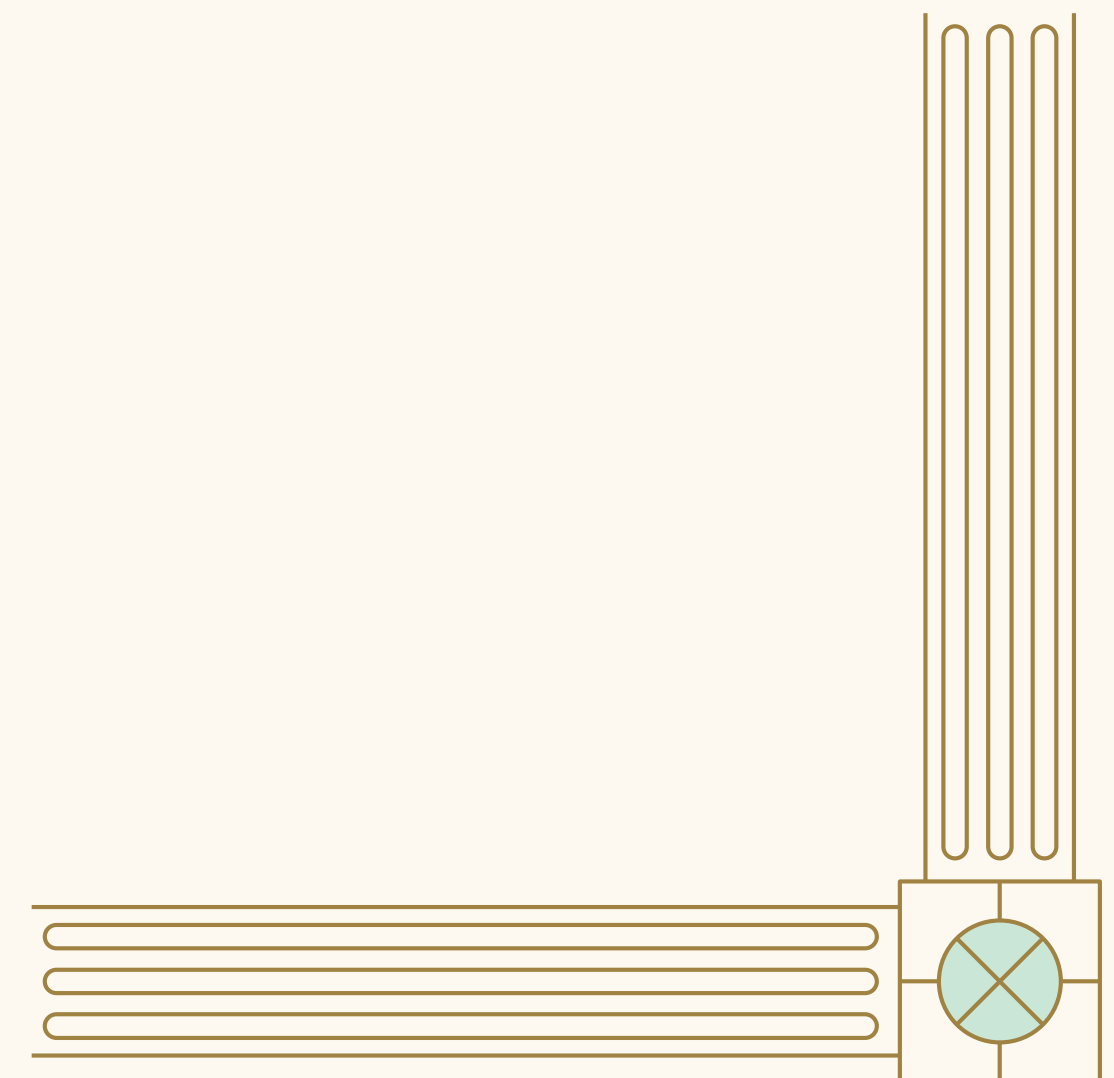
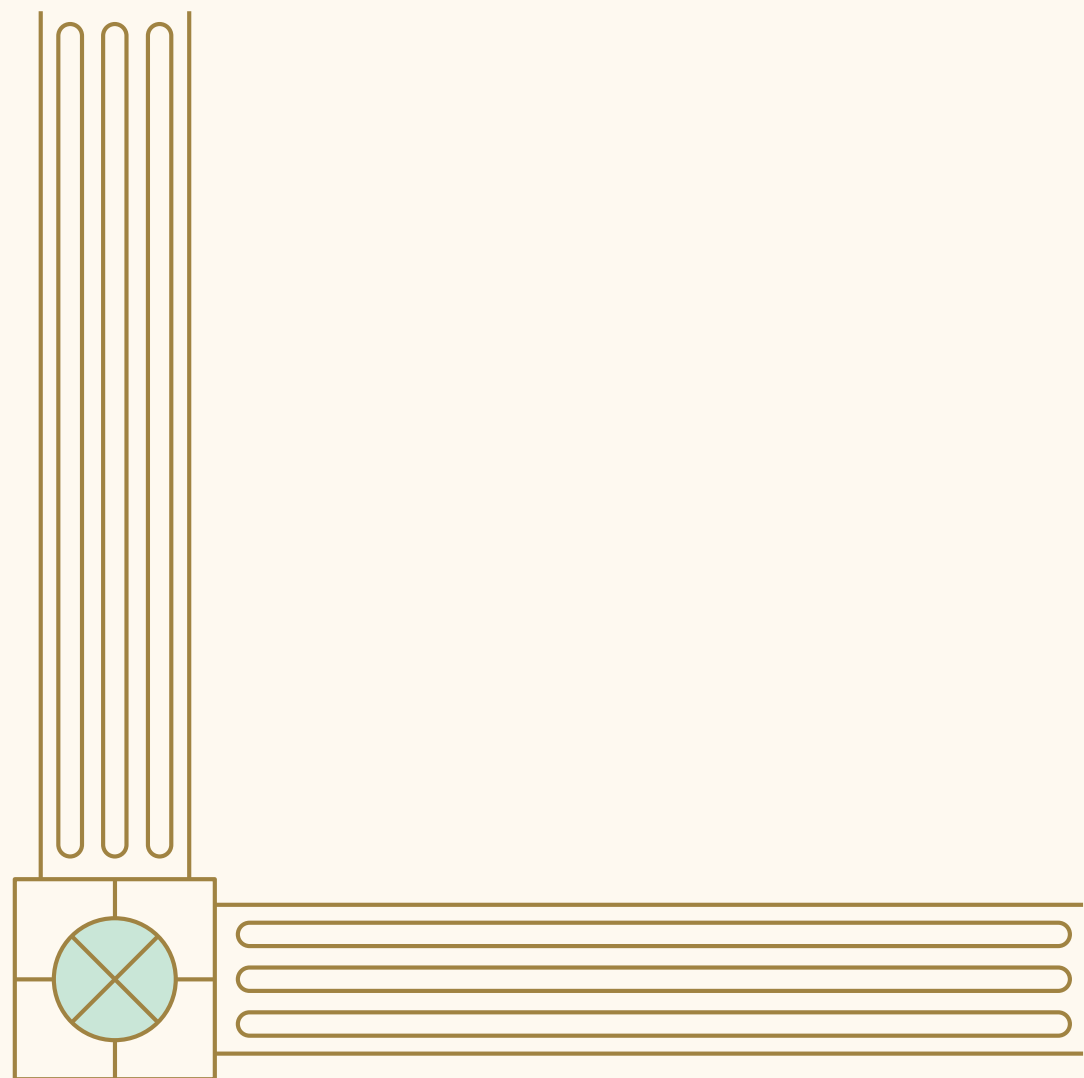




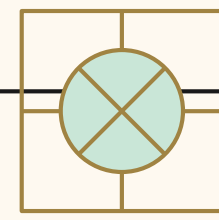
# Task 3

# Simulated Annealing

# Search







# NATURE

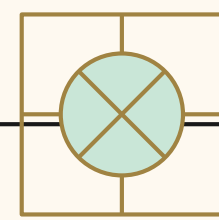
## Simulated Annealing Search

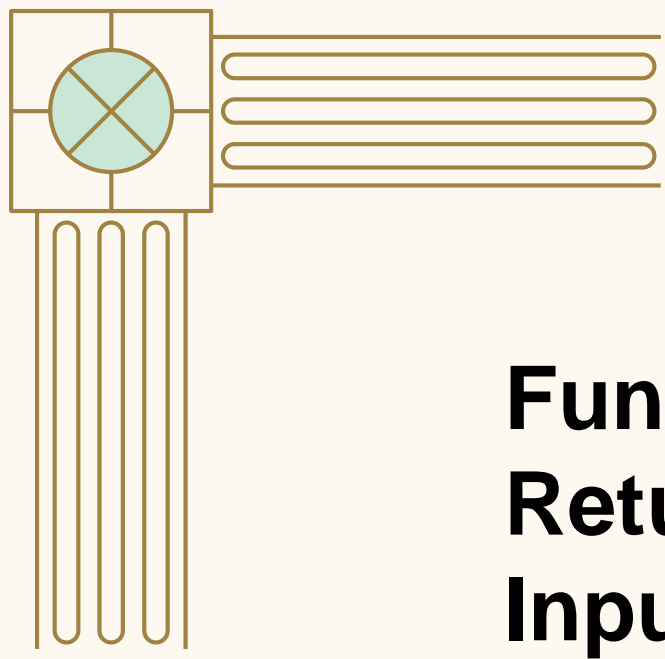
### ADVANTAGES

- + Global Optimization**
- + Flexibility**
- + Heuristic nature**
- + Simplicity**

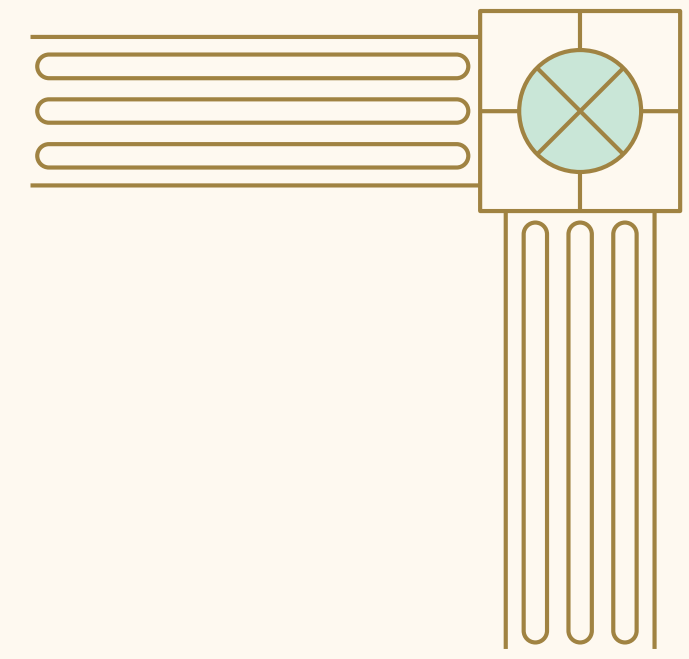
### DISADVANTAGES

- 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

$\Delta E$  <- evaluate next\_state - evaluate current\_state using problem

    If  $\Delta E > 0$  or random probability <  $\exp(\Delta E / T)$  then:

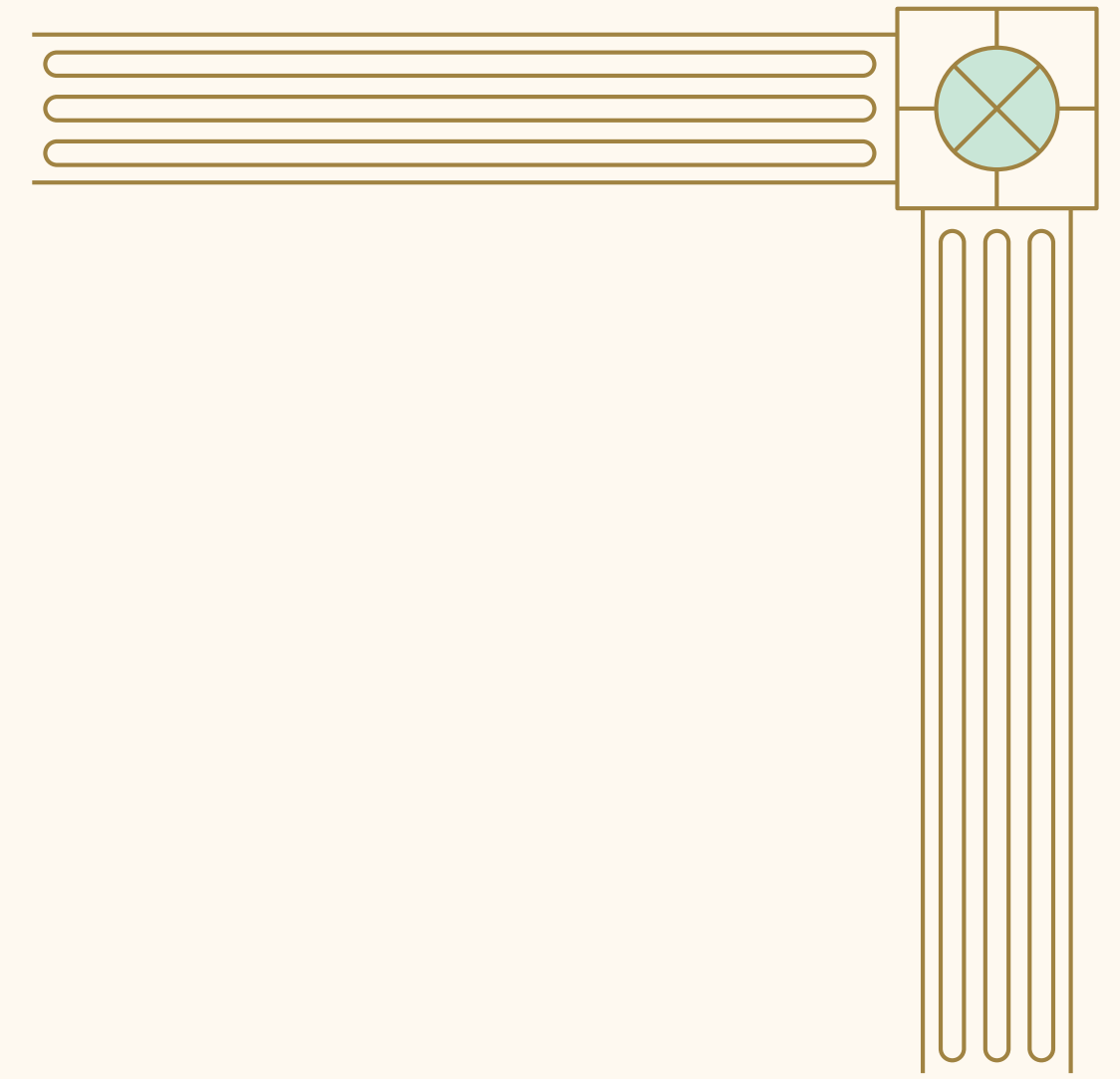
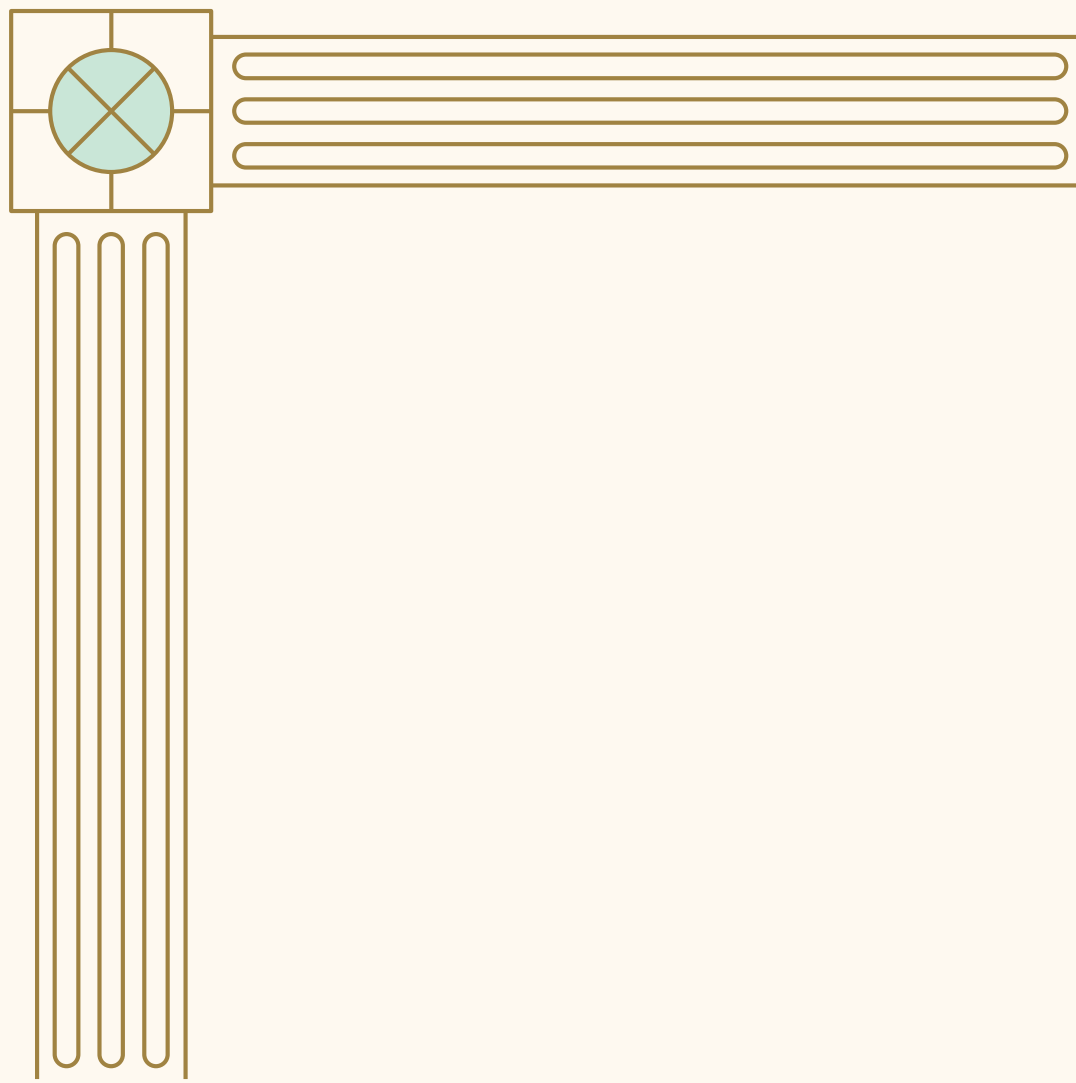
        Update current\_state to next\_state

        Append current\_state to path

Create xyz\_list from path with each state transformed to (x, y, Z value)

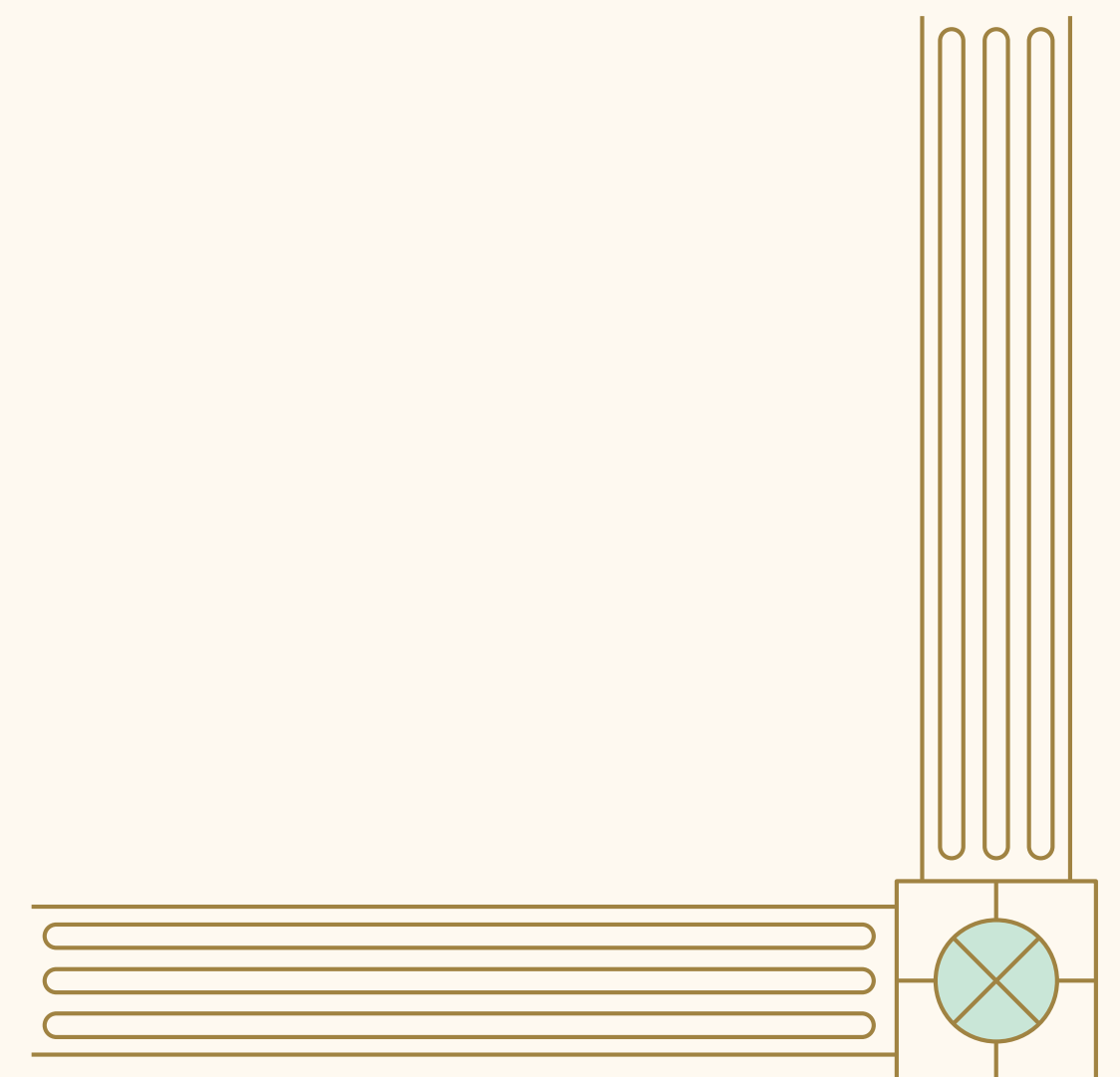
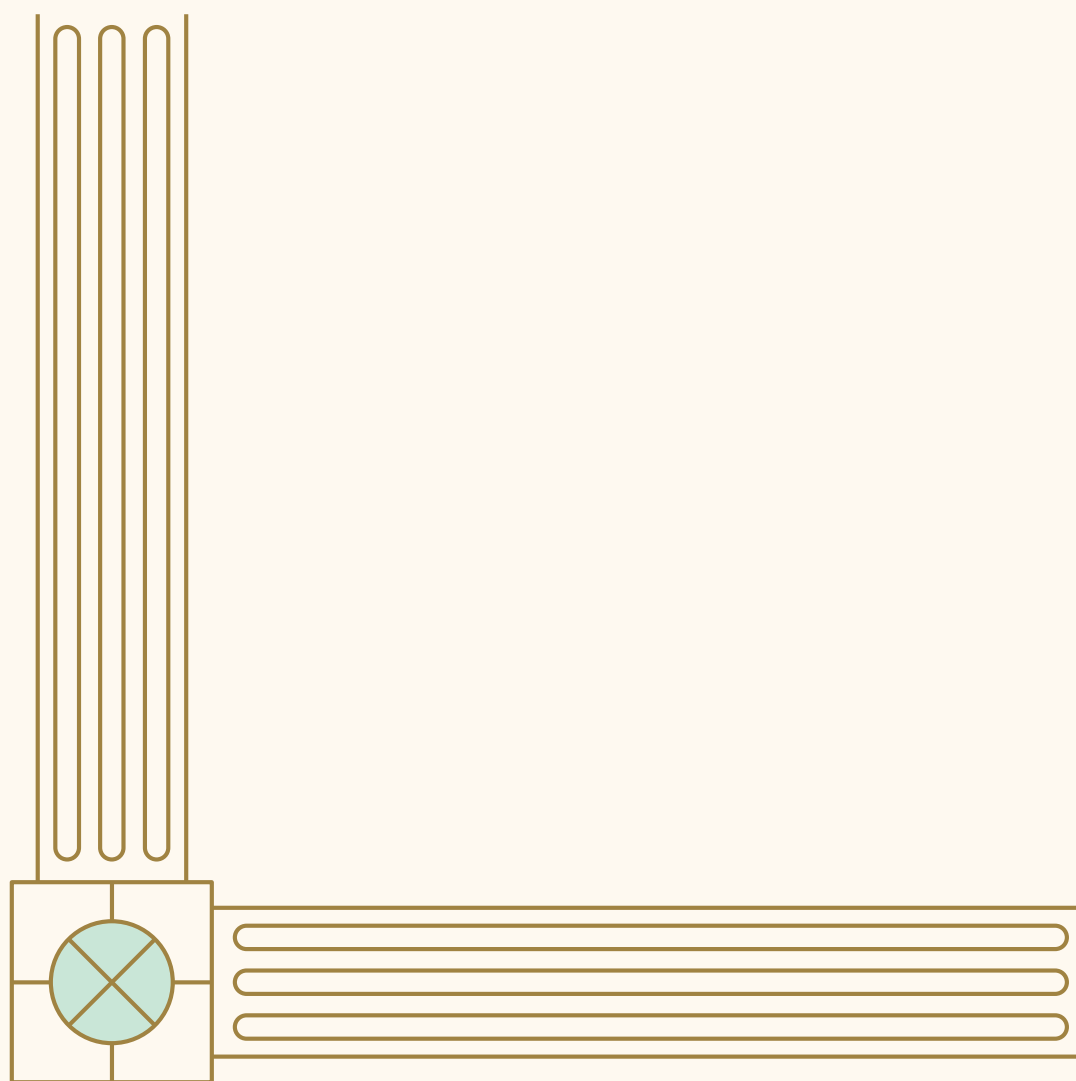
**Return** xyz\_list

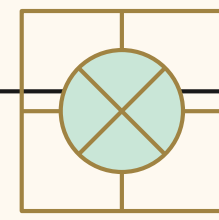




# Task 4

# Local Beam Search





NATURE

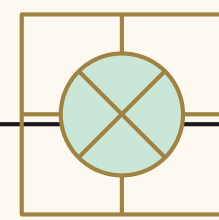
# Local Beam Search

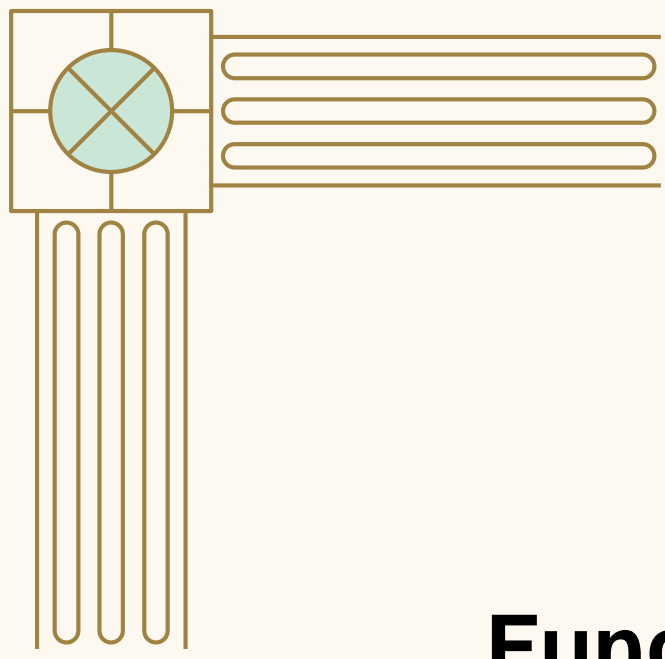
## ADVANTAGES

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

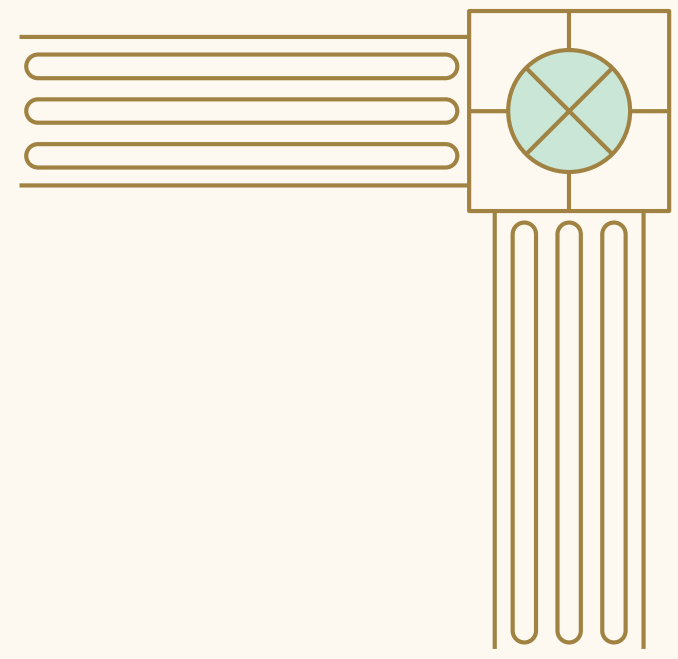
## DISADVANTAGES

- 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

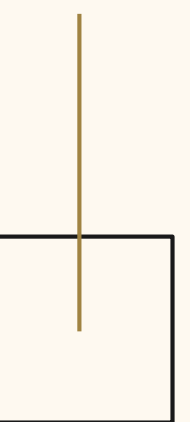
    If the first element's evaluation in new\_beam is not less than any other's in new\_beam then:

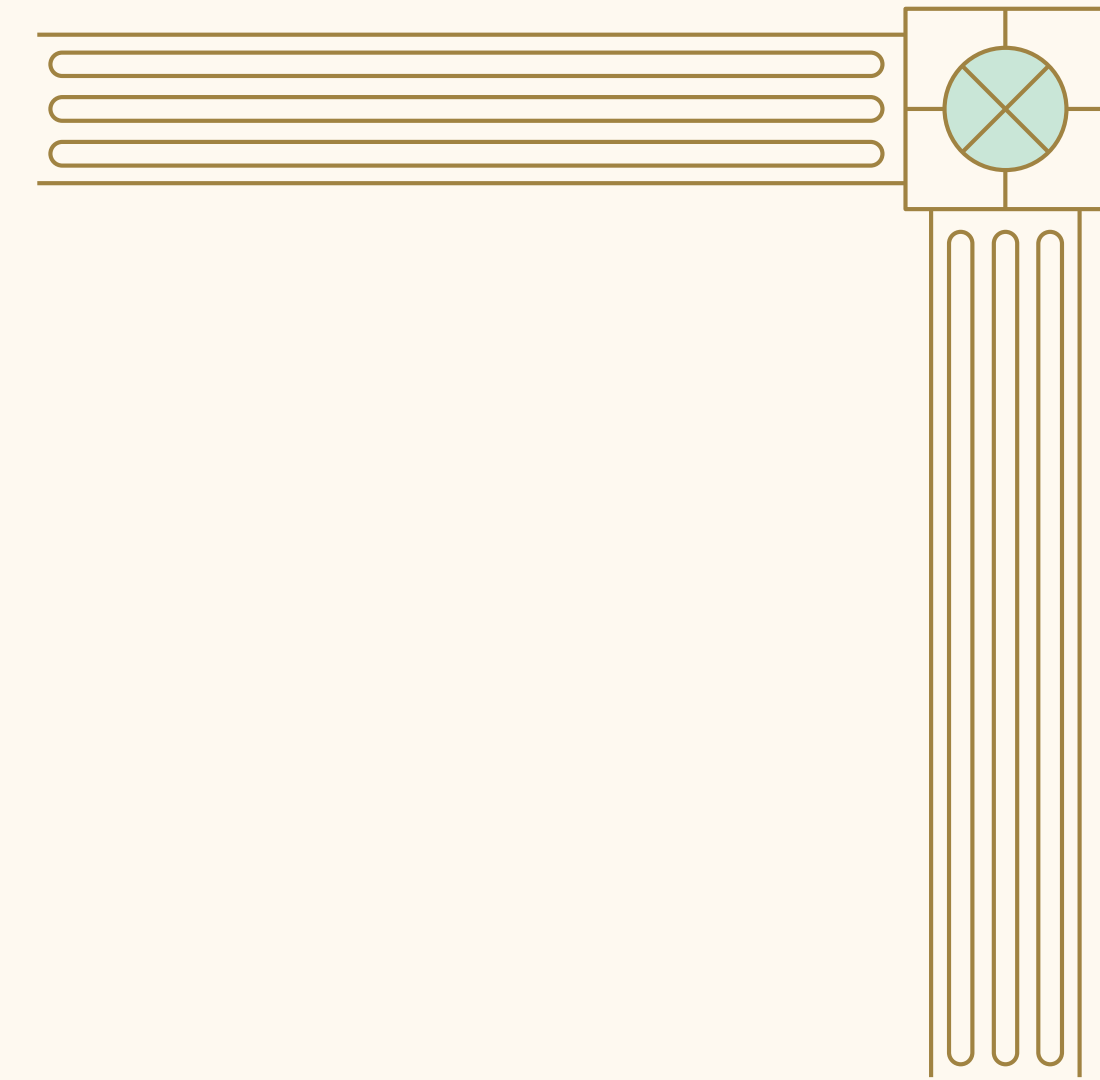
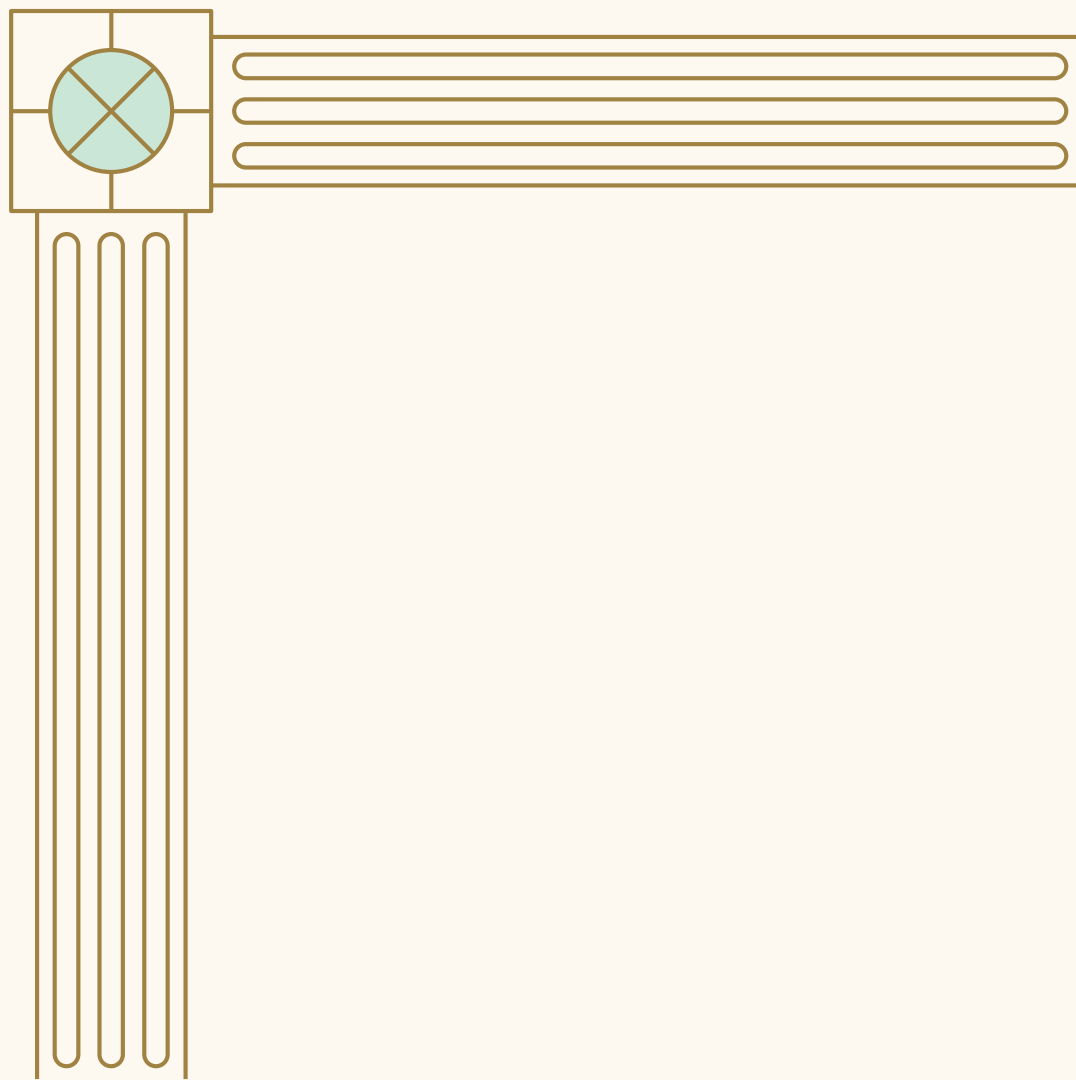
        - Exit the loop

    Update beam to new\_beam

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

**Return** path





**THE END**

