**TASK 1.1 - Xây dựng Mô hình Bài toán (Logic + Phân tích)**

**Phần 1: Mô hình hóa State Space**

* [ ] Định nghĩa trạng thái (State)
* [ ] Xác định trạng thái ban đầu
* [ ] Xác định 4 trạng thái đích
* [ ] Liệt kê tất cả hành động (Actions)
* [ ] Xây dựng hàm chuyển trạng thái (Transition Model)
* [ ] Định nghĩa hàm chi phí đường đi (Path Cost)

**Phần 2: Thiết kế Cấu trúc Dữ liệu**

* [ ] Thiết kế class State
* [ ] Thiết kế class Action
* [ ] Thiết kế class Node
* [ ] Thiết kế class Problem
* [ ] Thiết kế class Heuristic
* [ ] Thiết kế class AStar
* [ ] Vẽ Class Diagram

**Phần 3: Cài đặt Thuật toán**

* [ ] Implement A\* tổng quát
* [ ] Implement Priority Queue
* [ ] Implement kiểm tra trạng thái đích
* [ ] Implement truy vết đường đi
* [ ] Test với các trường hợp cơ bản

# 1. MÔ HÌNH STATE SPACE

## 1.1 Trạng thái (State) và trạng thái ban đầu

* **Biểu diễn:** Ma trận 3×3 chứa các số từ 1-8 và ô trống (0)
* **Ví dụ:**

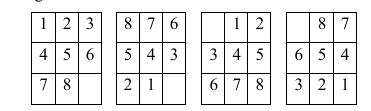
[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

## 1.2 Trạng thái Đích (Goal States)

Có **4 trạng thái đích** khác nhau:



## 1.3 Hành động (Actions)

**Loại 1: Swap các ô kề nhau có tổng = 9**

* Điều kiện: A + B = 9, A và B kề nhau (ngang/dọc)
* Các cặp hợp lệ: (1,8), (2,7), (3,6), (4,5)

**Loại 2: Swap góc chéo**

* Góc trên-trái ↔ Góc dưới-phải
* Góc trên-phải ↔ Góc dưới-trái
* **Lưu ý:** Ô trống (0) KHÔNG được swap

**1.4 Hàm Chuyển trạng thái**

State → Action → New State

* Kiểm tra tính hợp lệ của action
* Thực hiện swap
* Trả về trạng thái mới

**1.5 Chi phí (Cost)**

* Mỗi hành động có chi phí = 1
* Chi phí đường đi = số bước đi

# 2. CẤU TRÚC DỮ LIỆU

## 2.1 Class State

class State:

- board: List[List[int]] # Ma trận 3x3

- blank\_pos: Tuple[int, int] # Vị trí ô trống

+ \_\_init\_\_(board)

+ get\_valid\_actions() → List[Action]

+ apply\_action(action) → State

+ is\_goal() → bool

+ \_\_eq\_\_(other) → bool

+ \_\_hash\_\_() → int

+ to\_tuple() → Tuple

## 2.2 Class Action

class Action:

- type: str # "sum9\_swap" hoặc "corner\_swap"

- pos1: Tuple[int, int]

- pos2: Tuple[int, int]

+ \_\_init\_\_(type, pos1, pos2)

+ \_\_str\_\_() → str

## 2.3 Class Node

class Node:

- state: State

- parent: Node

- action: Action

- path\_cost: int # g(n)

- heuristic: int # h(n)

- f\_score: int # f(n) = g(n) + h(n)

+ \_\_init\_\_(state, parent, action, path\_cost, heuristic)

+ get\_path() → List[Action]

+ \_\_lt\_\_(other) → bool # So sánh cho priority queue

## 2.4 Class Problem

class Problem:

- initial\_state: State

- goal\_states: List[State]

+ \_\_init\_\_(initial\_state)

+ is\_goal(state) → bool

+ get\_successors(state) → List[Tuple[State, Action, int]]

## 2.5 Class Heuristic

class Heuristic:

+ calculate(state, goal\_states) → int

+ is\_admissible() → bool

+ is\_consistent() → bool

## 2.6 Class AStar

class AStar:

- problem: Problem

- heuristic: Heuristic

- open\_list: PriorityQueue

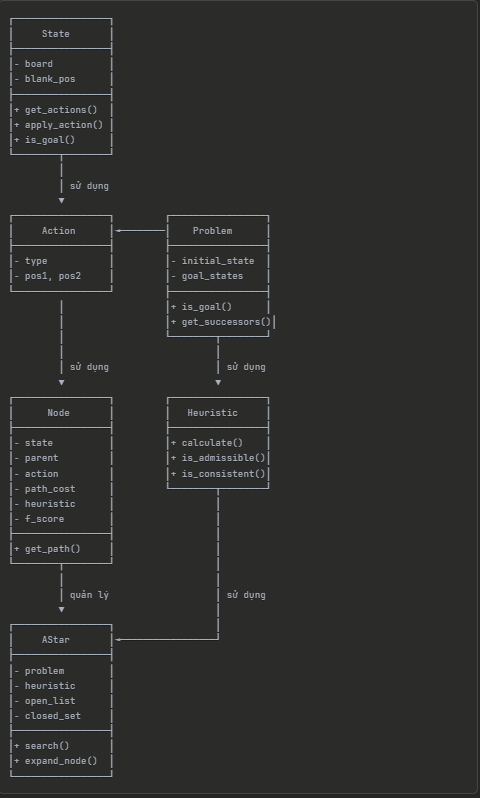
- closed\_set: Set[State]

+ \_\_init\_\_(problem, heuristic)

+ search() → Tuple[List[Action], int, dict]

+ expand\_node(node) → List[Node]

# 3. CLASS DIAGRAM



# 4. THUẬT TOÁN A \* TỔNG QUÁT

|  |
| --- |
| **Pseudo Code:**  function A\_STAR(problem, heuristic):  initial\_node = Node(problem.initial\_state, g=0, h=heuristic)  open\_list = PriorityQueue()  open\_list.push(initial\_node)  closed\_set = Set()  while open\_list not empty:  current = open\_list.pop() // Node có f nhỏ nhất  if problem.is\_goal(current.state):  return current.get\_path()    closed\_set.add(current.state)  for successor in problem.get\_successors(current.state):  child\_state, action, cost = successor  if child\_state in closed\_set:  continue  g = current.path\_cost + cost  h = heuristic.calculate(child\_state)  child\_node = Node(child\_state, current, action, g, h)  open\_list.push(child\_node)  return None // Không tìm thấy lời giải |