Chuyển đổi mã nguồn C++ sang Python
Thuật toán LaCAM2 cho bài toán MAPF

Sinh viên: **Bùi Xuân Tùng** MSSV: **20224905**

Giáo viên hướng dẫn: **ThS. Nguyễn Tiến Thành**

https://github.com/PakerB/Project-3---MAPF

Nội dung

Bài toán MAPF (Multi-Agent Path Finding)

Đầu vào

Bản đồ lưới có vật cản + N agent (start/goal).

Ràng buộc

- **Va chạm đỉnh (vertex collision)**: không có 2 agent đứng cùng một ô tại cùng thời điểm.
- **Va chạm cạnh/hoán đổi (swap collision)**: không có 2 agent đổi chỗ cho nhau trong cùng 1 bước.

Mục tiêu

Tìm lộ trình cho tất cả agent đến goal; có thể tối ưu makespan hoặc tổng chi phí.

LaCAM2 là gì?

- LaCAM2: **Lazy Constraint Addition Search** cho MAPF.
- Ý tưởng: chỉ thêm ràng buộc khi thật sự cần, thay vì liệt kê mọi ràng buộc ngay từ đầu.
- Kết hợp **high-level search** (cấu hình toàn cục) và **low-level** (PIBT).

Cấu trúc module chính

File	Vai trò
`graph.py`	Đồ thị lưới: Vertex/Graph, neighbor, hash config
`instance.py`	Load map/scenario, tạo start/goal
`dist_table.py`	Heuristic distance table (BFS lazy)
`planner.py`	**Core**: LaCAM2 search + PIBT + xử lý swap
`lacam2.py`	Entry: hàm `solve(...)`
`post_processing.py`	Validate solution + metrics
`utils.py`	Deadline, log, random helper

Luồng chạy tổng quát

1. Tạo `Instance` từ map/scenario (starts/goals).
2. Tạo `Planner` và `DistTable` (heuristic).
3. Lặp tìm kiếm: tạo cấu hình kế tiếp bằng **PIBT**, tránh va chạm.
4. Nếu đạt cấu hình goal: backtrack để xuất đường đi.
5. Hậu kiểm nghiệm bằng `post_processing`.

Nguyên tắc port C++ → Python

- Giữ đúng **logic thuật toán** trước; tối ưu sau.
- Thay con trỏ/struct bằng **class + reference**.
- STL: `vector/map/queue` → Python: `list/dict/deque`.
- Tách module rõ ràng: Graph / Instance / Heuristic / Planner / Validate.

Bảng đối chiếu nhanh

Khối chức năng	**C++ (gốc)**	**Python (port)**
Đồ thị	Graph/Vertex	`graph.py`
Bài toán	Instance	`instance.py`
Heuristic	DistTable (BFS)	`dist_table.py`
Tìm kiếm	Planner/HNode/LNode	`planner.py`
API	solve(...)	`lacam2.py`
Hậu kiểm	feasible + metrics	`post_processing.py`

Graph/Vertex: biểu diễn bản đồ lưới

- Đọc file `.map`: lấy `width/height` + ma trận ký tự.
- Bỏ ô vật cản (`@` hoặc `T`), tạo `Vertex` cho ô trống.
- Mỗi `Vertex` lưu danh sách neighbor (4 hướng) để agent có thể đi.

So sánh: Graph/Vertex (C++ vs Python)

graph.cpp 9

lacam2 > lacam2 > src > graph.cpp > ...

```cpp
#include "../include/graph.hpp"

Vertex::Vertex(uint _id, uint _index)
    : id(_id), index(_index), neighbor(Vertices())
{
}

Graph::Graph() : V(Vertices()), width(0), height(0) {}
Graph::~Graph()
{
  for (auto& v : V)
    if (v != nullptr) delete v;
  V.clear();
}

// to load graph
static const std::regex r_height = std::regex(R"(height\s(\d+))");
static const std::regex r_width = std::regex(R"(width\s(\d+))");
static const std::regex r_map = std::regex(R"(map)");

Graph::Graph(const std::string& filename) : V(Vertices()), width(0), height(0
{
  std::ifstream file(filename);
  if (!file) {
    std::cout << "file " << filename << " is not found." << std::endl;
    return;
  }
  std::string line;
  std::smatch results;

  // read fundamental graph parameters
  while (getline(file, line)) {
    // for CRLF coding
    if (*(line.end() - 1) == 0x0d) line.pop_back();

    if (std::regex_match(line, results, r_height)) {
      height = std::stoi(results[1].str());
    }
```

graph.py

lacam2_py > lacam2 > graph.py > ...

```python
"""
Định nghĩa đồ thị cho LaCAM2
"""

import re
from typing import List, Optional


class Vertex:
    """Đỉnh của đồ thị đại diện cho một vị trí"""

    def __init__(self, vertex_id: int, index: int):
        self.id = vertex_id  # chỉ số cho V trong Graph
        self.index = index   # chỉ số cho U, width * y + x, trong Graph
        self.neighbor: List['Vertex'] = []

    def __repr__(self):
        return str(self.index)

    def __str__(self):
        return str(self.index)


# Định nghĩa kiểu dữ liệu
Vertices = List[Optional[Vertex]]
Config = List[Vertex]  # tập hợp vị trí của tất cả agent


class Graph:
    """Biểu diễn đồ thị dạng lưới"""

    def __init__(self, filename: Optional[str] = None):
        self.V: List[Vertex] = []  # without None
        self.U: List[Optional[Vertex]] = []  # with None
        self.width: int = 0
        self.height: int = 0

        if filename:
```

Instance: định nghĩa bài toán MAPF

- Lưu **Graph** + **starts/goals** cho N agent.
- 3 cách khởi tạo: từ scenario, từ chỉ số (test), hoặc sinh ngẫu nhiên.
- Hàm `is_valid` kiểm tra nhất quán dữ liệu.

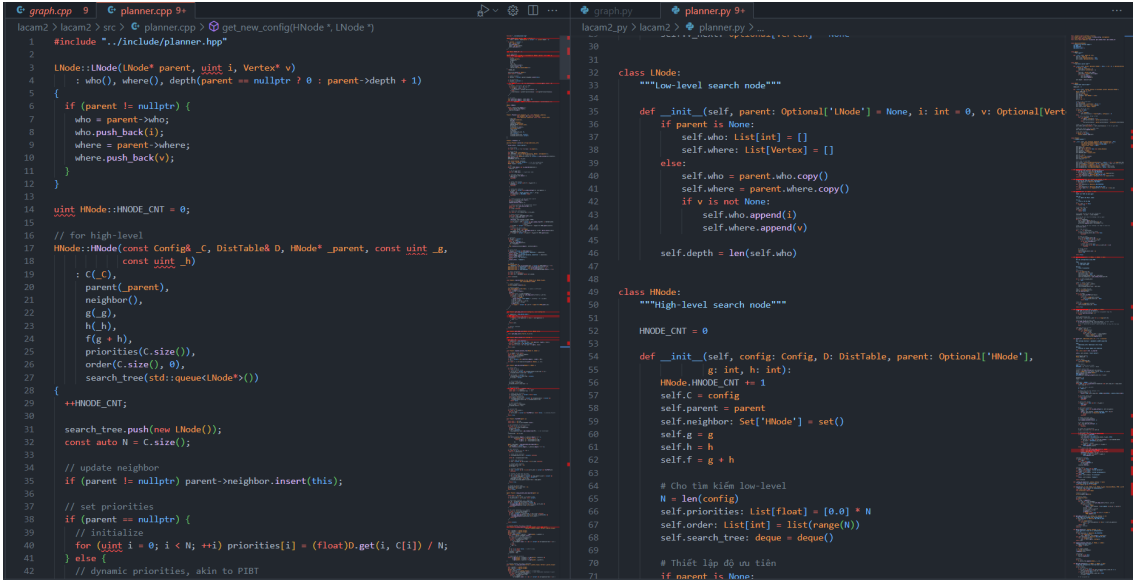
DistTable: heuristic khoảng cách (BFS + lazy)

- Mục tiêu: truy vấn nhanh khoảng cách ngắn nhất từ một vertex về goal của agent.
- BFS được mở rộng **lazy**: chỉ chạy đến khi chạm vertex cần truy vấn.
- Lợi ích: tiết kiệm thời gian/bộ nhớ khi không cần full all-pairs.

Planner: trái tim của LaCAM2

- **HNode**: cấu hình toàn cục C của tất cả agent.
- **LNode**: gán một phần (ai $\rightarrow$ đâu) trong 1 timestep.
- Tạo cấu hình kế tiếp bằng **PIBT**, kiểm soát vertex + swap conflicts.

So sánh: HNode & NNode (C++ vs Python)



PIBT (Priority Inheritance with Backtracking)

- Mỗi agent chọn candidate: đứng yên hoặc đi sang neighbor.
- Ưu tiên candidate gần goal hơn (theo DistTable) + tie-breaker ngẫu nhiên.
- Nếu bị chặn, agent có thể "đẩy" agent khác đi (kế thừa ưu tiên) để tránh kẹt.

So sánh: PIBT (C++ vs Python)

graph.cpp 9 planner.cpp 9+

lacam2 > lacam2 > src > planner.cpp > get_new_config(HNode *, LNode *)

```cpp
bool Planner::get_new_config(HNode* H, LNode* L)
/*

bool Planner::funcPIBT(Agent* ai)
{
  const auto i = ai->id;
  const auto K = ai->v_now->neighbor.size();

  // get candidates for next locations
  for (auto k = 0; k < K; ++k) {
    auto u = ai->v_now->neighbor[k];
    C_next[i][k] = u;
    if (MT != nullptr)
      tie_breakers[u->id] = get_random_float(MT);  // set tie-breaker
  }
  C_next[i][K] = ai->v_now;

  // sort
  std::sort(C_next[i].begin(), C_next[i].begin() + K + 1,
            [&](Vertex* const v, Vertex* const u) {
              return D.get(i, v) + tie_breakers[v->id] <
                     D.get(i, u) + tie_breakers[u->id];
            });

  Agent* swap_agent = swap_possible_and_required(ai);
  if (swap_agent != nullptr)
    std::reverse(C_next[i].begin(), C_next[i].begin() + K + 1);

  // main operation
  for (auto k = 0; k < K + 1; ++k) {
    auto u = C_next[i][k];

    // avoid vertex conflicts
    if (occupied_next[u->id] != nullptr) continue;

    auto& ak = occupied_now[u->id];

    // avoid swap conflicts
    if (ak != nullptr && ak->v_next == ai->v_now) continue;

    // reserve next location
    occupied_next[u->id] = ai;
```

graph.py planner.py 9+

lacam2_py > lacam2 > planner.py > ...

```python
class Planner:

    def funcPIBT(self, ai: Agent) -> bool:
        """Hàm PIBT cho một agent"""
        i = ai.id
        K = len(ai.v_now.neighbor)

        # Lấy các ứng viên cho vị trí tiếp theo
        self.C_next[i] = []
        for k in range(K):
            u = ai.v_now.neighbor[k]
            self.C_next[i].append(u)
            if self.MT is not None:
                self.tie_breakers[u.id] = self.MT.random()
        self.C_next[i].append(ai.v_now)

        # Sắp xếp theo khoảng cách + tie-breaker
        self.C_next[i].sort(
            key=lambda v: self.D.get(i, v) + self.tie_breakers[v.id]
        )

        # Kiểm tra swap
        swap_agent = self.swap_possible_and_required(ai)
        if swap_agent is not None:
            self.C_next[i].reverse()

        # Thao tác chính
        for k in range(K + 1):
            u = self.C_next[i][k]

            # Tránh xung đột đỉnh
            if self.occupied_next[u.id] is not None:
                continue

            ak = self.occupied_now[u.id]

            # Avoid swap conflicts
            if ak is not None and ak.v_next == ai.v_now:
                continue

            # Reserve next location
            self.occupied_next[u.id] = ai
```

Tránh swap collision

- Swap: $A : u \to v$ và $B : v \to u$ trong cùng một bước.
- Planner có các hàm mô phỏng cục bộ để quyết định:
  - swap có **thể** xảy ra không?
  - swap có **bắt buộc** không?
- Khi swap "required": thay đổi thứ tự candidate/ưu tiên để phá bế tắc.

Entry point: `lacam2.solve(...)`

- Hàm `solve` là API chính: tạo Planner và gọi `planner.solve()`.
- Tách interface/core giúp dễ dùng và dễ test.

Hậu kiểm nghiệm (post_processing.py)

- Kiểm tra start/goal đúng.
- Kiểm tra vertex collision theo từng timestep.
- Kiểm tra move hợp lệ (đứng yên hoặc sang ô kề).
- Kiểm tra swap collision.

So sánh C++ vs Python

C++

- Nhanh, tối ưu bộ nhớ
- Mã phức tạp (con trỏ/template)

Python

- Dễ đọc, dễ debug/test
- Chậm hơn do overhead runtime