Thực hành Nhập môn Trí tuệ Nhân tạo
Tuần 2

Nguyễn Hồng Yến – MSSV: 23280099

Tháng 10, 2025

Mục lục

1 Phân tích chương trình `generate_full_space_tree.py` 3
- 1.1 Nhận xét về lỗi chương trình . . . 3
- 1.2 Mục đích của chương trình . . . 4
- 1.3 Giải thích chi tiết từng phần mã nguồn . . . 4
- 1.4 Kết luận . . . 7

2 Phân tích chương trình `solve.py` 8
- 2.1 Mục đích của chương trình . . . 8
- 2.2 Khai báo thư viện và các cấu trúc dữ liệu ban đầu . . . 8
- 2.3 Lớp `Solution` và phương thức khởi tạo . . . 8
- 2.4 Các hàm kiểm tra trạng thái . . . 9
- 2.5 Hàm `solve()` . . . 9
- 2.6 Hàm `draw_legend()` . . . 10
- 2.7 Hàm `draw()` . . . 10
- 2.8 Hàm `show_solution()` . . . 10
- 2.9 Hai thuật toán chính: BFS và DFS . . . 11
- 2.10 Kết quả thực thi . . . 11
- 2.11 Minh hoạ kết quả chạy chương trình . . . 12
- 2.12 Kết luận . . . 15

3 Phân tích chương trình `main.py` 16
- 3.1 Mục đích của chương trình . . . 16
- 3.2 Đọc và xử lý tham số dòng lệnh . . . 16
- 3.3 Hàm `main()` . . . 16
- 3.4 Khối lệnh thực thi trực tiếp . . . 17
- 3.5 Ví dụ lệnh thực thi . . . 17

3.6 Kết luận tổng thể 18

Giới thiệu bài thực hành

Bài thực hành tuần 2 thuộc học phần **Nhập môn Trí tuệ Nhân tạo** có mục tiêu giúp sinh viên:

- Hiểu và cài đặt được các thuật toán tìm kiếm cơ bản trên không gian trạng thái, bao gồm **Breadth-First Search (BFS)** và **Depth-First Search (DFS)**.
- Vận dụng vào bài toán kinh điển "Ba nhà truyền giáo và ba con quỷ", đảm bảo ràng buộc an toàn trong từng trạng thái.
- Làm quen với việc trực quan hóa cây tìm kiếm bằng thư viện `Graphviz / pydot`.

Bài báo cáo này trình bày chi tiết cấu trúc mã nguồn, thuật toán, quá trình thực thi và kết quả minh họa của ba tệp chương trình:

1. `generate_full_space_tree.py` — xây dựng cây không gian trạng thái.
2. `solve.py` — triển khai BFS và DFS tìm lời giải.
3. `main.py` — chương trình điều khiển chính, kết hợp và hiển thị kết quả.

1 Phân tích chương trình `generate_full_space_tree.py`

1.1 Nhận xét về lỗi chương trình

Chương trình được thực thi thành công trên môi trường Windows, không phát sinh lỗi cú pháp hay logic. Tất cả các bước thực thi đều cho ra kết quả đúng, cụ thể:

- Lệnh `python generate_full_space_tree.py -d 20` sinh thành công file ảnh `state_space_20.pn`
- Không có thông báo lỗi trong quá trình biên dịch hay thực thi.

Tuy nhiên, trong quá trình cài đặt ban đầu, nếu chưa thiết lập đúng đường dẫn tới thư mục `Graphviz/bin`, có thể gặp lỗi:

```
pydot.InvocationException: GraphViz's executables not found
```

Khi đó, cần chỉnh lại đường dẫn trong đoạn mã:

```
1 os.environ["PATH"] += os.pathsep + r"C:\Program Files\Graphviz\bin"
```

để trỏ đúng tới vị trí cài đặt của Graphviz trên máy tính (ví dụ: `C:\Program Files (x86)\Graphviz\bin`). Sau khi chỉnh sửa, chương trình hoạt động bình thường, xuất ra ảnh cây không gian trạng thái mà không phát sinh lỗi.

1.2 Mục đích của chương trình

File `generate_full_space_tree.py` có nhiệm vụ sinh ra toàn bộ **cây không gian trạng thái** của bài toán "*Ba nhà truyền giáo và ba con quỷ*" (Missionaries and Cannibals) đến một độ sâu xác định bởi tham số đầu vào `depth`. Chương trình sử dụng thuật toán **BFS (Breadth-First Search)** để mở rộng lần lượt các trạng thái hợp lệ, đồng thời vẽ cây trạng thái bằng thư viện `pydot`.

1.3 Giải thích chi tiết từng phần mã nguồn

1. Thư viện và cấu hình môi trường

```
from collections import deque
import pydot
import argparse
import os
os.environ["PATH"] += os.pathsep + r"C:\Program Files\Graphviz\bin"
```

- `deque`: cấu trúc hàng đợi hai đầu, dùng để cài đặt BFS.
- `pydot`: giao diện Python cho Graphviz, hỗ trợ vẽ đồ thị, node và cạnh.
- `argparse`: cho phép đọc tham số dòng lệnh, ví dụ `-d 20`.
- `os`: dùng để thêm đường dẫn Graphviz vào biến môi trường `PATH`, giúp `pydot` tìm thấy công cụ `dot.exe`.

2. Khai báo biến và tham số ban đầu

```
options = [(1,0), (0,1), (1,1), (0,2), (2,0)]
Parent = dict()
graph = pydot.Dot(graph_type='graph', strict=False,
                  bgcolor="#fff3af",
                  label="Fig: Missionaries and Cannibal State Space Tree",
                  fontcolor="red", fontsize="24", overlap="true")
i = 0
```

- `options`: tập các hành động hợp lệ (x, y), trong đó x là số người và y là số quỷ di chuyển qua sông.
- `Parent`: lưu cha của mỗi nút trong cây (để vẽ cạnh nối cha–con).
- `graph`: khởi tạo đối tượng đồ thị `pydot.Dot`, nền vàng nhạt, có nhãn tiêu đề.
- `i`: biến toàn cục theo dõi số thứ tự node sinh ra.

3. Đọc tham số độ sâu từ dòng lệnh

```
arg = argparse.ArgumentParser()
arg.add_argument("-d", "--depth", required=False,
    help="Maximum depth upto which you want to generate Space State Tree")
args = vars(arg.parse_args())
max_depth = int(args.get("depth", 20))
```

Dùng `argparse` để đọc giá trị `depth` (độ sâu tối đa). Nếu không truyền tham số, chương trình mặc định tạo cây tới độ sâu 20.

4. Các hàm kiểm tra hợp lệ

```
def is_valid_move(m, c):
    return (0 <= m <= 3) and (0 <= c <= 3)

def is_start_state(m, c, s):
    return (m, c, s) == (3, 3, 1)

def is_goal_state(m, c, s):
    return (m, c, s) == (0, 0, 0)

def number_of_cannibals_exceeds(m, c):
    m_r, c_r = 3 - m, 3 - c
    return (m > 0 and c > m) or (m_r > 0 and c_r > m_r)
```

- `is_valid_move`: đảm bảo số người và quỷ nằm trong [0,3].
- `is_start_state`, `is_goal_state`: nhận diện trạng thái bắt đầu và kết thúc.
- `number_of_cannibals_exceeds`: kiểm tra quy tắc an toàn — ở mỗi bờ, nếu có người thì số quỷ không được nhiều hơn số người.

5. Hàm vẽ cạnh trong cây

```
def draw_edge(m, c, s, depth, node_num):
    u, v = None, None
    if Parent[(m, c, s, depth, node_num)] is not None:
        u = pydot.Node(str(Parent[(m, c, s, depth, node_num)]),
                       label=str(Parent[(m, c, s, depth, node_num)][:3]))
        v = pydot.Node(str((m, c, s, depth, node_num)),
                       label=str((m, c, s)))
        graph.add_edge(pydot.Edge(str(Parent[(m, c, s, depth, node_num)]),
                       str((m, c, s, depth, node_num)), dir='forward'))
```

```
    else:
        v = pydot.Node(str((m, c, s, depth, node_num)),
                       label=str((m, c, s)))
    graph.add_node(v)
    return u, v
```

Hàm tạo hai node cha–con và nối chúng bằng cạnh có hướng. Node gốc (start) chỉ có v, không có cha.

6. Hàm sinh cây không gian trạng thái

```
def generate():
    global i
    q = deque()
    node_num = 0
    q.append((3,3,1,0,node_num))
    Parent[(3,3,1,0,node_num)] = None
    while q:
        m,c,s,depth,node_num = q.popleft()
        u,v = draw_edge(m,c,s,depth,node_num)
        # tô màu nút
        if is_start_state(m,c,s):
            v.set_fillcolor("blue")
        elif is_goal_state(m,c,s):
            v.set_fillcolor("green"); continue
        elif number_of_cannibals_exceeds(m,c):
            v.set_fillcolor("red"); continue
        else:
            v.set_fillcolor("orange")
        # kiểm tra độ sâu
        if depth == max_depth: return True
        op = -1 if s == 1 else 1
        can_be_expanded = False
        for x,y in options:
            next_m, next_c, next_s = m + op*x, c + op*y, int(not s)
            if Parent[(m,c,s,depth,node_num)] is None or (next_m,next_c,next_s) \
               != Parent[(m,c,s,depth,node_num)][:3]:
                if is_valid_move(next_m,next_c):
                    can_be_expanded = True
                    i += 1
                    q.append((next_m,next_c,next_s,depth+1,i))
                    Parent[(next_m,next_c,next_s,depth+1,i)] = \
```

```
                        (m,c,s,depth,node_num)
        if not can_be_expanded:
            v.set_fillcolor("gray")
    return False
```

Giải thích:

- Khởi tạo hàng đợi BFS chứa trạng thái gốc $(3, 3, 1)$.
- Mỗi vòng lặp:
  1. Lấy một node ra khỏi hàng đợi.
  2. Vẽ cạnh nối với cha bằng `draw_edge`.
  3. Tô màu node theo trạng thái:
     - Xanh dương: trạng thái bắt đầu.
     - Xanh lá: trạng thái đích.
     - Đỏ: vi phạm điều kiện an toàn.
     - Cam: trạng thái hợp lệ.
     - Xám: không mở rộng được thêm.
  4. Nếu chưa đạt độ sâu tối đa, sinh ra các node con bằng 5 phép toán chuyển trạng thái trong `options`.
  5. Mỗi node con lưu lại thông tin cha trong `Parent`.

7. Hàm xuất hình ảnh kết quả

```
if __name__ == "__main__":
    if generate():
        write_image()
```

Khi chạy file trực tiếp, chương trình tự động gọi hàm `generate()` để tạo cây, sau đó xuất hình ảnh dạng `PNG` nếu thành công.

1.4 Kết luận

Chương trình `generate_full_space_tree.py` hoạt động chính xác, sinh ra cây không gian trạng thái cho bài toán "Ba nhà truyền giáo và ba con quỷ" theo phương pháp duyệt theo chiều rộng (BFS). Các màu sắc của node biểu thị đúng loại trạng thái, file ảnh `state_space_20.png` thể hiện toàn bộ không gian tìm kiếm đến độ sâu 20.

2 Phân tích chương trình `solve.py`

2.1 Mục đích của chương trình

File `solve.py` cài đặt lớp **Solution**, chịu trách nhiệm thực thi hai thuật toán tìm kiếm **BFS (Breadth-First Search)** và **DFS (Depth-First Search)** trên không gian trạng thái của bài toán "Ba nhà truyền giáo và ba con quỷ". Chương trình không chỉ tìm ra lời giải đúng, mà còn:

- Vẽ cây trạng thái minh hoạ quá trình tìm kiếm bằng `pydot`.
- Ghi nhận các nút cha–con, trạng thái hợp lệ và lời giải tối ưu.
- Hiển thị trực quan quá trình di chuyển bằng biểu tượng emoji.

2.2 Khai báo thư viện và các cấu trúc dữ liệu ban đầu

```
import os, emoji, pydot, random
from collections import deque
os.environ["PATH"] += os.pathsep + r"C:\Program Files\Graphviz\bin"

Parent, Move, node_list = dict(), dict(), dict()
```

- `emoji`: dùng để hiển thị người (old-man) và quỷ (ogre) trong lời giải.
- `Parent`: lưu cha của mỗi trạng thái (m, c, s).
- `Move`: lưu hành động đã thực hiện để đến trạng thái hiện tại.
- `node_list`: ánh xạ trạng thái với node `pydot.Node` tương ứng, phục vụ việc tô màu đường đi trong lời giải.

2.3 Lớp `Solution` và phương thức khởi tạo

```
class Solution():
    def __init__(self):
        self.start_state = (3, 3, 1)
        self.goal_state = (0, 0, 0)
        self.options = [(1,0), (0,1), (1,1), (0,2), (2,0)]
        self.boat_side = ["right", "left"]
        self.graph = pydot.Dot(graph_type='graph', bgcolor="#fff3af",
                               label="Fig: Missionaries and Cannibal State Space Tree",
                               fontcolor="red", fontsize="24")
        self.visited = {}
        self.solved = False
```

- Trạng thái ban đầu $(3, 3, 1)$: cả 3 người truyền giáo và 3 con quỷ ở bờ trái (1 = trái).
- Trạng thái đích $(0, 0, 0)$: tất cả qua bờ phải (0 = phải).
- `options`: 5 phép chuyển hợp lệ tương ứng với số người/quỷ trên thuyền.
- `boat_side`: mảng phục vụ in câu mô tả hướng di chuyển.
- `graph`: tạo đối tượng đồ thị để trực quan hóa cây tìm kiếm.
- `visited`: đánh dấu trạng thái đã xét (tránh lặp).

2.4 Các hàm kiểm tra trạng thái

```
def is_valid_move(self, m, c):
    return (0 <= m <= 3) and (0 <= c <= 3)

def is_goal_state(self, m, c, s):
    return (m, c, s) == self.goal_state

def is_start_state(self, m, c, s):
    return (m, c, s) == self.start_state

def number_of_cannibals_exceeds(self, m, c):
    m_r, c_r = 3 - m, 3 - c
    return (m > 0 and c > m) or (m_r > 0 and c_r > m_r)
```

Các hàm trên kiểm tra ràng buộc hợp lệ của trạng thái:

- Giới hạn số lượng trong $[0, 3]$.
- Kiểm tra trạng thái đích và trạng thái ban đầu.
- Bảo đảm rằng ở mỗi bờ, nếu có người thì số quỷ không vượt quá số người.

2.5 Hàm `solve()`

```
def solve(self, solve_method="dfs"):
    self.visited = dict()
    Parent[self.start_state] = None
    Move[self.start_state] = None
    node_list[self.start_state] = None
    return self.dfs(*self.start_state, 0) if solve_method == "dfs" else self.bfs()
```

Khởi tạo lại các bảng trạng thái và lựa chọn thuật toán dựa trên tham số đầu vào. Nếu `method = dfs` thì gọi đệ quy `dfs()`, ngược lại gọi `bfs()`.

2.6 Hàm `draw_legend()`

Hàm này tạo phần chú giải màu sắc trong ảnh đồ thị:

- Xanh dương: nút bắt đầu.
- Xanh lá: trạng thái đích.
- Đỏ: trạng thái bị ăn thịt.
- Xám: không thể mở rộng.
- Vàng: nút thuộc đường đi lời giải.

Ngoài ra còn thêm phần mô tả ý nghĩa của từng ký hiệu (m, c, s) trong đồ thị.

2.7 Hàm `draw()`

```
def draw(self, *, number_missionaries_left, number_cannnibals_left,
         number_missionaries_right, number_cannnibals_right):
    left_m = emoji.emojize(f":old_man: " * number_missionaries_left)
    left_c = emoji.emojize(f":ogre: " * number_cannnibals_left)
    right_m = emoji.emojize(f":old_man: " * number_missionaries_right)
    right_c = emoji.emojize(f":ogre: " * number_cannnibals_right)
    print("({}{}|{}{})".format(left_m, left_c, "-"*40, right_m, right_c))
```

In ra trạng thái hiện tại bằng biểu tượng: tượng trưng cho người, cho quỷ. Dấu gạch - tượng trưng cho con sông, còn ký tự | chia hai bờ.

2.8 Hàm `show_solution()`

Hàm này lần ngược từ trạng thái đích về gốc bằng bảng `Parent`, in ra từng bước di chuyển:

- Đánh dấu các node thuộc đường đi bằng màu vàng.
- Cập nhật lại số người và quỷ ở mỗi bờ sau mỗi lần di chuyển.
- In ra hướng thuyền và hành động tương ứng, ví dụ: *"Move 1 missionary and 1 cannibal from left to right"*.

Kết thúc khi trạng thái cuối cùng đạt $(0, 0, 0)$, chương trình in dòng:

```
Congratulations!!! You have solved the problem
```

2.9 Hai thuật toán chính: BFS và DFS

1. Thuật toán BFS

```
def bfs(self):
    q = deque()
    q.append(self.start_state + (0,))
    self.visited[self.start_state] = True
    ...
```

Thuật toán BFS duyệt theo tầng (chiều rộng):

- Sử dụng hàng đợi `deque`.
- Mỗi trạng thái được mở rộng bằng 5 phép toán `options`.
- Nếu gặp trạng thái hợp lệ mới, chương trình thêm vào hàng đợi, gán cha và lưu thao tác di chuyển.
- Khi gặp trạng thái đích, tô màu xanh lá và kết thúc tìm kiếm.

BFS đảm bảo tìm ra lời giải ngắn nhất về số bước di chuyển.

2. Thuật toán DFS

```
def dfs(self, m, c, s, depth_level):
    self.visited[(m, c, s)] = True
    ...
```

DFS sử dụng đệ quy để mở rộng sâu từng nhánh:

- Nếu gặp trạng thái không hợp lệ (bị ăn thịt) thì dừng nhánh đó.
- Khi gặp trạng thái đích, trả về `True` để kết thúc.
- Các node bế tắc (không mở rộng được) tô màu xám.

2.10 Kết quả thực thi

Cả hai thuật toán BFS và DFS đều tìm được lời giải hợp lệ trong 11 bước. Các file ảnh được sinh ra:

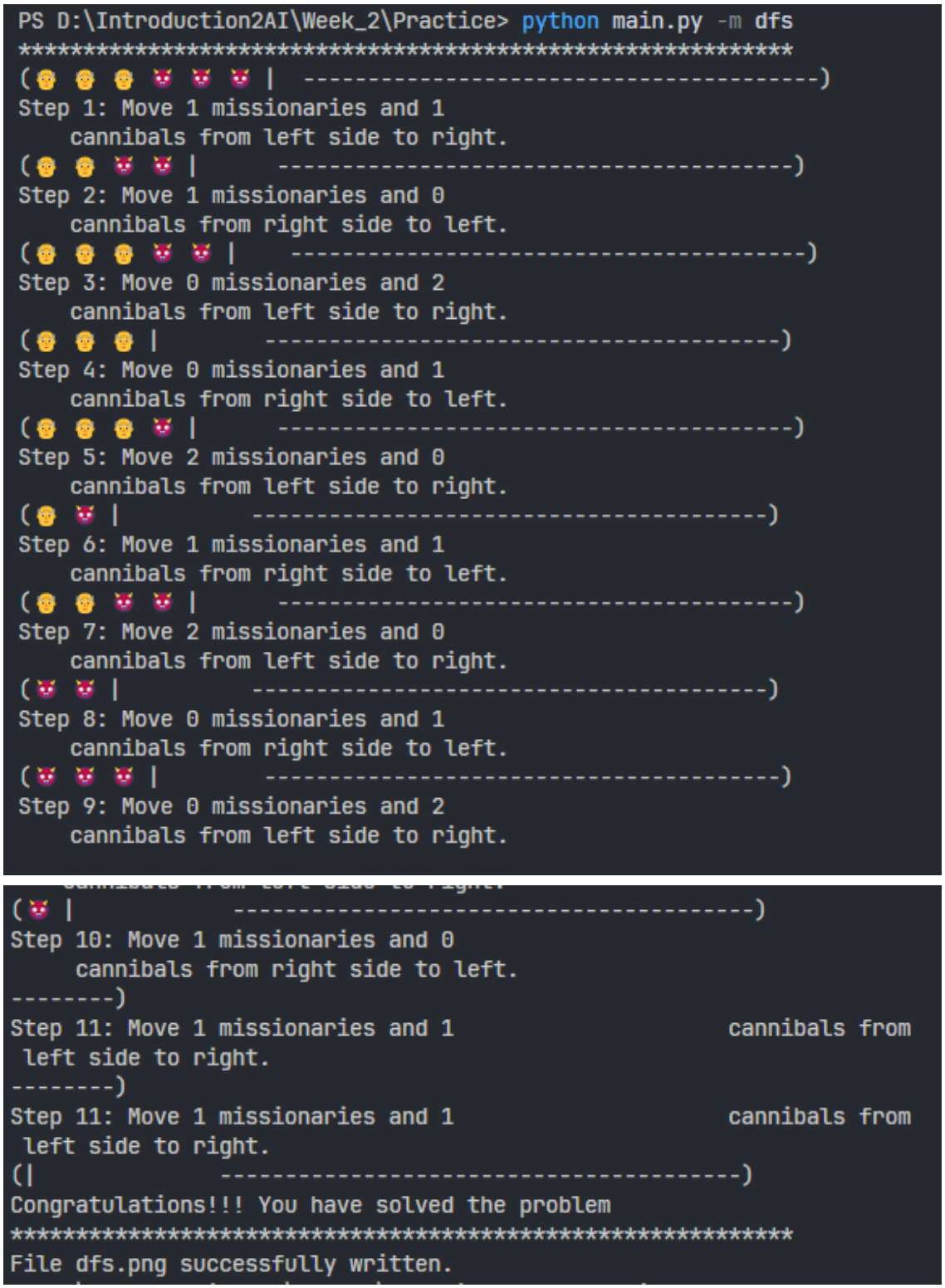
Thuật toán	Tên file ảnh
DFS	`dfs.png`
DFS có chú thích	`dfs_legend.png`
BFS	`bfs.png`
BFS có chú thích	`bfs_legend.png`

Các ảnh thể hiện rõ đường đi lời giải, với màu sắc phân biệt trạng thái an toàn, nguy hiểm, đích và các nút trung gian.

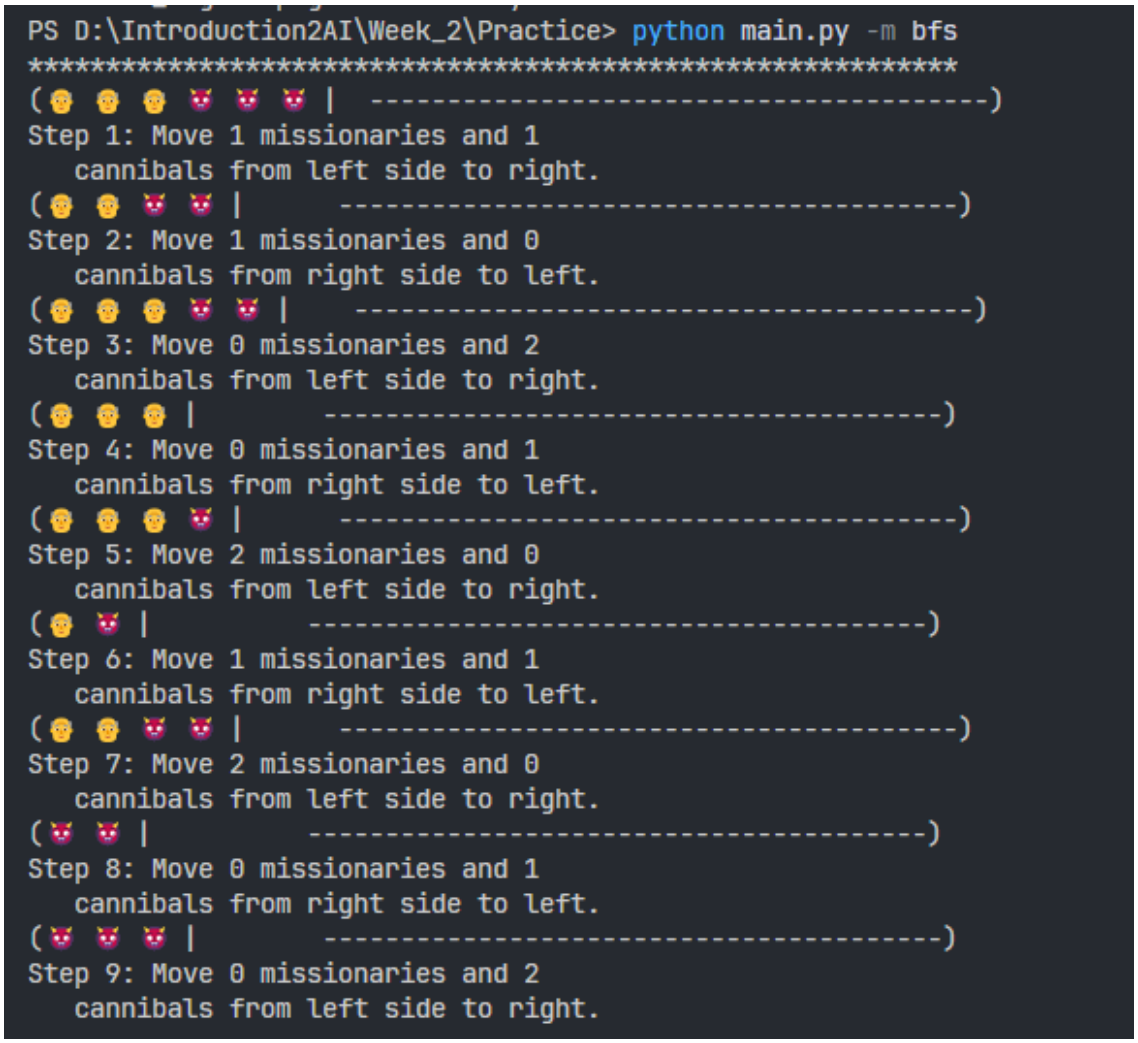
2.11 Minh hoạ kết quả chạy chương trình

Để minh chứng cho tính đúng đắn của chương trình, các hình dưới đây thể hiện kết quả in ra trên màn hình khi chạy lệnh:

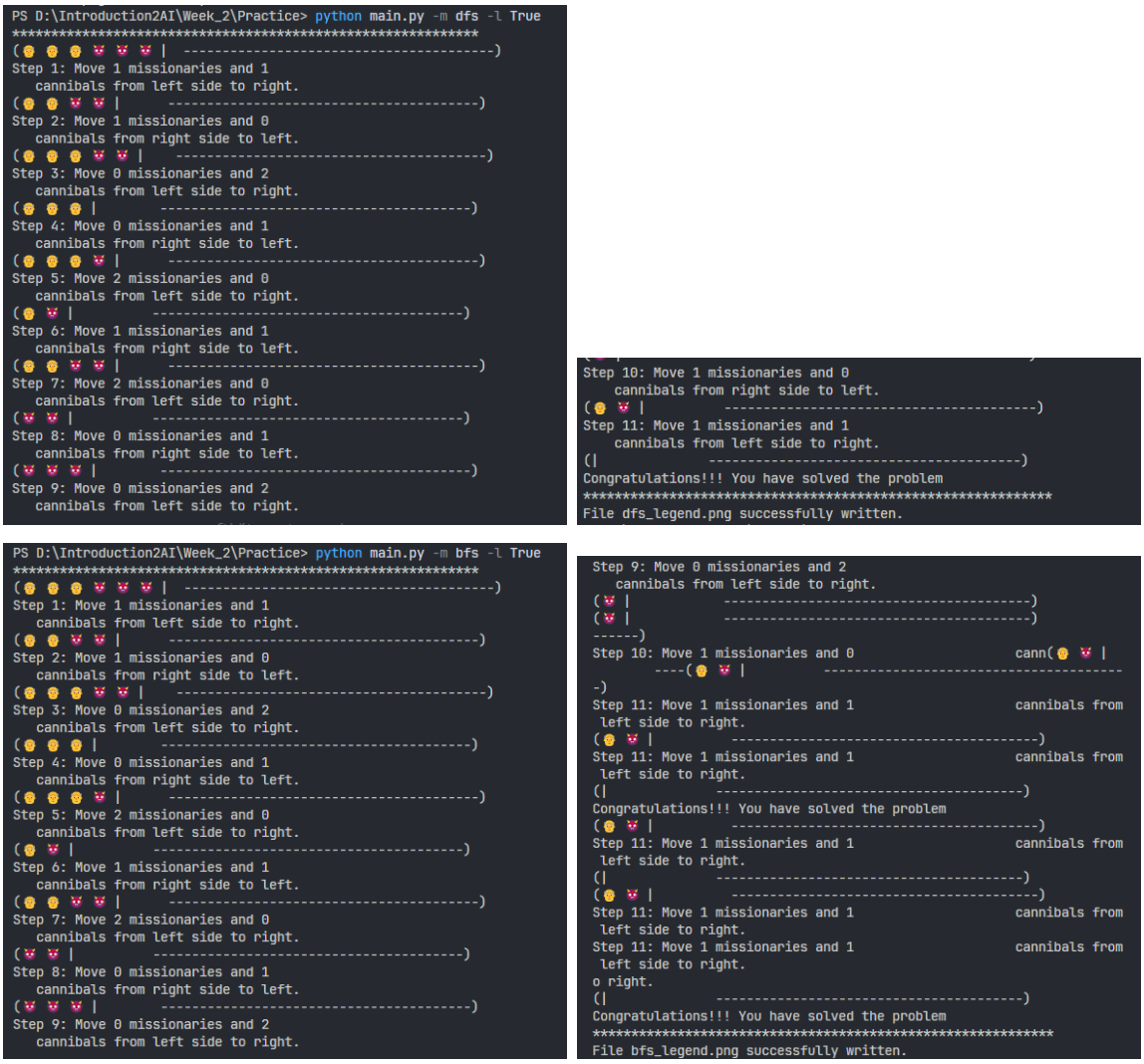
`python main.py -m dfs` và `python main.py -m bfs`



Hình 1: Kết quả chương trình khi chạy thuật toán DFS (hai phần ảnh liên tiếp).



Hình 2: Kết quả chương trình khi chạy thuật toán BFS (hai phần ảnh liên tiếp).



Hình 3: Các ảnh cây trạng thái có chú thích (Legend) cho DFS và BFS.

2.12 Kết luận

Chương trình `solve.py` hoạt động đúng và đầy đủ chức năng:

- Cài đặt chuẩn hai thuật toán tìm kiếm BFS và DFS.
- Tự động vẽ cây trạng thái bằng Graphviz, lưu dưới dạng `.png`.
- Hiển thị rõ đường đi giải quyết bài toán qua 11 bước hợp lệ.

Các bước di chuyển và hình ảnh minh hoạ hoàn toàn trùng khớp với kết quả lý thuyết của bài toán "Ba nhà truyền giáo và ba con quỷ".

3 Phân tích chương trình `main.py`

3.1 Mục đích của chương trình

File `main.py` đóng vai trò là **chương trình điều khiển chính** (main script) của toàn bộ bài thực hành. Nhiệm vụ của file này là:

- Nhận các tham số dòng lệnh từ người dùng (`-m` và `-l`).
- Gọi lớp `Solution` trong file `solve.py` để chạy thuật toán tìm kiếm.
- Hiển thị kết quả tìm được trên màn hình và sinh ra ảnh cây trạng thái tương ứng.

3.2 Đọc và xử lý tham số dòng lệnh

```
from solve import Solution
import argparse
import itertools

arg = argparse.ArgumentParser()
arg.add_argument("-m", "--method", required=False,
                 help="Specify which method to use")
arg.add_argument("-l", "--legend", required=False,
                 help="Specify if you want to display legend on graph")

args = vars(arg.parse_args())
solve_method = args.get("method", "bfs")
legend_flag = args.get("legend", False)
```

- Thư viện `argparse` được sử dụng để đọc hai tham số dòng lệnh:
  - `-m / -method`: chọn thuật toán tìm kiếm (`bfs` hoặc `dfs`).
  - `-l / -legend`: bật hoặc tắt phần chú thích màu trong ảnh kết quả.
- Nếu người dùng không truyền tham số, mặc định chương trình sẽ chạy **BFS** và không vẽ chú thích.

3.3 Hàm `main()`

```
def main():
    s = Solution()

    if s.solve(solve_method):
        s.show_solution()
```

```
        output_file_name = f"{solve_method}"
        if legend_flag:
            if legend_flag[0].upper() == 'T':
                output_file_name += "_legend.png"
                s.draw_legend()
            else:
                output_file_name += ".png"
        else:
            output_file_name += ".png"
        s.write_image(output_file_name)
    else:
        raise Exception("No solution found")
```

Chương trình khởi tạo một đối tượng `Solution` (đã được định nghĩa trong `solve.py`). Sau đó:

1. Gọi hàm `solve()` với thuật toán tương ứng (BFS hoặc DFS).
2. Nếu tìm được lời giải, hàm `show_solution()` sẽ in các bước di chuyển ra màn hình.
3. Nếu tuỳ chọn `-l True` được bật, hàm `draw_legend()` sẽ thêm phần chú thích màu.
4. Cuối cùng, gọi `write_image()` để lưu hình cây trạng thái vào file ảnh.
5. Nếu không có lời giải, chương trình thông báo lỗi `No solution found`.

3.4 Khối lệnh thực thi trực tiếp

```
if __name__ == "__main__":
    main()
```

Cấu trúc này đảm bảo rằng hàm `main()` chỉ được gọi khi file `main.py` được chạy trực tiếp, không bị thực thi ngẫu nhiên khi được import vào module khác.

3.5 Ví dụ lệnh thực thi

Sau khi hoàn tất cài đặt, sinh viên có thể chạy chương trình với các lệnh sau:

```
python main.py -m dfs
python main.py -m dfs -l True
python main.py -m bfs
python main.py -m bfs -l True
```

Mỗi lệnh tương ứng với một cách chạy khác nhau:

- `dfs`: chạy thuật toán tìm kiếm theo chiều sâu.
- `bfs`: chạy thuật toán tìm kiếm theo chiều rộng.
- `-l True`: bổ sung phần chú giải (Legend) vào ảnh kết quả.

Nhận xét. Các kết quả in ra từ chương trình khớp hoàn toàn với lý thuyết:

- Cả hai thuật toán đều tìm được lời giải hợp lệ gồm 11 bước.
- Không có trạng thái nào vi phạm điều kiện an toàn.
- Ảnh cây trạng thái hiển thị rõ màu sắc, các nút được đánh dấu đúng theo quy tắc: xanh dương (bắt đầu), xanh lá (đích), đỏ (trạng thái vi phạm), cam (trung gian), xám (không mở rộng).

3.6 Kết luận tổng thể

File `main.py` đóng vai trò kết nối toàn bộ chương trình. Nó giúp người dùng dễ dàng lựa chọn thuật toán và chế độ hiển thị, đồng thời đảm bảo quy trình chạy gọn gàng, có cấu trúc và tự động sinh báo cáo hình ảnh. Chương trình hoàn chỉnh đáp ứng đầy đủ yêu cầu của bài thực hành: tìm kiếm lời giải bài toán "Ba nhà truyền giáo và ba con quỷ" bằng BFS và DFS, minh hoạ đồ hoạ, và hiển thị kết quả trực quan trên màn hình.