BÁO CÁO BÀI TẬP THỰC HÀNH TUẦN 2

1. Cài đặt và thực thi chương trình

Em đã cài đặt 3 file code của cô và không tìm thấy lỗi trong khi thực thi chương trình bằng các dòng lệnh như:

```
pip install -r requirements.txt
python generate_full_space_tree.py -d 8 # (với d là độ sâu (depth=8))
python generate_full_space_tree.py -d 20 # (depth = 20)
python generate_full_space_tree.py -d 10 # phần làm tương tự với 10
python generate_full_space_tree.py -d 40 # phần làm tương tự với 40

python main.py -m dfs
python main.py -m dfs -l True # dfs với legend
python main.py -m bfs
python main.py -m bfs -l Trye # bfs với legend
```

Và tất cả đều xuất ra hình ảnh đúng theo yêu cầu.

2. Đọc hiểu code và trình bày

2.1. generate_full_space_tree.py

KHỞI ĐẦU

Bắt đầu với file `generate_full_space_tree.py` ta có các bước khởi đầu như sau:

Đây là phần code để thêm vào các thư viện sẽ dùng cho file này như `deque`, `pydot`, `argparse`, `os`. Trong đó:

- `deque`: là hàng đợi hai đầu
- `pydot`: giúp vẽ và thao tác với đồ thị, là giao diện cho Graphviz
- `argparse`: giúp định nghĩa và quản lý các tham số đầu vào của chương trình

Phần này là để set up tới thư mục bin của phần mềm Graphviz

```
os.environ["PATH"] += os.pathsep + '/usr/bin/dot'
```

Options là để đưa ra các bước đi hợp lệ, vì một thuyền chỉ đi được hai người nên cần phải đi `(x, y)` với `x` + `y` không lớn hơn 2 và `x` đại diện cho số người và `y` đại diện cho số quỷ

```
options = [(1, 0), (0, 1), (1, 1), (0, 2), (2, 0)]
```

Phần còn lại để định nghĩa đồ thị và các tham số đầu vào khi chạy chương trình.

CÁC HÀM KIỂM TRA

Hàm `is_valid_move` để kiểm tra rằng số lượng người sau khi move có hợp lý hay không tránh trường hợp vận chuyển `người` hay `quỷ` đi trong khi bờ đó không còn nhân vật tương ứng. Ví dụ:

Bên trái có 3 quỷ 0 người nhưng lại muốn vận chuyển tiếp 1 người từ bên trái đi sang phải $\rightarrow$ lỗi.

Hàm `write_image` xuất hình ảnh cây đã được tạo với độ xâu d ra ngoài.

Hàm `draw_edge` như cái tên, nó có nhiệm vụ vẽ ra các cạnh của đồ thị, trong đó các nodes biểu diễn trạng thái và các cạnh biểu diễn sự chuyển đổi giữa các trạng thái đó. Tất cả quá trình vẽ sử dụng thư viện `pydot` để vẽ.

Hai hàm `is_start_state` và `is_goal_state` dùng để kiểm tra xem trạng thái hiện tại có phải là trạng thái bắt đầu, hay là trạng thái kết thúc.

Hàm `number_of_cannibales_exceeds` để check xem nếu 1 bên bờ mà có tồn tại ít nhất 1 người và số lượng quỷ bên bờ đó lớn hơn 1 sẽ return về `True`, nghĩa là nhân vật người sẽ gặp nguy hiểm.

HÀM GENERATE

```
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
```

Phần này là để lấy ra biến `i` là biến toàn cục dùng để đánh số thứ tự các nút. Sau đó tạo một hàng đợi hai đầu và thêm trạng thái bắt đầu vào trong đó. Và đánh dấu Parent của trạng thái bắt đầu là `None`.

```
    while q:
        number_missionaries, number_cannibals, side, depth_level, node_num =
```

```
q.popleft()
        u, v = draw_edge(number_missionaries, number_cannibals, side,
depth_level, node_num)
```

Phần này để lặp đến khi nào mà hàng đợi không còn trạng thái, ta lấy ra nút đầu ở hàng đợi, sau đó vẽ cạnh từ trạng thái `u` sang trạng thái con `v` bằng hàm `draw_edge()`.

Và các câu lệnh rẽ nhánh sau đó dùng để phân biệt màu với các trạng thái:

- Trạng thái bắt đầu là màu xanh biển
- Trạng thái kết thúc là màu xanh lá cây
- Trạng thái không hợp lệ (nhân vật người gặp nguy hiểm) là màu đỏ → không thể mở rộng thêm
- Các trạng thái khác không phải các trạng thái trên được đánh là màu cam

```
        if depth_level == max_depth:
            return True
```

Dùng để kiểm tra xem cây có đạt được tới độ sâu tối đa hay không, nếu có trả về `True`, ngược lại trả về `False`.

```
        op = -1 if side == 1 else 1
        can_be_expanded = False
```

Phần `op` là dùng để xác định hướng di chuyển dựa trên vị trí của con thuyền. Nếu nó đang ở bên trái (side = 1) thì cần di chuyển sang bên phải (op = -1) và ngược lại.
Biến `can_be_expanded` để theo dõi xem nếu trạng thái hiện tại có được mở rộng ra tiếp các thái con hay không.

```
        for x, y in options:
            next_m, next_c, next_s = number_missionaries + op * x,
number_cannibals + op * y, int(not side)

            if Parent[(number_missionaries, number_cannibals, side,
depth_level, node_num)] is None or \
                (next_m, next_c, next_s) != Parent[(number_missionaries,
number_cannibals, side, depth_level, 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_level + 1, i))
```

```
                    Parent[(next_m, next_c, next_s, depth_level + 1, i)] =
(number_missionaries, number_cannibals, side, depth_level, node_num)
```

Nó duyệt tất cả các hành động mà có thể xảy ra tiếp theo, sau đó kiểm tra điều kiện như sau:

- Không trùng với trạng thái cha (tránh lặp lại)
- Có là trạng thái hợp lệ: dùng hàm `is_valid_move()`

Sau đó nếu qua được vòng kiểm tra sẽ được đưa vào hàng đợi và biến `can_be_expanded` sẽ thành `True`.

Từ đó nếu không có trạng thái con nào mà được mở rộng thì `can_be_expanded` sẽ vẫn là `False` → dừng thuật toán và trả về `False`.

Vậy trong hàm main(), nếu cây được mở rộng tới độ dài `max_depth` thì nghĩa là có thể tạo ra đồ thị, nếu không nó sẽ không thỏa mãn.

2.2. solve.py

Đây là file chứa class `Solution`, class dùng để giải thuật toán BFS và DFS trên cây tìm kiếm.

CÁC HÀM KHỞI TẠO

Hàm `__init__` đơn giản là để khởi tạo trạng thái ban đầu, kết thúc, các bước đi, đồ thị (vẽ bằng Graphviz), boat_side là thuyền đang ở bên nào, visited (để không lặp lại trạng thái đã duyệt qua), solved thể hiện rằng vấn đề đã được giải quyết chưa.

Các hàm vẽ đồ thị:

- `write_image()`: Xuất hình cây trạng thái ra để trực quan
- `draw_legend()`: Xuất hình cây trạng thái kèm theo các chú thích
- `draw()`: vẽ trạng thái hiện tại lên console với các emoji

CÁC HÀM KIỂM TRA

Tương tự file `generate_full_space_tree.py`, file này cũng có các hàm có chức năng tương tự như:

`is_valid_move`, `is_goal_state`, `is_start_state`, `number_of_cannibals_exceeds`

CÁC HÀM GIẢI BÀI TOÁN

Hàm `solve()` có nhận vào tham số `solve_method`, nó sẽ giải bài toán dựa trên tham số đó truyền vào gồm có "dfs" hoặc "bfs".

BFS

Trong hàm này, ta vẫn tạo ra hàng đợi, thêm start_state vào hàng đợi và visited. Sau đó ta lặp đến khi nào mà hàng đợi không còn phần tử nào. Quá trình hoạt động như sau:

- Lấy ra nút đầu tiên của hàng đợi → Vẽ cạnh
- Xét xem nút này là thuộc trạng thái nào để gán màu vẽ hình
- Đặt hướng di chuyển của thuyền là -1 nếu side = 1 và ngược lại là 1 nếu side = 0. Khởi tạo `can_be_expanded` để giám sát xem cây có mở rộng được thêm không.
- Duyệt qua tất cả các hành động trong `option` để tạo ra các trạng thái mới
- Với mỗi hành động, kiểm tra xem nó có trong visisted không, có phải là một trạng thái hợp lệ không. Nếu đúng thì thêm vào hàng đợi, và đặt `can_be_expanded = True`
- Nếu không thể mở rộng, nghĩa là `can_be_expanded = False` thì dừng thuật toán và trả về `False`, nghĩa là không thể giải được với BFS.

DFS

Hàm DFS trong class `Solution()` này sử dụng kỹ thuật đệ quy quay lui.

Đầu tiên khởi tạo DFS:

```
def dfs(self, number_missionaries, number_cannibals, side, depth_level):
    self.visited[(number_missionaries, number_cannibals, side)] = True

    u, v = self.draw_edge(number_missionaries, number_cannibals, side,
depth_level)
```

Đánh dấu trạng thái đầu vào là đã được thăm, sau đó tiến hành dùng `draw_edge` để vẽ trong đồ thị để visualize.

Sau đó là các bước để kiểm tra trạng thái hiện tại là trạng thái thế nào để phân màu vẽ.

```
    if self.is_start_state(number_missionaries, number_cannibals, side):
        v.set_style("filled")
        v.set_fillcolor("blue")
    elif self.is_goal_state(number_missionaries, number_cannibals, side):
        v.set_style("filled")
        v.set_fillcolor("green")
        return True
    elif self.number_of_cannibals_exceeds(number_missionaries,
number_cannibals):
        v.set_style("filled")
        v.set_fillcolor("red")
        return False
    else:
```

```
        v.set_style("filled")
        v.set_fillcolor("orange")
```

Tương tự như code `generate_full_space_tree()`:

- Trạng thái bắt đầu → xanh dương
- Trạng thái kết thúc → xanh lá → trả về True vì đã tìm thấy lời giải
- Trạng thái không hợp lệ → là khi số quỷ lớn hơn số người với số người ≥ 1 → màu đỏ → trả về False để dừng tiếp tục
- Trạng thái khác → màu cam → tiếp tục mở rộng các trạng hái tiếp theo theo chiều sâu

Duyệt trạng thái con

```
solution_found = False
    operation = -1 if side == 1 else 1

    can_be_expanded = False

    for x, y in self.options:
        next_m, next_c, next_s = number_missionaries + operation * x,
number_cannibals + operation * y, int(not side)

        if (next_m, next_c, next_s) not in self.visited:
            if self.is_valid_move(next_m, next_c):
                can_be_expanded = True
                Parent[(next_m, next_c, next_s)] = (number_missionaries,
number_cannibals, side)
                Move[(next_m, next_c, next_s)] = (x, y, side)
                node_list[(next_m, next_c, next_s)] = v

                solution_found = (solution_found or self.dfs(next_m, next_c,
next_s, depth_level + 1))
                if solution_found:
                    return True
```

Trong vòng for nó sẽ tạo ra trạng thái con thông qua các hành động đã được định nghĩa trong `options`, sau đó kiểm tra tính hợp lệ và visited, cuối cùng gọi đệ quy `dfs` tới trạng thái mới vừa tạo luôn và gán nó cho `solution_found`, nếu `solution_found` là True, nghĩa là đã tìm thấy trạng thái kết thúc thì sẽ trả về `True`.

Quay lui

```
if not can_be_expanded:
        v.set_style("filled")
        v.set_fillcolor("gray")

    self.solved = solution_found
    return solution_found
```

Nếu trạng thái không thể mở rộng thêm (nghĩa là không còn trạng thái con nào hợp lệ), nó sẽ được vẽ màu xám, và trả về `solution_found = False`.

2.3. main.py

Trong file này chỉ đơn giản là thêm vào thư viện `argparse`, và từ file `solve.py` thêm class `Solution` vào.

Nó lấy từ console khi chạy file ra -m là chỉ định thuật toán để chạy (bfs hoặc dfs), -l chỉ định có vẽ ra hình ảnh có legend hay không (True hoặc False)

Sau đó khởi tạo class Solution và chạy thuật toán đã nhập vào, cuối cùng xuất ra hình ảnh tương ứng.