Bài tập thực hành Nhập môn trí tuệ nhân tạo Tuần 2

Họ và tên: Nguyễn Công Tiến Dũng

MSSV: 22280014

- Sau khi cài đặt chương trình thì chương trình đã thực thi thành công mà không báo lỗi.
- Phân tích lại ý nghĩa của các file code đã cho:

  - Đầu tiên là file *generate_full_space_tree.py*:
    Code trên xây dựng và vẽ một cây không gian trạng thái (state space tree) cho bài toán "Missionaries and Cannibals" sử dụng thư viện pydot để vẽ đồ thị và argparse để cấu hình tham số.
    1. Dùng thư viện argparse để phân tích đối số truyền vào từ dòng lệnh
    2. Khởi tạo một số biến ban đầu:
       Dùng module os để thiết lập đường dẫn tới thư mục của Graphviz, giúp vẽ đồ thị.
       Option: khả năng di chuyển của “Missionaries” và “Cannibals” tương ứng với x và y.
       Parent: Dictionary lưu trữ các trạng thái cha của từng trạng thái.
       graph: Tạo đối tượng pydot.Dot làm đồ thị.
       argparse: Dùng để thiết lập độ sâu tối đa cho không gian trạng thái của cây, mặc định độ sâu là 20.
    3. Các hàm hỗ trợ
       is_valid_move(): Kiểm tra xem số lượng người truyền giáo và kẻ ăn thịt người có nằm trong phạm vi hợp lệ (0 đến 3) hay không.
       write_image(): Lưu đồ thị đã vẽ thành file ảnh PNG.
       draw_edge(): Hàm vẽ các cạnh của đồ thị giữa các trạng thái. Mỗi trạng thái là một node và các cạnh biểu diễn các bước di chuyển hợp lệ. Dùng các thư viện như Graphviz và pydot để xây dựng đồ thị thông qua trạng thái u,v được kết nối với nhau bằng edge
       is_start_state() và is_goal_state(): Xác định trạng thái bắt đầu và trạng thái mục tiêu của bài toán.
       number_of_cannibals_exceeds(): Kiểm tra xem số lượng kẻ ăn thịt người có vượt quá số lượng người truyền giáo ở bất kỳ phía nào của sông hay không.
    4. Hàm generate()
       Hàm generate() triển khai thuật toán duyệt đồ thị BFS (Breadth-First Search) để tạo cây không gian trạng thái từ trạng thái ban đầu (3, 3, 1)

(ba người truyền giáo, ba kẻ ăn thịt người và thuyền ở bên trái) đến trạng thái mục tiêu (0, 0, 0).
Trong khi duyệt, mỗi trạng thái được thêm vào hàng đợi q dưới dạng tuple chứa số lượng người truyền giáo, kẻ ăn thịt người, vị trí của thuyền, mức độ sâu hiện tại và số node.
Mỗi node được vẽ với màu đặc trưng:

- Xanh lam cho trạng thái bắt đầu, kiểm tra trạng thái bắt đầu bằng hàm is_start_state().
- Xanh lá cho trạng thái mục tiêu, dùng hàm is_goal_state() để kiểm tra
- Đỏ cho các trạng thái không hợp lệ (số kẻ ăn thịt người vượt số người truyền giáo), kiểm tra bằng hàm number_of_cannibals_exceeds()
- Cam cho trạng thái hợp lệ cho các trường hợp còn lại
- Xám cho trạng thái không thể mở rộng.

Hàm cũng lưu trạng thái cha (Parent) của mỗi trạng thái con để xây dựng cây

Khi chạy chương trình nó sẽ gọi đến hàm generate() nếu thành công thì sẽ trả về ảnh không gian cây trạng thái ở file PNG

- Tiếp đến là file *solve.py*:
  1. Khởi tạo các biến toàn cục:
     Thiết lập module os để lưu đường dẫn tới Graphviz.
     Parent: Lưu trạng thái cha của mỗi trạng thái.
     Move: Lưu lại các bước di chuyển nào đã dẫn đến trạng thái đó.
     Node_list: Lưu các node đồ thị tương ứng với các trạng thái để dễ dàng tô màu các trạng thái có trong lời giải.
  2. Class Solution: chứa các hàm chính để giải quyết bài toán

     2.1 Hàm __init__(): Thiết lập trạng thái ban đầu (start_state) và trạng thái mục tiêu (goal_state) của bài toán. Self.options() các phép di chuyển với các cặp giá trị (missionaries, cannibals).self.graph() sử dụng pydot để vẽ các trạng thái không gian tìm kiếm.

     2.2 Hàm is_valid_move: kiểm tra điều kiện ràng buộc số lượng của missionaries và cannibals trong khoảng từ 0 đến 3 trên mỗi bờ.

     2.3 Hàm is_goal_state() và is_start_state(): Tương tự như ở file generate_full_space_tree.py, dung để kiểm tra node ở trạng thái mục tiêu hay ở trạng thái ban đầu.

     2.4 Hàm number_of_cannibals_exceeds(): dùng để kiểm tra liệu số cannibals có vượt quá số lượng missionaries hay không(kiểm tra trên 2 bờ bên trai và bên phải)

2.5 Hàm write_image(): Dùng để lưu đồ thị ở dạng PNG sau khi vẽ bằng pydot. Nếu có lỗi thì hiển thị lỗi bằng Exception và dừng chương trình.

2.6 Hàm solve(): với tham số truyền vào là solve_method bằng “dfs” hay “bfs” để có thể chọn cách giải quyết bài tóa bằng thuật toán lựa chọn mặc định là “dfs”. Đồng thời cũng khởi tạo các trạng thái bắt đầu để bắt đầu tìm kiếm như Parent, Move và node_list.

2.7 Hàm draw_legend(): Hàm này sẽ được thực thi để vẽ đồ thị kèm theo legend(chú thích) lên đồ thị để giải thích ý nghĩa của các trạng thái.

2.7.1 Đầu tiên khởi tạo một subgraph bằng pydot.Cluster() với biến graphlegend.
2.7.2 Thêm các node vào Cluster: Các nút biểu thị trạng thái khác nhau trong bài toán, mỗi nút có một nhãn và một màu sắc riêng biệt.
+ fillcolor=”blue”: Nút được tô màu xanh
+ label=”Start Node”: Nhãn hiển thị của nút là Start Node.
+ fontcolor=”white”: màu chứ nhãn là trắng.
+ width=”2”, fixedsized=”true”: Điều chỉnh kích thước và cố định kích thước của nút.

2.7.3 Tương tự cho các node khác (từ 2 đến 7) lần lượt đại diện cho các trạng thái:

+ node2: Nút đã bị “Killed” với màu đỏ.

+ node3: Nút “Solution” với màu vàng .

+ node4: Nút không thể mở rộng với màu xám.

+ node5: Nút “Goal” màu xanh lá.

+ node6: Đây là nút đặc biệt khi chứa phần mô tả trạng thái bài toán Missionaries và Cannibals với (m, c, s) biểu diễn trạng thái và số lượng Missionaries(m), số Cannibals© và vị trí thuyền(s). 1 là bờ bên trái, 0 là bờ bên phải.

+ node7: Nút có con với màu vàng.

2.7.4 Cuối cùng là them Cluster Legend vào đồ thị chính và nối các nút:

+Thêm Cluster bằng lệnh:
self.graph.add_subgraph(graphlegend)

+Thêm các cạnh pydot.Edge(…, style=”invis”) được thêm giữa các nút để xắp xếp chúng theo thứ tự nhất định mà không vẽ đường nối.

2.8 Hàm draw(): Hàm này in trạng thái hiện tại trên console với các biểu tượng bằng emoji tương ứng với số lượng Missionaries và Cannibals tương ứng trên mỗi bờ.

2.9 Hàm show_solution(): Hàm này lần theo tưng bước từ trạng thái mục tiêu(goal_state) về trạng thái ban đầu(start_state), hiển thị các trạng thái lên bờ và in các bước di chuyển cần thiết để giải quyết bài toán.

2.9.1 khởi tạo các biến:

path, steps, nodes: Các danh sách này lưu lại đường đi(path), các bước di chuyển(steps), và các nút tương ứng trong đồ thị(nodes).

2.9.2 Truy ngược các bước:

+ Lặp vòng lặp While cho đến khi state trả về None

+ Mỗi vòng lặp: thêm trạng thái hiên tại vào danh sách path.append(state). Thêm bước di chuyển tương ứng vào steps, nơi Move là dic() lưu thông tin bươc di chuyển (m, c, s). Thêm các nút tương ứng từ danh sách các nút nodes.append(node_list[state]). Cuối cùng là cập nhật state bằng Parent[state]

+ Sau khi hoàn thành vòng lặp, steps và nodes được đảo ngược lại để bắt đầu từ trạng thái đầu đến trạng thái kết thúc steps, nodes = steps[::-1], nodes[::-1]

2.9.3 Khởi tạo số lượng Missionaries và Cannibals: bên trái (3, 3) bên phải (0, 0)

2.9.4 In trạng thái bắt đầu bằng gọi lệnh self.draw(…) để vẽ trạng thái ban đầu

2.9.5 Duyệt qua các bước và cập nhật trạng thái: duyệt bằng for các bước trong steps và các nút trong nodes (bỏ qua bước bắt đầu). Sau đó thiết lập style nút là “filled” và đặt màu nền nút thành màu vàng.

2.9.6 Di chuyển và cập nhật số lượng:
"Step {i+1}: Move {number_missionaries} missionaries and {number_cannibals} \ cannibals from {self.boat_side[side]} to {self.boat_side[int(not side)]}": in ra các bước hiện tại

Kiểm tra op=-1 thì thuyền ở bờ bên phải(sẽ trừ số lượng ở bờ trái và cộng vào bờ phải) hoặc 1 nếu thuyền ở bờ trái.

2.9.7 Vẽ trạng thái sau mỗi bước bằng hàm draw() trên màn hình console. Và sau đó in lời chúc mừng khi hoàn thành khi kết thúc vòng lặp

2.10 Hàm draw_edge(): dùng để thêm một cạnh giữa các nút trong đồ thị, biểu diễn một bước di chuyển trong bài toán Missionaries and Cannibals

2.10.1 Khởi tạo nút u, v trong đó u là nút cha v là nút con.

2.10.2 Kiểm tra xem trạng thái hiện tại có cha if Parent[(number_missionaries, number_cannibals, side)] is not None. Tiếp đến tạo nút cha(u) bằng pydot.Node() và them nó vào đồ thị. Tạo tươn tự cho nút con(v). Cuối cùng là tạo edge bằng pydot.Edge() để nối giữa nút u và v.

2.10.3 Nếu rơi vào nút không có cha tức trạng thái bắt đầu thì chỉ cần tạo nút v mà không cần tạo cạnh

2.10.4 Trả về 2 nút u, v

3. Thuật toán tìm kiếm:

3.1 Hàm bfs():

Triển khải xây dựng tìm kiếm bằng giải thuật bfs để tìm kiếm theo chiều rộng. Sử dụng hàng đợi q giúp duyệt trạng thái theo thứ tự FIFO

B1: Thêm trạn thái ban đầu vào q bằng q.append() kèm theo chiều sâu depth_level=0

B2: Dùng vòng lặp while q để lặp đến khi q rỗng. Dùng q.popleft() để lấy ra số lượng Missionaries, Cannibals và side.

B3: Khởi tạo 2 nút u, v bằng trạng thái lấy ra từ q và vẽ edge giữa chúng.

B4: Kiểm tra và gắn màu cho trạng thái: Nếu là start thì gán màu xanh dương, nếu là đích thì gan màu xanh lá, nếu không hợp lệ thì gán màu đỏ, còn binh thường thì để màu cam.

B5: Tính toán và them trạng thái tiếp theo: Dùng op để xác định hướng di chuyển của thuyền nếu side là 1 thì op=-1 thì di chuyển về bờ trái. Ngược lại op là 1. Sau đó dùng vòng lặp để chọn options di chuyển ch trạng thái kế tiếp.

B6: Kiểm tra và thêm trạng thái hợp lệ: Kiểm tra đã được thăm bằng visited() tiếp đến là kiểm tra di chuyển hợp lệ bằng is_valid_move() nếu đúng thì can_be_expanded = True và thêm node vào trong hàng

đợi q với depth_level + 1. Sau đó lưu trạng thái Parent, Move và cập nhật node_list.

B7: Kiểm tra nếu can_be_expanded == False thì đánh dấu trạng thái bằng màu xám.

B8: Trả về False nếu hàng đợi rỗng mà vẫn k tìm thấy trạng thái đích.

3.2 Hàm dfs(): dùng để tìm kiếm theo chiều sâu trong bài toán Missionaries and Cannibals.

B1: Đánh dấu trạng thái đã thăm self.visited(…) = True

B2: Vẽ cạnh giữa các trạng thái bằng cách khởi tạo 2 nút cha(u), con(v) và dùng draw_edge() để vẽ 2 cạnh kết nối.

B3: Xác định trạng thái hiện tại và gán màu tương tự như thuật toán bfs ở trên.

B4: Khởi tạo biến để duyệt qua các trạng thái tiếp theo. Solution_found=False là biến để theo dõi nếu tìm thấy giải pháp phù hợp trong các trạng thái con, dung operation xác định hướng di chuyển và biến can_be_expanded=False.

B5: dùng vòng lặp for duyệt qua các options(x, y) và khởi tạo trạng thái tiếp theo (next_m, next_c, next_s). Sau đó kiểm tra nếu chưa visited() và có hướng di chuyển hợp lệ self.is_valid_move() thì đánh dấu can_be_expanded=True, lưu trạng thái Parent và Move, node_list. Gọi đệ quy dfs() trên trạng thái con nếu tìm thấy giải pháp(solution_found = True) thì trả về True.

B6: Nếu trạng thái không thể mở rộng thì tô màu xám cho nút v.

B7: Cập nhật trạng thái self.solved = solution_found và trả vê solution_found.

- cuối cùng là file *main.py*:
  1. Import thư viện:

     Import class Solution của file solve.py: from solve import Solution.

     Thư viện argparse để giúp phân tích đối số từ dòng lệnh, cho phép dễ dạng lựa chọn các phương pháp giải (method) hay hiển thị chú thích(legend) trong hình hay không.

  2. Thiết lập Argument Parser giúp sử lí tham số truyền từ dòng lệnh:
     -m / --method: Cho phép người dùng chọn phương pháp giải mặc định là BFS.
     -l / --legend: Cho phép người dùng chỉ định có muốn hiển thị chú thích trên bản đồ hay không.

3. Phân tích các đối số truyền từ dòng lệnh bằng vars() để chuyển đổi sang dictionary. solve_method và legend_flag sẽ khởi tạo các giá trị mặc định nếu không có đối số được truyền vào.

4. Hàm main():

- Khởi tạo đối tượng s = Solution()

- Gọi phương thức giải s.solve(solve_method) và nếu có lời giải thì trả về s.show_solution() để hiển thị chuỗi các bước giải trên console.

- Tạo output_file_name trường hợp có legend và không có legend tương ứng với solve_method.

- Lưu ảnh của cây trạng thái thành file ảnh với tên output_file_name bằng s.write_image(output_file_name)

- Đoạn mã if __name__ == “__main__” đảm bảo chỉ chạy khi mã này được thực thi trực tiếp, không phải khi được import từ tệp khác.