Trường Đại học Khoa học Tự Nhiên
ĐHQG.TPHCM

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

TPHCM, ngày 4 tháng 11 năm 2024

BÁO CÁO THỰC HÀNH – NHẬP MÔN TRÍ TUỆ NHÂN TẠO – LẦN 1

Họ và tên: Nguyễn Thanh Kiên.
MSSV: 22110092.

Bài toán:

Có 3 người truyền giáo và 3 con quỷ ở bờ bên trái của một con sông, cùng với con thuyền có thể chở được 1 hoặc 2 người. Nếu số quỷ nhiều hơn số người truyền giáo trong một bờ thì số quỷ sẽ ăn thịt số người truyền giáo. Tìm các để đưa tất cả qua bờ sông bên kia (bên phải) sao cho số người không ít hơn số quỷ ở cùng 1 bờ (bên trái hay bên phải), nghĩa là không ai bị ăn thịt. Gọi (a, b, k) với 0 <= a, b <= 3 , trong đó a là số người, b là số con quỷ ở bên bờ bên trái, k = 1 nếu thuyền ở bờ bên trái và k = 0 nếu thuyền ở bờ bên phải. Khi đó, không gian trạng thái của bài toán được xác định như sau:

- Trạng thái ban đầu là (3, 3, 1).
- Thuyền chở qua sông 1 người, hoặc 1 con quỷ , hoặc 1 người và 1 con quỷ, hoặc 2 người, hoặc 2 con quỷ => các phép toán chuyển từ trạng thái này sang trạng thái khác là: (1, 0), (0, 1), (1, 1), (2, 0), (0, 2) (trong đó (x, y) là số người và số quỷ di chuyển từ bờ bên trái qua bờ bên phải hay ngược lại).
- Trạng thái kết thúc là (0, 0, 0).

Tổng quát về code:

1. May mắn thay, các chương trình em gõ và thực thi đều không bị lỗi nào hết, do đã install pip, pydot, graphviz.
2. Để giải quyết bài toán này, ta chia ra thành 3 chương trình với các mục đích khác nhau:

a. Chương trình Generate Full Space Tree: Sử dụng phương pháp *tạo cây trạng thái đầy đủ* để biểu diễn tất cả các trạng thái có thể của bài toán, từ trạng thái ban đầu đến các trạng thái trung gian và cuối cùng, trong đó mỗi trạng thái đại diện cho một cách sắp xếp số lượng nhà truyền giáo và con quỷ ở mỗi bên sông cùng với vị trí của thuyền. . Mục tiêu của chương trình là xây dựng một cây trạng thái đầy đủ để mô tả các bước di chuyển hợp lệ trong bài toán, từ trạng thái khởi đầu đến trạng thái mục tiêu, đồng thời hiển thị cây này dưới dạng đồ họa.

b. Chương trình Solve: Sử dụng thuật toán tìm kiếm (như BFS hoặc DFS) để duyệt qua các trạng thái khả thi và tìm ra con đường từ trạng thái ban đầu đến trạng thái đích. Mục đích là giải bài toán bằng cách tìm dãy chuyển đổi trạng thái hợp lệ từ trạng thái khởi đầu (tất cả đều ở một bờ) sang trạng thái mục tiêu (tất cả đều ở bờ đối diện).

c. Chương trình Main: Sau khi tìm thấy đường đi hợp lệ, chương trình này sẽ xuất kết quả hoặc hiển thị chi tiết các bước từ trạng thái ban đầu đến trạng thái đích. Mục đích để cung cấp một cách dễ hiểu và có thể theo dõi trực quan kết quả của bài toán, giúp người dùng hoặc người học thấy rõ dãy trạng thái và hành động cụ thể.

Generate_Full_Space_Tree

- Trước hết, ta xây dựng các Option – các đường đi có thể đi được: options = [(1, 0), (0, 1), (1, 1), (0, 2), (2, 0)] hay thuyền chở qua sông 1 người, hoặc 1 con quỷ , hoặc 1 người và 1 con quỷ, hoặc 2 người, hoặc 2 con quỷ

- Tiếp theo, Dòng os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz2.38/bin/' được dùng để thiết lập đường dẫn cho Graphviz. Điều này cần thiết vì *pydot* dựa vào Graphviz để tạo và lưu cây trạng thái dưới dạng ảnh. Việc thêm đường dẫn giúp chương trình có thể gọi trực tiếp các công cụ của Graphviz để vẽ đồ thị, đảm bảo quá trình tạo ảnh cây trạng thái diễn ra trơn tru.

- Tiếp tục, ta tạo một đồ thị sử dụng thư viện pydot. Đồ thị này sẽ lưu trữ các nút và cạnh để biểu diễn cây trạng thái, với các thuộc tính là đồ thị không có hướng (nhưng trong bài này các cạnh sẽ được thêm vào với hướng từ Node Cha tới Node con.), màu nền vàng nhạt, gán nhãn fig: Missionaries and Cannibal State Space Tree với cỡ chữ 24.
- Ta sử dụng đối số arg để tạo định nghĩa "depth" để người dùng có thể nhập độ sâu tối đa cho cây. Vì vậy có thể nhập -d hay -depth để giới hạn độ sâu, nếu không cung cấp giá trị, max_depth mặc định sẽ bằng 20.

```
graph = pydot.Dot(graph_type='graph',strict=False, bgcolor="#fff3af", label="fig: Missionaries and Cannibal State Space Tree", fontcolor="red", fontsize="24", overla

# To track node
i = 0

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))
```

Chương trình đã tạo ra 7 hàm phục vụ mục đích khác nhau:

1. Hàm is_valid_move để kiểm tra xem số lượng nhà truyền giáo và con quỷ có thuộc khoảng [0, 3] hay không?

2. Hàm write_image dùng để ghi lại đồ thị cây dưới dạng tệp ảnh PNG với file name là state_space_{max_depth} với max_depth do người dùng nhập hoặc mặc định là 20 nư đã giải thích ở trên. Hàm này sử dụng khối try...except để bắt lỗi, trong trường hợp có lỗi xảy ra khi ghi ảnh (chẳng hạn như đường dẫn không hợp lệ, thiếu quyền ghi, hoặc lỗi do thư viện), khối except sẽ bắt lỗi và gán thông tin lỗi cho biến e, với thông tin chi tiết từ e. Khối try...except đảm bảo rằng chương trình vẫn tiếp tục chạy, không bị dừng đột ngột và cung cấp thông tin lỗi khi không thể ghi ảnh, thay vì dừng đột ngột.

3. Hàm draw_edge dùng để vẽ các cạnh và Node trong cây, trong đó:

    - u đại diện cho Node cha và Node con của cây. Node này được lấy từ từ điển Parent dựa trên trạng thái hiện tại của các nhà truyền giáo, con quỷ, và vị trí thuyền.

    - v là Node đại diện cho trạng thái hiện tại được truyền vào hàm draw_edge. Node này cũng có thể là node gốc (root node) nếu không có cha

- Với trường hợp đặc biệt của node gốc (không có node cha), ta sẽ sử dụng một hàm if else, trong đó, phần if:

    - **IF**: Sử dụng để xử lý các node không phải là node gốc (node có cha). Nếu node hiện tại có một trạng thái cha (node cha không phải None), draw_edge sẽ tạo ra cả u (node cha) và v (node hiện tại). Sau đó, nó thêm cạnh từ u tới v bằng cách sử dụng pydot.Edge(...). Cạnh này biểu diễn mối liên hệ giữa node cha và node hiện tại trong cây trạng thái.

    - **ELSE**: Nếu Parent[...] is None, điều đó có nghĩa đây là node khởi đầu (gốc) của cây trạng thái. Node gốc này không có cha, vì vậy không cần tạo u. Chỉ có v được tạo ra để đại diện cho node khởi đầu, và node này được thêm vào đồ thị dưới dạng một node đơn lẻ, không có cạnh nối với các node khác.

4. Hàm is_start_state sẽ khởi tạo trạng thái ban đầu và trả về giá trị (3, 3, 1).

5. Hàm is_goal_state sẽ khởi tạo trạng thái kết thúc, trạng thái GOAL và trả về giá trị (0, 0, 0).

6. Hàm number_of_cannibals_exceeds sẽ kiểm tra xem trạng thái hiện tại có vi phạm điều kiện hay không (số lượng con quỷ không được vượt quá số lượng nhà truyền giáo ở bất kỳ bờ nào).

7. Hàm generate làm hàm chính và quan trọng. Nó chịu trách nhiệm tạo ra cây, sử dụng hàng đợi (queue) để duyệt qua các trạng thái bằng cách kiểm tra từng bước di chuyển hợp lệ dựa trên các quy tắc của bài toán. Cụ thể:

    - Khởi tạo hàng đợi q với vai trò lưu trữ các trạng thái cần được duyệt trong quá trình duyệt cây, với trạng thái ban đầu là 3 nhà truyền giáo, 3 con quỷ, thuyền ở bên trái, độ sâu 0, số node là 0) được thêm vào q và đánh dấu là node gốc (node này không có cha).

- Vòng lặp while: lấy từ trạng thái từ hàng đợi q ra để xử lý, với mỗi lần lặp, hàm q.popleft() sẽ lấy trạng thái đầu tiên trong hàng đợi.

- Các câu lệnh điều kiện, if, elif, else sẽ tô màu các trạng thái, và Nếu depth_level đạt đến độ sâu tối đa (max_depth) đã chỉ định, cây trạng thái sẽ ngừng mở rộng thêm nhánh mới và kết thúc hàm.

- Vòng lặp for: Duyệt qua các hành động có thể thực hiện được từ trạng thái hiện tại bằng cách sử dụng danh sách options, trong đó:

    - x: số lượng nhà truyền giáo.

    - y: số lượng con quỷ

    - Vòng lặp for sử dụng biến op để đặt hướng di chuyển phụ thuộc vào vị trí của thuyền (-1 nếu thuyền đang ở bên phải, 1 nếu ở bên trái). Sau đó, sử dụng các biến next_m, next_c, next_s để tính số lượng nhà truyền giáo và con quỷ sau khi thực hiện hành động (x, y), cùng vị trí thuyền (next_s). Cuối cùng, nó kiểm tra điều kiện của các trạng thái tiếp theo, nếu node hiện tại không phải là node gốc, và trạng thái tiếp theo không phải là trạng thái cha của node hiện tại, thì tiếp tục kiểm tra tính hợp lệ của trạng thái.

    - is_valid_move(next_m, next_c): Kiểm tra nếu trạng thái tiếp theo có số lượng hợp lệ của nhà truyền giáo và kẻ ăn thịt (không vượt quá 3 và không âm), nếu hợp lệ, trạng thái được thêm vào q, đồng thời Parent ghi lại cha của trạng thái này.
- Cuối cùng, biến can_be_expanded là cờ để kiểm tra xem node hiện tại có sinh ra thêm trạng thái con hợp lệ nào không. Nếu node hiện tại không thể sinh ra trạng thái hợp lệ nào (can_be_expanded == False), điều này có nghĩa là node này là một trạng thái "kết thúc", không thể mở rộng thêm. Sau đó tô màu node đó.

Sau đó khi thực thi chương trình trong CMD, nếu name "__main__" thì nó sẽ kiểm tra hàm generate, nếu hàm này không có lỗi thì nó sẽ trả lại hàm write_image là xuất ra hình ảnh.

Solve

- Ta khởi tạo một lớp Solution, mục tiêu của lớp này là tìm kiếm và hiển thị tất cả các bước di chuyển cần thiết để đưa tất cả các nhà truyền giáo và con quỷ từ bờ bên trái sang bờ bên phải mà không vi phạm các quy tắc.
- Lớp Solution sử dụng 13 hàm, cụ thể:

1. Sử dụng init để khởi tạo tham số self với trạng thái ban đầu, trạng thái mục tiêu, các tùy chọn di chuyển cho thuyền, và cấu hình cho đồ họa (sử dụng pydot để vẽ cây trạng thái) và tham số thăm (visited), và giải (mặc định là False – chưa giải được).

2. Hàm is_valid_move để kiểm tra xem số lượng nhà truyền giáo và con quỷ có thuộc khoảng [0, 3] hay không?

3. Hàm is_goal_state kiểm tra xem trạng thái hiện tại có phải là trạng thái goal (0, 0, 0) hay không?

4. Hàm is_start_state kiểm tra xem trạng thái hiện tại có phải là trạng thái bắt đầu (3, 3, 1) hay không?

5. Hàm number_of_cannibals_exceeds kiểm tra xem số lượng con quỷ có vượt quá số nhà truyền giáo ở bất kỳ bờ nào hay không. (3 – 4 – 5 giống như chương trình 1)

6. Hàm write_image - giống như Chương trình 1 cũng với mục đích hi hình ảnh cây trạng thái vào tệp PNG.

7. Hàm solve: có giá trị mặc định là DFS, tức là, nếu không có tham số nào được truyền vào, hàm sẽ sử dụng phương pháp tìm kiếm theo chiều sâu. Sau đó, hàm khởi tạo các từ điển rỗng, trạng thái khởi đầu, hành động, và Node đều rỗng. Cuối cùng, hàm gọi phương thức dfs nếu tham số solve_method là "dfs" hoặc gọi phương thức bfs nếu là "bfs".Khi gọi hàm dfs, *self.start_state sẽ phân tích tuple (3, 3, 1) thành ba tham số riêng biệt (3, 3, 1) và 0 sẽ được truyền vào như độ sâu ban đầu. Còn nếu không, nếu bfs được chọn thì nó sẽ được gọi.

8. Hàm draw_legend sử dụng để vẽ chú thích cho các yếu tố trong đồ thị. Hàm khởi tạo graphlegend để tạo một nhóm nút (cluster) với tên là "legend".Và các thuộc tính để làm nổi bật phần chú thích. Hàm này tạo ra 7 loại Node khác nhau tương ứng với các vai trò khác nhau

9. Hàm draw được sử dụng để hiển thị trạng thái hiện tại của trò chơi trên console bằng cách sử dụng emoji, với các tham số đầu vào lượng nhà truyền giáo con quỷ ở bên trái và bên phải của bờ sông.

10. Hàm solution_show bắt đầu với việc khởi tạo các danh sách rỗng dùng để lưu trữ các trạng thái, bước di chuyển, và các nút tương ứng. Sau đó, nó sử dụng 1 vòng lặp while thu thập thông tin về chuỗi trạng thái từ trạng thái mục tiêu về trạng thái khởi đầu và lưu trữ nó, cuối cùng, vòng lặp cập nhật state bằng Parent[state] để tiếp tục đi lên chuỗi cha cho đến khi không còn nút cha nào (tức là state trở thành None).Sau đó, hàm sử dụng một vòng lặp for, lặp qua từng bước trong giải pháp, bắt đầu từ bước thứ hai (bỏ qua bước đầu tiên là trạng thái khởi đầu). Tiếp tục, nó sử dụng một điều kiện if để kiểm tra xem Node hiện tại có phải là trạng thái khởi đầu hay không ? Sau đó, nó in ra thông tin về bước di chuyển cụ thể, cho biết số lượng nhà truyền giáo và con quỷ di chuyển từ bên nào sang bên nào và cập nhật lại số lượng. Cuối cùng, nó vẽ lại trạng thái sau mỗi bước.

11. Cũng tương tự như chương trình 1, hàm draw_edge edge dùng để vẽ các cạnh và Node trong cây.

12. Hàm bfs và dfs là hai hàm quan trọng trong chương trình 2 giúp ta giải quyết bài toán. Trước hết, ở hàm bfs:

- Ta tạo một hàng đợi q để lưu trạng thái cần khám phá và thêm trạng thái khởi đầu vào hàng đợi và đánh dấu nó là đã được thăm.

- Tiếp theo, sử dụng một vòng lặp While, khi hàng đợi không rỗng, lấy trạng thái đầu tiên ra và phân tích nó. Vòng lặp này chủ yếu vẽ các cạnh giống như chương trình 1.

- Sử dụng một vòng lặp for với hai biến x, y cũng giống chương trình 1. Tuy vậy, ta tính toán kỹ các tham số next,

  - **next_m:** Tính toán số lượng nhà truyền ở bên còn lại (sau khi di chuyển). Sử dụng op, nếu side là bên phải (1), op sẽ là -1, do đó giảm số lượng nhà truyền giáo; ngược lại, nếu side là bên trái (0), op sẽ là 1, tăng số lượng nhà truyền giáo
  - **next_c:** Tương tự như next_m, tính toán số lượng con quỷ ở bên còn lại.
  - **next_s:** Xác định bên mới của thuyền. int(not side) sẽ đổi giá trị side từ 0 thành 1 hoặc từ 1 thành 0.

- Ta sử dụng điều kiện if để kiểm tra xem trạng thái mới (next_m, next_c, next_s) đã được thăm hay chưa. Nếu chưa, ta có thể tiếp tục xử lý trạng thái này. Điều này giúp tránh việc quay lại những trạng thái đã được khám phá trước đó, tránh lặp vô hạn và giảm thiểu số lượng trạng thái cần kiểm tra.

- Thêm 1 điều kiện if điểm tra xem di chuyển đến trạng thái (next_m, next_c) có hợp lệ hay không.

- Sau đó, ta ghi lại thông tin về trạng thái, di chuyển, node.

13. Hàm dfs cũng tương tự như hàm bfs, tuy vậy, nó không sử dụng vòng lặp while mà thay vào đó draw_edge luôn. DFS thường sử dụng đệ quy, trong đó hàm gọi chính nó để xử lý các trạng thái con. Do đó, DFS không cần vòng lặp while để kiểm soát quá trình tìm kiếm, mà nó xử lý thông qua các cuộc gọi hàm đệ quy. Ngược lại, BFS sẽ khám phá tất cả các nút ở một mức độ trước khi chuyển sang mức độ tiếp theo. Nó sử dụng một hàng đợi (deque) để lưu trữ các nút cần được

xử lý, và điều này yêu cầu một vòng lặp while để tiếp tục cho đến khi hàng đợi trống.

- Hơn nữa, trong vòng lặp for của dfs, vòng lặp này cũng kiểm tra tất cả các tùy chọn di chuyển cho trạng thái hiện tại, nhưng nếu di chuyển đến trạng thái tiếp theo là hợp lệ, nó sẽ không thêm vào hàng đợi mà sẽ gọi đệ quy hàm dfs cho trạng thái mới. Điều này cho phép DFS đi sâu vào các nhánh mà không cần chờ đợi xử lý tất cả các trạng thái ở mức hiện tại như BFS.

Cuối cùng ta tạo biến boolean Solution_found trong hàm DFS (Depth-First Search) được sử dụng để theo dõi xem liệu thuật toán đã tìm thấy giải pháp cho bài toán hay chưa. solution_found được khởi tạo với giá trị False trước khi bắt đầu tìm kiếm. Trong quá trình tìm kiếm, nếu thuật toán tìm thấy trạng thái nào đó là trạng thái mục tiêu (goal state), biến này sẽ được cập nhật thành True. Sau đó, ta gán giá trị của solution_found vào thuộc tính self.solved. Điều này cho phép theo dõi xem bài toán đã được giải quyết hay chưa. Và, ta trả về giá trị của solution_found, cho biết liệu thuật toán DFS đã tìm thấy giải pháp cho bài toán hay chưa. Nếu một giải pháp được tìm thấy trong bất kỳ cuộc gọi đệ quy nào, giá trị này sẽ là True, ngược lại sẽ là False.

Main

1. Chương trình main cho phép ta chọn phương thức giải quyết một bài toán, hiển thị giải pháp và xuất kết quả ra file hình ảnh.

2. Ta tạo hai tham số method và legend cho phép người dùng chỉ định phương thức giải quyết.và cho phép người dùng chọn xem có muốn hiển thị chú thích trên đồ thị hay không.

3. Tiếp tục, ta lấy giá trị của các tham số từ dòng lệnh. Nếu không có tham số nào được cung cấp, solve_method mặc định là "bfs" và legend_flag là False.

   solve_method = args.get("method", "bfs") legend_flag = args.get("legend", False)

4. Ta tạo một hàm main với việc sử dụng lớp solution ở chương trình 2, với hàm if để gọi phương thức solve của đối tượng s với phương thức đã chỉ định. Nếu giải pháp được tìm thấy, hiển thị giải pháp trên console.

5. Tiếp tục ta xử lý tên file xuất ra, output file name, nếu legend_flag được kích hoạt, vẽ chú thích và thêm _legend vào tên file, nếu không thì output chỉ có dạng .png

6. Cuối cùng ta throw 1 exception nếu không tìm được giải pháp.

7. Sau đó sử dụng 1 điều kiện if để ta chạy trên console và trả về hàm main.

Người báo cáo

Nguyễn Thanh Kiên