BÁO CÁO THỰC HÀNH NHẬP MÔN TRÍ TUỆ NHÂN TẠO

Tuần 2

Bài toán. The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from ananalytical viewpoint (Amarel, 1968).

(a) Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.

(b) Implement and solve the problem optimally using an appropriate search algorithm. Is it a good idea to check for repeated states?

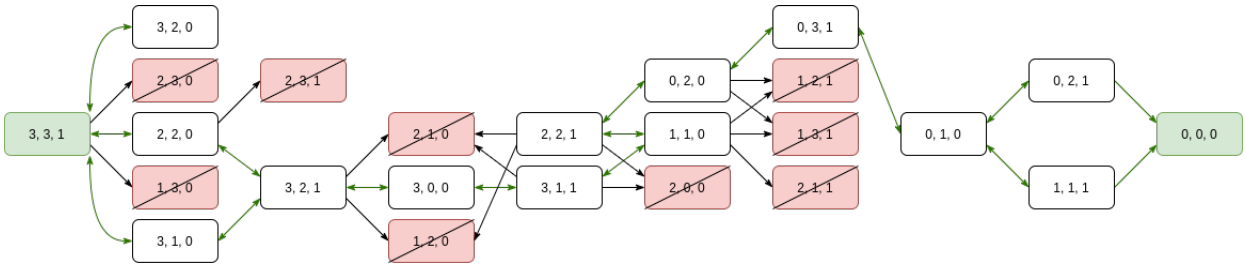
1 Phân tích bài toán

Gọi (a, b, k) với $0 \leq a, b \leq 3$, trong đó a là số người, b là số con quỷ ở bên bờ tả ngạn vào các thời điểm mà thuyền ở bờ này hoặc bờ kia, $k = 1$ nếu thuyền ở bờ tả ngạn và $k = 0$ nếu thuyền ở bờ hữu ngạn. 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 2 người, hoặc 2 con quỷ, hoặc 1 người và 1 con quỷ.
- Trạng thái kết thúc là $(0, 0, 0)$.

2 Sơ đồ trạng thái

Vẽ sơ đồ liệt kê tất cả các khả năng có thể để tìm lời giải của bài toán (search tree).



3 Xây dựng graph

Tạo **class MCGraph** có 3 thành phần là trạng thái vào lúc xét bên bờ tả ngạn (**state**), số bước đã thực hiện tính đến thời điểm hiện tại (**num_of_moves**) và trạng thái cha của trạng thái hiện tại (**parent**) với các phương thức là **is_goal_state()**, **is_valid()**, **is_killed()**, **generate_child()** và **find_solution_dfs()**.

```
4   class MCGraph:
5       def __init__(self, state = (3, 3, 1), num_of_moves = 0, parent = None):
6           self.state = state
7           self.num_of_moves = num_of_moves
8           self.parent = parent
```

Phương thức **is_valid()** kiểm tra một trạng thái được xét có hợp lệ hay không. Một trạng thái được gọi là hợp lệ khi trong trạng thái đó, số lượng người và số lượng quỷ nằm trong đoạn $[0, 3]$ và chỉ số chỉ vị trí chiếc thuyền chỉ nhận giá trị 0 hoặc 1.

```
16      def is_valid(self):
17          missionaries = self.state[0]
18          cannibals = self.state[1]
19          boat = self.state[2]
20          if missionaries < 0 or missionaries > 3:
21              return False
22          if cannibals < 0 or cannibals > 3:
23              return False
24          if boat < 0 or boat > 1:
25              return False
26          return True
```

Phương thức **is_killed()** kiểm tra một trạng thái được xét có bị loại hay không. Một trạng thái được gọi là bị loại khi số lượng con quỷ lớn hơn số lượng người ở cả hai bên bờ.

```
28      def is_killed(self):
29          missionaries = self.state[0]
30          cannibals = self.state[1]
31          if missionaries > 0 and missionaries < cannibals:
32              return True
33          # Check for the other side
34          if missionaries > cannibals and missionaries < 3:
35              return True
36          return False
```

Phương thức **generate_child()** thực hiện tạo các trạng thái con tiếp theo của trạng thái đang xét. Tại mỗi trạng thái có tối đa 5 trạng thái con ứng với việc thêm hoặc bớt số lượng quỷ và người ứng với số lượng người và quỷ có thể có trên chiếc thuyền: $(1, 0), (0, 1), (2, 0), (0, 2), (1, 1)$. Mỗi trạng thái sẽ thêm số lượng người và quỷ nếu chỉ số vị trí chiếc thuyền là 0, ngược lại sẽ giảm.

```
38      def generate_child(self):
39          children = []
40          op = -1 # Subtract
41          if self.state[2] == 0:
42              op = 1 # Add
43          miss, cans, boat_side = self.state
44          for mis in range(3):
45              for can in range(3):
46                  new_state_vars = (miss + op*mis, cans + op*can, boat_side + op)
47                  new_state = MCGraph(new_state_vars, self.num_of_moves + 1, self)
48                  if mis + can >= 1 and mis + can <= 2 and new_state.is_valid():
49                      children.append(new_state)
50          return children
```

Phương thức **find_solution_dfs()** thực hiện việc tìm lời giải tối ưu nhất cho bài toán trên bằng thuật toán tìm kiếm **Depth First Search**.

4 Sử dụng thuật toán Depth First Search để tìm lời giải tối ưu

4.1 Ý tưởng thuật toán DFS

Từ trạng thái gốc ban đầu, xác định và duyệt qua các trạng thái kề xung quanh trạng thái gốc vừa xét. Tiếp tục quá trình duyệt qua các trạng thái kề trạng thái vừa xét cho đến khi đạt được kết quả cần tìm hoặc duyệt qua tất cả các trạng thái. Tại mỗi bước chọn trạng thái để phát triển (trạng thái được sinh ra trước các trạng thái chờ phát triển khác), thêm trạng thái được chọn vào danh sách đã duyệt để đánh dấu. Danh sách này được xử lý như hàng đợi **Stack**.

4.2 Khởi tạo

- Một ngăn xếp **frontier**,
- List **explored** để kiểm soát các trạng thái đã duyệt qua để tránh bị trùng lắp. Tuy nhiên, vẫn sẽ có trường hợp trùng trên **explored**, nhưng sẽ có ảnh hưởng không nhiều đến kết quả,
- List **solutions** chứa các kết quả đường đi tối ưu được tìm thấy bằng thuật toán DFS.

4.3 Các bước thực hiện

- Bước 1: Thêm node gốc $(3, 3, 1)$ vào ngăn xếp **frontier** để chờ được xét.
- Bước 2: Kiểm tra tập **frontier** có rỗng không?

  - Nếu tập **frontier** không rỗng, lấy một node ra khỏi tập **frontier** làm node đang xét **current_node** và chuyển sang bước 3.
  - Nếu tập **frontier** rỗng, thoát khỏi vòng lặp và thông báo không tìm được lời giải.

- Bước 3: Tạo các node con của node đang xét và lưu vào **childs**, sau đó duyệt từng node con vừa tạo. Nếu node con đang xét có trạng thái **state** chưa xuất hiện trong **explored**, ta kiểm tra xem node đang xét có trạng thái có bị loại hay không, nếu nó là trạng thái bị loại, ta xét tiếp node con tiếp theo, ngược lại ta chuyển sang bước 4.

- Bước 4: Ta tiếp tục kiểm tra node con đang xét có trạng thái **state** có phải trạng thái đích $(0, 0, 0)$ hay không? Nếu nó là trạng thái đích, ta thêm node con đó vào **solutions**. Sau đó, ta thêm **child** vào **frontier** và **child_state** vào **explored** để đánh dấu node đã duyệt qua.

- Bước 5: Hàm tìm kiếm lời giải cho bài toán này trả về **solutions** chứa tất cả các đường đi tối ưu nhất.

4.4 Cài đặt hàm tìm kiếm lời giải theo thuật toán DFS

```
52      def find_solution_dfs(self):
53          root = self
54          frontier = LifoQueue()
55          frontier.put(root)
56          explored = []
57          solutions = []
58          while True:
59              if frontier.empty():
60                  break
61              current_state = frontier.get()
62              childs = current_state.generate_child()
63              for child in childs[::-1]:
64                  child_state = child.state
65                  if child_state not in explored:
66                      if child.is_killed():
67                          continue
68                      elif child.is_goal_state():
69                          solutions.append(child)
70                      frontier.put(child)
71                      explored.append(child_state)
72          return solutions
```

4.5 Kết quả

```
(0, 0, 0) <-- (0, 2, 1) <-- (0, 1, 0) <-- (0, 3, 1) <-- (0, 2, 0) <-- (2, 2, 1) <-- (1, 1, 0) <-- (3, 1, 1) <-- (3, 0, 0) <-- (3, 2, 1) <-- (3, 1, 0) <-- (3, 3, 1)
```

4.6 Nhận xét

Đây là một ý tưởng tốt để kiểm tra các trạng thái lặp đi lặp lại. Khi duyệt qua các trạng thái con của trạng thái đang xét, có nhiều con dẫn đến các trạng thái mà trước đó đã duyệt qua. Nếu một thuật toán không kiểm tra các trạng thái lặp đi lặp lại này, nó có thể dễ dàng bị mắc kẹt trong vòng lặp vô hạn.