CS76 - Missionaries and Cannibals

Himadri Narasimhamurthy - 19W - 1/10/19

Introduction

This is a basic implementation of the missionaries and cannibals AI problem. The way it is implemented comes with two different sets of rules, which you can choose by inputting a boolean in your search.

1. The first rule set is that the boat is a safe zone, which means that if a cannibal is within the boat on a side of the river - he cannot eat any of missionaries - he is alone in the boat. This allows for drop offs in the boat rather than everyone having to get out and risk being eaten.

2. The second is the traditional rule where if the boat is one one side of the river, all the people count in the figure of whether they can be eaten or not. There is no safety for have just come across on the boat.

Initial Discussion and Introduction

First, we deal with the state of the program. In order for our search algorithms to search the proper graphs, we need to come up with the upper bound for the number of states and then a way to generate states.

The state: We will be keeping track of the state in a tuple of (M, C, B) where M is number of missionaries on the left side, C is number of cannibals on the left side, and B is the position of the boat (1 - left, 0 - right). Our start is (3,3,1) and goal is (0,0,0)

```
Total options for states:
    4 (3,2,1,0 missionaries) * 4 (3,2,1,0 missionaries) * 2 (left or right) = 32 tota
l states
    This (32) is the upper bound since some of these states are impossible
        - more cannibals than missionaries on a side
```

The State Diagram

- see the project file for the state diagram (sd.png)

In the state diagram - we can see the states three steps into the problem, starting from 3,3,1.

Code Design

Since much of the structure was provided, I followed the implementation suggestions which were in the comments of that code. Before doing so - I added some small tests to fully understand how the SearchSolution and CannibalProblem classes function.

Here is what I added to test the get_successors method I would later write:

```
    test_cp = CannibalProblem((5, 5, 1))
    print(test_cp.get_successors((2, 3, 0), True))
    print(test_cp.get_successors((4, 1, 0), True))
    print test_cp
```

Building the model

The majority of this section can be found in CannibalProblem.py where I wrote the get*opp(state), get*successors(state), methods to check if states were valid, dependent on the rule selected, and a simple goal_test method.

Get_Opp method this method gets the opposite sides state in order to check the validity of both sides - I use this and it"s two conditions to implement the two rules.

Get_Successors method this method uses a brute force tactic of coming up with the states that are possible, and then valid, from any given state. Depending on the side of the boat, we either add/subtract one or two missionaries or cannibals from the state.

sb_safe this method checks validity of the states with the rule that the boat is a safe zone or does not count as a part of the total missionaries and cannibals on the shore.

strictrules_safe this method is the traditional ruled validity checking method.

goal_test this method is a simple method which checks if we are at the goal vertex.

BFS

Below is the beginning of the code I used to implement BFS in my searches. I used a dictionary to keep tracks of the backpointers from nodes and a dequeue to pop off the earliest added nodes. Then, we finally went through the dictionary to get the path to the goal, which we popped off the deque.

```
def bfs_search(search_problem, safe_boat):
    start = search_problem.start_state

    back_dict = {}
    back_dict[start] = None

    q = deque()
    q.append(start)
    .
    .
    .
```

Memoizing DFS

Does memoizing dfs save significant memory with respect to breadth-first search? Why or why not?

DFS without memoizing, especially in a tree can save significant memory however with memoizing DFS, we potentially have to save the entire frontier as visited which means that it does not save significant memory compared to BFS. Both search algorithms would have to save all the "visited" nodes in the worst case scenario - with path checking, we save memory but maybe compromise slightly on time since we need to recheck the path each time.

Recursize Path Checking DFS

Does path-checking depth-first search save significant memory with respect to breadth-first search? Draw an example of a graph where path-checking dfs takes much more run-time than breadth-first search; include in your report and discuss.

Path checking DFS does save significant space in most cases of trees but we risk the chance of being stuck in loops in non-directed graphs. It does save space even if it takes more time because we don't have to save all of the visited nodes as we do in BFS, we only need to know the path.

- See the project folder for the drawing (dfs.png)

We implemented backchaining for DFS (see here) as well in order to save even more space in terms of keeping track of the path.

```
def get_path(n):
backpath = []
backpath.append(n.state)
while n.parent != None:
    backpath.append(n.parent.state)
    n = n.parent
return backpath
```

Iterative Deepening Search

On a graph, would it make sense to use path-checking dfs, or would you prefer memoizing dfs in your iterative deepening search?

On a graph, we should use path checking dfs. With the large memory space needed to keep track of the frontier, we should be checking the path instead of keeping track of all visited. Frontier could be very large on a graph.

Here is the call to dfs which we run every time for the amount of times that our loop runs, or our depth_limit.

```
    for depth in range(depth_limit):
    solution = dfs_search(search_problem, depth)
```

Lossy Missionaries and Cannibals

Design a problem where no more than E missionaries could be eaten, where E is some constant.

The State: (Number of Missionaries, Numbers of Cannibals, Number of Missionaries Eaten, Boat Side)

I think that we just need to change the method where we generate the model of states through successors since we can now have a variable number of total missionaries and cannibals. This means that our total number of states is equal to **the number without this condition * E+1**. We would have to change the functions to get*successors, get*opposite, and to check validity of the states.