Program 1 Report: Missionaries and Cannibals

Preston Wipf and James O'Neal

I. Methodology

I wanted to include a disclaimer here regarding my project. Initially, I was told by another member of the class that we could work together in a team of 3 (unsafely assuming this had come with instructor permission). I was notified by my former team on Monday that we could not work as a 3 man cell, and I was the odd man out. Tuesday I found a person I found out there was another person in a similar situation and thus this project attempt, is a bit of a last minute scramble to form a new team and complete the project from scratch so as to not violate the integrity of our former teams. In order to do so, we completely changed methodologies to help distinguish our approach / code from theirs, but ran into a lot of problems in doing so. Those methodologies will be fleshed out in the following subsections.

A. General Approach

Our general approach to this problem wanted to move away from the knowledge base we were working from on our initial teams which were a class-based systems where each instance of the class represented the state of the left bank, right bank and the boat. The differentiation was an attempt to create the state as a simple matrix, which involved not only managing the matrices across the program, but also associating matrix to matrix relationships such as parent matrices. The result is a bit hacky.

B. States

As I mentioned the states are represented by a 2x3 matrix in which each row represents a bank and each column is cannibals, missionaries and boat location (similar to how it is seen in the start and goal files). The implementation was a pandas dataframe which made some operations really convenient (referring to matrix indexes by row and column name), and others extremely difficult (adding to a priority queue which requires a "truth" value which is ambiguous for a dataframe).

One other point to mention is that the explored states are stored in a dictionary or hash table which is keyed on a string representing the state. So given a matrix [[0,0,0], [3,3,1]], the key would be "000331" as this uniquely identifies the state.

C. Successors

The purpose behind the approach to states above is mainly for the computation of successors. With the state represented as a matrix, the five possible moves for each state can be created by simple matrix addition. So a state of [[0,0,0], [1,3,3]] could be added to [[0,2,1], [0,-2,-1]] to generate and action of moving the boat from one bank to the other with 2 cannibals in it. Further, each direction could be changed by simply multiplying the matrix by a constant factor of -1.

Successors which pass the validation check are added to a queue which represents the fringe or frontier. In this implementation queue is a parameter to the *find_successors()* function as the queue type is dependent on the algorithm being executed. Last point of successors is a quick note that a successor is associated to its parent state by giving it the key or hash of its parent state.

D. Validation

As just mentioned, there are two main pieces of validation in the software: validation of successors (ie is the proposed state legitimate) and validation of a goal state. In the former case, it is a simple check to ensure there are no negative values and that no missionaries will be eaten by cannibals. Note that there is no validation for the members in the boat as this logic is handled by the five possible actions listed in the problem description. While the latter state compares the number of missionaries, number of cannibals and boat locations in a given state against the goal state.

E. Algorithm Parameters

Lastly, is the parameters. In this case each search algorithm takes only two parameters, the start state and the goal state. The start state is put into the fringe queue and the goal state is validated against on each iteration. For the queue of each algorithm, it is initialized within the algorithm and its type is dependent on which algorithm is executing it. For example, bfs uses a FIFO queue, dfs and iddfs use LIFO queues and the A* algorithm uses a Priority queue.

The A* algorithm also makes use of the *hueristic()* function which takes a potential successor and the goal state as inputs and calculates the hueristic of the move to the successor state as the sum of the difference between the goal state's left bank and the successors state's left bank divided by the size of the boat (2). In other words, it is the number of people yet to make it to the destination bank divided by the number of people who can go in a single trip. This heuristic was accepted because ofdue to its admissibility. Because in each instance the boat requires at least one person to bring it back to the opposing bank, the rate at which people are moved between banks is one per trip.

II. Results

Algorithm	Test 1	Test 2	Test 3
BFS	(12,28)	(34,184)	(378,8974)
DFS	(12,16)	(48,94)	(1798,4599)
IDDFS	(12,99)	(48,3590)	(???, ∞)
A*	(12,15)	(34,112)	(378,3851)

Key: (*solution_depth, num_nodes_expanded*)

III. Discussion

This is a difficult section to reconcile given the state of the program. Going in, the assumptions were that A* would be the superior algorithm and that we would see relatively poor, but similar performance from DFS and BFS. Given that IDDFS has a space complexity of $O(b\hat{d})$ and that it executes each layer *m - (d - 1)* times where m is the max depth and d is the depth of the layer, we expected it to use more space than the other algorithms. It was also expected for DFS to use less space than both BFS and IDDFS, but we also expected to find that DFS would demonstrate the lack of completeness of BFS, IDDFS and A*.

One problem that has not been resolved as of this report is an intermittent issue that results in an infinite loop or truncated solution path in the larger trees. We have yet to isolate the root of the problem, but it is suspected that the problem lies somewhere in the hacky class-less state association.

IV. Conclusion

Our conclusion here is dependent on not only the results we have seen from previous work, but also our previous understanding of the tradeoffs of the different algorithms. That said, given an admissible, well-thought out heuristic, A* search is the best search algorithm. It performs the best in terms of the number of nodes expanded, and returns a solution depth that is on par or better than the rest of the algorithms. The results returned from DFS were, as expected, lacking both in optimality and completeness while IDDFS returned the optimal path, but in an exponential time and space complexity that far exceeded the others. I don't think there is anything surprising here from the conclusions. The results more reinforce previous understanding of the algorithms than present any sort of novel view of their performance.