Report

COSC276, Fall2021, PA1, Xuedan Zou

A. Description

Three kinds of algorithms are implemented here for this Fox Problem: BFS(breadth-first search), DFS(path-checking depth-first search) and IDS(iterative deepening search).

To begin with, we use tuple (a, b, c) to describe there are a chickens, b foxes and c boats on the side. A FoxProblem class is designed to store the necessary constents of our defined problem, to give out any successor legal states of any given state and
finally to check if our goal of this problem is reached. This class is extremely useful
as it is the key to generate the search graph of our given problem. The node in this graph wraps state object generated by successor funtion in FoxProblem class, and also contains a pointer to the partent node. These nodes are generated during the searching process, also means that we don't need to generate the whole graph in the begining and store it in the memory! If the goal state were found, then the path should be provided. We use a backchaining function to do so. This function calls its parent recursively until finding the start node and so the solution has been found.

BFS method traverses the whole search graph generated by FoxProblem class and expands shallowest unexpanded nodes. Note that here the BFS runs on the graph and has to remember all visited nodes to avoid repeat visit. The **set** data structure provided by python helps us to remmeber every states BFS has visited. BFS method is realized by a while structure, saving those unexpanded nodes by the **deque** data structure which works as a queue and is usually more effective than a linked list structure. Our graph based BFS method is complete and can always find the optimal solution(if existed) to this problem.

DFS method, compared with BFS, traverses the search graph and always expanding the deepest unexpanded nodes. Since this DFS works on a graph and is path-checking, it tracks only of states on the currently explored path and makes sure none of these states will be visited agin to ensure no loop on the current path. In order to be memory efficient, DFS uses a recursive way named as path_checking method to check if the current state was visited before. Overall, DFS method is also implemented by a recursive strucutre, it passes the current depth,node and solution to each sub DFS function. DFS method is complete since its path-checking, but not always optimal since once it finds any solution it will stop regardless of there may be other shallower solution.

Finally, IDS method modifies DFS method by limiting the depth-limit during the search process. It increases the search of depth from 0 to the pre-set depth-limit. Just simply include the DFS method in a while structure and gradually add the depth-limit value of DFS. If only solution was found then IDS would stop and that is the optimal solution.

B. Evaluation

All of these three algorithms work fine to this problem and the given test cases. Note that BFS can not always find the optimal solution. Take the test beginning statement (6, 3, 1) for example, DFS will spend much more time on that and finding out a solution with a length of 26. IDS and BFS are able to find the optimal solution with a lengh of only 16 and BFS is the fastest. Genearlly, for most cases BFS runs faster than DFS and IDS is the slowest with apparent longer time.

Note that here I implement my IDS based on the previous path-checking DFS. It may perform diffrent based on memoizing DFS.

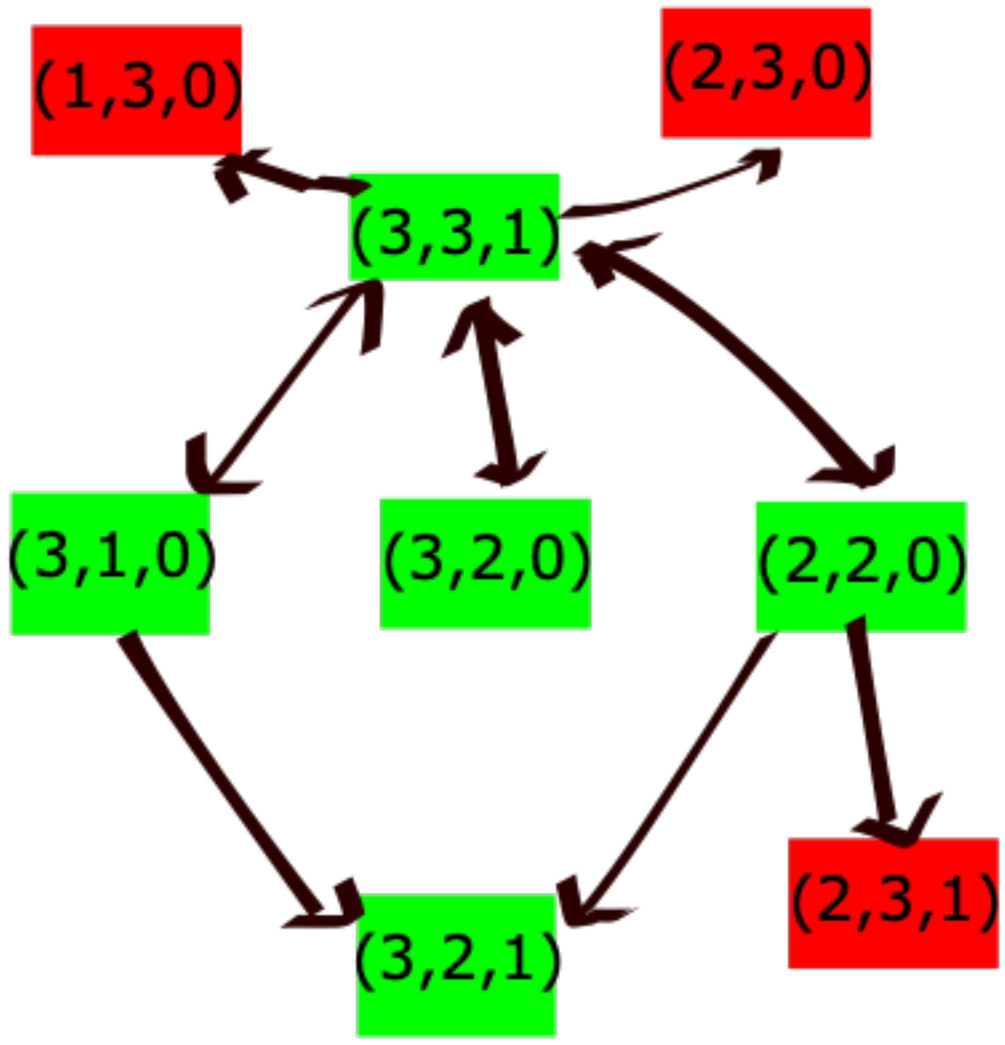
C. Discussion

1. States

Consider there are *a chickens, b foxes and 1 boat*, then upper bound on the number of states can be (a+1)*(b+1)*2 without considering the legality. Obviously the number of chickens changes from 0 to a with a+1 possiblities. Similarly there are b+1 possiblities for the number of foxes and 2 possiblites for the boat status. According to multiplication principle the upper bound should be the products of them and is (a+1)*(b+1)*2.

There are some not legal states like (a, b, 0) when all chickens and foxes are on the current side but there is no boat, since the boat can not row itself it is impossible. Other not legal states are when there are more foxes than chickens on either side, then those foxes will eat chickens.

Now let's look at the example of the original problem: 3 chickens, 3 foxes and 1 boat. Part of the search graph is shown in the below image (green box means legal state and red box means not legal state) :



In this case the initial state is expressed as (3, 3, 1). The numbers in this triple mean the number of chickens, foxes and boats seperately. Then 1 chicken, 2 chickens, 1 fox, 2 foxes or 1 chicken together with 1 fox can be sent to the opposite land by boat. This leads to the possible successors be (1, 3, 0), (2, 3, 0), (3, 1, 0), (3, 2, 0), (2, 2, 0). Notice that in the cases of (1, 3, 0) and (2, 3, 0) there are more foxes than the chicken so these states are not legal. We should also check the number of foxes and chickens in the opposite land. This can be easily done by minus the current number of specific animal with the orgirnal one. Thoses states with more foxes than the existed chickens (but no chicken will be legal) are not legal either. Luckily here all of the states (3, 1, 0), (3, 2, 0) and (2, 2, 0) are legal. Finally, for each leagl states, we continue get its successors and the result is shown in the above image. Note that all of these three states can be returned to the original state (3, 3, 1) but in graph search the repeat state can not be visited so (3, 3, 1) will not be added to the waited expanded states in the next term.

2. Breadth-first search

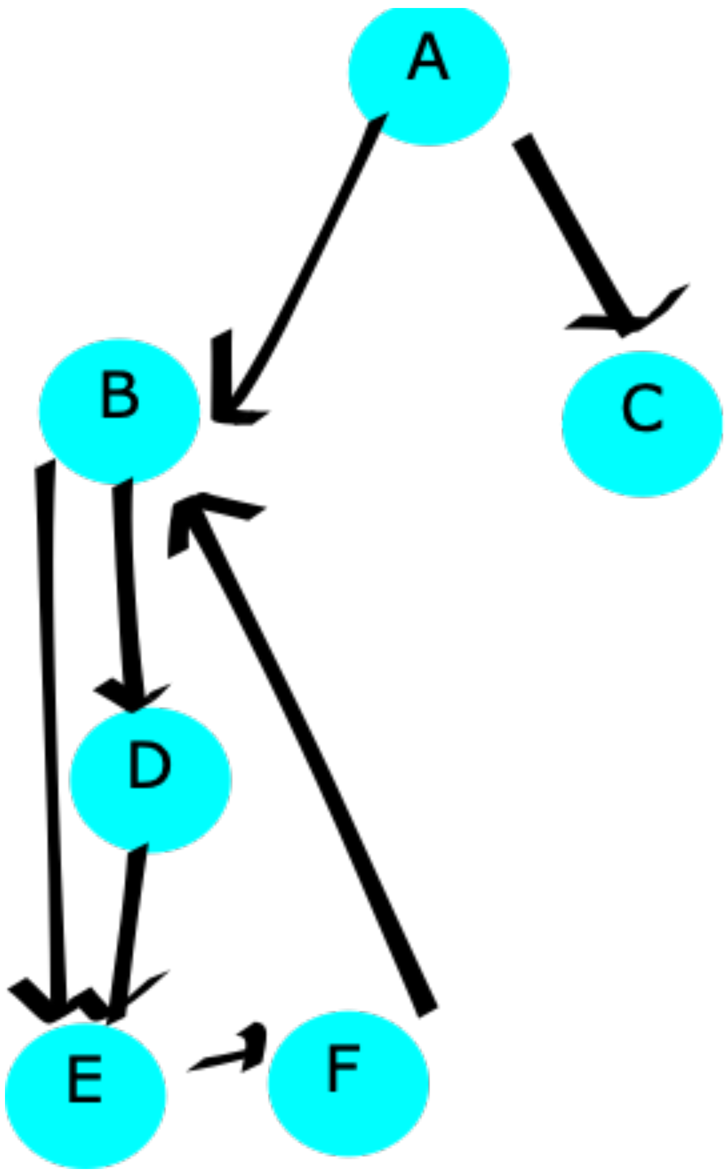
Below is an important code part in the BFS method——the backchaining method. It uses a recursive call structure to get the path from the start state to the current given state.

```
# if the goal is found then give out the path
    def backchaining(self, pass_list=[]):
        list = pass_list
        list.append(self.state)
        if(self.parent != None):
            result_list = self.parent.backchaining(list)
            return result_list
        else:
            return list
```

Using BFS method to our (3, 3, 1) start case discussed before, the best solution includes 12 steps: (3, 3, 1)->(3, 1, 0)->(3, 2, 1)->(3, 0, 0)->(3, 1, 1)->(1, 1, 0)->(2, 2 ,1)->(0, 2, 0)->(0, 3, 1)->(0, 1, 0)->(1, 1, 1)->(0, 0, 0). The total number of nodes been visited in this process is 16.

3. Path-checking depth-fisrst search

The memory saved by path-checking depth-first search depends on the ending depth the DFS algorithm reaches. In most general cases since DFS uses recursive call method to get the path while BFS uses *set* to do so it will take much more memory for BFS. However, if DFS reache too deep while the optimal result is short then DFS may not save that much memory.



In the above image, our goal is to reach node C from node A. Apprently if we use DFS here it will first get deep into the left part and spend much more time than BFS.(Here BFS will only explore A, B, C and find the solution A -> C).

It also depends if memorizing DFS save significant memory with respect to BFS. Generally BFS saves all of the visited nodes before finding the optimal solution and takes a lot of memory. Compared, general path-checking DFS only has to keep track of the chain nodes from top to bottom so it will save significant memory. However, if DFS does not eliminate those redundant paths and saves all nodes that have been visited in data structure like *set* then it depends on the size of the specific search graph and may spend even more memory than BFS.

4. Iterative deepening search

To better discuss the time and space memory aspects between path-checking DFS and memoizing DFS, we use *b* to refer to the maximum branching factor of the search graph, *d* to refer to the depth of the shallowest solution, *m* to refer to the maximum depth of the state space. The time of memoizing DFS should be *O(b^m)* while of path-checking DFS should be *O(mb^m))*. Generally memoizing DFS will be faster. However, the memory of memoizing DFS is *O(b^m)* while of path-checking DFS is *O(m)*, path-checking DFS surely save much more memory in most of the time. As a result, if memory is important then path-checking DFS will be a big plus, otherwise memoizing DFS will be much faster. However, BFS only uses the time *O(b^d)* and the memory *O(b^d)*. Note that here according to our defination m can not smaller than d, BFS will have the best performance in this kind of graph search problems and there is no need to use both kinds of DFS. If the memory does very important then consider path-checking DFS.

5. Lossy chickens and foxes

If at most no more than E chickens could be eaten, the *get_successors* and *checksafe* funciton in *FoxProblem* class should be changed accordingly. We should add a new constent like *total_eaten* to save E, and for each state we should save the total eaten chickens since the begining. When each time calling *checksafe* check if added the possible eaten chickens this time to the previous total eaten chickens is still samller than *total_eaten*. If so, then that state can be legal.
The upper bound on the number of possible states regaredless of the legality here is
the same as the original one: (a+1)*(b+1)*2 where a, b respects to the original number of chickens and foxes respectively.