COSC 76 Project 1: Missionaries and Cannibals

Seungjae Jason Lee

Introduction

Objective

Objective of the project is to use different search methods (bfs, dfs ids) to solve the Missionaries and Cannibals problem. There are two big parts in this project to achieve the objective: implementing the problem and implementing different search methods.

Game Rule

There are M missionaries, N missionaries and a boat on the left side of the river. Missionaries can be killed by cannibals whenever there are more cannibals than missionaries on any sides of the river. As for the boat, it can carry at most 2 people and must carry at least one person. With these rules, the objective of the game is move everyone to the other side of the river with the boat without losing any missionaries.

Initial Discussion

Whenever we use search methods, it is recommended that we find the upper bound to estimate the complexity of the search. For our problem, we can first describe our states as tuples with three integers, which indicate numbers of missionaries, cannibals, and boat on the left side of the river. Assuming that there are M missionaries, N cannibals and a single boat, we know that there are M+1, N+1, 2 possible numbers. Thus, regardless of legality of states, the upper bound for the number of states is M+1 * N+1 * 2. Thus, for our simple example of (3,3,1), it would be 32.

Code Design

As mentioned before in the objectives, the design consists of two parts: the problem and search methods.

The first part is implemented in CannibalProblem.py. CannibalProblem classifies what the problem is and how states and actions are defined for the problem. Also, it has other functions required to build a tree, which is explained under **Building the Model**.

The second part is implemented in uninformed_search.py, which has three different search methods

implemented. Details explained below.

Building the Model

These are the methods of CannibalProblem class that describes states of the game to build a tree.

- '**init**'(self, start_state=(3, 3, 1)): method called when creating object. Sets the start and goal state of the problem.
- get_successors(self, state): returns a list of legal successors of the given state
- build-successors(self, state, boat-cap, boat-location): helper function for get_successors. Returns successors for given information(state, number of people in the boat and the location of the boat).
- legal-state(self, state): checks if a state is legal. Checks if there are more missionaries than cannibals or no missionaries on both sides of the river.
- goal_test(self, state): checks if the given state is the goal test. Returns True or False.

Breath First Search

First, I made SearchNode class in uninformed_search.py, which stores a state of a problem and its parent.

Then, I made bfs-search function, which takes search_problem object as an input. The function returns the shortest path(list) of states for the given problem if it exists. If not, returns an empty list.

I used deque to implement queue and set to keep track of visited nodes in O(1) complexity. As for the backchaining, I used recursion to find the parents of nodes starting from the goal state to the root state.

```
def backchaining(node):
    if node.get_parent() == None:
        array = [node.get_state()]
        return array
    return backchaining(node.get_parent()) + [node.get_state()]
```

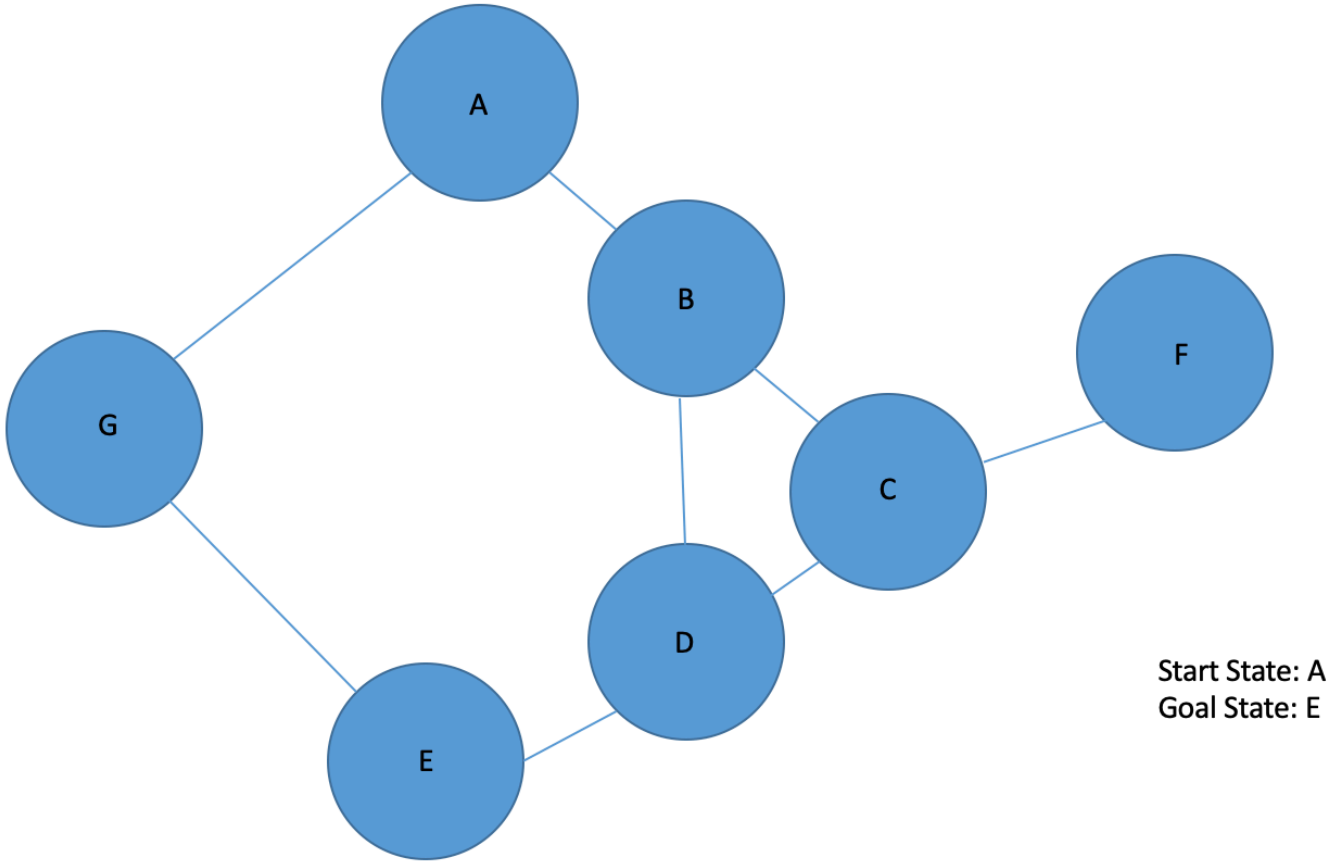
Memoizing depth-first search

Discussion: Memoizing dfs is memory efficient compared to bfs. For both this dfs and bfs, to keep track of the visited nodes, we use O(n) memory where n is number of states. The only difference would be how we find the path first and not make it necessarily more efficient memory-wise.

Path-Checking depth-first search

For the implementation of path-checking dfs, I used recursion to go deeper into a tree. To keep track of the path, I used SearchSolution.py, which helps the algorithm avoid infinite loop by selecting visited node. As the algorithm traverses through the tree and finds the goal state, it returns the path that is stored in SearchSolution object. If it fails to find a solution within given depth limit, it returns empty list.

Discussion: Path checking dfs may be faster than bfs depending on where the goal is relative to the depth of the tree. If the goal is not deep as relative to the tree, bfs may be faster than dfs as it checks less states, which is also true for graphs. For example, let us observe the example graph shown below.



In the following picture, if dfs decides to go for B after A(start state), it will take at least 4 states to find E, which is the goal state. (A - B - D - E). However, for bfs, it will find the shortest path, which is (A-G-E) as it checks each depth.

Iterative deepening search

For IDS, I used for loops and called dfs with different depth limit input(1 to user assigned depth_limit) until either the algorithm finds a path or depth limit is reached. I tried bfs, dfs, and ids on (5,4,1) problem with boat

capacity of 3 and checked that bfs and ids does offer shortest path. The picture of result is shown below.

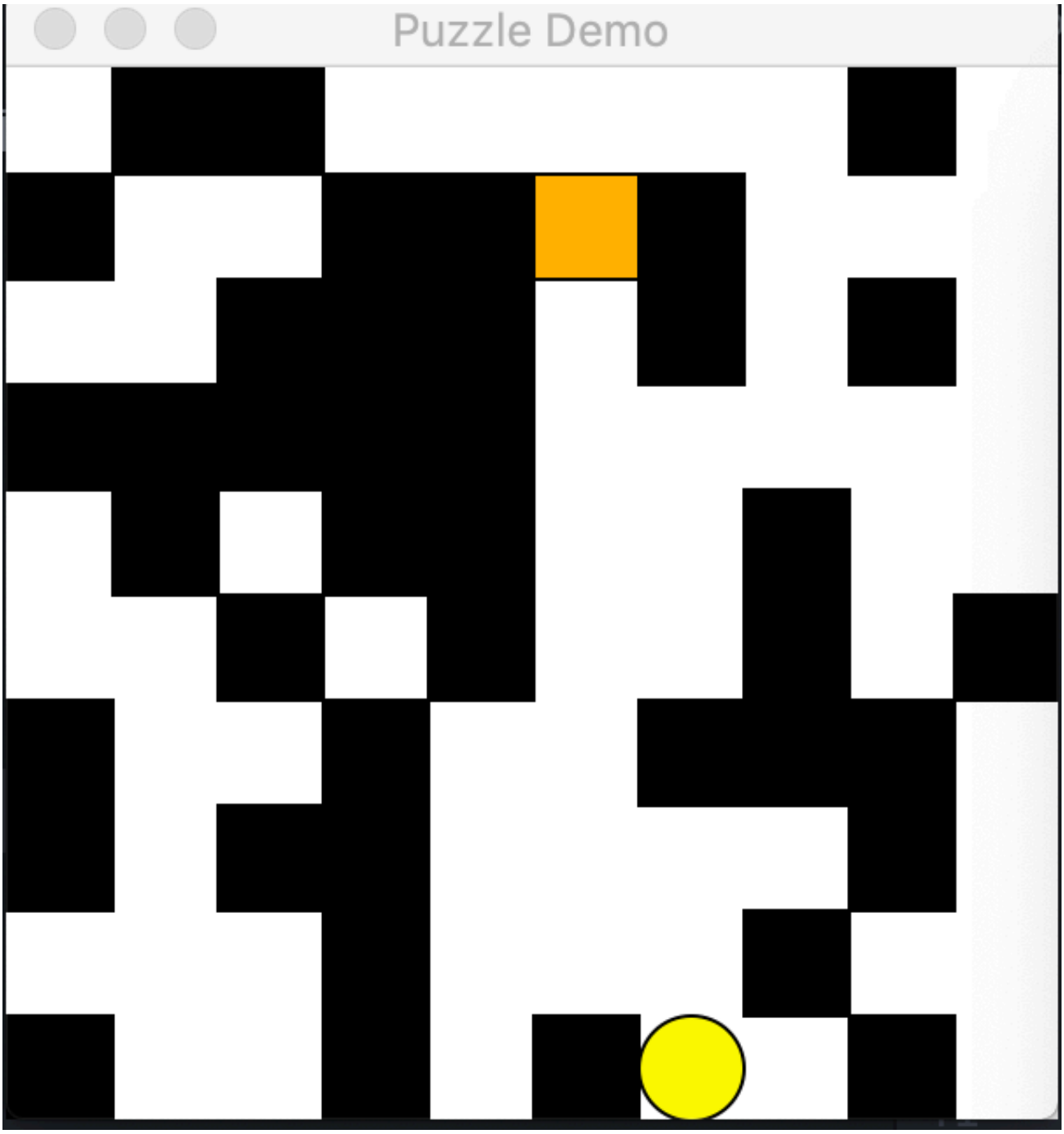
```
def ids_search(search_problem, depth_limit=100):
    for i in range(depth_limit):
        search_result = dfs_search(search_problem, depth_limit = i)
        if search_result != []:
            return search_result
    return []
```

```
Seungjae-Macbook-Pro:provided SJLEE$ python3 cannibals.py
bfs:  [(5, 4, 1), (5, 1, 0), (5, 2, 1), (2, 2, 0), (3, 2, 1), (1, 1, 0), (2, 1, 1), (0, 0, 0)]
dfs:  [(5, 4, 1), (5, 2, 0), (5, 3, 1), (4, 3, 0), (4, 4, 1), (3, 3, 0), (4, 3, 1), (3, 2, 0), (3, 3, 1), (2, 2, 0), (3, 2, 1), (2, 1, 0),
 (2, 2, 1), (1, 1, 0), (2, 1, 1), (1, 0, 0), (1, 1, 1), (0, 1, 0), (0, 2, 1), (0, 0, 0)]
ids:  [(5, 4, 1), (5, 1, 0), (5, 2, 1), (2, 2, 0), (3, 2, 1), (1, 1, 0), (2, 1, 1), (0, 0, 0)]
```

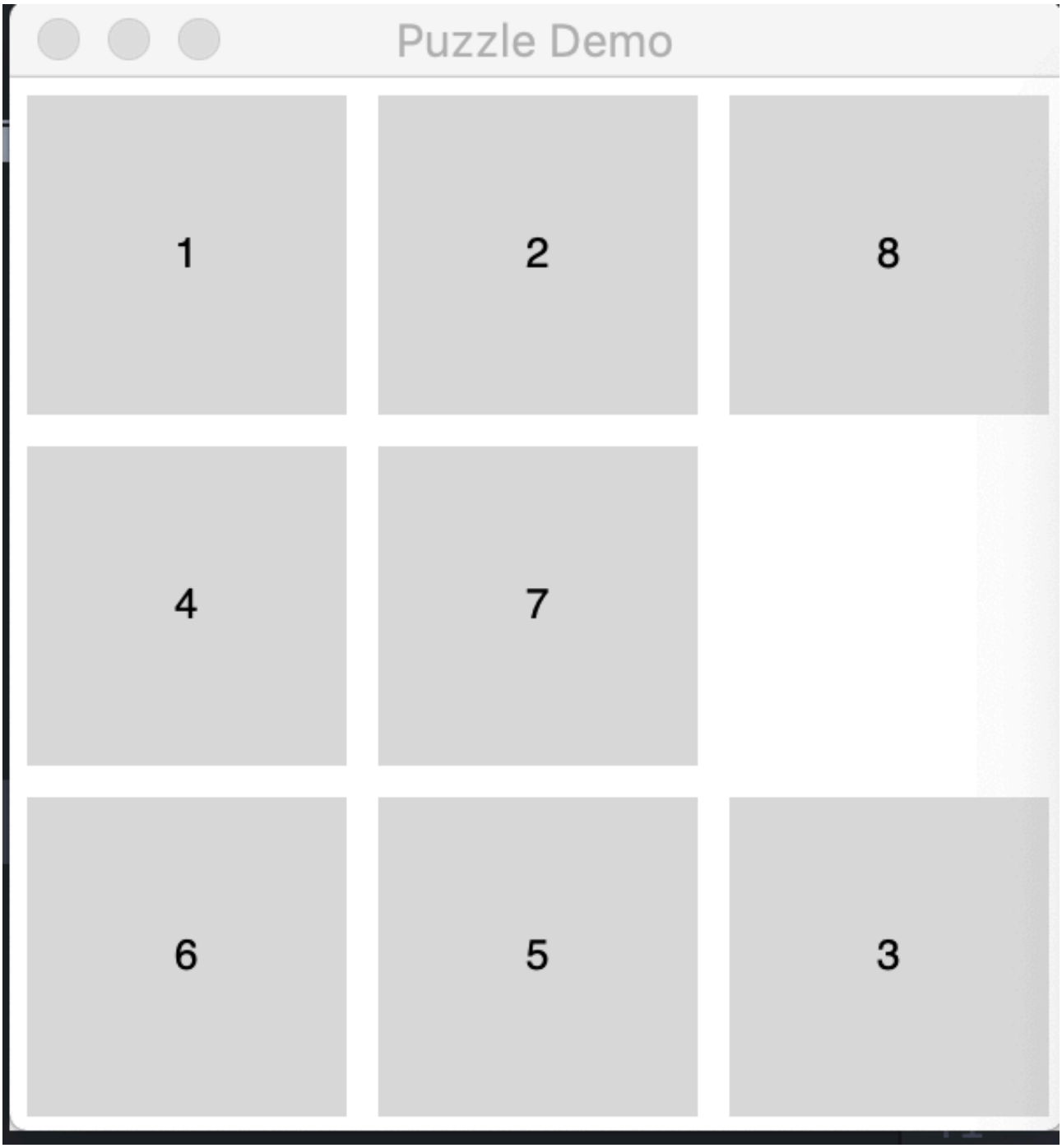
Discussion: One of the key aspect of graph is that it can have many loops and repeating paths. Since IDS and DFS only checks current path while BFS checks all visited nodes, BFS would be most efficient for small (not so big) graphs. However, if the graph is big, I would use DFS over IDS. Although IDS does give shortest path, it throws away the work of previous DFS calls, and would be less efficient for graph.

Extension

Extension 1: I made my own search problem. Under robot game.py, I made a problem object that takes a dimension of a maze. Then, it makes a 2D square array of given dimension and assigns wall with pre-assigned probability. Then, it chooses a start state and a goal state of the robot randomly. Then, I wrote successor method that would move the robot to up, down, right, left, if there arent any walls blocking it. For better display of the output, I also made a graphic feature of the problem, which can be seen by running robot test.py. Demo video is attached under robot_demo.



Extension 2: I made my own search problem: 8-puzzle problem. The problem object is implemented in puzzle.py and the test with graphic simulation can be seen by running puzzle_test.py. Demo video is attached under puzzle demo.



Extension 3: I made cannibal problem more general by allowing any number of boat cap. When making CannibalProblem object, one just needs to assign boat_max a number and it overrides the default value. The picture below shows the result for 5,4,1 problem with 3 boat capacity. When the capacity was 2, 5,4,1 problem was unsolvable; however, we can clearly see that 3 boat capacity does led to more states and more complexity as the problem becomes solvable.

```
Seungjae-Macbook-Pro:provided SJLEE$ python3 cannibals.py
bfs:  [(5, 4, 1), (5, 1, 0), (5, 2, 1), (2, 2, 0), (3, 2, 1), (1, 1, 0), (2, 1, 1), (0, 0, 0)]
dfs:  [(5, 4, 1), (5, 2, 0), (5, 3, 1), (4, 3, 0), (4, 4, 1), (3, 3, 0), (4, 3, 1), (3, 2, 0), (3, 3, 1), (2, 2, 0), (3, 2, 1), (2, 1, 0),
 (2, 2, 1), (1, 1, 0), (2, 1, 1), (1, 0, 0), (1, 1, 1), (0, 1, 0), (0, 2, 1), (0, 0, 0)]
ids:  [(5, 4, 1), (5, 1, 0), (5, 2, 1), (2, 2, 0), (3, 2, 1), (1, 1, 0), (2, 1, 1), (0, 0, 0)]
```