

Missionaries and Cannibal Problem

The Problem Statement

Find the solution of the Missionaries and Cannibals problem using both Breadth-First Search(BFS) and Depth First Search(DFS)

Missionaries and Cannibals Problem

In this popular game, there are three missionaries and three cannibals waiting on the left shore to cross a river. They have one boat, only two people can ride the boat at one time and the boat always needs at least one person to steer it. There is also the constraint of never letting the number of cannibals exceed the no of missionaries on either shore.

Defining States of the Missionaries and Cannibals Problem

The states are defined using a Python Dictionary.

State Space: {'M': no of Missionaries, 'C': no of Cannibals, 'B': State of the boat}

M represents the number of Missionaries and C represents the number of Cannibals on the left shore. (Only left shore is represented to avoid confusion)

The state of the boat is represented with B and can have values 0 or 1.

0 being the left shore and 1 being the right.

Initial State : {'M': 3, 'C': 3, 'B': 0}

This can be interpreted as 3 missionaries and 3 cannibals in the left shore.

Goal State: {'M': 3, 'C': 3, 'B': 0}

Which can be interpreted as 0 missionaries and cannibals on the left shore, implying everyone is on the right shore.

Which Search was better?

The algorithm we used for searching Breadth-First and Depth First makes use of queue and stack respectively. We also avoided repeated nodes and searched on the following order, if the states were acceptable.

1st: One missionary and One cannibal

2nd: 2 cannibals

3rd: 2 missionaries

4th: 1 cannibal

5th: 1 missionary

For our algorithm, the Depth First Search visited lower no of nodes to get to the solution than the Breadth-First Search.

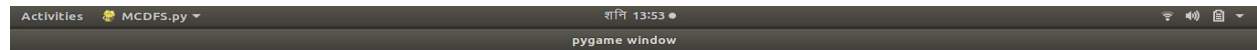
Methodology

To solve this problem we use Python Programming Language and Spyder environment.

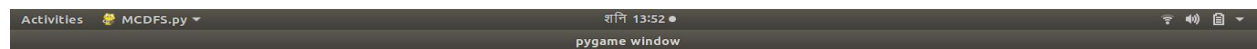
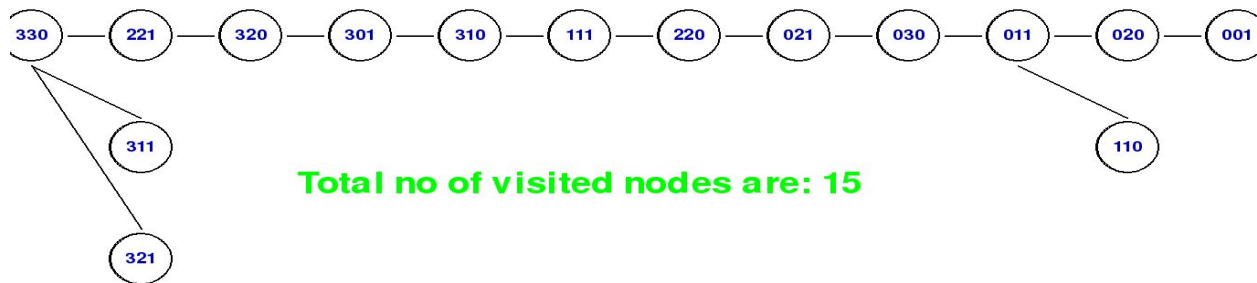
The logic behind the searching algorithm uses simple programming concepts like functions, stack, queue, dictionary, tuple and loops and to make a tree, we used pygame library.

Output

The outputs in the window:



BFS:



DFS:

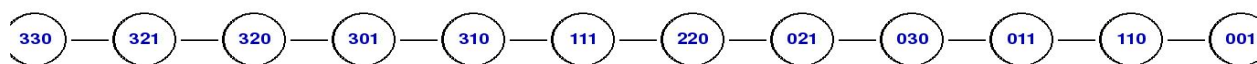
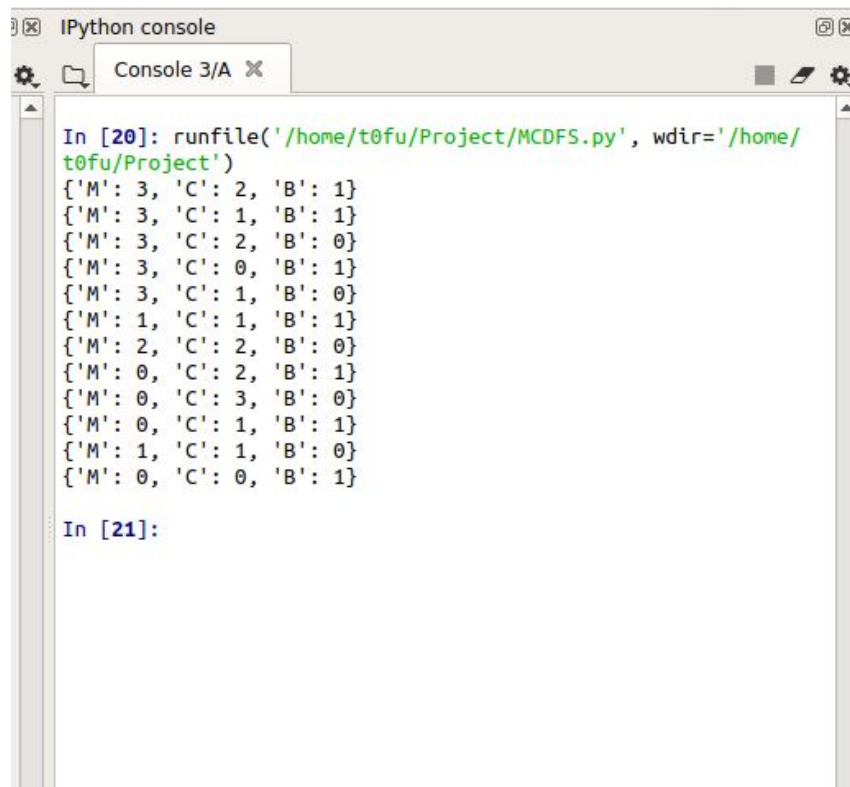


Fig: BFS and DFS tree

```
In [19]: runfile('/home/t0fu/Project/MCBFS.py', wdir='/home/
t0fu/Project')
{'M': 2, 'C': 2, 'B': 1}
{'M': 3, 'C': 1, 'B': 1}
{'M': 3, 'C': 2, 'B': 1}
{'M': 3, 'C': 2, 'B': 0}
{'M': 3, 'C': 0, 'B': 1}
{'M': 3, 'C': 1, 'B': 0}
{'M': 1, 'C': 1, 'B': 1}
{'M': 2, 'C': 2, 'B': 0}
{'M': 0, 'C': 2, 'B': 1}
{'M': 0, 'C': 3, 'B': 0}
{'M': 0, 'C': 1, 'B': 1}
{'M': 0, 'C': 2, 'B': 0}
{'M': 1, 'C': 1, 'B': 0}
{'M': 0, 'C': 0, 'B': 1}

In [20]:
```

Fig: BFS output in the console



```
IPython console
Console 3/A

In [20]: runfile('/home/t0fu/Project/MCDFS.py', wdir='/home/
t0fu/Project')
{'M': 3, 'C': 2, 'B': 1}
{'M': 3, 'C': 1, 'B': 1}
{'M': 3, 'C': 2, 'B': 0}
{'M': 3, 'C': 0, 'B': 1}
{'M': 3, 'C': 1, 'B': 0}
{'M': 1, 'C': 1, 'B': 1}
{'M': 2, 'C': 2, 'B': 0}
{'M': 0, 'C': 2, 'B': 1}
{'M': 0, 'C': 3, 'B': 0}
{'M': 0, 'C': 1, 'B': 1}
{'M': 1, 'C': 1, 'B': 0}
{'M': 0, 'C': 0, 'B': 1}

In [21]:
```

Fig: DFS in the console

Note to execute the program:

MCDFS.py and MCBFS.py can be executed in spyder.

A pdf file containing the program's source code is also attached.