Practical - 3

October 26, 2023

Write a program to implement missionaries and cannibals problem using `Depth First Search algorithm` .

[6]:
```
from typing import Optional, List, Tuple
import random, copy, time, heapq
import matplotlib.pyplot as plt
import numpy as np

class State:
    def __init__(self, N = 3, intital=False) -> None:
        self.M1 = N
        self.C1  = N
        self.M2: int = 0
        self.C2: int = 0
        self.boat_side = 1
        self.max = N
        self.initial = intital

    def __lt__(self, other):
        return self.M1 < other.M1
    def __hash__(self):
        return hash((self.M1, self.C1, self.M2, self.C2, self.boat_side))
    def __eq__(self, other):
        return self.M1 == other.M1 and self.C1 == other.C1 and self.M2 == other.
 ↪M2 and self.C2 == other.C2 and self.boat_side == other.boat_side
    def __str__(self) -> str:
        island1 = 'M' * self.M1 + 'C' * self.C1
        island2 = 'M' * self.M2 + 'C' * self.C2
        if self.initial:
            return f''' {island1} ------> {island2}'''
        if self.boat_side == 2:
            return f''' {island1} ------> {island2}'''
        else:
            return f''' {island1} <------  {island2}'''

class DFS_MC:
    @staticmethod
    def gen_initial_state(N : int):
```

```
        return State(N=3)
    @staticmethod
    def is_valid_state(state: State) -> bool:
        # Check if the state is valid (no more cannibals than missionaries on either side)
        if (state.C1 > state.M1 > 0) or (state.C2 > state.M2 > 0):
            return False
        if ( state.C1 > state.max or state.C1 < 0) or ( state.M1 > state.max or state.M1 < 0) or ( state.C2 > state.max or state.C2 < 0) or ( state.M2 > state.max or state.M2 < 0):
            return False
        return True
    @staticmethod
    def check_visited(state, visited_states):
        for s in visited_states:
            if state == s:
                return False
        return True
    @staticmethod
    def gen_child_states(state: State,states, visited_states: List[State]) -> List[State]:
        # Define the possible actions (moving missionaries and cannibals)
        actions = [(1, 0), (2, 0), (0, 1),  (1, 1), (0, 2)]
        for action in actions:
            # Determine the direction (from 1 to 2 or vice versa)
            if state.boat_side == 1:
                next_state = State()
                next_state.M1 = state.M1 - action[0]
                next_state.C1 = state.C1 - action[1]
                next_state.M2 = state.M2 + action[0]
                next_state.C2 = state.C2 + action[1]
                next_state.boat_side = 2
            else:
                next_state = State()
                next_state.M1 = state.M1 + action[0]
                next_state.C1 = state.C1 + action[1]
                next_state.M2 = state.M2 - action[0]
                next_state.C2 = state.C2 - action[1]
                next_state.boat_side = 1
            # Check if the generated state is valid and hasn't been visited before
            if DFS_MC.is_valid_state(next_state) and DFS_MC.check_visited(next_state,visited_states):
                states.append(next_state)
        return states
class Run_DFS:
    @staticmethod
```

```
    def dfs_search(initial_state: State):
        visited_states = set()
        stack = [(initial_state, [])]
        visited_states.add(initial_state)
        while stack:
            current_state, path = stack.pop()
            if DFS_MC.check_visited(current_state,visited_states):
                visited_states.add(current_state)
            if current_state.M1 == 0 and current_state.C1 == 0:
                return path
            child_states = DFS_MC.gen_child_states(current_state, [],␣
↪visited_states)
            for child_state in child_states:
                stack.append((child_state, path + [child_state]))
        return None

class Run_DFS_Tree:
    @staticmethod
    def dfs_tree(initial_state: State, max_depth: int):
        visited_states = set()
        stack = [(initial_state, None)]  # (state, parent_state) pairs
        tree = {initial_state: None}  # Dictionary to store the tree structure
        while stack:
            current_state, parent_state = stack.pop()
            visited_states.add(current_state)
            if current_state.M1 == 0 and current_state.C1 == 0:
                return tree
            if max_depth is None or len(tree[current_state]) < max_depth:
                child_states = DFS_MC.gen_child_states(current_state, [],␣
↪visited_states)

                for child_state in child_states:
                    if child_state not in tree:
                        tree[child_state] = current_state  # Update the tree
                        stack.append((child_state, current_state))
        return tree
```

[7]:
```
# 5. Driver Function
__name__ == '__main__'
if __name__ == "__main__":
    initial_state = State(N=3,intital=True)
    dfs_path = Run_DFS.dfs_search(initial_state)
    print(initial_state)
    if dfs_path:
        print("DFS Solution Path:")
        for i, state in enumerate(dfs_path):
            print(f"Step {i + 1}:")
```

```
            print(state)
            print("--------------------")
    else:
        print("No solution found.")
```

```
 MMMCCC ------>
DFS Solution Path:
Step 1:
 MMMC ------> CC
--------------------
Step 2:
 MMMCC <------  C
--------------------
Step 3:
 MMM ------> CCC
--------------------
Step 4:
 MMMC <------  CC
--------------------
Step 5:
 MC ------> MMCC
--------------------
Step 6:
 MMCC <------  MC
--------------------
Step 7:
 CC ------> MMMC
--------------------
Step 8:
 CCC <------  MMM
--------------------
Step 9:
 C ------> MMMCC
--------------------
Step 10:
 CC <------  MMMC
--------------------
Step 11:
  ------> MMMCCC
--------------------
```