Assignment 1

CP468
October 4th 2019

Austin Bursey , Aaron Exley, Tim McGill, Joseph Myc

1 Problem Formulation

A problem is defined by 5 items :

1. $InitialState$
2. $Actions$
3. $Transitionspace$
4. $GoalTest$
5. $Pathcost$

1. We define our initial state as:

3:3 L 0:0

We choose this notation to view the count/ratio of missionaries to cannibals on any given side. The left ration is 3:3 since there is three missionaries and three cannibals on the left river bank. The right ration is 0:0 in the initial state due to nobody being on the right river bank. The "L" signifies the position of the boat, "L" for Left and "R" for Right.

2. All of the available Action-state pairs are listed below :

$$
\begin{aligned}
&Action(3:3L0:0) = \{MR, MMR, MCR, CCR, CR\}\\
&Action(3:2R0:1) = \{CL\}\\
&Action(3:1R0:2) = \{CL, CCL\}\\
&Action(2:2R1:1) = \{ML, MCL, CL\}\\
&Action(3:2L0:1) = \{MR, MMR, MCR, CCR, CR\}\\
&Action(3:1R0:2) = \{CL, CCL\}\\
&Action(3:0R0:3) = \{CL, CCL\}\\
&Action(3:1L0:2) = \{MR, MMR, MCR, CR\}\\
&Action(1:1R2:2) = \{ML, MML, MCL, CCL, CL\}\\
&Action(2:2L1:1) = \{MR, MMR, MCR, CCR, CR\}\\
&Action(0:2R3:1) = \{ML, MML, MCL, CL\}\\
&Action(0:3L3:0) = \{CCR, CR\}\\
&Action(0:1R3:2) = \{ML, MML, MCL, CCL, CL\}\\
&Action(1:1L2:2) = \{MR, MCR, CR\}\\
&Action(0:2L3:1) = \{CCR, CR\}
\end{aligned}
$$

*All states with that are playable states are shown here. We define a playable state as a state that is not the goal state nor a state in which the cannibals outnumber the missionaries causing a game over.

3. Transition model

A boat carries missionaries and /or cannibals from one side of the river to the other. Can move up to two people from one side to the other.

The next state is determined by the current state, (# of cannibals on left, # of Missionaries on left — (# of cannibals on right, # of Missionaries on right) and the action (combination of passengers to take to the other side of the river), the next state is one that does not allow for number of cannibals to exceed the number of missionaries on either side of the river.

4.

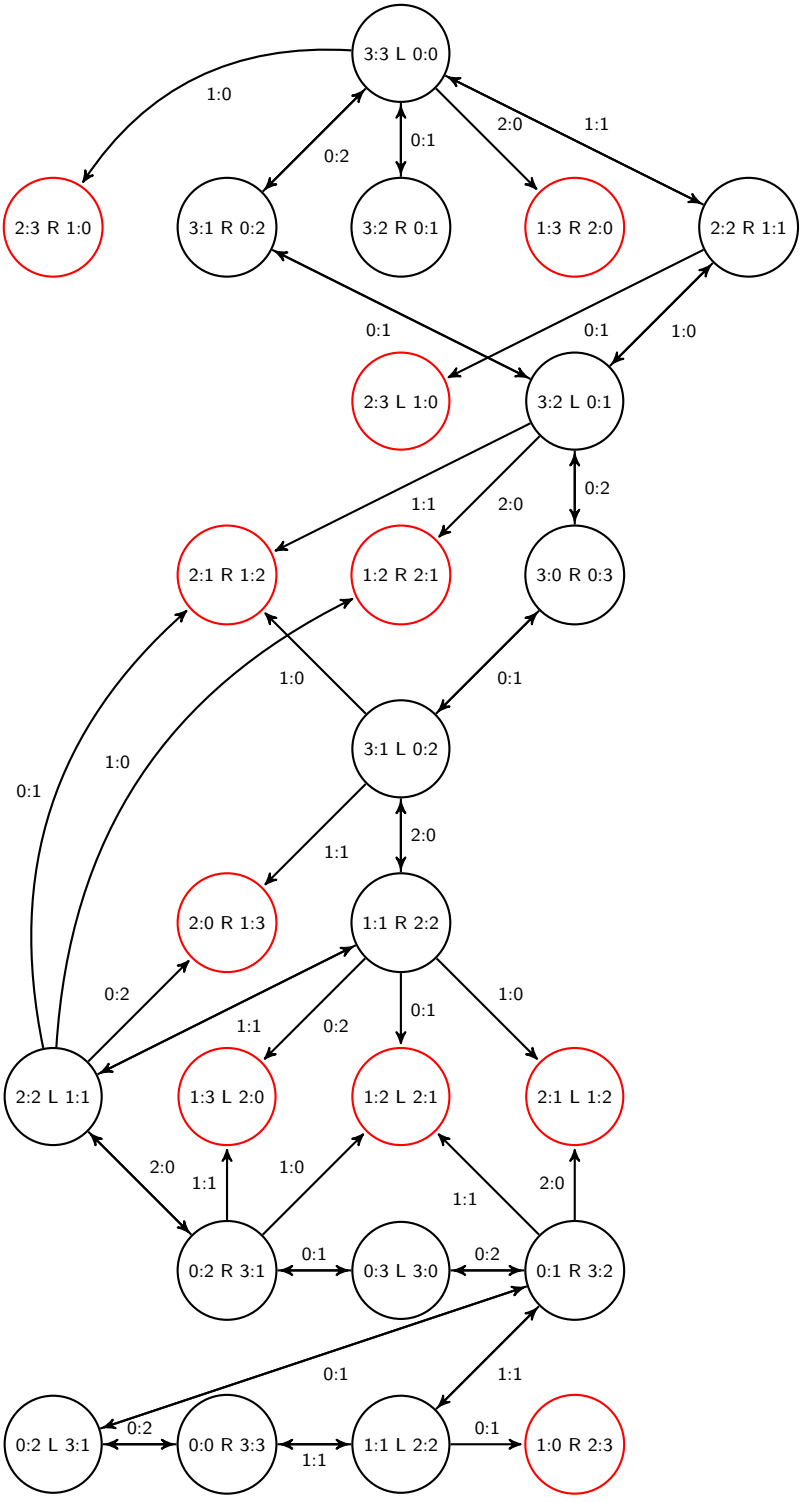
The goal text is explicitly when the state is (0:0 R 3:3) or to explain in words, when all three missionaries and cannibals are on the right river bank.

5.

The path cost we decided is each action has a value of 1. Thus the Function would be:

cost(next state) = cost(current state) + 1

2 Complete State Space



3 Implementation of A* using python

```
from itertools import combinations
from copy import deepcopy
from queue import Queue
from heapq import heappop, heappush

solutions = []
hashTable= \{\}
SOLUTION = [[0, 0], [3, 3], 'R']

class Tree:

    def __init__(self, state, parent = None, action = None, pathCost = 0):
        self.children = [] # The children of the node
        self.state = state # The state of the node [[ML, CL], [MR, CR], 'Side boat is on']
        self.parent = parent # The parent of the node
        self.action = action # The action taken to get here
        self.pathCost = pathCost # The cost to get to the node, used for BFS
        # The key version of the node used for the key in a dictionary
        self.key = "{}{}{}{}{}".format(self.state[0][0],
self.state[0][1],self.state[1][0],self.state[1][1],self.state[2])
        # A*
        self.g = -1 # cost to get to the node, used for A*
        self.f = -1 # The cost to get to the node + a guess of how for it
#needs to go, used for A*

        # A Check if the current state is a solution and add it to solutions if it is
        if state == SOLUTION:
            solutions.append(self)
            return

    def getChildren(self):
        '''
        Returns: (Tree[]) The array of children of the current node
        '''
        return self.children
```

```
def createChildren(self):
    '''
    Description: Creates the possible children then returns the result
    Returns: (Tree[]) The array of children of the current node
    '''

    options = ''
    count = 0
    children = []

    # Determines the location of the boat
    if self.state[2] == 'L':
        # Adds 1 or 2 M and C to create the possible options
        # That can be done
        options += min(self.state[0][0], 2) * 'M'
        options += min(self.state[0][1], 2) * 'C'

        # Counts the total number of people on the side
        count = min(self.state[0][0] + self.state[0][1], 2)
    else:
        options += min(self.state[1][0], 2) * 'M'
        options += min(self.state[1][1], 2) * 'C'
        count = min(self.state[1][0] + self.state[1][1], 2)

    # Generates all the choose 1 and choose 2 options from the current state
    comb = set(combinations(options, 1))
    if count > 1:
        comb = comb.union(set(combinations(options, 2)))

    # Sort the options so the order is consistant
    # This is needed because the combinations function produces a random order
    comb = sorted(comb)

    # Go through the combinations and append a new tree node based on the M and C's
    for com in comb:

        m = com.count('M')
        c = com.count('C')

        # Filter out any children that would lead back to the parent state
        if self.action is None or not (m == self.action[0] and c == self.action[1]):
            children.append(Tree(self.move((m, c)), self, (m, c), self.pathCost + 1))

    return children
```

```
    def AStarAddAllChildren(self):
        '''
        Description: Used for A*
                     Gets the array of children add adds them to the children array
                     and returns the result
        Returns: (Tree[]) The array of children
        '''
        self.children = self.createChildren()
        return self.children

    def addAllChildren(self):
        '''
        Description: Used for BFS
                     Gets the array of children add adds them to the children array
                     if the current node is a valid node
        '''
        if self.check():
            self.children = self.createChildren()

    def move(self, moves):
        '''
        Description: applies a move to a node
        '''
        newState = deepcopy(self.state)

        if newState[2] == 'L':
            newState[1][0] += moves[0]
            newState[1][1] += moves[1]
            newState[0][0] -= moves[0]
            newState[0][1] -= moves[1]
        else:
            newState[0][0] += moves[0]
            newState[0][1] += moves[1]
            newState[1][0] -= moves[0]
            newState[1][1] -= moves[1]

        newState[2] = 'L' if newState[2] == 'R' else 'R'

        return newState

    def checkValid(self):
        '''
        Description: Checks if a node is valid
        Returns: (Bool) False if C > M at any point True otherwise
        '''
        return not (((self.state[0][0] < self.state[0][1]) and self.state[0][0] != 0) or ((s
```

```
    def check(self):
        '''
        Description: Checks if a node is valid and not used
        Returns: (Bool) False if C > M at any point and the node hasn't been used yet True o
        '''
        return self.checkValid() and not self.isInTree()

    def isInTree(self):
        '''
        Description: Checks if the current node has been used (ie is it in the tree)
        Returns: (Bool) True if the node is in the tree, False otherwise
        '''

        value = hashTable.get(self.key)
        if value == None: #value doesn't exist in the table
            hashTable[self.key] = True
            return False

        return True

    def __lt__(self, value):
        return self.f < value.f

    def __str__(self, level=0):
        ret = "\t" * level + repr(self.state) + "\n"
        for child in self.children:
            ret += child.__str__(level + 1)
        return ret

    def __repr__(self):
        return str(self.state) + " " + str(self.f)

def AStar(start):
    '''
    Description: Runs the A* algorithm from the given start node
    Prints: The nodes searched and the optimal solution found
    '''

    frontier = [start]

    checked = {}

    start.f = h(start)
    start.g = 0
```

```
    while len(frontier) != 0:

        current = heappop(frontier)

        if current.state == SOLUTION:
            print(start)
            print_solution(current)
            return

        checked[current.key] = True

        for child in current.AStarAddAllChildren():

            if checked.get(child.key):
                continue

            gScore = current.g + 1

            if child.checkValid() and (child.g == -1 or gScore < child.g):

                child.parent = current
                child.g = gScore
                child.f = child.g + h(child)

                if child not in frontier:
                    heappush(frontier, child)
    return None

def h(node):
    '''
    Description: The heuristic for A*
                 h = (number of people on left side) - 1
                 Is admissible because every round trip (boat move to right side then back)
                 results in at most A net of 1 person (except the last move
      since at least one person has to return.
    '''
    return node.state[0][0] + node.state[0][1] - 1

def BFS(root):
    '''
    Description: Performs a BFS search from the given start node
    Prints: The nodes searched and the optimal solution found
    '''
    head = Queue()
    head.put(root)
```

```
    while (head.empty() is not True ):
        node = head.get()
        node.addAllChildren()

        for child in node.getChildren():
            head.put(child)

    print(root)

    for solution in solutions:
        print_solution(solution)

def print_solution(end):
    '''
    Description: Prints the solution from the given end point
    Prints: The solution
    '''
    s = []

    node = end

    while node is not None:
        s.append(node)
        node = node.parent

    print("=" * 20)
    while len(s) != 0:
        n = s.pop()

        p = "({:3} {:3} | {:3} {:3})[{:1}]".format(n.state[0][0] * 'M', n.state[0][1] * 'C',
                                                   n.state[1][0] * 'M', n.state[1][1] * 'C',
        print(p)

    print("=" * 20)
    print()

def main():
            # Left side, Right side
            #[[M, C], [M, C], 'Side boat is on']
            # Cost of each action is 1
    print("BFS")
    root = Tree([[3, 3], [0, 0], 'L'])
    BFS(root)
    print("=" * 20)
    print("AStar")
    AStar(Tree([[3, 3], [0, 0], 'L']))
```

```
if __name__ == "__main__":
    main()
```

The output of the code is the optimal solution :

```
AStar
[[3, 3], [0, 0], 'L']
    [[3, 2], [0, 1], 'R']
    [[3, 1], [0, 2], 'R']
        [[3, 2], [0, 1], 'L']
            [[3, 0], [0, 3], 'R']
                [[3, 1], [0, 2], 'L']
                    [[2, 1], [1, 2], 'R']
                    [[2, 0], [1, 3], 'R']
                    [[1, 1], [2, 2], 'R']
                        [[1, 2], [2, 1], 'L']
                        [[1, 3], [2, 0], 'L']
                        [[2, 1], [1, 2], 'L']
                        [[2, 2], [1, 1], 'L']
                            [[2, 1], [1, 2], 'R']
                            [[2, 0], [1, 3], 'R']
                            [[1, 2], [2, 1], 'R']
                            [[0, 2], [3, 1], 'R']
                                [[0, 3], [3, 0], 'L']
                                    [[0, 1], [3, 2], 'R']
                                        [[0, 2], [3, 1], 'L']
                                            [[0, 0], [3, 3], 'R']
                                        [[1, 1], [2, 2], 'L']
                                        [[1, 2], [2, 1], 'L']
                                        [[2, 1], [1, 2], 'L']
                                [[1, 2], [2, 1], 'L']
                                [[1, 3], [2, 0], 'L']
            [[2, 2], [1, 1], 'R']
            [[2, 1], [1, 2], 'R']
            [[1, 2], [2, 1], 'R']
    [[2, 3], [1, 0], 'R']
    [[2, 2], [1, 1], 'R']
        [[2, 3], [1, 0], 'L']
        [[3, 2], [0, 1], 'L']
            [[3, 1], [0, 2], 'R']
            [[3, 0], [0, 3], 'R']
            [[2, 1], [1, 2], 'R']
            [[1, 2], [2, 1], 'R']
    [[1, 3], [2, 0], 'R']
```

```
====================
(MMM CCC |         )[L]
(MMM C   |      CC )[R]
(MMM CC  |      C  )[L]
(MMM     |      CCC)[R]
(MMM C   |      CC )[L]
(M   C   | MM   CC )[R]
(MM  CC  | M    C  )[L]
(    CC  | MMM  C  )[R]
(    CCC | MMM     )[L]
(    C   | MMM  CC )[R]
(    CC  | MMM  C  )[L]
(        | MMM  CCC)[R]
====================
```