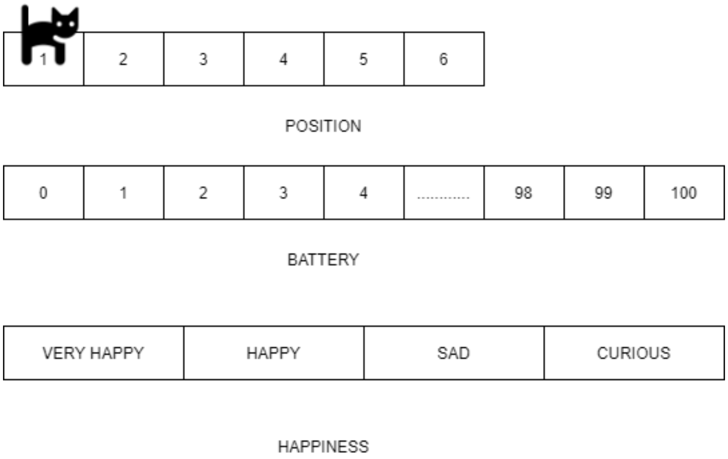
CS 6364 HW 1

Rohith Peddi, RXP190007

September 22, 2020

1 INTELLIGENT AGENTS



State space	Possible Percepts	Possible Actions

Positions(6), Battery(101)
Happiness(6)
	Clap, Pet, Bump	Walk, Purr, Meow,
		Blink, Stop, Turn around

1.1 State Space

A factored representation of states, where each state is described by three variables : Position(P), Battery(B), Happiness(H) = S(P, B, H)

Total number of states : {1,2,3,4,5,6} x {0, 1, ... 100} x {VH, H, S, C}

$\implies$ (6)(101)(4) = 2424

1.2 State update functions

```
State = UPDATE-STATE(state, action, percept, model)

Happiness + 1 = UPDATE-KITTY[Happiness, Clap]
Happiness + 2 = UPDATE-KITTY[Happiness, Pet]
Happiness + 1 = UPDATE-KITTY[Happiness, Bump]
Max(0, Battery - 5) = UPDATE-KITTY[Battery, Bump]

Battery - 1 = UPDATE-KITTY[Battery, Walk]
Battery - 4 = UPDATE-KITTY[Battery, Turn around]
Battery - 3 = UPDATE-KITTY[Battery, Purrs]
Battery - 2 = UPDATE-KITTY[Battery, Meows]

Position - 1 = UPDATE-KITTY[Position, Walk]
Position = UPDATE-KITTY[Position, Purr|Meow|Blink|Stop|Start]
```

When battery is discharged no more actions are possible

GENERIC PSEUDO CODE FOR UTILITY BASED AGENTS

```
function UTILITY-BASED-AGENT(percept) returns an action
    """
    persistent: state, the agent's current conception of the world state
                model, describes how the next state depends on current state and action
                utility-function, describes agent's utility function
                plan, a sequence of actions to take, empty initially
                action, most recent action, none initially
    """

    state = UPDATE-STATE(state, action, percept, model)
    if plan is empty then
        plan = PLAN(state, utility-function, model)
    action = FIRST(plan)
    plan = REST(plan)
return action
```

function ***PLAN*** *in the code for utility based agent generates the next set of actions that maximise the utility function of the kitty*

```
Function PLAN has the following inputs:
1. Current state
2. Utility Function
3. Model

As the state consists of (P,B,H) where Battery is included.
Once Battery reaches 0. plan becomes empty with no action.

After the battery is revived, function PLAN generates sequences of ACTIONS
```

1.3 Utility function

OBSERVATIONS

As there is no particular goal state which is evidently visible. We can consider the following formulations for goal state

1. Reaching a particular square in box with Maximum Happiness and Maximum battery
2. For the given percept sequence choosing actions inorder to maximise Happiness and Maximize battery life

So possible utility functions are:

1. $S(P, B, H) = -|P - 6| + H * 10 + B$ (Considering position 6 as goal state)
2. $S(P, B, H) = H * 10 + B$

Considering following for each state of H, Happiness i,e VH = 4, H = 3, S = 0, C = 1

2 PROBLEM FORMULATION FOR SEARCH

2.1 Problem Formulation

Formulate the search problem formally. State clearly how you represent states, (including the initial state and the goal state(s)), what actions are available in each state, including the cost of each action. (15 points)

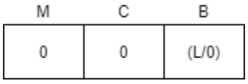
SOLUTION

Problem formulation components					
States	Initial state	Actions	Transition model	Goal test	Path cost

2.1.1 STATES

A state is determined by the following factors:

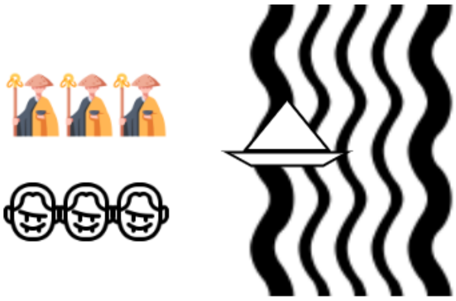
1. Number of missionaries on the destination side
2. Number of cannibals on the destination side
3. Side in which boat is present (Origin side(L/0), Destination side(R/1))

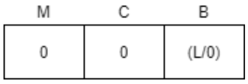


M	C	B
0	0	(L/0)

Total possible world states = $4 * 4 * 2 = 32$.
It comprises of both legal and illegal states.
Eg: State (1,2,1) is illegal as Number of cannibals on any side cannot exceed number of missionaries

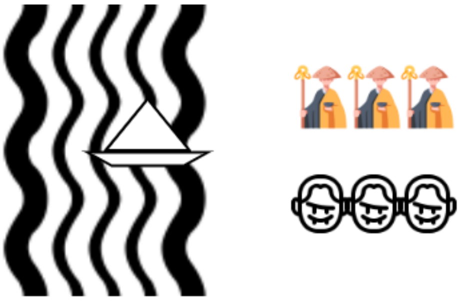
2.1.2 INITIAL STATE

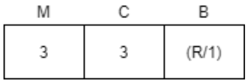




M	C	B
0	0	(L/0)

2.1.3 FINAL STATE





M	C	B
3	3	(R/1)

2.1.4 ACTIONS

Following are the different type of actions :

1. M - Transfer of 1 Missionary to the other side
2. C - Transfer of 1 Cannibal to the other side
3. MM - Transfer of 2 Missionaries to the other side
4. CC - Transfer of 2 Cannibals to the other side
5. MC - Transfer of 1 Missionary and 1 Cannibal to the other side

But possible actions for each state are limited by the actual number of missionaries and cannibals on the side and with the restriction of number of cannibals not exceeding the number of missionaries at any side of the river.

Following are possible states with possible actions and associated cost

State	[Legal action, Cost, Next state]		
(0,0,0)	[C, 1, (0,1,1)]	[CC, 1, (0,2,1)]	[MC, 1, (1,1,1)]
(0,1,0)	[C, 1, (0,2,1)]	[CC, 1, (0,3,1)]	[M, 1, (1,1,1)]
(0,2,0)	[C, 1, (0,3,1)]	[MM, 1, (2,2,1)]	
(0,3,0)			
(1,0,0)			
(1,1,0)	[MC, 1, (2,2,1)]	[MM, 1, (3,1,1)]	
(1,2,0)			
(1,3,0)			
(2,0,0)			
(2,1,0)			
(2,2,0)	[MC, 1, (3,3,1)]	[M, 1, (3,2,1)]	
(2,3,0)			
(3,0,0)	[C,1,(3,1,1)]	[CC, 1, (3,2,1)]	
(3,1,0)	[C, 1, (3,2,1)]	[CC, 1, (3,3,1)]	
(3,2,0)	[C, 1, (3,3,1)]		
(3,3,0)			
(0,0,1)			
(0,1,1)	[C, 1, (0,0,0)]		
(0,2,1)	[C, 1, (0,1,0)]	[CC, 1, (0,0,0)]	
(0,3,1)	[C, 1, (0,2,0)]	[CC, 1, (0,1,0)]	
(1,0,1)			
(1,1,1)	[MC, 1, (0,0,0)]	[M, 1, (0,1,0)]	
(1,2,1)			
(1,3,1)			
(2,0,1)			
(2,1,1)			
(2,2,1)	[MC, 1, (1,1,0)]	[MM, 1, (0,2,0)]	
(2,3,1)			
(3,0,1)			
(3,1,1)	[C, 1, (3,0,0)]	[MM, 1, (1,1,0)]	
(3,2,1)	[C, 1, (3,1,0)]	[M, 1, (2,2,0)]	[CC, 1, (3,0,0)]
(3,3,1)			

2.2 Solution for problem

2.2.1 Strategy/Motivation

1. Uniform cost search strategy is used for finding the solution for the problem
2. It is uninformed search which gives optimal solution
3. As all path costs equal to 1, this shall be equivalent to a breadth first search
4. Explored set is used to not consider the repeated states

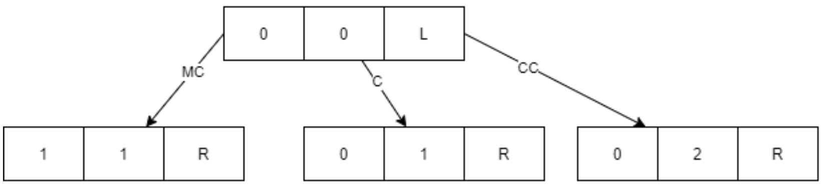
2.2.2 Solution

Obtained Path : (Node, Path cost)

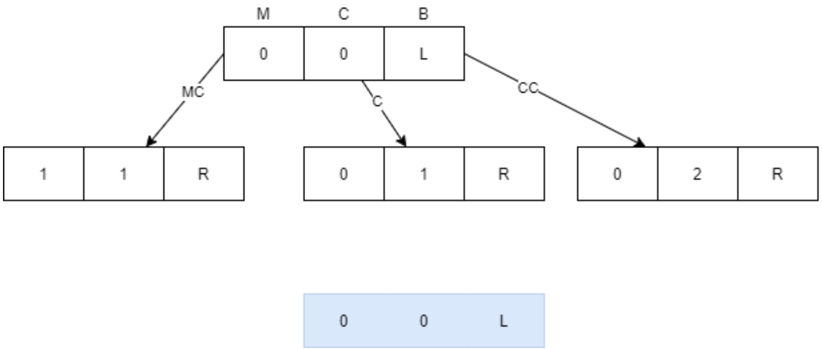
- (0, 0, 0), 0 [Initial state]
- (1, 1, 1), 1
- (0, 1, 0), 2
- (0, 3, 1), 3
- (0, 2, 0), 4
- (2, 2, 1), 5
- (1, 1, 0), 6
- (3, 1, 1), 7
- (3, 0, 0), 8
- (3, 2, 1), 9
- (2, 2, 0), 10
- (3, 3, 1), 11 [Goal state]

2.2.3 Solution using strategy

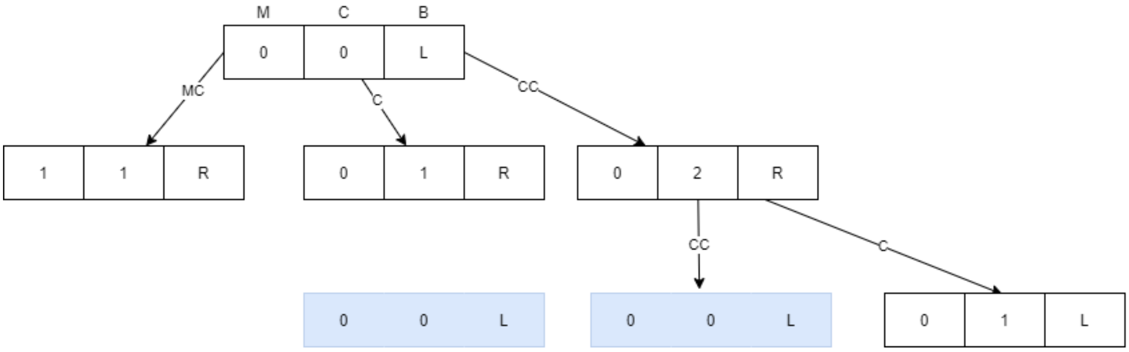
```
--------------------------------------------------------------------------------
UCS STEP : 1
1. Current Node : (0, 0, 0)
2. Evaluation function : 0
3. Frontier :
(1, <Node (0, 1, 1)>)
(1, <Node (1, 1, 1)>)
(1, <Node (0, 2, 1)>)
4. Explored :
[(0, 0, 0)]
5. Children :
[((0, 2, 1), 1), ((0, 1, 1), 1), ((1, 1, 1), 1)]
```



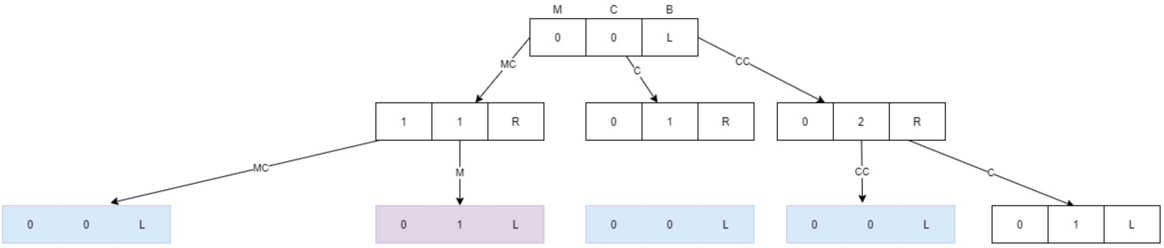
```
--------------------------------------------------------------------------------
UCS STEP : 2
1. Current Node : (0, 1, 1)
2. Evaluation function : 1
3. Frontier :
(1, <Node (0, 2, 1)>)
(1, <Node (1, 1, 1)>)
4. Explored :
[(0, 1, 1), (0, 0, 0)]
5. Children :
[((0, 0, 0), 2)]
```



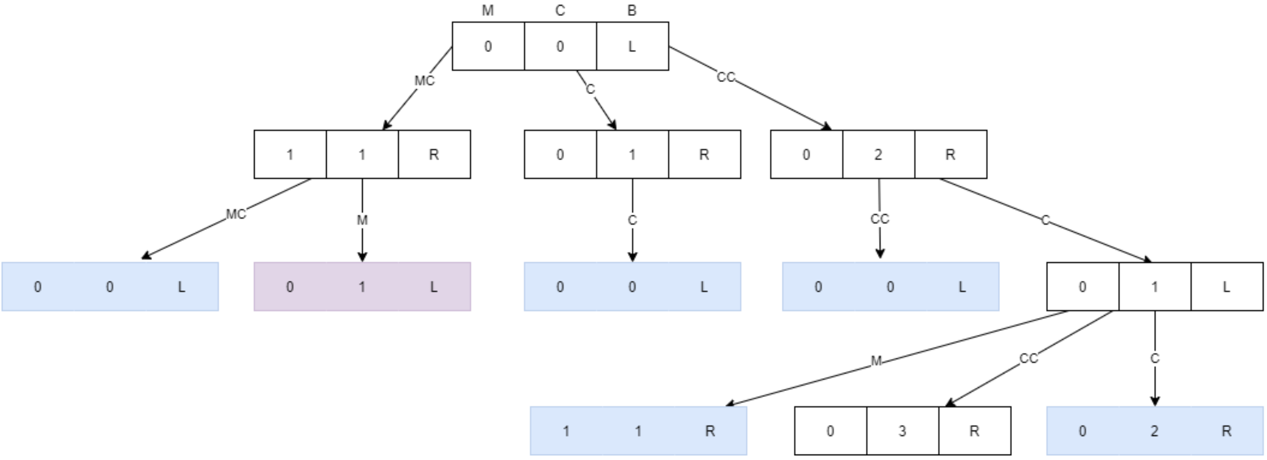
```
--------------------------------------------------------------------------------
UCS STEP : 3
1. Current Node : (0, 2, 1)
2. Evaluation function : 1
3. Frontier :
(1, <Node (1, 1, 1)>)
(2, <Node (0, 1, 0)>)
4. Explored :
[(0, 1, 1), (0, 2, 1), (0, 0, 0)]
5. Children :
[((0, 1, 0), 2)]
```



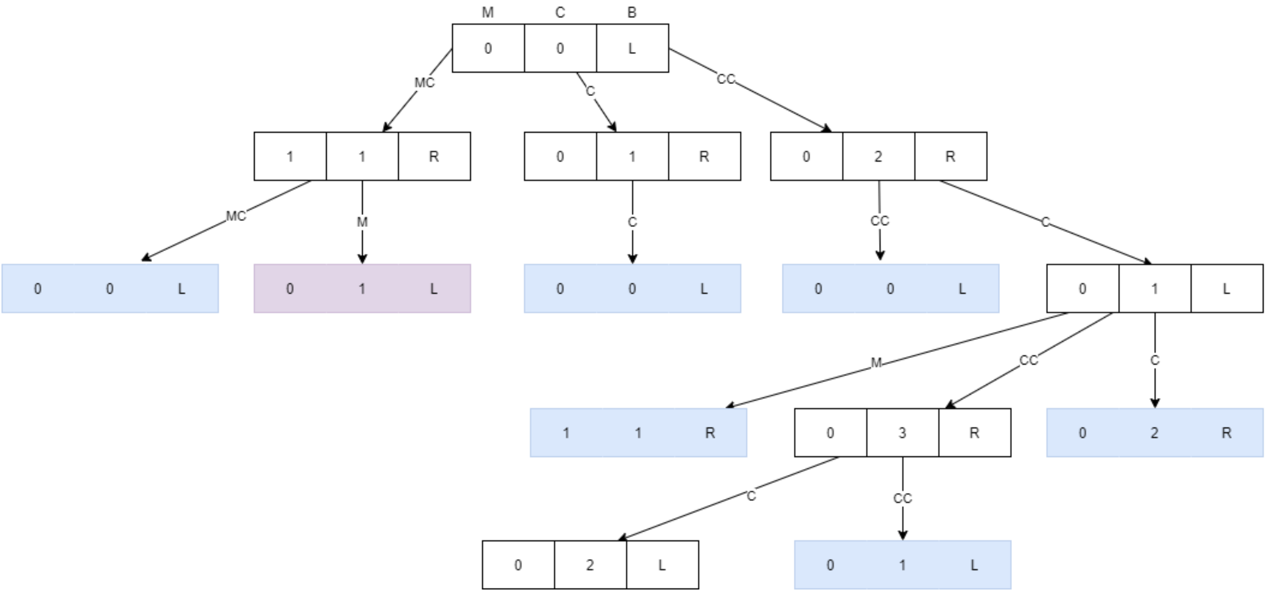
```
--------------------------------------------------------------------------------
UCS STEP : 4
1. Current Node : (1, 1, 1)
2. Evaluation function : 1
3. Frontier :
(2, <Node (0, 1, 0)>)
4. Explored :
[(0, 1, 1), (0, 2, 1), (0, 0, 0), (1, 1, 1)]
5. Children :
[((0, 0, 0), 2), ((0, 1, 0), 2)]
```



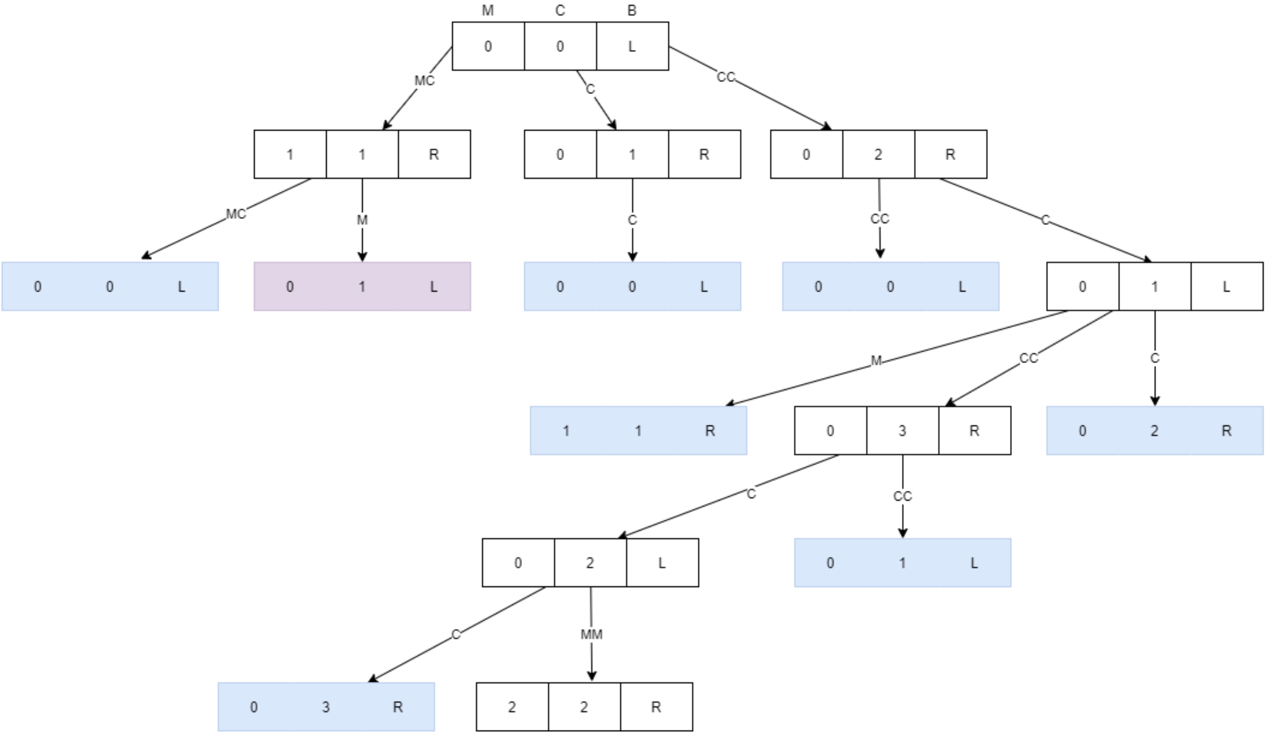
```
--------------------------------------------------------------------------------
UCS STEP : 5
1. Current Node : (0, 1, 0)
2. Evaluation function : 2
3. Frontier :
(3, <Node (0, 3, 1)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 1, 0), (0, 0, 0), (0, 2, 1)]
5. Children :
[((0, 3, 1), 3), ((1, 1, 1), 3), ((0, 2, 1), 3)]
```



```
--------------------------------------------------------------------------------
UCS STEP : 6
1. Current Node : (0, 3, 1)
2. Evaluation function : 3
3. Frontier :
(4, <Node (0, 2, 0)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 1, 0), (0, 3, 1), (0, 0, 0), (0, 2, 1)]
5. Children :
[((0, 2, 0), 4)]
```

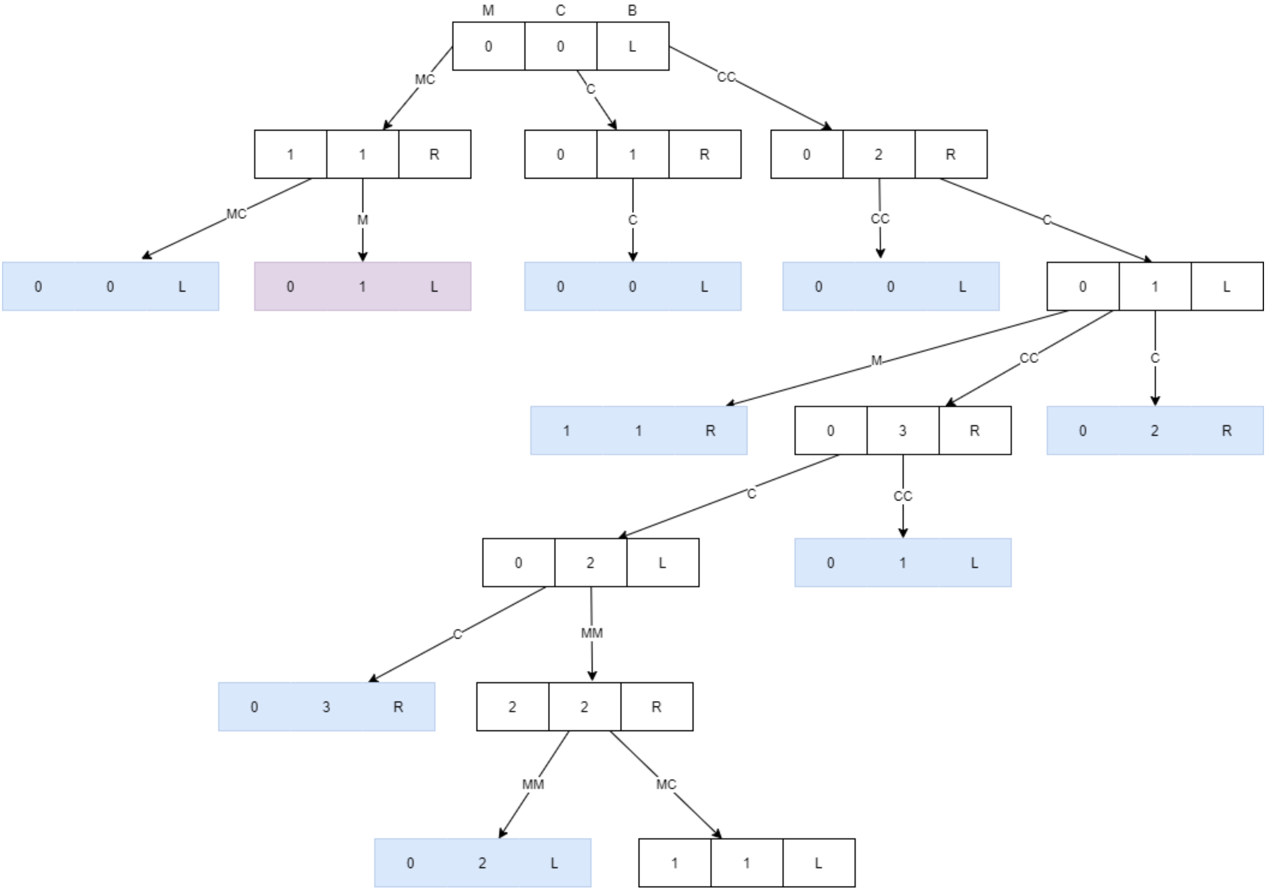


```
--------------------------------------------------------------------------------
UCS STEP : 7
1. Current Node : (0, 2, 0)
2. Evaluation function : 4
3. Frontier :
(5, <Node (2, 2, 1)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 2, 0), (0, 1, 0), (0, 3, 1), (0, 0, 0), (0, 2, 1)]
5. Children :
[((0, 3, 1), 5), ((2, 2, 1), 5)]
```

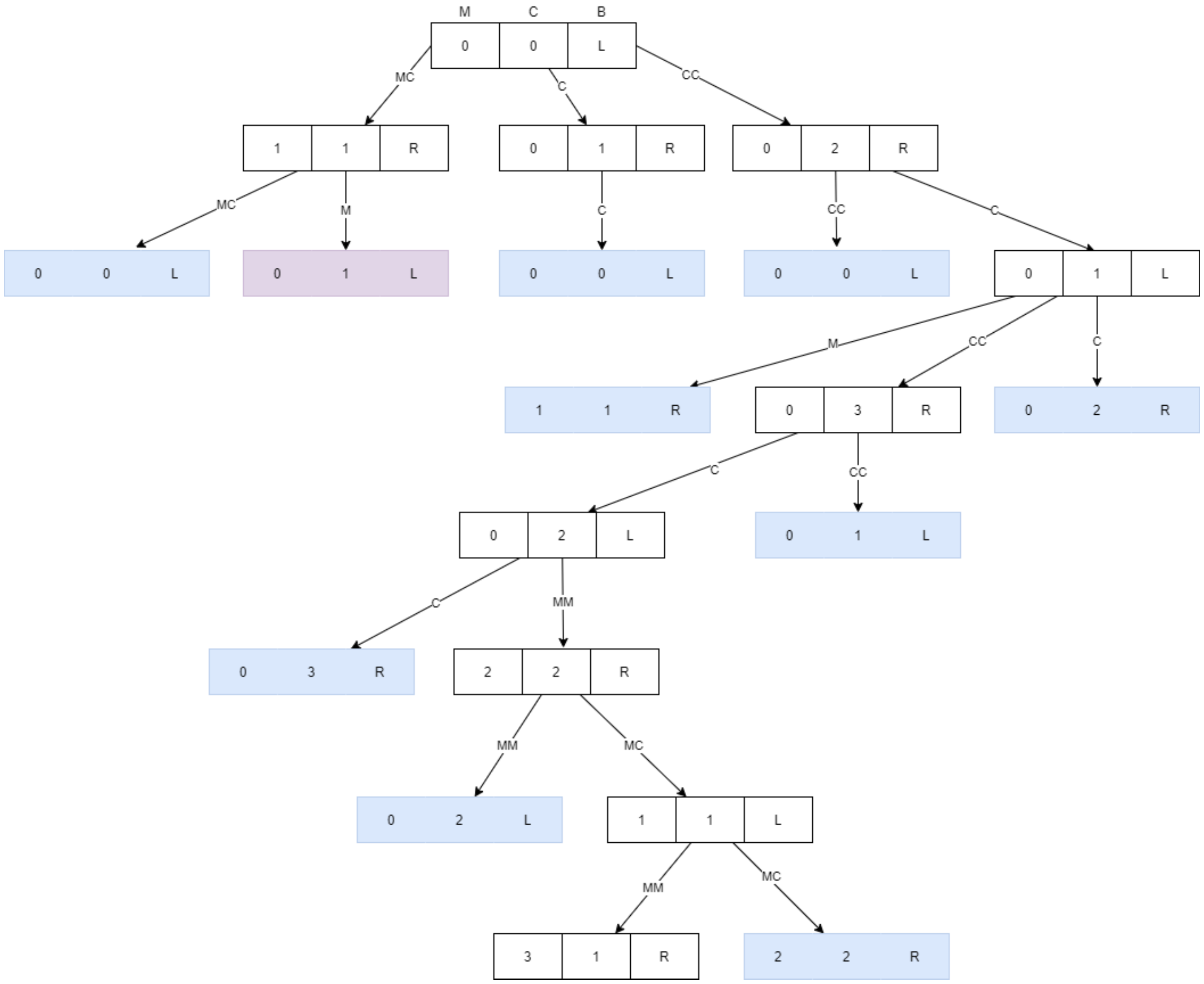


```
--------------------------------------------------------------------------------
UCS STEP : 8
1. Current Node : (2, 2, 1)
2. Evaluation function : 5
3. Frontier :
```

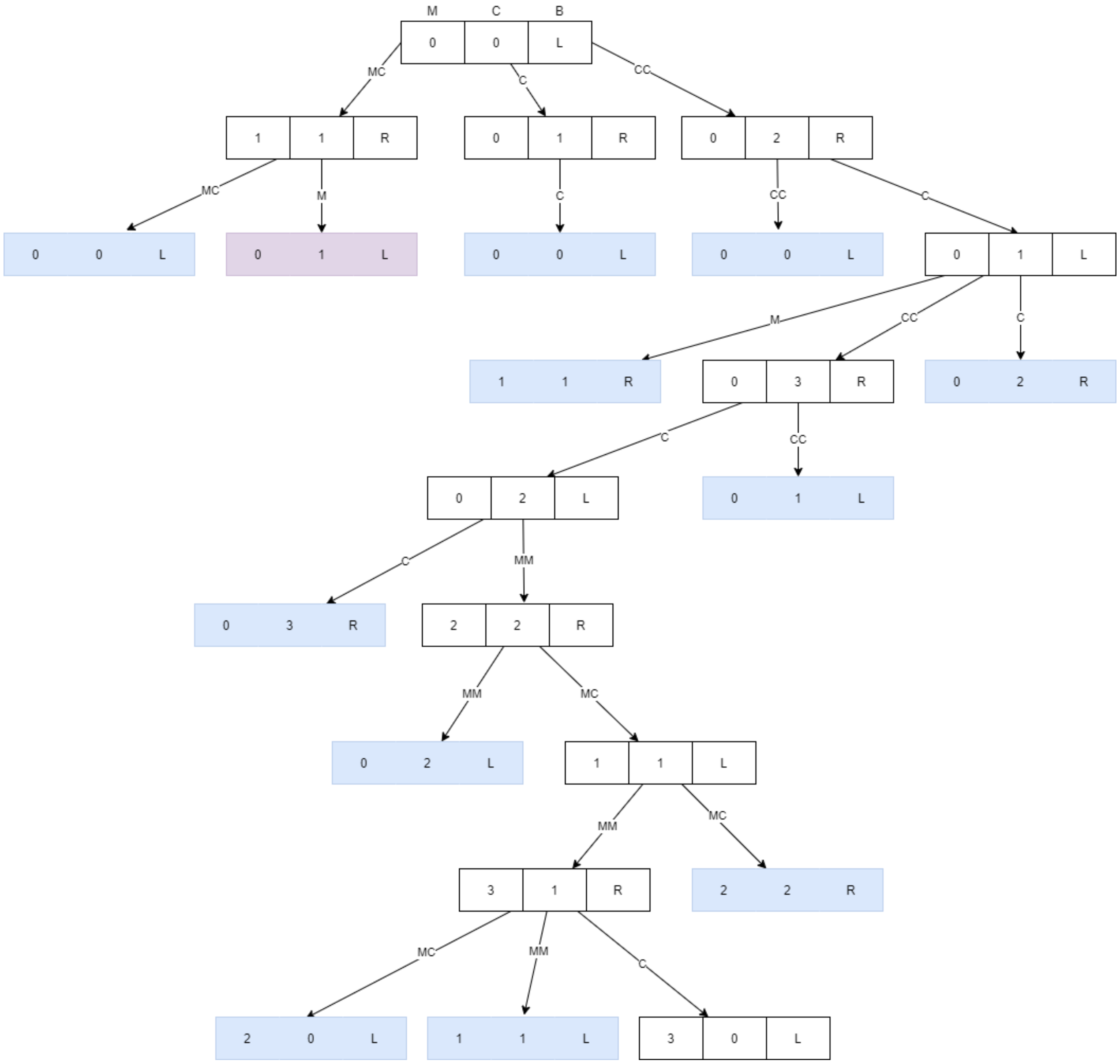
```
(6, <Node (1, 1, 0)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 2, 0), (0, 1, 0), (2, 2, 1), (0, 3, 1), (0, 0, 0), (0, 2, 1)]
5. Children :
[((0, 2, 0), 6), ((1, 1, 0), 6)]
```



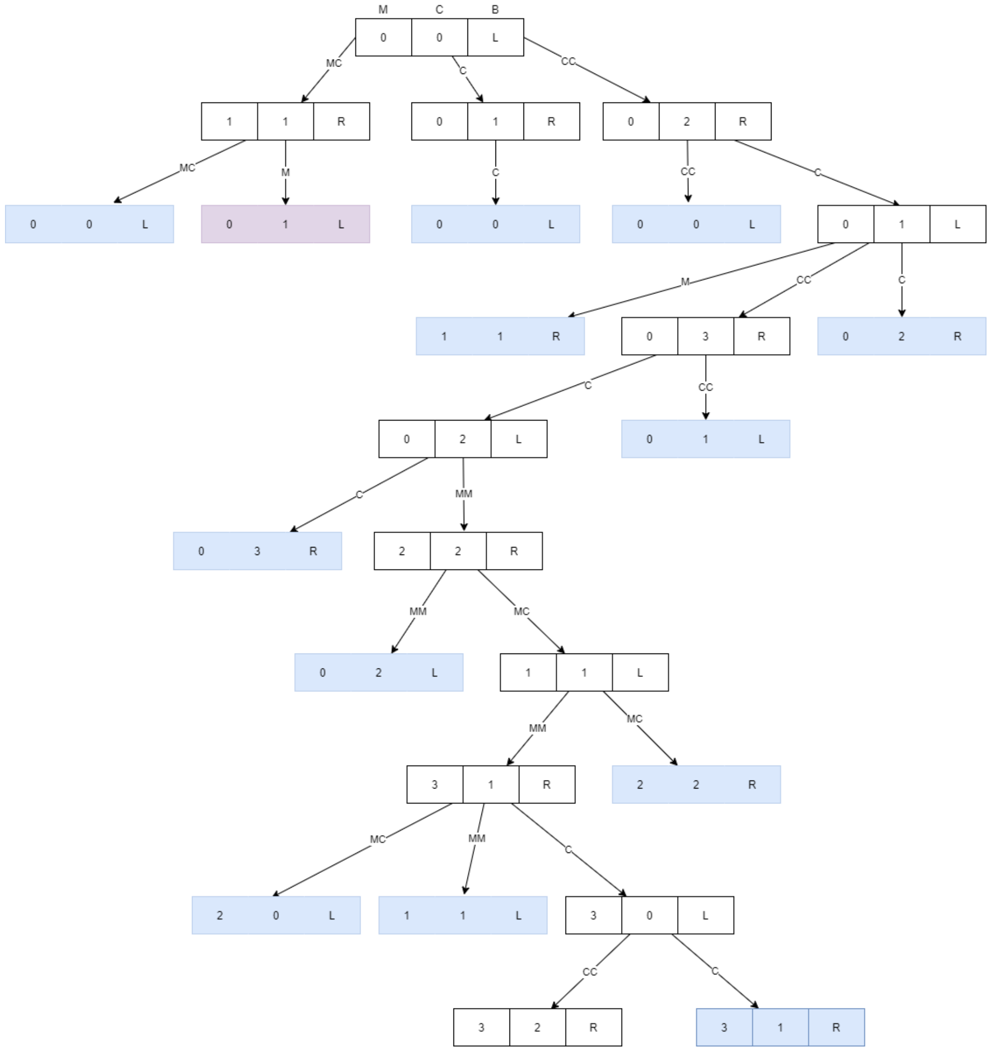
```
--------------------------------------------------------------------------------
UCS STEP : 9
1. Current Node : (1, 1, 0)
2. Evaluation function : 6
3. Frontier :
(7, <Node (3, 1, 1)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 2, 0), (1, 1, 0), (0, 1, 0), (2, 2, 1), (0, 3, 1), (0, 0, 0), (0, 2, 1)]
5. Children :
[((3, 1, 1), 7), ((2, 2, 1), 7)]
```



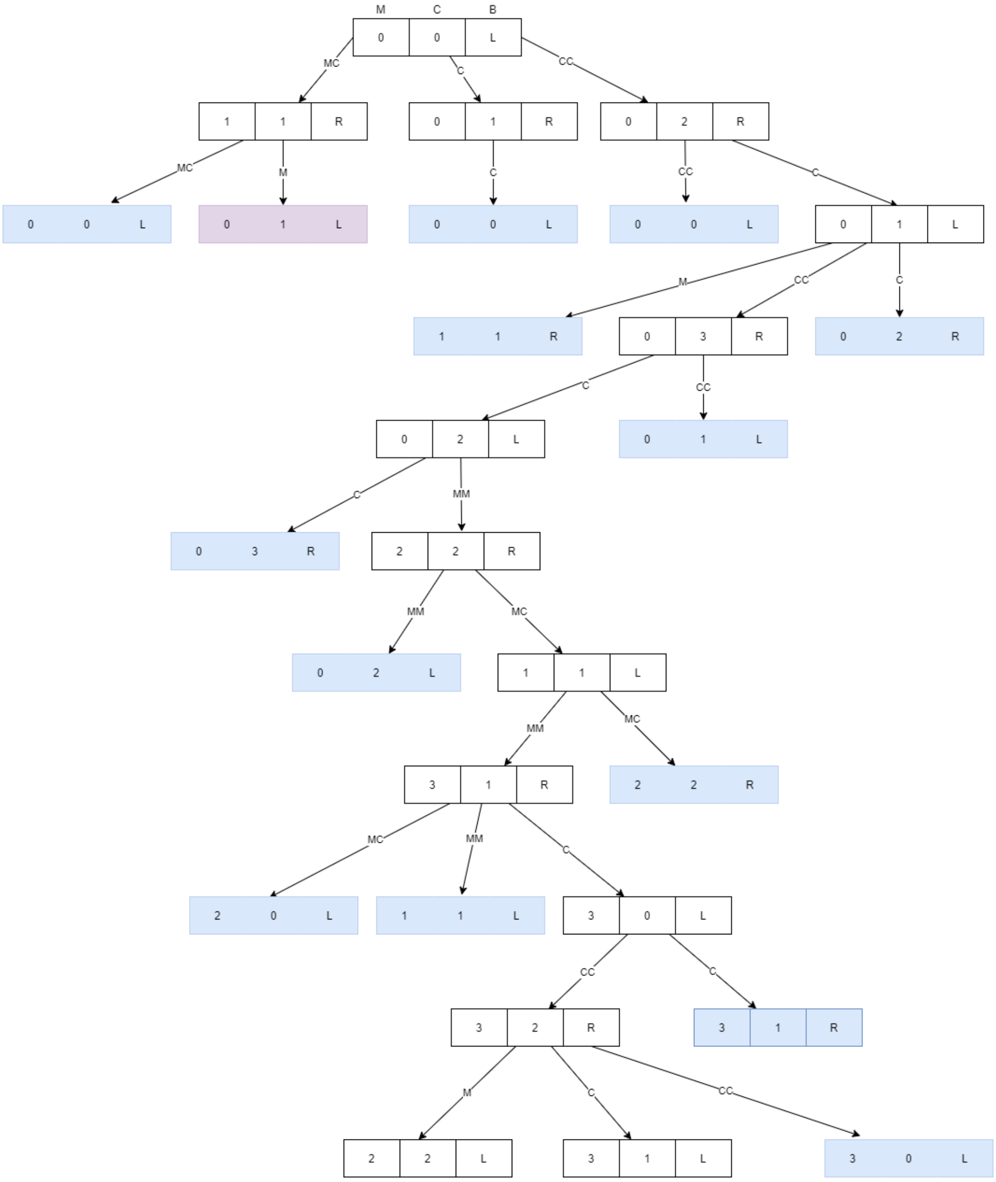
```
--------------------------------------------------------------------------------
UCS STEP : 10
1. Current Node : (3, 1, 1)
2. Evaluation function : 7
3. Frontier :
(8, <Node (3, 0, 0)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 2, 0), (1, 1, 0), (3, 1, 1), (0, 1, 0), (2, 2, 1), (0, 3, 1), (0, 0, 0), (0,
5. Children :
[((3, 0, 0), 8), ((1, 1, 0), 8)]
```



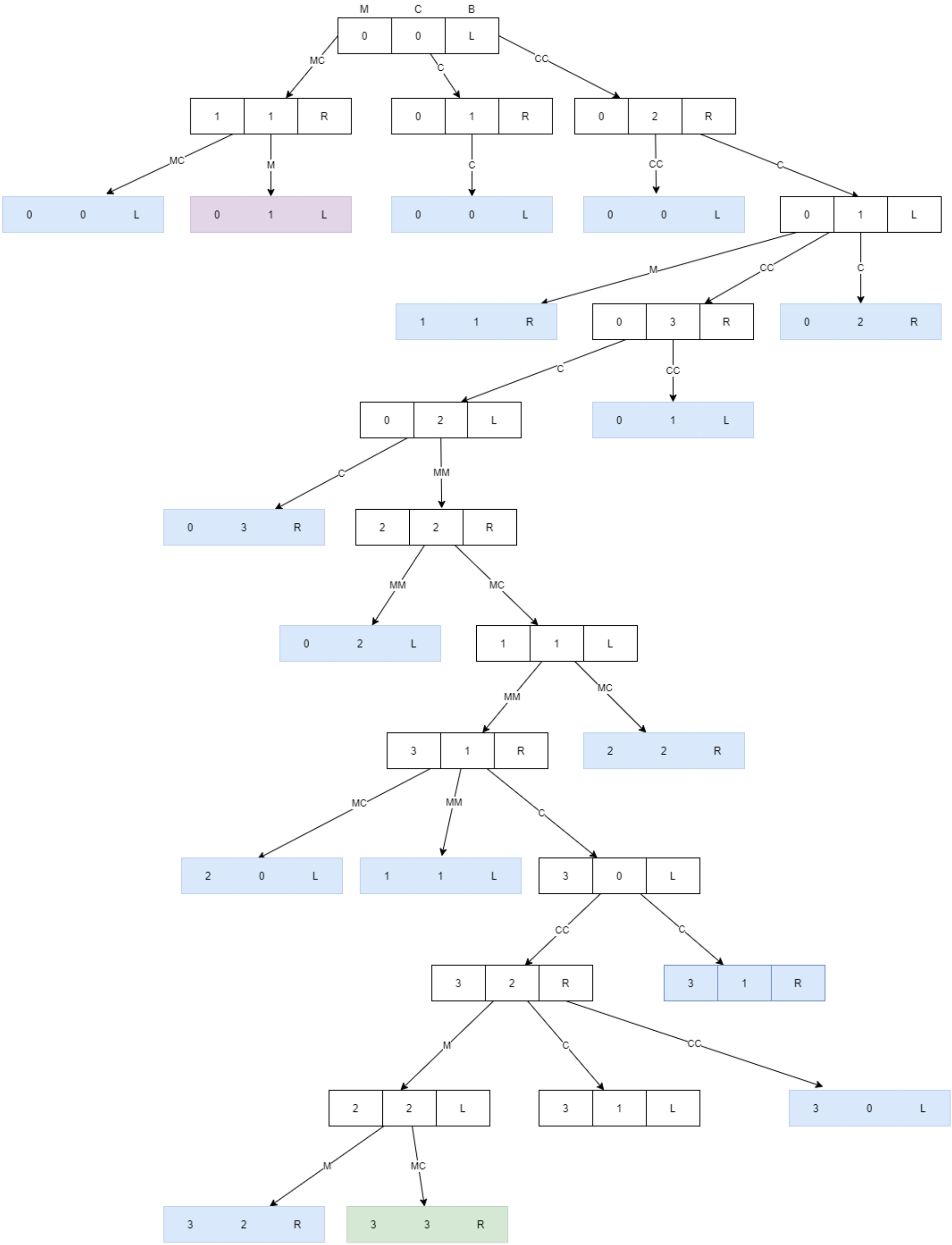
```
---------------------------------------------------------------------------------
UCS STEP : 11
1. Current Node : (3, 0, 0)
2. Evaluation function : 8
3. Frontier :
(9, <Node (3, 2, 1)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 2, 0), (1, 1, 0), (3, 1, 1), (0, 1, 0), (2, 2, 1), (0, 3, 1), (3, 0, 0), (0,
5. Children :
[((3, 1, 1), 9), ((3, 2, 1), 9)]
```



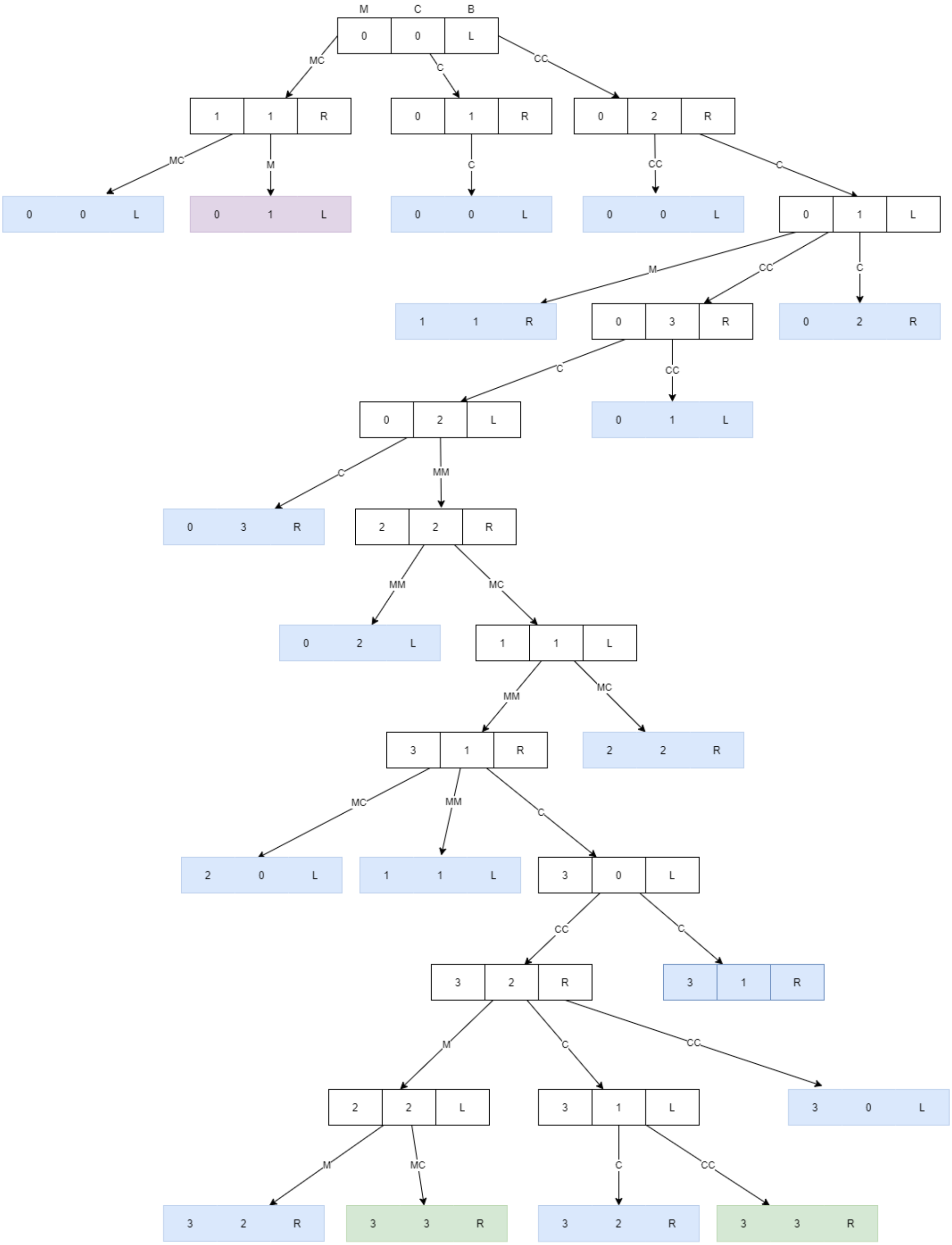
```
--------------------------------------------------------------------------------
UCS STEP : 12
1. Current Node : (3, 2, 1)
2. Evaluation function : 9
3. Frontier :
(10, <Node (2, 2, 0)>)
(10, <Node (3, 1, 0)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 2, 0), (1, 1, 0), (3, 1, 1), (0, 1, 0), (3, 2, 1), (2, 2, 1), (0, 3, 1), (3,
5. Children :
[((3, 1, 0), 10), ((2, 2, 0), 10), ((3, 0, 0), 10)]
```



```
--------------------------------------------------------------------------------
UCS STEP : 13
1. Current Node : (2, 2, 0)
2. Evaluation function : 10
3. Frontier :
(10, <Node (3, 1, 0)>)
(11, <Node (3, 3, 1)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 2, 0), (1, 1, 0), (3, 1, 1), (0, 1, 0), (3, 2, 1), (2, 2, 1), (0, 3, 1), (2,
5. Children :
[((3, 2, 1), 11), ((3, 3, 1), 11)]
```



```
--------------------------------------------------------------------------------
UCS STEP : 14
1. Current Node : (3, 1, 0)
2. Evaluation function : 10
3. Frontier :
(11, <Node (3, 3, 1)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 2, 0), (1, 1, 0), (3, 1, 1), (0, 1, 0), (3, 2, 1), (2, 2, 1), (3, 1, 0), (0,
5. Children :
[((3, 2, 1), 11), ((3, 3, 1), 11)]
```


M C B
0 0 L
MC
C
CC
1 1 R
0 1 R
0 2 R
MC
M
C
CC
C
0 0 L
0 1 L
0 0 L
0 0 L
0 1 L
M
CC
C
1 1 R
0 3 R
0 2 R
C
CC
0 2 L
0 1 L
C
MM
0 3 R
2 2 R
MM
MC
0 2 L
1 1 L
MM
MC
3 1 R
2 2 R
MC
MM
C
2 0 L
1 1 L
3 0 L
CC
C
3 2 R
3 1 R
M
C
CC
2 2 L
3 1 L
3 0 L
M
MC
C
CC
3 2 R
3 3 R
3 2 R
3 3 R

2.3 PROGRAMMING PART

2.3.1 Solution using AIMA code base

Use the implementations of the search strategies available in the aima code; e.g. https://github.com/aimacode/aima-java to write the program that will search for the solution to Missionaries and Cannibals Problem. (10 points)

SOLUTION

1. We formulate the problem with the help of a subclass of class Problem
2. Create an instance of the problem with initial and goal states
3. Using breadth first tree search strategy from the AIMA code base we find the solution

Code for problem class

```
class MissionaryCannibalProblem(Problem):
    def __init__(self, initial, goal):
        Problem.__init__(self, initial, goal)
        self.state = initial

    def actions(self, state):
        """
        State : (M,C,B)
        1. Boat on the left side [Indicated by B=0]
            1. Number of missionaries on the left side (0,1,2,3) [Indicated by ’3-M’]
                1. Number of cannibals on the left side (0,1,2,3) [Indicated by ’3-C’]
        2. Boat on the right side [Indicated by B=1]
            1. Number of missionaries on the right side (0,1,2,3) [Indicated by ’M’]
                1. Number of cannibals on the right side (0,1,2,3) [Indicated by ’C’]

        Based on M, C, B each state has a set of legal and illegal actions available.
        We return the legal actions available for each state
        """
        if state[2] == 0:  # Boat on the left
            if state[0] == 0:  # 3 missionaries on the left
                if state[1] == 0:
                    return [’MC’, ’C’, ’CC’]
                elif state[1] == 1:
                    return [’M’, ’C’, ’CC’]
                elif state[1] == 2:
                    return [’MM’, ’C’]
                else:
                    return []
            elif state[0] == 1:  # 2 missionaries on the left
                if state[1] == 1:
                    return [’MC’, ’MM’]
                else:
                    return []
            elif state[0] == 2:  # 1 missionary on the left
                if state[1] == 2:
                    return [’MC’, ’M’]
                else:
                    return []
            else:  # 0 missionary on the left
                if state[1] <= 1:
                    return [’C’, ’CC’]
                elif state[1] == 2:
                    return [’C’]
                else:
                    return []
        elif state[2] == 1:  # Boat on the right side
            if state[0] == 0:  # 3 missionaries on the left
                if state[1] == 0:
                    return []
                elif state[2] == 1:
                    return [’C’]
                else:
                    return [’C’, ’CC’]
            elif state[0] == 1:  # 2 missionaries on the left
                if state[1] == 1:
                    return [’M’, ’MC’]
                else:
                    return []
            elif state[0] == 2:  # 1 missionary on the left
                if state[1] == 2:
                    return ['MM', 'MC']
                else:
                    return []
            else:  # No missionary on the left
                if state[1] == 1:
                    return ['MM', 'C']
                elif state[1] == 2:
                    return ['M', 'C', 'CC']
                else:
                    return []
        else:
            return []

    def result(self, state, action):
        state = list(state)
        if state[2] == 0:  # Boat on the left, (For actions we add them)
            if action == 'M':
                state[0] = state[0] + 1
                state[2] = 1
            elif action == 'MM':
                state[0] = state[0] + 2
                state[2] = 1
            elif action == 'MC':
                state[0] = state[0] + 1
                state[1] = state[1] + 1
                state[2] = 1
            elif action == 'C':
                state[1] = state[1] + 1
                state[2] = 1
            else:
                state[1] = state[1] + 2
                state[2] = 1
        elif state[2] == 1:  # Boat on the right, (For actions we subtract them)
            if action == 'M':
                state[0] = state[0] - 1
                state[2] = 0
            elif action == 'MM':
                state[0] = state[0] - 2
                state[2] = 0
            elif action == 'MC':
                state[0] = state[0] - 1
                state[1] = state[1] - 1
                state[2] = 0
            elif action == 'C':
                state[1] = state[1] - 1
                state[2] = 0
            else:
                state[1] = state[1] - 2
                state[2] = 0

        state = tuple(state)
        self.state = state
        return self.state
```

Code for obtained Path

```
"""
---------------------------------------------------------------------------------
1. Using the implementation of breadth first tree search to find solution
---------------------------------------------------------------------------------
"""

bread_first_tree_search_solution = breadth_first_tree_search(missionary_cannibal_problem)
for node in bread_first_tree_search_solution.path():
    print(node.state)
```

Obtained Path

$(0,\ 0,\ 0) \implies MC, (1, 1, 1) \implies M, (0, 1, 0) \implies CC, (0, 3, 1) \implies C, (0, 2, 0) \implies MM, (2, 2, 1) \implies MC, (1, 1, 0) \implies MM, (3, 1, 1) \implies C, (3, 0, 0) \implies CC, (3, 2, 1) \implies M, (2, 2, 0) \implies MC, (3, 3, 1)$

2.3.2 Path to solution generated by code

Provide the path to the solution generated by your code, indicating each state it has visited when using: (a) uniform-cost search; (b) iterative deepening search; (c) greedy best-first search; (d) A* search and (e) recursive best-first search (1 point/strategy). Describe clearly how you have implemented each state in the search space. (5 points)

For all the problems considered below each state is defined as tuple (M,C,B) and above Problem class is used for finding the path in each strategy

1. UNIFORM COST SEARCH

```python
def uniform_cost_search_modified(problem, g):
    g = memoize(g, 'g')
    node = Node(problem.initial)
    frontier = PriorityQueue('min', g)
    frontier.append(node)
    explored = set()
    counter = 0
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        children = set()
        for child in node.expand(problem):
            children.add((child.state, g(child)))
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                if g(child) < frontier[child]:
                    del frontier[child]
                    frontier.append(child)
        counter += 1
        print_strategy_items('UCS', counter, node, g, explored, frontier, children)
    return None


def print_strategy_items(strategy, counter, node, f, explored, frontier, children):
    print('--------------------------------------------------------------------------------')
    print(strategy + ' STEP : ' + str(counter))
    print('1. Current Node : ' + str(node.state))
    print('2. Evaluation function : ' + str(f(node)))
    print('3. Frontier : ')
    frontier.print_items()
    print('4. Explored : ')
    print(list({k: 1 for k in explored}))
    print('5. Children : ')
    print(list({k: 1 for k in children}))

# CALLING THE ABOVE MODIFIED FUNCTION
uniform_cost_search_solution = uniform_cost_search_modified(missionary_cannibal_problem, lambda n: n.path_cost)

# PRINT THE GENERATED PATH
for node in bread_first_tree_search_solution.path():
    print(node.action, node.state, node.path_cost)
```

Obtained path for UCS strategy

Format : [Action that generated node, Node, Path cost]

$[(0,\ 0,\ 0),\ 0] \implies [MC,(1,1,1),1] \implies [M,(0,1,0),2] \implies [CC,(0,3,1),3] \implies [C,(0,2,0),4] \implies [MM,(2,2,1),5] \implies [MC,(1,1,0),6] \implies [MM,(3,1,1),7] \implies [C,(3,0,0),8] \implies [CC,(3,2,1),9] \implies [M,(2,2,0),10] \implies [MC,(3,3,1),11]$

2. ITERATIVE DEEPENING SEARCH

Depth limited Tree search is used in this, so explored set is not considered

```python
"""
Considered Tree search paradigm, So explored set is not used.
"""
def depth_limited_search_modified(problem, limit=50):
    frontier = [(Node(problem.initial))]  # Stack
    counter = 0
    while frontier:
        node = frontier.pop()
        children = set()
        counter += 1
        if problem.goal_test(node.state):
            print_dls(limit, counter, node, frontier, children)
            return node
        elif node.path_cost == limit:
            print_dls(limit, counter, node, frontier, children)
            continue
        for child in node.expand(problem):
            children.add(child.state)
            if child not in frontier:
                frontier.append(child)
        print_dls(limit, counter, node, frontier, children)
    return 'cutoff'


def print_dls(depth, counter, node, frontier, children=set()):
    print('--------------------------------------------------------------------------------')
    print('Depth : ' + str(depth) + ', Step : ' + str(counter))
    print('1. Current Node : ' + str(node.state))
    print('2. Frontier : ')
    print("".join([str(x) for x in frontier]))
    print('3. Children : ')
    print(list({k: 1 for k in children})) if len(children) > 0 else print('[]')


def iterative_deepening_search_modified(problem):
    for depth in range(sys.maxsize):
        result = depth_limited_search_modified(problem, depth)
        if result != 'cutoff':
            return result

# CALLING MODIFIED ITERATIVE DEEPENING SEARCH METHOD
iterative_deepening_search_solution = iterative_deepening_search_modified(
    missionary_cannibal_problem)

# PRINT THE GENERATED PATH
for node in iterative_deepening_search_solution.path():
    print(node.action, node.state)
```

Obtained path for Iterative Deepening Strategy

Format : [Action that generated node, Node]

$[(0,\ 0,\ 0)] \implies [CC(0,2,1)] \implies [C(0,1,0)] \implies [CC(0,3,1)] \implies [C(0,2,0)] \implies [MM(2,2,1)] \implies [MC(1,1,0)] \implies [MM(3,1,1)] \implies [C(3,0,0)] \implies [CC(3,2,1)] \implies [C(3,1,0)] \implies [CC(3,3,1)]$

Check Heuristic Consistency

From here as we use graph search strategy with explored set. We need to verify whether the heuristic used is consistent or not, in order to find the optimal solution.

```python
"""
Heuristic considered for the problem
"""

def h(self, node):
    """
    Here we define the following heuristic function:
    h(M,C,B) = ceil([(3-M) + (3-B)]/(Boat Capacity))
    [This is a consistent heuristic (Optimal for graph search strategies)]
    """
    current_state = node.state
    return math.ceil(((3 - current_state[0]) + (3 - current_state[1]))/2)
```

To verify the consistency of the heuristic function we use the code from EXTRA CREDIT section of (Problem 3). After verification we find that the heuristic considered above is indeed consistent.

3. GREEDY BEST FIRST SEARCH

```python
def best_first_search_modified(strategy, problem, f, display=False):
    f = memoize(f, 'f')
    node = Node(problem.initial)
    frontier = PriorityQueue('min', f)
    frontier.append(node)
    explored = set()
    counter = 0
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            if display:
                print(len(explored), "paths have been expanded and", len(frontier), "paths remain in the frontier")
            return node
        explored.add(node.state)
        children = set()
        for child in node.expand(problem):
            children.add((child.state, f(child)))
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                if f(child) < frontier[child]:
                    del frontier[child]
                    frontier.append(child)
        counter += 1
        print_strategy_items(strategy, counter, node, f, explored, frontier, children)
    return None

# CALLING GREEDY BEST FIRST SEARCH METHOD
h = memoize(missionary_cannibal_problem.h, 'h')
greedy_best_first_search_solution = best_first_search_modified('GBFS', missionary_cannibal_problem, lambda n: h(n))

# PRINT THE GENERATED PATH
for node in greedy_best_first_search_solution.path():
    print(node.action, node.state, node.path_cost)
```

Obtained Path

Format : [Node, path cost]

$[(0,0,0)\ 0] \implies [CC(0,2,1)1] \implies [C(0,1,0)2] \implies [CC(0,3,1)3] \implies [C(0,2,0)4] \implies [MM(2,2,1)5] \implies [MC(1,1,0)6] \implies [MM(3,1,1)7] \implies [C(3,0,0)8] \implies [CC(3,2,1)9] \implies [M(2,2,0)10] \implies [MC(3,3,1)11]$

4. A* SEARCH

```
# CALLING A* SEARCH METHOD
h = memoize(missionary_cannibal_problem.h, 'h')
a_star_search_solution = best_first_search_modified('A*', missionary_cannibal_problem, lambda
    n: n.path_cost + h(n))

# PRINT THE GENERATED PATH
for node in a_star_search_solution.path():
    print(node.action, node.state, node.path_cost)
```

Obtained Path

Format : [Action, Node, Path cost]
[(0, 0, 0) 0] $\implies [CC(0,2,1)1] \implies [C(0,1,0)2] \implies [CC(0,3,1)3] \implies [C(0,2,0)4] \implies [MM(2,2,1)5] \implies [MC(1,1,0)6] \implies [MM(3,1,1)7] \implies [C(3,0,0)8] \implies [CC(3,2,1)9] \implies [M(2,2,0)10] \implies [MC(3,3,1)11]$

5. RECURSIVE BEST FIRST SEARCH

```python
def recursive_best_first_search_modified(problem, h=None):
    h = memoize(h or problem.h, 'h')
    node = Node(problem.initial)
    node.f = h(node)

    def RBFS(problem, node, flimit):
        if problem.goal_test(node.state):
            return node, 0  # (The second value is immaterial)
        successors = node.expand(problem)
        if len(successors) == 0:
            return None, np.inf
        for s in successors:
            s.f = max(s.path_cost + h(s), node.f)
        while True:
            # Order by lowest f value
            successors.sort(key=lambda x: x.f)
            best = successors[0]
            if best.f > flimit:
                print_rbfs(flimit, best, np.inf, node.state, node.parent.state)
                return None, best.f
            if len(successors) > 1:
                alternative = successors[1].f
            else:
                alternative = np.inf
            print_rbfs(flimit, best, alternative, node.state, best.state)
            result, best.f = RBFS(problem, best, min(flimit, alternative))
            if result is not None:
                return result, best.f

    result, bestf = RBFS(problem, node, np.inf)
    return result

def print_rbfs(flimit, best, alternative, current_city, next_city):
    print('------------------------------------------------------------')
    print('1. f_limit: ' + str(flimit))
    print('2. best: ' + str(best.f))
    print('3. alternative: ' + str(alternative))
    print('4. current-city: ' + str(current_city))
    print('5. next-city: ' + str(next_city))
    print('------------------------------------------------------------')
    return


# CALLING THE FUNCTION
rbfs_result = recursive_best_first_search_modified(missionary_cannibal_problem)

# PRINT THE GENERATED PATH
for node in rbfs_result.path():
    print(node.action, node.state, node.path_cost)
```

Obtained Path

Format : [Action, Node, Path cost]

$[(0,0,0)\ 0] \implies [CC(0,2,1)1] \implies [C(0,1,0)2] \implies [CC(0,3,1)3] \implies [C(0,2,0)4] \implies [MM(2,2,1)5] \implies [MC(1,1,0)6] \implies [MM(3,1,1)7] \implies [C(3,0,0)8] \implies [CC(3,2,1)9] \implies [M(2,2,0)10] \implies [MC(3,3,1)11]$

2.4 EXTRA CREDIT

1. UNIFORM COST SEARCH

```
    --------------------------------------------------------------------------------
UCS STEP : 1
1. Current Node : (0, 0, 0)
2. Evaluation function : 0
3. Frontier :
(1, <Node (0, 1, 1)>)
(1, <Node (1, 1, 1)>)
(1, <Node (0, 2, 1)>)
4. Explored :
[(0, 0, 0)]
5. Children :
[((0, 2, 1), 1), ((0, 1, 1), 1), ((1, 1, 1), 1)]
--------------------------------------------------------------------------------
UCS STEP : 2
1. Current Node : (0, 1, 1)
2. Evaluation function : 1
3. Frontier :
(1, <Node (0, 2, 1)>)
(1, <Node (1, 1, 1)>)
4. Explored :
[(0, 1, 1), (0, 0, 0)]
5. Children :
[((0, 0, 0), 2)]
--------------------------------------------------------------------------------
UCS STEP : 3
1. Current Node : (0, 2, 1)
2. Evaluation function : 1
3. Frontier :
(1, <Node (1, 1, 1)>)
(2, <Node (0, 1, 0)>)
4. Explored :
[(0, 1, 1), (0, 2, 1), (0, 0, 0)]
5. Children :
[((0, 1, 0), 2)]
--------------------------------------------------------------------------------
UCS STEP : 4
1. Current Node : (1, 1, 1)
2. Evaluation function : 1
3. Frontier :
(2, <Node (0, 1, 0)>)
4. Explored :
[(0, 1, 1), (0, 2, 1), (0, 0, 0), (1, 1, 1)]
5. Children :
[((0, 0, 0), 2), ((0, 1, 0), 2)]
--------------------------------------------------------------------------------
UCS STEP : 5
1. Current Node : (0, 1, 0)
2. Evaluation function : 2
3. Frontier :
(3, <Node (0, 3, 1)>)
4. Explored :
[(0, 1, 1), (1, 1, 1), (0, 1, 0), (0, 0, 0), (0, 2, 1)]
5. Children :
[((0, 3, 1), 3), ((1, 1, 1), 3), ((0, 2, 1), 3)]
--------------------------------------------------------------------------------
```

Rest of the steps included at the end

2. ITERATIVE DEEPENING SEARCH

Used Tree Search for depth limited search, So there is no explored node section. Explored set is not appropriate here as a node can be reached from different paths with different path costs. If it is approached at higher depth initially when it is equal to limit that node is not expanded but when considered from another path where depth is much lesser than the initial one, it is expanded and children of node are considered

```
        --------------------------------------------------------------------------------
Depth : 0, Step : 1
1. Current Node : (0, 0, 0)
2. Frontier :
3. Children : []
--------------------------------------------------------------------------------
Depth : 1, Step : 1
1. Current Node : (0, 0, 0)
2. Frontier : <Node (1, 1, 1)><Node (0, 1, 1)><Node (0, 2, 1)>
3. Children : [(0, 1, 1), (1, 1, 1), (0, 2, 1)]
--------------------------------------------------------------------------------
Depth : 1, Step : 2
1. Current Node : (0, 2, 1)
2. Frontier : <Node (1, 1, 1)><Node (0, 1, 1)>
3. Children : []
--------------------------------------------------------------------------------
Depth : 1, Step : 3
1. Current Node : (0, 1, 1)
2. Frontier : <Node (1, 1, 1)>
3. Children : []
--------------------------------------------------------------------------------
Depth : 1, Step : 4
1. Current Node : (1, 1, 1)
2. Frontier :
3. Children : []
--------------------------------------------------------------------------------
Depth : 2, Step : 1
1. Current Node : (0, 0, 0)
2. Frontier : <Node (1, 1, 1)><Node (0, 1, 1)><Node (0, 2, 1)>
3. Children : [(0, 1, 1), (1, 1, 1), (0, 2, 1)]
--------------------------------------------------------------------------------
Depth : 2, Step : 2
1. Current Node : (0, 2, 1)
2. Frontier : <Node (1, 1, 1)><Node (0, 1, 1)><Node (0, 1, 0)>
3. Children : [(0, 1, 0)]
--------------------------------------------------------------------------------
Depth : 2, Step : 3
1. Current Node : (0, 1, 0)
2. Frontier : <Node (1, 1, 1)><Node (0, 1, 1)>
3. Children : []
--------------------------------------------------------------------------------
```

Rest of the steps included at the end

3. GREEDY BEST FIRST SEARCH

```
        --------------------------------------------------------------------------------
GBFS STEP : 1
1. Current Node : (0, 0, 0)
2. Evaluation function : 3
3. Frontier : (2, <Node (0, 2, 1)>), (3, <Node (0, 1, 1)>), (2, <Node (1, 1, 1)>)
4. Explored : [(0, 0, 0)]
5. Children : [((0, 2, 1), 2), ((0, 1, 1), 3), ((1, 1, 1), 2)]
--------------------------------------------------------------------------------
GBFS STEP : 2
1. Current Node : (0, 2, 1)
2. Evaluation function : 2
3. Frontier : (2, <Node (1, 1, 1)>), (3, <Node (0, 1, 1)>), (3, <Node (0, 1, 0)>)
4. Explored : [(0, 2, 1), (0, 0, 0)]
5. Children : [((0, 1, 0), 3)]
--------------------------------------------------------------------------------
GBFS STEP : 3
1. Current Node : (1, 1, 1)
2. Evaluation function : 2
3. Frontier :
(3, <Node (0, 1, 0)>)
(3, <Node (0, 1, 1)>)
4. Explored : [(1, 1, 1), (0, 2, 1), (0, 0, 0)]
5. Children : [((0, 1, 0), 3), ((0, 0, 0), 3)]
--------------------------------------------------------------------------------
GBFS STEP : 4
1. Current Node : (0, 1, 0)
2. Evaluation function : 3
3. Frontier :
(2, <Node (0, 3, 1)>)
(3, <Node (0, 1, 1)>)
4. Explored : [(1, 1, 1), (0, 2, 1), (0, 0, 0), (0, 1, 0)]
5. Children : [((0, 2, 1), 2), ((0, 3, 1), 2), ((1, 1, 1), 2)]
--------------------------------------------------------------------------------
GBFS STEP : 5
1. Current Node : (0, 3, 1)
2. Evaluation function : 2
3. Frontier :
(2, <Node (0, 2, 0)>)
(3, <Node (0, 1, 1)>)
4. Explored : [(0, 2, 1), (0, 1, 0), (0, 3, 1), (0, 0, 0), (1, 1, 1)]
5. Children : [((0, 2, 0), 2)]
--------------------------------------------------------------------------------
GBFS STEP : 6
1. Current Node : (0, 2, 0)
2. Evaluation function : 2
3. Frontier :
(1, <Node (2, 2, 1)>)
(3, <Node (0, 1, 1)>)
4. Explored : [(0, 2, 0), (0, 2, 1), (0, 1, 0), (0, 3, 1), (0, 0, 0), (1, 1, 1)]
5. Children : [((2, 2, 1), 1), ((0, 3, 1), 2)]
--------------------------------------------------------------------------------
```

Rest of the steps included at the end

4. A* SEARCH

```
A* STEP : 1
1. Current Node : (0, 0, 0)
2. Evaluation function : 3
3. Frontier : (3, <Node (0, 2, 1)>), (4, <Node (0, 1, 1)>), (3, <Node (1, 1, 1)>)
4. Explored : [(0, 0, 0)]
5. Children : [((0, 2, 1), 3), ((1, 1, 1), 3), ((0, 1, 1), 4)]
--------------------------------------------------------------------------------
A* STEP : 2
1. Current Node : (0, 2, 1)
2. Evaluation function : 3
3. Frontier : (3, <Node (1, 1, 1)>), (4, <Node (0, 1, 1)>), (5, <Node (0, 1, 0)>)
4. Explored : [(0, 2, 1), (0, 0, 0)]
5. Children : [((0, 1, 0), 5)]
--------------------------------------------------------------------------------
A* STEP : 3
1. Current Node : (1, 1, 1)
2. Evaluation function : 3
3. Frontier : (4, <Node (0, 1, 1)>), (5, <Node (0, 1, 0)>)
4. Explored : [(1, 1, 1), (0, 2, 1), (0, 0, 0)]
5. Children : [((0, 1, 0), 5), ((0, 0, 0), 5)]
--------------------------------------------------------------------------------
A* STEP : 4
1. Current Node : (0, 1, 1)
2. Evaluation function : 4
3. Frontier : (5, <Node (0, 1, 0)>)
4. Explored : [(1, 1, 1), (0, 2, 1), (0, 0, 0), (0, 1, 1)]
5. Children : [((0, 0, 0), 5)]
--------------------------------------------------------------------------------
A* STEP : 5
1. Current Node : (0, 1, 0)
2. Evaluation function : 5
3. Frontier : (5, <Node (0, 3, 1)>)
4. Explored : [(0, 1, 1), (0, 2, 1), (0, 1, 0), (0, 0, 0), (1, 1, 1)]
5. Children : [((1, 1, 1), 5), ((0, 3, 1), 5), ((0, 2, 1), 5)]
--------------------------------------------------------------------------------
A* STEP : 6
1. Current Node : (0, 3, 1)
2. Evaluation function : 5
3. Frontier : (6, <Node (0, 2, 0)>)
4. Explored : [(0, 1, 1), (0, 2, 1), (0, 1, 0), (0, 3, 1), (0, 0, 0), (1, 1, 1)]
5. Children : [((0, 2, 0), 6)]
--------------------------------------------------------------------------------
```

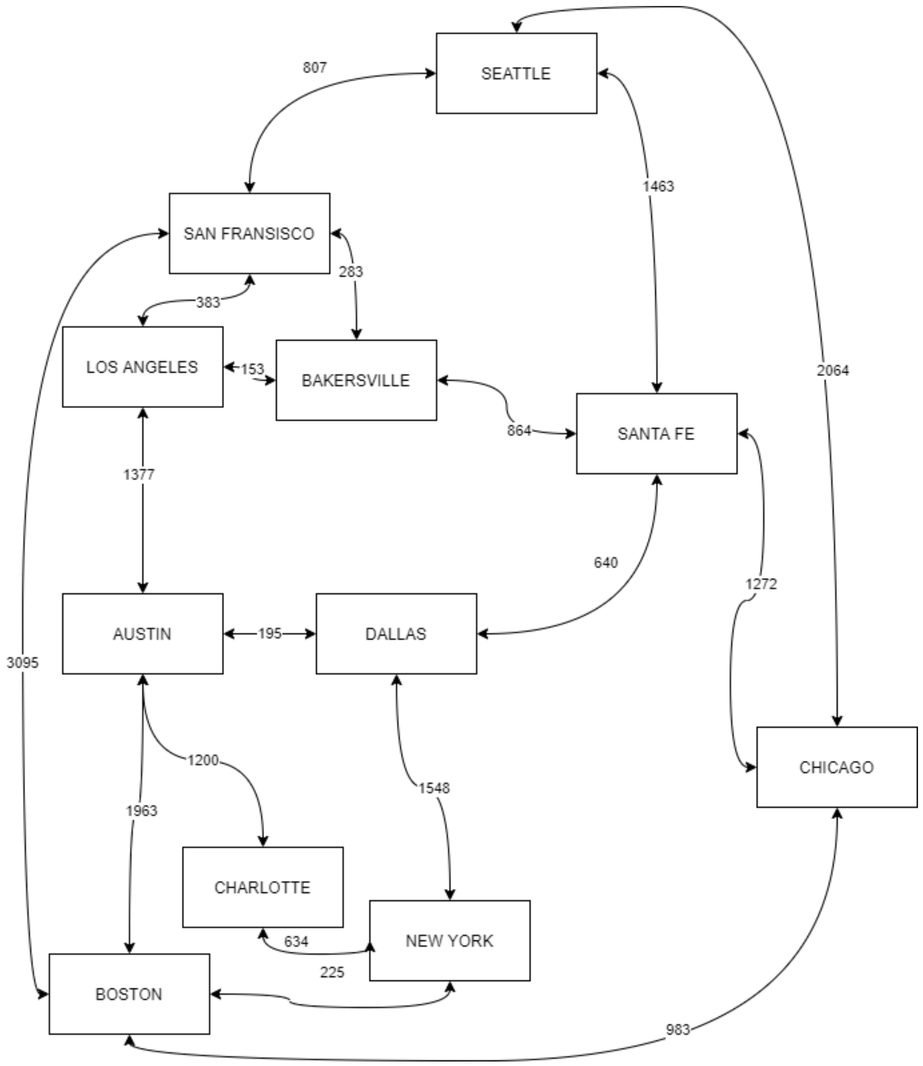
Rest of the steps included at the end

5. RECURSIVE BEST FIRST SEARCH

```
        ------------------------------------------------------------
1. f_limit: inf
2. best: 3
3. alternative: 3
4. current node: (0, 0, 0)
5. next node: (1, 1, 1)
------------------------------------------------------------
1. f_limit: 3
2. best: 5
3. alternative: inf
4. current node: (1, 1, 1)
5. next node: (0, 0, 0)
------------------------------------------------------------
1. f_limit: inf
2. best: 3
3. alternative: 4
4. current node: (0, 0, 0)
5. next node: (0, 2, 1)
------------------------------------------------------------
1. f_limit: 4
2. best: 5
3. alternative: inf
4. current node: (0, 2, 1)
5. next node: (0, 0, 0)
------------------------------------------------------------
1. f_limit: inf
2. best: 4
3. alternative: 5
4. current node: (0, 0, 0)
5. next node: (0, 1, 1)
------------------------------------------------------------
1. f_limit: 5
2. best: 5
3. alternative: inf
4. current node: (0, 1, 1)
5. next node: (0, 0, 0)
------------------------------------------------------------
1. f_limit: 5
2. best: 5
3. alternative: 5
4. current node: (0, 0, 0)
5. next node: (1, 1, 1)
------------------------------------------------------------
1. f_limit: 5
2. best: 7
3. alternative: inf
4. current node: (1, 1, 1)
5. next node: (0, 0, 0)
------------------------------------------------------------
1. f_limit: 5
2. best: 5
3. alternative: 6
4. current node: (0, 0, 0)
5. next node: (0, 2, 1)
------------------------------------------------------------
```

Rest of the steps included at the end

3 SEARCHING FOR ROAD TRIPS



3.1 RBFS PROGRAM

You contemplate to search for a trip back to Dallas from Seattle, having access to a road map which should be implemented as a graph.
Use the aima code to find the solution and return a simulation of the RBFS strategy by generating automatically the following five values: (1) f_limit; (2) best; (3) alternative; (4) current-city; and (5) next-city for each node visited. (10 points)

<u>SOLUTION</u>

```
"""
Problem formulation for the Search Problem
"""

us_map = UndirectedGraph((dict(
    LosAngeles=dict(SanFrancisco=383, Austin=1377, Bakersville=153),
    SanFrancisco=dict(Bakersville=283, Seattle=807),
    Seattle=dict(SantaFe=1463, Chicago=2064),
    Bakersville=dict(SantaFe=864),
    Austin=dict(Dallas=195, Charlotte=1200),
    SantaFe=dict(Dallas=640),
    Boston=dict(Austin=1963, Chicago=983, SanFrancisco=3095),
    Dallas=dict(NewYork=1548),
    Charlotte=dict(NewYork=634),
    NewYork=dict(Boston=225),
    Chicago=dict(SantaFe=1272)
)))
us_map.locations = dict(
    Austin=(0, 182),
    Charlotte=(0, 929),
    SanFrancisco=(0, 1230),
    LosAngeles=(0, 1100),
    NewYork=(0, 1368),
    Chicago=(0, 800),
    Seattle=(0, 1670),
    SantaFe=(0, 560),
    Bakersville=(0, 1282),
    Boston=(0, 1551),
    Dallas=(0, 0)
)


class USRoadMapProblem(Problem):
    """The problem of searching a graph from one node to another."""

    def __init__(self, initial, goal, graph):
        super().__init__(initial, goal)
        self.graph = graph

    def actions(self, A):
        """The actions at a graph node are just its neighbors."""
        return list(self.graph.get(A).keys())

    def result(self, state, action):
        """The result of going to a neighbor is just that neighbor."""
        return action

    def path_cost(self, cost_so_far, A, action, B):
        return cost_so_far + (self.graph.get(A, B) or np.inf)

    def find_min_edge(self):
        """Find minimum value of edges."""
        m = np.inf
        for d in self.graph.graph_dict.values():
            local_min = min(d.values())
            m = min(m, local_min)
        return m

    def h(self, node):
        """h function is straight-line distance from a node’s state to goal."""
        locs = getattr(self.graph, ’locations’, None)
        if locs:
            if type(node) is str:
                return int(distance(locs[node], locs[self.goal]))
            return int(distance(locs[node.state], locs[self.goal]))
        else:
            return np.inf

    def value(self, state):
        pass
```

Simulation of RBFS using AIMA code generates the following :

1. STEP - 1

flimit	inf
best	2023
alternative	2037
current-city	Seattle
next-city	Santa Fe

2. STEP - 2

flimit	2037
best	2103
alternative	inf
current-city	Santa Fe
next-city	Seattle (Unwinding back)

3. STEP - 3

flimit	inf
best	2037
alternative	2103
current-city	Seattle
next-city	San Francisco

4. STEP - 4

flimit	2103
best	2290
alternative	inf
current-city	San Francisco
next-city	Seattle (Unwinding back)

5. STEP - 5

flimit	inf
best	2103
alternative	2290
current-city	Seattle
next-city	Santa Fe

6. STEP - 6

flimit	2290
best	2103
alternative	3535
current-city	Santa Fe
next-city	Dallas

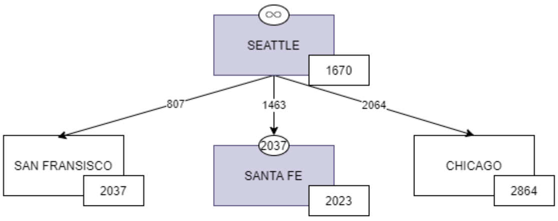
Once we reach the goal state the algorithm terminates. As we reach the goal state within flimit (f-value of best alternative path available from any ancestor of the current node) there is no other path that could reach goal state lesser than this.

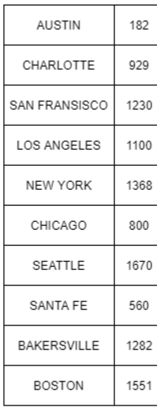
3.2 RBFS MANUAL

Find the same solution manually and specify how you have computed the following five values: (1) f_limit; (2) best; (3) alternative; (4) current-city; and (5) next-city for each node visited. Specify at each step the current node and the next node. (5 points)

SOLUTION

1. STEP - 1

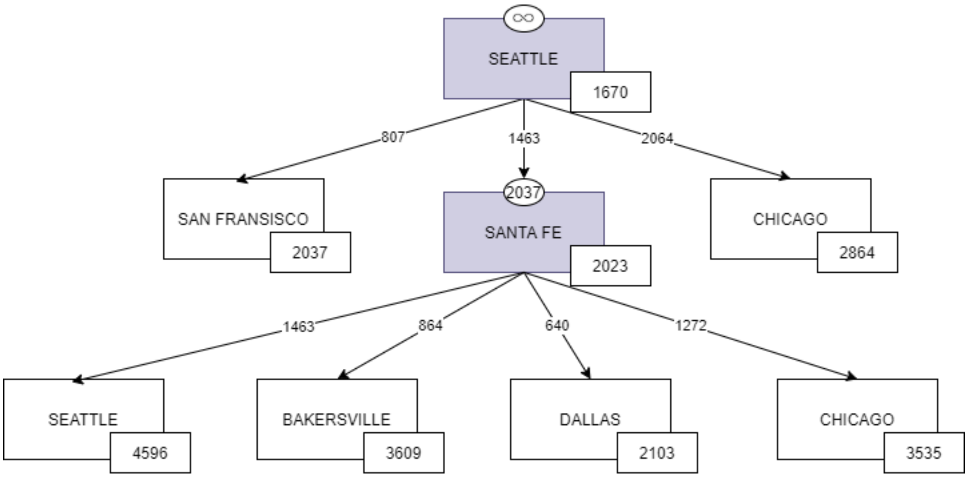




AUSTIN	182
CHARLOTTE	929
SAN FRANSISCO	1230
LOS ANGELES	1100
NEW YORK	1368
CHICAGO	800
SEATTLE	1670
SANTA FE	560
BAKERSVILLE	1282
BOSTON	1551

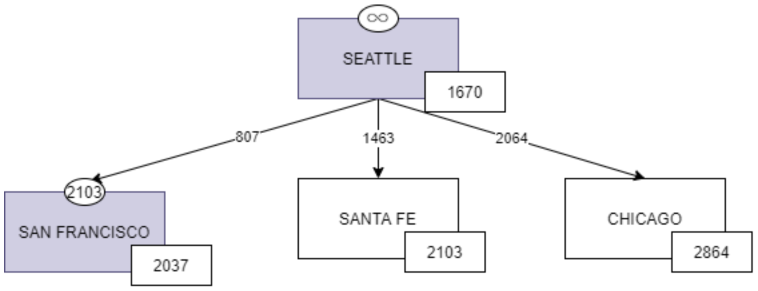
flimit	inf
best	2023
alternative	2037
current-city	Seattle
next-city	Santa Fe

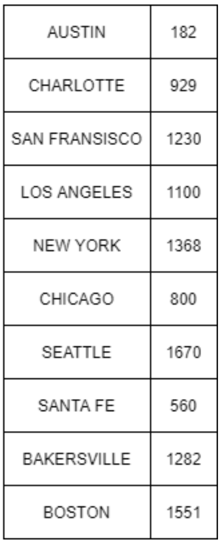
2. STEP - 2



flimit	2037
best	2103
alternative	inf
current-city	Santa Fe
next-city	Seattle (Unwinding back)

3. STEP - 3

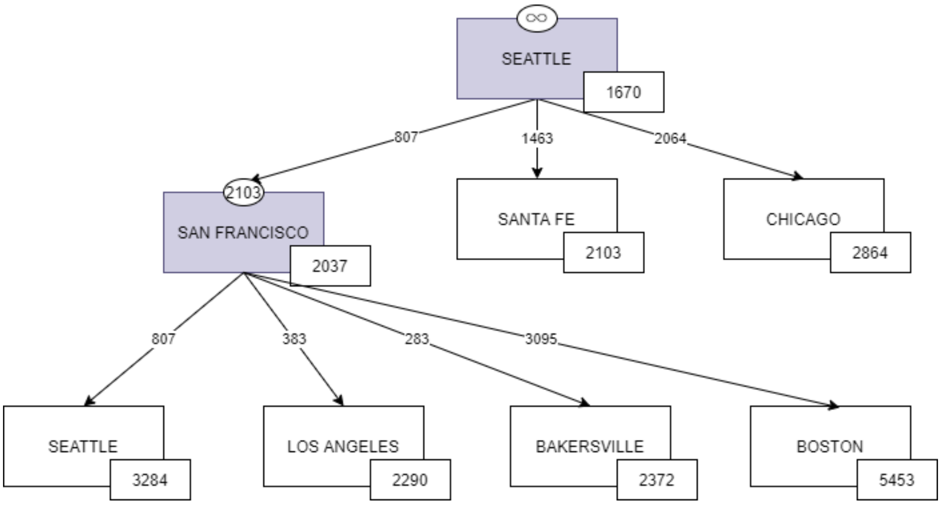




AUSTIN	182
CHARLOTTE	929
SAN FRANSISCO	1230
LOS ANGELES	1100
NEW YORK	1368
CHICAGO	800
SEATTLE	1670
SANTA FE	560
BAKERSVILLE	1282
BOSTON	1551

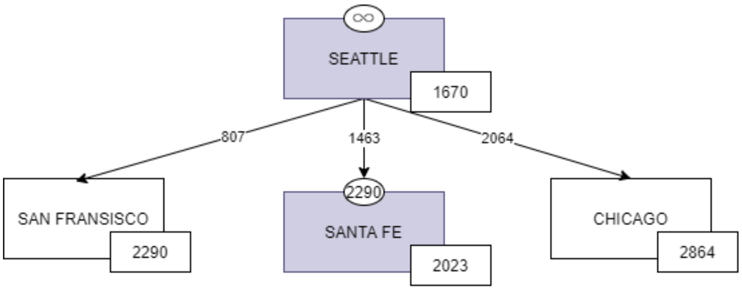
flimit	inf
best	2037
alternative	2103
current-city	Seattle
next-city	San Francisco

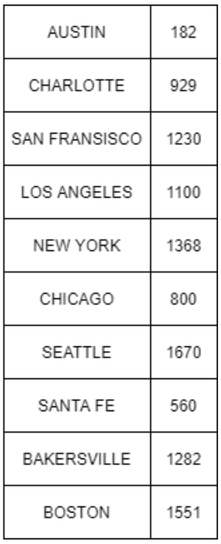
4. STEP - 4



flimit	2103
best	2290
alternative	inf
current-city	San Francisco
next-city	Seattle (Unwinding back)

5. STEP - 5

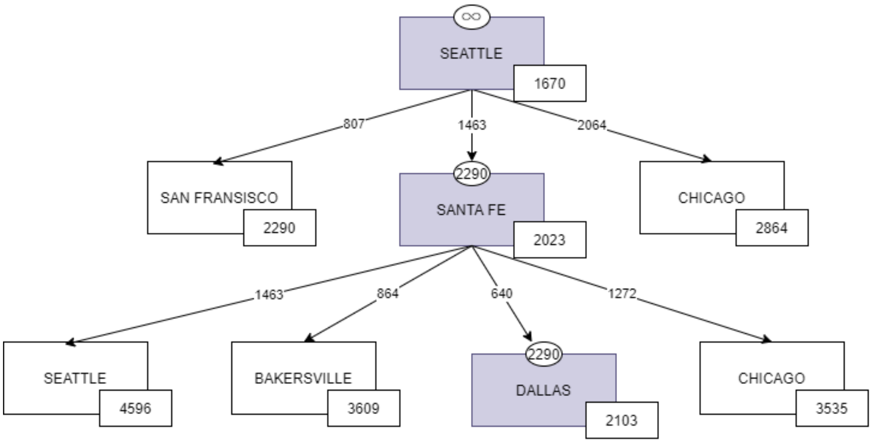


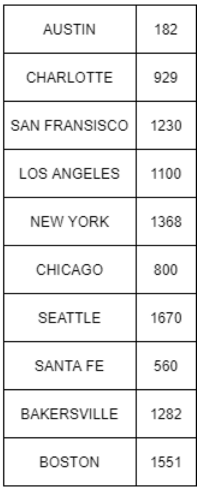


AUSTIN	182
CHARLOTTE	929
SAN FRANSISCO	1230
LOS ANGELES	1100
NEW YORK	1368
CHICAGO	800
SEATTLE	1670
SANTA FE	560
BAKERSVILLE	1282
BOSTON	1551

flimit	inf
best	2103
alternative	2290
current-city	Seattle
next-city	Santa Fe

6. STEP - 6





AUSTIN	182
CHARLOTTE	929
SAN FRANSISCO	1230
LOS ANGELES	1100
NEW YORK	1368
CHICAGO	800
SEATTLE	1670
SANTA FE	560
BAKERSVILLE	1282
BOSTON	1551

flimit	2290
best	2103
alternative	3535
current-city	Santa Fe
next-city	Dallas

3.3 A* PROGRAM

```
Perform the same search for your optimal road trip from Seattle to Dallas when using
A*. List the contents of the frontier and explored list for each node that you visit during
search, in the format:
[Current node: X; Evaluation function (current node)=???;
Explored Cities: [ ....]; Frontier: [(City_X, f(City_X)), ....]; ]
```

SOLUTION

As we have not verified the heuristic to be consistent. Inorder to find the solution for A* we should ideally use a Tree Search algorithm as we know that heuristic provided is an admissible one.

1. STEP - 1

Current Node	Seattle
Evaluation function	1670
Explored cities	[Seattle]
Frontier	[(Santa Fe, 2023), (Chicago, 2864), (San Francisco, 2037)]

2. STEP - 2

Current Node	Santa Fe
Evaluation function	2023
Explored cities	[Seattle, Santa Fe]
Frontier	[(San Francisco, 2037), (Dallas, 2103), (Chicago, 2864), (Bakersville, 3609)]

3. STEP - 3

Current Node	San Francisco
Evaluation function	2037
Explored cities	[Seattle, Santa Fe, San Francisco]
Frontier	[(Dallas, 2103), (Los Angeles, 2290), (Chicago, 2864), (Bakersville, 2372), (Boston, 5453)]

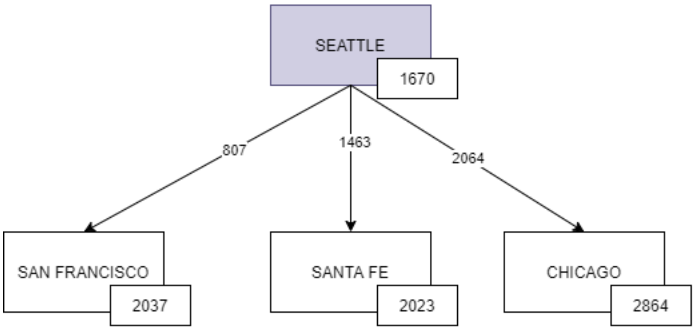
4. STEP - 4

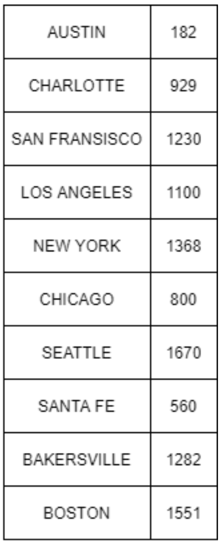
Current Node	Dallas
Evaluation function	2103
Explored cities	[Dallas, Seattle, Santa Fe, San Francisco]
Frontier	Reached Goal

3.4 A* MANUAL

```
Find the same solution manually, specifying:
(a) the current node; If it is not a goal node then also:
(b) the children of the current node and the value of their evaluated function
(detailing how you have computed it)
(c) The current path from Seattle to the current city + the cost
(d) The Contents of the Explored Cities list
(e) The Contents of the Frontier
(f) Next node.
```

1. STEP - 1

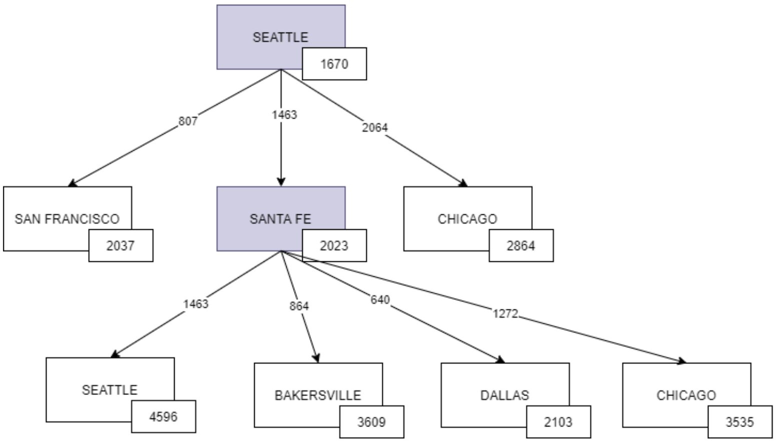


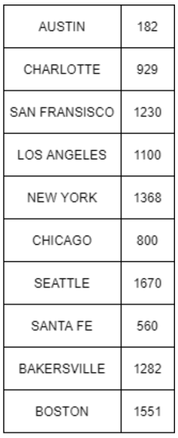


AUSTIN	182
CHARLOTTE	929
SAN FRANSISCO	1230
LOS ANGELES	1100
NEW YORK	1368
CHICAGO	800
SEATTLE	1670
SANTA FE	560
BAKERSVILLE	1282
BOSTON	1551

Current Node	(Seattle, 1670)
Children	[(Santa Fe, 2023), (Chicago, 2864), (San Francisco, 2037)]
Current Path	[Seattle]
Explored cities	[Seattle]
Frontier	[(Santa Fe, 2023), (Chicago, 2864), (San Francisco, 2037)]
Next Node	Santa Fe

2. STEP - 2

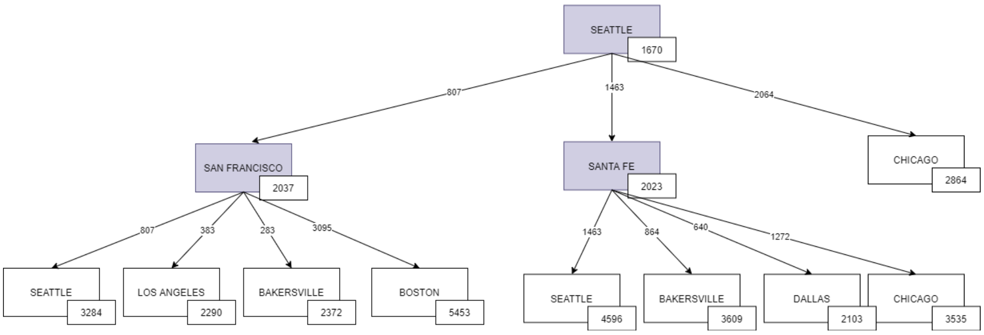




AUSTIN	182
CHARLOTTE	929
SAN FRANSISCO	1230
LOS ANGELES	1100
NEW YORK	1368
CHICAGO	800
SEATTLE	1670
SANTA FE	560
BAKERSVILLE	1282
BOSTON	1551

Current Node	(Santa Fe, 2023)
Children [Seattle, Santa Fe]	[(Dallas, 2103), (Chicago, 3535), (Bakersville, 3609), (Seattle, 4596)] heightCurrent Path
Explored cities	[Seattle, Santa Fe]
Frontier	[(San Francisco, 2037), (Dallas, 2103), (Chicago, 2864), (Bakersville, 3609)]
Next node	San Francisco

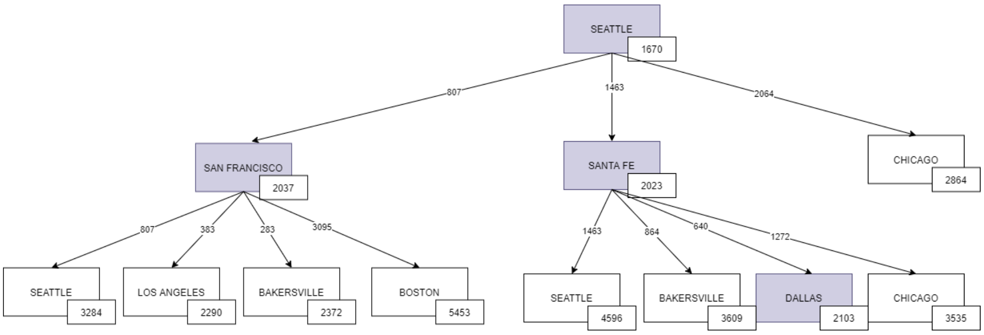
3. STEP - 3



AUSTIN	182
CHARLOTTE	929
SAN FRANSISCO	1230
LOS ANGELES	1100
NEW YORK	1368
CHICAGO	800
SEATTLE	1670
SANTA FE	560
BAKERSVILLE	1282
BOSTON	1551

Current Node	(San Francisco, 2037)
Children	[(Seattle, 3284), (Los Angeles, 2290), (Bakersville, 2373), (Boston, 5453)]
Current Path	[Seattle, San Francisco]
Explored cities	[Seattle, Santa Fe, San Francisco]
Frontier	[(Dallas, 2103), (Los Angeles, 2290), (Chicago, 2864), (Bakersville, 2372), (Boston, 5453)]
Next Node	Dallas

4. STEP - 4



AUSTIN	182
CHARLOTTE	929
SAN FRANSISCO	1230
LOS ANGELES	1100
NEW YORK	1368
CHICAGO	800
SEATTLE	1670
SANTA FE	560
BAKERSVILLE	1282
BOSTON	1551

Current Node	GOAL STATE (Dallas, 2103)

3.4.1 EXTRA CREDIT

`Extra-credit Write a program that will check if the heuristic provided in this problem is consistent, given the road graph TOTAL: 20 points`

STRATEGY

1. A heuristic is consistent if for every node n and every successor n' of n generated by an action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n'
2. $h(parent) \leq c(parent, action, child) + h(child)$
3. We consider the code for greedy best first graph search method and check consistency for each child node when we expand a node
4. If for any node and child in the graph equation 2 doesn't hold we determine the heuristic to be not consistent

```python
def best_first_graph_search_consistent_heuristic_check(problem, f, h, display=False):
    h = memoize(h, 'h')
    f = memoize(f, 'f')
    node = Node(problem.initial)
    frontier = PriorityQueue('min', f)
    frontier.append(node)
    explored = set()
    is_consistent = True
    while (frontier.__len__() > 0) and is_consistent:
        node = frontier.pop()
        """
        1. Here we comment out check for goal state
        2. A heuristic is consistent if for every node n and every successor n' of n generated by an action a,
        the estimated cost of reaching the goal from n is no greater than the step cost of getting to n'
        plus the estimated cost of reaching the goal from n'
        3. h(parent) <= c(parent, action, child) + h(child)
        """
        # if problem.goal_test(node.state):
        #     if display:
        #         print(len(explored), "paths have been expanded and", len(frontier), "paths remain in the frontier")
        #     return node
        explored.add(node.state)
        for child in node.expand(problem):
            print('----------------------------------------------------------------------')
            print('Checking consistency for ')
            print('n : ' + str(node.state))
            print('n\' : ' + str(child.state))
            print('h(n) : ' + str(h(node)))
            print('h(n\') : ' + str(h(child)))
            print('h(n) - h(n\') : ' + str(h(node) - h(child)))
            print('c(n,a,n\') : ' + str(child.path_cost) + ' - ' + str(node.path_cost) + ' = ' + str(child.path_cost-node.path_cost))

            if h(node) > (child.path_cost - node.path_cost) + h(child):
                print('NOT CONSISTENT')
                return False
            else:
                print('ARC CONSISTENT')

            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                if f(child) < frontier[child]:
                    del frontier[child]
                    frontier.append(child)
    return True

def consistent_heuristic_check(problem, h=None, display=False):
    h = memoize(h or problem.h, 'h')
    return best_first_graph_search_consistent_heuristic_check(problem, lambda n: n.path_cost + h(n), lambda n: h(n), display)
```

OBSERVATION

1. For the above search problem we find the heuristic produced to be inconsistent.
2. For nodes **(LosAngeles, Bakersville)** difference between heuristic functions is ***1282-1100 = 182***. But the path cost as presented is **153**
3. Heuristic misses to be optimistic and is greater than the actual path cost considered.

PROBLEM 2 EXTRA CREDIT : Complete output for each strategy is placed in another file named outputs.txt