1(a).<u>**Aim**</u> **:** Implement Exhaustive search techniques using BFS.

<u>**Source Code**</u> **:**

```
def bfs(graph,node,goal):
visited=[]
queue=[]
visited.append(node)
queue.append(node)
while(len(queue)!=0):
p=queue.pop()
print(p,end=" ")
if(p==goal):
print("Found")
return
for i in graph[p]:
if i not in visited:
visited.append(i)
queue.append(i)
graph=eval(input('Enter graph: '))
s=int(input('Enter source: '))
bfs(graph,s)
```

<u>**Output 1**</u> **:**

```
Enter graph: {0:[1,2],1:[0,4],2:[0,3],3:[2,4],4:[1,3,5,6],5:[4,6],6:[4,5]}
Enter source: 2
2 3 4 6 5 1 0
```

<u>**Output 2**</u> **:**

```
Enter graph: {0:[1,2], 1:[2], 2:[0,3],3:[3]}
Enter source: 0
0 2 3 1
```

1(b).Aim : Implement Exhaustive search techniques using DFS.

Source Code :

```
def dfs(graph,node):
visited=[]
queue=[]
visited.append(node)
queue.append(node)
while(len(queue)!=0):
p=queue.pop()
print(p,end=" ")
for i in graph[p]:
if i not in visited:
visited.append(i)
queue.append(i)
graph=eval(input('Enter graph: '))
s=int(input('Enter source: '))
dfs(graph,s)
```

Output 1 :

```
Enter graph: {0:[1,2],1:[3],2:[3,4],3:[4],4:[0]}
Enter source: 0
0 2 4 3 1
```

Output 2 :

```
Enter graph: { 0:[1,2], 1:[2],2:[0,3],3:[3]}
Enter source: 0
0 2 3 1
```


1(c).Aim : Implement Exhaustive search techniques using Uniform Cost Search.

Source Code :

```
g=eval(input('Enter graph: '))
c=eval(input('Enter costs: '))
s=input('Enter source: ')
d=input('Enter destination: ')
q=s
l=[]
cost=0
minc=[]

def add():
k=0
if q not in g:
return
for i in g[q]:
l.append([cost+c[q][k],q,i])
if i==d:
minc.append(cost+c[q][k])
k+=1
```

```
if q!=d:
add()

while l:
j=min(l)
print(j)
q=j[2]
cost=j[0]
l.remove(j)
if q!=d:
add()

print('Minimum cost =',min(minc))
```


D. Venkata Sivaji
Y20CS044

<u>Output</u> :

```
Enter graph: {'Sibiu':['Fagaras','Rimnicu Vilcea'], 'Fagaras':['Bucharest'], 'Rimnicu Vilcea':['Pitesti'], 'Pitesti':['Bucharest']}
Enter costs: {'Sibiu':[99,80], 'Fagaras':[211], 'Rimnicu Vilcea':[97], 'Pitesti':[101]} Enter source: Sibiu
Enter destination: Bucharest
[80, 'Sibiu', 'Rimnicu Vilcea']
[99, 'Sibiu', 'Fagaras']
[177, 'Rimnicu Vilcea', 'Pitesti']
[278, 'Pitesti', 'Bucharest']
[310, 'Fagaras', 'Bucharest']
Minimum cost = 278
```

1(d).<u>Aim</u> : Implement Exhaustive search techniques using Depth-First Iterative Deepening.

<u>Source Code</u> :

```
g=eval(input('Enter graph: '))
s=int(input('Enter source: '))
t=int(input('Enter target: '))
depth=int(input('Enter depth: '))

def DFS(d):
visited=[s]
stack=[s]
```

```
check=[0]
while stack:
f=stack.pop()
print(f,end=' ')
p=check.pop()
if f==t:
print(' Target found within given depth')
return 1
if p+1>d:
continue
for neighbor in g[f]:
if neighbor not in visited:
check.append(p+1)
visited.append(neighbor)
stack.append(neighbor)
return 0

def IDDFS():
for i in range(depth+1):
print('Depth',i,': ',end='')
if DFS(i):
break
print()

IDDFS()
```


<u>Output 1</u> :

```
Enter graph: {1:[2,5],2:[3,4],3:[8,9],5:[6,7],7:[10,11]}
Enter source: 1
Enter target: 6
Enter depth: 2
Depth 0 : 1
Depth 1 : 1 5 2
Depth 2 : 1 5 7 6 Target found within given depth
```

Output 2 :

Enter graph: {1:[2,5],2:[3,4],3:[8,9],5:[6,7]}
Enter source: 1
Enter target: 7
Enter depth: 2
Depth 0 : 1
Depth 1 : 1 5 2
Depth 2 : 1 5 7 Target found within given depth

1(e).Aim : Implement Exhaustive search techniques using Bidirectional.

Source Code :

```
from collections import defaultdict
graph=defaultdict(list)
edges=eval(input('Enter edges: '))
```

```
for i in edges:
graph[i[0]].append(i[1])
graph[i[1]].append(i[0])
src=int(input('Enter initial state: '))
dest=int(input('Enter goal state: '))
visit_f=[src]
visit_b=[dest]
front_f,top_f,front_b,top_b=0,0,0,0

def bfs_f():
if src==dest:
return src
global front_f,top_f
while front_f<=top_f:
s=visit_f[front_f]
for i in graph[s]:
if i not in visit_f:
visit_f.append(i)
top_f=top_f+1
if i in visit_b:
return i
else:
k=bfs_b()
if k!=-1: return k
front_f=front_f+1
return -1

def bfs_b():
global front_b,top_b
while front_b<=top_b:
s=visit_b[front_b]
for i in graph[s]:
if i not in visit_b:
visit_b.append(i)
top_b=top_b+1
```



```
if i in visit_f:
return i
else:
k=bfs_f()
```

```
if k!=-1: return k
front_b=front_b+1
return -1
def remove(l,g):
a=l.index(g)
for i in range(a,0,-1):
if [l[i],l[i-1]] not in edges and [l[i-1],l[i]] not in edges:
l.remove(l[i-1])
a=l.index(g)
return a

g=bfs_f()
if g==-1:
print('No intersection')
else:
print('\nIntersection Node:',g)
a=remove(visit_f,g)
b=remove(visit_b,g)
k=visit_f[:a]+visit_b[b::-1]
print('Path :',k)
```

Output 1 :

```
Enter edges: [[1,2],[1,5],[2,3],[2,4],[3,8],[3,9],[5,6],[5,7],[7,10],[7,11]]
Enter initial state: 1
Enter goal state: 9
Intersection Node: 2
Path : [1, 2, 3, 9]
```

Output 2 :

```
Enter edges: [[1,2],[1,5],[2,3],[2,4],[3,8],[3,9],[5,6]]
Enter initial state: 1
Enter goal state: 6
Intersection Node: 5
Path : [1, 5, 6]
```

2(a).<u>Aim</u> : Implement water jug problem with Search tree generation using BFS.

<u>Source Code</u> :

```
from collections import deque
x_capacity = int(input("Enter Jug 1 capacity:"))
y_capacity = int(input("Enter Jug 2 capacity:"))
end = int(input("Enter target volume:"))

def bfs(start, end, x_capacity, y_capacity):
path = []
front = deque()
front.append(start)
visited = []
#visited.append(start)
while(not (not front)):
current = front.popleft()
x = current[0]
y = current[1]
path.append(current)
if x == end or y == end:
print("Found")
return path
# rule 1
if current[0] < x_capacity and ([x_capacity, current[1]] not in visited):
front.append([x_capacity, current[1]])
visited.append([x_capacity, current[1]])
# rule 2
if current[0]>0 and ([0,current[1]] not in visited):
front.append([0,current[1]])
visited.append([0,current[1]])
# rule 3
if current[1] < y_capacity and ([current[0], y_capacity] not in visited):
front.append([current[0], y_capacity])
visited.append([current[0], y_capacity])
#rule 4
if current[1]>0 and ([current[0],0] not in visited):
front.append([current[0],0])
visited.append([current[0],0])

#rule 5
if (current[0]+current[1])<=x_capacity and current[1]>0 and
([current[0]+current[1],0] not in visited):
```

front.append([current[0]+current[1],0])
visited.append([current[0]+current[1],0])

#rule 6
if (current[0]+current[1])<=y_capacity and current[0]>0 and
([0,current[0]+current[1]] not in visited):
front.append([0,current[0]+current[1]])
visited.append([0,current[0]+current[1]])
#rule 7
if (current[0]+current[1])>=x_capacity and current[1]>0 and
([x_capacity,current[1]-(x_capacity-current[0])] not in visited):
front.append([x_capacity,current[1]-(x_capacity-current[0])])
visited.append([x_capacity,current[1]-(x_capacity-current[0])])
#rule 8:
if (current[0]+current[1])>=y_capacity and current[0]>0 and ([current[0]-
(y_capacity-current[1]),y_capacity] not in visited):
front.append([current[0]-(y_capacity-current[1]),y_capacity])
visited.append([current[0]-(y_capacity-current[1]),y_capacity])
return ("Not found")
def gcd(a, b):
if a == 0:
return b
return gcd(b%a, a)
start = [0, 0]
if end % gcd(x_capacity,y_capacity) == 0:
print(bfs(start, end, x_capacity, y_capacity))
else:
print("No solution possible for this combination.")

<u>Output 1</u> :

Enter Jug 1 capacity:5
Enter Jug 2 capacity:3
Enter target volume:2
Found
[[0, 0], [5, 0], [0, 3], [0, 0], [5, 3], [2, 3]]

Output 2 :

Enter Jug 1 capacity:4
Enter Jug 2 capacity:3
Enter target volume:2
Found
[[0, 0], [4, 0], [0, 3], [0, 0], [4, 3], [1, 3], [3, 0], [1, 0], [3, 3], [0, 1], [4, 2]] **Output 3** :

Enter Jug 1 capacity:4
Enter Jug 2 capacity:2
Enter target volume:1
No solution possible for this combination.

2(b).Aim : Implement water jug problem with Search tree generation using DFS.

Source Code :

```
x_capacity = int(input("Enter Jug 1 capacity:"))
y_capacity = int(input("Enter Jug 2 capacity:"))
end = int(input("Enter target volume:"))

def dfs(start, end, x_capacity, y_capacity):
```

```
path = []
front = []
front.append(start)
visited = []
#visited.append(start)
while(not (not front)):
current = front.pop()
x = current[0]
y = current[1]
path.append(current)
if x == end or y == end:
print("Found")
return path
# rule 1
if current[0] < x_capacity and ([x_capacity, current[1]] not in visited):
front.append([x_capacity, current[1]])
visited.append([x_capacity, current[1]])
# rule 2
if current[0]>0 and ([0,current[1]] not in visited):
front.append([0,current[1]])
visited.append([0,current[1]])
# rule 3
if current[1] < y_capacity and ([current[0], y_capacity] not in visited):
front.append([current[0], y_capacity])
visited.append([current[0], y_capacity])
#rule 4
if current[1]>0 and ([current[0],0] not in visited):
front.append([current[0],0])
visited.append([current[0],0])
#rule 5
if (current[0]+current[1])<=x_capacity and current[1]>0 and
([current[0]+current[1],0] not in visited):
front.append([current[0]+current[1],0])
visited.append([current[0]+current[1],0])
```



```
#rule 6
if (current[0]+current[1])<=y_capacity and current[0]>0 and
([0,current[0]+current[1]] not in visited):
front.append([0,current[0]+current[1]])
visited.append([0,current[0]+current[1]])
```

```
#rule 7
if (current[0]+current[1])>=x_capacity and current[1]>0 and
([x_capacity,current[1]-(x_capacity-current[0])] not in visited):
front.append([x_capacity,current[1]-(x_capacity-current[0])])
visited.append([x_capacity,current[1]-(x_capacity-current[0])])
#rule 8:
if (current[0]+current[1])>=y_capacity and current[0]>0 and ([current[0]-
(y_capacity-current[1]),y_capacity] not in visited):
front.append([current[0]-(y_capacity-current[1]),y_capacity])
visited.append([current[0]-(y_capacity-current[1]),y_capacity])
return ("Not found")
def gcd(a, b):
if a == 0:
return b
return gcd(b%a, a)
start = [0, 0]
if end % gcd(x_capacity,y_capacity) == 0:
print(dfs(start, end, x_capacity, y_capacity))
else:
print("No solution possible for this combination.")
```

<u>Output 1</u> :

```
Enter Jug 1 capacity:3
Enter Jug 2 capacity:5
Enter target volume:4
Found
[[0, 0], [0, 5], [3, 2], [0, 2], [2, 0], [2, 5], [3, 4]]
```

<u>Output 2</u> :

```
Enter Jug 1 capacity:4
Enter Jug 2 capacity:3
Enter target volume:2
Found
[[0, 0], [0, 3], [3, 0], [3, 3], [4, 2]]
```


3a.<u>Aim</u> : Implement Missionaries and Cannibals problem with Search tree generation using BFS

Source Code :

```
from collections import deque
def is_valid_state(state):
m_left, c_left, m_right, c_right, boat = state
if (m_left < 0 or c_left < 0 or m_right < 0 or c_right < 0 or
(m_left != 0 and m_left < c_left) or
(m_right != 0 and m_right < c_right)):
return False
return True

def generate_next_states(state):
m_left, c_left, m_right, c_right, boat = state
possible_states = []
if boat == 'left':
for m in range(3):
for c in range(3):
if m + c > 2 or m + c == 0:
continue
new_state = (m_left - m, c_left - c, m_right + m, c_right + c, 'right')
if is_valid_state(new_state):
possible_states.append(new_state)
else:
for m in range(3):
for c in range(3):
if m + c > 2 or m + c == 0:
continue
new_state = (m_left + m, c_left + c, m_right - m, c_right - c, 'left')
if is_valid_state(new_state):
possible_states.append(new_state)

return possible_states

def bfs(start_state, goal_state):
visited = set()
queue = deque([(start_state, [])])
while queue:
current_state, path = queue.popleft()
visited.add(current_state)
if current_state == goal_state:
```

```
return path
for next_state in generate_next_states(current_state):
if next_state not in visited:
queue.append((next_state, path + [next_state]))
visited.add(next_state)
return []

start_state = (3, 3, 0, 0, 'left')
goal_state = (0, 0, 3, 3, 'right')
visited = set()
start_state = (3, 3, 0, 0, 'left')
goal_state = (0, 0, 3, 3, 'right')
print("bfs solution:")
solution = bfs(start_state, goal_state)
if solution:
for i, state in enumerate(solution):
print(f"Step {i+1}: {state}")
else:
print("No solution found.")
```

<u>Output</u> :

```
bfs solution:
Step 1: (3, 1, 0, 2, 'right')
Step 2: (3, 2, 0, 1, 'left')
Step 3: (3, 0, 0, 3, 'right')
Step 4: (3, 1, 0, 2, 'left')
Step 5: (1, 1, 2, 2, 'right')
Step 6: (2, 2, 1, 1, 'left')
Step 7: (0, 2, 3, 1, 'right')
Step 8: (0, 3, 3, 0, 'left')
Step 9: (0, 1, 3, 2, 'right')
Step 10: (0, 2, 3, 1, 'left')
Step 11: (0, 0, 3, 3, 'right')
```

3b.Aim : Implement Missionaries and Cannibals problem with Search tree generation using DFS

Source Code :

```
from collections import deque
def is_valid_state(state):
m_left, c_left, m_right, c_right, boat = state
if (m_left < 0 or c_left < 0 or m_right < 0 or c_right < 0 or
(m_left != 0 and m_left < c_left) or
(m_right != 0 and m_right < c_right)):
return False
return True

def generate_next_states(state):
m_left, c_left, m_right, c_right, boat = state
possible_states = []
if boat == 'left':
for m in range(3):
for c in range(3):
if m + c > 2 or m + c == 0:
continue
new_state = (m_left - m, c_left - c, m_right + m, c_right + c, 'right')
if is_valid_state(new_state):
possible_states.append(new_state)
else:
for m in range(3):
for c in range(3):
if m + c > 2 or m + c == 0:
continue
new_state = (m_left + m, c_left + c, m_right - m, c_right - c, 'left')
if is_valid_state(new_state):
possible_states.append(new_state)

return possible_states

def dfs(current_state, goal_state, path, visited):
visited.add(current_state)
if current_state == goal_state:
return path
for next_state in generate_next_states(current_state):
if next_state not in visited:
```

```
solution = dfs(next_state, goal_state, path + [next_state], visited)
if solution:
return solution
return None

start_state = (3, 3, 0, 0, 'left')
goal_state = (0, 0, 3, 3, 'right')
visited = set()
start_state = (3, 3, 0, 0, 'left')
goal_state = (0, 0, 3, 3, 'right')
print("dfs solution:")
solution = dfs(start_state, goal_state, [start_state], visited)
if solution:
for i, state in enumerate(solution):
print(f"Step {i+1}: {state}")
else:
print("No solution found.")
```

<u>Output</u> :

```
dfs solution:
Step 1: (3, 3, 0, 0, 'left')
Step 2: (3, 1, 0, 2, 'right')
Step 3: (3, 2, 0, 1, 'left')
Step 4: (3, 0, 0, 3, 'right')
Step 5: (3, 1, 0, 2, 'left')
Step 6: (1, 1, 2, 2, 'right')
Step 7: (2, 2, 1, 1, 'left')
Step 8: (0, 2, 3, 1, 'right')
Step 9: (0, 3, 3, 0, 'left')
Step 10: (0, 1, 3, 2, 'right')
Step 11: (0, 2, 3, 1, 'left')
Step 12: (0, 0, 3, 3, 'right')
```

4(a).<u>Aim</u> : Implement Vacuum World problem with Search tree generation using BFS.

<u>Source Code:</u>

```
from queue import Queue
class State:
def init__(self, agent_position, room_a, room_b):
self.agent_position = agent_position
self.room_a = room_a
self.room_b = room_b
self.left = None
self.right = None
self.clean = None

def get_states(self):
next_states = []
if self.left:
next_states.append(self.left)
if self.right:
next_states.append(self.right)
if self.clean:
next_states.append(self.clean)
return next_states

class Agent:
def init__(self, first_state, goal_state1, goal_state2):
self.first_state = first_state
self.goal_state1 = goal_state1
```

```
self.goal_state2 = goal_state2

def run_bfs(self):
is_initial_state = False
queue = Queue()
checked = set()
queue.put(self.first_state)
checked.add(self.first_state)

while not queue.empty():
current = queue.get()
if not is_initial_state:
is_initial_state = True
print("\nInitial State:", current.state_name)
else:
```



```
print("\nMove to state", current.state_name)
self.prompt_attributes(current)
if current == self.goal_state1 or current == self.goal_state2:
print("Final State:", current.state_name)
return True
elif not current.get_states():
print("Final state wasn't found or reached!")
return False
else:
print("Next possible states:", [state.state_name for state in
current.get_states()])
for state in current.get_states():
if state not in checked:
queue.put(state)
checked.add(state)

def prompt_attributes(self, current):
print("Vacuum is in room", (current.agent_position))
if current.room_a:
print("Room A is clean")
else:
print("Room A is dirty")

if current.room_b:
```

```
print("Room B is clean")
else:
print("Room B is dirty")

def main():
# Input Section
print("Enter the initial state of each room where: 1 - Clean | 2 - Dirty")
status_a = int(input("Enter the state of the first room: ")) == 1
status_b = int(input("Enter the state of the second room: ")) == 1

print("Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B") position =
int(input("Enter the initial position: "))

# Create state instances
state1 = State(None, status_a, status_b)
state2 = State(None, status_a, status_b)
state3 = State(None, status_a, status_b)
state4 = State(None, status_a, status_b)
```



```
state5 = State(None, status_a, status_b)
state6 = State(None, status_a, status_b)
state7 = State(None, status_a, status_b)
state8 = State(None, status_a, status_b)
# Setup and connect each state
state1.agent_position = state1
state1.right = state2
state1.clean = state5
state1.state_name = "State 1"

state2.agent_position = state2
state2.left = state2
state2.right = state4
state2.state_name = "State 2"

state3.left = state3
state3.right = state4
state3.clean = state7
state3.state_name = "State 3"

state4.agent_position = state4
```

```
state4.left = state4
state4.right = state4
state4.state_name = "State 4"
state5.left = state5
state5.right = state6
state5.state_name = "State 5"

state6.agent_position = state6
state6.left = state6
state6.right = state8
state6.state_name = "State 6"

state7.left = state7
state7.right = state8
state7.state_name = "State 7"

state8.agent_position = state8
state8.left = state8
state8.right = state8
state8.state_name = "State 8"
# Run agent
```



```
initial_state = None
if position == 1:
if status_a and status_b:
initial_state = state7
elif status_a:
initial_state = state5
elif status_b:
initial_state = state3
else:
initial_state = state1
elif position == 2:
if status_a and status_b:
initial_state = state8
elif status_a:
initial_state = state6
elif status_b:
initial_state = state4
else:
```

```
initial_state = state2
else:
print("\nInitial state is unknown...")
return
agent = Agent(initial_state, state7, state8)
if agent.run_bfs():
print("\nAgent achieved the goal.")
else:
print("\nAgent failed to reach the goal.")

if name == "_main ":
main()
```

Output 1 :

Enter the initial state of each room where: 1 - Clean | 2 - Dirty
Enter the state of the first room: 2
Enter the state of the second room: 2
Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B Enter the initial position: 1

Initial State: State 1
Room A is dirty
Room B is dirty

Next possible states: ['State 2', 'State 5']

Move to state State 2
Room A is dirty
Room B is dirty
Next possible states: ['State 2', 'State 4']

Move to state State 5
Room A is dirty
Room B is dirty
Next possible states: ['State 5', 'State 6']

Move to state State 4
Room A is dirty
Room B is dirty
Next possible states: ['State 4', 'State 4']

Move to state State 6
Room A is dirty
Room B is dirty
Next possible states: ['State 6', 'State 8']

Move to state State 8
Room A is dirty
Room B is dirty
Final State: State 8
Agent achieved the goal.

Output 2 :

Enter the initial state of each room where: 1 - Clean | 2 - Dirty
Enter the state of the first room: 1
Enter the state of the second room: 1
Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B Enter the initial position: 1

Initial State: State 7
Room A is clean
Room B is clean
Final State: State 7
Agent achieved the goal.

4(b).Aim : Implement Vacuum World problem with Search tree generation using DFS.

Source Code :

```
class State:
```

```
def_init__(self, agent_position, room_a, room_b):
self.agent_position = agent_position
self.room_a = room_a
self.room_b = room_b
self.left = None
self.right = None
self.clean = None

def get_states(self):
next_states = []
if self.left:
next_states.append(self.left)
if self.right:
next_states.append(self.right)
if self.clean:
next_states.append(self.clean)
return next_states

class Agent:
def_init__(self, first_state, goal_state1, goal_state2):
self.first_state = first_state
self.goal_state1 = goal_state1
self.goal_state2 = goal_state2

def run_dfs(self):
visited = set()
return self.dfs(self.first_state, visited)

def dfs(self, current_state, visited):
if current_state in visited:
return False

visited.add(current_state)
print("\nMove to state", current_state.state_name)
self.prompt_attributes(current_state)
if current_state == self.goal_state1 or current_state == self.goal_state2:
print("Final State:", current_state.state_name)
return True
```



```
next_states = current_state.get_states()
```

```
for state in next_states:
if self.dfs(state, visited):
return True
return False

def prompt_attributes(self, current):
print("Vacuum is in room", str(current.agent_position))
if current.room_a:
print("Room A is clean")
else:
print("Room A is dirty")
if current.room_b:
print("Room B is clean")
else:
print("Room B is dirty")

def main():
# Input Section
print("Enter the initial state of each room where: 1 - Clean | 2 - Dirty")
status_a = int(input("Enter the state of the first room: ")) == 1
status_b = int(input("Enter the state of the second room: ")) == 1
print("Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B") position =
int(input("Enter the initial position: "))
# Create state instances
state1 = State(None, status_a, status_b)
state2 = State(None, status_a, status_b)
state3 = State(None, status_a, status_b)
state4 = State(None, status_a, status_b)
state5 = State(None, status_a, status_b)
state6 = State(None, status_a, status_b)
state7 = State(None, status_a, status_b)
state8 = State(None, status_a, status_b)

# Setup and connect each state
state1.agent_position = state1
state1.right = state2
state1.clean = state5
state1.state_name = "State 1"

state2.agent_position = state2
state2.left = state2
```

```
state2.right = state4
state2.state_name = "State 2"

state3.left = state3
state3.right = state4
state3.clean = state7
state3.state_name = "State 3"

state4.agent_position = state4
state4.left = state4
state4.right = state4
state4.state_name = "State 4"

state5.left = state5
state5.right = state6
state5.state_name = "State 5"

state6.agent_position = state6
state6.left = state6
state6.right = state8
state6.state_name = "State 6"

state7.left = state7
state7.right = state8
state7.state_name = "State 7"

state8.agent_position = state8
state8.left = state8
state8.right = state8
state8.state_name = "State 8"

# Run agent
initial_state = None
if position == 1:
if status_a and status_b:
initial_state = state7
elif status_a:
initial_state = state5
elif status_b:
initial_state = state3
else:
initial_state = state1
```

```
elif position == 2:
if status_a and status_b:
initial_state = state8
elif status_a:
initial_state = state6
elif status_b:
initial_state = state4
else:
initial_state = state2
else:
print("\nInitial state is unknown...")
return

agent = Agent(initial_state, state7, state8)
if agent.run_dfs():
print("\nAgent achieved the goal.")
else:
print("\nAgent failed to reach the goal.")

if name == "_main ":
main()
```

<u>Output 1</u> :

Enter the initial state of each room where: 1 - Clean | 2 - Dirty
Enter the state of the first room: 2
Enter the state of the second room: 2
Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B Enter the initial position: 2

Move to state State 2
Room A is dirty
Room B is dirty

Move to state State 4
Room A is dirty
Room B is dirty

Agent failed to reach the goal.

Output 2 :

Enter the initial state of each room where: 1 - Clean | 2 - Dirty
Enter the state of the first room: 2
Enter the state of the second room: 1
Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B Enter the initial position: 1

Move to state State 3
Room A is dirty
Room B is clean

Move to state State 4
Room A is dirty
Room B is clean

Move to state State 7
Room A is dirty
Room B is clean
Final State: State 7
Agent achieved the goal.

5(a).<u>Aim</u> : Implement the Greedy Best First Search Algorithm.

<u>Source Code</u> :

```
class Node:
def_init__(self, v, weight):
self.v = v
self.weight = weight

class pathNode:
def_init__(self, node, parent):
self.node = node
self.parent = parent

def addEdge(u, v, weight):
adj[u].append(Node(v, weight))

def GBFS(h, V, src, dest):
openList = []
```

```
closeList = []
openList.append(pathNode(src, None))
while openList:
currentNode = openList[0]
currentIndex = 0
for i in range(len(openList)):
if h[openList[i].node] < h[currentNode.node]:
currentNode = openList[i]
currentIndex = i
openList.pop(currentIndex)
closeList.append(currentNode)
if currentNode.node == dest:
path = []
cur = currentNode
while cur:
path.append(cur.node)
cur = cur.parent
path.reverse()
return path
for node in adj[currentNode.node]:
if node.v not in [x.node for x in openList] and node.v not in [x.node for x in closeList]:
openList.append(pathNode(node.v, currentNode))
```



```
return []
```

```
V = int(input("Enter the number of vertices: "))
adj = [[] for _ in range(V)]
# Getting input for edges and weights
print("Enter the edges (u v weight), one per line (press enter to stop):")
while True:
edge = input().split()
if len(edge) != 3:
break
u, v, weight = map(int, edge)
addEdge(u, v, weight)
```

```
# Getting user input for the source and destination nodes.
src = int(input("Enter the source node: "))
dest = int(input("Enter the destination node: "))
# Getting the heuristic values for each node.
h = []
for i in range(V):
h_value = int(input("Enter the heuristic value for node " + str(i) + ": "))
h.append(h_value)

path = GBFS(h, V, src, dest)
if path:
print("Shortest path:", " -> ".join(str(node) for node in path))
else:
print("No path found from source to destination.")
```

Output 1 :

Enter the number of vertices: 10
Enter the edges (u v weight), one per line (press enter to stop):
0 1 3
0 2 2
1 3 4
1 4 1
2 5 3
2 6 1
5 7 5
6 8 2
6 9 3

Enter the source node: 0
Enter the destination node: 9
Enter the heuristic value for node 0: 12
Enter the heuristic value for node 1: 4
Enter the heuristic value for node 2: 7
Enter the heuristic value for node 3: 3
Enter the heuristic value for node 4: 8
Enter the heuristic value for node 5: 2
Enter the heuristic value for node 6: 4

Enter the heuristic value for node 7: 9
Enter the heuristic value for node 8: 13
Enter the heuristic value for node 9: 0
Shortest path: 0 -> 2 -> 6 -> 9

<u>Output 2</u> :

Enter the number of vertices: 10
Enter the edges (u v weight), one per line (press enter to stop):
0 1 3
0 2 2
1 3 4
1 4 1
2 5 3
2 6 1
5 7 5
6 8 2
6 9 3

Enter the source node: 0
Enter the destination node: 7
Enter the heuristic value for node 0: 12
Enter the heuristic value for node 1: 4
Enter the heuristic value for node 2: 7
Enter the heuristic value for node 3: 3
Enter the heuristic value for node 4: 8
Enter the heuristic value for node 5: 2
Enter the heuristic value for node 6: 4
Enter the heuristic value for node 7: 9
Enter the heuristic value for node 8: 13
Enter the heuristic value for node 9: 0
Shortest path: 0 -> 2 -> 5 -> 7

5(b).<u>Aim</u> : Implement the A* algorithm.

Source Code :

```
from collections import deque
class Graph:
def_init__(self, adjacency_list):
self.adjacency_list = adjacency_list
def get_neighbors(self, v):
return self.adjacency_list[v]
def h(self, n):
H = {
'A': 1,
'B': 1,
'C': 1,
'D': 1
}
return H[n]

def a_star_algorithm(self, start_node, stop_node):
open_list = set([start_node])
closed_list = set([])
g = {}
g[start_node] = 0
parents = {}
parents[start_node] = start_node
while len(open_list) > 0:
n = None
for v in open_list:
if n == None or g[v] + self.h(v) < g[n] + self.h(n):
n = v;
if n == None:
print('Path does not exist!')
return None
if n == stop_node:
reconst_path = []
while parents[n] != n:
reconst_path.append(n)
n = parents[n]
reconst_path.append(start_node)
reconst_path.reverse()
print('Path found: {}'.format(reconst_path))
return reconst_path
```

```
for (m, weight) in self.get_neighbors(n):
if m not in open_list and m not in closed_list:
open_list.add(m)
parents[m] = n
g[m] = g[n] + weight
else:
if g[m] > g[n] + weight:
g[m] = g[n] + weight
parents[m] = n
if m in closed_list:
closed_list.remove(m)
open_list.add(m)
open_list.remove(n)
closed_list.add(n)
print('Path does not exist!')
return None
adjacency_list = {
'A': [('B', 1), ('C', 3), ('D', 7)],
'B': [('D', 5)],
'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
start_node = input("Enter the start node: ")
goal_node = input("Enter the goal node: ")
graph1.a_star_algorithm(start_node,goal_node)
```

Output 1 :

```
Enter the start node: A
Enter the goal node: D
Path found: ['A', 'B', 'D']
['A', 'B', 'D']
```

Output 2 :

```
Enter the start node: A
Enter the goal node: C
Path found: ['A', 'C']
['A', 'C']
```

6.Aim : Implement 8-puzzle problem using A* algorithm.

Source Code :

```
from heapq import heappop, heappush
class PuzzleNode:
def_init__(self, state, parent=None, g=0, h=0):
self.state = state
self.parent = parent
self.g = g
self.h = h
def_lt (self, other):
return (self.g + self.h) < (other.g + other.h)
def get_blank_position(state):
for i in range(3):
for j in range(3):
if state[i][j] == 0:
return i, j
def get_manhattan_distance(row1, col1, row2, col2):
return abs(row1 - row2) + abs(col1 - col2)
def get_heuristic_value(state, goal_state):
h = 0
for i in range(3):
for j in range(3):
if state[i][j] != 0:
value = state[i][j]
goal_row, goal_col = find_position(goal_state, value)
h += get_manhattan_distance(i, j, goal_row, goal_col)
return h
def get_valid_moves(row, col):
moves = []
if row > 0:
moves.append((-1, 0)) # Move blank tile up
if row < 2:
moves.append((1, 0)) # Move blank tile down
if col > 0:
moves.append((0, -1)) # Move blank tile left
if col < 2:
moves.append((0, 1)) # Move blank tile right
return moves
def get_new_state(state, move):
```

```
row, col = get_blank_position(state)
```



```
new_state = [row[:] for row in state]
new_row, new_col = row + move[0], col + move[1]
new_state[row][col] = new_state[new_row][new_col]
new_state[new_row][new_col] = 0
return new_state
def print_state(state):
for row in state:
print(row)
print()
def is_goal_state(state, goal_state):
return state == goal_state
def find_position(state, value):
for i in range(3):
for j in range(3):
if state[i][j] == value:
return i, j
def get_solution_path(node):
path = []
while node is not None:
path.append(node.state)
node = node.parent
path.reverse()
return path
def solve_puzzle(initial_state, goal_state):
open_set = []
closed_set = set()
h = get_heuristic_value(initial_state, goal_state)
initial_node = PuzzleNode(initial_state, g=0, h=h)
heappush(open_set, initial_node)
while open_set:
current_node = heappop(open_set)
closed_set.add(tuple(map(tuple, current_node.state)))
if is_goal_state(current_node.state, goal_state):
return get_solution_path(current_node)
row, col = get_blank_position(current_node.state)
moves = get_valid_moves(row, col)
for move in moves:
new_state = get_new_state(current_node.state, move)
```

```
if tuple(map(tuple, new_state)) not in closed_set:
g = current_node.g + 1
h = get_heuristic_value(new_state, goal_state)
new_node = PuzzleNode(new_state, parent=current_node, g=g, h=h)
```



```
heappush(open_set, new_node)
return None
print("Enter the initial state (0 represents the blank tile):")
initial_state = []
for i in range(3):
row = list(map(int, input().split()))
initial_state.append(row)
print("Enter the goal state:")
goal_state = []
for i in range(3):
row = list(map(int, input().split()))
goal_state.append(row)
solution = solve_puzzle(initial_state, goal_state)
if solution is not None:
print("Solution found!")
for state in solution:
print_state(state)
else:
print("No solution found.")
```

Output 1 :

initial_state = [[1, 2, 3], [0,4,6], [7, 5, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Solution found!
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

[1, 2, 3]
[4, 5, 6]

[7, 0, 8]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

<u>Output 2</u> :

initial_state = [[2,8,3], [1,6,4], [7, 0,5]]
goal_state = [[1, 2, 3], [8,0,4], [7, 6,5]]
Solution found!
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

7.<u>Aim</u> : Implement AO* algorithm for General graph problem.

<u>Source Code:</u>

```
def Cost(H, condition, weight = 1):
cost = {}
if 'AND' in condition:
AND_nodes = condition['AND']
Path_A = ' AND '.join(AND_nodes)
PathA = sum(H[node]+weight for node in AND_nodes)
cost[Path_A] = PathA
OR_nodes = condition['OR']
Path_B =' OR '.join(OR_nodes)
PathB = min(H[node]+weight for node in OR_nodes)
cost[Path_B] = PathB
return cost

def update_cost(H, Conditions, weight=1):
Main_nodes = list(Conditions.keys())
Main_nodes.reverse()
least_cost= {}
for key in Main_nodes:
```

```
condition = Conditions[key]
print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))
c = Cost(H, condition, weight)
H[key] = min(c.values())
least_cost[key] = Cost(H, condition, weight)
return least_cost

def shortest_path(Start,Updated_cost, H):
Path = Start
if Start in Updated_cost.keys():
Min_cost = min(Updated_cost[Start].values())
key = list(Updated_cost[Start].keys())
values = list(Updated_cost[Start].values())
Index = values.index(Min_cost)

Next = key[Index].split()
if len(Next) == 1:
Start =Next[0]
Path += '<--' +shortest_path(Start, Updated_cost, H)
# ADD TO PATH FOR AND PATH
```



```
else:
Path +='<--('+key[Index]+') '
Start = Next[0]
Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '
Start = Next[-1]
Path += shortest_path(Start, Updated_cost, H) + ']'
return Path

H=eval(input('Enter nodes with heuristic costs: '))
Conditions=eval(input('Enter graph: '))
weight = 1
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H)
```

Output 1 :

Enter nodes with heuristic costs: {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}
Enter graph: {'A': {'OR': ['B'], 'AND': ['C', 'D']},'B': {'OR': ['E', 'F']},'C': {'OR': ['G'], 'AND':

['H', 'I']},'D': {'OR': ['J']}}
Updated Cost :
D : {'OR': ['J']} >>> {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}
Shortest Path :
A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]

Output 2 :
Enter nodes with heuristic costs: {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9} Enter graph: {'A': {'OR': ['B'], 'AND': ['C', 'D']},'B': {'OR': ['E', 'F']}}
Updated Cost :
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 8, 'B': 9}
Shortest Path :
A<--(C AND D) [C + D]

8(a).Aim : Implement Game trees using MINIMAX algorithm.

Source Code :
```
import math
def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):
# Base case: targetDepth reached
if curDepth == targetDepth:
return scores[nodeIndex]
if maxTurn:
return max(minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),
minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth))
else:
return min(minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth),
minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth))

num_nodes = int(input("Enter the number of nodes: "))
```

```
scores = []
print("Enter the scores for each node:")
for i in range(num_nodes):
score = int(input(f"Node {i}: "))
scores.append(score)
treeDepth = math.log(len(scores), 2)
print("The optimal value is:", minimax(0, 0, True, scores, int(treeDepth)))
```

<u>Output 1</u> :

Enter the number of nodes: 8
Enter the scores for each node:
Node 0: 3
Node 1: 5
Node 2: 6
Node 3: 9
Node 4: 1
Node 5: 2
Node 6: 0
Node 7: -1
The optimal value is: 5

<u>Output 2</u> :

Enter the number of nodes: 8
Enter the scores for each node:
Node 0: -1
Node 1: 4
Node 2: 2
Node 3: 6
Node 4: -3
Node 5: -5
Node 6: 0
Node 7: 7

The optimal value is: 4

Output 3 :

Enter the number of nodes: 8
Enter the scores for each node:
Node 0: 2
Node 1: 3
Node 2: 5
Node 3: 9
Node 4: 0
Node 5: 1
Node 6: 7
Node 7: 5
The optimal value is: 3

8(b).Aim : Implement Game trees using Alpha-Beta pruning.

Source Code :

```
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
if depth == 0:
return values[nodeIndex]
if maximizingPlayer:
```

```
best = MIN
# Recur for left and right children
for i in range(0, 2):
val = minimax(depth - 1, nodeIndex * 2 + i, False, values, alpha, beta)
best = max(best, val)
alpha = max(alpha, best)
# Alpha Beta Pruning
if beta <= alpha:
break
return best
else:
best = MAX
# Recur for left and right children
for i in range(0, 2):
val = minimax(depth - 1, nodeIndex * 2 + i, True, values, alpha, beta)
best = min(best, val)
beta = min(beta, best)
# Alpha Beta Pruning
if beta <= alpha:
break
return best
if name == "_main ":
# Take user input for values
values = []
num_nodes = int(input("Enter the number of nodes: "))
print("Enter the values for each node:")
for i in range(num_nodes):
value = int(input(f"Node {i}: "))
values.append(value)
depth = int(input("Enter the depth: "))
print("The optimal value is:", minimax(depth, 0, True, values, MIN, MAX))
```


Output 1 :

Enter the number of nodes: 8
Enter the scores for each node:
Node 0: 3
Node 1: 5
Node 2: 6
Node 3: 9
Node 4: 1
Node 5: 2
Node 6: 0
Node 7: -1
Enter the depth: 4
The optimal value is: 5

Output 2 :

Enter the number of nodes: 8
Enter the scores for each node:
Node 0: -1
Node 1: 4
Node 2: 2
Node 3: 6
Node 4: -3
Node 5: -5
Node 6: 0
Node 7: 7
Enter the depth: 4
The optimal value is: 4
The optimal value is: 3

9.Aim : Implement Crypt arithmetic problems.

Source Code :

```
import itertools

def number(n,d):
t=0
for i in n:
t=d[i]+(t*10)
return t

def test(l,s,d):
sum=0
for i in l:
sum=sum+number(i,d)
if sum==number(s,d):
return 1
return 0

def check(d):
for i in d.keys():
if i in c and d[i]==0:
return 1
return 0

l=input('Enter list of strings: ').split()
s=input('Enter output string: ')

c=[]
for i in l:
c.append(i[0])
c.append(s[0])

p=list(set(''.join(l)+s))
q=len(p)

k=list(itertools.permutations(range(0,10),q))

d={}
f=0
for i in k:
```

```
for j in range(q):
```



```
d[p[j]]=i[j]
if check(d):
continue
if test(l,s,d) == 1:
f=1
print(d)
print('Solution found')
break

if f==0: print('No solution found')
```

Output 1 :

Enter list of strings: send more
Enter output string: money
{'d':7, 'y': 2, 'e': 5, 'o': 0, 'r': 8, 's': 9, 'n': 6, 'm': 1}
Solution found

Output 2 :

Enter list of strings: your you
Enter output string: heart
{'y': 9, 'a': 3, 'o': 4, 'h': 1, 'r': 6, 't': 8, 'u': 2, 'e': 0}
Solution found