6. Implement Vacuum World problem with Search tree generation using

    a. BFS b. DFS

Source Code:

```
from collections import deque

def move(state):
    """Move vacuum between A (left) and B (right)."""
    state = list(state)
    if state[1] == '1':  # If vacuum at A, move to B
        state[1] = '0'
        state[3] = '1'
    else:  # If vacuum at B, move to A
        state[1] = '1'
        state[3] = '0'
    return ''.join(state)

def clean(state):
    """Clean the current location if dirty."""
    state = list(state)
    if state[1] == '1' and state[0] == '1':  # If vacuum at A & A is dirty
        state[0] = '0'
    if state[3] == '1' and state[2] == '1':  # If vacuum at B & B is dirty
        state[2] = '0'
    return ''.join(state)

def make_tree(start, final_states):
    """Generate a tree of possible states using vacuum world transitions."""
    tree = {start: []}
    all_nodes = [start]
    visited = set()
    queue = deque([start])
```

```
    while queue:
        source = queue.popleft()
        if source in final_states:
            break

        # Generate child nodes by moving and cleaning
        for action in [move, clean]:
            new_state = action(source)
            if new_state not in visited:
                visited.add(new_state)
                all_nodes.append(new_state)
                tree[source].append(new_state)
                tree[new_state] = []
                queue.append(new_state)
                if new_state in final_states:
                    break

    return tree

def bfs(start, final_states, tree):
    """Breadth-First Search (BFS) to find a solution path."""
    queue = deque([[start]])
    visited = set()

    while queue:
        path = queue.popleft()
        state = path[-1]

        if state in final_states:
            return path  # Return the solution path

        if state not in visited:
```

```
            visited.add(state)
            for child in tree[state]:
                new_path = path + [child]
                queue.append(new_path)

    return None

def dfs(start, final_states, tree):
    """Depth-First Search (DFS) to find a solution path."""
    stack = [[start]]
    visited = set()

    while stack:
        path = stack.pop()
        state = path[-1]

        if state in final_states:
            return path  # Return the solution path

        if state not in visited:
            visited.add(state)
            for child in tree[state]:
                new_path = path + [child]
                stack.append(new_path)

    return None

# User input
start = input("Enter start state (e.g., '1101' where 1=dirty, 0=clean, vacuum=position): ")
final_states = {'0100', '0001'}  # Both locations must be clean
```

```
# Build state transition tree
tree = make_tree(start, final_states)
print("\nGenerated State Tree:")
for key, values in tree.items():
    print(f"{key} -> {values}")

# Solve using BFS
solution_bfs = bfs(start, final_states, tree)
if solution_bfs:
    print("\nBFS Solution Path:")
    print(" -> ".join(solution_bfs))
else:
    print("\nNo solution found using BFS.")

# Solve using DFS
solution_dfs = dfs(start, final_states, tree)
if solution_dfs:
    print("\nDFS Solution Path:")
    print(" -> ".join(solution_dfs))
else:
    print("\nNo solution found using DFS.")
```

Output:

```
Enter start state (e.g., '1101' where 1=dirty, 0=clean, vacuum=position
): 0101

Generated State Tree:
0101 -> ['0001']
0001 -> []

BFS Solution Path:
0101 -> 0001

DFS Solution Path:
   101 -> 0001
```

---

7. Implement the following

a. Greedy Best First Search
b. A* algorithm

Source Code:

a)

```
# Greedy Best First Search Function
def greed(cost, h, source, destination):
    op = []  # Open list (nodes to explore)
    c = []   # Closed list (visited nodes)

    while source != destination:
        op.append(source)
        children = []
        children_f = []

        # Finding child nodes
        for i in range(len(cost[0])):
            if cost[source][i] > 0:
                children.append(i)
                children_f.append(h[i])  # Store heuristic values

        if len(children) > 0:
            index = children_f.index(min(children_f))  # Select lowest heuristic
            source = children[index]  # Move to the best node

    op.append(source)  # Add destination to path
    return op, c
n = int(input("Enter number of nodes in the graph: "))  # Number of nodes
cost = [[0] * n for i in range(n)]
print("Nodes are named as:", *range(1, n+1))
print("Enter path costs as: <from_node> <to_node> <cost>")
print("Enter 'x' to stop input.")
```

```
# Taking cost input
while True:
    s = input("Enter: ")
    if s.lower() == 'x':
        break
    s = s.split()
    cost[int(s[0])-1][int(s[1])-1] = int(s[2])

# Taking heuristic values
h = list(map(int, input("Enter heuristic values in order: ").split()))

# Taking source and destination
source = int(input("Enter source node: ")) - 1
destination = int(input("Enter destination node: ")) - 1

# Running the algorithm
op, c = greed(cost, h, source, destination)

# Output path
print("\nNodes travelled are:")
print(" -> ".join(str(i+1) for i in op))
```

Output:

```
Enter number of nodes in the graph: 5
Nodes are named as: 1 2 3 4 5
Enter path costs as: <from_node> <to_node> <cost>
Enter 'x' to stop input.
Enter: 1 2 2
Enter: 1 3 4
Enter: 2 4 1
Enter: 3 5 3
Enter: 4 5 2
Enter: x
Enter heuristic values in order: 7 6 2 1 0
Enter source node: 1
Enter destination node: 5

Nodes travelled are:
1 -> 3 -> 5
```

b)

```
#A* algorithm
import heapq

def astar(n, cost, h, src, dest):
    pq = []  # Priority queue for nodes
    heapq.heappush(pq, (h[src], 0, src, [src]))  # (f(n), g(n), node, path)

    while pq:
        f, g, node, path = heapq.heappop(pq)

        if node == dest:
            return path, g  # Return path and total cost

        for nxt in range(n):
            if cost[node][nxt] > 0:  # If there's an edge
                new_g = g + cost[node][nxt]  # Update cost g(n)
                new_f = new_g + h[nxt]  # f(n) = g(n) + h(n)
                heapq.heappush(pq, (new_f, new_g, nxt, path + [nxt]))

    return None, float('inf')  # No path found

# Input
n = int(input("Enter number of nodes: "))
cost = [[0] * n for _ in range(n)]

print("Enter edges as: <from> <to> <cost> (x to stop)")
while True:
    s = input("Enter: ")
    if s.lower() == 'x': break
    u, v, c = map(int, s.split())
    cost[u-1][v-1] = c  # Convert to 0-based index
```

```
h = list(map(int, input("Enter heuristics: ").split()))
src = int(input("Enter source: ")) - 1
dest = int(input("Enter destination: ")) - 1

# Run A* algorithm
path, total_cost = astar(n, cost, h, src, dest)

# Output
if path:
    print("Path:", " -> ".join(str(p+1) for p in path))
    print("Cost:", total_cost)
else:
    print("No path found!")
```

Output:

```
Enter number of nodes: 5
Enter edges as: <from> <to> <cost> (x to stop)
Enter: 1 2 2
Enter: 1 3 4
Enter: 2 4 1
Enter: 3 5 3
Enter: 4 5 2
Enter: x
Enter heuristics: 7 6 2 1 0
Enter source: 1
Enter destination: 5
Path: 1 -> 3 -> 5
Cost: 7
```

8. Implement 8-puzzle problem using A* algorithm

Source Code:

```
import heapq


# Check if puzzle is solvable
def is_solvable(p):
    p = p.replace("_", "")
    inv = sum(p[i] > p[j] for i in range(len(p)) for j in range(i + 1, len(p)))
    return inv % 2 == 0


# Get possible moves
def get_moves(state):
    moves = []
    i = state.index("_")
    swap = [(i, i+1), (i, i-1), (i, i+3), (i, i-3)]  # Right, Left, Down, Up

    for a, b in swap:
        if 0 <= b < 9 and not (i % 3 == 2 and b == i + 1) and not (i % 3 == 0 and b == i - 1):
            t = list(state)
            t[a], t[b] = t[b], t[a]
            moves.append("".join(t))

    return moves


# Manhattan Distance heuristic
def heuristic(s, goal):
    return sum(abs(i//3 - goal.index(s[i])//3) + abs(i%3 - goal.index(s[i])%3) for i in range(9) if s[i] != "_")


# A* Algorithm
def solve_puzzle(start, goal):
```

```
    if not is_solvable(start):
        return None, "No solution found."

    pq = [(heuristic(start, goal), 0, start, [start])]
    visited = set()

    while pq:
        _, g, state, path = heapq.heappop(pq)
        if state == goal:
            return path, g  # Solution found

        if state in visited:
            continue
        visited.add(state)

        for nxt in get_moves(state):
            if nxt not in visited:
                heapq.heappush(pq, (g + 1 + heuristic(nxt, goal), g + 1, nxt, path + [nxt]))

    return None, "No solution found."

# Input
s = input("Enter start state: ")
g = input("Enter goal state: ")

# Solve
path, moves = solve_puzzle(s, g)

# Output
if path:
    print("\nSolution Steps:")
```

```
    for p in path:
        print(p[:3], "\n" + p[3:6], "\n" + p[6:], "\n")
    print("Total moves:", moves)
else:
    print(moves)
```

Output:

```
Enter start state: 1234_6758
Enter goal state: 12345678_

Solution Steps:
123
4_6
758

123
456
7_8

123
456
78_

Total moves: 2
```

---

9. Implement AO* algorithm for General graph problem

Source Code:

```
import heapq
import ast


def cost(H, condition, weight=1):
    costs = {}
    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node] + weight for node in AND_nodes)
        costs[Path_A] = PathA
```

```
    if 'OR' in condition:
        OR_nodes = condition['OR']
        Path_B = ' OR '.join(OR_nodes)
        PathB = min(H[node] + weight for node in OR_nodes)
        costs[Path_B] = PathB
    return costs

def update_cost(H, Conditions, weight=1):
    main_nodes = list(Conditions.keys())[::-1]  # Reverse the node order
    least_cost = {}
    for key in main_nodes:
        condition = Conditions[key]
        c = cost(H, condition, weight)
        print(f"{key}: {Conditions[key]} >>> {c}")
        H[key] = min(c.values())
        least_cost[key] = c
    return least_cost

def shortest_path(Start, Updated_cost, H):
    Path = Start
    if Start in Updated_cost:
        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(' AND ') if ' AND ' in key[Index] else key[Index].split(' OR ')

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

```
        Path += '[' + ' + '.join(shortest_path(n, Updated_cost, H) for n in Next) + ']'
    return Path


# Use ast.literal_eval() for safe parsing
H = ast.literal_eval(input('Enter nodes with heuristic costs (dict format): '))
Conditions = ast.literal_eval(input('Enter graph structure (dict format): '))


weight = 1
print('Updated Cost:')
Updated_cost = update_cost(H, Conditions, weight)


start_node = input('Enter start node: ')
print('Optimal Path:', shortest_path(start_node, Updated_cost, H))
```

Output:

```
Enter nodes with heuristic costs (dict format): {'A': 10, 'B': 8, 'C':
5, 'D': 7, 'E': 3, 'G': 0}
Enter graph structure (dict format): {'A': {'OR': ['B', 'C']}, 'B': {'A
ND': ['D', 'E']}, 'C': {'OR': ['G']}, 'D': {'OR': ['G']}, 'E': {'OR': [
'G']}}
Updated Cost:
E: {'OR': ['G']} >>> {'G': 1}
D: {'OR': ['G']} >>> {'G': 1}
C: {'OR': ['G']} >>> {'G': 1}
B: {'AND': ['D', 'E']} >>> {'D AND E': 4}
A: {'OR': ['B', 'C']} >>> {'B OR C': 2}
Enter start node: A
Optimal Path: A <-- (B OR C) [B <-- (D AND E) [D <-- G + E <-- G] + C <
-- G]
```

10. Implement Game trees using
    a. MINIMAX algorithm
    b. Alpha-Beta pruning

Source Code:

a) #minimax

```
import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):
    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))

scores = list(map(int,input("Enter scores:").split()))
treeDepth = int(math.log(len(scores), 2))

print("The optimal value is:", minimax(0, 0, True, scores, treeDepth))
```

Output:

```
Enter scores:7 5 2 9
The optimal value is: 5
```

b) #alpha beta pruning

```
import math

MAX, MIN = math.inf, -math.inf

def minimax(d, i, isMax, v, α, β):
   if d == 3 or i >= len(v):  # Check if i is within bounds
      return v[i] if i < len(v) else 0  # Return 0 if out of bounds

   if isMax:
      best = MIN
      for j in range(2):
         best = max(best, minimax(d + 1, i * 2 + j, False, v, α, β))
         α = max(α, best)
         if β <= α:
            break
      return best
   else:
      best = MAX
      for j in range(2):
         best = min(best, minimax(d + 1, i * 2 + j, True, v, α, β))
         β = min(β, best)
         if β <= α:
            break
      return best

# Input
v = list(map(int, input("Enter at least 8 scores (space-separated): ").split()))
while len(v) < 8:
   print("Please enter at least 8 values for a full tree.")
   v = list(map(int, input("Enter at least 8 scores (space-separated): ").split()))
```

Output

print("Optimal value:", minimax(0, 0, True, v, MIN, MAX))

Output:

```
Enter at least 8 scores (space-separated): 2 9 2 6 26 29 7 17 18
Optimal value: 17
```

---

11. Implement Crypt arithmetic problems.

Source Code:

```
import itertools


def number(word, digit_map):
    """Convert a word into a number using the digit map."""
    return int(''.join(str(digit_map[letter]) for letter in word))


def solve_cryptarithmetic(puzzle):
    """Solve a cryptarithmetic puzzle using brute force and constraints."""

    words = puzzle.split()
    unique_chars = set(''.join(words))  # Unique letters in the puzzle

    if len(unique_chars) > 10:
        return "Invalid puzzle: More than 10 unique characters (Only 0-9 available)"

    leading_chars = {word[0] for word in words if len(word) > 1}  # First letters (excluding single-char words)

    for digits in itertools.permutations(range(10), len(unique_chars)):
        digit_map = dict(zip(unique_chars, digits))

        # Ensure leading characters are not mapped to 0
```

```
        if any(digit_map[char] == 0 for char in leading_chars):
            continue

        # Check if sum of all words except last equals the last word
        if sum(number(word, digit_map) for word in words[:-1]) == number(words[-1], digit_map):
            return digit_map

    return "No solution found"

# Input and execution
puzzle = input("Enter the cryptarithmetic puzzle (words separated by spaces): ")
solution = solve_cryptarithmetic(puzzle)
print("Solution:", solution)
```

Output:

```
Enter the cryptarithmetic puzzle (words separated by spaces): SEND MORE
MONEY
Solution: {'M': 1, 'O': 0, 'E': 5, 'R': 8, 'Y': 2, 'N': 6, 'S': 9, 'D':
7}


Enter the cryptarithmetic puzzle (words separated by spaces): LOYAL TRU
ST HONEST
Solution: Invalid puzzle: More than 10 unique characters (Only 0-9 avai
lable)
```