1. FAMILY RELATIONSHIP TREE

```
from collections import defaultdict
class Solution:
    def __init__(self, head_name):
        self.family = defaultdict(list)
        self.head = head_name
        self.dead = set()

    def birth(self, p_name, c_name):
        self.family[p_name].append(c_name)
    def death(self, name):
        self.dead.add(name)
    def inheritance(self):
        ans = []
        def depth_search(current):
            if current not in self.dead:
                ans.append(current)
            for child in self.family[current]:
                depth_search(child)
        depth_search(self.head)
        return ans
ob = Solution('Paul')
ob.birth('Paul', 'Zach')
ob.birth('Zach', 'Emma')
ob.birth('Paul', 'David')
ob.birth('David', 'Sophia')
ob.death('David')
result = ob.inheritance()
print(result)
```

OUTPUT:

['Paul', 'Zach', 'Emma']

2. WATER JUG PROBLEM

```
from collections import defaultdict
def waterJugSolver(amt1, amt2, jug1=4, jug2=3, aim=2, visited=defaultdict(bool)):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1 - amt1)), amt2 - min(amt2, (jug1 - amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2 - amt2)), amt2 + min(amt1, (jug2 - amt2))))
    else:
        return False
print("Steps: ")
waterJugSolver(0, 0)
```

OUTPUT:

```
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
```

3. BEST FIRST SEARCH

```
from queue import PriorityQueue
def best_first_search(graph, actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True
    while not pq.empty():
        u = pq.get()[1]
        print(u, end=" ")
        if u == target:
            brea
        for v, c in graph[u]:
            if not visited[v]:
                visited[v] = True
                pq.put((c, v))
        print()
def main():
    v = int(input("Enter the number of vertices in the graph: "))
    graph = [[] for _ in range(v)]

    e = int(input("Enter the number of edges in the graph: "))
    print("Enter edges in the format 'source destination cost':")
    for _ in range(e):
        x, y, cost = map(int, input().split())
        graph[x].append((y, cost))
        graph[y].append((x, cost))
    source = int(input("Enter the source node: "))
    target = int(input("Enter the target node: "))
    print("Best First Search Path:")
    best_first_search(graph, source, target, v)
if __name__ == "__main__":
    main()
```

OUTPUT

Enter the number of vertices in the graph: 14

Enter the number of edges in the graph: 13

Enter edges in the format 'source destination cost':

0 1 3

0 2 6

0 3 5

1 4 9

1 5 8

2 6 12

2 7 14

3 8 7

8 9 5

8 10 6

9 11 1

9 12 10

9 13 2

Enter the source node: 0

Enter the target node: 9

Best First Search Path:

0 1 2 3 4 5 6 7 8 9

4.DEPTH FIRST SEARCH

```
graph = {'5': ['3', '7'], '3': ['2', '4'], '7': ['8'], '2': [], '4': ['8'], '8': []}
visited = set()
def dfs(visited, graph, node):
    # function for DFS
    if node not in visited:
        print(node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
print("Following is the Depth-First Search:")
dfs(visited, graph, '5')
```

OUTPUT:

```
Following is the Depth-First Search:
5
3
2
4
8
7
```

5. DEPTH LIMIT SEARCH

```python
class DFS:
    def main(self):
        nv = int(input("Enter the number of vertices: "))
        ne = int(input("Enter the number of edges: "))
        cost = [[0] * nv for _ in range(nv)]

        print("Enter the edges (format: vertex1 vertex2):")
        for _ in range(ne):
            v1, v2 = map(int, input().split())
            cost[v1][v2] = 1

        s = int(input("Enter the source vertex: "))
        depth_limit = int(input("Enter the depth limit: "))  # Set the depth limit
        visit = [False] * nv
        self.DLSTraversal(cost, s, visit, depth_limit)  # Call DLS function

    def DLSTraversal(self, cost, s, visit, depth_limit):
        self.DLSRecursive(cost, s, visit, depth_limit, 0)

    def DLSRecursive(self, cost, s, visit, depth_limit, depth):
        if depth == depth_limit:
            return
        visit[s] = True
        print(s, end=" ")

        for i in range(len(cost)):
            if cost[s][i] != 0 and not visit[i]:
                self.DLSRecursive(cost, i, visit, depth_limit, depth + 1)
if __name__ == "__main__":
    dfs_instance = DFS()
    dfs_instance.main()
```

OUTPUT:

Enter the number of vertices: 4

Enter the number of edges: 5

Enter the edges (format: vertex1 vertex2):

0 1

0 2

1 2

2 3

3 1

Enter the source vertex: 0

Enter the depth limit: 4

0 1 2 3

6. A* SEARCH ALGORTIHM

```
import heapq
def a_star(start, goal, graph, heuristic):
    open_list = [(0, start)]
    closed_list = set()
    g_scores = {start: 0}
    parents = {start: None}
    while open_list:
        f, current = heapq.heappop(open_list)
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = parents[current]
            return path[::-1]
        closed_list.add(current)
        for neighbor in graph[current]:
            if neighbor in closed_list: continue
            g = g_scores[current] + graph[current][neighbor]
            if neighbor not in [n[1] for n in open_list]:
                heapq.heappush(open_list, (g + heuristic(neighbor, goal), neighbor))
            elif g < g_scores[neighbor]:
                i = [n[1] for n in open_list].index(neighbor)
                open_list[i] = (g + heuristic(neighbor, goal), neighbor)
            parents[neighbor] = current
            g_scores[neighbor] = g
    return None
graph = {
    'A': {'B': 5, 'C': 10},'B': {'D': 15, 'E': 20}, 'C': {'F': 5},'D': {'G': 25},'E': {'G': 20},'F': {'G': 10},'G': {}}
def heuristic(node, goal):
    h = {'A': 35, 'B': 30, 'C': 25, 'D': 20, 'E': 15, 'F': 10, 'G': 0}
    return h[node]
```

```
start = input("Start node: ")
goal = input("Goal node: ")
path = a_star(start, goal, graph, heuristic)
if path:
    print(f"Path from {start} to {goal}:")
    print(path)
else:
    print(f"No path found from {start} to {goal}.")
```

OUTPUT:

```
Start node: A
Goal node: G
Path from A to G:
['A', 'C', 'F', 'G']
```

7. 4 QUEENS PROBLEM

```
def place(pos):
    global a
    for i in range(1, pos):
        if a[i] == a[pos] or abs(a[i] - a[pos]) == abs(i - pos):
            return False
    return True
def print_sol(N):
    global cnt
    global a
    cnt += 1
    print(f"\n\nSolution {cnt}:\n")
    for i in range(1, N + 1):
        for j in range(1, N + 1):
            if a[i] == j:
                print("Q\t", end="")
            else:
                print("*\t", end="")
        print()
def queen(n):
    global cnt
    global a
    cnt = 0
    k = 1
    a = [0] * 30
    a[k] = 0
    while k != 0:
        a[k] += 1
        while a[k] <= n and not place(k):
            a[k] += 1
        if a[k] <= n:
            if k == n:
```

```
            print_sol(n)
        else:
            k += 1
            a[k] = 0
    else:
        k -= 1
if __name__ == "__main__":
    N = int(input("Enter a number: "))
    queen(N)
    print(f"\nTotal solutions = {cnt}")
```

OUTPUT:

```
Enter a number: 4
Solution 1:
*    Q    *    *
*    *    *    Q
Q    *    *    *
*    *    Q    *
Solution 2:
*    *    Q    *
Q    *    *    *
*    *    *    Q
*    Q    *    *

Total solutions = 2
```

8. GRAPH COLORING

```
def addEdge(adj, v, w):
    adj[v].append(w)
    adj[w].append(v)
    return adj
def greedyColoring(adj, V):
    result = [-1] * V
    result[0] = 0
    available = [False] * V
    for u in range(1, V):
        for i in adj[u]:
            if result[i] != -1:
                available[result[i]] = True
        cr = 0
        while cr < V:
            if available[cr] == False:
                break
            cr += 1
        result[u] = cr
        for i in adj[u]:
            if result[i] != -1:
                available[result[i]] = False
    for u in range(V):
        if result[u] == 0:
            print("Vertex", u, "---> color Red")
        elif result[u] == 1:
            print("Vertex", u, "---> color Green")
        elif result[u] == 2:
            print("Vertex", u, "---> color Blue")
        else:
            print("Vertex", u, "---> color Yellow")
g = [[] for i in range(5)]
g = addEdge(g, 0, 1)
```

```
g = addEdge(g, 0, 2)
g = addEdge(g, 1, 2)
g = addEdge(g, 1, 3)
g = addEdge(g, 2, 3)
g = addEdge(g, 3, 4)
print("Coloring of the graph is:")
greedyColoring(g, 5)
```

OUTPUT:

```
Coloring of the graph is:
Vertex 0 ---> color Red
Vertex 1 ---> color Green
Vertex 2 ---> color Blue
Vertex 3 ---> color Red
Vertex 4 ---> color Green

0 (Red) --- 1 (Green)
|         / |
|       /   |
|     /     |
|   /       |
| /         |
2 (Blue) --- 3 (Red) --- 4 (Green)
```

9. KANREN

```
from kanren import run, fact, var
from kanren.assoccomm import eq_assoccomm as eq
from kanren.assoccomm import commutative, associative
addition = "add"
multiplication = "mul"
fact(commutative, multiplication)
fact(commutative, addition)
fact(associative, multiplication)
fact(associative, addition)
x, y, z = var('a'), var('b'), var('c')
originalPattern=(multiplication,(addition, z, x),y)
ex1 = (multiplication, 9, (addition, 5, 1))
ex2 = (addition,5,(multiplication,8,1))
ex3=(multiplication,59,(addition,234,34))
print(run(0, (x, y, z), eq(originalPattern,ex1)))
print(run(0, (x, y, z), eq(originalPattern,ex2)))
print(run(0, (x, y, z), eq(originalPattern,ex3)))
```

OUTPUT

```
((1,9,5),(5,9,1),(1,9,5))
()
((34,59,234),(234,59,34),(34,59,234))
```

10. KNOWLEDGE REPRESENTATION

a. likes(ram,mango)

Query- likes(ram, What)

b. girl(seema)

Query- girl(Who)

c. like(bill,cindy)

Query- like(Who,cindy)

d. red(rose)

Query- red(What)

e. owns(john,gold)

Query- owns(Who,What)