Relationship

```
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,parent,child):
        self.family[parent].append(child)

    def death(self,name):
        self.dead.add(name)

    def inheritance(self):
        ans=[]
        def depth(current):
            if current not in self.dead:
                ans.append(current)
                for child in self.family[current]:
                    depth(child)
        depth(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)
```

Waterjug problem

```
from collections import defaultdict
def waterjug(amt1,amt2,jug1=4,jug2=3,aim=2,visited=defaultdict(bool)):

    if(amt1==aim and amt2==0) or (amt1==0 and amt2==aim):
        print(amt1,amt2)
```

```
        return True

    if visited[(amt1,amt2)]==False :
        print(amt1,amt2)
        visited[(amt1,amt2)]=True
        return (waterjug(amt1,0) or waterjug(0,amt2) or waterjug(jug1,amt2) or waterjug(amt1,jug2)
or (waterjug(amt1+min(amt2,(jug1-amt1)),amt2-min(amt2,(jug1-amt1)))) or
waterjug(amt1-min(amt1,(jug2-amt2)),amt2+min(amt1,(jug2-amt2))))

    else:
        return False
print("Steps")
waterjug(0,0)
```

4 queens

```
def is_safe(board,row,col):
    return all(board[i] != col and abs(board[i]-col)!=row-i for i in range(row))

def solve_n_queens(n,row=0,board=[]):
    if row == n:
        return [board[:]]
    solutions=[]

    for col in range(n):
        if is_safe(board,row,col):
            solutions.extend(solve_n_queens(n,row+1,board+[col]))

    return solutions

solutions=solve_n_queens(4)
print(solutions)
```

Depth First Search

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

visited=set()
def dfs(graph,visited,node):
    if node not in visited:
        print(node)
```

```
        visited.add(node)

        for neighbour in graph[node]:
            dfs(graph,visited,neighbour)

dfs(graph,visited,4)
```

Best First Search

```
from queue import PriorityQueue

def bfs(graph, start, goal, heuristic):
    priority_queue = PriorityQueue()
    priority_queue.put((heuristic[start], [start]))

    while not priority_queue.empty():
        _, path = priority_queue.get()
        current_node = path[-1]

        if current_node == goal:
            return path

        for neighbor in graph[current_node]:
            if neighbor not in path:
                new_path = path + [neighbor]
                priority_queue.put((heuristic[neighbor], new_path))

# Example usage:
graph = {
    'S': ['A', 'B'],
    'A': ['C','D'],
    'C':[],
    'D':[],
    'B': ['E','F'],
    'E': ['H'],
    'F': ['I','G'],
    'I':[],
    'G':[],
    'H':[]
}

start_node = 'S'
goal_node = 'G'
```

```
heuristic = {'A':12,'B':4,'C':7,'D':3,'E':8,'F':2,'H':4,'I':9,'S':13,'G':0}
print(heuristic)# Simple heuristic (number of neighbors)

bfs_path = bfs(graph, start_node, goal_node, heuristic)
print(f'BFS Path from {start_node} to {goal_node}: {bfs_path}')
```

Depth Limit Search

```
def depth_limited_search(graph, start, goal, depth_limit):
    visited = set()

    def dfs(node, depth):
        if depth > depth_limit:
            return False
        if node == goal:
            return True
        if node in visited:
            return False
        visited.add(node)
        for neighbor in graph.get(node, []):
            if dfs(neighbor, depth + 1):
                return True

        return False

    return dfs(start, 0)

# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': ['G'],
    'G': []
}

start_node = 'A'
goal_node = 'G'
depth_limit = 2
```

```
result = depth_limited_search(graph, start_node, goal_node, depth_limit)

if result:
    print(f"There is a path from {start_node} to {goal_node} within the depth limit.")
else:
    print(f"No path found from {start_node} to {goal_node} within the depth limit.")
```

Graph Colouring

```
def greedy_coloring(graph):
    colors = {}
    available_colors = ['red', 'green', 'blue', 'yellow']

    for vertex in graph:
        used_colors = set(colors[neighbor] for neighbor in graph[vertex] if neighbor in colors)
        for color in available_colors:
            if color not in used_colors:
                break

        colors[vertex] = color

    return colors


if __name__ == "__main__":

    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'C', 'D'],
        'C': ['A','B', 'D'],
        'D': [ 'B','C','E'],
        'E': ['D']
    }

    coloring_result = greedy_coloring(graph)

    print("Vertex\tColor")
    for vertex, color in coloring_result.items():
        print(f"{vertex}\t{color}")
```