```
# -*- coding: utf-8 -*-
"""2320040009 - shaik hanifa (Py code file 1).ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/19URfj9awH7BLk0uyiSvOHZWWy9MiX2X7

# 1. UNIFORMED SEARCH TECHNIQUE

***DFS***
"""

def dfs(g, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            print(node)
            visited.add(node)
            stack.extend(reversed(g[node]))

g = {
    10: [20, 30],
    20: [10, 40, 50],
    30: [10, 50],
    40: [20, 60],
    50: [20, 30],
    60: [40]
}
dfs(g, 10)

"""***BFS***"""

from collections import deque

def bfs(g, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node)
            visited.add(node)
            for n in g[node]:
                if n not in visited:
                    queue.append(n)

g = {
    10: [20, 30],
    20: [10, 40, 50],
    30: [10, 50],
    40: [20, 60],
    50: [20, 30],
    60: [40]
}
bfs(g, 10)

"""**IDDFS**"""

def iddfs(g, start, max_depth):
    def dfs(node, depth, visited):
        if depth == 0:
            return
        print(node)
```

```
        visited.add(node)
        for neighbor in g[node]:
            if neighbor not in visited:
                dfs(neighbor, depth - 1, visited)

    for depth in range(1, max_depth + 1):
        print(f"Depth {depth}:")
        visited = set()
        dfs(start, depth, visited)
        print()

# Graph with updated numbers as nodes
g = {
    10: [20, 30],
    20: [10, 40, 50],
    30: [10, 50],
    40: [20, 60],
    50: [20, 30],
    60: [40]
}

# Start IDDFS from node 10 with a maximum depth limit of 3
iddfs(g, 10, 3)

"""# 2.INFORMED SEARCH TECHNIQUE

**A*SEARCH**
"""

import heapq

def astar(graph, start, goal, h):
    queue = [(h[start], start, 0)]
    visited = set()
    while queue:
        f, node, g = heapq.heappop(queue)
        if node in visited:
            continue
        print(node)
        visited.add(node)
        if node == goal:
            break
        for neighbor, cost in graph[node]:
            if neighbor not in visited:
                new_g = g + cost
                new_f = new_g + h[neighbor]
                heapq.heappush(queue, (new_f, neighbor, new_g))

# Graph: (neighbor, cost) pairs
graph = {
    10: [(20, 1), (30, 4)],
    20: [(40, 1), (50, 4)],
    30: [(50, 2)],
    40: [(60, 2)],
    50: [(60, 1)],
    60: []
}

# Heuristic function
h = {
    10: 5,
    20: 3,
    30: 2,
    40: 2,
    50: 1,
    60: 0
}
```

```
astar(graph, 10, 60, h)

"""**BFS(Best first search)**"""

import heapq

def best_first_search(graph, start, goal, h):
    queue = [(h[start], start)]
    visited = set()

    while queue:
        _, node = heapq.heappop(queue)
        if node in visited:
            continue
        print(node)
        visited.add(node)
        if node == goal:
            break
        for neighbor, _ in graph[node]:
            if neighbor not in visited:
                heapq.heappush(queue, (h[neighbor], neighbor))

graph = {
    10: [(20, 1), (30, 4)],
    20: [(40, 1), (50, 4)],
    30: [(50, 2)],
    40: [(60, 2)],
    50: [(60, 1)],
    60: []
}

h = {
    10: 5,
    20: 3,
    30: 2,
    40: 2,
    50: 1,
    60: 0
}

best_first_search(graph, 10, 60, h)

"""# 3.ADVERSAL SEARCH TECHNIQUE

**Alpha Beat prunning**
"""

import math

MIN = -math.inf
MAX = math.inf

def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta):
    if depth == 3:
        return values[node_index]

    if maximizing_player:
        max_eval = MIN
        for i in range(2):
            eval = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
```

```
        min_eval = MAX
        for i in range(2):
            eval = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

values = [4, 7, 8, 10, 2, 3, 1, -2]
result = alpha_beta(0, 0, True, values, MIN, MAX)
print("The optimal value after applying alpha-beta pruning is:", result)

"""**min max program**"""

def minimax(depth, is_maximizing, values):
    if depth == len(values):
        return values[depth - 1]

    if is_maximizing:
        best_value = float('-inf')
        for i in range(2):
            val = minimax(depth + 1, False, values)
            best_value = max(best_value, val)
        return best_value
    else:
        best_value = float('inf')
        for i in range(2):
            val = minimax(depth + 1, True, values)
            best_value = min(best_value, val)
        return best_value

values = [7, 15, 9, 3, 5, 11, 18, 4]
result = minimax(1, True, values)
print("The optimal value calculated by the minimax algorithm is:", result)
```