***Python Code File 1***

from collections import deque

import heapq

# Uninformed Search: Breadth-First Search (BFS)

def bfs(graph, start, goal):

queue = deque([[start]])

visited = set()

while queue:

path = queue.popleft()

node = path[-1]

if node in visited:

continue

visited.add(node)

if node == goal:

return path

for neighbor in graph.get(node, []):

new\_path = list(path)

new\_path.append(neighbor)

queue.append(new\_path)

return None

# Informed Search: A\* Search

def a\_star(graph, start, goal, h):

open\_list = []

heapq.heappush(open\_list, (0, start))

came\_from = {}

g\_score = {start: 0}

f\_score = {start: h(start)}

while open\_list:

\_, current = heapq.heappop(open\_list)

if current == goal:

return reconstruct\_path(came\_from, current)

for neighbor, cost in graph.get(current, []):

tentative\_g\_score = g\_score[current] + cost

if tentative\_g\_score < g\_score.get(neighbor, float('inf')):

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = tentative\_g\_score + h(neighbor)

heapq.heappush(open\_list, (f\_score[neighbor], neighbor))

return None

def reconstruct\_path(came\_from, current):

path = [current]

while current in came\_from:

current = came\_from[current]

path.append(current)

return path[::-1]

# Adversarial Search: Minimax Algorithm with Alpha-Beta Pruning

def minimax(board, depth, alpha, beta, maximizing\_player):

if depth == 0 or is\_terminal(board):

return evaluate(board)

if maximizing\_player:

max\_eval = float('-inf')

for move in get\_all\_possible\_moves(board, True):

evaluation = minimax(move, depth - 1, alpha, beta, False)

max\_eval = max(max\_eval, evaluation)

alpha = max(alpha, evaluation)

if beta <= alpha:

break

return max\_eval

else:

min\_eval = float('inf')

for move in get\_all\_possible\_moves(board, False):

evaluation = minimax(move, depth - 1, alpha, beta, True)

min\_eval = min(min\_eval, evaluation)

beta = min(beta, evaluation)

if beta <= alpha:

break

return min\_eval

def is\_terminal(board):

# Implementation depends on the game

pass

def evaluate(board):

# Implementation depends on the game

pass

def get\_all\_possible\_moves(board, is\_maximizing\_player):

# Implementation depends on the game

pass

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Graph for BFS and A\*

graph = {

'A': [('B', 1), ('C', 3)],

'B': [('D', 1), ('E', 5)],

'C': [('F', 2)],

'D': [],

'E': [('F', 1)],

'F': []

}

# BFS

print("BFS Path:", bfs(graph, 'A', 'F'))

# Heuristic function for A\*

def heuristic(node):

h\_values = {'A': 5, 'B': 4, 'C': 2, 'D': 6, 'E': 2, 'F': 0}

return h\_values.get(node, float('inf'))

# A\* Search

print("A\* Path:", a\_star(graph, 'A', 'F', heuristic))