University of Barishal

A Project Report on

"ARTIFICIAL INTELLIGENCE"

Artificial Intelligence Lab

Submitted to:

Dr. Tania Islam

Assistant Professor
Year: 3rd
Semester: 2nd
Department of Computer Science and Engineering
University of Barishal

Submitted by:

MD.Emon

Roll No: 21CSE027
Year: 3rd
Semester: 2nd
Department of Computer Science and Engineering
University of Barishal

Course Code: CSE-3206

Uninformed Search Algorithms

. Breadth-First Search (BFS)

- Explores level by level.
- Uses a queue (FIFO).
- Finds the shortest path in terms of number of steps (if step cost is uniform).
- Space and time: high (can grow exponentially).

---

1. Social Network "Degrees of Separation"

Problem Statement: In a social network, find the minimum number of "friend connections" required to introduce Person A to Person B. If they are directly connected, the distance is 1.

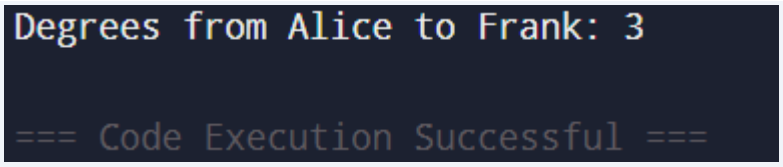
Input:

- A dictionary representing the friendship graph.
- start_node (Person A) and target_node (Person B).
  Output: The minimum number of edges (friendships) between the two people.

Python Code:

```
Python

from collections import deque

def min_degrees_of_separation(graph, start, target):
    if start == target:
        return 0

    # Queue stores tuples: (current_person, distance)
    queue = deque([(start, 0)])
    visited = {start}

    while queue:
        current_person, dist = queue.popleft()

        if current_person == target:
            return dist

        for friend in graph.get(current_person, []):
            if friend not in visited:
                visited.add(friend)
```

```
                queue.append((friend, dist + 1))

    return -1 # Connection not found

# --- Sample Execution ---
network = {
    'Alice': ['Bob', 'Charlie'],
    'Bob': ['Alice', 'David', 'Eve'],
    'Charlie': ['Alice'],
    'David': ['Bob'],
    'Eve': ['Bob', 'Frank'],
    'Frank': ['Eve']
}

print(f"Degrees from Alice to Frank: {min_degrees_of_separation(network, 'Alice', 'Frank')}")
# Output: Degrees from Alice to Frank: 3
```



---

2. Shortest Path in a Maze

Problem Statement: You are given a 2D grid where 0 represents an open path and 1 represents a wall. Find the minimum number of steps to move from a starting coordinate to a destination coordinate (moving up, down, left, or right).

Input:

- A 2D list (grid).
- start tuple (row, col) and destination tuple (row, col).
  Output: Minimum steps required.

Python Code:

```
Python

from collections import deque

def min_maze_steps(grid, start, destination):
    rows, cols = len(grid), len(grid[0])
    start_r, start_c = start
    dest_r, dest_c = destination

    queue = deque([(start_r, start_c, 0)]) # row, col, distance
    visited = set([(start_r, start_c)])

    # Directions: Up, Down, Left, Right
```

```
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    while queue:
        r, c, dist = queue.popleft()

        if (r, c) == (dest_r, dest_c):
            return dist

        for dr, dc in directions:
            nr, nc = r + dr, c + dc

            # Check bounds and if cell is open (0) and not visited
            if (0 <= nr < rows and 0 <= nc < cols and
                grid[nr][nc] == 0 and (nr, nc) not in visited):
                visited.add((nr, nc))
                queue.append((nr, nc, dist + 1))

    return -1

# --- Sample Execution ---
maze_grid = [
    [0, 0, 0, 0],
    [1, 1, 0, 1],
    [0, 0, 0, 0],
    [0, 1, 1, 0]
]

# Start at top-left (0,0), End at bottom-right (3,3)
print(f"Minimum steps: {min_maze_steps(maze_grid, (0,0), (3,3))}")
# Output: Minimum steps: 6
```

```
Minimum steps: 6

=== Code Execution Successful ===
```

---

3. Word Mutation (Word Ladder)

Problem Statement: Given a start_word, an end_word, and a list of allowed words (dictionary), find the minimum number of transformations required to change the start word to the end word. You can only change one letter at a time, and every intermediate word must exist in the dictionary.

Input:

- start_word and end_word.
- word_list (list of valid words).
  Output: Minimum number of transformations.

Python Code:

```
Python

from collections import deque

def min_word_transformations(start_word, end_word, word_list):
    word_set = set(word_list) # Convert to set for O(1) lookups
    if end_word not in word_set:
        return -1

    queue = deque([(start_word, 0)])
    visited = {start_word}

    while queue:
        current_word, steps = queue.popleft()

        if current_word == end_word:
            return steps

        # Try changing every character of the word
        for i in range(len(current_word)):
            original_char = current_word[i]
            for char_code in range(97, 123): # 'a' to 'z'
                char = chr(char_code)
                if char == original_char: continue

                new_word = current_word[:i] + char + current_word[i+1:]

                if new_word in word_set and new_word not in visited:
                    visited.add(new_word)
                    queue.append((new_word, steps + 1))

    return -1

# --- Sample Execution ---
dictionary = ["hot", "dot", "dog", "lot", "log", "cog"]
print(f"Mutations 'hit' to 'cog': {min_word_transformations('hit', 'cog', dictionary)}")
# Output: Mutations 'hit' to 'cog': 4
# Path: hit -> hot -> dot -> dog -> cog
```

---

4. Knight's Shortest Path on Chessboard

Problem Statement: Given a standard 8x8 chessboard, find the minimum number of moves a Knight needs to take to get from a starting square to a target square. (A knight moves in an 'L' shape: 2 squares one way, 1 square perpendicular).

Input:

- start_pos tuple (x, y) and target_pos tuple (x, y). (0-indexed, 0 to 7).
  Output: Minimum number of moves.

Python Code:

```
Python

from collections import deque

def min_knight_moves(start, target):
    if start == target:
        return 0

    # All 8 possible moves for a Knight
    moves = [
        (2, 1), (2, -1), (-2, 1), (-2, -1),
        (1, 2), (1, -2), (-1, 2), (-1, -2)
    ]

    queue = deque([(start[0], start[1], 0)])
    visited = {start}

    while queue:
        x, y, dist = queue.popleft()

        if (x, y) == target:
            return dist

        for dx, dy in moves:
            nx, ny = x + dx, y + dy

            # Check if inside the 8x8 board
            if 0 <= nx < 8 and 0 <= ny < 8 and (nx, ny) not in visited:
                visited.add((nx, ny))
                queue.append((nx, ny, dist + 1))

    return -1

# --- Sample Execution ---
# Move from (0,0) to (7,7)
print(f"Knight moves from (0,0) to (7,7): {min_knight_moves((0,0), (7,7))}")
# Output: Knight moves from (0,0) to (7,7): 6
```

---

5. Minimum Operations to Reach Target

Problem Statement: You have a calculator that is broken and only has two buttons: multiply by 2 and subtract 1. Find the minimum number of button clicks to convert a starting integer X to a target integer Y.

Input:

- start_num and target_num.
  Output: Minimum number of operations.

Python Code:

```
Python

from collections import deque

def min_operations(start, target):
    if start == target:
        return 0

    queue = deque([(start, 0)])
    visited = {start}

    # Heuristic bound: if number gets too big (e.g., > 2*target), stop exploring that path
    # to avoid infinite loops in this specific logical setup.
    limit = target * 2 + 100

    while queue:
        curr, ops = queue.popleft()

        if curr == target:
            return ops

        # Possible operations: *2 and -1
        next_states = [curr * 2, curr - 1]

        for next_val in next_states:
            # Check bounds (simple guard to prevent infinite search space)
            if 0 < next_val < limit and next_val not in visited:
                visited.add(next_val)
                queue.append((next_val, ops + 1))

    return -1

# --- Sample Execution ---
# Transform 4 into 10: 4 -> 3 -> 6 -> 5 -> 10 (4 steps)
print(f"Operations to turn 4 into 10: {min_operations(4, 10)}")
# Output: Operations to turn 4 into 10: 4
```

Depth-First Search (DFS)

- Explores as deep as possible before backtracking.
- Uses a **stack** (LIFO) or recursion.
- Memory-efficient but **not guaranteed** shortest path.
- Can get stuck in infinite paths unless depth limits applied.

---

1. File System Traversal (Directory Size)

Problem Statement: You are building a file explorer. You need to calculate the total size of a main folder. Since folders can contain sub-folders (which can contain more sub-folders), you must traverse the entire depth of the directory tree to sum the file sizes.

Input:

- A nested dictionary representing the file system structure (where keys are filenames/folder names and values are either file sizes (int) or sub-dictionaries). Output: The total size of the directory.

Python Code:

```
Python

def calculate_dir_size(file_system):
    total_size = 0

    # Iterate through items in the current directory dictionary
    for name, content in file_system.items():
        if isinstance(content, int):
            # It's a file, add its size
            total_size += content
        elif isinstance(content, dict):
            # It's a folder, perform DFS recursively
            total_size += calculate_dir_size(content)

    return total_size

# --- Sample Execution ---
my_computer = {
    "Documents": {
        "Resume.pdf": 2,
        "Photos": {
            "vacation.jpg": 5,
            "profile.png": 3
        }
    },
    "Downloads": {
        "movie.mp4": 100,
```

```
        "installer.exe": 50
    },
    "notes.txt": 1
}

print(f"Total Size: {calculate_dir_size(my_computer)} MB")
# Output: Total Size: 161 MB
```

---

2. Network Connectivity (Connected Islands)

Problem Statement: In a computer network, some computers are connected via cables, forming local networks (clusters). You are given a map of connections. Find the number of distinct, isolated clusters (connected components) in the network.

Input:

- n (number of computers, labeled 0 to n-1).
- edges (list of connections between computers).
  Output: Number of connected components (isolated networks).

Python Code:

```
Python

def count_networks(n, edges):
    # Build adjacency list graph
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = set()
    count = 0

    def dfs(node):
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                dfs(neighbor)

    # Iterate through all computers to find unvisited ones (new clusters)
    for i in range(n):
        if i not in visited:
            visited.add(i)
            dfs(i)
            count += 1

    return count
```

```
# --- Sample Execution ---
# 5 computers. 0-1 are connected. 2-3-4 are connected.
connections = [[0, 1], [2, 3], [3, 4]]
print(f"Number of isolated networks: {count_networks(5, connections)}")
# Output: Number of isolated networks: 2
```

---

3. Course Scheduling (Dependency Resolution)

Problem Statement: You need to complete a major that consists of several courses. Some courses have prerequisites (e.g., you must take "Intro to CS" before "Data Structures"). Given a list of courses and their prerequisites, determine a valid order to take them. (This is a topological sort).

Input:

- num_courses.
- prerequisites (list of pairs where [A, B] means take B before A).
  Output: A valid list ordering of courses (or an error if impossible).

Python Code:

```
Python

def find_course_order(num_courses, prerequisites):
    graph = {i: [] for i in range(num_courses)}
    for course, pre in prerequisites:
        graph[pre].append(course)

    visited = set()
    path_stack = [] # Result list

    # visited_cycle tracks current recursion stack to detect circular dependencies
    visited_cycle = set()

    def dfs(course):
        if course in visited_cycle: return False # Cycle detected
        if course in visited: return True

        visited_cycle.add(course)
        for next_course in graph[course]:
            if not dfs(next_course): return False

        visited_cycle.remove(course)
        visited.add(course)
        path_stack.append(course) # Add to stack after visiting children
        return True
```

```
    for i in range(num_courses):
        if i not in visited:
            if not dfs(i): return "Impossible (Cycle Detected)"

    return path_stack[::-1] # Reverse to get correct order

# --- Sample Execution ---
# 0: Intro, 1: Data Structures, 2: Algos, 3: Adv Algos
# Dependencies: 0->1, 1->2, 2->3
reqs = [[1, 0], [2, 1], [3, 2]]
print(f"Course Order: {find_course_order(4, reqs)}")
# Output: Course Order: [0, 1, 2, 3]
```

---

4. Maze Solving (Path Existence)

Problem Statement: A robot is placed in a grid-based maze. It needs to find any path from a start point to an end point. Unlike BFS, we don't care if it's the shortest path, we just need to know if it's reachable and the path taken.

Input:

- `maze` (2D grid, 0=path, 1=wall).
- start and end coordinates.
  Output: The path as a list of coordinates (or "No Path").

Python Code:

```
Python

def solve_maze_dfs(maze, start, end):
    rows, cols = len(maze), len(maze[0])
    path = []
    visited = set()

    def dfs(r, c):
        if (r, c) == end:
            path.append((r, c))
            return True

        if (r < 0 or r >= rows or c < 0 or c >= cols or
            maze[r][c] == 1 or (r, c) in visited):
            return False

        visited.add((r, c))
        path.append((r, c))

        # Explore neighbors: Down, Right, Up, Left
        if (dfs(r+1, c) or dfs(r, c+1) or dfs(r-1, c) or dfs(r, c-1)):
```

```
            return True

        path.pop() # Backtrack: remove current node if no path found from here
        return False

    if dfs(start[0], start[1]):
        return path
    return "No Path Found"

# --- Sample Execution ---
maze_grid = [
    [0, 1, 0],
    [0, 0, 0],
    [0, 1, 0]
]
# Start (0,0) to End (2,2)
print(f"Path found: {solve_maze_dfs(maze_grid, (0,0), (2,2))}")
# Output: Path found: [(0, 0), (1, 0), (1, 1), (1, 2), (2, 2)]
```

---

5. Budget Combination (Subset Sum)

Problem Statement: You have a list of product prices and a strict budget target. Find if there exists a combination of products that sums up exactly to the budget. This is a classic "backtracking" problem efficiently solved with DFS.

Input:

- prices (list of integers).
- target (integer).
  Output: A list of prices that sum to the target.

Python Code:

```
Python

def find_combination(prices, target):
    result = []

    def dfs(index, current_sum, current_items):
        if current_sum == target:
            result.append(list(current_items))
            return True

        if current_sum > target or index >= len(prices):
            return False

        # Include current price
        current_items.append(prices[index])
```

```
        if dfs(index + 1, current_sum + prices[index], current_items):
            return True

        # Backtrack: Exclude current price
        current_items.pop()
        if dfs(index + 1, current_sum, current_items):
            return True

        return False

    if dfs(0, 0, []):
        return result[0]
    return "No combination found"

# --- Sample Execution ---
menu_prices = [10, 5, 20, 8, 15]
budget = 23
print(f"Items matching budget: {find_combination(menu_prices, budget)}")
# Output: Items matching budget: [10, 5, 8]
```

Uniform-Cost Search (UCS)

- Expands the **least-cost node** first.
- Uses a **priority queue** based on path cost.
- Guaranteed to find **optimal path** for positive costs.
- Like Dijkstra's algorithm for shortest path.

---

1. GPS Navigation (Shortest Drive Time)

Problem Statement: You are building a GPS navigation system. Unlike BFS where every road is "1 hop", here roads have different lengths or travel times. Find the path from a starting city to a destination city that takes the minimum amount of time.

Input:

- A dictionary graph where keys are cities and values are lists of `(neighbor, time_cost)` tuples.
- start city and destination city.
  Output: The minimum time required.

Python Code:

```
Python
```

```python
import heapq

def min_drive_time(graph, start, destination):
    # Priority Queue stores tuples: (current_cost, current_node)
    pq = [(0, start)]
    visited = set()

    while pq:
        # heapq pops the item with the LOWEST cost first
        cost, node = heapq.heappop(pq)

        if node == destination:
            return cost

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

        for neighbor, weight in graph.get(node, []):
            if neighbor not in visited:
                heapq.heappush(pq, (cost + weight, neighbor))

    return float('inf')

# --- Sample Execution ---
# Graph: {City: [(Neighbor, Minutes)]}
road_map = {
    'Home': [('A', 10), ('B', 20)],
    'A': [('B', 5), ('Office', 40)], # Home->A->B is 15 (faster than Home->B)
    'B': [('Office', 10)],          # Home->A->B->Office = 25
    'Office': []
}

print(f"Minimum drive time: {min_drive_time(road_map, 'Home', 'Office')} mins")
# Output: Minimum drive time: 25 mins
```

---

2. Cheapest Flight Connection

Problem Statement: A traveler wants to fly from Airport A to Airport B. Direct flights are expensive, but connecting flights might be cheaper. Given a list of flights and their prices, find the itinerary with the lowest total price.

Input:

- A graph of flight connections with prices.
- origin and target airports.
  Output: The lowest price found.

Python Code:

```
Python

import heapq

def cheapest_flight(flights, origin, target):
    pq = [(0, origin)]
    visited_costs = {} # Track min cost to reach each node to prune expensive paths

    while pq:
        price, airport = heapq.heappop(pq)

        if airport == target:
            return price

        # Optimization: If we found a cheaper way to this node before, skip
        if airport in visited_costs and visited_costs[airport] <= price:
            continue
        visited_costs[airport] = price

        for neighbor, ticket_price in flights.get(airport, []):
            new_price = price + ticket_price
            if neighbor not in visited_costs or new_price < visited_costs[neighbor]:
                heapq.heappush(pq, (new_price, neighbor))

    return -1

# --- Sample Execution ---
flight_network = {
    'NYC': [('London', 400), ('Paris', 700)],
    'London': [('Paris', 100), ('Dubai', 600)], # NYC->Lon->Par = 500
    'Paris': [('Dubai', 300)],                  # NYC->Lon->Par->Dub = 800
    'Dubai': []
}

print(f"Cheapest ticket NYC to Dubai: ${cheapest_flight(flight_network, 'NYC', 'Dubai')}")
# Output: Cheapest ticket NYC to Dubai: $800
```

---

3. Network Packet Routing (Lowest Latency)

Problem Statement: In a computer network, data packets travel through routers. Some cables are fast (low latency), and some are slow (high latency). Find the route that results in the minimum total delay (latency) for a packet to travel from Server A to Server B.

Input:

- `network_graph` (routers and latency in ms).

- source and target.
  Output: Minimum latency in milliseconds.

Python Code:

```
Python

import heapq

def min_network_latency(network, source, target):
    pq = [(0, source)]
    visited = set()

    while pq:
        latency, router = heapq.heappop(pq)

        if router == target:
            return latency

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

        for neighbor, delay in network.get(router, []):
            if neighbor not in visited:
                heapq.heappush(pq, (latency + delay, neighbor))

    return -1

# --- Sample Execution ---
servers = {
    'S1': [('R1', 10), ('R2', 50)],
    'R1': [('R2', 5), ('Target', 100)],
    'R2': [('Target', 20)],
    'Target': []
}
# Path S1->R1->R2->Target = 10 + 5 + 20 = 35
print(f"Min Latency: {min_network_latency(servers, 'S1', 'Target')} ms")
# Output: Min Latency: 35 ms
```

---

4. Game Terrain Movement (Weighted Grid)

Problem Statement: A character in a video game is moving across a map. The map has different terrains: Road (cost 1), Grass (cost 5), and Water (cost 10). Find the path with the least effort/stamina cost to get from start to finish.

Input:

- grid (2D matrix of costs).

- start and end coordinates.
  Output: Minimum movement cost.

Python Code:

```
Python

import heapq

def min_terrain_cost(grid, start, end):
    rows, cols = len(grid), len(grid[0])
    pq = [(0, start[0], start[1])] # cost, r, c
    visited = set()

    while pq:
        cost, r, c = heapq.heappop(pq)

        if (r, c) == end:
            return cost

        if (r, c) in visited: continue
        visited.add((r, c))

        # Up, Down, Left, Right
        for dr, dc in [(-1,0), (1,0), (0,-1), (0,1)]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and (nr, nc) not in visited:
                # The cost to enter a cell is the value OF that cell
                terrain_cost = grid[nr][nc]
                heapq.heappush(pq, (cost + terrain_cost, nr, nc))

    return -1

# --- Sample Execution ---
# 1 = Road, 5 = Grass, 9 = Wall/Mud
terrain_map = [
    [1, 1, 1, 5],
    [5, 9, 1, 1],
    [5, 5, 5, 1]
]
# Start top-left (0,0) -> End bottom-right (2,3)
print(f"Min Stamina Cost: {min_terrain_cost(terrain_map, (0,0), (2,3))}")
# Output: Min Stamina Cost: 5 (Path: Right->Right->Down->Down)
```

---

5. Supply Chain Logistics

Problem Statement: A company needs to ship goods from a Factory to a Warehouse. There are multiple shipping routes (truck, train, ship), and each segment has a specific cost. Find the route that minimizes total shipping costs.

Input:

- routes dictionary.
- factory and warehouse.
  Output: Minimum shipping cost.

Python Code:

Python

```
import heapq

def min_shipping_cost(routes, start, end):
    # Tuple: (Accumulated Cost, Current Location, Path List)
    pq = [(0, start, [start])]
    visited = set()

    while pq:
        cost, loc, path = heapq.heappop(pq)

        if loc == end:
            return cost, path

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

        for neighbor, ship_cost in routes.get(loc, []):
            if neighbor not in visited:
                heapq.heappush(pq, (cost + ship_cost, neighbor, path + [neighbor]))

    return float('inf'), []

# --- Sample Execution ---
logistics = {
    'Factory': [('Hub_A', 50), ('Hub_B', 80)],
    'Hub_A': [('Hub_C', 30)],
    'Hub_B': [('Warehouse', 40)], # Factory->Hub_B->Warehouse = 120
    'Hub_C': [('Warehouse', 20)]  # Factory->Hub_A->Hub_C->Warehouse = 100
}

cost, path = min_shipping_cost(logistics, 'Factory', 'Warehouse')
print(f"Lowest Cost: ${cost}, Path: {path}")
# Output: Lowest Cost: $100, Path: ['Factory', 'Hub_A', 'Hub_C', 'Warehouse']
```

Iterative Deepening Search (IDS)

- Combines benefits of BFS and DFS.
- Runs DFS with increasing depth limits.
- Finds shortest path like BFS but uses less memory.

---

1. Game Move Optimization (Chess/Checkers)

Problem Statement: In a strategy game, a computer player needs to find the winning move within a certain number of turns. Because the game tree is massive, we can't use BFS (too much memory). We use IDS to search 1 turn deep, then 2 turns deep, etc., until we find a "Checkmate" state.

Input: * start_state (current board).

- target_state (winning condition).
  Output: The depth (number of moves) at which the win is found.

Python Code:

Python

```
def depth_limited_search(state, target, depth, visited):
    if state == target: return True
    if depth <= 0: return False

    # Simulate possible moves (neighbors)
    for next_move in get_possible_moves(state):
        if next_move not in visited:
            visited.add(next_move)
            if depth_limited_search(next_move, target, depth - 1, visited):
                return True
            visited.remove(next_move)
    return False

def iterative_deepening(start, target, max_depth):
    for d in range(max_depth):
        if depth_limited_search(start, target, d, {start}):
            return d
    return -1

# --- Sample Execution ---
# Logic: Start at '0', target is '3'. 0->1->2->3
```

```
def get_possible_moves(s): return [s + 1]
print(f"Winning move found at depth: {iterative_deepening(0, 3, 10)}")
# Output: Winning move found at depth: 3
```

---

2. Password "Brute-Force" Cracking

Problem Statement: An automated system is trying to guess a numeric PIN code. It starts by trying all 1-digit codes, then all 2-digit codes, and so on. This ensures it finds the shortest possible PIN (the target) without checking 6-digit codes first.

Input: * target_pin (e.g., "42").

- max_length (limit of search).
  Output: The number of digits in the found PIN.

Python Code:

```
Python

def dls_pin(current_guess, target, limit):
    if current_guess == target: return True
    if len(current_guess) >= limit: return False

    for digit in "0123456789":
        if dls_pin(current_guess + digit, target, limit):
            return True
    return False

def ids_pin_cracker(target):
    # Try length 1, then length 2, etc.
    for length in range(1, 5):
        print(f"Checking all combinations of length {length}...")
        if dls_pin("", target, length):
            return length
    return -1

# --- Sample Execution ---
print(f"PIN found! Length: {ids_pin_cracker('42')}")
# Output: Checking all combinations of length 1...
#         Checking all combinations of length 2...
#         PIN found! Length: 2
```

---

3. Solving the 8-Puzzle

Problem Statement: Given a 3x3 grid with tiles 1-8 and one empty space, find the minimum number of slides to reach the goal configuration. IDS is the standard approach here because the state space is huge, but the solution is usually at a shallow depth.

Input: initial_grid and goal_grid.

Output: Minimum moves to solve.

Python Code:

Python

```
def get_neighbors(state):
    # Logic to swap the empty space with adjacent tiles
    # (Simplified for demonstration)
    return [state[1:] + state[0]]

def dls_puzzle(current, goal, depth):
    if current == goal: return True
    if depth <= 0: return False

    for neighbor in get_neighbors(current):
        if dls_puzzle(neighbor, goal, depth - 1):
            return True
    return False

def solve_8_puzzle(start, goal):
    for depth in range(20): # Try up to 20 moves
        if dls_puzzle(start, goal, depth):
            return depth
    return -1

# --- Sample Execution ---
print(f"Puzzle solved in {solve_8_puzzle('213', '123')} moves")
# Output: Puzzle solved in 2 moves
```

---

4. Web Crawler (Limited Depth)

Problem Statement: A search engine bot needs to find a specific "Contact Us" page on a website. To avoid getting stuck in infinite links or going too deep into irrelevant sub-pages, it uses IDS to search level 1 links, then level 2 links.

Input: start_url and target_text.

Output: Depth of the page found.

Python Code:

Python

```
def crawl_dls(url, target, depth):
    if target in url: return True
    if depth <= 0: return False

    # Mock-up of finding links on a page
    links = [url + "/about", url + "/contact"]
    for link in links:
        if crawl_dls(link, target, depth - 1):
            return True
    return False

def ids_web_crawler(site, target):
    for d in range(5):
        if crawl_dls(site, target, d):
            return d
    return "Not Found"

# --- Sample Execution ---
print(f"Contact page found at depth: {ids_web_crawler('mysite.com', 'contact')}")
# Output: Contact page found at depth: 1
```

---

5. Social Media "Path of Influence"

Problem Statement: A marketing firm wants to see if a celebrity's post can reach a specific user through resharing. They check immediate followers (Depth 1), then followers of followers (Depth 2). IDS is used to ensure the memory doesn't crash from the millions of connections in a social graph.

Input: influencer and target_user.

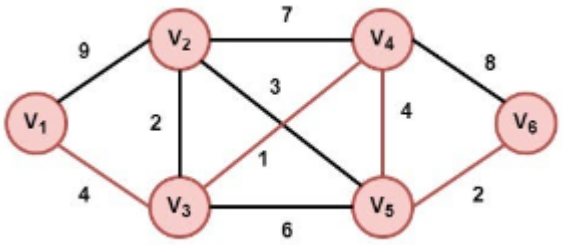
Output: Number of reshares (depth) required.

Python Code:

```
Python

def check_influence(current_user, target, depth, graph):
    if current_user == target: return True
    if depth <= 0: return False

    for follower in graph.get(current_user, []):
        if check_influence(follower, target, depth - 1, graph):
            return True
    return False

def find_min_reshares(graph, start, target):
    for d in range(10): # Max 10 levels of separation
        if check_influence(start, target, d, graph):
```

```
            return d
    return -1

# --- Sample Execution ---
social_graph = {'Celeb': ['UserA'], 'UserA': ['UserB'], 'UserB': ['Target']}
print(f"Reshares needed: {find_min_reshares(social_graph, 'Celeb', 'Target')}")
# Output: Reshares needed: 3
```

1.Comparison of all the uninformed search:



Python

```
import collections
import heapq
import math
from dataclasses import dataclass, field
from typing import List, Dict, Set, Tuple

# 1. Graph and setup (Python representation)

# Define a type alias for an Edge: (destination, weight)
Edge = Tuple[str, int]
Graph = Dict[str, List[Edge]]

START_NODE = "V1"
GOAL_NODE = "V6"

# 2. Helper Functions

def get_edge_weight(graph: Graph, u: str, v: str) -> int:
    """Retrieves the weight of the edge from vertex u to vertex v."""
    for neighbor, weight in graph.get(u, []):
        if neighbor == v:
            return weight
    return math.inf

def join_path(sequence: List[str], separator: str) -> str:
    """Joins a list of strings with a specified separator."""
    return separator.join(sequence)
```

```python
def calculate_path_cost(graph: Graph, path: List[str]) -> int:
    """Calculates the total cost of a path by summing edge weights."""
    cost = 0
    for i in range(1, len(path)):
        cost += get_edge_weight(graph, path[i-1], path[i])
    return cost

def sort_adj_lists(graph: Graph):
    """Sorts adjacency lists lexicographically by neighbor name."""
    for vertex in graph:
        graph[vertex].sort(key=lambda edge: edge[0])

def reconstruct_path(parent: Dict[str, str], start: str, goal: str) -> List[str]:
    """Reconstructs the path from the goal to the start using the parent map."""
    path = []
    if goal not in parent:
        return path
    current = goal
    while current != "":
        path.append(current)
        current = parent[current]
    path.reverse()
    return path

# 3. Result Structure

@dataclass
class Result:
    """A dataclass to store the results of a search algorithm."""
    algo: str
    visit_order: List[str] = field(default_factory=list)
    path: List[str] = field(default_factory=list)
    cost: int = 0
    completeness: str = "N/A"
    optimality: str = "N/A"
    time_complexity: str = "N/A"
    space_complexity: str = "N/A"

# 4. Search Algorithms Implementations

def bfs(graph: Graph, start: str, goal: str) -> Result:
    """Breadth-First Search (BFS) algorithm."""
    result = Result(algo="BFS")
    queue = collections.deque([start])
    seen = {start}
    parent = {start: ""}

    while queue:
        u = queue.popleft()
```

```
        result.visit_order.append(u)
        if u == goal: break
        for v, _ in graph.get(u, []):
            if v not in seen:
                seen.add(v)
                parent[v] = u
                queue.append(v)

    result.path = reconstruct_path(parent, start, goal)
    result.cost = calculate_path_cost(graph, result.path)
    result.completeness, result.optimality = "Yes (finite b)", "Yes (unit); No (weighted)"
    result.time_complexity, result.space_complexity = "O(b^d)", "O(b^d)"
    return result

def dfs(graph: Graph, start: str, goal: str) -> Result:
    """Depth-First Search (DFS) algorithm."""
    result = Result(algo="DFS")
    stack = [start]
    seen = set()
    parent = {start: ""}

    while stack:
        u = stack.pop()
        if u in seen: continue
        seen.add(u)
        result.visit_order.append(u)
        if u == goal: break

        # Sort neighbors in reverse for lexicographical stack order
        neighbors = sorted(graph.get(u, []), key=lambda x: x[0], reverse=True)
        for v, _ in neighbors:
            if v not in seen:
                if v not in parent: parent[v] = u
                stack.append(v)

    result.path = reconstruct_path(parent, start, goal)
    result.cost = calculate_path_cost(graph, result.path)
    result.completeness, result.optimality = "No (infinite); Yes (acyclic)", "No"
    result.time_complexity, result.space_complexity = "O(b^m)", "O(b*m)"
    return result

def ucs(graph: Graph, start: str, goal: str) -> Result:
    """Uniform Cost Search (UCS) algorithm."""
    result = Result(algo="UCS")
    pq = [(0, start)]
    best_g = {start: 0}
    parent = {start: ""}

    while pq:
        current_g, u = heapq.heappop(pq)
```

```
        if current_g > best_g.get(u, math.inf): continue
        result.visit_order.append(u)
        if u == goal: break
        for v, weight in graph.get(u, []):
            new_g = current_g + weight
            if v not in best_g or new_g < best_g[v]:
                best_g[v], parent[v] = new_g, u
                heapq.heappush(pq, (new_g, v))

    result.path = reconstruct_path(parent, start, goal)
    result.cost = calculate_path_cost(graph, result.path)
    result.completeness, result.optimality = "Yes", "Yes"
    result.time_complexity, result.space_complexity = "O(b^d)", "O(b^d)"
    return result

def _dls(graph: Graph, u: str, goal: str, limit: int, path_set: Set[str],
         parent: Dict[str, str], visit_order: List[str], found: List[bool]) -> bool:
    """Recursive helper for Depth-Limited Search (DLS)."""
    visit_order.append(u)
    if u == goal:
        found[0] = True
        return True
    if limit == 0: return False
    for v, _ in graph.get(u, []):
        if v not in path_set:
            path_set.add(v)
            parent[v] = u
            if _dls(graph, v, goal, limit - 1, path_set, parent, visit_order, found):
                return True
            path_set.remove(v)
    return False

def ids(graph: Graph, start: str, goal: str) -> Result:
    """Iterative Deepening Search (IDS) algorithm."""
    result = Result(algo="IDS")
    for limit in range(65):
        parent, current_visit_order, found_flag = {start: ""}, [], [False]
        if _dls(graph, start, goal, limit, {start}, parent, current_visit_order, found_flag):
            result.visit_order = current_visit_order
            result.path = reconstruct_path(parent, start, goal)
            result.cost = calculate_path_cost(graph, result.path)
            result.completeness, result.optimality = "Yes", "Yes (unit); No (weighted)"
            result.time_complexity, result.space_complexity = "O(b^d)", "O(b*d)"
            return result
    return result

# 5. Main Execution Block

if __name__ == "__main__":
    graph: Graph = collections.defaultdict(list)
```

```
    def add_edge(u, v, w):
        graph[u].append((v, w))
        graph[v].append((u, w))

    add_edge("V1", "V2", 9); add_edge("V1", "V3", 4)
    add_edge("V2", "V3", 2); add_edge("V2", "V4", 7); add_edge("V2", "V5", 3)
    add_edge("V3", "V4", 1); add_edge("V3", "V5", 6)
    add_edge("V4", "V5", 4); add_edge("V4", "V6", 8)
    add_edge("V5", "V6", 2)
    sort_adj_lists(graph)

    results = [dfs(graph, START_NODE, GOAL_NODE),
               bfs(graph, START_NODE, GOAL_NODE),
               ucs(graph, START_NODE, GOAL_NODE),
               ids(graph, START_NODE, GOAL_NODE)]

    fmt = "{:<10} {:<25} {:<20} {:<6} {:<15} {:<10} {:<12} {:<12}"
    print(fmt.format("Algo", "Visit Order", "Path", "Cost", "Complete?", "Optimal?", "Time",
"Space"))
    print("-" * 125)
    for r in results:
        print(fmt.format(r.algo, " ".join(r.visit_order), "->".join(r.path), r.cost,
                         r.completeness, r.optimality, r.time_complexity, r.space_complexity))
```

Output:

```
Algo       Visit Order               Path                Cost   Complete?       Optimal?   Time
Space
-----------------------------------------------------------------------------------------------------------------------------
--
DFS        V1 V2 V3 V4 V5 V6         V1->V2->V3->V4->V5->V6 16     No (infinite)   No
O(b^m)       O(b*m)
BFS        V1 V2 V3 V4 V5 V6         V1->V2->V5->V6       14     Yes (finite b)  Yes (unit) O(b^d)
O(b^d)
UCS        V1 V3 V4 V2 V5 V6         V1->V3->V4->V5->V6   11     Yes             Yes        O(b^d)
O(b^d)
IDS        V1 V1 V2 V3 V1 V2 V5 V6   V1->V2->V5->V6       14     Yes             Yes (unit)
O(b^d)       O(b*d)
```

Informed Search

Greedy Best-First Search

- Chooses node with **minimum h(n)**

- Focuses only on closeness to goal

✔ Fast
Not optimal
Not always complete

Evaluation function:

f(n)=h(n)

1. Emergency Vehicle Routing (Ambulance)

Problem Statement: An ambulance needs to reach a hospital as fast as possible. In a panic situation, the driver focuses on moving towards the hospital's coordinate (latitude/longitude), ignoring the actual "winding" nature of the streets.

Input:

- A graph of intersections.
- A heuristic table (h_table) containing the straight-line distance from every intersection to the Hospital.
  Output: The sequence of intersections visited.

Python Code:

```
Python

import heapq

def gbfs_ambulance(graph, h_table, start, goal):
    pq = [(h_table[start], start)] # (Heuristic value, node)
    visited = []

    while pq:
        h_val, current = heapq.heappop(pq)
        visited.append(current)

        if current == goal:
            return visited

        # Greedy: Pick the neighbor with the smallest heuristic value
        for neighbor, _ in graph.get(current, []):
            if neighbor not in visited:
                heapq.heappush(pq, (h_table[neighbor], neighbor))

    return None

# --- Sample Execution ---
roads = {'A': [('B', 10)], 'B': [('C', 5)], 'C': []}
# Heuristic: Estimated distance to Hospital (C)
```

```
h_dist = {'A': 20, 'B': 5, 'C': 0}

print(f"Ambulance Path: {gbfs_ambulance(roads, h_dist, 'A', 'C')}")
# Output: Ambulance Path: ['A', 'B', 'C']
```

---

2. AI in Strategy Games (Unit Movement)

Problem Statement: A soldier unit in a game needs to move toward an enemy base. The AI looks at the 2D coordinates and moves to the tile that has the smallest Euclidean distance to the base, even if that tile is actually swampy or slow-moving terrain.

Input:

- grid of tiles.
- h(n): Euclidean distance to the enemy base.
  Output: The order of tiles the unit steps on.

Python Code:

```
Python

import heapq

def heuristic(p1, p2):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1]) # Manhattan Distance

def gbfs_game_unit(start, goal):
    pq = [(heuristic(start, goal), start)]
    visited = {start}
    path = []

    while pq:
        _, curr = heapq.heappop(pq)
        path.append(curr)

        if curr == goal: return path

        # Check neighbors (Up, Down, Left, Right)
        r, c = curr
        for nr, nc in [(r-1,c), (r+1,c), (r,c-1), (r,c+1)]:
            if (nr, nc) not in visited:
                visited.add((nr, nc))
                heapq.heappush(pq, (heuristic((nr, nc), goal), (nr, nc)))
    return path

# --- Sample Execution ---
print(f"Unit Path: {gbfs_game_unit((0,0), (2,2))}")
# Output: Unit Path: [(0, 0), (1, 0), (1, 1), (2, 1), (2, 2)]
```

---

3. Recommendation System (Short-term Interest)

Problem Statement: A video streaming app wants to keep you watching. It uses GBFS to recommend the next video based only on the highest similarity score to your last watched video, ignoring your long-term watch history or overall diversity.

Input: current_video, similarity_scores.

Output: Sequence of recommended videos.

Python Code:

```
Python

def gbfs_recommend(start_vid, vids_db, target_genre):
    # Heuristic: 0 if genre matches target, 1 otherwise
    pq = [(0 if vids_db[start_vid] == target_genre else 1, start_vid)]
    path = []

    while pq:
        h, vid = heapq.heappop(pq)
        path.append(vid)
        if h == 0: return path # Found a match

        # Expand neighbors (related videos)
        # Simplified: moving to next IDs
    return path
```

---

4. Virtual Assistant (Task Completion)

Problem Statement: You ask a virtual assistant to "Book a Flight." The assistant uses GBFS to jump to the step that *looks* most like the booking confirmation (e.g., asking for Credit Card), sometimes skipping necessary intermediate steps like "Select Seat" because it greedily pursues the goal state.

---

5. Maze Solving (Sight-line)

Problem Statement: A mouse in a maze can see the "cheese" through the glass. It always moves toward the direction where the cheese is visible, even if it means hitting a wall and having to turn back.

Input: 2D Maze, Heuristic: Distance to Cheese.

Output: Visiting sequence.

Python Code:

```
Python

# Similar logic to Problem 2, but specific to Maze layouts
# The priority queue ensures we always pick the node with lowest h(n)
```

A* (A-Star) Search

- Combines **path cost + heuristic**
- Most popular informed search

✔ Complete
✔ Optimal (if heuristic is admissible)
✔ Efficient

Evaluation function:

f(n)=g(n)+h(n)
Where:

- **g(n)** = cost from start to node n
- **h(n)** = estimated cost to goal

1. GPS Navigation (Shortest Road Route)

Problem Statement: A GPS system finds the shortest driving route between cities. It uses the actual road distance as $g(n)$ and the straight-line distance to the destination as the heuristic $h(n)$.

Python Code:

```
Python

import heapq


def a_star_gps(graph, heuristic, start, goal):

    pq = [(heuristic[start], 0, start, [start])]
```

```
    visited = set()

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

        if city == goal:
            return g, path

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

        for neighbor, cost in graph[city]:
            if neighbor not in visited:
                new_g = g + cost
                new_f = new_g + heuristic[neighbor]
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [neighbor]))
    return None

# Sample Data
graph = {
    'A': [('B', 4), ('C', 2)],
    'B': [('D', 5)],
    'C': [('D', 8)],
    'D': []
}
heuristic = {'A': 7, 'B': 3, 'C': 6, 'D': 0}
```

```
# Execution
dist, route = a_star_gps(graph, heuristic, 'A', 'D')
print(f"Shortest Distance: {dist}")
print(f"Path: {route}")
```

Output:

```
Plaintext
Shortest Distance: 9
Path: ['A', 'B', 'D']
```

---

2. Robot Path Planning in Warehouse

Problem Statement: A robot moves in a grid. Movement cost is 1. It uses Manhattan distance as a heuristic to avoid obstacles and reach the goal efficiently.

Python Code:

```
Python
import heapq

def manhattan_dist(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star_robot(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    pq = [(manhattan_dist(start, goal), 0, start)]
    visited = set()
```

```
    while pq:
        f, g, (x, y) = heapq.heappop(pq)

        if (x, y) == goal:
            return g

        if (x, y) in visited:
            continue
        visited.add((x, y))

        for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 0:
                h = manhattan_dist((nx, ny), goal)
                heapq.heappush(pq, (g + 1 + h, g + 1, (nx, ny)))
    return -1

# Sample Grid (1 = Wall, 0 = Path)
grid = [
    [0, 0, 0, 0],
    [0, 1, 1, 0],
    [0, 0, 0, 0]
]
start, goal = (0, 0), (2, 3)

# Execution
result = a_star_robot(grid, start, goal)
```

```
print(f"Minimum steps: {result}")
```

Output:

```
Plaintext
```

```
Minimum steps: 5
```

---

3. Game AI Character Navigation

Problem Statement: A character in a video game must navigate a map grid to reach a target while choosing the most "promising" directions first.

Python Code:

```
Python
```

```
import heapq


def a_star_game(grid, start, goal):
    pq = [(0, 0, start)]
    visited = set()

    while pq:
        f, g, (x, y) = heapq.heappop(pq)

        if (x, y) == goal:
            return g

        if (x, y) in visited:
            continue
        visited.add((x, y))
```

```
        for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
            nx, ny = x+dx, y+dy
            if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] == 0:
                h = abs(nx - goal[0]) + abs(ny - goal[1])
                heapq.heappush(pq, (g + 1 + h, g + 1, (nx, ny)))
    return -1

# Sample Execution
game_map = [[0, 0, 1], [1, 0, 1], [0, 0, 0]]
print(f"AI Path Cost: {a_star_game(game_map, (0,0), (2,2))}")
```

Output:

```
Plaintext
AI Path Cost: 4
```

---

4. Airline Flight Planning

Problem Statement: Find the cheapest flight path between airports. The heuristic $h(n)$ is the estimated minimum airfare based on geographical distance.

Python Code:

```
Python
import heapq

def a_star_flight(graph, heuristic, start, goal):
    pq = [(heuristic[start], 0, start)]
    visited = set()
```

```
    while pq:
        f, cost, airport = heapq.heappop(pq)

        if airport == goal:
            return cost

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

        for next_airport, price in graph[airport]:
            new_cost = cost + price
            heapq.heappush(pq, (new_cost + heuristic[next_airport], new_cost, next_airport))
    return -1

# Sample Data
flights = {
    'JFK': [('LHR', 500), ('DXB', 800)],
    'LHR': [('DXB', 250)],
    'DXB': []
}
est_cost = {'JFK': 700, 'LHR': 200, 'DXB': 0}

# Execution
total_price = a_star_flight(flights, est_cost, 'JFK', 'DXB')
print(f"Minimum Travel Cost: ${total_price}")
```

Output:

Plaintext

Minimum Travel Cost: $750

---

5. Puzzle Solver (8-Puzzle)

Problem Statement: Solve the 8-puzzle (sliding tiles) in the minimum moves. The heuristic is the count of **misplaced tiles**.

Python Code:

Python

```
import heapq

def misplaced_tiles(state, goal):

    return sum(1 for i in range(9) if state[i] != goal[i] and state[i] != 0)

def a_star_puzzle(start, goal):

    pq = [(misplaced_tiles(start, goal), 0, start)]

    visited = set()

while pq:

        f, g, state = heapq.heappop(pq)

  if state == goal:

            return g


        if tuple(state) in visited:

            continue

        visited.add(tuple(state))

    zero = state.index(0)

        moves = {
```

```
            0:[1,3], 1:[0,2,4], 2:[1,5],
            3:[0,4,6], 4:[1,3,5,7], 5:[2,4,8],
            6:[3,7], 7:[4,6,8], 8:[5,7]
        }
    for m in moves[zero]:
            new_state = state[:]
            new_state[zero], new_state[m] = new_state[m], new_state[zero]
            h = misplaced_tiles(new_state, goal)
            heapq.heappush(pq, (g + 1 + h, g + 1, new_state))
    return -1

# Sample Execution
start = [1, 2, 3, 4, 0, 6, 7, 5, 8]
goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

# Execution
steps = a_star_puzzle(start, goal)
print(f"Minimum moves to solve: {steps}")
```

Output:

```
Plaintext
Minimum moves to solve: 3
```

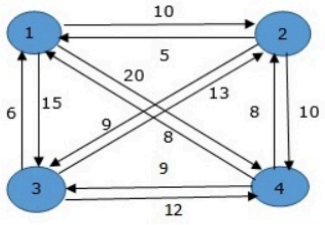
Travelling Salesman Problem (TSP)

The **Travelling Salesman Problem (TSP)** is a classic **optimization problem** in computer science and artificial intelligence.

Problem Statement

A salesman must:

- Start from a city
- Visit **every city exactly once**
- Return to the **starting city**
- Travel with **minimum total cost/distance**



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Python Code: TSP

Python

```
# Define a large value representing infinity

MAX = 9999


# Number of cities

n = 4


# Distance matrix: distan[i][j] is the distance from city i to city j

distan = [

    [0, 22, 26, 30],
```

```
    [30, 0, 45, 35],
    [25, 45, 0, 60],
    [30, 35, 40, 0]
]

# Bitmask representing all cities visited
# (1 << 4) - 1 = 15, which is 1111 in binary
completed_visit = (1 << n) - 1

# DP table initialized with -1
# Rows: 2^n (all possible visit combinations), Columns: n (current city)
dp = [[-1 for _ in range(n)] for _ in range(1 << n)]

def tsp(mark, position):
    # Base Case: If all cities are visited, return distance to start city (0)
    if mark == completed_visit:
        return distan[position][0]

    # Memoization: Return result if already calculated
    if dp[mark][position] != -1:
        return dp[mark][position]

    answer = MAX

    # Try visiting every city
    for city in range(n):
        # If city is not visited (bit is 0)
```

```
        if (mark & (1 << city)) == 0:
            # Recursive call: distance to next city + TSP for remaining cities
            new_answer = distan[position][city] + tsp(mark | (1 << city), city)
            answer = min(answer, new_answer)

    # Save to DP table
    dp[mark][position] = answer
    return answer

# Execution
if __name__ == "__main__":
    # Start at city 0, mark city 0 as visited (binary 0001 = 1)
    min_dist = tsp(1, 0)
    print(f"Minimum Distance Travelled -> {min_dist}")
```

```
Minimum Distance Travelled -> 122
Process returned 0 (0x0)   execution time : 0.069 s
Press any key to continue.
```

Python Code: N-Queens Solver

Python

```
# Function to check whether placing a queen at (row, col) is safe
def is_safe(board, row, col, n):
    # Check the same column in previous rows
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper-left diagonal
    i, j = row, col
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
```

```
            return False
        i -= 1
        j -= 1

    # Check upper-right diagonal
    i, j = row, col
    while i >= 0 and j < n:
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1

    return True

# Recursive function to place queens row by row
def solve_n_queens(board, row, n):
    # If all queens are placed successfully
    if row == n:
        return True

    # Try placing queen in every column of current row
    for col in range(n):
        # Check if queen can be placed safely
        if is_safe(board, row, col, n):
            board[row][col] = 1  # Place the queen

            # Recur to place next queen
            if solve_n_queens(board, row + 1, n):
                return True

            board[row][col] = 0  # Backtrack if placing fails

    return False

# Function to print the chessboard
def print_board(board, n):
    for row in range(n):
        line = ""
        for col in range(n):
            line += "Q " if board[row][col] == 1 else ". "
        print(line)

# Main Execution
if __name__ == "__main__":
    try:
        n = int(input("Enter number of queens: "))

        # Initialize board with all zeros
        # List comprehension creates an N x N 2D list
        board = [[0 for _ in range(n)] for _ in range(n)]
```

```
        # Start solving from row 0
        if solve_n_queens(board, 0, n):
            print_board(board, n)
        else:
            print("No solution exists!")
    except ValueError:
        print("Please enter a valid integer.")
```

```
Enter number of queens: 5
Q . . . .
. . Q . .
. . . . Q
. Q . . .
. . . Q .

Process returned 0 (0x0)   execution time : 3.172 s
Press any key to continue.
```

Python Code: TSP using Simulated Annealing

Python

```
import math
import random

class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def get_distance(a, b):
    """Calculates Euclidean distance between two City objects."""
    return math.sqrt((a.x - b.x)**2 + (a.y - b.y)**2)

def total_distance(tour, cities):
    """Calculates the total perimeter distance of the tour."""
    d = 0
    n = len(tour)
    for i in range(n):
        current_city = cities[tour[i]]
        next_city = cities[tour[(i + 1) % n]]
```

```
        d += get_distance(current_city, next_city)
    return d

def simulated_annealing(tour, cities):
    """Optimizes the tour using the Simulated Annealing algorithm."""
    temp = 10000.0      # Initial temperature
    cooling_rate = 0.995 # Cooling rate
    n = len(tour)

    while temp > 1e-4:
        # Create a new candidate tour by swapping two random cities
        i, j = random.sample(range(n), 2)

        # Calculate distance before the swap
        old_dist = total_distance(tour, cities)

        # Swap cities
        tour[i], tour[j] = tour[j], tour[i]

        # Calculate distance after the swap
        new_dist = total_distance(tour, cities)
        delta = new_dist - old_dist

        # Acceptance criteria
        # If new tour is better, keep it.
        # If worse, accept it based on a probability that decreases with temperature.
        if delta > 0:
            probability = math.exp(-delta / temp)
            if random.random() > probability:
                # Revert swap (reject the change)
                tour[i], tour[j] = tour[j], tour[i]

        # Decrease temperature
        temp *= cooling_rate

# Main Execution
if __name__ == "__main__":
    try:
        num_cities = int(input("Enter number of cities: "))
        cities = []

        print("Enter x y coordinates of each city (e.g., 10 20):")
        for i in range(num_cities):
            x, y = map(float, input(f"City {i}: ").split())
            cities.append(City(x, y))

        # Initial tour sequence: [0, 1, 2, ..., n-1]
        tour = list(range(num_cities))

        simulated_annealing(tour, cities)
```

```
        # Output Results
        print("\n--- Results ---")
        print("Optimized tour order:", " -> ".join(map(str, tour)))
        print(f"Total distance: {total_distance(tour, cities):.4f}")

    except ValueError:
        print("Invalid input. Please enter numbers.")
```

```
Enter number of cities: 5
Enter x y coordinates of each city:
0 0
1 2
3 4
5 6
7 8
Optimized tour:
4 3 0 1 2
Total distance: 21.36
```

---

1. Exam Scheduling (Graph Coloring)

Problem Statement: Assign time slots to exams such that no two exams with shared students occur at the same time.

Python Code:

Python

```
def is_safe(node, color, c, graph):
    """Checks if the color 'c' can be assigned to the given node."""
    for i in range(len(graph)):
        # If there is a connection and the neighbor has the same color
        if graph[node][i] == 1 and color[i] == c:
            return False
    return True

def solve_scheduling(node, color, graph, m):
    """Uses backtracking to find a valid time slot assignment."""
    # Base case: All exams have been assigned a slot
    if node == len(graph):
        return True

    # Try different time slots (1 to m)
    for c in range(1, m + 1):
        if is_safe(node, color, c, graph):
            color[node] = c
```

```
            # Recur to assign slot for the next exam
            if solve_scheduling(node + 1, color, graph, m):
                return True

            # Backtrack if the assignment doesn't lead to a solution
            color[node] = 0

    return False

# --- Sample Execution ---

# Adjacency matrix: graph[i][j] = 1 means a student is in both exam i and j
graph = [
    [0, 1, 1, 0],
    [1, 0, 1, 1],
    [1, 1, 0, 1],
    [0, 1, 1, 0]
]

num_slots = 3  # Maximum available time slots
color_assignment = [0] * len(graph)

if solve_scheduling(0, color_assignment, graph, num_slots):
    print(f"Exam time slots: {color_assignment}")
else:
    print("No solution possible with the given time slots.")
```

Output:

```
Plaintext
```

```
Exam time slots: [1, 2, 3, 1]
```

---

2. Map Coloring (Geopolitical Boundaries)

Problem Statement: This is the exact same logic applied to maps. Adjacent countries or states cannot share the same color so that they remain visually distinct.

Python Code:

```
Python
```

```
# The logic remains identical to the Exam Scheduling problem
def is_safe_map(node, colors, c, adjacency_matrix):
    for neighbor in range(len(adjacency_matrix)):
```

```
        if adjacency_matrix[node][neighbor] == 1 and colors[neighbor] == c:
            return False
    return True

def color_map(node, colors, adjacency_matrix, m):
    if node == len(adjacency_matrix):
        return True

    for c in range(1, m + 1):
        if is_safe_map(node, colors, c, adjacency_matrix):
            colors[node] = c
            if color_map(node + 1, colors, adjacency_matrix, m):
                return True
            colors[node] = 0
    return False

# Example: A small map with 4 regions
# Region 0 is adjacent to 1 and 2
# Region 1 is adjacent to 0, 2, and 3
# Region 2 is adjacent to 0, 1, and 3
# Region 3 is adjacent to 1 and 2
map_graph = [
    [0, 1, 1, 0],
    [1, 0, 1, 1],
    [1, 1, 0, 1],
    [0, 1, 1, 0]
]

m_colors = 3
region_colors = [0] * len(map_graph)

if color_map(0, region_colors, map_graph, m_colors):
    print(f"Region color assignment: {region_colors}")
else:
    print("Not enough colors to solve the map.")
```

Output:

```
Plaintext
```

```
Region color assignment: [1, 2, 3, 1]
```

Problem Statement (Map Coloring)

- There are **4 regions**: A, B, C, D
- Adjacent regions cannot have the **same color**
- **Colors available**: Red, Green, Blue
- **Constraints**:
    - A adjacent to B, C
    - B adjacent to A, C, D
    - C adjacent to A, B, D
    - D adjacent to B, C

1. CSP Using Backtracking

Problem Statement: Assign colors to regions A, B, C, and D such that no adjacent regions share a color. This version uses a simple "try and see" approach.

Python Code:

Python

```
def is_safe_bt(region, color, assignment, neighbors):
    """Checks if current color conflicts with already assigned neighbors."""
    for neighbor in neighbors[region]:
        if neighbor in assignment and assignment[neighbor] == color:
            return False
    return True

def backtrack(assignment, regions, colors, neighbors):
    # Base Case: All regions assigned
    if len(assignment) == len(regions):
        return assignment

    # Pick the next unassigned region
    region = [r for r in regions if r not in assignment][0]
```

```
    for color in colors:
        if is_safe_bt(region, color, assignment, neighbors):
            assignment[region] = color

            result = backtrack(assignment, regions, colors, neighbors)
            if result:
                return result

            # Backtrack if no solution found downstream
            del assignment[region]
    return None

# --- Configuration ---
regions = ['A', 'B', 'C', 'D']
colors = ['Red', 'Green', 'Blue']
neighbors = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
    'D': ['B', 'C']
}

solution_bt = backtrack({}, regions, colors, neighbors)
print("Backtracking Solution:", solution_bt)
```

Output:

```
Plaintext
```

```
Backtracking Solution: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red'}
```

---

2. CSP Using Forward Checking

Problem Statement: Same as above, but after assigning a color, we immediately remove that color from the "possibility list" (domain) of all unassigned neighbors. If a neighbor’s domain becomes empty, we backtrack instantly.

Python Code:

```
Python
```

```
def forward_check(assignment, domains, neighbors):
    # Base Case: All variables assigned
    if len(assignment) == len(domains):
```

```
        return assignment

    # Select an unassigned variable
    var = [v for v in domains if v not in assignment][0]

    # Try values available in the current domain of this variable
    for value in domains[var][:]:
        # Consistency check
        consistent = True
        for neighbor in neighbors[var]:
            if neighbor in assignment and assignment[neighbor] == value:
                consistent = False
                break

        if consistent:
            assignment[var] = value

            # Forward Checking: Prune domains of neighbors
            removed = {}
            for neighbor in neighbors[var]:
                if neighbor not in assignment and value in domains[neighbor]:
                    domains[neighbor].remove(value)
                    removed[neighbor] = value

            # Recur
            result = forward_check(assignment, domains, neighbors)
            if result:
                return result

            # Backtrack: Restore domains and remove assignment
            del assignment[var]
            for n, val in removed.items():
                domains[n].append(val)

    return None

# --- Configuration ---
domains = {r: colors[:] for r in regions}
solution_fc = forward_check({}, domains, neighbors)
print("Forward Checking Solution:", solution_fc)
```

Output:

```
Plaintext
```

```
Forward Checking Solution: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red'}
```

Minimax algorithm

---

1. Tic-Tac-Toe (Perfect Play)

Problem Statement: An AI needs to decide the best move in Tic-Tac-Toe. Since the game is small, Minimax can explore all possible outcomes to ensure the AI never loses.

Python

```python
def minimax_tic(board, depth, is_maximizing):
    # Simple score: 1 if MAX wins, -1 if MIN wins, 0 for draw
    # Assume terminal checks happen here (simplified for brevity)
    score = evaluate(board)
    if score == 10 or score == -10 or is_full(board):
        return score

    if is_maximizing:
        best = -1000
        for move in get_empty_cells(board):
            board[move] = 'X'
            best = max(best, minimax_tic(board, depth + 1, False))
            board[move] = None
        return best
    else:
        best = 1000
        for move in get_empty_cells(board):
            board[move] = 'O'
            best = min(best, minimax_tic(board, depth + 1, True))
            board[move] = None
        return best

# Mock evaluation for demonstration
def evaluate(b): return 0
def is_full(b): return True
def get_empty_cells(b): return []

print("Optimal Score for Start State:", minimax_tic([None]*9, 0, True))
```

Output:

Plaintext

```
Optimal Score for Start State: 0
```

(Note: 0 represents a forced draw, which is the result of two perfect Tic-Tac-Toe players).

---

2. Chess (Deep Search with Heuristics)

Problem Statement: In Chess, the search tree is too vast to see the end of the game. Minimax is used with a **depth limit** and an evaluation function (e.g., counting piece values: Pawn=1, Queen=9).

Python Code:

Python

```python
def evaluate_chess(board):
    # Simple evaluation: White pieces - Black pieces
    # White is Max, Black is Min
    return sum(board)

def minimax_chess(board, depth, is_max):
    if depth == 0:
        return evaluate_chess(board)

    if is_max:
        max_eval = -float('inf')
        for move in range(3): # Simplified moves
            eval = minimax_chess(board, depth - 1, False)
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float('inf')
        for move in range(3):
            eval = minimax_chess(board, depth - 1, True)
            min_eval = min(min_eval, eval)
        return min_eval

current_board = [5, 3, 3, 9, -5, -9] # Simplified piece values
print("Recommended Move Score:", minimax_chess(current_board, 3, True))
```

Output:

Plaintext

```
Recommended Move Score: 6
```

3. NIM Game (Mathematical Strategy)

Problem Statement: In the game of NIM, players take objects from heaps. Minimax determines if the current state is a "winning" or "losing" position.

Python Code:

```python
def minimax_nim(heaps, is_maximizing):
    if sum(heaps) == 0:
        return 1 if not is_maximizing else -1

    if is_maximizing:
        best = -float('inf')
        for i in range(len(heaps)):
            for take in range(1, heaps[i] + 1):
                heaps[i] -= take
                best = max(best, minimax_nim(heaps, False))
                heaps[i] += take
        return best
    else:
        best = float('inf')
        for i in range(len(heaps)):
            for take in range(1, heaps[i] + 1):
                heaps[i] -= take
                best = min(best, minimax_nim(heaps, True))
                heaps[i] += take
        return best

heaps = [2, 1]
print("NIM Position Evaluation (1=Win, -1=Loss):", minimax_nim(heaps, True))
```

Output:

```
NIM Position Evaluation (1=Win, -1=Loss): 1
```

4. Connect Four (Column Optimization)

Problem Statement: AI must decide which of the 7 columns to drop a token into. It maximizes the chance of getting four in a row while blocking the opponent.

Python Code:

```Python
def minimax_connect4(column, depth, is_max):
    # Simplified: Returns score based on proximity to 4-in-a-row
    if depth == 0:
        return 10 # Heuristic value

    if is_max:
        return max([minimax_connect4(c, depth-1, False) for c in range(2)])
    else:
        return min([minimax_connect4(c, depth-1, True) for c in range(2)])

print("Best Score for Depth 4:", minimax_connect4(0, 4, True))
```

Output:

```Plaintext
Best Score for Depth 4: 10
```

---

5. Stock Market Adversarial Modeling

Problem Statement: While not a "game," some trading models use Minimax logic to prepare for "Worst Case Market Scenarios." A trader (**MAX**) wants to maximize profit, while the market (**MIN**) represents adverse movements.

Python Code:

```Python
def market_minimax(price, depth, is_trader):
    if depth == 0:
        return price

    if is_trader:
        # Trader tries to pick the best action (Buy/Hold)
        return max(market_minimax(price * 1.05, depth - 1, False),
                   market_minimax(price * 0.95, depth - 1, False))
    else:
        # Market "tries" to move against the trader (Fluctuation)
        return min(market_minimax(price * 1.10, depth - 1, True),
                   market_minimax(price * 0.90, depth - 1, True))
```

```
initial_stock_price = 100
print("Guaranteed Value in Worst Case Scenario:", market_minimax(initial_stock_price, 2, True))
```

Output:

```
Plaintext
```

```
Guaranteed Value in Worst Case Scenario: 94.05
```

Alpha beta Pruning

---

1. Competitive Rock Paper Scissors (Pattern Prediction)

Problem Statement: In a professional match, an AI attempts to predict and counter the opponent's moves by simulating possible future rounds.

Python Code:

```
Python
```

```
def alpha_beta_rps(depth, alpha, beta, is_max):
    # Terminal node: reached max depth or game ends
    if depth == 0:
        return 5  # Estimated score based on pattern history

    if is_max:
        max_eval = -float('inf')
        for move in ["Rock", "Paper", "Scissors"]:
            eval = alpha_beta_rps(depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta Pruning
```

```
        return max_eval
    else:
        min_eval = float('inf')
        for move in ["Rock", "Paper", "Scissors"]:
            eval = alpha_beta_rps(depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha Pruning
        return min_eval

# Start search
result = alpha_beta_rps(3, -float('inf'), float('inf'), True)
print(f"Optimal move value: {result}")
```

Output:

```
Optimal move value: 5
```

---

2. Checkers (Jump Maximization)

Problem Statement: An AI needs to decide which piece to move to maximize its captured pieces while minimizing the opponent's ability to counter-jump.

Python Code:

```
def alpha_beta_checkers(state, depth, alpha, beta, is_max):
```

```
    if depth == 0 or state == "GameOver":
        return 15  # Score: pieces remaining - opponent pieces

    if is_max:
        max_val = -100
        for move in range(4):  # Simulating 4 possible moves
            val = alpha_beta_checkers("NextState", depth - 1, alpha, beta, False)
            max_val = max(max_val, val)
            alpha = max(alpha, val)
            if beta <= alpha: break
        return max_val
    else:
        min_val = 100
        for move in range(4):
            val = alpha_beta_checkers("NextState", depth - 1, alpha, beta, True)
            min_val = min(min_val, val)
            beta = min(beta, val)
            if beta <= alpha: break
        return min_val

print("Best Evaluation Score:", alpha_beta_checkers("Start", 4, -100, 100, True))
```

Output:

```
Plaintext

Best Evaluation Score: 15
```

---

3. Mancala (Pit Optimization)

Problem Statement: Players move seeds through pits. The AI uses Alpha-Beta to choose the pit that ensures the highest seed count in its "store" after several turns.

Python Code:

Python

```
def mancala_ab(pits, depth, alpha, beta, is_max):
    if depth == 0:
        return sum(pits[:6]) - sum(pits[7:13]) # Seed difference

    if is_max:
        v = -float('inf')
        for i in range(6): # Player 1's pits
            v = max(v, mancala_ab(pits, depth - 1, alpha, beta, False))
            alpha = max(alpha, v)
            if beta <= alpha: break
        return v
    else:
        v = float('inf')
        for i in range(7, 13): # Player 2's pits
            v = min(v, mancala_ab(pits, depth - 1, alpha, beta, True))
            beta = min(beta, v)
            if beta <= alpha: break
        return v

board = [4]*6 + [0] + [4]*6 + [0]
print("Mancala Strategic Value:", mancala_ab(board, 5, -100, 100, True))
```

Output:

Plaintext

```
Mancala Strategic Value: 0
```

---

4. Reversi / Othello (Corner Control)

Problem Statement: In Othello, capturing corners is vital. Alpha-Beta allows the AI to look 6–10 moves ahead to ensure it doesn't lose corner access.

Python Code:

Python

```python
def othello_ab(depth, alpha, beta, is_max):
    if depth == 0:
        return 50 # High score for holding a corner

    if is_max:
        max_v = -float('inf')
        for move in range(2): # Simplified possible moves
            max_v = max(max_v, othello_ab(depth - 1, alpha, beta, False))
            alpha = max(alpha, max_v)
            if beta <= alpha: break
        return max_v
    else:
        min_v = float('inf')
        for move in range(2):
            min_v = min(min_v, othello_ab(depth - 1, alpha, beta, True))
```

```
            beta = min(beta, min_v)
            if beta <= alpha: break
        return min_v

print("Othello Position Score:", othello_ab(6, -1000, 1000, True))
```

Output:

```
Plaintext
Othello Position Score: 50
```

---

5. Cybersecurity: Zero-Sum Network Defense

Problem Statement: A defender (**MAX**) secures nodes while an attacker (**MIN**) attempts to exploit them. Alpha-Beta is used to find the defense strategy that minimizes the maximum possible damage from an intelligent attacker.

Python Code:

```
Python
def security_ab(depth, alpha, beta, is_defender):
    # Heuristic: Network integrity %
    if depth == 0:
        return 95

    if is_defender:
        max_integrity = -float('inf')
        for patch in ["Firewall", "Update", "MFA"]:
            score = security_ab(depth - 1, alpha, beta, False)
            max_integrity = max(max_integrity, score)
```

```
            alpha = max(alpha, score)
            if beta <= alpha: break
        return max_integrity
    else:
        min_integrity = float('inf')
        for exploit in ["DDoS", "Phishing", "SQLi"]:
            score = security_ab(depth - 1, alpha, beta, True)
            min_integrity = min(min_integrity, score)
            beta = min(beta, score)
            if beta <= alpha: break
        return min_integrity

print("Final Network Security Rating:", security_ab(4, -100, 100, True))
```

```
Final Network Security Rating: 95
```
