**Research-Based Assignment on Uninformed and Informed Search Strategies**

**Part 1: Understanding Search Strategies (Conceptual Questions)**

**Task 1: Definitions and Characteristics**

1. Define **Uninformed Search** and provide two examples.
2. Define **Informed Search** and provide two.
3. Compare **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** in terms of:
   * Completeness
   * Optimality
   * Time Complexity
   * Space Complexity
4. Why is *A Search*\* considered better than **Greedy Best-First Search (GBFS)**?

Search Strategies

**1. Uninformed Search**

Uninformed search, also known as blind search, is a search strategy that does not use any additional information about the problem other than the definition of the problem. It relies solely on the algorithm's mechanics to find a solution.

Examples:

- Breadth-First Search (BFS)

- Depth-First Search (DFS)

**2. Informed Search**

Informed search, also known as heuristic search, is a search strategy that uses additional information about the problem, such as heuristics or evaluation functions, to guide the search towards more promising areas of the search space.

Examples:

- Greedy Best-First Search (GBFS)

- A\* Search

1. **Comparison of BFS and DFS**

### ****1. Completeness****

**BFS:** ✅ **Complete** – It will always find a solution if one exists, because it explores all nodes at the current depth before moving deeper.

**DFS:** ❌ **Not complete** (in general) – It may go down an infinite path and never find a solution if the tree/graph is infinite.

### ****2. Optimality****

**BFS:** ✅ **Optimal** (only if all step costs are equal) – It finds the shallowest (i.e., least cost in equal-weighted graphs) goal node first.

**DFS:** ❌ **Not optimal** – It may find a longer or more costly solution before a shorter one.

### ****3. Time Complexity****

Assuming:

**b** = branching factor (maximum number of successors of any node)

**d** = depth of the shallowest goal node

**m** = maximum depth of the search tree

**BFS:** O(b^d) – Because it explores all nodes level by level until depth d.

**DFS:** O(b^m) – Because it may explore the deepest paths, even without reaching the goal early.

### ****4. Space Complexity****

**BFS:** O(b^d) – Stores all nodes at the current level; high memory usage.

**DFS:** O(m) – Only needs to store nodes along the current path from the root to the leaf.

**4. A Search vs. Greedy Best-First Search (GBFS)\***

- Optimality: A\* Search is guaranteed to find the optimal solution if the heuristic function is admissible (never overestimates the true cost) and consistent (the estimated cost to reach the goal is always less than or equal to the true cost). GBFS, on the other hand, may find a suboptimal solution.

- Informedness: A\* Search takes into account both the estimated cost to reach the goal (heuristic function) and the cost of reaching the current node (cost function). GBFS only considers the estimated cost to reach the goal.

- Robustness: A\* Search is more robust and can handle a wider range of problems, including those with non-uniform edge costs. GBFS can get stuck in local optima if the heuristic function is not well-designed.

Overall, A\* Search is a more informed and robust search strategy that can provide better solutions than GBFS in many cases.

**Task 2: Problem-Solving Scenarios**

For each scenario below, suggest the most appropriate search strategy (from the allowed list) and justify your choice:

1. Finding the shortest path in a grid.
2. Solving an 8-puzzle problem with a heuristic (Research based).
3. Exploring all possible moves in a game tree without heuristic knowledge (Research based)

### ****1. Finding the shortest path in a grid****

**✅ Suggested Strategy:** **Breadth-First Search (BFS)**  
**✅ Justification:**

In an unweighted grid (where each move has the same cost), **BFS** is guaranteed to find the shortest path to the goal.

BFS explores all paths level by level, ensuring the first time the goal is found is through the shortest possible path (in terms of number of moves).

**Complete** and **Optimal** under equal step costs.

### ****2. Solving an 8-puzzle problem with a heuristic****

**✅ Suggested Strategy:** **A\* Search**  
**✅ Justification:**

The 8-puzzle is a classic heuristic-based problem with a manageable state space.

A\* uses both the cost so far (**g(n)**) and the estimated cost to goal (**h(n)**), making it efficient.

With a good heuristic like **Manhattan Distance** or **Misplaced Tiles**, A\* finds optimal solutions much faster than blind searches.

**Complete**, **Optimal** (with admissible heuristics), and significantly **more efficient** than uninformed search methods.

### ****3. Exploring all possible moves in a game tree without heuristic knowledge****

**✅ Suggested Strategy:** **Depth-First Search (DFS)** or **Minimax (without evaluation function)**  
**✅ Justification:**

If no heuristic is available, and the goal is to **explore all possible moves**, then DFS is more memory-efficient for deep game trees.

In turn-based games (e.g., chess, tic-tac-toe), **Minimax** is used for decision-making. Without a heuristic (evaluation function), it must go to terminal states, which makes **DFS** (used inside Minimax) suitable.

While not optimal or efficient in all cases, DFS is feasible for full exploration when memory is a concern.

**Part 2: Implementation-Based Tasks**

**Task 3: Implementing BFS and DFS**

1. Write a Python program to implement **BFS** for finding the shortest path in a graph.
2. Modify the same program to implement **DFS** and compare the paths obtained.
3. Analyze which algorithm is more efficient for this problem and why (Research based).

### ****1. BFS Implementation (Shortest Path)****

python

CopyEdit

from collections import deque

def bfs\_shortest\_path(graph, start, goal):

visited = set()

queue = deque([[start]]) # Queue stores paths

while queue:

path = queue.popleft()

node = path[-1]

if node == goal:

return path

if node not in visited:

visited.add(node)

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

new\_path = list(path)

new\_path.append(neighbor)

queue.append(new\_path)

return None

### ✅ ****2. DFS Implementation (Path - Not necessarily shortest)****

python

CopyEdit

def dfs\_path(graph, start, goal):

visited = set()

stack = [[start]] # Stack stores paths

while stack:

path = stack.pop()

node = path[-1]

if node == goal:

return path

if node not in visited:

visited.add(node)

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

new\_path = list(path)

new\_path.append(neighbor)

stack.append(new\_path)

return None

### ✅ ****Sample Graph and Test****

python

CopyEdit

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = 'A'

goal\_node = 'F'

bfs\_path = bfs\_shortest\_path(graph, start\_node, goal\_node)

dfs\_path\_result = dfs\_path(graph, start\_node, goal\_node)

print("BFS Path (Shortest):", bfs\_path)print("DFS Path:", dfs\_path\_result)

### ✅ ****3. Analysis: Which is More Efficient?****

| **Criteria** | **BFS** | **DFS** |
| --- | --- | --- |
| **Goal** | Finds **shortest path** | May find **any path** |
| **Completeness** | ✅ Complete | ❌ May not find goal in infinite tree |
| **Optimality** | ✅ Optimal (for unweighted graphs) | ❌ Not optimal |
| **Space** | Uses **more memory** | More **memory-efficient** |
| **Time** | May take longer in **wide graphs** | May be faster but **unreliable** |

**Task 4: Implementing Greedy Best-First Search (GBFS) and A**\*

1. Implement **Greedy Best-First Search** using a simple heuristic (e.g., Manhattan distance for a grid).
2. Extend the program to implement *A Search*\* with the same heuristic.
3. Compare the number of nodes explored by GBFS and A\* in a given maze.

### ****Assumptions****

Grid is a 2D list where:

0 = free space

1 = wall

Movement allowed in 4 directions: up, down, left, right

### ✅ ****1. Greedy Best-First Search (GBFS)****

python

CopyEdit

import heapq

def manhattan(a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1])

def greedy\_best\_first\_search(grid, start, goal):

rows, cols = len(grid), len(grid[0])

visited = set()

heap = [(manhattan(start, goal), start)]

came\_from = {start: None}

nodes\_explored = 0

while heap:

\_, current = heapq.heappop(heap)

nodes\_explored += 1

if current == goal:

break

if current in visited:

continue

visited.add(current)

for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:

neighbor = (current[0]+dx, current[1]+dy)

if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:

if grid[neighbor[0]][neighbor[1]] == 0 and neighbor not in visited:

heapq.heappush(heap, (manhattan(neighbor, goal), neighbor))

if neighbor not in came\_from:

came\_from[neighbor] = current

# Reconstruct path

path = []

curr = goal

while curr and curr in came\_from:

path.append(curr)

curr = came\_from[curr]

path.reverse()

return path, nodes\_explored

### ✅ ****2. A\* Search Implementation****

python

CopyEdit

def a\_star\_search(grid, start, goal):

rows, cols = len(grid), len(grid[0])

visited = set()

heap = [(manhattan(start, goal), 0, start)]

came\_from = {start: None}

g\_cost = {start: 0}

nodes\_explored = 0

while heap:

\_, cost\_so\_far, current = heapq.heappop(heap)

nodes\_explored += 1

if current == goal:

break

if current in visited:

continue

visited.add(current)

for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:

neighbor = (current[0]+dx, current[1]+dy)

if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:

if grid[neighbor[0]][neighbor[1]] == 0:

new\_g = g\_cost[current] + 1

if neighbor not in g\_cost or new\_g < g\_cost[neighbor]:

g\_cost[neighbor] = new\_g

f\_cost = new\_g + manhattan(neighbor, goal)

heapq.heappush(heap, (f\_cost, new\_g, neighbor))

came\_from[neighbor] = current

# Reconstruct path

path = []

curr = goal

while curr and curr in came\_from:

path.append(curr)

curr = came\_from[curr]

path.reverse()

return path, nodes\_explored

### ✅ ****3. Test and Comparison****

python

CopyEdit

grid = [

[0, 0, 0, 0, 1],

[1, 1, 0, 1, 0],

[0, 0, 0, 0, 0],

[0, 1, 1, 1, 1],

[0, 0, 0, 0, 0]

]

start = (0, 0)

goal = (4, 4)

gbfs\_path, gbfs\_nodes = greedy\_best\_first\_search(grid, start, goal)

a\_star\_path, a\_star\_nodes = a\_star\_search(grid, start, goal)

print("Greedy BFS Path:", gbfs\_path)print("Greedy BFS Nodes Explored:", gbfs\_nodes)

print("A\* Path:", a\_star\_path)print("A\* Nodes Explored:", a\_star\_nodes)

**Part 3: Research and Analysis**

**Task 5: Case Study on Real-World Applications (Research based)**

Research and write a short report (200-300 words) on:

* **One real-world application of BFS/DFS** (e.g., web crawling, social networks).
* **One real-world application of A**\* (e.g., robotics, GPS navigation).
* Discuss why an informed search is preferred over an uninformed search in these cases.

#### ****1. BFS in Web Crawling****

Web crawling is a fundamental component of search engines like Google. It involves systematically browsing the World Wide Web to index web pages. **Breadth-First Search (BFS)** is commonly used in this process. Starting from a seed URL, the crawler visits all directly linked pages (level 1), then all pages linked from those pages (level 2), and so on. This ensures a wide coverage of the internet without getting trapped in deep but less important sections. BFS is effective here because it helps in prioritizing higher-level or more “popular” pages early, which are typically more relevant.

#### ****2. A\* in GPS Navigation****

**A\*** is widely used in GPS navigation systems like Google Maps or vehicle routing applications. The algorithm finds the shortest or fastest path between two points on a road network. It combines the actual distance traveled (g-cost) with an estimate of the remaining distance (h-cost, often using straight-line or Euclidean distance). This makes A\* highly efficient and optimal for real-time route planning, where both accuracy and speed are critical, especially in dynamic environments like city traffic.

#### ****3. Why Informed Search is Preferred****

In real-world scenarios like GPS navigation or robotics, **informed search algorithms like A\*** are preferred over uninformed methods like BFS or DFS because they incorporate domain-specific knowledge (heuristics). This significantly reduces the number of nodes explored, increases efficiency, and ensures optimal solutions in complex or time-sensitive environments. Uninformed methods lack this guidance and may explore irrelevant paths, wasting resources and time. Hence, informed search offers a practical balance between performance and optimality in real-world applications.

**Task 6: Limitations and Trade-offs**

1. What are the main limitations of **Greedy Best-First Search**?
2. Under what conditions does **A**\* fail to find an optimal solution?
3. Why might **DFS** be impractical for large search spaces despite its low memory usage?

### ****1. What are the main limitations of Greedy Best-First Search (GBFS)?****

**Not Optimal**: GBFS only considers the heuristic value (h(n)) and ignores the actual path cost (g(n)), so it may choose a seemingly promising path that turns out to be suboptimal.

**Heuristic-Dependent**: Its success heavily relies on the accuracy of the heuristic. A poor heuristic can mislead the search, resulting in longer runtimes or failure.

**Incomplete**: In graphs with cycles or infinite paths, GBFS may loop indefinitely without finding a solution unless extra mechanisms (like visited sets) are used.

**Can Get Stuck in Local Minima**: GBFS may get trapped exploring one direction extensively if the heuristic consistently favors it, missing better options elsewhere.

### ****2. Under what conditions does A\* fail to find an optimal solution?****

**Heuristic is Not Admissible**: If the heuristic overestimates the true cost to the goal (h(n) > actual cost), A\* may prioritize wrong paths and return a non-optimal solution.

**Heuristic is Inconsistent**: If the heuristic doesn't satisfy the consistency condition (i.e., h(n) ≤ cost(n, n') + h(n')), it may revisit nodes unnecessarily, affecting performance or correctness.

**Resource Limitations**: A\* can consume large amounts of memory and time in very large or complex state spaces, potentially causing it to fail due to system constraints.

### ****3. Why might DFS be impractical for large search spaces despite its low memory usage?****

**No Optimality**: DFS doesn't guarantee the shortest or least-cost path; it might find a longer or inefficient one first.

**Incomplete in Infinite Spaces**: DFS can get trapped in infinitely deep branches, never finding the goal even if a solution exists

**Repetitive Exploration**: Without visited-node tracking, it may revisit the same states multiple times, wasting time.

**Path Blindness**: DFS goes deep without knowing if it’s moving closer to the goal, which is inefficient in wide or deep spaces with many dead ends.

**Submission Guidelines**

* Include **code implementations** (Python files) for Tasks 3 and 4.
* Push the assignment file (along with Python files) to your GitHub repositories.