# Title: Implementation of A* Search and Best First Search Algorithm and Differentiation from BFS and DFS

## 1. Objectives

The objectives of this lab are:

- To understand and implement A* Search and Best First Search (GBFS) algorithms.
- To analyze and compare these algorithms with Breadth-First Search (BFS) and Depth-First Search (DFS).
- To evaluate their efficiency in terms of time complexity, optimality, and space utilization.

## 2. Introduction

Search algorithms are fundamental in Artificial Intelligence for pathfinding and decision-making.

- **A* Search:** Uses both cost (g(n)) and heuristic (h(n)) to find the optimal path efficiently.
- **Best First Search (BFS):** A greedy algorithm that selects nodes based on the heuristic function alone, often leading to faster but non-optimal solutions.
- **Breadth-First Search (BFS)**: Explores all paths level by level, guaranteeing the shortest path in unweighted graphs.
- **Depth-First Search (DFS)**: Explores deeply before backtracking, which is memory-efficient but may not find the best solution.

**Key Differences:**

- A* balances cost (g(n)) and heuristic (h(n)) to guarantee the shortest path.
- Best First Search uses only h(n), making it faster but sometimes misleading.
- BFS is complete and optimal for unweighted graphs but requires more memory.
- DFS is useful for deep searches but not optimal.

## 3. Lab Work Implementation

### A* Search Algorithm

**Logic:**

- Uses a priority queue (heapq) to always expand the most promising node based on f(n) = g(n) + h(n).
- Ensures the shortest path by balancing cost and heuristic values.

**Python Code for A Search***

```
import heapq


def a_star(graph, start, goal, heuristic):
    priority_queue = [(0, start, [start])]
    g_cost = {start: 0}

    while priority_queue:
        cost, node, path = heapq.heappop(priority_queue)

        if node == goal:
            return path
        for neighbor, weight in graph.get(node, {}).items():
            new_cost = g_cost[node] + weight
            if neighbor not in g_cost or new_cost < g_cost[neighbor]:
                g_cost[neighbor] = new_cost
                f_cost = new_cost + heuristic[neighbor]
                heapq.heappush(priority_queue, (f_cost, neighbor, path + [neighbor]))

    return None
```

## Best First Search Algorithm

Logic:

- Uses priority queue but selects nodes only based on h(n).
- Faster than A* but does not guarantee the shortest path.

**Python Code for Best First Search**

```
from heapq import heappop, heappush


def best_first_search(graph, start, goal, heuristic):
    priority_queue = [(heuristic[start], start, [start])]
    visited = set()

    while priority_queue:
        _, node, path = heappop(priority_queue)
        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in graph.get(node, {}):
                heappush(priority_queue, (heuristic[neighbor], neighbor, path + [neighbor]))

    return None
```

## Breadth-First Search Algorithm

Logic:

- Uses a queue (FIFO) to explore level by level.
- Guarantees shortest path in unweighted graphs.

**Python Code for BFS**

```
from collections import deque


def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        node, path = queue.popleft()
        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in graph.get(node, {}):
                queue.append((neighbor, path + [neighbor]))

    return None
```

## Depth-First Search Algorithm

**Logic:**

- Uses a stack (LIFO) to explore deeply before backtracking.
- Memory-efficient but not guaranteed to find the best solution.

**Python Code for DFS**

```
def dfs(graph, start, goal):
    stack = [(start, [start])]
    visited = set()

    while stack:
        node, path = stack.pop()
        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in graph.get(node, {}):
                stack.append((neighbor, path + [neighbor]))

    return None
```

## 4. Result Analysis and Comparison

### Comparison of Algorithms

| Algorithm | Uses Heuristic? | Guarantees Shortest Path? | Space Complexity | Completeness |
|---|---|---|---|---|
| A* Search | ✅ Yes | ✅ Yes | High | ✅ Yes |
| Best First Search | ✅ Yes | ❌ No | Medium | ❌ No |
| BFS | ❌ No | ✅ Yes | High | ✅ Yes |
| DFS | ❌ No | ❌ No | Low | ❌ No |

### Observations:

1. A* Search finds the shortest path most efficiently but requires more memory.
2. Best First Search is fast but can get trapped in local minima.
3. BFS guarantees the shortest path in unweighted graphs but is slower than A*.
4. DFS is better for deep searches but is not guaranteed to find the optimal path.

## 5. Discussion and Conclusion

- A* Search is the best choice when both efficiency and accuracy are required, making it widely used in robotics, navigation, and AI decision-making.
- Best First Search is suitable for quick searches but does not always find the optimal solution.
- BFS is preferred when shortest paths are needed in unweighted graphs.
- DFS is useful for deep exploration but may not be the best for pathfinding.

Thus, A Search remains the most reliable algorithm for optimal pathfinding in AI applications.

## 6. References


- Stuart Russell, Peter Norvig - "Artificial Intelligence: A Modern Approach"
- Wikipedia: A* Search Algorithm
- GeeksforGeeks: Best First Search Algorithm


[NOTE: Full codes with visualization are available at github.com/ar-sayeem/AI-Lab