# PROJECT REPORT

## A Comparative Analysis of AI Search Algorithms for Real-World Navigation

**Course:** Introduction to Artificial Intelligence
**Team:** The Wayfinders
**Members:** Youssef, Salma, Huda, Yomna, Mahmoud, and Mohammed.
**Submission Date:** December 19, 2025
**GitHub Repository:** GitHub - youssef-darrag/real-world-path-visualizer

## Executive Summary

This report details the implementation and analysis of six fundamental AI search algorithms applied to real-world pathfinding problems. The MapPath Visualizer application demonstrates how Breadth-First Search (BFS), Depth-First Search (DFS), Depth-Limited Search (DLS), Iterative Deepening Search (IDS), Uniform-Cost Search (UCS), and A* Search perform on actual geographical maps using OpenStreetMap data. Our analysis reveals significant differences in algorithm performance, particularly highlighting the efficiency gains achieved through heuristic guidance in A* Search.

# System Architecture

## Technology Stack:

- **Core Language:** Python 3.8
- **GUI Framework:** Tkinter with TkinterMapView extension
- **Map Processing:** OSMnx 1.6.0
- **Graph Operations:** NetworkX 3.1

## Data Flow Pipeline:

1. **Map Acquisition:** Download street network from OpenStreetMap
2. **Graph Conversion:** Convert to NetworkX graph with edge weights
3. **Algorithm Execution:** Run selected search algorithm
4. **Visualisation:** Display step-by-step exploration on map
5. **Metrics Collection:** Track performance statistics

# Algorithm Implementations

## Breadth-First Search (BFS)

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

    while queue:
        current, path = queue.popleft()
        if current == goal:
            return path
        for neighbor in graph.neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))
    return None
```

*Features: FIFO queue, complete, optimal for unweighted graphs*

## Depth-First Search (DFS) && Depth-Limited Search (DLS)

```
def dfs(graph, start, goal, depth_limit=None):
    stack = [(start, [start], 0)]
    visited = set()

    while stack:
        node, path, depth = stack.pop()

        if node == goal:
            return path

        if depth_limit and depth >= depth_limit:
            continue

        if node not in visited:
            visited.add(node)
            for neighbor in graph.neighbors(node):
                stack.append((neighbor, path + [neighbor], depth + 1))
```

*Features: LIFO stack, incomplete, not optimal, memory efficient*

## Iterative Deepening Search (IDS)

```
def iterative_deepening(graph, start, goal, max_depth=100):
    for depth in range(max_depth):
        result = depth_limited_search(graph, start, goal, depth)
        if result is not None and result != "CUTOFF":
            return result
    return None
```

*Features: Combines BFS completeness with DFS memory efficiency*

## Uniform-Cost Search (UCS)

```
def uniform_cost_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put((0, start, [start]))
    explored = set()

    while not frontier.empty():
        cost, current, path = frontier.get()
        if current == goal:
            return path, cost
        if current not in explored:
            explored.add(current)
            for neighbor in graph.neighbors(current):
                new_cost = cost + graph[current][neighbor]['weight']
                frontier.put((new_cost, neighbor, path + [neighbor]))
    return None, float('inf')
```

*Features: Priority queue, optimal for weighted graphs*

### A Search*

```python
def a_star_search(graph, start, goal, heuristic):
    frontier = PriorityQueue()
    frontier.put((heuristic(start, goal), 0, start, [start]))
    g_costs = {start: 0}

    while not frontier.empty():
        _, g_cost, current, path = frontier.get()
        if current == goal:
            return path, g_cost
        for neighbor in graph.neighbors(current):
            new_g = g_cost + graph[current][neighbor]['weight']
            if neighbor not in g_costs or new_g < g_costs[neighbor]:
                g_costs[neighbor] = new_g
                f_cost = new_g + heuristic(neighbor, goal)
                frontier.put((f_cost, new_g, neighbor, path + [neighbor]))
    return None, float('inf')
```

*Features: f(n) = g(n) + h(n), heuristic: Euclidean distance*

# Visualization System

- **Real-time map display** with OpenStreetMap tiles
- **Color-coded visualization:**
  - Green: Start node
  - Red: Goal node
  - Blue: Explored nodes
  - green: Frontier nodes
  - Purple: Optimal path
- **Interactive controls:** Play, reset, random selection
- **Statistics panel:** Real-time performance metrics

# Algorithm Analysis

## Theoretical Analysis

| Algorithm | Complete | Optimality | TC | SC |
|---|---|---|---|---|
| BFS | Yes | Yes* | O(b^d) | O(b^d) |
| DFS | No | No | O(b^m) | O(bm) |
| DLS | No | No | O(b^l) | O(bl) |
| IDS | Yes | Yes* | O(b^d) | O(bd) |
| UCS | Yes | Yes | O(b^(C*/ε)) | O(b^(C*/ε)) |
| A* | Yes | Yes** | O(b^d) | O(b^d) |

*Optimal for unweighted graphs only
*With admissible heuristic

**TC** = time Complexity

**SC** = space Complexity

**b** = branching factor

**d** = depth

**m** = maximum depth

**l** = depth limit

**C*** = Optimal solution cost

**ε** = minimum edge cost

# Practical Behavior Observations

## BFS Behavior:

- Expands uniformly in all directions
- Guarantees shortest path in terms of node count
- Performs poorly on weighted graphs
- High memory consumption on dense graphs

## DFS Behavior:

- Tends to explore deep branches first
- Can get stuck in infinite loops without cycle checking
- Memory efficient but often finds suboptimal paths
- Performs poorly on graphs with many dead ends

## UCS Behavior:

- Systematically expands cheapest paths first
- Optimal for weighted graphs but slower than A*
- Frontier grows quickly in complex graphs
- Requires edge weight information

## A* Behavior:

- Guided directly toward goal by heuristic
- Most efficient with good heuristic
- Balance between exploration and exploitation
- Performance depends heavily on heuristic quality

# Performance Comparison

## Experimental Setup

**Test Environment:**

- Hardware: ryzen 5 5500, 16GB RAM
- Software: Arch linux, Python 3.8
- Test Maps: Three urban areas of varying complexity
- Map Sizes: Small (100 nodes), Medium (500 nodes), Large (1000+ nodes)

**Metrics Collected:**

1. Execution time (milliseconds)
2. Nodes explored before solution
3. Path length (meters)
4. Memory Usage
5. Solution optimality (compared to ground truth)

## Performance Results: Table 1: Small Map Performance (100 nodes)

| Algorithm | Time (ms) | Nodes Explored | Path Length | Memory Usage | Optimal? |
|---|---|---|---|---|---|
| BFS | 45 | 62 | 850m | 85 | No |
| DFS | 32 | 28 | 1200m | 42 | No |
| DLS (d=15) | 38 | 35 | - | 45 | - |
| IDS | 52 | 65 | 850m | 55 | Yes |
| UCS | 68 | 48 | 780m | 72 | Yes |
| A* | 25 | 18 | 780m | 30 | Yes |

## Table 2: Medium Map Performance (500 nodes)

| Algorithm | Time (ms) | Nodes Explored | Path Length | Memory Usage | Optimal? |
|---|---|---|---|---|---|
| BFS | 210 | 285 | 1450m | 320 | No |
| DFS | 155 | 142 | 2100m | 180 | No |
| DLS (d=25) | 165 | 158 | - | 195 | - |
| IDS | 245 | 290 | 1450m | 210 | Yes |
| UCS | 320 | 210 | 1320m | 285 | Yes |
| A* | 95 | 65 | 1320m | 85 | Yes |

## Table 3: Large Map Performance (1000+ nodes)

| Algorithm | Time (ms) | Nodes Explored | Path Length | Memory Usage | Optimal? |
|---|---|---|---|---|---|
| BFS | 520 | 642 | 2100m | 750 | No |
| DFS | 380 | 325 | 2850m | 420 | No |
| DLS (d=40) | 420 | 385 | - | 480 | - |
| IDS | 580 | 655 | 2100m | 520 | Yes |
| UCS | 780 | 485 | 1980m | 650 | Yes |
| A* | 185 | 128 | 1980m | 165 | Yes |

# Key Performance Insights

## Time Efficiency Ranking:

1. **A*** (Fastest - heuristic guidance)
2. **DFS** (Depth-first exploration)
3. **DLS** (Controlled depth)
4. **BFS** (Systematic but exhaustive)
5. **IDS** (Repeated depth-limited searches)
6. **UCS** (Cost-based, computationally intensive)

## Memory Efficiency Ranking:

1. **DFS/DLS** (Depth-based, smaller frontier)
2. **A*** (Heuristic-guided frontier)
3. **IDS** (Depth-limited memory usage)
4. **UCS** (Priority queue management)
5. **BFS** (Large frontier storage)

## Solution Quality:

- **UCS and A*** consistently found optimal paths
- **BFS and IDS** optimal only for unweighted paths
- **DFS** often found significantly longer paths
- **DLS** sometimes failed to find solution within depth limit

# Real-World Considerations

### Urban Network Characteristics Affecting Performance:

1. **Branching Factor Variation:** Intersections vs. straight roads
2. **Dead Ends:** Challenge DFS and DLS
3. **Weight Distribution:** Affects UCS and A*
4. **Geographical Constraints:** Rivers, parks create search barriers
5. **Network Density:** Dense urban vs. sparse suburban

# Project Achievements

**Complete Implementation:** All six algorithms correctly implemented
**Real-World Integration:** Successful use of OpenStreetMap data
**Educational Visualization:** Clear step-by-step algorithm display
**Performance Analysis:** Comprehensive metrics collection
**Comparative Insights:** Clear algorithm trade-off analysis

# Team Contributions

- **Youssef** : Algorithm implementation (A*), testing and document integration
- **Salma** : Algorithm implementation (DFS & DLS) and testing integration
- **Huda** : Algorithm implementation (BFS & IDS), testing and document integration.
- **Yomna** : Algorithm implementation (UCS), testing and control integration
- **Mahmoud** : GUI, real-map visualization system and testing analysis
- **Mohamed** : Grid-visualization and performance analysis