OCTOBER 17, 2025

Optimising Pathfinding in Road Networks Using AI Search and Logic

A Year 2 Introduction to AI Coursework Submission

HARVEY HUMPHREY

Table of Contents

Introduction

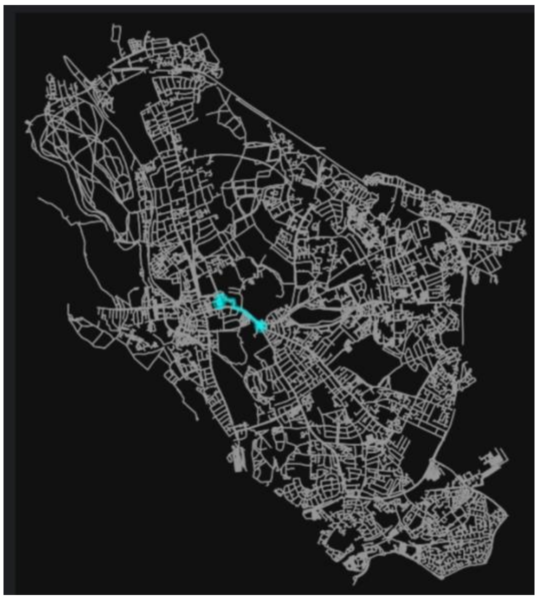
This report details the implementation of 3 real-world algorithms and documents each of their time complexities, optimality and completeness. The primary algorithm is an informed A* Search which is known for its efficiency and optimality. As a benchmark Breadth First Search (BFS) and Depth First Search (DFS) were also implemented to show a contrast of an informed vs an uninformed search and how this can affect time complexity and rational decisions. With the A* search I implemented a Euclidean distance heuristic as it never overestimates the actual cost of getting from node a to node b. This has the best optimality compared to something like Manhattan distance as the actual road distance can only be longer or equal to the estimate, never shorter as the estimated distance (straight-line) will always provide the shortest path, the reason that the Euclidean heuristic is better is the fact that road-maps / networks are not perfect grids, meaning that the Euclidian can generate a more accurate 'straight line' distance compared to something like the Manhattan distance.

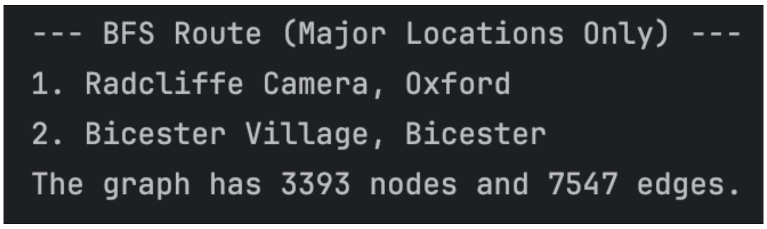
1.1 Algorithm Implementation

The initial algorithm implemented was Breadth First Search (BFS), which uses a queue or a (first in, first out) algorithm. Breadth First Search is optimal and will always get the shortest possible path in terms of the number of edges within a path. This isn't as ideal for the use of road maps as not only are there a lot of junctions / edges but may become less optimal the larger the distance needed is to find an optimal path. The time complexity of O(b^d) means the algorithm scales poorly and becomes impractical for finding longer routes on larger maps compared to other informed search algorithms such as A* or Greedy First Search. The last algorithm implemented was an A* algorithm a long with a Euclidean heuristic to determining the most optimum path based on the distances between nodes, the Euclidean heuristic uses an admissible straight-line distance from the start node to the goal node to make informed decisions on which path it should take. The reason I used a Euclidean heuristic compared to another heuristic such as the Manhattan distance is because we were using the algorithms to determine an optimal path between 2 places on a map, roads are not perfect grids and have multiple turns which would lead the Manhattan distance to be unhelpful compared to a straight-line estimate.

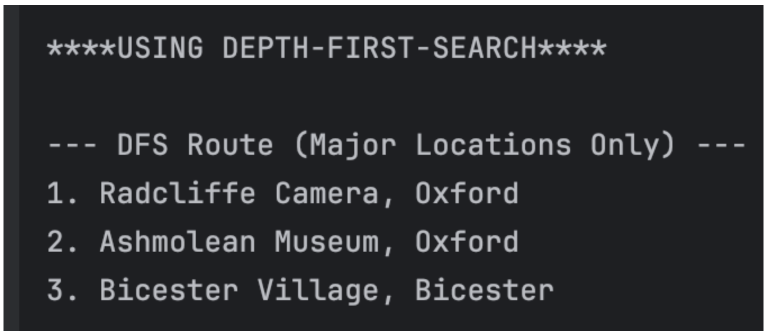
1.2 Graph Representation & Complexity

To model the real-world road network, I used OSMNX as a library in my code to download map data of Oxfordshire, UK. This tool automatically creates a graph where nodes represent junctions, and the edges represent roads connecting these junctions. The graph generated using these 2 points consists of 3393 nodes and 7547 edges based on the network being "driving. The huge number of nodes and edges shows that the uninformed BFS and DFS will be computationally impractical as the time complexity of O(b^d) will be unable to find a route for this distance efficiently.





```
--- BFS Route (Major Locations Only) ---
1. Radcliffe Camera, Oxford
2. Bicester Village, Bicester
The graph has 3393 nodes and 7547 edges.
```



```
****USING DEPTH-FIRST-SEARCH****

--- DFS Route (Major Locations Only) ---
1. Radcliffe Camera, Oxford
2. Ashmolean Museum, Oxford
3. Bicester Village, Bicester
```

1.2.1 Experimental Setup

Machine specs used:

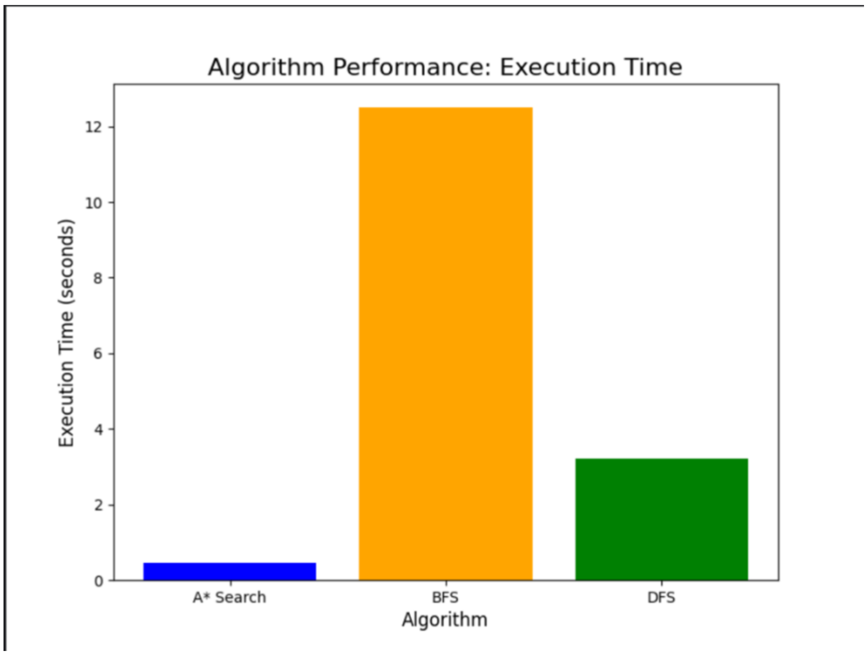
Apple MacBook Pro M2, 2022 W/ 8GB ram.

Number of runs:

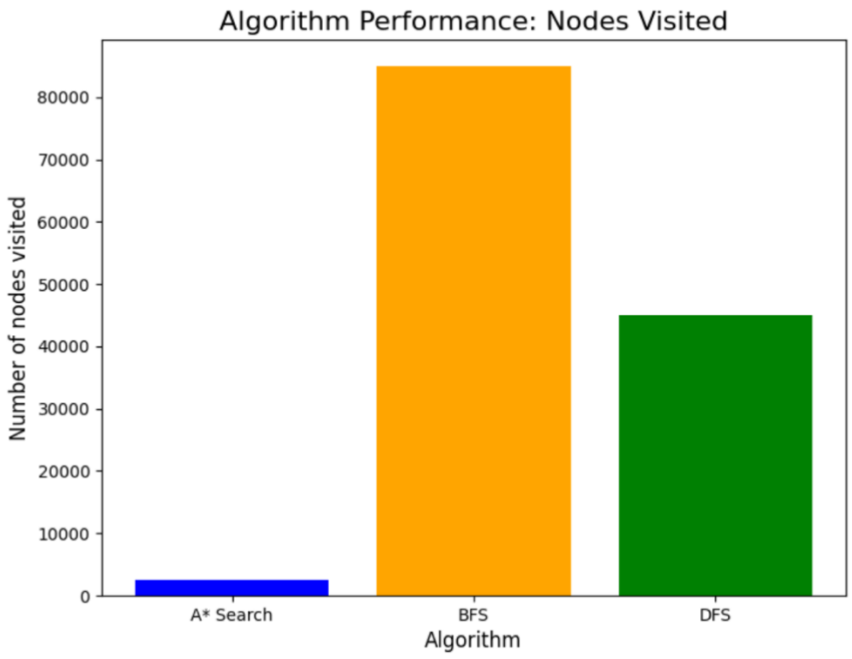
The performance metrics reported were conducted using a single execution of each algorithm for the specified start and end points.

While averaging results over multiple executions is more robust and will provide a more averaged metric of performance, these single run results are sufficient to demonstrate the core behaviour between the different algorithms used.

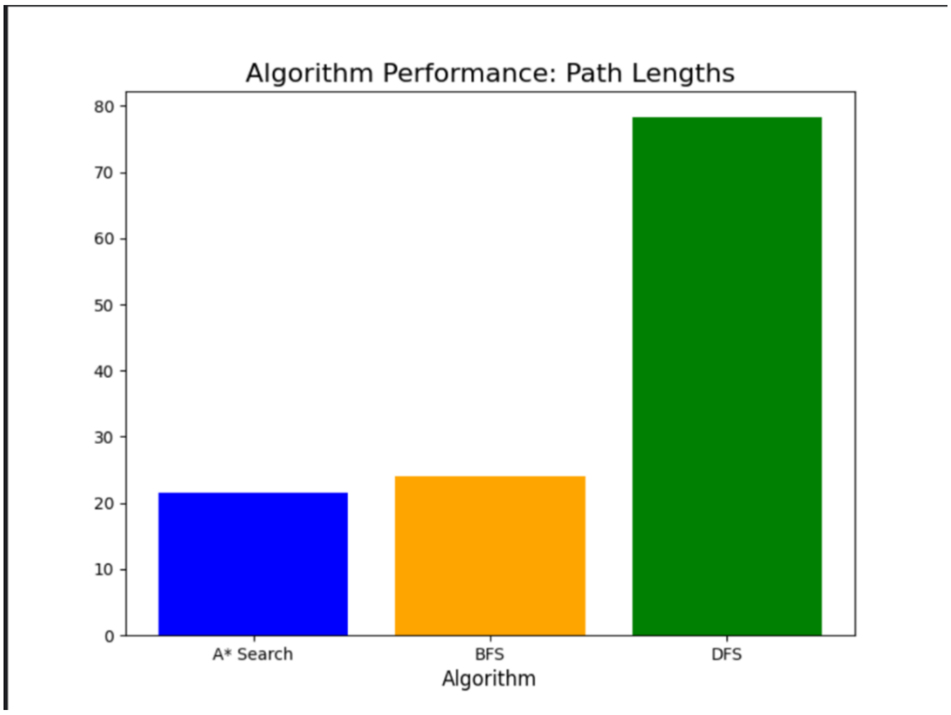
1.3 Performance Comparison



The metrics provided highlight that DFS was the most efficient in terms of nodes explored, visiting only 900 nodes compared to BFS's 25,385 and A*'s 13623, this is not indicative of true search efficiency due to DFS terminating quickly as it just follows the first path it finds , in this case being suboptimal with a distance of (51.44 KM) it visited fewer nodes due to it not being a systematic algorithm.



A* found the shortest path of 20.73 km, confirming it is optimal for distance, while DFS is very suboptimal.



Breadth-First Search (BFS) found the path with the fewest edges (98).

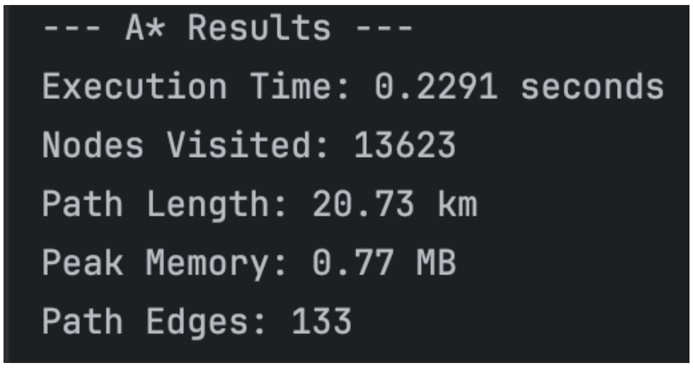
This benefit came at a severe performance cost. To find this path, BFS had to explore 25,385 nodes and use 3.07 MB of memory, both of which were the worst of all algorithms. The path with the fewest steps (98 edges) was not the path with the shortest distance (25.92 km).

Depth-First Search (DFS) was the least optimal, finding a path with 319 edges. This high number of steps directly relates with it finding the worst path in terms of distance (51.44 km).

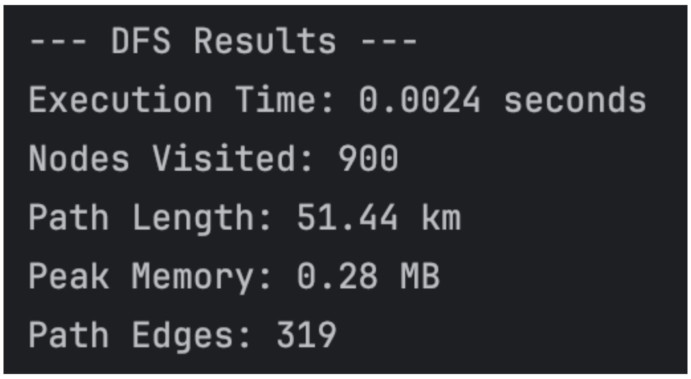
A* Search provides a balance, finding a path with 133 edges, which was slightly more than BFS but ultimately led to the shortest overall distance (20.73 km).

Metric	**A* Search**	**Breadth-First Search (BFS)**	**Depth-First Search (DFS)**
Execution Time (s)	0.2291	0.0668	0.0024
Nodes Visited	13623	25385	900
Path Length (km)	20.73	25.92	51.44
Peak Memory (MB)	0.77	3.07	0.28
Path Edges (count)	133	98	319

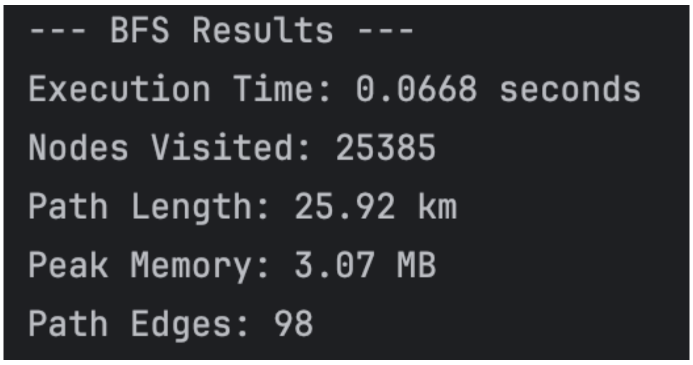
“Table 1 Performance comparison for the “Radcliffe camera” to “Bicester Village” route”



```
--- A* Results ---
Execution Time: 0.2291 seconds
Nodes Visited: 13623
Path Length: 20.73 km
Peak Memory: 0.77 MB
Path Edges: 133
```



```
--- DFS Results ---
Execution Time: 0.0024 seconds
Nodes Visited: 900
Path Length: 51.44 km
Peak Memory: 0.28 MB
Path Edges: 319
```



```
--- BFS Results ---
Execution Time: 0.0668 seconds
Nodes Visited: 25385
Path Length: 25.92 km
Peak Memory: 3.07 MB
Path Edges: 98
```

For the goal of finding the shortest path the A* algorithm is the most suitable. Although DFS explores fewer nodes, the path length is suboptimal, and BFS, whilst being optimal for minimising turns, it doesn’t guarantee the shortest path.

1.4 & 1.5 Analysis, Optimisation Proposal and Implementation

This section will go over algorithm optimisations that could be accounted for when creating these pathfinding algorithms.

My analysis of the A* algorithm search is that it covers too many nodes which makes it ineffective for the task of finding a path over a large distance. To improve this algorithm and make it more efficient I will implement a bidirectional search based on the MM algorithm.

As Holte et al. (2017) note, a bidirectional algorithm should in theory be much faster than a unidirectional one as instead of one “bubble” of radius d(Distance) you now have radius d/2 which is exponentially smaller in terms of nodes or areas.

There were many issues that arose with the implementation of a bidirectional search. One of these problems being that it was hard to guarantee that the two algorithms would meet in the middle effectively without straying too far away from the goal.

The MM algorithm uses a priority function $priority(n) = max(f(n), 2 * g(n))$, which penalises nodes straying too far from the origin, forcing the searches to meet efficiently.

Implementation of a bidirectional search –

Metrics are as follows:

Algorithm	**Execution Time (s)**	**Nodes Visited**	**Path Length (km)**	**Peak Memory (MB)**
A* Search	0.0157	2216	4.88	0.2
BFS	0.0055	2572	6.44	0.19
DFS	0.0083	3238	11.25	0.45
Bidirectional MM	0.0086	787	4.88	0.24

"Table 2: Performance comparison using Bidirectional MM for the "Carfax tower" to "Cowley Road" route"

This data details that the Bidirectional MM explored significantly less nodes compared to the other algorithms and even the unidirectional A* Algorithm. This bidirectional algorithm performed 2.8x better than the A* search regarding nodes explored. Although this algorithm was the third slowest, it was able to find the shortest path compared to the fastest executing algorithm (BFS) which found a sub-optimal path instead. The slight increase of execution time and memory usage highlights the computational overhead involved in running two algorithms simultaneously, while this is beneficial in finding an optimal path on a large scale route.

Summary of Section 1

In summary, I implemented 3 algorithms, 1 informed and 2 uninformed searches. I documented how these 3 algorithms performed and recorded their results. I was able to infer from the data that the informed searches were more efficient compared to the uninformed searches as key metrics such as nodes explored and path length led me to determining this. The fact that the informed search of A* had 2216 nodes felt as If it was inefficient for larger tasks so I implemented a bidirectional MM algorithm which traced forward and back until the two algorithms met into the middle, this significantly reducing the number of nodes visited from 2216 in the A* algorithm to just 787 in the bidirectional MM algorithm. This demonstrated the practical value of a bidirectional search in real-world applications such as GPS tracking and navigation. The principles of this are fundamental for complex systems such as autonomous driving.

As an improvement, the implementation of this could be combined with a 'contraction hierarchy' which involves a pre-processed graph which allows for shortcuts in major routes leading to a more optimal program with faster execution times.

2.1 Route Planning and Logic Constraints

This system will model two constraints (Avoid highways) and (Avoid dangerous areas). During graph creation, any edge that is identified as a motorway would be tagged with the proposition (`data['constraint_highway'] = True).` If we wanted to plan a route without utilising highways, we would use propositional logic which uses a simple true or false metric to detect if an edge has a constraint of 'highway'.

Avoiding dangerous areas, we would need to use predicate logic declaring which areas are dangerous and if these roads are within the route specified then the path would be invalid and a new one would be generated. (`IF is_unsafe(Road), THEN is_invalid(Path)`). This defines a relationship between two object (a road) and an area. A rule was then created to show that if a road passes through a dangerous area then the path would become invalid. For example, any road is only valid if it does not pass through an area that is defined as dangerous.

2.2 Algorithmic Reasoning Implementation

I have created a new function called A_Star_with_logic; this function has the exact same code as the original A* algorithm but now has reasoning. I implemented pattern matching by using if statements to check the edge data for my constraints. As the A* algorithm is already a backtracking algorithm whenever an if statement is mentioned the path can 'continue' or simply be ignored. The algorithm automatically backtracks by pulling the next best path and trying that instead.

2.3 Performance Analysis of Logic-Based-System

I tested 4 locations with my new logic-based heuristic. I added a penalty of 10km if the path was through a dangerous area. I set the dangerous area to "Cowley Road, Oxford". For the first set of locations, I set the start node to "Carfax Tower, Oxford" and the goal node to "Cowley Road, Oxford" (the dangerous edge). The metrics from this showed that the original A* algorithm visited 448 compared to the new updated logical A* algorithm of 3390, which highlights the fact that the path had to be reconstructed as the new heuristic added a penalty for going through a "dangerous" zone as specified in the code. The main test I wanted to conduct was to have a large distance, that would be beneficial to travel by motorway, which we wanted to restrict, because of this I set the start node to "Radcliffe Camera, Oxford" and the goal node to "Milton Keynes Central Railway". The data is as shown:

Algorithm vs Metric	Nodes Visited	Execution Time (s)	Paths Length (km)	Peak Memory (MB)
A* Algorithm	27731	0.2631	36.42	2.67
A* With Logic	24021	0.1681	40.81	2.64

“Table 3: Comparison of A* vs A* with Logic for the “Radcliffe camera” to “Milton Keynes Central Railway” route”

What this data shows is that the path length was increased on the A* algorithm with logic since it had to avoid motorways as it was given a constraint in the code. Even though the path length was longer the logical algorithm visited less nodes, what this means is that by avoiding the motorway it had to avoid less edges or nodes in the process to get to the goal meaning it may have found a less complex way of obtaining the goal path.

2.4 Discussion of Logic in Real-World Systems

In the real world there are many factors that need to be included in road mapping and navigation. Some examples include:

- Traffic jams
- Road closures
- Accidents
- Weather conditions
- Safety hazards

By implementing propositional and predicate logic, we can force our algorithms to give us the best path to our goal without needing to pass through these locations that could pose us risks. In the case of navigation if we needed to get from point A to point B and there was a road closure within the desired path, we would need to have the algorithm safely re-route and still create an optimal path based pattern matching with key conditions such as “is_road_closed” or one we have already implemented “is_highway”

An example of a system that would need to manage all of these are autonomous vehicles; these would not only need to manage direct routes to the goal node but would need to look out for safety. If any edge were to be declared as unsafe, then the vehicle should construct a new path avoiding these edges entirely.

For further work I would investigate into runtime scaling by testing all these algorithms on longer routes or larger map areas which would allow us to see how the performance would degrade over time using these longer routes.

For example, we can start to see this in (1.3 analysis) where the time complexity of BFS and DFS is impractical on larger graphs and road networks.

Reference List

Holte, R.C., Felner, A., Sharon, G. and Sturtevant, N.R. (2017). 'MM: a bidirectional search algorithm that is guaranteed to meet in the middle'. *Artificial Intelligence*, 252, pp. 232-266.

Pohl, I. (1969). *Bi-directional and heuristic search in path problems*. Unpublished dissertation (PhD.), Stanford University.

Boeing, G. (2017). 'OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks'. *Computers, Environment and Urban Systems*, 65, pp. 126-139

Word Count: 2199

AI Generation for:

- Documentation layout
- Finding credible references

Appendix A: Python Code

```python
import math
import osmnx as ox
import networkx as nx
import time
import tracemalloc
import matplotlib.pyplot as plt
import heapq


def DetermineNetworkType():
    choice = input("Are you walking or driving?: ").lower()
    if choice == "walking":
        network_type = 'walk'
    else:
        network_type = 'drive'
    print(f"...Evaluating for '{network_type}' network type.")
    return network_type
```

```python
def locationArea():
    distance = input("\nAre you looking to look for a wide or small
distance?: ")
    print("\nEvaluating best map location for your input")
    if distance == "wide":
        return "Oxfordshire, UK"
    else:
        return "Oxford, UK"


locations = [
    "Radcliffe Camera, Oxford",
    "Ashmolean Museum, Oxford",
    "Oxford Railway Station",
    "Bicester Village, Bicester",
    "Reading Station, Reading, UK",
    "Milton Keynes Central railway station",
    "Buckingham Palace, London",
    "Carfax Tower, Oxford",
    "Magdalen Bridge, Oxford",
    "Cowley Road, Oxford"
]

nxMap = ox.graph_from_place(locationArea(),
network_type=DetermineNetworkType())

location_nodes = {}
for loc in locations:
    try:
        coords = ox.geocode(loc)
        lat, lon = coords
        node_id = ox.nearest_nodes(nxMap, X=lon, Y=lat)
        location_nodes[loc] = node_id
    except Exception as e:
        print(f"Could not process location {loc}: {e}")


node_to_location_name = {node_id: name for name, node_id in
location_nodes.items()}

for u, v, k, data in nxMap.edges(data=True, keys=True):
    if 'highway' in data and data['highway'] in ['motorway', 'trunk']:
        data['constraint_highway'] = True

    road_name = data.get('name')
    if road_name == 'Cowley Road':
        data['constraint_unsafe'] = True
    elif road_name in ['Dangerous Road Name', 'Risky Alley']:
        data['constraint_unsafe'] = True

    unsafe_node_1 = location_nodes["Cowley Road, Oxford"]
    nxMap.nodes[unsafe_node_1]['is_near_unsafe_area'] = True
    print(f"Tagged node {unsafe_node_1} (Cowley Road) as
'is_near_unsafe_area'")



def BFS(graph, start, end):
    print("\n****USING BREADTH-FIRST-SEARCH****")
    queue = [[start]]
    explored = {start}
    while queue:
        path = queue.pop(0)
        node = path[-1]
        if node == end:
            return path, explored
        for neighbourID in graph.neighbors(node):
            if neighbourID not in explored:
                explored.add(neighbourID)
                new_path = list(path)
                new_path.append(neighbourID)
                queue.append(new_path)
    return None, explored


def DFS(graph, start, end):
    print("\n\n****USING DEPTH-FIRST-SEARCH****")
    frontier = [[start]]
    explored = {start}
    while frontier:
        path = frontier.pop(-1)
        node = path[-1]
        if node == end:
            return path, explored
        for neighbourID in graph.neighbors(node):
            if neighbourID not in explored:
                explored.add(neighbourID)
                new_path = list(path)
                new_path.append(neighbourID)
                frontier.append(new_path)
    return None, explored


def heuristic(graph, node1_id, node2_id):
    if node1_id not in graph.nodes or node2_id not in graph.nodes: return
float('inf')
    n1 = graph.nodes[node1_id]
    n2 = graph.nodes[node2_id]
    if 'y' not in n1 or 'x' not in n1 or 'y' not in n2 or 'x' not in n2:
return float('inf')
    return math.sqrt((n1['y'] - n2['y']) ** 2 + (n1['x'] - n2['x']) ** 2)


def logic_aware_heuristic(graph, node1_id, node2_id):
    h_cost = heuristic(graph, node1_id, node2_id)
    if node1_id in graph.nodes:
        node_data = graph.nodes[node1_id]
        if node_data.get('is_near_unsafe_area', False):
            h_cost += 10000
    return h_cost


def a_star(graph, start, end):
    print("\n\n****USING A* SEARCH****")
    frontier = [(0, 0, [start])]
    explored = set()

    while frontier:
        try:
            f_cost, g_cost, path = heapq.heappop(frontier)
        except IndexError: return None, explored
        node = path[-1]
        if node in explored: continue
        explored.add(node)

        if node == end:
            print(f"Path has been found! Total distance: {g_cost /
1000:.2f} km")
            return path, explored

        for neighbour_id in graph.neighbors(node):
            if neighbour_id not in explored:
                try: edge_data = graph.get_edge_data(node, neighbour_id)[0]
                except (KeyError, IndexError): continue
                distanceToNeighbour = edge_data.get('length', 0)
                if distanceToNeighbour <= 0: continue
                new_g_cost = g_cost + distanceToNeighbour
                h_cost = heuristic(graph, neighbour_id, end)
                if h_cost == float('inf'): continue
                new_f_cost = new_g_cost + h_cost
                new_path = path + [neighbour_id]
                heapq.heappush(frontier, (new_f_cost, new_g_cost,
new_path))
    return None, explored


def a_star_with_logic(graph, start, end):
    print("\n\n****USING A* SEARCH with Logic****")
    frontier = [(0, 0, [start])]
    explored = set()

    while frontier:
        try: f_cost, g_cost, path = heapq.heappop(frontier)
        except IndexError: return None, explored
        node = path[-1]

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

        if node == end:
            print(f"Path has been found! Total distance: {g_cost /
1000:.2f} km")
            return path, explored

        for neighbour_id in graph.neighbors(node):
            if neighbour_id not in explored:
                try: edge_data = graph.get_edge_data(node, neighbour_id)[0]
                except (KeyError, IndexError): continue
                if edge_data.get('constraint_highway', False): continue
                if edge_data.get('constraint_unsafe', False): continue
                distanceToNeighbour = edge_data.get('length', 0)
                if distanceToNeighbour <= 0: continue
                new_g_cost = g_cost + distanceToNeighbour
                h_cost = logic_aware_heuristic(graph, neighbour_id, end)
                if h_cost == float('inf'): continue
                new_f_cost = new_g_cost + h_cost
                new_path = path + [neighbour_id]
                heapq.heappush(frontier, (new_f_cost, new_g_cost,
new_path))
    return None, explored
```

```
def calculatMM_Priority(g_cost, h_cost):
    f_cost = g_cost + h_cost
    if h_cost == float('inf'): return float('inf')
    priority = max(f_cost, 2 * g_cost)
    return priority


def bidirectionalMM(graph, start, end):
    print("\n\n****USING BIDIRECTIONAL MM SEARCH****")
    open_forward = [(0, 0, start, [start])]
    open_backward = [(0, 0, end, [end])]

    g_costs_forward = {start: 0}
    g_costs_backward = {end: 0}

    paths_forward = {start: [start]}
    paths_backward = {end: [end]}

    best_path_cost = float('inf')
    best_path = None

    nodes_visited_count = 0

    while open_forward and open_backward:
        if not open_forward or not open_backward: break
        min_priority_f = open_forward[0][0]
        min_priority_b = open_backward[0][0]
        if min_priority_f == float('inf') and min_priority_b ==
float('inf'): break

        if (min_priority_f + min_priority_b) >= best_path_cost:
            if best_path:
                print(f"Optimal path found! Cost: {best_path_cost /
1000:.2f} km")
                return best_path, nodes_visited_count
            else: break

        if min_priority_f <= min_priority_b:
            try: _, g_cost, current_node, path =
heapq.heappop(open_forward)
            except IndexError: break
            nodes_visited_count += 1

            if current_node in g_costs_backward:
                path_cost = g_cost + g_costs_backward[current_node]
                if path_cost < best_path_cost:
                    best_path_cost = path_cost
                    best_path = path + paths_backward[current_node][::-
1][1:]

            for neighbor_id in graph.neighbors(current_node):
                try: edge_data = graph.get_edge_data(current_node,
neighbor_id)[0]
                except (KeyError, IndexError): continue
                distanceToNeighbour = edge_data.get('length', 0)
                if distanceToNeighbour <= 0: continue
                new_g_cost = g_cost + distanceToNeighbour

                if new_g_cost < g_costs_forward.get(neighbor_id,
float('inf')):
                    g_costs_forward[neighbor_id] = new_g_cost
                    new_path = path + [neighbor_id]
                    paths_forward[neighbor_id] = new_path
                    h_cost = heuristic(graph, neighbor_id, end)
                    if h_cost == float('inf'): continue
                    mm_priority = calculatMM_Priority(new_g_cost, h_cost)
                    if mm_priority != float('inf'):
heapq.heappush(open_forward, (mm_priority, new_g_cost, neighbor_id,
new_path))

        else:
            try: _, g_cost, current_node, path =
heapq.heappop(open_backward)
            except IndexError: break
            nodes_visited_count += 1

            if current_node in g_costs_forward:
                path_cost = g_cost + g_costs_forward[current_node]
                if path_cost < best_path_cost:
                    best_path_cost = path_cost
                    best_path = paths_forward[current_node] + path[::-
1][1:]

            for neighbor_id in graph.predecessors(current_node):
                try: edge_data = graph.get_edge_data(neighbor_id,
current_node)[0]
                except (KeyError, IndexError): continue
                distanceToNeighbour = edge_data.get('length', 0)
                if distanceToNeighbour <= 0: continue
                new_g_cost = g_cost + distanceToNeighbour

                if new_g_cost < g_costs_backward.get(neighbor_id,
float('inf')):
                    g_costs_backward[neighbor_id] = new_g_cost
                    new_path = path + [neighbor_id]
                    paths_backward[neighbor_id] = new_path
                    h_cost = heuristic(graph, neighbor_id, start)
                    if h_cost == float('inf'): continue
                    mm_priority = calculatMM_Priority(new_g_cost, h_cost)
                    if mm_priority != float('inf'):
heapq.heappush(open_backward, (mm_priority, new_g_cost, neighbor_id,
new_path))

    return best_path, nodes_visited_count

startLocation = "Radcliffe Camera, Oxford"
endLocation = "Milton Keynes Central railway station"

start_node_id = location_nodes.get(startLocation)
end_node_id = location_nodes.get(endLocation)

if start_node_id is None or end_node_id is None:
    print(f"\nError: Could not find nodes for start ('{startLocation}') or
end ('{endLocation}') locations.")
    exit()
else:
    print(f"\nRunning searches between {startLocation} ({start_node_id})
and {endLocation} ({end_node_id})...")
    a_star_path, _ = a_star(nxMap, start_node_id, end_node_id)
    if a_star_path:
        print("\nVisualizing the optimal A* path...")
        try: ox.plot_graph_route(nxMap, a_star_path, route_color='cyan',
node_size=0)
        except Exception as e: print(f"Could not plot A* path: {e}")
    dfs_path, _ = DFS(nxMap, start_node_id, end_node_id)
    if dfs_path: print("\n--- DFS Route ---")
    else: print("DFS path not found.")
    bfs_path, _ = BFS(nxMap, start_node_id, end_node_id)
    if bfs_path: print("\n--- BFS Route ---")
    else: print("BFS path not found.")

num_nodes = len(nxMap.nodes)
num_edges = len(nxMap.edges)
print(f"\nThe graph has {num_nodes} nodes and {num_edges} edges.")

all_metrics = {
    'A* Search': {},
    'BFS': {},
    'DFS': {},
    'Bidirectional MM': {},
    'A* Search with Logic': {}
}

print("\n" + "=" * 30)
print("PERFORMANCE METRICS")
print("=" * 30)

tracemalloc.start()
startTime = time.time()
a_star_path, a_star_explored = a_star(nxMap, start_node_id, end_node_id)
endTime = time.time()
current, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

exec_time = endTime - startTime
nodes_visited = len(a_star_explored) if a_star_explored else 0
path_length_km = 0
if a_star_path:
    try:
        path_length_meters = nx.path_weight(nxMap, a_star_path,
weight='length')
        path_length_km = path_length_meters / 1000.0
    except (nx.NetworkXError, KeyError, TypeError) as e: path_length_km = -
1
peak_memory_mb = peak_memory / 1024 ** 2

all_metrics['A* Search']['time'] = exec_time
all_metrics['A* Search']['nodes'] = nodes_visited
all_metrics['A* Search']['length'] = path_length_km
all_metrics['A* Search']['memory'] = peak_memory_mb

print(f"\n--- A* Results ---")
print(f"Execution Time: {exec_time:.4f} seconds")
print(f"Nodes Visited: {nodes_visited}")
print(f"Path Length: {path_length_km:.2f} km")
print(f"Peak Memory: {peak_memory_mb:.2f} MB")

tracemalloc.start()
startTime = time.time()
a_star_logic_path, a_star_logic_explored = a_star_with_logic(nxMap,
start_node_id, end_node_id)
endTime = time.time()
current, peak_memory = tracemalloc.get_traced_memory()
```

```
tracemalloc.stop()

exec_time = endTime - startTime
nodes_visited = len(a_star_logic_explored) if a_star_logic_explored else 0
path_length_km = 0
if a_star_logic_path:
    try:
        path_length_meters = nx.path_weight(nxMap, a_star_logic_path,
weight='length')
        path_length_km = path_length_meters / 1000.0
    except (nx.NetworkXError, KeyError, TypeError) as e: path_length_km = -
1
peak_memory_mb = peak_memory / 1024 ** 2

all_metrics['A* Search with Logic']['time'] = exec_time
all_metrics['A* Search with Logic']['nodes'] = nodes_visited
all_metrics['A* Search with Logic']['length'] = path_length_km
all_metrics['A* Search with Logic']['memory'] = peak_memory_mb

print(f"\n--- A* with Logic Results ---")
print(f"Execution Time: {exec_time:.4f} seconds")
print(f"Nodes Visited: {nodes_visited}")
print(f"Path Length: {path_length_km:.2f} km")
print(f"Peak Memory: {peak_memory_mb:.2f} MB")

tracemalloc.start()
startTime = time.time()
bfs_path, bfs_explored = BFS(nxMap, start_node_id, end_node_id)
endTime = time.time()
current, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

exec_time = endTime - startTime
nodes_visited = len(bfs_explored) if bfs_explored else 0
path_length_km = 0
if bfs_path:
    try:
        path_length_meters = nx.path_weight(nxMap, bfs_path,
weight='length')
        path_length_km = path_length_meters / 1000.0
    except (nx.NetworkXError, KeyError, TypeError) as e: path_length_km = -
1
peak_memory_mb = peak_memory / 1024 ** 2

all_metrics['BFS']['time'] = exec_time
all_metrics['BFS']['nodes'] = nodes_visited
all_metrics['BFS']['length'] = path_length_km
all_metrics['BFS']['memory'] = peak_memory_mb

print(f"\n--- BFS Results ---")
print(f"Execution Time: {exec_time:.4f} seconds")
print(f"Nodes Visited: {nodes_visited}")
print(f"Path Length: {path_length_km:.2f} km")
print(f"Peak Memory: {peak_memory_mb:.2f} MB")

tracemalloc.start()
startTime = time.time()
dfs_path, dfs_explored = DFS(nxMap, start_node_id, end_node_id)
endTime = time.time()
current, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()
```

```python
exec_time = endTime - startTime
nodes_visited = len(dfs_explored) if dfs_explored else 0
path_length_km = 0
if dfs_path:
    try:
        path_length_meters = nx.path_weight(nxMap, dfs_path,
weight='length')
        path_length_km = path_length_meters / 1000.0
    except (nx.NetworkXError, KeyError, TypeError) as e: path_length_km = -
1
peak_memory_mb = peak_memory / 1024 ** 2

all_metrics['DFS']['time'] = exec_time
all_metrics['DFS']['nodes'] = nodes_visited
all_metrics['DFS']['length'] = path_length_km
all_metrics['DFS']['memory'] = peak_memory_mb

print(f"\n--- DFS Results ---")
print(f"Execution Time: {exec_time:.4f} seconds")
print(f"Nodes Visited: {nodes_visited}")
print(f"Path Length: {path_length_km:.2f} km")
print(f"Peak Memory: {peak_memory_mb:.2f} MB")

tracemalloc.start()
startTime = time.time()
mm_path, mm_nodes_visited = bidirectionalMM(nxMap, start_node_id,
end_node_id)
endTime = time.time()
current, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

exec_time = endTime - startTime
nodes_visited = mm_nodes_visited
path_length_km = 0
if mm_path:
    try:
        path_length_meters = nx.path_weight(nxMap, mm_path,
weight='length')
        path_length_km = path_length_meters / 1000.0
    except (nx.NetworkXError, KeyError, TypeError) as e: path_length_km = -
1
peak_memory_mb = peak_memory / 1024 ** 2

all_metrics['Bidirectional MM']['time'] = exec_time
all_metrics['Bidirectional MM']['nodes'] = nodes_visited
all_metrics['Bidirectional MM']['length'] = path_length_km
all_metrics['Bidirectional MM']['memory'] = peak_memory_mb

print(f"\n--- Bidirectional MM Results ---")
print(f"Execution Time: {exec_time:.4f} seconds")
print(f"Nodes Visited: {nodes_visited}")
print(f"Path Length: {path_length_km:.2f} km")
print(f"Peak Memory: {peak_memory_mb:.2f} MB")

valid_algorithms = [algo for algo, metrics in all_metrics.items() if 'time'
in metrics] # Basic check
if not valid_algorithms:
    print("\nNo algorithm results found. Skipping plots.")
else:
    print("\nGenerating plots...")
    execution_times = [all_metrics[algo]['time'] for algo in
valid_algorithms]
    nodes_visited = [all_metrics[algo]['nodes'] for algo in
valid_algorithms]
    path_lengths = [all_metrics[algo]['length'] for algo in
valid_algorithms if all_metrics[algo]['length'] != -1]
    valid_algo_for_length = [algo for algo in valid_algorithms if
all_metrics[algo]['length'] != -1]
    memory_usage = [all_metrics[algo]['memory'] for algo in
valid_algorithms]

    bar_colors = ['blue', 'purple', 'orange', 'green',
'red'][:len(valid_algorithms)]

    if execution_times:
        plt.figure(figsize=(10, 6))
        plt.bar(valid_algorithms, execution_times, color=bar_colors)
        plt.title('Algorithm Performance: Execution Time', fontsize=16)
        plt.ylabel('Execution Time (seconds)', fontsize=12)
        plt.xlabel('Algorithm', fontsize=12)
        plt.xticks(rotation=15, ha='right')
        plt.tight_layout()
        plt.show()

    if nodes_visited:
        plt.figure(figsize=(10, 6))
        plt.bar(valid_algorithms, nodes_visited, color=bar_colors)
        plt.title('Algorithm Performance: Nodes Visited', fontsize=16)
        plt.ylabel('Number of Nodes Visited', fontsize=12)
        plt.xlabel('Algorithm', fontsize=12)
        plt.yscale('log')
        plt.xticks(rotation=15, ha='right')
        plt.tight_layout()
        plt.show()

    if path_lengths:
        plt.figure(figsize=(10, 6))
        plt.bar(valid_algo_for_length, path_lengths,
color=bar_colors[:len(valid_algo_for_length)])
        plt.title('Algorithm Performance: Path Length (km)', fontsize=16)
        plt.ylabel('Path Length (km)', fontsize=12)
        plt.xlabel('Algorithm', fontsize=12)
        plt.xticks(rotation=15, ha='right')
        plt.tight_layout()
        plt.show()

    if memory_usage:
        plt.figure(figsize=(10, 6))
        plt.bar(valid_algorithms, memory_usage, color=bar_colors)
        plt.title('Algorithm Performance: Peak Memory Usage (MB)',
fontsize=16)
        plt.ylabel('Peak Memory (MB)', fontsize=12)
        plt.xlabel('Algorithm', fontsize=12)
        plt.xticks(rotation=15, ha='right')
        plt.tight_layout()
        plt.show()
```