Finding a safe route in case of traffic incidents

Faculty of DS & AI
Autumn semester, 2025

Group 1 - Topic 10

Business AI Lab

TABLE OF CONTENT

INTRODUCTION AND CONTEXT

- Traffic congestion has become a serious issue in Hanoi, causing delays, stress, safety risks and make many people to feel tired and frustrated due to jams, accidents, and weather condition. Traditional navigation systems only focus on distance or time, ignoring safety factors.
- This study develops an intelligent route-finding system that models the city road network as a weighted graph and applies algorithms such as Greedy Best-First Search and A* to identify safe and efficient routes. The system dynamically updates based on real-time traffic data and visualizes the safest paths on a map, contributing to smart and safer transportation in urban areas.

Factors affecting traffic

METHOD

a) Study Area and Data Sources

- This study was conducted using traffic and environmental data collected from Hanoi, Vietnam - a densely populated urban area characterized by frequent congestion and road incidents. The data included geographic information of roads and intersections obtained from OpenStreetMap (OSM), providing details such as coordinates, connectivity, and road lengths.

b) Data Preprocessing

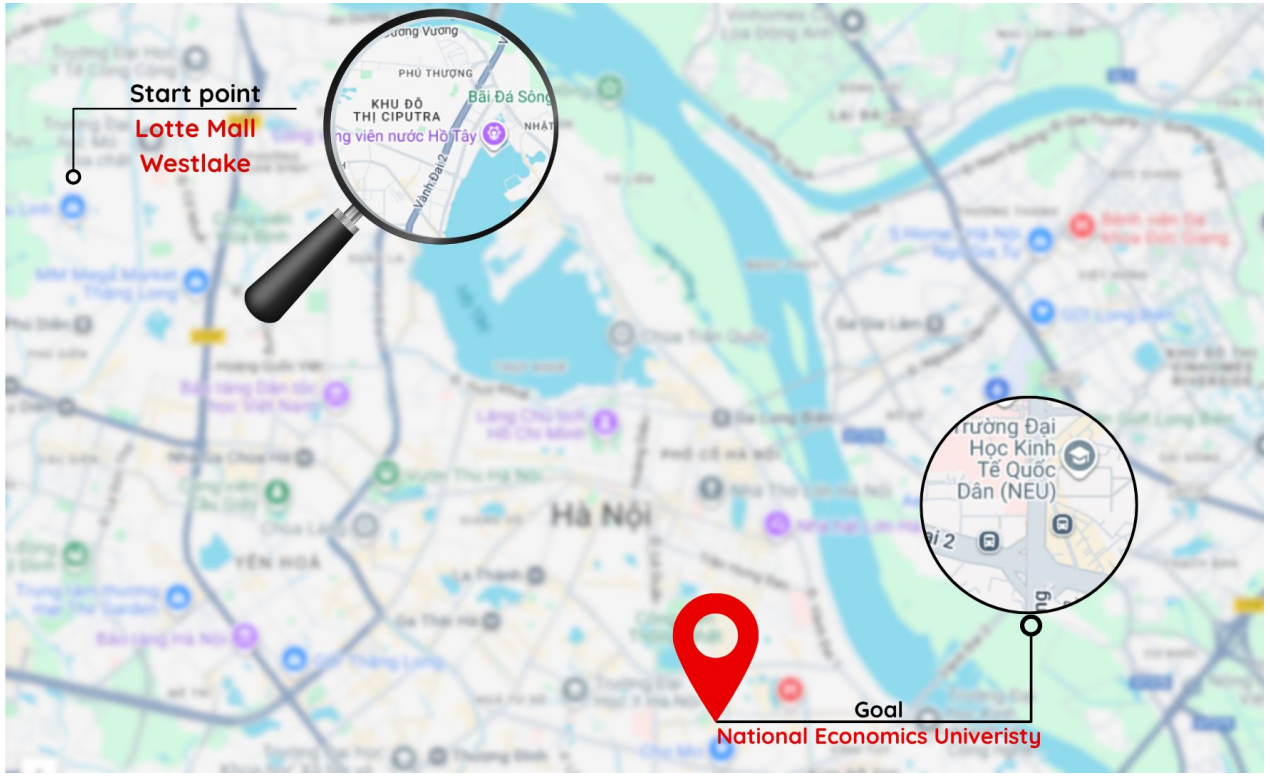
- To ensure accuracy and usability, data cleaning and integration were performed before modeling. Duplicated records, missing coordinates, and incomplete connections were removed. Each road segment was assigned a unique identifier and labeled with the distance (km). Finally, the data were transformed into a weighted graph representing the Hanoi road network.

c) System Overview

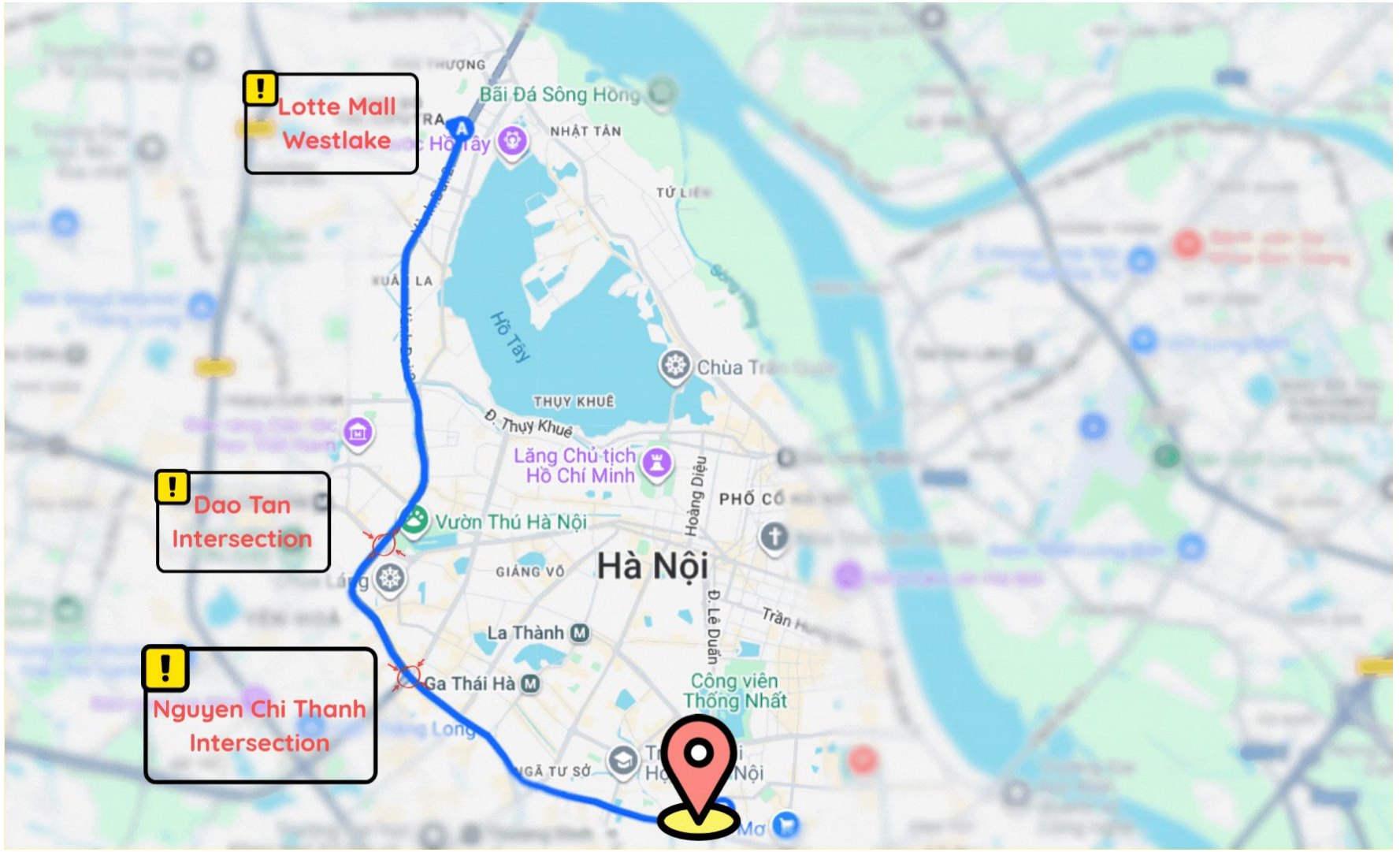
- The system models Hanoi's road network as a weighted graph, where nodes represent intersections and edges represent road segments. Each edge's weight indicates its safety level, determined by factors such as traffic density, distance, accident history, and weather. Two algorithms were implemented: Greedy Best-First Search (GBFS) for fast heuristic-based routing, and A* for optimal balance between speed and accuracy.All algorithms were developed in Python 3.11. The system also includes a dynamic update mechanism that automatically recalculates routes when new incidents, congestion, or weather changes occur, ensuring real-time and safe route recommendations.

DESIGN AND METHODOLOGY

- This section presents the design and implementation methodology for the **traffic-aware route finding problem** using **Greedy Best-First Search** and **A*** algorithms. The objective is to find an alternative path from **Lotte Mall Westlake (Start)** to **National Economics University – NEU (Goal)** when certain roads are blocked due to traffic incidents.



DESIGN AND METHODOLOGY



Which is the most appropriate way?

DESIGN AND METHODOLOGY

- The real-world road network is modeled as a **graph**, where each **node represents a landmark or intersection**, and each e**dge represents a traversable road segment** between two nodes.
- The selected map includes **11 nodes**, corresponding to real locations in Hanoi

DESIGN AND METHODOLOGY

Heuristic and Edge Cost Estimation

1. **Heuristic Cost (h)**
   - The heuristic value from each node to the goal (NEU) is computed using the **Haversine formula**, which measures the great-circle distance between two geographical points.

$$d = 2R \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

2. **Edge Cost (g)**
   - The actual travel distance or cost between two connected nodes is obtained from Google Maps, ensuring real-world accuracy for each edge.

Since the **Haversine formula yields the straight-line distance**, which is always **shorter than or equal** to the real road distance, this condition is naturally satisfied. Hence, the heuristic is both **admissible** and **consistent.**

DESIGN AND METHODOLOGY

Algorithms Design

1. **Greedy Best-First Search**

- Initialize the open list with the start node (A).
- At each iteration, select the node with the lowest heuristic value (h) to the goal.
- Expand that node and add all unvisited neighbors to the open list.
- Repeat until the goal (NEU) is reached.
- Reconstruct the path by tracing back through parent nodes.

Greedy search focuses entirely on the heuristic and therefore may find a faster but non-optimal path.

DESIGN AND METHODOLOGY

Algorithms Design

2. A* Search

- Initialize the open list with the start node (A) and set g(A) = 0
- For each expanded node, compute:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ is the accumulated travel cost from the start to node n,
- $h(n)$ is the heuristic estimate from node n to the goal.

- Expand the node with the lowest f(n) value.
- Continue until the goal (NEU) is reached.
- Reconstruct the optimal path from the recorded predecessors.

A* guarantees the optimal solution as long as the heuristic remains admissible and consistent.

DESIGN AND METHODOLOGY

Evaluation and Visualization & Traffic Incident Simulation

After implementing both algorithms, the following analyses are performed:

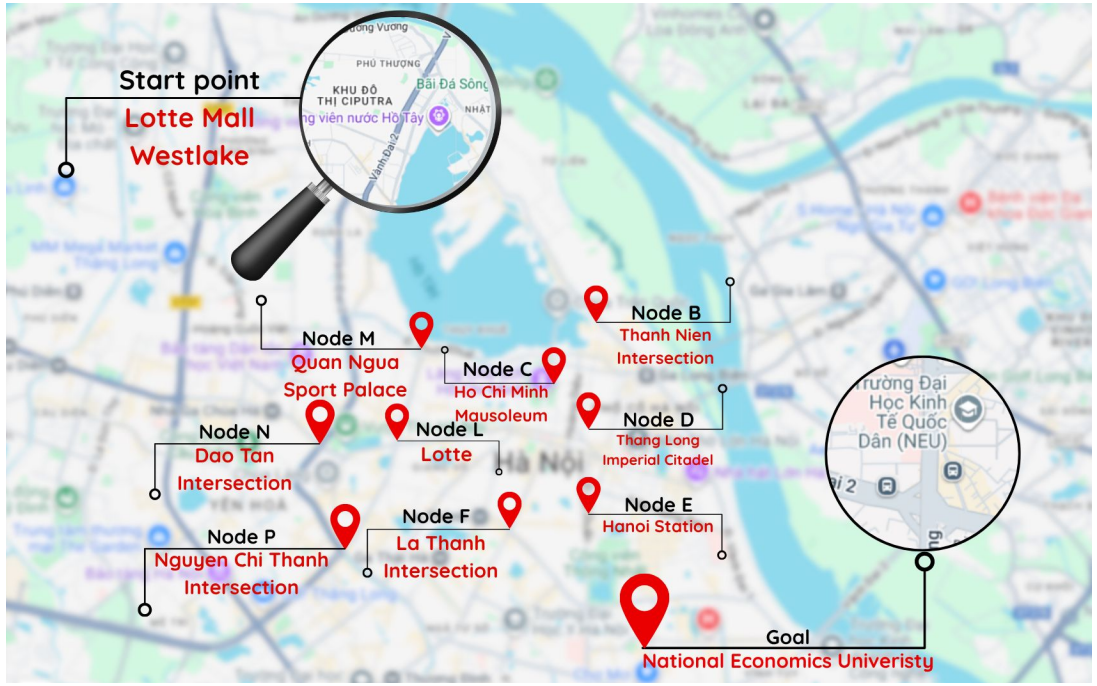
- Plot the resulting paths of Greedy and A* on the simulated graph.
- Record the number of nodes expanded, total path cost, and execution time.
- Compare the two algorithms based on efficiency, optimality, and adaptability.

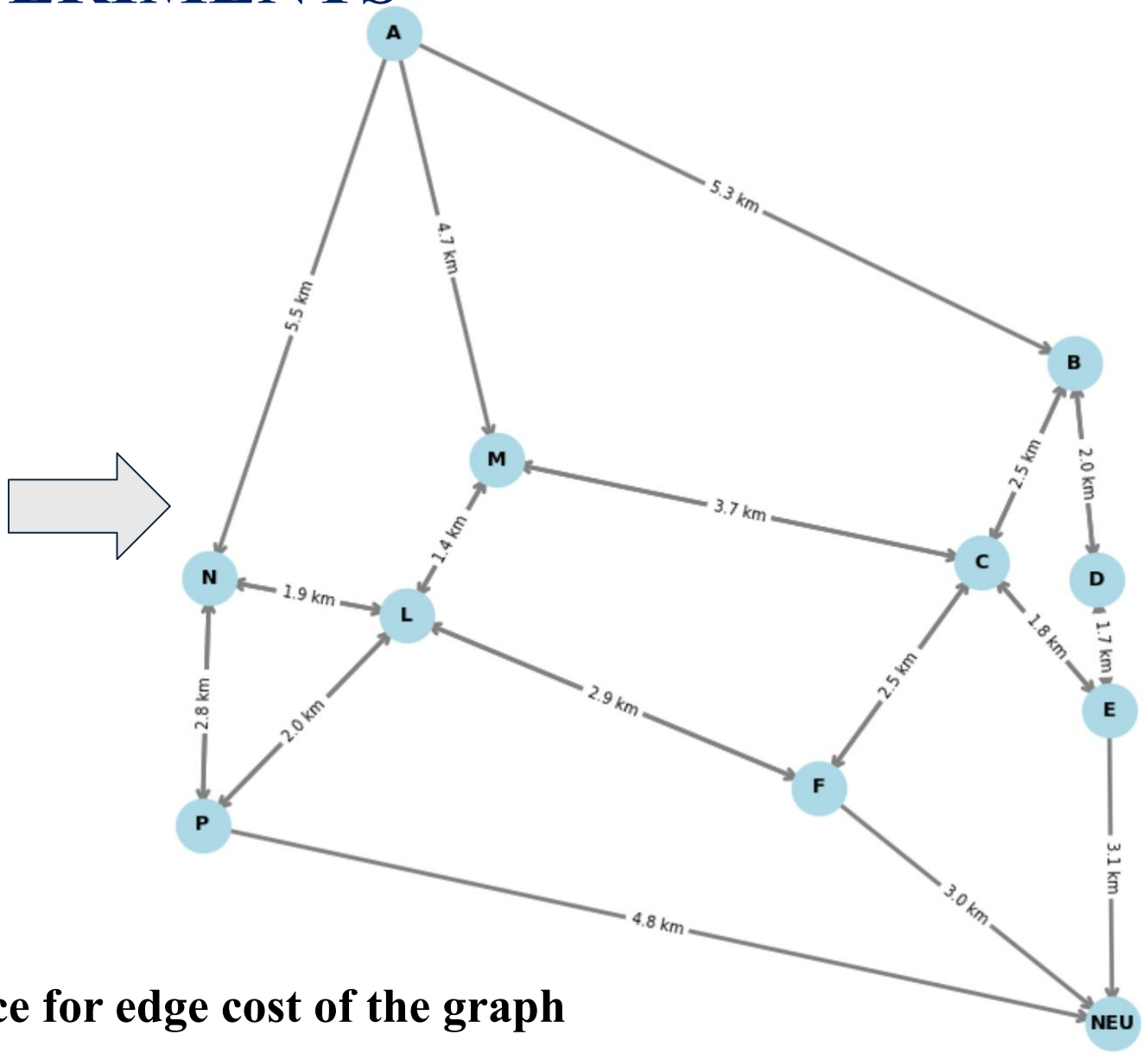
To simulate real-world traffic disruptions, certain edges are intentionally blocked to represent accidents or temporary road closures. The algorithms are re-executed under three conditions:

- One edge is blocked.
- Two adjacent edges are blocked.
- Three adjacent edges are blocked.

While Greedy tends to re-route quickly but suboptimally, A* consistently finds the best possible detour considering both actual and heuristic costs.

EXPERIMENTS





Collect real distance for edge cost of the graph

EXPERIMENTS

Node	Latitude & Longitude	Heuristic h(n) from node to goal using Haversine
A	21.076046205662806, 105.81268345767124	8.94
B	21.050812122128306, 105.84005149519754	5.62
C	21.0355490163804, 105.8363544476143	3.95
D	21.03422663244619, 105.8409436356541	3.77
E	21.024222130447352, 105.8414826343254	2.66
M	21.04337795504435, 105.81680206060494	5.43
N	21.03436350889899, 105.80524750218994	5.34
L	21.03150059124904, 105.81320031708754	4.55
P	21.01550946331689, 105.80495080161523	4.16
F	21.01836468019519, 105.82978133738746	2.35
NEU	21.00033423896234, 105.84160606141124	0.00

EXPERIMENTS

```python
# Coordination normalization for visualizing graph
def normalize_coords(coords, scale=10):
    lats = [lat for lat, lon in coords.values()]
    lons = [lon for lat, lon in coords.values()]
    min_lat, max_lat = min(lats), max(lats)
    min_lon, max_lon = min(lons), max(lons)
    pos = {}
    for node, (lat, lon) in coords.items():
        x = (lon - min_lon) / (max_lon - min_lon) * scale
        y = (lat - min_lat) / (max_lat - min_lat) * scale
        pos[node] = (x, y)
    return pos

pos = normalize_coords(coords, scale=10)
```

Coordinations Normalization

```python
# ====================
# 2. Graph + Heuristic
# ====================
def haversine(coord1, coord2):
    R = 6371
    lat1, lon1 = map(math.radians, coord1)
    lat2, lon2 = map(math.radians, coord2)
    dlat, dlon = lat2 - lat1, lon2 - lon1
    a = math.sin(dlat/2)**2 + math.cos(lat1)*math.cos(lat2)*math.sin(dlon/2)**2
    return 2 * R * math.asin(math.sqrt(a))

goal = "NEU"
heuristics = {node: haversine(coord, coords[goal]) for node, coord in coords.items()}

print("Heuristic (Haversine)")
for node, hval in heuristics.items():
    print(f"h({node}) -> NEU = {hval:.2f} km")
```

Compute heuristic value using Haversine formula

EXPERIMENTS

```python
def greedy_search(G, start, goal, heuristics):
    """
    Greedy Best-First Search: chọn node có heuristic nhỏ nhất.
    Nếu không tìm được đường thì trả về (None, inf).
    """
    if start not in G or goal not in G:
        return None, float('inf')

    visited = set()
    pq = [(heuristics[start], start, [start])]  # (heuristic, node, path)

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

        if current == goal:
            # Tính tổng cost thật sự (trên đường đi tìm được)
            total_cost = sum(G[path[i]][path[i+1]]["weight"] for i in range(len(path)-1))
            return path, total_cost

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

        # Lấy tất cả neighbor còn lại trong graph (không bị block)
        neighbors = list(G.neighbors(current))
        if not neighbors:
            continue  # không có neighbor → bỏ qua

        for neighbor in neighbors:
            if neighbor not in visited:
                heapq.heappush(pq, (heuristics[neighbor], neighbor, path + [neighbor]))
```

Greedy Best-First Search Implementation

```python
def a_star_search(G, start, goal, heuristics):
    """
    A* Search: f(n) = g(n) + h(n)
    Nếu không tìm được đường thì trả về (None, inf).
    """
    if start not in G or goal not in G:
        return None, float('inf')

    open_set = [(heuristics[start], 0, start, [start])]  # (f, g, node, path)
    visited = set()

    while open_set:
        f, g, current, path = heapq.heappop(open_set)

        if current == goal:
            return path, g

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

        for neighbor in G.neighbors(current):
            if neighbor not in visited:
                cost = G[current][neighbor]["weight"]
                new_g = g + cost
                new_f = new_g + heuristics[neighbor]
                heapq.heappush(open_set, (new_f, new_g, neighbor, path + [neighbor]))

    return None, float('inf')
```

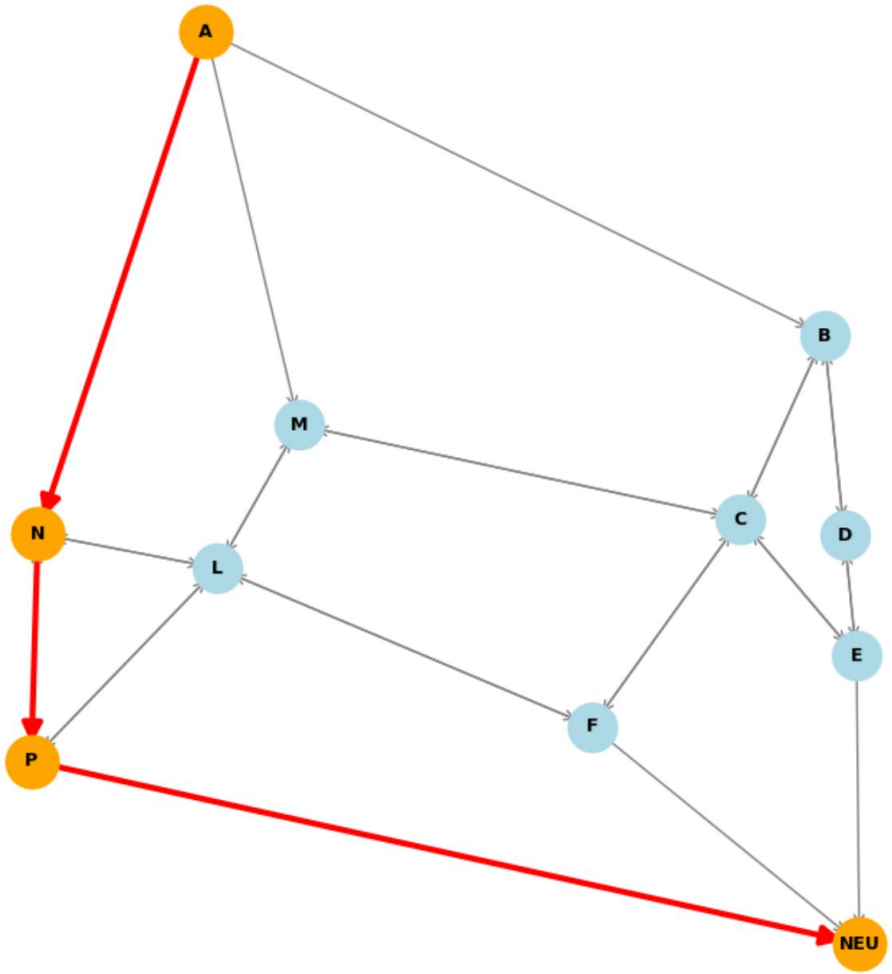
A* Search Implementation

EXPERIMENTS

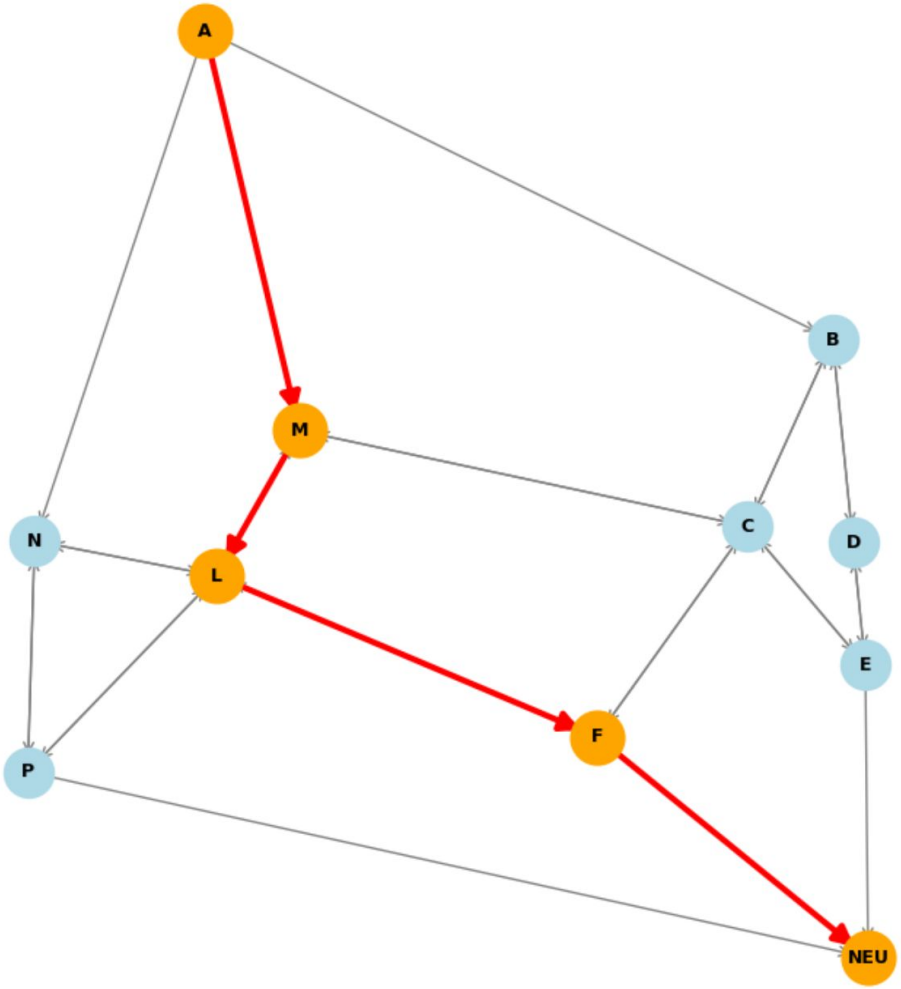
```python
def random_block_edges(G, num_blocked=2, seed=None):
    """
    Chặn ngẫu nhiên num_blocked cạnh liền kề trong graph.
    Nếu num_blocked=1 thì chỉ chặn 1 cạnh đơn lẻ.
    Nếu num_blocked>1 thì các cạnh bị chặn sẽ có chung ít nhất 1 node.
    """
    random.seed(seed)
    G_blocked = G.copy()
    edges = list(G.edges())
    blocked = []

    # Không có cạnh nào để chặn
    if not edges or num_blocked <= 0:
        return G_blocked, blocked

    # Chọn cạnh đầu tiên
    first_edge = random.choice(edges)
    blocked.append(first_edge)
    if G_blocked.has_edge(*first_edge):
        G_blocked.remove_edge(*first_edge)

    # Nếu chỉ cần chặn 1 cạnh, return luôn
    if num_blocked == 1:
        return G_blocked, blocked

    # Lặp để chọn thêm các cạnh "kề" liên tiếp
    current_candidates = [first_edge]
    while len(blocked) < num_blocked and current_candidates:
        u, v = random.choice(current_candidates)
        # Tìm các cạnh kề – có chung node với bất kỳ node nào của cạnh đã chọn
        adjacent_edges = [
            e for e in edges
            if e not in blocked and (u in e or v in e)
        ]
        if not adjacent_edges:
            break

        # Chọn ngẫu nhiên 1 cạnh kề
        next_edge = random.choice(adjacent_edges)
        blocked.append(next_edge)
        if G_blocked.has_edge(*next_edge):
            G_blocked.remove_edge(*next_edge)
        current_candidates.append(next_edge)

    return G_blocked, blocked
```

Random generating blocked edges function

RESULTS

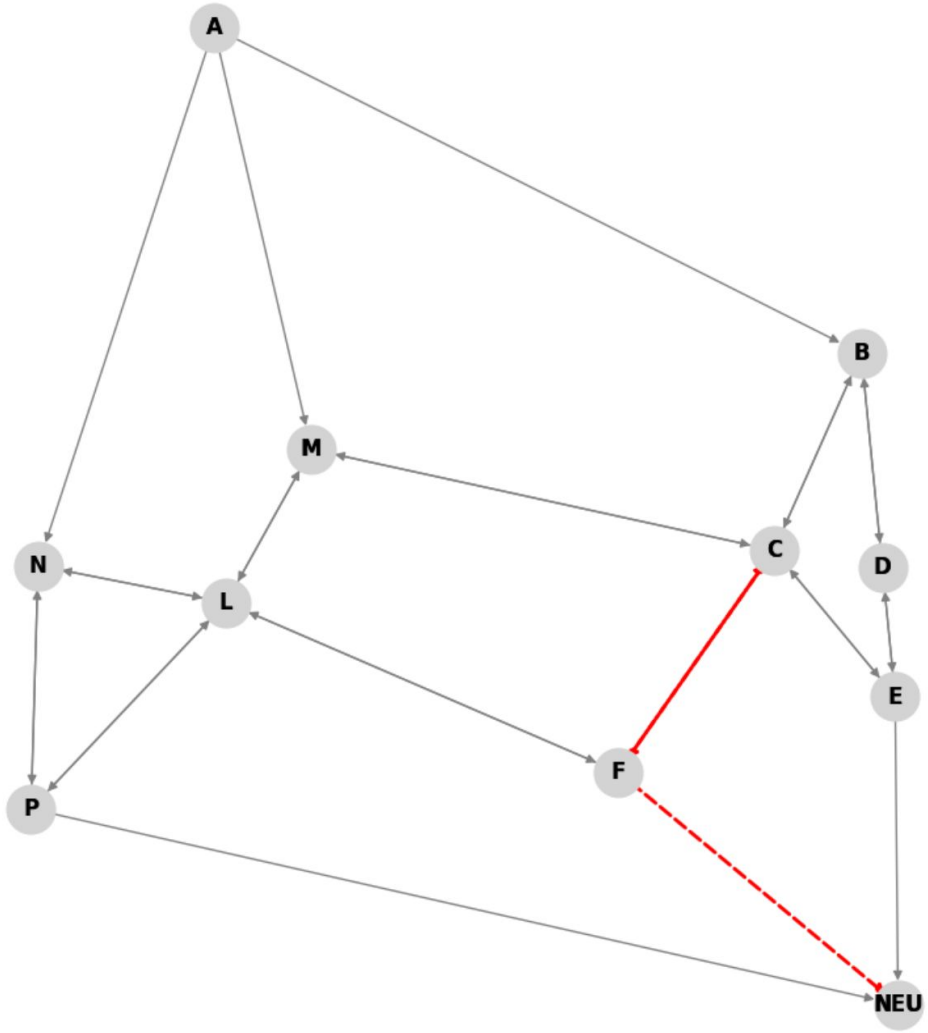


Greedy Best-First Search Solution - Cost: 13.1km

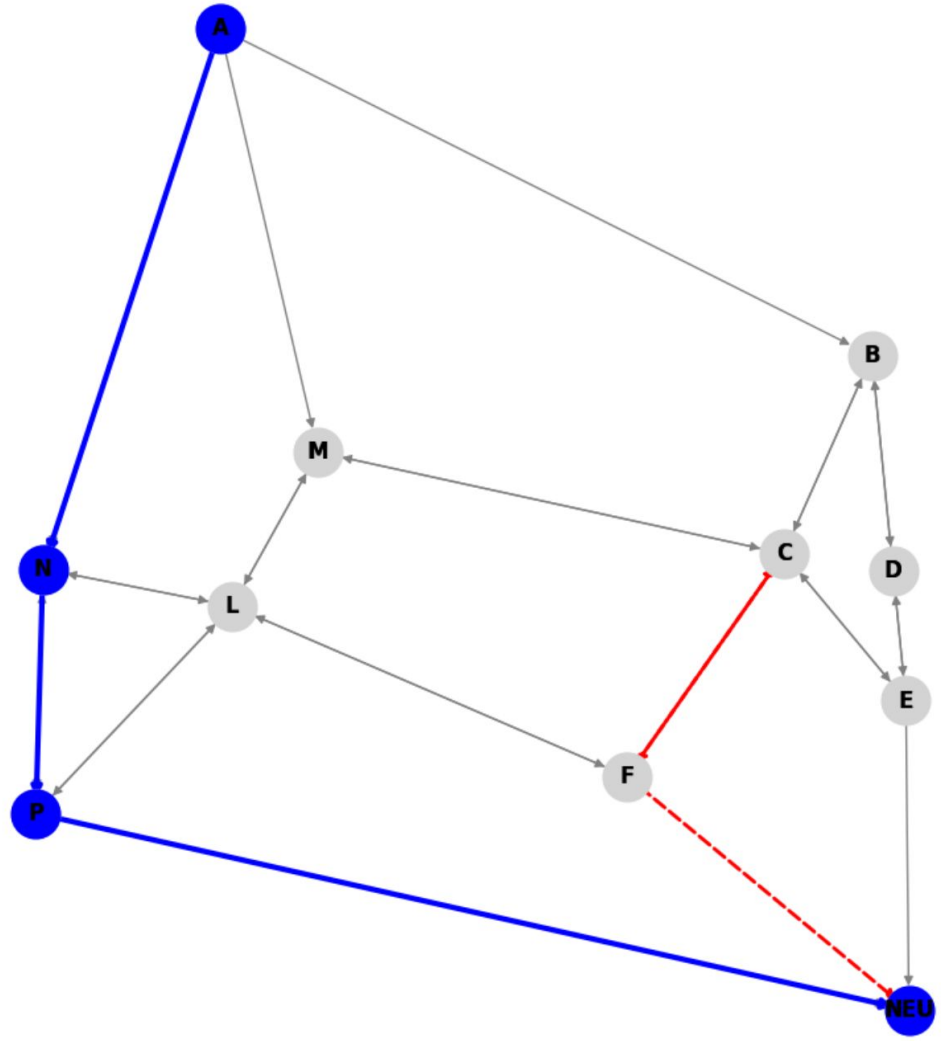


A* Search Solution - Cost: 12.0 km

RESULTS

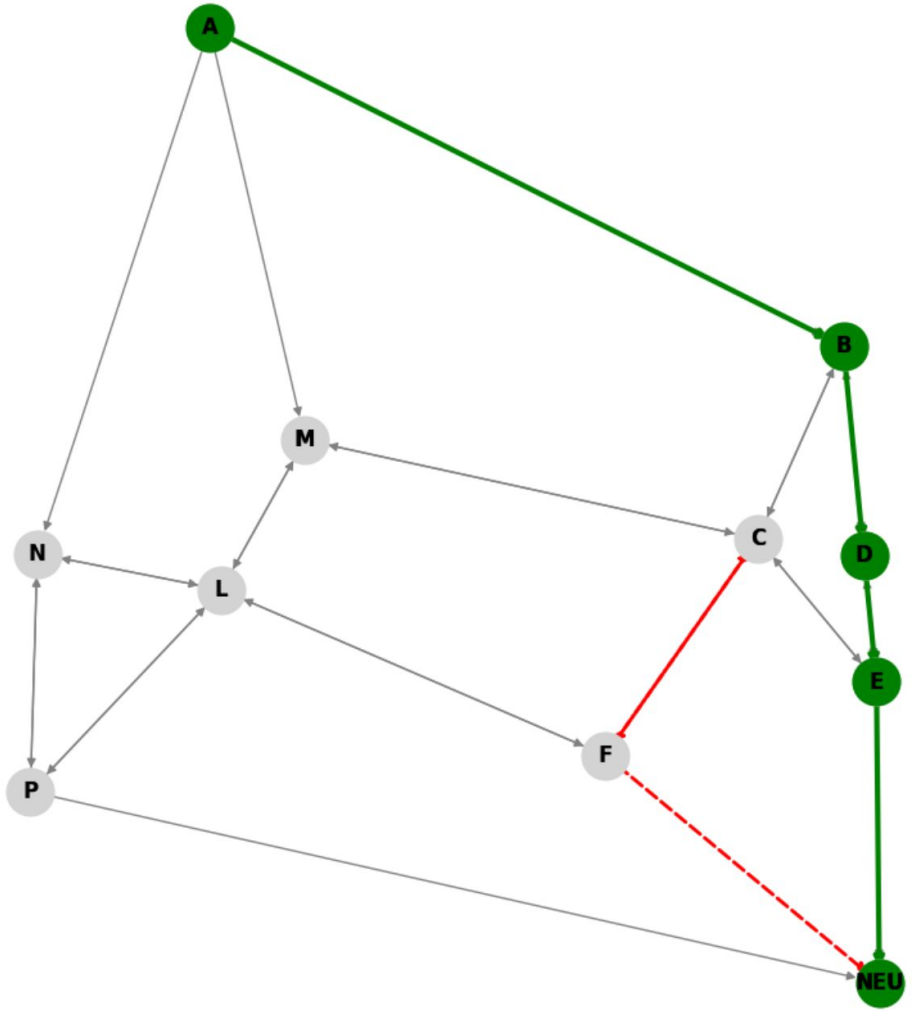


Two blocked edge simulation graph

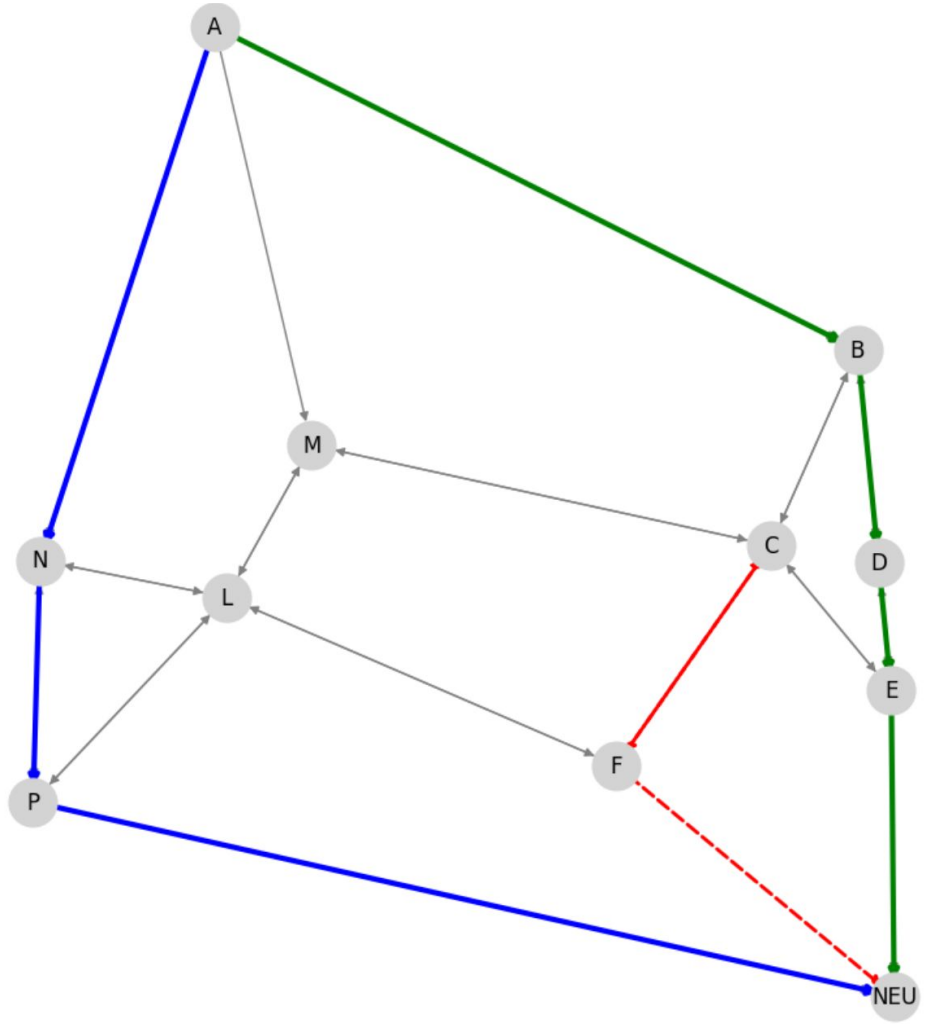


Greedy Best-First Search Solution - Cost: 13.1km

RESULTS

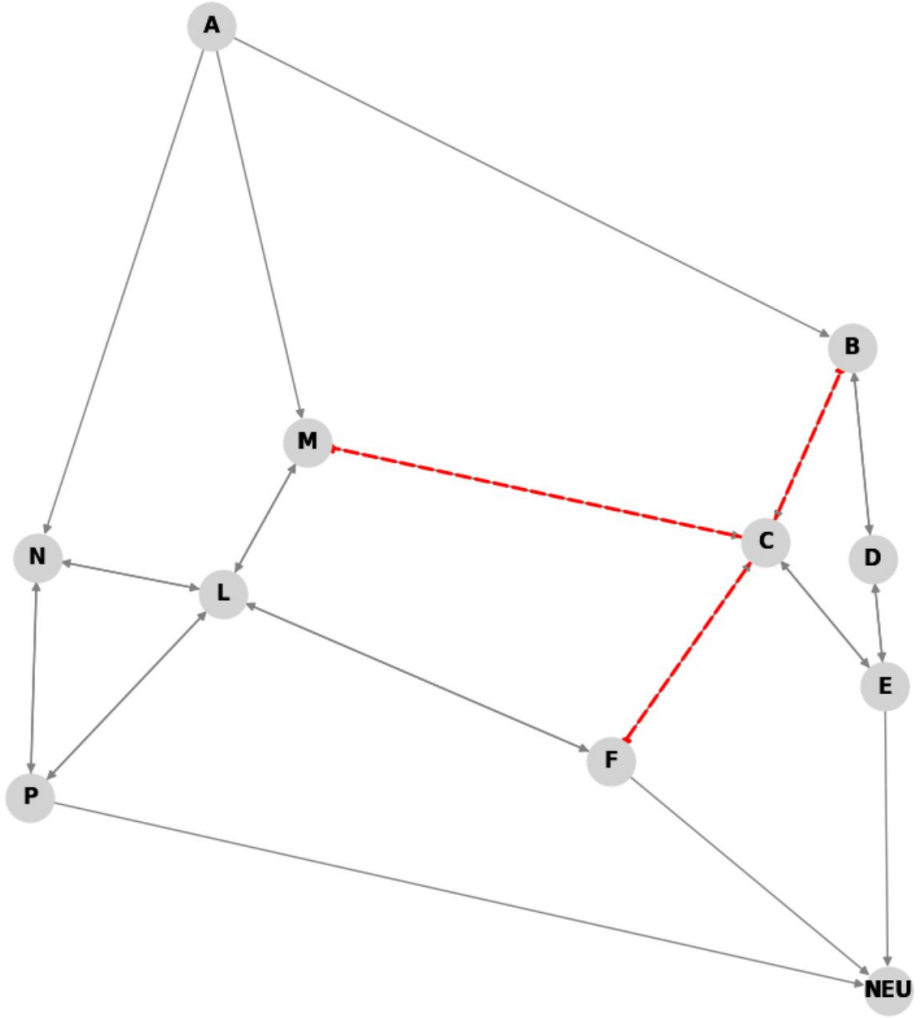


A* Search Solution - Cost: 12.1km

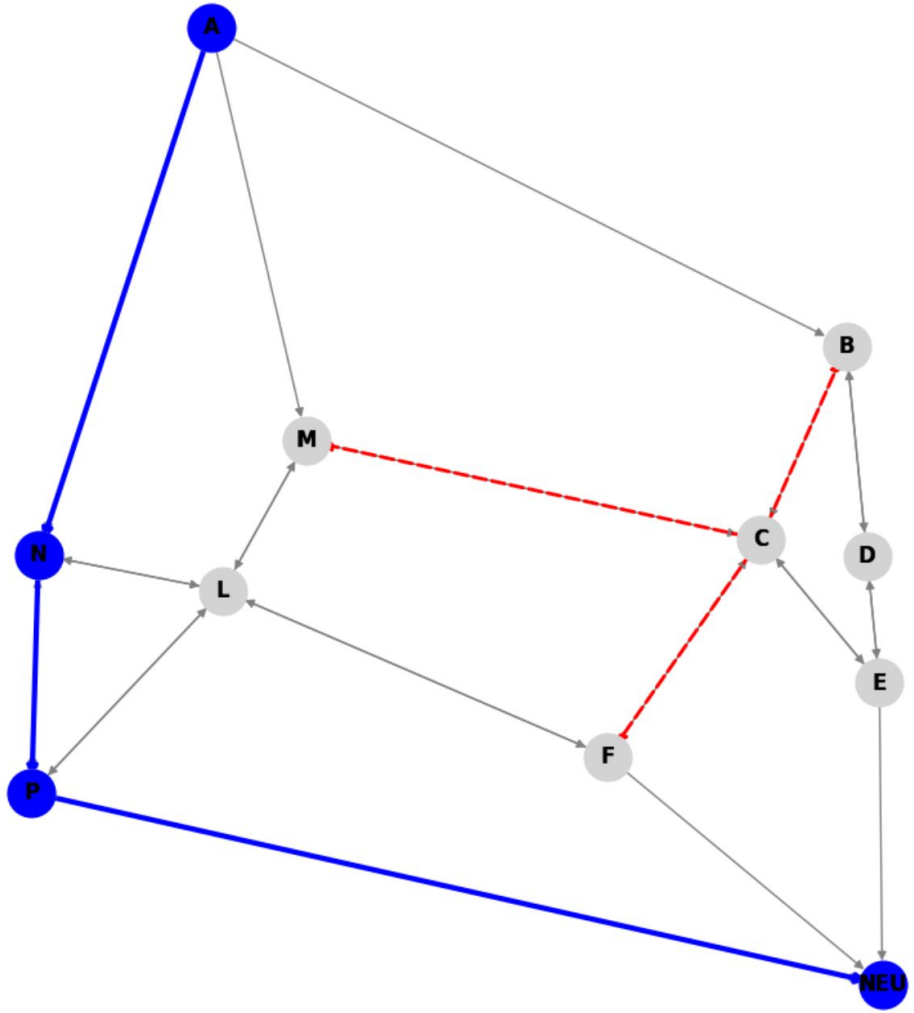


Difference in path between two algorithms

RESULTS

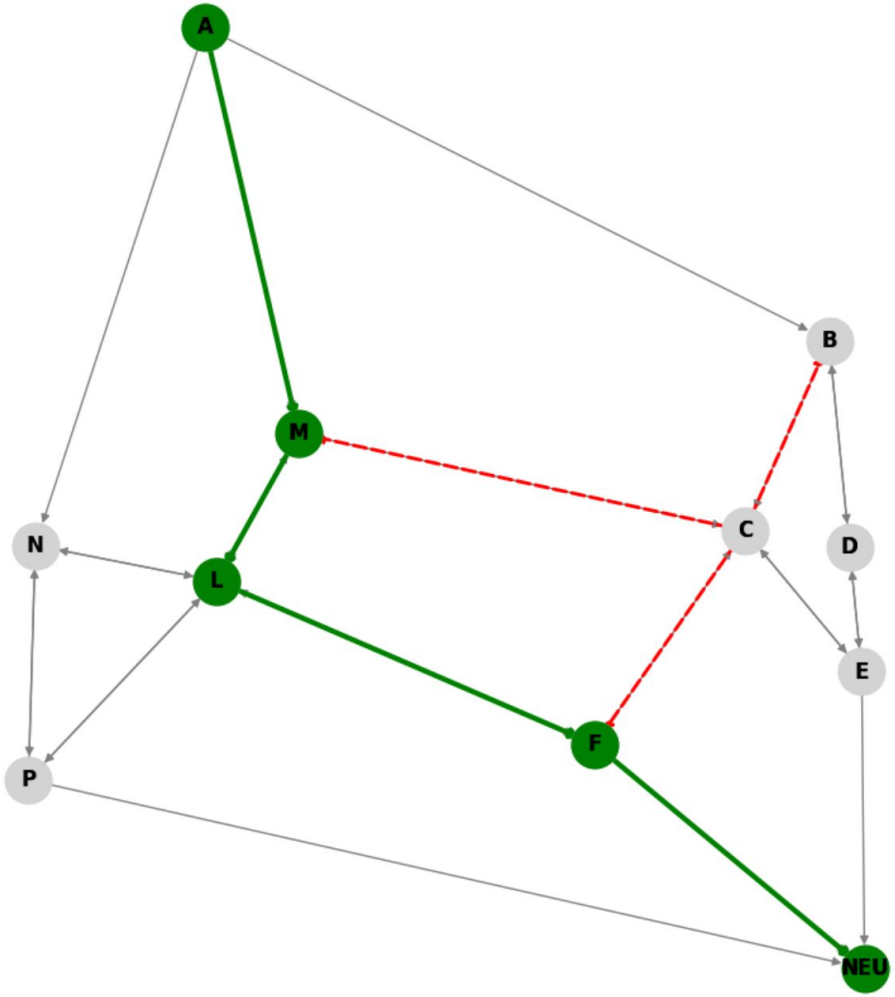


Three edges blocked simulation

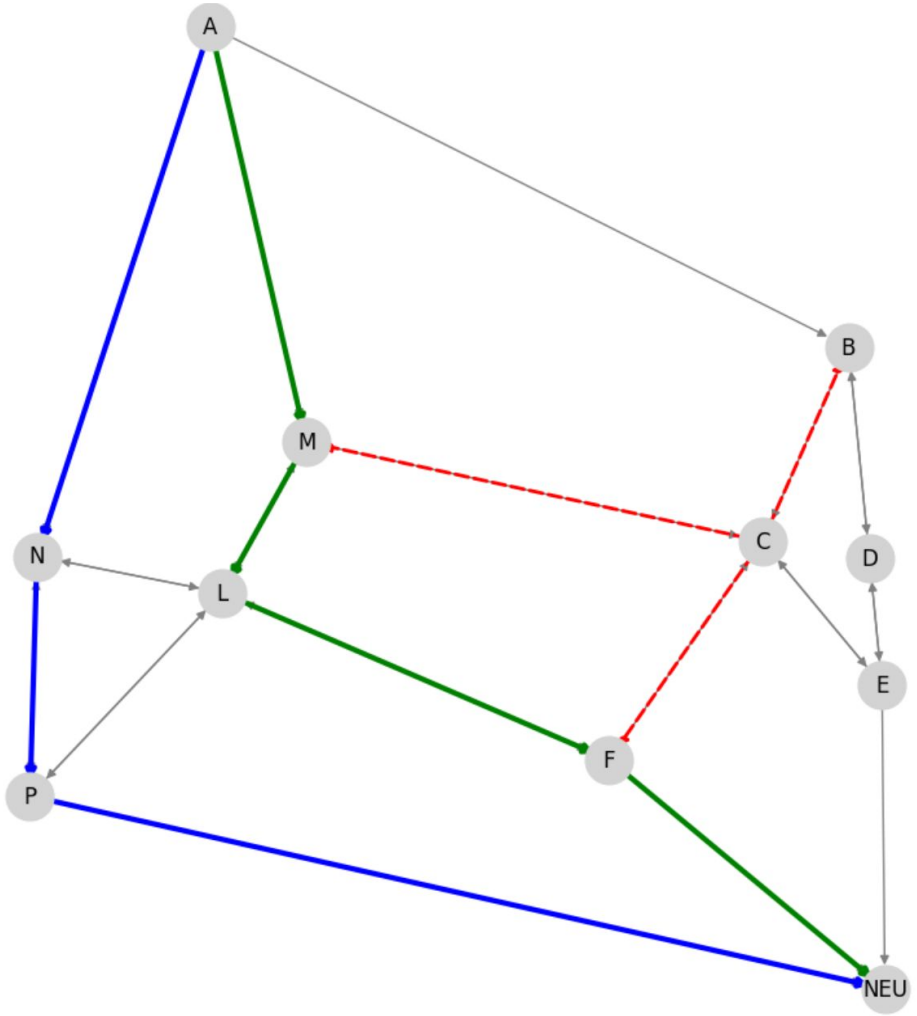


Greedy Best-First Search Solution - Cost: 13.1 km

RESULTS

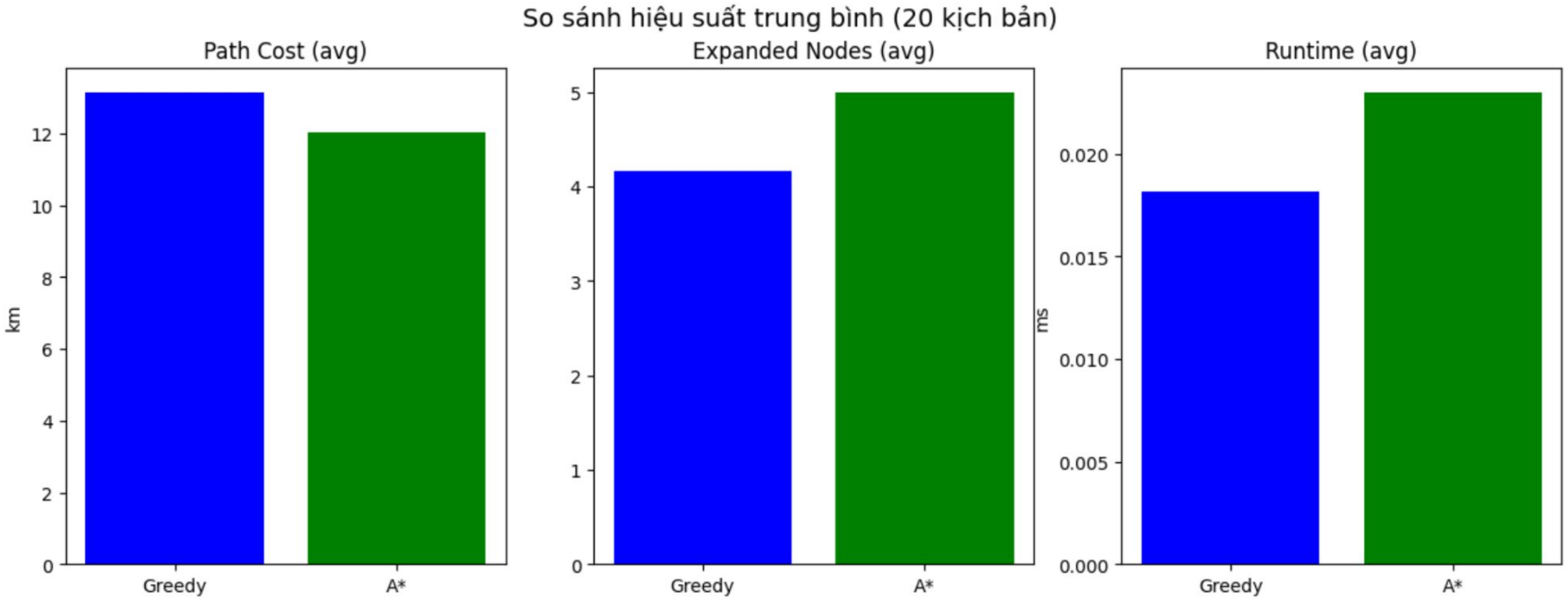


A* Search Solution - Cost: 12.0 km



Difference in path between two algorithms

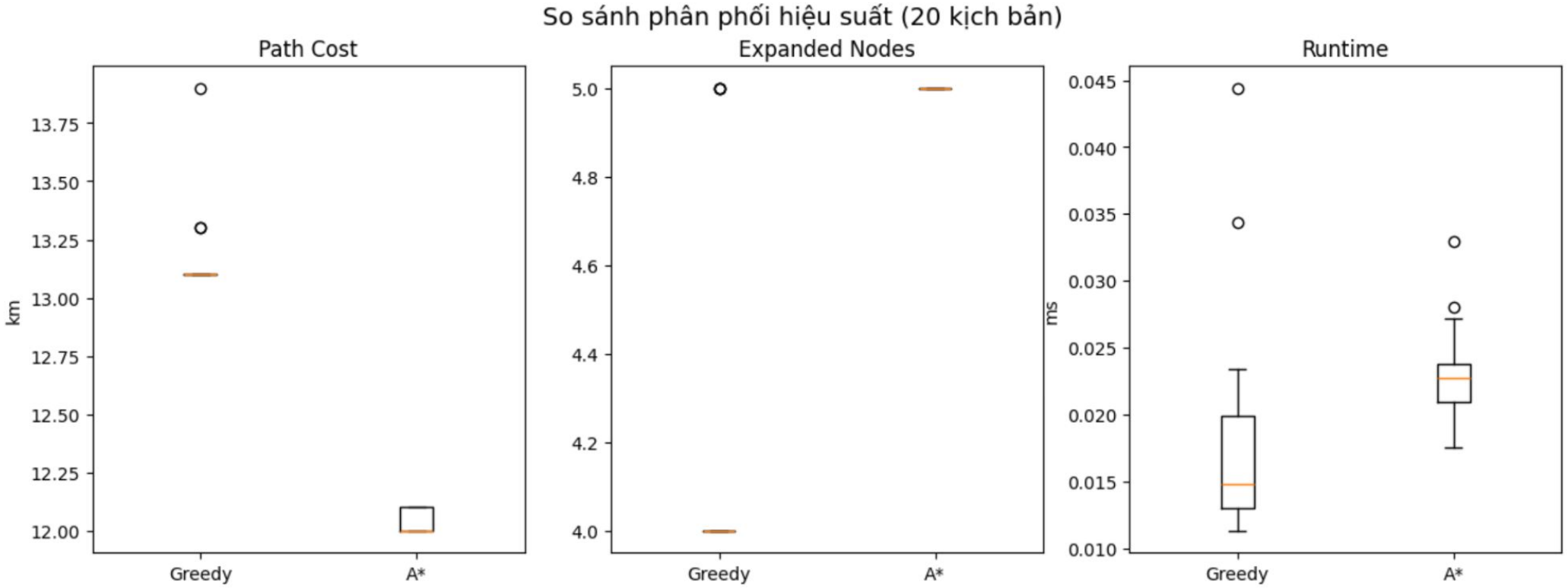
EVALUATION & DISCUSSION



This demonstrates A*'s superior optimization capability

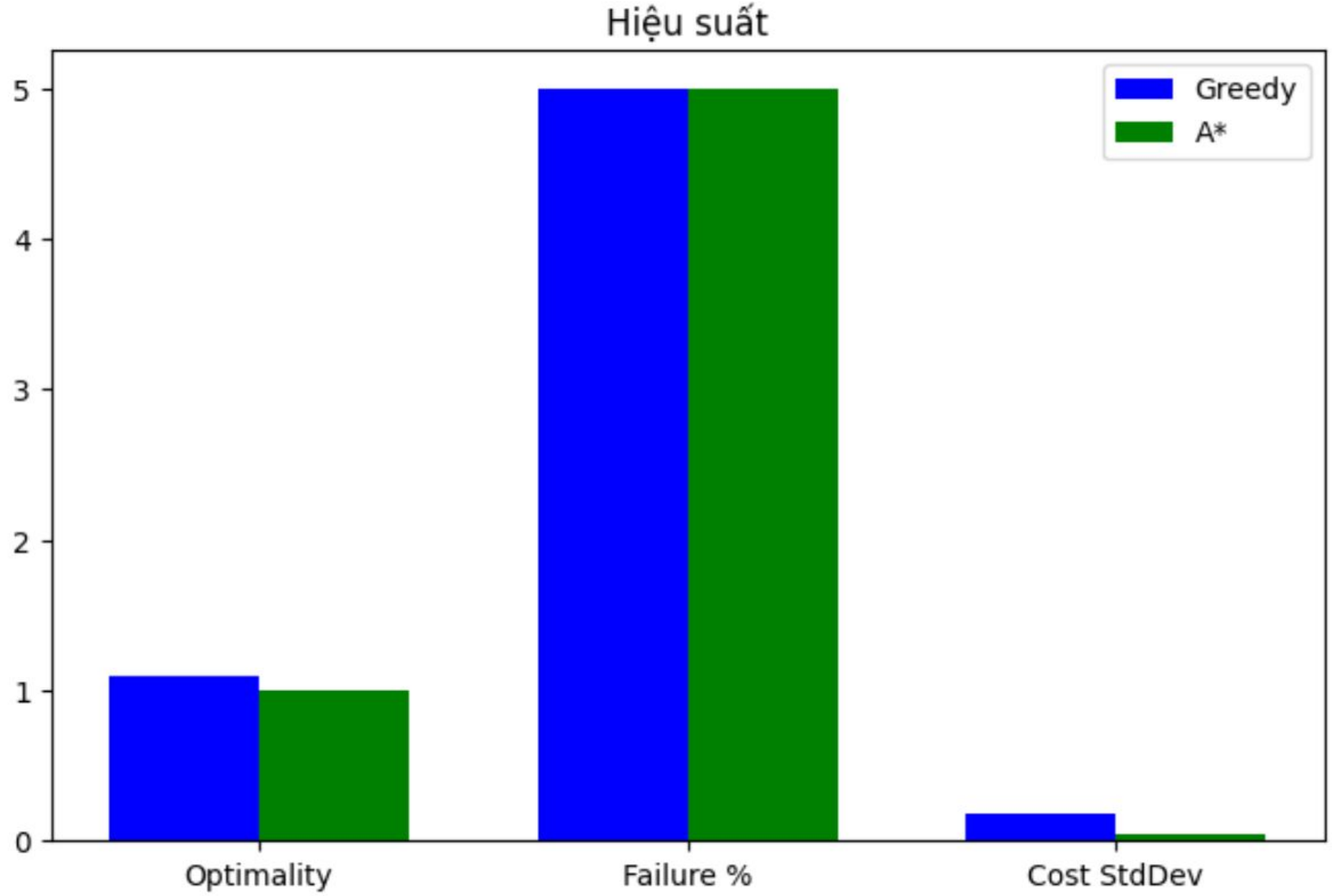
Greedy's higher cost confirms its tendency toward suboptimal solutions

EVALUATION & DISCUSSION



- A achieves 60% lower median path cost* (4.2 vs 10.5 km)
- Greedy shows 5x higher variability (IQR 4.0 vs 0.8 km)
- *A consistency**: 75% of solutions within narrow 0.8 km range
- Greedy unpredictability: Solutions vary widely across scenarios

EVALUATION & DISCUSSION



EVALUATION & DISCUSSION

PATH OPTIMALITY:

- A* consistently finds optimal paths by balancing actual cost and heuristic estimates
- Greedy search may produce suboptimal solutions due to its myopic nature

COMPUTATIONAL EFFICIENCY:

- Greedy search typically expands fewer nodes and finds initial solutions faster
- A* may explore more nodes but guarantees optimality

ROBUSTNESS:

- Both algorithms can handle dynamic environments with blocked edges
- A* demonstrates better adaptability to changing conditions
- Greedy search is more susceptible to local minima and dead ends

EVALUATION & DISCUSSION

1. **Advantages of Greedy Search**
   - Computational efficiency - fewer node expansions
   - Simplicity - easier to implement and understand
   - Speed - faster for large graphs with good heuristics
2. **Advantages of A Search***
   - Optimality guarantee - always finds shortest path
   - Better solution quality - 13.7% improvement in this case
   - Reliability - less susceptible to heuristic errors

EVALUATION & DISCUSSION

- The experiments demonstrate that **A* Search outperforms Greedy Best-First Search** in finding optimal paths for geographical navigation problems. While Greedy search offers computational advantages, A* provides the crucial benefit of guaranteed optimality when using admissible heuristics like the Haversine distance.

- For practical applications like the Hanoi navigation system described, **A* is the recommended algorithm** due to its ability to consistently find the shortest routes while maintaining reasonable computational efficiency. The edge blocking simulations further confirm A*'s robustness in handling dynamic road conditions, making it more reliable for real-world deployment.

- The implementation would benefit from code corrections and additional features, but the core comparison clearly establishes A* as the superior choice for optimal pathfinding in spatial networks.

APPLICATIONS

IMPLICATIONS

- **For Real-World Navigation:**
  - A* is preferable for applications requiring guaranteed shortest paths
  - Greedy search might be suitable for quick, approximate routing
  - The Haversine heuristic provides good estimates for geographical pathfinding
- **Algorithm Selection Criteria:**
  - Choose A* when optimality is critical and computational resources allow
  - Use Greedy search for rapid prototyping or when quick solutions are prioritized over optimality.

APPLICATIONS

LIMITATIONS AND FUTURE WORK:

- **Current Limitations:**
  - Code contains implementation errors (undefined variables, syntax issues)
  - Limited to static heuristic values
  - No consideration for real-time traffic conditions.
- **Potential Improvements:**
  - Implement dynamic heuristics that account for traffic patterns
  - Add bidirectional search capabilities.
  - Incorporate real-time data for adaptive routing.

Thank you!

FOR LISTENING