*Discussion

* Strengths of the Implemented Solution

The hybrid algorithm combining bidirectional A* with ALT preprocessing demonstrates significant improvements o

Efficiency in Large-Scale Networks: The ALT preprocessing reduced query times by 50–70% compared to Dijkstra'

Deterministic Behavior: Neighbor sorting in `algorithms.py` ensured consistent execution:

[language=Python] sorted(graph.neighbors(...), key=lambda x: graph[...][x][0]['length'])

Effective Visualization: The Animator class provided critical insights through:

Blue edges: Explored paths

Green path: Optimal route

Side-by-side algorithm comparisons

* Limitations and Challenges

Static Preprocessing: `ALTPreprocessor` computes landmarks only once during initialization.

Hardcoded Cost Function: Exclusive use of edge length:

[language=Python] min(data['length'] for data in graph[u][v].values())

Absence of Contraction Hierarchies: Limits scalability for continental-scale networks.

Suboptimal Landmark Selection: Current greedy selection in `select_landmarks()`.

* Future Improvements

*Contraction Hierarchies Integration [language=Python] class CHPreprocessor: def $_{init_{(self,graph):self.node_order=self.com}}$

def $_compute_node_order(self) : return sorted(graph.nodes, key = lambda n : self._edge_difference(n))$

* Dynamic Graph Support [language=Python] class DynamicALTPreprocessor(ALTPreprocessor): def update$_edge($

*Customizable Cost Functions [language=Python] def bidirectional$_alt_query(..., cost_function =' length') : edge_lengt$

*Enhanced Landmark Selection [language=Python] from networkx.algorithms.centrality import betweenness$_centrat$

def $_select_landmarks(self) : centrality = betweenness_centrality(self.graph, weight =' length')self.landmarks = s$