Table of Contents

I. Pathfinding and Search Formulation

Goal Formulation

Goal State: The user-specified goal city

Goal test: When expanding a node, check $current\_city\ ==\ goal\_city$. If true, stop.

Problem Formulation

Initial State: The start city given by the user.

Actions: From a city, you can travel along any adjacent highway edge to a neighboring city.

Transition model: $Result(state,\ action)$ = move from the current city to the adjacent cities via the chosen road segment.

Path cost:

- Unweighted (BFS/DFS): each action costs 1 hop.
- Weighted: action cost equals edge miles; path cost is the sum of miles.

II. Frontier type per algorithm; explored set; node vs state

BFS	**DFS**	**UCS (Dijkstra)**	**Greedy**	**A***
Frontier: FIFO queue	Frontier: stack(LIFO)	Frontier: min-priority queue	Frontier: min-priority queue keyed by h(n)	Frontier: min-priority queue
Explored Sets: keeps cities already removed from the queue	Explored Sets: keeps cities already removed from the queue	Explored Sets: $closed\ set\ +\ best\_g[s$ (lowest known cost)	Explored Set: cities popped from the queue	Explored Set: $closed\ set\ +\ best$
Node vs State: Node = (city, parent, depth) State = city name	Node vs State: Node = (city, parent, depth) State = city	Node vs State: Node = (city, parent, g) State = city	Node vs State: Node = (city, parent, h) State = city	Node vs State: Node = (city, parent, f) State = city

III. **Graph representation choice** of data structure and justification

Choice: Adjacency list implemented as a dictionary of lists.

Justification:

- Efficient neighbor expansion:
- Memory efficient for sparse graphs: Highway networks are sparse (each city connects to only a few others). Adjacency lists only store existing edges; no wasted space like in an adjacency matrix.
- Readable and flexible: lists store both neighbor name and edge weight, so you can use the same structure for unweighted (hops) and weighted (miles) algorithms
- Scales well: Space = O(V + E)

IV. Heuristic definition and admissibility argument

The landmark heuristic precomputes the distance from all nodes to the landmark node. It uses these values to compute an estimated difference between two nodes by finding the difference between them. The precomputation means that the A* search algorithm saves time by just looking up the stored values

V. Design decisions

We have input validation to make sure the input is within the graph and will return a message informing the user if there is no destination.

The DFS algorithm uses a stack data structure to explore as far down a path as possible before backtracking. It uses a visited list to prevent revisiting nodes and causing infinite loops. It then traverses the path back in reverse to print out the nodes that it had visited in order.

Dijkstra's algorithm was designed with a helper function to traverse the nodes it had visited in reverse order and print the path, rather than being built into the function. This allowed it to be used in other algorithms, such as the greedy best-first and A* algorithms.

For the implementation of A* and greedy best-first search, Dijkstra's algorithm was used with an added heuristic.

VI. Analysis

- **Algorithm comparison**

  BFS and Dijkstra's all returned the same best path cost when the same inputs were entered.

  DFS (supplemented with a guardrail that prevents looping) always found a path, but it was not guaranteed to find the shortest path.

- **Time and space considerations per algorithm**

  Where E = edges and V = vertices

  - **BFS:**

    Time complexity: O(V + E)

    Space complexity: O(V)

  - **DFS:**

    Time complexity: O(V + E)

    Space complexity: O(V)

  - **Dijsktras:**

    Time complexity: O(E + VlogV)

    Space complexity: O(V + E)

  - **GBFS:**

    Time complexity: O(ElogV)

    Space complexity: O(V)

  - **A***

    Time complexity: O(E + VlogV)

    Space complexity: O(V)