problem: design self driving strategy with algorithms

1. Brainstorming:
assume:
optimal tour route from pickup to destination
does this just mean a to b (well technically a to b to c or even d because the car has a starting location and may have to charge, but ?
are multiple stops going to be a regular expectiation? I wouldn't expect that for a robotaxi.
so lets just assume a to b
Can elevation just be factored into 1-way traffic since our nodes likely should be bi-directional for traffic anyways?

maximize profit by reducing car travel by reducing recharge frequency
    => use as little electricity as possible and charge at lowest energy assuming some kind of next trip length?
    =>traffic causes electricity costs to increase, but we are commited to the optimal tour by distance or by electricity?

minimize customer cost with dynamic pricing?
    -demand based => more demand => lower prices (traditional taxis kind of work this way if you can find people to share a ride with you)

vehical availability impacts route selection?????   fewer cars means you have less routes?
I’m assuming I’m just responsible for routes and roads, not any of the geocoding from gps coordinates to street address.

if we make a graph:
    nodes:
    charging stations (charge rate included. that can differ),
    traffic stops (intersections/exit holdups),
    customer locations+destinations, (could be like heatmapped or just assumed to general shared ones?)
    maybe proper routing stuff like highway on-ramps and exists, general stuff that traffic flow is designed around
    could probably come up with something else to turn into nodes to, but basically anything we might want to put edges between, especially if the node itself could have some sort of cost

edges:
    distance (default)
    add onto the edge value with traffic level
    subtract from it by demand level
    estimate electricity cost from elevation change scalar, speed limit, traffic scalar, stop/go-ishness
    required charge from taking route
    honestly the nodes are more focused, but we could add TONS of shit to make our edges more optimal

potential algorithms of use:
djikistra?
BFS?
A*
graph partitioning (we ignore all but a local subset of our graph to focus on more calculations between a certain set of nodes/edges)

I'm actually looking at some interesting open source projects+papers for similar ideas:
Further Reading (I’ve glanced through these some):
https://github.com/Telenav/open-source-spec/blob/master/routing_basic/doc/crp.md
https://github.com/Telenav/open-source-spec/blob/master/routing_basic/doc/graph_partition.md
https://project-osrm.org/
https://valhalla.github.io/valhalla/
https://github.com/valhalla/valhalla
https://mapsplatform.google.com/maps-products/routes/
The google maps api actually has some neat ideas in their advertising like optimizing for different types of vehicles.

```
class Node:
  - id
  - location (coordinates)
  - is_charging_station (boolean)
  - additional_properties (e.g., amenities)

class Edge:
  - source_node
  - target_node
  - distance
  -elevation, but elevation could be included as part of the traffic scalar or something too?
  - traffic_level (dynamic)
  -traffic_level (predicted or regular)
  - demand_level (dynamic)
  -demand_level (predicted or regular)
  - electricity_consumption_rate
  - price_factor
```

for some of these factors, getting the info and quanitifying them is their entire own new function and algorithm that I think could be left out as they'd be similar projects in terms of scope each on their own.

cost functions (well, a few of them):

electricity_used = base_electricity_rate * distance * traffic_scalar *

price = base_price_rate * distance / (demand_scalar)

profit = customer_price - electricity costs - operation_upkeep_costs

considerations:
traffic and demand change constantly
I don't know a lot about how stop lights work and they're main traffic gates in urban areas

optimal tour might always not be most battery or time efficient, but let's just assume it ishness

even with just a 2 digit number of nodes, this could get out of hand to an ABSURD degree

battery technology is kind of behind where it should be and is closed source....

would a computer even be able to or want to make these calculations on the spot quickly? the customer is waiting, we'd probably want most of this pre-done and cached somewhere

urban and rural optimizations might differ

can we integrate both real time traffic and predicted pattern traffic?

how do we quantify each step into our edges
how do we go from location with a phone's GPS to a starting node?
each step is a thing we express with words we need to turn into numbers and back.
Valhalla for instance, uses data endpoints from OpenStreetMap to get its inputs

A* is very intensive so we would need very small numbers of nodes and very large numbers of partitions which would still result in large numbers of nodes in larger graphs where each parititon is a node.
Apparently it is the standard though despite there beting better ways to do more node hits in TSP methods.

REMEMBER MAIN INSTRUCTION RULES:
COMPANY PROFIT (MAXIMIZE)
CUSTOMER COST (MINIMIZE)
TOUR COST (MINIMIZE in terms of fuel/time/distance)

2. brute force approach:
enumerate all possible paths from start node to end node,
pick the shortest one using above factors like distance, traffic, high demand lowering the price (I still don't really get this, but I don't make the rules), increasing profits by lowering charging spending
select the shortest path.

side loop (this is so small in comparison to the rest of this it barely matters):
if it's a path that exceeds remaining fuel, charge up first
if it's a path that exceeds total fuel, charge up partway.

brute force pseudocode:
```
function findOptimalRoute(graph, pickup, destination, batteryCapacity, currentBatteryLevel):
    allPaths = []
    profitByPath = {}
    customerCostByPath = {}

    // Step 1: Generate all possible paths
    function generateAllPaths(currentNode, destinationNode, currentPath, visitedNodes):
        if currentNode == destinationNode:
```

```
        allPaths.append(copy(currentPath))
        return

    visitedNodes.add(currentNode)

    for each neighbor in graph.getNeighbors(currentNode):
        if neighbor not in visitedNodes:
            currentPath.append(neighbor)
            generateAllPaths(neighbor, destinationNode, currentPath, visitedNodes)
            currentPath.pop()  // Backtrack

    visitedNodes.remove(currentNode)

// call the recursive function to generate all paths
generateAllPaths(pickup, destination, [pickup], set())

// Step 2: Evaluate each path
for path in allPaths:
    remainingBattery = currentBatteryLevel
    totalElectricityUsed = 0
    totalCustomerCost = 0
    isPathFeasible = true

    for i from 0 to path.length - 2:
        currentNode = path[i]
        nextNode = path[i+1]
        edge = graph.getEdge(currentNode, nextNode)

        // calculate electricity consumption based on traffic
        electricityForSegment = edge.distance * (1 + edge.trafficLevel * TRAFFIC_FACTOR)

        //check if battery is sufficient
        if remainingBattery < electricityForSegment:
            // Need to recharge???
            if currentNode.hasChargingStation:
                // Recharging cost (time and money)
                rechargeCost = calculateRechargeCost(remainingBattery, batteryCapacity)
                totalElectricityUsed += rechargeCost
                remainingBattery = batteryCapacity
            else:
                // Can't complete this path without running out of battery
                isPathFeasible = false
                break
            // Update battery level
            remainingBattery -= electricityForSegment
            totalElectricityUsed += electricityForSegment

            // Calculate customer cost based on demand (inverse surge pricing)
            segmentCost = edge.distance * BASE_PRICE * (1 / (1 + edge.demandLevel *
DEMAND_FACTOR))
            totalCustomerCost += segmentCost

        if isPathFeasible:
            // Calculate company profit
            profit = totalCustomerCost - totalElectricityUsed * ELECTRICITY_COST

            profitByPath[path] = profit
            customerCostByPath[path] = totalCustomerCost

    // Step 3: Select the optimal path (maximizing profit, minimizing customer cost)
    // We could use a weighted approach to balance the two objectives
    bestPath = null
    bestScore = 0

    for path in profitByPath.keys():
        // Normalize values between 0 and 1
        normalizedProfit = profitByPath[path] / max(profitByPath.values())
        normalizedCost = 1 - (customerCostByPath[path] / max(customerCostByPath.values()))

        // Weighted score (can adjust weights based on company priorities)
        score = PROFIT_WEIGHT * normalizedProfit + CUSTOMER_COST_WEIGHT *
normalizedCost

        if score > bestScore:
            bestScore = score
            bestPath = path

    return bestPath
```

3. time complexity of brute force

this is like a terrible version of distance matrix problems we've done in class.
this gets so big and out of hand on such an atrocious level. And we'd probably need one for every hour of each day of the week and some extra stuff for holiday or event traffic.

Assuming E is our number of edges in the graph, then this is O(V! * E) time and space complexity. This is a ludicrously awful approach. Let alone for a potentially infinite number of nodes and edges.

We've already done a really good job quantifying it into a graph in idea, but there's probably even dumber solutions that are still much better because the brute force is too much, we can partition and compartmentalize, and maybe generalize/simplify a few things that are a litlte extraneous.
At this point we'd be better off using existing map APIs and just borrowing google or someone else's pre-existing work.
And that work already uses similar shortcut ideas to what I've suggested by focusing traffic into main freeways entrances as nodes even then connecting those to more minor directions.
I don't want to know what that level of API access costs though, especially if we were doing it live, but we could handle that with the graph partitioning.

4. Algorithm planning:

A few of the more basic things from the brute force can be held over like the fuel checks and how we develop the cost itself
The main time saving angle in our math comes from avoiding needing to calculate every possible route to compare.
I kind of like a greedy approach since no matter what we do, we cut some corners, but I think I'll settle on A* with some partitioning and other basic sorting out
Some research says it’s standard for this but other research contradicts that, but I think it looks good.

Core Algorithm: A* search

Key design choices:
Graph Hierarchy:
By partitioning travel tours into local streets, arterial roads, and highways, we can heavily limit our nodes needed to properly travel.

Multi-Objective Cost:

Explain that the A* gScore minimizes a combined cost function: segmentCost = operationalCost - profit = operationalCost - (customerRevenue - electricityCost - rechargeCost). Detail how customerRevenue includes dynamic pricing (inverse demand) and electricityCost includes traffic/road type factors.

The enhanced state and enhanced heuristic are just there to help with pruning options a little earlier.

There's probably better ways to handle battery levels like just a literal if loop, but I wanted more things to be pre-processable.

5. Pseudocode

```
// Pre-processing: Partition the network
      function partitionNetworkAndPrecompute(graph):
        // Partition the graph into hierarchical levels
        partitions = {
          HIGHWAY_LEVEL: [],
          ARTERIAL_LEVEL: [],
          LOCAL_LEVEL: []
        }

        // Assign each node to appropriate level based on road classification
        for each node in graph.nodes:
          if node.isHighway:
            partitions[HIGHWAY_LEVEL].add(node)
          else if node.isArterial:
            partitions[ARTERIAL_LEVEL].add(node)
          else:
            partitions[LOCAL_LEVEL].add(node)

        // Identify boundary nodes between partitions
        boundaryNodes = findBoundaryNodes(partitions)

        // Pre-compute optimal paths between boundary nodes with metrics
        boundaryPaths = []
        for source in boundaryNodes:
          for target in boundaryNodes:
            if source != target:
              // Calculate path using standard A* on static/average conditions
              path = calculateOptimalPath(graph, source, target)
              if path:
                // Store detailed metrics for this boundary path
                boundaryPaths.add({
                  startNode: source,
                  endNode: target,
                  path: path,
                  estimated_distance: calculatePathDistance(path),
                  estimated_electricity: calculatePathElectricity(path),
                  estimated_time: calculatePathTime(path),
                  estimated_customer_cost: calculatePathCustomerCost(path),
                  estimated_combined_cost: calculatePathCombinedCost(path)
                })

        return {partitions, boundaryNodes, boundaryPaths}
```

```
    // Enhanced state representation including partition information
    class EnhancedState:
        constructor(node, batteryLevel, partition):
            this.node = node
            this.batteryLevel = batteryLevel // Battery level upon reaching this node
            this.partition = partition

        // Critical for efficient set operations and pruning
        equals(other):
            return this.node == other.node &&
                   abs(this.batteryLevel - other.batteryLevel) < BATTERY_TOLERANCE &&
                   this.partition == other.partition

        // Important for hash-based collections
        hashCode():
            // Discretize battery level to prevent explosion of states
            return hash(this.node) ^ hash(discretizeBatteryLevel(this.batteryLevel)) ^
hash(this.partition)

    // Helper function to discretize battery levels
    function discretizeBatteryLevel(exactBatteryLevel):
        // Convert continuous battery values to discrete levels
        // This significantly reduces the state space size
        return round(exactBatteryLevel / BATTERY_RESOLUTION) *
BATTERY_RESOLUTION

    // Main routing function
    function findOptimalRoute(graph, pickup, destination, batteryCapacity,
currentBatteryLevel):
        // Get or compute partitioned network data
        networkData = partitionNetworkAndPrecompute(graph)

        // Identify partitions containing pickup and destination
        pickupPartition = findContainingPartition(pickup, networkData.partitions)
        destPartition = findContainingPartition(destination, networkData.partitions)

        // Initialize data structures
        openSet = new PriorityQueue() // Ordered by fScore
        closedSet = new Set()        // States fully processed
        cameFrom = new Map()          // For path reconstruction
        gScore = new Map()           // Cost from start to state

        // Initialize with start node
```

```
startState = new EnhancedState(pickup, currentBatteryLevel, pickupPartition)
gScore[startState] = 0
fScore = enhancedHeuristic(startState, destination, networkData)
openSet.add(startState, fScore)

while not openSet.isEmpty():
    // Get state with lowest fScore
    currentState = openSet.pop()
    currentNode = currentState.node
    currentBattery = currentState.batteryLevel
    currentPartition = currentState.partition

    // Check if destination reached
    if currentNode == destination:
        return reconstructPath(cameFrom, currentState, gScore)

    // Mark as processed to avoid revisiting
    closedSet.add(currentState)

    // --- Option 1: Explore regular graph neighbors ---
    for each neighborNode in graph.getNeighbors(currentNode):
        edge = graph.getEdge(currentNode, neighborNode)
        neighborPartition = findContainingPartition(neighborNode, networkData.partitions)

        // Special handling for freeway transitions
        if isFreewayTransition(currentNode, neighborNode):
            costs = calculateFreewayTransitionCosts(edge, currentState)
        else:
            costs = calculateSegmentCostsAndProfit(edge, currentState)

        electricityForSegment = costs.electricity
        segmentCost = costs.segmentCost // (-profit + operationalCost)

        // Check battery feasibility
        // Cannot reach this neighbor if battery insufficient and not at charging station
        if currentBattery < electricityForSegment:
            if not (neighborNode.hasChargingStation and currentBattery >= MIN_ARRIVAL_BATTERY):
                continue // Skip this neighbor, battery insufficient

        // Calculate new battery level after traveling to neighbor
        newBatteryLevel = currentBattery - electricityForSegment
        rechargeCostAdjustment = 0 // Additional cost if recharge occurs
```

```
            // Handle charging if neighbor has station and we should charge
            // Either we arrived with low battery or charging is strategically beneficial
            if neighborNode.hasChargingStation and
               (newBatteryLevel < RECHARGE_THRESHOLD ||
                isStrategicToCharge(neighborNode, destination)):

                // Calculate recharge cost from current level to capacity
                rechargeInfo = calculateRechargeCostAndTime(
                    max(0, newBatteryLevel), // Can't have negative battery
                    batteryCapacity
                )

                rechargeCostAdjustment = rechargeInfo.cost
                newBatteryLevel = batteryCapacity // Fully recharged

            // Create neighbor state with new battery level
            neighborState = new EnhancedState(neighborNode, newBatteryLevel,
neighborPartition)

            // Skip if we've seen this state with better battery
            if closedSet.contains(neighborState) and
               closedSet.get(neighborState).batteryLevel >= newBatteryLevel:
                continue

            // Calculate total cost to reach this state
            tentativeGScore = gScore[currentState] + segmentCost +
rechargeCostAdjustment

            // If we found a better path to this state
            if not gScore.contains(neighborState) or tentativeGScore <
gScore[neighborState]:
                cameFrom[neighborState] = currentState
                gScore[neighborState] = tentativeGScore
                fScore = tentativeGScore + enhancedHeuristic(neighborState, destination,
networkData)

                // Add or update in open set
                if openSet.contains(neighborState):
                    openSet.updatePriority(neighborState, fScore)
                else:
                    openSet.add(neighborState, fScore)

        // --- Option 2: Explore hierarchical shortcuts (Boundary Paths) ---
```

```
            // Only consider shortcuts if current node is a boundary node
            if currentNode in networkData.boundaryNodes:
                // Get boundary paths that make progress toward destination
                for boundaryPath in getRelevantBoundaryPaths(currentNode, destination,
networkData):
                    targetNode = boundaryPath.endNode
                    targetPartition = findContainingPartition(targetNode, networkData.partitions)

                    pathElectricity = boundaryPath.estimated_electricity
                    pathCombinedCost = boundaryPath.estimated_combined_cost

                    // Check if battery sufficient for this boundary path
                    if currentBattery < pathElectricity:
                        // Skip if insufficient battery for this shortcut
                        continue

                    newBatteryLevel = currentBattery - pathElectricity
                    targetState = new EnhancedState(targetNode, newBatteryLevel,
targetPartition)

                    // Skip if already processed with better battery
                    if closedSet.contains(targetState):
                        continue

                    tentativeGScore = gScore[currentState] + pathCombinedCost

                    // If we found a better path via hierarchy
                    if not gScore.contains(targetState) or tentativeGScore < gScore[targetState]:
                        cameFrom[targetState] = currentState
                        gScore[targetState] = tentativeGScore
                        fScore = tentativeGScore + enhancedHeuristic(targetState, destination,
networkData)

                        // Add or update in open set
                        if openSet.contains(targetState):
                            openSet.updatePriority(targetState, fScore)
                        else:
                            openSet.add(targetState, fScore)

        // No path found
        return null

    // Calculate costs for regular road segments
    function calculateSegmentCostsAndProfit(edge, currentState):
        // Calculate electricity consumption based on traffic and road type
        baseConsumption = BASE_CONSUMPTION_RATE * edge.distance
        trafficFactor = 1 + (edge.trafficLevel * TRAFFIC_IMPACT_FACTOR)
        roadTypeFactor = getRoadTypeEfficiencyFactor(edge.roadType)
        electricity = baseConsumption * trafficFactor * roadTypeFactor

        // Calculate customer cost with inverse surge pricing (more demand = lower price)
        basePrice = BASE_PRICE * edge.distance
        demandFactor = 1 / (1 + (edge.demandLevel * DEMAND_DISCOUNT_FACTOR))
        customerCost = basePrice * demandFactor

        // Calculate operational cost based on vehicle availability
        availabilityFactor = 1 + ((1 - edge.vehicleAvailability) *
AVAILABILITY_IMPACT_FACTOR)
        operationalCost = BASE_OPERATIONAL_COST * edge.distance * availabilityFactor

        // Calculate profit
        profit = customerCost - (electricity * ELECTRICITY_COST_PER_UNIT)

        // Cost function A* minimizes: (-profit + operationalCost)
        segmentCost = -profit + operationalCost

        return {electricity: electricity, segmentCost: segmentCost}

    // Calculate costs for freeway transitions (on/off ramps)
    function calculateFreewayTransitionCosts(edge, currentState):
        if edge.source.isFreeway and not edge.target.isFreeway:
            // Off-ramp (deceleration uses less electricity)
            electricityFactor = OFF_RAMP_FACTOR
        else:
            // On-ramp (acceleration uses more electricity)
            electricityFactor = ON_RAMP_FACTOR

        // Calculate electricity with ramp-specific factors
        baseConsumption = BASE_CONSUMPTION_RATE * edge.distance
        trafficFactor = 1 + (edge.trafficLevel * TRAFFIC_IMPACT_FACTOR)
        electricity = baseConsumption * trafficFactor * electricityFactor

        // Customer cost calculation remains similar
        basePrice = BASE_PRICE * edge.distance
        demandFactor = 1 / (1 + (edge.demandLevel * DEMAND_DISCOUNT_FACTOR))
        customerCost = basePrice * demandFactor

        // Operational cost with ramp-specific considerations
```

```
            availabilityFactor = 1 + ((1 - edge.vehicleAvailability) *
AVAILABILITY_IMPACT_FACTOR)
            // Ramps may have additional operational costs due to increased wear
            operationalCost = BASE_OPERATIONAL_COST * edge.distance * availabilityFactor *
RAMP_OP_COST_FACTOR

            // Calculate profit
            profit = customerCost - (electricity * ELECTRICITY_COST_PER_UNIT)

            // Cost function A* minimizes: (-profit + operationalCost)
            segmentCost = -profit + operationalCost

            return {electricity: electricity, segmentCost: segmentCost}

        // Enhanced heuristic function that estimates remaining cost
        function enhancedHeuristic(state, destination, networkData):
            // Basic distance heuristic - geographic distance
            // THIS MUST BE ADMISSIBLE (never overestimate true cost)
            distance = calculateDistance(state.node, destination)

            // Estimate minimum electricity consumption to destination
            // Uses direct distance * minimum consumption rate (best case)
            estimatedMinElectricity = distance * MIN_CONSUMPTION_RATE

            // Minimum cost per distance unit (best case scenario)
            minCostPerDistance = calculateMinPossibleCostPerUnit()

            // Base cost heuristic (distance * min cost per unit)
            baseCostHeuristic = distance * minCostPerDistance

            // Partition-aware component if in different partitions
            partitionHeuristic = 0
            if state.partition != findContainingPartition(destination, networkData.partitions):
                // Estimate minimum cost of crossing partitions
                partitionHeuristic = estimateMinPartitionTransitionCost(
                    state.partition,
                    findContainingPartition(destination, networkData.partitions)
                )

            // Battery heuristic component - conservative estimate
            batteryHeuristic = 0
            if state.batteryLevel < estimatedMinElectricity:
                // We'll need to recharge at least once - add minimum possible recharge cost
                // This is admissible if it's the minimum possible recharge cost
                batteryHeuristic = MIN_RECHARGE_COST

            // Combine all heuristic components
            return max(0, baseCostHeuristic + partitionHeuristic + batteryHeuristic)

        // Function to determine if it's strategic to charge at this location
        function isStrategicToCharge(node, destination):
            // Consider charging if:
            // 1. Node is close to highway entrance (charge before highway)
            // 2. Few charging stations ahead in route
            // 3. Destination is far and charging stations are sparse

            // Simple implementation - charge if near highway and destination is far
            return node.isNearHighway and calculateDistance(node, destination) >
LONG_DISTANCE_THRESHOLD

        // Helper function for path reconstruction
        function reconstructPath(cameFrom, finalState, gScore):
            path = [finalState.node]
            currentState = finalState

            // Trace back the path
            while currentState in cameFrom:
                currentState = cameFrom[currentState]
                path.insert(0, currentState.node)

            // Calculate metrics for the final path
            totalDistance = calculatePathDistance(path)
            totalElectricity = calculatePathElectricity(path)
            totalCustomerCost = calculatePathCustomerCost(path)
            totalProfit = calculatePathProfit(path)

            return {
                path: path,
                distance: totalDistance,
                electricity: totalElectricity,
                customerCost: totalCustomerCost,
                profit: totalProfit,
                cost: gScore[finalState]
            }

        // Helper function to get relevant boundary paths
        function getRelevantBoundaryPaths(currentNode, destination, networkData):
            relevantPaths = []
            // Get all boundary paths starting from current node
            for path in networkData.boundaryPaths:
                if path.startNode == currentNode:
                    // Only consider paths that make progress toward destination
                    if isProgressTowardDestination(path.endNode, currentNode, destination):
                        relevantPaths.append(path)

            return relevantPaths

        // Helper function to check if a node is closer to destination
        function isProgressTowardDestination(node, currentNode, destination):
            currentDistance = calculateDistance(currentNode, destination)
            nodeDistance = calculateDistance(node, destination)

            // Progress is made if the new node is closer to destination
            // With some tolerance for slight detours
            return nodeDistance < currentDistance * PROGRESS_FACTOR
```

6. Complexity Analysis:

Variables:
B: boundary nodes, these are nodes on the edges of partitions
V: vertices/nodes
E: edges
P: partitions (this is like 3 in the current design, but could easily be implemented further).
K: battery level, generalized to 10 mile blocks of driving energy range (this is bad design in case of emergencies in retrospect, but I'm not the boss), realistically 20-50 possibilities

TIME: Pre-processing:O(B^2 * (E+ V log V)

O(V+E): partitioning and classifying all nodes and edges, a liner scan over the graph.

O(V): identifying boundaries

O(B^2 * (E + V log V)) realistically this is O (B^2) where we compute paths between boundary nodes and each path uses A*to evaluate

Runtime: O(V * K log(K * V)

V * K comes from battery levels, and nodes

Evaluating our edges: Olog(V * K) comes from our priority queue, could be as bad V*K as every edge might need to be considered for every battery level
This is of course way better than a factorial time complexity from our brute force. Also some of the battery level stuff can just be an input or client side, this could be a client-side input.

Space complexity: **(O (B^2 * V + E):**
V+E: storing all graph information
B^2 * V: worst case of boundary nodes and normal nodes

Space Runtime Storage: O(V*K)
We just need the general space here.

In general this is just a lot better than any factorial.