Routing Policy Evaluation for Electric Vehicle Charging Systems: A Discrete-Event Simulation Study

Ava Birtwistle
V00970244
Mateya Berezowsky
V00994661

Professor: Dr. Kui Wu
University of Victoria

Computer Science
CSC 446/546
Section A02

December 2025

Contents

Contents **1**

List of Figures **3**

1 Introduction **4**

1.1 Problem Description 4
1.2 Goals of the Simulation Study 4

2 System Model **5**

2.1 Description of the Real System 5
2.2 Modeling Approach 6
2.3 Queueing Model and Service Discipline 6
2.4 Assumptions and Simplifications 7
2.4.1 Travel in the Simulation Plane 7
2.4.2 Battery 8

3 Methodology **10**

3.1 Architecture and Flow 10
3.1.1 Classes 10
3.2 Performance Metrics 18
3.3 Parameter Setup and Justification 18
3.4 Simulation Run Length and Termination Criteria 19
3.5 Scenario Descriptions 20

4 Results **21**

5 Discussion **26**

5.1 Interpretation of Results 26
5.1.1 Run with Original Parameters 26
5.1.2 Decreasing Minimum Charge Amount 26
5.2 Engineering Implications 27

6 Conclusion **28**
6.1 Summary of Findings 28
6.2 System Improvement Recommendations 28
6.3 Limitations 29

A Appendix A **31**

List of Figures

2.1 The area represented by the simulation plane. 5
2.2 The representation of the service area used in the simulation model. . . 7

3.1 The logical steps in `system.py`. 14
3.2 The routing policies used in the simulation. 16

4.1 The original run. 22
4.2 The minimum charge amount is 10% 23
4.3 The balking threshold battery level is reduced to 25%. 25

1 Introduction

1.1 Problem Description

Electric vehicles (EVs) have experienced rapid growth in British Columbia, which now has the highest EV adoption rate in North America [1]. As demand on the public fast-charging network increases, congestion has become a significant factor affecting system performance and driver satisfaction. According to BC Hydro's Public EV Charging Evaluation Report, 76% of surveyed BC Hydro network members in urban areas are willing to wait ten minutes or less for a charger, indicating that most drivers expect immediate or near-immediate access [3]. In non-urban areas, tolerance is somewhat higher, with 46% of drivers willing to wait fifteen minutes or more, but overall the survey highlights a strong preference for minimal queueing.

While BC Hydro provides real-time station availability through its public charging map, routing decisions remain entirely user-driven, often leading to unoptimized station utilization and overall longer time spent trying to publicly charge vehicles. As EV's become more prominent, understanding how routing strategies could improve charging accessibility is relevant. To explore this, we develop a simulation of EV arrivals within a defined region of Victoria, BC, and evaluate alternative routing policies to assess their impact on total time spent accessing public charging, including both travel time and queueing delay.

1.2 Goals of the Simulation Study

The goal of the study is to provide a model to evaluate what vehicle routing policies reduce the total amount of time that a car spends from when a driver decides that they want to charge their vehicle to when they have completed charging.

2

System Model

2.1 Description of the Real System

The system studied represents a small network of electric vehicle (EV) charging stations located within a defined service area. The physical environment is modeled as a rectangular region approximately 7.5 km by 13 km. Three charging stations are situated within this region at the following coordinates:

Station #	x (km)	y (km)
1	3.62	2.93
2	9.29	4.91
3	10.32	1.74

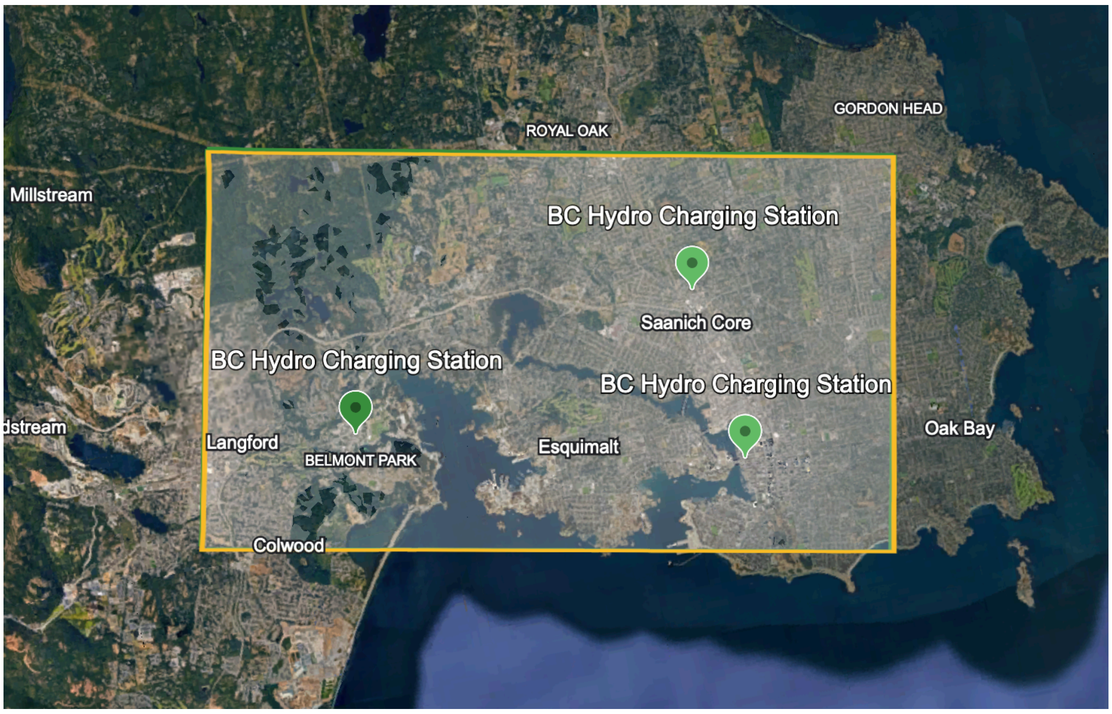

Figure 2.1: *The area represented by the simulation plane.*

Each station contains two chargers: one fast charger and one slow charger, which provide different power levels and therefore different charging durations. Vehicles travel through the region over time and differ in their battery capacities, initial state-of-charge (SoC), and the amount of energy required to reach their desired charge level.

When drivers arrive at a station, they begin charging if a charger is open. If both chargers are already in use, the driver must wait until a charger becomes free before charging can start. The length of time a vehicle remains at the station depends on its initial SoC, its target charge level, and the power rating of the charger it eventually uses. Once charging is completed, the vehicle leaves the station and exits the system.

2.2 Modeling Approach

The simulation model approximates the real environment using a 7.5 km by 13 km rectangular plane that preserves the spatial relationships between the real life stations modeled. Vehicles are assumed to have known GPS coordinates and initial SoC at the moment routing decisions are made, consistent with how a navigation service or routing API would operate.

The simulation is implemented as a discrete-event system, in which events types corresponding to vehicle arrivals to the system, vehicle arrivals to chargers, and departures.

Travel times between vehicles and stations are computed using Euclidean distance, and charging durations are computed using the vehicle's initial SoC, target SoC, battery capacity, and the power rating of the assigned charger. Figure 2.2 shows the simplified geometric representation used in the model. In order to model real time factors such as drive time, the euclidean distance is used to compute the drive time for the vehicle to the charger at a speed of 30km per hour.

2.3 Queueing Model and Service Discipline

Each station is represented as a two-server, single-queue model, with one server corresponding to a fast charger and the other to a slow charger. Arrivals to the system are generated by a Poisson process. Vehicles are routed to a charging station according to the active routing policy. When a vehicle arrives and both chargers are busy, it is placed in a single FCFS waiting line. When a charger becomes available, the next vehicle in line is assigned to it. If the total number of cars already in the queue and currently en route to the queue exceeds 20, and the vehicle's battery level is above a specified threshold, the vehicle is routed to a different station when possible; otherwise, it balks.

Service times are computed from the energy required for the vehicle to reach its target state of charge and the power rating of the assigned charger, according to the

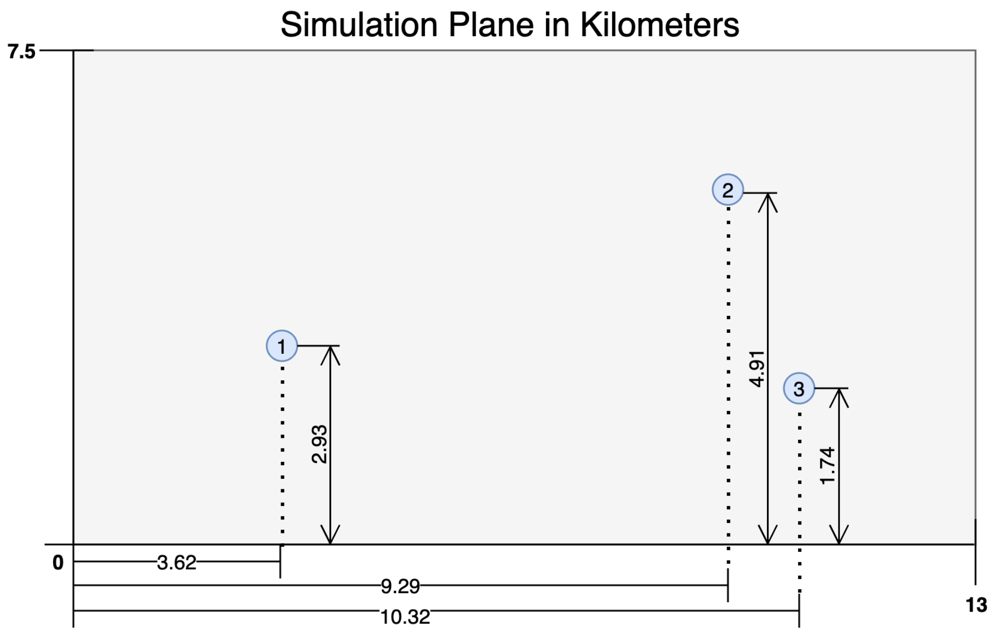

Figure 2.2: *The representation of the service area used in the simulation model.*

following equation [4]:

$$T_{\text{minutes}} = 60 \cdot \frac{(SOC_{\text{target}} - SOC_{\text{initial}}) \cdot \text{Battery Capacity (kWh)}}{100 \cdot \text{Charger Power (kW)}}.$$

Because of this, each station can be viewed as an $M/G/2$ system with two servers that operate at different rates.

2.4 Assumptions and Simplifications

The following assumptions are applied in the simulation model:

2.4.1 Travel in the Simulation Plane

To model vehicle movement within the simulation plane, the following assumptions and simplifications are made:

1. The service area is represented as a two-dimensional plane. Consequently, the following effects are not captured:
   - Elevation changes when computing distances and drive times,
   - Battery consumption differences caused by uphill or downhill travel, and
   - Any elevation-related adjustments that would be required when applying the model to regions with significant terrain variation.
2. Vehicles are assumed to be able to travel freely across the 2-D plane.

- Bodies of water, buildings, parks, and other areas where vehicles cannot drive in the real world are not represented in the simulation.

3. Vehicle travel is modeled as straight-line (Euclidean) movement rather than movement along real roads. This means:
    - The path between two points is taken as the geometric shortest distance, not the actual route a car would drive.
    - Drive time is based on this distance and a constant speed, so it reflects an average travel time rather than a detailed road-specific estimate.
    - Road features such as turns, intersections, traffic signals, and detours are not included.
4. The simulation does not enforce that vehicles arrive at a station in the same order they are created or routed. This reflects real-world behaviour:
    - A later-created vehicle may arrive earlier if it is closer to the station.
    - Travel times vary due to distance, traffic conditions, and individual driving behaviour.
    - Routing decisions are made when a vehicle enters the system, not when it reaches the station, so the queue length observed at routing time may differ from the queue on arrival.
5. Vehicles are assumed to travel at a constant average speed of 30 km/h. This simplifies several real-world factors that are not modeled in the simulation:
    - Variation in road speeds
    - Traffic congestion
    - Elevation changes
    - Individual driving behaviour
    - The use of Euclidean distance rather than road-network paths.
6. Vehicles and charging stations are treated as points in the plane. This implies:
    - A vehicle may share the exact coordinates of a station, in such cases, the vehicle is assumed to already be at the station, and the drive time for that routing decision is set to zero.

2.4.2 Battery

1. Each vehicle's initial SOC is generated at the time of vehicle generation and sampled uniformly between 20% and 60%.
    - The initial SoC is considered as a known factor by the system so is elligible to use in routing descisions
2. Each vehicle's target SOC is selected uniformly between its initial SOC plus 20% to 80%.
    - The target SOC is not known to the system.

- This models the driver choosing how much they want to charge based on their own plans or needs.

3. The vehicle's state of charge after the drive is computed from the distance to the station using
$$\mathrm{SOC}_{\mathrm{after}} = \mathrm{SOC}_{\mathrm{initial}} - \frac{d \cdot E_{\mathrm{drive}}}{C_{\mathrm{battery}}}.$$
   - Driving reduces SOC according to this calculation
   - SOC is not updated continuously during travel, and
   - The resulting value is used when determining charging (service) time.
4. Battery lost during idle periods while waiting in the line up is not factored into the simulation.

3

Methodology

3.1 Architecture and Flow

3.1.1 Classes

The overall architecture for the system is made of 6 classes that model the physical characteristic of the system and the behavioural characteristics.

`car.py`

The Car class models vehicle objects generated in the system. Each vehicle is instantiated with a set of known initial parameters including spawn position, initial state of charge (SoC), and the list of `station_meta` objects which include the reachable stations and the relational data between the car and that station. Additional attributes required later in the simulation, including drive time to the routed station, charging time, queueing time, and total time in the system, are initialized as `None` and are populated once the routing policy assigns the car to a station Listing A.1.

When a car arrival event is generated in the system the following steps occur:

1. The cars arrival time to the system is recorded as the current simulation clock.
2. The car's position is sampled uniformly from the simulation plane based on Equation 3.1
$$(x, y) \sim U([X_{\min}, X_{\max}] \times [Y_{\min}, Y_{\max}]). \tag{3.1}$$
3. The initial battery level (SoC) is generated as a floating-point percentage sampled uniformly within the allowable battery range according to Equation 3.2.
$$\text{SoC}_{\text{initial}} \sim U([B_{\min}, B_{\max}]) \tag{3.2}$$
This value is known by the system and used for eliminating unreachable stations and the computing the battery remaining after a cars drive used for the service rate calculation.

4. The target charge level (SoC) is generated as a floating-point percentage sampled uniformly between the initial SoC plus a minimum charge amount of $\Delta 20\%$ and the maximum charge amount of $SOC_{MAX} = 80\%$ [2].

$$\text{SoC}_{\text{target}} \sim U\left([\text{SoC}_{\text{initial}} + 0.20,\ 0.80]\right) \tag{3.3}$$

   This reflects the variability in human charging behaviour, as vehicles do not all charge to the same final SoC in the real world. The target level is determined when the vehicle is created, but it is not available to the routing logic and is treated as an unknown parameter within the system.
5. A list of stations that can be reached by the car is constructed. For each station in the system, the distance from the car, the associated drive time, and the estimated SoC upon arrival are recorded in a `Station_Meta` object. Any station that cannot be reached is excluded from the list. This list is later used during routing, allowing policies to select a station without requiring modifications to the underlying data structure.

   - The Euclidean distance from the spawn point to the station,

$$d_{\text{km}} = \sqrt{(x - x_s)^2 + (y - y_s)^2}, \tag{3.4}$$

   - The estimated SoC after driving,

$$\begin{aligned} E_{\text{drive}} &= d_{\text{km}} \cdot \texttt{ENERGY\_CONSUMPTION\_RATE}, \\ \text{SOC}_{\text{after}} &= \text{SOC}_{\text{initial}} - \frac{E_{\text{drive}}}{\texttt{BATTERY\_CAPACITY}} \times 100 \end{aligned} \tag{3.5}$$

     Where E_{drive} is the energy consumed from the drive.

   Stations for which the estimated SoC falls below `MIN_BATTERY_THRESHOLD` are discarded as unreachable. The remaining stations are stored as `Station_Meta` objects containing the distance, drive time (computed using `SPEED_KM`), and $\text{SOC}_{\text{after}}$.
6. Fields used later in routing, queueing, and charging are initialized but not filled at creation:

   - `routed_drive_time`,
   - `routed_arrival_time`,
   - `time_charging`,
   - `time_in_queue`,
   - `total_time_in_system`.

   These are updated later by the routing module and the charging station processes.

charging_station.py

The charging station object represents a single service node in the system. Three stations are modelled, each equipped with one fast charger, one slow charger, and a shared FCFS queue Listing A.4.

The behaviour of each charging station is characterized by the following:

1. **Initialization:** Each station is configured with a unique identifier, a fixed x,y coordinate within the simulation plane, a reference to the global simulation clock, charger availability states, an internal FCFS queue, and event-type mappings for arrivals and departures.
2. **Arrival Processing:** When an arrival event is generated, charger availability is evaluated. If a fast charger is available, the vehicle is sent there otherwise it is sent to the slow charger. If both are occupied, the vehicle joins the queue.
3. **Service-Time Calculation:** After a vehicle is assigned a charger, the service time in minutes is determined. The service time for a vehicle is defined as the duration for the vehicle to charge from its post travel SoC to its target charge level. This is calculated by first determining the delta which the car needs to charge. Then the time in minutes needed for the vehicle to gain that delta is calculated according to Equation 3.6. The computed service time is then used to schedule the corresponding departure event [4].

$$T_{\text{minutes}} = 60 \cdot \frac{(SOC_{\text{target}} - SOC_{\text{initial}}) \cdot \text{Battery Capacity (kWh)}}{100 \cdot \text{Charger Power (kW)}}. \tag{3.6}$$

4. **Departure Processing:** When a departure event occurs, the associated charger is freed. If the queue contains waiting vehicles, the next vehicle is admitted, its waiting time is recorded, its service time is computed, and its departure event is scheduled. If the queue is empty, the charger remains idle.

station_meta.py

This class is used to store the station-specific values that depend on a vehicles initial spawn point. When a vehicle is instantiated, a list of `station_metadata` objects is constructed Listing A.5.

This class stores station-specific quantities that depend on a vehicle's initial state. For each vehicle, a list of metadata objects is constructed at creation time for all reachable charging stations, capturing values such as distance, drive time, and estimated state of charge after travel. This metadata is later used by the routing logic to compare and select a destination station.

This class is used to represent the association between a vehicle and a charging station. Charging stations are modeled independently as physical infrastructure, and a vehicle's routed station is not known at the time of vehicle generation. The routing part of the simulation needs specific information for a vehicles relationship to the stations within the system.

system.py

The `EV_Charging_System` object represents the entry point and global controller of the simulation. It initializes core parameters, manages event execution, coordinates routing decisions, and maintains system-level performance statistics Listing A.2.

The behaviour of the EV charging system is characterized by the following:

1. **Initialization:** The routing policy is selected, the required number of completed services is recorded, and statistical counters for wait times, balking, and reneging are initialized. A global simulation clock and event-list priority queue are created, and the first system-arrival event is scheduled using an exponentially distributed inter-arrival time. Three charging stations are instantiated at fixed spatial coordinates and linked to the global simulation time. These coordinates remain constant throughout simulation runs. A void-counter vector is initialized to track cars en route to each station. The flow logic can be seen in Figure 3.1.
2. **Main Simulation Loop:** The simulation repeatedly executes events until the required number of serviced vehicles has been reached. At each iteration, the timing function removes the earliest event from the event list, advances the simulation clock, and identifies the corresponding event type and car (if applicable). The appropriate arrival or departure handler is invoked. Upon completion of all required departures, system-wide performance statistics are computed and reported.
3. **System Arrival Processing:** When a system-arrival event occurs, the next arrival is scheduled using an exponential interarrival time. Routing is then performed according to the selected policy. If no acceptable station is found, the customer balks otherwise, the travel time to the selected station is determined, and an arrival-at-station event is scheduled.
4. **Reneging Logic:** Reneging is evaluated during each system-arrival event. If a station queue exceeds five vehicles and the sixth vehicle has waited more than fifteen minutes, the vehicle abandons the queue and is counted as a reneging event. This was selected in accordance to a BC hydro study on driver willingness to wait in queues [2].

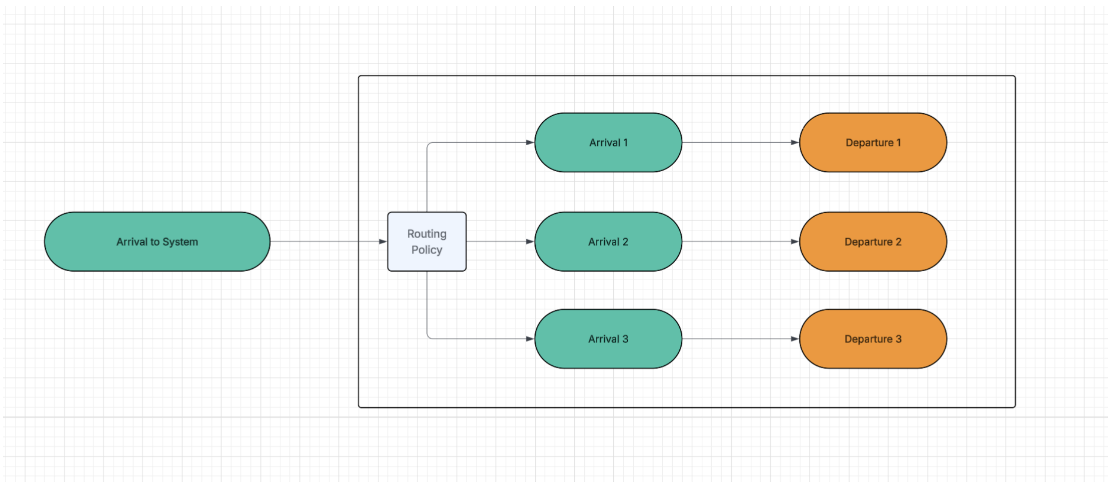

Figure 3.1: *The logical steps in* `system.py`.

routing.py

The `Routing` object determines the charging station each arriving vehicle is assigned to. It applies the selected routing strategy, evaluates station admissibility, and updates all relevant state variables associated with the routing decision. The routing component functions as the decision-making layer between the vehicle and its set of reachable stations Listing A.3.

The behaviour of the routing component is characterized by the following and follows Figure 3.2:

1. **Policy Initialization and Lookup:** The routing object stores a reference to the vehicle being routed, the routing policy selected in `system.py`, and the void-counter vector that tracks vehicles currently en route to each station. An internal mapping binds each routing policy enumerator to its corresponding implementation. During routing, the `route()` method retrieves the appropriate policy function for the name specified in Listing A.6 and delegates execution to it.
2. **Closest-Station-First Routing:** Under this policy, reachable stations are sorted implicitly by drive time, and candidates are evaluated in order of increasing travel distance. For each station, a verification step assesses effective queue length, SoC upon station arrival, and predefined system thresholds. If the station is admissible, the routing decision is finalized; otherwise, the next closest station is examined. If no reachable stations satisfy the verification criteria, the vehicle balks.
3. **Shortest-Estimated-Wait Routing:** This policy constructs an estimated total delay for each reachable station by summing the drive time and a time-scaled queue wait estimate. Stations that fail the verification step are discarded. Among the remaining candidates, the station that minimizes estimated total delay is selected. If all candidates are rejected, no routing decision is made and the vehicle balks.
4. **Station Verification Logic:** Verification is performed using the `_verify_station_()` method. A station is deemed inadmissible if its effective queue length exceeds the global maximum queue threshold and the vehicle's post-drive SoC is above the defined balking level. This ensures that vehicles with sufficient battery consider alternative stations and prevents excessive concentration at congested locations.
5. **Routing Finalization and State Updates:** When a station is selected, the routing object updates all relevant vehicle fields, including drive time, SoC upon arrival, and expected station arrival timestamp. The void-counter associated with the chosen station is incremented to reflect a vehicle currently in transit. These updates maintain consistency in subsequent arrival, queueing, and departure operations across the system.

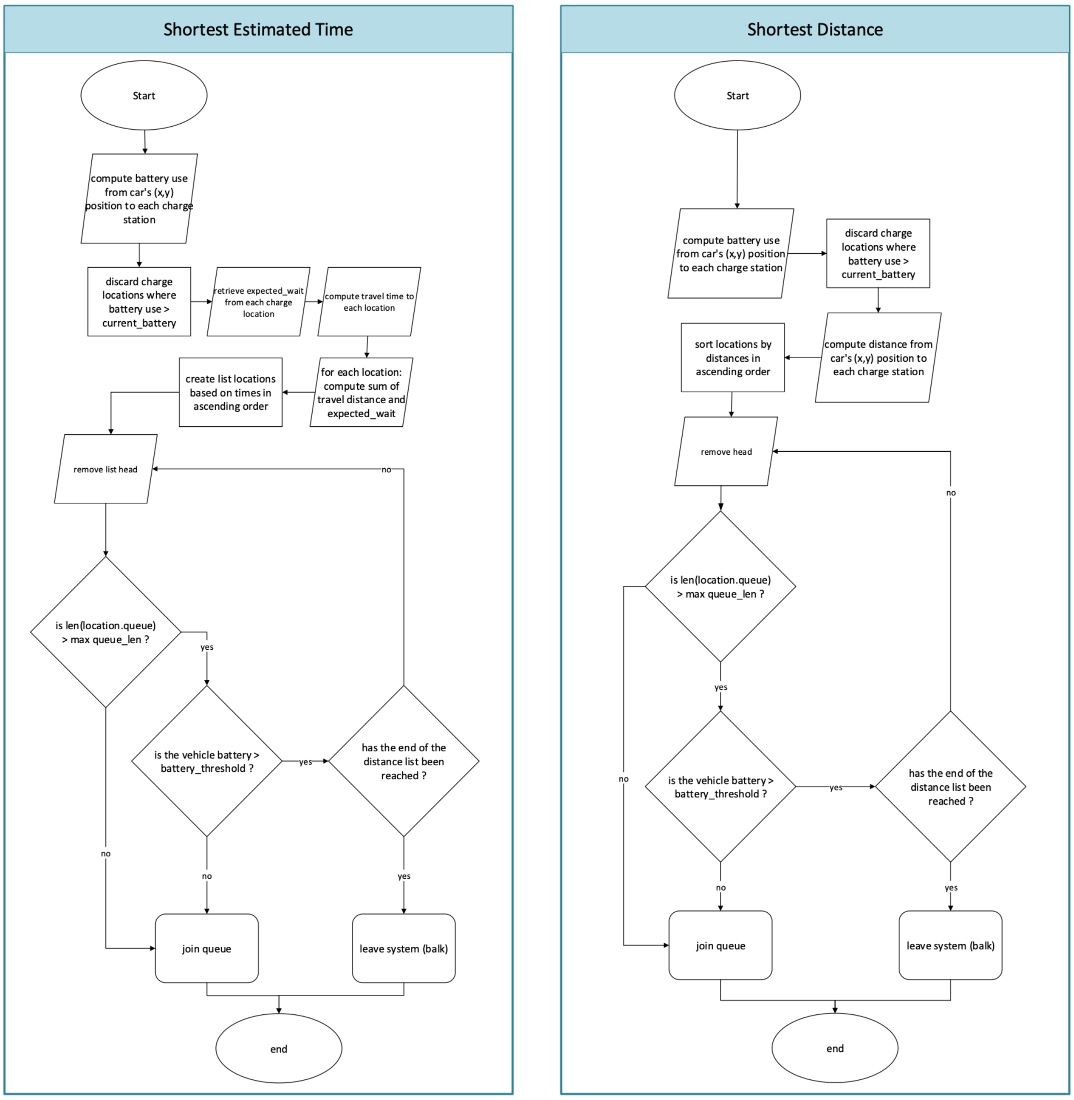
Shortest Estimated Time
Start
compute battery use from car's (x,y) position to each charge station
discard charge locations where battery use > current_battery
retrieve expected_wait from each charge location
compute travel time to each location
for each location: compute sum of travel distance and expected_wait
create list locations based on times in ascending order
remove list head
is len(location.queue) > max queue_len ?
yes
no
is the vehicle battery > battery_threshold ?
yes
no
has the end of the distance list been reached ?
no
yes
join queue
leave system (balk)
end
Shortest Distance
Start
compute battery use from car's (x,y) position to each charge station
discard charge locations where battery use > current_battery
compute distance from car's (x,y) position to each charge station
sort locations by distances in ascending order
remove head
is len(location.queue) > max queue_len ?
no
yes
is the vehicle battery > battery_threshold ?
yes
no
has the end of the distance list been reached ?
no
yes
join queue
leave system (balk)
end

Figure 3.2: *The routing policies used in the simulation.*

constants.py

This defines the constants used throughout the simulation and can be referenced in Listing A.7. Further details on their selection is available in Section 3.3.

event.py

Utilizes Enum to define the event types used in the simulation in `system.py` Listing A.8.

routing_policies.py

Uses an `Enum` to define the available routing policy names within the system. This allows the system to automatically invoke the routing function associated with each policy and decouples the routing logic so that future routing policies can be added Listing A.6.

3.2 Performance Metrics

The following performance metrics were collected to evaluate system throughput and delay across different simulation seeds and routing policy runs.

Table 3.1: *Performance metrics collected by the EV charging simulation.*

Metric	Description
Number of Cars Processed	Total number of vehicles that completed charging and departed the system.
Average Time in System	Mean time a vehicle spends in the system, total_time_in_system/num_cars_processed.
Average Wait Time	Mean waiting delay before charging begins, including driving, total_wait_time/num_cars_processed.
Total Balking Events	Number of vehicles that refused to join the system because no acceptable station was available at routing time.
Total Reneging Events	Number of vehicles that abandoned a queue after waiting longer than the allowed threshold.

3.3 Parameter Setup and Justification

To ensure that the simulation reflects realistic EV-charging behaviour in British Columbia, all parameters were chosen using BC Hydro documentation, typical EV specifications, and standard assumptions from discrete-event modelling. The parameters remained fixed across all replications unless otherwise noted, allowing for controlled comparisons between routing policies.

Vehicle Energy Parameters

A battery capacity of 75 kWh from Equation 3.2 was used to represent a mid-range electric vehicle. The energy consumption rate was set to 0.20 kWh/km, consistent with reported values for urban and suburban driving conditions. Initial battery levels were sampled uniformly between predefined minimum and maximum thresholds to capture heterogeneous arrival states among users.

Charging Behaviour

The minimum charge increment was set to 20%, reflecting typical user behaviour where drivers generally do not charge from very low to full capacity. The target state-of-charge (SoC) was sampled from an interval bounded below by the initial SoC plus the minimum increment and above by an 80% cap, modelling real-world patterns where users avoid unnecessarily long charging sessions.

Travel and Spatial Parameters

Vehicles were spawned uniformly across a specified two-dimensional region representing the service area. A fixed travel speed of 30 km/h was used to approximate mixed traffic environments. Euclidean distance was employed to compute travel times between vehicle spawn points and station coordinates.

Charging Station Parameters

Each charging station is configured with one fast charger with a charging power or $200kW$, and one slow charger, corresponding to $4.8kW$ [3]. The charger power rates along with the battery capacity defined in Section 3.3, are used in Equation 3.6 to determine the the service time for a vehicle to reach it's target state of charge.

Arrival Process

System arrivals were generated using an exponential interarrival distribution, consistent with standard Poisson-arrival assumptions in queueing systems. The mean interarrival time was selected to approximate one vehicle every five minutes which translates to approximately twelve vehicles arriving to the system per hour [6].

Queueing and Balking Thresholds

A maximum queue length of 10 was used to represent a queue length that a driver who does not urgently need to charge would realistically be willing to wait for, and to avoid routing vehicles to heavily congested stations [2]. When the queue length exceeds this value, vehicles with battery levels above the balking threshold either check any remaining stations they can reach or do not enter the system. The queue length includes both vehicles waiting at the station and vehicles that have been assigned to the station but have not yet arrived. This was decided as considering only the current physical queue would underestimate the queue length at time of arrival.

3.4 Simulation Run Length and Termination Criteria

Each replication of the simulation was executed with a target sample size of 100,000 serviced cars. The simulation terminated only after 100,000 cars had completed service which means customers who balked or reneged were not counted as serviced; therefore, their departures did not contribute toward the sample size. The main event loop continued to run until the total number of serviced cars reached the required sample size for the replication.

3.5 Scenario Descriptions

To examine how modelling assumptions influence overall system behaviour, three alternative parameter configurations were considered. Each scenario modifies one aspect of the baseline parameters described in Section 3.3, allowing us to isolate the effect of that specific change.

Baseline Scenario

The first scenario uses the parameter selection defined in Section 3.3 , which was chosen to represent typical and realistic EV-charging behaviour in British Columbia. These parameters reflect average charging patterns, typical battery levels on arrival, and long-term conditions under relatively stable interarrival times.

Decreasing the Minimum Charge Amount

In the second scenario, the minimum charge increment is reduced. Instead of sampling the target state-of-charge from an interval beginning at the initial SoC plus 20%, the lower bound is decreased so that vehicles may request smaller charge amounts. For example, the target SoC may be drawn from

$$U([\mathrm{SoC}_{\mathrm{initial}} + 0.10,\ 0.80]),$$

allowing drivers to seek increments as small as 10%. This scenario reflects environments where shorter charging sessions are more common, such as dense urban areas or settings where drivers make frequent but brief stops.

Decreasing the Balk Threshold

The final scenario modifies the balk threshold, which determines the minimum battery level at which a vehicle is willing to enter the system. Lowering this threshold allows vehicles to decline service more readily when stations appear too congested. This represents situations where drivers are more selective about entering queues, or where alternative charging opportunities outside the system may be available.

4

Results

Run with Original Parameters

The original run showed that the paired differences were positive for all seeds, as presented in Table 4.1. The mean paired difference was $\bar{D} = 0.9192$ minutes, and the 95% confidence interval of $[0.8473, ; 0.9911]$ lay entirely above zero, as shown in Table 4.2.

Table 4.1: *Paired differences between routing policies for each seed (original run).*

Seed	Difference (CSF – SEW)
3	0.8824
200	1.0196
303	0.9180
670	0.8854
1000	0.8907

Table 4.2: *CRN summary statistics for paired differences (original run).*

Statistic	Value
Mean Difference ($\bar{D}$)	0.9192
Half-width (H)	0.0719
95% Confidence Interval	$[0.8473,\ 0.9911]$

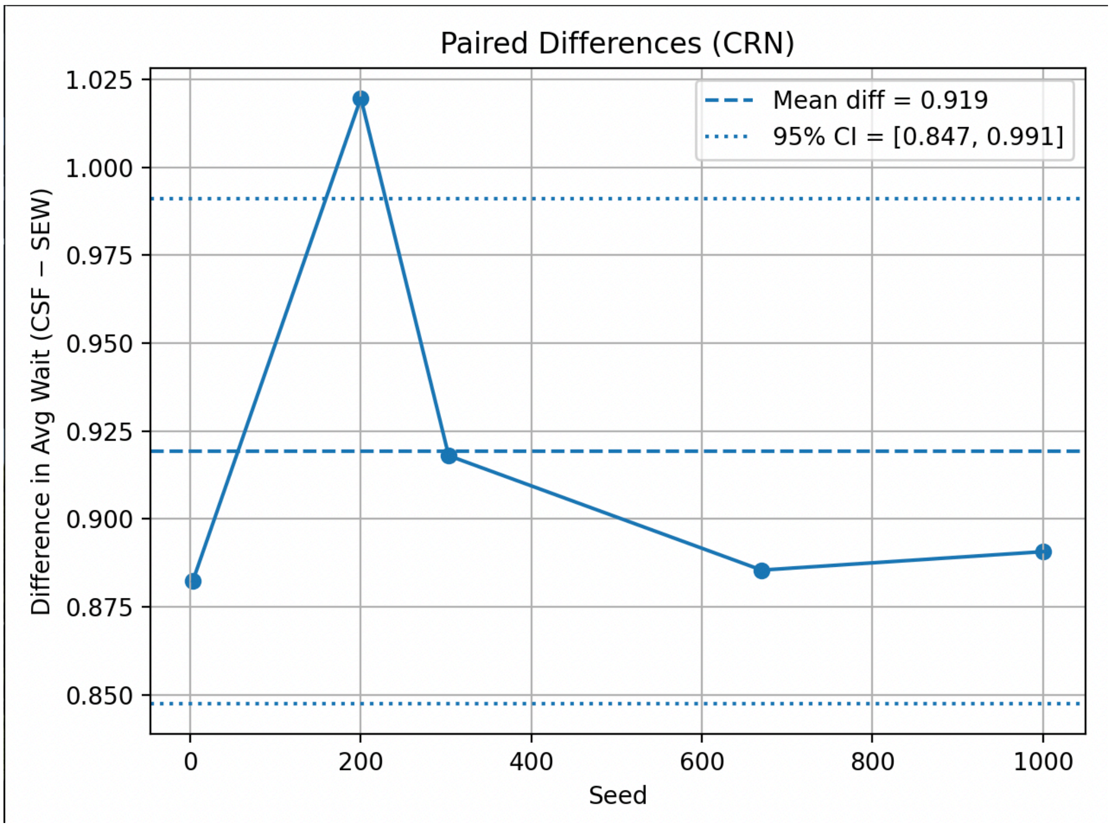

Figure 4.1: *The original run.*

Decreasing Minimum Charge Amount

The paired differences for this scenario are reported in Table 4.3, and all values are negative. The confidence interval plot for this scenario is shown in Figure 4.2.

Table 4.3: *Paired differences in average wait time for the confidence.py scenario.*

Seed	Paired Difference (minutes)
3	-0.3634
200	-0.3369
303	-0.3705
670	-0.4081
1000	-0.3980

Table 4.4: *CRN summary statistics for paired differences (minimum charge amount scenario).*

Statistic	Value
Mean Difference ($\bar{D}$)	-0.3754
Half-width (H)	0.0353
95% Confidence Interval	$[-0.4107, -0.3401]$

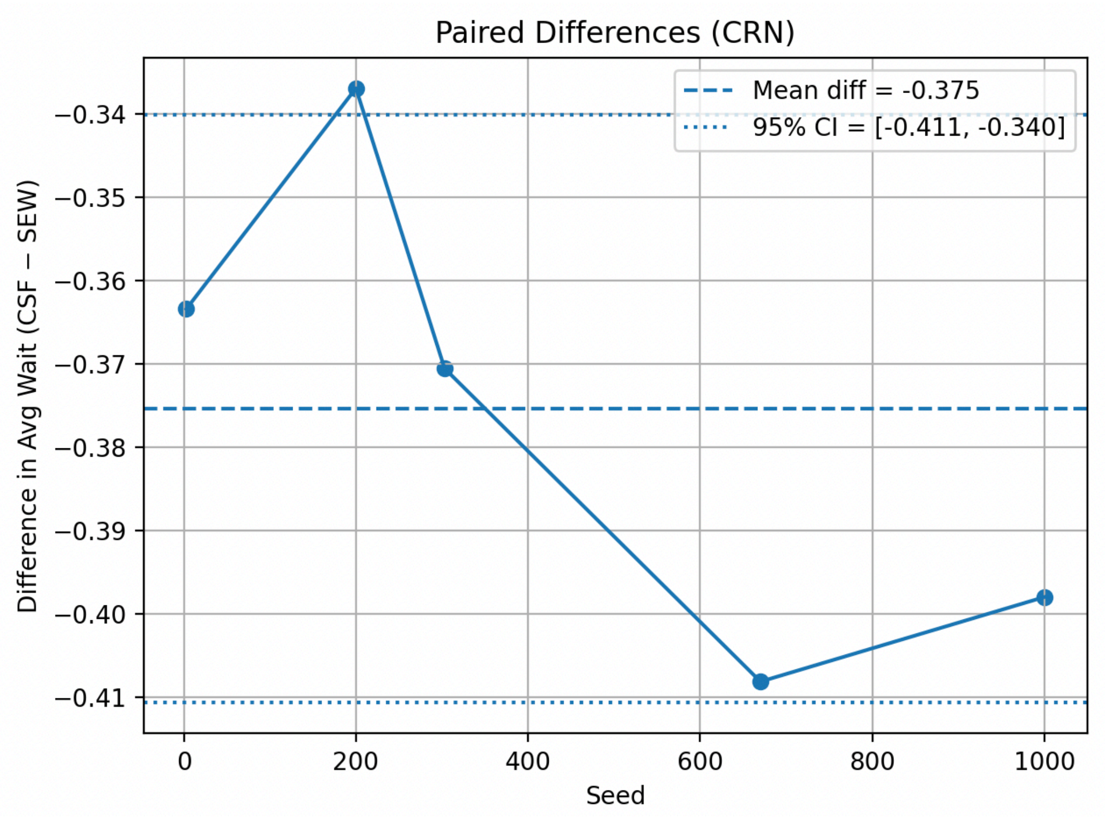

Figure 4.2: *The minimum charge amount is 10%*

Decreasing the Minimum Balk Battery Percentage

In suburban settings where most drivers have access to home chargers and residential areas are located relatively close to urban centers, the threshold at which drivers are willing to balk is typically lower. To represent this in the model, the battery threshold for balking was reduced from 60% to 25%. Under this setting, any vehicle arriving with at least 25% state of charge could choose not to enter the system if the expected wait times at all stations were sufficiently long.

The paired differences for this scenario are shown in Table 4.6. The mean paired difference was $\bar{D} = 0.9191$ minutes, and the 95% confidence interval of $[0.8477, ; 0.9905]$ lay entirely above zero, as reported in Table 4.5. The associated confidence interval plot is shown in Figure 4.3.

Table 4.5: *Paired differences in average wait time under the reduced balk threshold (25%).*

Seed	Paired Difference (minutes)
3	0.8824
200	1.0190
303	0.9177
670	0.8854
1000	0.8912

Table 4.6: *Summary statistics for paired differences under the reduced balk threshold scenario.*

Statistic	Value
Mean Difference ($\bar{D}$)	0.9191
Half-width (H)	0.0714
95% Confidence Interval	$[0.8477,\ 0.9905]$

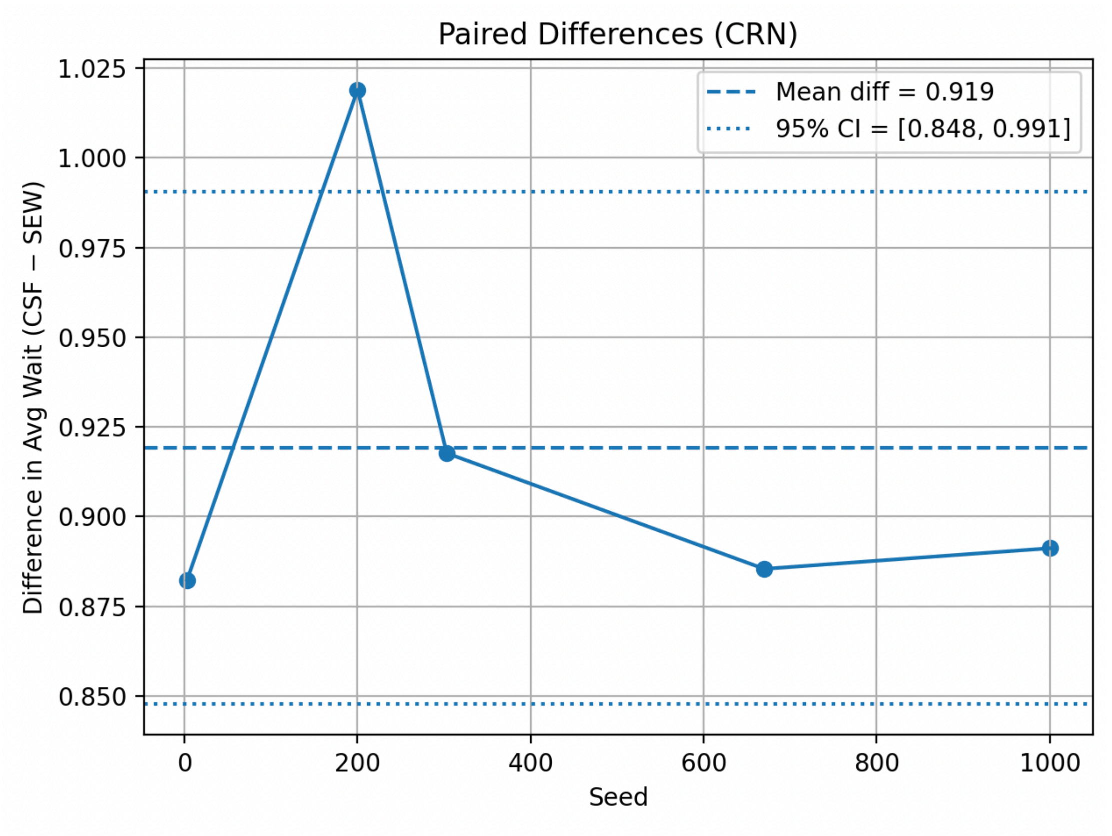

Figure 4.3: *The balking threshold battery level is reduced to 25%.*

5 Discussion

5.1 Interpretation of Results

5.1.1 Run with Original Parameters

The results shown in Figure 4, indicate the shortest wait time routing algorithm was consistently preferred over the closest station first algorithm under the initial simulation parameters. The paired differences in Table 4.1, were positive for every seed, and the confidence interval lay entirely above zero Table 4.2. From this it can be inferred that the observed improvement of the routing policy was stable rather than a result of seed-specific variation. This is further supported through the logical conclusion that when larger charge amounts are demanded, the delays arising from queueing are more influential on the overall wait that any additional delay incurred from travel time. In real-world settings, this behavior would be expected in mixed environments where moderate charging demands are common and station spacing is neither extremely dense nor widely dispersed. Examples of these would be commuter corridors or suburban–highway boundary regions, where vehicles tend to arrive with lower battery levels and undergo longer charging sessions than in dense urban areas but there are still many commuters who charge for short time periods.

5.1.2 Decreasing Minimum Charge Amount

In this scenario, the target charge level is sampled from $U([\mathrm{SoC}_{\mathrm{initial}} + 0.10, ; 0.80])$. Even though the upper bound remained at 80%, minimum charge lowered from 20% to 10% reduced the charge time for some vehicles. The overall affect on the system was a decreased average service time and reduced service time variability. With shorter service times, queues clear faster and total delay is affected more by travel time than by queueing. As shown in **??** and Figure 4.2, the paired differences are negative and the confidence interval lies below zero. As a result, routing to the closest station performs better. This points to routing cars with a closest station first policy being most beneficial in locations where users request smaller charge increments such as city locations where charge station availability is dense, drivers are making fewer trips, and when drivers cannot stay long such as during the work day.

Decreasing the Minimum Balk Battery Percentage

Lowering the threshold for balking increased balking but also allowed more cars to avoid joining queues, which led to decreased wait times for vehicles that did enter the system which is consistent with queueing theory principles [5]. The positive paired differences in Table 4.6 and the confidence interval in Table 4.5 indicate that the shortest wait time routing policy continued to outperform the closest station first policy under this setting. The mean paired difference was slightly lower than in the original run, but the interval lay entirely above zero, so an advantage for the shortest wait time policy was still present.

5.2 Engineering Implications

The results are relevant for city design and indicate that routing strategies for charging infrastructure should consider the operating environment. For example, in areas where vehicles typically require larger charging increments and where larger traffic is observed, routing based on expected wait times would provide the most benefit. In contrast, environments characterized by shorter charging sessions and high station density would be better served by routing policies that minimize travel time. The balking scenario suggests that system performance can be improved by enabling or encouraging drivers with adequate state of charge to defer charging when wait times are long, in an effort to reduce congestion. For engineering design, this implies that a single routing strategy may not be optimal for all contexts, and that adaptive or environment-specific routing approaches should be considered.

6 Conclusion

6.1 Summary of Findings

The results of this study showed that routing performance was determined by the interaction between charging duration, queueing pressure, and station spacing. When charging sessions were long and variable, queueing delay was the dominant contributor to overall waiting time, and the shortest wait time routing policy produced the lowest delays. When the minimum charge requirement was reduced and charging durations became shorter, queues cleared more quickly and total delay became influenced more by travel distance than by queueing, which resulted in better performance by the closest station first policy. Reducing the balk threshold further decreased system load and improved waiting times for both routing policies. In contrast, dense urban environments with high station availability and shorter charging durations do not create the same queueing dynamics, so the advantage associated with selecting the station with the shortest expected wait is reduced.

6.2 System Improvement Recommendations

It was observed that routing effectiveness depended on the operating environment, suggesting that routing systems should be adapted rather than fixed. In settings with moderate or high charging demand and longer service durations, routing based on expected wait time would be expected to yield better performance. In dense urban regions with short charging sessions, routing based on travel distance would be expected to be more effective. Improvements in system performance could be achieved by incorporating driver behaviour, such as balking or flexibility in charging time, into routing frameworks. This would allow peak congestion to be reduced by enabling drivers with sufficient state of charge to postpone charging. More robust performance could be achieved through adaptive routing systems in which routing decisions are updated based on local demand conditions, time of day, and real-time station status.

6.3 Limitations

1. **Objective limited to minimizing wait time.** The simulation was designed to minimize waiting time but did not incorporate cost considerations. In practice, a small increase in waiting time may be outweighed by the additional travel distance, battery consumption, or monetary cost associated with driving to a farther station.
2. **Limited station set.** Only three stations in Victoria were modeled, and they were assumed to function identically. Real charging networks contain stations with different charger types, power ratings, spatial distributions, and reliability characteristics.
3. **Assumption of uniform EV distribution.** Vehicles were assumed to be evenly dispersed across the simulated region. Real-world EV traffic patterns are influenced by commuter flows, land use, and clustering, and therefore do not follow a uniform spatial distribution.
4. **Constant interarrival rate.** A constant arrival rate was assumed, although BC Hydro survey data indicate that usage of public fast chargers is concentrated between 07:00 and 16:00. Time-varying arrival rates would therefore provide a more realistic representation of demand.

Bibliography

[1] BC Hydro, "From 1 to 6.6 million: Electric vehicles by the numbers," BC Hydro, Tech. Rep., Jan. 2024, [Online]. Available: https://www.bchydro.com/news/conservation/2024/ev-by-the-numbers.html. [Accessed: Feb. 9, 2025].

[2] BC Hydro, *Charging help centre*, https://www.bchydro.com/powersmart/electric-vehicles/public-charging/charging-help-centre.html, Accessed: 2025-12-11, 2025. [Online]. Available: https://www.bchydro.com/powersmart/electric-vehicles/public-charging/charging-help-centre.html.

[3] BC Hydro, "Public electric vehicle charging service rates evaluation report: Year one," BC Hydro, Tech. Rep. Year One Evaluation Report, Aug. 2025, [Online]. Available: https://www.bchydro.com/content/dam/BCHydro/customer-portal/documents/corporate/regulatory-planning-documents/regulatory-filings/reports/2025-08-29-bchydro-public-ev-charging-service-rates-evaluation-report-year-1.pdf. [Accessed: Feb. 9, 2025].

[4] B. J. Kirby and J. D. Kueck, "Spinning reserve provided from responsive loads," Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-2004/20, Mar. 2004, [Online]. Available: https://www.osti.gov/servlets/purl/2580296. [Accessed: Feb. 9, 2025]. [Online]. Available: https://www.osti.gov/servlets/purl/2580296.

[5] A. Pazgal, "Comparison of customer balking and reneging behavior to queueing theory predictions: An experimental study," *Operations Research Letters*, vol. 36, no. 2, pp. 186–190, 2008, [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0305054806003066. [Accessed: Feb. 9, 2025]. DOI: 10.1016/j.orl.2006.09.001. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0305054806003066.

[6] P. Pourvaziri, S. Golmohammadi, S. Razavi, and S. Yazdani, "Planning of electric vehicle charging stations: An integrated model for location, capacity and pricing," *Journal of Cleaner Production*, vol. 421, p. 140716, 2024. DOI: 10.1016/j.jclepro.2024.140716. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1366554524001595.

A

Appendix A

```python
import numpy as np

from station_meta import Station_Meta
from typing import Iterable
from charging_station import Charging_Station
from constants import ENERGY_CONSUMPTION_RATE, BATTERY_CAPACITY, MIN_BATTERY_THRESHOLD,
↪ BATTERY_MIN, BATTERY_MAX, TARGET_MAX_FINAL_BATTERY, MIN_CHARGE_AMOUNT, X_MIN,
↪ X_MAX, Y_MIN, Y_MAX, SPEED_KM

class Car:
    position: tuple[float, float] # (x,y) coordinate
    battery_level_initial: float # initial battery level (%)
    soc_after_drive: float | None # estimated SoC (%) after driving to routed station
    system_arrival_time: float  # time car was spawned in the system
    reachable_stations: list[Station_Meta] # list of reachable station meta objects
    time_charging: float | None # time spent charging (minutes)
    target_charge_level: float # target charge level (%)
    routed_drive_time: float | None # drive time to routed station (minutes)
    routed_arrival_time: float # arrival time at station
    time_in_queue: float # time spent in queue (minutes)
    total_time_in_system: float | None # total time in system (minutes)

    def __init__(self, system_arrival_time: float, stations:
    ↪ Iterable[Charging_Station]):
        self.system_arrival_time = system_arrival_time
        self.position = self._set_position()
        self.battery_level_initial = self._set_battery_level_initial()
        self.target_charge_level = self._set_target_charge_level()
        self.reachable_stations = self._set_reachable_stations(stations)

        # Updated once car is routed
        self.routed_drive_time = None
        self.routed_arrival_time = 0.0
        self.time_charging = None
        self.total_time_in_system = None
        self.time_in_queue = 0.0

    def _set_position(self) -> tuple[float, float]:
        x = np.random.uniform(X_MIN, X_MAX)
        y = np.random.uniform(Y_MIN, Y_MAX)
        return (x, y) # the car spawn position

    def _set_target_charge_level(self) -> float:
        """
        Sets a target charge level between defined min and max final battery levels.
        In the real world, this simulates a human decision of how much to charge.
        Since the system is simulating routing for example by using an app, the system
        ↪  has no knowledge
        of the target charge level the user is thinking of when they want to charge.
        Because the system has no knowledge of this target charge level, it cannot use
        ↪  it in routing decisions
        and must rely on other metrics such as distance, estimated wait time, etc.

        Returns the target charge level (%).
        """
        return np.random.uniform(self.battery_level_initial + MIN_CHARGE_AMOUNT,
        ↪  TARGET_MAX_FINAL_BATTERY)

    def get_total_time_in_system(self, sim_time: float) -> float:
        """
        Computes the total time the car has been in the system.
        Returns total time in system (minutes).
        """
        return sim_time - self.system_arrival_time

    def _set_battery_level_initial(self) -> float:
        """
        Sets an initial battery level between defined min and max battery levels.
        Returns the initial battery level (%).
        """
        return np.random.uniform(BATTERY_MIN, BATTERY_MAX) # initial battery level (%)

    def _set_reachable_stations(self, stations: Iterable[Charging_Station]) ->
    ↪  list[Station_Meta]:
        """
        Builds the list of reachable stations.

        For each station, compute distance, drive time, and SoC after drive.

        Returns a list of Station_Meta objects only for stations the car can reach.
        """
        reachable_stations = [] # list of station meta objects

        for station in stations:
            # get the estimate soc after drive
            distance_km = self._get_euclidian_to_station(station)
            soc_after_drive = self.get_estimated_soc_after_driving_km(distance_km)

            if soc_after_drive is None:
                continue # car cannot reach this station, check the next station

            # otherwise station is reachable, create station meta object
            drive_time_minutes = distance_km / (SPEED_KM / 60)  # get the drive time in
            ↪  minutes
            station_meta = Station_Meta(
                station, # station object
                distance_km, # eculidian distance from spawn point of car to station
                drive_time_minutes, # drive time from spawn point of car to station
                soc_after_drive # estimated soc after driving to station
            )
            # add to reachable stations list
            reachable_stations.append(station_meta)

        return reachable_stations # return list of station_meta objects

    def get_estimated_soc_after_driving_km(self, distance_km) -> float | None:
        """
        Computes the expected SoC (%) after driving from the EV's current position
        to the specified charging station.

        Returns soc_after_drive, or None if the vehicle cannot reach it the station
        ↪  (SoC would fall below MIN_BATTERY_THRESHOLD).
        """
        # energy used to drive there (one-way)
        energy_used = distance_km * ENERGY_CONSUMPTION_RATE

        # SoC after driving to the station (percent)
        soc_after_drive = self.battery_level_initial - (energy_used / BATTERY_CAPACITY)
        ↪  * 100

        # if the car cannot even physically reach the station
        if soc_after_drive < MIN_BATTERY_THRESHOLD:
            return None
        return soc_after_drive

    def _get_euclidian_to_station(self, station) -> float:
        """
        Compute the Euclidean distance from the cars spawn point to the station
        ↪  specified.

        Returns the distance in kilometers.
        """
        station_position = station.position # (x,y) coordinate of station
        distance_km = np.sqrt((self.position[0] - station_position[0])**2 +
        ↪  (self.position[1] - station_position[1])**2)
        return distance_km

```

Listing A.1: *The car.py class.*

```
import heapq
```

```python
import numpy as np

from routing_policies import RoutingPolicy
from event import EventType
from charging_station import Charging_Station
from car import Car
from routing import Routing

class EV_Charging_System:
    def __init__(self, routing_policy, num_delays_required, seed):
        self.routing_policy = routing_policy
        self.num_delays_required = num_delays_required
        self.num_cars_processed = 0
        self.total_time_in_system = 0.0
        self.total_wait_time = 0.0
        self.total_wait_time_queue = 0.0
        self.total_balking = 0
        self.total_reneging = 0
        self.seed = seed
        np.random.seed(seed)
        self.wait_times = []

        self.mean_interarrival_time = 5
        self.sim_time = 0.0
        self.void_counter = [0, 0, 0]  # List to track cars on the way to each station

        # Event list
        self.event_queue = []   # (time, event_type, payload)

        # Schedule first system arrival
        first_arrival = self.sim_time + self.expon(self.mean_interarrival_time)
        heapq.heappush(self.event_queue,
                       (first_arrival, EventType.ARRIVAL_SYSTEM, None)) # Push event
                       ↪ without routing

        # Stations
        self.stations = [
            Charging_Station(1, [3.62, 2.93], lambda: self.sim_time),   # Belmont park
            ↪ area
            Charging_Station(2, [9.29, 4.91], lambda: self.sim_time),   # Uptown / NE
            ↪ side
            Charging_Station(3, [10.32, 1.74], lambda: self.sim_time),   # West /
            ↪ highway area
        ]

    def timing(self):

        if not self.event_queue:
            raise Exception("Event queue empty - simulation cannot continue.")

        # Pop next min time event
        self.sim_time, self.next_event_type, self.routing =
        ↪  heapq.heappop(self.event_queue)

        # determine if there's a car
        if self.routing and hasattr(self.routing, "car"):
            self.event_car = self.routing.car
        else:
            self.event_car = None

    def arrival_system(self):
        # Schedule next system arrival
        next_arrival = self.sim_time + self.expon(self.mean_interarrival_time)
        heapq.heappush(self.event_queue,
                    (next_arrival, EventType.ARRIVAL_SYSTEM, None))

        # Create the car check if reneging and then routing
        car = Car(system_arrival_time=self.sim_time, stations=self.stations)
        self.reneging()
        routing = Routing(car, self.routing_policy, void_counter=self.void_counter)

        # Actually perform routing and set routed_station
        chosen_station = routing.route()
        routing.routed_station = chosen_station  # (if not set inside route()) follow
        ↪  up

        if routing.routed_station is None:
            self.total_balking += 1
            return

        # print(f"Car {car.battery_level_initial} routed to station
        ↪  {routing.routed_station.station.station_id}")
        station_choice_id = routing.routed_station.station.station_id

        # Select the correct arrival event type based on the id we retrieved from
        ↪  station choice
        station_event = [
            EventType.ARRIVAL_STATION_1,
            EventType.ARRIVAL_STATION_2,
            EventType.ARRIVAL_STATION_3,
        ][station_choice_id - 1]

        # Schedule arrival to the station - this is the time it takes the car to drive
        ↪  there
        heapq.heappush(self.event_queue, (car.routed_arrival_time, station_event,
        ↪  routing))

    def expon(self, mean): # generate exponential random variable
        """
        Generate an exponential random variable with the given mean.

        :param mean: The mean of the exponential distribution.
        :return: A random variable drawn from the exponential distribution.
        """
        return -mean * np.log(np.random.uniform())

    def record_departure(self, car):
        """
        Records the departure of a car from the system, updating statistics.

        :param car: The car that is departing.
        """
        # retrieve the total time in system for this car and add to total
        wait = car.time_in_queue + car.routed_drive_time
        self.wait_times.append(wait)
        self.total_time_in_system += car.get_total_time_in_system(self.sim_time)

        # retrieve the wait time (drive + queue) for this car and add to total
        self.total_wait_time_queue += car.time_in_queue
        self.total_wait_time += (car.time_in_queue + car.routed_drive_time) # total
        ↪  wait time includes drive time
        self.num_cars_processed += 1

    def reneging(self):
        for station in self.stations:
            queue = station.queue
            if len(queue) <= 5:
                continue

            sixth_car = queue[5]
            time_waiting =  self.sim_time - sixth_car.routed_arrival_time
            # If the 6th car has waited too long, it reneges
            if time_waiting > 15:  # minutes
                queue.pop(5)
                self.total_reneging += 1

    def print_results(self):
        """Prints the final simulation results."""
        if self.num_cars_processed > 0:
            avg_time_in_system = self.total_time_in_system / self.num_cars_processed
            avg_wait_time = self.total_wait_time / self.num_cars_processed
        else:
            avg_time_in_system = 0.0
            avg_wait_time = 0.0

        print("\n" + "="*50)
        print("Simulation Results")
        print(f"Total Cars Processed: {self.num_cars_processed}")
        print(f"Total Time in System: {self.total_time_in_system:.2f} minutes")
        print(f"Total Wait Time (incl. drive): {self.total_wait_time:.2f} minutes")
        print("-" * 50)
        print(f"Average Time in System: {avg_time_in_system:.2f} minutes")
        print(f"Average Wait Time (incl. drive): {avg_wait_time:.2f} minutes")
        print(f"Total Balking Events: {self.total_balking}")
        print(f"Total Reneging Events: {self.total_reneging}")
        print(F"Simulation end time: {self.sim_time:.2f} minutes")
        print("="*50)

    def main(self):
        while self.num_cars_processed < self.num_delays_required:
            self.timing() # - to get the next event

            match self.next_event_type:
                # to each event other than system arrival pass the event queue and the
                ↪  car

                case EventType.ARRIVAL_SYSTEM:
                    self.arrival_system()

                case EventType.ARRIVAL_STATION_1:
                    self.stations[0].arrival(self.routing, self.event_queue)

                case EventType.ARRIVAL_STATION_2:
                    self.stations[1].arrival(self.routing, self.event_queue)

                case EventType.ARRIVAL_STATION_3:
                    self.stations[2].arrival(self.routing, self.event_queue)

                case EventType.DEPARTURE_STATION_1_FAST:
                    self.stations[0].departure_fast(self.event_queue)
                    self.record_departure(self.routing)

                case EventType.DEPARTURE_STATION_1_SLOW:
                    self.stations[0].departure_slow(self.event_queue)
                    self.record_departure(self.routing)

                case EventType.DEPARTURE_STATION_2_FAST:
                    self.stations[1].departure_fast(self.event_queue)
                    self.record_departure(self.routing)

                case EventType.DEPARTURE_STATION_2_SLOW:
                    self.stations[1].departure_slow(self.event_queue)
                    self.record_departure(self.routing)

                case EventType.DEPARTURE_STATION_3_FAST:
                    self.stations[2].departure_fast(self.event_queue)
                    self.record_departure(self.routing)

                case EventType.DEPARTURE_STATION_3_SLOW:
                    self.stations[2].departure_slow(self.event_queue)
                    self.record_departure(self.routing)

        self.print_results()

if __name__ == "__main__":
    sim = EV_Charging_System(RoutingPolicy.CLOSEST_STATION_FIRST, 100000, 100)
    sim.main()
```

Listing A.2: *The system.py class.*

```
from car import MIN_BATTERY_THRESHOLD
from station_meta import Station_Meta
from routing_policies import RoutingPolicy
from event import EventType
from constants import MAX_QUEUE_LENGTH, TIME_FACTOR, BALK_BATTERY_LEVEL

class Routing:
    def __init__(self, car, routing_policy: RoutingPolicy, void_counter: list[int]):
        self.car = car # the car being routed
        self.routing_policy = routing_policy # the routing policy to use, this is an
        ↪  ENUM and chosen in system.py
        self.routed_station = None
        self.void_counter = void_counter # store the void counter for use in routing
        ↪  decisions

        # mapping of policies to functions, function names are defined in ENUM, the car
        ↪  will routing will call the correct function based on the policy chosen
        self._policy_map = {
            RoutingPolicy.CLOSEST_STATION_FIRST: self._closest_station_first,
            RoutingPolicy.SHORTEST_ESTIMATED_WAIT: self._shortest_estimated_wait
        }

    def route(self) -> Station_Meta | None:
        """
        Docstring for route which selects the routing policy to use and calls the
        ↪  appropriate function

        :param self: Description
        :rtype: Station_Meta | None
        """
        try:
            # look up the correct function to call base on the policy provided
            policy_fn = self._policy_map[self.routing_policy]
        except KeyError: # policy not found
            raise ValueError(f"Unknown routing policy: {self.routing_policy}")

        return policy_fn()  # if the policy exists call it

    def _closest_station_first(self) -> Station_Meta | None:
        """
        Implements the closest station first routing policy.

        :param self: Description
        :rtype: Station_Meta | None
        """
        # get the list of reachable stations for this car
        stations: list[Station_Meta] = list(self.car.reachable_stations)
        while stations: # while there are still stations to check
            closest = min(stations, key=lambda s: s.drive_time_minutes) # get the
            ↪  closest station based on drive time
```

```python
            # if the queue length at the closest station is acceptable or the car is low on battery choose it
            if (verify := self._verify_station_(closest, closest.soc_after_drive, void_counter=self.void_counter)) != -1: # if we are allowed to route to the station
                # Apply routing decision cleanly
                self._apply_routing_decision(closest, void_counter=self.void_counter)
                return closest
            else:
                stations.remove(closest) # remove and check next closest

        return None # if we reach here no stations were suitable and None were chosen, car balks

    def _shortest_estimated_wait(self) -> Station_Meta | None:
        stations: list[Station_Meta] = list(self.car.reachable_stations)
        wait_times = {}

        for station_meta in stations:
            drive_time = station_meta.drive_time_minutes

            # compute estimated queue wait (keep raw int result separate to avoid reassigning int to float)
            wait_time_ahead_raw = self._verify_station_(station_meta,
                                                        station_meta.soc_after_drive,
                                                        void_counter=self.void_counter)

            # if station is invalid (<0) skip it
            if wait_time_ahead_raw < 0:
                continue

            # ensure we use a float for the scaled wait time to avoid type issues
            wait_time_ahead = wait_time_ahead_raw * TIME_FACTOR

            total_est = wait_time_ahead + drive_time
            wait_times[station_meta] = total_est

        if not wait_times:
            return None

        # pick minimum
        chosen = min(wait_times, key=lambda s: wait_times[s])
        self._apply_routing_decision(chosen, void_counter=self.void_counter)
        return chosen

    def _verify_station_(self, station_meta: Station_Meta, soc_after_drive: float, void_counter: list[int]) -> int:
        """
        Helper for eliminating the stations that have too long of a queue from consideration
        Returns the queue length >=0 if valid, returns -1 if false
        """
        q_len = station_meta.get_effective_queue_length(void_counter=void_counter) # get the number of cars in the queue and on the way to the station
        if q_len > MAX_QUEUE_LENGTH and soc_after_drive > BALK_BATTERY_LEVEL: # if the queue length is too long and the car has enough battery to consider other stations
            return -1 # if we shouldn't consider this station return -1
        return q_len # if we can consider this stations return the queue length

    def _apply_routing_decision(self, chosen: Station_Meta, void_counter: list[int]) -> None:
        """
        Updates all car fields after selecting a routing destination.
        """
        void_counter[chosen.get_station_id() - 1] += 1 # increment the void counter for the chosen station indicating a car is somewhere in the simultion
        self.car.routed_station = chosen # update routed station with station_meta object
        self.car.routed_arrival_time = self.car.system_arrival_time + chosen.drive_time_minutes
        self.car.soc_after_drive = chosen.soc_after_drive
        self.car.routed_drive_time = chosen.drive_time_minutes
```

Listing A.3: *The routing.py class.*

```python
from __future__ import annotations
from typing import TYPE_CHECKING, Callable, List, Tuple

if TYPE_CHECKING:
    from car import Car

import heapq

from event import EventType
from constants import (
    BATTERY_CAPACITY,
    FAST_CHARGER_POWER_KW,
    SLOW_CHARGER_POWER_KW
)

class Charging_Station:
    station_id: int
    position: Tuple[float, float]
    sim_time: Callable[[], float]

    fast_charger_status: int
    slow_charger_status: int

    current_estimated_wait_time: float
    queue: List["Car"]

    mean_fast_service: float
    mean_slow_service: float

    arrival_event: EventType
    depart_fast_event: EventType
    depart_slow_event: EventType

    def __init__(self, station_id: int, position: Tuple[float, float], sim_time:
    ↪  Callable[[], float]):
        self.station_id = station_id
        self.position = position
        self.sim_time = sim_time   # lambda returning current sim time

        self.fast_charger_status = 0
        self.slow_charger_status = 0

        self.current_estimated_wait_time = 0.0 # in minutes

        self.queue = []   # Store Cars that are waiting to charge

        self.mean_fast_service = 0.5
        self.mean_slow_service = 1.0

        # event type mapping
        self.arrival_event = {
            1: EventType.ARRIVAL_STATION_1,
            2: EventType.ARRIVAL_STATION_2,
            3: EventType.ARRIVAL_STATION_3
        }[station_id]

        self.depart_fast_event = {
            1: EventType.DEPARTURE_STATION_1_FAST,
            2: EventType.DEPARTURE_STATION_2_FAST,
            3: EventType.DEPARTURE_STATION_3_FAST
        }[station_id]

        self.depart_slow_event = {
            1: EventType.DEPARTURE_STATION_1_SLOW,
            2: EventType.DEPARTURE_STATION_2_SLOW,
            3: EventType.DEPARTURE_STATION_3_SLOW
        }[station_id]

    # car is passed in from system ( so lowest time in event queue)
    def arrival(self, routing, event_queue: list[tuple[float, EventType, "Car"]]):
        idx = routing.routed_station.get_station_id() - 1
        routing.void_counter[idx] -= 1 if routing.void_counter[idx] > 0 else None
        car = routing.car
        # if both chargers busy - join queue
        if self.fast_charger_status == 1 and self.slow_charger_status == 1:
            self.queue.append(car)
            return

        # is the fast charger free?
        if self.fast_charger_status == 0:
            self.fast_charger_status = 1

            # compute service time and schedule departure
            service_time = self.compute_charge_time(car.target_charge_level,
            ↪  car.soc_after_drive, FAST_CHARGER_POWER_KW)
            car.time_charging = service_time
            depart_time = self.sim_time() + service_time

            #schedule departure event
            heapq.heappush(event_queue,
                (depart_time, self.depart_fast_event, car))
            return

        # is the slow charger free?
        if self.slow_charger_status == 0:
            self.slow_charger_status = 1

            # compute service time and schedule departure
            service_time = self.compute_charge_time(car.target_charge_level,
            ↪  car.soc_after_drive, SLOW_CHARGER_POWER_KW)
            car.time_charging = service_time # set the car's service time
            depart_time = self.sim_time() + service_time # compute departure time

            # Schedule thedeparture event
            heapq.heappush(event_queue,
                (depart_time, self.depart_slow_event, car))

            return

    # the next event was a departure from the fast charger
    def departure_fast(self, event_queue):

        # queue empty?
        if len(self.queue) == 0:
            self.fast_charger_status = 0
            return

        # take next car from queue
        next_car = self.queue.pop(0)
        self.fast_charger_status = 1
        next_car.time_in_queue = self.sim_time() - next_car.routed_arrival_time  # set
        ↪  the car's time in queue

        # compute service time and schedule departure
        service_time = self.compute_charge_time(next_car.target_charge_level,
        ↪  next_car.soc_after_drive, FAST_CHARGER_POWER_KW)
        next_car.time_charging = service_time  # set the car's service time
        depart_time = self.sim_time() + service_time

        heapq.heappush(event_queue,
            (depart_time, self.depart_fast_event, next_car))


    # slow charger
    def departure_slow(self, event_queue):
        if len(self.queue) == 0:
            self.slow_charger_status = 0
            return

        next_car = self.queue.pop(0)
        self.slow_charger_status = 1
        next_car.time_in_queue = self.sim_time() - next_car.routed_arrival_time

        # compute service time and schedule departure
        service_time = self.compute_charge_time(next_car.target_charge_level,
        ↪  next_car.soc_after_drive, SLOW_CHARGER_POWER_KW)
        next_car.time_charging = service_time  # set the car's service time
        depart_time = self.sim_time() + service_time

        heapq.heappush(event_queue,
            (depart_time, self.depart_slow_event, next_car))

    def compute_charge_time(self, target_charge_level, soc_after_drive, charge_rate_kw:
    ↪  float) -> float:
        """
        Computes the estimated charge time (in hours) for the given car
        based on its target charge level. Affected by the service rate of the charger
        ↪  it gets assigned to.
        Args:
            target_charge_level (float): The target charge level for the car.
            battery_level_initial (float): The initial battery level of the car.
            charge_rate_kw (float): The charging rate of the charger in kW.
            Returns:
            float: Estimated charge time in minutes
        """
        soc_diff = (target_charge_level - soc_after_drive) / 100.0 # fraction of
        ↪  battery to charge
        time_in_minutes = ((soc_diff * BATTERY_CAPACITY) / charge_rate_kw) * 60.0
        return time_in_minutes # return the service time in minutes

```

Listing A.4: *The charging_station.py class.*

```
from charging_station import Charging_Station

# Metadata about a station for a specific car
# This includes distance, drive time, and estimated SoC after driving there
# Used in routing decisions
# the actual routed station object is not stored here and only in the car object
class Station_Meta:
    station: Charging_Station           # the actual station object
    distance_km: float                  # the euclidean distance for the ev to the
    ↪  station
    drive_time_minutes: float           # travel time
    soc_after_drive: float              # SoC (%) after getting there

    def __init__(self, station, distance_km, drive_time_minutes, soc_after_drive):
        self.station = station          # the actual station object this meta refers to
        self.distance_km = distance_km  # the euclidean distance for the ev to the
        ↪ station
        self.drive_time_minutes = drive_time_minutes # the travel time to the station
        ↪ in minutes for the ev
        self.soc_after_drive = soc_after_drive # SoC (%) after the ev gets there

    def get_drive_time_minutes(self) -> float:
        return self.drive_time_minutes  # return the drive time in minutes

    def get_station_id(self) -> int:
        return self.station.station_id  # return the station id

    def get_effective_queue_length(self, void_counter) -> int:
        return len(self.station.queue) + void_counter[self.station.station_id - 1]  #
        ↪ return the current queue length at the station and the current length of
        ↪ void cars for the station
```

Listing A.5: *The station_meta.py class.*

```
from enum import Enum

class RoutingPolicy(str, Enum):
    CLOSEST_STATION_FIRST = "closest_station_first"
    SHORTEST_ESTIMATED_WAIT = "shortest_estimated_wait"
```

Listing A.6: *An enum for defining the available routing policy names.*

```
SLOW_CHARGER_POWER_KW = 4.8    # BC Hydro Level 2 ~ 4.8 kW
FAST_CHARGER_POWER_KW = 200    # BC Hydro Fast Charger ~ 200 kW Level 3

MAX_QUEUE_LENGTH = 10  # maximum acceptable queue length
TIME_FACTOR = 15.0  # factor to estimate wait times in queue

# Constraints for the maximum and minimum battery levels for generated cars
BATTERY_MIN = 20 # 30% OST study recommended this assumption for the equation used for
↪ service time
BATTERY_MAX = 60 # 60%
BALK_BATTERY_LEVEL = 25 # if battery level is below this, car will balk if no stations
↪ are reachable (%)
TARGET_MAX_FINAL_BATTERY = 80    # max % they will charge up to
ENERGY_CONSUMPTION_RATE: float= 0.20# (kWh/km)
BATTERY_CAPACITY: float = 75.0 # (kWh)
MIN_BATTERY_THRESHOLD = 19 # minimum battery level to consider driving to a station (%)
MIN_CHARGE_AMOUNT = 20  # minimum amount to charge (%)

X_MIN = 0.0 # minimum x coordinate for simulation area
X_MAX = 13 # maximum x coordinate for simulation area
```

```
Y_MIN = 0.0 # minimum y coordinate for simulation area
Y_MAX = 7.5 # maximum y coordinate for simulation area
SPEED_KM = 30 # average speed in km/h
```

Listing A.7: *The constants used throughout the system.*

```
from enum import Enum

class EventType(Enum):
    NONE = 0
    ARRIVAL_SYSTEM = 1
    ARRIVAL_STATION_1 = 2
    ARRIVAL_STATION_2 = 3
    ARRIVAL_STATION_3 = 4
    DEPARTURE_STATION_1_FAST = 5
    DEPARTURE_STATION_1_SLOW = 6
    DEPARTURE_STATION_2_FAST = 7
    DEPARTURE_STATION_2_SLOW = 8
    DEPARTURE_STATION_3_FAST = 9
    DEPARTURE_STATION_3_SLOW = 10
```

Listing A.8: *The event Enum.*

```
import csv
from system import EV_Charging_System
from routing_policies import RoutingPolicy

SEEDS = [3, 200, 303, 670, 1000]
POLICIES = [
    RoutingPolicy.CLOSEST_STATION_FIRST,
    RoutingPolicy.SHORTEST_ESTIMATED_WAIT,
]
NUM_DELAYS_REQUIRED = 100000
OUTPUT_FILE = "simulation_results.csv"

def run_replications():

    table = [{"seed": seed} for seed in SEEDS]

    for policy in POLICIES:
        pol_name = policy.name.lower()

        for i, seed in enumerate(SEEDS):
            sim = EV_Charging_System(
                policy,
                num_delays_required=NUM_DELAYS_REQUIRED,
                seed=seed
            )
            sim.main()

            wait_times = sim.wait_times
```

```python
            avg_wait = sum(wait_times) / len(wait_times) if wait_times else 0

            table[i][pol_name] = avg_wait

    fieldnames = ["seed"] + [p.name.lower() for p in POLICIES]
    with open(OUTPUT_FILE, "w", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(table)

if __name__ == "__main__":
    run_replications()

```

Listing A.9: *The script used for running the simulation runs.*

```python
import csv
import math
import matplotlib.pyplot as plt
from scipy.stats import t

INPUT_FILE = "simulation_results.csv"

def load_results():
    seeds = []
    p1 = []
    p2 = []
    with open(INPUT_FILE, "r") as f:
        reader = csv.DictReader(f)
        for row in reader:
            seeds.append(int(row["seed"]))
            p1.append(float(row["closest_station_first"]))
            p2.append(float(row["shortest_estimated_wait"]))
    return seeds, p1, p2

def compute_crn_confidence_interval(p1, p2):
    R = len(p1)
    D = [p1[i] - p2[i] for i in range(R)]

    D_bar = sum(D) / R
    S2_D = sum((d - D_bar)**2 for d in D) / (R - 1)
    se = math.sqrt(S2_D / R)

    # 95% CI
    t_val = t.ppf(1 - 0.025, R - 1)
    H = t_val * se

    return D, D_bar, H

def plot_differences(seeds, D, D_bar, H):
    plt.figure()
    plt.axhline(D_bar, linestyle="--", label=f"Mean diff = {D_bar:.3f}")
    plt.axhline(D_bar + H, linestyle=":", label=f"95% CI = [{D_bar - H:.3f}, {D_bar +
    ↪  H:.3f}]")
    plt.axhline(D_bar - H, linestyle=":")

    plt.scatter(seeds, D)
    plt.plot(seeds, D)

    plt.title("Paired Differences (CRN)")
    plt.xlabel("Seed")
    plt.ylabel("Difference in Avg Wait (CSF - SEW)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    seeds, p1, p2 = load_results()
    D, D_bar, H = compute_crn_confidence_interval(p1, p2)

    print("\nPaired Differences:")
    for s, d in zip(seeds, D):
        print(f"Seed {s}: {d:.4f}")

    print(f"\nMean Difference (D) = {D_bar:.4f}")
    print(f"Half-width (H) = {H:.4f}")
    print(f"95% CI = [{D_bar - H:.4f}, {D_bar + H:.4f}]")

    plot_differences(seeds, D, D_bar, H)

```

Listing A.10: *The script responsible for the confidence interval determination.*