Bus Charging Schedule Simulated Annealing with MILP Constraints

Alexander Brown

October 3, 2022

Contents

1 Introduction

This document outlines the simulated annealing (SA) approach to the bus charging scheduling problem utilizing Mixed Integer Linear Programming (MILP) constraints as the method of determining feasible charging schedules. The problem statement is as follows: given a set of routes for a fleet of Battery Electric Buses (BEB), generate an optimal charging schedule to minimize the consumption cost (amount of electricity used over a certain time) and the demand cost (rate at which electricity is being used) within the constraints that the buses must maintain sufficient charge to complete the working day and do not have any delays in their respective routes.

Simulated Annealing (SA) shall be introduced and utilized as a means of finding the global optimum. The SA algorithm shall be constrained by a set of Mixed Integer Linear Program (MILP) constraints derived from the Position Allocation Problem (PAP). These constraints are set in place to ensure validity of the proposed charging schedules. A set of objective functions describing consumption cost and demand cost, as stated above, shall be minimized to reduce power consumption and total cost of using the BEBs.

2 Problem Description

Given a set of bus arrivals to a charging station $i = \{1, ..., I\} \in \mathbb{Z}$ with a set of chargers to be queued $q = \{1, ..., Q\} \in \mathbb{Z}$ where the bus is identified by an identification number $b = \{1, ..., B\} \in \mathbb{Z}$. Each bus arrival, i, can be represented by a the tuple: $(b_i, a_i, e_i, u_i, d_i, v_i, \eta_i)$, in which the ordered elements denote the bus identification number arrival time to the station, departure time from the station, initial charging time, charge end time, the charger queue for the bus to be placed into, and the initial State Of Charge (SOC).

It is assumed that each visit occurs over the time horizon $[T_1, T_2]$. The set of all arrivals is represented by the set $\mathbb{I} = \{(b_i, a_i, e_i, u_i, d_i, v_i, \eta_i) : b_i \in B, a_i, e_i, u_i, d_i \in [T_1, T_2], v_i \in Q\}$. The concept of "arrivals" in derived from the PAP [7]. This idea of arrivals is useful in the sense that it is easy to describe the state of any arbitrary arrival; however, it also assumes that each arrival is unique (i.e. no two bus arrives twice) therefore a system must be put in place to track each bus over each arrival. That is why a bus identifier is placed in the tuple, and in that way each bus is linked over each arrival.

For each arrival a bus must be placed in a singular queue, $v_i \in Q$. The bus is not allowed to change queues mid-charge. The amount of time the bus is allowed to charge is dictated by the scheduled arrival time and required departure time, $[a_i, e_i]$. Although an arrival must be placed in a queue, if a bus does not require much charge, or none at all, partial charges, or no charging, is allowed. It is not allowed for the bus to charge over its battery capacity limit. The battery charging rate is modeled as linear, which remains accurate up to about an SOC of 80% charge [5].

Each bus arrival, with the exception of the first and last arrival for each bus, has a paired "route" that the bus must perform. This route, as one would expect, cause the bus to discharge by some certain amount. This paper assumes an average discharge over a period of time allowing an estimation to be calculated for each route, Δ_i. The charge supplied while at the station is assumed to supply enough charge for each route (battery charge does not deplete to zero) with an additional battery capacity percentage, m, acting as a safety factor.

The scheduler's task shall be to schedule the set of arrivals $\mathbb{I}$ to fulfill the minimum charge requirements over the time horizon $[T_1, T_2]$ as well as minimize the demand cost, particularly through peak periods, as well as minimize over the consumption cost. The functions and constraints are discussed in further detail in section subsection 3.2.

3 Optimization Problem

This sections introduces the problem in the form of the objective functions as well MILP constraints. The objective function is required to allow comparisons between candidate solutions. In the context of this formulation, the objective function is broken down into two major components as alluded in the introduction: consumption cost and demand cost. The constraints ensure that candidate solutions are in the feasible region. They are composed of a series of equation defined by decision variables which are unknown variables that are manipulated in the attempt to optimize the objective functions and input variables predefined input variables that are assumed to be known. Furthermore, the decision variables have components that are directly and indirectly manipulated.

3.1 Parameter Definitions

This section defines the input variables and decision variables used by the system. The input variables are the parameters that are assumed to be known prior to optimizing the system. The decision variables are the values that the SA algorithm has the freedom to manipulate. The values produced by the SA algorithm will be interpreted as a candidate charging solution. This is further described in section 4.

Table 1: Table of variables used in the paper.

Variable	**Description**
Input constants	
C	Penalty method gain factor
B	Number of buses in use
I	Number of total visits
$J(v, u, d)$	Objective function
K	Local search iteration amount
Q	Number of chargers
$\mathcal{T}$	Time horizon
T	Temperature
Input variables	
Δ_i	Discharge of visit over route i
α_b	Initial charge percentage time for bus b
β_i	Final charge percentage for bus i at the end of the time horizon
δ_i	Discharge rate for vehicle i per mile
ϵ_q	Cost of using charger q
κ_b	Battery capacity for bus b
ρ_i	Route distance after visit i
ξ_i	Value indicating the next index visit i will arrive
a_i	Arrival time of visit i
b_i	ID for bus visit i
e_i	Time visit i must exit the station
k	Local search iteration k
m	Minimum charge percentage allowed for each visit
r_q	Charge rate of charger q
Decision Variables	
η_i	Initial charge for visit i
ψ_i	Binary term to enable/disable charge penalty for visit i
d_i	Detach time from charger for visit i
$p_{dem}(t)$	Demand cost
s_i	Amount of time spent on charger for visit i (service time)
u_i	Initial charge time of visit i
v_i	Assigned queue for visit i

3.1.1 Input Variables

The input values of any MILP system are defined prior to the solving of the system. They define initial conditions, known state properties, etc. Roughly following the order in Table 1, each variable will be introduced.

Δ_i is the amount power required to complete the bus route after visit i. Because there is no route after the last visit, $\Delta_I = 0$. The discharge for visit i is defined by equation Equation 1.

$$\Delta_i = \delta_i * \rho_i \tag{1}$$

Where δ_i is the amount of energy consumed by the bus per mile and ρ_i is the route mileage after visit i. As discussed before, since there is no route after the last visit $\rho_I = 0$. α_b is the initial SOC percentage of bus b at the beginning of the working day. The initial SOC for bus b can be represented as

$$\eta_1 = \alpha_b * \kappa_b. \tag{2}$$

Where κ_b is the battery capacity for bus b, η_1 is the initial charge for bus visit 1. η_i will be further discussed in subsubsection 3.1.2. ϵ_q is the cost for assigning a charger to queue q. This parameter is utilized by the objective function and is further discussed in subsection 3.2. ξ_i represents the next arrival index for bus b_i. In other words, given a set of bus visit IDs $b = \{1, 2, 3, 1\}$, $\xi_1 = 4$. a_i and e_i are the arrival and departure times of bus visit i to the station, respectively. k represents the local iteration search for the SA algorithm. This is further discussed in section 4. Lastly, r_q represents the rate of charge for the charger in queue q. As will be discussed in subsection 3.2, fast chargers and slow chargers relate to high and low costs, ϵ_q, respectively.

3.1.2 Decision Variables

Decision variables are the defined by the optimizer and are therefore unknown prior to running the optimization algorithm. In this case the optimizer is SA. Once SA has been ran and each of the decision variables have been specified, the fitness of the solution is defined by the objection functions outlined in subsection 3.2. The variables will be broken into two sections: direct and indirect decision variables. Decision variables that are direct are values that the system has direct control over and indirect variables are those that are influenced by the direct.

Direct Decision Variables Decision variables that are direct are variables that can be immediately chosen by SA. The first decision variable introduced is ψ_i. This value is a boolean decision variable, $\psi_i \in \{0, 1\}$, that either enables of disables the charge penalty defined in subsection 3.2. The next two variables are u_i and d_i. They represent the initial and final charging times. These values must remain within range of the arrival time and departure time for visit i, $[a_i, e_i]$. The last direct decision variable is the queue that bus visit i can be placed in to charge, $v_i \in q$.

Indirect Decision Variables Indirect decision variables are variables that are dependent on direct decision variables. For example η_i is the initial charge for visit i. These variables are chained together per bus by using the bus identifier, b, and next index, ξ_i. The initial charges must be chained so that the battery charge can be calculated per bus as it is charged and discharged over each visit, $[u_i, d_i]$. p_{dem} is the demand cost of the overall charging schedule. It is calculated at after all the decision variables have been assigned. This is further described in subsection 3.2.

3.2 Objective Function

The objective function is used to compare the fitness of different candidate solutions against one another. This objective function takes in a set input variables and decision variables to calculate some value of measure. The calculated objective function value can either be maximized or minimized. The desired option is dependent on the problem to be solved as well as the formulation of said objective function. Let J represent the objective function. The objective function for this problem has four main considerations: charger assignment, consumption cost, demand cost, peak hours, and sufficient charge.

Suppose the objective function is of the form $J = AC(u_i, d_i, v_i, \eta_i) + PC(u_i, d_i, v_i)$. $AC(u_i, d_i, v_i, \eta_i)$ is the assignment cost, and $PC(u_i, d_i, v_i)$ is the power usage cost. The assignment cost represents the costs of assigning a bus to a particular queue as well as the chosen charging period, $[u_i, d_i]$ as shown in Equation 3.

$$AC(u_i, d_i, v_i, \eta_i) = \sum_{i=1}^{I} \epsilon_{v_i}(d_i - u_i) + \frac{1}{2} C\psi_i(\eta_i - m\kappa_i)^2 \tag{3}$$

Where $v_i \in q$ is the charger index, u_i is the initial charge time, d_i is the detach time for visit i, ψ_i is a binary decision variable, m is the minimum charge percentage allowed, κ_i is the battery capacity for visit i, and η_i is the initial charge for visit i. The first term in the summation represents the calculation of the cost for assigning a bus to queue q. The second term is the penalty function that is either enabled or disabled by ψ_i which is discussed in subsection 3.3. This form is the most common form that penalty methods are found in [6]. Note that the variables ψ_i and η_i are both decision variables that are being multiplied together. This is called a bilinear term. Using a traditional MILP solver, this would require linearization [9]; however, because SA handles nonlinearities easily these bilinear terms will be ignored [8].

The power cost can begin to be de,fined with the consumption cost:

$$PC(u, d, v) = DemandCost(schedule) + \sum_{i=1}^{I} ConsumptionCost(v_i, u_i, d_i)$$

where $ConsumptionCost(v_i, u_i, d_i)$ returns the energy in KWH given the charger index v_i and time spent on the charger d_i as shown in Algorithm 1.

Algorithm: ConsumptionCost
Input: Charger assignment, start charge time, end charge time: (v, u, i)
Output: Consumption cost
begin
| **return** $r[v_i](d_i - u_i)$
end

Algorithm 1: Method describing the consumption cost for a single visit

Peak 15 should also be taken into consideration. Peak 15 is defined as:

$$p_{15}(t) = 1/15 \int_{t-15}^{t} p(\tau) d\tau$$

which represents the energy used over the last 15 minutes. Because worst case must be assumed to always ensure enough power is supplied

$$p_{max}(t) = \max_{\tau \in [0,t]} p_{15}(\tau)$$

Which retains the largest p_{15} found. The demand charge is then determined by

$$p_{dem}(t) = \max(p_{fix}, p_{max}(t))s_r$$

where s_r is the demand rate. Which, again, retains the largest p_{15} value with a starting, fixed value of p_{fix}. To calculate this numerically, an integration algorithm is required to iteratively calculate the $p_{15}(t)$. In turn, $p_{dem}(T)$ can be defined. This process is defined in Algorithm 2.

```
Algorithm: DemandCost
Input: Candidate solution: (schedule)
Output: Demand cost: (p-dem)
begin
    p15 ← ∅;
    for dt ← 0 to T do
        Union(p15, Integrate(schedule,(dt,dt+15)))
    end
    p-old ← p-new ← p-dem ← p-fix;
    foreach element p in p15 do
        p-old ← p-new;
        p-new ← p;
        if p-new > p-old then
            p-dem ← p-new;
            p-old ← p-new;
        end
    end
    return p-dem
end
```

Algorithm 2: Algorithm to calculate the demand cost.

From this we can write:

$$PC(u, d, v) = DemandCost(schedule) + \sum_{i=1}^{I} ConsumptionCost(v_i, u_i, d_i)$$

3.3 Constraints

Now that a method of calculating the fitness of a schedule has been established, a method for determining the feasibility of a schedule must be established. Feasible schedules require

- No overlap in time
- No overlap in space
- Bus receives enough charge
- Bus is not overcharged
- Departs on time

These set of requirements can be summarized by the constraints that follow:

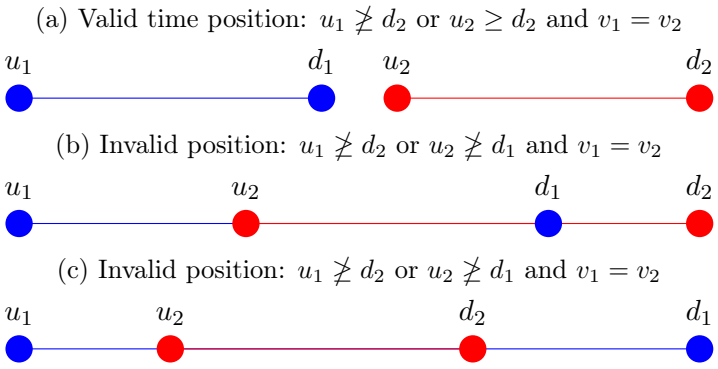

Figure 1: Set of possible collisions between two buses in the same queue.

$(u_i \geq d_j$ or $u_j \geq d_i)$ and $v_i = v_j$	Valid queue position/time
$\Delta_i = \delta_i(a_{\xi_i} - d_i)$	Calculate discharge of bus during route
$\eta_{\xi_i} = \eta_i + \text{ConsumptionCost}(v_i, a_i, e_i) - \Delta_i$	Charge constraint
$\kappa_i \geq \eta_i + \text{ConsumptionCost}(v_i, a_i, e_i)$	Ensure the bus is not charged over its maximum capacity
$a_i \leq u_i \leq (T - s_i)$	Arrival time $<$ initial charge time $<$ maximum initial charge time
$d_i \leq e_i$	Detach time should be less than or equal to departure
$s_i = d_i - u_i$	Time spent on charger is equal to the difference of the attach and de

Where the valid queue position/time constraint is as defined in [1] and depicted in Figure 1. Also note that the η constraints can only be verified *after* the schedule has been generated as the initial charge for each visit is based from the previous charger selection and charge time.

4 Simulated Annealing

SA is a local search (exploitation oriented) single-solution based (as compared to population based) meta-heuristic approach in which its main advantage is simply [2]. This model is named after its analogised process where a crystalline solid is heated then allowed to cool very slowly until it achieves its most regular possible crystal lattice configuration [3]. There are five key components to SA:

- Initial Temperature
- Cooling schedule (temperature function)
- Generation mechanism
- Acceptance criteria
- Local search iteration count (temperature change counter)

The initial temperature and cooling schedule are used to regulate the speed at which the solution attempts to converge to the best known solution. When the temperature is high, SA encourages exploration. As it cools down (in accordance to the cooling schedule), it begins to encourage local exploitation of the solution [?].

4.1 Cooling Equation (Experimental)

There are three basic types of cooling equations as shown in Figure 2. A linear cooling schedule is defined by

$$T[n] = T[n-1] - \Delta_0$$

with $T[0] = T_0$ and $\Delta_0 = 1/2\ C^\circ$ in Figure 2. A geometric cooling schedule is mostly used in practice [4]. It is defined by

$$T[n] = \alpha T[n-1]$$

where $\alpha = 0.995$ in Figure 2. An Exponential cooling schedule is defined by the difference equation is define as

$$T[n] = e^{\beta} T[n-1]$$

where $\beta = 0.01$ in Figure 2. The initial temperature, T_0, in the case of Figure 2, is set to $500^\circ\ C$ and each schedule's final temperature is $1\ C^\circ$.

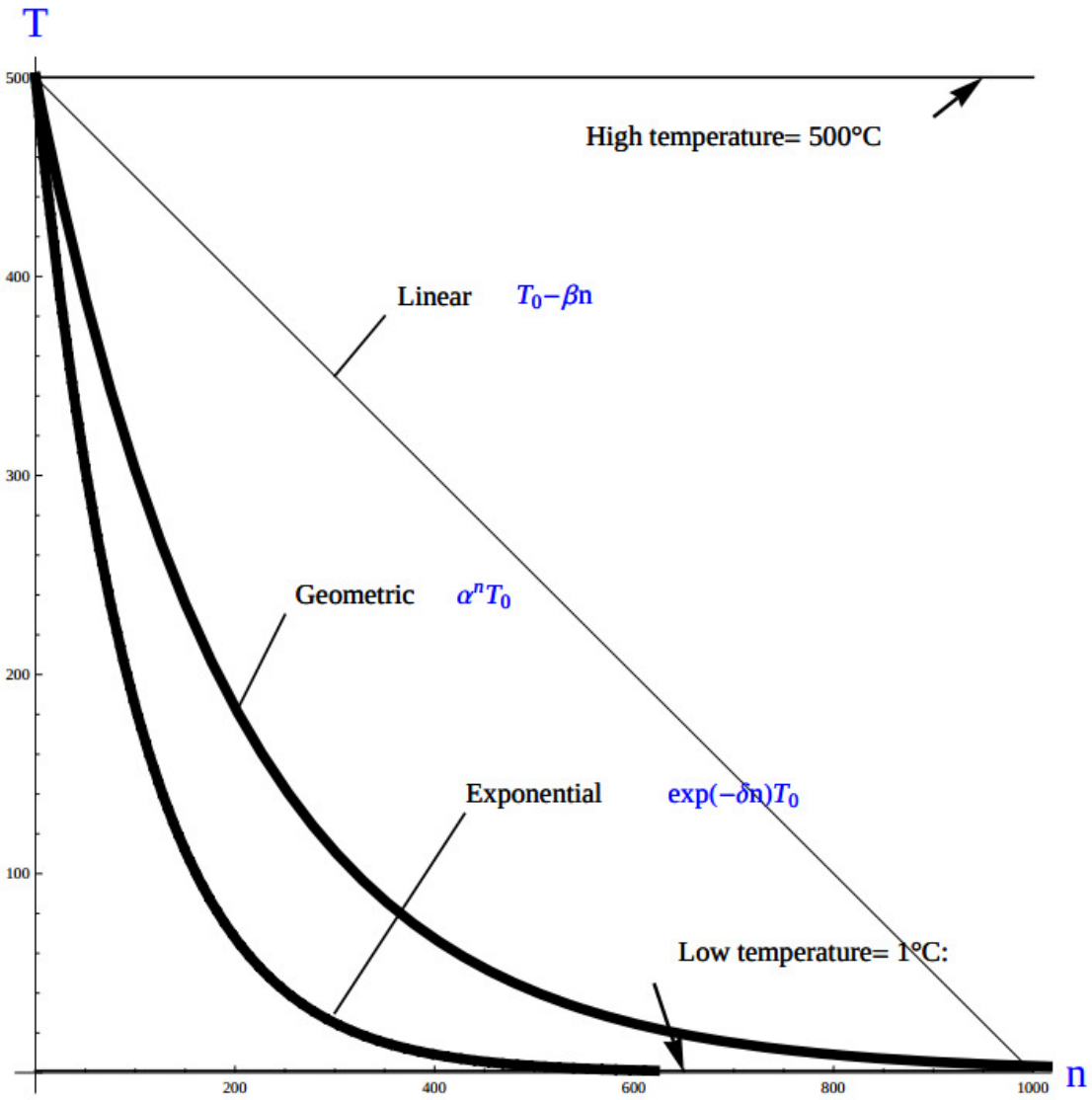

Figure 2: Cooling equations

4.2 Acceptance Criteria

Acceptance criteria describes the method to accept or reject a given candidate solution. In SA, if a new candidate solution is more fit than the currently stored solution it is always accepted as the new solution. However, within SA, worse candidate solutions may be accepted as the new solution. The probability of accepting the candidate solution is described by the function $\exp(\frac{J(new-sol)-J(old-sol)}{T})$ where $J()$ is the objective functions described in subsection 3.2.

4.3 Generation Mechanisms

Generation mechanisms in SA are used to generate random solutions to propose to the optimizer. For the case of the bus generation, five generation mechanism shall be used:

- New visit
- Slide visit
- New charger
- Remove
- New window

These generator mechanisms will in turn be utilized by three wrapper functions. One of them being to generate a set of bus route data and the other two used to generate candidate solutions to the bus routes. These routines are defined as follows:

- Route generation, Figure 4, which utilizes route metadata as shown Figure 5
- Schedule generation, Figure 6
- Tweak schedule, Figure 7

4.3.1 Generators

This section describes and outlines the algorithm pool for the different generator types that are utilized in the wrapper functions. Note that to satisfy constraints, B extra dummy chargers with a power of 0 KW will be added to the array of valid chargers. When a bus is not to be placed on a charger, it will be placed in the queue $v_i \in \{Q, ..., Q + b\}$. Where Q is the total amount of chargers and b is the bus id.

New visit The new visit generator describes the process of moving bus b from the idle queue, $v_i \in \{Q, .., Q + b\}$ to a valid charging queue, $v_i \in \{0, .., Q\}$. A list of tuples describing valid time, u_i and d_i, for each charger will be listed and randomly selected using a uniform distribution. The algorithm is defined in Algorithm 3.

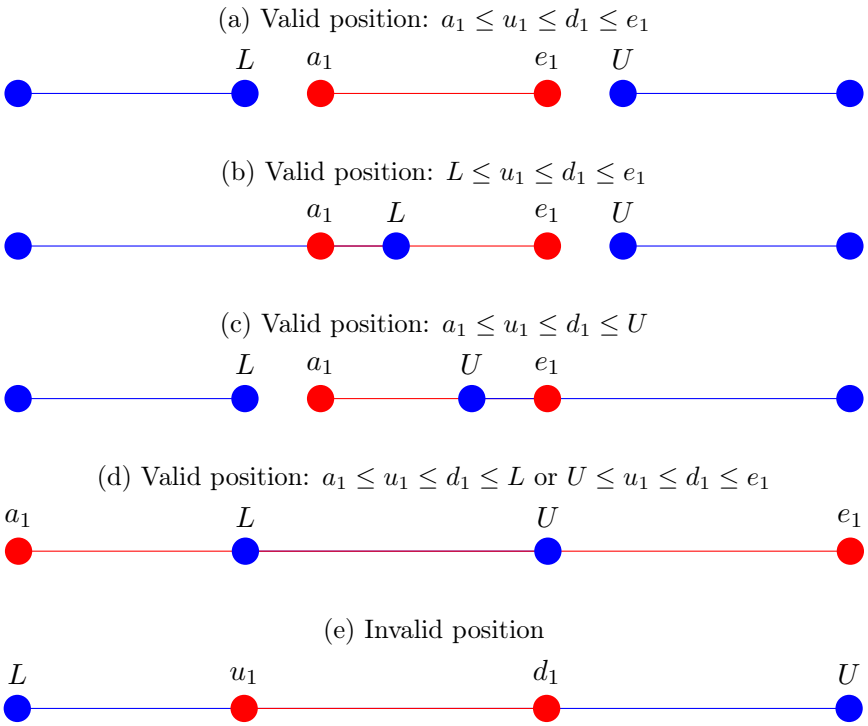

Figure 3: Outlines the different cases that requested time and charger allocated time can overlap

Algorithm: New Visit
Input: Visit index, route data, Charger data: (i, route-data, charger-data)
Output: Tuple of queue and valid time region: (v, u, d)
begin
  $a \leftarrow$ route-data[i].a;
  $e \leftarrow$ route-data[i].e;
  valid-visit $\leftarrow \emptyset$;
  for $q \leftarrow 0$ **to** Q **do**
    for *free-region* $\leftarrow$ **to** *charger-data[q]* **do**
      `Union(`*valid-visit, (q,*`findFreeTime(`*free-region, (a,e)*`)`*)*`)`;
    end
  end
  return $\mathbb{U}_{[valid-visit[0],valid-visit[length(valid-visit)-1]]}$
end

Algorithm 3: New visit algorithm

Where $\mathbb{U}_{[}a, b]$ is the continuous uniform distribution of a and b, `route-data` is the data generated in `RouteGeneration` (described in section 4.3.2), and `charger-data` are the time intervals allocated to buses. The algorithm to find free time is defined in Algorithm 4. The cases are depicted in Figure 3.

Algorithm: Find Free Time

Input: Lower and upper bound of available time and arrival and departure time for bus: (L, U, a, e)

Output: Tuple of initial and final charge times: (u, d)

begin

  if $L \leq a$ *and* $U \geq e$ **then**

    u $\leftarrow \mathbb{U}_{[a,e]}$;

    d $\leftarrow \mathbb{U}_{[u,e]}$;

  end

  else if $L > a$ *and* $U \geq e$ **then**

    u $\leftarrow \mathbb{U}_{[L,e]}$;

    d $\leftarrow \mathbb{U}_{[u,e]}$;

  end

  else if $L \leq a$ *and* $U < e$ **then**

    u $\leftarrow \mathbb{U}_{[a,U]}$;

    d $\leftarrow \mathbb{U}_{[u,U]}$;

  end

  else $L > a$ and $U < e$

    u $\leftarrow \emptyset$;

    d $\leftarrow \mathbb{U}_{[u,U]}$;

  end

  return *(u,d)*

end

Algorithm 4: Find free time algorithm searches and returns the available time frames

Slide visit Slide visit is used for buses that have already been scheduled. Because $a_i \leq u_i \leq d_i \leq e_i$ (arrival time is less than initial charge time which is less than the detatch time which is less than the time the bus exists the station), there may be some room to move u_i and d_i within the window $[a_i, e_i]$. Two new values, u_i and d_i are are selected with a uniform distribution to satisfy $a_i \leq u_i \leq d_i \leq e_i$.

Algorithm: Slide Visit

Input: Visit index, route data, Charger data: (i, route-data, charger-data)

Output: Tuple of queue, valid time region: (v, u, d)

begin

  $a \leftarrow$ route-data$[i].a$;

  $e \leftarrow$ route-data$[i].e$;

  $u \leftarrow \mathbb{U}_{[a,e]}$;

  $d \leftarrow \mathbb{U}_{[u,e]}$;

  return *(v,d)*

end

Algorithm 5: Slide Visit Algorithm

New charger Similar to new visit, this generator moves a bus from one queue to another; however, the new charger generator moves a bus from one charger queue to another, $v_i \in \{0, .., Q\}$. A new charger will be selected at random with a uniform distribution.

Algorithm: New Charger

Input: Visit index, route data, Charger data: (i, route-data, charger-data)
Output: Tuple of queue, valid time region: (v, u, d)
begin
  $a \leftarrow$ route-data[i].a;
  $e \leftarrow$ route-data[i].e;
  $v \leftarrow$ route-data[i].v;
  valid-visit $\leftarrow \emptyset$;
  for $q \leftarrow 0$ **to** Q *and* $q \neq v$ **do**
    for *free-region* $\leftarrow$ **to** *q.free* **do**
      `Union(`*valid-visit,* `findFreeTime(`*free-region, (a,e)*`))`;
    end
  end
  return $\mathbb{U}_{[valid-visit[0],valid-visit[length(valid-visit)-1]]}$
end

Algorithm 6: New Charger Algorithm

Remove The remove generator simply removes a bus from a charger queue and places it in its idle queue, $v_i \in \{Q, ..., Q + B\}$.

Algorithm: New Visit

Input: Visit index, route data, Charger data: (i, route-data, charger-data)
Output: Tuple of queue, time region: (v, u, d)
begin
  $v \leftarrow Q + b$;
  $u \leftarrow$ route-data[i].u;
  $d \leftarrow$ route-data[i].d;
  return (v, u, d)
end

Algorithm 7: Remove algorithm

New window New window is a combination of the remove and then new visit generators (section 4.3.1 and section 4.3.1).

Algorithm: New Window

Input: Visit index, route data, Charger data: (i, route-data, charger-data)
Output: Tuple of queue, valid time region: (v, u, d)
begin
  $v \leftarrow$ route-data[i].v;
  $u \leftarrow$ route-data[i].u;
  $d \leftarrow$ route-data[i].d;
  $(v, u, d) =$ `Remove(`v, u, d`)`;
  $(v, u, d) =$ `NewVisit(`v, u, d`)`;
  return (v, u, d)
end

Algorithm 8: New window algorithm

4.3.2 Generator Wrappers

This section covers the algorithms utilized to select and execute different generation processes for the SA process.

Route Generation The objective of route generation is to create a set of metadata about bus routes given the information in Figure 5. Specifically, the objective is to generate I routes for B buses. Each visit will have

- Initial charge (for first visit only)
- Arrival time
- Departure time
- Final charge (for finial visit only)

This is created by following the "GenerateSchedule" state in the state diagram found in Figure 4. In essence the logic is as follows: Generate B random numbers that add up to I visits (with a minimum amount of visits set for each bus). For each bus and for each visit, set a departure time that is between the range $[\min_{\text{rest}}, \max_{\text{rest}}]$ (Figure 5), set the next arrival time to be $j \cdot \frac{T}{\text{number-of-bus-visits}}$ where j is the j^{th} visit for bus b. Finally, calculate the amount of discharge from previous arrival to the departure time.

Algorithm: RouteGeneration
Input: Route YAML metadata path: (path)
Output: Array of route events: (route-data)
begin
  while *!schedule-created* **do**
    arrival-new ← 0.0;
    arrival-old ← 0.0;
    departure-time ← 0.0;
    num-visit ← `NumBusVisits(`B`)`;
    schedule-created ← false;
    for $b \in B$ **do**
      for $n \in num - visit[b]$ **do**
        arrival-old ← arrival-new;
        if $j = num - visit[b]$ **then**
          final-visit = true;
        end
        else
          final-visit = false;
        end
        departure-time ← `DepartureTime(`*arrival-old, final-visit*`)`;
        arrival-new ← current-visit*$\frac{T}{total-visit-count}$;
        discharge ← discharge-rate*(next-arrival-depart-time) ;
        `Union(`*route-data, (arrival-old, departure-time, discharge)*`)`;
      end
    end
    schedule-created ← `Feasible(`*route-data*`)`;
    `SortByArrival(`*route-data*`)`;
  end
end

Algorithm 9: Route generation algorithm

Algorithm: DepartureTime

Input: Previous arrival and final visit flag: (arrival-old and final-visit)
Output: Next departure time: (depart)
begin
  if *final-visit* **then**
    depart ← T;
  end
  else
    depart ← arrival-old + $\mathbb{U}_{[min-rest,max-rest]}$;
  end
  return *depart*
end

Algorithm 10: Departure time algorithm

Where `discharge-rate` is read from YAML data shown in Figure 5, the `Feasible` method is used to determine if the generated schedule is valid (conditions covered in subsection 3.3).

Schedule Generation The objective of this generator is to generate a candidate solution to the given schedule. To generate a candidate solution the generator is given the route schedule data that was previous generated. A bus is picked at random, $b \in B$, then a random route is picked for bus b. The new arrival generator is then utilized. This process is repeated for each visit. The state diagram is depicted in the state digram in Figure 6 and outlined in Algorithm 11.

Algorithm: ScheduleGeneration

Input: Route data: (route-data)
Output: Candidate charging schedule: (schedule)
begin
  schedule ← ∅;
  for *i in I* **do**
    bus ← $\mathbb{U}_{[0,B]}$;
    visit ← $\mathbb{U}_{[0,total-visit-count]}$;
    `Union(`*schedule*`,NewVisit(`*(visit.a, visit.e)*`))`;
  end
  return *schedule*
end

Algorithm 11: Schedule generation algorithm

Tweak Schedule As described in SA, local searches are also employed to try and exploit a given solution [**?**]. The method that will be employed to exploit the given solution is as follows: pick a bus, pick a visit, pick a generator. This state diagram is depicted in Figure 7 and outlined in Algorithm 12.

Algorithm: TweakSchedule

Input: Schedule candidate solution: (schedule)
Output: Perturbed schedule: (schedule)
begin
  for *i in I* **do**
    bus $\leftarrow$ $\mathbb{U}_{[0,B]}$;
    visit $\leftarrow$ $\mathbb{U}_{[0,total-visit-count]}$;
    generator $\leftarrow$ $\mathbb{U}_{[0,generator-count]}$;
    schedule $\leftarrow$ `GeneratorCallback` [generator](i, route-data, charger-data);
  end
  return *schedule*
end

Algorithm 12: Tweak schedule algorithm

5 Optimization Algorithm

This final section combines the generation algorithms and the optimization problem into a single algorithm. The objective is to outline the SA process from start to finish. Algorithm 9 generates a set of bus routes utilizing the route metadata in Figure 5. The initial temperature and cooling schedule will be selected and passed into the SA optimization algorithm. A new candidate solution will be generated. For each step in the cooling schedule will have K iterations to attempt to find a local maxima. Each perturbation to the system is then compared to the current candidate solution. If the new candidate solution is better it is kept; however, if the candidate solution is worse, the solution may still be kept with a probability exp(del-sol/T) as described in subsection 4.2. This process is summarized in Algorithm 13.

Algorithm: SA PAP

Input: Bus route metadata: (file-path)
Output: Optimal charging schedule: (schedule)
begin
  T_0 $\leftarrow$ `InitTemp()`;
  $T_{schedule}$ $\leftarrow$ `GetCoolSchedule()`;
  route-metadata $\leftarrow$ `LoadYaml(`*file-path*`)`;
  routes $\leftarrow$ `RouteGeneration(`*route-metadata*`)`;
  best-solution $\leftarrow$ $v \in$ `ScheduleGeneration(`*routes*`)`;
  foreach $T \in T_{schedule}(T_0)$ **do**
    candidate-solution $\leftarrow$ `ScheduleGeneration(`*routes*`)`;
    foreach $k \in K$ **do**
      del-sol $\leftarrow$ `J(`*candidate-solution*`) - J(`*best-solution*`)`;
      if *del-sol* $\leq$ *0* **then**
        best-solution $\leftarrow$ candidate-solution;
      end
      else if *del-sol* $\geq$ *0* **then**
        best-solution $\leftarrow$ candidate-solution with probability exp(del-solτ_k);
      end
      schedule $\leftarrow$ `TweakSchedule(`*schedule*`)`;
    end
  end
end

Algorithm 13: Simulated annealing approach to the position allocation problem

References

[1] Find if two rectangles overlap using c++.

[2] Michel Gendreau and Jean-Yves Potvin, editors. *Handbook of Metaheuristics*. Internationalseries in operation research & management science. Springer International Publishing, 3 edition, oct.

[3] Darrall Henderson, Sheldon H. Jacobson, and Alan W. Johnson. The theory and practice of simulated annealing. In *International Series in Operations Research &amp: Management Science*, pages 287–319. Kluwer Academic Publishers.

[4] Andre A. Keller. *Multi-Objective Optimization In Theory and Practice II: Metaheuristic Algorithms*. BENTHAM SCIENCE PUBLISHERS, mar 2019.

[5] Jing-Quan Li. Battery-electric transit bus developments and operations: A review. *International Journal of Sustainable Transportation*, 10(3):157–169, 2016.

[6] David G. Luenberger and Yinyu Ye. Penalty and barrier methods. *Linear and Nonlinear Programming*, page 401–433, 2008.

[7] Ahad Javandoust Qarebagh, Farnaz Sabahi, and Dariush Nazarpour. Optimized scheduling for solving position allocation problem in electric vehicle charging stations. In *2019 27th Iranian Conference on Electrical Engineering (ICEE)*, pages 593–597, 2019.

[8] Jordan Radosavljevic. *Metaheuristic Optimization in Power Engineering*. Energy Engineering. Institution of Engineering and Technology, Stevenage, England, June 2018.

[9] Maria Analia Rodriguez and Aldo Vecchietti. A comparative assessment of linearization methods for bilinear models. *Computers and Chemical Engineering*, 48:218–233, 2013.

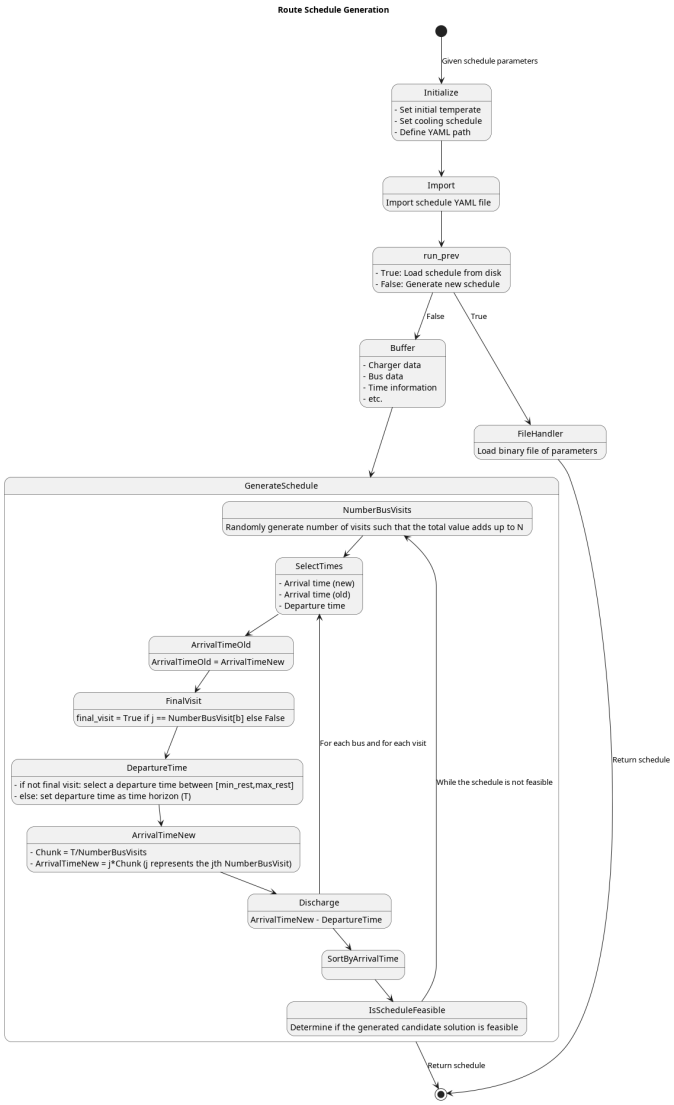

Figure 4: Route generation state diagram

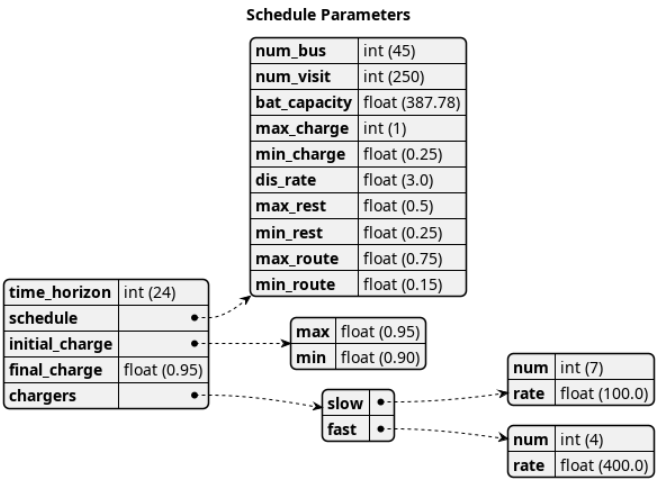

Figure 5: Route YAML file with example data

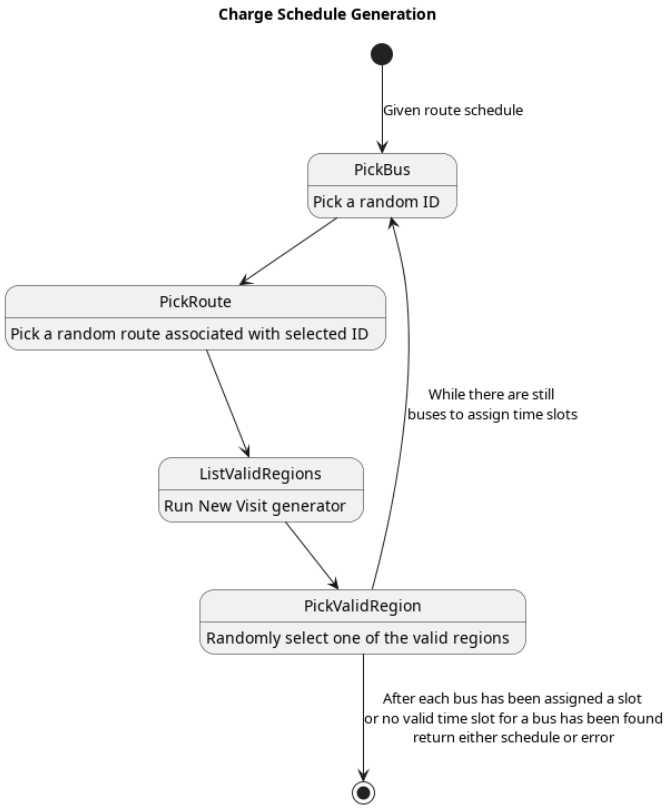

Figure 6: Charge solution state diagram

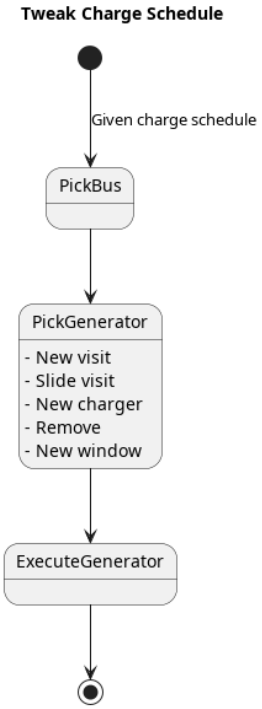

Figure 7: Solution tweak state diagram