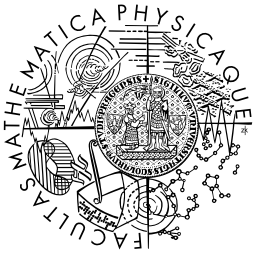

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jakub David

Energy optimization in a family house

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: RNDr. Jiří Fink, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor RNDr. Jiří Fink, Ph.D. for his guidance and for patiently answering all of my questions.

Title: Energy optimization in a family house

Author: Jakub David

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Fink, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The goal of this thesis is to compare various optimization algorithms for optimizing energy usage in residential households. We consider a model of a household with heat and power cogeneration, fixed electricity and domestic hot water consumption and devices with controllable start of their operation. For this problem, we use a mixed-integer linear programming solver, and we implemented local search, evolutionary algorithm and particle swarm optimization. We compare these algorithms on data measured from multiple different households.

Keywords: energy optimization, MILP, local search, nature inspired algorithms

Název práce: Optimalizace energie v rodinném domě

Autor: Jakub David

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jiří Fink, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Cílem této práce je porovnat různé optimalizační algoritmy pro optimalizaci spotřeby energie v domácnostech. Uvažujeme model domácnosti s kogenerací tepla a elektřiny, fixní spotřebou elektřiny a horké vody a se zařízenímy s ovladatelným začátkem provozu. K řešení tohoto problému používáme řešič smíšeně celočíselných lineárních úloh a implementovali jsme lokální prohledávání, evoluční algoritmus a optimalizaci hejnem částic. Tyto algoritmy provnáváme na datech naměřených v různých domácnostech.

Klíčová slova: optimalizace energie, MILP, lokální prohledávání, přírodou inspirované algoritmy

Contents

Introduction

Energy consumption in residential households makes a significant part of carbon dioxide emission [1]. In this thesis, we compare different algorithms for optimizing the energy usage in households. The objective of these algorithms is to minimize the cost of energies. Minimizing the cost is easier than minimizing peaks in the usage of energies [2]. Furthermore, a local heating problem can be solved using a greedy algorithm [3], but we consider a more complex model of a household.

We made a model of a household that we optimize the energy usage in. A house, in our model, is connected to the grid and is equipped with a Combined Heat and Power unit (mCHP). Furthermore, it has fixed electricity and hot water usage. In addition to the fixed electricity usage, there are electrical appliances such as a dishwasher or a washing machine. The start of operation of these appliances is set by the algorithms. The model is described in detail in Chapter 1.

In Chapter 2 we discuss data used to test the algorithms. We describe how we obtained the data, and how we processed it to the form used by the algorithms.

One of the approaches we use to solve the problem is mixed integer linear programming (MILP), described in Chapter 3. We reformulated the model of a household to a form appropriate for MILP and employed a MILP solver to optimize the model.

Next, we implemented various optimization algorithms ourselves. These algorithms use two different representations of a solution defined in Chapter 4. The first representation is quite straightforward modification of the model from Chapter 1. However, in the second representation, only a part of the solution is optimized by these algorithms. When these solutions need to be evaluated, we find the rest of the solution with a linear programming solver. The algorithms we implemented are local search (Chapter 5), evolutionary algorithm (Chapter 6) and particle swarm optimization (Chapter 7). First, we describe these algorithms. Then follows Chapter 8, where we evaluate different hyperparameters for these algorithms, and select the best ones. Finally, we evaluate these optimization approaches and present their comparison in Chapter 9.

In Attachments is a description of attached files (Attachment A.1), how to run the experiments conducted in this thesis (Attachment A.2) and details about the implementation of algorithms are described in Attachment A.3.

1 Problem statement

We consider a household equipped with a mCHP connected to a buffer for hot water and a battery. The mCHP consumes natural gas to simultaneously heat water and generate electricity. Furthermore, the household is connected to the grid to buy electricity.

Water heated by mCHP is for domestic hot water (DHW) consumption. This DHW consumption is uncontrollable, and we have a prediction of it for the whole planning period. We call this fixed hot water demand. Similarly, we have fixed electricity demand. It is electricity used by devices where we can not control when they are turned on.

However, in addition to that, we have time-shiftable devices. These devices represent electronic household appliances such as a washing machine or a dishwasher. Their start can be scheduled and every device has its own operation period. The operation period is a time window when the device can be started.

All the devices have to run exactly once during the planning period. In order to start one device multiple times, it has to be added as multiple devices with different, not overlapping, operation periods.

1.1 Mathematical model

We are using a discrete time model, so the planning period is split into T discrete time intervals of the same length. This gives us a set of time intervals $\mathcal{T} = \{1, \ldots, T\}$. We use 5 minute long time intervals.

In every time interval $t \in \mathcal{T}$ is fixed electricity demand D_t^e and fixed hot water demand D_t^h. Also, we have a set of time-shiftable devices $\mathcal{D}$. Each of these devices $d \in \mathcal{D}$ has an electricity consumption profile D_i^d for $i \in \{1, \ldots, l_d\}$, where l_d is the length of the profile. Moreover, it has a start of its operation period O_s^d and an end of its operation period O_e^d. Note that O_e^d can not be greater than $T - l_d + 1$.

If the mCHP is running, the amount of natural gas used in one time interval is G, H denotes the produced heat and E is the generated electricity. Otherwise, it uses and produces nothing. The buffer connected to the mCHP has a state of charge s_t^h for every time interval $t \in \mathcal{T}$. This is the amount of heat stored in the buffer. Additionally, the buffer has an upper bound for the state of charge U^h and the lower bound is naturally 0. At the start of the planning period, the state of charge of the buffer has initial value I^h.

The battery serves as a buffer for electricity. For every time interval $t \in \mathcal{T}$, s_t^e is the state of charge of the battery at the beginning of the interval. The state of charge has an initial value I^e and the capacity of the battery is limited by an upper bound U^e and the lower bound is 0. How much we can charge the battery in one time interval is also limited by an upper bound M_c^e and discharging is limited by an upper bound M_d^e. Both of the lower bounds are 0.

The price of electricity can change throughout the day, so we have different price P_t^e for each time interval $t \in \mathcal{T}$. For natural gas, on the other hand, we have only one price P^g.

Next, we have variables to control our model. Let x_t^- be a variable denoting the amount of electricity used to charge the battery in time interval $t \in \mathcal{T}$. Similarly,

we declare a variable x_t^+ to denote the electricity provided by the battery in time interval $t \in \mathcal{T}$. Let x_t^g be a variable denoting the amount of electricity used from the grid in time interval $t \in \mathcal{T}$. Let y_t be a binary variable that is true, if the mCHP is turned on in time interval $t \in \mathcal{T}$, and false otherwise. Let z_d be a variable that denotes in which time interval time-shiftable device $d \in \mathcal{D}$ starts. To simplify notation later on, we denote the electricity demand of all time-shiftable devices as

$$d_t^s = \sum_{\substack{d \in \mathcal{D} \\ z_d \leq t \\ t < z_d + l_d}} D_{t - z_d + 1}^d.$$

We also consider losses in this model. Storing electricity in the battery has loss L_s^e. In every time interval $t \in \mathcal{T}$ the amount of stored energy lost is $L_s^e s_t^e$. The heat buffer also has storage loss L_s^h, which works in the same way. Charging and discharging the battery have losses L_c^e and L_d^e respectively. $L_c^e x^-$ is the amount of electricity lost during charging and when discharging additional $L_d^e x^+$ is subtracted from the state of charge of the battery.

The variables and the states of the buffers are constrained by the following equations.

$$x_t^g + y_t E + x_t^+ = D_t^e + d_t^s + x_t^- \quad \text{for} \quad t \in \mathcal{T} \tag{1.1}$$
$$s_{t+1}^e = (1 - L_s^e) s_t^e + x_t^-(1 - L_c^e) - x_t^+(1 + L_d^e) \quad \text{for} \quad t \in \mathcal{T} \tag{1.2}$$
$$s_{t+1}^h = (1 - L_s^h) s_t^h - D_t^h + y_t H \quad \text{for} \quad t \in \mathcal{T} \tag{1.3}$$
$$s_1^e = I^e \tag{1.4}$$
$$s_1^h = I^h \tag{1.5}$$
$$0 \leq s_t^e \leq U^e \quad \text{for} \quad t \in \mathcal{T} \tag{1.6}$$
$$0 \leq s_t^h \leq U^h \quad \text{for} \quad t \in \mathcal{T} \tag{1.7}$$
$$0 \leq x_t^+ \leq M_d^e \quad \text{for} \quad t \in \mathcal{T} \tag{1.8}$$
$$0 \leq x_t^- \leq M_c^e \quad \text{for} \quad t \in \mathcal{T} \tag{1.9}$$
$$0 \leq x_t^g \quad \text{for} \quad t \in \mathcal{T} \tag{1.10}$$
$$y_t \in \{0, 1\} \quad \text{for} \quad t \in \mathcal{T} \tag{1.11}$$
$$z_d \in \{O_s^d, \ldots, O_e^d\} \quad \text{for} \quad d \in \mathcal{D} \tag{1.12}$$

Equation (1.1) ensures that electricity production equals electricity consumption. On the left-hand side of the equation are all the sources of electricity: the electricity used from the grid, electricity produced by the mCHP and electricity provided by the battery. On the right-hand side is the consumption of electricity: fixed electricity demand, the electricity demand of all the time-shiftable devices and the electricity used to charge the battery.

Equation (1.2) is the update of the state of charge of the battery. The term $(1 - L_s^e) s_t^e$ is the state of charge from the previous time interval after the energy storage loss is accounted for and terms $x_t^-(1 - L_c^e)$ and $x_t^+(1 + L_d^e)$ are the charging and discharging of the battery respectively.

The update of the state of charge of the heat buffer is (1.3). It has the term to account for the heat storage loss as well. However, the rest of the equation is different because we directly use hot water stored in the buffer to cover the hot

water demand, and we directly transfer the heated water from the mCHP to the buffer.

To initialize the states of charge of the buffers we have (1.4) and (1.5). The remaining equations (1.6) through (1.12) set the domains of our variables.

As the objective, we minimize the price of purchased natural gas and electricity. This can be calculated as the sum

$$\sum_{t \in \mathcal{T}} \left(x_t^g P_t^e + y_t G P^g \right). \tag{1.13}$$

2 Input Data

In this chapter we describe the sources of our data in Section 2.1 and their preprocessing in Section 2.2. Then, in Section 2.3 we explain how the data is stored in JSON file format. In Section 2.4 is described how we use the data in this thesis, and in Section 2.5 is a visualization of the data.

2.1 Description of used data

For electricity demand, we use a dataset containing a measured time series from several small businesses and residential households [4]. The data in this dataset was originally collected from a trial site in Konstanz (Germany) from October 2013 to December 2016 during a project called CoSSMic. The records for residential buildings include electricity imported from a grid, electricity consumption of various devices and photovoltaic energy generation (only for some households). These measurements were done in 1 minute long intervals.

Domestic hot water consumption comes from a dataset with profiles for different types of consumers [5]. This data is based on measurements from households in Québec (Canada) that were conducted between November 2006 and April 2007. From this raw data 12 different DHW draw profiles were created. There are profiles for average, median, sparing and profligate consumers, and each of these types further differs in temporal usage patterns. All of these profiles have measurements in 5 minute long time intervals.

We use day-ahead market prices for electricity and intra-day market prices for natural gas, both from the Czech Republic [6]. Electricity prices are available as an average cost of one MWh in Euros within a one-hour long period. Prices of natural gas, on the other hand, are only available as an average price of all transactions during one day, measured in Eur/MWh. All the prices used are from 2023

For the parameters of mCHP, we used a specification sheet for reference [7]. We also use a specification sheet for the heat buffer [8]. This specification sheet belongs to electric water heaters; however, we only need to know what sizes of hot water buffers are used in households and their losses.

Similarly, we used multiple specification sheets of battery systems for residential use [9, 10, 11, 12]. Although these batteries are designed to be used in conjunction with solar panels, it suffices as a reference to obtain the necessary values. Self-discharge of these batteries is 0.1–0.3% per day [13].

2.2 Data preprocessing

Here is how we process the data.

From the dataset for electricity demand, we use the data from four different residential buildings. Unfortunately, many measurements in this dataset are linearly interpolated, because these measurements were not recorded due to connection issues. Therefore, we discarded all days containing those records. Furthermore, the data was recorded as measurements from an electricity meter, so

we changed it to electricity used during the individual time intervals. Finally, we resampled the data from 1 min intervals to 5 min intervals used in our algorithms.

As electricity demand, we use a field called grid import and fields containing dishwasher and washing machine power consumption for device profiles. The power consumption of these devices has to be subtracted from the electricity demand to have the profiles of time-shiftable devices and electricity demand separate. Because of measuring errors, this causes the electricity demand to be negative in some time intervals, so we set the electricity demand to 0 in those time intervals. Then the power consumption of the devices is split into continuous segments to create multiple profiles for each device.

To create the operation periods for the time-shiftable devices, we use time intervals corresponding to the beginnings of the profiles in the original dataset. Based on these time intervals, the operation periods are generated randomly, and are at most 6 hours long. Also, the operation periods are shortened if necessary to prevent operation of a device outside the planning period and overlap of profiles belonging to one device in the original dataset.

The data for DHW consumption only requires to be converted from liters to kWh. In this conversion, we assume that the water is heated from 15 °C to 65 °C.

Electricity prices also only need to be converted from MWh to kWh, and all time intervals in each hour have assigned the same price.

The particular mCHP in the specification sheet has modulating output; however, our model does not support that. Consequently, we randomly assign each instance of the input data one fixed mode. Moreover, its maximum output is quite high for one household, so we limited thermal output to 5 kW, and natural gas consumption and electrical output proportionally to it.

From the specification sheets of the batteries, we obtained capacity and maximum charge and discharge power. For battery charging and discharging loss, we took roundtrip efficiency and split it evenly between the two.

The parameters of batteries and heat buffers are randomized too. Each instance of input data gets randomly assigned parameters from one battery (capacity, maximum charge and discharge power, charging and discharging loss) and parameters of one hot water buffer (capacity and storage loss). From the specification sheet of electric heaters, we only use the 200 and 300 liter variants.

The initial state of the battery is a random value from the interval 0 to one fifth of its capacity. The initial state of the heat buffer is also random. However, its minimum value is $D_1^h + D_2^h + D_3^h$. Otherwise, the mCHP might not be able to generate enough heat in the beginning to cover the hot water demand. The maximum value of the initial state is likewise one fifth of its capacity.

2.3 Input data structure

We store each instance of input data in a separate JSON file. These JSON files have the following properties:

- `time_interval_count`: integer, number of time intervals in the planning period
- `electricity_prices`: array of floats, prices of electricity in Eur/kWh for each time interval

- `gas_price`: float, price of natural gas in Eur/kWh for the whole planning period
- `electricity_demand`: array of floats, electricity demand in kWh for every time interval
- `water_demand`: array of floats, DHW consumption in kWh corresponding to all time intervals
- `mCHP`: object, parameters of the mCHP:
  - `gas_consumption`: float, natural gas required to run the mCHP for a time interval in kWh
  - `electricity_production`: float, electricity produced by the mCHP during one time interval in kWh
  - `heat_production`: float, heat produced by the mCHP during one time interval in kWh
- `electricity_buffer`: object, parameters of the battery:
  - `capacity`: float, maximum capacity of the battery in kWh
  - `initial_state`: float, initial capacity of the battery in kWh
  - `max_input`: float, maximum amount of power in kWh, that can be used to charge the battery during one time interval
  - `max_output`: float, maximum amount of power in kWh discharged during one time interval
  - `input_loss`: float, energy lost during charging
  - `output_loss`: float, energy lost during discharging
  - `storage_loss`: float, energy lost during one time interval
- `heat_buffer`: object, parameters of the hot water buffer:
  - `capacity`: float, maximum capacity in kWh
  - `initial_state`: float, initial capacity in kWh
  - `storage_loss`: float, energy lost during one time interval
- `devices`: array of objects, time-shiftable devices, each device has the following properties:
  - `name`: string, this property is not used by the algorithms; however, it is useful for identifying the device profiles
  - `profile`: array of floats, power consumption profile of the device in kWh
  - `operation_period_start`: integer, first time interval when the device can be started
  - `operation_period_end`: integer, last time interval when the device can be started

Note that the length of arrays `electricity_prices`, `electricity_demand` and `water_demand` has to be equal to `time_interval_count`. The number of devices is unlimited, and it can be zero.

2.4 Data Split

Our input data are divided into a development set and a test set. The development set is used to select hyperparameters for our algorithms and the test set is used for the final comparison of the algorithms. The difference between these sets is that they have electricity consumption and DHW demand from different households, and use prices from different days.

Each of these sets contains 40 files in total. We have files with 1 day, 2 days, 5 days and 10 days long operations and there are 10 files of each length.

2.5 Overview of the Data

To get an idea of what the data looks like, we have an example in this section. The following figures are graphs of electricity demand (Figure 2.1) and DHW demand (Figure 2.2), electricity prices (Figure 2.3) and profiles of time-shiftable devices from one data file in the development set. Figure 2.4 shows power consumption profile of a dishwasher and Figure 2.5 shows a profile of a washing machine.

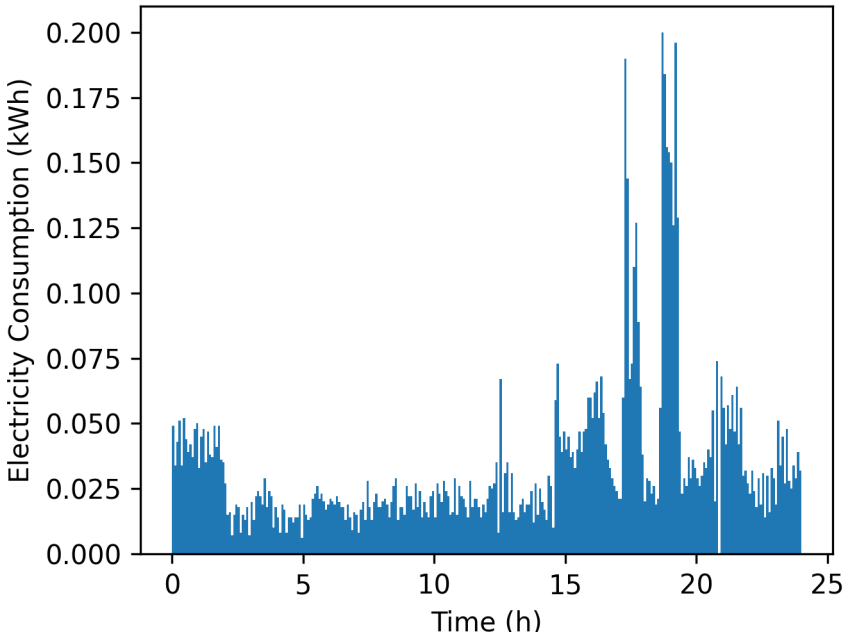

Figure 2.1 Electricity demand

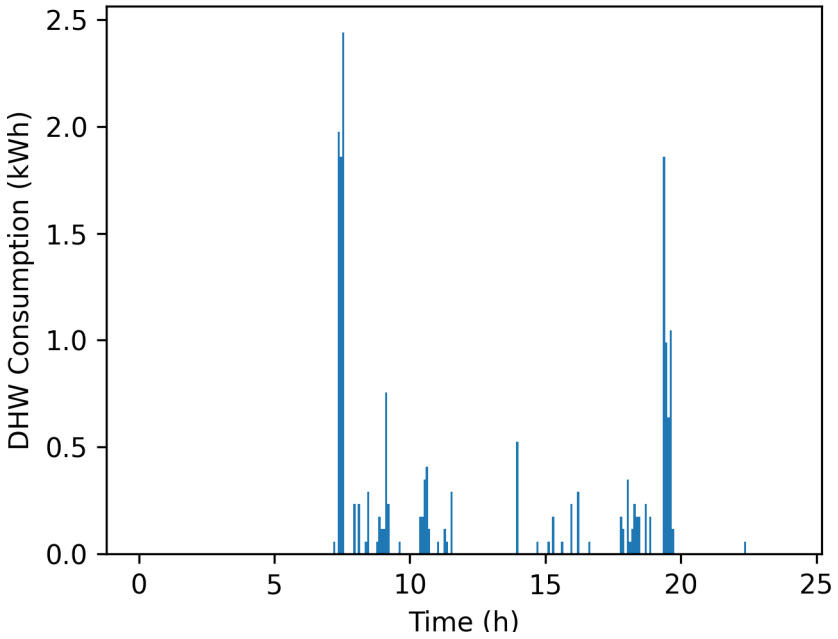

Figure 2.2 DHW demand

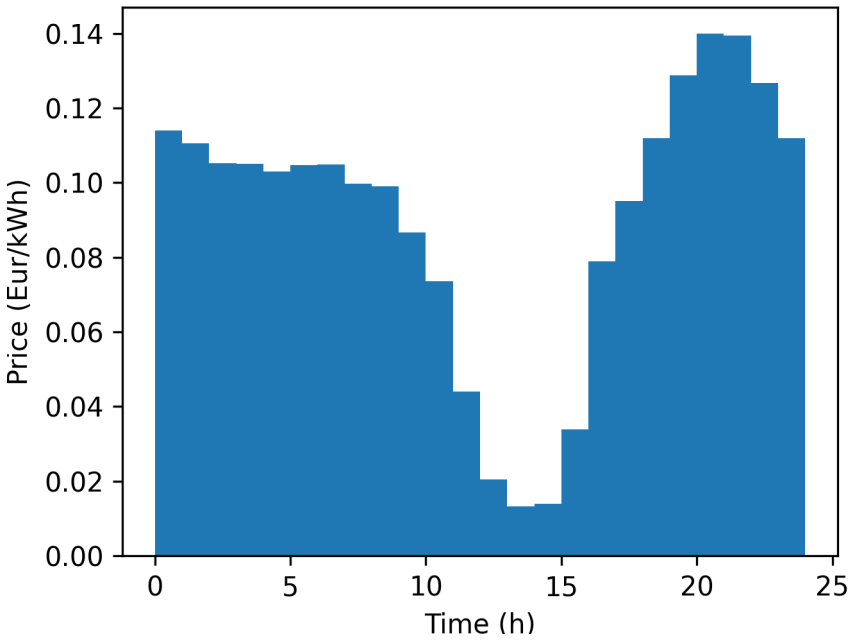

Figure 2.3 Electricity prices

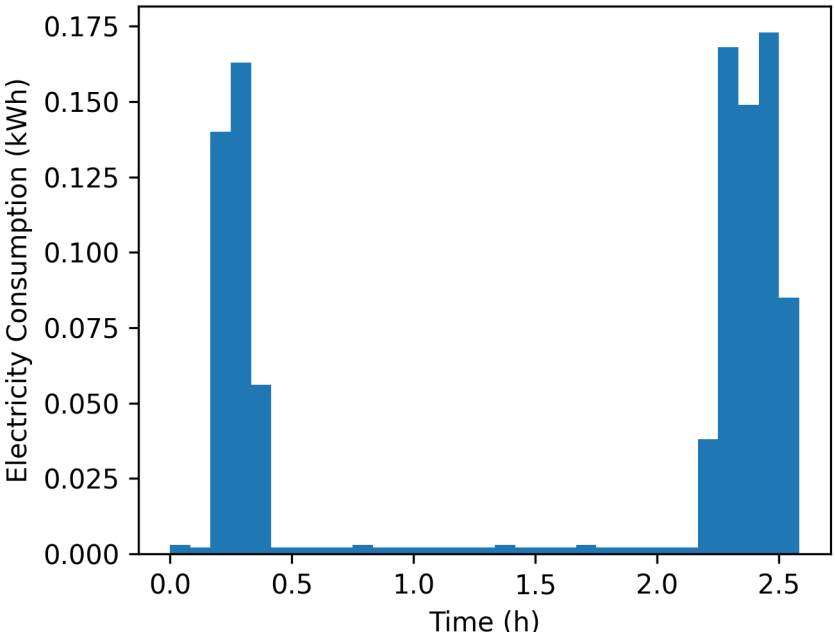

Figure 2.4 Electricity consumption profile of a dishwasher

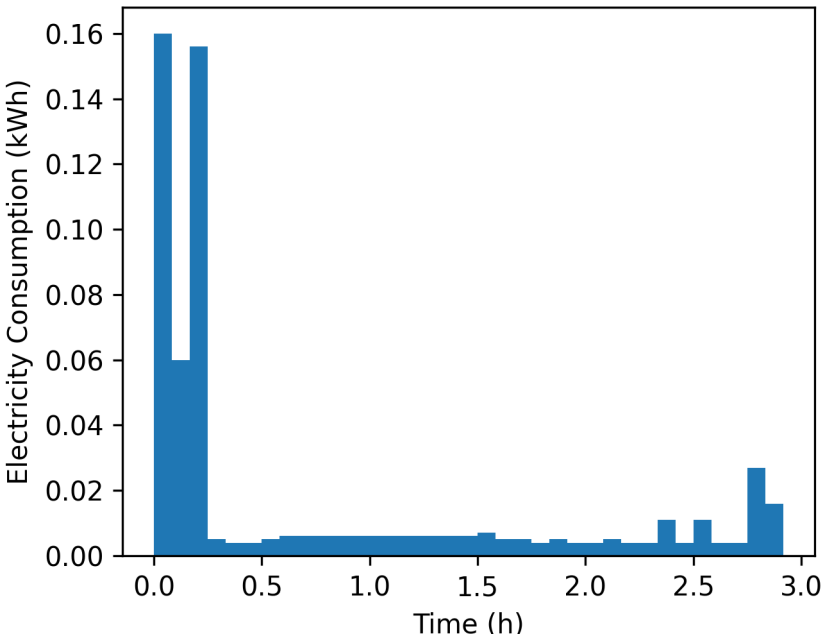

Figure 2.5 Electricity consumption profile of a washing machine

3 Mixed-integer linear programming

Linear programming (LP) is an approach for solving optimization problems. A problem in linear programming is called linear program. A linear program is composed of real variables, an objective function and constraints. Since this is linear programming, the objective function has to be linear and the constraints are linear equalities and inequalities. To solve a linear program means to find values of the variables that satisfy the given constraints and maximize or, depending on the problem, minimize the objective function.

In what we call linear programming, we use real variables. The method that uses integer variables is called integer programming. We have real variables and integer variables, so we need to use a method that allows both types of variables. This method is called mixed-integer linear programming (MILP).

Linear programs can be solved in polynomial time. However, integer programming is an NP-hard problem, and so is MILP [14].

3.1 MILP Model

To use MILP to solve our problem, we need to make a few changes to the model defined in Section 1.1. Instead of one variable z_d for time-shiftable device $d \in \mathcal{D}$, we define variables $z_{d,t}$ where $t \in \{O_s^d, \ldots, O_e^d\}$. Variable $z_{d,t}$ is 1, if device d starts in time interval t, and 0 otherwise. This definition requires that for every device $d \in \mathcal{D}$ exactly one variable $z_{d,t}$ has to be 1, and the rest of them have to be 0, so (1.12) is replaced by the following two equations:

$$z_{d,t} \in \{0,1\} \quad \text{for} \quad d \in \mathcal{D},\, t \in \{O_s^d, \ldots, O_e^d\} \tag{3.1}$$

$$\sum_{t=O_s^d}^{O_e^d} z_{d,t} = 1 \quad \text{for} \quad d \in \mathcal{D} \tag{3.2}$$

Equation (3.1) is the domain of the variables and (3.2) ensures that the device starts in exactly one time interval.

We can keep all the constraints (1.1) through (1.11). However, the definition of term d_t^s in equation (1.1) has to be changed because its value depends on variables z_d. The new definition is

$$d_t^s = \sum_{d \in \mathcal{D}} \delta_{s,t}^d$$

where $\delta_{s,t}^d$ is a profile of device d at time interval t, if the device starts in time interval s:

$$\delta_{s,t}^d = \begin{cases} D_{t-s+1}^d & \text{if } t \geq s \wedge t < s + l_d \\ 0 & \text{otherwise} \end{cases}$$

4 Solution Representation and Fitness

In this chapter, we discuss the representation of a solution used in local search, evolutionary algorithm and particle swarm optimization. In Section 4.1 we define two different representations, and in Section 4.2 is definition of their fitness functions. Finally, in section 4.3 we describe correction of infeasible solutions.

4.1 Solution Representation

A solution for our algorithms could be represented as it is defined in Section 1.1. However, to help the algorithms to find a better solution we propose two different representations.

In Subsection 4.1.1 we describe the first representation, which we named complete representation. The second representation is in Subsection 4.1.2, and we call it partial representation. Solution in complete representation is called complete solution and solution in partial representation is called partial solution.

4.1.1 Complete Representation

A complete solution uses only variables x_t, y_t and z_d. We explain variable x_t and how it changes calculations in our model.

Instead of variables x_t^- and x_t^+, we define variables x_t for each time interval $t \in \mathcal{T}$. If the battery is being charged in time interval t, then $x_t = x_t^-$; otherwise $x_t = -x_t^+$.

This representation requires us to change (1.1) and (1.2), and add a constraint for the variable domain. First, let's define a function to apply charging or discharging loss on variable x_t:

$$\text{Loss}(x) = \begin{cases} x(1 + L_d^e) & \text{if } x < 0 \\ x(1 - L_c^e) & \text{otherwise} \end{cases}$$

Now we can define the constraints as follows:

$$x_t^g + y_t E = D_t^e + d_t^s + x_t \quad \text{for} \quad t \in \mathcal{T} \tag{4.1}$$

$$s_{t+1}^e = (1 - L_s^e)s_t^e + \text{Loss}(x_t) \quad \text{for} \quad t \in \mathcal{T} \tag{4.2}$$

$$-M_d^e \leq x_t \leq M_c^e \quad \text{for} \quad t \in \mathcal{T} \tag{4.3}$$

The rest of the equations remain unchanged; however, equations (1.8) and (1.9) are no longer necessary.

Furthermore, the variables x_t^g are not a part of our solution. They can be simply calculated using (4.1) from the rest of the variables:

$$x_t^g = D_t^e + d_t^s + x_t - y_t E \tag{4.4}$$

4.1.2 Partial Representation

This representation is inspired by incomplete solution representation in [15].

A partial solution only stores values of variables controlling the mCHP and time-shiftable device starts. Since the variables for charging and discharging the battery are continuous, we use linear programming to find their optimal values given the values of variables for mCHP and time-shiftable devices.

The linear program has variables x_t^-, x_t^+ and x_t^g for $t \in \mathcal{T}$. The objective function and constraints do not change from the definition in Section 1.1. However, we only need to keep (1.1), (1.2), (1.4), (1.6), (1.8), (1.9) and (1.10). Moreover, y_t and d_t^s in (1.1) are considered as constants.

4.2 Fitness

The objective function defined in Section 1.1 is not enough for evaluating solutions because it works only on valid solutions. However, our algorithms also work with invalid solutions, so we need to define a fitness function that also calculates how close an infeasible solution is to being feasible.

In Subsection 4.2.1 we describe a penalty for infeasible solutions, and we define the fitness function for complete and partial solutions in Subsection 4.2.2 and Subsection 4.2.3 respectively.

4.2.1 Penalty

The algorithms produce solutions with variables that have values inside their valid ranges. Therefore, we only need penalty for states s_t^e, s_t^h and variables x_t^g, because they are not included in the solutions.

To calculate the penalties we have the following functions:

$$\psi_s(x,u) = \begin{cases} |x| & \text{if } x < 0 \\ x - u & \text{if } x > u \\ 0 & \text{otherwise} \end{cases}$$

$$\psi_g(x) = \begin{cases} |x| & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}$$

The function $\psi_s(x,u)$ is a penalty for state x with upper bound u, and $\psi_g(x)$ is used for variables x_t^g.

Furthermore, we define penalties for all variables in a solution:

$$\Psi_e = \sum_{t \in \mathcal{T}} \psi_s(s_t^e, U^e)$$
$$\Psi_h = \sum_{t \in \mathcal{T}} \psi_s(s_t^h, U^h)$$
$$\Psi_g = \sum_{t \in \mathcal{T}} \psi_g(x_t^g)$$

Ψ_e is a penalty for all the battery states, Ψ_h is a penalty for all the heat buffer states and Ψ_g is a penalty for all variables x_t^g.

If variables x_t, y_t and z_d of a solution have valid values and penalties Ψ_h, Ψ_e and Ψ_g are 0, then the solution is valid, because (1.1) is satisfied by assigning values to variables x_t^g by (4.4).

4.2.2 Complete Representation

The fitness function of a complete solution written in pseudocode can be seen in Program 1.

We first calculate the penalty for the solution. If the penalty is greater than tolerance, it is multiplied by a large constant to ensure its fitness is higher than fitness of a valid solution. The tolerance is there because of floating point arithmetic errors and because measurements in reality are not accurate anyway.

Otherwise, we return the objective calculated by (1.13).

Program 1 Fitness function of a complete solution

```
function fitness(solution):
    penalty = Ψ_h + Ψ_e + Ψ_g

    if penalty > 10^-4:
        return penalty · 10^7

    return Objective(solution)
```

4.2.3 Partial Representation

The fitness function of a partial solution written in pseudocode can be seen in Program 2.

Fitness for partial solutions is similar to complete solutions. Except there is only penalty for states s_t^h, and the objective is calculated by linear programming. If an optimal solution for the linear program cannot be found, the fitness function is ∞.

Program 2 Fitness function of a partial solution

```
function fitness(solution):
    penalty = Ψ_h

    if penalty > 10^-4:
        return penalty · 10^7

    result = SolveLP(solution)

    if result is optimal:
        return Objective(result)

    return ∞
```

4.3 Correction of infeasible solutions

If we have an infeasible solution, it either has overflowing or underflowing buffers or negative usage of electricity from the grid.

The correction algorithm for hot water buffer is in Program 3. Overflow is simple to fix, but more than one mCHP variable might need to be changed to fix underflow.

The correction of battery and grid variables is done simultaneously, as it can be seen in Program 4. In this case, it is more complicated because of the battery input and output losses, so we do the correction only approximately. Because of this, the correction does not guarantee to make the solution feasible.

Program 3 Correction of hot water buffer

```
s = I^h
for t = 1, ..., T:
    s = update_state(s)
    if s < -10^-4:
        s += cover_water_demand(t, -s)

    if s > U^h + 10^-4:
        y_t = 0
        s -= H

function cover_water_demand(t, amount):
    heat_generated = 0
    for i = t, ..., 1:
        if heat_generated >= amount:
            return heat_generated

        if y_i == 0:
            y_i = 1
            heat_generated += H(1 - L_s^h)^(t - i)

    return heat_generated
```

Program 4 Correction of battery

```
s = I^e
previous_s = s
for t = 1, ..., T:
    s = update_state(s)

    if s < 0:
        x_t -= s
        x_t^g -= s
        s = 0

    if x_t^g < 0:
        x_t -= x_t^g
        x_t^g = 0
        s = update_state(previous_s)

    if s > U^e:
        amount = (s - U^e)(1 + L_d^e)
        fix_battery(t, amount)
        s = U^e

    previous_s = s

function fix_battery(t, amount):
    for i = t, ..., 1:
        if amount <= 0:
            return

        a = min(x_t^g, amount)
        x_t -= a
        x_t^g -= a
        amount -= a

        amount = amount(1 - L_s^e)
```

5 Local Search

Local search algorithms are characterized by the fact that they only remember the current state they are in, and they consider only their neighboring states to improve the solution. For optimization problems, these algorithms use objective function to evaluate how good a state is.

We use local search algorithm called hill-climbing. It is a greedy algorithm, that is iteratively improving a solution while the objective function is increasing (for maximization problems) or decreasing (for minimization problems).

More about local search can be found in [16].

5.1 Implementation

In this section, we discuss our implementation of the hill-climbing algorithm.

To give a high-level overview how we implemented the hill-climbing algorithm, we first initialize the solution. The initialization is described in Subsection 5.1.2. Then until a termination condition is met, we apply operators described in Subsection 5.1.1. We terminate the algorithm after a specified time. These operators try to improve the solution. A list of operators is passed to the algorithm as an argument, and in every iteration all the operators are applied in the given order.

We use both complete solutions and partial solutions defined in Section 4.1. For evaluating solutions we use fitness function from Section 4.2.

5.1.1 Operators

To make improvements to the solution, we apply various operations on it. Every operator can make multiple changes to a solution in one call, so some of our operators have a boolean parameter `only_best`. If this parameter is set to true, only the best change made by the operator is applied to the solution; otherwise, we apply the changes that improve the solution immediately, and the following changes are made to the altered solution. If none of the changes improves the solution, it remains unchanged.

We have the following operators:

- `ImproveBattery(n,only_best)` This operator randomly selects two time intervals i and j. For interval i it selects value $a = x_i \cdot r$, where r is a random number from uniform distribution $U(0, 1)$. For interval j it uses $b = a \cdot l$, where l are losses for charging the battery in interval i, storing the electricity in the battery until interval j and then discharging it; or vice-versa if $j < i$. Finally, two variables are changed by formulas $x_i := x_i + a$ and $x_j := x_j - b$. This is repeated `n` times.

- `ImproveMchpToggle(only_best)` This operator tries the opposite value for every variable of the mCHP y_t.

- `ImproveMchpSwap(only_best)` If any two mCHP variables y_t belonging to time intervals t and $t + 1$ have opposite values, this operator swaps values of those variables.

- `ImproveMchpExtend(only_best)` It tries to extend every consecutive sequence of time intervals, in which the mCHP is turned on.
- `ImproveMchpShorten(only_best)` It tries to shorten every consecutive sequence of time intervals, in which the mCHP is turned on.
- `ImproveDeviceStartsAll` This operator chooses the best starting time for every time-shiftable device. All the time intervals in the device's operation period are considered. The values are tried separately for each time-shiftable device.
- `ImproveDeviceStartsShift` This operator shifts the starting time of every time-shiftable device by one time interval forward or backward, or leaves it set to the original value; whichever is the best. The values are tried separately for each time-shiftable device.

Some of these operators actually do a subset of operations of other operators. For example, every value tried by `ImproveDeviceStartsShift` is also tried by `ImproveDeviceStartsAll`. However, `ImproveDeviceStartsShift` requires fewer evaluations of the fitness function, which could be an advantage.

5.1.2 Initialization

To initialize the solution, we set the variables of time-shiftable devices to the middle of their operation periods. The mCHP is turned on only immediately before the hot water is needed. The electricity produced by mCHP, which is not consumed by fixed electricity demand, is used to charge the battery. The electricity from the battery is used in the earliest time intervals, when the mCHP does not cover the fixed electricity demand. The power consumption of time-shiftable devices is for battery initialization ignored for higher flexibility of the solution. We call this the greedy initialization.

An improved variant of initialization for complete solutions, which we also use, is initializing a partial solution and then making a complete solution from it. This essentially initializes the battery with linear programming.

6 Evolutionary Algorithm

Evolutionary algorithms are inspired by nature, and how organisms evolve and adapt to their environment. An evolutionary algorithm imitates natural selection. Only the individuals adapted to the environment survive and have offsprings, propagating better genetic information to the next generations.

To solve an optimization problem, evolutionary algorithm simulates a population of individuals. These individuals are initialized randomly (Section 6.1), and a fitness function is used to evaluate how good these individuals are. It selects individuals from the population for reproduction based on their fitness. This is parental selection (Section 6.2).

Then, these individuals are used to create their offsprings. Two parents are merged to create one or two offsprings, so each offspring has genetic information from both of his parents. This part is called recombination or crossover (Section 6.3).

Next, the offsprings are mutated (Section 6.4). Each information contained in the individual has a chance to change to a different value.

Finally, individuals are selected for the next generation (Section 6.5). It is called the environmental selection, and it can also use fitness to select better individuals.

This is repeated from parental selection to environment selection, until a termination condition is met. We terminate the algorithm after a specified time.

Both complete solutions and partial solutions defined in Section 4.1 are used to represent an individual. To evaluate an individual, we use the fitness functions defined in Section 4.2.

The theory in this chapter is written according to [17].

6.1 Initialization

The initialization of individuals is random. Generating values for battery variables is done by the continuous uniform probability distribution. The lower and upper bounds are $-M_d^e$ and M_c^e respectively, multiplied by coefficients $c_{\text{battery}}^{\text{lb}}$ and $c_{\text{battery}}^{\text{ub}}$ given as parameters. The variables for mCHP are set to 1 with a probability also given as a parameter, and 0 otherwise. The starts of time-shiftable devices are initialized by the discrete uniform probability distribution with time intervals from their operation periods.

To initialize a partial solution, only the initialization for mCHP variables and time-shiftable device start variables are used.

We also use the greedy initialization from local search defined in Subsection 5.1.2.

6.2 Parental Selection

For the parental selection, we use tournament selection. In tournament selection, every time we are selecting an individual, n individuals from the population

are randomly chosen, and the individual with the highest fitness among the chosen individuals is selected. We use tournament selection with $n = 2$.

6.3 Crossover

We use one point crossover for an array containing variables x_t and y_t. One point crossover randomly selects a point i from discrete uniform distribution $U(1, T)$. Then, it creates two offsprings. One receives variables x_t and y_t with t from 1 to i from the first parent, and variables x_t and y_t with t from $i + 1$ to T from the second parent. The other one receives the complement of those variables.

We choose to use the same point for both x_t and y_t in complete solutions because two different points would probably break the solutions more often. Partial solutions do the one point crossover only for variables y_t.

Every individual variable z_d is chosen separately at random from the parents and given to one offspring. The other offspring receives the other variable.

Finally, we do the correction of variables x_t and y_t described in Section 4.3 to help the algorithm find feasible solutions.

This crossover is applied to two parents with probability P_{cross}. If it is not applied, two offsprings are created by copying the parents.

6.4 Mutations

To mutate an individual, we add to variables x_t a random value from normal distribution $\mathcal{N}(0, \sigma_{\text{battery}})$ and clip the variable, if its value is outside its domain. Every variable y_t is changed to the opposite value with probability c_{mCHP}/T. Variables z_d are also mutated by adding value from $\mathcal{N}(0, \sigma_{\text{devices}})$. However, they need to be rounded to their nearest valid value. Variables σ_{battery}, σ_{devices} and c_{mCHP} are parameters of the mutation.

Every epoch each individual is mutated with probability P_{mut}.

6.5 Environmental Selection

In environmental selection we use elitism. Elitism selects k best parents for the next generation. The rest of the new generation are new offsprings. This ensures that we do not lose our best solutions found so far.

7 Particle Swarm Optimization

Particle swarm optimization (PSO) was introduced by Kennedy and Eberhart in 1995 [18], and the description in this chapter is according to [17].

PSO is a variation of an evolutionary algorithm. In this case, a population of individuals is a swarm of particles. A particle is a pair of real vectors $(\mathbf{p}, \mathbf{v})$, where $\mathbf{p}$ is a position of the particle and $\mathbf{v}$ is a velocity of the particle. Every particle also remembers their personal best location $\mathbf{p_{best}}$, a location where their fitness has been so far the best. The global best position $\mathbf{g_{best}}$ is also recorded. This is the position with the best fitness among all the particles.

The process of PSO can be imagined as particles moving about a space, each with its own velocity, gravitating towards their $\mathbf{p_{best}}$ and $\mathbf{g_{best}}$. The particles start at random positions with random velocities. In every iteration of the PSO algorithm the positions and velocities of all the particles are updated by the following equations:

$$\mathbf{v} = \omega\mathbf{v} + \phi_1\mathbf{r}_1 \odot (\mathbf{p_{best}} - \mathbf{p}) + \phi_2\mathbf{r}_2 \odot (\mathbf{g_{best}} - \mathbf{p}) \tag{7.1}$$

$$\mathbf{p} = \mathbf{p} + \mathbf{v} \tag{7.2}$$

Equation (7.1) is the update of velocity. The symbol $\odot$ in this equation is element-wise multiplication of vectors. Vectors $\mathbf{r}_1$ and $\mathbf{r}_2$ have values drawn from uniform distribution $U(0, 1)$, ϕ_1 and ϕ_2 are learning rates controlling how big the updates are, and ω is a parameter called inertia. The terms $\mathbf{p_{best}} - \mathbf{p}$ and $\mathbf{g_{best}} - \mathbf{p}$ give directions toward the personal and global bests respectively. The position is updated by adding the velocity to it in (7.2).

After a specified time, we terminate the algorithm. The PSO algorithm in pseudocode is in Program 5.

7.1 Representation

To represent particles, we use complete solutions and partial solutions described in Section 4.1. However, PSO only uses real variables, so a particle, instead of variables y_t and z_d, has real variables $\hat{y}_t$ and $\hat{z}_d$ with the following domains:

$$0 \leq \hat{y}_t \leq 1 \quad \text{for } t \in \mathcal{T} \tag{7.3}$$

$$O_s^d \leq \hat{z}_d \leq O_e^d \quad \text{for } d \in \mathcal{D} \tag{7.4}$$

Because the domains of variables are limited, the values of variables are clipped after adding the velocity. We call a particle corresponding to a complete solution a complete particle, and a partial particle corresponds to a partial solution.

Velocities are represented as particles, but their variables have no lower and upper bound.

To get a solution to our problem from a particle, we round the variables. After converting a particle to a solution, we do the correction of variables x_t and y_t described in Section 4.3.

Program 5 PSO algorithm

```
function pso(n):
    particles = randomly initialize n particles
    velocities = randomly initialize n velocities
    personal_bests = copy(particles)
    global_best = particle with the lowest fitness
    while termination condition is not met:
        for i = 1, ..., n:
            p, v = particles[i], velocities[i]
            p_best = personal_bests[i]
            r_1, r_2 = UniformVector(0, 1), UniformVector(0, 1)

            v = ωv + φ_1 r_1 ⊙ (p_best − p) + φ_2 r_2 ⊙ (g_best − p)
            p = p + v

            particles[i], velocities[i] = p, v

            if fitness(p) < fitness(p_best):
                p_best = p
                if fitness(p) < fitness(global_best):
                    global_best = p

    return global_best
```

7.2 Fitness

Fitness of a particle is evaluated by converting it to a solution, as described in Section 7.1. Then, a fitness function defined in Section 4.2 is used.

7.3 Initialization

The initialization of particles is identical to the initialization of individuals in the evolutionary algorithm described in Section 6.1. However, in PSO we also need to initialize velocities. For velocities, variables x_t are drawn from $U(-M_d^e c_{\mathbf{v}}, M_c^e c_{\mathbf{v}})$, $\hat{y}_t$ from $U(-c_{\mathbf{v}}, c_{\mathbf{v}})$ and $\hat{z}_d$ from $U(-l_d c_{\mathbf{v}}, l_d c_{\mathbf{v}})$. Variable $c_{\mathbf{v}}$ is a parameter of the initialization.

8 Hyperparameter Selection

In this chapter, we select the best hyperparameters for local search, evolutionary algorithm and PSO. To choose the hyperparameters, we only use the development set.

To select values for parameters in this chapter, we use grid search. Simply put, we select values for parameters we want to try and try every possible combination. The grid search is done on four data files of different lengths from the development set. To evaluate a combination of values we average the best fitness achieved on those four data files.

The results displayed in the following graphs are measured on the whole development set. We used MILP to get the optimum for every data file. These optima are at most 1.3% from the real optimum (gap parameter for the solver). In these graphs, we use percent error calculated as 100%·(`fitness - optimum`) / `optimum`, and display its mean calculated over the whole development set. Furthermore, if a solution was infeasible in the beginning, we removed the corresponding points in that graph, because it would be unreadable otherwise.

All the scripts that run these experiments are in subfolder `hyperparameters` of Attachment A.1.

8.1 Local Search

To determine the best combination of operators, we tried multiple combinations that make sense to us. For battery operators we consider

- `ImproveBattery(1,false)`,
- `ImproveBattery(100,false)` or
- `ImproveBattery(100,true)`.

For mCHP operators we have variants

- `ImproveMchpToggle(m)`;
- `ImproveMchpSwap(m)`;
- `ImproveMchpExtend(m)`, `ImproveMchpShorten(m)` and
- `ImproveMchpSwap(s)`, `ImproveMchpExtend(m)`, `ImproveMchpShorten(m)`.

Here variables `m` and `s` can be true or false, and extending and shortening operators always have the same value.

To determine the best combination of operators, we combined every above-mentioned battery operator variant with every listed mCHP operator combination and every time-shiftable device operator. The combinations of mCHP operators also differed in assignment of those parameter variables `m` and `s`. The operators are applied in order as they are mentioned and are tested for complete and partial representation separately.

The best operator combination for complete representation and partial representation is in Table 8.1.

Representation	Operators
Complete	ImproveBattery(100,false) ImproveMchpExtend(true) ImproveMchpShorten(true) ImproveDeviceStartsShift
Partial	ImproveMchpToggle(true) ImproveDeviceStartsAll

Table 8.1 The best operator combinations

Because the best parameters of the battery operator have `n` = 100, we tried values 10, 50, 100, and 200 for this parameter. Their comparison is in Figure 8.1.

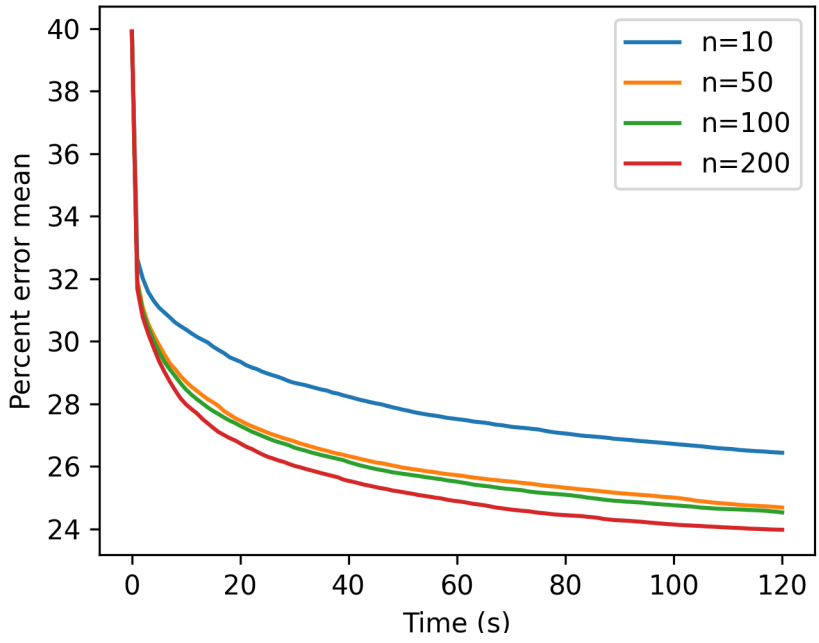

Figure 8.1 Influence of parameter n of the battery operator in local search

In Figure 8.2 is a comparison of complete and partial representation. Additionally, there is a complete solution initialized from a partial solution. Although the partial representation achieved better results, it did not improve the solution after greedy initialization very much. But it is already quite close to the optimum. The complete solution initialized from partial, on the other hand, did not improve at all. Note that one iteration in local search with partial representation took quite a long time, and the termination condition is checked only at the beginning of an iteration, so local search ran for longer than the complete representation. However, it can be seen in the graph that it did not improve any further.

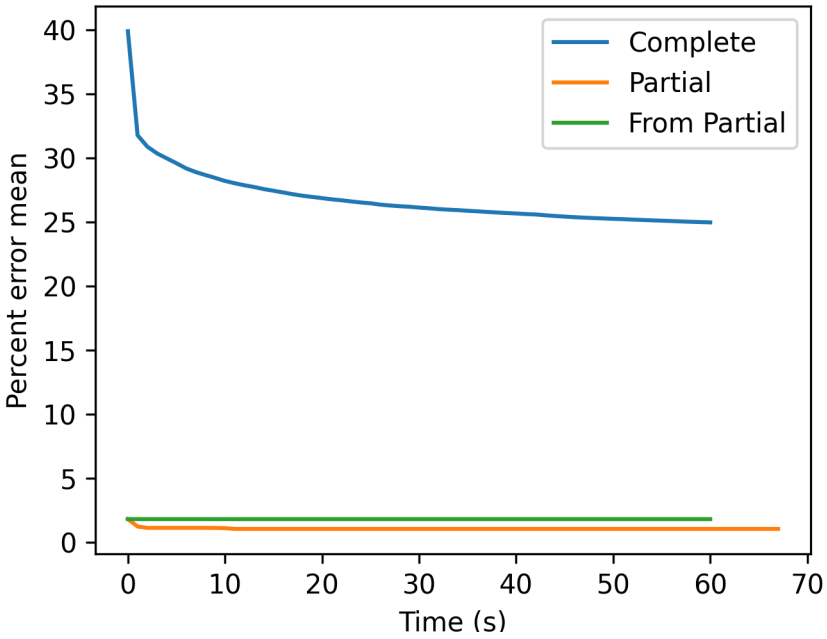

Figure 8.2 Comparison of hill climbing algorithm variants

8.2 Evolutionary Algorithm

Because there are a lot of parameters for the evolutionary algorithm, we divided the selection of parameters into three stages.

In the first stage, we used average values calculated from solutions initialized by the greedy algorithm to get initial values for the parameters of initialization. The mCHP is turned on 9% of the time, and charging and discharging of the battery is on average approximately 2% of their maxima M_c^e and M_d^e. Therefore, we used coefficients 0.04 for the battery initialization.

Parameters of mutation we just set to $\sigma_{\text{battery}} = 0.01$, $c_{\text{mCHP}} = 1$ and $\sigma_{\text{devices}} = 0.5$.

For the remaining parameters, we selected values $\{100, 500\}$ for parameter population size, $\{1, 5, 10\}$ for elite size, $\{0.1, 0.2\}$ for P_{mut} and $\{0.1, 0.2\}$ for P_{cross}. Then we tried every combination of these parameters.

The best combination can be seen in Table 8.2.

Representation	Population Size	Elite Size	P_{mut}	P_{cross}
Complete	100	5	0.2	0.8
Partial	100	10	0.1	0.8

Table 8.2 Best parameters for evolutionary algorithm

In the second stage, we ran a grid search for parameters of initialization with values $c^{\text{lb}}_{\text{battery}} = \{0.04, 0.1, 0.25, 0.5, 1.0\}$ and $c^{\text{ub}}_{\text{battery}} = \{0.04, 0.1, 0.25, 0.5, 1.0\}$. We kept the value of the parameter for initializing variables y_t, because the average

from greedy initialization is probably good. Results from this stage are in Table 8.3.

Representation	$c^{\text{lb}}_{\text{battery}}$	$c^{\text{ub}}_{\text{battery}}$
Complete	0.04	0.25

Table 8.3 Best initialization parameters for evolutionary algorithm

In Table 8.4 is the result of the third stage, where we ran a grid search for parameters of mutation with values $\sigma_{\text{battery}} = \{0.005, 0.01, 0.02, 0.05\}$, $c_{\text{mCHP}} = \{0.5, 1.0, 2.0\}$ and $\sigma_{\text{devices}} = \{0.3, 0.5, 1.0\}$.

Representation	σ_{battery}	c_{mCHP}	σ_{devices}
Complete	0.05	1.0	0.5
Partial		2.0	0.3

Table 8.4 Best mutation parameters for evolutionary algorithm

As can be seen in Table 8.5 and Figure 8.3, the correction of infeasible solutions plays a significant role.

Representation	Correction	Feasible solution found
Complete	On	100%
Complete	Off	17.5%

Table 8.5 Influence of correction for complete representation in the evolutionary algorithm

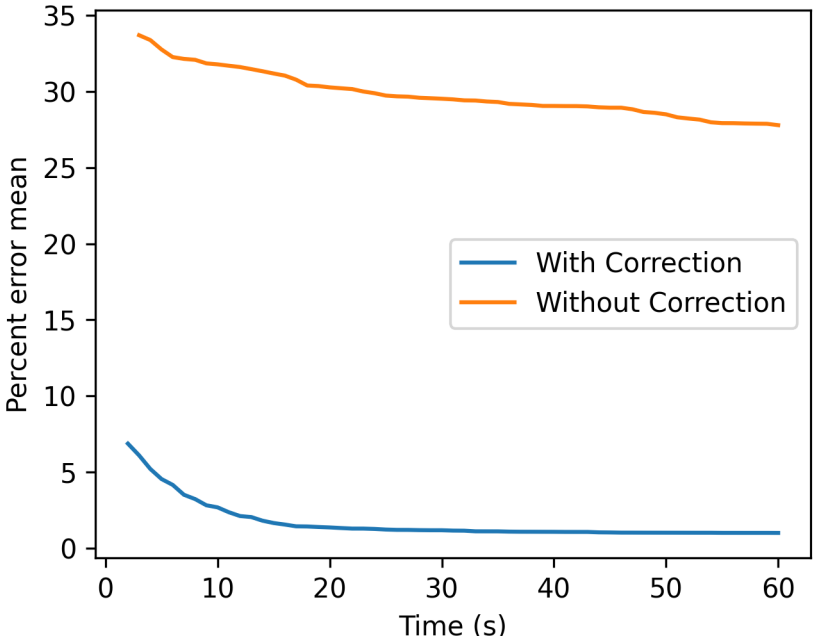

Figure 8.3 Influence of correction for partial representation in the evolutionary algorithm

Finally, in Figure 8.4 is a comparison of complete and partial representation. We also included solutions initialized by the greedy algorithm. The partial representation is better than the complete representation, but it only managed to get to fitness achieved just by using the greedy initialization.

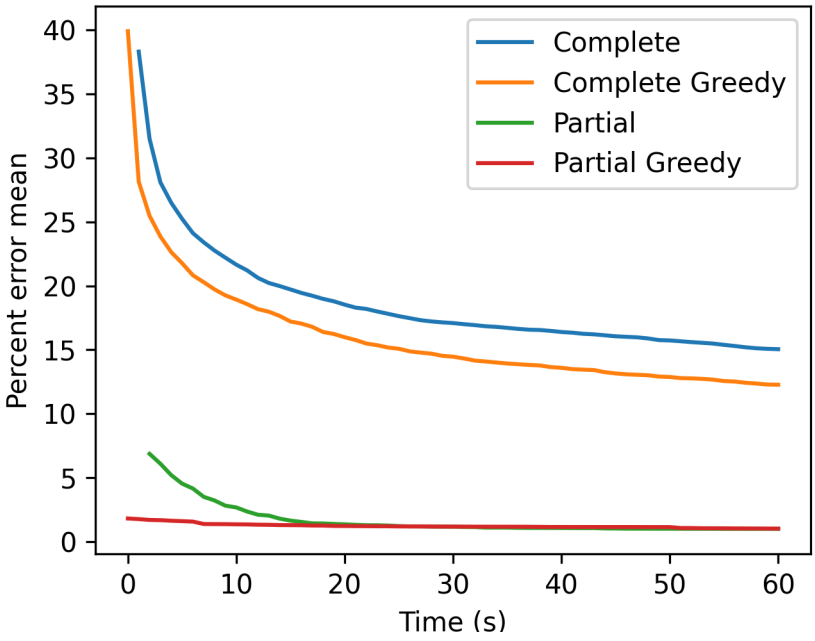

Figure 8.4 Comparison of evolutionary algorithm variants

8.3 PSO

For PSO we did the same process as for the evolutionary algorithm. We used the same values for initialization. However, PSO also has a parameter velocity coefficient, so we set it to 0.1. With these initial values, we ran a grid search for $\omega = \{0.3, 0.7\}$, $\phi_1 = \{1.3, 1.7\}$, $\phi_2 = \{1.3, 1.7\}$ and $\{100, 500\}$ for population size. The best combinations of these parameters are in table 8.6.

Representation	ω	ϕ_1	ϕ_2	Population Size
Complete	0.3	1.7	1.7	500
Partial	0.3	1.3	1.3	100

Table 8.6 Best parameters for PSO

Then we used grid search with values $\{0.1, 0.2, 0.5\}$ for the velocity coefficient $c_{\mathbf{v}}$, $c^{\text{lb}}_{\text{battery}} = \{0.04, 0.1, 0.3, 1.0\}$ and $c^{\text{ub}}_{\text{battery}} = \{0.04, 0.1, 0.3, 1.0\}$. For partial representation, we tried here only values $\{0.05, 0.1, 0.2, 0.3, 0.4, 0.5\}$ for $c_{\mathbf{v}}$. The result is in Table 8.7.

Representation	$c^{\text{lb}}_{\text{battery}}$	$c^{\text{ub}}_{\text{battery}}$	$c_{\mathbf{v}}$
Complete	1.0	0.04	0.2
Partial			0.1

Table 8.7 Best initialization parameters for PSO

Finally, we refined the parameter values selected in the beginning. For complete representation, we did this using a grid search with $\omega = \{0.2, 0.3, 0.4\}$, $\phi_1 = \{1.5, 1.7, 1.9\}$, $\phi_2 = \{1.5, 1.7, 1.9\}$ and $\{300, 500, 700\}$ for population size. For partial representation we used $\omega = \{0.2, 0.3, 0.4\}$, $\phi_1 = \{1.2, 1.3, 1.4\}$, $\phi_2 = \{1.2, 1.3, 1.4\}$ and $\{75, 100, 200\}$ for population size. The final combination of parameters is in Table 8.8.

Representation	ω	ϕ_1	ϕ_2	Population Size
Complete	0.3	1.9	1.5	700
Partial	0.2	1.4	1.2	100

Table 8.8 Best parameters for PSO run 2

Table 8.9 shows that the correction of infeasible solutions is also very important for PSO. In fact, for the partial representation, it is even more important.

We did the same comparison of complete and partial representation, as we did for the evolutionary algorithm. It can be seen in Figure 8.5. The results are similar to the evolutionary algorithm, but the complete representation did slightly better in this case.

Representation	Correction	Feasible solution found
Complete	On	100%
Complete	Off	45%
Partial	On	100%
Partial	Off	45%

Table 8.9 Influence of correction in PSO

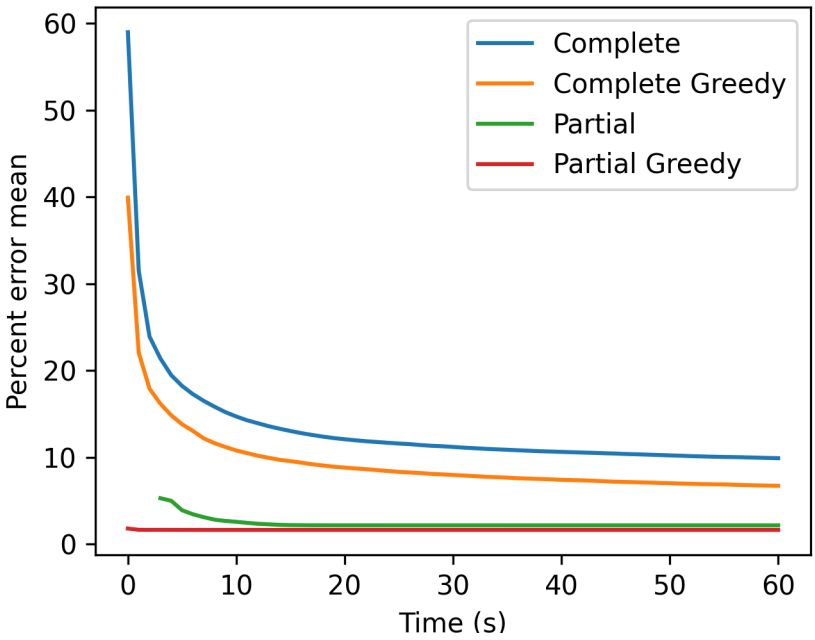

Figure 8.5 Comparison of PSO variants

9 Comparison of Algorithms

The graphs in this chapter are created the same way as in Chapter 8.

We decided to use the partial representation for local search, evolutionary algorithm and PSO, and random initialization for evolutionary algorithm and PSO in the comparison.

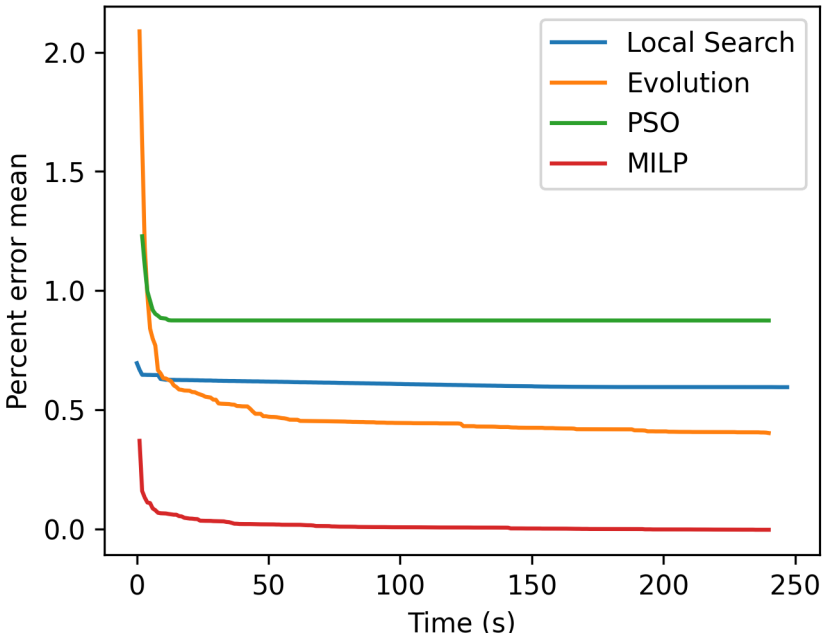

Figure 9.1 Comparison of algorithms

Figure 9.1 is the comparison of algorithms on the test dataset. We can see that MILP found the optimal solution. The rest of the algorithms also have percent error mean below 1%. However, local search started with a solution initialized by the greedy initialization, and it almost did not improve at all. This shows that greedily setting the heat buffer gives a good result, and further optimizing it only gives marginal improvements. Furthermore, PSO did not even manage to surpass the greedy initialization. On the other hand, evolutionary algorithm achieved percent error mean below 0.5%. It has the best results out of the approaches we implemented ourselves.

In Figure 9.2 are pie charts showing a breakdown of the sources of energy and the usage of energy, averaged over solution found by MILP for the test dataset. Notable is that time-shiftable devices account only for 1.1% of used energy. This is because opposed to a whole day of electricity usage in a household, there are not that many of them and they run only for a few hours.

To test how scalable it is, we took every 10 day long data file from the test set and copied every time-shiftable device 100 times. For reference, every new data file has 600–1900 time-shiftable devices. A comparison of the algorithms on this modified dataset is in Figure 9.3.

Again, MILP found the optimal solution, but note that construction of the MILP model took every time over a minute, and this time is not included in

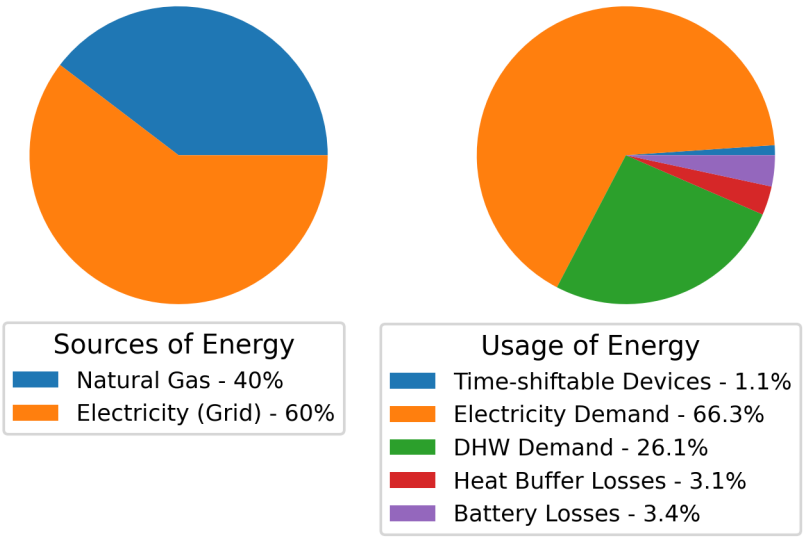

Figure 9.2 Sources of energy (left) and usage of energy (right)

the graph. However, we did not implement the construction of a model with performance in mind, so it probably could be improved. PSO, this time, also performed the worst, getting stuck in local optima. It achieved percent error mean of 5%. At least it managed to surpass the greedy initialization here. The evolutionary algorithm did worse with more devices, but it still achieved percent error mean below 3%.

The local search is very close to the optimum. However, one iteration took a long time. That is why it ran longer than the other algorithms since we check the termination condition only before every iteration. It managed to make only 4–9 iterations.

We have the breakdown of energy sources and energy usage for the modified test dataset in Figure 9.4. In this case, time-shiftable devices make 46.7% of used energy. The electricity used from the grid is higher than before and the battery losses are lower, so the time-shiftable devices use a lot of electricity directly from the grid. Thus, finding a good start time for every device can be done, in this case, by simply checking every start time for each device separately. That is probably why local search did so well.

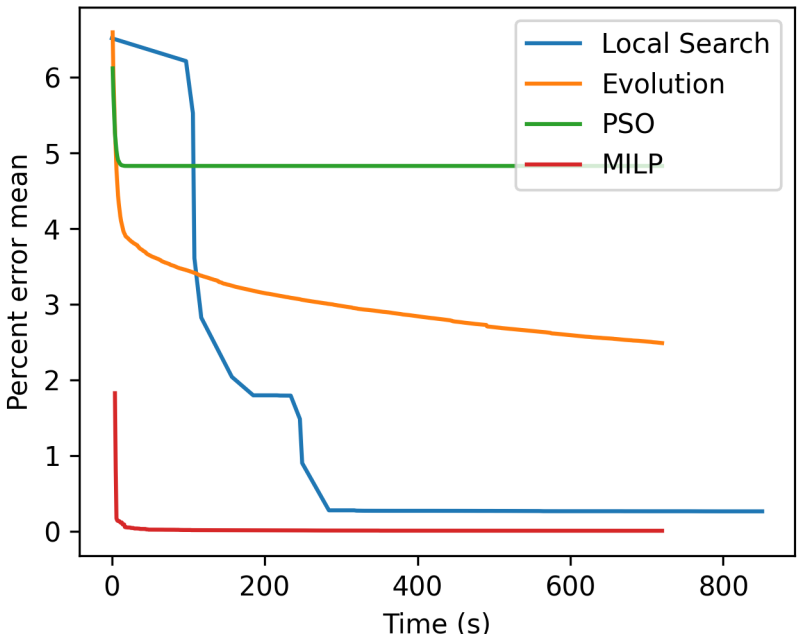

Figure 9.3 Comparison of algorithms, every time-shiftable device copied 100 times

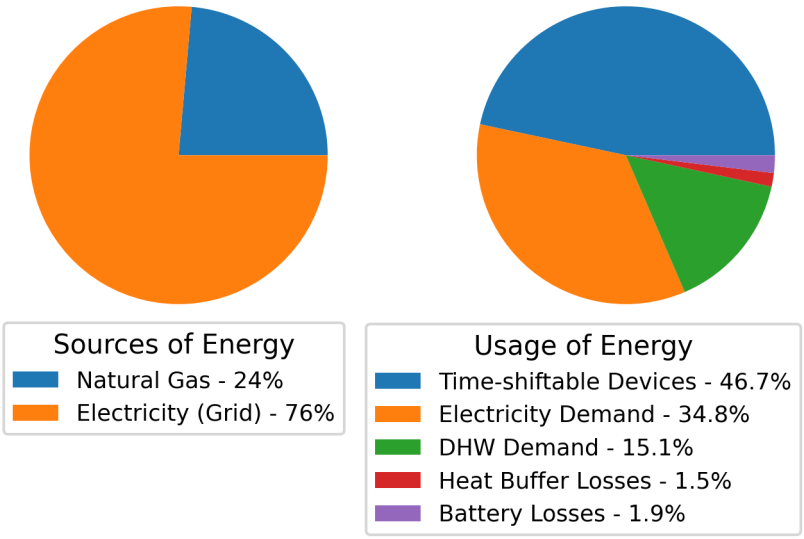

Figure 9.4 Sources of energy (left) and usage of energy (right), every time-shiftable device copied 100 times

Conclusion

The approach that performed the best is MILP. It achieves good results even if there are over 1000 time-shiftable devices. From the algorithms we implemented ourselves evolutionary algorithm generally performs the best. If there are a lot of time-shiftable devices, local search outperforms the evolutionary algorithm.

However, on the test dataset, we achieved very good results just by using the greedy initialization. So this problem is not so hard.

Bibliography

1. Goldstein, Benjamin; Gounaridis, Dimitrios; Newell, Joshua P. The carbon footprint of household energy use in the United States. *Proceedings of the National Academy of Sciences*. 2020, vol. 117, no. 32, pp. 19122–19130. Available from doi: 10.1073/pnas.1922205117.
2. Fink, Jiří; Hurink, Johann L. Minimizing costs is easier than minimizing peaks when supplying the heat demand of a group of houses. *European Journal of Operational Research*. 2015, vol. 242, no. 2, pp. 644–650. issn 0377-2217. Available from doi: 10.1016/j.ejor.2014.10.040.
3. Fink, Jiří; Hurink, Johann L. Greedy algorithm for local heating problem. *Discrete Optimization*. 2021, vol. 39, p. 100627. issn 1572-5286. Available from doi: 10.1016/j.disopt.2021.100627.
4. Data, Open Power System. *Data Package Household Data* [https://data.open-power-system-data.org/household_data/2020-04-15/]. 2020. Version 2020-04-15.
5. Edwards, Skai; Beausoleil-Morrison, Ian; Laperrière, André. Representative hot water draw profiles at high temporal resolution for simulating the performance of solar thermal systems. *Solar Energy*. 2015, vol. 111, pp. 43–52. issn 0038-092X. Available from doi: 10.1016/j.solener.2014.10.026.
6. Czech electricity, OTE, a.s. the; operator, gas market [https://www.ote-cr.cz/en]. [N.d.]. Accessed March 12, 2024.
7. Group, Axiom Energy. *MicroCHP Spec Sheet* [https://www.axiom-energy.com/_files/ugd/7a0ad4_a74da8efef054a6e9030649827b90169.pdf?index=true]. [N.d.]. Accessed March 11, 2024.
8. STIEBEL ELTRON spol. s r. o. *Technical data of an electric water heater* [https://www.stiebel-eltron.cz/cs/produkty-a-reseni/ohrev_vody/zasobnikove_ohrivacevody/stacionarni_ohrivacevodyod200l/shw-s/shw-200-s/technicka-data.product.pdf]. [N.d.]. Accessed March 22, 2024.
9. Panasonic. *EVERVOLT® Home Battery System Data Sheet* [https://ftp.panasonic.com/solar/brochure/home_battery_sheet.pdf]. [N.d.]. Accessed March 12, 2024.
10. Tesla. *Tesla Powerwall 2 Datasheet* [https://www.tesla.com/sites/default/files/pdfs/powerwall/Powerwall_2_AC_Datasheet_EN_NA.pdf]. [N.d.]. Accessed March 12, 2024.
11. Solution, LG Energy. *16H Prime data sheet* [https://www.lgessbattery.com/ImageServlet?imgPath=20230802154044964[20230802154044964].pdf&imageType=PRODUCT]. [N.d.]. Accessed March 12, 2024.
12. Solution, LG Energy. *10H Prime data sheet* [https://www.lgessbattery.com/ImageServlet?imgPath=20230802154016813[20230802154016813].pdf&imageType=PRODUCT]. [N.d.]. Accessed March 12, 2024.

13. Chen, Haisheng; Cong, Thang Ngoc; Yang, Wei; Tan, Chunqing; Li, Yongliang; Ding, Yulong. Progress in electrical energy storage system: A critical review. *Progress in Natural Science*. 2009, vol. 19, no. 3, pp. 291–312. ISSN 1002-0071. Available from DOI: 10.1016/j.pnsc.2008.07.014.

14. Matoušek, Jiří; Gärtner, Bernd. *Understanding and using linear programming*. Understanding and using linear programming. Berlin: Springer, 2007. Universitext. ISBN 3-540-30697-8.

15. Blum, Christian; Raidl, Günther R. *Hybrid Metaheuristics: Powerful Tools for Optimization*. Springer International Publishing, 2016. Artificial Intelligence: Foundations, Theory, and Algorithms. ISBN 9783319308838. ISSN 2365-306X. Available from DOI: 10.1007/978-3-319-30883-8.

16. Russell, Stuart; Norvig, Peter. *Artificial Intelligence: A modern approach, global edition*. Third. Pearson Education, Limited, 2016. ISBN 9781292153971.

17. Eiben, A.E.; Smith, J.E. *Introduction to Evolutionary Computing*. Second. Springer Berlin, Heidelberg, 2015. Natural Computing Series. ISBN 9783662448748. ISSN 1619-7127. Available from DOI: 10.1007/978-3-662-44874-8.

18. Kennedy, J.; Eberhart, R. Particle swarm optimization. In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. 1995, vol. 4, pp. 1942–1948. Available from DOI: 10.1109/ICNN.1995.488968.

19. Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2024. Available also from: https://www.gurobi.com.

List of Figures

List of Tables

List of Abbreviations

DHW domestic hot water
LP linear programming
mCHP combined heat and power unit
MILP mixed-integer linear programming
$c_{\mathbf{v}}$ velocity initialization coefficient in PSO
$c^{\text{lb}}_{\text{battery}}$ battery lower bound initialization coefficient
$c^{\text{ub}}_{\text{battery}}$ battery upper bound initialization coefficient
c_{mCHP} coefficient in probability of mutating variable y_t calculated as c_{mCHP}/T
$\mathcal{D}$ set of time-shiftable devices
D^d_i profile of time-shiftable device d
D^e_t electricity demand in time interval t
D^h_t DHW demand in time interval t
d^s_t electricity used by all time-shiftable devices in interval t, depends on z_d
E electricity produced by mCHP in one time interval
G natural gas consumed by mCHP in one time interval
H heat produced by mCHP in one time interval
I^e initial state of battery
I^h initial state of hot water buffer
L^e_c battery charging loss
l_d profile length of time-shiftable device d
L^e_d battery discharging loss
L^e_s battery storage loss
L^h_s hot water buffer storage loss
M^e_c battery maximum input power, upper bound for x^-_t
M^e_d battery maximum output power, upper bound for x^+_t
O^d_e end of operation period for time-shiftable device d
O^d_s start of operation period for time-shiftable device d
P^g price of natural gas
P_{cross} probability of crossover
P_{mut} probability of mutation
P^e_t price of electricity in time interval t
s^e_t state of battery in time interval t
s^h_t state of hot water buffer in time interval t
$\mathcal{T}$ set of time intervals
T number of time intervals
U^e battery capacity
U^h capacity of hot water buffer
x_t battery variable used in complete representation
x^-_t variable, charging of battery in time interval t
x^g_t variable, electricity used from the grid in time interval t
x^+_t variable, discharging of battery in time interval t
y_t variable, state of mCHP in time interval t
z_d variable, start of device d
σ_{battery} standard deviation for mutation of variables x_t
σ_{devices} standard deviation for mutation of variables z_d

A Attachments

A.1 Attached Files

Structure of the attachment:

Path	Description
`data`	data processing scripts, input data and data JSON schema
`experiments`	scripts for comparing algorithms and their results
`hyperparameters`	scripts for selection of hyperparameters and their results
`milp`	implementation of MILP and baselines used to compare algorithms
`solver`	implementation of local search, evolutionary algorithm and PSO
`install_requirements.jl`	dependencies for Julia
`requirements.txt`	dependencies for Python

A.2 User Documentation

To run the code in the Attachment A.1, installations of Julia programming language and Python are necessary. Required packages for Python are in `requirements.txt` and for julia in `install_requirements.jl`. They can be installed by running:

```
pip install -r requirements.txt
julia install_requirements.jl
```

Furthermore, GUROBI solver needs to be installed [19]. We ran the code on Linux with Python 3.11.8, Julia 1.10.2 and GUROBI 11.0.1.

Note that all the scripts in the attachment are expected to be launched from the root directory of the attachment. For example, script `create_datasets.py` in directory `data` should be launched as

```
python data/create_datasets.py
```

The datasets are created by Python script `data/create_datasets.py`, and `data/create_test2.py` crates the modified test dataset with every device copied 100 times. Because some data are downloaded from the internet, we included the datasets generated by these scripts in subdirectories `dev`, `test` `test2` of directory `data`, in case they are no longer accessible.

The optima for creating graphs in Chapter 8 and Chapter 9 are found by script `milp/baselines.py`.

Results from scripts in directory `hyperparameters` are used in Chapter 8. The following table shows what each script is used for.

File Name	Used For
`evolution_init.jl`	Table 8.3
`evolution_mutation.jl`	Table 8.4
`evolution_no_correction.jl`	Table 8.5 and Figure 8.3
`evolution_parameters.jl`	Table 8.2
`evolution_variants.jl`	Figure 8.4 and Figure 8.3
`hc_battery.jl`	Figure 8.1
`hc_operators.jl`	Table 8.1
`hc_variants.jl`	Figure 8.2
`init_params.jl`	initial parameters for random initialization
`pso_init.jl`	Table 8.7
`pso_no_correction.jl`	Table 8.9
`pso_parameters.jl`	Table 8.6
`pso_parameters2.jl`	Table 8.8
`pso_variants.jl`	Figure 8.5

Note that all of these scripts except `init_params.jl` need Julia to be run with argument `-p n`. This runs Julia with `n` worker processes. For example:

```
julia -p 6 hyperparameters/evolution_init.jl
```

Results from scripts `run_experiments.jl` and `run_milp.py` in directory `experiments` are used in Chapter 9, and `run_experiments.jl` also requires the argument `-p`.

Results of all of these scripts are included in the attachment.

A.3 Programmer Documentation

The implementation of MILP is in `milp/solve.py`. It contains a function that creates MILP model defined in Chapter 3 for a data instance. It is programmed in Python and uses solver GUROBI [19].

Data processing from Section 2.2 is implemented in files `data/preprocess.py` and `data/parameters.py`. These two files are then used to create datasets from Section 2.4 in `data/create_datasets.py`.

Implementation of local search, evolutionary algorithm and PSO is done in Julia. It is all inside one module `Solver` defined in file `solver/solver.jl`, and the implementation itself is separated into multiple files. File `solver/data.jl` contains loading of data instances and their representation. The function `load_data` has argument `schema_file`, which is a path to the JSON schema file. It can be found in `data/data.schema.json`. In `solver/solution.jl` is the representation of solutions, fitness and correction of infeasible solutions from Chapter 4. It also contains the initialization of solutions described in Subsection 5.1.2 and Section 6.1. Hill climbing (Chapter 5) is in `solver/hill_climbing.jl`. Evolutionary algorithm (Chapter 6) is in `solver/evolution.jl`, and `solver/pso.jl` contains PSO (Chapter 7).

This implementation uses the following abstract types:

- `Solution`: representation of a solution (Section 4.1)

- `Init`: initialization of a solution (Subsection 5.1.2 and Section 6.1)
- `Operator`: operator for the hill-climbing algorithm (Subsection 5.1.1)
- `Mutation`: mutation for the evolutionary algorithm (Section 6.4)
- `Particle`: representation of a particle in PSO (Section 7.1)
- `ParticleInit`: initialization of a particle (Section 7.3)

The Julia scripts that run the experiments run the algorithms for a short while before the actual experiments to ensure that the compilation time is not included in the results.