

Politecnico di Milano

Facoltà di ingegneria dell'informazione

Corso di laurea magistrale in ingegneria informatica



OPTIMIZATION HEURISTICS FOR RESIDENTIAL ENERGY LOAD MANAGEMENT

Relatore: Prof. E. Amaldi
Correlatore: Dott.ssa G. Carello

Tesi di laurea di
Claudio Giovanni Mattera
740334

A. A. 2011-2012

ABSTRACT

The MS thesis is concerned with the problem of scheduling the daily energy loads in a multihouse environment from the point of view of an energy retailer.

We assume that the residential users own a set of home appliances (washing machines, dishwashers, ovens, microwave ovens, vacuum cleaners, boilers, fridges, water purifiers, irons, TVs, personal computers and lights) that are supposed to be used during the day. Houses can also be equipped with Photo Voltaic (**PV**) panels, which produce energy in a discontinuous way, and batteries that allow the system to store and release energy when required. The day is subdivided into 96 timeslots of 15 minutes each. For each appliance, we suppose to know the load profile, that is, a set of successive timeslots with the corresponding amount of energy required.

Given the load profile of each appliance, the time windows in which the appliances must be executed, the physical characteristics of the batteries, the energy amount produced by the **PV** systems, the problem is that of scheduling the various appliances (assigning their starting timeslots) so as to minimize an appropriate objective function while respecting the maximum capacity of the meters (usually 3 kW). We consider minimizing the total maximum peak.

This Residential Energy Load Management Problem is a challenging extension of the classical Generalized Assignment Problem (**GAP**). Since the Mixed Integer Linear Programming (**MILP**) formulation can be solved within reasonable computing time only for small instances, we developed various methods to tackle medium-to-large size instances: a Greedy Randomized Adaptive Search Procedure (**GRASP**) to generate initial feasible solutions, a meta-heuristic à la Tabu Search (**TS**) to improve initial solutions, and other techniques based on the solution of reduced **MILP** problems. In the **TS** algorithm we proposed different types of moves (appliances shift, batteries charge or discharge...) to explore the neighbourhood.

We have tested our methods on a data set of 180 realistic instances with different number of houses (20, 200 and 400), **PV** panels and batteries. The solutions provided by the heuristics are compared with those obtained by solving the **MILP** model by using a state-of-the-art solver.

For instances without batteries all our heuristics yield high quality solutions – within 3% from the reference solution – in a short computing time for the largest instances. Heuristics that solve reduced **MILPs** achieved the same results even for instances with batteries.

SOMMARIO

La tesi riguarda il problema di organizzare i carichi di energia giornalieri in un ambiente multi case da punto di vista del grossista di energia.

Si assume che gli utenti domestici possiedono un set di dispositivi (lavatrici, lavastoviglie, forni, forni a microonde, aspirapolvere, scaldabagno, frigoriferi, purificatori per acqua, ferri da stiro, TV, computer e luci) che vengono utilizzati durante il giorno. Le case possono anche essere dotate di Pannelli Fotovoltaici (PV), che producono energia in modo discontinuo, e batterie che permettono al sistema di immagazzinare energia per rilasciarla quando necessario. Il giorno è suddiviso in 96 timeslot di 15 minuti ciascuno. Per ogni dispositivo si suppone di conoscere il profilo di carico, i. e., un set di timeslot successivi con il corrispondente ammontare di energia richiesta.

Dato il profilo di carico di ogni dispositivo, le finestre temporali in cui i dispositivi devono essere attivati, le caratteristiche fisiche delle batterie, l'ammontare di energia prodotta dai sistemi fotovoltaici, il problema è di schedare le varie attività (assegnando i timeslot di inizio) in modo da minimizzare una funzione obiettivo appropriata e da rispettare la capacità massima (solitamente 3 kW). Si considererà la minimizzazione del picco massimo totale.

Il problema della gestione del carico domestico è un'impegnativa estensione del classico Problema di Assegnamento Generalizzato (GAP). Dato che la formulazione di Programmazione Lineare Mista Intera (MILP) può essere risolta in tempo ragionevole solamente per piccole istanze abbiamo sviluppato diversi metodi per affrontare istanze medio-grandi: un algoritmo Greedy Randomized Adaptive Search Procedure (GRASP) per generare soluzioni iniziali ammissibili, una meta-euristica à la Tabu Search (TS) per migliorare le soluzioni iniziali, ed altre tecniche basate sulla risoluzione di problemi MILP ridotti. Abbiamo proposto diversi tipi di mosse (spostamento dei dispositivi, carica e scarica delle batterie...) per esplorare il vicinato nell'algoritmo TS.

Abbiamo testato i nostri metodi su un data set di 180 istanze realistiche con un diverso numero di case (20, 200, 400), pannelli fotovoltaici e batterie. Le soluzioni fornite dall'euristica sono confrontate con quelle ottenute risolvendo il modello MILP con un risolutore allo stato dell'arte.

Per istanze senza batterie tutte le nostre euristiche hanno prodotto soluzioni di alta qualità – entro il 3% dalle soluzioni di riferimento – in breve tempo di esecuzione per le istanze più grandi. Le euristiche che risolvono MILP ridotti hanno raggiunto gli stessi risultati anche per istanze con batterie.

CONTENTS

1	INTRODUCTION	1
1.1	Energy demand curve	2
1.1.1	Smooth demand curve	2
1.2	Smart grids	3
1.3	Energy Box	4
1.4	Dynamic pricing	5
1.4.1	The day ahead market	5
1.5	This thesis	6
1.5.1	Structure	6
2	RESIDENTIAL ENERGY LOAD MANAGEMENT PROBLEM	9
2.1	Residential appliances scheduling	9
2.1.1	An overview	9
	Photo Voltaic panels	11
	Selling energy to the network	11
	Batteries	11
2.1.2	Objectives	12
	Minimization of total cost	12
	Minimization of global maximal peak	12
	Tracking a given demand curve	12
2.1.3	Definition	12
2.1.4	An example	13
	Multiple houses	16
	Realistic example	16
2.2	Related and previous work	17
2.3	Generalized Assignment Problem	20
3	MIXED INTEGER LINEAR PROGRAMMING MODEL	21
3.1	Model	21
3.1.1	Sets	21
3.1.2	Parameters	22
3.1.3	Variables	23
3.1.4	Constraints	24
3.1.5	Objective functions	26
3.2	Compact MILP model	26
3.2.1	Constraints	27
3.2.2	Improvements	27
3.3	Final model	28
4	GENERATING AN INITIAL FEASIBLE SOLUTION	29
4.1	GRASP algorithm	29
4.2	GRASP for the Residential Energy Load Management Problem	30
4.2.1	Algorithm	30
4.2.2	Structure of a solution	30

	Energy profile	31	
4.2.3	Updating the energy profile	31	
	Adding a load	32	
	Removing a load	32	
	Batteries as loads	33	
4.2.4	GRASP in detail	34	
	Using batteries	34	
4.3	Objective functions	36	
4.3.1	Ideal demand curve	37	
	Maximal difference from ideal	37	
	p -norm of difference from ideal	37	
	Ideal curve in GRASP	41	
4.3.2	Implemented functions	41	
	Maximal aggregate peak	41	
	Maximal difference	41	
	Maximal difference and p -norm	43	
	Maximal difference plus p -norm	43	
	Maximal peak plus maximal difference plus p -norm	43	
4.3.3	Infeasible solutions	43	
4.4	Partial Linear Relaxation with reduced MILP	43	
5	TABU SEARCH HEURISTICS AND IMPROVING A SOLUTION	45	
5.1	Tabu Search	46	
5.2	Tabu Search for the Residential Energy Load Management Problem	47	
5.2.1	Shift move	48	
5.2.2	Swap move	48	
5.2.3	Battery move	48	
	Example	49	
5.2.4	MILP move	52	
5.2.5	MILP-batteries move	52	
5.2.6	MILP-zeroes-fixing move	52	
5.2.7	Large move	54	
5.2.8	Mixed move	54	
	Tabu Search control	54	
5.2.9	Tabu Moves	55	
5.2.10	Exceeding maximal local peak	57	
	Infeasible exploration	59	
5.2.11	Early stopping and diversification	60	
5.3	Local branching	60	
5.3.1	Local branching for the Residential Energy Load Management Problem	61	
	Refining	62	
6	COMPUTATIONAL RESULTS	65	
6.1	Objective functions	68	
6.1.1	p -norms	68	

6.2	GRASP	72
6.2.1	Filtering criteria	72
6.2.2	Infeasible generation	74
6.2.3	Batteries usage in GRASP	75
6.3	Tabu Search	76
6.3.1	Tabu Search with shift moves	76
6.3.2	Tabu Search with MILP moves	78
6.3.3	Tabu Search with mixed moves	78
6.3.4	Tabu Search with MILP-zeroes-fixing move and mixed moves	79
6.3.5	Tabu Moves	80
6.3.6	Early stop and diversification	80
6.3.7	Infeasible exploration	82
6.4	Partial Linear Relaxation and reduced MILP	82
6.5	Local branching	84
6.6	Comparison	86
6.7	Multi-threading	89
7	CONCLUDING REMARKS	91
7.1	Future work	92
A	IMPLEMENTATION	95
A.1	Data structures for efficient neighbourhood exploration	96
A.1.1	Incomplete energy profile	96
	Incomplete energy profile in other algorithms	100
A.1.2	Move semantic	100
A.2	Local search framework	100
A.2.1	The algorithm	101
	Early stop and diversification	101
A.2.2	Template policies	105
	Why template parameters?	105
	Policies	106
A.3	Multi-threading	106
	Bibliography	108

LIST OF FIGURES

Figura 1.1	Daily domestic energy demand	2
Figura 2.1	Example: activities load profiles	14
Figura 2.2	Example: initial demand curve	14
Figura 2.3	Example: demand curves for different starting time slot of D	15
Figura 2.4	Example: optimal demand curve	15
Figura 2.5	Example: multiple houses	16
Figura 2.6	Solutions demand curves (without batteries)	17
Figura 2.7	Solutions demand curves (with batteries)	17
Figura 4.1	Same maximal peak for different solutions	36
Figura 4.2	Same maximal difference for different solutions	38
Figura 4.3	Maximal difference for exponential ideal curve	38
Figura 4.4	Solutions with same maximal peak and difference from ideal	40
Figura 4.5	Solutions with diverse area of difference	40
Figura 4.6	Solutions with same area of difference	40
Figura 4.7	p -norms for various values of p	42
Figura 5.1	Battery move example	53
Figura 5.2	Mixed move flow chart without batteries	55
Figura 5.3	Mixed move flow chart with Battery moves	56
Figura 5.4	Mixed move flow chart with MILP-batteries moves	56
Figura 5.5	Mixed move flow chart with MILP-zeroes-fixing moves	56
Figura 5.6	Swapping two activities	58
Figura 5.7	Local branching iteration state chart	63
Figura 6.1	Box plot notation	67
Figura 6.2	GRASP with different objective functions, 200 houses	69
Figura 6.3	p -norms	70
Figura 6.4	GRASP filtering criteria, 200 houses without batteries	73
Figura 6.5	Feasible and infeasible GRASP	74
Figura 6.6	TS with shift moves, 400 houses without batteries	77
Figura 6.7	TS with MILP-zeroes-fixing move and mixed moves, typical local search progress	79
Figura 6.8	Tabu List usage	81
Figura 6.9	Infeasible exploration in TS, details	83
Figura 6.10	Comparison, gap from optimum	87
Figura 6.11	Comparison, computing time	88
Figura 6.12	Computing times when using multi-threading	89

Figura A.1	Local search framework's class diagram	102
Figura A.2	Concrete iteration types	105

LIST OF TABLES

Tabella 2.1	Oven's load profile	10
Tabella 2.2	Example: activities load profiles	13
Tabella 5.1	Move switch in case of empty neighbourhood	54
Tabella 6.1	Data set	66
Tabella 6.2	Reference solutions	67
Tabella 6.3	Objective functions ids	71
Tabella 6.4	GRASP filtering criteria	72
Tabella 6.5	Feasible and infeasible GRASP	74
Tabella 6.6	Batteries usage in GRASP	75
Tabella 6.7	TS with shift moves	77
Tabella 6.8	TS with MILP moves	78
Tabella 6.9	TS with mixed moves	78
Tabella 6.10	TS with MILP-zeroes-fixing move and mixed moves	79
Tabella 6.11	Early stop and diversification in TS	80
Tabella 6.12	Infeasible exploration in TS	82
Tabella 6.13	PLR and reduced MILP	84
Tabella 6.14	PLR and reduced MILP, instances solved	84
Tabella 6.15	Local branching	85

LIST OF ALGORITHMS

Algorithm 4.1	GRASP	35
Algorithm 5.1	Battery move, outer loops	50
Algorithm 5.2	Battery move, inner loops	51
Algorithm A.1	Neighbourhood's exploration, first version	97
Algorithm A.2	Neighbourhood's exploration, second version	97
Algorithm A.3	Neighbourhood's exploration, third version	98
Algorithm A.4	Neighbourhood's exploration, fourth and final version	99
Algorithm A.5	Complete local search	103
Algorithm A.6	Local search early stopping criterion	104

ACRONYMS

GRASP Greedy Randomized Adaptive Search Procedure

TS Tabu Search

TM Tabu Move

- TL Tabu List
- MILP Mixed Integer Linear Programming
- LP Linear Programming
- PLR Partial Linear Relaxation
- GAP Generalized Assignment Problem
- PV Photo Voltaic
- POD Plain Old Data
- CPU Central Processing Unit
- RTP Real Time Pricing
- PTA Peak To Average
- IBR Inclining Block Rates
- FIFO First-In First-Out
- EB Energy Box
- POSIX Portable Operating System Interface
- RELMP Residential Energy Load Management Problem

INTRODUCTION

In this chapter we introduce the background of the thesis. We mention the ongoing changes in the residential electric energy network, the importance of the demand curve in the contest of energy efficiency, the future opportunities available with the development of smart grids and dynamic pricing policies. Finally we state the objective of the thesis and its contribution to the field.

CONTENTS

1.1	Energy demand curve	2
1.1.1	Smooth demand curve	2
1.2	Smart grids	3
1.3	Energy Box	4
1.4	Dynamic pricing	5
1.4.1	The day ahead market	5
1.5	This thesis	6
1.5.1	Structure	6

Historically, electric networks had a top-down structure: the power plant produced energy, users signed a contract with the electrical company that specified only the maximal peak consumption, and they used energy without any kind of interaction with the network. Energy price was fixed and users had no reason to plan their energy usage, they just turned on their electrical devices whenever they needed them¹.

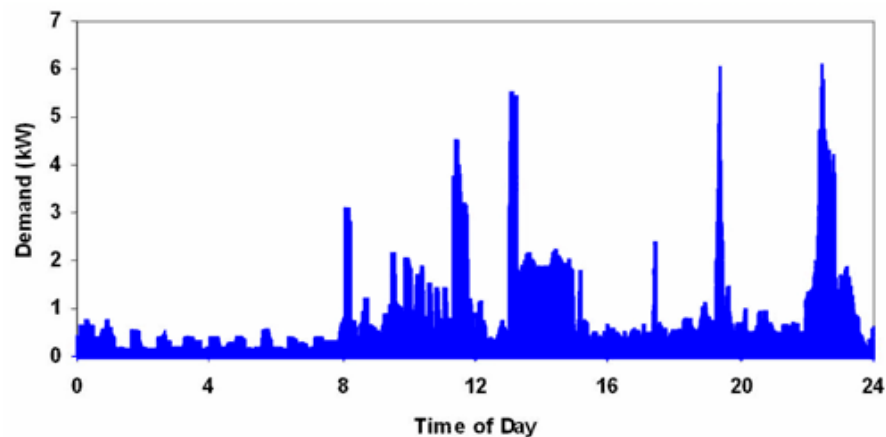
Since the electrical company had no way to control energy usage, there could happen that all users requested the maximal energy allowed by their contract, thus power plants needed to be constantly able to produce enough energy for this event – and the network itself needed to be able to sustain such energy flow. Residential energy usage has usually a periodic behaviour: people use little or no energy at night, and during the day they usually perform the same tasks (cooking, cleaning. . .) at roughly the same time. In Figure 1.1 we show the typical demand distribution during the day². We can see that there are peaks in energy consumption, for instance at eight in the morning most people use water boilers or microwave ovens to make breakfast, or at lunch time they use ovens. During other hours not much energy is used: peak usage is isolated, most of the time only a small fraction

Isolated peaks in daily domestic energy usage

¹ Dual rate tariffs had actually effect in making users aware of different energy prices. However, it is a very coarse grained tariff, as there are only two prices, one for each halve of the day. Moreover it is a static tariff, it does not reflect the actual state of the network and the *cost* of energy.

² Source: http://www.mpoweruk.com/electricity_demand.htm

Figure 1.1: Daily domestic energy demand



of this energy is requested. This also means that most of the time only a small fraction of the electrical network is used, which is therefore oversized.

1.1 ENERGY DEMAND CURVE

An electrical device requires energy in a continuous way, for this reason instead of the total *energy* consumption it is usually stated the *power* consumption, i. e. the amount of energy consumed per unit of time. A 1000 W device actually consumes 1000 J if it runs for 1 s, but if it runs for 1 h it will consume 3.6 MJ. If the power consumption is represented with respect to the time – or, equivalently, if the energy usage is discretized and the energy consumption is represented with respect to the time slices – the resulting graph is called the *energy demand curve*. We showed an example in Figure 1.1 (discretized for 2 min time slices).

The area below the graph between two points is the integral of the power – or the summation of energy slices – and it is the energy consumed between those two points. Thus, a peak is a time interval where much energy is consumed, and conversely a valley is a time interval where little energy is consumed. A 1000 W peak lasting 1 s consumes the very same amount of energy of a long 1 W valley lasting 1000 s: the energy demand curve does not represent how much energy is consumed, but *how* and *when* it is consumed.

1.1.1 Smooth demand curve

Smoothing³ the energy demand curve has various beneficial effects. As said before, the electrical network must be sized such as to sus-

³ *Smooth* is to be intended in the practical sense of *flat, regular*, without isolated peaks, not in the usual sense of infinitely continuously differentiable, i. e., belonging to C^∞ .

tain the maximal energy request. Oversized facilities are expensive to build and to maintain, and they often have to be in standby – consuming energy. A smooth demand curve means that the facilities are used in large fraction, and all the energy used to sustain the network is actually needed and used by the users.

Smoothing the energy demand allows to increase the number of users or the energy they can use without increasing the size of the existing facilities. This can be particularly important in highly densely populated areas like large cities, where it might just be impossible to deploy additional power plants, and also in developing countries, where most of the people is now starting having access to many modern energy hungry devices [Livengood and Larson, 2009, Black and Larson, 2007].

Besides helping in reducing facilities sizes, a smooth demand curve does even help in reducing carbon dioxide and other harmful emissions, as discussed in Mohsenian-Rad and Leon-Garcia [2010], Livengood and Larson [2009]. Environment friendly power plants – solar, wind and to some extent hydroelectric and ocean wave plants – handle well enough a low steady energy request, but they are not suited for short extremely high peaks. For such peaks environment unfriendly power plants – using fossil fuels – are needed.

Smooth demand curves are environmental friendly

1.2 SMART GRIDS

Lately the energy network – or *grid* – has been undergoing through some important changes: from a centralized top-down network, where there was one entity producing energy and many entities consuming it, it is becoming a decentralized grid, where, aside the larger power plants, there are many small energy producers, like **pv** panels or wind power plants. Potentially every single house, once only allowed to buy and consume energy, could produce and sell unused energy to the grid.

Although in some areas is already possible to sell exceeding energy to the grid, it should be noted that there is a transition in act. Most of the smart grid features are still unavailable at this moment, as the structure of the electric grid is still for the most part the old one we mentioned at the beginning of this chapter. However this is the direction toward which the grid is heading in the long run.

Perhaps the most important aspect of smart grids – the one that actually gives the name *smart* – is that, besides energy flow, there is also an *information flow* across the grid. The different entities in the grid can communicate with each other and modify their actions on the grid accordingly. For instance, if during a peak hour the main power plant reaches its production limit, it can asks the other entities to slightly reduce their energy consumption. Conversely, during an extremely sunny day few houses equipped with **pv** panels may pro-

Information flows across smart grids

duce more energy than the houses themselves need, so they sell the energy back to the grid and the main power plant can decrease its production.

1.3 ENERGY BOX

An Energy Box (**EB**) is an appliance that controls the electrical devices of a house, or a group of houses, proposed by [Livengood and Larson \[2009\]](#):

The Energy Box is proposed as a 24/7 background processor operating on a local computer or in a remote location, silently managing one's home or small business electrical energy usage hour-by-hour and even minute-by-minute.

Different levels of control may be implemented, but in principle the **EB** functions are:

- Detecting the power consumption of each device;
- Detecting the power production of **PV** panels or wind power plants;
- Knowing the energy requested by each device at each phase of its functioning;
- Detecting grid's changes of state, e. g. an energy shortage or an energy abundance, and changes of energy's price;
- Asking the user to specify an execution window for each device;
- Switching on and off devices according to some scheduling.

The **EB** would take care of actually deciding when to switching a device on, according to the execution window specified by the user. The user sets the soonest and latest times the device should start and the soonest and latest times it should finish. The **EB** is free to switch the device on at any time, as long as it respects these constraints. Another approach is to define a preferred time and a maximal slack: whenever a device is started before or later than its preferred time the user's comfort decreases, and the **EB** would need to maximize user's comfort.

The **EB** lifts the users of the burden of explicitly manage the usage of every electrical device. Even if the network could communicate to the houses, users should be ready to respond to changes of the state of the network, and turn on or off their device. With the **EB** they only have to set some preferences, and the devices are automatically managed without the need for interaction.

1.4 DYNAMIC PRICING

As said before, in the traditional electrical network the energy's price is fixed. Users are not interested in how they use energy, any way they do it they always pay the same, so they do it in the most convenient way. Introducing a dynamic pricing would make the users aware of different ways to use energy: the simplest way – which is actually used by some energy providers, e. g., the dual rate tariff is quite popular in Italy – is to differentiate the price in peak hours and non-peak hours. This way the users are induced to use some of their devices when energy is less expensive. Of course some activities need to be done at fixed time, so users will likely continue to use the cooker at launch and dinner time, but some other will be shifted, for instance users may decide to do the laundry at late evening.

*Dynamic pricing
raises users
awareness*

The main purpose for dynamic pricing is to smooth the energy demand curve (Figure 1.1) by shaving the peaks and filling the valleys.

1.4.1 *The day ahead market*

In the day ahead market policy the focus is on a group of houses. There are three main entities:

HOUSE OWNERS The users who consume energy with their electrical devices;

ENERGY PRODUCER The main entity who produces the energy in the grid (a power plant company);

ENERGY RETAILER An intermediate entity who buys energy from the producer and sells it to the single houses.

The retailer actually buys energy quotes a day ahead, i. e., it makes a promise to the producer that the day after it will request a given amount of energy, which it forecast it will be consumed by the users. The energy producer already knows how much energy will be selling to the retailer, it already knows how much energy it will have to produce. It no longer has to keep the power plants and the network in the condition of producing and dispatching the maximal amount of energy, so it might switch on only few plants or deactivate some energy routes. The producer is able to save money knowing in advance how much energy to produce, and this reflects on the energy's price: energy bought a day ahead is cheaper.

*Energy quotes are
bought a day ahead,
based on energy
demand forecast*

If for some reason the retailer did not produce a correct estimate of energy consumption, it is not able to keep its promises to the energy producer: it has to buy either more or less energy than previously stated. The energy producer allows this, but varying the amount of bought energy in the same day energy market is much more expensive than doing so in the day ahead market.

With the traditional electrical network such pricing policy would not be very effective, as there would be no easy way to predict how users will use energy the day after⁴. Smart grids and **EBS** on the other hand make possible to exploit this pricing policy: the users are asked to specify the execution window for each device for the day after, the centralized **EB** collects all data and output the predicted energy usage.

1.5 THIS THESIS

The Residential Energy Load Management Problem, i. e., finding the electrical devices scheduling that optimizes the shape of the demand curve, is very interesting and important with regard to the efficiency in domestic energy usage, and it has recently been the topic of many researches, as covered in Section 2.2 on page 17. As explained in Chapter 3, an existing Mixed Integer Linear Programming (**MILP**) model can be used to compute a solution which optimize a given objective function of the demand curve. Solving such model requires however extremely high computational effort, and for large instances it might even be impossible. The purpose of this thesis is to analyse and exploit the structure of the problem in order to develop an heuristic that is able to produce good solutions, e. g., solutions whose demand curve has low maximal peaks, in a reasonable time. The heuristic will be based on the aforementioned model, thus it will be possible to compare the results obtained through the two approaches.

As we shall see, most of the related researches focus on minimizing the total cost in a fully dynamic pricing policy scenario. A flat demand curve – which is the actual objective – is achieved as a side effect, since prices rise during peak hours and fall during valley hours. We will instead focus directly on flattening the demand curve, ignoring the energy price, and instead assuming a day ahead market scenario. Since energy is cheaper when bought on the day ahead and a flat demand curve means lower energy production cost, this might result in money saving as well.

As we observe in Section 2.3 on page 20, the problem can be modelled as an extension of the Generalized Assignment Problem (**GAP**), which is used to model a large number of phenomenons. Therefore it might be possible to adapt the resulting heuristic to problems with similar structures.

1.5.1 Structure

In Chapter 2 on page 9 we describe in detail the Residential Energy Load Management Problem. First we make an overview, then we give

⁴ Existing forecasting algorithms rely only on statistics over historical data. They ignore users preferences, and are ineffective when those are different from the expected.

a more formal definition, and lastly we make a complete example. Then we make a brief survey of previous and related work, and we mention the relation with the Generalized Assignment Problem.

In Chapter 3 on page 21 we introduce and describe in detail an extension of the Mixed Integer Linear Programming (**MILP**) model for the Residential Energy Load Management Problem proposed in Barbato et al. [2011a,b]. We also mention improvements to the model and computational issues for large instances.

In Chapter 4 on page 29 we describe a Greedy Randomized Adaptive Search Procedure (**GRASP**) to generate an initial feasible solution for the Residential Energy Load Management Problem problem. We define the structure of a solution and the procedures to update it when scheduling activities. We mention some issues in distinguish good solutions from bad solutions and we propose few objective functions to address these issues. We also describe an alternative method to generate an initial feasible solution through Partial Linear Relaxation (**PLR**).

In Chapter 5 on page 45 we describe a Tabu Search (**TS**) algorithm for the Residential Energy Load Management Problem. Initially we describe the characteristics of the **TS** meta-heuristic, then we explain in detail what kind of moves we implemented to explore the neighbourhood, how we control the algorithm to switch moves at runtime, how we enabled exploration of infeasible region and diversification. We also mention local branching algorithm as an alternative to **TS** to iteratively explore the solution space.

In Chapter 6 on page 65 we report and discuss the results obtained with the methods we developed. At first we describe the data set and show the reference results, then we report results for different kind of objective functions. We report results obtained with **GRASP** using different filtering criteria, enabling infeasible generation and enabling battery usage. We report results obtained with **TS**, for few variants and parameters: using shift moves, mixed moves, solving a reduced **MILP**, enabling diversification and infeasible exploration. We report results obtained with **PLR** with reduced **MILP** and local branching methods. In the end we compare the different methods and remark their advantages and disadvantages.

In Chapter 7 on page 91 we summarize the main contributions of the thesis and we mention a few directions for future work.

In Appendix A on page 95 we give a brief overview about the implementation and we discuss in detail some important or interesting aspects, such as: what data structures we used to efficiently explore the neighbourhood; how we implemented an extensible framework for local search methods.

RESIDENTIAL ENERGY LOAD MANAGEMENT PROBLEM

In this chapter we describe in detail the Residential Energy Load Management Problem. First we make an overview, then we give a more formal definition, and lastly we make a complete example. Then we make a brief survey of previous and related work, and we mention the relation with the Generalized Assignment Problem.

CONTENTS

2.1	Residential appliances scheduling	9
2.1.1	An overview	9
	Photo Voltaic panels	11
	Selling energy to the network	11
	Batteries	11
2.1.2	Objectives	12
	Minimization of total cost	12
	Minimization of global maximal peak	12
	Tracking a given demand curve	12
2.1.3	Definition	12
2.1.4	An example	13
	Multiple houses	16
	Realistic example	16
2.2	Related and previous work	17
2.3	Generalized Assignment Problem	20

2.1 RESIDENTIAL APPLIANCES SCHEDULING

The problem we deal with in this thesis is to find a daily schedule for electrical devices used in a set of cooperating houses so as to optimize a given objective function that is of interest for the energy retailer or the users. A MILP model for the problem is described in Chapter 3 on page 21.

If the energy demand curve is known a day ahead it is possible to effectively exploit the day ahead market without having to continuously adjust the energy consumption.

2.1.1 An overview

Electric devices in a house are called *appliances*. The day is divided in small time chunks, called *time slots*, and each appliance has a fixed duration measured in time slots.

Users specify a desired *execution window* for each appliance, i. e.

Day is discretized in time slots

Appliances must start in a user specified window

Time slot	1	2	3	4	5	6	7	8	9
Energy	800	800	800	800	800	800	0	0	800

Table 2.1: Oven’s load profile

the interval when the appliance can run. More precisely, for each appliance there is a minimal starting time – before that it cannot start – and a maximal ending time – by then it has to finish. Usually there is some flexibility, users can specify a more or less broad execution window. For instance if user comes home at 17:00 and wants to end dinner not later than 20:30 then the oven may be scheduled on any time after 17:15 and has to finish by 19:45.

We limit ourselves to the case where each appliance is executed exactly once during the day, no re-use nor skip is taken into consideration. Actually this is not a limitation since both two cases can be reduced to single execution appliances:

- if an appliance should run twice or more – the user might want to use the electric burner at breakfast, lunch and dinner – it is enough to introduce two or more identical appliances with non-overlapping execution windows;
- if an appliance should not run during a day – the oven is typically used more seldom than every single day – it is enough to set its length to zero.

Appliances consume amounts of energy at each time slot after they start

The execution of an appliance is divided into phases, and in each phase electric energy is consumed. A *phase* is a time interval of the same length of a time slot, and the set of all phases along with the energy consumed is called *load profile*. For instance the oven might consume 800 W during the first 30 minutes it is on, then it reaches the desired temperature and it stops using energy; but then the heat dissipates and it consumes again energy after 10 minutes for other 5 minutes. With 5 minutes long time slots the oven would have a 9 phases load profile, shown in Table 2.1.

When the execution of different appliances overlaps the total energy consumption for the house is of course the sum of the single appliances energy consumptions: using the microwave oven while watching TV requires more energy than doing only one thing at time. It is not possible to use arbitrarily as many appliances as possible however, the contract with the local energy retailer always sets a limit for residential usage, usually imposing a fixed limit on the absorbed power. Though it is sometime possible to exceed this limit we assume that the limit is strictly enforced.

It is possible to try to smooth the house’s energy demand curve by shifting appliances, filling one’s valleys with another’s peaks. Unfortunately, as pointed out in [Barbato et al. \[2011b\]](#), a single house does

A single house can only gain modest improvements

not usually have enough flexibility in its scheduling to achieve sensible improvements. When users collaborate, however, one's peaks can be routed to another's valleys: while energy curves of single houses may still be non-smooth, by combining them it is possible to make the aggregate curve sensibly smooth. A pair house, appliance is called *activity*.

Collaboration among different houses allows greater improvements

Photo Voltaic panels

Some houses may have Photo Voltaic (PV) panels that supply energy in a non predictable way during the day. Though in the model the source is called PV panels, it could also be wind power, or a mixture of different sources.

Obtaining reliable forecasting for PV panels energy production is an open problem [Picault et al., 2010], but in this thesis we assume that the energy production is known the day ahead. This energy can then be used for appliances instead of buying energy from the network.

Selling energy to the network

Whenever in a time slot PV panels produces more energy than it is required from the appliances the exceeding energy is sold to the network. Sometimes the contract between the energy retailer and the users sets an upper limit on the amount of energy that is sold in a time slot, as for bought energy.

Batteries

Some houses may have batteries that can be used to store energy for later usage. Batteries are still very rare to be found nowadays, but in the future this could change, especially with electric cars becoming more and more popular: an electric car can in fact be used as a battery.

Energy can be stored in batteries for later use

Batteries are in principle very simple entities: they charge, requiring energy, and they discharge, providing energy. More precisely a battery charges taking the energy from the network or from a PV panel; then it stores the energy, that can be later fetched from it to power an appliance or to be sold to the network. We assume that there is no significant energy loss in storing energy, even for a long period of time.

Batteries should not be charged and discharged frequently by small amount of energy because they suffer great wear from such usage. To avoid misusing a battery we introduced minimal charging and discharging constraints. Other constraints are related to physical limitations of batteries: maximal charging and discharging constraints and maximal and minimal energy storage. Finally, a battery can not both charge and discharge simultaneously during the same time slot.

We assume that each house that is equipped with batteries has the same set of batteries, and the instances used for testing are construc-

ted with only one battery. We also assume that all the houses with batteries have only one battery of the same type.

2.1.2 Objectives

Different objectives can be considered when managing in a centralized way the scheduling of a group of houses.

Minimization of total cost

Assuming the electric company uses a dynamic pricing policy, energy price varies during the day. At peak hours, when many users require energy, it is more expensive, while it usually gets cheaper during low peak hours. Moving heavy activities to low cost time slots allows to reduce the total cost.

As discussed in Chapter 3, the model assumes that the pricing policy is not actually dynamic. The electric company may impose a different price for each time slot, but these prices are assumed to be known a day ahead.

Minimization of global maximal peak

The global maximal peak is the maximal energy consumption by the entire aggregate of houses in a time slot, it is the highest point in the demand curve. Minimizing the global maximal peak leads to flatter demand curves, which has many advantages already discussed in Chapter 1.

Tracking a given demand curve

Given a desired demand curve it is possible to schedule the activities in order to make the actual demand curve as close as possible to it. Note that this is a generalization of the second objective, which can be achieved by following a flat demand curve.

2.1.3 Definition

We can now define the Residential Energy Load Management Problem. Given the following data:

- the set of houses;
- the set of appliances;
- the set of time slots;
- the execution window for each activity;
- the load profile for each appliance;

Activity	Length	Load				
A	5	1	1	1	1	2
B	3	2	2	2		
C	2	1	2			
D	4	3	3	1	1	

Table 2.2: Example: activities load profiles

- the limit on the energy absorbed from the network;

we want to compute the scheduling for all activity, i. e., the starting time slot for each activity, such as to optimize one of the mentioned objective functions.

We also consider two extensions, when **PV** panels and batteries are available. In that case we have the following additional data:


- the set of houses with batteries or **PV** panels;
- the energy produced by **PV** panels for each relevant house;
- the characteristics of each battery for each relevant house;


and we need to compute also the amount of energy stored and retrieved from batteries.

2.1.4 An example

Let's now consider a small single-house instance for an example. There are four different activities, whose loads we report in Table 2.2 and show in Figure 2.1. The total load is 23, and the day is divided in 9 time slots.

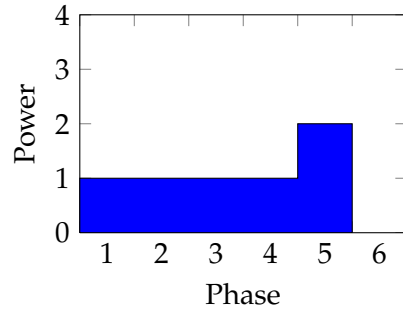
Let's assume that activities A, B and C start respectively in time slots 1, 3 and 7. We show the resulting demand curve in Figure 2.2.

Activity D could start in any time slots from 1 to 6. The resulting demand curves are shown in Figures 2.3a, 2.3b, 2.3c, 2.3d, 2.3e, 2.3f. The  area is the contribution of activity D to the demand curve.

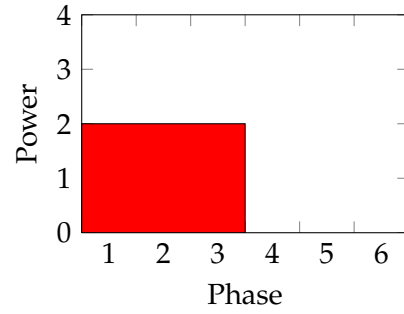
The six different starting time slots lead to different demand curves, whose maximal height is indicated with the dotted line . The best demand curves – with respect to the minimization of maximal peak – are the ones where activity D starts in time slots 4 or 5, and they have maximal height 4. An even better scheduling would be the one that we show in Figure 2.4, whose maximal height is 3.

In this simplified example activity D could start in any time slot. However, activities usually have a limited execution window, e. g., activity D must start after time slot 1 and finish by time slot 8. In such case, solutions in Figures 2.3a, 2.3b and 2.3f would be infeasible.

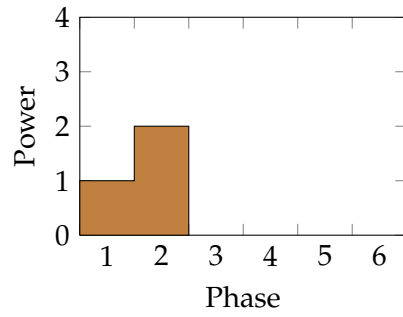
Figure 2.1: Example: activities load profiles



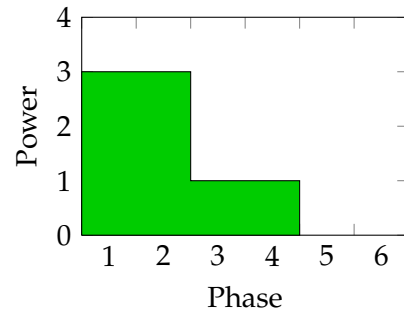
(a) Activity A load profile



(b) Activity B load profile



(c) Activity C load profile



(d) Activity D load profile

Figure 2.2: Example: initial demand curve

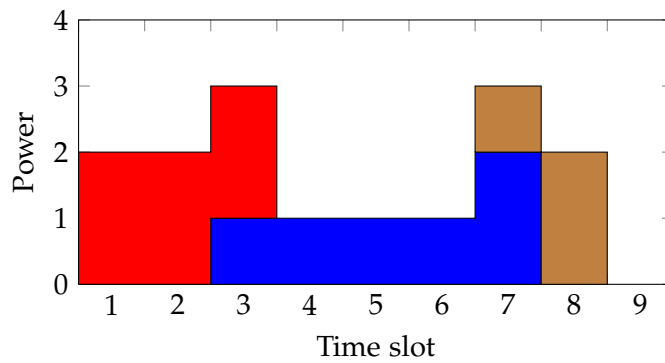


Figure 2.3: Example: demand curves for different starting time slot of D

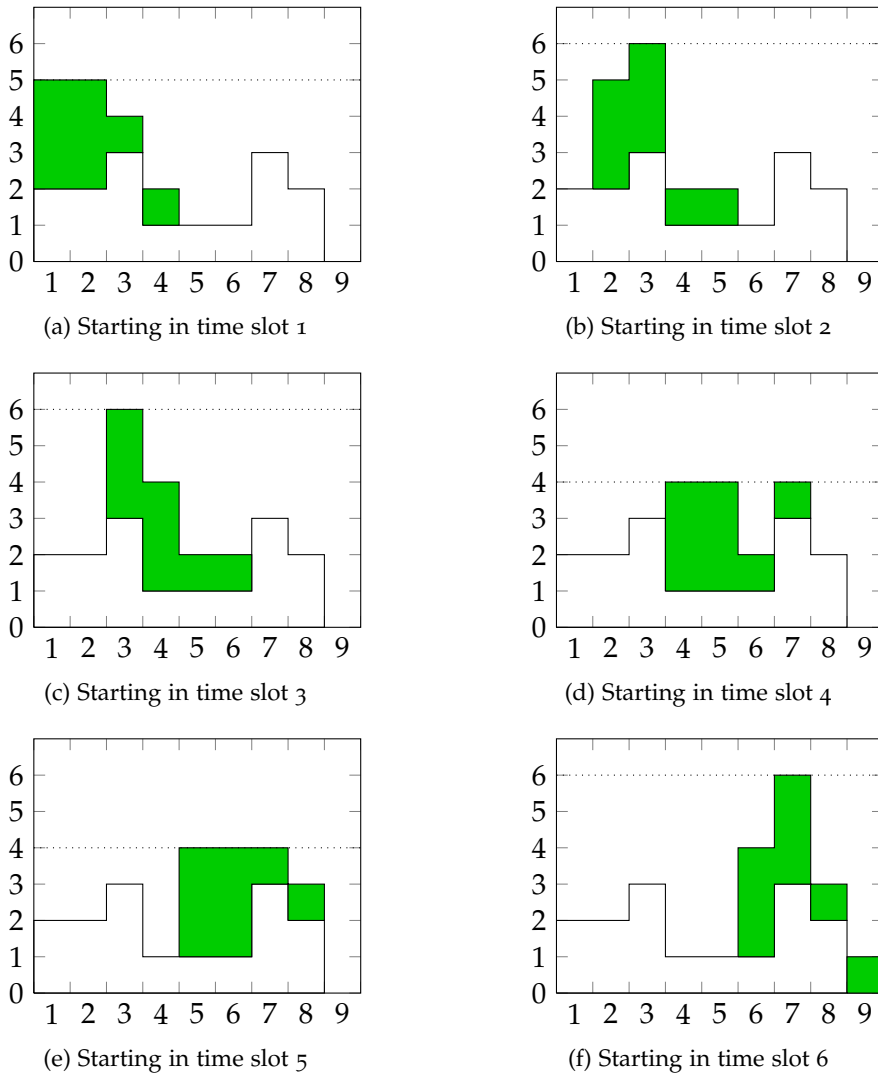


Figure 2.4: Example: optimal demand curve

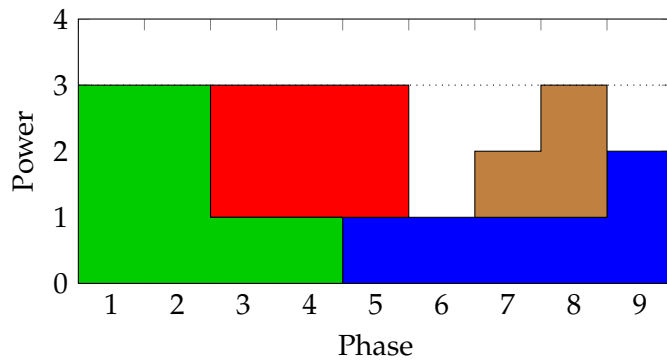
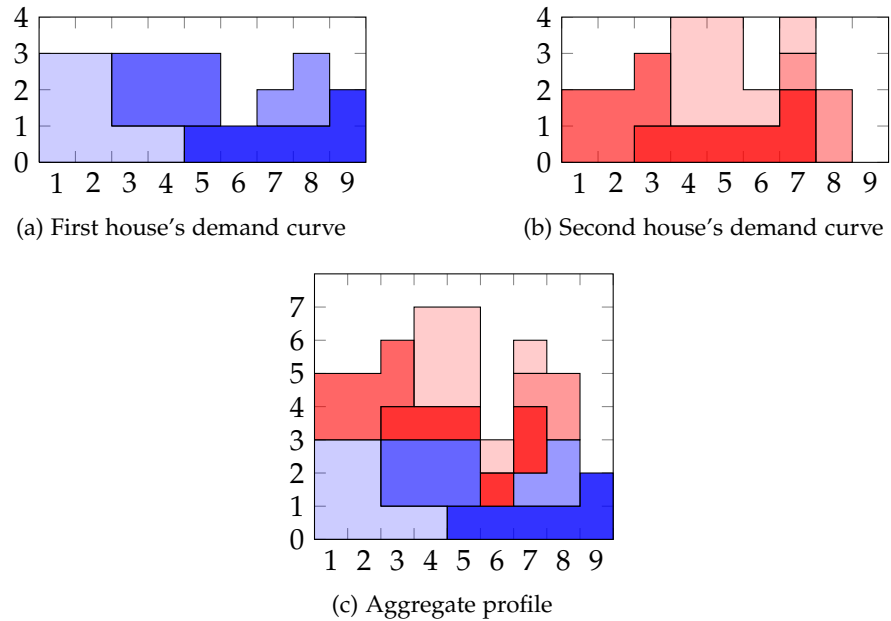


Figure 2.5: Example: multiple houses



Multiple houses

The previous example involved only a single house. In multiple houses scenario each house has its own local demand curve and the aggregate profile is the sum of all local profiles. We show an example with two houses in Figure 2.5. Each house has the same set of appliances of the previous example.

Realistic example

Finally, we show an example with realistic data. We show a low quality solution and a high quality one for the same instance of 200 houses, in the two cases when batteries are available or not. In Figures 2.6 and 2.7 we plot the demand curve for two different solutions of the same instance.

The low quality solutions are very similar. They have a few peaks, a large valley at the beginning of the day and also a large peak. The maximal peak is more than twice the ideal height.

The high quality solutions are instead different when batteries are available or not. In the latter case, by cleverly scheduling the activities, we got a solution flat and smooth. There still is a valley at the beginning of the day, but it is much narrower, and it is due to the fact that little or no activities can be scheduled at that time of night. When batteries are available it is even possible to fill this initial valley and shave the rest of the demand curve, since batteries have no constraints like activities execution window, and can be charged any

Figure 2.6: Solutions demand curves (without batteries)

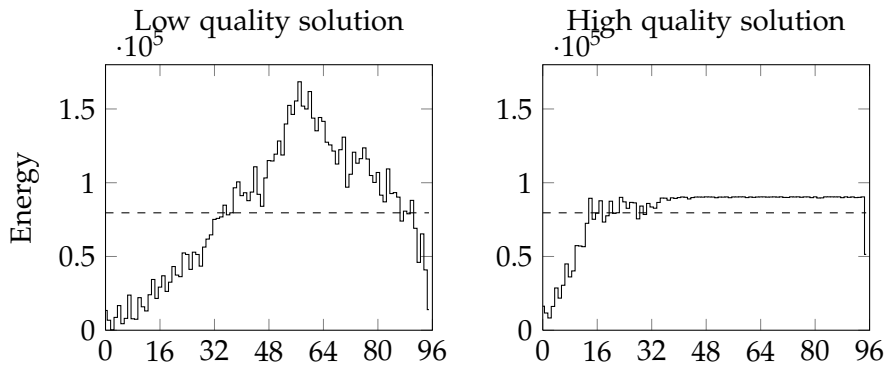
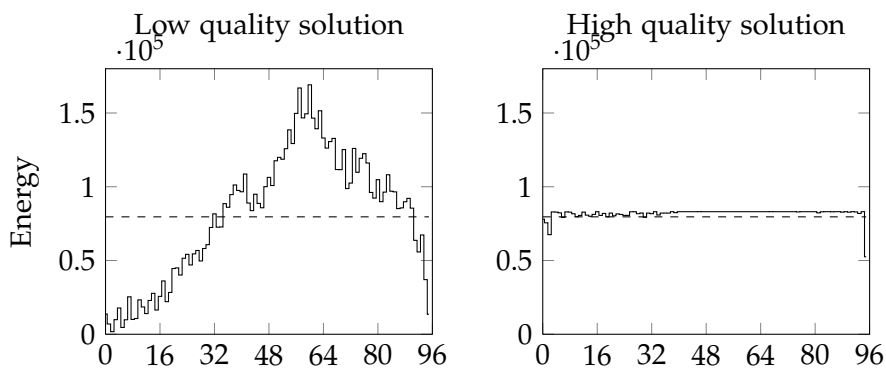


Figure 2.7: Solutions demand curves (with batteries)



time. Both high quality solutions are within 2% from the reference optimum.

2.2 RELATED AND PREVIOUS WORK

The ability to react to fast energy's price's changes is a requirement for the auto regulation of energy usage allowed by the smart grids. However domestic users lack this ability, and often even the time to reorganize the electrical devices schedules. With this motivation in [Livengood and Larson \[2009\]](#) the [EB](#) is introduced as a system to control electrical devices in response to dynamic pricing. Automatic scheduling is done via stochastic dynamic programming algorithms, but the [EB](#) supports other kind of algorithms in its algorithm bank, allowing to provide custom algorithms. This paper focuses on temperature controlling and battery charging loads.

A too high prices changes frequency is argued to cause fluctuations in the energy usage and thus advised to be avoided; a frequency of one hour is assumed. Finer grain control of order of minutes or even seconds is still possible: interruptible devices – e. g. clothes dryer – and devices which can function with different energy consumption – e. g. air conditioning – would be switched off or idled when the

grid is stressed, according to agreements between the users and the electric company.

In [Mohsenian-Rad and Leon-Garcia \[2010\]](#) an automated system for appliances scheduling in response to [IBR](#) combined Real Time Pricing ([RTP](#)) is proposed. In [Inclining Block Rates \(IBR\)](#) pricing policy energy's price grows if the hourly consumption is above a given threshold; users are then induced to use energy in a more regular way. A formal mathematical formulation is used for problem's definition and an algorithm for price prediction is used to forecast how the energy prices will vary in the short term future. The ability to predict prices has proved very effective in reducing the total cost for the users, as well as in reducing the [Peak To Average \(PTA\)](#) ratio of demand curve.

In [Kowahl and Kuh \[2010\]](#) the stochastic dynamic programming approach used in [Livengood and Larson \[2009\]](#) is discussed and argued to make assumptions – i. e. discretized states with known transitions probabilities – that do not hold for real world applications. A reinforcement learning approach is proposed, that is able to improve its performance over time. The softmax algorithm is used: decisions are taken from a pool with some probability. Whenever a decision lead to a lower cost its probability increases, and in the long run the probability distribution should favour optimal decisions. The algorithm was further adapted to reduce the number of probabilities by using Gaussian approximation and to simultaneously update neighbour states. The results approached the ones obtained with stochastic dynamic programming after about 100 days of learning.

In [Kishore and Snyder \[2010\]](#) is shown that, if the houses of a neighbourhood are controlled independently to shave the peaks in their energy demand curve by raising peaks energy's price, the result is that each house moves its heaviest loads to off-peaks intervals. The aggregate demand curve has then new peaks in those that were earlier off-peaks zones. A controller for multiple houses is then proposed. An upper limit over the aggregate energy consumption is imposed, and to each house is guaranteed a minimal amount of energy. If a house wants to request more energy it is made competing with possibly other houses with the same intent for the remaining energy available to the neighbour. A new single house dynamic programming scheduling algorithm is also proposed for accounting the case when the energy request is refused.

In [Ha et al. \[2006\]](#) a [TS](#) algorithm is proposed for the domestic energy scheduling and room temperature control. An anticipative layer predicts energy usage and production in the mid term future using learning mechanisms, and a reactive layer handles fast dynamics and short term fluctuations in predictions.

In [Agnētis et al. \[2011\]](#) the energy retailer is able to send price / volume signals to the users, i. e. asking to decrease the current load

under a certain threshold (the volume) in exchange for a reward (the price). Electrical devices are divided in *manageable* and *non manageable*, and the former is further divided in *adjustable* loads and *shiftable* loads – the former are devices that can function with different energy consumption, e. g. air conditioning; the latter are devices whose starting time may be shifted, e. g. washing machine. Non manageable loads – as well as distributed power generators like *PV* panels – are regarded as disturbance, but an estimate is assumed to be available.

A mathematical programming problem is defined using a weighted objective function involving three different objectives:

COST MINIMIZATION exploiting price's changes and retailer's price / volume signals;

MAXIMIZATION OF CLIMATIC COMFORT assumed to be directly related to adjustable loads;

SCHEDULING CONVENIENCE of the shiftable loads.

Different users may give different importance to each objective by setting different weights. Tests are run with four weights combinations (three unbalanced and one balanced profiles) and results for each profile, objective pair – e. g. cost minimization when using a pure scheduling convenience profile – are discussed.

In [Barbato et al. \[2011a,b\]](#) a **MILP** model is proposed for minimizing the total energy cost for cooperative and non-cooperative residential users. Two pricing policies are considered: a simple two-level peak and off-peak policy, and a dynamic policy where the price follows the grid's status (prices are assumed to be known beforehand). Users specify execution windows for the appliances with some degree of flexibility. Interruptible appliances are also considered in a variant of the model.

For the two-level pricing policy the saving is roughly independent of the execution windows flexibility, while for the dynamic pricing policy larger saving is possible for high flexibility scenarios. Interruptible appliances do not significantly improve the results. Batteries appear instead to be very effective when charged during off-peak hours and used to supply energy in peak hours.

In cooperative scenario a group of different houses are considered and their appliances are scheduled together, imposing a maximal peak constraint over the aggregate energy demand. Although the energy cost slightly raises with respect to the non-cooperative scenario, the demand curve is sensibly flatter and smoother.

This thesis is based on an extension of the model proposed in [Barbato et al. \[2011a,b\]](#), that will be described in detail in Chapter 3.

2.3 GENERALIZED ASSIGNMENT PROBLEM

The problem falls in the widely known category of Generalized Assignment Problems (GAPs). In the GAP there is a set of agents A and a set of jobs J [Martello et al., 2009]. Each job has a size s_j and each agent has a capacity c_a , any job can be assigned to any agent as long as the agent's total capacity is at least the sum of the capacities of assigned jobs. Each pair agent, job results in a gain $g_{a,j}$. The goal is to find the assignment of jobs to agents such that the total gain is maximal.

Setting the activity a to start in time slot t is equivalent to assign the job a to the agent t . Jobs sizes are given by activities load profiles, and agents capacities depend on the current scheduling and other parameters. Sizes and capacities are vectors rather than scalars, since they represent constraints that hold in multiple time slots.

MIXED INTEGER LINEAR PROGRAMMING MODEL

In this chapter we introduce and describe in detail an extension of the Mixed Integer Linear Programming ([MILP](#)) model for the Residential Energy Load Management Problem proposed in [Barbato et al. \[2011a,b\]](#). We also mention improvements to the model and computational issues for large instances.

CONTENTS

3.1	Model	21
3.1.1	Sets	21
3.1.2	Parameters	22
3.1.3	Variables	23
3.1.4	Constraints	24
3.1.5	Objective functions	26
3.2	Compact MILP model	26
3.2.1	Constraints	27
3.2.2	Improvements	27
3.3	Final model	28

3.1 MODEL

The Residential Energy Load Management Problem can be formulated as a [MILP](#) problem that can be solved with a [MILP](#) solver such as CPLEX.

3.1.1 Sets

The model has the following sets:

T The set of all time slots;

F The set of all phases¹;

H The set of all houses;

$H_p \subseteq H$ The set of all [pv](#) equipped houses;

$H_b \subseteq H$ The set of all battery equipped houses;

A The set of all appliances;

B The set of all batteries.

¹ Phases are used to define activities load profiles. They are somehow equivalent to time slots.

3.1.2 Parameters

For appliances

$ST_{h,a} \quad \forall h \in H, a \in A$ The latest starting slot for activity (h, a) ;

$ET_{h,a}^{\min} \quad \forall h \in H, a \in A$ The earliest ending slot for activity (h, a) ;

$ET_{h,a}^{\max} \quad \forall h \in H, a \in A$ The latest ending slot for activity (h, a) ;

$nt_a \quad \forall a \in A$ The length of appliance a ;

$lp_{a,f} \quad \forall a \in A, f \in F$ The power required by appliance a during its phase f .

The first three parameters define the activity's execution window. They are actually redundant since to decide the scheduling only the minimal and maximal starting slots are needed, plus the length, i. e.,

$$\begin{aligned} ST_{h,a}^{\min} &= \max(ST_{h,a}, ET_{h,a}^{\min} - nt_{h,a} + 1) \\ ST_{h,a}^{\max} &= ET_{h,a}^{\max} - nt_{h,a} + 1. \end{aligned} \quad (3.1)$$

However it might be more convenient for the user to specify all three of them.

For prices

$c_t \quad \forall t \in T$ Bought energy's price;

$g_t \quad \forall t \in T$ Sold energy's price.

For batteries

$\gamma_h^{\min} \quad \forall b \in B$ The minimal capacity for battery b ;

$\gamma_h^{\max} \quad \forall b \in B$ The maximal capacity for battery b ;

$\tau_h^{\min} \quad \forall b \in B$ The minimal amount of energy to charge for battery b ;

$\tau_h^{\max} \quad \forall b \in B$ The maximal amount of energy to charge for battery b ;

$\vartheta_h^{\min} \quad \forall b \in B$ The minimal amount of energy to discharge for battery b ;

$\vartheta_h^{\max} \quad \forall b \in B$ The maximal amount of energy to discharge for battery b ;

$ich_b \quad \forall b \in B$ The initial capacity for battery b (i. e. the capacity at time slot 0);

$\eta_n \quad \forall b \in B$ Efficiency for battery b .

For powers

$\pi_{h,t}^{pv}$ $\forall h \in H_p, t \in T$ The amount of energy produced by **PV** panel of house h at time slot t ;

$\pi_{h,t}^{in}$ $\forall h \in H, t \in T$ Maximal amount of energy the house h is allowed to buy at time slot t ;

$\pi_{h,t}^{out}$ $\forall h \in H, t \in T$ Maximal amount of energy the house h is allowed to sell at time slot t .

3.1.3 Variables

There are two types of variables: decisional variables and non decisional variables. The latter are defined in term of the former, so their value is given by the decisional variables. They exist only in order to simplify the model's expression, the model itself could be rewritten with only the decisional variables.

For activities and energy

$x_{h,a,t}$ $\forall h \in H, a \in A, t \in T$ This variable is set to 1 if activity (h, a) starts in time slot t , 0 otherwise;

$y_{h,t}$ $\forall h \in H, t \in T$ The amount of energy bought by house h at time slot t ;

$z_{h,t}$ $\forall h \in H, t \in T$ The amount of energy sold by house h at time slot t ;

$p_{h,a,t,f}$ $\forall h \in H, a \in A, t \in T, f \in F$ Energy used by activity (h, a) at time slot t if in that time slot it is at phase f , 0 otherwise.

Variable p is used in constraint 3.3. It was however aggregated in the compact model variant, as explained in Section 3.3.

For batteries

$e_{h,b,t}$ $\forall h \in H_b, b \in B, t \in T$ The energy stored in battery b of house h at time slot t ;

$\omega_{h,b,t}^c$ $\forall h \in H_b, b \in B, t \in T$ This variable is set to 1 if battery b in house h is charging at time slot t , 0 otherwise;

$\omega_{h,b,t}^d$ $\forall h \in H_b, b \in B, t \in T$ This variable is set to 1 if battery b in house h is discharging at time slot t , 0 otherwise;

$v_{h,b,t}^c$ $\forall h \in H_b, b \in B, t \in T$ The amount of energy charged in the battery b in house h at time slot t ;

$v_{h,b,t}^d$ $\forall h \in H_b, b \in B, t \in T$ The amount of energy discharged from the battery b in house h at time slot t .

3.1.4 Constraints

Single start for each activity

Each activity starts exactly once during the day, hence the value of $x_{h,a,t}$ is 1 only for one triple h, a, t – or, better, for activity (h, a) there is exactly one value for t such that $x_{h,a,t} = 1$, i. e.,

$$\sum_{t=ST_{h,a}}^{ET_{h,a}^{\max}-nt_a+1} x_{h,a,t} = 1 \quad \forall h \in H, a \in A. \quad (3.2)$$

Instantaneous power

At each phase of an activity the instantaneous power must be equal to the load profile for that phase, i. e.,

$$p_{h,a,t,f} = lp_{a,f} x_{h,a,(t-f+1)} \quad \forall h \in H, a \in A, t \in T, f \in F. \quad (3.3)$$

This formula picks the right phase at each time slot. If $x_{h,a,(t-f+1)} = 1$ for some t, f it means that the activity started f slots ago, hence in time slot t it is at phase f .

Contractual constraints

A single house is only allowed to buy and sell a limited amount of energy at each time slot, i. e.,

$$\begin{aligned} y_{h,t} &\leq \pi_t^{\text{in}} \quad \forall h \in H, t \in T \\ z_{h,t} &\leq \pi_t^{\text{out}} \quad \forall h \in H, t \in T. \end{aligned} \quad (3.4)$$

Balance equation

The law of conservation of energy must hold for this model: all the energy provided by any mean to a house must be somehow used. This constraint represents the balance between energy available to a house and energy used by that house, i. e.,

$$\underbrace{y_{h,t}}_A + \underbrace{\pi_{h,t}^{pv}}_B + \underbrace{\sum_{b \in B} v_{h,b,t}^d}_C = \underbrace{z_{h,t}}_D + \underbrace{\sum_{a \in A} \sum_{f \in F} p_{h,a,t,f}}_E + \underbrace{\sum_{b \in B} v_{h,b,t}^c}_F. \quad (3.5)$$

The terms in the left hand side of balance equation are the ones corresponding to available energy:

- A Energy bought from the net;
- B Energy generated by PV panels;
- C Energy discharged from batteries.

The terms in the right hand side of balance equation are the ones corresponding to used energy:

- D Energy sold to the net;
- E Energy consumed by appliances – $p_{h,a,t,f}$ is non zero only for the pairs (t, f) such that in time slot t the appliance is running phase f ;
- F Energy charged into batteries.

Of course it is possible that not every house $h \in H$ has PV panels or batteries; the constraint is actually split in these four ones:

$$\begin{aligned}
 A &= D + E & \forall t \in T, h \in H \setminus H_b \setminus H_p \\
 A + B &= D + E & \forall t \in T, h \in H \cap (H_p \setminus H_b) \\
 A + C &= D + E + F & \forall t \in T, h \in H \cap (H_b \setminus H_p) \\
 A + B + C &= D + E + F & \forall t \in T, h \in H \cap (H_b \cap H_p).
 \end{aligned}$$

Batteries constraints

During a given time slot a battery is either charging, discharging or idling, it is not possible to charge and discharge at the same time, i. e.,

$$\omega_{h,b,t}^c + \omega_{h,b,t}^d \leq 1 \quad \forall h \in H_b, b \in B, t \in T. \quad (3.6)$$

Batteries have minimal / maximal capacity, i. e.,

$$\begin{aligned}
 e_{h,b,t} &\leq \gamma_b^{\max} & \forall h \in H_b, b \in B, t \in T \\
 e_{h,b,t} &\geq \gamma_b^{\min} & \forall h \in H_b, b \in B, t \in T.
 \end{aligned} \quad (3.7)$$

Batteries have maximal charge / discharge rates. In order to prevent charging / discharging a battery of small amounts of energy, there exist additional constraints on minimal charge / discharge rates.

$$\begin{aligned}
 v_{h,b,t}^c &\leq \tau_b^{\max} \omega_{h,b,t}^c & \forall h \in H_b, b \in B, t \in T \\
 v_{h,b,t}^c &\geq \tau_b^{\min} \omega_{h,b,t}^c & \forall h \in H_b, b \in B, t \in T \\
 v_{h,b,t}^d &\leq \vartheta_b^{\max} \omega_{h,b,t}^d & \forall h \in H_b, b \in B, t \in T \\
 v_{h,b,t}^d &\geq \vartheta_b^{\min} \omega_{h,b,t}^d & \forall h \in H_b, b \in B, t \in T.
 \end{aligned} \quad (3.8)$$

Charge at time slot t is the charge at time slot $t - 1$ plus energy charged minus energy discharged – except of course at first time slot, i. e.,

$$e_{h,b,t} = \begin{cases} ich_b & \forall h \in H_b, b \in B, t = 1 \\ e_{h,b,t-1} + \eta_b v_{h,b,t}^c - \frac{1}{\eta_b} v_{h,b,t}^d & \forall h \in H_b, b \in B, t \in T \setminus \{1\}. \end{cases} \quad (3.9)$$

Maximal peak

This variable is the only one related to the entire set of houses, its lower bound is the maximum over all time slots of the energy bought by all houses, i. e.,

$$\text{peak} \geq \sum_{h \in H} y_{h,t} \quad \forall t \in T. \quad (3.10)$$

3.1.5 Objective functions

Minimize total cost

The objective is to minimize the difference between the money spent to buy energy and the money gained from selling energy, i. e.,

$$\min f(\underline{x}, \underline{v}^c, \underline{v}^d) = \sum_{t \in T} (c_t y_t - g_t z_t). \quad (3.11)$$

Minimize global maximal peak

The objective is to minimize the global maximal peak, i. e. the maximal height of demand curve, i. e.,

$$\min f(\underline{x}, \underline{v}^c, \underline{v}^d) = \text{peak}. \quad (3.12)$$

3.2 COMPACT MILP MODEL

During the thesis work we discovered a problem in the original model: Constraint (3.2) forced each activity to start exactly once inside its execution window, but it did not affect in any way what happens *outside* it. In a feasible solution some appliances might start many times².

Fixing this issue would have been trivial by just changing the constraint (3.2) to

$$\sum_{t \in T} x_{h,a,t} = 1 \quad \forall h \in H, a \in A, \quad (3.13)$$

so that only one variable $x_{h,a,t}$ were non zero for each activity (h, a) . But a better approach had been to partially rewrite the model in order to completely *remove* those useless variables. The new model introduces two new sets for each activity:

$T_{h,a}^s \subseteq T$ The time slots where activity (h, a) is allowed to start;

$T_{h,a}^c \subseteq T$ The additional time slots where activity (h, a) is allowed to run.

The two sets are contiguous and depend directly on the execution window and the length of each activity and are computed in a pre-processing phase, i. e.,

$$\begin{aligned} T_{h,a}^s &= [\max(ST_{h,a}, ET_{h,a}^{\min} - nt_{h,a} + 1), ET_{h,a}^{\max} - nt_{h,a} + 1] \\ T_{h,a}^c &= [ET_{h,a}^{\max} - nt_{h,a} + 1, ET_{h,a}^{\max}]. \end{aligned} \quad (3.14)$$

The $x_{h,a,t}$ variables are defined in term of $T_{h,a}^s$:

$$x_{h,a,t} = \begin{cases} 1 & \text{If activity } (h, a) \text{ starts in time slot } t \quad \forall t \in T_{h,a}^s. \\ 0 & \text{otherwise} \end{cases}$$

² They actually did. We noticed that by solving a reduced MILP: some appliances had multiple starting slots.

Now the guilty constraint (3.2) can be defined as

$$\sum_{t \in T_{h,a}^s} x_{h,a,t} = 1 \quad \forall h \in H, a \in A \quad (3.15)$$

without any other checks: it is not possible that some $x_{h,a,t}$ variable is 1 somewhere else, because *there exist no such variable outside the execution window*.

3.2.1 Constraints

Most of the constraints are now defined in term of $T_{h,a}^s$ and $T_{h,a}^c$.

Instantaneous power

The original constraint contained variable $x_{h,a,t}$ so it was updated to

$$\begin{aligned} p_{h,a,t,f} &= l p_{a,f} x_{h,a,(t-f+1)} \\ \forall h \in H, a \in A, t &\in T_{h,a}^s \cup T_{h,a}^c, \\ f &\in \{f : (t-f+1) \in T_{h,a}^s\}. \end{aligned} \quad (3.16)$$

3.2.2 Improvements

The compact model is equivalent to the (corrected) original model, but it has far less variables. MILP solvers decompose a single human readable constraint in many constraints for each variable, e. g., Constraint (3.4) would be decomposed in the $|H| |T|$ many constraints

$$\begin{aligned} y_{1,1} &\leq \pi_1^{in} \\ y_{1,2} &\leq \pi_2^{in} \\ &\vdots \\ y_{7,3} &\leq \pi_3^{in} \\ y_{7,4} &\leq \pi_4^{in} \\ &\vdots \end{aligned}$$

one for each pair (h, t) . Since some constraints are now defined in term of $T_{h,a}^s$ and $T_{h,a}^c$ instead of the whole T this means that many constraints were also removed. T^s is the subset of time slots where an activity can start. Most of the times it contains only few time slots, since execution windows rarely span more than a couple hours; the improvement is therefore quite significant.

Lesser variables and lesser constraints means lesser memory – and to some extent even faster computing time – so it is now possible to solve larger instances than the one solvable with the original model. For instance, we used CPLEX to check solutions feasibility during the

progress of the thesis: using the original model we could check only instances up to 35 houses, larger ones required too much memory; with improved model we could even check instances with 400 houses (CPLEX was executed on a machine with 2 GiB RAM).

3.3 FINAL MODEL

Another weakness of this mathematical model is that the variable $p_{h,a,f,t}$ is extremely large. A variant of the model was created by aggregating its values such that it depends only on the two variables h and t . p is defined as

$$p_{h,t} = \sum_{a \in A, f \in F, (t-f+1) \in T_{h,a}^s} l p_{a,f} x_{h,a,(t-f+1)} \quad \forall h \in H, t \in T.$$

Now the term E in the balance constraint (3.5) simply consists of $p_{h,t}$:

$$\underbrace{y_{h,t}}_A + \underbrace{\pi_{h,t}^{pv}}_B + \underbrace{\sum_{b \in B} v_{h,b,t}^d}_C = \underbrace{z_{h,t}}_D + \underbrace{p_{h,t}}_E + \underbrace{\sum_{b \in B} v_{h,b,t}^c}_F.$$

GENERATING AN INITIAL FEASIBLE SOLUTION

In this chapter we describe a Greedy Randomized Adaptive Search Procedure ([GRASP](#)) to generate an initial feasible solution for the Residential Energy Load Management Problem problem. We define the structure of a solution and the procedures to update it when scheduling activities. We mention some issues in distinguish good solutions from bad solutions and we propose few objective functions to address these issues. We also describe an alternative method to generate an initial feasible solution through Partial Linear Relaxation ([PLR](#)).

CONTENTS

4.1	GRASP algorithm	29
4.2	GRASP for the Residential Energy Load Management Problem	30
4.2.1	Algorithm	30
4.2.2	Structure of a solution	30
	Energy profile	31
4.2.3	Updating the energy profile	31
	Adding a load	32
	Removing a load	32
	Batteries as loads	33
4.2.4	GRASP in detail	34
	Using batteries	34
4.3	Objective functions	36
4.3.1	Ideal demand curve	37
	Maximal difference from ideal	37
	p -norm of difference from ideal	37
	Ideal curve in GRASP	41
4.3.2	Implemented functions	41
	Maximal aggregate peak	41
	Maximal difference	41
	Maximal difference and p -norm	43
	Maximal difference plus p -norm	43
	Maximal peak plus maximal difference plus p -norm	43
4.3.3	Infeasible solutions	43
4.4	Partial Linear Relaxation with reduced MILP	43

4.1 [GRASP](#) ALGORITHM

The greedy algorithm is an algorithm to generate an initial feasible solution step by step, making at each step the best local choice. Greedy Randomized Adaptive Search Procedure ([GRASP](#)), see e. g. [Feo and Resende \[1995\]](#), is a randomized extension of the greedy algorithm: at each step a set it makes a random choice among the best local choices.

Solution is built step by step, choosing among the best candidates

The procedure is repeated few times, and the best generated solution is returned.

4.2 GRASP FOR THE RESIDENTIAL ENERGY LOAD MANAGEMENT PROBLEM

4.2.1 Algorithm

The initial solution is empty, i. e., no activities are scheduled yet. We loop over all the activities¹, sorted by flexibility. For each activity we generate the set of candidates, i. e., the set of starting time slots belonging to the execution window of the activity. We evaluate each candidate with a given objective function and we filter out low quality candidates, according to a filtering criterion, and then we randomly pick the starting time slot.

4.2.2 Structure of a solution

In this section we describe the structure of a solution, the data structure used to evaluate a solution, and how we can add and remove activities while correctly updating such data structure.

A solution of the MILP model is completely determined by the values of the free variables:

$x_{h,a,t}$ determines when an activity starts during the day;

$\omega_{h,b,t}^d, \omega_{h,b,t}^c$ determines when a battery is charging or discharging;

$v_{h,b,t}^c, v_{h,b,t}^d$ determines how much a battery is charging or discharging.

All the other variables can be directly or indirectly obtained from these. For example it is possible to compute the energy bought from house h at time slot t using only the free variables as

$$y_{h,t} = \sum_{a \in A, t \in T_{h,a}^s \cup T_{h,a}^c} \sum_{f \in F} l p_{a,f} x_{h,a,(t-f+1)}.$$

In GRASP and the other methods however we make decision based on the value of some non free variables like $y_{h,t}$ or even their aggregates like $y_t = \sum_{h \in H} y_{h,t}$. Therefore we keep track of some of them as well.

A solution is then made of three parts:

1. the free variables: the starting slot for each activity and the battery flows for each house, battery and time slot;

¹ An activity is a pair of house, appliance: $(h,a) \in H \times A$.

2. some additional state variables: the batteries energy and the list of activities in each time slot;
3. an energy profile.

Energy profile

An *energy profile* can be thought of as the shape of a solution. It contains:

Energy profile represents the shape of a solution

- the energy bought by the single houses at each time slot;
- the energy sold by the single houses at each time slot;
- the energy bought by the whole aggregate at each time slot, i. e., the demand curve;
- the energy sold by the whole aggregate at each time slot.

We use the energy profile to evaluate a solution, and so in GRASP and other procedures we make decisions on which activity to move or which battery to charge and discharge depending on it.

The main reason to split the energy profile from the solution is that not only solutions have an energy profile: during a GRASP iteration we consider only one activity, and we create a set of *candidates to addition* as pairs of starting slot and resulting energy profile (plus battery information if needed).

4.2.3 *Updating the energy profile*

The energy profile is composed of three data structures:

BOUGHT_t energy bought by the aggregate at time slot t ;

SOLD_t energy sold by the aggregate at time slot t ;

$\text{FLOW}_{h,t}$ energy bought or sold by house h at time slot t .

Since a house can only either buy or sell energy in a time slot it is possible to use a single data structure: if flow is positive it means the house is buying $\text{Flow}_{h,t}$ energy during that time slot, otherwise it is selling $-\text{Flow}_{h,t}$ energy.

Two basic operations: adding and removing a load

There are only two basic operations that can modify the energy profile: a load ℓ is added to house h at one time slot t , or a load ℓ is removed from house h at one time slot t . Activities last usually more than one time slot, so adding or removing an activity means adding or removing a list of independent loads. In GRASP loads can not be removed, but only added.

Adding a load

At the beginning of **GRASP** no activity is scheduled yet, hence the energy profile is empty. If there are houses with solar panels all their energy is sold to the network. Then we start scheduling activities, adding loads to the profile.

When a load ℓ is added to house h at time slot t , it might be that the house is selling energy in that time slot. It means that there is still some unused energy from the solar panels

$$\text{EnergyAvailableInPV} \leftarrow \max(0, -\text{Flow}_{h,t})$$

which can be used for the new load instead

$$\text{EnergyFromPV} \leftarrow \min(\ell, \text{EnergyAvailableInPV}),$$

while the rest must be bought from the net

$$\text{EnergyFromNet} \leftarrow \ell - \text{EnergyFromPV}.$$

These equations handle gracefully all the three possible cases:

1. the energy for the entire load ℓ can be supplied by the solar panels, $\text{EnergyFromPV} = \ell$ and $\text{EnergyFromNet} = 0$;
2. the energy for the entire load ℓ must be bought from the net since no energy is left from solar panels, $\text{EnergyFromPV} = 0$ and $\text{EnergyFromNet} = \ell$;
3. some of the energy can be taken from the panels and the rest must be bought from the net.

Then we update energy profile with the newly added load:

$$\text{Bought}_t \leftarrow \text{Bought}_t + \text{EnergyFromNet}$$

$$\text{Sold}_t \leftarrow \text{Sold}_t - \text{EnergyFromPV}$$

$$\text{Flow}_{h,t} \leftarrow \text{Flow}_{h,t} + \ell.$$

Removing a load

In **GRASP** we only add loads to the energy profile. However, for the other methods that we developed and that we discuss in Chapter 5 we need to move a load from a time slot t_1 to a time slot t_2 , which is equivalent to removing it from the former and adding it to the latter. Thus it is enough to provide a procedure to remove a load ℓ from a house h at time slot t . We chose to cover the load removal procedure here, as it is the dual of load addition, even if it is irrelevant for **GRASP**.

When a load ℓ is removed from house h at time slot t the house might be buying some energy from the net

$$\text{BoughtSoFar} \leftarrow \max(0, \text{Flow}_{h,t}).$$

*Moving a load is
equivalent to
removing it from the
initial place and
adding it to the new
place*

If that quantity is less than ℓ then the rest of the energy must come from the solar panels

$$\text{FromPV} \leftarrow \max(0, \ell - \text{BoughtSoFar}),$$

hence by removing the load ℓ only the energy actually bought for the load will be saved

$$\text{NoLongerBought} \leftarrow \ell - \text{FromPV}.$$

These equations handle gracefully all the three possible cases:

1. all the energy was bought, $\text{FromPV} = 0$ and $\text{NoLongerBought} = \ell$;
2. all the energy came from the solar panels, $\text{FromPV} = \ell$ and $\text{NoLongerBought} = 0$;
3. some of the energy came from the panels, the other was bought from the net.

Then we update the energy profile after having removed the load:

$$\begin{aligned} \text{Bought}_t &\leftarrow \text{Bought}_t - \text{NoLongerBought} \\ \text{Sold}_t &\leftarrow \text{Sold}_t + \text{FromPV} \\ \text{Flow}_{h,t} &\leftarrow \text{Flow}_{h,t} - \ell. \end{aligned}$$

This procedure cleverly maximizes the solar panels usage: the energy is bought from the net only if there is not energy left in the panels.

Batteries as loads

Although in the previous sections we assumed that the sold energy came from the solar panels, this is not always true. $\text{Flow}_{h,t}$ represents just a pool of energy used for house h , if it is positive then some entity requires energy, if it is negative then some entity provides energy. When initially an empty solution is created the flows are set to the (negative) energy from the panels: they are the entities providing energy; activities are instead entities requiring energy. When both kind of entities are present at time slot t they compensate each other and the result is their algebraic sum.

Batteries are entities too. A charging battery requires energy, a discharging battery provides it (a battery is either charging or discharging during a time slot, it is not allowed to do both). So charging a battery of $v_{h,b,t}^c$ has the same effect of adding a load $\ell = v_{h,b,t}^c$ to house h at time slot t . On the converse discharging a battery of $v_{h,b,t}^d$ has the same effect of removing a load $\ell = v_{h,b,t}^d$ from house h at time slot t . Additional data structures must still be maintained because batteries has constraints on whether their load can be added or removed, but their effect on the energy profile is just a special case of adding and removing loads.

Using batteries is done by adding and removing loads

4.2.4 GRASP in detail

We show the complete GRASP in Algorithm² 4.1. Note that $ST_{h,a}^{\min}$ and $ST_{h,a}^{\max}$ are obtained by the model's data using (3.1).

Activities are sorted
by flexibility

The initial solution is empty, i. e., no activities are scheduled yet. The initial energy profile is flat and nil. At first we sort the activities by flexibility before scheduling them. Flexibility measures how much an activity starting time slot can be moved, and it is defined as the

$$\begin{aligned} \text{flexibility}_a &= ET_a^{\max} - \max(ET_a^{\min}, ST_a + nt_a) \\ &= ST_{h,a}^{\max} - ST_{h,a}^{\min} \end{aligned}$$

In this way the most *critical* activities, i. e., the ones that have a small execution window, are scheduled sooner than the others. A low-flexibility activity can start in only few time slots, and in latest stage of GRASP there is the chance that choosing any of such time slots result in an infeasible solution. High-flexibility activities can instead start in many more time slots, and therefore it is less likely that all of them result in an infeasible solution.

For each activity we generate a list of candidate starting time slots, corresponding to every time slot within the execution window. For each candidate we make a copy³ of the energy profile, add the activity in the candidate starting slot and compute the resulting energy profile. Each candidate is then evaluated with a given objective function, and low quality candidates are filtered out according to one of the following criteria:

Different criteria for
filtering out bad
candidates

1. keep the best N candidates;
2. keep all the candidates whose value is in the top α percentile, i. e., their value is less than

$$(1 - \alpha)(\text{worstValue} - \text{bestValue}) + \text{bestValue}, \quad 0 < \alpha < 1;$$

3. keep the best schedule and all the schedules with its same value (equivalent to the second criterion with $\alpha = 1$).

We then choose randomly the starting time slot for the current activity among the best ones. Finally we update the current solution and energy profile.

Using batteries

If there are batteries available it is possible use them to schedule activities in GRASP. We analyse the load profile of the activity to verify if

² We used the notation from Cormen et al. [2001] for the algorithms. In particular, the symbol \triangleright indicates a comment.

³ The actual algorithm makes an *incomplete* copy of the energy profile, see Section A.1.1 on page 96.

Algorithm 4.1 GRASP

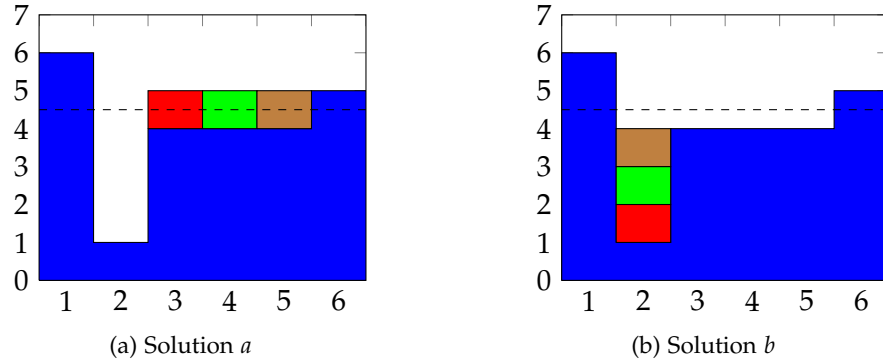
GRASP()

```

1  solution  $\leftarrow \emptyset$ 
2  energy-profile  $\leftarrow 0$ 
3  activities  $\leftarrow \{(h, a) \in H \times A\}$ 
4  SORT-BY-FLEXIBILITY(activities)
5  for  $(h, a) \in \textit{activities}$ 
6      do candidates  $\leftarrow \emptyset$ 
7           $\triangleright$  Loop over the execution window
8          for  $t \leftarrow ST_{h,a}^{\min}$  to  $ST_{h,a}^{\max}$ 
9              do  $m \leftarrow (h, a, t)$ 
10                  $p \leftarrow \text{COPY}(\textit{energy-profile})$ 
11                 ADD-ACTIVITY( $p, m$ )
12                 if FEASIBLE( $p$ )
13                     then candidates  $\leftarrow \textit{candidates} \cup (m, p)$ 
14                  $\triangleright$  All candidates were generated
15                 good-candidates  $\leftarrow \text{FILTER}(\textit{candidates})$ 
16                  $(m^*, p^*) \leftarrow \text{RANDOM}(\textit{good-candidates})$ 
17                 energy-profile  $\leftarrow p^*$ 
18                  $\triangleright$  Set the starting slot for activity  $(h, a)$ 
19                 solution  $\leftarrow \textit{solution} \cup m^*$ 
20                 UPDATE-ACTIVITIES-IN-EACH-TIME-SLOT(solution)
21 solution  $\leftarrow \textit{solution} \cup \textit{energy-profile}$ 
22 return solution

```

Figure 4.1: Same maximal peak for different solutions



the needed energy can be produced by batteries in some time slots. Batteries constraints must be satisfied in such timeslots, i. e., the discharged energy must be at least ϑ_b^{\min} . Then the sum of loads in these time slots is the total load to charge in the battery in earlier time slots, while loads in other time slots are normally purchased from the net (or taken from PV panels).

We check all previous time slots to see if it is possible to charge the battery with the required energy amount and if the capacity is enough to store it until the discharging time slots. For each alternative, if the resulting solution is feasible, we add it to the candidates list, hence battery-scheduled candidates compete with normal ones.

4.3 OBJECTIVE FUNCTIONS

In this section we mention some issues we experienced with objective functions and we investigate different objective functions in order to find the most suitable one for the Residential Energy Load Management Problem.

The objective function ranks solutions according on their quality. In a minimization problem solutions with low values have high quality.

At first we used the maximal peak of the aggregate demand curve; minimizing such objective function lead to flat demand curves. However, a large number of candidates – even if very different among themselves – had the same objective function value.

We provide an example of such behaviour in Figure 4.1: the two solutions have the same maximal peak, however solution *b* is more regular than solution *a*. If those were optimal solutions we could as well pick one randomly, but if we were going to schedule other activities we prefer the most regular one. However the maximal peak objective function can not distinguish between these two solutions. This effect is due to the *bottleneck objective function*, the solution's value depends only on the the single worst time slot.

4.3.1 *Ideal demand curve*

To overcome this problem we tested another objective function. Our aim is to minimize the maximal peak of demand curve, we want the most regular curve as possible. The *ideal* demand curve would completely flat, so that the energy requirement during the whole period would be constant. This coincides with the solution with minimal maximal peak. The demand curve is a rectangle, whose height is the demand curve's area – which is constant, it is the sum of all activities loads – divided by the length of the period, i. e., the number of time slots:

$$\text{ideal}_t = \frac{\sum_{h \in H, a \in A, f \in F} l p_{a,f}}{|T|}.$$

Ideally the demand curve should be flat

Maximal difference from ideal

Of course it is very unlikely that a scheduling corresponding to the ideal solution actually existed, however we can define a distance between the current solution and the ideal one, and minimize such distance. We started with minimizing the maximal difference from ideal

$$\begin{aligned} \text{diff}_t &= |\text{ideal}_t - y_t| \\ \min \max_t \text{diff}_t. \end{aligned}$$

Contrary to the expectation, the maximal difference objective function showed to perform extremely badly, much worse than the maximal peak one. The reason is due to the data instances used for tests: since very few activities could be started in the earliest time slots, very little amount of energy is consumed, and that amount is very far from ideal. Earliest time slots are thus a large lower bound to maximal difference objective function, as we show in Figure 4.2: the two solutions have nearly the same maximal difference value, but they are otherwise completely different. This is another case of bottleneck objective function.

Maximal difference still suffers from bottlenecks

The problem does not arise if the ideal solution is constructed such as to acknowledge that in earlier time slots energy consumption is low, as shown in figure 4.3.

p-norm of difference from ideal

Another approach other than minimizing the maximal difference is to minimize the difference's total area. This does not suffer from the previous problem, since the contribution from earlier time slots is only a part of the total area, and even if it has a large lower bound the total area can still be reduced by reducing contributions in other time slots.

Difference's area does not suffer from bottlenecks

Figure 4.2: Same maximal difference for different solutions

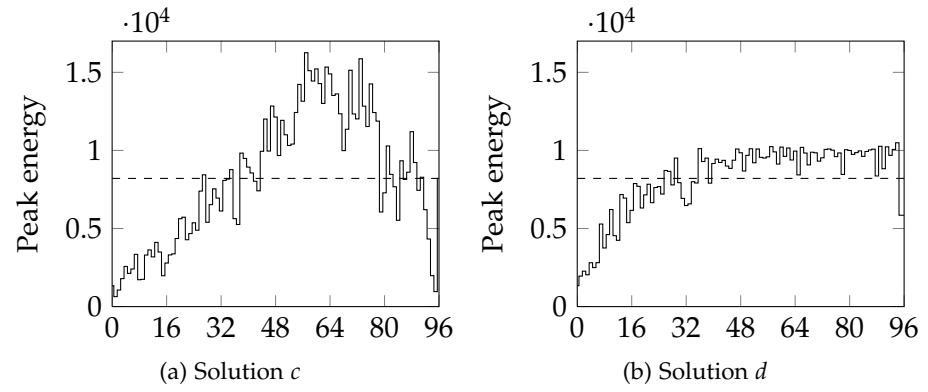
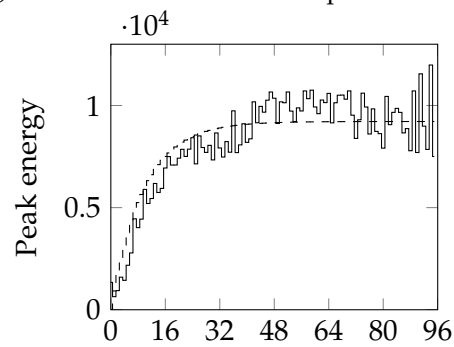



Figure 4.3: Maximal difference for exponential ideal curve



Two solutions are shown in Figure 4.4. In both solutions the maximal peak is 8 (in time slot 2) and the maximal difference from ideal is $34/8$ (in time slot 1), but solution e is more regular than solution f . The area bounded by the ideal and the actual curves, defined as the sum of difference in each time slots, i. e.,

$$\begin{aligned} \text{area} &= \sum_t |\text{diff}_t| \\ f(x_e) &= \frac{9}{4} + \frac{15}{4} + 6 \cdot \frac{1}{4} \\ &= 7.5 \\ f(x_f) &= \frac{9}{4} + \frac{15}{4} + 2 \cdot \frac{5}{4} + 2 \cdot \frac{1}{4} + 2 \cdot \frac{3}{4} \\ &= 10.5, \end{aligned}$$

and shown in red  in Figure 4.5, is instead very different, and captures more information about the solution.

The difference area has a huge drawback as well: a little difference over many time slots counts as a large difference over few time slots, the two solutions in Figure 4.6 have the same value. A better alternative is to generalize the area and use the p -norm of the difference

$$\|\text{diff}\|_p = \sqrt[p]{\sum_t |\text{diff}_t^p|},$$

so that short large differences produce a larger norm than long small differences. Different values for p give different results (see Section 6.1.1 on page 68), but the most effective ones seem to be 2 and 3. Note that the maximal difference is the limit of the p -norm

p-norm is effective to evaluate solutions

$$\max_t \text{diff}_t = \lim_{p \rightarrow \infty} \|\text{diff}\|_p = \|\text{diff}\|_\infty.$$

In the previous example the 2-norms are

$$\begin{aligned} f(x_e) &= \sqrt{\left(\frac{9}{4}\right)^2 + \left(\frac{15}{4}\right)^2 + 6 \cdot \left(\frac{1}{4}\right)^2} \\ &\simeq 4.42 \end{aligned}$$

$$\begin{aligned} f(x_f) &= \sqrt{\left(\frac{9}{4}\right)^2 + \left(\frac{15}{4}\right)^2 + 2 \cdot \left(\frac{5}{4}\right)^2 + 2 \cdot \left(\frac{1}{4}\right)^2 + 2 \cdot \left(\frac{3}{4}\right)^2} \\ &\simeq 4.85 \end{aligned}$$

$$\begin{aligned} f(x_g) &= \sqrt{4 \cdot (1)^2 + 4 \cdot (1)^2} \\ &\simeq 2.83 \end{aligned}$$

$$\begin{aligned} f(x_h) &= \sqrt{1 \cdot (4)^2 + 4 \cdot (1)^2} \\ &\simeq 4.72. \end{aligned}$$

Figure 4.4: Solutions with same maximal peak and difference from ideal

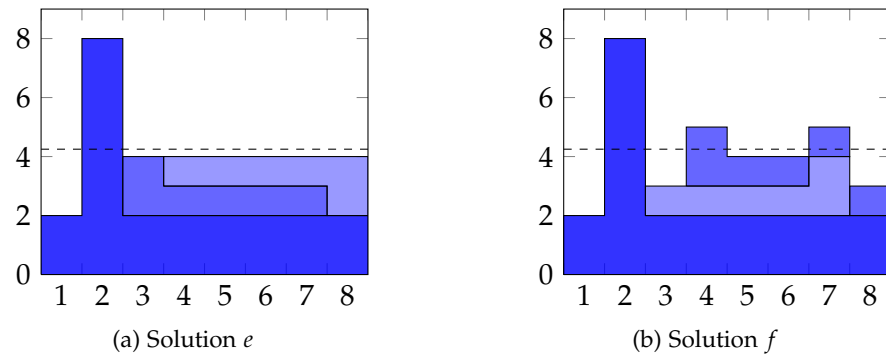


Figure 4.5: Solutions with diverse area of difference

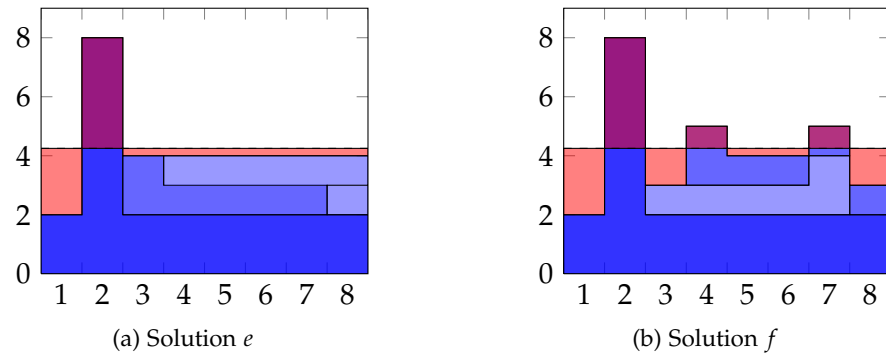
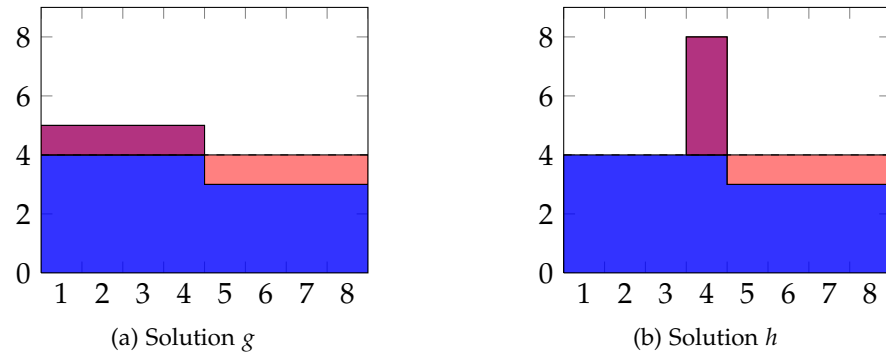


Figure 4.6: Solutions with same area of difference



For solutions e and f they are much closer than the 1-norms, since for $p > 1$ the p -norm is dominated by the largest coordinates, i. e., the first two time slots, which are the same for the two solutions. For the same reason for solutions g and h the 2-norms are no longer the same as the 1-norms, but instead they penalize solution d . For $p \rightarrow \infty$ the p -norm tends to the maximal coordinate, as shown in Figure 4.7.

The final steps we took in defining an objective function was to use information from all the functions described here to evaluate a solution.

Ideal curve in GRASP

During GRASP execution only a subset of activities are scheduled, hence the actual demand curve can be very far from the ideal one. The ideal curve area is taken as the energy of the activities *currently scheduled*.

4.3.2 Implemented functions

Different objective functions were implemented. The following notation is used:

$$\begin{aligned} \max(\underline{x}) &\triangleq \text{Max peak of solution } \underline{x} \\ \max(\underline{x} - I) &\triangleq \text{Max difference from ideal} \\ \|\underline{x} - I\|_p &\triangleq p\text{-norm of difference from ideal.} \end{aligned}$$

A solution is *lesser* than another if the former improves upon the latter.

Maximal aggregate peak

The original function from the mathematical model. Solution's value is its maximal aggregate peak. A solution is lesser than another if its maximal peak is lower.

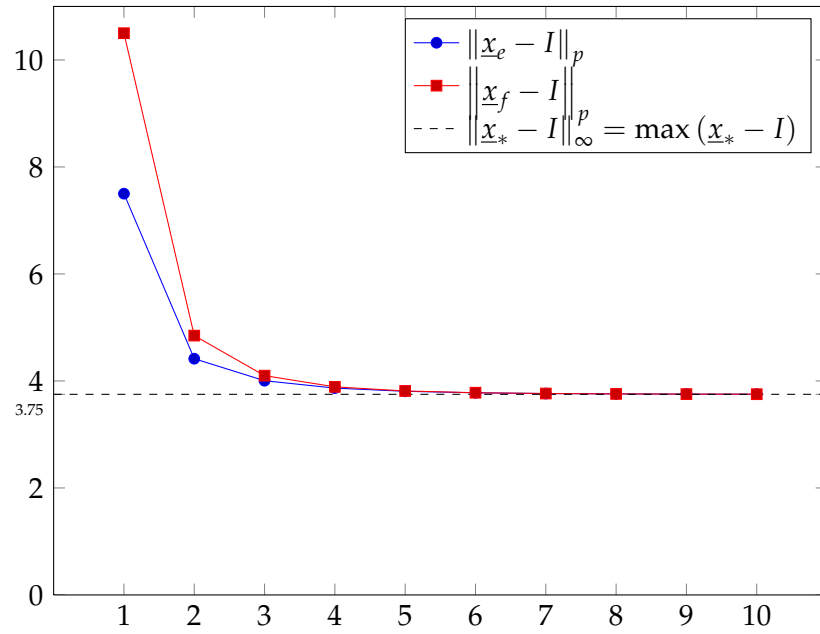
$$\begin{aligned} \text{Value}(\underline{x}) &= \max(\underline{x}) \\ \underline{x}_a < \underline{x}_b &\text{ if } \max(\underline{x}_a) < \max(\underline{x}_b) \end{aligned}$$

Maximal difference

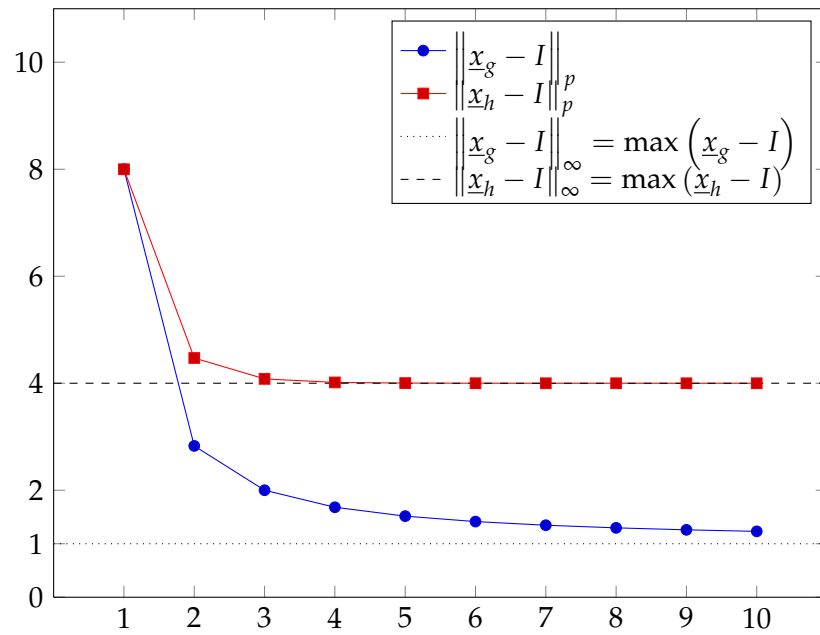
Solution's value is the maximal absolute value of difference from ideal. A solution is lesser than another if its maximal difference is lower.

$$\begin{aligned} \text{Value}(\underline{x}) &= \max(\underline{x} - I) \\ \underline{x}_a < \underline{x}_b &\text{ if } \max(\underline{x}_a - I) < \max(\underline{x}_b - I) \end{aligned}$$

Figure 4.7: p -norms for various values of p



(a) Solutions e and f



(b) Solutions g and h

Maximal difference and p-norm

Solution's value is the maximal absolute value of difference from ideal plus ten time the p -norm. A solution \underline{x}_a is lesser than a solution \underline{x}_b if \underline{x}_a 's maximal difference is strictly lesser than \underline{x}_b 's one, otherwise \underline{x}_a is lesser than \underline{x}_b if \underline{x}_a 's p -norm is lesser than \underline{x}_b 's one.

$$\text{Value}(\underline{x}) = \max(\underline{x} - I) + 10 \|\underline{x} - I\|_p$$

$$\underline{x}_a < \underline{x}_b \text{ if } \begin{cases} \max(\underline{x}_a - I) < \max(\underline{x}_b - I) & \max(\underline{x}_a - I) \neq \max(\underline{x}_b - I) \\ \|\underline{x}_a - I\|_p < \|\underline{x}_b - I\|_p \end{cases}$$

Maximal difference plus p-norm

Solution's value is the maximal difference plus the p -norm. A solution is lesser than another if its value is lower.

$$\text{Value}(\underline{x}) = \max(\underline{x} - I) + \|\underline{x} - I\|_p$$

$$\underline{x}_a < \underline{x}_b \text{ if } \max(\underline{x}_a - I) + \|\underline{x}_a - I\|_p < \max(\underline{x}_b - I) + \|\underline{x}_b - I\|_p$$

Maximal peak plus maximal difference plus p-norm

Solution's value is the maximal peak plus the maximal difference plus the p -norm. A solution \underline{x}_a is lesser than a solution \underline{x}_b if \underline{x}_a 's maximal peak is strictly lesser than \underline{x}_b 's one, otherwise \underline{x}_a is lesser than a solution \underline{x}_b if \underline{x}_a 's maximal difference is strictly lesser than \underline{x}_b 's one, otherwise \underline{x}_a is lesser than \underline{x}_b if \underline{x}_a 's p -norm is lesser than \underline{x}_b 's one.

$$\text{Value}(\underline{x}) = \max(\underline{x}) + \max(\underline{x} - I) + \|\underline{x} - I\|_p$$

$$\underline{x}_a < \underline{x}_b \text{ if } \begin{cases} \max(\underline{x}_a) < \max(\underline{x}_b) & \max(\underline{x}_a) \neq \max(\underline{x}_b) \\ \max(\underline{x}_a - I) < \max(\underline{x}_b - I) & \max(\underline{x}_a - I) \neq \max(\underline{x}_b - I) \\ \|\underline{x}_a - I\|_p < \|\underline{x}_b - I\|_p \end{cases}$$

4.3.3 *Infeasible solutions*

It is possible to instruct [GRASP](#) to generate solutions where the maximal local peak Constraint (3.4) does not hold. This is useful when we improve such solutions with local search using infeasible exploration, see Section 5.2.10 on page 57.

4.4 PARTIAL LINEAR RELAXATION WITH REDUCED MILP

To generate good initial feasible solutions we can also use the Partial Linear Relaxation (PLR) with reduced MILP procedure.

Integrality constraint is released to generate a reduced problem, which is solved with time limit

In the original **MILP** model all the $x_{h,a,t}$ variables are binary integer, i. e., they can only assume 1 or 0 values. This constraint is relaxed, producing a Partial Linear Relaxation of the original problem (variables $\omega_{h,b,t}^c$ and $\omega_{h,b,t}^d$ are still integer, but they are much fewer than variables $x_{h,a,t}$). This **MILP** problem can be solved to optimal in very short time with a **MILP** solver, at price that those variables might assume any real values in $[0, 1]$. Due to the constraint (3.2) (or (3.13)), however, we have that

$$\sum_t x_{h,a,t} = 1 \quad \forall h, a \in H, A.$$

In practice most of the $x_{h,a,t}$ variables are set to 0, which is already an integer. We fix a random subset of those variables to 0, we enforce the integrality constraint, and we solve this reduced **MILP** with a given time limit. This time the solution is feasible, and – although unlikely optimum – it might have high quality.

By fixing a variables $x_{h,a,t}$ to 0 we forbid activity (h, a) to start in time slot t . It can still start in any other feasible time slot, so even by fixing few of them, we are not making the scheduling too rigid.

TABU SEARCH HEURISTICS AND IMPROVING A SOLUTION

In this chapter we describe a Tabu Search (**ts**) algorithm for the Residential Energy Load Management Problem. Initially we describe the characteristics of the **ts** meta-heuristic, then we explain in detail what kind of moves we implemented to explore the neighbourhood, how we control the algorithm to switch moves at runtime, how we enabled exploration of infeasible region and diversification. We also mention local branching algorithm as an alternative to **ts** to iteratively explore the solution space.

CONTENTS

5.1	Tabu Search	46
5.2	Tabu Search for the Residential Energy Load Management Problem	47
5.2.1	Shift move	48
5.2.2	Swap move	48
5.2.3	Battery move	48
	Example	49
5.2.4	MILP move	52
5.2.5	MILP-batteries move	52
5.2.6	MILP-zeroes-fixing move	52
5.2.7	Large move	54
5.2.8	Mixed move	54
	Tabu Search control	54
5.2.9	Tabu Moves	55
5.2.10	Exceeding maximal local peak	57
	Infeasible exploration	59
5.2.11	Early stopping and diversification	60
5.3	Local branching	60
5.3.1	Local branching for the Residential Energy Load Management Problem	61
	Refining	62

In local search methods we start from an initial feasible solution and we improve it iteratively. At each iteration, given the current solution \underline{x}_k , we generate its *neighbourhood*

$$N_k = \{ \underline{x} : \exists m : \underline{x} = \underline{x}_k \oplus m \},$$

i. e., all the solutions which can be obtained by applying a move m^1 to \underline{x}_k , and we choose the next solution \underline{x}_{k+1} in N_k . This iterative process generates a sequence of improving solutions hopefully converging to

¹ The notation $x \oplus m$ stands for “apply move m to solution x ”.

Solution is improved iteratively by exploring other near solutions

an optimal solution. When x_{k+1} is chosen as the best solution in N_k the algorithm is known as the *steepest descent method*, because at each iteration it takes the most promising direction, and it stops at the first local minimum.

5.1 TABU SEARCH

Tabu Search (**TS**) is a meta-heuristic based on an iterative local search algorithm [Hertz et al., 1995, Glover and Laguna, 1998] which prevents stopping in local minima. At each iteration, when we choose the next solution, we add the inverse move to a Tabu List (**TL**). When exploring the neighbourhood we ignore all the moves in **TL**, i. e., the Tabu Moves (**TMS**), and then we choose the best neighbour, even if it is not improving the current solution. **TL** has fixed size and contains only recent moves.

Tabu Search allows to escape from local minima

TMS forbid to go back during local search: whenever we reach a local minimum, we continue and start climbing its walls. In the next iterations we can not fall again in the minimum, since moves that lead back to the local minimum are forbidden. In few iterations we will be – hopefully – outside the local minimum.

Aspiration criterion

The idea of **TMS** is to forbid cycles of solutions in **TS**, i. e., to forbid visiting already known solutions. However sometimes a **TM** can forbid to visit an unknown solution. If such solution is better than any one we found so far we ignore the **TM** and we insert it in the neighbourhood.

Intensification

With intensification we focus **TS** in a small region of solution space. For instance if we know the approximate region of a local minimum, we can intensify **TS** in order to obtain the actual minimum. To start intensification we can either reduce the **TL** size, so that less nearby solutions are forbidden, or we can use a custom objective function that increases penalty on solutions far from the current one for few iterations.

Diversification

Diversification is the dual of intensification: it drifts **TS** to a completely different region of solution space. To perform diversification we can either start over from a random initial solution, or we can use a custom objective function that increases penalty on solutions near the current one for few iterations.

Strategic oscillation

The solution space is explored moving from a feasible solution to another. It might happen that to reach a minimum we have to circumnavigate an infeasible region with a long sequence of feasible moves. By temporarily relaxing some constraints we can instead cross the infeasible region and reach the minimum in less iterations.

*Exploration crosses
infeasible regions*

TS has been successfully applied to a large range of discrete optimization problems [Glover and Laguna, 1998]. Ahuja et al. [2002] provides a survey of techniques useful in local search problems with very large neighbours.

5.2 TABU SEARCH FOR THE RESIDENTIAL ENERGY LOAD MANAGEMENT PROBLEM

In this section we describe in detail the algorithms we developed for the **TS** meta-heuristic.

We implemented a **TS** algorithm that supports different types of moves. At each iteration we generate all the possible moves $m \in M$ of the specified type and apply them the current solution, obtaining the neighbourhood. Then we rank all neighbours with a given objective function and we take the best one for the next iteration.

Different types of moves explore different types of neighbourhoods:

SHIFT AND SWAP MOVES A new solution is obtained from the current one by changing the starting time slots of one or more activities;

BATTERY MOVES A new solution is obtained from the current one by changing the usage of batteries, i. e., when and how much they are charged and discharged;

REDUCED MILP MOVES A new solution is obtained from the current one by solving a reduced **MILP**;

LARGE MOVE A new solution is obtained from the current one by rescheduling a large part of the activities using **GRASP**.

In order to obtain the best from each move type, we implemented a procedure to dynamically change the current move type. Some move types also support the exploration of the infeasible region. The heuristic can also detect when the solution have not improved in the latest iterations, and can decide to stop the exploration before reaching the maximal iterations limit. Instead of ending the exploration it is also able to perform diversification.

5.2.1 Shift move

An activity's starting time is shifted to another slot

A shift move is the simplest possible move: we shift an activity² (h, a) from a starting time slot t_{old} to another time slot t_{new} . In this iteration we first populate a list of *interesting* activities, i. e., activities that influence the maximal peak, then we shift each of these activities to all time slots within its execution window, generating a set of neighbour moves $N = \{(h, a, t_{\text{old}}, t_{\text{new}})\}$. We then select the move that generates the best neighbour. The size of the neighbour is $O(\text{hat})$.

5.2.2 Swap move

Neighbourhood's size is large for swap moves

A swap move resembles a sequence of two shift moves: we shift two activities $(h_1, a_1), (h_2, a_2)$ from starting time slots $t_{1,\text{old}}, t_{2,\text{old}}$ to $t_{1,\text{new}}, t_{2,\text{new}}$. h_1, h_2 could be the same, and so a_1, a_2 (though at least one between house and appliance must differ). The main difference from a sequence of two shift moves is that $t_{1,\text{new}} = t_{2,\text{old}}$ and $t_{1,\text{old}} = t_{2,\text{new}}$. Another difference is that even if the intermediate solution is infeasible the move is feasible as long as the final solution is so.

The size of the neighbourhood in this case is much larger than the one for shift moves: $O(h^2a^2)$ compared to $O(\text{hat}) \simeq O(ha)$: for a 200 houses and 11 appliance instance we have $4840000 \gg 2200$. Since exhaustive exploration of the neighbourhood is just out of reach, we sample the neighbourhood. We select a random activity (h_1, a_1) among the ones influencing the maximal peaks, starting at $t_{1,\text{old}}$. Then we select all activities (h_2, a_2) that can start at $t_{1,\text{old}}$ and if (h_1, a_1) can start at $t_{2,\text{old}}$ as well, then we swap them. If the result is feasible with respect to the local maximal peak then the iteration is concluded, as a stop-on-first-improvement policy is used (a minimal number of generated neighbours can be specified, though), otherwise we try another random pair of activities.

5.2.3 Battery move

Looking for a suitable pair of time slots, a first one with low peak where to charge the battery, and a second one where to discharge it to decrease peak

This move focuses only on batteries. We first find a time slot t_d in which a battery can be discharged, then we find a preceding time slot t_c in which the battery can be charged.

For each time slot t_d corresponding to one of the highest peaks, we loop over the battery-equipped houses. For each house we try anticipate buying the amount of energy ℓ that it buys in that time slot, somewhere before the current time slot, storing it in batteries, and releasing it in t_d . In t_d we must have the following:

1. It is possible to discharge at least ℓ energy. Otherwise we continue with the next house in the list.

² An activity is a pair of house, appliance: $(h, a) \in H \times A$.

2. The new battery flow, i. e., the old one plus ℓ , is above θ_b^{\min} . Otherwise we continue with the next house in the list.


To find a time slot t_c in which the battery is charged, we check the time slots from $t_d - 1$ to 1 verifying if:

1. There is enough battery storage available. If not, we continue with the next house in the list. We can not charge the battery in earlier time slots, since we have to store the energy in the battery until t_d , but in the current time slot the battery has not enough capacity.
2. In the current slot it is possible to charge the battery. Batteries can either idle, charge or discharge in a given time slot, if the latter is true then we can not charge it. However, an earlier time slot might still be suitable, so we continue with the previous time slot.
3. It is possible to buy at least ℓ energy. Otherwise an earlier time slot might still be suitable, so we continue with the previous time slot.
4. The new battery flow, i. e., the old one plus ℓ , is above τ_b^{\min} . Otherwise an earlier time slot might still be suitable, so we continue with the previous time slot.

If all tests succeeded then it is possible to buy ℓ energy at time slot t_c , storing it in batteries until t_d and then release it to the house. We produce a new solution member of the neighbourhood by applying these changes. After we tried all the combinations of house, t_c and t_d , we choose the best solution in the neighbourhood as next solution.

Algorithms [5.1 on the next page](#) and [5.2 on page 51](#) show the procedure in detail. *energy-profile** is the best neighbour's profile and it is overwritten in the bottommost loop; we do not show other variables containing solution's state for the best neighbour (e. g. t_d , t_c), but we store them alongside the best profile and we update the solution with them. Note that in the actual algorithm we store the best *incomplete* energy profile described in [Section A.1.1 on page 96](#) (the only difference is that in [line 18](#) the profile is not copied completely).

Example

We now show a simplified example of a battery move in [Figure 5.1](#). At first we identify the time slot with highest peak: time slot $t_d = 8$. In t_d there is a house with a battery that is buying the amount of energy indicated with . We can instead buy such energy in an earlier time slot t_c , storing it in the battery, and discharge the battery in t_d . The only feasible time slots for charging the battery are t_c^1 and t_c^2 (note that they are visited backwards). We generate the two demand

Algorithm 5.1 Battery move, outer loops

```

BATTERY-MOVE( $\underline{x}_0$ )
1   $energy-profile^* \leftarrow \infty$ 
2   $p_0 \leftarrow \text{ENERGY-PROFILE}(\underline{x}_0)$ 
3  for  $t_d \in \text{HIGHEST-PEAKS}$ 
4      do  $\triangleright$  Compute houses which buy much energy in  $t_d$ 
5           $interesting-houses \leftarrow \{(h, \ell) \dots\}$ 

6          for  $(h, \ell) \in interesting-houses$ 
7              do if  $v_{h,b,t_d}^c > 0$ 
8                  then  $\triangleright$  House  $h$  is charging, can not discharge
9                      Skip house  $h$ 

10                  $\triangleright$  Compute how much energy is possible to discharge in  $t_d$ 
11                  $\ell_d \leftarrow \min \left\{ \ell, \theta_b^{\max} - v_{h,b,t_d}^d, \gamma_b^{\max} - pB_{h,b,t_d-1}, \pi_{h,t}^{in} \right\}$ 

12                  $\triangleright$  Can discharge enough energy?
13                 if  $\ell_d < \ell$ 
14                     then Skip time slot  $t_d$ 
15                 if  $v_{h,b,t_d}^d + \ell_d < \theta_b^{\min}$ 
16                     then Skip time slot  $t_d$ 

17                 BATTERY-MOVE-INNER-LOOPS( $\underline{x}_0, t_d, h, \ell, energy-profile^*$ )

18  $\triangleright$  Update solution with new profile and battery information
19  $\underline{x} \leftarrow \text{UPDATE}(p_0, energy-profile^*, \dots)$ 

20 return  $\underline{x}$ 

```

Algorithm 5.2 Battery move, inner loops

BATTERY-MOVE-INNER-LOOPS($p_0, t_d, h, \ell, \text{energy-profile}^*$)

- 1 ▷ Loop backward and try charging in all previous slots
- 2 **for** $t_c \leftarrow t_d - 1$ **downto** 0
- 3 **do** ▷ Is there enough storage?
- 4 **if** $pB_{h,b,t_c} + \frac{1}{4} \ell_d > \gamma_b^{\max}$
- 5 **then** Skip time slot t_d ▷ discharge slot, not t_c

- 6 ▷ Is battery already discharging?
- 7 **if** $v_{h,b,t_c}^d > 0$
- 8 **then** Skip time slot t_c

- 9 ▷ Compute how much energy is possible to charge in t_c
- 10 $\text{from-ideal} \leftarrow \max \{0, \text{difference}_{t_c}\}$
- 11 $\ell_c \leftarrow \min \{ \gamma_b^{\max} - pB_{h,b,t_c}, \tau_b^{\max} - v_{h,b,t_c}^c, \text{from-ideal}, \pi_{h,t}^{\text{in}} - y_{h,t_c} \}$

- 12 ▷ Can charge enough energy?
- 13 **if** $v_{h,b,t_c}^c + \ell_c < \tau_b^{\min}$
- 14 **then** Skip time slot t_c
- 15 **if** $\ell_c < \ell$
- 16 **then** Skip time slot t_c

- 17 ▷ We can charge ℓ in t_c and discharge it in t_d
- 18 $p \leftarrow \text{COPY}(p_0)$
- 19 ADD-LOAD(p, ℓ, t_c)
- 20 REMOVE-LOAD(p, ℓ, t_d)
- 21 **if** $p < \text{energy-profile}^*$
- 22 **then** $\text{energy-profile}^* \leftarrow p$
- 23 ▷ Save also best h, ℓ, t_d and t_c
- 24 **return**

curves for the two cases when we buy energy in t_c^1 or t_c^2 , and we pick the best one, according to the given objective function (both solutions have the same maximal peak, but for instance charging in t_c^2 reduces also the maximal difference).

5.2.4 MILP move

MILP move is very different from the moves described so far. The idea is to solve a smaller MILP formulation obtained from the current solution by fixing the value of some variables.

Some variables are fixed and the reduced problem is solve with time limit

We fix the values of all the variables $x_{h,a,t}$ corresponding to activities that do not influence the maximal peaks; if the set of free variables is too big we fix other activities in order of increasing flexibility. We solve the reduced MILP problem and generate the next solution.

We do not explicitly generate a neighbourhood for this move, we let the MILP solver return the best neighbour it can find within the time limit.

5.2.5 MILP-batteries move

MILP-batteries move is also based on a reduced MILP problem. We fix all activities in their current starting time slot, and we leave batteries flows variables free. The MILP solver finds an improving batteries scheduling within a time limit.

Reduced problem's only free variables are battery ones

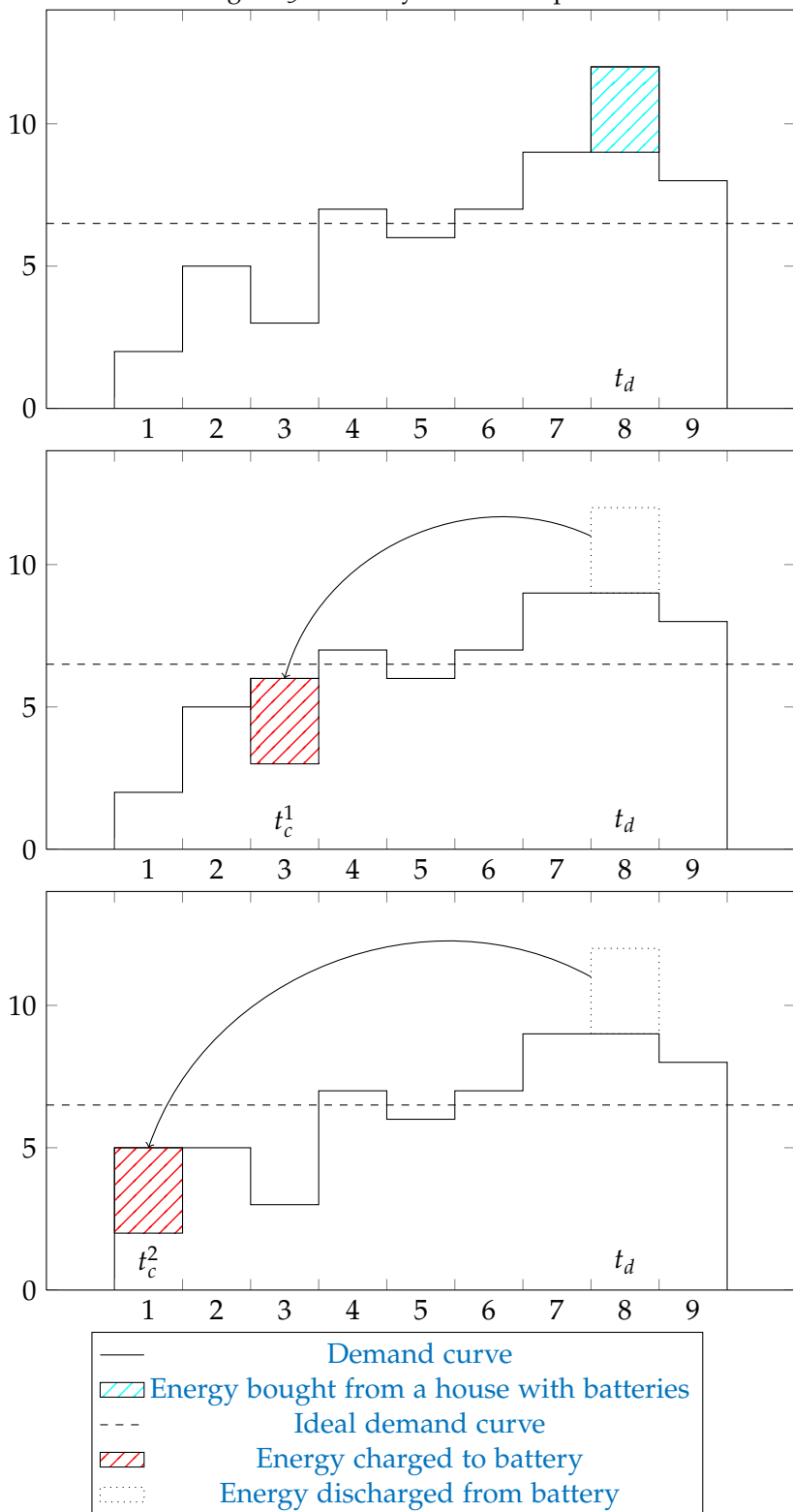
Note that this move is not supposed to be performed more than once in a row. If the MILP solver did not find the optimal solution for the reduced problem it would make no sense to repeat the move: indeed, it can not improve more. If it did not find the optimal solution within the time limit, it would likely not find it if run with the same parameters and inputs. This remark does not apply to the previous MILP move, since at each iteration the maximal peaks change and thus different activities get fixed.

Note that the only decisional variables left free are $v_{h,b,t}^c$, $v_{h,b,t}^d$, $\omega_{h,b,t}^c$ and $\omega_{h,b,t}^d$. Most of the integer variables become constants, so the problem is much closer to a Linear Programming (LP) problem and thus much easier and faster to solve.

5.2.6 MILP-zeroes-fixing move

This is another move that is based on a reduced MILP problem. The idea is similar to the one described in Section 4.4 on page 43, but instead of solving the PLR we compute the $x_{h,a,t}$ variables from the current solution's starting time slots. We fix the value of a random subset of these variables to 0 (not the one corresponding to the actual starting time slot), and we solve the reduced MILP with a time limit.

Figure 5.1: Battery move example



Current active move	Next active move
Swap	Shift / Battery / MILP-batteries
Battery	Shift
MILP-batteries	Shift
MILP-zeroes-fixing	Shift

Table 5.1: Move switch in case of empty neighbourhood

5.2.7 Large move

Solution is partially destroyed and rebuilt from scratch

This move is a *diversification move*, i. e., its purpose is not to improve the current solution, but to drift the exploration to a different region of the solution space. We remove a randomly chosen subset of activities from the solution and we reschedule them using GRASP. By changing the size of this subset it is possible to control how much the next solution should have in common with the previous one.

We generate only a single solution, which will be almost surely worse than the latest ones, but hopefully it will belong to a region of the solution space with a better local optimum.

5.2.8 Mixed move

Mixed move combines different types of move

We can use the moves described so far directly with TS. The shift move is actually quite effective in finding a good solution for instances without batteries, and the other moves manage to sensibly improve a solution. However they give the best results when they are combined and the heuristic can choose the move depending on the run-time state.

We defined mixed move as an abstraction, it wraps other moves and controls which one is active, i. e., used at current TS iteration. At the end of iteration it is possible to change the active move.

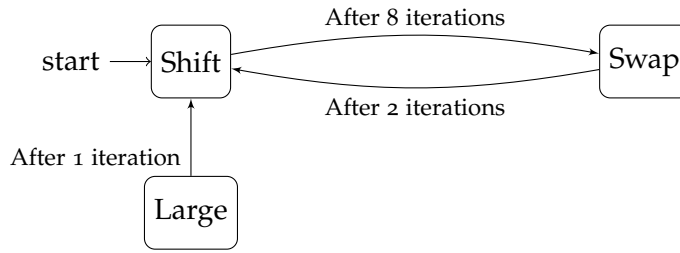
Sometimes in an iteration we can not find any solutions in the current neighbourhood, not even a worse one. For instance in a swap move is hard to find a suitable pair of activities, and the neighbourhood is only randomly sampled and not exhaustively explored; in battery move it may also happen that all useful time slots are filled. When the active move produces an empty neighbourhood, we switch to a different type of move, as we show in Table 5.1.

Tabu Search control

Active move type changes at runtime

After we complete an iteration we may replace the current active move with another one. The policy that controls this behaviour is slightly different whether batteries are available or not. In case of no batteries the shift move is sometimes interrupted to try a couple swap

Figure 5.2: Mixed move flow chart without batteries



moves (Figure 5.2). The idea is that higher order moves – moves that correspond to more than one shift move, as swap moves – might open paths closed to shift moves.

When batteries are available the aim changes: we want to use batteries to smooth the demand curve, so battery moves are performed as long as possible. In a perfect world we would not even need a single shift move, we would just charge batteries when the demand curve is above ideal, and discharge when it is below. Of course this world is imperfect, batteries have limited capacity and other constraints, besides, not all houses have batteries. Sooner or later it would be impossible to perform further battery moves. When an empty neighbourhood is found a short burst of shift and swap moves is performed, hoping that they will be able to make room for other battery moves (Figure 5.3).

When using **MILP**-batteries moves, we do not perform them more than once in a row, so after a single iteration the shift move becomes the active move (Figure 5.4).

When using **MILP**-zeroes-fixing moves, we perform them only once at the beginning of the execution. Although the solution does not improve in a significant way when **MILP**-zeroes-fixing moves are stacked, the solution benefits of some refinement done with simple mixed move (Figure 5.5 on the next page).

In all cases we perform a large move for a single iteration, and then the shift move becomes the active move. Large moves almost destroy the current solution with just a single iteration, two iterations in a row might destroy it completely. Large move can only be promoted to active move when a diversification is requested by the **TS** algorithm (e. g. when we could not improve the solution for a while).

5.2.9 Tabu Moves

When exploring the neighbourhood using the shift moves (Section 5.2.1) or the **MILP** moves (Section 5.2.4) we enabled the use of a Tabu List (**TL**). When we choose a shift move $(h, a, t_{\text{old}}, t_{\text{new}})$, we insert its inverse Tabu Move (**TM**) (h, a, t_{old}) into the **TL**, i. e., we forbid moves that shift activity (h, a) back to starting time slot t_{old} .

Figure 5.3: Mixed move flow chart with Battery moves

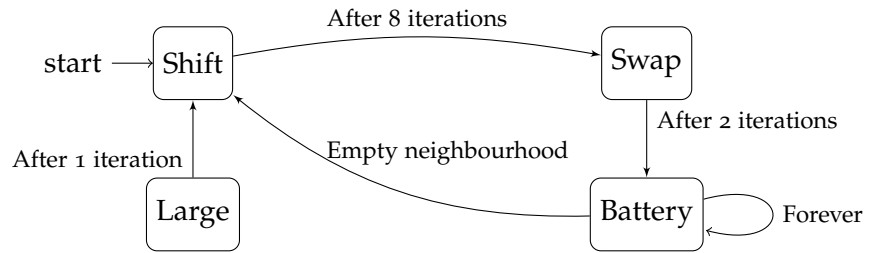


Figure 5.4: Mixed move flow chart with **MILP**-batteries moves

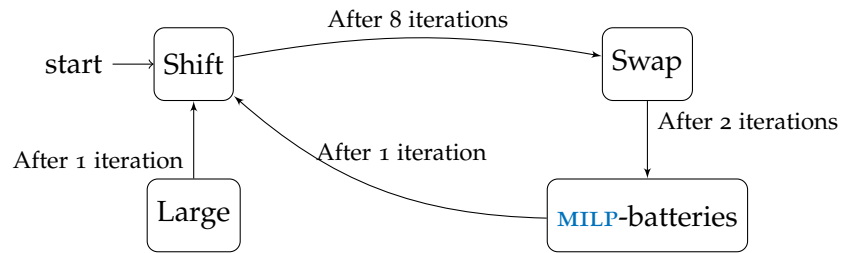
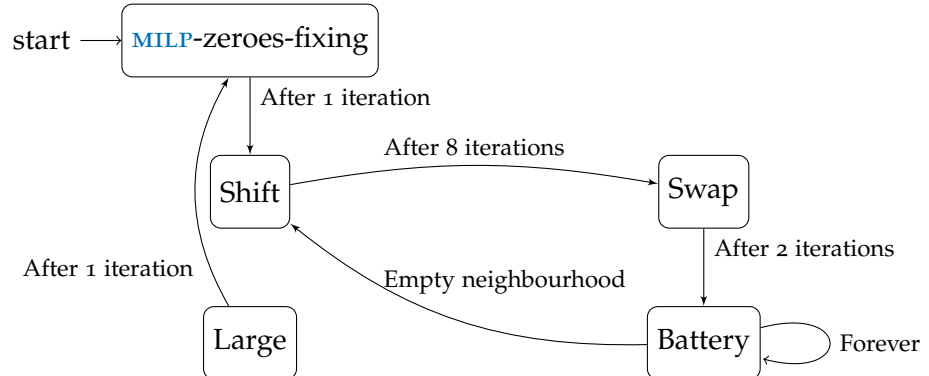


Figure 5.5: Mixed move flow chart with **MILP**-zeroes-fixing moves






When in the **MILP** move we choose a solution, we compute a sequence of shift moves from the initial solution to the new solution. We then add the inverse of each of these shift moves to the **TL**.

We implemented **TMS** only for these types of moves, since they are the only which are executed in long bursts. Battery moves, **MILP**-zeroes-fixing moves... are usually the active move for only few iterations, the corresponding **TMS** would get outdated very soon.

5.2.10 *Exceeding maximal local peak*

For each house the Constraint (3.4) imposes a maximal local peak, i. e., the retailer sets an upper bound to the energy available in one time slot. Until now, during local search all solutions were required to be feasible at each iteration, i. e., they had to satisfy all constraints. However it might be possible that by taking more than one infeasible steps the final solution is feasible and improved.

Strict feasibility might prevent reaching desirable solutions

Let's consider the following simplified instance: the day is divided in 2 time slots and there are 3 activities, red , blue  and green . Figures 5.6a and 5.6b show respectively the initial solution and the optimal solution (the maximal local peak is shown as ---). Red activity must start at the first time slot, so from the initial solution there are only two shift moves: move green activity to first time slot, shown in Figure 5.6d, and move blue activity to second time slot, shown in Figure 5.6c.

The two possible moves from initial solution \underline{x}_0 are $(G, 1)$ and $(B, 2)$, but the solutions

$$\begin{aligned} \underline{x}_1 &= \underline{x}_0 \oplus (G, 1) \\ \underline{x}_2 &= \underline{x}_0 \oplus (B, 2) \end{aligned}$$

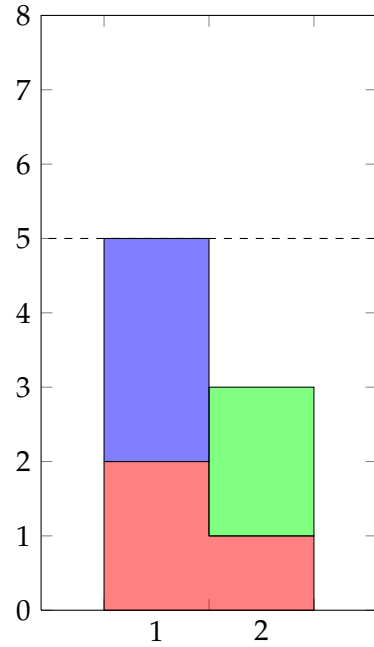
are both infeasible, while the optimal solution

$$\begin{aligned} \underline{x}^* &= \underline{x}_0 \oplus (G, 1) \oplus (B, 2) \\ &= \underline{x}_0 \oplus (B, 2) \oplus (G, 1) \end{aligned}$$

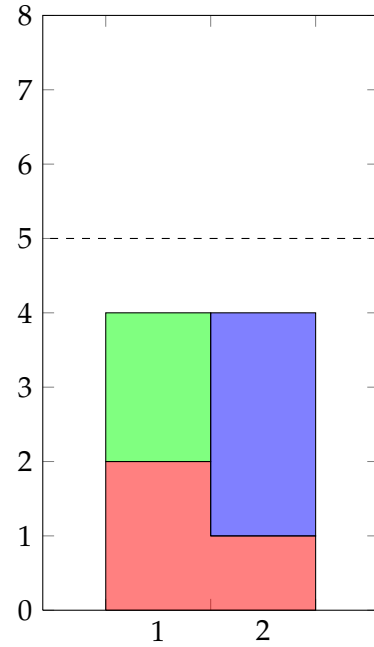
can only be reached by applying the two moves in sequence.

We can imagine the optimal solution being beyond an infeasible barrier of height 2: we can not go beyond the barrier travelling only one move a time, therefore the optimal solution can never be reached. A swap move handles this particular example, even if it means increasing the neighbourhood size from $O(\text{hat})$ to $O(h^2 a^2)$, however there might occur other cases where the barrier is of height 3 or even higher. We could consider chains of moves instead of single moves, but then the neighbourhood size would explode to $O(h^k a^k t^k)$ for chains of length k .

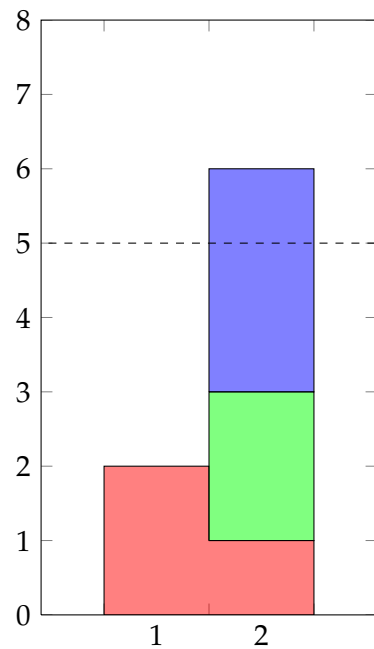
Figure 5.6: Swapping two activities



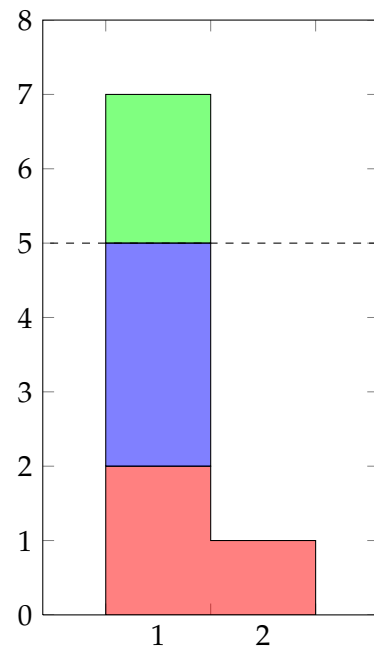
(a) Initial solution



(b) Optimal solution



(c) Shift blue activity to time slot 2



(d) Shift green activity to time slot 1

Infeasible exploration

A better approach is to relax the maximal local peak constraint for few iterations. This way there are no barriers at all and we can freely explore the solution space one move at time. To recover feasibility we proceed as follows:

Let's call π_t^{in} the maximal local peak, and $\max_t y_{h,t}$ the current local peak. Then we have for each house a *slack*, i. e., the distance from the feasibility,

$$s_h \triangleq \max \left(0, \max_t y_{h,t} - \pi_t^{\text{in}} \right) \quad \forall h \in H$$

$$s \triangleq \sum_{h \in H} s_h.$$

Then we define three phases in which the **TS** can operate:

STRICT FEASIBILITY PHASE The current solution is feasible and we discard all moves that generate infeasible solutions. The interesting activities³ are all activities that influence the maximal peaks. All slacks are zero.

INFEASIBILITY PHASE If a solution is infeasible it has strictly positive slacks for the maximal local peak constraint (i. e. $\exists (h, t) : \pi_t^{\text{in}} - y_{h,t} = s_{h,t} \gtrsim 0$). The sum of those slacks gives a measure of how much infeasible it is

$$\sum_{h \in H} s_h = \text{Solution's infeasibility.}$$

We do not discard any solution, hence the next solution could be infeasible. At the end of iteration we update the aggregate values of slacks for each house s_h . The interesting activities are again all activities that influence the maximal peaks.

LOOSE FEASIBILITY PHASE When we are in this phase it means that we have been exploring the infeasible region, but now we are trying to recover feasibility. We discard all moves that generates solutions not less infeasible than the current one, i. e. with

$$\sum_{h \in H} s_h \geq \sum_{h \in H} s_h^{\text{Current solution}},$$

since they would not help in reducing the infeasibility. If the next solution has all zero slacks then we recovered feasibility, and we switch to strict feasibility phase. We ignore **TMS** altogether in this phase (they might prevent to recover feasibility). In this phase the interesting activities are all and only activities

In infeasibility phase maximal local peak is ignored

In loose feasibility phase only solutions with lesser slacks are considered

³ *Interesting activities* are the activities considered in shift moves. Activities that are not interesting are ignored, so their starting time slot can not change.

of infeasible houses (i. e., houses with positive slacks). Moves involving already feasible houses would not reduce infeasibility (slacks can not go below zero), so there is no use in considering other than infeasible houses. Secondly, there is a chance that no activity influencing the maximal peaks influences also an infeasible time slot, hence by taking only the latter we might end up with an empty neighbourhood.

Not all kind of moves support infeasible exploration. **MILP** moves and battery **MILP** moves support only strictly feasible exploration, while battery moves only works in strictly feasible and infeasible phases. Shift and swap moves supports all three phases. Large move works also in all three phases.

5.2.11 *Early stopping and diversification*

Usually **TS** stops either after a time limit or a maximal number of iterations, but we developed an auxiliary algorithm that can detect whether it might be a good choice to stop before that.

We keep a queue of last $k \cdot t$ solutions, where t is the current **TL** length and k is a parameter. If the best solution found so far did not improve during the last $k \cdot t$ iterations⁴, we assume that the current region of the solution space was sufficiently explored.

We can then stop and return the best solution found so far, or we can try the diversification move defined in Section 5.2.7, which randomly reschedules half of activities using **GRASP**, clearing the last solutions queue. In the following iterations we will explore a different region in the solution space, and hopefully we will find a new local minimum.

5.3 LOCAL BRANCHING

Fischetti and Lodi [2003] proposed a local branching technique to solve large **MILP** problems, which tries to bring together the advantages of both **MILP** solvers and local search heuristics. Starting from an initial guess its neighbourhood is explored and the result is used as new guess for the next iteration. However the neighbourhoods are defined using **MILP** constraints and are explored with the **MILP** solver.

At iteration k , the solution space is divided in two subsets: solutions closer than m_k moves from the current solution x_k , and solutions further away than m_k moves. The first subset is explored with **MILP** solver with a time limit, and the best solution found is used as the next solution x_{k+1} . In the following iterations a new constraint is imposed to exclude already explored regions of solution space.

⁴ When using **TLs** we may hit the bottom of a local optimum, climbing up, and then improving again. By using the best solution *found so far* this does not count as improvement.

5.3.1 Local branching for the Residential Energy Load Management Problem

We implemented the algorithm proposed by Fischetti and Lodi [2003] without any relevant modification.

At first we need to define a function of distance between two solutions. We say that two solutions \underline{x}_j and \underline{x}_k are m steps distant if there are m activities that start at different time slots in the two solutions, i. e., their distance is⁵

$$\Delta(\underline{x}_j, \underline{x}_k) = \sum_{h \in H} \sum_{a \in A} \left(1 - x_{h,a, \text{Starting slot of } (h,a) \text{ in } \underline{x}_k}^j\right),$$

where “Starting slot of (h, a) in \underline{x}_k ” is the starting time slot of activity (h, a) in the solution \underline{x}_k . Batteries are left free.

Given a solution \underline{x}_k , we divide the solution space in the two regions: one closer to \underline{x}_k than m_k steps and one farther. To instruct the MILP solver to restrict on one of these two regions we add one of the following constraints:

$$\begin{aligned} \Delta(\underline{x}, \underline{x}_k) &\leq m_k \\ \Delta(\underline{x}, \underline{x}_k) &\geq m_k + 1. \end{aligned}$$

We solve the problem with these additional constraints within a time limit, and we take different actions depending on the result:

AN OPTIMAL SOLUTION WAS FOUND We optimally solved this region of solution space. We register the best upper bound and we exclude this region from following exploration by permanently adding the constraint

$$\Delta(\underline{x}, \underline{x}_k) \geq m_k + 1.$$

THE PROBLEM IS INFEASIBLE This region of solution space does not contain feasible solutions⁶. We increase m_k to explore a large region. If a diversification is possible we also reset the upper bound to ∞ . We exclude this region from following exploration by permanently adding the constraint

$$\Delta(\underline{x}, \underline{x}_k) \geq m_k + 1.$$

A FEASIBLE SOLUTION WAS FOUND We cannot exclude the current region from following exploration, since it could still contains

⁵ Here x stands for the starting time slot variables discussed in Chapter 3, while \underline{x} (note the underline) represents instead a whole solution. We do not use again the first x in the rest of local branching section.

⁶ This seldom means that there are no feasible solutions at all, but instead that those solutions are all worse than the current upper bound.

better solutions, so we limit ourselves to excluding the current solution by permanently adding the constraint

$$\Delta(\underline{x}, \underline{x}_k) \geq 1.$$

We then use the newly found solution \underline{x}_{k+1} for the next iteration, i. e., we divide the solution space in two regions, one close to \underline{x}_{k+1} and one far from \underline{x}_{k+1} .

NO SOLUTION WAS FOUND If a diversification is possible we replace the last *closeness* constraint with a trivial one to exclude the current solution; then we reset the upper bound to ∞ and we enlarge the current region by increasing m_k .

Otherwise we reduce the region's size, by decreasing m_k .

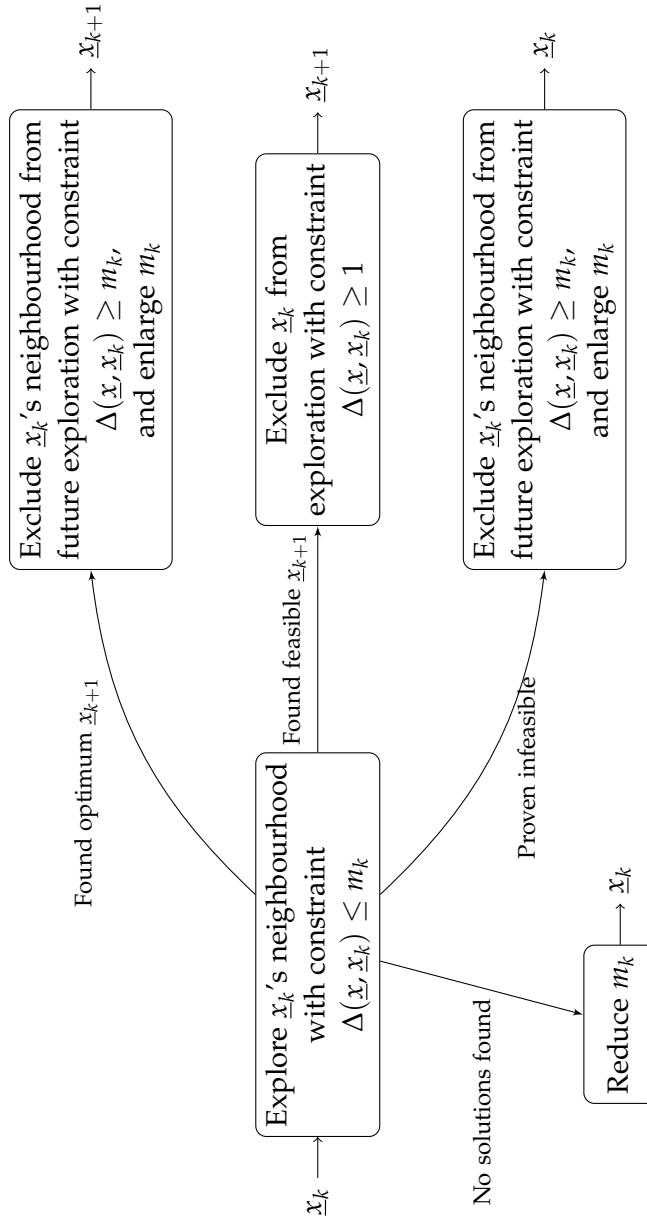
Finally, we solve a last **MILP** problem excluding all found solutions, all optimally solved regions and all proven infeasible regions. If we can optimally solve this last sub-problem, or if it can be proven infeasible, a global optimum was found. Otherwise we simply return the best solution we found.

We show a state chart of local branching iteration in Figure 5.7. The input is the current solution, and the output is the next solution.

Refining

When we find a feasible solution it may happen that a slightly better one was hidden by the additional constraints, but it is still feasible for the original problem. We perform a few iterations with **TS** using the simple shift moves described in Section 5.2.1 to avoid such risk.

Figure 5.7: Local branching iteration state chart



COMPUTATIONAL RESULTS

In this chapter we report and discuss the results obtained with the methods we developed. At first we describe the data set and show the reference results, then we report results for different kind of objective functions. We report results obtained with **GRASP** using different filtering criteria, enabling infeasible generation and enabling battery usage. We report results obtained with **TS**, for few variants and parameters: using shift moves, mixed moves, solving a reduced **MILP**, enabling diversification and infeasible exploration. We report results obtained with **PLR** with reduced **MILP** and local branching methods. In the end we compare the different methods and remark their advantages and disadvantages.

CONTENTS

6.1	Objective functions	68
6.1.1	p -norms	68
6.2	GRASP	72
6.2.1	Filtering criteria	72
6.2.2	Infeasible generation	74
6.2.3	Batteries usage in GRASP	75
6.3	Tabu Search	76
6.3.1	Tabu Search with shift moves	76
6.3.2	Tabu Search with MILP moves	78
6.3.3	Tabu Search with mixed moves	78
6.3.4	Tabu Search with MILP -zeroes-fixing move and mixed moves	79
6.3.5	Tabu Moves	80
6.3.6	Early stop and diversification	80
6.3.7	Infeasible exploration	82
6.4	Partial Linear Relaxation and reduced MILP	82
6.5	Local branching	84
6.6	Comparison	86
6.7	Multi-threading	89

Data set

For tests we used a fixed set of instances, shown in Table 6.1. There are three groups with 20, 200 and 400 houses. Each house has a fixed set of 11 appliances. In each group there are 10 different instances, where appliances load profiles and execution windows vary. For each instance there are variants with all combinations of 0 - 10% batteries and 0 - 10% - 20% **PV** panels. In total there are 60 different instances with 20 houses, 60 with 200 houses and 60 with 400 houses. Appliances load profiles and execution windows are realistic, but not real,

Houses	Batteries	PV panels	Instances
20	0, 2	0, 2, 4	1 to 10
200	0, 20	0, 20, 40	1 to 10
400	0, 40	0, 40, 80	1 to 10

Table 6.1: Data set

and were computed using information collected during a previous project [Carpentieri, 2012].

As we shall see performances over different kind of instances are heterogeneous, thus the aggregate over the whole data set is very imprecise. The two aspects that most influence this are batteries and number of houses.

Batteries make the problem much harder. For instance an heuristic might be able to solve a instance without batteries up to 2% gap from optimum, but it might be able to reach only 8% gap from optimum if we add even one single battery. This is also true when using the CPLEX solver, as batteries add more variables.

Instances with different number of houses are also difficult to compare. A single move actually affects a constant amount of energy over the whole energy profile, while the energy profile's total energy is of course directly tied with number of houses. It takes therefore more iterations to improve larger instances. On the other hand this means that steps are smaller in larger instances, so the solution is *finer grained*, and it is possible to reach better results in the long run by closely approaching the ideal solution.

For these reasons in the following we will make separate plots and tables for instances with and without batteries and for different houses number. PV panels presence does not seem to influence much the results, so we will aggregate the results over them.

Reference results

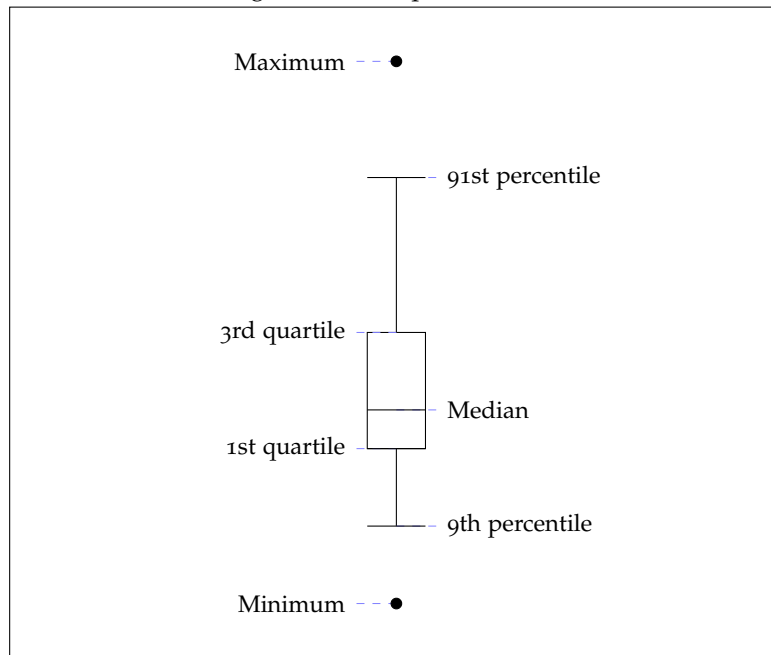
In Table 6.2 we show the results obtained with a MILP solver by minimizing the maximal peak of the demand curve. We used IBM ILOG CPLEX 12.4.0.0 and IBM ILOG AMPL 11.010 on a Intel(R) Xeon(R) 3.30GHz with 16GiB RAM. Note that CPLEX were also the solver we used to solve reduced MILP problems in our heuristics. Note that the computing times are always expressed in seconds, unless stated differently. These results were obtained by forcing CPLEX to use a single thread, so we did the same in all our tests, for both CPLEX and the heuristics, with the notable exception of the test in Section 6.7 on page 89.

CPLEX could not find the global optimum, but it produced a feasible solution and a lower bound. We computed the gap between that

Batteries	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
No	3.45%	3599.76	0.42%	5397.3	0.23%	10311.19
Yes	3.26%	3599.76	1.09%	5316.78	0.84%	10780.0

Table 6.2: Reference solutions

Figure 6.1: Box plot notation



solution and that lower bound. Except for small instances, the solution found was within 1% from the real optimum.

Box plot notation

We run many tests on the entire dataset, or a part of it, so the result is not a single value but actually a distribution of values. To compare distributions for different methods sometimes we used box plots, with the following notation:

The middle line is the median, the box itself is delimited by the first and third quartile, the two whiskers correspond the 9th and 91st percentiles, and the two dots correspond to the maximal and minimal sample in the collection. We show an example in Figure 6.1.

Gap from optimum

In order to compare the heuristic to the reference results we will show plots of the *relative gap from optimum* of a solution, which measures

the relative distance between a solution \underline{x} and the reference “optimal” one \underline{x}^* . It is defined as

$$\text{gap}_{\underline{x}} = \frac{\|\underline{x}\|_{\infty} - \|\underline{x}^*\|_{\infty}}{\|\underline{x}^*\|_{\infty}},$$

i. e., as the difference between the maximal peak of \underline{x} and the maximal peak of \underline{x}^* , divided by the maximal peak of \underline{x}^* . Note that \underline{x}^* is not the true optimum, but only the best solution found by the CPLEX solver after a long execution. It is therefore possible – although uncommon – to reach a negative gap.

6.1 OBJECTIVE FUNCTIONS

In this section we report results for different objective functions and different parameters.

We used different objective functions to generate 200 solutions with [GRASP](#) to investigate which one gives best results. In [Figure 6.2](#) we show the distribution of maximal aggregate peak only for 200 houses instances, however the results show the same qualitative behaviour for other subsets of data set, and for solutions obtained from [TS](#). The objective functions corresponding to the ids are listed in [Table 6.3](#).

The “maximal difference” objective function has extremely bad performance. Very little activities can start at the first time slots of the day, due to execution window constraints. Hence the aggregate peak during these time slots is always very small, i. e., very far from ideal, regardless of how close to the ideal the demand curve is in the following time slots (recall [Figure 4.2 on page 38](#)). The distributions are plotted a second time without this objective function, in order to better appreciate the differences among the other functions.

The “maximal peak” function gives also bad results compared to the others. The two functions that use both the maximal difference and the p -norm perform better, but the best results are obtained with the last function, that uses all three parameters: maximal peak, maximal difference and p -norm. It seems that, as discussed in [Section 4.3 on page 36](#), the best results are obtained when we could distinguish most among similar solutions.

All the following results in this chapter were obtained using the last objective function, except where stated differently.

6.1.1 p -norms

In [Figure 6.3](#) we show the results for different values of p when using the best objective function just shown in [TS](#) with shift moves for instances without batteries. The best value for p is 2, but the difference is rather small for other values.

Figure 6.2: GRASP with different objective functions, 200 houses

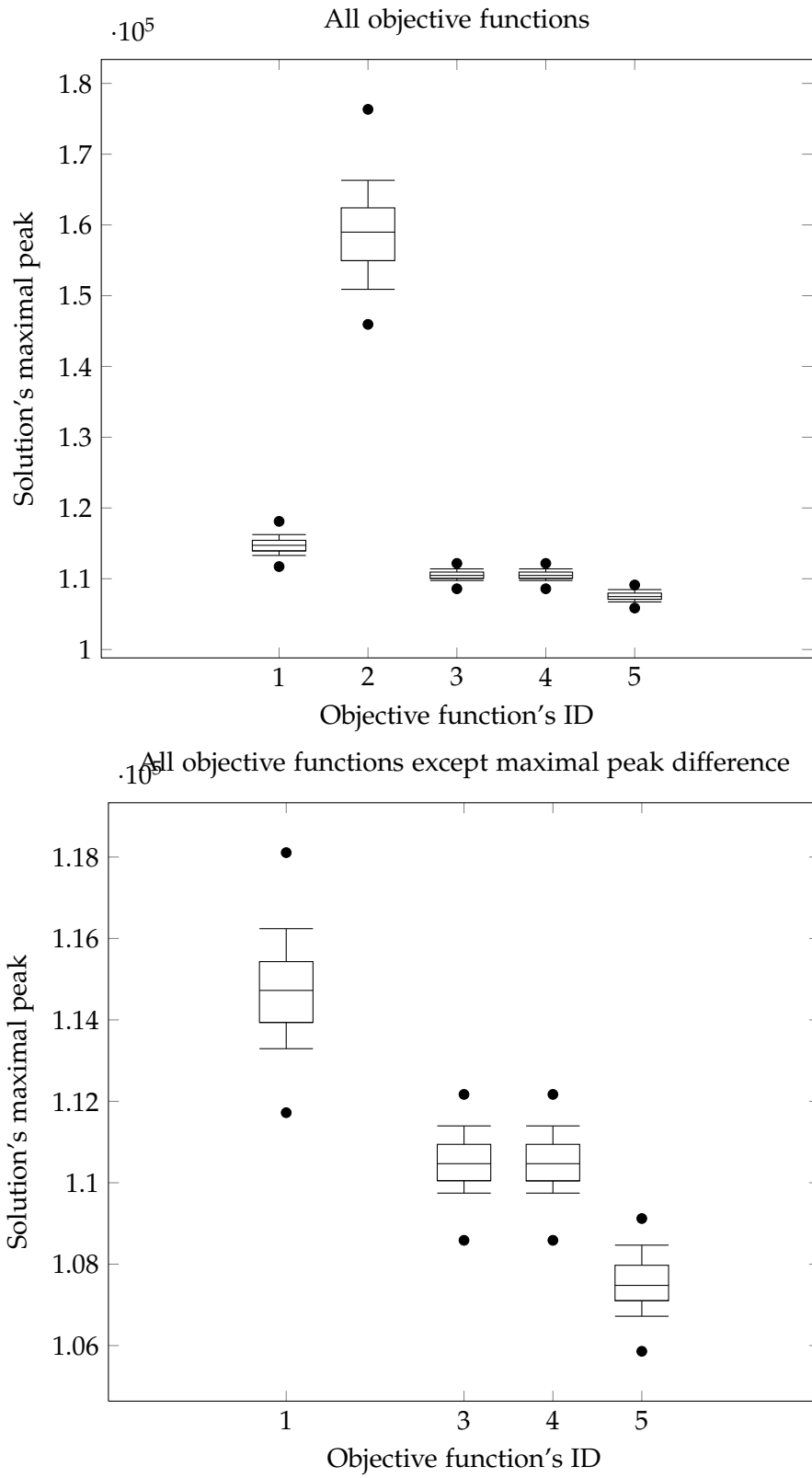
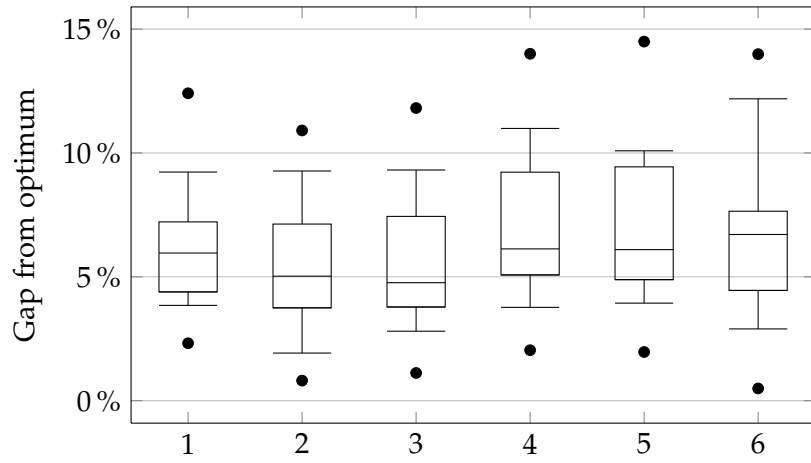
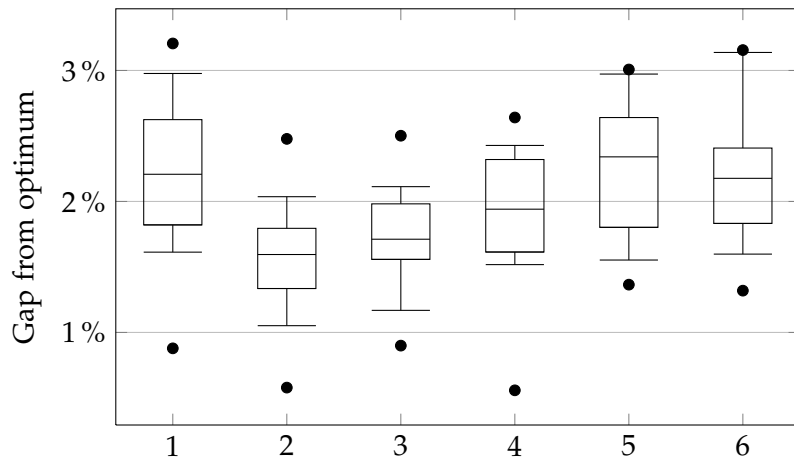
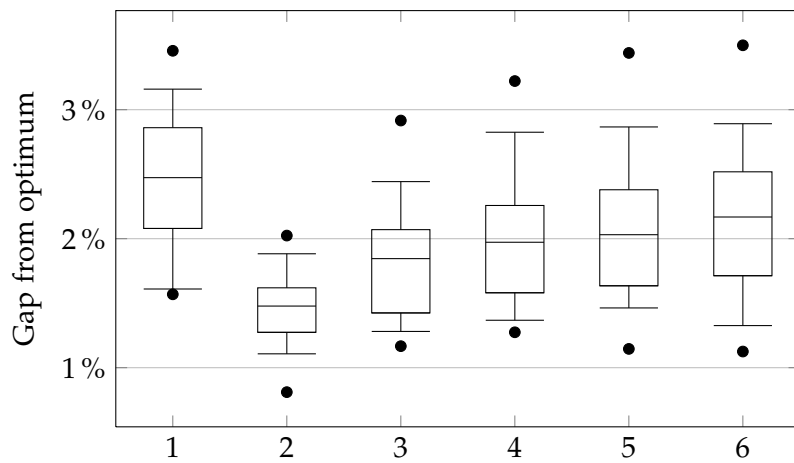


Figure 6.3: p -norms
20 houses

200 houses



400 houses



Id	Objective function
1	Maximal aggregate peak
2	Maximal difference from ideal
3	Maximal difference and p -norm
4	Maximal difference plus p -norm
5	Maximal aggregate peak plus maximal difference plus p -norm

Table 6.3: Objective functions ids

Criteria	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
Best 5	35.03%	3.4	21.60%	62.9	20.39%	208.83
Best 10	48.54%	3.5	33.65%	64.9	32.23%	204.93
Best 15	60.15%	3.7	45.15%	63.03	43.74%	206.07
Closest 0.8	28.15%	2.93	20.66%	59.33	20.85%	213.27
Closest 0.9	25.11%	2.93	17.46%	59.47	17.69%	202.67
Closest 0.95	23.40%	2.9	15.59%	60.2	15.83%	222.2
Closest 0.99	21.70%	2.83	13.89%	63.17	13.77%	209.3
Greedy	21.05%	2.7	13.63%	52.87	13.36%	194.0

Table 6.4: GRASP filtering criteria

6.2 GRASP

In this section we report the results for GRASP, for different filtering criteria and parameters.

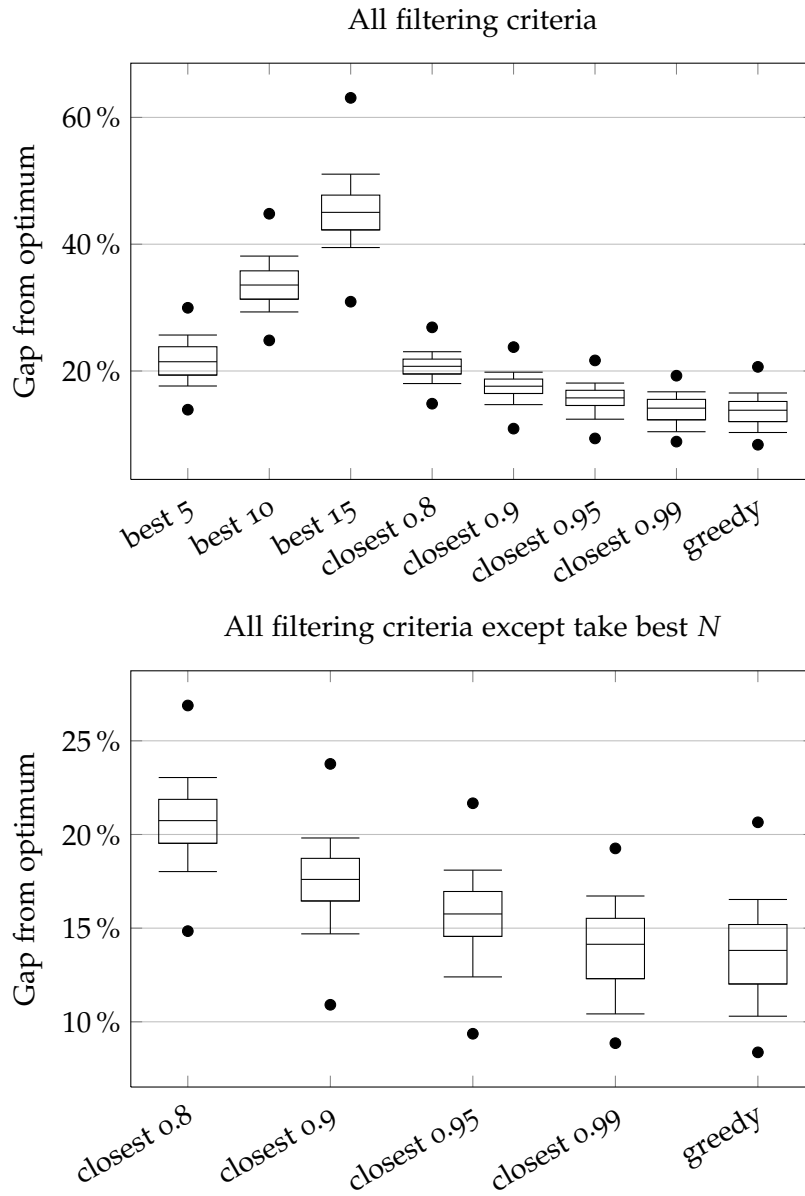
6.2.1 Filtering criteria

As explained in Section 4.2.1 on page 30, GRASP filters the best candidates at each iteration according to some criterion. In Table 6.4 we show the results when using different criteria and different parameters. For each case we generated 1000 solutions. We also show a box plot of the results for instances with 200 houses and no batteries in Figure 6.4, to better display the differences among different parameters. The distributions are plotted a second time without the “take best N ” criterion, in order to better appreciate the differences among the other criteria.

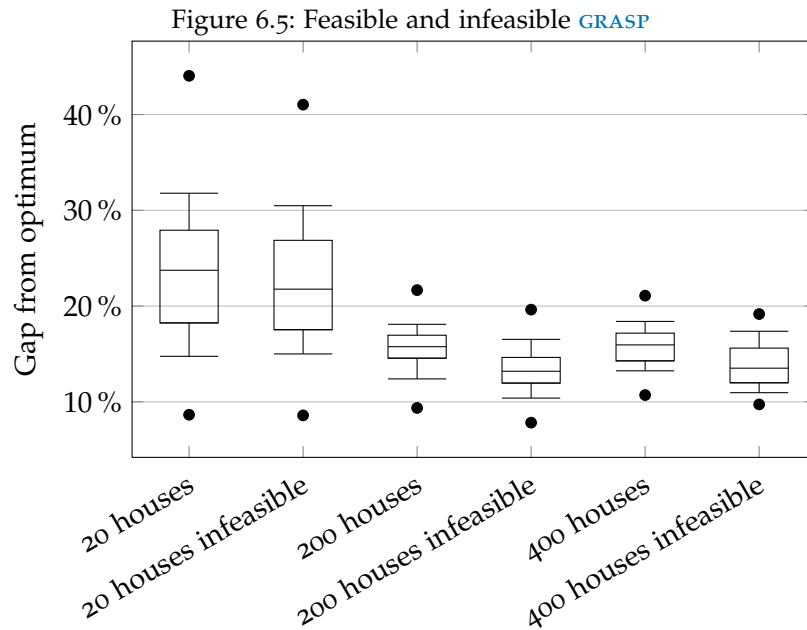
The “take best N ” criterion gives very bad results. The generated solution’s average gap from optimum appears to increase linearly with N , and even with low values of N it is much worse than other criteria. The “take closest α ” criterion gives much better results, and it also shows an increasing behaviour with decreasing the parameter α . This criterion with $\alpha = 1$ seems to be the most effective one.

Taking a fixed number of candidates has probably the worst effects in iterations where there is a small starting number of candidates, i. e., if there are 7 feasible starting time slot for an appliance then taking the best 5 may take low quality candidates. On the other hand, in iterations where there is a large number of candidates many of them might have similar values, so it is best to use a restrictive criterion to only take the best one, i. e., use a high value for α .

Figure 6.4: GRASP filtering criteria, 200 houses without batteries



	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
Feasible	23.30%	0.57	15.51%	7.8	15.69%	21.4
Infeasible	22.34%	0.57	13.28%	7.97	13.82%	21.0

Table 6.5: Feasible and infeasible **GRASP**

Note that batteries are ignored in **GRASP**, so the results in instances with batteries are much worse than in instances without batteries. In fact the generated solutions should be identical (up to randomly generated numbers, see Section A.3 on page 106), while the reference solution has a lower maximal peak.

Even with the most effective criteria the average gap is quite high, about 13% for large instances and up to 21% for small ones, hence the generated solutions need always be improved with **TS**.

6.2.2 Infeasible generation

In Table 6.5 and Figure 6.5 we compare the results obtained using **GRASP** when the local maximal peak constraint is or is not relaxed. Solutions are slightly better in the former case, the difference around 1%, and the computing time is roughly the same. Solutions are likely to be infeasible, but if we enable infeasible exploration in **TS**, it will start from a better solution. However, in the end it will have to recover feasibility. For each case we generated 100 solutions.

	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
Using batteries	39.74%	10.27	29.94%	127.1	29.33%	311.27
Ignoring batteries	34.70%	2.67	26.48%	45.4	26.80%	141.17

Table 6.6: Batteries usage in GRASP

6.2.3 Batteries usage in GRASP

It is possible to instruct GRASP to use batteries when constructing a solution. We compared the results for instances with batteries in Table 6.6. The results are slightly worse than if we ignore batteries, moreover the computing time is significantly longer. It is then a better choice to ignore batteries in GRASP altogether and let the local search phase make use of them.

6.3 TABU SEARCH

In this section we report the results for **TS**, for different kind of moves and parameters.

TS is an iterative algorithm, the solution's value changes at each iteration. It is then possible to plot the solution's value with respect to the iteration number (or the computing time) to represent how the algorithm is improving the solution. Aggregating these information from many executions is however not trivial, since each one would have a different trend. When representing multiple executions we plotted the average solution's value (or gap from optimum) from each execution, along with its standard deviation as error bars.

It might also happen that not all executions from the same batch last the same number of iterations. For instance an execution might reach its optimal value in less than the maximal iterations. Or it reaches earlier the maximal diversifications limit. For this reason the mean and standard deviation at higher iterations might be computed over a smaller set than the ones at lower iterations, and thus be less generalizable to the average case. This is often clear when the last point in progresses plots does not fit with the previous ones, or when it has a large standard deviation. When comparing final performances using box plots we use the value at the final iteration for every execution, so this is independent on how many iterations took to reach the optimal value.

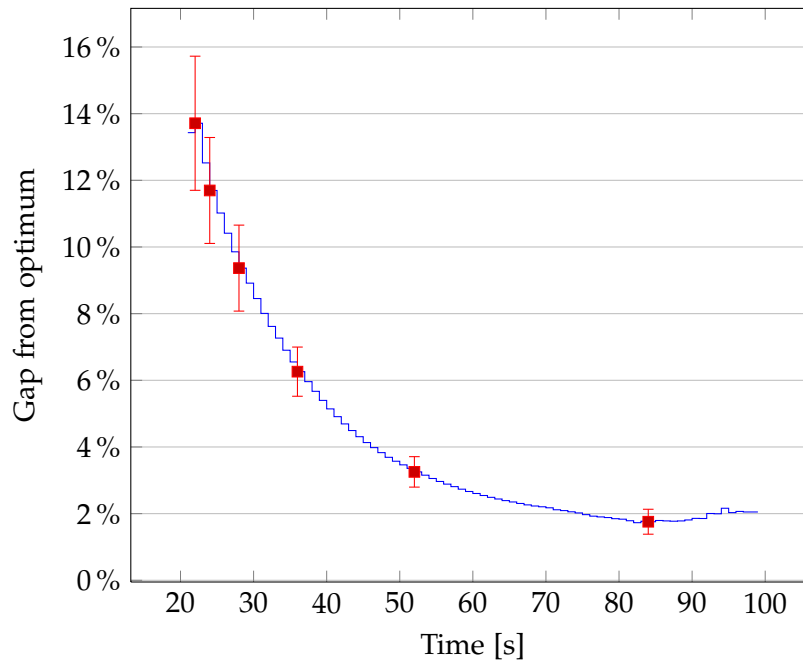
6.3.1 *Tabu Search with shift moves*

In Table 6.7 we show the results when using **TS** with shift moves. We generated 10 solutions with **GRASP** and we improved the best one with **TS**. **TS** stopped when it could not further improve the solution. The computing time includes the time to generate the initial solutions.

In Figure 6.6 we show the average trend for instances with 400 houses and no batteries. The qualitative behaviour is the same for other subsets of the data set. Early iterations result in large improvements; when instead the solution approaches the optimum the improvement slows down.

There is a significant difference for instances with a small number of houses and instances with a large number of houses. As we mentioned, in large instances there are many appliances to move, and their load profile is a tiny fraction of the total aggregate energy. It is therefore easier to fit them in time slots with a small (with respect to the total energy) residual capacity. For the same reason, computing time does not increase linearly with the number of houses, since for large instances more iterations are necessary to reach the optimal solution. Solutions for large instances are within 2% from reference solutions. Computing times is in average less than 80 seconds.

Batteries	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
No	5.33%	0.8	1.64%	24.3	1.54%	78.07
Yes	14.72%	0.87	11.33%	24.57	11.31%	79.03

Table 6.7: **TS** with shift movesFigure 6.6: **TS** with shift moves, 400 houses without batteries
Gap over elapsed time

Small instances are instead coarser grained. There are less appliances to move, and their load profile is much more significant with respect to the total energy. It is harder to fit them in time slots with a small (with respect to the total energy) residual capacity, since the total energy is much lower. Computing time is extremely short, about 1 second.

TS with shift moves ignores batteries. However we made an attempt to post-allocate them, i. e., after the **TS** stopped we charged all batteries in time slots where the demand curve was below the ideal curve and discharged them in the other time slots. This procedure is extremely fast and very simple, it was the first attempt we made to use batteries, but it did not succeed (solutions are farther than 10% from the reference ones), so we investigated the other mentioned algorithms (battery moves, **PLR**...).

Batteries	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
No	13.35%	15.03	13.70%	24.47	14.55%	43.83
Yes	26.11%	15.57	24.90%	48.63	25.68%	108.77

Table 6.8: **TS** with **MILP** moves

Batteries	Move	20 houses		200 houses		400 houses	
		Gap	Time	Gap	Time	Gap	Time
No		5.23%	0.87	1.58%	26.2	1.45%	89.73
Yes	Battery	13.93%	0.87	6.09%	34.93	5.76%	122.2
	MILP -batteries	12.57%	2.17	10.58%	43.3	12.70%	83.47

Table 6.9: **TS** with mixed moves

6.3.2 *Tabu Search with MILP moves*

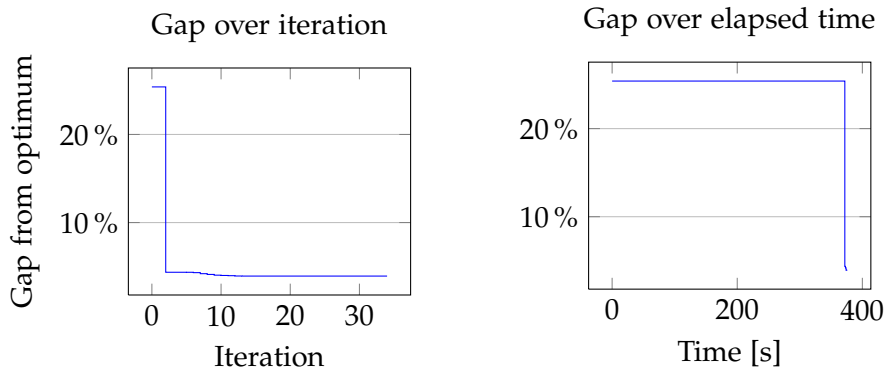
In Table 6.8 we show the results when using **TS** with **MILP** moves. This heuristic was our first attempt to improve a solution by solving a reduced **MILP**. In fact it produces solutions with the same quality as **GRASP**, over 13% from the reference, but in much longer time.

6.3.3 *Tabu Search with mixed moves*

In Table 6.9 we show the results when using **TS** with mixed moves. We run the tests with the same parameters of the **TS** with shift moves, but this time battery usage is embedded in the local search, as explained in Section 5.2.8 on page 54, using either battery moves (Section 5.2.3 on page 48) or **MILP**-batteries moves (Section 5.2.5 on page 52).

The results are comparable with the ones obtained using **TS** with shift moves in instances without batteries (in fact the only difference is a couple swap moves once in a while). When batteries are available the computing times are higher, but the gap from optimum is much lower: using shift moves it was not possible to reach a high quality solution, so **TS** stopped earlier. For large instances this method reached a gap of 6% from the reference solutions, for small instances instead it produced solutions with much worse quality, over 13%. In small instances only 2 houses out of 20 have batteries, it is hard for this method to exploit such little capacity.

MILP-batteries moves produced instead low quality solutions, over 10% from the reference. In the following, when we refer to “mixed moves”, we mean mixed moves with battery moves, not **MILP**-batteries moves.

Figure 6.7: **TS** with **MILP**-zeroes-fixing move and mixed moves, typical local search progress

Batteries	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
No	1.67%	31.23	0.63%	40.87	1.15%	62.8
Yes	2.39%	31.53	2.40%	79.7	2.78%	177.43

Table 6.10: **TS** with **MILP**-zeroes-fixing move and mixed moves

6.3.4 Tabu Search with **MILP**-zeroes-fixing move and mixed moves

In Figure 6.7 we show the typical solution's value trend when using **TS** with **MILP**-zeroes-fixing move (Section 5.2.6 on page 52) followed by mixed moves. The reduced **MILP** is solved once at the first iteration of **TS**, then shift and battery moves refine the obtained solution. The improvement due to the reduced **MILP** move is huge, however it does not come without cost: solving the reduced **MILP** problem is not done iteratively, so the gap from optimum stays constant for a long time, until the new solution is found. Moreover, memory usage increases significantly when using CPLEX.

In Table 6.10 we show the results for this method. We generated 10 solutions with **GRASP** and we improved the best one with **TS** with **MILP**-zeroes-fixing move and mixed moves. **TS** stopped when it could not further improve the solution.

Results are slightly worse for instances with batteries, but compared to the other algorithms the difference is much smaller (and values are in general better). Note that the **TS** refines the solution obtained solving the reduced **MILP**, which has already good quality. For every subset of data set this method produced solutions within 3% from the reference solutions. Computing time is longer compared to battery moves, but still less than 3 minutes for the largest instances.

Action	k	20 houses		200 houses		400 houses	
		Gap	Time	Gap	Time	Gap	Time
Stop	3	5.82%	0.8	1.75%	25.43	1.81%	91.13
	5	5.39%	0.87	1.71%	28.9	1.80%	93.1
	7	5.16%	1.03	1.70%	32.4	1.77%	103.7
Diversify	3	3.50%	2.97	1.34%	133.8	1.37%	524.3
	5	2.83%	3.5	1.31%	130.9	1.35%	516.27
	7	3.37%	3.63	1.33%	133.63	1.35%	509.3
Continue	∞	3.80%	3.47	1.52%	128.57	1.51%	487.43

Table 6.11: Early stop and diversification in **TS**

6.3.5 *Tabu Moves*

In Figure 6.8 we show the typical progress of solution's value in **TS** for a couple instances with 20 and 200 houses. The qualitative behaviour is the same for other subsets of the data set. Solution's value is almost always decreasing, especially for large instances. The heuristic does not fall into and escape from many local minima, so the **TS** actually behaves more like the steepest descent algorithm, at least until the latest iterations.

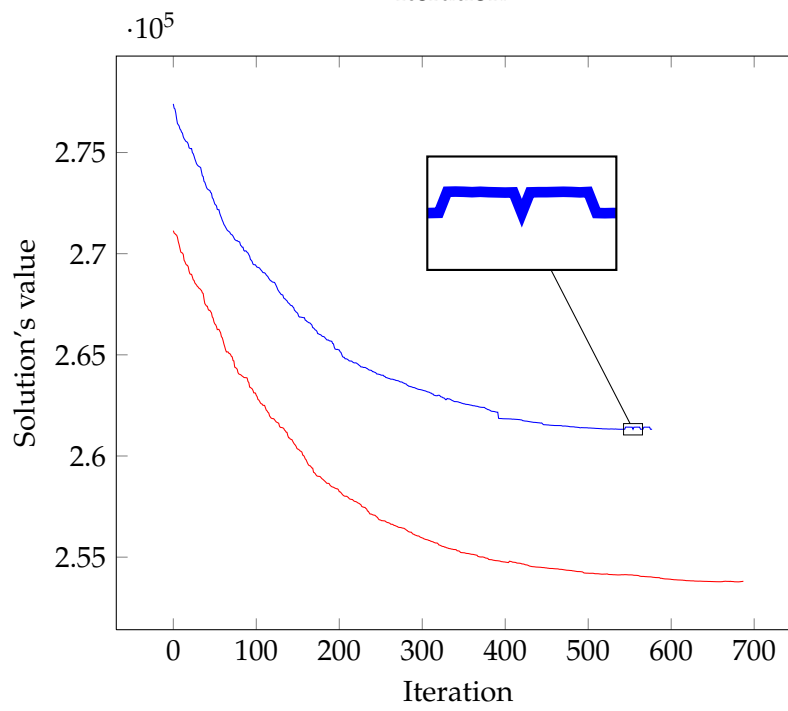
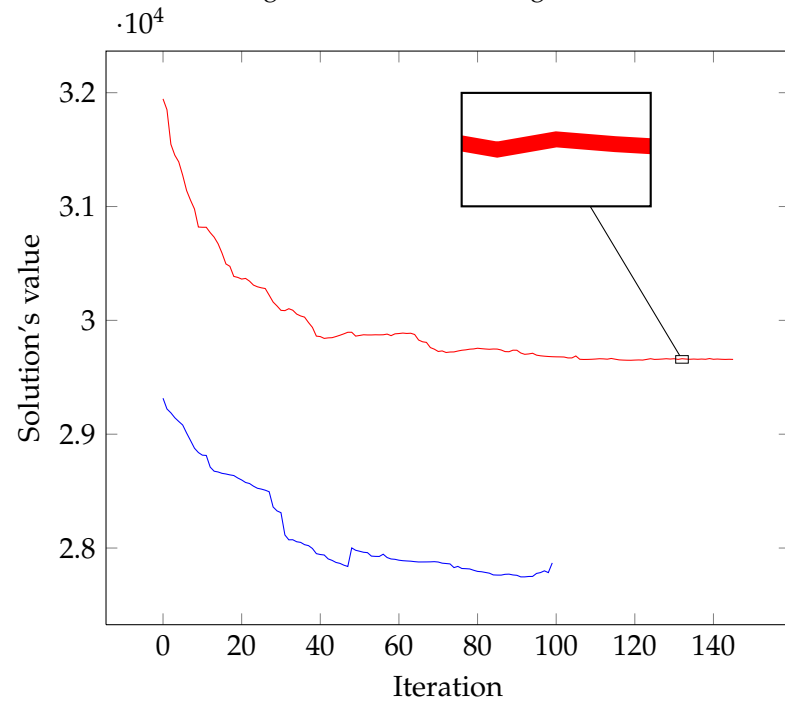
Since at each iteration the neighbourhood is quite large – each activity can be shifted to any feasible time slot – there are not many local minima when the solution is far from the optimum, there is often a large number of improving moves, so the Tabu List (**TL**) can not forbid all of them. This is no longer true in the latest iterations, where in facts we observe some oscillation.

6.3.6 *Early stop and diversification*

When **TS** could not manage to improve the current solutions in the past $k \cdot t$ iterations, where t is the current **TL** size, and k is a parameter, it can either stop, diversify or continue until it reaches the maximal iterations limit (Section 5.2.11 on page 60). In Table 6.11 we show the results when using different values of k and when diversifying or stopping.

There is no significant difference whether the heuristic stops as soon as no improvements are possible, instead of continuing or diversifying, while the computing time is much shorter, about 5 times lower.

Figure 6.8: Tabu List usage



Feasibility	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
Feasible	5.39%	0.87	1.71%	28.9	1.80%	93.13
Infeasible	6.35%	0.83	1.36%	37.23	1.35%	127.8

Table 6.12: Infeasible exploration in **TS**

6.3.7 Infeasible exploration

In Table 6.12 we show the results when enabling the exploration of the infeasible region. In that case we show both the best feasible gap and the best infeasible gap. We generated 10 solutions with **GRASP** and we improved the best one with **TS** with mixed moves. **TS** diversified when it could not further improve the solution, for up to 5 times. The computing time includes the time to generate the initial solutions.

Except for small instances, we achieved slightly better results but in longer time, about 1.3 times higher. While in feasible exploration as soon as a local minimum is reached **TS** can stop and return the solution, in infeasible exploration it must first bring it back to the feasible region, and then to find a feasible local minimum.

In the same plot we can see how a diversification can result in a lower local minimum.

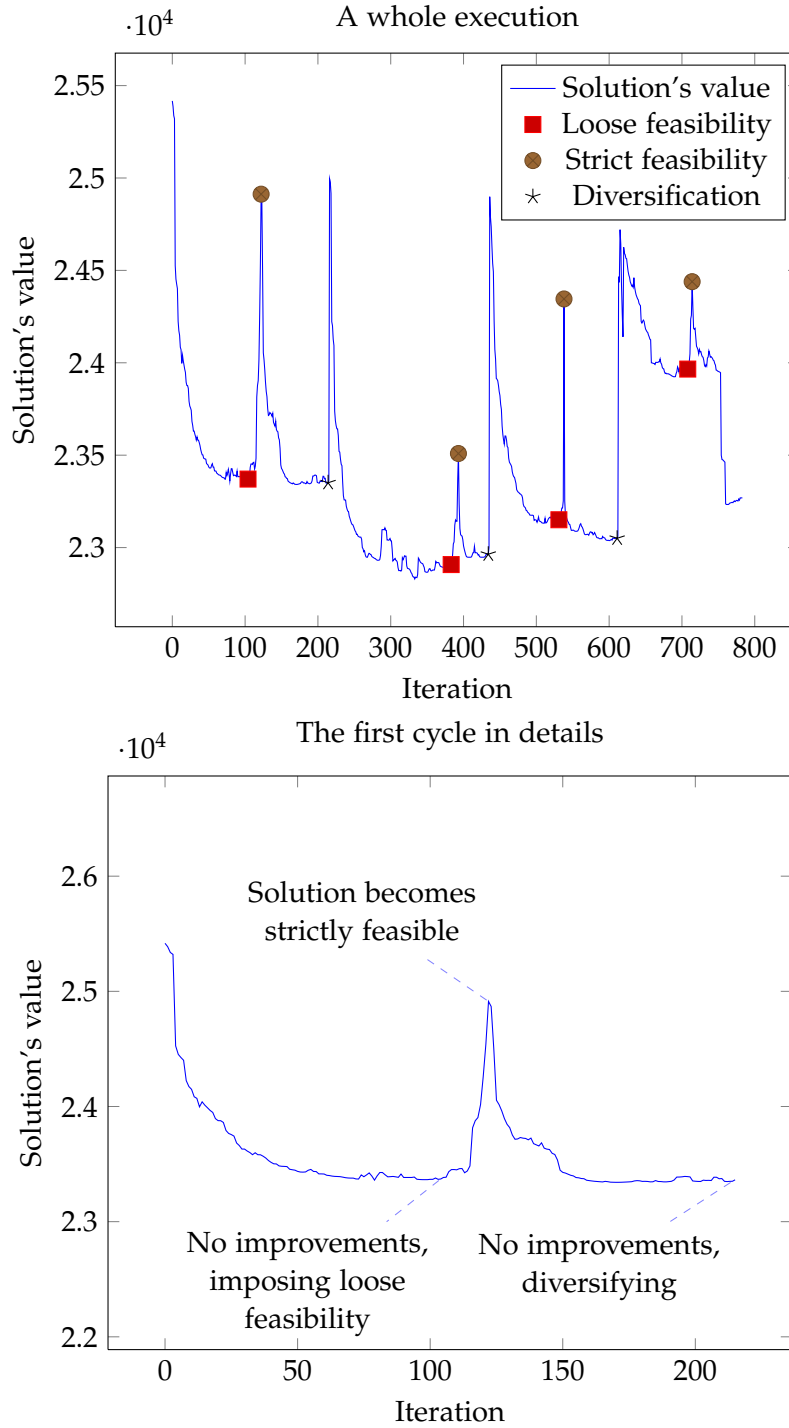
In Figure 6.9 we show the solution's value's progress against iteration when enabling infeasible exploration. **GRASP** generated an infeasible solution, so the local search starts in the infeasible phase. The solution is improved until no improvements occurred in the latest iterations (small fluctuations are due to Tabu Moves (**TMS**)). At this point the solution is in a local minimum, however it is not feasible, so the heuristic enters in the loosely feasible phase in order to drift the solution toward the feasible region. This is done by reducing the slacks until the solution is strictly feasible. At this point the heuristic enters in the strictly feasible phase: the solution is improved again until it reaches another local minimum, this time a feasible one.

At the end the heuristic either stops the local search or performs a diversification move. In the latter case the heuristic enters again in the infeasible phase, and the procedure starts over from the beginning.

6.4 PARTIAL LINEAR RELAXATION AND REDUCED **MILP**

In Table 6.13 we show the results for the **PLR** and reduced **MILP** method, described in Section 4.4 on page 43. We first computed the Partial Linear Relaxation with a time limit, then we fixed approximately 60% of variables already at 0, then we solved the resulting reduced **MILP** with a time limit. The solutions generated, called *initial*

Figure 6.9: Infeasible exploration in TS, details



Batteries		20 houses		200 houses		400 houses	
		Gap	Time	Gap	Time	Gap	Time
No	Initial	0.86%	31.0	0.19%	76.13	0.12%	188.67
	Improved	-0.07%	61.0	0.00%	138.14	0.03%	342.93
Yes	Initial	1.76%	31.1	2.73%	88.63	3.27%	227.1
	Improved	1.62%	60.3	2.29%	147.56	2.10%	384.36

Table 6.13: [PLR](#) and reduced [MILP](#)

Batteries	20 houses	200 houses	400 houses
No	10 out of 10	9 out of 10	9 out of 10
Yes	10 out of 10	7 out of 10	5 out of 10

Table 6.14: [PLR](#) and reduced [MILP](#), instances solved

in the table, were then improved using [TS](#) with mixed moves, for the same time limit. The time limit was set to 30, 60 and 150 seconds for instances with 20, 200 and 400 houses.

The initial solutions are very good, for instances without batteries the gap from reference is almost nil, while for instances with batteries is within 4%. We even manage to slightly improve them with [TS](#). However this method could not manage to solve all the instances, as we show in Table 6.14. Sometimes the time limit was too short to even produce a feasible solution.

For small instances the [PLR](#) is almost instantaneous, so the total computing time depends only on the reduced [MILP](#) and the local search. We see a similar effect for larger instances without batteries, the [PLR](#) is solved before the time limit. Sometimes, especially when no batteries are available, the reduced [MILP](#) is also solved before the time limit.

6.5 LOCAL BRANCHING

In Table 6.15 we show the results for local branching. We generated 10 solutions with [GRASP](#) and we improved the best one with local branching. Local branching could actually generate itself an initial solution, however it usually takes very long time, so we supplied one from [GRASP](#). We imposed a time limit on each local branching iteration of 30, 60 and 150 seconds for instances with 20, 200 and 400 houses.

For small instances local branching produced high quality solutions, within 2% from the reference, but in long time. For large instances, especially when batteries are available, it produced low qual-

Batteries	20 houses		200 houses		400 houses	
	Gap	Time	Gap	Time	Gap	Time
No	0.61%	92.53	0.41%	79.3	4.47%	1772.3
Yes	1.69%	92.87	5.81%	1444.24	13.04%	2627.4

Table 6.15: Local branching

ity solutions, up to 13% from reference, and the computing time grew to impracticable values.

6.6 COMPARISON

We run tests over the various methods in order to see which method computed the best solutions in a given time. We tried to set the time limit to a value meaningful for all the algorithms, but this was not possible. Some algorithms reached a local optimum in really short time. Some other algorithms, namely the ones that solved a reduced **MILP** problem, need enough time to generate at least a feasible solution. For these reasons the computing time were not the same for different methods, however we tried to impose comparable times for instances of the same size. In Figure 6.10 we show the average gap from optimum, and in Figure 6.11 we show the average computing time.

TS with shift moves produces good solutions when no batteries are available, within 2% from the reference for large instances. When instead batteries are available this method is not able to make use of them; in fact it generates almost the same solutions whether batteries are available or not, but of course in the former case the optimal is much lower. Solutions are also slightly worse for small instances, however the computing time is very short, about 1 second in average. Even for large instances it has the lower computing time, less than 90 seconds.

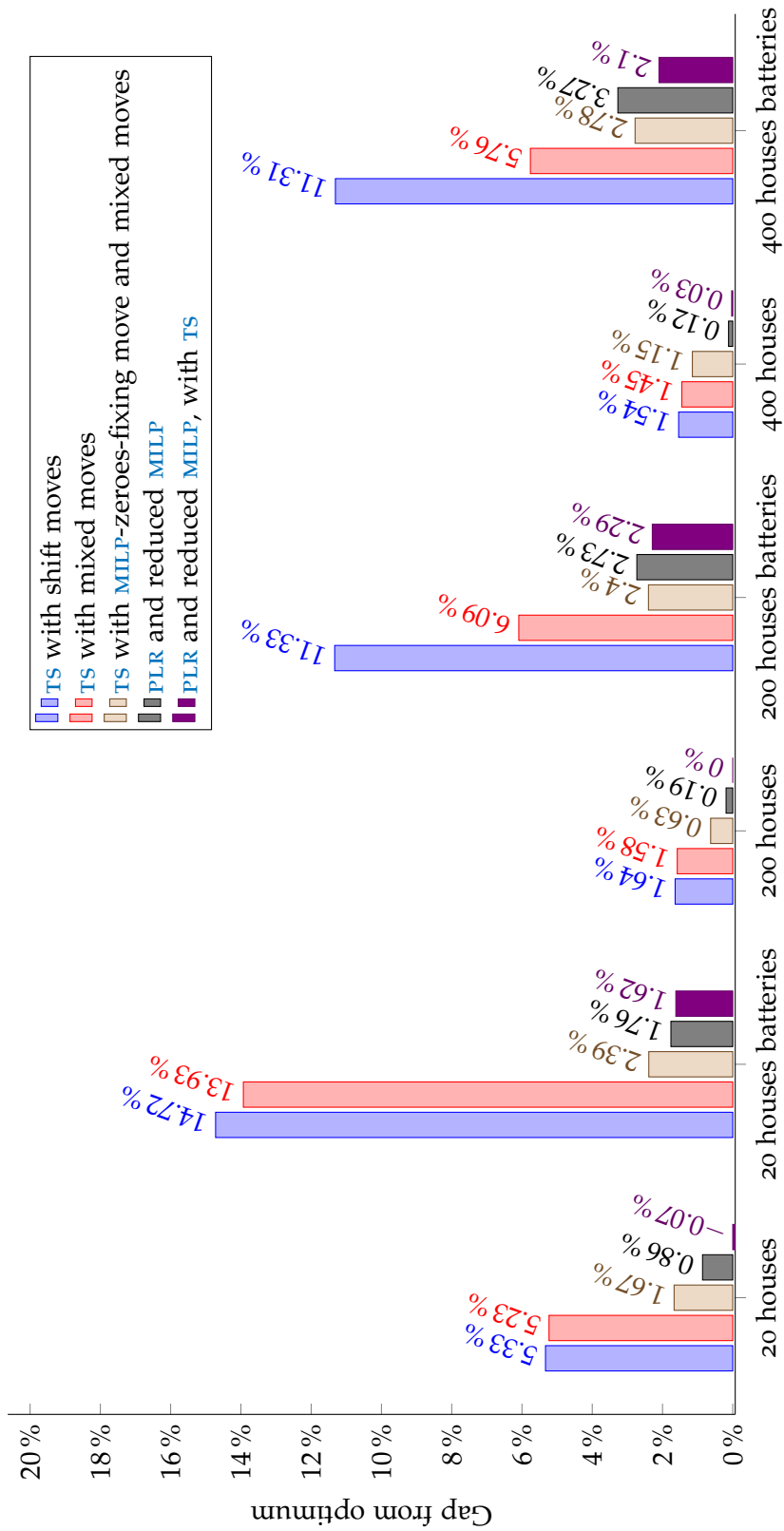
We can make the same observations for **TS** with mixed move when no batteries are available (mixed moves consist mostly of shift moves and battery moves). This method produced better solutions for instances with batteries, within 6% from reference, but not for small instances. Larger instances with batteries are still not very close to the reference, but the gap is smaller than when using shift moves.

Solutions obtained with **TS** with **MILP**-zeroes-fixing move and mixed moves are the closest to the reference for the **TS** variants; even in instances with batteries the gap from reference is very small, less than 3%. This method took a slightly longer time to finish, but it is probably the best choice to solve the Residential Energy Load Management Problem. One drawback is that solving a reduced **MILP** with the CPLEX solver may require a larger amount of memory, and for instances much larger than the ones considered it might even be impossible.

PLR and reduced **MILP**, followed or not by **TS**, gave the best solutions, the gap from reference is almost zero for instances without batteries, and below 3% otherwise. However this method took significantly longer than the others, up to 5 minutes, and in addition to that it could not even produce a feasible solution for some instances, especially when batteries were available. Moreover it suffers too from the drawback of solving a reduced **MILP**.

We excluded from the comparison the other methods that gave bad results or had longer computing time.

Figure 6.10: Comparison, gap from optimum



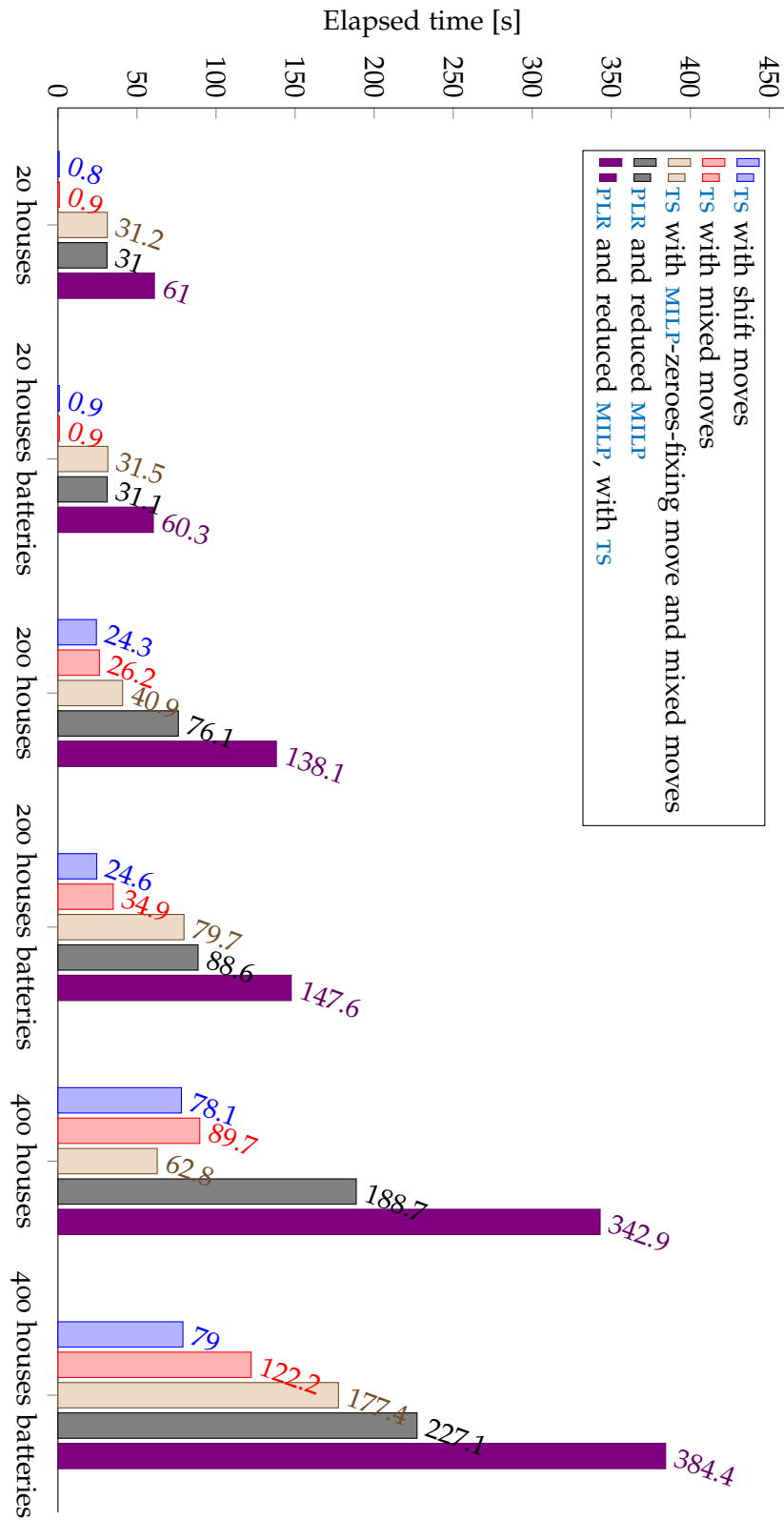
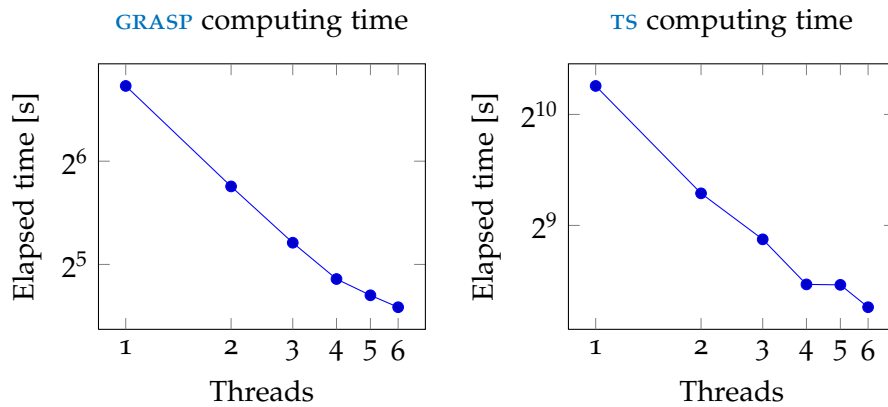


Figure 6.11: Comparison, computing time

Figure 6.12: Computing times when using multi-threading



6.7 MULTI-THREADING

In Figure 6.12 we show our program's computing time with respect to the number of threads running simultaneously. We generated 240 initial solutions with **GRASP** and improved the 24 best with 2048 iterations of **TS**. We run this test on a 4 **CPU** machine. Note that in these plots both axis have logarithmic scale. The computing time decreases approximately linearly over the number of working threads, as long as they correspond to physical **CPU**s. We make some remarks about multi-threading in Section A.3 on page 106.

CONCLUDING REMARKS

In this chapter we summarize the main contributions of the thesis and we mention a few directions for future work.

When we started working on the thesis the original [MILP](#) model was very inefficient, computing time was very high even for small instances with 10 houses and 11 appliances. For large instances with more than 100 houses it would not provide a feasible solution even after 1 hour of computing time. The model have been improved in parallel to this thesis. The two changes that we discussed in [Section 3.2 on page 26](#) and in [Section 3.3 on page 28](#) led to a much more compact model, which can be solved considerably faster. In some way this decreased the need for an heuristic, but on the other hand it allowed us to easily embed reduced [MILPs](#) in the heuristic, which actually led to better results.

We started by developing a Greedy Randomized Adaptive Search Procedure ([GRASP](#)) in order to generating initial feasible solutions ([Section 4.2.1 on page 30](#)). Such method turns out to be very fast but tends to generate low quality solutions.

In order to evaluate solutions quality we defined an energy profile ([Section 4.2.2 on page 30](#)) and procedures to incrementally update it when adding and removing loads ([Section 4.2.3 on page 31](#)). Since the maximal peak objective function was a bottleneck function we investigated and developed other objective functions ([Section 4.3 on page 36](#)).

We then developed a [TS](#) algorithm which involves different types of moves. We first focused on shift moves ([Section 5.2.1 on page 48](#)), which produced high quality solutions in short computational time for instances without batteries. Since batteries made the problem harder, we first tried to allocate them at the end of [TS](#), but this approach produced low quality solutions. The only effective way to take into account batteries is to optimize them during [TS](#), therefore we developed the battery move ([Section 5.2.3 on page 48](#)) and the [MILP](#)-battery move ([Section 5.2.5 on page 52](#)). The [TS](#) based on the former move gave better results, but the solutions were still not very close to optimal.

Finally we devised a [TS](#) variant using the [MILP](#)-zeroes-fixing move ([Section 5.2.6 on page 52](#)), which even when batteries are available produces near optimal “raw” solutions, that could further be improved with shift moves.

For comparison purposes we implemented also two alternative algorithms: Partial Linear Relaxation ([PLR](#)) with reduced [MILP](#), and local branching. The former gave very high quality solutions, but re-

quired higher computing time than **TS**: moreover it could solve only a subset of instances. For local branching, computing times turns out to be much higher (up to several order of magnitude) than for **TS**.

The heuristics we developed showed to be very effective, they generated solutions within 3% from the reference ones in very short computing time, less than 3 minutes for the largest instance, and this looks promising for even larger instances, for which we could not compute reference solutions.

7.1 FUTURE WORK

To conclude we mention a few directions for future work.

As far as the problem is concerned, it would be interesting to allow energy flow from a house to another. Whenever a house has exceeding energy it might give it to other houses. Batteries and **PV** panels might then be shared among the houses. Batteries-equipped houses might act as storage for the whole aggregate, collecting energy from the grid or the **PV**-equipped houses; the retailer might instead set up an internal market, where houses compete with the main energy producer in the grid.

In the problem the activities scheduling is optimized over a single day, from midnight to midnight. Users can not specify execution windows that end at late night. It would be interesting to overcome this limitation, either by chaining multiple days and optimizing over a longer period, or by finding a way to impose “boundary conditions”.

As far as the heuristic is concerned, it would be interesting to utilize non flat ideal solutions. In order to minimize the maximal peak of the demand curve, we minimized the distance between the current curve and the flat ideal one. However, there are no limitations on the shape of the ideal curve, the algorithms (and even the program) we developed can be used with other shapes¹. A non flat ideal curves might be useful in a scenario where **PV** panels produce a significant amount of energy. If a production forecast is available, the retailer might set an ideal curve which is higher during such time slots, in order to maximize **PV** panels usage.

The users specify their appliances execution windows a day ahead, so that the energy retailer can find the optimal scheduling and buy the energy in advance. If an user is in the need to use an appliance regardless of the scheduling, the retailer has to re-negotiate the energy supply. It might be interesting to let the user change the appliances execution window only few time slots in advance, fix the already started activities, and find the new optimal scheduling. The retailer would then re-negotiate for a smaller amount of energy.

¹ We actually made a couple experiments with an exponential increasing ideal curve (Figure 4.3 on page 38)

The most effective way to exploit batteries is to optimize them together with activities, as we partially did with battery moves. It would be interesting to analyse the structure of optimal solutions for small instances, in order to identify some structures that could be exploited to better combine batteries and activities.

IMPLEMENTATION

In this chapter we give a brief overview about the implementation and we discuss in detail some important or interesting aspects, such as:

- what data structures we used to efficiently explore the neighbourhood;
- how we implemented an extensible framework for local search methods.

CONTENTS

A.1	Data structures for efficient neighbourhood exploration	96
A.1.1	Incomplete energy profile	96
	Incomplete energy profile in other algorithms	100
A.1.2	Move semantic	100
A.2	Local search framework	100
A.2.1	The algorithm	101
	Early stop and diversification	101
A.2.2	Template policies	105
	Why template parameters?	105
	Policies	106
A.3	Multi-threading	106

We implemented all our algorithms in C++. We chose C++ because we thought it was the language with the best balance among control and expressiveness for the type of application we needed. The program is CPU bound, i. e., the CPU is the most used component. Higher level languages provide many convenient features such as bounds checking for arrays or garbage collection, but the user is forced to pay for them even when not used. Lower level languages grants instead even more control, e. g., no constructors nor destructors implicit calls, but lack most basic data structures such as vectors, lists or hashes, and other features such as exception handling and (partial) functional paradigm. We strived to use known best practices to make the code as safe, as readable and as efficient as possible.

The main core is written in plain standard C++98 (in some places we optionally enabled C++11 new feature move semantic, see Section A.1.2 on page 100), while we made use of Boost library for auxiliary operations, such as multi-threading and options parsing [Karls-son, 2005]. The program should be fairly portable, with the exception of the component which executes external processes, which currently

works only on the Windows® operating system¹ and on **POSIX** systems. We used the AMPL solver with the IBM ILOG CPLEX 12.4.0.0 solver for solving **MILP** problems.

A.1 DATA STRUCTURES FOR EFFICIENT NEIGHBOURHOOD EXPLORATION

As described in Section 4.2.2 on page 30, a solution contains:

- Starting slots for each activity (h, a) ;
- Flows for each house, battery and time slot (corresponding to $v_{h,b,t}^c - v_{h,b,t}^d$, i. e., when positive the battery is charging, when is negative the battery is discharging);
- Energies for each house, battery and time slot (corresponding to $e_{h,b,t}$);
- A list of all activities that influence each time slot;
- An energy profile, i. e., the data about how much energy is bought and sold by each house and by the aggregate for each time slot (corresponding to some combination of $y_{h,t}$).

The energy profile is actually a separate entity from a solution since there are other entities which can have an energy profile).

A.1.1 Incomplete energy profile

In this section we explain how we implemented the generation and exploration the neighbourhood at each Tabu Search (**TS**) iteration. This implementation detail turned out to be very important, as using the wrong data structures results in significantly longer execution times. We describe our initial implementation and how we improved it until the final one.

*Naïve
implementation of
neighbourhood
exploration results
in longer execution
time*

In **TS** we pick the best solution in the current solution's neighbourhood

$$\underline{x}_{k+1} \leftarrow \min N_k.$$

We might create the set N_k and then to find its minimum, but this requires very large memory. A better approach is to store only the current best solution in the neighbourhood during its exploration (the initial best solution has to have ∞ as value if the iteration can be non-improving), as we show in Algorithm A.1.

¹ Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Algorithm A.1 Neighbourhood's exploration, first version

```

FIND-BEST-NEIGHBOUR-1( $\underline{x}_0$ )
1   $\underline{x}^* \leftarrow \infty$ 
2  for  $m \in \text{ALL-POSSIBLE-MOVES}$ 
3      do
4           $\underline{x} \leftarrow \text{COPY}(\underline{x}_0)$ 
5           $\triangleright$  Compute the resulting solution after the move
6           $\text{APPLY-MOVE}(\underline{x}, m)$ 
7           $\text{UPDATE-OTHER-VARIABLES}(\underline{x})$ 
8          if  $\underline{x} < \underline{x}^*$ 
9              then  $\underline{x}^* \leftarrow \underline{x}$ 
10 return  $\underline{x}^*$ 

```

The “other variables” are those variables that are not used to evaluate the solution, e. g., the activities in each time slot. In the current iteration we do not need those variables, it is therefore useless to update them for solutions that will be discarded, we can do it only for the best solution found, as we show in Algorithm A.2.

Algorithm A.2 Neighbourhood's exploration, second version

```

FIND-BEST-NEIGHBOUR-2( $\underline{x}_0$ )
1   $\underline{x}^* \leftarrow \infty$ 
2  for  $m \in \text{ALL-POSSIBLE-MOVES}$ 
3      do
4           $\underline{x} \leftarrow \text{COPY}(\underline{x}_0)$ 
5           $\triangleright$  Compute the resulting solution after the move
6           $\text{APPLY-MOVE}(\underline{x}, m)$ 
7          if  $\underline{x} < \underline{x}^*$ 
8              then  $\underline{x}^* \leftarrow \underline{x}$ 
9           $\text{UPDATE-OTHER-VARIABLES}(\underline{x}^*)$ 
10 return  $\underline{x}^*$ 

```

There is a subtle problem with this method that makes the program extremely slow: we copy the entire initial solution in a temporary variable, and we update its energy profile. Then if there is an improvement we copy the entire current solution in the best solution variable. Unfortunately copying an entire solution is extremely expensive.

A solution contains the starting slots, the batteries energies and flows, the list of activities for each time slot and the energy profile. During neighbourhood exploration we only use the energy profile, since all the other variables are the same as the current solution. Even

*Copying a solution
is expensive*

if we save time by delaying updating these variables, we still waste time in copying them.

We then decoupled the energy profile from the solution. We are only interested in the neighbours energy profile – which is used by the objective function to evaluate them – not in any of the other variables. We can skip altogether copying the starting time slots, the batteries energy and flow. We do it *una tantum* at the end of iteration, as we show in Algorithm A.3.

Algorithm A.3 Neighbourhood’s exploration, third version

```

FIND-BEST-NEIGHBOUR-3( $\underline{x}_0$ )
1  ▷ Use the energy profile instead of the solution
2   $p^* \leftarrow \infty$ 
3   $m^* \leftarrow \text{INVALID-MOVE}$ 
4   $p_0 \leftarrow \text{ENERGY-PROFILE}(\underline{x}_0)$ 
5  for  $m \in \text{ALL-POSSIBLE-MOVES}$ 
6      do
7           $p \leftarrow \text{COPY}(p_0)$ 
8          ▷ Compute the resulting profile after the move
9           $\text{APPLY-MOVE}(p, m)$ 
10         if  $p < p^*$ 
11             then
12                  $p^* \leftarrow p$ 
13                  $m^* \leftarrow m$ 
14   $\underline{x}^* \leftarrow \text{UPDATE-PROFILE}(\underline{x}_0, p^*)$ 
15   $\text{UPDATE-OTHER-VARIABLES}(\underline{x}^*)$ 
16  return  $\underline{x}^*$ 

```

Finally, we can identify another set of variables that we could avoid copying. During the neighbourhood exploration we apply a set of moves to the initial solution, generating all the neighbours. Different moves involve different houses, but a single move involves a single house², and the energy profiles of all other houses are not affected by the move. We are copying the full set of houses energy profiles, while only one is actually changing.

*Only the relevant
house’s profile is
copied*

We then split the energy profile in three different structures:

GLOBAL PROFILE It stores the aggregate profiles (bought, sold, difference), space complexity is $O(t)$;

ENERGY PROFILE It stores a global profile and the full set of houses demand curves, space complexity is $O(t + ht)$;

² This is true for shift moves but false in general. The point is that a single move involves a *small* subset of the houses.

INCOMPLETE ENERGY PROFILE It stores a global profile and a subset of the houses demand curves, space complexity is $O(t + kt)$.

t and h are respectively the number of time slots and houses in the instance. k is the number of houses involved in the move, it is 1 for shift moves, and in general $k \ll h$.

For each neighbour we then make an incomplete copy of its energy profile, and we update only the global profile (which is used by the objective function to evaluate a solution) and the demand curves of the houses involved in the move. We fetch the other houses demand curves from the initial solution only at the end of the iteration. We show this final version in Algorithm A.4.

Algorithm A.4 Neighbourhood's exploration, fourth and final version

FIND-BEST-NEIGHBOUR-4(\underline{x}_0)

```

1  ▷ Use an incomplete energy profile
2   $ip^* \leftarrow \infty$ 
3   $m^* \leftarrow \text{INVALID-MOVE}$ 
4  for  $m \in \text{ALL-POSSIBLE-MOVES}$ 
5      do
6           $ip_0 \leftarrow \text{INCOMPLETE-COPY}(\underline{x}_0, h_m)$ 
7          ▷ Compute the resulting incomplete profile after the move
8           $ip \leftarrow \text{APPLY-MOVE}(ip, m)$ 
9          if  $ip < ip^*$ 
10             then
11                  $p^* \leftarrow p$ 
12                  $m^* \leftarrow m$ 
13   $\underline{x}^* \leftarrow \text{UPDATE-PROFILE}(\underline{x}_0, ip^*)$ 
14   $\text{UPDATE-OTHER-VARIABLES}(\underline{x}^*)$ 
15  return  $\underline{x}^*$ 

```

We can now wonder whether there is room for more improvements. For short activities we still copy the full house's demand curve while we actually need only few time slots (this is especially true for battery moves, which involve only a two time slots). Such a change would however be much more complicate. First of all we still need the whole profile for computing the p -norm of difference and comparing solutions, so we still have to loop over $O(kt)$ elements, whether we copy all of them or we leave some in the initial profile. Secondly, in multiple moves iterations (e. g. swap moves) we would have to keep track of which time slots are involved for each house, taking care of possible overlapping. An improvement would hardly justify the huge additional complexity it would cause.

Incomplete energy profile in other algorithms

In every algorithm where we copy the initial energy profile, apply a move on it, and compare it to the best one, we can use the incomplete profile to reduce execution time. We copy only the relevant houses demand curves while we later fetch the remaining – that are not used to compute the global profile nor for comparisons – directly from the original energy profile. This applies to every kind of moves, and to [GRASP](#) as well.

A.1.2 *Move semantic*

Another minor improvement for avoiding useless copies comes out-of-the-box with the new C++11 standard: move semantic. Whenever we build a temporary value and then copy it to another variable we are making an useless copy. The same is true when we return a complex value from a function and assign it to a variable: the copy constructor is called for the operation.

Objects are moved instead of copied and discarded

Using move semantic we can force *moving* variables instead of *copying* them. When we move a variable to another the latter is overwritten, and the former is emptied and left in a destructible state. There is no longer overhead to construct a complex variable and moving its value to another one.

A.2 LOCAL SEARCH FRAMEWORK

In this section we describe in detail the implementation of an extensible framework for local search algorithms. The framework can be used with different types of moves, and its behaviour can be specialized through different policies.

The *local search framework* is a set of template classes which implement local search and [TS](#). Local Search Framework is the main class, and it implements the basic local search algorithm: given an initial solution, its neighbourhood is iteratively explored for the next solution. *How* the neighbourhood is explored is not a Local Search Framework's concern, this operation is demanded to the Exploration Policy interface, it is therefore possible to seamlessly use the framework with different types of moves .

High level local search algorithm is decoupled from the neighbourhood exploration algorithm

The framework offers support for a Tabu List ([TL](#)), i. e., at each iteration it fetches the current Tabu Move ([TM](#)) from Exploration Policy and stores it in a list. At the next iteration it forwards the [TL](#) to Exploration Policy. The latter is so lifted from the burden of managing a [TL](#), possibly dynamically changing its size; it only has to provide a new [TM](#) at the end of its iteration, and it gets a valid [TL](#) at the next iteration. In [TS](#) an iteration may be non-improving, so the framework

also stores the best solution found so far – and the best *feasible* one, since it is possible to explore the infeasible region.

In Figure A.1 we show the class diagram of the Local Search Framework, and in Figure A.2 we show the concrete classes that implements the Exploration Policy interface. Each exploration policy explores a neighbourhood corresponding to a move, as described in Chapter 5 on page 45.

A.2.1 The algorithm

We show the complete procedure implemented in Local Search Framework in Algorithm A.5 on page 103. \underline{x}_0 is the initial solution, \underline{x}^* is the best solution found, $\underline{x}_{\text{feasible}}^*$ is the best feasible solution found, and \underline{x} is the current solution.

recent-list and *tabu-list* are First-In First-Out (FIFO) queues with maximal capacity (whenever their size grows above their capacity the least recent element is discarded). The former contains the best feasible solutions, and it is used to keep track of whether the solution is improving, and the latter contains the TMS associated with the recent moves.

state is set to the current feasibility state, i. e., STRICT-FEASIBILITY, LOOSE-FEASIBILITY or INFEASIBILITY. The algorithm executes different branches depending on in which state it is. *slacks* variable contains the slacks of infeasible solutions. Note that the neighbourhood exploration might return a neighbour which differs from the current solution only for a subset of houses; in this case only related slacks are updated.

Early stop and diversification

At the end of each iteration the framework might stop early depending on the result of a helper function, that we show in Algorithm A.6. The current solution is compared to the best solutions \mathbb{N} iterations ago, if it has not improved then the framework stops. \mathbb{N} must be larger than the TL size, otherwise the whole purpose of TMS is defeated: few non-improving iterations might be useful to climb up local minima's walls.

If the current solution was infeasible, then the framework switches to loose feasibility phase, in order to recover feasibility. Otherwise, if the current solution was instead strictly feasible, then the framework either returns the best feasible solution found, or diversifies. During loosely feasible phase most iterations are non-improving, so the framework never stops early.

In case of diversification the framework merely forwards the request to the Exploration Policy. In case the latter is not able to perform a diversification, the execution ends.

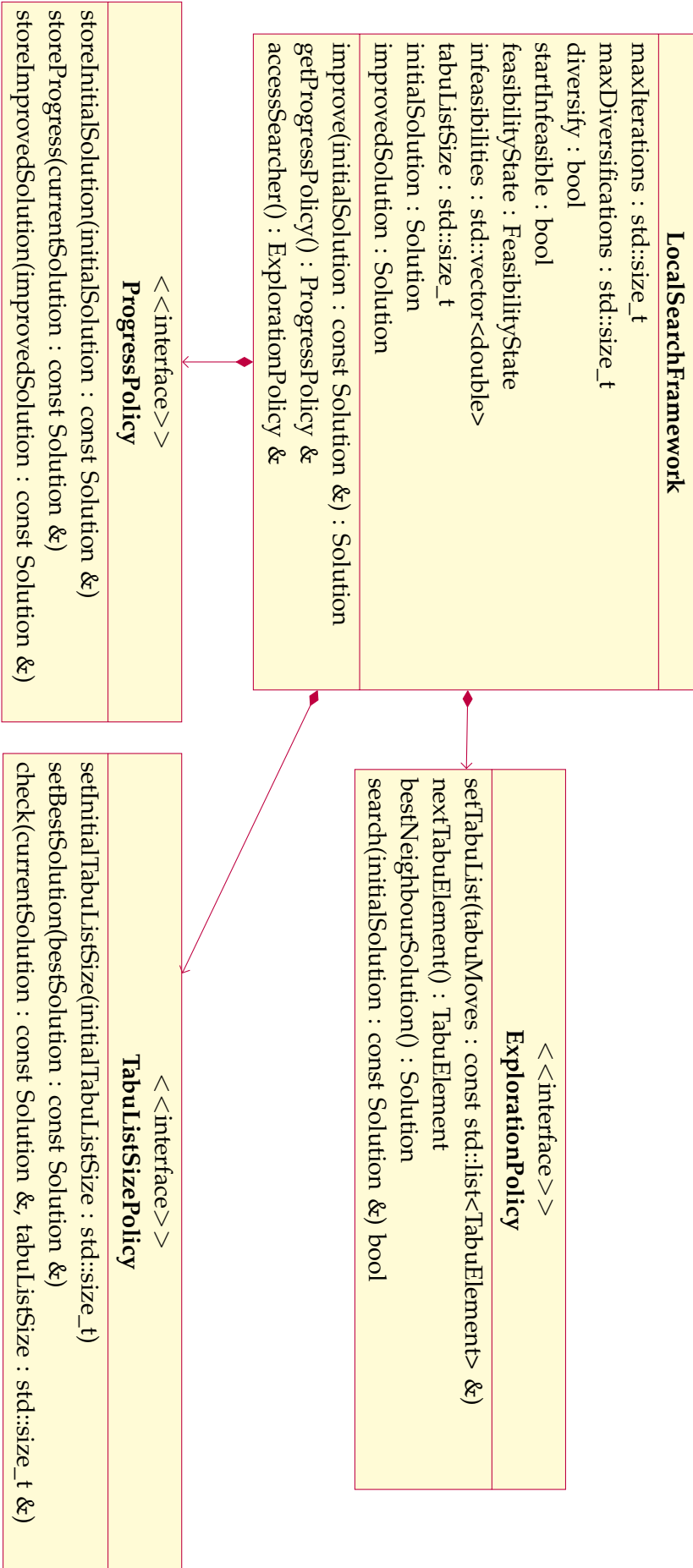


Figure A.1: Local search framework's class diagram

Algorithm A.5 Complete local search

```

LOCAL-SEARCH( $\underline{x}_0$ )
1   $\underline{x}^* \leftarrow \underline{x}_0$ 
2  if FEASIBLE( $\underline{x}_0$ )
3      then  $\underline{x}_{\text{feasible}}^* \leftarrow \underline{x}_0$ 
4      else  $\underline{x}_{\text{feasible}}^* \leftarrow \infty$ 
5   $\underline{x} \leftarrow \underline{x}_0$ 
6  recent-list  $\leftarrow \emptyset$ 
7  tabu-list  $\leftarrow \emptyset$ 
8  slacks  $\leftarrow$  INITIALIZE-SLACKS( $\underline{x}_0$ )
9  state  $\leftarrow$  INITIAL-FEASIBILITY-STATE
10 for  $i \leftarrow 0$  to MAX-ITERATIONS
11     do
12          $(\underline{x}, \text{next-tabu-move}) \leftarrow$  FIND-NEIGHBOUR( $\underline{x}, \text{tabu-list}, \text{slacks}$ )
13         tabu-list  $\leftarrow \text{tabu-list} \cup \text{next-tabu-move}$ 
14          $\triangleright$  Dynamically enlarge or reduce tabu list
15         UPDATE-TABU-LIST-SIZE()
16         STORE-PROGRESS( $\underline{x}$ )
17         if state  $\neq$  STRICT-FEASIBILITY
18             then slacks  $\leftarrow$  UPDATE-SLACKS( $\underline{x}$ )
19                 if state = STRICT-FEASIBILITY
20                      $\triangleright$  Previous solutions were infeasible, so they are discarded
21                     then recent-list  $\leftarrow \emptyset$ 
22                 if  $\underline{x} < \underline{x}^*$ 
23                     then  $\underline{x}^* \leftarrow \underline{x}$ 
24                 if  $\underline{x} < \underline{x}_{\text{feasible}}^* \wedge \text{state} = \text{STRICT-FEASIBILITY}$ 
25                     then  $\underline{x}_{\text{feasible}}^* \leftarrow \underline{x}$ 
26                 if SHOULD-STOP-EARLY?( $\underline{x}, \text{recent-list}, \text{state}$ )
27                     then stop
28 return  $\underline{x}^*$ 

```

Algorithm A.6 Local search early stopping criterion

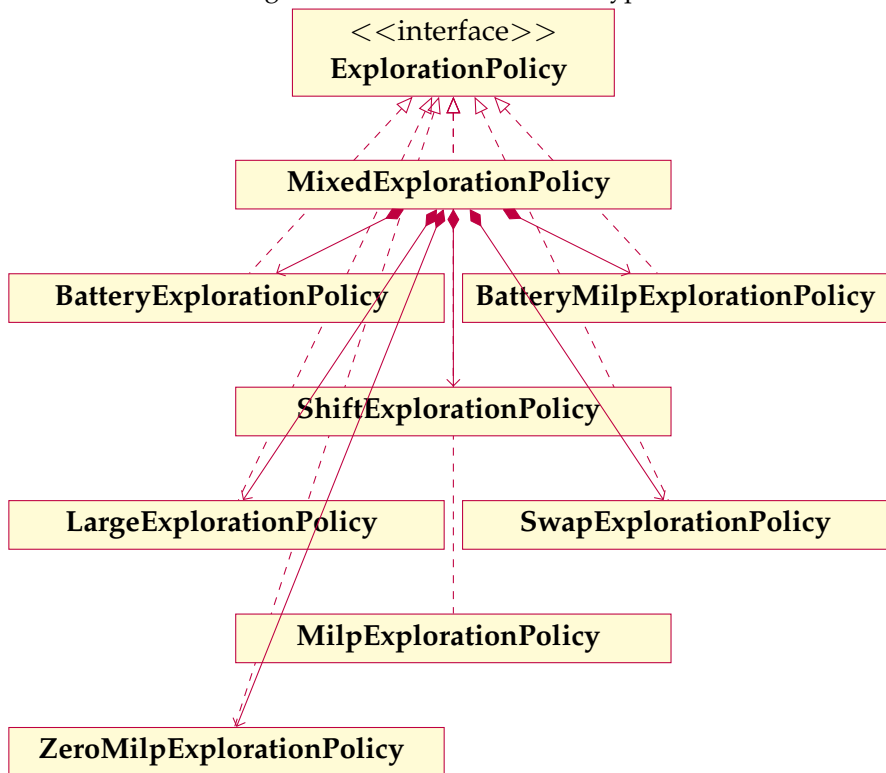
SHOULD-STOP-EARLY?(\underline{x} , *recent-list*, *state*)

```

1  if  $\underline{x} \geq \text{recent-list}_0$ 
2    then
3       $\triangleright$  The best solution has not improved over the last few iterations
4      if state = STRICT-FEASIBILITY
5        then
6           $\triangleright$  We hit the bottom of feasible region
7          if  $d < \text{MAX-DIVERSIFICATIONS}$ 
8            then  $d \leftarrow d + 1$ 
9               $\text{recent-list} \leftarrow \emptyset$ 
10             DIVERSIFY()
11          else
12            return true
13        elseif state = INFEASIBILITY
14          then
15             $\triangleright$  We hit the bottom of infeasible region
16             $\triangleright$  We head back to feasibility
17            state  $\leftarrow$  LOOSE-FEASIBILITY
18        else
19           $\triangleright$  The best solution has improved over the last few iterations
20           $\text{recent-list} \leftarrow \text{recent-list} \cup \underline{x}$ 
21  return false

```

Figure A.2: Concrete iteration types



A.2.2 Template policies

As said before, Local Search Framework class defines a general algorithm, relying on other entities to perform single steps of that algorithm. This is called *template method pattern*, and we implemented it using *template classes policies* (despite sharing the “template” name, the two concepts are not related).

Why template parameters?

At early stages of development there was only one local search method: the Tabu Search (TS) method with shift moves (Section 5.2.1 on page 48), implemented with regular functions. Later another method was added: the TS with MILP moves (Section 5.2.4 on page 52). Since they were both TS methods, they shared the same external algorithm which handles the TL and drives the exploration, their only difference is how to explore the neighbourhood of a solution. Hence, we wanted to factor out such external algorithm. This is a typical use case for the *template method pattern*: a common algorithm makes use of different functions depending on the chosen method. It can be implemented in different ways, the most common being abstract virtual functions implemented in derived classes. An alternative could be bare function pointers.

A basic algorithm is specialized using template method pattern

Virtual functions could take care of the behavioural aspect, but there was another issue: **TMs** were of different types in the two methods – they were either single moves or set of moves. What type should the **TL** had stored? Single moves could be represented as set of one move, but this is hardly elegant and lesser useful if new methods were added with entirely different moves.

Template parameters
allow to specialize
data types

The solution was then to use template parameters to specify iteration and **TM type**. The client code could *build* as many different frameworks as necessary, with different iteration and move types, as long as they implemented a given *compile time* interface (e. g. it must be possible to compare **TMs** and to store them in containers).

Policies

Policies are a way to implement template method pattern using template parameters in C++, proposed by Alexandrescu [2001]. A template parameter is expected to define a member function to implement a specialized algorithm, called in a template class member function. A client can then write different classes that implement different algorithms and feed them to the template class, obtaining many similar complete algorithms.

Client specializes
default algorithm
using template
policies

The framework uses template policies for various operations:

- Exploring the neighbourhood at each iteration.
- Managing **TL** size. A smart policy checks if the current solution is the same as the last minimal solution, i. e., if during the exploration we came back to an old solution. Chances are that this is a deep local minimum, so deep that **TL** is not long enough to prevent returning to it. In that case the **TL** size is doubled for few iterations.
- Storing progress at each iteration. It might be useful to store the solution value at each iteration for performance analysis.

In all cases default policies do nothing, but there exist other specialized policies (e. g. the various exploration policies shown in Figure A.2). Since template instantiation happens at compile time, the compiler has the chance to perform many micro-optimizations in order to minimize overhead. Anyway, the program spends the vast majority of the time *inside* exploration policy's function, so performance impact is unimportant at this level.

A.3 MULTI-THREADING

GRASP generation
and local search are
reentrant

Since each solution is created and improved independently of the other ones, the procedure is *reentrant*, therefore they can be executed in parallel using multi-threading. A thread pool is created with a

fixed number of threads, then each solution is created or improved as soon as there is a thread available. Since all the operations are **CPU** bound, this is only useful when running the program on a multi-processor or multi-core machine, in that case performance improvement is roughly linear over **CPU** number.

Solutions are independent, so even if created and improved in parallel – as long as the program’s parameters are the same – ought to be the same. However, some algorithms (namely **GRASP**) make use of random numbers. The random numbers generator itself is a thread-safe global object, the random numbers stream is therefore deterministic (it can of course be initialized with a random seed), but threads scheduling is indeterminate, and so is the order in which different threads will ask the generator for random numbers.

BIBLIOGRAPHY

- Alessandro Agnetis, Gabriella Dellino, Paolo Detti, Giacomo Innocenti, G. de Pascale, and Antonio Vicino. Appliance Operation Scheduling for Electricity Consumption Optimization. In *50th IEEE Conference on Decision and Control and European Control Conference*, pages 5899–5904, Orlando, Florida, 2011. ISBN 9781612847993. URL <http://www.nt.ntnu.no/users/skoge/prost/proceedings/cdc-ecc-2011/data/papers/0506.pdf>. (Cited on page 18.)
- R.K. Ahuja, Ö. Ergun, J.B. Orlin, and A.P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002. URL <http://www.sciencedirect.com/science/article/pii/S0166218X01003389>. (Cited on page 47.)
- Andrei Alexandrescu. *Modern C++ Design: Generic programming and design patterns applied*. Addison-Wesley, 2001. ISBN 0201704315. URL <http://erdani.com/index.php/books/modern-c-design/>. (Cited on page 106.)
- A. Barbato, A. Capone, G. Carello, M. Delfanti, M. Merlo, and A. Zaminga. Cooperative and Non-Cooperative house energy optimization in a Smart Grid perspective. In *2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–6. IEEE, June 2011a. ISBN 978-1-4577-0352-2. doi: 10.1109/WoWMoM.2011.5986478. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5986478>. (Cited on pages 7, 19, and 21.)
- A. Barbato, A. Capone, G. Carello, M. Delfanti, M. Merlo, and A. Zaminga. House energy demand optimization in single and multi-user scenarios. In *2011 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 345–350. IEEE, October 2011b. ISBN 978-1-4577-1702-4. doi: 10.1109/SmartGridComm.2011.6102345. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6102345>. (Cited on pages 7, 10, 19, and 21.)
- Jason W. Black and Richard C. Larson. Strategies to overcome network congestion in infrastructure systems. *Journal of Industrial and Systems Engineering*, (November), 2007. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.3390&rep=rep1&type=pdf>. (Cited on page 3.)
- Giuseppe Carpentieri. MILP Optimization models for domestic load management. Unpublished, 2012. (Cited on page 66.)

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT press, 2nd edition, 2001. ISBN 0-262-03293-7. URL <http://mitpress.mit.edu/books/introduction-algorithms>. (Cited on page 34.)
- Thomas A. Feo and Mauricio G. C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6(2):109–133, March 1995. ISSN 0925-5001. doi: 10.1007/BF01096763. URL <http://www.springerlink.com/index/10.1007/BF01096763>. (Cited on page 29.)
- Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical Programming*, 98(1-3):23–47, September 2003. ISSN 0025-5610. doi: 10.1007/s10107-003-0395-5. URL <http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s10107-003-0395-5>. (Cited on pages 60 and 61.)
- Fred Glover and Manuel Laguna. *Tabu search*. Springer, 1998. ISBN 9780792381877. URL <http://www.amazon.com/exec/obidos/ASIN/079239965X/ref=nosim/weisstein-20>. (Cited on pages 46 and 47.)
- Long Ha, Stephane Ploix, Eric Zamai, and Mireille Jacomino. Tabu search for the optimization of household energy consumption. In *2006 IEEE International Conference on Information Reuse & Integration*, pages 86–92. IEEE, September 2006. ISBN 0-7803-9788-6. doi: 10.1109/IRI.2006.252393. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4018470>. (Cited on page 18.)
- Alain Hertz, Eric Taillard, and Dominique De Werra. A tutorial on tabu search. In *Proc. of Giornate di Lavoro AIRO*, volume 95, pages 13–24, 1995. URL <http://red.cs.nott.ac.uk/~ajp/courses/g5baim/files/IntroTS.pdf>. (Cited on page 46.)
- Björn Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley Professional, 2005. ISBN 978-0-321-13354-0. URL <http://www.informit.com/store/beyond-the-c-plus-plus-standard-library-an-introduction-9780321133540>. (Cited on page 95.)
- Shaline Kishore and Lawrence V. Snyder. Control Mechanisms for Residential Electricity Demand in SmartGrids. In *2010 First IEEE International Conference on Smart Grid Communications*, pages 443–448. IEEE, October 2010. ISBN 978-1-4244-6510-1. doi: 10.1109/SMARTGRID.2010.5622084. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5622084>. (Cited on page 18.)
- Nathan Kowahl and Anthony Kuh. Micro-scale smart grid optimization. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, July 2010. ISBN 978-1-4244-6916-

1. doi: 10.1109/IJCNN.2010.5596726. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5596726>. (Cited on page 18.)
- D. Livengood and R. Larson. The Energy Box: Locally Automated Optimal Control of Residential Electricity Usage. *Service Science*, 1(1):1–16, March 2009. ISSN 2164-3962. doi: 10.1287/serv.1.1.1. URL <http://servsci.journal.informs.org/cgi/doi/10.1287/serv.1.1.1>. (Cited on pages 3, 4, 17, and 18.)
- Silvano Martello, Mauro Dell’amico, and Rainer Burkard. *Assignment problems*. SIAM: Society for Industrial and Applied Mathematics, 2009. ISBN 978-0898716634. URL <http://www.assignmentproblems.com/>. (Cited on page 20.)
- Amir-Hamed Mohsenian-Rad and Alberto Leon-Garcia. Optimal Residential Load Control With Price Prediction in Real-Time Electricity Pricing Environments. *IEEE Transactions on Smart Grid*, 1(2):120–133, September 2010. ISSN 1949-3053. doi: 10.1109/TSG.2010.2055903. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5540263>. (Cited on pages 3 and 18.)
- D Picault, B Raison, S Bacha, J. de la Casa, and J Aguilera. Forecasting photovoltaic array power production subject to mismatch losses. *Solar Energy*, 84(7):1301–1309, July 2010. ISSN 0038092X. doi: 10.1016/j.solener.2010.04.009. URL <http://www.sciencedirect.com/science/article/pii/S0038092X10001556>. (Cited on page 11.)

NOMENCLATURE

ACTIVITY A pair appliance, house, i. e. an electric device belonging to a particular house.

APPLIANCE Any kind of domestic electric device, e. g. an oven, a fridge, lightings or a boiler.

EXECUTION WINDOW The interval in which an activity can be scheduled. Lowly flexible activities, e. g., the fridge or the heating system, have short execution windows. Highly flexible activities have instead long execution windows.

LOAD PROFILE The energy consumed by an appliance for each time slots since it is turned on.

PHASE A phase is a time interval of the same size of a time slot. Activities load profiles give the amount of energy consumed in each phase.

SMOOTH DEMAND CURVE A smooth demand curve is a curve with low height, no isolated peaks, little variation from one peak to the nearby ones. This means that the energy consumption is very steady over time. It has no relation with the usual meaning in mathematics, i. e., a function belonging to C^∞ .

TEMPLATE Templates are a feature of the C++ programming language that allow functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

TIME SLOT A day is discretized in a finite number of time slots.

