Local Search for integrated Economic Dispatch and Unit Commitment problems

J. F. Hageraats

February 12, 2023

Abstract

I will present a novel way to model the hybrid Unit Commitment (UC) and Economic Dispatch (ED) problem as a variation of an interior point problem, which allows the problem to be tackled by common local-search metaheuristics. On top of that, my state representation is highly intuitive, adaptable and can accept any cost function and many constraints with relative ease. I will also compare my findings with the pre-existing literature and I will show that this method can find an improvement of a known optimum for at least one well known instance of the hybrid UC/ED problem. I will also show that this method exhibits interesting search behaviour which can be preferable in networks with renewable energy sources and various other robustness concerns.

Contents

1 Introduction

In today's world having a reliable supply of electricity has become a vital part of modern life, because large parts of society will quickly grind to a halt without it. We are facing a reality in which broadly applicable methods of energy storage are not yet available. Currently the generation of electricity is one of the main environmental polluters, most renewable energy producers are unpredictable, energy distribution networks place significant technical constraints on the routing of electrical power, and economic factors cannot be ignored. So it becomes vitally important to optimize the production scheduling of the generation of electrical power. The unit commitment (UC) and economic dispatch (ED) problems are concerned with finding methods to optimize the aforementioned production schedules.

As of yet, creating these schedules is hard and no general or easily adaptable methods exist for computing them. The creation of said schedules is further complicated by the fact that energy networks have different topologies, economic regimes and technical properties. These are the reasons that most methods for optimizing the UC problem for one network, are usually not easily portable to different networks, or to different network or generator operators.

Most solutions in the currently available literature focus on approaches based on variations of linear programming algorithms. This means that the instances of the problem, which often contain non-lineair objective functions, have to be approximated by a linear approximation of their objective functions. Methods based on local search do not have this linearity requirement, because such methods can be used to optimize any cost function. This flexibility of the local search method comes with the downside that it requires careful modeling of the problem under consideration. It is in this modeling phase where most previous attempts have been generally unsuccessful or only have succeeded for certain specific instances of the unit commitment problem.

Overall we think that the use of local search methods with regard to the Unit Commitment problem for electrical power generation, has not yet received the amount of scientific scrutiny it deserves.

Therefore, the purpose of this master thesis is to explore the feasibility of local-search methods for the unit-commitment problem with integrated dispatch. To this end, we show that we can cast the problem as an instance of a local search problem which bears some similarities to an interior point problem, which then has to be optimized through common meta-heuristics.

2 Overview of the Unit Commitment and Economic Dispatch problems

The unit commitment problem is typically concerned with finding a production schedule for a given power plant or energy distribution network, such that the energy demands can be met, and the various constraints imposed by the technical and operational limitations of the generators and the distribution network are respected. Typically this means that we have to compute a schedule for production, dispatch and optionally for transmission as well.

There are two common ways to look at these problems. The first is a classical way in which the problem is subdivided into a part in which the commitment status of the generators is computed separately from the dispatch of the generators. In the other (hybrid) approach, both the dispatch and the commitment schedules are calculated simultaneously.

While in this document we will only be considering the integrated approach which combines the calculation of the commitment and dispatch schedules into one algorithm, it is still required to have a basic understanding of the isolated forms of both problems.

2.1 Classical approach to Unit Commitment

The classical approach to compute a schedule for the unit commitment problem is only concerned with computing whether or not a specific generator or unit[1] (g_i) is on or off at a certain point (t_j) in time. This is usually represented as a binary vector in which a 0 means that g_i is off at t_j and a 1 means that g_i is on at t_j.

This in turn means that a commitment (on/off) schedule that provides a solution to the classical version of the Unit Commitment problem with n generators with a length of m time steps, is a binary $n \times m$ matrix in which a 0 at position (i, j) means that generator i is off at time j and that a 1 at position (i, j) means that generator i is on at time j.

Table 1: Example classical Unit Commitment schedule for GA10 instance (part 1)

Time	1	2	3	4	5	6	7	8	9	10	11	12
Unit 1	1	1	1	1	1	1	1	1	1	1	1	0
Unit 2	1	1	1	1	1	1	1	1	0	0	0	0
Unit 3	0	0	0	0	0	1	1	1	1	1	1	1
Unit 4	1	1	1	1	1	0	0	0	0	0	1	1
Unit 5	0	0	1	1	1	1	1	1	1	1	1	1
Unit 6	1	1	1	0	0	0	1	1	1	0	0	0
Unit 7	0	0	0	0	0	0	1	1	1	1	1	0
Unit 8	0	1	0	1	0	1	0	1	0	1	0	1
Unit 9	0	0	0	0	0	0	1	0	1	1	1	0
Unit 10	0	0	0	0	0	1	1	0	1	1	0	1

Table 2: Example classical Unit Commitment schedule for GA10 instance (part 2)

Time	13	14	15	16	17	18	19	20	21	22	23	24
Unit 1	0	0	0	0	0	0	0	1	1	1	1	1
Unit 2	0	0	0	0	1	1	1	1	1	1	1	1
Unit 3	0	0	0	0	0	1	1	1	1	1	1	1
Unit 4	1	1	1	0	0	0	0	0	1	1	1	1
Unit 5	1	1	0	0	0	0	0	0	0	0	0	0
Unit 6	1	1	1	1	1	1	0	0	0	0	0	0
Unit 7	0	0	1	1	1	0	0	0	1	1	1	0
Unit 8	1	1	0	0	1	1	1	1	1	0	0	0
Unit 9	0	0	0	1	0	1	0	1	1	1	1	0
Unit 10	0	0	1	1	1	0	1	1	1	0	1	0

The demands are usually given in the form of an integer- or real-valued vector of length m in which the value at position j denotes the demand of energy required at time t_j.

However, the classical Unit Commitment problem does not concern itself with assigning concrete levels of power output to each generator at each timestep, but it does concern itself with generating an on/off schedule in which energy demands are met while the constraints implied by all the generation and distribution equipment are respected.

This usually does simplify the study of the problem, but it could also be seen as an obstacle to finding good or better solutions, because some constraints which must be satisfied in a final solution are hidden from the algorithm if one only considers this formulation and representation.

However, for certain classes of algorithms, like genetic algorithms in Kazarlis [7] for example, this representation can be very useful.

[1] Note that the terms unit and generator are used interchangeably throughout this document, as is often seen in other literature on this problem.

2.2 Economic dispatch problem

Once a commitment schedule has been given or found, the economic dispatch problem has to be solved for the given commitment schedule if we want to obtain a feasible production schedule. The economic dispatch problem is concerned with minimizing the cost of said energy production production, while meeting the energy demands specified as stated in equation 1.

$$\text{minimize} \sum_{i=1}^{m} \sum_{j=1}^{n} C_i(P_{i,j}) + \text{SUC}(P_{i,j}) + \text{SDC}(P_{i,j}) \tag{1}$$

Subject to:

$$\sum_{i=1}^{m} P_{i,j} = D_j \qquad \forall j \in 1..n \tag{2}$$

Where C_i is the cost per unit of power of running generator g_i, $\text{SUC}(P_{i,j})$ is a function which defines the startup cost of generator g_i at time t_j, $\text{SDC}(P_{i,j})$ is a function which defines the shutdown cost of generator g_i at time t_j, $P_{i,j}$ is the power output of g_i at time $_j$, and D_j is the energy demand at time t_j.

2.3 Transmission

Sometimes we have instances of the UC-problem in which we also need to consider the network topology and/or have to compute a transmission schedule for the network. The transmission schedule dictates how the generated power should flow through the network of a network operator. This can again be formulated as a matrix in which at position (k, j) we denote the state of a switch k (s_k) in the network at time t (t_j).

We also have to make sure that our production schedule meets the energy demands which are specified beforehand. In the case where we do not have to consider the network topology, the demands can be modeled as a vector in which $d(t_j)$ denotes the total amount of power demanded at t_j. If we do have to consider the network topology as well, a simple vector does not suffice anymore. In this case our demands will be given in the form of a matrix in which each (k, j) position denotes the power demands $d(n_k, t_j)$ at network node $n_k \in 1..l$, at time t_j.

An example can be seen in equation 3.

$$\text{Demands} = \begin{bmatrix} d(n_1,t_1) & d(n_1,t_2) & \cdots & d(n_1,t_n) \\ d(n_2,t_1) & d(n_2,t_2) & \cdots & d(n_2,t_n) \\ \vdots & \vdots & \ddots & \vdots \\ d(n_k,t_1) & d(n_k,t_2) & \cdots & d(n_l,t_n) \end{bmatrix} \tag{3}$$

I have decided to leave experiments involving network topologies out of scope because of the very instance-specific implementations required and the increased workload this would cause. However, it's important to note that the way the problem is formulated does allow for the integration of network constraints with relative ease.

2.4 Integrated problem

Sometimes it's useful to combine the unit commitment and economic dispatch problem into a variant of the problem in which the on/off schedule and the power dispatch schedule are calculated simultaneously. This makes sense, because we are often not just interested in obtaining an on/off schedule, but we also need a detailed production schedule as well. Yet another reason to integrate both problems, is that local-search based approaches often require complicated additional procedures to ensure that the on/off schedule actually leads to a feasible production schedule.

The first thing we have to do to solve the UC-problem with an integrated approach, is to define a representation of a production schedule. In a production schedule, we need to know how much energy has to be produced by each generator g at time each timestep t. This means that a production schedule takes the form of a 2-dimensional matrix in which at position (i, j) we denote the outcome of $p(g_i, t_j)$, which is the amount of power that generator i (g_i) needs to produce at time j (t_j). An example can be seen in equation 4.

$$\text{Schedule} = \begin{bmatrix} p(g_1, t_1) & p(g_1, t_2) & \cdots & p(g_1, t_n) \\ p(g_2, t_1) & p(g_2, t_2) & \cdots & p(g_2, t_n) \\ \vdots & \vdots & \ddots & \vdots \\ p(g_m, t_1) & p(g_m, t_2) & \cdots & p(g_m, t_n) \end{bmatrix} \tag{4}$$

Note that, once a production schedule has been found, the corresponding on/off schedule can easily be found as well, by defining that whenever $p(g_i, t_j) > 0$ then g_i is on at t_j and that when $p(g_i, t_j) = 0$ the generator g_i is off at t_j. This on/off schedule can then serve as a solution to the Unit Commitment problem.

2.5 Objectives

If we set out to calculate a suitable schedule for the unit commitment problem, we typically have one or more objectives we wish to accomplish. We wish to accomplish those objectives by finding a schedule which respects all the operational, technical and demand constraints imposed by the equipment which makes up the generation and distribution-network.

The objectives we want to optimize are typically one or more of the following:

- Maximize total profits.
- Minimize total operating costs.
- Minimize fuel costs.
- Minimize power shortages.
- Minimize environmental pollution.

2.6 Problem constraints

Some of the technical constraints we have to consider for generation are:

- Minimum uptime: The minimum time a generator has to be turned on before it can be turned off again.
- Minimum downtime: The minimum amount of time a generator has to be off before it can be turned on again.
- The maximum amount of power a generator can output (P_{max}).
- The minimum amount of power a generator can output without having to shut down (P_{min}).
- The maximum amount of power a generator can output when it is turned on ($P_{startup}$).
- The maximum amount of power a generator can output right before it can be turned off ($P_{shutdown}$).
- Ramp up limit: The maximum amount of power by which the output of a generator can be increased between two subsequent time steps.
- Ramp down limit: The maximum amount of power by which the output of a generator can be decreased between two subsequent timesteps.
- Demand constraints which place a requirement on the schedule that at least a certain amount of power has to be available at a certain network node at time t_j.

This list of constraints can grow if additional requirements need to be modeled as well. Examples of potential extra constraints are the capacity constraints imposed by the cables when we need to consider the routing of power through the network. Renewables and energy storage systems, like batteries or pumped-hydroelectrical storage systems can complicate things even further and will require the introduction of additional constraints.

3 Related work

At least three literature surveys have been done on the topic of unit commitment by Padhy[13], Savaran[15] and Ackooij[19].

Padhy[13] provides an overview for the standard UC-problem. In it, we can find a basic formulation of the objectives, which is to either minimize operational cost or to maximize profits, and the various constraints at play in standard UC problems. Padhy[13] also notes that while UC for a single area (without a transmission network) has been extensively studied, the variants dealing with multiple areas (nodes) have not received enough attention yet. Padhy[13] does also note that latter variants of UC have to be considered combined with the viable economic dispatch problem, because of the constraints the transmission lines induce on the generated schedule. Furthermore, Padhy[13] lists an overview of studies done using algorithms based on exhaustive enumeration, priority listing, dynamic programming, (integer) linear programming, branch and bound, Lagrangian relaxation, interior point optimization, tabu-search, simulated annealing, expert systems, fuzzy systems, artificial neural networks, genetic algorithms, evolutionairy programming, ant colony search algorithms and hybrid models.

Padhy[13] notes that all listed techniques have their own drawbacks and advantages. Exhaustive enumeration is capable of finding optimal solutions, but it is not suitable for larger instances of the UC-problem.

Priority listing is a simple method which orders all generators based on lowest operational cost characteristics and then turns them on, based on lowest operational cost or some other criterion that leads to a certain ordering on the priority-list. While this might work for certain generator operators, it doesn't necessarily work for others.

Methods based on dynamic programming are capable of solving UC-problems of a variety of sizes, but treat time dependent startup costs, minimum uptime and minimum downtime constraints in a suboptimal way.

Methods based on branch and bound, (integer) lineair programming and Lagrangian relaxation, generally perform well, but are mathematically heavy and often require complex solvers. They also require that the cost-functions, which are often quadratic, are piecewise linearly approximated, implying that the solutions found are inherently suboptimal. However, in practice, these approximations are quite reasonable.

Methods based on simulated annealing are generally less mathematically complex and can be suitable for solving for UC, but generally require more CPU-time.

Savaran[15] adds a series of objectives which can be used if we have to solve UC-problems in different economic environments. Examples of these environments are stochastic environments, profit-, time- and emissions-based economic environments. However, most of the studies listed in their survey are not based on local search methods and rely on other methods like, for example, (integer) lineair programming, instead.

Ackooij[19] has provided the biggest literature survey and mostly focuses on UC under uncertainty (UUC). Uncertainty here can generally refer to two types of uncertainty. The first type are faults which can occur in the network. In these cases, objectives like minimization of non-delivered energy and maximization of network robustness can be optimized. The second type of uncertainty pertains to the uncertainty imposed by the demand forecasts and/or the production figures of renewable energy sources like solar, wind and run-of-river units. Ackooij[19] also discusses the aspects of energy storage systems (like pumping hydros for example) and transmission switching. However, the number of papers listed by Ackooij[19] which are concerned with local search approaches, is very small and only one of these listed papers ([18]) takes the ramping constraints into account. Their method consists of a simulated annealing algorithm which relies on changing random states of randomly chosen generators for its candidate generation.

Regarding algorithms based on local search:

Zhuang and Galiana[22] also use a simulated annealing algorithm to solve UC. They partition the set of constraints into "easy" and "difficult" constraints, where difficult constraints are penalized and easy constraints are not. What is remarkable about their algorithm, is that they model the generation limits, minimum up- and downtimes as "easy" constraints, while they model power balance and reserve and crew constraints as difficult constraints which are penalized. Crew constraints are constraints concerning the staff that needs to operate the power plant, while the reserve constraints are constraints with relation to

the amount of spinning reserve that has to be available. Their neighbourhood is calculated by changing the commitment of a random generator g_i at a random point in time t_j. They do have some cost improvements, but their simulated annealing method also requires a considerable amount of additional iterations and running time.

Matanwy et al[9] also randomly select a generator g_i at time t_j and detail an elaborate strategy to turn g_i off at t_j if it is on, and on if it is off, while taking great care to ensure that all constraints are still satisfied. Their approach is quite different from our current approach, because they generate a random unit at a random instant and then compute whether or not all constraints in the problem are still valid. Our proposed approach does not generate units randomly and stores as many constraints as possible into a data-structure in such a way that checking the validity of constraints becomes a fast operation. Yet Matanwy et al's[9] approach bears the most similarities with our proposed approach.

In a later paper Matanwy et al.[10] decribe a simulated annealing algorithm which specifically takes the long term hydro scheduling into account. This problem specifically focuses on a system with four hydro-systems that spill their water into each other's reservoir in a hierarchical way, such that the production of a generator upstream also directly influences the production of a unit downstream.

Mori[11] proposes a method based on tabu-search. The idea is that the number of neighbouring solution candidates is narrowed down by integrating the restrictions posed by the priority list of units into the tabu-search. They do this in such a way that, for a given load-curve, only the generation units that are direct neighbours of the given load-curve on the priority list, are considered in the tabu-search.

Lin et al[8] provide a hybrid algorithm based on tabu-search with flexible memory and evolutionary programming to solve just the economic dispatch problem in a power distribution network.

Rajan[4] describes a neural based tabu-search algorithm for solving unit commitment problems in which a neural network is used to generate trial solutions for the tabu-search. However, this algorithm requires an extensive repair mechanism to ensure that all constraints are met.

Annakage et al.[1] demonstrate that the running time of simulated annealing algorithms can be somewhat lowered by using a variant of the simulated annealing algorithm that uses speculative execution techniques to evaluate trial solutions in parallel.

Borghetti et al.[3] compare tabu-search and Lagrangian heuristics on a few realistic instances of the optimal short-term unit commitment problem. They highlight the strong and weak points of each technique. They find that both Lagrangian relaxation and tabu-search both usually find solutions which differ less than 1% of their total costs. They notice that Lagrangian relaxation is more suitable for handling instances with a larger number of generators, while for smaller instances, a straightforward implementation of tabu-search is able to find equally good results in less than a minute. They argue that further research into a method that integrates both algorithms is therefore justified.

Victore and Jeyakumar [20] demonstrate a hybrid tabu-search algorithm which can be used for both cost based (CBUC) and profit based (PBUC) instances of the unit commitment problem. In the CBUC variant of the problem the focus is solely on reducing the costs of generation, while in the PBUC problem, they also take economic factors, like the price at which the power can be sold, into consideration. They use a particle swarm optimization method to keep a population of feasible solutions and it also guides the selection of trial candidates (neighbours) that the tabu-search will evaluate first.

Simmopoulos et al.[18] present an algorithm based on tabu-search which generates trial solutions by a procedure which uses three different mechanisms to create pertubations of the current schedule in an alternating way. The first mechanism changes the states of a randomly chosen generator over a given time interval. The second mechanism changes the states of all generators for a randomly chosen hour. The third mechanism uses a complicated series of steps to ensure that feasible solutions are always obtained.

In a different paper by Simmopoulos et al.[17] an algorithm based on simulated annealing is presented. The algorithm deals with the reliability constraints by penalizing solutions in which the reliability constraints are not met.

Kazarlis et al.[7] present a genetic algorithm to solve the unit commitment problem. They also provide a detailed description of a relatively small (10 units, 24 hours) but realistic instance of the unit commitment problem, along with the costs of the best solutions they have found. This instance is used in some of the papers mentioned above as a benchmark for their algorithms. It is therefore quite useful as a starting point.

Kun-Yuan Huang et al.[5] provide an approach based on constraint logic programming to solve the unit commitment problem. Although this is also not directly relevant for a local search based solution,

this paper provides a detailed instance of a unit commitment problem that is slightly larger (38 units and 24 hours) than the instance provided by Kazarlis et al.[7].

Other papers which are less relevant for local search, but nevertheless interesting because they contain instances and/or benchmarking data for which instances are available at the time of writing are: Silbernagl et al.[16], Park et al. [14], Frangioni et al.[2] and Orero and Irving[12].

Finally there is the elaborate Dispa-SET study from the EU Joint Research Centre[6] which provides some complicated real-world instances and much more.

Park et al. [14] and Frangioni et al.[2] solve a 24-hour UC problem with 140 and 200 generators respectively with relative success. Of these two, only Park et al. [14] provide us with the score of the best solution they have found and therefore contains the largest "solved" solution against which we can benchmark. The other instances (with the exception of Dispa-SET[6]) all contain problems of smaller sizes, with various combinations of constraints. The Dispa-SET instances are much larger, but also much harder to compare.

An overview of the different instances and their sizes is included in Table 3. Machine readable definitions in .uc format of the instances in table 3 were kindly provided by Rogier Hans Wuijts who is the author of the .uc files and has used a number of these same instances to study dynamic programming algorithms for the Single-Unit Commitment problem with ramping limits in [21]. An overview of the different properties of these machine readable instances is shown in table 3.

Dataset	Generators	Timesteps	State variables	Additional Properties
Kazarlis et. al. [7]	10	24	240	Global optimum: 565825
Huang et. al. [5]	38	24	912	
Orero and Irving [12]	110	24	2640	
Park et. al. [14]	140	24	3360	
Frangioni et. al. [2]	200	24	4800	
Silbernagl et. al. [16]	54	576	31104	transmission
DispaSET [6]	304	8760	2663040	storage, transmission
DispaSET2 [6]	1442	8783	12665086	storage, transmission

Table 3: Number of generators, timesteps, generator states and non-regular constraints present in each instance.

Instance	GA10 [7]	TAI38 [5]	RTS54 [16]	A110 [12]	KOR140 [14]	RCUC200 [2]	DISPASET [6]	DISPASET2 [6]
Units	10	38	54	110	140	200	304	1442
Timesteps	24	24	576	24	24	24	8760	8783
Cost function	Quadratic	Quadratic	Quadratic	Quadratic	Quadratic	Quadratic	Quadratic	Quadratic
Transmission	No	No	Yes	No	No	No	Yes	Yes
Initial State	Yes	No	No	No	No	No	No	No
Pmin	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Pmax	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Ramp Up	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Ramp Down	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Time Dependent Startup Costs	Yes	Yes	No	No	Yes	No	Yes	Yes
Shutdown Limit	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Minimum Uptime	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Minimum Downtime	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 4: An overview of the various properties, constraints and cost functions present in the instances provided by Rogier Hans Wuijts.

4 Local search

Local search is a heuristic which is often used to solve computationally hard problems. It is an approach which takes an existing solution for a problem and then repeatedly modifies it in such a way that a new solution is generated, with the aim of discovering better solutions than the one it has currently found.

For the Unit Commitment problem, these changes can either be small changes, like ramping a unit which increases or decreases its power output at a certain point in time, or big changes, like turning a unit on or off or moving a sequence of loads from one unit to another.

However, before local search as a method can be used, we need the following prerequisites:

- A formally defined representation of a solution. (Section 5)
- A formally defined state-representation for the algorithm. (Section 9)
- An objective function to be optimized. (Section 8)
- A search strategy.
- Procedures to generate neighbouring solutions, which can be used to formulate a Neighbourhood description. (Section 11)
- A suitable set of constraints to model the feasibility of the problem. (Section 6)
- Suitable criteria to stop the search (Stopping criteria).
- A suitable set of test instances to test our local search on.
- A set of initial solutions or a method of generating initial solutions. (Section 13)
- A set of criteria which define whether or not a neighbouring solution is accepted.

4.1 The local search procedure

The general process of local search works along these lines:

1. Generate an initial solution.
2. Evaluate the stopping criterion.
3. If the stopping criterion is satisfied, terminate the search. Otherwise continue.
4. Generate the next neighbouring solution.
5. Test if the neighbour is invalid, reject the neighbour and go to step 2. Otherwise continue to step 6.
6. Evaluate acceptance criteria. If the neighbour is rejected, go to step 2. Otherwise continue to step 7.
7. Accept neighbour.
8. Update algorithm state.
9. Recalculate or update cost function.
10. Go to step 2.

This process is illustrated in a flowchart in figure 1.

4.2 Why local search?

We now have a basic understanding of what local search as a meta-heuristic entails. But what are the differences, advantages and potential downsides of using of local search instead of (integer) linear programming? In this section I will touch briefly on these matters.

The main difference between local search and lineair programming is the search procedure each employs. Local search is a very basic procedure which in itself is not very complex and if the problem can be modeled, it places very few restrictions on how the problem can be modeled. However, the success of local search is heavily dependent on the finer details of how the problem is modeled.

Lineair programming however, typically uses some variation of the simplex method. The simplex method puts more restrictions on how the problem can be modeled because it exploits very specific properties of higher dimensional shapes with convex geometries. This also means that for (I)LP-like

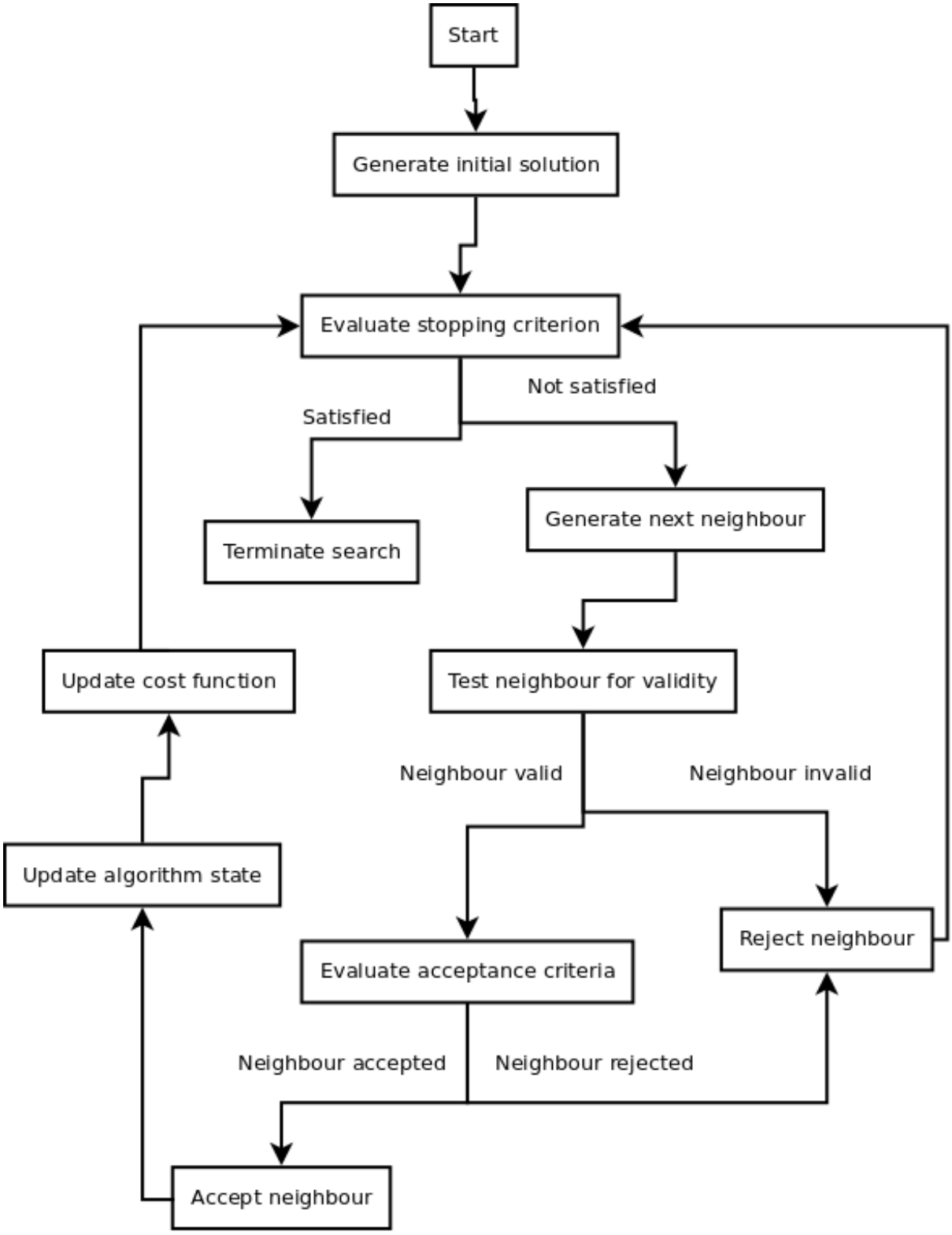


Figure 1: Flow chart of a typical local search

methods to work, the shape of the space enclosed by the higher dimensional convex shape, must be guaranteed to be convex.

To ensure that the space enclosed by all valid solutions is convex, the objective function of the unit commitment problem and all of its constraints are modeled as lineair higher dimensional surfaces which divide the higher-dimensional space into two parts.

The simplex method exploits the convex shape of the solution space by following the boundary of the enclosing space by moving along the lines, surfaces and points along which the various higher dimensional planes intersect.

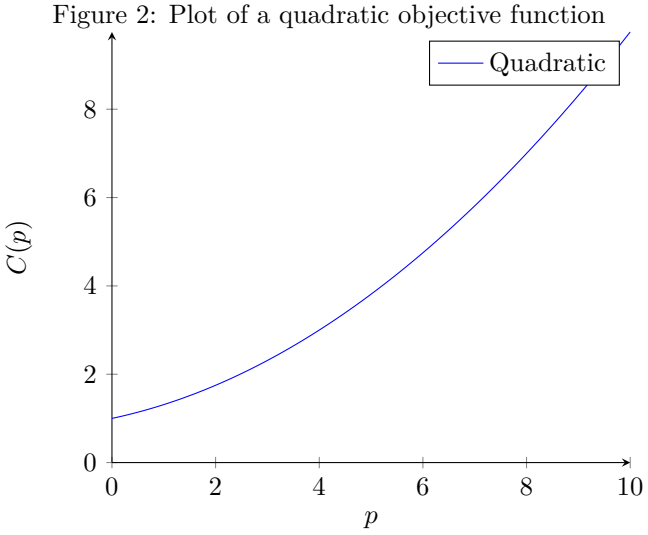
If the shape isn't convex, or if the given constraints and objective function cannot be modeled as higher dimensional linear planes, then the simplex method simply won't work as intended.

This is a problem, because most instances of the hybrid unit commitment problem, contain units which have quadratic cost functions.

The traditional way to deal with this, is to use linear approximations to model the quadratic cost functions. Usually this is fine because the margin of error is relatively small, but it is still important to note that the local search method is not bound by such restrictions on the cost functions or constraints, and therefore does not require the problem to be approximated, nor does it require the operator to define these approximations.

To illustrate this difference, an example of a quadratic cost function can be seen in figure 2. An example of a lineair approximation of a cost function that an (I)LP could use can be seen in figure 3 and in figure 4 you can see an overlay of how both relate to each other.

Figure 2: Plot of a quadratic objective function



Other aspects in which methods based on linear programming and local search differ, are in the way the problems can be specified. In a local search, the programmer has to model the entire problem by himself and then write that model into computer code, while linear programs only require a mathematical description of the problem which can often be given to a solver in terms of a matrix or a number of equations. This means that there are more general purpose software packages available for solving linear problems and they are more widely applicable, while a program for local search on the unit commitment problem, is usually only suitable to solve the unit commitment problem and cannot easily be transformed to solve other problems.

4.3 Types of local search

There are a number of strategies which can be employed by a local search algorithm. In this section I will cover four of the most prominent ones. These strategies are usually used to modify the way a local search algorithm deals with situations in which it gets stuck in local optima and how it can escape from a local optimum. Usually this involves either terminating the search or finding some way to jump over barriers in the landscape of the objective function.

Figure 3: Plot of its piecewise linear approximations

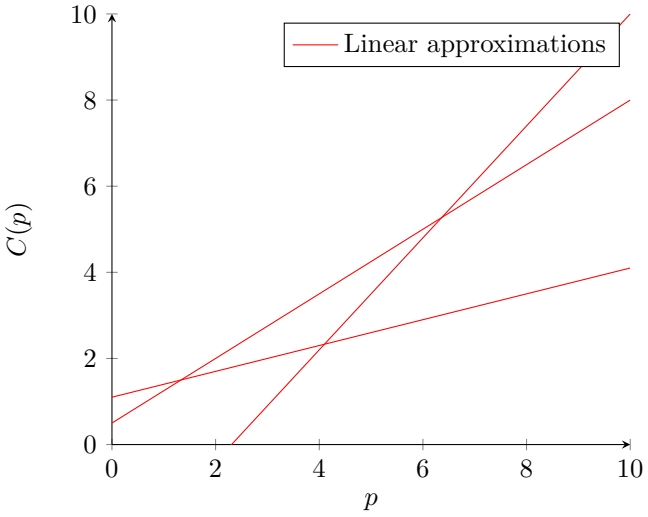
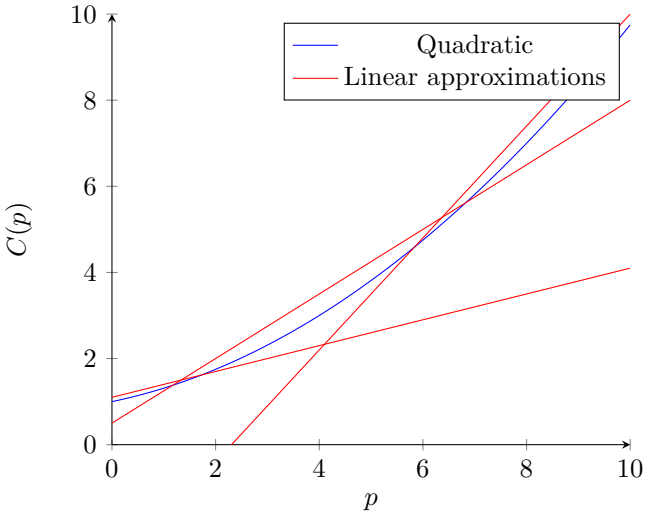


Figure 4: Plot of both a quadratic objective function and its piecewise linear approximation



A visual example of the landscape of the objective function, containing one local optimum, one gobal optimum and one barrier can be seen in figure 5.

4.3.1 Random walk

A random walk is a type of local search in which the acceptance criteria simply accept every generated neighbour which satisfies all constraints. Usually the best solution the random walk has seen, is kept in memory, while the search is allowed to walk randomly throughout all possible valid solutions of the problem. This type of search typically does not get stuck in local optima, but will often have trouble finding a good outcome within a reasonable amount of allotted time.

4.3.2 Greedy hillclimber

A local search which uses the greedy hillclimber metaheuristic is a type of local search in which the acceptance criteria accepts every neighbour which leads to a better outcome according to the cost function used to model a problem, and rejects every neighbour which leads to a solution with an outcome of the objective functions that is worse than the best solution which has been found so far.

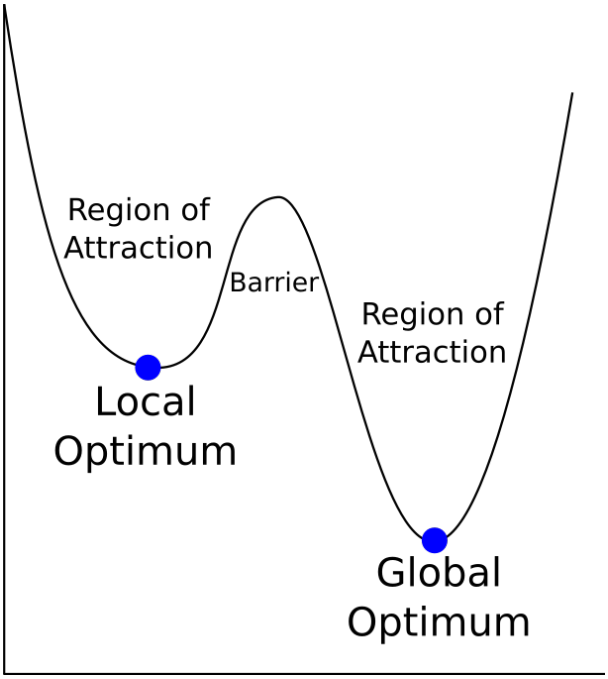


Figure 5: A visual example of a local optimum, a global optimum, a barrier and regions of attraction.

4.3.3 Best in neighbourhood

A local search which uses the 'best in neighbourhood' metaheuristic evaluates all neighbouring solutions which can be reached from the best solution it has found so far and then finally selects the best solution out of all neighbouring solutions it has evaluated. This method has the downside that it cannot be used in a procedure which does not have a enumerable and finite neighbourhood. This in turn means that it cannot be used in procedures which generate a neighbour randomly. It is also guaranteed that when given the same initial solution and initial parameters, it will always follow the same search path and terminate on the same final solution.

4.3.4 Simulated Annealing

A local search using the simulated annealing metaheuristic, is a local search which will generate neighbouring solutions randomly. These generated solutions can be better or worse than the solution the local search has currently discovered. If the outcome of the objective function for the neighbour which is under consideration is better than the best solution found so far, the simulated annealing heuristic will always accept the better solution. However, if the value of the objective function of the neighbour under consideration is worse than that of the current solution, simulated annealing will accept this neighbour with a certain probability. The probability that a worse solution is accepted depends on the difference in the result of its objective function compared to the result of the objective function of the current solution and the number of iterations which has gone by. As this process was inspired by the process of annealing in metallurgy, it uses the notion of a cooling schedule to gradually make the acceptance of a worse neighbour less likely. The main idea behind this is that in the beginning of the search, we can quickly get trapped in local optima which could be surrounded by barriers, which make it hard to escape from the local optimum if a worse neighbour can never be accepted. It should therefore be regarded as a technique to escape from a local optimum so we can prevent the search from stalling prematurely, which slowly becomes more restrictive in the magnitude of changes it allows as the search progresses.

The widely known equations 5 and 6 typically govern this process.

$$\text{temperature}(x, y) = 1 - \frac{x+1}{y} \tag{5}$$

Where x is the number of the current iteration in the search process and y is the maximum number of iterations after which the search process will terminate.

$$\text{probability}(r, s, t) = \exp(\frac{r - s}{t}) \tag{6}$$

Where r is the value of the objective function for the current solution, s is the value found by the objective function for the neighbouring solution and t is the value returned by the temperature function.

To determine whether or not a worse solution is accepted, we simply generate a random number q in the range $[0, 1]$ and if q is smaller than the calculated probability, we accept the current solution.

Note that the function presented in equation 6 is the conventional way of modeling the probability function of the simulated annealing heuristic. However, this definition as given in equation 6 is not usable for the unit commitment problem, because this definition makes it nearly impossible to overcome a barrier which has an objective function that is two times as high as the current value for r. However, in this problem the barriers of the objective function can be anywhere from about a factor of two to a hundred times as high as the current value of r. It's therefore not uncommon to have a very deep local optimum surrounded by relatively high barriers. To overcome this problem, I have updated the probability function to be more permissive of larger differences between r and s by modifying the probability equation as seen in equation 7.

$$\text{probability}(r, s, t) = \exp(\frac{r - s}{r \times t}) \tag{7}$$

5 Characterization of a solution

In section 2 we've seen a basic overview of the Unit Commitment and Economic Dispatch problems. We've also seen a characterization of what a schedule which qualifies as a solution looks like and what the differences between a solution to just the Unit Commitment, Economic Dispatch and the integrated problem are.

However, the problem under consideration is more complicated than the theoretic versions shown above in 2.4. In some of the instances under consideration, some of the units are modeled as a single unit, consisting of multiple identical subunits which each represent a generator with identical specifications in their own right. In this case we have i different units, which all have a fixed number of identical subunits. Let $s(i)$ denote the number of subunits corresponding to unit i. With these definitions, each pair of (g_i, s_k) for $k \in 1..k$ will now require a row in a matrix like the one shown in 4. This yields a matrix of the following form shown in equation 8.

$$\text{Schedule} = \begin{bmatrix} p(g_1, s(1)_1, t_1) & p(g_1, s(1)_1, t_2) & \cdots & p(g_1, s(1)_1, t_j) & \cdots & p(g_1, s(1)_1, t_n) \\ p(g_1, s(1)_2, t_1) & p(g_1, s(1)_2, t_2) & \cdots & p(g_1, s(1)_2, t_j) & \cdots & p(g_1, s(1)_2, t_n) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ p(g_1, s(1)_k, t_1) & p(g_1, s(1)_k, t_2) & \cdots & p(g_1, s(1)_k, t_j) & \cdots & p(g_1, s(1)_k, t_n) \\ p(g_2, s(2)_1, t_1) & p(g_2, s(2)_1, t_2) & \cdots & p(g_2, s(2)_1, t_j) & \cdots & p(g_2, s(2)_1, t_n) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ p(g_m, s(m)_k, t_1) & p(g_m, s(m)_k, t_2) & \cdots & p(g_m, s(m)_k, t_j) & \cdots & p(g_m, s(m)_k, t_n) \end{bmatrix} \tag{8}$$

We are searching for a schedule like this, in which every $p(i, s, j)$ value has been assigned a concrete value, such that all contraints imposed by the technical constraints of the units, the infrastructure and the forecasted demands at a given time are satisfied.

6 Modeling problem constraints

In this section I will describe the various constraints, what these constraints represent, how these can mathematically be modeled, and whether or not we have to treat these constraints as soft or hard constraints.

6.1 Maximum generation (P_{max})

The maximum power generation of generator g_i models the absolute maximum production capacity a generator can be run on at any given time. The production for this generator may never rise above this value. We will refer to this value as $P_{max}(g_i)$. This is a hard constraint, because it cannot be violated under any circumstances. Often this is the case because exceeding this limit would cause damage to real-world equipment.

This can trivially be modeled by a function which always returns P_{max} and puts an upper bound on the amount of power a generator can output.

The P_{max} constraint is violated if $p(i, s, j) > P_{max}$.

6.2 Minimum generation (P_{min})

The minimum power generation of generator g_i models the lowest production capacity a generator can be run on at any given time. If the production for this generator drops below this value, it has to be turned off. We will refer to this value as $P_{min}(g_i)$. This is a hard constraint, because the output of a generator can never fall below this threshold if it is running and therefore puts a lower bound on the output of a running generator.

However, because generators can also be turned off, we have to model this constraint as a function which returns $P_{min}(g_i)$ if a generator is on and 0 if a generator is turned off.

This gives us the formulation of equation 9 to calculate the lower bound imposed on the value of $p(i, s, j)$.

$$\text{calculateBound}(i, s, j) = \begin{cases} P_{min}(i) & \text{if unit } i \text{ subunit } s \text{ is generating at time } t \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Violation of the P_{min} constraint is tested with equation 10.

$$\text{isViolated}(i, s, j) = \begin{cases} p(i, s, j) < P_{min}(i) & \text{if unit } i \text{ subunit } s \text{ is generating at time } t \\ p(i, s, j) > 0 & \text{otherwise} \end{cases} \tag{10}$$

6.3 Ramp-up limit ($RU(i)$)

The ramp-up limit is the maximum amount of units of power by which the production of generator g_i can be increased at any given time t_j. We will refer to this value as $RU(i)$. This is a hard constraint which is imposed by technical restrictions of the generator.

This constraint puts both an upper and a lower bound on the value of $p(i, s, j)$. The ramp-up limit constraint limits the maximum value of $p(i, s, j)$ and depends on the value of $p(i, s, j - 1)$, which puts an upper bound on the value $p(i, s, j)$ can take on. Conversely, the ramp-up limit constraint also puts a lower bound on the value $p(i, s, j)$ can take on because of the value of $p(i, s, j + 1)$.

We should observe that because of this, the ramp-up limit is not just one constraint, but actually two: it has a forwards looking component and a backwards looking component. I have modeled these as two separate constraints.

The definitions I've used for the forwards looking ramp-up constraints can be seen in equations 11 and 12.

$$\text{calculateBound}(i, s, j) = p(i, s, j + 1) - RU(i) \tag{11}$$

$$\text{isViolated}(i, s, j) = p(i, s, j + 1) > p(i, s, j) + RU(i) \tag{12}$$

The definitions I've used for the backwards looking ramp-up constraints can be seen in equations 13 and 14.

$$\text{calculateBound}(i, s, j) = p(i, s, j - 1) + RU(i) \tag{13}$$

$$\text{isViolated}(i, s, j) = p(i, s, j) > p(i, s, j - 1) + RU(i) \tag{14}$$

6.4 Ramp-down limit ($RD(i)$)

The ramp-down limit is the maximum amount of units of power by which the production of generator g_i can be decreased at any given time t_j. We will refer to this value as $RD(i)$. This is a hard constraint which is imposed by technical restrictions of the generator.

This constraint also puts both an upper and a lower bound on the value of $p(i, s, j)$. The ramp-down limit constraint limits the minimum value of $p(i, s, j)$ and depends on the value of $p(i, s, j-1)$, which puts an lower bound on the value $p(i, s, j)$ can take on. Conversely, the ramp-down limit constraint also puts a upper bound on the value $p(i, s, j)$ can take on because of the value of $p(i, s, j+1)$.

We should observe that because of this, the ramp-down limit is also not just one constraint, but actually two: it also has a forwards looking component and a backwards looking component. I have again modeled these as two separate constraints.

The definitions I've used for the forwards looking ramp-down constraints can be seen in equations 15 and 16.

$$\text{calculateBound}(i, s, j) = p(i, s, j+1) + RD(i) \tag{15}$$

$$\text{isViolated}(i, s, j) = p(i, s, j) \geq p(i, s, j+1) + RD(i) \tag{16}$$

The definitions I've used for the backwards looking ramp-down constraints can be seen in equations 17 and 18.

$$\text{calculateBound}(i, s, j) = p(i, s, j-1) + RD(i) \tag{17}$$

$$\text{isViolated}(i, s, j) = p(i, s, j) < p(i, s, j-1) - RD(i) \tag{18}$$

6.5 Startup limit ($SU(i)$)

The startup limit is the maximum amount of power which generator g_i can produce at t_j if it was off at t_{j-1}. We will refer to this value as $SU(g_i)$. This is a hard constraint imposed by technical limitations of the generator. This constraint puts an upper bound on the value of $p(i, s, j)$.

The startup limit constraint limits the maximum value of $p(i, s, j)$ and depends on the value of $p(i, s, j-1)$, which puts an upper bound on the value $p(i, s, j)$ can take on.

The definitions I've used to model the startup limit constraints can be seen in equations 19 and 20.

$$\text{calculateBound}(i, s, j) = \begin{cases} SU(g_i) & \text{if unit } i \text{ subunit } s \text{ is not generating at time } t_{j-1} \\ +\infty & otherwise \end{cases} \tag{19}$$

$$\text{isViolated}(i, s, j) = \begin{cases} true & \text{if unit } i \text{ subunit } s \text{ is not generating at time } t_{j-1} \text{ and } p(i, s, j) > SU(g_i) \\ false & otherwise \end{cases} \tag{20}$$

6.6 Shutdown limit ($SD(i)$)

The shutdown limit is the maximum amount of power which generator g_i can produce at t_j if it is about to be off at t_{j+1}. We will refer to this value as $SD(g_i)$. This is a hard constraint imposed by technical limitations of the generator. This constraint puts an upper bound on the value of $p(i, s, j)$.

The shutdown limit constraint limits the maximum value of $p(i, s, j)$ and depends on the value of $p(i, s, j-1)$, which puts an upper bound on the value $p(i, s, j)$ can take on.

The definitions I've used to model the shutdown limit constraints can be seen in equations 21 and 22.

$$\text{calculateBound}(i, s, j) = \begin{cases} SD(g_i) & \text{if } p(i, s, j) > 0 \text{ and } p(i, s, j+1) \leq 0 \\ +\infty & otherwise \end{cases} \tag{21}$$

$$\text{isViolated}(i,s,j) = \begin{cases} true & p(i,s,j) > 0 \text{ and } p(i,s,j+1) \leq 0 \text{ and } p(i,s,j) > SD(g_i) \\ false & otherwise \end{cases} \tag{22}$$

6.7 Minimal uptime (minup(i))

The minimal uptime constraint models the minimum amount of time generator g_i has to be generating power before it can be shutdown. We will refer to this value as minup(g_i)). This is a hard constraint imposed by technical limitations of the generator. This constraint puts a lower limit on the value of $p(i,s,j)$.

The minimal uptime constraint limits the minimal value of $p(i,s,j)$ and depends on the value of $p(i,s,j-1)$ and uptime$(i,s,j-1)$, which puts a lower bound on the value $p(i,s,j)$ can take on.

The definitions I've used to model the minimal uptime constraints can be seen in equations 23 and 24.

$$\text{calculateBound}(i,s,j) = \begin{cases} P_{min}(g_i) & \text{if } p(i,s,j) > 0 \text{ and uptime}(i,s,j) \leq 0 \\ -\infty & otherwise \end{cases} \tag{23}$$

$$\text{isViolated}(i,s,j) = \begin{cases} true & \text{if uptime}(i,s,j-1) > 0 \text{ and uptime}(i,s,j-1) \leq \text{minup}(g_i) \text{ and uptime}(i,s,j) \leq 0 \\ false & otherwise \end{cases} \tag{24}$$

6.8 Minimal downtime (mindown(i))

The minimal downtime constraint models the minimum amount of time generator g_i has to be off before it can be switched on again. We will refer to this value as mindown(g_i)). This is a hard constraint imposed by technical limitations of the generator. This constraint puts an upper limit on the value of $p(i,s,j)$.

The minimal downtime constraint limits the maximal value of $p(i,s,j)$ and depends on the value of $p(i,s,j-1)$ and downtime$(i,s,j-1)$ and puts an upper bound on the value $p(i,s,j)$ can take on.

The definitions I've used to model the minimal uptime constraints can be seen in equations 25 and 26.

$$\text{calculateBound}(i,s,j) = \begin{cases} 0 & \text{if } p(i,s,j) \leq 0 \text{ and downtime}(i,s,j-1) \leq \text{mindown}(g_i) \\ +\infty & otherwise \end{cases} \tag{25}$$

$$\text{isViolated}(i,s,j) = \begin{cases} true & \text{if } p(i,s,j-1) \leq 0 \text{ and downtime}(i,s,j-1) < \text{mindown}(g_i) \text{ and } p(i,s,j) > 0 \\ false & otherwise \end{cases} \tag{26}$$

6.9 Demand constraints

Furthermore, the problem contains demand constraints. These are simply a number which indicate the amount of forecasted demand for power at a given point in time at a given node in the network. These are given as a vector of numbers for each node, in which every number corresponds to one timestep at each node in the network. I will treat these as soft constraints for reasons which will become clear in section 13.

7 Some observations about the hybrid Unit commitment and economic dispatch problem

With all of the above knowledge of the problem, we can now make some observations which will provide us with the insights we need to further model this problem.

The information from section 6 leads me to create table 5.

Table 5: Initial observations about constraints

Constraint	Upper bound	Lower bound	Forwards looking (t+1)	Backwards looking (t-1)
Pmin	No	Yes	No	No
Pmax	Yes	No	No	No
Ramp up	Yes	Yes	Yes	Yes
Ramp down	Yes	Yes	Yes	Yes
Startup limit	Yes	No	No	Yes
Shutdown limit	Yes	No	Yes	No
Minimum uptime	No	Yes	No	Yes
Minimum downtime	Yes	No	No	Yes

From this table it becomes clear that nearly all information we require to evaluate the constraints of this problem quickly, can be found in the schedule by looking at the value of $p(i, s, j)$ or one of its direct temporal neighbours $p(i, s, j - 1)$ or $p(i, s, j + 1)$. The only exceptions to this are the minimum uptime and minimum downtime constraints, but it's also clear that if we are willing to keep two variables of extra state which denote uptime(i, s, j) and downtime(i, s, j) for each combination of (i, s, j), we can also access this information from the direct temporal neighbours of (i, s, j). These constraints can all be evaluated in constant time, and the variables we'd have to store can also be updated in constant time in all operations which do not change the state of a unit i subunit s at time t_j from on to off or off to on.

In section 6 I've already touched upon the fact that the ramp up and ramp down constraints are not just one constraint each, but in practise manifest themselves as being two separate constraints with their own bounds and validity criteria each. For the sake of clarity I have separated these ramping constraints from the constraints in table 5 and listed how their various properties relate and turn out in table 6.

Table 6: Properties and relations of the ramping constraints

Constraint	Temporal direction	Upper bound	Lower bound
Ramp up	Forwards	No	Yes
Ramp up	Backwards	Yes	No
Ramp down	Forwards	Yes	No
Ramp down	Backwards	No	Yes

Another noteworthy observation is that the variables calculated by the upper and lower bounds formed by these constraints, will only change in $(i, s, j-1)$, (i, s, j) and $(i, s, j+1)$ if the value of $p(i, s, j)$ is updated. This in turn implies that it is possible to split the objective function of the hybrid unit commitment and economic dispatch problem into smaller components, which all individually affect the outcome of the objective function in an additive manner. However, because the manner in which the components of the cost functions are combined is additive, we can also recalculate only the parts of the cost function that actually change if we make a small change in a schedule, calculate the differences (delta δ) in cost of these components, and then add the deltas to the previously calculated total value of the objective function for the entire problem. The fact that we can save a lot of calculations this way, is extremely beneficial for an approach using local search.

While the feasible region is undoubtedly a higher dimensional polygon which is hard to imagine, it is possible to get an intuitive sense of what the feasible region containing all admissible solutions, which satisfy all hard constraints, looks like for one unit at a single point in time. An example of this representation can be seen in figure 6. From this figure we can derive that the value for $p(i, s, t)$ is limited by the values assigned to $(i, s, j - 1)$ and $(i, s, j + 1)$.

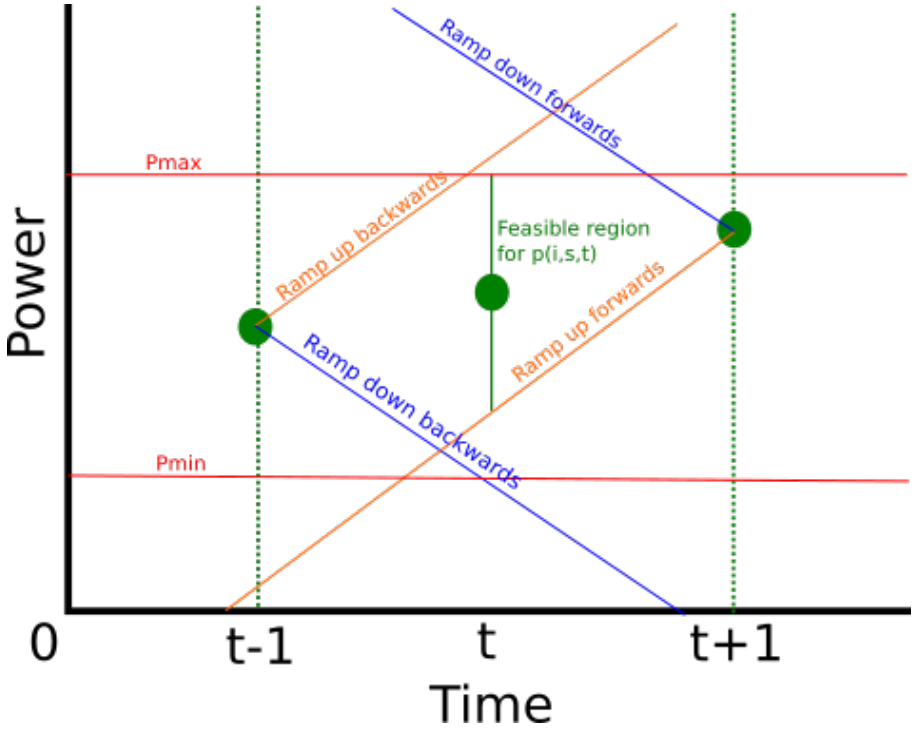


Figure 6: Intuitive visualisation of how various constraints restrict the feasible region of the value of $p(i, s, t)$. Note that the ramping constraints originate in both directions from each $p(i, s, t)\forall t$ but that only half or the constraints originating from $p(i, s, t-1)$ and $p(i, s, t+1)$ have been drawn for the sake of clarity.

We will exploit these observations in the construction of a suitable state representation for our local search algorithm.

8 Objective Functions (Cost Functions)

The instances under consideration all use equation 27 as their fuel cost function. This is the first component of the cost function.

$$f(p) = a + bp + cp^2 \tag{27}$$

In equation 27 p is the amount of power a generator generates at a given point in time and a, b and c are given constants which differ per unit.

However the instances under consideration have just one of two different functions for calculating the startup costs.

They can either use equation 28 in which FSC is the fixed startup cost constant, VSC is the variable startup cost constant and the constant λ are given for each unit,

$$StartupCost = \text{FSC} + \text{VSC} \times (1 - e^{(-\lambda)}) \tag{28}$$

or the instances can use a cost function which defines its variable startup cost in a piecewise manner. The intervals to define the startup cost in a piecewise manner are specified for each generator in each instance. In the instance files, the SCI colum denotes the intervals and the SCV colum defines the values corresponding to each interval. For example, the function for unit 1 in the GA10 instance looks like equation 29.

$$StartupCost = \begin{cases} 9000 & \text{if downtime } > 13 \\ 4500 & \text{if downtime } \leq 13 \end{cases} \tag{29}$$

Note that the functions modeling the startup cost, only apply at the points in time where a generator is being switched on from a state in which it was switched off.

9 Algorithmic state representation

In section 7 we have made some important observations which we can exploit to guide the design of our local search algorithm. At the core of this algorithm is always a data structure which represents the state of our search process. This data structure will model the feasible region of the problem and will keep track of all variables we need to decide whether or not a certain change to the schedule is admissible.

Due to the observations from section 7 I've come up with a representation which has six components at its core.

These six components are:

1. A UnitState.
2. A NodeState.
3. A TimeState.
4. A Node.
5. A Unit.
6. An Instance: An instance of the problem we are currently solving.

I will detail how these are modeled and how these are related to each other in this section.

A conceptual diagram of the horizontal and vertical relations between these components can be seen in figure 7 and 8.

9.1 The UnitState

Formally the UnitState class (or UnitState for short) can be regarded as a large tuple which at least contains:

- Indices: The i, s and j indices for its corresponding unit, subunit and time coordinates in the state-matrix to which this UnitState corresponds.
- Power: The value of $p(i, s, j)$ which will appear in the final solution.
- Cost: A cost variable which contains the partial contribution to the total cost of the current solution.
- Uptime: A variable which denotes the uptime of unit i, subunit s at time j in the schedule.
- Downtime: A variable which denotes the downtime of unit i, subunit s at time j in the schedule.
- InitialUptime: A variable which can be set if the given instance defines a default value for uptime which should be restored if at some point in the search process the unit is turned off after it has been turned on at t_j.
- InitialDowntime: A variable which can be set if the given instance defines a default value for downtime which should be restored if at some point in the search process the unit is turned on after it has been turned off at t_j.
- MaximumPowerConstraints: A list of all constraints which provide an upper bound on the value of $p(i, s, j)$.
- PowerUpperLimit: The minimum of all the results of the calculateBound function of all constraints in the list MaximumPowerConstraints.
- MinimumPowerConstraints: A list of all constraints which provide a lower bound on the value of $p(i, s, j)$.

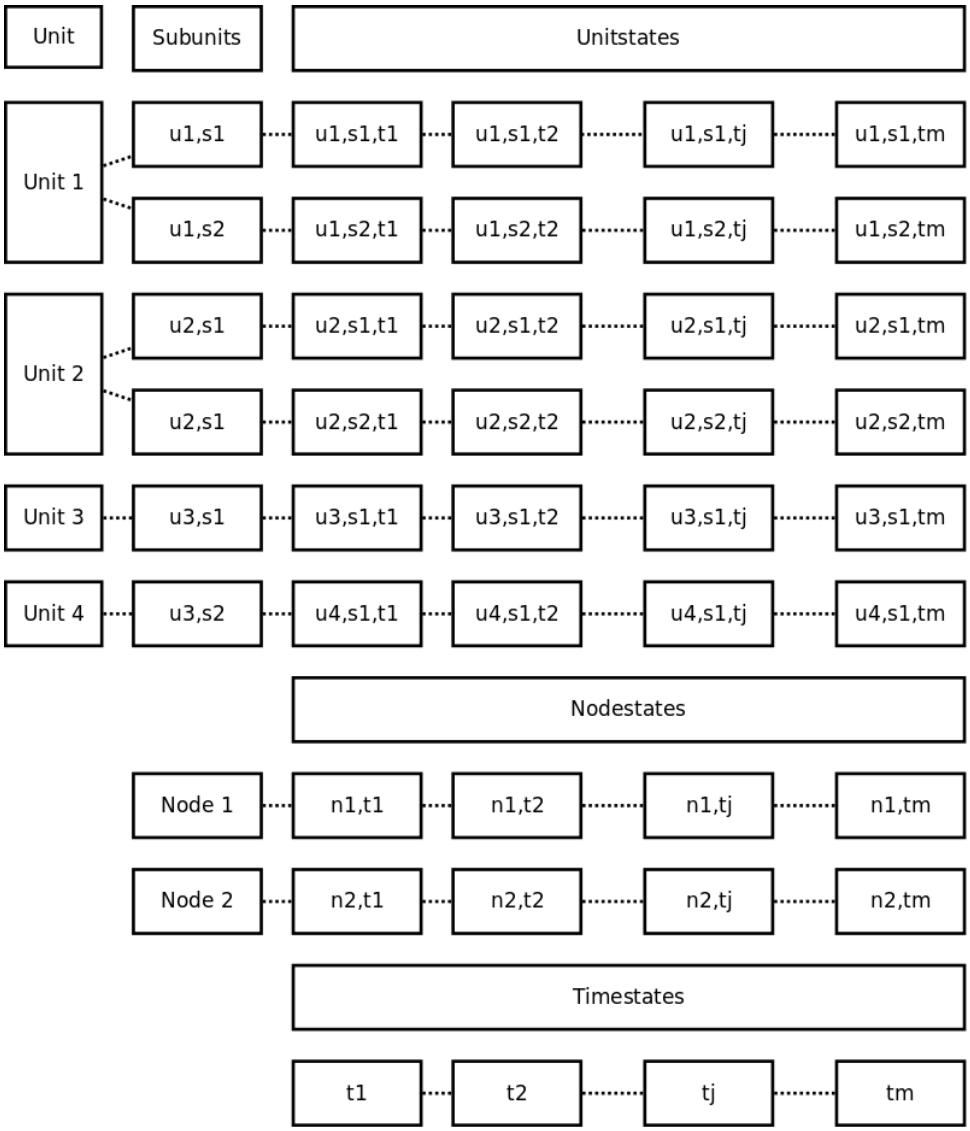


Figure 7: Horizontal relationships between the various components of the state description.

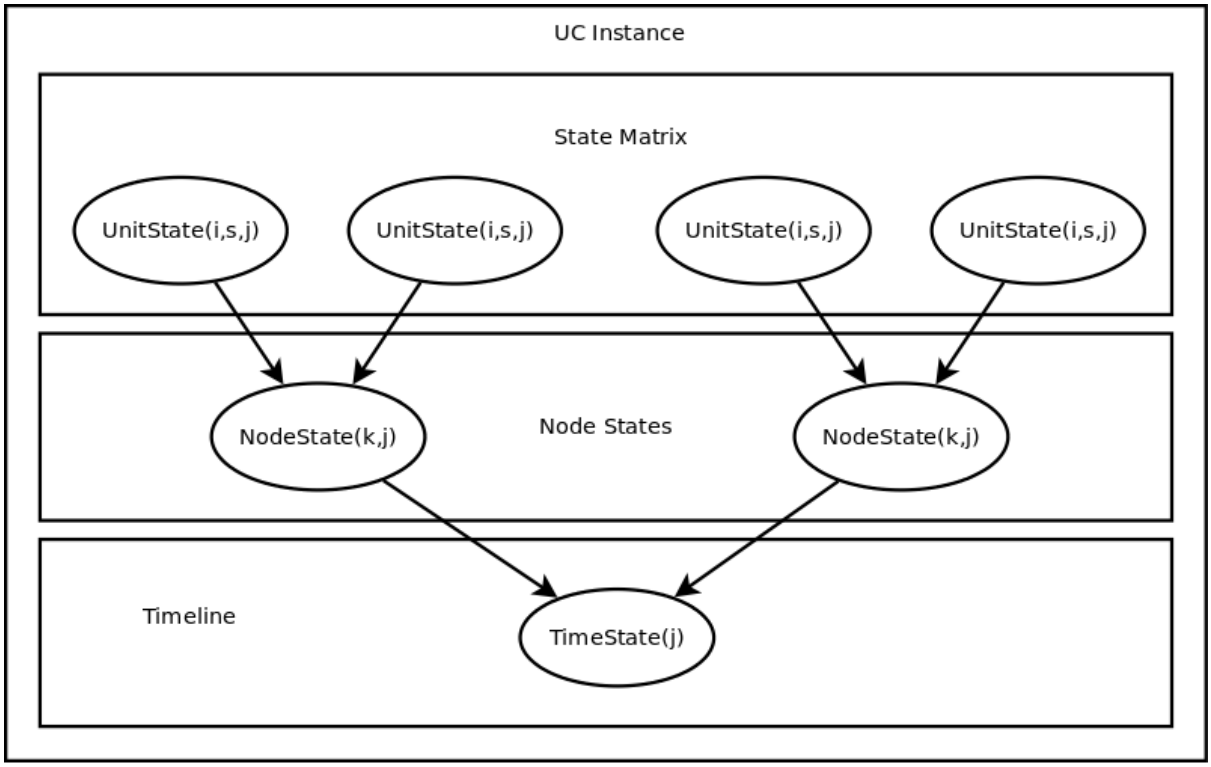


Figure 8: Vertical relationships between the various components of the state description.

- PowerLowerLimit: The maximum of all the results of the calculateBound function of all constraints in the list MinimumPowerConstraints.
- Unit: A pointer for quick access to the Unit variable which represents the unit to which this UnitState belongs and contains all relevant variables for this.
- NodeState: A pointer for quick access to the NodeState which corresponds to the NodeState for Node k at time t_j to which this Unit is connected.
- Instance: A pointer for quick access to the Instance object which represents this instance.
- Previous: A pointer for quick access to the UnitState $(i, s, j-1)$ which is chronologically the previous. If a previous UnitState does not exist, this pointer is nil.
- Next: A pointer for quick access to the UnitState $(i, s, j+1)$ which is chronologically the next. If a next UnitState does not exist, this pointer is nil.
- CostFunctionComponents: A list of the partial components of the cost function which apply to this UnitState.

Most of the variables in the UnitState-tuple are exactly what the description in the above list states. However the Cost, PowerUpperLimit, PowerLowerLimit and CostFunctionComponents require some additional explanation.

The CostFunctionComponents variable contains a list of all components of the cost function which apply to that UnitState. Among these are the functions for the startup and fuel cost as defined in section 8, which each comprise a CostFunctionComponent of their own. The value of the cost variable is then defined as the sum over all CostFunctionComponents.

To calculate the value of PowerUpperLimit, we take the minimum of the result of the calculateBound function of each constraint in the MaximumPowerConstraints list like denoted in equation 30.

$$PowerUpperLimit = \min_{\text{Constraint } c \in \text{ MaximumPowerConstraints}} \text{c.calculateBound}(i, s, j) \tag{30}$$

The value of PowerLowerLimit is calculated analogously, but instead of the minimum, we take the maximum of the result of the calculateBound function of each constraint in the MinimumPowerConstraints as shown in equation 31.

$$PowerLowerLimit = \max_{\text{Constraint } c \in \text{ MinimumPowerConstraints}} \text{c.calculateBound}(i, s, j) \tag{31}$$

Furthermore, the functions isGenerating (equation 32), isStartup (equation 33), isShutdown (equation 34), negativeSpinningReserve (equation 35), positiveSpinningReserve (equation 36) and operatingReserve (equation 37) are defined for the UnitState.

$$\text{isGenerating} = Power > 0 \tag{32}$$

$$\text{isStartup} = \begin{cases} true & \text{if the uptime of the previous UnitState is 0 and uptime} = 1 \\ false & \text{otherwise} \end{cases} \tag{33}$$

$$\text{isShutdown} = \begin{cases} true & \text{if this UnitState is generating and the next UnitState is not} \\ false & \text{otherwise} \end{cases} \tag{34}$$

$$\text{negativeSpinningReserve} = \begin{cases} power - PowerLowerLimit & \text{if this UnitState is generating} \\ 0 & \text{otherwise} \end{cases} \tag{35}$$

$$\text{positiveSpinningReserve} = \begin{cases} PowerUpperLimit - power & \text{if this UnitState is generating} \\ 0 & \text{otherwise} \end{cases} \tag{36}$$

$$\text{operatingReserve} = \begin{cases} SU(i) & \text{if this UnitState is not generating and can be started} \\ 0 & \text{if this UnitState is generating and cannot be started} \\ PositiveSpinningReserve & \text{otherwise} \end{cases} \tag{37}$$

9.2 The NodeState

The NodeState is a tuple containing the variables required to represent the state of a Node in the network at a certain point in time. It contains the following variables:

- Node: A pointer for quick access to the Node to which this NodeState belongs.
- Time: A pointer for quick access to the TimeState related to this NodeState.
- UnitStates: A list of UnitStates for quick access to the UnitStates related to this NodeState.
- Instance: A pointer for quick access to the Instance object which represents this instance.
- CurrentDispatch: The sum of the value of the Power variable of all UnitState tuples in UnitStates. This corresponds to the total amount of power generated at this Node at time t_j.
- Cost: The sum of the result of the objective functions incurred by all components of the cost-function for this NodeState which are present in CostFunctionComponents. This is the cost incurred by this NodeState at time t_j.
- OperatingReserve: The sum of the value returned by the operatingReserve function applied to all UnitState tuples in Unitstates. This corresponds to the total amount of power which can be brought online from an offline state at this Node at time t_j.

- PositiveSpinningReserve: The sum of the value returned by the PositiveSpinningReserve function applied to all UnitState tuples in Unitstates. This corresponds to the total amount of additional power which can be generated at this Node at time t_j.
- NegativeSpinningReserve: The sum of the value returned by the NegativeSpinningReserve function applied to all UnitState tuples in Unitstates. This corresponds to the total amount of power by which the generation can be lowered at this Node at time t_j.
- ActiveUnitStates: A count of all UnitStates in the list 'UnitStates' for which the function isGenerating is true. This corresponds to the total number of generators which are switched on at this Node at time t_j.
- CostFunctionComponents: A list of all relevant components of the cost function which apply to this Node at time t_j.

The functions demand (which simply returns the forecasted demand at node k at time t_j), calculateShortage (equation 38) and calculateSurplus (equation 39) are defined as well:

$$\text{calculateShortage} = \begin{cases} demand - currentDispatch & \text{if } demand - currentDispatch > 0 \\ 0 & \text{otherwise} \end{cases} \tag{38}$$

$$\text{calculateSurplus} = \begin{cases} currentDispatch - demand & \text{if } currentDispatch - demand > 0 \\ 0 & \text{otherwise} \end{cases} \tag{39}$$

9.3 The TimeState

The TimeState is a tuple containing the variables required to represent the state of some aggegrate variables for the entire network at a point in time t_j.

It contains the following variables:

- CurrentDispatch: The sum of the value of the CurrentDispatch variable for all NodeState tuples in NodeStates.
- totalDemand: The sum of the value returned by the demand-function for all NodeState tuples in NodeStates.
- Cost: The sum of the result of the objective functions incurred by all components of the cost-function for this TimeState which are present in CostFunctionComponents. This is the cost incurred by this TimeState for time t_j.
- Shortage: The sum of the value returned by the calculateShortage function applied to all NodeState tuples in NodeStates. This corresponds to the total amount of unmet power demand at this TimeState at time t_j.
- Surplus: The sum of the value returned by the calculateSurplus function applied to all NodeState tuples in NodeStates. This corresponds to the total amount of surplus power at this TimeState at time t_j.
- OperatingReserve: The sum of the value of the OperatingReserve variable of all NodeState tuples in NodeStates. This corresponds to the total amount of power which can be brought online from an offline state at this point in time (t_j).
- PositiveSpinningReserve: The sum of the value of the PositiveSpinningReserve variable of all NodeState tuples in NodeStates. This corresponds to the total amount of spinning reserve power available at time t_j.
- NegativeSpinningReserve: The sum of the value of the NegativeSpinningReserve variable of all NodeState tuples in NodeStates. This corresponds to the total amount of power by which the generation can be lowered at time t_j.

- ActiveUnitStates: The sum of the value of the ActiveUnitStates variable of all NodeState tuples in NodeStates. This corresponds to the total number of generators which are switched on at time t_j.
- TimeIndex: j because this variable represents the point in time (t_j) to which this TimeState corresponds.
- NodeStates: A list containing all NodeState tuples which model the state of a node in the network at time t_j.
- CostFunctionComponents: A list of all relevant components of the cost function which apply to this TimeState representing time t_j.

9.4 The Node

A Node is just a tuple which contains its id value and a demands vector in which $demands_j$ denotes the total amount of power demand at time t_j.

9.5 The Unit

The Unit is a tuple which contains all variables which define the characteristics of a generator or unit. Because there are a lot of variables, this is a relatively long list.

- Index: The index i of the unit.
- Id: The ID number of a unit.
- Count: The number of subunits which make up this composite unit if $count > 1$. If $count = 1$ this unit is not a composite unit and therefore will only have one subunit.
- P_{max}: The value used to define the P_{max} constraint in section 6.1.
- P_{min}: The value used to define the P_{min} constraint in section 6.2.
- a, b and c: The constants used in equation 27 which is used to calculate the fuel costs.
- RampUpLimit: The value returned by the function $RU(i)$ which is used to define the constraints for the ramp-up limit in subsection 6.3.
- RampDownLimit: The value returned by the function $RD(i)$ which is used to define the constraints for the ramp-down limit in subsection 6.4.
- StartupLimit: The value returned by the function $SU(i)$ which is used to define the constraints for the startup limit in subsection 6.5.
- ShutdownLimit: The value returned by the function $SD(i)$ which is used to define the constraints for the shutdown limit in subsection 6.6.
- MinUp: The value returned by the function minup(i) which is used to define the minimum uptime constraint in subsection 6.7.
- MinDown: The value returned by the function mindown(i) which is used to define the minimum downtime constraint in subsection 6.8.
- FixedStartupCost, VariableStartupCost and λ: The values for FSC, VSC and λ in equation 28.
- VariableStartupCostInterval: The respective values for the interval boundaries in the predicates of equation 29
- VariableStartupCostValue: The respective values for the results in equation 29.

9.6 The Instance

The instance is the tuple which contains the entire state representation for the local search for a given instance of the hybrid unit commitment and economic dispatch problem. This consists of the following variables:

- Units: A list containing all Units in this instance.
- Nodes: A list containing all Nodes in this instance.
- Time: A number which denotes the number of TimeSteps in this instance.
- StateMatrix: A 3-dimensional array of UnitStates for which the first dimension denotes the unit component (g_i), the second dimension denotes the subunit if the unit is clustered (s) and the third dimension denotes the time dimension (t_j) to which the UnitState stored at that position corresponds.
- NodeStates: A 2-dimensional array of NodeStates in which the first dimension denotes the number of the Node (n_k) in the network and the second dimension denotes the time dimension (t_j) to which the NodeState stored at that position corresponds.
- Timeline: An array of TimeStates of which each TimeState stored at index j in the array corresponds to the state at time t_j.
- TotalDemand: A variable which contains the sum of all demands at all NodeStates in the NodeStates array.
- TotalCost: A variable which stores the result of the sum over all CostFunctionComponents which have currently been added to the state representation.
- TotalDispatch: The total amount of energy generated by all units at all timesteps with the current schedule.
- Shortage: The total amount of unmet energy demands (shortage) with the current schedule.
- Surplus: The total amount of surplus energy generated with the current schedule.
- OperatingReserve: The combined total amount of operating reserve present in the schedule across all UnitStates at all Nodes at all points in Time.
- PositiveSpinningReserve: The combined total amount of positive spinning reserve present in the schedule across all UnitStates at all Nodes at all points in Time.
- NegativeSpinningReserve: The combined total amount of negative spinning reserve present in the schedule across all UnitStates at all Nodes at all points in Time.
- TotalUnitStates: The total number of UnitState tuples present in this instance.
- ActiveUnitStates: The total number of UnitState tuples in this instance at which the generator is on.

Note that the variables TotalDemand, TotalCost, TotalDispatch, Shortage, Surplus, OperatingReserve, NegativeSpinningReserve, PositiveSpinningReserve and ActiveUnitStates can all be used as performance measures for the current schedule.

10 Updating the state representation

In section 9 we have seen how the state in this local search algorithm was modeled, but what we have not discussed is a procedure for how we can update all those variables to keep them consistent.

Fortunately this representation allows us to do so with a relatively simple procedure. We have to tell this procedure two things: The new value we want to write to the power-variable of a given UnitState and whether or not we want to also call the update procedure without changing the power-value of its direct neighbours (the next UnitState $(i, s, j + 1)$ and the previous UnitState $(i, s, j - 1)$).

The update procedure for the state of a UnitState works as follows:

1. Store the old values for Cost, OperatingReserve, PositiveSpinningReserve, NegativeSpinningReserve and on/off status of the generator which belongs to the given UnitState at (i, s, j).
2. Calculate the difference between the old value of power and the new value we are going to replace it with. Call this value δp.
3. Update power with the new power value.
4. Recalculate all the state state variables which depend on the value of power. These are the Uptime, Downtime, PowerUpperLimit as mandated by the constraints in MaximumPowerConstraints, PowerLowerLimit as mandated by the constraints in MinimumPowerConstraints and finally the value of Cost mandated by summing over the values returned by calculateBound from all CostFunctionComponents in CostFunctionComponents.
5. Calculate the difference between the old and new values of Cost, OperatingReserve, PositiveSpinningReserve and NegativeSpinningReserve. Call these $\delta Cost$, $\delta OperatingReserve$, $\delta PositiveSpinningReserve$ and $\delta NegativeSpinningReserve$.
6. Update the variables TotalCost and TotalDispatch in the Instance tuple by doing $TotalCost = TotalCost + \delta Cost$ and $TotalDispatch = TotalDispatch + \delta Power$.
7. Calculate $\delta ActiveUnitStates = \begin{cases} -1 & \text{if the generator was previously on and is now turned off} \\ 1 & \text{if the generator was previously off and is now turned on} \\ 0 & otherwise \end{cases}$
8. Call the state update procedure for NodeState while supplying δpower, δOperatingReserve, δPositiveSpinningReserve, δnegativeSpinningReserve and δActiveUnitStates.
9. If the flag to update the previous and next UnitStates is given, call the update procedure for the UnitState $(i, s, j - 1)$ and $(i, s, j + 1)$ with their respective currently assigned power values and the flag to update their neighbours set to false.

The procedure to update the variables of the NodeState is very similar to the update procedure for the UnitState and works as follows:

1. Receive $\delta power$, $\delta OperatingReserve$, $\delta PositiveSpinningReserve$, $\delta negativeSpinningReserve$ and $\delta ActiveUnitStates$ from the UnitState update procedure.
2. Store the old values for Cost, Shortage, Surplus and ActiveUnitStates.
3. Update $CurrentDispatch = CurrentDispatch + \delta power$
4. Update $OperatingReserve = OperatingReserve + \delta OperatingReserve$
5. Update $PositiveSpinningReserve = PositiveSpinningReserve + \delta PositiveSpinningReserve$
6. Update $NegativeSpinningReserve = NegativeSpinningReserve + \delta NegativeSpinningReserve$
7. Update $ActiveUnitStates = ActiveUnitStates + \delta ActiveUnitStates$
8. Recalculate Cost mandated by summing over the values returned by calculateBound from all CostFunctionComponents in CostFunctionComponents.

9. Calculate the difference between the old and new values of Cost, Shortage and Surplus. Call these values $\delta Cost$, $\delta Shortage$ and $\delta Surplus$.

10. Update the variable TotalCost in the Instance tuple by doing $TotalCost = TotalCost + \delta Cost$.

11. Call the state update procedure for TimeState while supplying $\delta power$, $\delta Shortage$, $\delta Surplus$, $\delta OperatingReserve$, $\delta PositiveSpinningReserve$, $\delta NegativeSpinningReserve$ and $\delta ActiveUnitStates$.

The procedure to update the variables of the TimeState is again very similar to the two above and works as follows:

1. Receive δpower, δShortage, δSurplus, δOperatingReserve, δPositiveSpinningReserve, δNegativeSpinningReserve and δActiveUnitStates from the NodeState update procedure.

2. Update $CurrentDispatch = CurrentDispatch + \delta power$

3. Update $Shortage = Shortage + \delta Shortage$

4. Update $Surplus = Surplus + \delta Surplus$

5. Update $OperatingReserve = OperatingReserve + \delta OperatingReserve$

6. Update $PositiveSpinningReserve = PositiveSpinningReserve + \delta PositiveSpinningReserve$

7. Update $NegativeSpinningReserve = NegativeSpinningReserve + \delta NegativeSpinningReserve$

8. Update $ActiveUnitStates = ActiveUnitStates + \delta ActiveUnitStates$

9. Recalculate Cost mandated by summing over the values returned by calculateBound from all CostFunctionComponents in CostFunctionComponents.

10. Update the variable Shortage in the Instance $Shortage = Shortage + \delta Shortage$

11. Update the variable Surplus in the Instance $Surplus = Surplus + \delta Surplus$

12. Update the variable OperatingReserve in the Instance $OperatingReserve = OperatingReserve + \delta OperatingReserve$

13. Update the variable PositiveSpinningReserve in the Instance $PositiveSpinningReserve = PositiveSpinningReserve + \delta PositiveSpinningReserve$

14. Update the variable NegativeSpinningReserve in the Instance $NegativeSpinningReserve = NegativeSpinningReserve + \delta NegativeSpinningReserve$

15. Update the variable ActiveUnitStates in the Instance $ActiveUnitStates = ActiveUnitStates + \delta ActiveUnitStates$

16. Update the variable Cost in the Instance $TotalCost = TotalCost + \delta Cost$

The conceptual steps can be seen in figure 9.

We now have a relatively flexible and extendable state-representation for the hybrid unit commitment and economic dispatch problem which allows for:

- Simulation of a schedule: The representation itself contains all the variables a simulation of a schedule requires.
- Memoization: Most of the variables do not have to be considered or recalculated and the variables we do have to update, can be updated with only the differences in their values (δ).
- Fast updates: Most of the updates to variables can be done in contant or near-constant amounts of time.

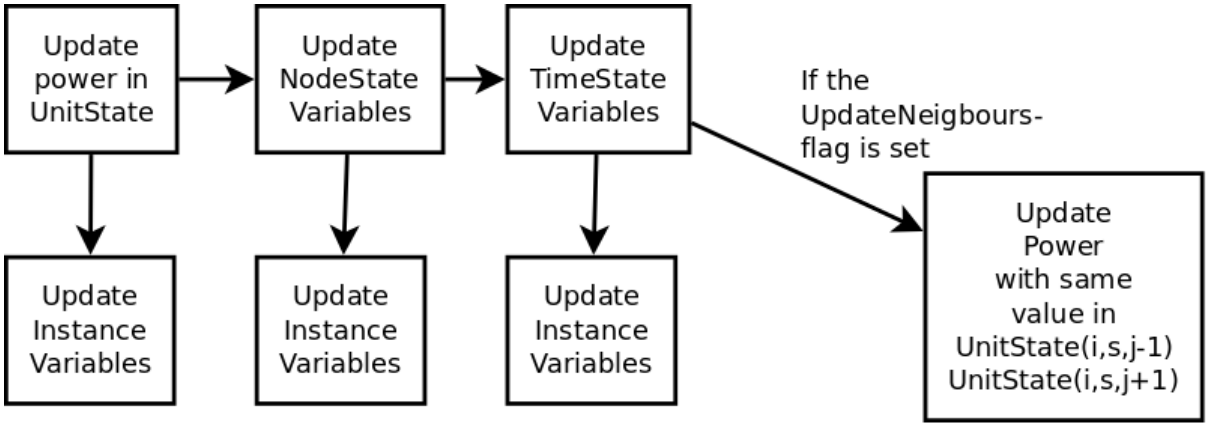


Figure 9: Conceptual overview of the procedure to update all state variables

- Fast checking of validity for constraints: For each power value we change, we can check in constant time if that value falls within an admissible range (between PowerUpperLimit and PowerLower-Limit) and therefore, if a change we'd like to make to this problem is inside or outside of the feasible region of the problem.
- The flexible cost functions: We can attach multiple CostFunctionComponents to various Unit-States, NodeStates and TimeStates.
- Flexible extension: If we want to introduce some additional variables to keep track of different performance measures, it's now relatively easy to add those to the respective UnitState, NodeState or TimeState. It's also relatively straightforward to adapt the update procedures to keep track of these additional variables as well.
- A limited number of factors we have to take into account while working on the other parts of the local search.

We have succeeded in modeling the higher-dimensional, non-linear and non-convex problem space of the hybrid UC+ED problem and we can traverse it by modifying the power values in the UnitStates. Because we have now succeeded in modeling the problem space and know where the boundaries of the feasible region are, this allows us to approach this local search problem as if it is somewhat similar to an Interior Point Problem.

This also means that we are no longer constrained by the limitations of the Linear Programming heuristic, which traverses the boundary region of a higher dimensional convex polygon.

11 Local Search Operators

In this section I will discuss the operators I've implemented to mutate the state representation to a neighbouring state representation. For convenience, I have called these operators "moves" because they move the state of the search from one solution, to one of its neighbouring solutions.

Once generated, a move strictly belongs to only one given Instance at one given iteration of the search and needs to implement three operations:

1. isValid: A predicate used to check whether or not the neighbouring state this move would move the schedule to, is still within the feasible region of the problem space.
2. apply: The apply operation executes the mutation of the state such that once it returns, the state representation reflects the schedule of the new neighbour.
3. undo: The undo operation reverts the apply operation. Due to the very diverse way the apply function can be implemented, it's up to each individual move type to implement a mechanism which stores the power-values of the previous state before it changes them in such a way that the undo operation can satisfy its stated purpose.

In the following subsections I will discuss the move-types (operators) I've implemented. The Ramp operator is used to perform all operations required to solve the Economic Dispatch part of the hybrid unit commitment and economic dispatch problem because it is the operator which can change the values of power dispatched from the generators. The other operators (TurnOnAtPmin, TurnOff, StartupPush-Back and ExtendRunTime) are the operators which implement the unit commitment part of the hybrid problem. The implementations of these operators can be found in the moves package of the source code.

11.1 Ramp

A ramp move changes the power value of a single UnitState in the instance. To do so, it requires coordinates i, s and j so it can find the corresponding UnitState in the state matrix stored in the Instance tuple. It also requires a δpower value which denotes the amount of change which has to be applied to the power-value of the UnitState tuple. Its validity check is to simply test if the unit is generating and the new power value would be bigger than the P_{min} and P_{max} values of the Unit to which this UnitState corresponds and if it does, test if the value is within the range between the PowerLowerLimit and PowerUpperLimit values stored in the UnitStateTuple. To implement the undo-operation, we simply store the previous power-value in variable and write it back to the power-value of the UnitState tuple (i, s, j) once the undo function is called.

11.2 TurnOnAtPmin

TurnOnAtPmin will, as its name indicates, turn on a specific generator at a certain point in time. To do so, it requires coordinates i, s and j so it can find the corresponding UnitState in the state matrix stored in the Instance tuple.

It has to perform one of the following operations:

- Turn on the generator for the remaining duration of the schedule, because the shutdown time is past the end of the schedule.
- Turn on the generator for the next minup(i) timesteps and ensure that the next mindown(i) timesteps after that are updated, so the uptimes and downtimes are correct.
- Turn on the generator for the next minup(i) timesteps and combine the interval that is currently being brought online, with an interval which follows the current in which the generator is already generating, and update the remainder of this interval.

11.3 TurnOff

TurnOff will simply check the variables in UnitState (i, s, j) to see if it can turn off the generator at all the succeeding points in time until the next startup or the end of the schedule is reached. It can do this if UnitState (i, s, j) is a startup moment, or at any time in which $uptime > minup(i)$. It will turn generators off by setting the UnitState's power value to 0.

11.4 StartupPushBack

StartupPushback can be applied in the interval in which the minimal uptime constraint has not been satisfied yet, but the remainder of the interval in which the generator is turned on, is longer than $minup(i)$. It will turn the generator off for any UnitState between the previous startup and the current time t_j if the requirement for its application is met. It will turn generators off by setting the UnitState's power value to 0.

11.5 ExtendRunTime

The ExtendRunTime move can extend an interval in which the unit was previously on before t_j up until time t_j. It will turn generators on by setting the UnitState's power value to $P_{min}(i)$ for all the unitstates between the previous shutdown and t_j.

12 Stategies for neighbourhood exploration

In this section I will describe the various strategies we can employ to explore the search neighbourhood. I have termed these MoveGenerators in the source code, because these strategies generate a move which then changes the state representation such that it represents a neighbouring solution. There are currently two main categories of MoveGenerators. The first category are MoveGenerators which rely on random numbers, while MoveGenerators of the second category will iteratively traverse the schedule from left to right and top to bottom and will try every possible move at every possible unitstate in the schedule in a brute-force way.

12.1 RandomRampMoveGenerator

This is a move generator which always creates a Ramp move, but it will choose its values i, s and j which are then used to select a UnitState randomly. Once the UnitState has been selected, a new power value will be generated according to equation 40.

$$\text{newPower} = PowerLowerLimit + ((PowerUpperLimit - PowerLowerLimit) \times Random([0,1]) \quad (40)$$

If the IntegerNumbers flag is set to true, the value of newPower will be rounded to the nearest integer value and otherwise no rounding will be done. Once we have the value of $newPower$ we determine the difference (δPower) between the current Power value at UnitState (i, s, j) which is the last value we need to generate a random move.

This move generator will ensure these values are within the valid array bounds of the StateMatrix in the Instance tuple, but that's all it will check. This means that it can also select a UnitState in which it is impossible to ramp a generator. If this happens, the move this generator returns, will be invalid and as a consequence, it will be rejected by the local search algorithm.

12.2 EnhancedRandomRampMoveGenerator

This MoveGenerator works the same as the RandomRampMoveGenerator in subsection 12.1, but with the extra added requirement that it will only generate Ramp moves for UnitStates in which the state of the generator is on. This decreases the number of rejected moves at the cost of having to spend a little more time. It selects the UnitState for which it will generate a Ramp move as follows:

1. Iterate through all TimeState tuples in the TimeLine and keep a list of all TimeStates for which the variable ActiveUnitStates is greater than 0.
2. Select a random TimeState from the previous list 1.
3. Iterate through all UnitStates at time t_j and keep a list of UnitStates at which the generator is active.
4. Select a random UnitState from the list of step 3.
5. Generate the value for δPower just like the RandomRampMoveGenerator in 12.1.

12.3 RandomBestGuessRampMoveGenerator

The RandomBestGuessRampMoveGenerator will randomly select a UnitState from the StateMatrix. It will then try to match the power value at that UnitState to correct the discrepancy between the demand and dispatch at that point in time. This means that if there is an energy shortage or surplus, the power value will be increased or decreased to the point where either the discrepancy is resolved or the PowerUpperLimit or PowerLowerLimit of the UnitState are hit.

12.4 RandomExtendRunTimeMoveGenerator

The RandomExtendRunTimeMoveGenerator simply generates valid values for i, s and j and will then return an ExtendRunTime move which may or may not be valid.

12.5 RandomPowerOnMoveGenerator

The RandomPowerOnMoveGenerator simply generates valid values for i, s and j and will then return a TurnOnAtPmin move which may or may not be valid.

12.6 RandomStartUpPushBackMoveGenerator

The RandomStartUpPushBackMoveGenerator simply generates valid values for i, s and j and will then return a StartUpPushBack move which may or may not be valid.

12.7 RandomTurnOffMoveGenerator

The RandomTurnOffMoveGenerator simply generates valid values for i, s and j and will then return a TurnOff move which may or may not be valid.

12.8 RandomMoveGenerator

The RandomMoveGenerator does not generate any move by itself. It instead accepts a list of tuples of which each tuple will contain a value c and one of the other RandomMoveGenerator types listed in this section. On its creation, it will calculate $c_{total} = \sum_{\forall c} c$.

Once it gets the request to generate a new move, it will select one of the RandomMoveGenerators it was given on its creation. The probability that a given RandomMoveGenerator is selected is $Pr(\frac{c}{c_{total}})$. This way we can influence the probabilities of generating a certain move-type.

12.9 IterativeExtendRunTimeMoveGenerator

The IterativeExtendRunTimeMoveGenerator will iterate through all UnitStates in left-to-right and top-to-bottom order and return a ExtendRunTime move with the respective i, s and j values.

12.10 IterativePowerOffMoveGenerator

The IterativePowerOffMoveGenerator will iterate through all UnitStates in left-to-right and top-to-bottom order and return a TurnOff move with the respective i, s and j values.

12.11 IterativePowerOnMoveGenerator

The IterativePowerOnMoveGenerator will iterate through all UnitStates in left-to-right and top-to-bottom order and return a PowerOnAtPmin move with the respective i, s and j values.

12.12 IterativeSamplingRampMoveGenerator

The IterativeSamplingRampMoveGenerator will iterate through all UnitStates in left-to-right and top-to-bottom order and return a Ramp move with the respective i, s and j values. However, it will sample the domain between the PowerLowerLimit and PowerUpperLimit values of the UnitState tuple with a given resolution to calculate the value of δPower which is passed to a Ramp move. It generates Ramp moves with the same values for i, s and j multiple times sequentially, but with different values for δPower so the entire range between PowerLowerLimit and PowerUpperLimit can be sampled, before it moves on to the next UnitState.

12.13 IterativeCompositeMoveGenerator

Just like the RandomMoveGenerator in subsection 12.8, the IterativeCompositeMoveGenerator receives a list of IterativeMoveGenerators at its creation. As its name suggests, it will iterate through these move generators one by one until their neighbourhood is exhausted.

12.14 A final note on MoveGenerators

In this section we have seen that the MoveGenerators we use to generate moves which can take us to neighbouring solutions, can be subdivided into two broad categories: random and iterative move generators. While the neighbourhood presented by the RandomMoveGenerators is impossible to exhaust, this is not the case for the iterative MoveGenerators. Because the neighbourhood of some move generators can be exhausted, the local search needs to contain a facility which can be used to signal to a MoveGenerator if the last move it generated was accepted or rejected. For the RandomMoveGenerators this signal is irrelevant, but the IterativeMoveGenerators need these signals to update their internal state.

Consequently, we also need a mechanism for the MoveGenerators to signal to the Local Search that its neighbourhood is exhausted. While the RandomMoveGenerators will never send this signal (their neighbourhood cannot be exhausted), the IterativeMoveGenerators do need to be able to send this signal. The IterativeCompositeMoveGenerator from subsection 12.13 even relies on this signal to advance to the next MoveGenerator it has on its list.

It is important to note, that the Local Search I've built for this master thesis has been extended with these facilities. The local search has also been extended with a mechanism by which it can request a "last attempt" move to prevent the search from stalling and therefore terminating once its neighbourhood is exhausted. This mechanism is very useful for implementing the best in neighbourhood heuristic from subsection 4.3.3.

13 Obtaining initial solutions and the role of the cost function

There are various methods we can employ to obtain initial solutions for our local search. We could for example simply assume that the power value in all UnitStates is set to 0. This means that the search will always start with the assumption that all generators are off at any given time.

However, if we were to do this, the local search would never start at all, because in most instances, the minimum possible fuel cost is achieved when all units are turned off and no power is generated at all. So if we apply this in a naive way, we won't get anywhere with most search heuristics. There might be a way to make a random walk which simply accepts any neighbour (even if it makes the solution worse) work, but unfortunately then we also have no way of determining if a solution found during the random walk is actually better than the solution in which all units are turned off. So we have to do something else.

The second naive idea is to simply turn all units on at their P_{max} value at the start of the search. However, this also won't work, because we are minimizing a cost function and the minimal cost is always the instance in which all units are turned off. Therefore the local search will quickly favor all neighbours which ramp generators down and turn them off. Once the search finally terminates, we are left with the same schedule as we had in previous situation: The one in which all units are turned off for the entire duration of the schedule. So this doesn't work either.

There is another trick which is commonly used to solve various optimization problems which can be modeled as linear programs which might be useful here: Put penalty costs on situations which are undesirable outcomes in your final solution. Considering the fact that the data structure and state update procedures I have defined in sections 9 and 10 are highly flexible and can quite easily process an additional CostFunctionComponent, I have decided to introduce a CostFunctionComponent which can be used to introduce a penalty cost for every unit of energy that is not generated. This CostFunctionComponent is defined in equation 41. This CostFunctionComponent is then attached to every TimeState in the timeline of the Instance before the local search is started. Once the search terminates, we remove all CostFunctionComponents in the problem and then add the legitimate ones which truly belong to that specific instance again. When we've done that, we do one last pass over the state datastructure to recalculate all state variables which ensures that the costs and the schedule we produce are correct.

$$calculateCost = \begin{cases} \text{Shortage} \times \text{PenaltyCost} & \text{if } CurrentDispatch < TotalDemand \\ 0 & \text{otherwise} \end{cases} \tag{41}$$

Now we have a mechanism which ensures that the local search doesn't immediately optimize to the solution in which all generators are turned off. This however immedately poses a new problem: "How much should the penalty for non-delivered energy be?". Unfortunately, this is different for each and every

instance of the hybrid UC/ED problem and I have not been able to find a pattern I could exploit. So I did a Fermi-estimation by just running a couple of Simulated Annealing searches to first discover the correct order of magnitude for the penalty cost. This means that I ran multiple Simulated Annealing searches with multiples of 10 as penalty value, so 10, 100, 1000, 10000 and so on. For the GA10 instance, with a PenaltyCost of 10, the search didn't move at all from the state in which all units were turned off. A PenaltyCost of 100 seemed to give somewhat acceptable results. A PenaltyCost of 1000 seemed to give results with a worse score than 100, but it also had less instances in which demands could not be met. A PenaltyCost of 10000 appeared to be more or less indistinguishable from the case where the costs were set at 1000.

So this seemed to suggest that for the GA10 instance, the optimal range for the PenaltyCost was to be found somewhere between 10 and 1000. It was also clear that the end of the optimal range would be somewhere in the interval $[100, 1000]$, but it wasn't clear on which side of 100 that range would start.

Then I ran a couple more and longer Simulated Annealing searches, but this time with values 50, 75, 100, 200 and 500 for the PenaltyCost.

Using this procedure for all instances, we arrive at the appropriate values for PenaltyCost shown in table 7.

Instance	Values
GA10	100, 200 or 500
TAI38	20000
A110	75 or 100
KOR140	100, 200 or 500
RCUC200	200 or 500

Table 7: An overview of values which work well for the PenaltyCost for each instance.

Surprisingly it turns out that the answer to the research question "How do we construct an initial solution?" is: "We don't. We just exploit the properties of the data-structure which we use as a state representation. Then we put a penalty on the undesirable situation of non-delivered energy. We then use a Fermi-estimation to get an idea of the desirable order of magnitude for the PenaltyCost. We then further narrow down on preferable values by splitting the space between two orders of magnitude to arrive at values which work reasonably well for the PenaltyCost constant."

I do have to note that this is only possible, because we can treat the demand constraints as soft constraints, and because the values returned by the objective function can span multiple orders of magnitude as well.

14 Experimental environment

In this section, I will give details about my experimental environment and setup.

14.1 Experimental environment

These experiments were done on a Lenovo ThinkPad L480 and L580 which both have an Intel Core i5-8250 CPU and 32 GigaBytes of ram and Debian 11 (Linux) as operating system. The algorithms were implemented in Java and the experiments have been compiled and run with java-17-openjdk-amd64.

To keep the results reproducible, the runtime of the experiments has been limited by a stopping criterion which terminates the search after a fixed number of iterations. However, for practical purposes I have tied the names of the experiments to their expected running time on the wall-clock. To this end, I have tied the number of iterations a search is allowed to run, to the number of seconds implied by "15m" (15 minutes or 900 seconds) and "1h" (1 hour or 3600 seconds) by assuming that the search will always progress at a fictional constant rate of 250.000 iterations per second. This means that a 15 minute search may run for at most $15 \times 60 \times 250000 = 225000000$ iterations and that a 1 hour long search may run for $3600 \times 250000 = 9000000000$ iterations. Note that this gross indication of the expected running time, is just a very loose indication which often does not match the actual running time. The actual running time often differed from the actual running time on the computer the experiments were ran on too. However,

the results produced by my experiments should be reproducible as all random generators are seeded from one generator, which is always seeded with 0. These two factors combined should make the randomness in my experiments entirely deterministic, so that apart from the running times, the experimental results should in theory be entirely reproducible if one were to run the main method of the Experiments class in the localSearchCore package.

14.2 A note on the experiment naming and the files and folders the code will produce

By default my program will look for the .uc files in the instance directory, which it expects to be in the working directory from which the program was started. In that same working directory it will create two folders. One of these folders is named "tex" and the other one is named "logs". Throughout these folders a naming scheme is used which follows the convention that the files and folders will be separated by hyphen-minus signs. The order in which the various naming elements will appear is as follows:

1. Name of instance (e.g. "GA10")
2. Name of search strategy (e.g. "simulated annealing")
3. Name of defined move-generation strategy (e.g. "Random")
4. Shorthand notation for the estimated given maximum runtime (e.g. "r15m" or "r1h")
5. A value used as a penalty cost per unit of non-delivered energy (e.g. "p10", "p100", "p1k" or "p10k")
6. A number denoting which experimental run produced the file.
7. A somewhat descriptive name of what was output into each file.

In the log directory, the program writes logs which dump nearly everything contained in the entire state of each experimental run in a somewhat human readable format. In the tex directory, the program writes compilable latex-files which will can be used to generate very large PDF-documents which can create nice tables and plots of the solutions produced by an experimental run. In this same directory, the program also writes a lot of tsv-files which contain the tables which serve as the input for the pgfplots package used in the Latex documents. So both folders contain multiple representations in various forms of the final schedules found by the program. In the code, in the Experiment class in the localsearchCore package, there are three switches which can be used to control the volume of output generated by the .tex documents which one can change if desired. Finally I also want to note that the LocalSearch class in the LocalsearchCore package contains a switch called integerMoves. If this switch is set to true, the various moveGenerators will try to generate only Integer moves by rounding the random moves they generate. If this switch is set to false, the various moveGenerators will not do any rounding and just generate any floating point number within the specified range. By default, this switch is set to true.

15 Stopping criteria

The following four basic stopping criteria have been implemented.

- Stop after a fixed number of search iterations.
- Stop after a fixed amount of time on the wall-clock has passed.
- Stop if the solution has not been improved for a fixed number of iterations.
- Stop if the solution has not been improved after a fixed amount of time on the wall-clock has passed.

In addition to the four stopping criteria listed above, I've also implemented three stopping criteria which can be used to combine different stopping criteria.

- A disjunctive stopping criterion which evaluates to true if one or more of its supplied stopping criteria evaluates to true.
- A conjunctive stopping criterion which evaluates to true if all its supplied stopping criteria evaluate to true.
- An inverse stopping criterion which can be used to negate the value of its supplied stopping criterion.

16 Experiments

In this section I will describe how the various components which have been described throughout this document are combined to perform experiments to assess the feasibility of furter research into local search methods for the hybrid Unit Commitment and Economic Dispatch problem.

16.1 Experimental setup

We now have a very flexible set of building blocks which can be combined to construct various types of integrated local search algorithms for the UC+ED problem.

Before we start defining complete configurations for local search algorithms, we first need to discuss how we have to combine the various RandomMoveGenerators to arrive at a RandomMoveGenerator which is suitable for a simulated annealing local search.

We can do so by combining the following elements into a RandomMoveGenerator as follows:

- Strictly one of the following: EnhancedRandomRampMoveGenerator, RandomRampMoveGenerator or BestGuessRandomRampMoveGenerator. The value we assign to c to define the probability is 400.
- A RandomStartUpPushBackMoveGenerator. The value we assign to c to define the probability is 100.
- A RandomTurnOffMoveGenerator. The value we assign to c to define the probability is 100.
- A RandomExtendRunTimeMoveGenerator. The value we assign to c to define the probability is 100.
- A RandomPowerOnMoveGenerator. The value we assign to c to define the probability is 100.

Notice how the c values of the RandomMoveGenerators which generate moves which can turn generators on and off add up to 400, which is exactly the same as the value assigned to the EnhancedRandomRampMoveGenerator or RandomRampMoveGenerator and how the total adds up to 800. Because these values are chosen like this, there is either a 50% chance that we randomly generate a move which tries to ramp the problem and a 50% chance that we randomly generate a move which tries to switch generators on and off.

This distribution has been deliberately chosen, because we are trying to solve two separate problems in the same local search algorithm. By chosing the values like this, there is now a an equal probability for the local search to work on the Unit Commitment or the Economic Dispatch part of this problem. Which side the local search will work on, is purely left up to chance.

I will use this distributions for all experiments which utilize RandomMoveGenerators.

For example, we can run a local search algorithm which uses the simulated annealing heuristic if we combine these elements:

- A RandomMoveGenerator
- A DisjunctiveStoppingCriterion consisting of a FixedNumberOfIterations stopping criterion with its number of iterations set to 250000 times the number of seconds we want the search to run and an IterationsSinceLastImprovement stopping criterion with its number of iterations set to 60 times 250000.

- A SimulatedAnnealingAcceptanceStrategy with its maximum number of iterations set to 250000 times the number of seconds we want the search to run. This parameter is used in the temperature function of simulated annealing defined in equation 5.

We can create a greedy hillclimber like this:

- A RandomMoveGenerator
- A DisjunctiveStoppingCriterion consisting of a FixedNumberOfIterations stopping criterion with its number of iterations set to 250000 times the number of seconds we want the search to run and an IterationsSinceLastImprovement stopping criterion with its number of iterations set to 60 times 250000.
- A GreedyHillClimberAcceptanceStrategy.

We can create a random walk like this:

- A RandomMoveGenerator
- A DisjunctiveStoppingCriterion consisting of a FixedNumberOfIterations stopping criterion with its number of iterations set to 250000 times the number of seconds we want the search to run and an IterationsSinceLastImprovement stopping criterion with its number of iterations set to 60 times 250000.
- An AlwaysAcceptAcceptanceStrategy.

We can create a best in neighbourhood local search like this:

- An IterativeMoveGenerator
- A DisjunctiveStoppingCriterion consisting of a FixedNumberOfIterations stopping criterion with its number of iterations set to 250000 times the number of seconds we want the search to run and an IterationsSinceLastImprovement stopping criterion with its number of iterations set to 60 times 250000.
- A BestInNeighbourhoodAcceptanceStrategy as the main acceptance strategy.
- A GreedyHillClimberAcceptanceStrategy as the acceptance strategy for the situation in which the neighbourhood generated by the IterativeMoveGenerator is exhausted.

In early testing the best in neighbourhood and random walk search strategies did not yield any satisfying results when compared to the greedy hillclimber and simulated annealing strategies and since this was in line with expectations and therefore not surprising, I decided it was not feasible to do any extended experiments with these strategies.

So the main focus of my experiments was placed on the greedy hillclimber and simulated annealing.

With the information from section 14 and in particular the naming scheme from subsection 14.2, the configuration parameters used for the extended experiments can be derived from their name. For all configurations, I have done 50 different local search runs, except for the configurations involving the A110 and RCUC200 instances, for which I have done 8 runs each. I have cut down on the number of runs for these instances, because these were the last two instances under consideration and by then a quite clear pattern had emerged.

17 Results

Let's first define what it means for a schedule to be successful. A schedule is successful if all demands are met, such that there are no points in time with a shortage greater than zero. We can then calculate the ratio of successful versus unsuccessful schedules for the various configurations of our experiments. The results of this are shown in table 17.

Table 8: Success ratios of various experiments

ExperimentName	numberOfRuns	SuccesRate
A110-SimulatedAnnealing-EnhancedRandom-r1h-p100	8	1.0
GA10-HillClimberGreedy-Random-r15m-p10	50	None
GA10-HillClimberGreedy-BestGuessRandom-r15m-p10	50	None
GA10-HillClimberGreedy-EnhancedRandom-r15m-p10	50	None
GA10-HillClimberGreedy-Random-r15m-p100	50	0.18
GA10-HillClimberGreedy-BestGuessRandom-r15m-p100	50	None
GA10-HillClimberGreedy-EnhancedRandom-r15m-p100	50	0.16
GA10-HillClimberGreedy-Random-r15m-p1k	50	0.3
GA10-HillClimberGreedy-BestGuessRandom-r15m-p1k	50	None
GA10-HillClimberGreedy-EnhancedRandom-r15m-p1k	50	0.26
GA10-HillClimberGreedy-Random-r15m-p10k	50	0.32
GA10-HillClimberGreedy-BestGuessRandom-r15m-p10k	50	None
GA10-HillClimberGreedy-EnhancedRandom-r15m-p10k	50	0.26
GA10-SimulatedAnnealing2-Random-r1h-p50	50	None
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p50	50	None
GA10-SimulatedAnnealing2-Random-r1h-p75	50	None
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p75	50	None
GA10-SimulatedAnnealing2-Random-r1h-p100	50	0.14
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p100	50	0.02
GA10-SimulatedAnnealing2-Random-r1h-p200	50	1.0
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p200	50	1.0
GA10-SimulatedAnnealing2-Random-r1h-p500	50	1.0
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p500	50	0.94
GA10-SimulatedAnnealing-Random-r15m-p10	50	None
GA10-SimulatedAnnealing-BestGuessRandom-r15m-p10	50	None
GA10-SimulatedAnnealing-EnhancedRandom-r15m-p10	50	None
GA10-SimulatedAnnealing-Random-r15m-p100	50	0.22
GA10-SimulatedAnnealing-BestGuessRandom-r15m-p100	50	None
GA10-SimulatedAnnealing-EnhancedRandom-r15m-p100	50	0.24
GA10-SimulatedAnnealing-Random-r15m-p1k	50	1.0
GA10-SimulatedAnnealing-BestGuessRandom-r15m-p1k	50	None
GA10-SimulatedAnnealing-EnhancedRandom-r15m-p1k	50	1.0
GA10-SimulatedAnnealing-Random-r15m-p10k	50	1.0
GA10-SimulatedAnnealing-BestGuessRandom-r15m-p10k	50	None
GA10-SimulatedAnnealing-EnhancedRandom-r15m-p10k	50	1.0
KOR140-SimulatedAnnealing-Random-r1h-p100	50	None
KOR140-SimulatedAnnealing-EnhancedRandom-r1h-p100	50	None
KOR140-SimulatedAnnealing-Random-r1h-p200	50	None
KOR140-SimulatedAnnealing-EnhancedRandom-r1h-p200	50	None
KOR140-SimulatedAnnealing-Random-r1h-p500	50	None
KOR140-SimulatedAnnealing-EnhancedRandom-r1h-p500	50	None
KOR140-SimulatedAnnealing-Random-r1h-p1k	50	None
KOR140-SimulatedAnnealing-EnhancedRandom-r1h-p1k	50	None
RCUC200-SimulatedAnnealing-Random-r1h-p100	8	None
RCUC200-SimulatedAnnealing-Random-r1h-p200	8	None
RCUC200-SimulatedAnnealing-Random-r1h-p500	8	0.125
TAI38-SimulatedAnnealing-EnhancedRandom-r1h-p20k	50	1.0

There are a few noticeble patterns. The first one is that the BestGuessRandom is clearly inferior. Its succes rate is always None.

The higher the PenaltyCost, the more likely we are to find a schedule with a high succes rate. This was expected.

It is also clear at a glance that if the local search finds a solution without unmet demands, it is very likely that nearly all runs will succeed in finding a solution without unmet demands. I have already hinted at this pattern in subsection 16.1 by stating that if 8 runs succeed, it is highly likely that all additional runs will also succeed.

Another pattern we can read from table 17 is that there does not appear to be a large difference between the RandomRampMoveGenerator and the EnhancedRandomRampMoveGenerator.

It appears to be the case that every time we try to do something clever, our efforts do not bear fruit.

However, there are also a couple of positive observations to be made. For example, the best score found by the GA10-SimulatedAnnealing2-Random-r1h-p200 experiment is 591252.55457. Remember from table 3 in section 3 that the global optimum for this instance is known to be 565825. This is about a 4.5% difference with respect to the global optimum, which was found by Kazarlis et al. [7] by brute-forcing the GA10 instance with a DP algorithm.

Another nice result is the schedule we've found for TAI38-SimulatedAnnealing-EnhancedRandom-r1h-p20k. The cost of this schedule is about $2.056 * 10^8$, which is slightly better than the best value of $2.138 * 10^8$ reported by Huang et al. in Table 6 of their paper [5].

This suggests that a solution which does no clever tricks other than making a detailed model of the problem space, has the potential to find solutions which only differ from the global optimum by a few percent.

17.1 Thoughts about the validity of the success-ratio as a metric

In the beginning of this section, we have seen that if we look at success rates, the local search method doesn't look very reassuring. However, that is only if we use the definition of an infeasible schedule in which we designate any schedule in which we have a shortage of energy that is strictly greater than 0 as infeasible.

Therefore I argue, that this might not be a good metric to determine the success or failure for the feasibility of a schedule, because the demands we are given, are forecasted values which rarely turn out exactly as they were forecasted.

I also remark that if the shortages are small enough, which they often are, it might be possible to make the schedule feasible with relatively small changes to the schedule. It is important to keep that in mind as well.

Fortunately, our approach allows for the easy bookkeeping of additional state variables which hold additional information about the state of the system. This is where the additional variables which model the operating reserve, positive spinning reserve and negative spinning reserve come into the picture.

To illustrate this point we need to take a look at an example schedule which is included in subsection 17.2.

There is no particular reason to choose this schedule over any other, other than that it is the first of a run with reasonable parameters. Yet this schedule exhibits the typical traits of many of the solutions a local search with this state-representation calculates.

In figure 17.2.1 we can see that the demand and dispatch match eachother almost perfectly except for one peak moment at time 11 at which this solution is 7 units of energy short.

If we then take a look at table 17.2.1, we can see that at time 11, there are still 4 units of Positive Spinning Reserve left which could bring the shortage at time 11 down to only 3 units.

If we then take a look at times 10 and 12 we see that there is more than enough Positive and Negative Spinning Reserve capacity to ramp a couple of units up or down to compensate for the 3 to 7 units of energy we are short at time 11 and this can probably also be done without making noticeble changes in the amount of power dispatched at times 10 and 12.

This brings us to the typical behaviour of a local search which uses simulated annealing in a problem which behaves as an interior point problem. That is: The search tends to move through the middle of the feasible region while staying some distance away from the boundaries. This behaviour is almost the polar opposite of algorithms using linear programming.

Table 9: Success ratios of various experiments

ExperimentName	BestScoreWithoutShortage	SuccesRate
A110-SimulatedAnnealing-EnhancedRandom-r1h-p100	4428953.969578764	1.0
GA10-HillClimberGreedy-Random-r15m-p10	None	None
GA10-HillClimberGreedy-BestGuessRandom-r15m-p10	None	None
GA10-HillClimberGreedy-EnhancedRandom-r15m-p10	None	None
GA10-HillClimberGreedy-Random-r15m-p100	629764.0471400002	0.18
GA10-HillClimberGreedy-BestGuessRandom-r15m-p100	None	None
GA10-HillClimberGreedy-EnhancedRandom-r15m-p100	625617.5357200002	0.16
GA10-HillClimberGreedy-Random-r15m-p1k	647551.2105900003	0.3
GA10-HillClimberGreedy-BestGuessRandom-r15m-p1k	None	None
GA10-HillClimberGreedy-EnhancedRandom-r15m-p1k	640272.7524700001	0.26
GA10-HillClimberGreedy-Random-r15m-p10k	647551.2105900003	0.32
GA10-HillClimberGreedy-BestGuessRandom-r15m-p10k	None	None
GA10-HillClimberGreedy-EnhancedRandom-r15m-p10k	637788.8110899999	0.26
GA10-SimulatedAnnealing2-Random-r1h-p50	None	None
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p50	None	None
GA10-SimulatedAnnealing2-Random-r1h-p75	None	None
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p75	None	None
GA10-SimulatedAnnealing2-Random-r1h-p100	591259.3546000001	0.14
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p100	591387.7778899994	0.02
GA10-SimulatedAnnealing2-Random-r1h-p200	591252.55457	1.0
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p200	591375.1288199999	1.0
GA10-SimulatedAnnealing2-Random-r1h-p500	609896.9320699999	1.0
GA10-SimulatedAnnealing2-EnhancedRandom-r1h-p500	609182.6644600001	0.94
GA10-SimulatedAnnealing-Random-r15m-p10	None	None
GA10-SimulatedAnnealing-BestGuessRandom-r15m-p10	None	None
GA10-SimulatedAnnealing-EnhancedRandom-r15m-p10	None	None
GA10-SimulatedAnnealing-Random-r15m-p100	593778.7539499997	0.22
GA10-SimulatedAnnealing-BestGuessRandom-r15m-p100	None	None
GA10-SimulatedAnnealing-EnhancedRandom-r15m-p100	592960.3534499996	0.24
GA10-SimulatedAnnealing-Random-r15m-p1k	630814.52726	1.0
GA10-SimulatedAnnealing-BestGuessRandom-r15m-p1k	None	None
GA10-SimulatedAnnealing-EnhancedRandom-r15m-p1k	633470.5980000002	1.0
GA10-SimulatedAnnealing-Random-r15m-p10k	649032.9910499996	1.0
GA10-SimulatedAnnealing-BestGuessRandom-r15m-p10k	None	None
GA10-SimulatedAnnealing-EnhancedRandom-r15m-p10k	649693.2709499998	1.0
KOR140-SimulatedAnnealing-Random-r1h-p100	None	None
KOR140-SimulatedAnnealing-EnhancedRandom-r1h-p100	None	None
KOR140-SimulatedAnnealing-Random-r1h-p200	None	None
KOR140-SimulatedAnnealing-EnhancedRandom-r1h-p200	None	None
KOR140-SimulatedAnnealing-Random-r1h-p500	None	None
KOR140-SimulatedAnnealing-EnhancedRandom-r1h-p500	None	None
KOR140-SimulatedAnnealing-Random-r1h-p1k	None	None
KOR140-SimulatedAnnealing-EnhancedRandom-r1h-p1k	None	None
RCUC200-SimulatedAnnealing-Random-r1h-p100	None	None
RCUC200-SimulatedAnnealing-Random-r1h-p200	None	None
RCUC200-SimulatedAnnealing-Random-r1h-p500	4.084530338869913E7	0.125
TAI38-SimulatedAnnealing-EnhancedRandom-r1h-p20k	2.0556055808769998E8	1.0

I also have to note that the schedules this algorithm generates, are very robust because they tend to have fairly large margins for the Positive and Negative Spinning Reserves, which means that at most points in time, there is a lot of room to anticipate fluctuating demand. This is a very desirable property if one has to consider the fluctuating output of renewable energy sources.

17.2 GA10-SimulatedAnnealing-EnhancedRandom-r15m-p100-0000

Table 10: Core statistics of GA10-SimulatedAnnealing-EnhancedRandom-r15m-p100-0000

Statistic	Value
total cost	593,542.35
total demand	27,100
total dispatch	27,094
total shortage	7
total surplus	1
active UnitStates	168
shortage instants	1
surplus instants	1

17.2.1 Timeline

Figure 10: Demand and dispatch of schedule GA10-SimulatedAnnealing-EnhancedRandom-r15m-p100-0000

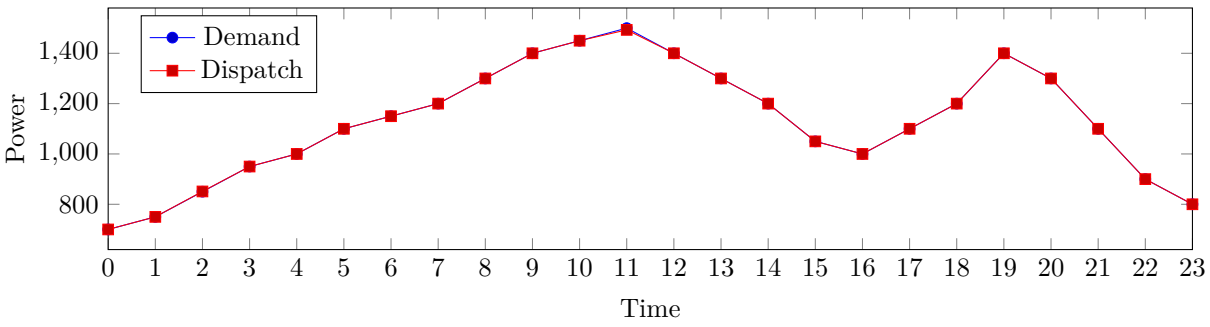


Figure 11: Shortage and surplus of schedule GA10-SimulatedAnnealing-EnhancedRandom-r15m-p100-0000

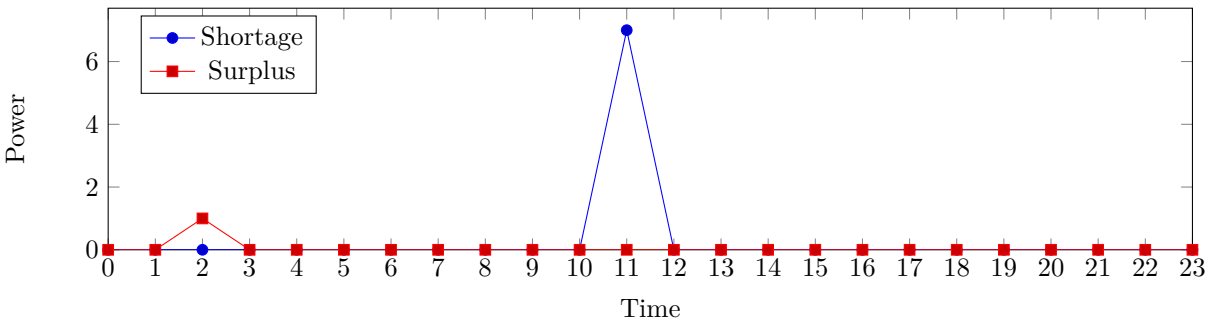


Figure 12: OR, PSR and NSR of schedule GA10-SimulatedAnnealing-EnhancedRandom-r15m-p100-0000

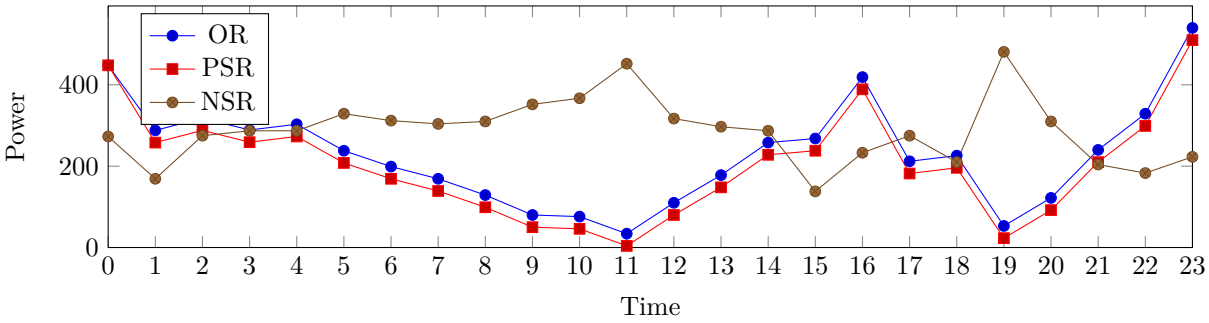


Table 11: Timeline of GA10-SimulatedAnnealing-EnhancedRandom-r15m-p100-0000

Time	Demand	Dispatch	Cost	Shortage	Surplus	OR	PSR	NSR
0	700	700	0	0	0	448	448	273
1	750	750	0	0	0	288	258	169
2	850	851	0	0	1	318	288	275
3	950	950	0	0	0	289	259	287
4	1,000	1,000	0	0	0	303	273	287
5	1,100	1,100	0	0	0	238	208	329
6	1,150	1,150	0	0	0	199	169	312
7	1,200	1,200	0	0	0	169	139	304
8	1,300	1,300	0	0	0	129	99	310
9	1,400	1,400	0	0	0	80	50	352
10	1,450	1,450	0	0	0	76	46	367
11	1,500	1,493	0	7	0	34	4	452
12	1,400	1,400	0	0	0	110	80	317
13	1,300	1,300	0	0	0	178	148	297
14	1,200	1,200	0	0	0	258	228	287
15	1,050	1,050	0	0	0	268	238	138
16	1,000	1,000	0	0	0	419	389	233
17	1,100	1,100	0	0	0	212	182	275
18	1,200	1,200	0	0	0	226	196	210
19	1,400	1,400	0	0	0	53	23	481
20	1,300	1,300	0	0	0	122	92	310
21	1,100	1,100	0	0	0	240	210	204
22	900	900	0	0	0	329	299	183
23	800	800	0	0	0	540	510	223

18 Conclusions

We have presented an integrated local search algorithm for the hybrid unit commitment and economic dispatch problem. At the time of writing, the author is not aware of the existance of another local search algorithm which can solve both problems simultaneously within the same procedure. Additionally, the algorithm can be modified and extended with relative ease, because of the state representation we have presented in this document. Because of the way the problem is formulated, it is also relatively easy to extend the current formulation such that network transmission flows could also be incorporated into the local search, if a future reader would so desire. The costs of the generated schedules are comparable to the costs found by other researchers. On top of that, we have succeeded in modeling the feasible region of the problem in a flexible, extensible and non-linear way which can be understood intuitively.

The author began this endavour because he suspected that it was possible that local search for the unit commitment problem had not yet received the scientific attention it deserved. While there is some pre-existing literature on this topic, this assumption turned out to be mostly correct.

19 Opportunities for future research

There are many paths we can take from this point on if one wants to research the subject of local search for the hybrid unit commitment and economic dispatch problem further. One of such opportunities lies in the search for new strategies for Neighbourhood exploration. Another opportunity lies in the potential to extend the state representation further such that it becomes possible to also pull the optimization of the load-flow problem into the local search procedure. Besides that, there is also potential to be found in the discovery of additional performance measures for the energy generating system by extending the tuples and update procedures of the state representation with additional variables which could directly model various other parameters, like for example pollution. Finally, one could also take it upon themselves to explore the feasibility of other acceptance strategies, like beam search or tabu-search for example. Or they could try to extend the formulation presented here such that it also becomes usable for the RTS45, Dispa-SET and Dispa-SET2 instances, which I have not touched upon due to their highly specific properties, like the presence of a transmission network with renewables and energy storage facilities.

20 Bibliography

References

[1] U.D. Annakkage, Thanisa Numnonda, and N.C. Pahalawaththa. Unit commitment by parallel simulated annealing. *Generation, Transmission and Distribution, IEE Proceedings-*, 142:595 – 600, 12 1995.

[2] Frangioni Antonio, Claudio Gentile, and Fabrizio Lacalandra. Tighter approximated milp formulations for unit commitment problems. *Power Systems, IEEE Transactions on*, 24:105 – 113, 03 2009.

[3] Alberto Borghetti, Frangioni Antonio, Fabrizio Lacalandra, Andrea Lodi, S. Martello, Carlo Alberto Nucci, and A. Trebbi. Lagrangian relaxation and tabu search approaches for the unit commitment problem. *2001 IEEE Porto Power Tech Proceedings*, 3:7 pp. vol.3, 02 2001.

[4] C. Christober Asir Rajan. Neural based tabu search method for solving unit commitment problem with cooling-banking constraints. *Serbian Journal of Electrical Engineering*, 6:57–74, 03 2009.

[5] Kun Yuan Huang, Hong Tzer Yang, and Ching Lien Huang. New thermal unit commitment approach using constraint logic programming. pages 176–185, 1 1997. Proceedings of the 1997 20th IEEE International Conference on Power Industry Computer Applications ; Conference date: 11-05-1997 Through 16-05-1997.

[6] Hidalgo Gonzalez I. Zucker A. Kavvadias, K. and S. Quoilin. Integrated modelling of future eu power and heat systems: The dispa-set v2.2 open-source model. *EU JRC Technical Report*, 2018.

[7] S. A. Kazarlis, A. G. Bakirtzis, and V. Petridis. A genetic algorithm solution to the unit commitment problem. *IEEE Transactions on Power Systems*, 11(1):83–92, Feb 1996.

[8] Whei-Min Lin, Fu-Sheng Cheng, and Ming-Tong Tsay. "an improved tabu search for economic dispatch with multiple minima,". *Power Systems, IEEE Transactions on*, 17:108 – 112, 03 2002.

[9] A.H. Mantawy, Youssef Abdel-Magid, and Shokri Selim. A simulated annealing algorithm for unit commitment. *Power Systems, IEEE Transactions on*, 13:197 – 204, 03 1998.

[10] A.H. Mantawy, Soliman Soliman, and Mo El-Hawary. A new tabu search algorithm for the long-term hydro scheduling problem. pages 29 – 34, 02 2002.

[11] H. Mori and O. Matsuzaki. Embedding the priority list into tabu search for unit commitment. In *2001 IEEE Power Engineering Society Winter Meeting. Conference Proceedings (Cat. No.01CH37194)*, volume 3, pages 1067–1072 vol.3, Jan 2001.

[12] S.O. Orero and M.R. Irving. Large scale unit commitment using a hybrid genetic algorithm. *International Journal of Electrical Power & Energy Systems*, 19(1):45 – 55, 1997.

[13] N. P. Padhy. Unit commitment-a bibliographical survey. *IEEE Transactions on Power Systems*, 19(2):1196–1205, May 2004.

[14] Jong-Bae Park, Yun-Won Jeong, Joong-Rin Shin, and Kwang Lee. An improved particle swarm optimization for nonconvex economic dispatch problems. *Power Systems, IEEE Transactions on*, 25:156 – 166, 03 2010.

[15] B. Saravanan, Siddharth Das, Surbhi Sikri, and D. P. Kothari. A solution to the unit commitment problem—a review. *Frontiers in Energy*, 7(2):223–236, Jun 2013.

[16] Matthias Silbernagl, Matthias Huber, and René Brandenberg. Improving Accuracy and Efficiency of Start-up Cost Formulations in MIP Unit Commitment by Modeling Power Plant Temperatures. *arXiv e-prints*, page arXiv:1408.2644, Aug 2014.

[17] D. N. Simopoulos, S. D. Kavatza, and C. D. Vournas. Reliability constrained unit commitment using simulated annealing. *IEEE Transactions on Power Systems*, 21(4):1699–1706, Nov 2006.

[18] Dimitris Simopoulos, Stavroula Kavatza, and Costas Vournas. Unit commitment by an enhanced simulated annealing algorithm. *Power Systems, IEEE Transactions on*, 21:68 – 76, 03 2006.

[19] W. van Ackooij, I. Danti Lopez, A. Frangioni, F. Lacalandra, and M. Tahanan. Large-scale unit commitment under uncertainty: an updated literature survey. *Annals of Operations Research*, 271(1):11–85, Dec 2018.

[20] T. A. A. Victoire and A. E. Jeyakumar. Unit commitment by a tabu-search-based hybrid-optimisation technique. *IEE Proceedings - Generation, Transmission and Distribution*, 152(4):563–574, July 2005.

[21] Rogier Hans Wuijts, Marjan van den Akker, and Machteld van den Broek. An improved algorithm for single-unit commitment with ramping limits. *Electric Power Systems Research*, 190:106720, 2021.

[22] F. Zhuang and F. D. Galiana. Unit commitment by simulated annealing. *IEEE Transactions on Power Systems*, 5(1):311–318, Feb 1990.

21 How to contact the author

You can get in touch with the author by sending an e-mail to: unit [dot] commitment [at] hageraats [dot] org. You need to replace the [dot] with a . and the [at] with an @ symbol to obtain a valid e-mail address.