# Scheduling Algorithms for the Smart Grid

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

**genehmigte**

**Dissertation**

von

**Lukas Franz Josef Barth**

| | |
|---|---|
| Tag der mündlichen Prüfung: | 15.5.2020 |
| Erste Referentin: | Prof. Dr. Dorothea Wagner |
| Zweiter Referent: | Prof. Dr. Veit Hagenmeyer |

# Danksagung

Diese Dissertation ist der – vorläufige – Höhepunkt meiner Reise durch die Informatik, die im Herbst 2009 begann. Eine meiner ersten Vorlesungen hieß „Theoretische Grundlagen der Informatik“ und wurde von meiner heutigen Doktormutter Dorothea Wagner gehalten. Die Vorlesung hat mich damals für die theoretische Informatik begeistert, und an ihrem Ende machte Dorothea den versammelten Studierenden ein Angebot: Gerne würde sie Abschlussarbeiten auch ins Ausland vermitteln. Als ich zwei Jahre später in ihrem Büro saß, stand das Angebot offenbar noch, und sie (und mein späterer Kollege Ignaz Rutter) machten es möglich, dass ich ein Semester lang bei einem Aufenthalt bei und als Teil der Arbeitsgruppe von Stephen Kobourov an der University of Arizona „Forschungsluft schnuppern“ konnte. In meinem darauf folgenden Masterstudium habe ich die Vorlesungen von Dorotheas Lehrstuhl mehr oder weniger „durchgespielt“, und im Jahr 2016 hat Dorothea mir angeboten, in ihrer Gruppe zu promovieren. Ich möchte allen, die meinen Weg in die Forschung so geebnet haben – allen voraus Dorothea Wagner – dafür danken.

Als Teil von Dorotheas Gruppe durfte ich mit großartigen Kolleginnen und Kollegen zusammenarbeiten, denen ich (in alphabetischer Reihenfolge) dafür danken möchte: Moritz Baum, Guido Brückner, Valentin Buchhold, Lars Gottesbüren, Sascha Gritzbach, Michael Hamann, Paul Jungeblut, Tamara Mchedlidze, Benjamin Niedermann, Martin Nöllenburg, Roman Prutkin, Marcel Radermacher, Ignaz Rutter, Jonas Sauer, Ben Strasser, Torsten Ueckerdt, Franziska Wegner, Matthias Wolf, Tim Zeitz und Tobias Zündorf. Die Zeit am Institut besteht ja zum Glück nicht nur aus Arbeit, und so danke ich euch allen neben der fachlichen Zusammenarbeit auch für große Emotionen bei epischen Kicker-Matches, für nervenzerreißende Damenwasserballweltmeisterschaften im legendären Gernsbach, für unzählige gute Gespräche und das warme Gefühl, nicht der einzige zu sein, der manchmal an LaTeX, der Forschung oder Studierenden verzweifelt. Es ist ein offenes Geheimnis, dass die eigentlichen Chefs einer jeden Arbeitsgruppe die Sekretärinnen (und Sekretäre) und Systemadministratoren (und Systemadministratorinnen) sind. Ich danke Lilian Beckert, Isabelle Junge, Ralf Kölmel, Laurette Lauffer, Tanja Wehrmann für all die Unterstützung beim Kampf mit der Technik und der Bürokratie.

Als Teil der ersten Generation des Graduiertenkollegs „Energy Status Data“ hatte ich eine zweite Gruppe von Kolleginnen und Kollegen, von denen ich viel gelernt habe. Allen diesen gilt mein Dank für die fruchtbare interdisziplinäre Zusammenarbeit, insbesondere meiner „Selbstorganisierten Studiengruppe“ bestehend aus Nicole Ludwig,

Esther Mengelkamp und Philipp Staudt. Für die Ermöglichung des Graduiertenkollegs möchte ich auch Herr Böhm danken.

Frei nach Karl Valentin gilt für eine Dissertation: „Promovieren ist schön, macht aber viel Arbeit." Bei dieser Dissertation haben mich korrekturlesenderweise Sascha Gritzbach, Matthias Wolf und Sophie Fendel unterstützt. Vielen Dank dafür, dass ihr euch das angetan habt. Vielen Dank auch an Marcel Radermacher, Tobias Zündorf und Valentin Buchhold, denen ich jeweils viel Zeit damit gestohlen habe, Fragen der Typographie, der richtigen Darstellung von Diagrammen oder der Schreibweise von Fachbegriffen zu erörtern. Den Gutachtern dieser Dissertation, Dorothea Wagner und Veit Hagenmeyer, möchte ich ebenfalls danken.

Abschließend möchte ich denen danken, die außerhalb der Universität dafür gesorgt haben, dass ich heute diese Dissertation einreichen kann. Danke an meine Eltern, die mir nicht nur das Studium ermöglicht, sondern die auch schon viel früher dafür gesorgt haben, dass ich allen meinen Interessen nachgehen konnte. Und nicht zuletzt besonderen Dank an Sophie, dafür dass du die Belastung, die meine Promotion auch für uns beide war, immer mitgetragen und mich zu jeder Zeit unterstützt hast.

# Deutsche Zusammenfassung

Diese Dissertation behandelt Themen rund um das Scheduling von elektrischen Lasten mit einem Bezug zu zukünftigen Smart Grids.

Die Problematik, Prozesse, die mit einem (elektrischen) Verbrauch einhergehen, zu schedulen, ergibt sich im Kontext von Smart Grids insbesondere bei der sogenannten *Demand Response*. Zukünftige Energiesysteme werden hauptsächlich auf erneuerbare Stromquellen wie Solar- oder Windenergie setzen, die im Gegensatz zu herkömmlichen Energieträgern wie Kohle- oder Kernenergie in ihrer Erzeugung nicht steuerbar sind. Da in einem Energienetz die Erzeugung aber zu jedem Zeitpunkt dem Verbrauch entsprechen muss, können diese erneuerbaren Erzeuger unsere bisherigen Kraftwerke nicht ohne weitere Änderungen ersetzen. Zu den Strategien, um damit umzugehen, gehört neben dem Ausbau von Speichern und Übertragungsnetzen auch die Demand Response (DR). Demand Response zielt darauf ab, den Verbrauch kurzfristig so zu verändern, dass er besser zur Erzeugung passt. Die naheliegendste Veränderung ist, Prozesse, die zeitlich flexibel sind, zu verschieben. Einen besonderen Fokus legt diese Arbeit dabei auf Verfahren, die *Peak Shaving* zum Ziel haben, das heißt die Reduktion von Verbrauchsspitzen. Dies ist insofern nützlich, als Kapazitäten im Energiesystem, sowohl was die mögliche Erzeugung als auch was die Übertragung angeht, auf diese Spitzenlasten ausgelegt werden müssen.

Diese Arbeit befindet sich an der Schnittstelle mehrerer weitläufiger Forschungsfelder: Zum einen ist der Bereich der Demand Response derzeit sehr aktiv, einen Überblick gibt beispielsweise Siano [Sia14]. Was den Scheduling-Aspekt angeht, ähnelt vor allem das Project Scheduling aus dem Bereich des Operations Research den hier behandelten Problemstellungen. Einen Überblick hierüber geben Brucker und Knust [BK12] und Węglarz [Węg99]. Insbesondere das *Time-Constrained Project Scheduling Problem* (TCPSP), beziehungsweise das spezifischere *Resource Acquirement Cost Problem* (RACP) (auch als *Resource Investment Problem* (RIP) bekannt) kommen den hier betrachteten Problemen nahe.

Ein großer Teil der Optimierung zu den genannten Problemen, insbesondere auch im Bereich der Demand Response, findet mittels gemischt-ganzzahlig linearer Programme (engl. *Mixed-Integer Linear Programs*, MIPs) statt. Deshalb, und weil MIPs ein wertvoller Benchmark für eigene Algorithmen sein können, befasst sich der erste Teil dieser Dissertation mit Themen rund um MIP-Formulierungen für solche Scheduling-Probleme. Zunächst wird in Kapitel 3 ein MIP-Framework vorgestellt, das die Modelle zahlreicher anderer Arbeiten vereint. Diese Veröffentlichung bietet einen Überblick über die in der Literatur modellierten Merkmale flexibler elektrischer

Lasten. Darauf baut ein MIP-Framework auf, das fast alle dieser Merkmale vereint. Mit einer experimentellen Evaluation wird gezeigt, dass die resultierenden Modelle trotz der Mächtigkeit des Frameworks für realistische Instanzgrößen immer noch in vertretbarer Zeit optimiert werden können.

Bei diesem Modell — ebenso wie bei den meisten MIP-Modellen in der Literatur — wird davon ausgegangen, dass für jeden Prozess ein Maß an Flexibilität Teil des Modells, also der Eingabe, ist. Das hat den Nachteil, dass die zeitliche Flexibilität jedes einzelnen Prozesses von z. B. einer Industrieanlage im Voraus bekannt sein muss. Dies hat sich als eine unrealistische Annahme herausgestellt. In Hinblick darauf stellt Kapitel 4 ein weiteres Modell-Framework vor. Anstelle einer zeitlichen Flexibilität pro Prozess bekommen diese neuen MIPs ein einziges, globales Maß für Flexibilität als Nebenbedingung. Die Lösung eines solchen MIPs kann als eine Empfehlung dafür aufgefasst werden, welche Prozesse bei einer zeitlichen Verschiebung den größten Nutzen erzielen würden.

Beide bisher vorgestellten MIP-Modelle haben gemeinsam, dass sie auf einer zeitexpandierten Formulierung basieren, deren Komplexität mit dem zeitlichen Horizont der Probleminstanz wächst. Mit diesem Problem befasst sich die Umformulierung in Kapitel 5, die zu effizienter optimierbaren MIP-Modellen führt. Kapitel 6 rundet diesen ersten Teil der Arbeit mit einem Benchmark-Datensatz für Scheduling-Probleme rund um Demand Response ab.

Neben der modellgetriebenen Optimierung stellt diese Arbeit in ihrem zweiten Teil zwei Heuristiken vor, die zur Optimierung der durch die Modelle spezifizierten Probleme geeignet sind. Eine erste Heuristik, die Resource Utlitization Scheduling Heuristic (RUSH), vorgestellt in Kapitel 7, ist darauf ausgelegt, auch auf großen Instanzen sehr schnell zu akzeptablen Ergebnissen zu gelangen. Dazu wird die Ressourcennutzung in diskrete Niveaus zerlegt und es wird in einer effizienten Datenstruktur vorgehalten, in welchen Zeitintervallen welche Niveaus der Ressourcennutzung erreicht werden. Mithilfe dieser Information lässt sich für jeden Prozess sehr effizient berechnen, an welche Zeitpunkte er verschoben werden könnte, um die Ressourcennutzung zu reduzieren. Dieser Vorgang wird iterativ wiederholt und so die Lastspitze gesenkt.

Die zweite, in Kapitel 8 vorgestellte Heuristik fasst das Scheduling-Problem teilweise als Graphproblem auf: Ist ein Abhängigkeitsgraph für ein Scheduling-Problem gegeben, so ergibt sich ein "schnellstmöglicher" Schedule, in dem jeder Prozess so früh wie vom Abhängigkeitsgraphen erlaubt gestartet wird. Dies erlaubt es, einen Graphen mit einem Schedule zu assoziieren. Es lässt sich zeigen, dass zu jedem gegebenen Scheduling-Problem und Abhängigkeitsgraphen ein "augmentierter" Graph, das heißt ein Supergraph des ursprünglichen Abhängigkeitsgraphen, existiert, sodass dessen assoziierter Schedule eine optimale Lösung des ursprünglichen Problems ist. Die Vorgehensweise der Heuristik zielt darauf ab, diesen Graphen anzunähern. Dazu werden iterativ neue Kanten in den Abhängigkeitsgraphen eingefügt. Eine neue Kante

im Abhängigkeitsgraphen trennt zwei Prozesse zeitlich voneinander, kann also dazu genutzt werden, um die Spitzenlast zu senken, indem zwei Prozesse, die beide parallel zur Zeit der Spitzenlast ausgeführt werden, voneinander getrennt werden.

Beide Heuristiken werden intensiv experimentell evaluiert. Dabei werden auf Echtweltdaten basierende Probleminstanzen generiert und verwendet. Die Ergebnisse werden verglichen mit Ergebnissen aus der MIP-Optimierung, aber auch mit denen eines anderen aktuellen Optimierungsalgorithmus für dieses Problem, GRASP [Pet+14]. Es wird gezeigt, dass die neuen Heuristiken bessere Ergebnisse als die bestehenden Verfahren berechnen bzw. Ergebnisse für Instanzen berechnen, auf denen die MIP-Modelle nicht mehr praktikabel sind. Die aus Echtweltdaten generierten Test-Datensätze werden als Benchmarks ebenfalls veröffentlicht.

Der letzte Teil der Dissertation betrachtet einige algorithmische Grundlagen für Scheduling-Algorithmen. Ein Thema, dem sich die Arbeit intensiv widmet, sind Datenstrukturen, die für Scheduling-Algorithmen von besonderer Wichtigkeit sind. Hierbei handelt es sich um besondere Formen von Suchbäumen, den dynamischen Segmentbäumen. Da deren Effizienz maßgeblich von den zugrundeliegenden selbstbalancierenden binären Suchbäumen abhängt, evaluiert Kapitel 9 verschiedene Varianten, unter anderem klassische Rot-schwarz-Bäume oder gewichtsbalancierte Bäume, aber auch weniger bekannte Varianten wie Zip Trees. Insbesondere Letztere erweisen sich als eine gute Wahl zum Engineering der dynamischen Segmentbäume. Da die evaluierten gewichtsbalancierten Bäume ebenfalls einige Punkte zum Tuning anbieten, widmet sich Kapitel 10 deren Engineering. Eine spannende Einsicht ist hier, dass die klassische Parametrisierung von gewichtsbalancierten Bäumen oft keine gute Wahl ist.

Ein weiteres Ergebnis dieses Teils der Arbeit, vorgestellt in Kapitel 11, ist die Einsicht, dass bei Scheduling-Problemen der Art, wie sie in dieser Dissertation betrachtet werden, die lokale Komplexität ausschlaggebend ist. Der zeitliche Horizont eines solchen Problems ist also bezüglich der Komplexität zweitrangig, solange die Komplexität zu jedem möglichen Zeitpunkt — das heißt insbesondere die Anzahl der Prozesse, die möglicherweise zu diesem Zeitpunkt ausgeführt werden könnten — nicht zu hoch ist. Zu dieser Einsicht gelangt man mittels eines einfachen, exakten Exponentialzeit-Algorithmus, dessen Zeitkomplexität polynomiell wird, sobald man die erwähnte lokale Komplexität beschränkt.

# Contents

# 1 Introduction

Energy systems all over the world are in upheaval. Germany with its *Energiewende* is one of the pioneers of transitioning an energy-hungry society from traditional, fossil-fuel based electricity generation to cleaner, renewable energies. Figure 1.1 shows the development of the primary energy carriers in the German electricity generation from 1990 to today. Starting at a meager 6.6% of the energy mix in 2000, the Energiewende has boosted the share of renewables in the German electricity mix to 40.1% in 2019. At the same time, the reliance on coal has been drastically reduced. In recent years, similar transitions have picked up speed in other places, for example in China and India — so much speed in fact that one could argue that these countries have overtaken Germany. For example, with a total of 200 GW of installed generation capacity in 2017 [ISE20], Germany added just 1.66 GW of installed photovoltaics in that year [Fed19]. In the same year, China, with an installed total capacity of 1750 GW [Ren18], expanded its photovoltaics capacity by 53 GW [den19]. Even in countries critical of renewable energy sources such as the United States of America with its intensely coal- and gas-focused electricity generation, the speed of solar parks being built is ever increasing.

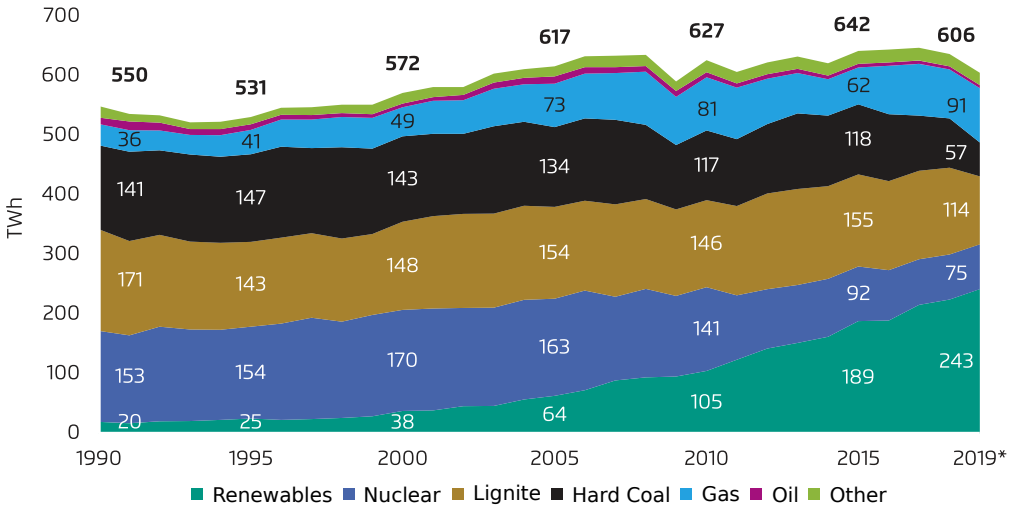



**Figure 1.1:** Development of primary energy carriers in German electricity generation from 1990 to today. Source: Agora Energiewende (2020). (* preliminary data)

In 2017, the U.S.A. installed a total of 10.6*GW* of photovoltaics, which amounts to 30% of newly installed generation capacity [Ass17].

Electrical grids are behemoths. The European so-called *Continental Synchronous Area* alone connects more than 400 million customers to about 700 gigawatts of generator power. It is a single electrical grid spanning the area from Portugal to Turkey and from Denmark to Algeria, with plans to expand into the Baltics and larger parts of northern Africa. Tightly coupled to it via high-voltage direct current connections are other large networks such as the British, Norwegian, Swedish and Finnish electrical grid. On the other side of the globe, China is pushing towards unification of its two currently existing wide-area electrical grids, with a combined generation capacity of more than 1.8 terawatts [Ren18].

Electrical grids are fragile. In November 2006, the routine shutdown of a single transmission line in Germany — planned well in advance — caused a chain reaction of line overloads, protective shutdowns and finally the split of the continental synchronous area into three separate areas, cutting the European electrical grid diagonally across Germany and Austria. The final report on the incident by the European Network of Transmission System Operators for Electricity (ENTSO-E) [Tra06] lists grave consequences: The effects of the incident reached as far as Morocco, which had to emergency-disconnect 300 megawatts of loads, causing a blackout for many consumers. In total, more than 17 gigawatts of loads had to be disconnected from the grid, especially in western Europe. The reasons for the failure are manifold and reach from last-minute changes to the shutdown schedule to the misconfiguration of line load limits. One major contributing factor, however, was the fact that at the moment the first line was intentionally shut down, it carried ten gigawatts of wind-produced electricity from the north of Germany into the southern parts of the grid — a situation that becomes more and more frequent with an increasing share of renewables.

Electrical grids are challenged. They are traditionally designed under the assumption that the amount of generated power can be tightly controlled to match the demand. On shorter time scales, the massive angular momentum of hundreds of turbines, each rotating one hundred tons of steel at 3000 revolutions per minute, is able to gracefully absorb any suddenly occurring demand spikes. Another assumption is that electricity flows from few central points — the power plants — towards the consumers. All three assumptions are challenged in electrical systems primarily based on solar and wind generation: Neither of them can be arbitrarily controlled, making the infamous *Dunkelflaute* the dread of German electrical systems engineers. At the same time, too much renewable generation can also cause problems, as Figure 1.2 illustrates. It shows the electricity generation, disaggregated into conventional and renewable generation, during a week in June 2019. It also displays the resulting day-ahead price for a megawatt-hour of electricity. On the 8th of June, when large amounts of solar energy were available and consumption was low, the market price dropped

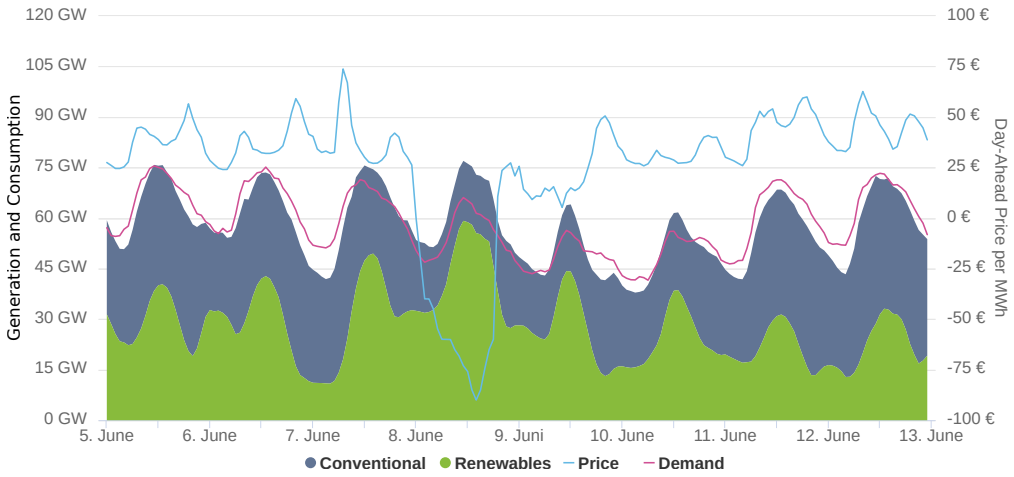



**Figure 1.2:** Electricity prices at the German day-ahead spot market and the corresponding mix, from 5.6.2020 to 16.6.2020. Source: Agora Energiewende (2020).

to about −90 €/MWh. Electricity producers were paying a considerable amount of money for their energy to be consumed. This happened because first curtailing the remaining conventional generation and ramping it back up shortly after would have been prohibitively expensive. This situation arises more and more frequently in the German electricity market — for a total of 134 hours distributed over 25 days in the year 2018 [Bun19]. Regarding the amount of angular momentum that can be used to instantly absorb abrupt changes in demand, even though wind turbines of course do have a rotating shaft, rotating mass is drastically reduced. Finally, generation capacity in future energy systems will connect to the transmission grid in a multitude of points instead of few central points. Not only are solar fields and wind turbines much more dispersed than traditional power plants, but consumers have also started to install solar panels on their roofs, scattering generation even further.

In the face of these challenges, the scientific, commercial and political communities involved with energy systems are developing mitigation strategies. The most prominent categories of mitigation strategies are *network expansion*, energy *storage*, *sector coupling* and *demand side management*. Expanding the electrical grid, i.e., building new transmission lines and reinforcing existing ones, helps to smooth out renewable generation at a continental scale. Network expansion achieves in the spatial dimension, what storage does along the dimension of time: Smoothing generation and consumption over time addresses the problem of renewables, especially solar energy, producing electricity at a schedule which only partially fits the demand curve. Sector coupling can also serve as a buffering mechanism by flexibly converting energy between electricity, gas and heat whenever one of the sectors has a surplus. However, by combining

power-to-gas technology with the gas distribution network it is even thinkable to exploit sector coupling to alleviate congestion in the electricity transmission networks.

The topic of this thesis fits into the category of demand side management (DSM). The term is used for a wide range of techniques and policies aimed at influencing when, where, and how much energy is consumed. The aim of DSM usually is to better match the demand to the (renewable) generation available at any point in time. DSM measures can be loosely grouped into two categories: The first aims at improving the *amount* of energy consumed, for example by increasing the energy efficiency of devices. The second tries to control *when* energy is consumed, i.e., to control the power demand. This latter category is called *demand response* (DR). The United States Department of Energy [US 06] defines demand response as "changes in electric usage by end-use customers from their normal consumption patterns [...] to induce lower electricity use at times of high wholesale market prices or when system reliability is jeopardized". The scheduling techniques presented in this thesis are intended to facilitate demand response.

demand response

The field surrounding DR is currently very active. Siano [Sia14] provides a general, broad survey. Deng et al. [Den+15] present a survey with a focus on mathematical modeling, while Vardakas et al. [VZV15] provide a general overview of methods used for demand side management. Haider et al. [HSE16] review the literature focused on demand side management involving residential customers. Pudjianto and Strbac [PS17] as well as Strbac [Str08] consider the possible benefits from and challenges to the implementation of DR.

Simple techniques for DR include time-of-use electricity tariffs or dynamic tariffs with price signals communicated e.g. via internet, which try to incentivize the consumer to consume power when plenty of generation is available. In fact, traditional night storage heaters can be seen as a demand response scheme, encouraging consumers to use electricity at night using a cheaper night tariff.

On the other hand, approaches to DR that do not rely on consumers controlling their devices themselves envision central coordinators to communicate with consumers' devices via a smart grid. This communication enables consumers to transfer a certain amount of control over their devices to the coordinator, and allows the coordinator to schedule consumption to optimally fit the available generation. Techniques involving such a central coordinator are usually called *centralized* or *direct* demand response. Other techniques, such as distributed-computing approaches based on game theoretic techniques, for example by Barbato et al. [Bar+15], range in the gray area between centralized and purely consumer-reliant DR. Just like DR based on time-of-use tariffs, centralized DR based on price signals has been evaluated in reality, for example by Larsen et al. [Lar+17] in a setting with 1900 households on the island of Bornholm. In this setting, the peak demand could be reduced by up to 27%.

Centralized approaches require the controller to frequently solve complex scheduling problems. Depending on the goal of the controller, the required speed of optimization can be substantial. One possible use for centralized DR is to provide *balancing energy*. Balancing energy is the term used for the amount of idle electricity generators held in reserve to react to fast changes in electricity demand. This reserve generation capacity is traded on a market similar to the market used for the actual electricity generation. The German energy system knows three forms of balancing energy: a primary reserve that must respond withing 30 seconds, a secondary reserve required to be fully available in five minutes, and a tertiary reserve operating on a scale of between five to 60 minutes. Thus, to participate in any of these markets, the controlled devices must be rapidly rescheduled. But even for plain marked-based electricity trading, fast scheduling would be beneficial — as an example, the German intra-day market operates on a fifteen-minute basis.

The scheduling problems arising from these scenarios are close to project scheduling problems, a well-studied area of research in operations research. In his survey of the field, Węglarz [Węg99, page ix] defines project scheduling problems as "problems of allocating scarce resources over time to perform a given set of activities". In smart grid scheduling problems, the most obvious scarce resource is electricity, however, incorporating more resources can be beneficial. If sector coupling is taken into consideration, gas and heat can be additional resources. Also, capturing (electric) storage such as batteries in scheduling problems can sometimes be done via additional resources. Brucker and Knust [BK12] provide another broad overview of the field, as do Demeulemeester and Herroelen [DH02]. Research into optimization of project scheduling problems is plentiful. On the one hand, model-based optimization is widely used, and many modeling techniques have been proposed, as for example discrete-time formulations by Pritsker et al. [PWW69], Rieck et al. [RZG12], or Naber and Kolisch [NK14], two event-based approaches by Koné et al. [Kon+11], or flow-based modeling techniques by Artigues et al. [AMR03]. Another very popular optimization technique are genetic (respective evolutionary) algorithms, as for example presented by Ballestín [Bal07], Ranjbar et al. [Ran13] or Ponsz-Tienda et al. [Pon+13].

Aside from classical scheduling research, the field of energy informatics has yielded results in the field of optimization techniques applicable to demand response, too. A survey of scheduling techniques with an explicit focus on demand response is not known to the authors. However, scheduling time-flexible electrical loads is closely related to scheduling virtual power plants (which can incorporate flexible loads) and certain optimization problems in the operation in microgrids. In that field, Nosratabadi et al. [NHG17] provide a survey on scheduling techniques. However, we are unsure about the quality of that survey; for example, the authors treat centralized DR (called *direct load control* in their terminology) as a form of incentive-based DSM (see [NHG17, Section 10]), which blurs the boundaries between different forms of DSM.

Much effort has gone into optimization of smart buildings; for example, Allerding et al. [All+12] use evolutionary algorithms to optimize smart homes, Bradac et al. [BKF15] use mixed-integer linear programming and Mahmood et al. [Mah+16] use particle swarm optimization. A multitude of other studies could be named; for a survey, see Haider et al. [HSE16]. Another active cluster of research focuses on the optimization of processes in industry, often with a focus on specific industrial plants. Ashok and Banerjee [AB00] look at a fertilizer plant, Ashok [Ash06] optimizes a steel plant and Eissa [Eis11] evaluates a time-of-use tariff scheme for industrial consumers in Saudi Arabia. Finn and Fitzpatrick [FF14] evaluate real-time pricing for industrial customers. Uncoupled from the actual scheduling technique, Merkert et al. [Mer+15] provide an overview over challenges and opportunities for industrial consumers.

However, there is little research explicitly developing scheduling heuristics for an application in demand response. In most cases, model-based optimization, i.e., mathematical programming, is used. The survey by Nosratabadi et al. lists a total of 57 publications related to the virtual power plant problem (see [NHG17, Table 6]). Out of these, 51 use mathematical programming for optimization. Of the remaining six, one uses a game-theoretic approach, i.e., distributed computation, and one does not formally optimize at all but uses simulation based on MATLAB. The remaining four papers each employ well-established metaheuristics: Twice, genetic algorithms are used, one paper uses particle swarm optimization, and one uses a greedy randomized adaptive search procedure (GRASP). We use this latter work by Petersen et al. [Pet+14] in Chapter 8 as the competitor algorithm for our own heuristic.

It seems fair to conclude from this that looking into heuristics tailored specifically to smart grid scheduling problems is a worthwhile undertaking — if one takes the extra step of experimentally evaluating them and showing their applicability, and ideally: superiority, to previous heuristics. Furthermore sorting, assessing and unifying the bulk of mathematical-programming-based techniques could contribute to a clearer field of research.

This thesis strives to contribute towards both these goals.

## Thesis Outline

The contributions in this thesis are divided into three parts. We start by investigating ways to describe and mathematically model demand side flexibility in Part I. We thoroughly evaluate the suggested modeling techniques and compare them to models from literature. In Part II, we present two heuristics that can be employed to optimize large-scale scenarios, and experimentally demonstrate the effectiveness of the two heuristics. Finally, in Part III, we dive deeper into some algorithmic foundations of scheduling. We describe data structures used to facilitate some operations in the

heuristics mentioned above, and we look into the fixed-parameter tractability of smart grid scheduling.

### Part I: Modeling

The first part is concerned with modeling scheduling problems that capture smart grid scheduling, and using model-based optimization, while always trying to manage the balancing act of capturing enough aspects in our models for them to be employed in reality, and keeping the optimization complexity of the models manageable.

**Chapter 3: A Comprehensive Modeling Framework for Demand Side Flexibility.** We start by creating a mixed-integer linear programming framework that unites the modeling power of several other mixed-integer linear programs from literature. In an experimental evaluation, we demonstrate that using the resulting models for optimization is still reasonable.

**Chapter 4: Exploring the Benefits of Flexibilization in Industrial Contexts.** Based on the modeling technique from Chapter 3, we present a modified modeling technique that is able to optimize electricity usage with less required domain knowledge: Instead of requiring information about the flexibility of the individual processes to be scheduled, a global measure of flexibility is specified in this model. The output of the model can be interpreted as a suggestion concerning the processes that should be made flexible. A domain expert can use repeated re-optimization of the model with changed parameters to explore how and with how much effort electricity usage can be improved.

**Chapter 5: An Order-Based Model for the Resource Acquirement Cost Problem.** We complement the presentation of the two modeling frameworks from chapters 3 and 4 with an investigation into how to speed up the underlying modeling technique. We suggest a new modeling technique that can be used as a drop-in replacement of the previous models, and demonstrate that the resulting models can be optimized more efficiently.

**Chapter 6: Industrial Demand Side Flexibility: A Benchmark Data Set.** Considering the multitude of published optimization methods for scheduling problems in smart grids, having a way of comparing these methods against each other is imperative. Experimental evaluation plays the most important role in this case, and good experiments require good input data on which the experiments can be footed. We present a set of instances for scheduling problems that can serve as a benchmark for various algorithms. Our analysis of the presented instances illustrates that the

instances are neither too hard nor too easy, so that one can see interesting differences between optimization methods using our instance set.

### Part II: Heuristics

In the second part, we present two heuristic algorithms to optimize the smart grid scheduling problems modeled in Part I. The optimization objective throughout this part is peak shaving, i.e., the minimization of the demand peak. Both heuristics are based on a fast iterative process, providing a simple time-quality tradeoff.

**Chapter 7: Exploiting Flexibility in Smart Grids at Scale.**
The first heuristic, RUSH, is based on the simple idea that repeatedly moving jobs which currently contribute to the peak demand will result in the reduction of the peak value. The main focus of this heuristic is speed, i.e., to very quickly achieve results with an acceptable quality.

**Chapter 8: Shaving Peaks by Augmenting the Dependency Graph.**
The second presented heuristic, SWAG, approaches the scheduling problem as a graph problem, using a dependency graph on the processes as representation of a possible solution. Inserting new dependencies, i.e., edges, into the graph, allows the algorithm to separate two processes that were running concurrently. Doing this repeatedly with jobs that participate in the peak allows us to reduce the peak demand by solely manipulating the dependency graph.

### Part III: Algorithmic Foundations

The third part of the thesis examines several more fundamental aspects of scheduling algorithms, such as supporting data structures and computational complexity.

**Chapter 9: Efficiently Finding Peaks Using Dynamic Segment Trees.**
The key to success for both presented heuristics in Part II is the availability of fast primitive operations. One such operation is the determination of the time interval during which the peak demand occurs in a given schedule, as well as the magnitude of the peak demand. We present a variation of the dynamic segment tree, a data structure that can be used to answer these queries very efficiently. We tune the dynamic segment tree by swapping out its underlying tree and changing the way annotations work.

**Chapter 10: Engineering Top-Down Weight-Balanced Trees.**
Weight-balanced trees are a form of balancing binary search trees, which can be used as a basis for the aforementioned dynamic segment trees. With weight-balanced trees, rebalancing can be done either in a top-down or in a bottom-up fashion. Both

rebalancing algorithms additionally have a set of parameters that control their inner workings. We perform a thorough investigation of the effects of parameter choices and arrive at the conclusion that unconventional parameter choices lead to the best performance, and that top-down rebalancing is the superior method.

**Chapter 11: TCPSP is Fixed-Parameter Tractable in a Local Measure .**
A theoretic aspect of smart grid scheduling problems that we looked into aside from supporting data structures is parameterized complexity. The scheduling problems encountered in this thesis are computationally very hard problems — to the point that the theoretic informatics community believes that deterministic algorithms will never be able to solve instances of realistic size to optimality. One way of dealing with this hardness is to isolate the factors that make instances hard to solve. Then, if one is able to control these factors, one might be able to optimize instances of large size after all. We present such a factor which captures how complex a scheduling instance is at every point in time.

# 2 Preliminaries

This chapter introduces basic notation and concepts used throughout this thesis. It mainly focuses on the various scheduling problems touched and their relation. For basic algorithmic concepts, we refer to the work of Cormen et al. [Cor+09]. In addition, we assume familiarity with basic concepts related to graphs and recommend the work by Diestel [Die17] as a reference.

## 2.1 Scheduling

Scheduling problems are optimization or decision problems that ask for the assignment of *start times* to *job*s. A job is an abstract object that can have various properties depending on the concrete scheduling problem, but usually each job has at least a *processing time*, i.e., an amount of time that the job must be executing after it was started. We call the set of all jobs $J$ and use $n = |J|$ as the number of jobs. The jobs in $J$ are $j_1, j_2, \ldots j_n$. A *schedule* $S = (s_1, s_2, \ldots s_n)$ is then an assignment of a start time $s_i$ to each job $j_i$. Usually, jobs are associated with the usage of *resources*. If such an association is given, each schedule results in a resource usage over time, also called a *resource profile*. Most scheduling scenarios either place constraints on the resource usage, or aim to somehow optimize the shape of the resource profile.

start time
job
processing time
schedule
resource
resource profile

Note that this model does not allow for jobs to be interrupted and resumed or restarted after they were started once. To model these possibilities, a schedule would need to hold more information than just one start time per job. The models used throughout this thesis never allow for jobs to be interrupted (also called *preemption*). However, in Section 4.5 of Chapter 4 and Section6.2.2 of Chapter 6 we briefly outline how the models used can be made to mimic interruptible jobs.

preemption

### 2.1.1 Machine Scheduling

So far, we specified neither constraints on the start times of jobs nor an optimization criterion. For the first group of scheduling problems, so-called *machine scheduling* problems, the most important constraint is that only a certain number of jobs may execute concurrently. The number of jobs that is allowed to execute in parallel is the number of machines, and each machine can execute only one job at a time. Depending on the concrete model used, additional constraints may be given. One popular example is to limit the execution of individual jobs to a certain time window. In this case, each job is associated with a *release time*, which is the earliest possible start time of the

machine scheduling
release time

deadline
window

job, and a *deadline*, which is the first point in time when the job must have stopped running. Together, they form the *window* of the job. Another common constraint is the presence of dependencies between jobs, which are usually given in the form of a partial order on the job set. In its simplest form, a dependency from $j_a$ to $j_b$ means that $j_a$ can start only after $j_b$ has finished running.

Brucker and Knust [BK12] give an overview over various machine scheduling problems. In their earlier survey, Graham et al. [Gra+79] not only provide an overview over the field, but also propose a notation to classify machine scheduling problems. In this notation, every problem is assigned three fields $\alpha$, $\beta$ and $\gamma$. Field $\alpha$ contains information about the available machines, $\beta$ describes the jobs and their constraints and $\gamma$ specifies the objective function. Using this notation, the "scheduling zoo" database by Dürr et al. [Dür+16] provides a convenient way of searching for problems and known algorithms or hardness results.

machine
minimization

A variant of machine scheduling problems that is close to several of the problems under study in this thesis is *machine minimization* scheduling. For such problems, the number of machines, i.e., the maximum number of concurrently running jobs, is not fixed. Instead, other constraints are given, and the objective is to find a schedule that minimizes the number of necessary machines.

SRDM

To illustrate that even seemingly simple scheduling problems can already be algorithmically hard, consider such a machine minimization problem, namely Scheduling with Release Times and Deadlines on a Minimum Number of Machines (SRDM). In SRDM, additionally to a processing time, each job is associated with a release time and deadline as defined above. The objective is to find a schedule that minimizes the number of machines. Cieliebak et al. [Cie+04] have shown this problem not only to be $\mathcal{NP}$-hard, but also show that there cannot be an approximation algorithm with an approximation factor of $2 - \epsilon$ for any $\epsilon > 0$.

### 2.1.2 Project Scheduling

project
scheduling
resource

In contrast to machine scheduling introduced above, *project scheduling problems* usually associate jobs with *m resources*, for $m \geq 1$. Each job requires a certain amount (possibly zero) of each resource. Machine scheduling then becomes a special case of project scheduling, in which $m = 1$ and every job requires exactly one unit of the sole resource.

A more thorough overview over project scheduling problems than this section can give as well as some algorithmic approaches is given by Węglarz [Węg99] as well as Brucker and Knust [BK12]. In his survey, Węglarz concisely defines project scheduling problems to be "problems of allocating scarce resources over time to perform a given set of activities". The relationship to scheduling electrical loads in smart grids becomes immediately clear: Here, the main (and in most cases only) resource we care for is the electrical power required by the tasks to be run.

resource-
constrained

Project scheduling problems can be coarsely classified into two categories:*resource-*

*constrained*project scheduling problems (RCPSP) and *time-constrained* project scheduling problems (TCPSP). Problems of the category RCPSP usually have a hard usage limit on each resource, and the most common objective is to minimize the finishing time of the project, i.e., have all jobs completed as early as possible. On the other hand, TCPSP problems have fixed time limits, either on a per-job basis or globally, and the objective depends on the resource usage during the schedule.

time-constrained

This thesis is mostly exploring the possibilities of peak shaving, i.e., the objective is to reduce the maximum power demand. This naturally leads to a TCPSP problem. The Resource Acquirement Cost Problem (RACP), first defined by Möhring [Möh84][1], is a good fit. To establish notation that is used throughout much of this thesis, we now formally define the RACP.

RACP

An instance $I$ of RACP consists of $m \in \mathbb{N}^+$ resources and a set $J$ of $n$ jobs. Each $j_i \in J$ is a four-tuple: $j_i = (r_i, d_i, p_i, u_i)$. Here, $r_i$ specifies the earliest possible start (the *release*) of $j_i$, while $d_i$ specifies the *deadline*, i.e., the first point in time when $j_i$ must be finished. The *processing time* $p_i$ specifies the amount of time that $j_i$ must run without interruption after being started. Finally, the *usage* vector $u_i \in \mathbb{R}^m$ specifies the resource usage. For a resource $\rho \in \{1, \dots m\}$, $u_{i,\rho}$ determines how much of resource $\rho$ job $j_i$ requires. Additionally, for each resource $\rho$, a *weighting factor* $w_\rho$ is given.

release
deadline
processing time
usage

The task is to determine a schedule $S = (s_1, s_2, \dots s_n)$ which assigns a start time $s_i \in [r_i, d_i - p_i]$ to each job $j_i$. Given such a schedule, we can determine the usage for resource $\rho$ at time $t$ as

$$U_\rho(t) = \sum_{\substack{i\,:\,s_i \le t \wedge \\ s_i + p_i > t}} u_{i,\rho} \tag{2.1}$$

This sums over all jobs that run at $t$, i.e., that have been started at $t$ (enforced by $s_i \le t$) and have not yet finished (enforced by $s_i + p_i > t$). From this, we can get the costs at $t$ as $C(t) = \sum_{\rho=1}^{m} w_\rho \cdot U_\rho(t)$. The objective is then to find the schedule that minimizes $\max_t C(t)$.

Another common aspect of project scheduling problems is the presence of *precedence constraints*. These constraints place a requirement on the relation of pairs of jobs. The simplest form are finish-start constraints. For a job pair $(j_a, j_b)$, a finish-start constraint means that job $j_b$ can start only after $j_a$ has finished, or formally that $s_b \ge s_a + p_a$. The more general form of *time lag* constraints allows to specify arbitrary amounts of time (instead of just $p_a$) that must pass between the start of a job $j_a$ and the start of job $j_b$. We usually specify these in the form of a partial function $L : J \times J \nrightarrow \mathbb{Z}$. If $(j_a, j_b) \in L$, that means that $s_b \ge s_a + L(j_a, j_b)$ must hold.

precedence constraint
time lag

For smart grid scheduling, we often only care for one resource, namely electrical power demand. We call the RACP restricted to $m = 1$ (and without the $w_l$ factors, which become irrelevant) the Single-Resource Acquirement Cost Problem (S-RACP).

S-RACP

[1] Note that in literature, the same problem is also knows as Resource Investment Problem (RIP)

# Part I

# Modeling

# 3 A Comprehensive Modeling Framework for Demand Side Flexibility

The increasing share of renewable energy generation in the electricity system comes with significant challenges, such as the volatility of renewable energy sources. To tackle those challenges, great hopes lie with demand response. The cornerstone of (especially centralized) demand response is the coordination of flexible electrical loads. To facilitate this coordination, flexible electrical loads need to be modelled and optimized. Although extensive research exists that describes, models and optimises various processes with flexible electrical demands, there is no unified notation. Additionally, most descriptions are very process-specific and cannot be generalised.

In this chapter, we develop a comprehensive modeling framework to mathematically describe demand side flexibility in smart grids while integrating a majority of constraints from different existing models. We provide a universally applicable modeling framework for demand side flexibility and evaluate its practicality by looking at how well mixed-integer linear program (MIP) solvers are able to optimize the resulting models, if applied to artificially generated instances. From the evaluation, we derive that our model improves the performance of previous models while integrating additional flexibility characteristics.

This chapter is based on joint work with Nicole Ludwig, Esther Mengelkamp and Philipp Staudt [Bar+18c].

## 3.1 Introduction

While many societies aim at shifting their energy mix towards *renewable energy sources* (RES), the stability of the current electrical grid relies on a centralised dispatch of generation, as explained for example by Schleicher-Tappeser [Sch12]. The high fluctuations of RES in supply, as well as strong intra-day patterns e. g. in the case of solar energy, are challenges for a smooth integration. On the demand side, the traditional consumer behaviour is strenuous for the power grid as it results in high peaks and low valleys of the electric load. Currently, this fluctuation is compensated by conventional steerable power plants to ensure a reliable operation of the electricity grid. As more and more intermittent renewable sources generate electricity, this balancing technique is threatened, as Weidlich et al. [Wei+12] point out. However, the decrease in supply side flexibility might be offset by an increase in demand side flexibility. Therefore, one possibility to ease the integration of RES is to steer the consumer demand and adapt it to the supply side. Strbac [Str08] elaborates on this possibility and arrives at the result that sufficient demand flexibility can significantly reduce

the necessary installed generator capacity of renewables. *Demand side management* (DSM) summarizes measures that foster an energy consumption that is more easy for the energy system to fulfil. Among DSM techniques, *demand response* (DR) denotes techniques that aim at short-term changes in demand, for example by postponing currently running processes.

Our objective is to design a comprehensive modeling framework to capture and schedule demand side flexibility. Scheduling energy loads, hence exploiting the flexibility in the system to enhance grid stability or reduce energy costs for the consumer, is not a new idea. However, previous work predominantly employs very application-specific formulations to describe the loads and their characteristics to be scheduled. This specificity results in a vast amount of different modeling formulations. Existing modeling techniques are usually not readily transferable to new data sets or different use cases. In this context, it is especially noteworthy that demand side flexibility of private households and industrial applications exhibits very different characteristics. For example, household appliances can usually run independently from each other while industrial processes often depend on other production steps. As the considered papers always focus on only one application, no formulation (known to the authors) exists that integrates all of the features necessary to comprehensively describe demand side flexibility across multiple domains and applications. This variety of formulations in the literature makes it difficult to compare the modeling approaches, their respective results and adaptability.

We present a novel comprehensive modeling framework in the field of energy informatics to represent flexibility in households as well as in an industrial context. Based on current literature, we classify the most important characteristics of flexibility represented in previous work and incorporate the majority of them in a single modeling framework. We combine currently existing, wide-ranging research and, to our knowledge, are the first to integrate the different approaches into a single modeling framework. We complement the presentation of the framework with a practical evaluation that shows the optimization performance of the models resulting from our technique.

The various flavours of scheduling problems arising when scheduling time-flexible electrical loads are similar to problems well known in the field of operations research, especially in the area of project scheduling. The best-fitting group of problems from project scheduling is called Time-Constrained Project Scheduling Problem (TCPSP). Related to these problems, and better understood, is a group of problems called Resource-Constrained Project Scheduling Problem (RCPSP). The modeling framework presented in this chapter is an adaption of the discrete-time modeling approach presented by Pritsker et al. [PWW69] for the RCPSP.

This chapter is structured as follows. In Section 3.2, we give a short overview of existing literature concerning demand side flexibility and management. Following

this, we describe common features found in the literature modeling demand flexibility in Section 3.3. Section 3.4 introduces our modeling framework which is evaluated with regards to its performance in the following Section 3.5. We discuss our work in Section 3.6, before giving an outlook and a conclusion in Section 3.7.

## 3.2 Related Work

Demand side management is currently seeing a growing interest from researchers. A variety of authors has been dealing with demand flexibility of private households. Consequently, they ignore most characteristics of industrial loads. For example, He et al. [He+13] provide a classification of household flexibility along different dimensions, while Allerding et al. [All+12] focus on developing demand response for private households. Gottwalt et al. [Got+16] also concentrate on private households, however, they incorporate several additional restrictions. Scott et al. [Sco+13] characterize the flexibility of individual household devices. However, the description is tailored to specific appliances and therefore not domain independent. In [Feh+14], Fehrenbach et al. show that thermal appliances and specifically the expansion of heat pump use may have the largest flexibility potential of private households. Du and Lu [DL12] provide a scheduling algorithm for those thermal appliances. Their work is extended by Alizadeh et al. [Ali+15], who differentiate between curtailable thermal loads and other deferrable loads. Household behaviour with regards to the provision of flexibility and effects on electricity costs is simulated by Gottwalt et al. [Got+11]. They conclude that saving potentials for households are moderate when compared to the investment in smart meter technology. Contrary to this result, Setlhaolo et al. [SXZ14] come to the conclusion that a reduction of up to 25% of the electricity costs of private households is possible. The investigation by Soares et al. [SGA14] also considers customer dissatisfaction besides the monetary compensation. In [Sou+11], Sou et al. model household appliances as an interruptible sequence of uninterruptible tasks and demonstrate that in their model, energy costs can be significantly lowered by scheduling these tasks.

An introduction to using demand side flexibility as a means to integrate renewable generation is given by Palensky and Dietrich [PD11]. Other research has established that fluctuations of renewable generation can be offset by demand side flexibility, as for example shown by Strbac [Str08]. However, the authors argue that it is hard to determine how to compensate the providers of the demand side flexibility. Halvorsen and Larsen [HL01] describe the effects of appliance endowment and additional investment on the ability to provide flexibility. A new approach for a scheduling algorithm was developed by Ströhle et al. [Str+14] to match uncertain supply with different demand packages to maximize total welfare. The optimal combination of private household

flexibility is investigated by Gärttner et al. [GFW16], providing recommendations to flexibility portfolio aggregators.

Ashok and Banerjee [AB00] pioneer the field of industrial demand response. Their model is specified in [Ash06] but leaves certain restrictions for future research. An extensive description of characteristics of demand side flexibility beyond residential flexibility is given by Petersen et al. [Pet+13]. The same authors also develop a first taxonomy for flexibility but chose not to incorporate a variety of characteristics of flexibility [Pet+14]. Paulus and Borggrefe [PB11] establish that demand side management bears considerable monetary potential in energy intensive industries. Qureshi et al. [QGJ14] develop a model to investigate economic potential of demand response in office buildings. In [SP96], Schilling and Pantelides present a MIP model that schedules tasks with recurring instances. Mitra et al. [Mit+12] as well as Moon and Park [MP14] consider scheduling with regards to electricity costs for industrial production. Oudalov et al. [OCB07] use batteries to reduce demand peaks.

## 3.3 Modeling Flexibility

In this section, we systematically categorize the most important aspects and features of demand side flexibility that are captured by models present throughout literature. Based on the presented features of flexibility, we classify the most relevant models of demand side flexibility with regards to the features they incorporate in Table 3.1.

1. **Time Frame.** States whether the described model uses discrete time steps or continuous time.

2. **Interruptible Jobs.** The model allows for interruptible jobs, i. e., jobs which do not have to be executed consecutively. We do not distinguish between the ability to stop jobs at any time, or at predefined time slots. *Indirect modeling:* Models which allow for interdependent jobs with arbitrary time lags (see below) enable us to split up interruptible jobs into small chunks and connect these with dependencies. This way, the original job can either be executed consecutively (if all chunks are scheduled consecutively) or with interruptions. (Negative) lag times can be used to specify maximal interruption times. Thus, all models supporting interdependent jobs with arbitrary time lags with indirectly support interruptible jobs.

3. **Storage.** The model allows to include some form of storage possibility. *Indirect modeling:* Storage can be modelled indirectly via a special kind of dummy jobs which can be moved forward to simulate charging of the storage. The place where dummy jobs were moved away from then has more power available, simulating getting energy out of storage.

4. **Interdependent Jobs.** Jobs can have predecessors, allowing a job to be scheduled only as soon as all its predecessors are completed. Optionally, time lags can be associated with dependencies, enforcing a certain amount of time to pass between the finish of a predecessor and the earliest start of a successor.

5. **Earliest Start Time.** Jobs can be associated with an earliest start time and may not be scheduled before that time.

6. **Deadline.** Jobs can be associated with a deadline and must be scheduled such that they are finished at that time. From this, the possibility of an overall deadline directly follows.

7. **Multiple Resources.** The model can contain more resources than electrical energy alone, and jobs may require amounts of more than one resource simultaneously.

**Table 3.1:** Comparison of the integrated flexibility features in related work to our modeling framework. Checkmarks mean yes, crosses mean no. A dash means that a feature is not applicable (see detailed feature description). Checkmarks in parentheses indicate that a feature is not explicitly modelled, but can be expressed approximately using other modeling features. We did not check for possibilities to use this indirect modeling approach for the related work.

| | [All+12] | [AB00] | [Ash06] | [CMB02] | [FHM14] | [Got+16] | [Luo+98] | [Mit+12] | [MP14] | [OCB07] | [Pet+13, Pet+14] | [SP96] | [Sou+11] | **This Chapter** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Time Frame [a] | d | d | d | c | d | d | d | d | d | d | d | d | c | d |
| Interruptible Jobs | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | (✓) |
| Storage | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ [b] | ✓ | ✓ | (✓) |
| Interdependent Jobs | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Earliest Start Time | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Deadline | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Multiple Resources | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Base loads | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Modes | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Drain, Losses | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Ramping | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Multiple Runs | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| Down-/Uptime | – | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | – |
| Production | – | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | – |

[a] d = discrete, c = continuous
[b] Only in [Pet+13]

8. **Base loads.** Uncontrollable loads, i.e., jobs that must be scheduled at a specific time, may be part of the model. *Indirect modeling:* Base loads can be modelled indirectly if earliest start times *and* deadlines are present, or if deadlines *and* interdependent jobs are present, by inserting dummy jobs that can only be scheduled at the specified times.

9. **Modes.** Jobs may have multiple modes, where every mode is a combination of a run time and resource requirements. Each job can have a possibly large set of possible operation modes. Modes with less required resources usually take longer. The scheduler can decide in which mode to run a job. We assume all modes of a job to be of equal value, i.e., things like product quality do not depend on the chosen mode of a job.

10. **Drain, Losses.** Energy spent on the execution of a job may drain over time, i.e., another job which is scheduled later might need to replenish energy (and thus use more resources or take longer) if it is scheduled late.

11. **Ramping.** Jobs may be associated with a ramping function of some kind, describing how resource usage slowly increases when the job is started and decreases when the job finishes. Ramping might be unnecessary if another job is executed right before the ramping job starts or directly after the job. If ramping is necessary, the runtime of a job usually increases.

12. **Multiple Runs.** Every job can either be scheduled once or multiple times, over the span of the optimization period. Multiple runs are most useful when we choose the time horizon in such a way that we need to meet a production target.

13. **Down-/Uptime.** Jobs can be associated with a fixed amount of time where they need to be shut down after running (downtime), or a fixed amount of time that they have to be used (uptime). In contrast to the flexible description of minimum and maximum runtime, this is a fixed amount of time. This makes only sense when multiple runs (see above) are possible.

14. **Production.** Jobs can be associated with a production output, and the whole schedule has to meet a production target. Again, this only makes sense when multiple runs are possible, as otherwise each jobs has to run exactly once and the production is fixed.

Table 3.1 summarizes the papers we examined and gives an overview of the features considered. The features we address with our new formulation are indicated with check marks. Brackets indicate that we can reasonably express a certain feature *indirectly*, although we do not meet all subtleties encountered in the literature presented. Features not yet included in our modeling framework are marked with crosses. In total, we take 14 different features into account, which we describe as follows:

## 3.4 Optimization Model

In this section, we describe our proposed comprehensive modeling framework. We incorporate the majority of features which we found in related papers and summarized in Table 3.1.

The basis for our model are *jobs*, representing uninterruptible processes that require a certain amount of electrical power during their execution. We associate a *duration* with each job.

job

duration

The optimization model that results from our modeling technique is a mixed-integer linear program (MIP). Given an instance with $n$ jobs in which job $i$ can be run in $M_i \geq 1$ different modes and the latest job deadline is $D_{\max}$, the decision variables consist of two groups of binary variables. A group $s_i(t)$ of variables indicating whether job $i$ starts running at time step $t$ and a group of variables $m_{i,j}$ indicating whether job $i$ is run in mode $j$. This limits the number of decision variables to $n \cdot D_{\max} + \sum M_i$. All other variables are derived from these decision variables. Table 3.2 lists all used variables as well as the constants that are part of the input instance.

For each job $i$, $\tilde{P}_i$ specifies the *base power requirement* of $i$. This base power requirement can be modified by the mode coefficients, if the job has more than one possible mode. The same holds for the *base run time* $\tilde{T}_i$. Job $i$ can be run in $M_i$ different *modes*. For each mode $m \in \{1, \dots M_i\}$, the actual power requirement for job $i$ is $\phi_{i,m} \cdot \tilde{P}_i$, and the actual run time is $\psi_{i,m} \cdot \tilde{T}_i$ if job $i$ is run in mode $m$. If $M_i = 1$, it makes sense to set $\phi_{i,1} = \psi_{i,1} = 1$, and $\tilde{P}_i$ becomes the actual power demand and $\tilde{T}_i$ becomes the actual run time. The *deadline* $D_i$ specifies the first time step at which $i$ must be finished, and $R_i$ specifies the *release time*, i.e., the first time step at which it can run.

base power requirement

base run time

mode

deadline

release time

For a pair of jobs $i$ and $j$, the value of $L_{i,j} \in \mathbb{Z} \cup \{-\infty\}$ specifies the minimal time that must pass between the start of $i$ and the start of $j$. Note that we allow negative values. If there should be no dependency between $i$ and $j$, setting $L_{i,j} = -\infty$ will achieve this.

Related to dependencies is the concept of *drain*. Certain processes, especially in chemical industry, require more power (resp. longer run time at the same power) if another process was finished too far in the past. The classical example are materials that need to be heated and cool down over time. We capture this — in a linear fashion — in the $\tau_{i,j}$ input constant for two jobs $i$ and $j$. When there are $k$ time steps between the end of $i$ and $j$, the run time of job $j$ is extended by $\tau_{i,j} \cdot k$. Of course, energy does not drain limitless, e.g., at some point, a material cooling down reaches environmental temperature. Therefore, $\hat{\tau}_{i,j}$ places a limit on the possible run time extension. A similar concept is *ramping*. Some processes can not be arbitrarily switched on or off. We capture this by a series of ramping jobs that can be prepended to a job $j$ if job $i$ is finished too long before $j$ starts. The number of possible ramping jobs before $j$ is $\Lambda_j$. Parameter $\delta_{i,j,k}$ determines the number of time steps that can pass between the end of $i$ and the start of $j$ before ramping job $k$ must be executed before $j$ can start. For

drain

ramping

the ramping jobs, $\mu_{i,k}$ determines the power demand of the $k$th ramping job for job $i$. Finally, certain objective functions (see below) require a specification of how much power can be used at time $t$ without incurring any cost, and how much the cost is for all power exceeding that amount. This is specified by $F_t$ and $c_t$.

Given an input instance specified like this, the model has two types of decision variables. For job $i$ and time step $t$, $s_{i,t}$ indicates that $i$ starts at time step $t$. For a job $i$ with multiple modes, $m_{i,k}$ indicates that job $i$ runs in mode $k$.

Note that with this specification, we consider only one resource, which we call *power*. However, incorporating multiple resources can be easily done by just duplicating all variables and constraints regarding resources for each additional resource, and combining the objective functions (see below). For simplicity sake, we do not formally specify that here.

**Objective Functions.** Our modeling technique can easily accommodate multiple objectives, each of which is relevant in a smart grid scheduling context, and which can of course also be combined.

overshoot minimization

The first one is an *overshoot minimization* objective. The goal is to buy as little energy from the grid as possible, assuming the presence of some "free" but non-steerable generation, for example by photovoltaics. We thus minimize the difference between self-produced electricity $F_t$ and the power $\hat{P}_t$ needed to perform the desired processes. Using energy from the grid is penalised with a cost $c_t$. The objective function is then

$$\min \sum_t c_t \cdot \left(\max\left(\hat{P}_t - F_t, 0\right)\right). \tag{3.1}$$

peak shaving

Another objective frequently seen in literature is that of *peak shaving*. Here, the maximal power usage is to be minimized. The objective function for peak shaving is

$$\min \left(\max_t \hat{P}_t\right) \tag{3.2}$$

Now that we have introduced the model's variables and objectives, we present the constraints. We do this in two blocks. The first block consists of the constraints that provide the basic workings of the model, binding together the decision variables and resource usage. The second block consists of constraints that model specific features from Section 3.3. The first block of constraints is:

**Table 3.2:** Variables used in the modeling framework, with the input constants at the top, the decision variables in the middle and the derived variables in the bottom part of the table.

| | |
|---|---|
| | Model Constants (Input) |
| $n$ | Number of jobs |
| $\tilde{P}_i$ | Base power requirement of job $i$ |
| $\tilde{T}_i$ | Base run time of job $i$ |
| $M_i$ | Number of different modes for job $i$ |
| $\phi_{i,m}$ | Mode coefficient for time adjustment of job $i$ in mode $m$ |
| $\psi_{i,m}$ | Mode coefficient for power adjustment of job $i$ in mode $m$ |
| $D_i$ | Deadline of job $i$ |
| $R_i$ | Release time of job $i$ |
| $L_{i,j}$ | Minimum time lag between job $i$ and $j$, measured in time steps from the end of $i$ to the start of $j$ |
| $\tau_{i,j}$ | Runtime extension coefficient for the separation of jobs $i$ and $j$ |
| $\hat{\tau}_{i,j}$ | Limit on separation-caused run time extension for jobs $i$ and $j$ |
| $\Lambda_i$ | Maximum number of ramping steps for job $i$ |
| $\delta_{i,j,k}$ | Number of time steps between the end of job $i$ and the start of job $j$ before job $j$ must execute ramping step $k$ before executing the actual job |
| $\mu_{i,k}$ | Power requirement of job $j$'s $k$th ramping step |
| $F_t$ | Power available "for free" at time step $t$ |
| $c_t$ | Cost of using one unit of energy above $F_t$ at time $t$ |
| | Decision Variables |
| $s_{i,t}$ | Binary variable, becomes 1 if and only if job $i$ starts at time step $t$ |
| $m_{i,k}$ | Binary variable, indicating if job $i$ is to be run in mode $k$ |
| | Derived Variables |
| $\tilde{\phi}_i$ | Effective time adjustment coefficient of job $i$ |
| $\tilde{\psi}_i$ | Effective power adjustment coefficient of job $i$ |
| $P_i$ | Power requirement of job $i$ in its selected mode |
| $T_i$ | Run time of job $i$ in its selected mode |
| $\hat{P}_t$ | Total power requirement at time step $t$ |
| $\sigma_i$ | Time step in which job $i$ starts |
| $\eta_i$ | First time step in which job $i$ is finished |
| $M$ | Large constant used to switch constraints on an off |
| $\rho_{i,k}$ | Binary variable indicating whether job $i$ must execute its $k$th ramping step |
| $\omega_{i,t}$ | Ramping power of job $i$ at time $t$ |

$$\forall i \in \{1, \dots n\} : \quad \sum_t s_{i,t} = 1 \tag{3.3}$$

$$\forall i \in \{1, \dots n\} : \quad \sum_j m_{i,j} = 1 \tag{3.4}$$

$$\forall i \in \{1, \dots n\} : \quad \sigma_i = \sum_t t \cdot s_{i,t} \tag{3.5}$$

$$\forall i \in \{1, \dots n\} : \quad P_i = \tilde{\psi}_i \cdot \tilde{P}_i \tag{3.6}$$

$$\forall i \in \{1, \dots n\} : \quad \eta_i = \sigma_i + T_i \tag{3.7}$$

$$\forall i \in \{1, \dots n\} : \quad \eta_i \leq D_i \tag{3.8}$$

$$\forall i \in \{1, \dots n\} : \quad \sigma_i \geq R_i \tag{3.9}$$

$$\forall i, j \in \{1, \dots n\} : \quad \sigma_i + L_{i,j} \leq \sigma_j \tag{3.10}$$

$$\forall i \in \{1, \dots n\} : \quad \tilde{\psi}_i = \sum_j m_{i,j} \psi_{i,j} \tag{3.11}$$

$$\forall i \in \{1, \dots n\} : \quad \tilde{\phi}_i = \sum_j m_{i,j} \phi_{i,j} \tag{3.12}$$

Equation (3.3) ensures that each job is scheduled once. Similarly, Equation (3.4) ensures that for every job, exactly one mode is selected. Equation (3.5) binds $\sigma_i$ to the start time of job $i$ by summing over all time instances times the indicator whether the job starts in this instance. During its execution, each job needs has a power demand of $P_i$ which depends on the mode the job is running in and its base power $\tilde{P}_i$ (Equation (3.6)). The effective power adjustment coefficient $\tilde{\psi}_i$ for job $i$ is determined in (3.11). In a similar fashion, the run time is changed by the job's mode — we defer the exact explanation of how this works until later. However, the effective run time adjustment coefficient $\tilde{\phi}_i$ is determined just as $\tilde{\psi}_i$ is (3.12).

Each job has to be finished before its deadline is reached (3.8), with the finish time depending on the start time and the actual run time of the job (3.7). Also, a job cannot start before its release time (3.9).

If a time lag is defined between jobs $i$ and $j$, the start time of the job $i$ and the start time of job $j$ need to be separated by at least their minimum time lag $L_{i,j}$ (Equation (3.10)).

We now describe the remaining necessary constraints which require a more detailed explanation.

**Time Extension for Drain and Modes.** Recall that $\tilde{T}_i$ is the base time requirement for Job $i$, and $\phi_{i,j}$ (resp. $\psi_{i,j}$) is the power (resp. time) mode coefficients for the mode being run in. These coefficients determine how the power requirement (resp. run time) changes if mode $j$ is selected, i.e., if $m_{i,j} = 1$.

Additionally, the actual run time may depend on several drain factors $\tau_{a,i}$. The drain factors indicate a run time extension of $i$ if job $i$ is not started immediately after job $a$, as the energy that drained between the execution of $a$ and $i$ has to be replenished. However, at some point we may assume that all energy has drained and that $i$'s run time does not increase any further because of drain. This limit is imposed by $\hat{\tau}_{a,i}$.

In total, the resulting constraint on the runtime $T_i$ of $i$ is

$$\forall i \in \{1, \dots, n\}: \quad T_i = \tilde{\phi}_i \cdot \tilde{T}_i + \sum_k \min\left(\tau_{k,i}\left(\sigma_i - \eta_k\right), \hat{\tau}_{k,i}\right). \tag{3.13}$$

Here, the sum sums over all jobs $k$ that might precede $i$. For jobs that do not precede $i$, or for which no drain is desired, $\tau_{k,i}$ should be set to zero, thereby making those terms irrelevant. Note that this part can never become negative, because $k$ being a predecessor of $i$ forces $i$ to start only after $k$ has finished, i.e., $\sigma_i \geq \eta_k$.

Of course, the minimum function is not a linear function. However, it can be linearized using standard techniques.

**Ramping.** The ramping of job $j$ is modelled via a series of dummy jobs, each of length 1, describing the steps in the ramping process. Whether the $\lambda$th ramping job must be executed is denoted by $\rho_{j,\lambda}$, where $\lambda \in \{1, \dots, \Lambda_j\}$. Here, $\Lambda_j$ is the maximum number of steps necessary to reach the power input needed for job $j$ to start. At which ramping step we start depends on the time distance between the end of the last predecessor job $i$, i.e., $\eta_i$ and the start time of the job $j$ that needs ramping, i.e., $\sigma_j$. We check if we execute ramping step $\lambda$ by introducing one of the following constraints for every predecessor $i$ of $j$:

$$\forall i, j \in \{1, \dots, n\}, \forall \lambda \in \{1, \dots, \Lambda_j\}: \quad \rho_{j,\lambda} \cdot M \geq (\sigma_j - \eta_i - \delta_{i,j,\lambda}), \tag{3.14}$$

where $M$ is a suitably large constant. Then, $\rho_{j,\lambda}$ must become 1 if the right side is larger than 0, i. e., if $i$ and $j$ are separated by more than $\delta_{i,j,\lambda}$ time steps. The $\lambda$th ramping step of job $j$ must be executed $\lambda$ time steps before the start of job $j$, because $\lambda - 1$ further ramping steps will follow. With this, the amount of power required for ramping job $j$ at time step $t$ can be formulated as

$$\forall j \in \{1, \dots n\}, \forall t: \quad \omega_{j,t} = \sum_{\lambda=1}^{\Lambda_j} \rho_{j,\lambda} \cdot s_{j,t+\lambda} \cdot \mu_{j,\lambda}. \tag{3.15}$$

Here, $\omega_{j,t}$ becomes $\mu_{j,\lambda}$, i. e., the power for $j$'s $\lambda$th ramping step, if and only if $j$ is started in time step $t + \lambda$ and $\rho_{j,\lambda}$, i.e., the indicator if the $\lambda$th ramping step must be executed, is 1.

**Table 3.3:** Properties of the four sets of generated instances. Intervals $[a, b]$ indicate numbers chosen uniformly at random between $a$ and $b$, inclusively.

| Name | # Jobs | # Dependencies | # Dependencies with Drain | Net Job Slack |
|---|---|---|---|---|
| Jobs (Set A) | {50, 100, 150, 200, 250, 300} | [0, 1000] | 0 | [0, 30] |
| Dependencies (Set B) | 200 | {0, 100, 500, 1000, 2000, 3000} | 0 | [0, 30] |
| Drain (Set C) | 200 | 1000 | {0, 100, 200, 500, 900} | [0, 30] |
| Slack (Set D) | 200 | [0, 10000] | 0 | {1, 25, 50, 75, 100} |
| Slack with few Dep. (Set E) | 200 | 200 | 0 | {1, 25, 50, 75, 100} |

**Total Power Requirement.** The total power requirement at time step $t$ is described as the sum over the power of all running jobs at time step $t$ and the power used of the jobs currently ramping

$$\forall t: \quad \hat{P}_t = \sum_i \left( P_i \sum_{t - T_i < t' \leq t} s_{i,t'} \right) + \sum_j \omega_{j,t} \tag{3.16}$$

**Linearization.** Some of the constraints described by us are not linear per se. See for example Equation (3.15), where $\rho_{j,\lambda}$ and $s_{j,t}$ — both variables, not constants — are multiplied. However, for two binary variables $a$ and $b$, such a multiplication can easily be linearized if the product contributes only positively to the objective function, i.e., if a solution where the product is 0 is preferred.

Let $c$ be a third binary variable indicating whether the product $a \cdot b$ is 1. Then it is enough to introduce the constraint $c \geq a + b - 1$. We can replace $a \cdot b$ with $c$ everywhere. If $a$ and $b$ are both 1, then $c$ must be 1. In all other cases, $c$ will be set to 0, since an optimum solution prefers the product to be 0.

## 3.5 Experimental Evaluation

We experimentally evaluate the models resulting from our modeling framework by generating random instances, optimizing the MIPs for 15 minutes and measuring the MIP gap, a measure for the difference between best feasible solution found and best proven lower bound. We evaluate the framework with peak shaving as objective

function. We chose this objective since it does not need to make any assumptions as to $F_t$ and $c_t$, which we assume to be highly dependent on the concrete scenario.

We conduct all experiments on a machine with 16 Intel Xeon E5-2670 cores at 2.6 GHz and 64GB of RAM, using Gurobi 6.5 as a solver.[1] We allow Gurobi to use all cores of the machine.

We generated five separate sets of instances. For each of the five sets, Table 3.3 shows the number of jobs, number of dependencies between pairs of jobs, number of dependencies that are associated with a drain, and the (net) *slack* jobs have in the instances. The *slack* of a job is its deadline minus the release time minus the run time of the job. The slack gives an indication of the amount of freedom one has during scheduling. The *net* slack compensates for the fact that in the presence of dependencies, the earliest possible start time of a job does not just depend on the release time, but also on the start times of its predecessors. Thus, a lower bound for the earliest start time of a job is the maximum of all its predecessors' earliest start times plus their respective run times. The net slack takes this lower bound *and* the release time into account.

slack

net slack

In Table 3.3, intervals like $[a, b]$ indicate that the value was chosen uniformly at random between $a$ and $b$ for every instance. A set like $\{a, b, c\}$ indicates that we generated instances for each of the values $a$, $b$ and $c$. For each possible combination of values in every row of the table, we generated 30 instances, for a total of 810 instances. We set the objective for all instances to minimize the peak power requirement. The power requirement for every job has been drawn from a normal distribution with mean 5 and standard deviation 2.

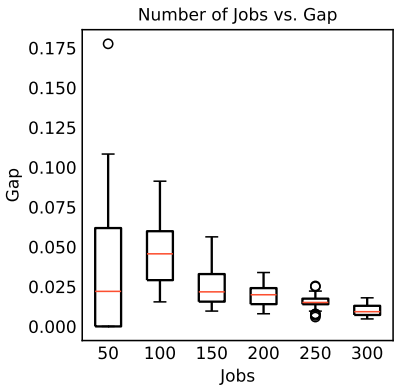
In the following, we analyse the *MIP gap* between best found feasible solution and best lower bound. Formally, let $C_{\text{bound}}$ be the cost of the best lower bound found by the optimizer and $C_{\text{feasible}}$ be the cost of the best found feasible solution, then the gap is defined as $(C_{\text{feasible}} - C_{\text{bound}})/C_{\text{feasible}}$. For instances where no feasible solution was found, we set the gap to 1.

MIP gap

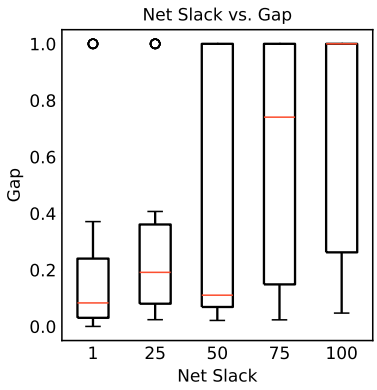
Figure 3.1c shows the effect of the number of dependencies on the gap achieved after 15 minutes. We can see that for up to 100 dependencies, all instances stay below a 2% gap. Even for 1000 dependencies, almost all instances can be solved up to a 4% gap. However, the gap increases superlinearly with the number of dependencies.

In Figure 3.1a, we present the same plot for a varying number of jobs. A counter-intuitive result is that while the gap first increases from 50 to 100 jobs, it decreases from there on. An explanation for this is the fact that the gap is a relative measure. As we keep adding jobs (keeping the global deadline and release time fixed), the absolute value of the optimum solution increases. A fixed (absolute) difference between the best feasible solution and the best lower bound becomes a lower gap as the optimum

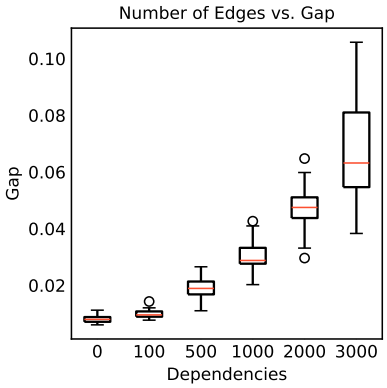
[1] See `https://www.gurobi.com`



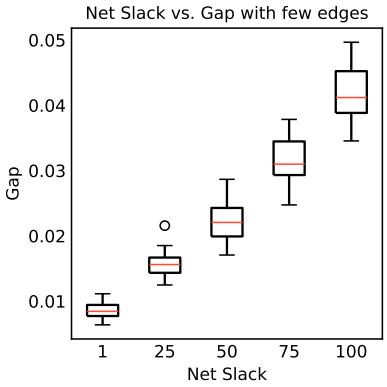

**(a)** Varying the number of jobs (Set A)



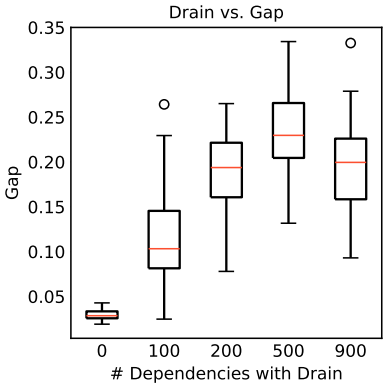

**(b)** Varying the amount of slack (Set D)




**(c)** Varying the number of dependencies (Set B)




**(d)** Varying the amount of slack with few dependencies (Set E)




**(e)** Varying the number of dependencies with drain (Set C)

**Figure 3.1:** The effect of varying different parameters in instance generation. Red lines indicate the median, boxes indicate upper and lower quartile. Whiskers show the extend of the remaining results, with outliers being shown as circles.

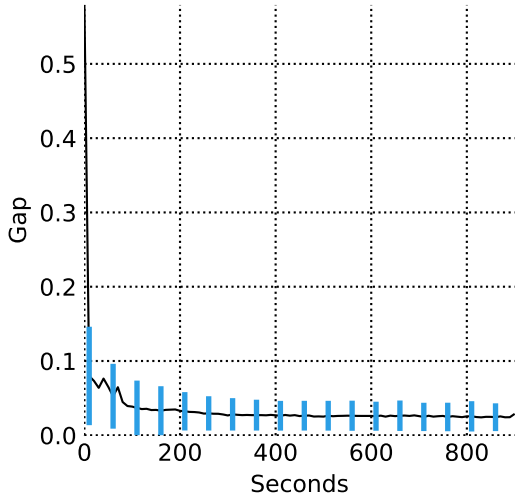



**Figure 3.2:** Convergence speed of the MIP solutions. The black line indicates the median over all runs. The blue bars indicate the area in which 75% of all runs fall.

solution increases, which manifests here. However, note that even in the worst case, with 100 jobs, the majority of the instances could be solved to a gap of 5% or below.

Figure 3.1b shows the net slack of all jobs versus the achieved gap. It is visible that large net slacks strongly increase the computational complexity of the model. Note that the mean duration of all jobs is 10, i.e., a net slack of 50 says that the net window of every job is already six times its duration. Furthermore, Figure 3.1d shows results of the same experiment where we kept the number of dependencies moderate, namely at 200. The gap again increases with the size of the net slack, however even for a net slack of 100, the gap never gets larger than 5%. Thus, a large number of dependencies combined with a lot of slack is what drives complexity here.

Regarding the effect of dependencies with drain, Figure 3.1e shows the gap for different numbers of dependencies with associated drains. As you can see from Table 3.3, we kept the number of dependencies constant at 1000 and vary the fraction of dependencies with drain. We can see that drain significantly raises the complexity of the model, even if just 10% of the dependencies are associated with drain. However, the complexity does not strictly increase with the number of dependencies with associated drain: If too many dependencies are incentivized to have jobs placed closely to each other, the flexibility in the model decreases and results improve slightly, as can be seen.

We finally take a look at the speed with which the MIP solver converges to the optimum solution in Figure 3.2. The black line shows the median of the achieved gap over all MIP runs at different points in time. The blue bars indicate the upper

respectively lower quartiles. We can see that within the first 200 seconds, the MIP gap drops to below 10% on average. After 200 seconds further improvement is relatively slow.

**Direct comparison with [Pet+14].** Petersen et al. [Pet+14] also give a MIP formulation of a problem which is a subset of the problem our framework can solve. They state that their MIP, executed on a standard laptop, was able to solve five out of twenty generated instances before hitting memory limits, and for the five solved instances, average execution time was eight minutes. We tried to generate twenty instances based on the same parameters as they did, i. e., ten instances each corresponding to their *Portfolio*$(25, 100)$ and *Portfolio*$(50, 100)$ settings. In [Pet+14], the authors define a *Portfolio*$(N, K)$ "*as a randomly generated portfolio of* $N$ *local units with* $K_{\text{Run}} \in \{2, 3, 4, 5\}, \overline{P} \in \{1, 2, 3, 4\}$, *and* $K_{\text{End}} \in \{1, 2, \ldots, K\}$". A local unit is in there definition a flexible consumer, corresponding to a job in our formulation. Unfortunately, the authors do not state how $P_{\text{Dispatch}}$, described by $F_t$ in our formulation, is selected. For the given portfolio settings, the average power consumption over the (expected) optimization period is 4.4 respectively 8.8, thus we selected $P_{\text{Dispatch}} = 5$ and $P_{\text{Dispatch}} = 9$. This should result in relatively difficult instances since, in an optimum solution, jobs must be distributed as uniformly as possible.

We solved these instances using our model on a standard laptop with 12 GB of RAM and a quad-core CPU running at 2.4 GHz. Gurobi was able to solve all instances to optimality within less than a second and a peak memory usage of less than 35 MB. The fast computation suggests that our framework indeed results in fairly tractable MIP models.

## 3.6 Discussion

We present a comprehensive modeling framework for demand side flexibility incorporating most of the characteristics from Table 3.1. In our framework, we currently do not include the features multiple runs, down-/uptime and production. All these features only make sense in a setting where part of the optimization is the selection of how often to execute a certain job. In such a setting, jobs are usually associated with a (monetary) profit. This setting is orthogonal to a setting where each job must be executed exactly once. Without a specific production target, scheduling all jobs exactly once seems most fitting for smart grid scheduling. This results in a fixed output for all possible schedules.

In the future, we are looking to extend the modeling framework and include the remaining flexibility constraints. Simultaneously, we plan to evaluate the representation and interdependences of the individual constraints theoretically and with real-life case studies. In current research, it seems unclear what realistic test instances that

cover a lot of possible real-life scenarios, look like. This is a topic for further research on its own.

Additionally, further research should investigate the optimal degree of flexibility in production processes. In our model, flexibility has zero marginal costs. However, providing a particular level of flexibility usually incurs a certain amount of costs and resources that need to be considered. As generating and providing energy usually incurs production costs, the unused self-produced energy also needs to be further considered. Therefore, non-utilization should be penalized in the optimization problem. A solution approach to this is the direct inclusion of energy storage capacities. Energy storage can help out by saving the otherwise unused energy for a certain amount of time. Nevertheless, storage costs will also occur and need to be considered in the optimization problem. Currently, we are only considering costs that occur for additional consumption of electricity meaning that we minimize the absolute area difference between production and consumption.

As we have discussed before, not all flexibility aspects are yet included in our experimental implementation even though we consider them in the mathematical model. However, we expect the remaining characteristics to be of lower computational complexity as those that we have already incorporated. Thus, their influence on the optimality gap and runtime should be smaller than the impacts of the characteristics we already evaluated in Section 3.5. A complete evaluation of our model's empirical complexity is subject to further research.

We also point out that, for now, we use Gurobi 6.5's standard configurations to solve the mixed-integer linear programs resulting from our modeling framework. These standard configurations work adequately for our random instances. However, tuning these could lead to improved performance. This approach might become useful in time-critical real-life implementation scenarios.

## 3.7 Conclusion

In this chapter, we presented and evaluated a modeling framework which allows a universal representation of demand side flexibility. Thus, we address a gap in the modeling of flexibility as current research has introduced a variety of models which are suitable for specific problem instances but neglect the characteristics of demand side flexibility for other applications. After an extensive review of existing literature, we aggregate a coherent list of demand side flexibility features from research. We then create a framework to integrate most of these into one consistent model. After introducing the modeling framework mathematically, it is evaluated using randomly created problem instances, and the performance is measured. We measure the performance as the occurring optimality gap and show that our model performs well computationally while considering a wide range of features. We focus on the minimization of externally

procured energy and peak shaving. In future work, we will consider the economic implications of providing and investing in flexibility. Our model advances current research as it can be universally used to describe flexibility for different applications and improves the comparability of optimization algorithms.

# 4 Exploring the Benefits of Flexibilization in Industrial Contexts

We introduce a novel approach to demand side management: Instead of using flexibility that needs to be defined by a domain expert, we identify a small subset of processes of e. g. an industrial plant that would yield the largest benefit if they were time-shiftable.

To find these processes we propose, implement and evaluate a framework that takes power usage time series of industrial processes as input and recommends which processes should be made flexible to optimize for several objectives. The technique combines and modifies a motif discovery algorithm with a scheduling algorithm based on mixed-integer programming.

We show that even with small amounts of newly introduced flexibility, significant improvements can be achieved, and that the proposed algorithms are feasible for realistically sized instances. We thoroughly evaluate our approach based on real-world power demand data from a small electronics factory.

This chapter is based on joint work with Veit Hagenmeyer, Nicole Ludwig and Dorothea Wagner [Bar+18b].

## 4.1 Introduction

It has almost become folklore within the energy research community that creating and exploiting demand side flexibility can and should be a response to the growing amount of intermittent, non-dispatchable generation in future energy systems based on renewable energy sources. A body of literature (e. g. [FN16, KR13, Tan14]) looks into this from different angles, exploring and demonstrating usefulness, applicability and computational feasibility.

Many of the approaches, especially those employing centralized variants of *demand side management* (DSM) resp. *demand response* (DR), need to solve an optimization problem that is a particular case of a project scheduling problem and can be formulated as follows. Given a set of processes, each associated with electrical power demand, and given specific flexibility for each process, run each process at the right time such that some objective regarding the total power usage is optimized. The objective, the processes, and the form of the flexibility can take many forms.

However, even though research in this field is plentiful, implementations of demand response in practice, especially *centralized* DR, are scarce. There are many reasons for this, not the least important of which is that much of the Smart Grid infrastructure so far only exists on paper, or that the current electricity market has insufficient incentives to provide flexibility.

We believe there are other significant roadblocks for centralized demand response. First, DR often focuses on households, neglecting that shifting demand from industrial customers would have a stronger impact due to the amount of energy expended. This focus is mainly happening because contrary to households, opportunity costs for shifting demand in industry, i. e., changing a production schedule, can be very high, making most price-based DR schemes infeasible.

Additionally, we have found that operators of industrial plants are most often unable to specify what the flexibility of their processes is. Designing production processes with demand flexibility in mind has not been a focus in the past. Neither has analyzing and documenting the flexibility hidden in existing industrial processes. Thus, confronting plant operators with the option of DR, we realized that many feel that this is an all or nothing choice and decide that demand response is just not for them.

Finally, while there are several sets of household consumption data available (such as the REDD data set by Kolter and Johnson [KJ11]), such data is very sparse in the industrial context, additionally impeding research in this area.

We hope that we can overcome all of these obstacles with a different approach: Instead of solving scheduling problems where flexibility must be specified a priori per process, we suggest taking industrial plants as they are currently operated and analyzing which processes would yield the most substantial benefit if they were flexible. We hope that such an analysis can be a powerful tool for industrial plant managers in motivating and implementing demand response.

**Our Contribution.** We propose, implement and evaluate a new framework that takes power consumption time series of industrial processes as input and indicates which processes should be made flexible to optimize for several objectives. With this framework, we want to answer the question: How much flexibility do we need? Thus, given all the processes we have, how many must be made flexible (and by how much) to get improvements regarding the energy consumption. To the best of our knowledge, we are the first to propose this approach to demand response. Based on a recently published set of power consumption time series from an electronics factory, we show that we can achieve notable improvement even with little newly-created flexibility.

**Outline.** Before describing the proposed technique, we formally define the terminology and problems used throughout this chapter in Section 4.1.1 and summarize related work in Section 4.2. Then, we describe our approach and its individual steps in Section 4.3. We evaluate our approach in Section 4.4 and discuss our results in Section 4.7, concluding the chapter in Section 4.8.

### 4.1.1 Problem Definition

The Flexibilization Project Scheduling Problem (FPSP) is the main problem of this chapter. Contrary to usual scheduling problems, we do not start with a set of jobs and ask for a schedule. Instead, we are given an existing schedule on a set of jobs plus some limitations on the addable flexibility as inputs. We first formally define a *schedule*:

schedule
job
start time
processing time

**Definition 1** (Schedule)**.** *A* schedule *is a set of $n$ triples $(c_i, p_i, u_i) \in \mathbb{N} \times \mathbb{N} \times \mathbb{R}$, for $i \in \{1, 2, \dots n\}$. Each triple describes a* job *in the schedule. We identify the triple $(c_i, p_i, u_i)$ with job $i$. For job $i$, the field $c_i$ indicates the (current)* start time *of job $i$ in the schedule, $p_i$ indicates the* processing time *(or* duration*) of the job, and $u_i$ specifies the amount of power that job $i$ uses during execution.*

Note that for the sake of simplicity, in this definition and throughout this chapter, all jobs have constant power demand during their execution. While this certainly is a simplification, we argue in sections 4.4.1 and 4.6 why it is probably an acceptable simplification for many industrial processes, and describe in Section 4.5 how our approach can easily be adapted to jobs with non-constant power demand, at the expense of computational complexity.

Next, we define the terms in which we talk about flexibility. Given a schedule as in Definition 1, two integers $\hat{T} \in \mathbb{N}$ and $\hat{J} \in \mathbb{N}$ are used to limit the amount of flexibility that may be created in the schedule. Here, $\hat{J}$ specifies how many jobs may be moved away from their original start times, and $\hat{T}$ specifies by how much time steps jobs may be moved in total.

With this, we can now define the problem examined throughout this chapter:

Flexibilization
Project
Scheduling
Problem

**Definition 2** (Flexibilization Project Scheduling Problem)**.**
*Given are a schedule as in Definition 1 and flexibilization limits $\hat{T}$ and $\hat{J}$.*
*Find for each job $i \in \{1, 2, \dots n\}$ a new start time $s_i \in \mathbb{N}$ such that*

- *the number of jobs $i$ for which $s_i \neq c_i$ is at most $\hat{J}$*
- *the total deviation from the current start times is at most $\hat{T}$, i.e.,*

$$\sum_{i=1}^{n} |c_i - s_i| \leq \hat{T}$$

Any $S$ which satisfies the conditions from Definition 2 is a *feasible* solution to the FPSP problem. Slightly abusing our notation, we also call such an $S$ a *schedule*. While specifying $\hat{T}$ and $\hat{J}$ of course means that to apply our framework one still has to specify these limits on flexibility, flexibility does not have to be specified on a per-job basis. Therefore it becomes easy to explore what improvements we can achieve at the "cost"

of what amount of flexibility — as we do in Section 4.4. Also, we outline in Section 4.8 how one can truly get rid of having an amount of flexibility as input parameter if one can estimate the costs of introducing new flexibility.

Note that so far we have only defined what a *feasible* solution is, not what makes a solution *optimal*. In fact, we explore different objective functions, the first of which models *peak shaving*. To do so, we need the total power usage at a certain point in time. For a feasible schedule $S$, let $U_t$ be the amount of power that is used during time step $t$, i. e.,

peak shaving

$$U_t = \sum \{u_i \mid s_i \leq t \wedge s_i + p_i > t\}.$$

**Definition 3** (Flexibilization Project Scheduling Problem with Peak Shaving (FPSP-PS))**.**
*For an instance of FPSP, let $\hat{U}$ be the maximum amount of power used by concurrently executing jobs throughout $S$:*

FPSP-PS

$$\hat{U} = \max_t U_t.$$

*The problem is then, find the feasible schedule that minimizes $\hat{U}$.*

While peak shaving is an objective that is relevant in real-world applications, it is often combined with available *generation*; the objective is to reduce the peaks of power demand which cannot be served by the generation. To model such objectives, we introduce an amount of generation for each time step: Let $G_t \in \mathbb{R}$ units of power be available in time step $t$. With this, we can define the next possible objective:

**Definition 4** (Flexibilization Project Scheduling Problem with Peak Shaving and Generation (FPSP-PSG))**.**
*Given an instance of FPSP and generation $G_t$, let $\tilde{U}$ be the maximum amount of power used by concurrently executing jobs throughout $S$ which cannot be met by own generation (the peak* residual *load)*

FPSP-PSG

$$\tilde{U} = \max_t (\max(U_t - G_t, 0)).$$

*The problem is then, find the feasible schedule that minimizes $\tilde{U}$.*

The third and last examined objective looks at *overshoot minimization*. In this setting we try to minimize the total amount of energy that can not be served by available generation:

overshoot minimization

**Definition 5.** *Flexibilization Project Scheduling Problem with Overshoot Minimization (FPSP-OM)*
*Given an instance of FPSP and generation $G_t$, find the feasible schedule that minimizes*

FPSP-OM

$$\sum_{t=0}^{\infty} \max(U_t - G_t, 0).$$

Note that the mixed-integer programming approach presented in Section 4.3.4 is able to optimize for all of these objectives. The modeling approach presented is based on the technique introduced in Chapter 3, where we also show how to adapt to various objective functions.

## 4.2 Related Work

The need for more flexibility in energy usage has been well established, e. g. by Taneja et al. [Tan14]. For an analysis of the possible benefits of demand response see for example Strbac [Str08] or Feuerriegel and Neumann [FN16].

The existing literature on demand response, in general, is vast and spans a significant area of applications and problems. For example, Gong et al. [Gon+15] investigate how DR can be used for households while still preserving their privacy, a profoundly important question which also results in the mentioned lack of support by industry to participate in such measures. Additionally, DR has been looked at not only for households but also for example for data centers e. g. by Klingert et al. [Kli+15]. However, we found the resulting schemes not to be transferable to industrial customers of the kind we consider. In contrast to our approach, Zehir et al. [Zeh+17] focus on getting small customers to participate in demand response, where the machines they can change are more related to that of households and not of manufacturing customers.

There are studies concerned with how much flexibility can be provided by the consumers (for example by D'hulst et al. [Dhu+15]), even on a device level (see e.g. Truong et al. [Tru+16]), and how much this flexibility is worth — for example by Pudjianto and Strbac [PS17], Feuerriegel and Neumann [FN16] or Ambrosius et al. [Amb+18]. However, we are not aware of anyone investigating what amount of demand side flexibility is necessary to improve the energy consumption significantly, especially not for industrial customers. Furthermore, many papers schedule flexible demands (for an extensive overview see Chapter 3) without looking into how many of those demands need to be changed by the scheduler to improve the energy pattern.

Peak minimization is one of the primary goals for many applications. For example, Liu et al. [Liu+13] and Zhao et al. [ZLC17] schedule loads to avoid specific peak demands, the first for data centers, the latter for electric vehicle charging. In our formulation, we focus on scheduling non-preemptive but deferrable loads, as do for example O'Brien and Rajagopal [OR15].

To find the patterns in the industrial load time series, we use a motif discovery technique. There exist other algorithmic techniques to find starting processes and monitor appliances, most prominently non-intrusive load monitoring, pioneered by Hart [Har92], advanced further for example by e. g. Ardakanian et al. [Ard+17] or Rollins and Banerjee [RB14]. However, these methods are not applicable in our case,

mainly due to the lack of labels in our data and thus the need to work without supervision.

Our approach also touches the field of project scheduling. Research in this area is vast, many variations of problem settings have been explored. For an extensive survey, see Węglarz [Węg99].

## 4.3 The Framework

In this section, we describe all steps of our proposed framework in the order in which we apply them. We start by describing the input data that we use for evaluation in Section 4.3.1. We then present a motif discovery algorithm which we use to detect the individual industrial processes from the usage data in Section 4.3.2. Not strictly part of our framework, but necessary for our evaluation is the generation of synthetic test data. We do this so that we can evaluate our approach on more than one data set. We explain the generation of synthetic data in Section 4.3.3. On the discovered processes (resp. our synthetic data), we run a model-based scheduling algorithm, which we present in Section 4.3.4.

For the sake of clarity, we introduce some terminology first. The initial data is a set of electrical consumption time series from *machines*, one time series per machine. We assume that a machine can either be running a *process* or be idle. The part of a machine's time series where a process is running is denoted a (time series) *sequence*. If several such sequences are similar, we assume they describe the same process. In this case, we say they are *occurrences* of the same *motif* (we explain this in more detail in Section 4.3.2). The mean motifs are then used to generate synthetic test data, which are sets of *instances*, each consisting of *jobs*.

### 4.3.1 Data

Our input data is the HIPE data set published recently by Bischof et al. [Bis+18], which is gathered at a small-scale electronics factory operated by the Institute for Data Processing and Electronics[1] at KIT. It is a set of time series of the apparent power in kVA of nine machines, which range from soldering furnaces to pick-and-place machines. The data was gathered over almost a year, from December 2016 to October 2017, with sub-minute resolution. See Section A.1 on how to get access to the raw data as well as the instances we create from the data set, as well as a more in-depth description of the raw data.

Some of the machines are frequently running in standby mode. Consequently, their power is above zero even while no process is running. To ease the analysis later and only consider the running processes without the standby times, we distinguish

[1] https://www.ipe.kit.edu/english/index.php

between an *active state* and a *passive state* of those machines. The active state means the machine is running a process, while the passive state means the machine is either off or in a standby mode. We determine the two states for each machine with the help of a $k$-means clustering algorithm, with which we cluster the power demands of each machine individually. Setting $k = 2$ leaves us with a cluster for each state. As we are only interested in the active state of the machine, we set the power demand to zero for all points in the passive cluster.

The points in time where a series goes from zero to non-zero power demand are the *start points* of a *sequence*. The sequence ends when the power demand goes back to zero, at its *end point*. To be able to compare sequences in a meaningful way later, we normalize the lengths of all sequences on a per-machine basis. For each machine, we determine the 80% quantile of the lengths of its sequences. We scale all sequences of this machine to this length for our motif discovery. We chose 80% because, on the one hand, stretching sequences comes with less data loss than compressing. On the other hand, taking the longest sequence would increase the impact of outliers.

### 4.3.2 Motif Discovery

The process with which we generate the instances that we test our approach on in Section 4.3.3 assumes as input a set of discovered patterns in the power demand time series described in Section 4.3.1. Such a frequently reoccurring pattern is also named a *motif*. The sequences that are associated with a certain motif are called the *occurrences* of the motif.

motif
occurrence

We do not go into detail here on how the motif discovery employed by us works. We use the approach suggested by Ludwig et al. [Lud+17] and refer the reader to that paper for a detailed description.

Figure 4.1 shows a discovered motif with all its occurrences in gray lines. The colored lines represent the mean, 20% quantile and 80% quantile of all the occurrences. These lines can give a feeling for how a process might look like for this machine. We show all discovered motifs in Section A.3.

### 4.3.3 Generation of Synthetic Instances

To evaluate the feasibility and usefulness of our proposed approach, we need many instances of the FPSP problem which emulate real-world processes. Therefore, we generate artificial instances for the FPSP problem from the motifs discovered as described in Section 4.3.2. In our generated instances, the start times, job durations and power requirements are statistically derived from the discovered motifs. These characteristics are key factors with regards to peak power demands during a schedule. Since we preserve these characteristics of the discovered motifs, we expect our generated instances to adequately resemble reality.

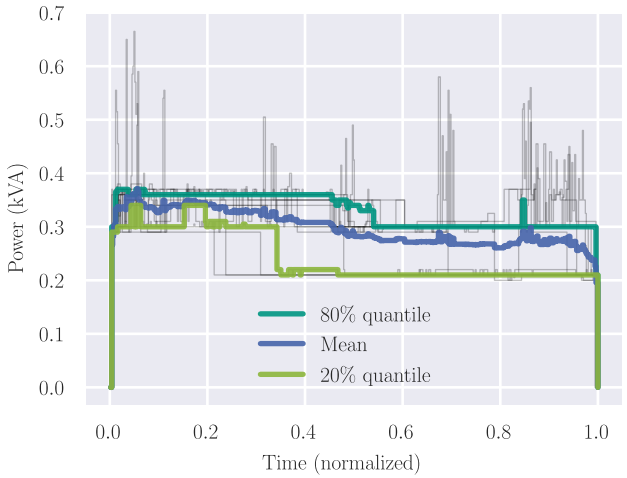



**Figure 4.1:** The discovered motif A. Each gray curve shows one occurrence, with the length of all occurrences being normalized to 1. Their shared common features, roughly represented by the mean and upper and lower quantiles, constitutes the motif.

To this end, we describe every motif by three normal distributions: One energy distribution, one start time distribution, and one duration distribution. Each of these distributions is determined by fitting a normal distribution to the lengths, start times and energy consumptions of the respective motif's occurrences. The start time distribution is actually a mixture of normal distributions: We assume that the same process might have several times within a day at which it usually starts. To factor this in, we cluster the start times of every motif's occurrences (using affinity propagation) and generate a normal distribution for every cluster. We then create a mixture distribution for that motif's start times, weighting each normal distribution by the size of its cluster. However, for most motifs, only a single cluster was found.

We generate instances (i. e., sets of jobs) with a fixed number of jobs by repeatedly randomly selecting a motif (weighted by the number of the motif's occurrences) and then generating a job for this motif. For each job, we generate a duration by randomly drawing from the respective motif's length distribution. We discard any length of less than one time step. Similarly, we determine a start time for this job by randomly drawing from the respective motif's start time distribution. Finally, we randomly generate an energy consumption for the job by drawing from the motif's energy distribution. The job's power demand is set as energy consumption divided by duration.

Since drawing values from normal distributions may yield extreme results in few cases, which nonetheless are sufficient to substantially skew the complexity and results

of the scheduling problem, we discard any values that deviate from the mean by more than three times the standard deviation.

### 4.3.4 Scheduling

We now describe a mixed integer program (MIP) that models and optimizes the FPSP problem. We base our MIP on the modeling technique from Chapter 3, which is intended for classic smart grid scheduling problems, i. e., assumes jobs given with fixed flexibility, expressed in terms of earliest starts, deadlines, etc. The approach is able to model many real-world processes' features encountered in literature, such as ramping, energy drain or interdependent jobs. We only give a very rough overview over the most important parts of the technique, and then describe how to extend it to cope with the FPSP problem.

The MIP technique in Chapter 3 is based on a discrete-time formulation for the Resource-Constrained Project Scheduling Problem. At the core of the model, a binary variable is created for every job and every time step at which this job could start. For every job, exactly one of its start indicator variables must be one, fixing the start time of the respective job. The start time variable of a job $i$ is $\sigma_i$, and its finishing time is $\eta_i$. Variables for the power usage at every time step can be built using the start indicator variables.

The foundation for our modification of this technique are *window extensions*. These are a way of expressing the current start times $c_i$ and the flexibilization limits from FPSP in terms of a "classic" project scheduling problem, which works with release times and deadlines instead.

window extension

Instead of every job having a desired start time $c_i$, we assume every job $i$ to have a release time $r_i$ (i. e., an earliest start time) and a deadline $d_i$ (i. e., a time when the job must be finished).

We transform an FPSP problem by first setting $r_i = c_i$ and $d_i = c_i + p_i$ for all jobs $i$. This way, we get a problem that we can immediately plug into the MIP technique from Chapter 3, but in which every job $i$ is forced to be started at its current start time $c_i$, because we have made its *window* (the time between release and deadline) small enough. In a second step, we now extend the MIP to allow for some jobs to be executed outside their window, i. e., to *extend* their window. This window extension is tailored such that it honors $\hat{T}$ and $\hat{J}$ of the FPSP instance.

window

In the MIP framework, the jobs' windows are enforced with the simple constraints[2]

$$\begin{aligned} \sigma_i &\geq r_i \qquad \forall i \\ \eta_i &\leq d_i \qquad \forall i. \end{aligned}$$

[2]Note that in Chapter 3, deadline and release are denoted as $D_i$ and $R_i$ instead of $d_i$ and $r_i$.

We introduce two new variables per job $i$, namely $\overleftarrow{x_i}$ and $\overrightarrow{x_i}$, the *left window extension* and *right window extension* of job $i$, respectively. We then change the aforementioned constraints to

$$\sigma_i \geq r_i - \overleftarrow{x_i} \qquad \forall i \tag{4.1}$$

$$\eta_i \leq d_i + \overrightarrow{x_i} \qquad \forall i. \tag{4.2}$$

The additional constraints to uphold $\hat{T}$ are simple

$$\overleftarrow{x_i} \geq 0 \quad \wedge \quad \overrightarrow{x_i} \geq 0 \qquad \forall i, \tag{4.3}$$

$$\sum_{i=1}^{n} \left( \overleftarrow{x_i} + \overrightarrow{x_i} \right) \leq \hat{T}. \tag{4.4}$$

To also uphold the job move limit $\hat{J}$, we need to introduce a binary variable indicating that a job was *not* moved. We will call this variable $\tilde{c}_i$. We force $\tilde{c}_i$ to become zero if job $i$ was moved with this constraint

$$\tilde{c}_i \in \{0, 1\} \qquad \forall i, \tag{4.5}$$

$$\tilde{c}_i \leq 1 - \frac{\overleftarrow{x_i} + \overrightarrow{x_i}}{\hat{T}} \qquad \forall i. \tag{4.6}$$

With this, it is easy to limit the number of moved jobs

$$\sum_{i=1}^{n} \tilde{c}_i \geq n - \hat{J}. \tag{4.7}$$

Note that with constraint 4.6, we get $\hat{T}$ as a coefficient in the constraint matrix, potentially giving us constraint coefficients of greatly varying magnitude. This can cause problems for the numeric stability of the resulting MIP model, as we further discuss in Section 4.4.4.

### Modeling further Constraints

Real-world scenarios most likely require more constraints to be placed on jobs than what we model in this chapter. However, since the MIP framework from Chapter 3 is very flexible, many additional constraints are easy to include. As an example, the MIP framework handles dependencies between jobs (in the form of *time lags*), energy drain for postponed jobs, and hard release times and deadlines.

### Modeling individual Move Costs

In realistic scenarios, moving some jobs in time might be way more effort than moving other jobs. The model can account for this with a slight modification: We introduce a weighting factor $w_i \in \mathbb{R}$ for every job $i$. We then modify Constraint 4.4 to:

$$\sum_{i=1}^{n} w_i \left( \overleftarrow{x_i} + \overrightarrow{x_i} \right) \leq \hat{T} \tag{4.8}$$

This way, moving some jobs counts stronger towards the $\hat{T}$ limit than others.

### Modeling Fluctuating Power Demands

As noted in Section 4.3.3, all jobs are assumed to have constant power demand. However, the MIP used technique can approximate jobs with non-constant power demand. This approximation is accomplished by using the feature of time lags mentioned above, i.e., constraints on the order in which jobs are scheduled. We defer a detailed description of this technique to Section 4.5.

## 4.4 Evaluation

Having introduced our framework and the data with which we work, we now evaluate our approach. We start with describing the motifs we find, and the instance sets generated, before assessing each of our problem sets individually.

### 4.4.1 Discovered Motifs

Given the nine machines, we find a total of 15 motifs in our time series data. The resulting motifs for each machine can be found in Figure A.8 in the appendix. We have also included an overview over our parameters used in Table A.1 in the appendix. As we can see there, most of the motifs are block-shaped. This fact gives us reason to believe that many processes can be adequately approximated by our assumption of constant power demand. For an in-depth discussion of this assumption see Section 4.6. The parameters for the motif discovery algorithm in this chapter are tailored to our specific problem at hand. For example, we use a relatively small alphabet size for most machines as their variations are small. We also choose all parameters in such a way that the algorithm can classify most sequences without assigning all of them to a single motif. All sequences which are not classified as belonging to one of the motifs are classified as noise and excluded from further analysis. In future work, we might want to do an extensive evaluation of our parameter choices. However, we expect the settings we have chosen to be sufficiently useful for our problems at hand.

**Table 4.1:** Statistics of $\hat{T}$ values for all possible values of $\Theta$.

| | $\hat{T}$ (hours) | |
|---|---|---|
| $\Theta$ | Mean | Std. Dev. |
| 0.005 | 3.6 | 0.20 |
| 0.01 | 7.2 | 0.39 |
| 0.02 | 14.3 | 0.77 |
| 0.03 | 21.5 | 1.16 |
| 0.04 | 28.7 | 1.54 |

### 4.4.2 Instance Sets

We generate several sets of instances from the data as described in sections 4.3.1 through 4.3.3, resembling the input data from Section 4.3.1 to varying degrees. We publish all the instance sets together with a description of the data format, see Section A.1 for how to obtain them.

**Set PS-Nonuniform.** This set is the first set of instances with which we evaluate FPSP-PS. We proceed like described in Section 4.3.3: Job lengths, job power demands and job start times are generated from the normal distributions fitted to the discovered motifs. Since in realistic scenarios, the optimization horizon likely is more than one day, we generate instances that span five days. The start times of our discovered motifs are times within a day. Thus, for every job, we not only pick a start time from the motif's start time distribution but also pick uniformly at random at which of the five days the job starts. One time step corresponds to five minutes. We generate instances with 150 jobs each.

Regarding the possible choices for $\hat{T}$, the amount of total job movement allowed, it seems reasonable to specify $\hat{T}$ relative to the instance size. We therefore introduce $\Theta$ and set $\hat{T} = \Theta \cdot \sum_i p_i$. Here, $\Theta$ specifies the fraction of cumulative duration that jobs may be moved. We test all of $\Theta \in \{0.005, 0.01, 0.02, 0.03, 0.04\}$. Table 4.1 shows statistics about the values that result for $\hat{T}$ for the various values for $\Theta$. We see that the values for $\hat{T}$ range from about 3.5 hours to about 30 hours.

For $\hat{J}$, the number of jobs allowed to be moved, we investigate all of $\hat{J} \in \{3, 6, 9\}$, for a total of 15 different flexibilization limits. We generate 30 sets of 150 jobs each, and pair them with every flexibilization limit from above, leading to a total of 450 instances.

**Set PS-Uniform.** As we see in our evaluation (see Section 4.4.4), the heterogeneity in power demand between the generated jobs (arising from heterogeneous power

demand in the discovered motifs) has a significant influence on the computational feasibility and the possible optimization benefits of the instances. We do not wish to bias our conclusions on the basis of this phenomenon, which may not occur in other workloads. Hence, we generate the PS-Uniform instance set in which we use a single, fixed normal distribution for all jobs' power demands (with mean 30 and standard deviation 10). Aside from this, we proceed as for the PS-Nonuniform set.

**Set PSG.** In the PSG set, we again explore peak shaving, but with fluctuating generation. This set corresponds to a setting where e. g. solar generation is available and one tries to minimize the peak residual load, which corresponds to FPSP-PSG.

We use a solar generation curve for one day derived from total solar generation data for Germany, Austria, and Luxembourg with quarter-hourly time resolution, which was retrieved from ENTSO-E.[3] For every quarter hour, we average the production from all summer days in the year 2016. In our instances, we set available generation (i. e., $G_t$) on all five days based on this curve, scaling the curve such that in total, 20% of the total energy demand in each instance is provided via solar energy. Aside from that, we generate instances as described for the PS-Nonuniform set.

**Set OM.** With the OM set, we evaluate an overshoot minimization objective, i. e., the FPSP-OM problem.

For the available generation, we assume that we can meet 65% of the total energy requirement in each instance by own generation. This number results from our calculations of the energy consumption and production in the summer month of BASF based on Hagenmeyer et al. [HLH14]. Thus, we assume this to be a realistic figure for a large-scale chemical plant. As BASF's power plants are steam-controlled, the generation in the winter months is so high that they can sell excess energy. However, during the summer months, the opposite is true, and they have to buy energy from the grid, which is more expensive. We assume the generation to be a flat curve in our calculations. For a steam-controlled power plant and a time horizon of five days, this is a realistic assumption since steam demand usually fluctuates relatively little. Formally, we set

$$G_t = \frac{\sum_i (p_i \cdot u_i) \cdot 0.65}{5 \cdot 24 \cdot 60/5} \qquad \forall t.$$

In this calculation, the denominator comes from the number of five-minute intervals in the five-day scheduling horizon. For this instance set, we use the objective from FPSP-OM, but proceed as for the PS-Nonuniform set otherwise.

[3] Via Open Power Systems Data:
`https://data.open-power-system-data.org/time_series/`

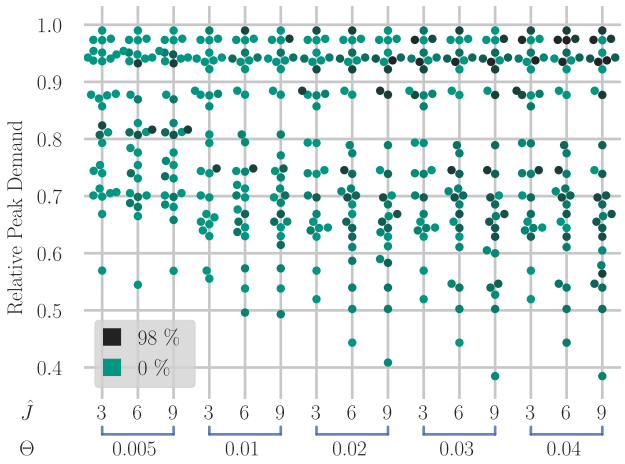



**Figure 4.2:** Relative reduction in peak demand for the PS-Nonuniform set, one point per instance. The columns are the different settings for $\hat{J}$ and $\Theta$. The Y-axis indicates the change in peak demand after optimization. Color indicates the remaining MIP gap when the optimization was stopped. Results are summarized in Table A.5.

### 4.4.3 Evaluation Environment

For all instances, we build the MIP models according to Section 4.3.4. We optimize every model using Gurobi 7.0. We also evaluated optimizing using CPLEX 12.8, but we achieve slightly better results using Gurobi for our models. We use a system with dual AMD EPYC 7601 CPUs (each having 64 physical CPU cores) with 512 GB of RAM. We optimize every model for 45 minutes, allowing for 20 threads per solver, running 6 solvers in parallel.

MIP gap

**MIP Gap.** In the following, we report the *MIP gaps* after optimization together with the solution quality. The MIP gap is the relative difference between the best feasible solution found and the best lower bound that the solver was able to prove, normalized by the best feasible solution. A large MIP gap is an indicator that further optimization could potentially find better solutions. Please note that we optimize with a focus on finding high-quality solutions.[4] Thus, we could have further reduced MIP gaps at the cost of solution quality. We performed parameter tuning via Gurobi's auto-tuning tool. However, the default settings produced the best results for us.

### 4.4.4 Evaluation of FPSP-PS and FPSP-PSG

We start by looking at the sets that represent peak shaving objectives. Figure 4.2 summarizes our results for the PS-Nonuniform set. Every dot represents one instance. Every column of dots represents one combination of $\hat{J}$ and $\Theta$. The $y$-coordinate of the dot indicates the respective instance's peak demand after optimization divided by the peak demand before optimization, which we define as *relative peak demand*. We use the dots' colors to indicate the MIP gap achieved for the respective instance, where darker colors indicate larger MIP gaps, i. e., worse optimization, and the lightest color indicates that the instance was solved to optimality. Table A.5 in the appendix reports numerical results. In Figure 4.2 we see that in the most extreme cases, peak demand is reduced by more than 60%. We also see that the improvement is mostly distributed evenly between 0% and about 40%, and that increasing $\hat{J}$ or $\Theta$ does not result in significant improvements. Figures A.4b and A.4a (in the appendix) give an insight into how the peak demand for every instance changed when increasing $\Theta$ or $\hat{J}$, respectively. We find that increasing $\Theta$ yields larger peak reductions than increasing $\hat{J}$, and that improvements gradually diminish with larger values for $\Theta$ (resp. $\hat{J}$). However, the optimization gets harder with increasing $\Theta$. Since the reported MIP gaps after optimization were large for many instances for $\Theta \in \{0.03, 0.04\}$, results may improve for those parameters if one optimizes the instances further. Figure A.1 (in the appendix) shows the MIP gaps of every instance.

relative peak demand

We perform a statistical significance test (Wilcoxon's signed-rank test[5]) on our findings. For every consecutive pair of $\hat{J}$ (resp. $\Theta$) values, while keeping the $\Theta$ (resp. $\hat{J}$) value fixed, we compute the $p$-value, which indicates how likely it is that the change in improvements is random happenstance instead of an effect of altering $\hat{J}$ (resp. $\Theta$). We report all values in Table 4.2.

We want to assume significance with 95% confidence, i. e., say that a change is significant if the $p$-value is below 0.05. However, throughout the whole of this chapter, we perform a total of 88 such tests. With an error probability of 5%, we thus expect erroneously reporting significance for four tests. To compensate for this, one can apply a Bonferroni correction, which essentially means assuming significance only when the $p$-value is below $0.05/88 \approx 0.00057$.

We see that changing $\Theta$ from 0.02 to 0.03 and 0.04 does probably not result in significant improvements for the PS-Nonuniform set. This lack of improvement might be because the optimization problem becomes too hard, but could also be because we already achieve optimal results for many instances with $\Theta = 0.02$ (see below). Aside from that, the only non-significant change is changing $\hat{J}$ from 6 to 9 at $\Theta = 0.005$ and

[4] We set the MIPFocus parameter to 1 for Gurobi.

[5] Because we have many ties in our data, the way such ties are handled is important in our case. We use the approach suggested by Pratt [Pra59].

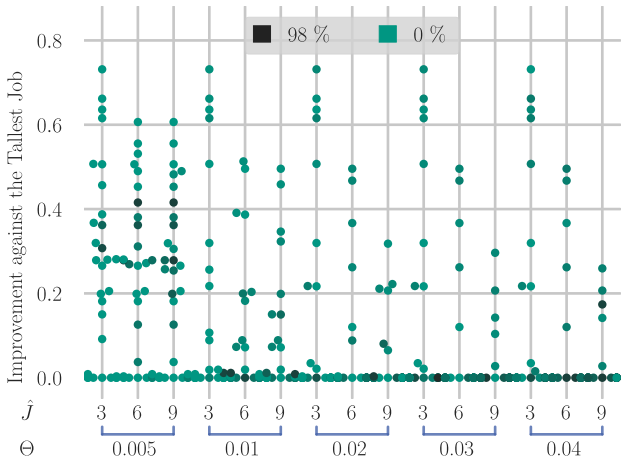



**Figure 4.3:** Change of the difference between the peak demand and the demand of the tallest job after optimization in the PS-Nonuniform set. The $y$ value is determined by taking the difference between peak demand and demand of the tallest job after optimization, and dividing it by the same value before optimization.

$\Theta = 0.04$, which is like because the little possible movement in time can already be optimally distributed over six jobs, resp. because the instances got too hard.

The very uniform distribution of improvements between 0% and 40% poses the question for validity, hence we looked into characteristics of the instances that allow for an unusually large or small improvement. We discovered that instances which allow for almost no improvement always contain jobs with very high power demands compared to all other jobs' power demands, i. e., substantial heterogeneity in power demands. This makes sense, since the job with the largest power demand (which we call the *tallest* job) is a lower bound for the overall peak power demand. The margin for optimization is at most the difference between the overall peak power demand and the demand of the tallest job.

tallest job

We therefore also evaluate how well our optimization performs within this margin, i. e., how the difference between peak demand and demand of the tallest job changes, which we define as the *improvement against the tallest job*. Figure 4.3 shows the results. A value of 0 indicates that after optimization, the overall peak power demand equals the demand of the tallest job. Figure 4.3 shows that for the majority of instances, we are in fact able to achieve this optimum. For all other instances, we improve by at least 20% within the margin between tallest job and original peak demand.

Since the peak demand being dominated by single jobs seems like a peculiar property of our instances, we create the PS-Uniform set, where we compensate for this characteristic. Regarding the results for the PS-Uniform set, shown in Figure 4.5 (numerical

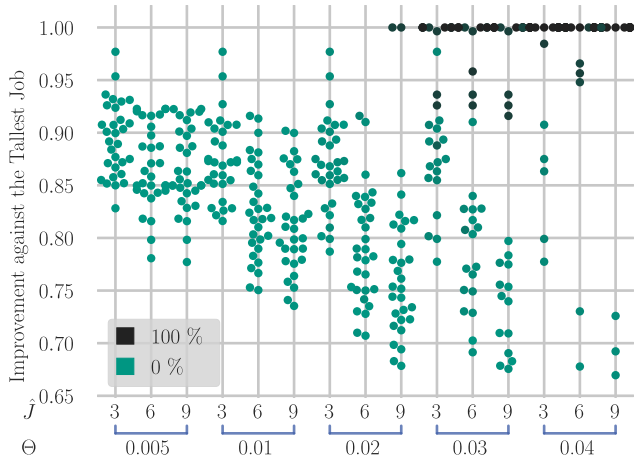



**Figure 4.4:** Change of the difference between the peak demand and the demand of the tallest job after optimization in the PS-Uniform set. The $y$ value is determined by taking the difference between peak demand and demand of the tallest job after optimization, and dividing it by the same value before optimization.

**Table 4.2:** $p$-Values for the change of one parameter in the PS-Nonuniform set. Values highlighted in green indicate that changing one of $\hat{J}$ and $\Theta$, while keeping the other one constant, results in a statistically significant change in improvements. Values in blue are significant only before Bonferroni correction.

| | 3 | → | 6 | → | 9 |
|---|---|---|---|---|---|
| 0.005 | | $< 10^{-4}$ | | 0.00089 | |
| ↓ | $< 10^{-4}$ | | $< 10^{-5}$ | | $< 10^{-5}$ |
| 0.01 | | $< 10^{-4}$ | | 0.00051 | |
| ↓ | $< 10^{-4}$ | | $< 10^{-4}$ | | $< 10^{-5}$ |
| 0.02 | | $< 10^{-4}$ | | 0.00039 | |
| ↓ | 0.0018 | | 0.014 | | 0.0032 |
| 0.03 | | 0.00029 | | 0.00014 | |
| ↓ | 0.012 | | 0.047 | | 0.035 |
| 0.04 | | $< 10^{-4}$ | | 0.0018 | |

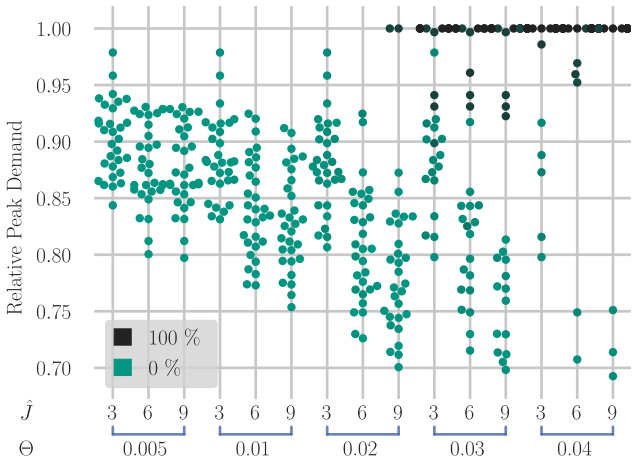



**Figure 4.5:** Relative reduction in peak demand for the PS-Uniform set, one point per instance. Columns are the different settings for $\hat{J}$ and $\Theta$. The Y axis indicates the change in peak demand after optimization. Color indicates how well the instance could be optimized. Results are summarized in Table A.6.

results in Table A.6 in the appendix), we see that even with power demands being drawn from a single distribution, peak demand reductions of 5% to 30% are realistic. We also see that solving these instances is a lot harder than the instances from the PS-Nonuniform set. We report the MIP gaps in Figure 4.6. Here, an interesting trend can be seen: While almost all instances for $\Theta \leq 0.02$ could be optimized to within 20% gap, often to optimality, the MIP gaps for $\Theta \geq 0.03$ are mostly above 60%. Thus, it seems like the computational complexity grows rapidly with $\hat{T}$ — this is certainly because of the larger solution space, but might also be exacerbated by the numerical stability issues with constraint (4.6) mentioned in Section 4.3.4.

The results for the PS-Uniform set are more tightly clustered, which makes sense since the instances are more similar to each other. For this set, we also evaluate how well we optimize within the margin between overall peak power demand and demand of the tallest job, which we report in Figure 4.4. Since in PS-Uniform, the peak power demand is not dominated by single jobs anymore, this now correlates closely with the absolute improvement. We again perform significance analysis as for the PS-Nonuniform set, the results of which we report in Table A.2 (in the appendix). We can see that some of the changes that were not significant for PS-Nonuniform, especially increasing $\Theta$, have now become significant. This change supports the assumption that the non-significance for the PS-Nonuniform set in these cases is because optimal values have already been achieved for $\Theta = 0.02$.

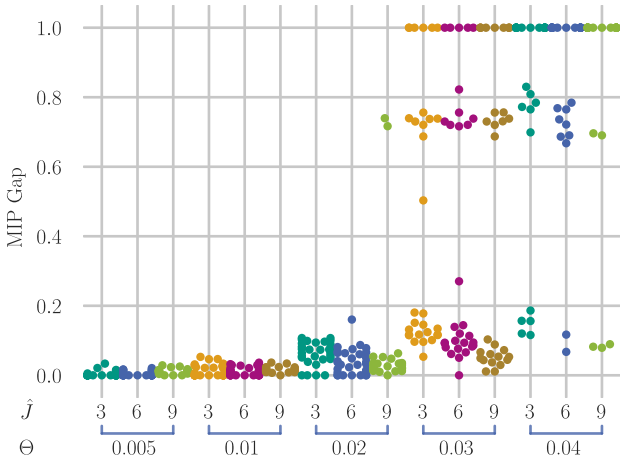



**Figure 4.6:** MIP gaps for the various settings of $\hat{J}$ and Θ in the PS-Uniform instance set. Every dot corresponds to one instance. Colors are used to distinguish the columns.

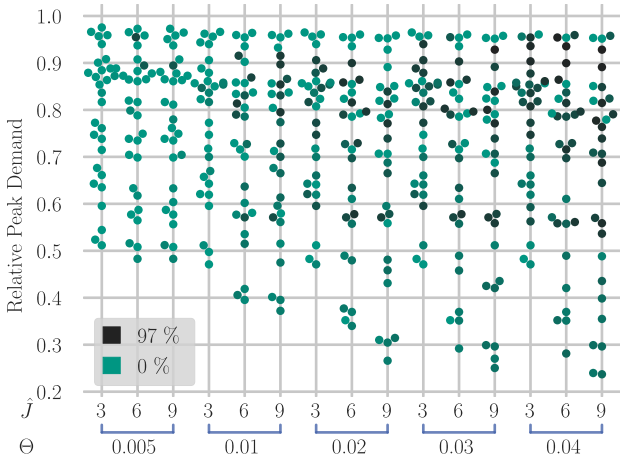



**Figure 4.7:** Relative reduction in peak demand for the PSG set, one point per instance. The columns are the different settings for $\hat{J}$ and Θ. The Y axis indicates the change in peak demand after optimization. Color indicates the remaining MIP gap when the optimization was stopped. Results are summarized in Table A.7.

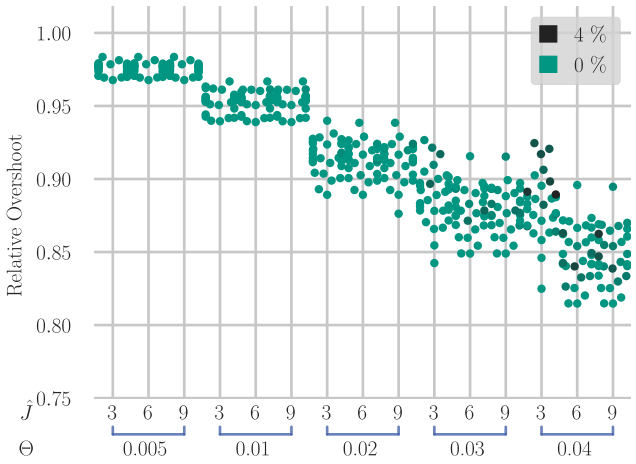



**Figure 4.8:** Results for set OM, one point per instance. The columns are the different settings for $\hat{J}$ and $\Theta$. The Y axis indicates the change in overshoot after optimization. Color indicates how well the instance could be optimized. Results are summarized in Table A.8. For better readability we removed one outlier at $\Theta = 0.01, \hat{J} = 3$ with a value of ca. 0.5.

The final instance set about peak shaving is the PSG set, in which we assume solar generation. We report the improvements in Figure 4.7 and numerical results as well as the effect of changing both parameters in Table A.7 and Figure A.6 (both in the appendix). We see that the distribution of improvements is similar to the PS-Nonuniform set. We can assume that tall jobs again have a large impact in this instance set. However, the absolute values regarding improvement are much better than for PS-Nonuniform: In most extreme cases, we are able to reduce the residual peaks by almost 80%. Most improvements are evenly distributed between 5% and 50%. This seems plausible since in the PSG set, (residual) peak reduction can not only be achieved by avoiding concurrent execution of jobs, but also by moving jobs towards the peaks of the solar generation curve.

We report MIP gaps in Figure A.2 in the appendix, which are not worse than for the PS-Nonuniform set. Significance values are reported in Table A.3. We see that almost all changes in parameter choice lead to significant improvements. Overall, our approach seems to be able to exploit the benefits of a given generation curve without increasing the computational complexity.

### 4.4.5 Evaluation of FPSP-OM

We evaluate the overshoot minimization objective from FPSP-OM with the OM instance set, the results of which we report in Figure 4.8 and Table A.8 (in the appendix). We see strong clustering of the results, indicating that with, e. g., $\Theta = 0.02$, one can expect

the amount of energy to overshoot generation, i. e., the amount of energy that must be bought from the grid, to decrease between 6% and 12%. For FPSP-OM, apparently $\hat{T}$ plays a crucial role, while $\hat{J}$ yields only minor improvements. These observations can be seen from figures A.7a and A.7b. Computational complexity increases slightly with decreasing $\hat{J}$. However, the reported MIP gaps (see Figure A.3 in the appendix) are drastically smaller than for our peak shaving sets. We solve all instances to at most 4% MIP gap, and in fact, solve most of them to optimality. We again do a significance analysis, reported in Table A.4. Here, we see that almost all parameter changes result in significant improvements.

## 4.5 Modeling Fluctuating Demand via Job Chains

In this section, we give some details on how the MIP model presented in Section 4.3.4 can be extended to better approximate real-world processes. The MIP modeling technique from Chapter 3, which we build upon in this chapter, allows to specify what is called *minimum time lags* between two jobs $i$ and $j$. This time lag $L_{i,j}$ determines the number of time steps that must pass between the start of $i$ and the start of $j$. Together with the start time variable $\sigma_i$ for every job $i$, this results in constraints of the form

$$\sigma_j \geq \sigma_i + T_{i,j} \qquad \forall\ i, j.$$

An interesting aspect is that the $T_{i,j}$ may be negative. Using this, we can form fixed chains of jobs. Say we have three jobs 1, 2 and 3, which all have processing time ($p_i$) of 1. If we set $T_{1,2} = 1$, $T_{2,3} = 1$ and $T_{3,1} = -2$, it must hold that

$$\begin{aligned}\sigma_2 &\geq \sigma_1 + 1,\\ \sigma_3 &\geq \sigma_2 + 1,\\ \sigma_1 &\geq \sigma_3 - 2.\end{aligned}$$

Which results in

$$\begin{aligned}\sigma_2 &= \sigma_1 + 1,\\ \sigma_3 &= \sigma_2 + 1.\end{aligned}$$

We can now assign different power demands to each of the three jobs. Such a situation is sketched in Figure 4.9, where the three rectangles represent the three jobs: The width of each rectangle is the job's duration and the height of the rectangle is the job's power demand. We see that the blue curve, which might represent the power

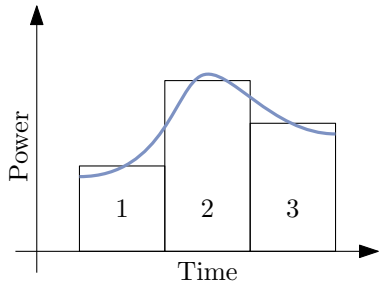



**Figure 4.9:** A fixed chain of three jobs 1, 2 and 3, which are used to approximate a job that has the blue demand curve.

demand curve of some industrial process, can be approximated more closely by this chain of jobs than if we only used one job.

When using this approach, one must pay attention to correctly encode the $\hat{T}$ / $\hat{J}$ limits in the instance. In Section 4.3.4, we fix every job at its start time (by setting release and deadline correctly). When using job chains, only one of the jobs of each chain must be fixed like this. Otherwise, moving a chain of $k$ jobs by $t$ time steps would count as $k$ jobs regarding $\hat{J}$ and would contribute $k \cdot t$ units to $\hat{T}$. However, fixing only one job's window also fixes all other jobs in the chain.

It is important to note that this approach usually leads to a significant increase in the computational complexity of the resulting model. Tweaking the model such that this technique becomes feasible for large instances is beyond the scope of this chapter.

## 4.6 Motif Analysis

We now analyze the discovered motifs further, especially with regard to the question whether jobs with constant power demand are a reasonable approximation of the motifs, and if not, how much better the approximation becomes when we allow to split the jobs into multiple blocks as outlined in Section 4.5. Note that all discovered motifs are presented in Figure A.8 in the appendix.

The occurrences of motifs correspond to stepwise functions: Every point in the occurrence's power demand time series results in one step in its power demand function. Let $o$ be an occurrence. We then call the (stepwise) function mapping a point in time to the power demand of the occurrence at that time $P_o \colon [0, 1] \to \mathbb{R}$ (note that occurrences are normalized, thus a point in time is in $[0, 1]$). The main question is how well we can approximate these functions with other stepwise functions *of low complexity*, i. e., with few steps. Note that a job with a constant power demand corresponds to a stepwise function with exactly one step, a chain of two jobs corresponds to a stepwise function with up to two steps, and so on. Let $\tilde{P}_{o,k} \colon [0, 1] \to \mathbb{R}$ be such a function with at most $k$ steps, which tries to approximate $P_o$.

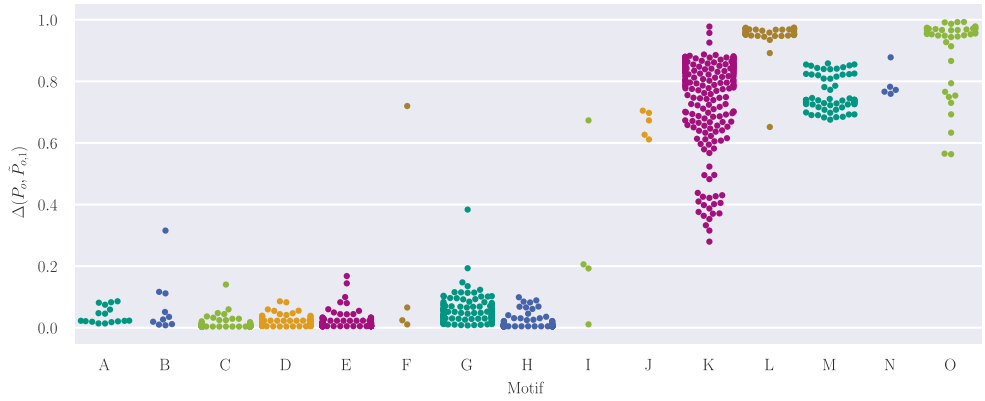



**Figure 4.10:** The difference between all occurrences' $P_o$ an and their respective optimal $\tilde{P}_{o,1}$

We need some notion of the difference between $P_o$ and $\tilde{P}_{o,k}$. We suggest the following metric:

$$\Delta\left(P_o, \tilde{P}_{o,k}\right) = \frac{1}{N_o} \int_0^1 \left(P_o(t) - \tilde{P}_{o,k}(t)\right)^2 \mathrm{d}t.$$

Here, $N_o$ is a normalization factor to make different motifs comparable, calculated as $N_o = \int_0^1 P_o(t)^2 \,\mathrm{d}t$. We can compute the value of the integral without actually integrating, since both functions are discrete in $t$. This metric penalizes deviations of $\tilde{P}_{o,k}$ from $P_o$ with a quadratic term. We assume a deviation that is large in magnitude but short in time to be worse than a deviation which is small in magnitude but long in time, because deviations of large magnitude might hide exactly the peaks in power demand that we are interested in reducing.

To analyze how block-shaped our motifs really are, we explore different values for $k$, i.e., the number of blocks to decompose the occurrences into. For each $k$ and each occurrence, we fit[6] a stepwise function $\tilde{P}_{o,k}$ to the $P_o$ of the respective occurrence $o$. Our first attempt is $k = 1$, i. e., a step function with exactly one step, representing jobs with constant power demand. We see the value of $\Delta(P_o, \tilde{P}_{o,1})$ in Figure 4.10. The $x$ axis groups the occurrences by their motif. We sort motifs by how well they are approximable for $k = 1$.

We cannot say what values for $\Delta(P_o, \tilde{P}_{o,k})$ are good or bad: The question of what is an acceptable approximation must be answered by the person using our framework. However, we can clearly see that nine of our fifteen discovered motifs are a lot better approximated by a job with constant power demand than the remaining six. This seems intuitively correct when looking at the motifs in Figure A.8.

[6] Using a black-box SLSQP optimizer.

We can also see how $\Delta(P_o, \tilde{P}_{o,k})$ changes when we go from $k = 1$ to $k = 2$, which is shown in Figure 4.11a. We see that the change is substantial for the six motifs that were not well approximated before. Especially motifs L, N and O seem to profit from a two-step function. When looking at these motifs in figures A.8l, A.8n and A.8o, that seems plausible.

We see a similar effect when going to $k = 5$ (see Figure 4.11b) and $k = 10$ (see Figure 4.11c): The $\Delta(P_o, \tilde{P}_{o,k})$ values shrink gradually for the six motifs which are not very block-shaped, although the improvement is less than for going from $k = 1$ to $k = 2$.

We can thus conclude that the technique proposed in Section 4.5 has benefits: Being able to approximate the motifs with stepwise functions of more than one step most likely brings the results of the optimization closer to reality. Especially allowing for two jobs instead of one might be a worthwhile option. However, we can also conclude that a constant function is already a good approximation for the majority of the discovered motifs, and is not completely outlandish for the rest of the motifs either.

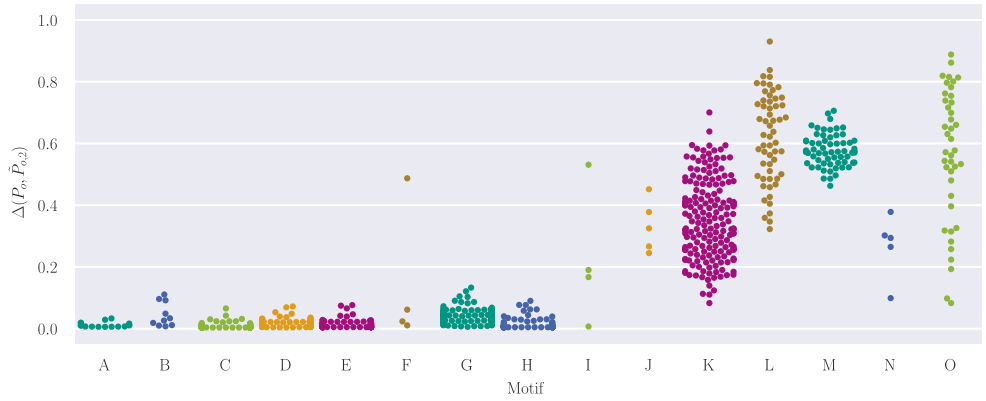
## 4.7 Discussion

For all examined variations of the FPSP problems, we can show that with a relatively small amount of flexibility, significant improvements in the target metric can be achieved. Since all our test data is founded on real energy consumption data obtained from a factory, we assume our results to apply to real-world scenarios. However, real industrial processes come with more constraints than we were able to respect within the scope of the present chapter. Since our optimization is based on an MIP framework (which supports additional constraints such as process dependencies etc.), many additional constraints should be straightforward to model.

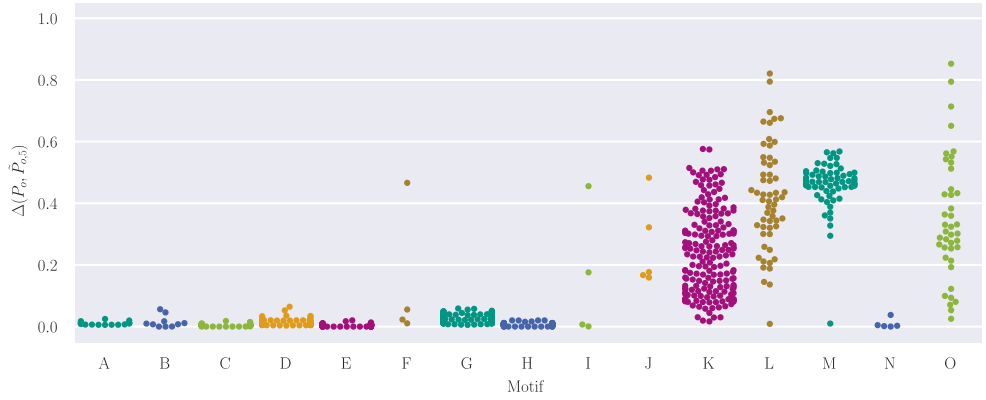
We discovered that the possible improvements depend a lot on the heterogeneity of the process' power demands. However, even for the heterogeneous instance sets derived from our real-world data, possible improvements were promising.

For the problem variants that assume available generation (FPSP-PSG and FPSP-OM), we need to choose the amount of generation. While we could obtain a solar generation curve from real-world data, we need to fix the total amount of energy available via generation somewhat arbitrarily. However, we have no reason to believe that our approach works significantly better or worse if we choose this amount differently. We were able to show that our approach is in fact suitable to reduce the peak residual demand as well as the amount of energy that must be bought from the grid.

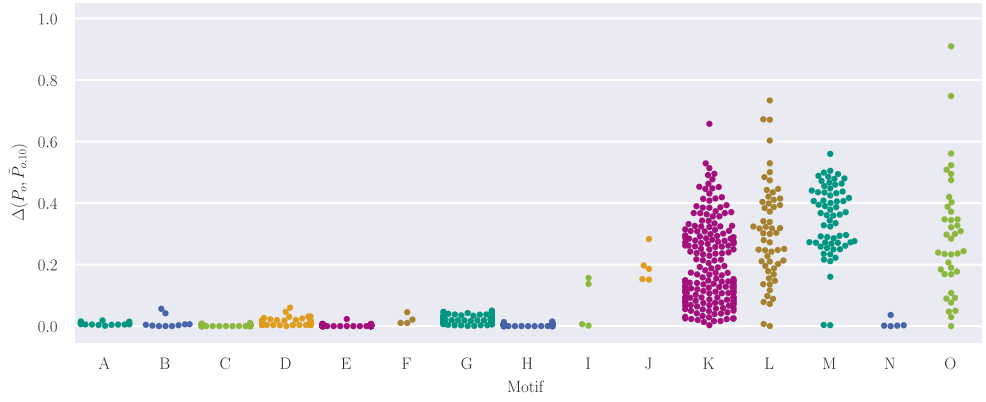
Another discrepancy between our test instances and real-world processes is the fact that we assume the power demand of each process to be constant over time. However, we have argued in sections 4.4.1 and 4.6 why this is probably a reasonably close approximation of the motifs we discovered, which is why we believe that we can




**(a)** $k = 2$




**(b)** $k = 5$




**(c)** $k = 10$

**Figure 4.11:** The difference between all occurrences' $P_o$ and and their respective optimal $\tilde{P}_{o,k}$

even use constant-demand jobs as an approximation for many real industrial processes. Furthermore, we have shown in Section 4.5 how to get around this limitation.

### 4.7.1 Optimization Aspects

Regarding the computational complexity of optimizing the MIPs of our approach, we can see that the most important parameter (besides the instance size) is $\hat{T}$. This is likely because for a large $\hat{T}$, the MIP formulation needs to create many start-indication variables ($s_i(t)$ in the original framework), quickly increasing the size of the underlying MIP model. Also, the numerical stability of the model probably suffers from large $\hat{T}$ values, as noted in Section 4.3.4. We also discover that substantial heterogeneity of power demands, while decreasing possible peak improvements, is beneficial for the complexity of the optimization problem.

Given all that, most of the time we can optimize instances of realistic sizes, spanning a whole working week with five-minute resolution, within 45 minutes to acceptable MIP gaps. We therefore think that our approach can not just be beneficial in reality, but can also be applied to realistically sized problems.

## 4.8 Conclusion & Outlook

We have shown a technique that can be used to guide flexibilization efforts in industrial processes. Our technique starts with consumption data obtained from the current operation of an industrial plant, and it ends with an indication which processes would be most beneficial if they were more flexible. We have shown that our technique can lead to significant improvements and should be applicable in real-world industrial processes.

We started this work promising to provide a tool that can suggest how much flexibility should be created in a plant's processes to achieve a certain level of power demand optimization. Our results indicate that there is most likely no need to flexibilize all processes. Starting with only a few small changes in the operation of the machines can already improve the energy consumption. This comparatively little necessary effort gives us hope that more flexibility in industrial processes is achievable and not a daunting prospect for any process manager. On the other hand, future work is needed to verify that our motif discovery technique really detects realistic workloads. For that, cooperation with domain experts is necessary.

In the future, we can think of several extensions of our approach. While we currently require fixed limits to be set for the amount of new flexibility to create (in terms of $\hat{T}$ and $\hat{J}$), it would be straightforward to allow for a weighting between improvement in peak demand (or overshoot) and the required new flexibility by making $\hat{T}$ and $\hat{J}$ variables and including them in the objective function. However, this requires a reasonable estimate of the (financial) costs of adding flexibility to processes compared

to the costs associated with peak demand or overshoot. Also, it would be easy to price the flexibilization on a per-job basis in the objective, to account for some processes to be more costly to make flexible than others.

From an algorithmic perspective, one should look into finding efficient heuristics for optimizing problems of the FPSP family, to enable the application of our approach to large-scale industrial processes. Also, incorporating uncertainties into the model should be a future step that would be beneficial for practical relevance.

# 5 An Order-Based Model for the Resource Acquirement Cost Problem

In this chapter, we present a novel modeling technique for project scheduling problems arising in the context of smart grids. The modeling technique takes on the same problem as the technique presented in Chapter 3. In this chapter, we set a strong focus on the optimization performance of the resulting models. We perform an extensive experimental evaluation to confirm that the new modeling technique does in fact lead to a performance gain.

## 5.1 Introduction

In chapters 3 and 4, we presented two mixed-integer linear programs that optimize schedules of time-flexible electrical loads — in a classical Time-Constrained Project Scheduling Problem (TCPSP) setting in Chapter 3, and in a setting with a global constraint on flexibility in Chapter 4.

Both modeling techniques have in common that they are based on a discrete-time mixed integer linear program, based on the original idea by Pritsker et al. [PWW69]. Time is discretized into a number of time steps, and variables are introduced for each of these time steps. These variables indicate whether each job starts at a certain time step. The binary variables are then used to determine the demand at each time step for each resource. This approach has two obvious disadvantages: First, modeling continuous time is impossible. Instead, the user of the models has to decide on a smallest time scale that the model can represent, i.e., a time resolution. Second, the model's size grows with the number of time steps. Thus the user might have to decide on a coarser than desirable time resolution to still be able to optimize the resulting models within reasonable run time.

There also are scheduling models that work without time discretization. Artigues et al. [AMR03] propose what they call a *flow-based* formulation for the Resource-Constrained Project Scheduling Problem, but which should be adaptable to the TCPSP. The idea of the model is that jobs can, once they finish, pass on the resources they used to later jobs. Koné et al. [Kon+11] build two variants of an *event-based* model, which use a start and stop event per job. They compare their model to the flow-based model and the discrete-time model by Pritsker et al. [PWW69]. They come to the conclusion that their event-based model is superior to both the flow-based and the discrete-time model.

In general, the problems tackled by these models belong to the area of project scheduling problems. The field of project scheduling research is vast — for a good overview, we refer the reader to Węglarz [Węg99].

**Our Contribution.** In this chapter, we present a novel approach. The modeling technique we present here is applicable to optimize for peak shaving, and allows for most of the features representable by the model presented in Chapter 3 — see Section 5.3.2. However, it does require neither variables that are associated with time steps nor event variables. Instead, the core of the model is build by variables indicating the relative order of job pairs.

Together with a formal description of our novel approach, we present an in-depth experimental evaluation of the new model, comparing it against the discrete-time and event-based approach. Our evaluation demonstrates that our modeling technique is superior to the event-based technique, and that our technique should be preferred over the discrete-time approach for instances with a fine time resolution.

**Overview.** After defining required notation and the problem under study in Section 5.2, we present in Section 5.3 the new order-based modeling technique, our main contribution. In Section 5.4, we introduce an adaption of the event-based model by Koné et al. [Kon+11] (which is intended for the RCPSP) to the TCPSP. This event-based model serves (together with the discrete-time model from Chapter 3) as a competitor model. We provide an experimental evaluation of the three modeling techniques in Section 5.5.

## 5.2 Preliminaries

We start by formally stating the problem under examination during this chapter and introducing some necessary notation. The problem is a form of the Resource Acquirement Cost Problem (RACP) introduced in Chapter 2, Section 2.1.2. We are given a problem instance as set of $m$ *resources* $R = \{1, 2, \ldots, m\}$ and a set of $n$ *jobs* $J = \{1, 2, \ldots, n\}$. To weight the resources against each other in the objective function, we are given a weight vector $w \in R^m$, where $w_i$ indicates the weight of resource $i$. Each job $j$ is associated with a *processing time* $p_j$, which specifies how long the job must be run without interruption, a *release time* $r_j$ and a *deadline* $d_j$. Each job has a resource *usage* vector $u_j \in \mathbb{R}^m$, where $u_{j,i}$ indicates the amount of resource $i$ that job $j$ needs while executing.

resource
job
processing time
release time
deadline
usage
time lag

Dependencies between jobs are given as a partial function from job pairs to *time lags* between those jobs: $L\colon J \times J \nrightarrow \mathbb{N}$.

The objective is to assign start times $S_j \in \mathbb{R}$ for every $j \in J$ such that:

- release times and deadlines are upheld: $\forall j\colon S_j \geq r_j \wedge S_j + p_j \leq d_j$,

- time lags are upheld: $\{i, j\} \in L \Rightarrow S_i + L(i, j) \leq S_j$,
- we achieve *peak shaving*: Let $U_{i,t}$ be the amount of resource $i$ that is used at time point $t$, then $\max_t \left(\sum_{i \in R} w_i U_{i,t}\right)$ is minimized among all possible choices for the start times $S$. peak shaving

Given such an instance, we define the *ply* of a job $j$, denoted as $\Pi(j)$, as the number of other jobs the windows of which overlap with the window of $j$, i.e.: ply

$$\Pi(j) = |\{a \in J : r_a < d_j \wedge d_a > r_j\}|$$

## 5.3 The Order-Based Model

In this section, we present our main contribution: a new, order-based modeling technique for RACP instances. The idea behind the technique starts with the insight that if all jobs have constant resource demand, the peak demand for each resource must occur at the start of a job, since resource usage only increases when a job starts. Thus, we do not as in the discrete-time model have variables capturing resource usage at every time step, or as in the event-based model variables capturing the usage at every event, but a set of variables capturing the resource usage at the start of each job.

The order-based model has for each pair of jobs $a, b$ a set of variables that expresses whether job $a$ is running when job $b$ starts. This way, the resource usage during the start of $b$ can be easily expressed.

To check whether $a$ runs when $b$ starts, we only need to distinguish three cases:

- $a$ starts after the start of $b$ or
- $a$ starts before (or at the same time as) $b$, and
  - $a$ finishes before $b$ starts or
  - $a$ does not finish before $b$ starts.

Only in the last case does $a$ run when $b$ starts. The model is based on creating binary variables that capture the above relationship: The variable $D_{a,b}$ ($D$ for "distinct") becomes 1 only if $b$ starts *after* $a$ is finished, i.e., if $S_a + p_a \leq S_b$. The variable $O_{a,b}$ ($O$ for "order") must be 1 if $a$ starts *before* (or at the same time as) $b$, i.e., if $S_a \leq S_b$. Note that the indices always indicate the intended order: $\{O, D\}_{x,y}$ always means "first $x$, then $y$". Also note that we specified the necessary variable assignments only to one side — for example, the variable assignment for $O_{a,b}$ is unspecified if $a$ does not start before $b$. However, this does not invalidate the correctness of our model, as we will see further below.

With these variables, job $y$ is active during the start of $x$ if and only if $O_{y,x} = 1$ (i.e., it has been started) and $D_{y,x} = 0$ (i.e., it has not finished yet), thus, if $O_{y,x} - D_{y,x} = 1$.

Note that the reverse, namely $D_{y,x} = 1$ and $O_{y,x} = 0$ can never happen. Thus, we can express the amount of a resource $\rho \in R$ being used at the start ob job $j$ as:

$$u_{j,\rho} + \sum_{k \in J} (O_{k,j} - D_{k,j}) \cdot u_{k,\rho}$$

### 5.3.1 Full Description

After the high-level description of the order-based modeling technique, we now formalize the approach. For a S-RACP instance as defined in Section 5.2, the variables are:

$$\forall j \in J: \quad S_j \in \mathbb{R} \tag{5.1}$$
$$\forall a, b \in J: \quad D_{a,b} \in \{0, 1\} \tag{5.2}$$
$$\forall a, b \in J: \quad O_{a,b} \in \{0, 1\} \tag{5.3}$$
$$\forall j \in J, \forall \rho \in R: \quad U_{\rho,j} \in \mathbb{R} \tag{5.4}$$
$$\hat{U} \in \mathbb{R} \tag{5.5}$$

The variables $S_j$ hold the start times for all jobs $j$. As explained above, $D_{a,b}$ encodes whether job $a$ ends before job $b$ starts, and $O_{a,b}$ indicates that $a$ starts before (or at the same time as) $b$ starts. For each resource $\rho$ and each job $j$, variable $U_{\rho,j}$ captures the amount of resource $\rho$ that is used in the moment that job $j$ starts. Finally, $\hat{U}$ captures the maximal (weighted) total resource usage. The constraints on these variables are:

$$\forall a, b \in J: \quad D_{a,b} - \frac{S_b - (S_a + p_a)}{M} \leq 1 \tag{5.6}$$
$$\forall a, b \in J: \quad O_{a,b} - \frac{S_b - S_a}{M} > 0 \tag{5.7}$$
$$\forall j \in J, \forall \rho \in R: \quad u_{j,\rho} + \sum_{k \in J} (O_{k,j} - D_{k,j}) \cdot u_{k,i} \leq U_{i,j} \tag{5.8}$$
$$\forall j \in J: \quad \sum_{i \in R} U_{i,j} \cdot w_j \leq \hat{U} \tag{5.9}$$
$$\forall j \in J: \quad S_j \geq r_j \tag{5.10}$$
$$\forall j \in J: \quad S_j + p_j \leq d_j \tag{5.11}$$
$$\forall (a, b) \in L: \quad S_a + L(a, b) \leq S_b \tag{5.12}$$

The objective is to minimize $\hat{U}$.

Constraint (5.6) forces $D_{a,b}$ to become 0 if $b$ does not start after $a$ ends. Note that $(S_b - (S_a + p_a))/M$ must always be in $[-1, 1]$ for an appropriately chosen $M$, and will

be non-negative only if $b$ starts after $a$ ends. This is a requirement for $D_{a,b}$ being 1. On the other hand, Constraint (5.7) forces $O_{a,b}$ to become 1 if $a$ starts before (or at the same time) as $b$ starts. Note that $S_a - S_b$ becomes negative (resp. zero) if $a$ starts before (resp. at the same time as) $b$, thus $O_{a,b}$ must be set to 1 in this case.

Constraint (5.8) uses $O$ and $D$ to compute the usage of a resource $i$ at the start of job $j$. Aside from $u_{j,i}$, we must add the usage of all jobs $k$ that overlap the start of $j$. A job $k$ overlaps the start of job $j$ if and only if $O_{k,j} = 1$ and $D_{k,j} = 0$, i.e., if $k$ starts before or at the same time as $j$, but $j$ does not start after $k$ has finished. Constraint (5.9) again builds the maximum weighted usage.

Finally, constraints (5.10), (5.11) and (5.12) enforce release times, deadlines and time lags in a straightforward way.

It might seem strange that we enforce the value of $D_{a,b}$ to be 0 only for the case that $b$ does not start after $a$ ends — if $b$ *does* start after $a$ ends, $D_{a,b}$ can be either 1 or 0. Similarly, we have such a "one-sided" constraint for $O_{a,b}$. Note however that a job $a$ only contributes to the demand at start of job $b$ if $O_{a,b} = 1$ and $D_{a,b} = 0$. Thus, this is the case we need to enforce using constraints. The other case — i.e., that not $O_{a,b} = 1$ and $D_{a,b} = 0$ in case that job $a$ does not execute during the start of job $b$ — will be preferred by the optimization.

### 5.3.2 Viable Model Features

In Section 3.3 of Chapter 3, we introduced a list of *features* that can be modelled by the MIP framework presented in that chapter. In this section, we give a short overview over which of those features can still be implemented with our new model. First, a list of the features mentioned in Chapter 3 is directly modelled in the model presented in Section 5.3.1: *Earliest start times*, *deadlines* and *interdependent jobs* (i.e., time lags between jobs) are directly a part of the model. Also, *multiple resources* can be specified. As with the discrete-time model, *base loads* can be modelled by introducing jobs $j$ for which $d_j - r_j = p_j$, i.e., which are fixed to one point in time.

If one wishes to model different *modes* or *drain*, the technique used in the discrete-time model is to introduce a variable for the processing time of each job instead of treating the processing time as a constant (see Section 3.4 of Chapter 3). Note that in the order-based model, we never multiply any $p_i$ with a variable. Thus, making $p_i$ a variable (called $T_i$ in the discrete-time model from Chapter 3) does not impede the linearity of the model. We can therefore incorporate multi-mode jobs and drain in the same way as in the discrete-time model. Similarly, *ramping* is modelled in the discrete-time model by a series of dummy jobs that are switched on or off depending on whether ramping is necessary. This part can be directly taken over into the order-based model.

### 5.3.3 Model Size

The $O_{a,b}$ and $D_{a,b}$ variables dominate the number of variables with each $n^2$ variables. The number of constraints is also in $O(n^2)$, dominated by the constraints (5.7), (5.6) and (5.8). In particular, each $O_{a,d}$ and $D_{a,d}$ appear in exactly one constraint of each (5.6), (5.7) and (5.8).

While the above model is correct, it introduces many unnecessary variables. We actually only need $D_{a,b}$ and $O_{a,b}$ variables for jobs the windows of which do overlap and between which there is no defined minimum time lag. This automatically also reduces the number of constraints of type (5.8), since they only exists for $a, b \in J$ for which $O_{a,d}$ and $D_{a,d}$ exist.

After this reduction, the total number of $O_{a,b}$ (resp. $D_{a,b}$) variables is $\sum_{j \in J} \Pi(j)$, which in turn reduces the number of constraints of types (5.6), (5.7) and (5.8) also to $\sum_{j \in J} \Pi(j)$ each.

Aside from the number of variables and constraints, the number of nonzero coefficient matrix entries is an important metric. Each $\{O, D\}_{a,b}$ participates in exactly one constraint of types (5.6), (5.7) and (5.8), respectively, for a total of $O(n \cdot \Pi)$ nonzeros. Similarly, each $S_a$ variable participates in at most $2\Pi$ constraints of type (5.7) and (5.6) each, again for a total of $O(n \cdot \Pi)$ nonzeros.

So far, we have not yet considered time lags in this analysis. Every defined time lag contributes one more constraint of type (5.12), which adds two nonzero coefficient matrix entries.

## 5.4 Competitor Model: Event-Based Model

In this section, we present an adaption of a known modeling technique that we use to compare our new approach to. Koné et al. [Kon+11] have shown this technique to be superior to discrete-time and flow-based models. The event-based model outlined in this section is a straightforward adaption of the event-based formulation from Koné et al. [Kon+11] — intended for the Resource-Constrained Project Scheduling Problem — to the Resource Acquirement Cost Problem. In its original form for the RCPSP, bounds are placed on the resource usages. To adapt to the TCPSP, those usages are now incorporated into the objective function. After that, we only have to add constraints to enforce hard deadlines on jobs.

The idea behind the event-based model is to consider every job start and every job end as an *event*. With $n$ jobs, that yields $2n$ events. For every job $j$, we introduce $2n$ binary variables $s_{j,k}$, indicating whether job $j$ starts at event $k$, and $2n$ binary variables $e_{j,k}$, indicating whether job $j$ ends at event $k$. We require jobs to end at an event that is later than their start event, and allow just one job start or job end to be assigned to each event. The resource usage at each event can then be computed from the resource

usage at the previous event plus the usage of the job starting at this event resp. minus the usage of the job ending at the event.

Finally, we need to express events' times to enforce deadlines. For every event $k$, we introduce a variable $T_k$ that holds the point in time at which event $k$ happens. To express events' times, it is sufficient to, for every job $j$, force the time between its start-event and its end-event to be exactly $p_j$, i.e., the jobs duration.

In total, the variables are:

$$\forall j \in J, \forall k \in \{1, \dots 2n\}: \quad s_{j,k} \in \{0, 1\} \tag{5.13}$$
$$\forall j \in J, \forall k \in \{1, \dots 2n\}: \quad e_{j,k} \in \{0, 1\} \tag{5.14}$$
$$\forall k \in \{1, \dots, 2n\}: \quad T_k \in \mathbb{R} \tag{5.15}$$
$$\forall k \in \{1, \dots, 2n\}, \forall \rho \in R: \quad U_{\rho,k} \in \mathbb{R}^+ \tag{5.16}$$
$$\forall \rho \in R: \quad \tilde{U}_\rho \in \mathbb{R}^+ \tag{5.17}$$
$$\hat{U} \in \mathbb{R} \tag{5.18}$$

Here, the $s_{j,k}$ are the variables associating the start of job $j$ with event $k$. Conversely, $e_{j,k}$ assign the ends. The $T_k$ variables denote the time point at which event $k$ happens. For resource $\rho$, the variable $U_{\rho,k}$ captures the resource usage at event $k$. The maximum usage is captured in $\tilde{U}_\rho$. Finally, the total weighted resource usage is combined in $\hat{U}$. Set $d_{\max} = \max_j d_j$. Then, the constraints are:

$$\forall j \in J: \quad \sum_k s_{j,k} = 1 \tag{5.19}$$
$$\forall j \in J: \quad \sum_k e_{j,k} = 1 \tag{5.20}$$
$$\forall k \in [2, 2n], \forall \rho \in R: \quad U_{\rho,k-1} + \sum_{j \in J} s_{j,k} \cdot u_{j,\rho} - \sum_{j \in J} e_{j,k} \cdot u_{j,\rho} \leq U_{\rho,k} \tag{5.21}$$
$$\forall \rho \in R: \quad \sum_{j \in J} s_{j,1} \cdot u_{j,\rho} \leq U_{\rho,1} \tag{5.22}$$
$$\forall \rho \in R, \forall k \in [1, 2n]: \quad U_{\rho,k} \leq \tilde{U}_\rho \tag{5.23}$$

$$\sum_{\rho \in R} \tilde{U}_\rho \cdot w_\rho \leq \hat{U} \tag{5.24}$$

$$\forall a \in [1, 2n), b \in (a, 2n]: \quad T_a \leq T_b \tag{5.25}$$

$$\forall a, b \in [1, 2n], a < b, j \in J: \quad T_a + ((s_{j,a} + e_{j,b} - 1) \cdot p_j) \leq T_b \tag{5.26}$$

$$\forall a, b \in [1, 2n], a < b, (i, j) \in L: \quad T_a + ((s_{i,a} + s_{j,b} - 1) \cdot L_{a,b}) \leq T_b \tag{5.27}$$

$$\forall j \in J, k \in [1, 2n]: \quad T_k + ((1 - s_{j,k}) \cdot r_j) \geq r_j \tag{5.28}$$

$$\forall j \in J, k \in [1, 2n]: \quad T_k - ((1 - e_{j,k}) \cdot (d_{\max} - d_j)) \leq d_j \tag{5.29}$$

The objective to be minimized is $\hat{U}$.

Constraints (5.19) and (5.20) ensure that every job is assigned exactly one event where it starts and one event where it ends. Constraint (5.21) computes the resource usage at an event $k$ for resource $j$. This usage consists of the usage for resource $j$ at event $k-1$ plus anything that starts at $k$ minus anything that ends at $k$. Constraint (5.22) handles the special case of the first event. Constraint (5.23) determines the maximum amount required of each resource, while constraint (5.24) builds the maximum weighted usage, thus the objective to be minimized.

Constraint (5.25) enforces the times associated with the events to be ordered. Constraint (5.26) computes (a lower bound on) the times at which an event $k$ happens: If a job $j$ starts at event $a$ and ends at event $b$, then the difference between $T_a$ and $T_b$ must be at least $p_j$. Note that $s_{j,a} + e_{j,b} - 1$ becoming negative is not a problem, since for $a < b$, $T_a \leq T_b$ must hold anyways. Similarly, Constraint (5.27) places a lower bound on the time that must elapse between two events associated with the starts of jobs between which a minimum time lag was specified.

Finally, constraints (5.28) and (5.29) use the computed event times to enforce release times and deadlines. These must be introduced for every pair of job and event and switched on and off with a big-M switch.

A note on the order of events: If multiple jobs start and end at the same time, this will result in multiple events being assigned the same time. In this case, the constraint (5.25) makes no statement as to the order in which job starts and ends should appear. For example, event $k$ could be assigned as the end of job $a$, event $k + 1$ as the start of job $b$, event $k + 2$ as the start of job $c$, and event $k + 3$ as the end of job $d$. Now, the computed demand at event $k + 1$ does include the demands of jobs $b$ and $d$, although $d$ has already finished, and although $c$ should be started at this time. This is not a problem: First, no job's demand will be missed, since we only care for the maximum demand of each resource (see constraint (5.23)). The last event at each time step will include the resource demand of all jobs started at that time step. Also, since that maximum shall be minimized, an optimum solution will always end all jobs at events before the event that contributes to the maximum usage. Similarly, it is not necessary to constrain the number of job starts and ends that are assigned to each event. The

usage at the event will be valid even if multiple starts and ends are assigned to the same event.

### 5.4.1 Reducing Variable Count

Similar to the way we reduced the order-based formulation's size in Section 5.3.3, we can also reduce the size of the event-based models. If we know for a job $j$ that at least $k$ other jobs must be finished before $j$ can start (either because of jobs' windows or because of dependencies), we do not need to introduce $s_{j,l}$ or $e_{j,l}$ variables for $l < 2k$, since at least $2k$ events before the start-event of $j$ must be covered by other jobs starting and finishing before $j$. The reverse is also true: For jobs certainly starting after $j$ has finished, we can similarly reduce the variables for $j$.

### 5.4.2 Model Size

In the basic form without the size reduction from Section 5.4.1, the number of variables is dominated by $2n^2$ variables from either (5.13) and (5.14), thus $4n^2$ binary variables. With the modification from Section 5.4.1, the number of either (5.13) and (5.14) variables for job $j$ is bounded by its ply $\Pi(j)$, for a total of $2 \sum_{j \in J} \Pi(j)$. This follows from the fact that the jobs not overlapping $j$'s window, i.e. which are not counted for $\Pi(j)$, must be finished before $j$ starts resp. start after $j$ finishes. In the presence of lags, the number is potentially reduced further.

The number of constraints is worse: Constraint (5.26) yields $\Theta(n^3)$ constraints for the non-reduced variant, and $\sum_{j \in J} \Pi(j)^2$ in the reduced form. Constraint (5.27) occurs $\Theta(n^2 \cdot \mathrm{dom}(L)) \in O(n^4)$ times, where $\mathrm{dom}(L)$ is the domain of the partial function $L$. Thus, while dependencies between jobs might reduce the number of $s_{j,k}$ and $e_{j,k}$ variables, they drastically increase the number of constraints.

## 5.5 Experimental Evaluation

The benefit of the model presented in this chapter lies in an improved optimization performance. While the analysis of the model size presented in sections 5.4.2 and 5.3.3 already indicates that our modeling technique results in smaller models than what the event-based model can achieve, we now present an experimental evaluation of the different modeling techniques to substantiate the claim that our models are more efficient.

We evaluate three modeling techniques: The new order-based technique from Section 5.3, the event-based technique from Section 5.4, and the discrete-time technique from Chapter 3. We use two different instance sets for our evaluation. Since we assume the order-based approach to outperform the discrete-time model on instances with a fine time resolution, we generate a set of instances with comparatively fine

resolution, called instance set $\mathcal{A}$. In this set, we generate instances with scheduling horizons picked uniformly at random between three and seven days, in one-minute time resolution. The duration of the jobs to be scheduled is picked from a normal distribution with a mean of eight hours and a standard deviation of two hours, and their slack (i.e., the window size minus the duration) is picked from a normal distribution with mean 12 hours and a standard deviation of five hours. We also draw for every instance an average ply from a uniform distribution between 10 and 30. For every instance, we then keep generating jobs until this chosen average ply $p$ is achieved, i.e., until the sum of the window sizes of all generated jobs is at least $p$ times the scheduling horizon. We generate a total of 200 instances like this.

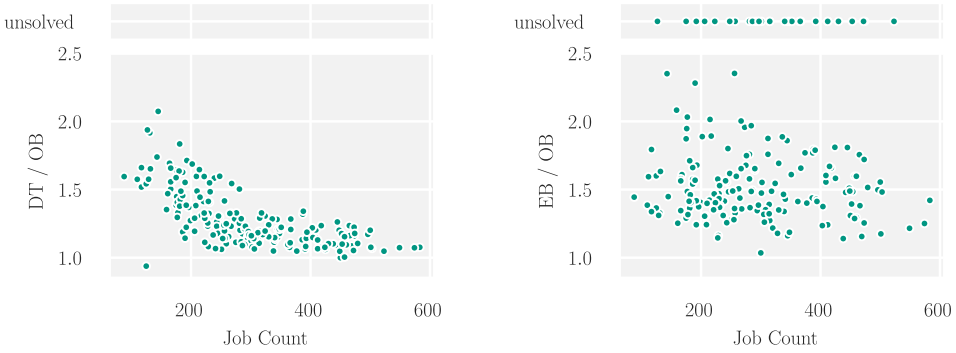
Additionally, we apply all three models to the benchmark instances presented in Chapter 6, which we call instance set $\mathcal{B}$ throughout this section.

All experiments were conducted by first building the respective models, then optimizing the models using Gurobi 7.0.2. The machines used are equipped with 64 GBs of memory and two Intel® Xeon® E5-2670 CPUs, each having eight cores running at 2.6 GHz. Gurobi was allowed to use 15 parallel threads, and we used 15 minutes as time limit.

### 5.5.1 Optimization Performance

We first look at the achieved optimization performance. Figure 5.1 shows the quality that the two competitor MIPs computed on all instances of set $\mathcal{A}$ relative to the quality computed on the same instances by the order-based MIP. For the discrete-time MIP, we see that with one exceptions, the order-based MIP computes better solutions. For instances with few jobs, the advantage goes up to a factor of two, for instances with more jobs, that advantage decreases. On most instances, the order-based MIP has an advantage of between 10% and 50%. The order-based MIP out-competes the event-based MIP on all instances, by about between 20% an 100% on most instances. For many instances, the event-based MIP does not even find feasible solutions. We also see that the advantage of the order-based MIP does not decrease with increasing number of jobs.

For set $\mathcal{B}$, we only plot relative results for the discrete-time MIP, since the event-based MIP was not able to solve even a single instance. The results for the discrete-time MIP are shown in Figure 5.2. We see that for instances with more instances but less fine time resolution, the discrete-time MIP fares better than the order-based approach. The order-based MIP still produces feasible solutions in the given time consistently for up to 2000 jobs, and does not work anymore past approximately 3000 jobs. As long as the order-based MIP still produces solutions, the discrete-time MIP's peak demand is pretty evenly distributed between half the peak demand and the same peak demand as for the order-based models. Again, the discrete-time model produces feasible solutions for all instances.




**(a)** Discrete-Time MIP **(b)** Event-Based MIP

**Figure 5.1:** The objective value achieved by the competitor MIPs ("DT": Discrete-Time, "EB": Event-Based) for each instance in instance set $\mathcal{A}$, one dot per instance. On the $y$ axis is the quality achieved by the competitor MIP normalized by the quality achieved by the order-based MIP on the same instance. Instances are sorted by job count on the $x$ axis. The upper band contains a dot for every instance that the competitor MIP found no feasible solution for.

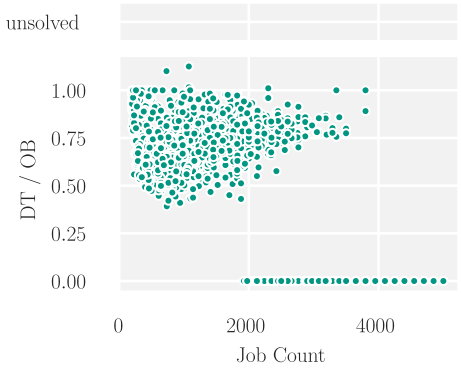



**Figure 5.2:** The objective value achieved by the discrete-time MIP for each instance in instance set $\mathcal{B}$, one dot per instance. On the $y$ axis is the quality achieved by the discrete-time MIP normalized by the quality achieved by the order-based MIP on the same instance. Instances are sorted by job count on the $x$ axis. An $y$-value of 0 indicates that the order-based MIP did not find a feasible solution.

### 5.5.2 Empirical Model Sizes

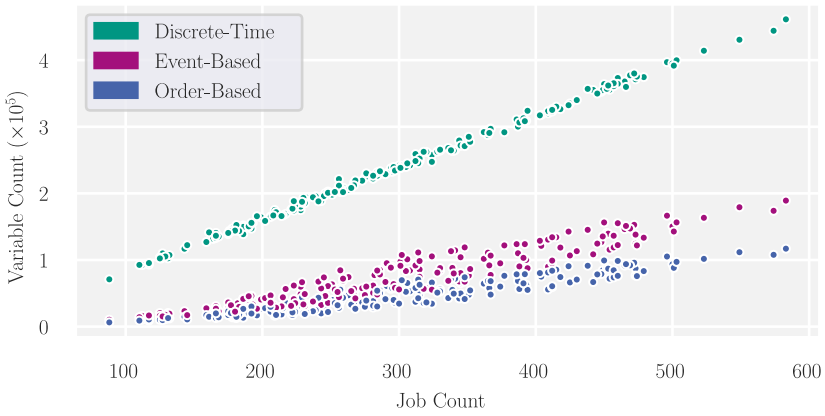
Further insight into the performance that can be expected for optimizing the created MIP models can be gained by looking at the size of the models. We again start by analyzing the instances in set $\mathcal{A}$. Figure 5.3a shows the number of variables in the models for all three modeling techniques, again ordered by job count on the $x$ axis. Even though one can see the slight curve of the quadratic increase for the event-based and order-based models, the (linearly growing) number of variables in the discrete-time model is still considerably higher for these instances. A similar picture arises when looking at the number of nonzero entries in the coefficient matrix, displayed in Figure 5.3c, although here the advantage of the order-based approach becomes more apparent. Looking at the number of constraints in Figure 5.3b, the reason for the larger number of nonzeroes with the event-based models becomes apparent: With its cubically increasing number of constraints, the event-based approach dwarfs both the discrete-time and the the order-based approach in terms of constraints.

Closely related to model sizes is the question of model construction time, i. e., the time it takes to construct the respective models before any optimization can start. Figure 5.4 presents the times it takes to build all the models discussed in this section. We see that while the order-based models stays below five seconds, both the discrete-time and the event-based models suffer from the large number of variables resp. constraints.

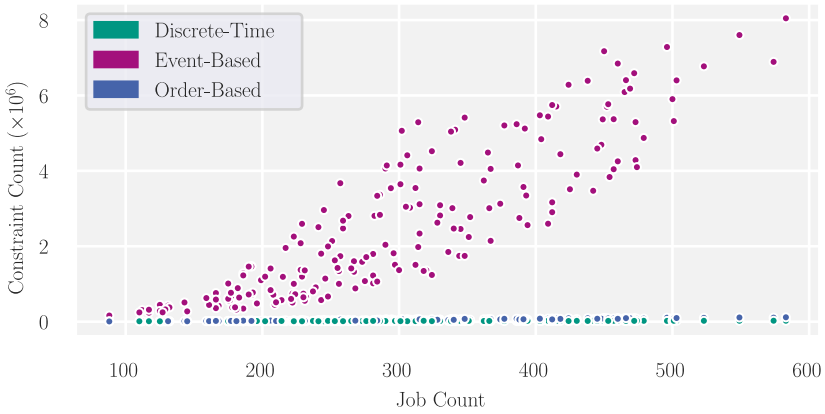
For the instances in set $\mathcal{B}$, the first interesting phenomenon is that the event-based approach is not able to compute any solution for any instance. The reason for this can easily be seen from the numbers of generated constraints and nonzero matrix entries, shown in figures 5.5b and 5.5c, respectively. The cubic growth in the number of constraints rapidly overtakes the other two models. The fact that there are no more points plotted for the event-based models past approximately 500 jobs is explained by the fact that after this point, model generation did exhaust the machine's memory. It is interesting to see that the number of nonzero matrix entries for the order-based model does in fact not exceed that of the discrete-time model that much even for instances with a large number of jobs. However, the number of variables (shown in Figure 5.5a) and constraints does. Thus, the coefficient matrix is a lot denser for the discrete-time models.

## 5.6 Conclusion

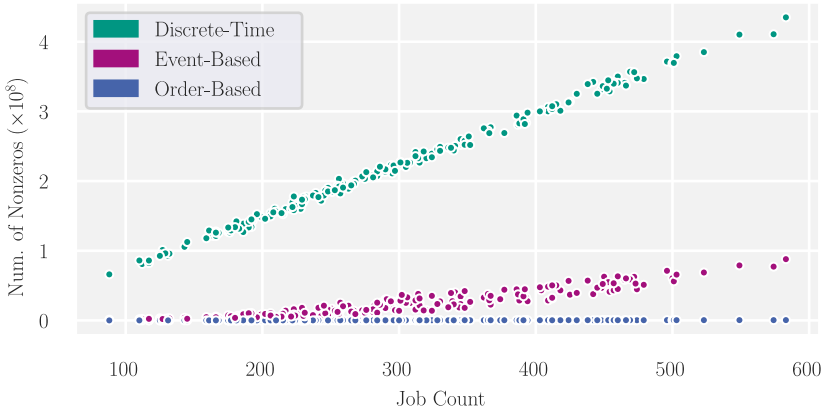
We have presented a novel modeling technique for Time-Constrained Project Scheduling Problem family. The technique is based on capturing the temporal relation between pairs of jobs, and is thus not affected by the chosen time resolution. In a theoretic analysis of the model sizes of our technique, we have shown that the produced models are smaller than those produced by an event-based approach from literature. Dependent on the number of jobs in an instance and the chosen




**(a)** Number of variables




**(b)** Number of constraints




**(c)** Number of nonzero coefficient matrix entries

**Figure 5.3:** Comparison of the model sizes by the three different techniques for the instances in set $\mathcal{A}$. Every dot corresponds to one model, ordered by the number of jobs on the $x$ axis.

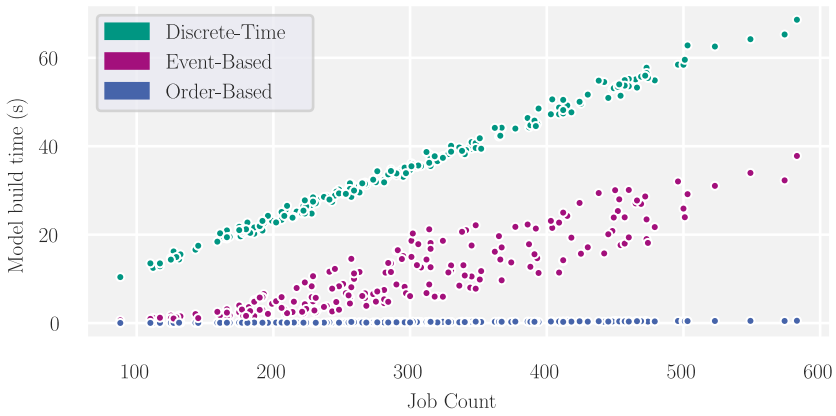
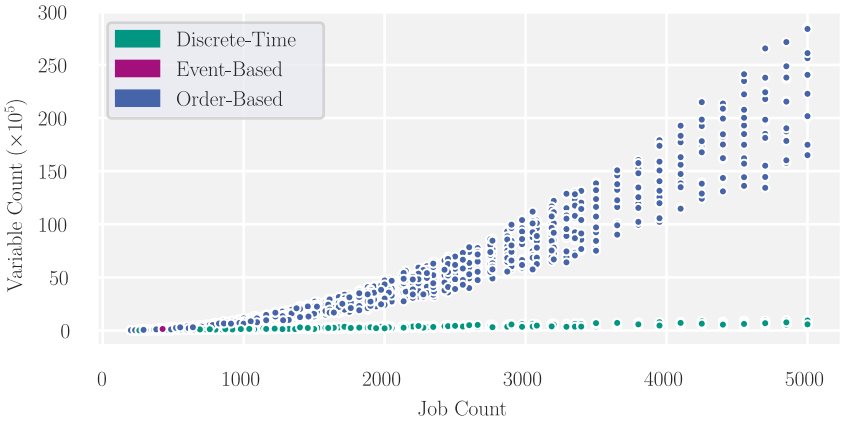


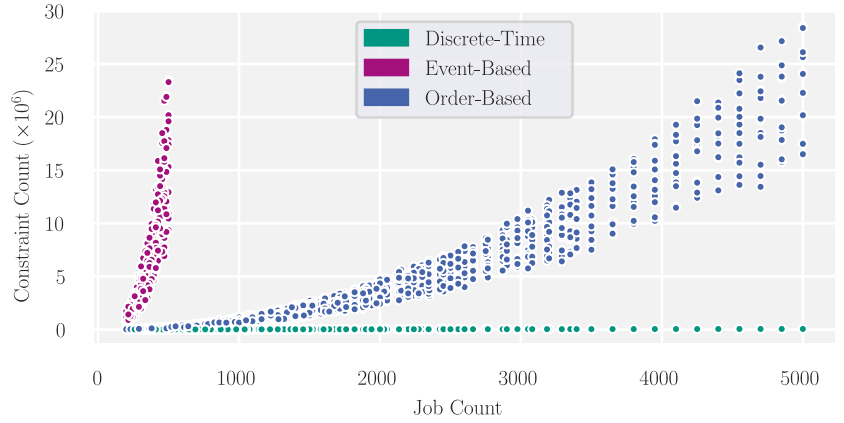

**Figure 5.4:** Comparison of the time (in seconds) it takes to construct the respective models for each instance in set $\mathcal{A}$. Every dot corresponds to one model, ordered by the number of jobs on the $x$ axis.

time resolution, our models are also smaller than a different discrete-time approach from literature. An experimental evaluation corroborates these findings on a set of benchmark instances. Our experimental evaluation also shows that for certain sets of instances, our technique produces models that are significantly easier to optimize than the models produced by the two competitor techniques. However, our evaluation also shows that if time resolution is low and the number of jobs is large, one should chose the discrete-time modeling technique.

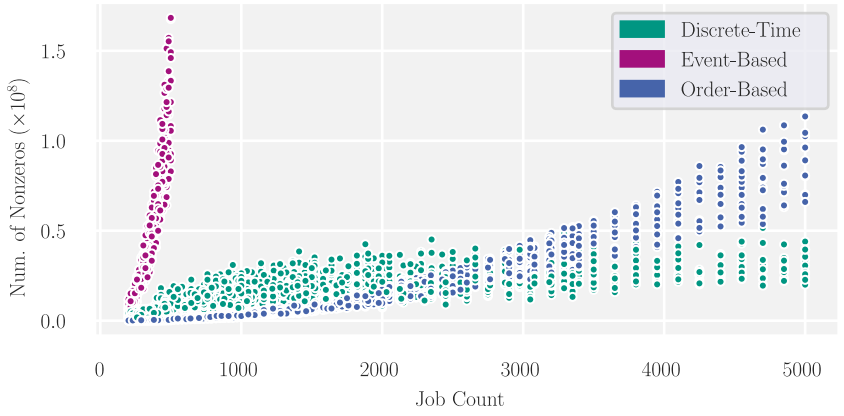
In conclusion, we think that the order-based approach can be a well-fitting tool to optimize project scheduling problems with high time resolution. This would be even more the case if one could incorporate more of the modeling features presented in Chapter 3 into the new technique in the future.




**(a)** Number of variables




**(b)** Number of constraints




**(c)** Number of nonzero coefficient matrix entries

**Figure 5.5:** Comparison of the model sizes by the three different techniques for the instances in set $\mathcal{B}$. Every dot corresponds to one model, ordered by the number of jobs on the $x$ axis.

# 6 Industrial Demand Side Flexibility: A Benchmark Data Set

To cope with the new demands of an electrical grid based on mostly renewable energy, more flexibility on the demand side is needed. To test new demand response strategies, energy consumption data sets which come with some information about the inherent flexibility of the processes are needed. However, such data sets — especially related to industrial customers — are often commercially sensitive and thus not published.

In the present chapter, we introduce a new benchmark data set containing scheduling scenarios of industrial processes with flexibility information. The instances are based on a real-world data set of a small scale industrial facility, from which we extract process characteristics using a novel motif discovery technique. Based on these characteristics, we generate a set of benchmark instances. We provide an in-depth analysis of the benchmark data set and show that it is suitable to evaluate smart-grid scheduling techniques.

This chapter is based on joint work with Nicole Ludwig, Dorothea Wagner and Veit Hagenmeyer [Lud+19b].

## 6.1 Introduction

Societies around the globe aim for future electricity grids which rely mostly on renewable energy sources (RES). Unfortunately, the intermittent nature of the RES and the fact that they cannot be controlled makes integrating them into today's grid difficult. One difficulty is that there is often a mismatch between the supply from RES and the actual power demand. While increasing transmission and storage capacities is one option to ease the integration, another one is generally referred to as *demand side management* (DSM). DSM includes all measures which aim at changing behaviour in the energy usage by demand side actors. DSM has been discussed extensively in the literature, for example by Finn and Fitzpatrick [FF14] or Boogen et al. [BDF17]. The sub-field of DSM which focuses on real-time changes in energy consumption is termed *demand response* (DR). An important assumption underlying all DR approaches is that the consumers have some kind of temporal or operational flexibility. Thus, they can either change *when* they use energy (temporal flexibility), or *how* they use energy (operational flexibility).

Information about the flexibility of individual consumers, especially industrial ones, is not readily available. This leads to difficulties in testing new strategies and ideas for DR or comparing different strategies with each other. To test different algorithms and frameworks, one can use data from or resembling smart meters (see for example

Gottwalt et al. [Got+11]) or grid data (see Logenthiran et al. [LSS12]). However, most authors either do not publish the data their analysis is based on or synthesize the whole data set. The synthetic data can range from being entirely made up (as for example in the case of Petersen et al. [Pet+14]), being modelled with specific appliances in mind (see for example Li et al. [Li+12]) or being generated based on data but without using algorithms to extract information from this data, as for example Yaw et al. [Yaw+14] do. Benchmark data sets play an essential role in making research comparable and more accessible. For general project scheduling, for example, there exists the PSPLIB benchmark data set by Kolisch and Sprecher [KS97], which the related literature uses heavily. However, this benchmark data set is not rooted in real-world data, as it relates to general project scheduling problems without any specific application in mind. Specifically for resource-constrained project scheduling, Kolisch et al. [KSS99] have published a data set. Again, the data set is not derived from real-world data. Moreover, no such benchmark data set exists for demand side flexibility in industrial processes.

Recently, the HIPE data set, a real-world data set with smart meter measurements from industrial machines, has been published by Bischof et al. [Bis+18]. This data set contains power demand time series from a set of machines in a small-scale electronics factory. However, the data set consists only of a relatively small amount of machines and there is no readily available information about their flexibility. Hence, in the present chapter, we extract process characteristics from the real-world data set of industrial machines, generate more process instances based on this information and infer flexibility attributes. More specifically, we use a novel algorithm based on motif discovery to find regular process patterns in each machine and extract information on when each process starts throughout the day as well as how many different processes can be identified in each machine. Based on this information we generate instances that model real-world scenarios with available demand side flexibility. To the best of our knowledge, we are the first to create such artificial instances based on pattern recognition through motif discovery. We evaluate the found patterns and show that the data can be used to evaluate performance-critical scheduling algorithms on workloads that resemble real-world scenarios.

The remainder of the chapter is structured as follows. We start with a short introduction to a scheduling problem which we use to evaluate the data set in Section 6.2. In Section 6.3 we outline the methodology to extract the process pattern from the power demand time series. We then describe how we generate benchmark instances based on this information in Section 6.4. Section 6.5.1 goes into detail about the origin of the time series data as well as the exact parameters chosen to generate our benchmark data set. It also explains how to obtain our data set. After that, we describe the respective processes found (Section 6.6), and evaluate their behaviour in scheduling algorithms (Section 6.7).

## 6.2 Preliminaries

The main contribution of the data set described in Section 6.5 is a set of real-world-data based benchmark instances for certain scheduling problems. The problem at hand arises in smart grids when flexible electrical demands can be moved in time to optimize various objectives. In this section, we introduce such a scheduling problem, which we also use in Section 6.7.2 to evaluate the suitability of the benchmark instances. Additionally, we show in Section 6.2.2 how the assumptions of the problem in Section 6.2.1 can be relaxed.

### 6.2.1 Single-Resource Project Scheduling

In a first step, we define the problem under the assumption that all processes have constant power demand during their execution. In Section 6.2.2 we describe how the defined problem can be used to optimize scenarios where processes' power demand changes over time without the need for a more elaborate model.

In the scheduling problem, every time-movable process constitutes one *job*. Let $j_i$ be a job. Then, $j_i$ has a *processing time* $p_i$, i. e., a duration for which it must be executed without interruption. The job also has a *release time* $r_i$, which is the earliest time during which the job can execute, and a *deadline* $d_i$, which is the earliest time at which the job must be finished. Finally, every job has a *usage* $u_i$, which is the (constant) amount of power required by $j_i$ during its execution.

job
processing time
release time
deadline
usage

Given a set of $n$ such jobs $J = \{j_1, j_2, \dots j_n\}$, a problem instance also has an edge-weighted directed acyclic graph $G = (J, D, w)$ on $J$, with edge set $D \subset J \times J$ and weight function $w : D \to \mathbb{Z}$. The edge set $D$ defines *dependencies* between two jobs, while the weight function indicates the necessary *time lag* between the two jobs. If $(j_a, j_b) \in D$, then $j_b$ can start at the earliest $w((j_a, j_b))$ time steps after $j_a$ has started.

dependency
time lag

Based on these definitions, we now define the problem used throughout this chapter, which models a peak shaving scenario.

**Problem 1** (Single-Resource Acquirement Cost Problem (S-RACP))**.** *Given $J$ and $G$ as defined above, the Single-Resource Acquirement Cost Problem is to find a start time for each job such that the peak demand of the resulting schedule is minimized.*

S-RACP

### 6.2.2 Non-Constant Power Demands

The problem defined in Section 6.2.1 assumes power demands to be constant over time. This assumption might be an especially unrealistic and simplifying one. Therefore, we describe in this section how the model from Section 6.2.1 can be used to model jobs with fluctuating power demand.

We assume the power demand function of a process to be a stepwise function. To model a stepwise power demand function for a process, we perform a *block decomposi-*

block decomposition

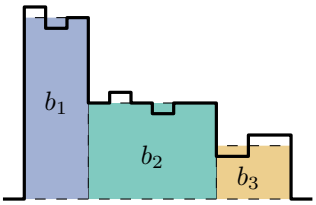



**Figure 6.1:** Decomposition of a stepwise power demand function into blocks. The fat black line is the original demand function with 11 steps. The three blocks $b_1$, $b_2$ and $b_3$ approximate this function.

*tion*, which is illustrated in Figure 6.1. For a perfect representation of a stepwise power demand function with $k$ steps, we decompose the respective process into $k$ blocks. From the perspective of the scheduling problem, each of the blocks is an individual job. However, we will use the term *block* for a job with constant power demand that is part of a decomposition of a process with non-constant power demand. It is easy to see that the $k$ blocks of a process, when executed consecutively, behave like a single job with the appropriate stepwise power demand function.

block

Thus, we must make sure that the blocks are executed consecutively and without any pauses between them. To this end, we use the dependencies introduced in Section 6.2.1. Let $b_1, b_2, \dots b_k$ be the $k$ blocks that we decomposed a process into. Then, for every $i \in \{1, \dots k-1\}$, we add $(b_i, b_{i+1})$ to $D$, with a weight $w((b_i, b_{i+1})) = p_i$. With this, no block can start before its predecessor has finished (but immediately after). Finally, we add $(b_k, b_1)$ to $D$ with $w((b_1, b_k)) = -1 \cdot \sum_{i=1}^{k-1} p_i$. This negative lag forces the last block to start *at the latest* $\sum_{i=1}^{k-1} p_i$ time steps after $b_1$ started. Combined, these dependencies cause the chain of $b_1, b_2, \dots b_k$ to be executed concurrently.

When modeling a real-world process (with a stepwise power demand function) in this way, the obvious $k$ to choose is the number of steps in the respective power demand function. As mentioned before, this would result in a perfect representation of the original stepwise power demand function. However, one can also choose a smaller number. If one chooses $k$ smaller, the series of blocks can not reflect the original process' power demand function perfectly, but only approximate it. In Section 6.7, we take a closer look at how well this approximation works with a low value for $k$, as well as how the value $k$ influences the complexity of S-RACP instances.

## 6.3 Finding Process Patterns

motif

The benchmark instance generation process in Section 6.4 takes as input *motifs* detected in the power demand time series. A motif is a (frequently reoccurring) pattern in the power demand curve. Since the detected motifs correspond to patterns in the time series, certain sequences in the power demand time series can be associated with a

motif. We assume that every point in the time series belongs to at most one motif, i.e., that motifs do not overlap. We call the sequences associated with a motif the *occurrences* of that motif.

occurrence

We do not go into detail here how the motif discovery is done. It is a typical unsupervised learning problem to find patterns in time series. The most popular methods for finding these patterns include different forms of clustering. The motif discovery technique used by us is a variation of the motif discovery algorithm by Chiu et al. [CKL03]. For details on the motif discovery technique, we refer the reader to the work by Ludwig et al. [Lud+17], which explains the approach in detail.

For this chapter, we just assume to be given a set of motifs and their respective occurrences. As final step in the pattern finding process, we determine block decompositions for each detected occurrence, as necessary for the process to represent jobs with non-constant power demand introduced in Section 6.2.2. In Section 6.7, we specify a measure for what we consider a good block decomposition of an occurrence. For each occurrence, and each $k$, i. e., each number of blocks the occurrence is to be decomposed into, we use a sequential least-squares programming solver to find a decomposition into $k$ blocks that optimizes the measure from Section 6.7. The block decompositions of all occurrences are part of the data set we release, see Section 6.5 for how to obtain them.

## 6.4 Generating S-RACP Instances

After detecting motifs, their occurrences and their block decompositions, we generate jobs that together form instances of the Single-Resource Acquirement Cost Problem (see Section 6.2.1). Several parameters influence the generation. First, the *instance size* specifies the number of jobs per instance. Also, a *time horizon* must be given, i.e., the latest deadline of any job, assuming that the earliest release time is 0. For the individual jobs, we first must specify the *block count*, i. e., into how many blocks every process should be decomposed. Also, we must specify how much flexibility we assume to be part of the instance, which is done in terms of a *window growth mean*, a *window growth standard deviation* and a *window base factor*.

Instance generation works by creating a set of *job generators*, one for each motif and each block count $k$, and then generating jobs from these generators up to the desired instance size. The idea behind the job generators is to fit random distributions to the respective motif's occurrences. When generating a job that should be divided into $k$ blocks, we start with the respective motifs' occurrences that have been decomposed into $k$ blocks as per Section 6.3. For each block, we obtain its length and total energy consumption, for a total of $2k$ values for each of the motif's occurrences. To these values, we fit a $2k$-variate Gaussian Mixture Model (GMM). We choose Gaussian Mixture Models because they are universal density approximators, as shown by e.g. Plataniotis

and Hatzinakos [PH00], meaning they can approximate any given probability density with arbitrary precision (under the condition that the GMM has enough components). Since we do not want to make any assumptions about the underlying distribution of the duration and energy values, GMMs seem appropriate. Drawing from this distribution results in the shape of a new job: For each of the job's blocks, we get a duration and an energy consumption, from which we determine the power demand.

To determine a release time and a deadline, we first fit a distribution to the start times of the motif's occurrences. However, the start times do empirically not fit a (mixed) normal distribution well. Thus, we instead use a mixture of uniform distributions. To do so, we first cluster the start times using the DBSCAN algorithm by Ester et al. [Est+96] to account for the assumption that there might be multiple separate time spans throughout a day during which the respective process is usually started. Then, we determine the 0.1 and 0.9 quantile of each determined cluster to account for outliers. These form the lower and upper limit of one uniform distribution each. We weight each uniform distribution by the number of occurrences assigned to the respective cluster. Randomly selecting one such uniform distribution by their weight and then drawing from that distribution yields a preliminary start time $s$.

However, we need a release time $r$ and and a deadline $d$ (together forming the *window* of the job). We obtain them by determining a window size $w$ and then setting $r = s - (w/2)$ and $d = s + p + (w/2)$ (with $p$ being the processing time, see Section 6.2.1). We determine $w$ in two components, the *window base* $w_b$ and the *window growth* $w_g$. The window base is meant to reflect the flexibility we see in the real-world data. However, the factory we retrieved the real-world data from was so far not managed with demand response in mind, thus we assume that more flexibility could be created if operations were changed to facilitate DR, which is why we add the window growth component.

window base
window growth

The window base $w_b$ is determined by the span of the uniform distribution we drew the start time from multiplied by the window base factor. We assume that the more flexible a process is, the larger the spans of its uniform start-time distributions will be. The window growth is determined by drawing from a normal distribution with the specified window growth mean and window growth standard distribution.

To generate an instance with $n$ jobs, we $n$ times perform a weighted selection on the set of job generators. We weight the generators by the number of occurrences in the respective motif. Each time, we generate one job using the selected generator. For each job generated in this way, the release times and deadlines produced by the job generator are based on time-of-day. Therefore we finally move each generated job to a random day within the time horizon. Note that although the S-RACP as defined in Section 6.2.1 allows for dependencies between jobs, we only use these dependencies for the block decomposition as described in Section 6.2.2. The real-world data we obtained (see Section 6.5.1) does not contain any satisfactory information

about dependencies between processes (see Section 6.6), thus we do not incorporate these into the generated instance sets.

### 6.4.1 Grouped Generation

To evaluate the effect of $k$, i. e., the number of blocks into which jobs are decomposed, we generate groups of instances that differ only in the value of $k$. We do this by first generating an instance with $k = 1$. Then, to generate an instance with $k = 2$, we iterate over all jobs in the $k = 1$ instance. For each such job, we generate a $k = 2$ job from the same motif, scale the job so that the total duration and energy consumption is the same as for the $k = 1$ job, and set the same window. We proceed in the same way for all values for $k$.

## 6.5 The Benchmark Data Set

Based on the process to generate instances as described in Section 6.4, we now describe the real-world data in which the instance sets are based, the concrete instance sets we generated and explain how to obtain these instance sets and the accompanying auxiliary data.

### 6.5.1 Data Origin

Real-world data forms the basis for our generated instances. This real-world data comes from a data set of smart meter measurements in a small scale electronics factory and is called HIPE (see Bischof et al. [Bis+18]). In the present chapter, we use a subset in machines and superset in time of the originally published HIPE data set. The instances are generated based on 6 machines: a chip press, a high temperature oven, a screen printer, a soldering oven, a vacuum oven and a washing machine. We only use this subset of machines since for each of the other machines, the data quality for the selected time range (see below) was questionable for various reasons. All of the machines have been equipped with smart meters which record several quantities such as voltages, currents, frequencies etc. at a frequency of 50Hz. Out of a large number of measured quantities we only consider the measured active power. The first measurement we use is the from 31.12.2016 10:00 pm, while the last measurement is from 31.12.2018 10:59 pm. We thus use two full years of data. We down-sample the data to one minute resolution, where the one minute values are the mean values from the original 50Hz measurements during that minute. Due to some problems during the recording of the measurements, not all machines have data for all minutes in the considered time period. For the machines with a complete set of power values we consider a total of 1,051,260 minutes. For more information on the origin of the data and the machines we refer the interested reader to Bischof et al. [Bis+18].

**Table 6.1:** Generated instance sets parameters. Three values $(a, b, c)$ in a cell indicate that a range of parameters was chosen: from minimum $a$ to maximum $b$ (inclusive), with steps of size $c$.

| Parameter Name | Chosen Values |
|---|---|
| Job Count | (200, 500, 15) |
| Window Growth Mean | (50, 200, 50) |
| Window Growth Std.Dev. | 20 |
| Window Base Factor | (0.05, 0.15, 0.05) |
| Time Horizon | 5 days |
| Time Resolution | 1 minute |
| Block Count | {1, 2, 3, 4, 5, 7, 10} |

### 6.5.2 Data Set Parameters and Publication

The instance generation process from Section 6.4 requires several parameters to be set. Table 6.1 list the parameters we choose for the instance set we generate. If a table cell contains three values like $(a, b, c)$, that means we choose all values from $a$ to $b$ (inclusive) in increments of $c$. Between all parameters where we choose more than one value, we take the Cartesian product to obtain the final parameter space. The chosen parameter space results in a total of 1764 instances.

We publish the instance set generated as above together with some auxiliary data as a separate data publication [Lud+19a] accessible at

`https://publikationen.bibliothek.kit.edu/1000094324`

The data archive itself contains a detailed description of its contents and the file formats. The instance file format is suitable to be used with the TCPSPSuite software package,[1] which is what we used for all optimizations performed during the evaluation. The auxiliary data includes a description of the motifs discovered (as described in Section 6.3) as well as for every instance the best solution we computed during our evaluation. These solutions can be used as a baseline for benchmarks.

## 6.6 Evaluation: Characteristics of the Patterns

In Section 6.3 we have outlined how to extract the patterns from the machine time series before we generate a bigger set of instances. In this section, we want to briefly characterize the patterns we have found using the method described above on the HIPE data set. Table 6.2 summarises how many sequences and patterns were found for each

[1] `https://github.com/kit-algo/TCPSPSuite`

**Table 6.2:** Characteristics of the individual machines and the patterns found in the sequences. Where $\bar{E}$ is the average energy needed in a sequence per machine and $\bar{n}$ is the average length of a sequence per machine.

| Machine | Sequences | Pattern | $\bar{E}(kW)$ | $\bar{n}(min)$ |
|---|---|---|---|---|
| High Temperature Oven | 226 | 7 | 1.13 | 74.50 |
| Screen Printer | 173 | 1 | 0.32 | 285.51 |
| Soldering Oven | 206 | 4 | 1.89 | 146.32 |
| Vacuum Oven | 572 | 7 | 0.41 | 12.57 |
| Washing Machine | 66 | 4 | 1.95 | 188.47 |
| Chip Press | 51 | 2 | 1.09 | 433.63 |

machine as well as the average energy and time they needed. Dependencies among machines often are an argument against any flexibility in operation. We thus want to gather some information from the real machines that help us to determine whether they are dependent on each other. For this purpose, we examine the correlation between the start and end times of all machines, which could indicate a temporal dependency.

As can be seen in Figure 6.2, the these correlations are all relatively low. Therefore, we assume that either all intermediary products of the machines can be stored efficiently in between operating dependent machines, or that the machines do not depend on each other much. Either way, we assume we can ignore dependencies for the moment, as already mentioned in Section 6.4.

We go on to check for other correlations between the operation of the machines and other factors. As shown in Table 6.3, most machines show a correlation between the start time of a process and the length of the process, with shorter processes starting later in the day than longer processes. The main reason for this seems to be the fact that working hours are roughly between 6 am and 6 pm. Thus, any machine or process

**Table 6.3:** Pearson correlation between the length of the processes and the time they are started for all machines. Three asterisks indicate $p < 0.01$.

| Machine | $\rho$ |
|---|---|
| Vacuum oven | -0.01 |
| Washing machine | -0.15 |
| Chip press | -0.42*** |
| High temperature oven | -0.17*** |
| Screen printer | -0.48*** |
| Soldering oven | -0.39*** |

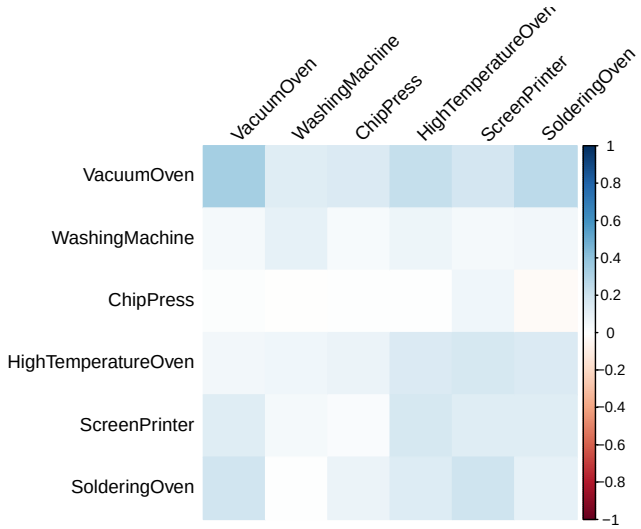



**Figure 6.2:** Pearson correlation between the end times (in minutes after midnight) and start times (in minutes after midnight) among all processes.

which needs supervision or needs to have ended before the workers go home is not started late in the day.

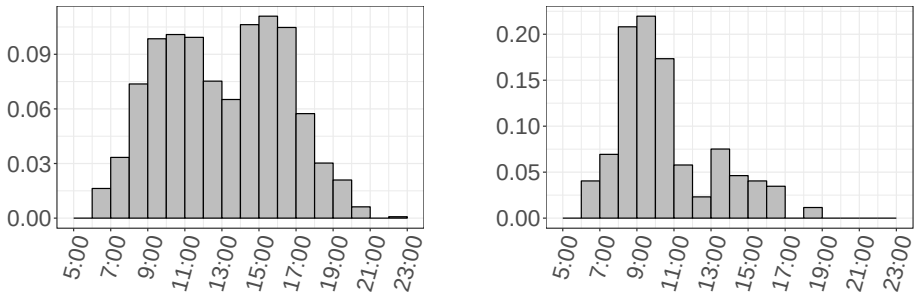
The dependency on working hours seems to also be relevant during the lunch break. As we can see in Figure 6.3 there is a significant drop in starting times of processes during noon. In total, most processes get started during the peaks occurring before and after the lunch break. Most machines exhibit the pattern seen in Figure 6.3a, with the exception of the screen printer which is more often started before noon.

Overall, the starting times are spread over the whole working day and there seems to be only little time restriction on the processes other than when the workers have a break or go home.

## 6.7 Evaluation: Block Decomposition Granularity

Since the data has one-minute time resolution, the power demand curve for every occurrence detected in Section 6.3 is a stepwise function, with one step per minute. However, for many algorithmic approaches, for example scheduling algorithms, the run-time complexity increases significantly if the demand functions have too many steps. Many approaches even can not deal with non-constant demand, i.e., require a demand function with exactly one step.

For this reason, in Section 6.4 we generate jobs with varying, but low numbers of steps in their power demand function. We also call the number of steps in the demand function the number of *blocks* that we decompose a job into. In this section,




**(a)** Hours during which all machines are started.

**(b)** Hours during which the screen printer is started.

**Figure 6.3:** Density of processes started during each hour of a day for all machines Figure 6.3a and the most unusual start time distribution found for the screen printer Figure 6.3b.

we analyze the effects that the number of blocks of a job has. In Section 6.7.1, we evaluate how closely we can approximate the original power demand functions with various numbers of blocks. In Section 6.7.2, we look at the increase in optimization complexity that comes with a rising number of blocks, using a mixed-integer program for the S-RACP as an example. Finally, in Section 6.7.3, we look at how well solutions for instances with low $k$ values can be transferred to the same instances with high $k$ values.

### 6.7.1 Approximation of the Original Power Demand Curve

In Section 6.2.2 we describe a block decomposition that allows to approximate the (stepwise) power demand curve of some original process with a varying number of steps (resp. blocks).

In this section, we analyze how close a given power demand curve with a low number of steps can be to the original curve it tries to approximate. As original curves, we take the occurrences discovered during motif discovery. First, we need a distance measure between two stepwise functions. Given the stepwise demand function of an occurrence $o$ as $P_o$, and a stepwise demand function with $k$ steps $\tilde{P}_{o,k}$ that approximates $P_o$, we use the measure

$$\Delta\left(P_o, \tilde{P}_{o,k}\right) = \frac{1}{N_o} \int_0^\infty \left(P_o(t) - \tilde{P}_{o,k}(t)\right)^2 \mathrm{d}t. \tag{6.1}$$

Here, $N_o$ is a normalization factor determined as $N_o = \int_0^\infty P_o(t)\ \mathrm{d}t$, i.e., the total energy of the original occurrence. Intuitively, the distance between two stepwise functions should correlate with the area between the two curves. However, we assume that —

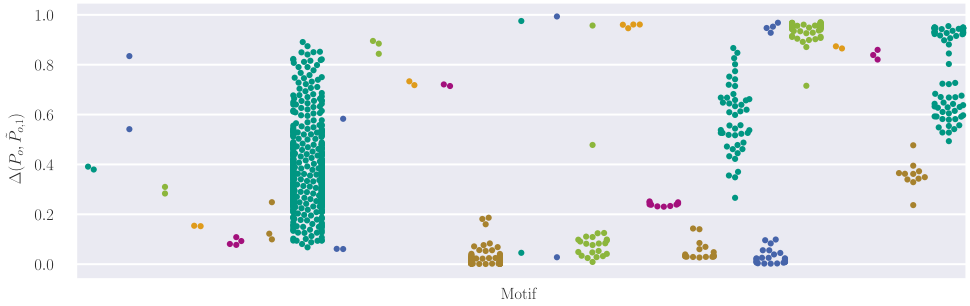



**Figure 6.4:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 1$.

especially for peak shaving applications — a large deviation over a short time is worse than a small deviation over a longer time, which is why we square the difference inside the integral.

We now investigate how well the detected occurrences can be approximated with a low number of steps (resp. blocks) according to this measure. Note that this is very different from generating jobs as described in Section 6.4 and then computing (6.1) for each job. A job is never generated from a single occurrence, and therefore does not approximate any single occurrence's power demand function. Instead, for a given $k$, we compute for every occurrence of every motif a block decomposition that minimizes (6.1). If $\Delta(P_o, \tilde{P}_{o,k})$ becomes small for a given occurrence $o$ and its optimal $k$-block approximation $\tilde{P}_{o,k}$, that means that occurrence $o$ can be approximated well using only $k$ blocks.

Figures 6.4, 6.5 and 6.6 show the $\Delta$ values for $k$ in $\{1, 5, 10\}$ and every occurrence. Larger plots, as well as plots for all $k$ in 1 to 10 plus 15 and 20, can be found in the appendix in Section B.1. In the plots, every dot is one occurrence, which are arranged into columns by their motif. We see that there are some motifs the occurrences of which can be well approximated with only one block. However, for many motifs, one block is not enough for a good approximation. On the other hand, we can also see that the benefit of using more than 5 blocks is small.

Figure 6.7 shows a line plot of the change in $\Delta$ for changing values of $k$ (on the $x$ axis). Here, for every occurrence $o$, and every $k \in \{1, \ldots 10\}$, we set the $y$ value to $\Delta(P_o, \tilde{P}_{o,k}) - \Delta(P_o, \tilde{P}_{o,20})$, giving an insight into how much one can improve the approximation for that occurrence when changing $k$ from its respective value to 20. Here, we again see that from $k = 1$ to $k = 5$, there is a sharp drop in $\Delta$ values, with the curve being rather flat afterwards. Thus, we can conclude that for the processes we mined from the time series data, a block decomposition into 5 blocks might be a good compromise between accuracy and instance complexity.

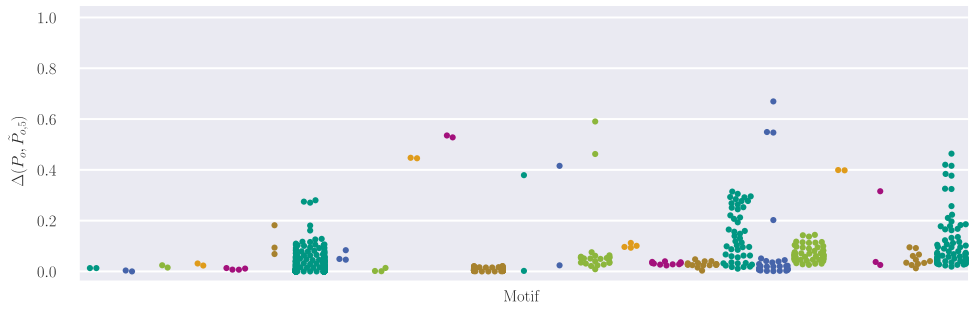



**Figure 6.5:** Δ measures for each occurrence, ordered by motif (on the x axis), for $k = 5$.

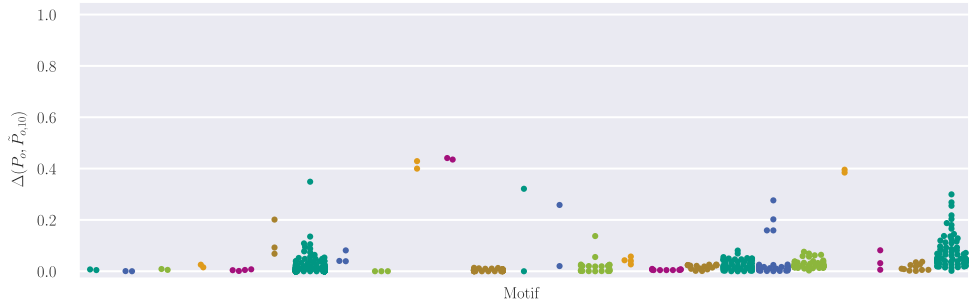



**Figure 6.6:** Δ measures for each occurrence, ordered by motif (on the x axis), for $k = 10$.

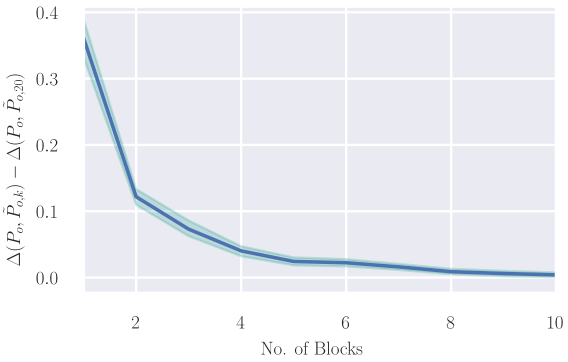


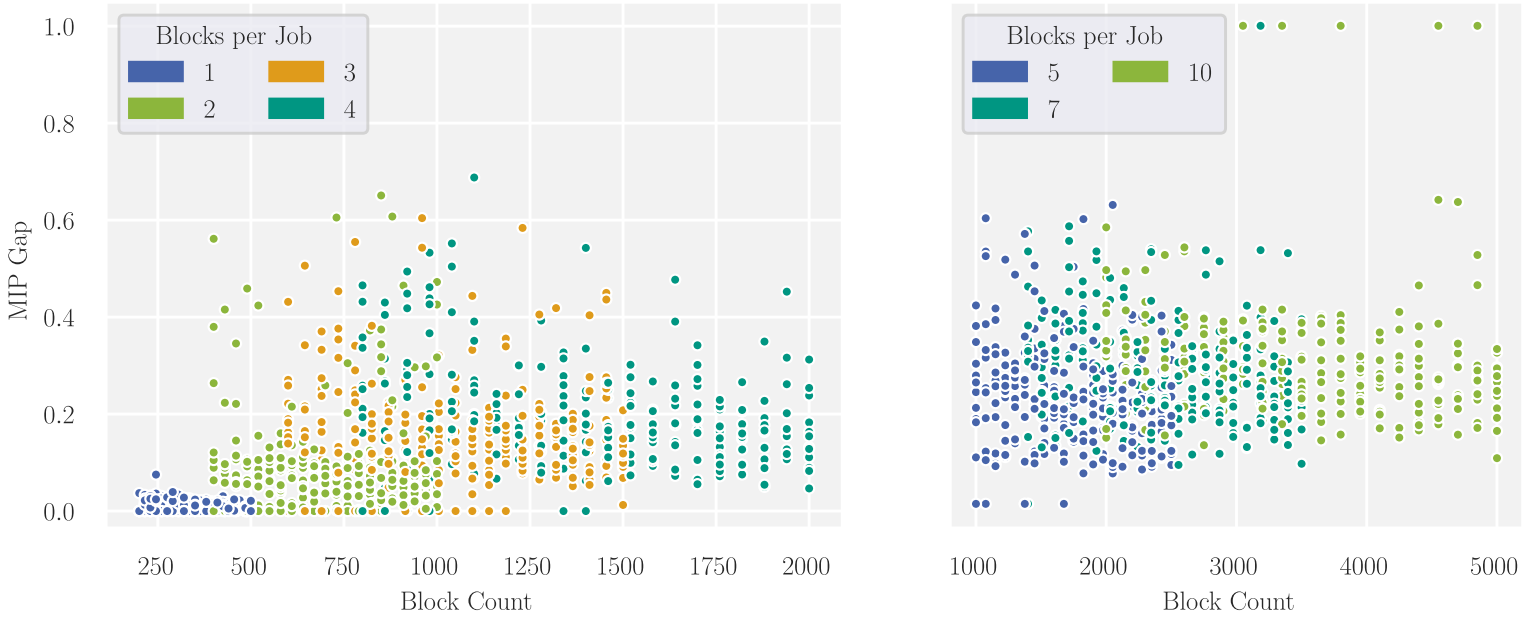

**Figure 6.7:** $\Delta(P_o, \tilde{P}_{o,k}) - \Delta(P_o, \tilde{P}_{o,20})$ for all occurrences, with $k$ being on the horizontal axis. The solid blue line indicates the mean, and all lines for all occurrences fall within the green shaded area.

### 6.7.2 Scheduling Complexity

When choosing the number of blocks to decompose jobs into, one important consideration is the time complexity of the optimization problem one intends to solve. In this section we explore how the complexity of optimizing S-RACP using a mixed-integer program changes with changing block decomposition.

To this end, we use the MIP formulation introduced in Chapter 3, which supports all necessary features for the block decomposition as described in Section 6.2.2 (such as dependencies which negative time lags). We optimize every instance for 30 minutes using Gurobi 7.0 with 16 threads on a machine with 16 Intel® Xeon® E5-2670 CPUs and 64 GBs of RAM. All resulting models fit into the available RAM. Figure 6.8 shows how complexity changes with increasing number of blocks in the instance. Every dot is one optimized instance. The $y$ axis indicates the MIP gap achieved after 30 minutes, the $x$ axis reports the number of blocks in the instance. The color of a dot indicates how many blocks every job in the respective instance has been decomposed into.

We see that for the instances with one block per job, the solver usually achieves a MIP gap of at most 5%. With two blocks per job, the achieved MIP gaps already increase significantly. For most instances, they go up to around 10%, however there is a sizeable fraction of instances that can only be optimized to around 40% MIP gap. Going to three blocks per job again worsens MIP gaps, however there seems to be no drastic further deterioration for four and five blocks per job. For seven and ten blocks per job, most instances can still be optimized to below 40% MIP gap, however we now have some instances with a MIP gap of 100%, i. e., for which the solver could not find any feasible solution.




**(a)** $k \leq 4$ **(b)** $k \geq 5$

**Figure 6.8:** MIP gaps achieved after 30 minutes of optimization versus number of blocks in the instance. Every dot represents one instance. Color denotes the number of blocks each job is decomposed into. For readability reasons split into block decompositions with less than 5 blocks (a) and at least 5 blocks (b). Where the gap is 1.0, the solver did not find any feasible solution within 30 minutes.

These results indicate that if one is to choose at least three blocks per job, one might as well go with a finer decomposition, since it does not increase computational complexity significantly. However, it is already questionable whether expected MIP gaps around 40% are still acceptable. If not, one is restricted to coarser block decompositions, or one needs to invest more computational power or find more performant models.

### 6.7.3 Quality of Schedules with Few Blocks

Regarding the decision of how many blocks to decompose jobs into, an obvious question is how well a solution for a low-block decomposition translates to a high-block decomposition of the same jobs. If the solutions translate well, one does not gain much by choosing a higher number of blocks, and since the number of blocks increases computational complexity (see Section 6.7.2), it would be advisable to chose a low number of blocks.

As described in Section 6.4.1, we generated instances that are suitable to evaluate this question: We generated groups of instances, within which the same processes are decomposed into varying numbers of blocks. The maximum number of blocks we decomposed each job into is 10. We evaluate the quality of a low-block solution as follows: For each job in the low-block instance, determine its computed start time. Then, set that start time as the start time of the corresponding job in the high-block (with $k = 10$) instance. Doing this for all jobs in an instance leads to a new solution for the $k = 10$ instance. We determine the factor between the quality of the so constructed solution and the best solution computed for the $k = 10$ instance.

Figure 6.9 shows the results of this evaluation. Every dot is one instance. The $x$ axis depicts the number of blocks that the jobs in the instances in the respective column were decomposed into. The $y$ axis shows the quality of the solution transferred from $k = \{1, 2, 3, 4, 5, 7\}$ divided by the quality directly computed on the respective $k = 10$ instance. We see a downward trend. While for many $k = 1$ instances, the transfer results in a deterioration by up to 40%, for $k = 7$, the deterioration is mostly limited to 20%. There are some instances where the transferred solution is better than the solution computed on the $k = 10$ instance — this is likely because the $k = 10$ instance was harder to solve and could not be well optimized within the time limit.

Because of these mixed results, we conclude that the approach of transferring a solution from one block decomposition to another itself has a stronger influence on the result than the number of blocks. Therefore, it must be decided on a case-by-case basis whether this approach is valid in practice.

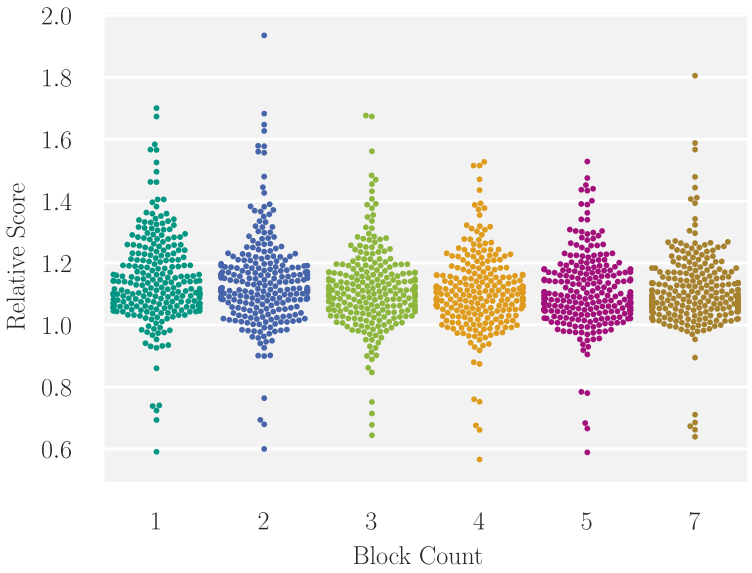



**Figure 6.9:** Every dot depicts an instance with $k$ as per the $x$ axis. The value on the $y$ axis is the relation between the solution obtained by transferring the best solution of the respective instance to the corresponding instance with $k = 10$, and the best solution computed for the corresponding $k = 10$ instance.

## 6.8 Conclusion

We presented a new benchmark data set of industrial demand side flexibility scheduling scenarios. The data set is based on real-world smart meter information from a small industrial facility and has been up-sampled to address large scale problems. The instances in the data set vary in terms of their size, the assumed amount of flexibility and the complexity of their processes' power demand functions, such that the data set covers a wide range of conceivable scheduling problems. We evaluated how the precision of the approximation of the found patterns with different blocks influences the scheduling performance and find that the complexity does not increase much as the job is split in more than three blocks. Additionally, there is no straightforward answer to the question how good a schedule with few blocks is compared to a schedule with more blocks per job. One should decide the used block count on a case-by-case basis, based on the need for accuracy weighted against the need for performance. Please note that it is in no way necessary that all jobs are decomposed into the same number of blocks.

Overall, we believe that the benchmark data set can be used to evaluate scheduling techniques dealing with demand response. The instances are complex enough to

provide a challenge, yet because of the large parameter space diverse enough to point out strengths and weaknesses in the algorithms to be evaluated. Our evaluation has shown that the block decomposition we perform is to a certain extend suitable to reflect the original power demand curves, thus we may assume that our instances resemble real-world scenarios.

In the future, it would be interesting to enrich the data set with additional constraints from real-world scenarios, such as dependencies between processes, storage constraints, and others.

# Part II

# Heuristics

# 7 Exploiting Flexibility in Smart Grids at Scale

Large parts of energy systems all over the world are undergoing drastic changes at the moment. Two of these changes are the increasing share of intermittent generation technologies and the advent of the smart grid. A possible application of smart grids is *demand response*, i.e., the ability to influence and control power demand to match it with fluctuating generation.

We present a heuristic approach to coordinate large amounts of time-flexible loads in a smart grid with the aim of peak shaving with a focus on algorithmic efficiency. A practical evaluation shows that our approach scales to large instances and produces results that come close to optimality.

This chapter is based on joint work with Dorothea Wagner [BW18].

## 7.1 Introduction

The electrical energy system of the future will be based on a *smart grid*, i.e., the confluence of communications and power transmission technology. An anticipated feature of smart grids is *demand response* (DR), which is defined as "changes in electric usage by end-use customers from their normal consumption patterns [...] to induce lower electricity use at times of high wholesale market prices or when system reliability is jeopardized" by the U.S. Department of Energy [US 06]. Demand response can be facilitated by various means, such as time-of-use pricing, dynamic tariffs with price signals, or centralized demand response. In the latter case, a central authority controls devices connected to the smart grid. These devices allow to control their load, for example by shifting it to a different time, modifying the shape of the load curve, or shedding the load altogether. An extensive survey regarding the possibilities of smart grids and demand response is given by Siano [Sia14].

Throughout this chapter, we deal with centralized demand response (also called direct load control) and *load shifting*, i.e., the shifting of the unmodified load curve of devices to a more desirable time. While this might technically be possible in the foreseeable future, the owners of the devices might not want to relinquish control over their devices. This can be addressed with (financial) incentives for allowing such control; however, these considerations are beyond the scope of this chapter.

Desirable and undesirable times for electrical loads can be the result of a high level of renewable generation in the electrical system. In contrast to conventional generation such as coal, gas or nuclear power, solar and wind power cannot be dispatched, i.e., the generation cannot be controlled to match a fluctuating demand. It can therefore

be desirable to shift loads away from times of high demand or low solar and wind availability.

From this arises the objective of *peak shaving*, where the goal is to minimize the maximum power requirement that exceeds the renewable generation. Intuitively, this peak corresponds to the maximum capacity of the conventional generation that needs to be activated to satisfy all demand. Hence, demand response for peak shaving can reduce the necessary installed conventional capacity, as for example Zibelman and Krapels [ZK08] show. Earle et al. [EKM09] also come to this result, but also argue that the positive effect is diminished if the demand response is uncertain, e. g. because it is based on price signals and relies on customers acting on these signals. With direct load control, such uncertainties can be avoided.

### 7.1.1 Related Work

Topics revolving around the smart grid are currently very active throughout the energy community. Fang et al. [Fan+11] give an overview over the developments in the field of smart grids. Approaches to exploit the flexibility offered by smart devices can be loosely separated into two groups, one considering flexibility in household contexts, one considering industry contexts. In the first group, Allerding et al. [AMS14] present an evolutionary algorithm aimed at scheduling devices within a smart building that is equipped with generation. Li et al. [Li+12] present a mixed-integer linear programming approach to schedule household appliances. Pedrasa et al. [PSM10] use particle swarm optimization to schedule electrical loads in households, where some loads can be shed.

In the context of industry, Ashok [Ash06] looks at steel plants and argues that these have a large potential for saving money by using their flexibility. Mitra et al. [Mit+12] consider continuous energy intensive processes and state a Mixed-Integer Program (MIP) to optimize these under fluctuating energy prices.

Some work has also gone into abstracting and unifying both household and industry contexts: Petersen et al. [Pet+14] develop a taxonomy of flexible electrical loads. Gottwalt et al. [Got+16] describe how to select a portfolio of flexible electric loads to achieve maximum utility.

Besides the energy community, the task of scheduling flexible electric demands touches two fields of research: in operations research, the Time-Constrained Project Scheduling Problem (TCPSP) is well known, which contains our problem as a special case. Guldemond et al. [Gul+08] give an overview over work related to the TCPSP and propose a solution technique in which jobs may miss their deadline (or need to be completed in "overtime"). The first MIP formulation for the problem is given by Deckro and Hebert [DH89].

The second field touched is the computer science field of machine scheduling. Here, the problem of minimizing the number of machines to schedule a set of jobs is similar to our problem. Cieliebak et al. [Cie+04] first introduce this idea, show its hardness

and present efficient algorithms for special cases. Chuzhoy et al. [Chu+04] present an approximation algorithm to this problem. In machine scheduling problems, jobs usually only take one machine simultaneously. When applying machine scheduling approaches to smart grid scheduling, machines correspond to the power requirement of a job; thus, these approaches only directly apply to smart grid scheduling scenarios where all electrical loads have the same power requirement.

### 7.1.2 Contribution and Outline

We present an iterative heuristic that minimizes electricity usage peaks using load shifting of directly controllable loads in smart grids — the *Resource Utilization Scheduling Heuristic* (RUSH).

Resource Utilization Scheduling Heuristic

Benefits from using flexibility in smart grids increase with the number of controlled loads. Thus, being able to optimize flexible devices at large scales is crucial. Most of prior research focuses on modeling sophisticated constraints or optimization criteria, but neglects algorithmic efficiency. Also, many approaches are based on metaheuristics which are not fine-tuned to the problem at hand. Our presented approach is focused on speed of computation even for very large instances. According to the principle of *algorithm engineering*, we intend to increase the model complexity in future research while trying to maintain algorithmic efficiency.

Section 7.2 formalizes the problem under study. In Section 7.3, we describe the details of the RUSH heuristic in detail. Section 7.4 presents an experimental evaluation of RUSH, analyzing its performance and comparing result quality to results obtained via a mixed-integer linear program. In Section 7.5, we conclude our work and outline what further steps we are going to take with this research.

## 7.2 Problem Formulation

We define an instance of our problem as a set of (electrical) loads with release times, deadlines, execution times and power requirements. These loads can represent for example individual cycles of devices such as refrigerators or A/C units, runs of one-off devices such as ovens, or batches in industrial processes.

Formally, let $n$ be the number of loads. Then, each load $j \in \{1, \dots n\}$ is described by a tuple

$$(r_j, d_j, p_j, u_j) \in \mathbb{N}^3 \times \mathbb{R}$$

where $r_j$ (the *release time*) is the first time step in which $j$ may be executed, $d_j$ (the *deadline*) is the first time step in which $j$ must be finished, $p_j$ (the duration or *processing time*) is the number of time steps $j$ must be active consecutively and $u_j$ (the *usage*) is the amount of power required by $j$ during its execution. Note that we model time

release time
deadline
processing time
usage

steps as discrete, while power is continuous. Given all loads, we define a global release time $R = \min_i\{r_i\}$ and a global deadline $D = \max_i\{d_i\}$.

The objective is to find an assignment $s \in \mathbb{N}^n$ of start times that minimizes the peak power requirement over all time steps. For such an assignment to be *feasible*, it must hold for every load $j$ that $s_j \geq r_j$ and $s_j + p_i \leq d_i$. We will call a feasible assignment a *schedule*.

schedule

Note that this model aims to minimize the peak demand. However, as stated in the introduction, one is usually interested in minimizing the peak amount of power that exceeds renewable generation, the so-called *residual load*. In our model, this can easily be achieved by introducing a set of immovable (by virtue of deadlines and release times) jobs that represent the difference between the amount of renewable generation available during the respective time interval and the maximum renewable generation.

residual load

## 7.3 Resource Utilization Scheduling Heuristic

In this section, we describe how RUSH works, starting with a high level overview.

Given a problem instance as defined in Section 7.2 and a feasible schedule to the problem (i.e., a start-time assignment $s \in \mathbb{N}^n$), we define its *demand profile* as a sequence of intervals, each of which marks a time span in which the schedule requires roughly the same amount of power.

demand profile

Formally, let $U_s(t)$ be the power requirement at time step $t$ induced by schedule $s$. To group time steps with roughly the same power requirement together, we define a set of power *levels*. Let $\lambda$ be a parameter specifying the size of each power level. We then say that during time step $t$, the schedule $s$ executes in power level $l$ if and only if $l\lambda \leq U_s(t) < (l+1)\lambda$.

For a given schedule, profile $\mathcal{P} = (\mathcal{I}, \mathcal{L})$ consists of a sequence of intervals $\mathcal{I}$ and a function $\mathcal{L} : \mathcal{I} \to \mathbb{R}^+$. Here, $\mathcal{I}$ is a sequence of consecutive, disjunct, right-open intervals spanning the whole scheduling horizon, i.e., $\cup_i \mathcal{I}_i = [R, D]$, $\cap_i \mathcal{I}_i = \emptyset$ and $\sup(\mathcal{I}_i) = \min(\mathcal{I}_{i+1})$. Correspondingly, for $\mathcal{I}_i \in \mathcal{I}$, $\mathcal{L}(\mathcal{I}_i)$ states the power level that the schedule is in during $\mathcal{I}_i$.

The working principle of RUSH is to first generate a feasible schedule by scheduling every load at its release time, and then repeatedly pick a load $j$, determine the highest power level during the scheduled execution of $j$ (let this be $\hat{l}$) and then move $j$ so that the total time the schedule executes in power level $\hat{l}$ is minimized while not increasing the time executed in any level above $\hat{l}$.

For such an iterative approach it is desirable to have the individual iterations execute as efficiently as possible, so that the algorithm arrives at a satisfactory solution as quickly as possible. Because of this, most computations performed by RUSH can be implemented as a set of simple operations on sets of intervals. Abusing notation a bit, we treat $\mathcal{I}$ as a function which is the inverse of $\mathcal{L}$: Let $\mathcal{I}(l) = \{\mathcal{I}_i : \mathcal{L}(\mathcal{I}_i) = l\}$ be the

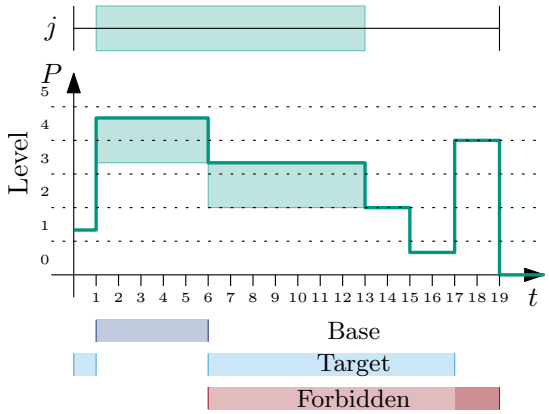



**Figure 7.1:** Example setting before moving load $j$. The green rectangle corresponds to the length and power requirement of $j$. On the top, the release and deadline of $j$ are shown together with $j$. Below is a graphical depiction of a possible profile, its discretization into levels, and $j$'s contribution to it. Base level, target and forbidden sets are indicated below.

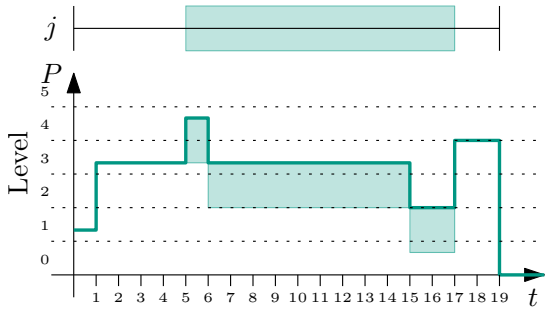



**Figure 7.2:** Example setting after moving load $j$.

set of intervals in the profile where the schedule is in level $l$. Also, for a fixed load $j$, let $\mathcal{P}^j = (\mathcal{I}^j, \mathcal{L}^j)$ be the profile derived from $\mathcal{P}$ by removing load $j$.

For a fixed load $j$, define the *base level* (denoted $l^*$) of load $j$ as the maximum level in $\mathcal{P}^j$ during the scheduled execution in $j$. Consider Figure 7.1. Here, the maximum power demand *below* the highlighted load $j$ is in level 3. Thus, $l^* = 3$ in this example.

Intuitively, executing $j$ during any of $\mathcal{I}^j(l^*)$, i.e., the times in which the schedule without $j$ executes in the base level, results in power consumption roughly equal to the old maximum power consumption during the execution of $j$. Thus, we try to minimize the amount of $j$ we schedule during $\mathcal{I}^j(l^*)$. We also forbid scheduling any of $j$ during any $\mathcal{I}^j(l^* + k)$ for any $k > 0$, i.e., no part of $j$ may be scheduled on top of a level that is higher than the base level. This leaves $\mathcal{T}_j = \cup_{i \in \{0,\dots l^*-1\}} \mathcal{I}^j(i)$ as the desirable area to schedule $j$ in, called the *target*. In the example of Figure 7.1, the target is everything in level 2 and below (after $j$ has been removed), i.e., $\mathcal{T}_j = \{[0, 1), [6, 17)\}$. We further

define a set of *forbidden* intervals, in which $j$ may not be started. We do so by taking all intervals of levels above $l^*$ in $\mathcal{I}^j$ and extending these intervals by $p_j - 1$ to the left:

$$\mathcal{F}_j = \left\{ [\max(0, a - (p_j - 1)), b) : [a, b) \in \bigcup_{i > l^*} \mathcal{I}^j(i) \right\}$$

In the example of Figure 7.1, the only time the profile (after removal of $j$) is in a level above $l^* = 3$ is in $[17, 19)$. Since $j$ has length 12, that results in $\mathcal{F}_j = \{[6, 19)\}$. It is easy to see that starting $j$ during any of these intervals would cause a part of $j$ to be scheduled during a higher power usage than before.

Observing that $d_j - p_j + 1$ is the latest time step at which we can start $j$ without missing its deadline, we now combine everything. We want to find a start point in $[r_j, d_j - p_j + 1) \setminus \mathcal{F}_j$ that maximizes $j$'s overlap with $\mathcal{T}$. We can show that to find this optimum position, it is sufficient to check the borders of the intervals in $[r_j, d_j) \setminus \mathcal{F}_j$ plus all time steps $t$ for which $j$ would be right-aligned in an interval in $\mathcal{T}_j \cap [r_j, d_j)$, i.e., the set of candidates for start positions is

$$\begin{aligned} C_j = \quad & \{x : [x, \cdot) \in ([r_j, d_j - p_j + 1) \setminus \mathcal{F}_j)\} \\ & \cup \{x : [\cdot, x) \in ([r_j, d_j - p_j + 1) \setminus \mathcal{F}_j)\} \\ & \cup \{\max(x - p_j, 0) : [\cdot, x) \in \mathcal{T} \cap [r_j, d_j)\} \end{aligned}$$

For each $\hat{s} \in C_j$, we check the size of $[\hat{s}, \hat{s} + p_j) \cap \mathcal{T}_j$ and select the start point with the largest overlap.

Putting everything together, one iteration of RUSH works in these five steps:

1. Randomly select a load $j \in \{1, \ldots n\}$
2. Remove $j$ from the current schedule
3. Compute $\mathcal{I}^j, \mathcal{L}^j, \mathcal{T}_j, \mathcal{F}_j$ and $C_j$ as above
4. In $C_j$, find the start candidate that results in the largest overlap with $\mathcal{T}_j$
5. Re-insert $j$ into the schedule at this point

In the example in Figure 7.1, the set of candidates is $\{0\} \cup \{5\} \cup \{5\}$ as per the above formula. Indeed, setting $s_j = 5$ results in the largest overlap of $j$ with the target of $\{[0, 1), [6, 17)\}$. Moving $j$ to start at time step 5 results in $j$ being executed for only one time step inside level 4, as seen in Figure 7.2 instead of five time steps in Figure 7.1.

**Efficient Implementation.** The sets computed in Step 3 above can all be computed via efficient set operations from $\mathcal{L}$ and $\mathcal{I}$. More precisely, for $\mathcal{T}_j$ (resp. $\mathcal{F}_j$), we need unions in the style of $\cup_{i \geq k} \mathcal{I}(i)$ (resp. $\cup_{i \leq k} \mathcal{I}(i)$) for some value $k$. To facilitate efficient access to these unions, we store each of them directly. However, since in our case $\cup_{i \geq k} \mathcal{I}(i) = [R, D) \setminus \cup_{i \leq k-1} \mathcal{I}(i)$, it suffices to store $\cup_{i \leq k-1} \mathcal{I}(i)$ for all possible levels $k$. We do so using Boost's ICL data structures.[1]

## 7.4 Experimental Evaluation

We evaluate RUSH experimentally by using it to schedule a set of instances of the problem introduced in Section 7.2. Runs of RUSH were conducted on a machine with four Intel Xeon E5-1630 cores at 3.7 GHz (of which only one was used) and 128 GB of RAM, the baseline mixed-integer linear program (MIP) was run on a machine with 16 Intel Xeon E5-2670 cores at 2.6 GHz and 64GB of RAM, using Gurobi 6.5 as a solver.

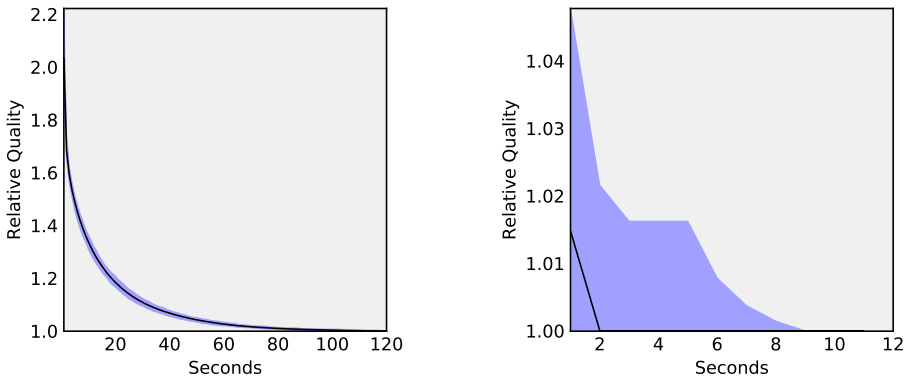
We generated two groups of instances: One set of medium-sized instances and a set of very large instances. For the medium instances, the baseline MIP yields acceptable lower bounds within reasonable time, allowing us to compare the solution quality obtained by RUSH to a lower bound. On the set of large instances, we demonstrate the scalability of RUSH.

For the set of medium-sized instances, the number of loads per instance was drawn uniformly at random from the range $[100, 400]$. For the large instances, we generated 10000 loads per instance.

The duration of all loads was also randomly drawn, from a normal distribution with a mean of 30, a standard deviation of 20, and a minimum value of 1. For each load, we assigned the release time uniformly at random between 0 and 200 (0 and 2000 for the large instances), and the *slack*, that is the difference between deadline minus release time and execution time, i.e., the amount of flexibility of a load, uniformly between 0 and 200 (0 and 2000 for large instances).

Finding randomly generated instances which adequately represent real smart grid scheduling problems is not easy, and has been done in various ways throughout literature. Petersen et al. [Pet+14] randomly chose all loads lengths from $\{2, 3, 4, 5\}$, all power requirements from $\{1, 2, 3, 4\}$ and the deadline for each load from $\{1, \ldots 100\}$ while all loads are released at $t = 1$. Li et al. [Li+12] explicitly model four household appliances: kettles, toasters, ovens and refrigerators. Each has a unique power consumption and pattern of release, deadline and duration. Yaw et al. [Yaw+14] also model individual household appliances, however they base their models on consumption profiles obtained from the REDD data set [KJ11]. We intentionally chose to generate our instances randomly and not by explicit modeling, since we think that a small number of household appliances alone are not sufficient to represent the various

[1] http://www.boost.org/doc/libs/1_64_0/libs/icl/doc/html/index.html




**(a)** Large Instances **(b)** Medium Instances

**Figure 7.3:** Convergence speed of RUSH ⟨4000⟩. Measurements were taken every second. All measurements fall within the blue area. The black line indicates the median value at every point in time. All results are normalized by the optimum value RUSH ⟨4000⟩ computes on the respective instance.

demands in the electrical system. By picking values from a continuous distribution instead of a fixed set we tried to have as much diversity among the loads as possible.

On each of these instances, we ran the baseline MIP for a maximum of 1200 seconds. For most instances, an optimal solution could not be found in this time. However, from the MIP runs we obtain lower bounds on the optimal solution quality as well as non-optimal feasible solutions. The MIP used to compute a baseline is the formulation presented in Chapter 3. The basic idea is to introduce a matrix of $n \times (D - R)$ binary variables $x_{jt}$, where $x_{jt} = 1$ if and only if load $j$ starts in time step $t$.

### 7.4.1 Results

It is easy to see that the quality achieved by RUSH depends on the number of levels that RUSH discretizes the power requirement into. We ran RUSH on all instances with 20, 40, 60, 200, 1000 and 4000 levels each. We denote RUSH with $k$ power levels as RUSH⟨$k$⟩.

We first examine the speed with which RUSH converges against a solution. The individual iterations become more expensive the finer the power consumption is discretized, i.e., the more power levels RUSH uses. Thus, we mainly look at the speed of RUSH⟨4000⟩, the largest number of levels we evaluate.

Figure 7.3 shows for every of the runs of RUSH⟨4000⟩ on each instance of medium resp. large size how close each computation got to the final result after what time. Note that a $y$-value of 1.0 in this case does not indicate the global optimum of the respective

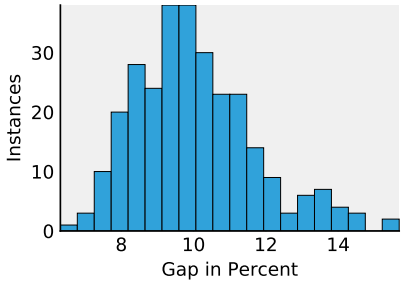
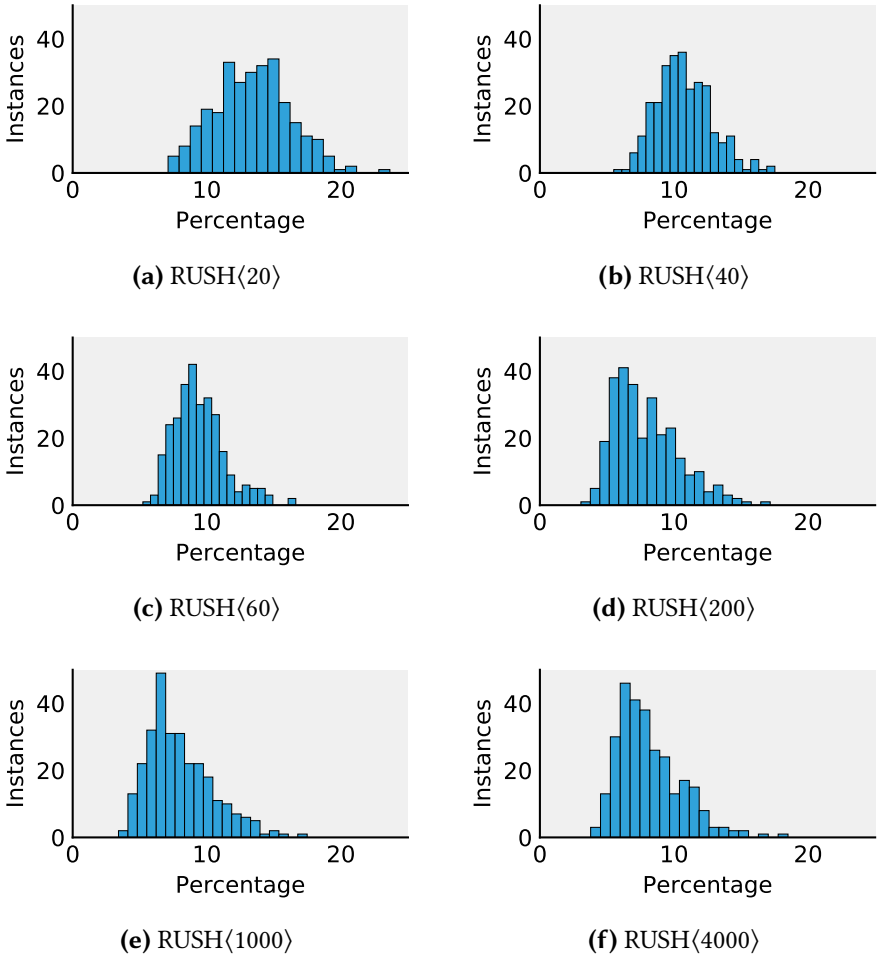



**Figure 7.4:** Histogram of the MIP gaps achieved by the MIP solver.




**(a)** RUSH⟨20⟩ **(b)** RUSH⟨40⟩

**(c)** RUSH⟨60⟩ **(d)** RUSH⟨200⟩

**(e)** RUSH⟨1000⟩ **(f)** RUSH⟨4000⟩

**Figure 7.5:** Relative quality of variations of RUSH compared to the baseline derived from the MIP.

**Table 7.1:** Median, upper quartile and maximum values for the quality gap achieved by the RUSH variants.

| Algorithm | Median | 0.75-quantile | Max |
|---|---|---|---|
| RUSH⟨20⟩ | 13.5 | 15.3 | 23.7 |
| RUSH⟨40⟩ | 10.5 | 12.1 | 17.5 |
| RUSH⟨60⟩ | 9.1 | 10.4 | 16.6 |
| RUSH⟨200⟩ | 7.4 | 9.4 | 17.2 |
| RUSH⟨1000⟩ | 7.4 | 9.4 | 17.5 |
| RUSH⟨4000⟩ | 7.6 | 9.4 | 18.5 |

instance, but rather the optimal value that RUSH achieved. Measurements were taken every second. While on medium instances, the near-optimum value is achieved after ten seconds for all runs, it takes about 100 seconds on the large instances. On medium instances, the median solution is near-optimal after two seconds already.

We now take a look at the quality of the computed solutions. The MIP is not suited to solve a significant number of medium-sized instances to optimality within reasonable time. Therefore, we take the average between lower bound and quality of the best feasible solution found by the MIP as baseline. Since the MIP does not compute anything useful on the large instances, the quality comparison is only done on the medium instances. Figure 7.4 shows a histogram of the gaps that the MIP achieved on the medium instances after 1200 seconds, i.e., the ratio between best feasible solution found and best lower bound.

Figure 7.5 show histograms of the gap between the baseline and the solutions achieved by RUSH with the various levels. Table 7.1 reports median, upper quartile and maximum values. It can be seen that going from 20 to 200 levels, the average solution quality improves significantly, even though the worst case is not improved by much after 40 levels. From 200 levels upwards, result quality does not improve much anymore. In fact, solution quality does even deteriorate marginally when going from 1000 to 4000 levels. This could be explained by the fact that in RUSH⟨4000⟩, individual iterations are more expensive and therefore less iterations can be computed within the given time limit.

## 7.5 Conclusion and Future Work

We have presented RUSH, an iterative heuristic to exploit flexibility in smart grids on a large scale. We have shown that RUSH yields results with good quality in short time. Still, this approach is research in progress.

There are several directions that seem promising. First, RUSH can be improved to yield even better results. Second, RUSH can be extended to be applicable to more complex models. Finally, a more thorough evaluation of RUSH, and a direct comparison against prior algorithmic approaches should be done.

To improve RUSH, we intend to find a smarter way of (randomly) selecting the load to be moved in each iteration. Also, we did not yet pay any attention to the solution RUSH starts with. Using a simple list scheduling heuristic instead of initially scheduling every load at its release time might lead to faster convergence.

In terms of extending RUSH, the first thing that comes to mind is adding dependencies between loads, i.e., allowing loads to only start after some predecessors finished. What needs to be done is to limit the feasible range of start candidates in Step 4 of the algorithm presented in Section 7.3. Regarding the evaluation, while we show that RUSH arrives at good solutions in very short time, solution quality and performance should be directly compared to competing algorithmic solutions such as the GRASP-based metaheuristic by Petersen et al. [Pet+14] or the PDM heuristic by Yaw et al. [Yaw+14].

Finally, since our heuristic works iteratively, a technique like simulated annealing to climb out of local optima seems applicable: After every iteration, one can decide to accept the modified solution or go back to the previous solution. A scheme in which this decision is driven by some cool down factor as in simulated annealing might be beneficial.

# 8 Shaving Peaks by Augmenting the Dependency Graph

Demand response (DR) is an important building block for future energy systems, since it mitigates the non-dispatchable, fluctuating power generation of renewables. For centralized DR to be implemented on a large scale, considerable amounts of electrical demands must be scheduled rapidly with high time resolution. To this end, we present the Scheduling With Augmented Graphs (SWAG) heuristic. SWAG uses simple, efficient graph operations on a job dependency graph to optimize schedules with a peak shaving objective. The graph-based approach makes it independent of the time resolution and incorporates job dependencies in a natural way. In a detailed evaluation of the algorithm, SWAG is compared to optimal solutions computed by a mixed-integer program. A comparison of SWAG to another state-of-the-art heuristic on a set of instances based on real-word consumption data demonstrates that SWAG outperforms this competitor, in particular on hard instances.

This chapter is based on joint work with Dorothea Wagner [BW19d].

## 8.1 Introduction

In large parts of the world, the electrical energy system is changing towards larger shares of renewable generation. While this is highly desirable, it presents the maintainers of these systems with a new challenge: In the past, power generation could be controlled and be made to match the power demand. Renewable power plants can often not be controlled, fluctuate in generation and one must rely on uncertain forecasts for wind and solar irradiation. There exist multiple strategies to still ensure that generation matches demand, most notably energy storage, the expansion of the transmission network and *demand response* (DR). Demand response is a general term for techniques which influence the power demand at certain times to better match what is generated. The U.S. Department of Energy [US 06] defines DR as "changes in electric usage by end-use customers from their normal consumption patterns [...] to induce lower electricity use at times of high wholesale market prices or when system reliability is jeopardized."

One way to categorize DR strategies is by how the customer is motivated to participate, i.e., by the reward structure. Usual strategies include time-of-use tariffs or flexibility auctions (see Siano [Sia14]). Another important dimension of DR techniques is how the flexibility provided by the consumers is coordinated and controlled. One can differentiate between indirect control (e.g. via time-of-use tariffs), decentralized control (e.g. using decentralized algorithms), or direct load control, in which a central entity

controls a set of flexible electrical demands. This *direct load control* is the scenario we focus on, without regard for how consumers are rewarded in this setting. We assume that the electrical demands can be separated into discrete processes (or *jobs*), and that these processes can be moved in time by the central controller. In this work, we do not consider the possibilities that the demand of processes can be changed in their shape or magnitude, or that certain processes could be shed altogether.

In an energy system with a high share of renewable generation, peak demand must often be accommodated by rapidly responding, dispatchable conventional power plants such as gas turbines, or large battery storage. Also, the transmission and distribution networks must be dimensioned for peak demand. To reduce the costs for building these networks and having this energy storage or generation capacity available, *peak shaving*, i.e., reducing the maximal power demand as much as possible, is an important optimization criterion. To achieve peak shaving, scheduling algorithms are necessary which are able to schedule large amounts of loads quickly. Speed is essential for these scheduling applications — not only because trading at energy exchanges happens at a high pace, but also because it will be necessary to rapidly respond to changes in the scheduling scenario as generation fluctuates unexpectedly or the set of processes that need to be scheduled changes. Also, especially in scenarios reflecting the processes of large industrial plants, one has to expect and cope with large problem instances.

### 8.1.1 Our Contribution

SWAG

We present the Scheduling With Augmented Graphs (SWAG) heuristic, a graph-based scheduling algorithm for flexible electrical demands focused on industrial settings. The algorithm is based on a job-dependency graph, which captures finish-start dependencies between the processes to be scheduled. This approach results in a much smaller solution space than algorithms that directly work with start times have to cope with. Also, this representation makes it possible to schedule with arbitrary time resolution without impacting efficient computation.

We provide an in-depth evaluation of our algorithm on instances that are based on real-world consumption data. We demonstrate the utility of the algorithm on large instances and compare it to a state-of-the-art algorithm by Petersen et al. [Pet+14] as well as a mixed-integer program. We publish everything we implemented, including the comparison algorithm as well as the mixed-integer program, and our test instance set.

### 8.1.2 Related Work

The field of demand response has received much attention lately. Among the numerous reviews of the field, a general survey is provided by Siano [Sia14], while Vardakas et al. [VZV15] provide a survey with focus on the methodology for implementing DR. A

survey with a stronger focus on the modeling of DR problems is provided by Deng et al. [Den+15]. A review by Good et al. [GEM17] examines the challenges and enablers that DR faces.

Scheduling problems aside from the smart grid have been looked into intensively both by the computer science and the operations research community. The field of computer science mostly focuses on *machine* scheduling, where discrete processes must be assigned to machines. This flavor of scheduling sometimes comes in the form of *real-time scheduling*. A review is given by Chen et al. [CPW98]. There are machine minimization problems with the objective of scheduling a set of jobs on a minimum number of machines. These are effectively a special case of the problem we consider, namely the case where all jobs would have unit power demand. Such a problem is first considered by Cieliebak et al. [Cie+04], who show its $\mathcal{APX}$-hardness and also give approximation algorithms for two special cases. Approximation algorithms for further special cases are given by Yu and Zhang [YZ09].

The problem considered in this chapter is a special case of what in the operations research community is known as the Resource Acquirement Cost Problem (RACP), itself a special case of the Time-Constrained Project Scheduling Problem (TCPSP). An overview over various project scheduling problems is provided by Węglarz [Węg99]. The RACP has first been tackled by Möhring [Möh84] and Demeulemeester [Dem95]. Recently, Guldemond et al. [Gul+08] use an ILP to solve a variant of RACP, while Ranjbar [Ran13] uses a metaheuristic based on path relinking.

There also is a considerable amount of work on scheduling with special regards to the smart grid. In Chapter 3, we provide an overview over various approaches based on mathematical programming. One of the earliest works is by Hsu and Su [HS91], who use a dynamic programming approach. While this technique yields optimal results, it does not scale to large instances. For large instances, Petersen et al. [Pet+14] use a metaheuristic to solve a problem similar to the one considered in this work. Yaw et al. [Yaw+14] give two simple combinatoric algorithms for peak demand scheduling problems with certain constraints. Logenthiran et al. [LSS12] use an evolutionary algorithm to schedule large amounts of loads in a simulated scenario.

## 8.2 Preliminaries

This section formalizes the considered problem and introduces notation used throughout the chapter.

### 8.2.1 The Problem

The problem under study in this chapter is the Single-Resource Acquirement Cost Problem (S-RACP), which is a special case of the Resource Acquirement Cost Problem (RACP).

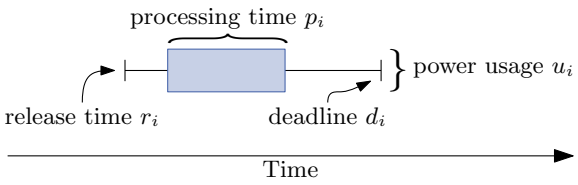



**Figure 8.1:** A graphical representation of a job in an S-RACP instance. With the release time, deadline and processing time as given, the box can be moved within the two whiskers, while the box's height represents the job's power demand.

job
dependency graph
release time
deadline
processing time
usage

An *instance* of S-RACP consists of a set $J$ of *jobs* and a directed *dependency graph* $\mathcal{G}$, the latter of which captures finish-start dependencies between the jobs to be scheduled. We assume $n$ to be the number of jobs, i.e., $n = |J|$. In the job set $J = \{j_1, j_2, \dots j_n\}$, each job $j_i$ is a four-tuple: $j_i = (r_i, d_i, p_i, u_i) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{R}$. For job $j_i$, the *release time* $r_i$ states the earliest time at which $j_i$ can be executed. In turn, $d_i$ states the *deadline*, i.e., the time at which $j_i$ must be finished. The *processing time* $p_i$ indicates how long $j_i$ must be executed without interruption. Finally, $u_i$ specifies the *usage* of $j_i$, i.e., how much power (the single resource of the project scheduling problem) $j_i$ requires. Such a job is depicted in Figure 8.1. For the dependency graph $\mathcal{G} = (V, E)$ we set $V = J$, i.e., we treat the jobs $J$ as vertices. The edge-set $E$ of $\mathcal{G}$ specifies dependencies between jobs: An edge $(j_a, j_b) \in E$ indicates that $j_b$ can start only after $j_a$ has finished.

schedule

Given such an instance, a *schedule* $S$ is a set of start times, one for each job: $S = (s_1, s_2, \dots s_n) \in \mathbb{N}^n$. Such a schedule is *feasible* if:

- Every job respects its limits, i.e., for all $i \in \{1, \dots n\}$, it holds that $s_i \geq r_i$ and $s_i + p_i \leq d_i$,
- and dependencies are respected, i.e., if $(j_a, j_b) \in E$, then $s_a + p_a \leq s_b$.

For a (feasible) schedule, the demand at a point in time $t$ is the sum of the power demands of all jobs active during $t$. The *peak demand* is then the maximum over all demands. A feasible schedule is an *optimal* schedule, if there is no other feasible schedule with less peak demand. Formally, we state the S-RACP as follows:

S-RACP

**Problem 2** (Single-Resource Acquirement Cost Problem (S-RACP))**.**
*Given a problem instance as $J$ and $\mathcal{G}$ as defined above, find the feasible schedule $S^*$ with the minimal peak demand.*

The S-RACP problem as defined above can be classified in the classification scheme by Herroelen et al. [HDD99] as $1|cpm, \rho_j, \delta_j|av$.

### 8.2.2 Notation

We now introduce some notation used throughout this chapter. First, we need a special kind of schedules.

Given an S-RACP instance (as $J$ and $\mathcal{G}$), we can define the *left-shifted* schedule that corresponds to $\mathcal{G}$ as the schedule in which every job starts as early as possible. left-shifted

**Definition 6** (Left-Shifted Schedule)**.**
*A schedule $S = (s_1, \dots s_n)$ is a left-shifted schedule for the Single-Resource Acquirement Cost Problem instance determined by $J$ and $\mathcal{G}$ if for every $j_i \in J$:*

$$s_i = \max\left(\{s_k + p_k : (j_k, j_i) \in E\} \cup \{r_i\}\right)$$

Note that such a schedule can be computed from $\mathcal{G}$ by a simple topological sort of $G$, which is equivalent to the well-known critical-path method (e.g., see [BK12], Chapter 3).

**Optimality.** The algorithm we present works by gradually inserting edges into $\mathcal{G}$. The schedule computed is a left-shifted schedule of the modified graph. Thus, the solution computed by SWAG must always be a left-shifted schedule. It is therefore interesting to show that for every S-RACP instance with $\mathcal{G}$ as dependency graph, there exists a supergraph $G$ of $\mathcal{G}$ (i.e., $G = (J, E')$ with $E' \supseteq E$), such that the left-shifted schedule for $G$ is an optimal schedule for the S-RACP instance.

**Lemma 8.1** (Preservation of Optimality)**.** *Let $J$ and $\mathcal{G} = (J, E)$ be an instance of S-RACP. Then there exists an optimal schedule that is a left-shifted schedule for some dependency graph $G = (J, E')$ with $E \supseteq E'$.*

*Proof.* Let $S^*$ be an optimal schedule. Create $G = (J, E')$ such that $(j_a, j_b) \in E'$ if and only if job $j_a$ ends before $j_b$ starts, i.e., $E' = \{(j_a, j_b) : S^*_a + p_a \leq S^*_b\}$. Since $S^*$ is a feasible schedule, this graph respects the dependencies in $\mathcal{G}$, i.e., it holds that $E' \supseteq E$. Now, let $S'$ be the left-shifted schedule for $G$. The peak demand of $S'$ can not be lower than for $S^*$ by assumption of optimality. Assume that the peak demand of $S'$ is larger than that of $S^*$. Then there must be at least one pair of jobs $j_c$ and $j_d$ executing concurrently in $S'$ but not in $S^*$. However, if $j_c$ and $j_d$ do not execute concurrently in $S^*$, there is by construction an edge between them in $G$, and they cannot execute concurrently in $S'$. □

## 8.3 Scheduling With Augmented Graphs

In this section, we describe the SWAG algorithm using pseudocode to illustrate. However, the description often omits implementation details, especially ways of implementing the described methods in an efficient way. We also publish our actual C++

implementation of SWAG, along with the competitor algorithms used in the evaluation. See Section 8.5 on how to obtain our implementation. To simplify the description of the algorithm, we assume the initial dependency graph $\mathcal{G}$ to be empty in the following. The algorithm modifies the edges of $G$ at various places. If the input instance does have dependencies (i.e., if $\mathcal{G}$ is not empty), one only has to make sure never to delete any of the edges of $\mathcal{G}$. The SWAG algorithm has several parameters (see Section 8.5.2 for how to choose them). Throughout this chapter, we always typeset SWAG's parameters underlined. See Table 8.2 for a full list of parameters.

During its execution, the SWAG algorithm holds and modifies a dependency graph $G$. From this graph results at every point during the execution a left-shifted schedule as defined in Definition 6. The start time of a job according to this left-shifted schedule is referred to simply as the *start* of a job. Also from the graph results for each job a *latest finish* time, i.e., latest the time the job may be executing such that no job misses its deadline. In the left-shifted schedule corresponding to $G$, there is some time interval such that the cumulative demand of all jobs executing during that interval is maximal, i.e., there is no other interval with a larger cumulative demand.[1] We call this interval the *peak range*, and the power demand during this interval the *peak demand*.

The algorithm follows the well-known (e.g., see [Kol96]) pattern of working on a *representation* for a schedule, and then applying a *schedule generation scheme* to create a schedule from the representation. We use the graph $G = (J, E')$ as representation. Then, we use the generation of a left-shifted schedule as described in Section 8.2.2 as schedule generation scheme. The main work lies in creating $G$ such that the peak demand is minimized. To this end, our algorithm starts with $G = \mathcal{G}$ and iteratively adds edges to $G$, i.e., *augments* the graph.

In the following, we first give a high-level overview over SWAG in Section 8.3.1, hiding much of the detail. We then present in sections 8.3.2 and 8.3.3 more detailed descriptions of several aspects as well as some insights into how to make the algorithm more efficient.

### 8.3.1 Algorithm Details

We start the explanation of SWAG by giving a big picture overview, which is outlined in Algorithm 1. The SWAG algorithm works in iterations, which corresponds to the loop from line 2 to line 17. At the start of every iteration, $S$, the left-shifted schedule that corresponds to the current dependency graph $G$, is computed (line 3). Together with $S$, the earliest possible starts for every job admitted by $G$ (variable $start$), and thereby the start times of every job in the left-shifted schedule $S$, are computed. Also computed is $latestFinish$, the latest possible finishing time for every job, i.e., for every job the latest point in time during which it can still execute without any job missing its

[1] If there are multiple intervals with maximal power demand, one may choose one arbitrarily. However, since the power demands are real numbers, this is highly unlikely.

deadline. From these values, the algorithm then determines the peak range in the form of *peakBegin* and *peakEnd*, and the jobs executing during peak demand (lines 4–5).

**Algorithm 1:** The SWAG Algorithm

**Input:** $G$ : Dependency Graph

```
1  originalG ← G;
2  while time limit not reached do
3      S, start, latestFinish ← leftShiftedSchedule(G);
4      peakBegin, peakEnd ← determinePeakRange(S);
5      peakJobs ← {j ∈ J | s_j < peakEnd ∧ s_j + p_j > peakBegin}
6      candidateEdges ← {(u, v) |
7              u, v ∈ peakJobs ∧
8              start[u] + p_u > start[v] ∧
9              start[u] + p_u + p_v ≤ latestFinish[v]};
10     if |candidateEdges| = 0 then
           // The algorithm is blocked
11         newCandidateEdge ← unblockByDeletion(peakJobs);
12         if (deletionsSinceLastReset > deletionsBeforeReset )
13             or (newCandidateEdge = Null) then
14             G ← originalG;                                  // Reset
15     else
16         e ← randomSelection(candidateEdges);
17         G.insert(e);
```

At the core of every iteration, the algorithm now needs to determine which edge to insert to extend a feasible schedule. We call the possible edges to be inserted *edge candidates*. Inserting an edge candidate $(j_u, j_v)$ is useful for reducing the current peak demand if $j_u$ and $j_v$ execute concurrently within the current peak range (see lines 7, 8). Inserting such an edge candidate would separate two jobs that currently contribute to the peak demand. Among these useful edge candidates, an edge candidate $(j_u, j_v)$ is *feasible* if inserting $(j_u, j_v)$ into $G$ would make the left-shifted schedule of $G$ a feasible schedule, which is the case exactly if the duration of $j_u$ plus the duration of $j_v$ is not greater than the time between the latest possible finishing time for $j_v$ and the earliest possible start time for $j_u$ (line 9). We only ever insert feasible edge candidates, thereby making sure that the left-shifted schedule of $G$ always stays feasible.

If at least one feasible edge candidate exists, the algorithm picks one at random, inserts it and starts the next iteration. Note that we explored various strategies of weighting this random selection, however empirically, picking one of the feasible edge candidates uniformly gave the best results. If no feasible edge candidate exists, we say the algorithm is *blocked*. This happens because edges inserted earlier cause

every useful edge candidate to become infeasible. Therefore, at least some of the edges inserted earlier must be removed again. For this, there are two possibilities: The algorithm can either reset $G$ back to $\mathcal{G}$ (i.e., delete all inserted edges, lines 13–14), or selectively find a small set of edges the deletion of which makes at least one useful edge candidate feasible (line 11).

**Algorithm 2:** Pseudocode for the UnblockByDeletion function.

**Input:** *peakJobs* : Set of jobs participating in peak demand
**Output:** $(s, t)$ : Edge to delete

```
1  Function unblockByDeletion(peakJobs)
2      for i ∈ {1, ..., deletionTrials} do
3          s, t ←randomSelection(peakJobs);
4          overlap ← duration[s] + duration[t] − (latestFinish[t] − start[s]);
5          delForwardSet, deleteForwardMovement ←
6              findDeletionEdgesForward(t, overlap);
7          delBackwardSet, deleteBackwardMovement ←
8              findDeletionEdgesBackward(s, overlap);
9          if deleteForwardMovement + deleteBackwardMovement ≥ overlap
             then
10             G.delete(delForwardSet, delBackwardSet);
11             return (s, t);
12     return Null;
```

### 8.3.2 Selecting Edges for Deletion

When the algorithm is blocked, it has run into a local minimum and must perturb the current solution to climb out of the minimum. Before just restarting the algorithm, the heuristic tries to find a small set of edges to delete to unblock the current situation. Note that we stated earlier that we assume the initial graph $\mathcal{G}$ to be empty to simplify the description of the algorithm. In fact, this step is the only step where one must pay attention not to delete edges of $\mathcal{G}$ when finding edges to be deleted.

SWAG does not just delete edges at random, but tries to find edges the deletion of which likely unblocks the algorithm. The procedure to find such edges is outlined as Algorithm 2.

The search for edges to be deleted works by iterating over the (infeasible) edge candidates, i.e., edges that we would like to insert, but for which can not do so without missing a deadline. The algorithm iteratively tries to make edge candidates between two jobs in the *peakJobs* set feasible. The process to make an edge candidate $(j_a, j_b)$ feasible is a two-step process: First, it is determined by how much deadlines would

be missed if $(j_a, j_b)$ would be inserted into $G$. This is the *overlap* computed in line 4. Note that since $s$ and $t$ are part of the *peakJobs* set and we did not find a feasible edge candidate, the duration of both jobs must be greater than the time between the latest possible finishing time for $t$ and the earliest possible start time for $s$ — otherwise, $(j_s, j_t)$ would be a feasible edge candidate. Thus *overlap* is always positive.

When selecting a set of edges to be deleted, the algorithm must ensure that deleting the edges allows to move $j_a$ enough to the left and $j_b$ enough to the right, such that the sum of both movements is at least *overlap*. If we delete such a set of edges, the edge candidate $(j_a, j_b)$ becomes feasible and we can insert $(j_a, j_b)$, thereby separating two jobs in the *peakJobs* set and completing an iteration of SWAG.

In line 6, the search for a set edges that can be deleted s.t. $j_a$ can be moved to the left for up to *overlap* steps starts. The analog search is done for $j_b$ in line 8. If this finds two edge sets such that the total possible movement is at least *overlap*, the edges are deleted and there is a new feasible useful edge candidate $(j_a, j_b)$ (see lines 9–11). If not, the algorithm tries to make a different edge candidate feasible, for up to deletionTrials trials.

The searches in lines 6 and 8 are equivalent, so we only describe *findDeletionEdges-Forward*, outlined in Algorithm 3. The search is a depth-limited breadth-first search on the edges of the graph. The search progresses iteratively, and at every time holds a queue of edges that are to be deleted. We start with all outgoing edges of $j_b$. In every step, we remove the first edge from the queue, say that edge is $(j_x, j_y)$. We then insert all outgoing edges of $j_y$ into the queue, and therefore the set of edges to be deleted. This way, the set of edges to be deleted progressively grows in size and distance from $j_b$. After every such replacement, we evaluate the quality of the current set of edges to be deleted. The quality decreases the more edges the set contains, and if the resulting possible movement of $j_b$ is less than *overlap*.

### 8.3.3 Optimizations

The heuristic as described so far is functional, but can benefit from optimizations in various places. We now describe three optimizations we implemented and evaluated. For an insight into the effects these optimizations have, see Section 8.5.3.

**Deferred Propagation.** In Algorithm 1, at the beginning of each iteration (in lines 3 and 4), up-to-date values for the starts and latest finishes of all jobs are needed, and from this the peak demand is computed. See below for how to efficiently compute the peak demand. Here, we explain how to efficiently retrieve starts and latest finishes. Instead of recomputing them at the start of every iteration, it is possible to maintain the current start and latest finish value for all jobs and update them as needed. Whenever an edge $(j_a, j_b)$ is inserted into $G$, such an update becomes necessary.

**Algorithm 3:** Pseudocode for the FindDeletionEdgesForward function.

**Input:** *startJob* : Job from which on to search for edges to be deleted
**Input:** *overlap* : Amount by which *startJob* should be moved to the left
**Output:** *bestSolution* : Best found set of edges to delete
**Output:** *bestMovement* : How much movement of *startJob* the deletion of the edge set enables

```
1  Function findDeletionEdgesForward(startJob, overlap)
       /* Entries are pairs of an edge and a depth. The initial
          entry ((⊥, startJob), 0) is not a real edge, but is needed
          to start the loop below with startJob.                    */
2      edgesToDelete ← makeList(((⊥, startJob), 0));
3      bestQuality ← ∞;
4      while edgesToDelete.notEmpty() do
5          e, depth ← edgesToDelete.popFront();
6          (v, w) ← e;
7          if depth ≤ deletionMaxDepth then
               /* One step in the edge BFS: Replace (v, w) with
                  outgoing edges of w.                              */
8              for f ∈ w.outgoingEdges do
9                  edgesToDelete.pushBack((f, depth + 1));
           /* Pretend to remove edges to determine new latest
              finishes                                              */
10         G.removeEdges(edgesToDelete);
11         newLF ←computeLatestFinish (startJob);
12         G.insertEdges(edgesToDelete);
13         movement ← newLF − latestFinish[startJob];
14         quality ← |edgesToDelete| +
               (undermovePenalty · max(0, overlap − movement));
15         if quality < bestQuality then
16             bestSolution ← markedEdges;
17             bestQuality ← quality;
18     return bestSolution, bestMovement;
```

The update is done by propagating new starts throughout $G$, starting in $j_b$, and propagating new latest finishes throughout the reverse graph of $G$ starting in $j_a$. However, especially for large, dense graphs, doing this propagation after every inserted edge is expensive.

Assuming that the current peak range does not change after an edge has been inserted (i.e., we need to insert more edges to remove the current peak), the next selected edge will be between two jobs overlapping the current peak range. Thus, in this case it is sufficient to only propagate starts and latest finishes to jobs overlapping the current peak range. Since these jobs are close to $j_a$ and $j_b$ in $G$, this is an inexpensive operation. We say we *defer* the full propagation. The algorithm records at which jobs the propagation stopped, and continues the propagation as necessary.

Deferred propagation might cause the computation of the peak range at the beginning of the an iteration to be incorrect — the peak could have moved to some other place in the schedule, which is not detected because changes in starts have not been propagated to the jobs that are involved in the new peak. To mitigate this, the algorithm must do a full propagation every couple of iterations, which is determined via the parameter completePropagationAfter.

**Determining the peak demand.** In Algorithm 1, at the beginning of each iteration (in line 4), the peak value and peak range is determined from the job starts. The trivial way of doing this would involve sorting all jobs by their start times, and then iterating this list, keeping track of how much demand is active at which point. Doing this would require $O(n \log n)$ time for the sorting step, which is too expensive.

Instead of recomputing peak demand and range each time it is required, we use a dynamic segment tree as described by van Kreveld and Overmars [KO93] to efficiently maintain these values. For each job, we insert a segment with the length of the job's duration into the segment tree, with the start point corresponding to the job's start time. Whenever we update a job's start time (see above), we also update the segment in the dynamic segment tree, which can be done in $O(\log n)$ time. In [KO93], segments in the segment tree are associated with some kind of segment ID, with the effect that one can query which segments are active at a certain point. SWAG does not need this functionality, but needs to efficiently determine cumulative demands at certain points. Thus, we instead associate every segment in the tree with the power demand of the corresponding job. This way, the dynamic segment tree allows us to retrieve the cumulative power demand at a specific point in time in $O(\log n)$ time. With some additional annotations of the tree's vertices, we can even retrieve the peak range and the peak value in $O(1)$ time.

**Batched Edge-Candidate Generation.** In Algorithm 1, line 6, we determine all feasible edge candidates between two jobs executing during the peak range. For large

instances, this set can become very large, thus expensive to compute. We therefore apply two optimizations: First, we only recompute the set of feasible edge candidates when necessary, i.e., when the peak range changes. Second, we do not compute the whole set at once. Usually, it will be sufficient to insert few edges to remove the current peak and shift the peak range somewhere else. Therefore, we first only generate a small batch of feasible edge candidates, generating more on demand when the generated candidates are depleted and the peak range has not shifted.

## 8.4 Competitor Algorithm: GRASP

We implemented the scheduling heuristic by Petersen et al. [Pet+14] to compare SWAG to. The algorithm described in that work is a combination of a metaheuristic called *greedy randomized adaptive search procedure* (GRASP) and a simple local search in the form of a hill climber. For simplicity reasons, we refer to the combination of both algorithms as GRASP in this chapter.

The authors present and evaluate multiple variants of their algorithm. We implemented and evaluated all variants, most notably the *sorted* and *random* variants of the GRASP step, as well as the *uniform* and the *weighted* variants of the hill climber. An in-depth discussion on how we chose which parameters and why is given in Section 8.5.2.

The GRASP algorithm cannot originally cope with release times and dependencies. However, both constraints are straightforward to add. GRASP works by iteratively trying to place jobs at a different time within the time window the respective job is allowed to run in. To incorporate dependencies and release times, one must only make sure to correctly constrain this window by the release time and possible predecessor or successor jobs. Also, in their work, Petersen et al. use GRASP to optimize for a slightly different objective. However, since GRASP is a metaheuristic, the objective can be switched without any changes to the actual algorithm.

We performed tuning on the GRASP algorithm as described in Section 8.5.2, and found several surprising insights. First, we consistently got better results when using the *random* variant instead of the *sorted* variant, which stands in contrast to what the original GRASP authors found (see Table IV in [Pet+14]). This could be explained by the fact that our instances are larger than the test instances in [Pet+14], thus sorting consumes more time in our case. For the random GRASP variant, the *uniform* hill climber consistently outperformed the *weighted* variant in our tests, while the results in [Pet+14] seem to favor the weighted variant. The optimal parameters determined by our tuning are shown in Table 8.1. Consistent with the findings in [Pet+14] is that the smallest possible values are chosen for $m$ and $l$, while rather high values are chosen

**Table 8.1:** Chosen parameters for GRASP. Corresponds to Table IV in [Pet+14].

| Parameter | Value |
|---|---|
| m | 1 |
| l | 1 |
| n | 200 |
| Hill Climber Iterations | 10 |
| Hill Climber Type | uniform |
| GRASP Type | random |

for the number of hill climber iterations.[2] This means that effectively, the hill climber does the main part of the algorithm's work.

To make the comparison between SWAG and GRASP as fair as possible, we implemented all parts that need access to (peak) demands once as a simple array-based approach, of which we assume that the authors of [Pet+14] use it, and once using the dynamic segment tree that we also use for our SWAG implementation. During our tuning, the dynamic-segment-tree based approach consistently outperformed the array-based variant, thus we used it for the evaluation.

## 8.5 Evaluation

We performed three kinds of evaluation of the SWAG algorithm: First, we compared the solutions computed by SWAG to near-optimal solutions computed by a mixed-integer program (MIP) on small instances, on which MIPs are still a feasible solution technique. Then, we investigated the influence that various SWAG parameters and properties of the instances have on the computed solution quality. Finally, we compared SWAG to the GRASP algorithm by Petersen et al. [Pet+14].

All experiments have been executed on machines with Intel® Xeon® E5-2670 CPUs with 16 cores and 64 GBs of RAM. The MIPs were solved by Gurobi 7.0.2, running one solver with 16 threads for 30 minutes. For the SWAG and GRASP heuristics, we always ran 15 experiments in parallel and used a time limit of 5 seconds for all experiments.

**Code and Data Publication.** All our code, including the implementations of SWAG, GRASP and the mixed-integer linear program, as well as all test instances are publicly available as a separate data publication [BW19a]. While that publication contains a snapshot of the code used for this publication, a more recent version of the optimization software can be found at

[2] In [Pet+14], the authors use a time limit instead of an iteration limit for the hill climber. We used an iteration limit for finer control.

https://github.com/kit-algo/TCPSPSuite/.

### 8.5.1 Instance Sets

We used a total of three sets of instances to evaluate SWAG: A set of small instances, called $I_{\text{small}}$, to compare SWAG to the MIP, since the MIP is not able to cope with larger instances. Second, a set of large instances, called $I_{\text{large}}$ to compare SWAG to GRASP. Optimization complexity for an Single-Resource Acquirement Cost Problem instance is not only driven by the number of jobs, but also by the jobs' window sizes, since the size of the solution space increases with more possibilities to place a job. Therefore, we generated a third set of instances with larger window sizes, called $I_{\text{window}}$.

All sets were generated based on the HIPE data set by Bischof et al. [Bis+18], which was obtained from a small-scale electronics factory at the Institute for Data Processing and Electronics at the Karlsruhe Institute of Technology. A detailed description of that data set can be found in [Bis+18]. From this factory data, we got power demand time-series from six machines, with sub-minute resolution. The machines are a chip press, a screen printer, a vacuum oven, a high temperature oven, a soldering machine and a chip washing machine.

In a first step, we detected patterns (which we call *processes*) in this data using a technique adapted from Ludwig et al. [Lud+17]. The technique is very similar to the technique described by Ludwig et al. [Lud+19b] for their benchmark data set generation. This pattern recognition subdivides every time series into *sequences*. Each sequence is a consecutive series of points in the time series and belongs to exactly one process. The sequences belonging to a process are the *occurrences* of a process. In total, we detected 16 different processes in the input data.

From these detected patterns, a representation is built for each process based on probability distributions. Each occurrence of a process is characterized by three parameters: The duration of the occurrence, the energy consumed during the occurrence and the time of day that the occurrence started at. Thus, the set of occurrences of a process results in a set of three-dimensional points.

Initial experiments revealed that for the duration and the consumed energy, a normal distribution is a good fit. Therefore, for each process, a bivariate Gaussian Mixture Model (GMM) was fit to the duration and energy components of the process' point set. We used a mixture model since we assume that the same process can be run in different modes, which would be captured by the mixture model having more than one component. The start time is not represented well by a normal distribution. We still assume that there might be several points of time in a day around which a process is usually started. Therefore, the start times of a process were first clustered using the

DBSCAN[3] algorithm [Est+96]. Then, the 0.1 and 0.9 quantile of every cluster were determined and taken as the lower respective upper limit of a uniform distribution. This results in one uniform distribution per cluster. The uniform distributions were weighted by the number of points in the respective cluster.

To generate a job as defined in Section 8.2.1, release time, deadline, duration and power demand are necessary. First, select one of the detected processes by a weighted selection, with weights being the number of occurrences of each process. Then, draw one sample of duration and energy from the corresponding GMM. The duration becomes the duration of the new job, the power demand is determined by the drawn energy divided by the duration. To draw a start time, first select one of the uniform start-time distributions of the selected process by their weight, then draw from that distribution. Finally, the deadline must be determined, which is equivalent to determining the window size. We assume that there is a certain flexibility immanent to the process we have created the job from. We assume that this flexibility is correlated with the difference between the maximum and minimum of the uniform distribution that we drew the start time from, therefore this difference becomes the first component of the window size. Additionally, we suppose that there is additional flexibility, which can not be seen from the data at hand, since in the past, no effort has been made to shift the processes in time. Therefore, draw a second component of the window from a normal distribution the parameters of which must be set. We call this amount the *window growth*, and the final window size is the sum of the span of the start-time normal distribution and the window growth.

With this approach, we generated the following three sets:

**Large Instance Set.** For the large instance set $I_{\text{large}}$, we did everything as described above, and generated between 500 and 1500 jobs per instance, for a total of 300 jobs. For the window growth distribution, we used a mean value of 100 minutes and a standard deviation of 20 minutes. All instances have a time horizon of five days, in one minute resolution.

**Small Instance Set.** For the small instance set $I_{\text{small}}$, we did everything as for $I_{\text{large}}$, only that we limited the number of jobs to between 50 and 150, and set the time horizon to 3 days. We generated a total of 200 instances.

**Window Instance Set.** The window instance set $I_{\text{window}}$ is used to evaluate the effect that larger window sizes have on the performance of SWAG. To this end, we generated an instance set equal to $I_{\text{large}}$, with the only exception that we used a mean value of 500 minutes for the window growth parameter.

[3] With $\epsilon$ such that occurrences starting 30 minutes from each other are considered to be close, and a minimum number of 5 points per dense region.

### 8.5.2 Parameter Tuning

Both the SWAG and the GRASP algorithm need to be parameterized. To have a fair comparison, we determined both parameter sets using the same technique, which we outline in this section. A systematic technique is necessary since both algorithms have a parameter space which is far too large to select satisfying parameters by just eyeballing results. We describe the technique we use in an abstract way, uncoupled from the actual algorithms we tune.

Let $P = \{\rho_1, \rho_2, \dots \rho_k\}, \rho_i \in \mathbb{R}$ be a set of (numeric) parameters for some algorithm. We start by computing a grid search on $P$. For every $\rho_i$, we select a set of $l_i$ possible values $\Gamma_i = \{\gamma_{i,1}, \gamma_{i,2}, \dots \gamma_{i,l_i}\}$. The $\Gamma_i$ are chosen somewhat arbitrary — however, we try mostly regularly spaced values in a range we consider reasonable, and add some more extreme values on both ends of the range to check whether our reasoning was wrong. The idea is that if the optimal value for a $\rho_i$ falls outside of the range of values in $\Gamma_i$, one of the two extreme values at its ends should be chosen. In this case, we repeat the tuning with an adjusted $\Gamma_i$. All the $\Gamma_i$ form a set of possible configurations $C = \Gamma_1 \times \Gamma_2 \times \Gamma_3 \dots \Gamma_k$. We run the algorithm to be tuned with every configuration in $C$ on each of the instances in the respective instance set.

From the set of results of this grid search, we must now select a good configuration. A good configuration is one that not only produces good results, but which is also similar to other configurations that produce good results. If a configuration produces good solutions, but all similar configurations don't, then it is highly likely that either this is a measurement error, e.g, because of random noise, or that this configuration over-fits the instance set. Therefore, we score configurations in a two-step process. First, we compute for every configuration a preliminary score solely based on the performance of that configuration, and then adjust this score by the scores of similar configurations to create the final score. We start by normalizing all solution qualities to the best solution quality computed on the respective instance. Then, for every configuration, we add up the normalized qualities for all instances. Thus, a configuration giving the best solution for *all* instances, in a set of $k$ instances, would get a preliminary score of $k$. A configuration that consistently always is worse than the optimum by a factor of 2 would get a preliminary score of $2k$.

To define the distance of two configurations, we use the $L_1$ distance, also knows as Manhattan distance. Let $P$ and $Q$ be two configurations, then $\text{dist}(P, Q) = \sum_i |P_i - Q_i|$. The influence of a neighboring solution decreases quadratically with distance, and we set the total weight of the neighboring solutions in a configuration's final score to 0.3, thus

$$\text{final}(P) = 0.7 \cdot \text{preliminary}(P) + 0.3 \sum_{Q \in C, Q \neq P} \frac{\text{preliminary}(Q)}{\text{dist}(P, Q)^2} N$$

with $N$ being a normalization factor of $N = \sum_{Q \in C, Q \neq P} \text{dist}(P, Q)^2$. We finally select the configuration with the smallest final score.

**Table 8.2:** Chosen parameters for SWAG. We use separate parameter sets for the set of small instances and the sets of large instances.

| | Value for set(s) | |
|---|---|---|
| Parameter | $I_{\text{small}}$ | $I_{\text{large}}$, $I_{\text{window}}$ |
| deletionTrials | 300 | 400 |
| completePropagationAfter | 0 | 0 |
| deletionsBeforeReset | 150 | 300 |
| deletionMaxDepth | 0 | 1 |
| batchsize | 7 | 6 |
| undermovePenalty | 5 | 10 |

The results of tuning SWAG can be found in Table 8.2. Note that we use separate sets of parameters for small and large instances. The tuned parameters of GRASP can be found in Table 8.1. Since we use GRASP only on the large instances, there is only one set of parameters here.

### 8.5.3 SWAG analysis

This section presents a first analysis of SWAG, starting by comparing results for the $I_{\text{small}}$ instance set computed by SWAG to results computed using a mixed-integer linear program (MIP) adapted from Chapter 3. The MIP was optimized for 1800 seconds using 16 parallel threads, while SWAG was executed for 5 seconds in a single thread. The MIP found optimal solutions for 97 of the 200 instances, and closed the MIP gap to within 5% for 178 instances. While these numbers look good, increasing the amount of time spent on MIP optimization does not allow us to optimize significantly larger instances using the MIP, since the major limiting factor for optimizing larger instances using the MIP is memory consumption.

Figure 8.2 shows a dot for every instances in $I_{\text{small}}$, where the $x$ coordinate specifies the number of jobs in the instance, and the $y$ coordinate specifies the peak value of the SWAG solution divided by the peak value of the MIP's solution.

We see that on $I_{\text{small}}$, SWAG computed the optimal solution within 5 seconds on many instances, especially on the smaller ones. In total, 99 instances were solved to optimality by SWAG. Of the 100 instances with less that 100 jobs, 63 were solved to optimality. As the instances grow in size, more instances could not be solved to optimality anymore by SWAG, with the factor between the MIP solution and the SWAG solution reaching 24% in the worst case.

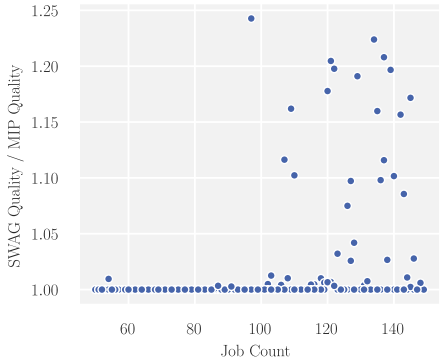



**Figure 8.2:** Quality computed by SWAG compared to MIP solution on $I_{\text{small}}$, ordered by number of jobs in the instance.

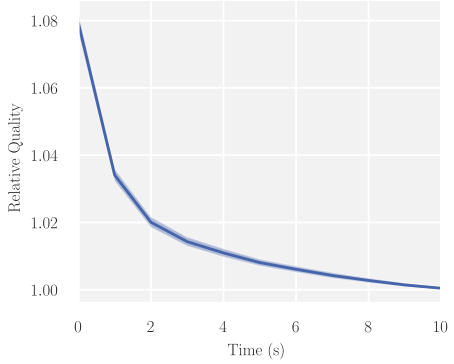



**Figure 8.3:** Convergence speed of SWAG. The $y$ axis indicates the quality of the best found solution at a certain point in time relative to the best solution found after 10 seconds. The blue line is the median over all instances, the shaded area indicates the 0.1 / 0.9 quantiles.

### Parameter Impact

We now investigate how the change of various parameters impacts the performance of SWAG. This analysis is done on the $I_{\text{large}}$ instance set. Note that the time resolution of the instance does not affect the SWAG performance, as SWAG is purely based on dependencies, this is why we do not evaluate it. Many other approaches, such as the (discrete-time) MIP approach, is influenced by time resolution.

Looking at the — according to the tuning — optimal choice for the parameter completePropagationAfter (see Table 8.2 and Section 8.3.3), deferred propagation is turned off in the optimal parameter set. Thus, we conclude that the disadvantage of not having the correct peak range at every point in time outweighs the advantage of not having to update all starts and latest finishes every time.

Similarly, the choice for deletionMaxDepth (see Algorithm 3) limits the depth of the edge breadth-first-search used to search for edges to be deleted to 1 on the large instances, which makes the BFS very shallow. On small instances, it is even set to 0. It stands to reason that always setting the parameter to 0, which would result in simply always picking the incoming (resp. outgoing) edges of $s$ (resp. $t$) in Algorithm 2 would not impair quality too much.

To summarize these two findings, we can conclude that often, simplicity and not doing too much work seems to be a decisive advantage.

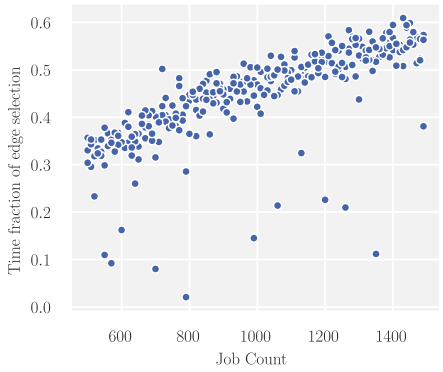



**Figure 8.4:** Fraction of run time spent on finding feasible edge candidates


**Figure 8.5:** Histogram of the relative score of GRASP compared to SWAG on $I_{\text{large}}$.

### Convergence Speed

We look into how fast the solutions found by SWAG converge. We performed this analysis on the $I_{\text{large}}$ instance set, using a maximum run time of 10 seconds. Every 0.1 seconds, we sampled the currently found best solution. After the run was completed, we normalized the solution qualities of all samples taken during that run by the final solution quality of the respective run, i.e., a value of 1.1 would indicate a solution that is 10% worse than the solution found after 10 seconds.

Figure 8.3 shows the results. The blue line indicates the median value over all instances, while the shaded bands indicate the 0.9 and 0.1 quantiles, i.e., only 10% of the runs fall above or below the blue band. We can see that already with the first samples after 0.1 seconds, we were usually reasonably close to the final solution, to within about 8% in the median. After 2 seconds, we can expect to be within 2% of the final solution, after 4 seconds within 1%. Thus, the choice of using 5 seconds as run time for all other experiments seems justified.

**Complexity of Algorithm Parts.** We now present a deeper insight into where the SWAG algorithm spends the majority of its time. Figure 8.4 shows the fraction of the total run time that SWAG spent on finding feasible edge candidates by instance size, roughly corresponding to lines 6 to 9 in Algorithm 1. We see that this is the most expensive step, needing up to 60% of the total run time. We also see that the necessary time increased with instance size. This is not surprising, since the size of the *peakJobs* set generally increases with the number of jobs, and the number of edges that must be checked for feasibility is quadratic in the size of this set.

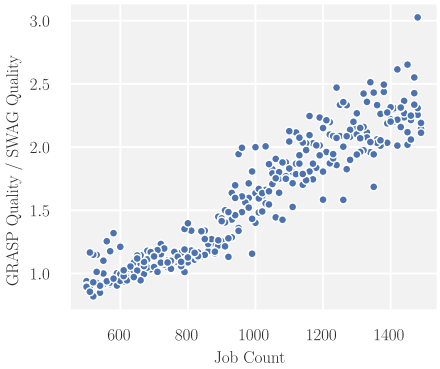



**Figure 8.6:** Comparison between SWAG and GRASP by job count, on $I_{\text{large}}$. Every dot is one instance. The y coordinate corresponds to the solution computed by GRASP divided by the solution computed by SWAG.

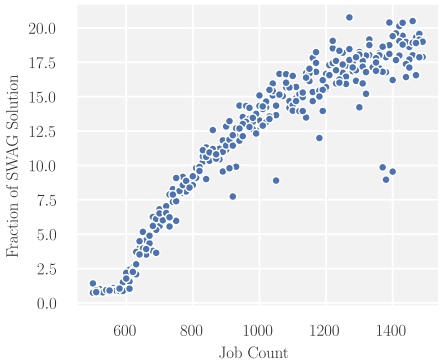



**Figure 8.7:** Comparison between SWAG and GRASP by job count, on $I_{\text{window}}$. Every dot is one instance. The y coordinate corresponds to the solution computed by GRASP divided by the solution computed by SWAG.

### 8.5.4 Comparison SWAG vs. GRASP

We now present how SWAG compares to GRASP on the instance set $I_{\text{large}}$ with parameters chosen as shown in tables 8.1 and 8.2. Figure 8.5 shows a histogram of the solution qualities computed by GRASP relative to the respective solution computed by SWAG. The value on the $x$ axis states by which factor the peak demand computed by GRASP is worse than the peak demand computed by SWAG, the $y$ axis just counts the instances. We see that there are some instances in which GRASP performed slightly better than SWAG, in the best case by about 18%. For 268 of 300 instances, SWAG outperformed GRASP, by a factor of up to 3. Figure 8.6 shows the results by number of jobs in the instance. We can see that with increasing instance size, the advantage of SWAG over GRASP grows rapidly. SWAG computed better results than GRASP on all instances with more than 670 jobs.

Besides instance size, the complexity of Single-Resource Acquirement Cost Problem is mainly driven by the window sizes of the jobs, as larger windows increase the solution space. We examined the behavior of SWAG on instances with larger window sizes on the instance set $I_{\text{window}}$, see Section 8.5.1 on how we enlarged the jobs' windows. The performance of SWAG compared to GRASP is depicted in Figure 8.7. Here, we see that starting at an instance size of around 600, the advantage of SWAG over GRASP grows drastically, up to a factor of almost 20 for the largest instances. Comparing Figure 8.6 and Figure 8.7, SWAG's better scalability in terms of instance size seems to be reinforced by instances with larger window sizes.

## 8.6 Conclusion

In this chapter we have presented SWAG, a heuristic to schedule large amounts of time-flexible loads in a smart grid. SWAG uses a graph-based representation, which makes it independent of the time resolution and allows to incorporate process dependencies. We evaluated SWAG on benchmark instances derived from real-world energy time series, showing SWAG to be very efficient in this use case. Our evaluation also shows that SWAG outperforms GRASP on this energy-related data set, especially on larger instances and instances with large window sizes. On small instances, SWAG could demonstrate its effectiveness by solving a large portion of the instances to optimality. We could also demonstrate that the solutions found by SWAG converge within few seconds.

In the future, it would be interesting to apply SWAG to more general scheduling scenarios, such as settings in which processes' power demand changes over time. Also, we have seen that the two main factors determining the complexity of an Single-Resource Acquirement Cost Problem instance are the instance size and the jobs' window sizes. Therefore, further research into what realistic scheduling scenarios in a smart grid would look like — especially in terms of window sizes — is necessary. Also, we determined that SWAG spends large parts of the work in determining feasible edge candidates. Finding a more efficient way of doing this would further increase SWAG's efficiency.

In conclusion, we believe that SWAG can be used as a building block of a future energy system, helping to schedule loads and shaving peaks.

# Part III

# Algorithmic Foundations

# 9 Efficiently Finding Peaks Using Dynamic Segment Trees

In this chapter, we introduce *zipping segment trees*, a special form of dynamic segment trees, which can be used to efficiently determine the location and magnitude of demand peaks in schedules. We utilize the recently introduced zip tree data structure. This task is closely related to that of performing a *stabbing query*.

Stabbing queries in sets of intervals are usually answered using segment trees. A dynamic variant of segment trees has been presented by van Kreveld and Overmars [KO93], which uses red-black trees to do rebalancing operations. This chapter presents *zipping segment trees* — dynamic segment trees based on zip trees, which were recently introduced by Tarjan et al. [TLT19]. To facilitate zipping segment trees, we show how to uphold certain segment tree properties during the operations of a zip tree. We present an in-depth experimental evaluation and comparison of dynamic segment trees based on red-black trees, weight-balanced trees and several variants of the novel zipping segment trees. Our results indicate that zipping segment trees perform better than rotation-based alternatives.

This chapter is based on joint work with Dorothea Wagner [BW20b].

## 9.1 Introduction

A common task in computational geometry, but also many other fields of application, is the storage and efficient retrieval of segments (or more abstract: intervals). In this thesis, we require such a data structure for the heuristics presented in chapters 7 and 8, where we use it to localize the demand peak in a given schedule. The question of which data structure to use is usually guided by the nature of the retrieval operations, and whether the data structure must by dynamic, i.e., support updates.

For scheduling algorithms, the concrete task is to determine all jobs that run at a specified time $t$, which amounts to a classic so-called *stabbing query* in an interval storing data structure. Such a stabbing query can be formulated as follows: Given a set of intervals on $\mathbb{R}$ and a query point $x \in \mathbb{R}$, report all intervals that contain $x$.

stabbing query

For the static case, a *segment tree* is the data structure of choice for this task. It supports stabbing queries in $O(\log n)$ time (with $n$ being the number of intervals). Segment trees were originally introduced by Bentley [Ben77]. While the classic segment tree is a static data structure, i.e., is built once and would have to be rebuilt from scratch to change the contained intervals, van Kreveld and Overmars present a dynamic version [KO93], the so-called *dynamic segment tree* (DST).

segment tree

dynamic segment tree

Dynamic segment trees are applied in many fields. Solving problems from computational geometry certainly is the most frequent application, for example for route planning based on geometry [EJW05] or labeling rotating maps [GNR16]. However, DSTs are also useful in other fields, for example internet routing [CL07].

To apply the dynamic segment tree to scheduling problems, we treat a job in a schedule as an interval between its starting and finishing point, and associate the intervals with a weight, i.e., the (in the simplest case single-resource) demand of the corresponding job. We store these intervals in a dynamic segment tree that we extend with an additional annotation at every node. This annotation allows us to determine the peak demand as well as the time interval during which the peak demand occurs in $O(1)$.

zipping segment trees

In this chapter, we present an adaption of dynamic segment trees, so-called *zipping segment trees*. Our main contribution is replacing the usual red-black-tree base of dynamic segment trees with zip trees, a novel form of balancing binary search trees introduced recently by Tarjan et al. [TLT19]. On a conceptual level, basing dynamic segment trees on zip trees yields an elegant and simple variant of dynamic segment trees. Only few additions to the zip tree's two rebalancing methods are necessary. On a practical level, we can show that zipping segment trees outperform dynamic segment trees based on red-black or weigh-balanced trees in our experimental setting.

We start this chapter by sketching some necessary concepts in Section 9.2. We then outline the operations of dynamic segment trees in Section 9.3. In Section 9.4 we introduce zipping segment trees, our main contribution. We experimentally compare traditional dynamic segment trees to our new zipping segment trees in Section 9.5.

## 9.2 Preliminaries

In this chapter we discuss data structures built upon balanced binary search trees. In this section we introduce the terminology necessary to formally specify these data structures.

binary tree
root
parent
children
leaf

A (rooted) *binary tree* is a connected graph consisting of nodes and edges, where one node is chosen as *root*. Every node except the root has one incoming edge, connecting the node to its *parent*. Every node has up to two outgoing edges, connecting it to up to two *children*. Nodes without any outgoing edges are called *leaves*. A tree does not contain any cycles, thus for two nodes $a$, $b$, the path between $a$ and $b$ exists (because of connectedness) and is unique. We call the length of the unique path from the root to a node $a$ the *depth* of $a$. For two nodes $a$, $b$, we define the *lowest common ancestor* of $a$ and $b$, written as $\mathrm{LCA}(a, b)$, to be the node on the path from $a$ to $b$ with the smallest depth.

depth
lowest common ancestor
ordered tree

A tree is *ordered* if we label one of its outgoing edges to be *left* and the other one to be *right*. For ordered trees, we use the notation $L(v)$ to refer to the left child of $v$

and $R(v)$ to refer to the right child of $v$. If one of the children is missing, we write that $L(v) = \perp$ resp. $R(v) = \perp$. We sometimes also need to explicitly reference the outgoing edges instead of the children, especially when edges are annotated. We denote the left outgoing edge of $v$ as $\vec{L}(v)$ and the right outgoing edge of $v$ as $\vec{R}(v)$. An ordered tree induces a *tree order* on the nodes. In this order, we say a node $a$ comes before node $b$ if and only if the path from $\text{LCA}(a, b)$ to $a$ starts with $(\text{LCA}(a, b), L(\text{LCA}(a, b)), \dots)$ *or* the path from $\text{LCA}(a, b)$ to $b$ starts with $(\text{LCA}(a, b), R(\text{LCA}(a, b)), \dots)$.

tree order

If the nodes of an ordered binary tree are associated with ordered values, for example values from $\mathbb{R}$, the tree can form a *search tree*. We call the associated value of a node $v$ its *key*. An ordered binary tree is a search tree if it has the property that the left child (if present) of any node $v$ always has a smaller-or-equal key as $v$, and the right child always has a key larger than that of $v$. This is equivalent to the order of the nodes in terms of keys being the same as the tree order of the nodes. In such a search tree, every possible key value, even values that are not associated with any node in the tree, has a *search path*. This path is constructed by starting with the root, and then repeatedly descending to the left if the search key is smaller-or-equal than the current node's key, or right otherwise.

search tree

key

search path

A common operation used in rebalancing search trees without changing the node order is a node *rotation*. Figure 9.2 displays a counter-clockwise rotation around node $a$. After the rotation, the former right child of $a$, node $c$, is $a$'s new parent, and the former left child of $c$ (if any) is the new right child of $a$.

rotation

A concept we need for zip trees are the two *spines* of a (sub-) tree. We also talk about the spines of a node, by which we mean the spines of the tree rooted in the respective node. The *left spine* of a subtree is the path from the tree's root to the previous (compared to the root, in tree order) node. Note that if the root (call it $v$) does is not the overall smallest node, the left spine exits the root left, and then always follows the right child, i.e., it looks like $(v, L(v), R(L(v)), R(R(L(v))), \dots)$. Conversely, the *right spine* is the path from the root node to the next node compared to the root node. Note that this definition differs from the definition of a spine by Tarjan et al. [TLT19].

spine

### 9.2.1 Union-Copy Data Structure

Dynamic segment trees in general carry annotations of sets of intervals at their vertices or edges. These set annotations must be stored and updated somehow. To achieve the run time guarantees in [KO93], van Kreveld and Overmars introduce the *union-copy* data structure to manage such sets.

union-copy

Sketching this data structure would be out of scope for this chapter. It is constructed by intricately nesting two different types of union-find data structures: a textbook union-find data structure using union-by-rank and path compression (see for example Seidel and Sharir [SS05]) and the $UF(i)$ data structure by La Poutré [La 90].

For this chapter, we just assume this union-copy data structure to manage sets of items. It offers the following operations[1]:

**createSet()** Creates a new empty set in $O(1)$.

**deleteSet()** Deletes a set in $O(1 + k \cdot F_N(n))$, where $k$ is the number of elements in the set, and $F(n)$ is the time the *find* operation takes in one of the chosen union-find structures.

**copySet(A)** Creates a new set that is a copy of $A$ in $O(1)$.

**unionSets(A,B)** Creates a new set that contains all items that are in $A$ or $B$, for two disjunct sets $A$ and $B$, in $O(1)$.

**createItem(X)** Creates a new set containing only the (new) item $X$ in $O(1)$.

**deleteItem(X)** Deletes item $X$ from *all* sets in $O(1 + k)$, where $k$ is the number of sets $X$ is in.

## 9.3 Dynamic Segment Trees

This section recapitulates the workings of dynamic segment trees as presented by van Kreveld and Overmars [KO93] and outlines some extensions. Before we describe the dynamic segment tree, we shortly describe a classic static segment tree and the *segment tree property*. For a more thorough description, see de Berg et al. [Ber+08, page 10.2]. Segment trees store a set $\mathcal{I}$ of $n$ intervals. Let $x_1, x_2, \ldots x_{2n}$ be the ordered sequence of interval end points in $\mathcal{I}$. For the sake of clarity and ease of presentation, we assume that all interval borders are distinct, i.e., $x_i > x_{i+1}$. We also assume all intervals to be closed. See Section 9.3.2 for the straightforward way of dealing with equal interval borders as well as arbitrary combinations of open and closed interval borders.

segment tree property

In the first step, we forget whether an $x_i$ is a start or an end of an interval. The intervals

$$(-\infty, x_1), [x_1, x_1], (x_1, x_2), [x_2, x_2], \ldots (x_{2n-1}, x_{2n}), [x_{2n}, x_{2n}], (x_{2n}, \infty)$$

are called the *elementary intervals* of $\mathcal{I}$. To create a segment tree, we create a leaf node for every elementary interval. On top of these leaves, we create a binary tree. The exact method of creating the binary tree is not important, but it should adhere to some balancing guarantee to provide asymptotically logarithmic depths of all leaves.

elementary intervals

[1] The data structure presented by van Kreveld and Overmars provides more operations, but the ones mentioned here are sufficient for this chapter.

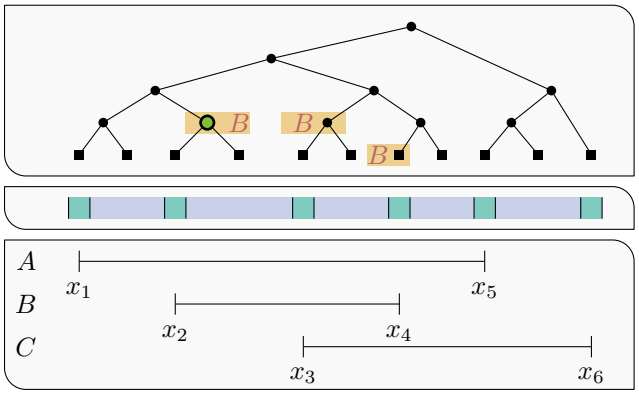



**Figure 9.1:** A segment tree (top) for three intervals (bottom). The middle shows the elementary intervals. Note that the green intervals do actually contain just one point and are only drawn fat so that they can be seen. The nodes marked with $B$ are the nodes that carry the annotation for interval $B$.

Such a segment tree is outlined in Figure 9.1. The lower box indicates the three stored intervals and their end points $x_1, \ldots x_6$. The middle box contains a visualization of the elementary intervals, where the green intervals are the $[x_i, x_i]$ intervals (note that while of course they should have no area, we have drawn them "fat" to make them visible) while the blue intervals are the $(x_i, x_{i+1})$ intervals. The top box contains the resulting segment tree, with the square nodes being the leaf nodes corresponding to the elementary intervals, and the circular nodes being the inner nodes.

We associate each inner node $v$ with the union of all the intervals corresponding to the leaves in the subtree below $v$. In Figure 9.1, that means that the larger red inner node is associated with the intervals $[x_2, x_3)$, i. e., the union of $[x_2, x_2]$ and $(x_2, x_3)$, which are the two leaves beneath it.

Recall that a segment tree should support fast stabbing queries, i.e., for any query point $q$, should report which intervals contain $q$. To this end, we annotate the nodes of the tree with sets of intervals. For any interval $I$, we annotate $I$ at every node $v$ such that the associated interval of $v$ is completely contained in $I$, but the associated interval of $v$'s parent is not. In Figure 9.1, the annotations for $B$ are shown. For example, consider the larger green node. Again, its associated interval is $[x_2, x_3)$, which is completely contained in $B = [x_2, x_4]$. However, its parent is associated with $[x_1, x_3)$, which is not contained in $B$. Thus, the large red node is annotated with $B$.

A segment tree constructed in such a way is semi-dynamic. Segments cannot be removed, and new segments can be inserted only if their end points are already end points of intervals in $I$. To provide a fully dynamic data structure with the same properties, van Kreveld and Overmars present the dynamic segment tree [KO93]. It relaxes the property that intervals are always annotated on the topmost nodes the

weak segment tree property

associated intervals of which are still completely contained in the respective interval. Instead, they propose the *weak segment tree property*: For any point $q$ and any interval $I$ that contains $q$, the search path[2] of $q$ in the segment tree contains exactly one node that is annotated with $I$. For any $q$ and any interval $J$ that does not contain $q$, no node on the search path of $q$ is annotated with $J$. Thus, collecting all annotations along the search path of $q$ yields the wanted result, all intervals that contain $q$. It is easy to see that this property is true for segment trees: For any interval $I$ that contains $q$, some node on the search path for $q$ must be the first node the associated interval of which does not fully contain $q$. This node contains an annotation for $I$.

Dynamic segment trees also remove the distinction between leaf nodes and inner nodes. In a dynamic segment tree, every node represents an interval border. To insert a new interval, we insert two nodes representing its borders into the tree, adding annotations as necessary. To delete an interval, we remove its associated nodes. If the dynamic segment tree is based on a classic red-black tree, both operations might require rotations to rebalance the tree. If annotations were not to be fixed after a rotation, such a rotation could cause the weak segment tree property to be violated. Also, the nodes removed when deleting an interval might have carried annotations, which also potentially violates the weak segment tree property.

We must thus fix the weak segment tree property during rotations. We must also make sure that any deleted node does not carry any annotations, and we must specify how we add annotations when inserting new intervals.

### 9.3.1 Red-Black Tree Operations

In this subsection, we provide details on how annotations can be repaired after red-black trees (or other rotation-based balancing binary search trees) have been rebalanced.

Since this simplifies notation, we change the notation of annotations: We assume not nodes to be annotated, but edges. Recall from Section 9.2 that we denote the left (resp. right) outgoing edge of a node $v$ as $\vec{L}(v)$ (resp. $\vec{R}(v)$). For an edge $e$, we denote its annotated set of intervals as $S(e)$. We also assume every node, even leaves, to have two outgoing (potentially annotated) edges — if a node $v$ has no left (resp. right) child, we write $L(v) = \perp$ (resp. $R(v) = \perp$). Transforming a node-annotation into an edge-annotation is trivial: The annotation of every node becomes the annotation of its incoming edge, except for the root node, which has no incoming edge. For the root node, its annotation is added to the annotations of both its outgoing edges.

Of the operations necessary for a dynamic segment tree, the problem of adding annotations after inserting a new interval is the easiest: Since the way of assigning annotations in static (or semi-dynamic) segment trees does fulfill the weak segment

[2] Recall the definition from Section 9.2.

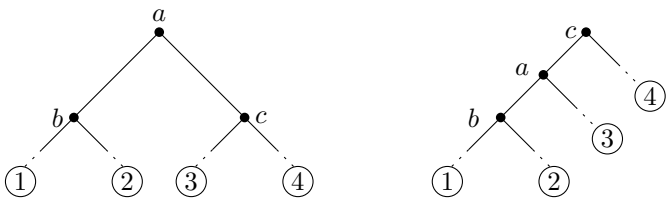



**Figure 9.2:** A counter-clockwise rotation around node $a$.

tree property, we just use this method: When inserting an interval $I$, we annotate it at every node $v$ (resp. its incoming edge) such that the associated interval of $v$ is completely contained in $I$, and the associated interval of $v$'s parent is not contained in $I$. Algorithmically, this is easy to do. Let $l$ and $u$ be the two nodes associated with the lower resp. upper border of $I$. Let $l = v_0^<, v_1^<, v_2^<, \ldots, \hat{v}, v_1^>, v_2^>, \ldots v_k^>, u = v_{k+1}^>$ be the unique path from $l$ to $u$, and let $\hat{v}$ be the node on this path with the least depth in the tree. Then, the node $\hat{v}$ is the lowest common ancestor of $l$ and $u$ in the tree, and the first node in which the search paths for $l$ and $u$ differ. The $v_i^<$ nodes are the nodes that are only on the search path for $l$, and the $v_i^>$ nodes are the nodes that are only on the search path for $u$. Note that this path may degenerate if e.g. $\text{LCA}(l, u) = l$ — in this case, $\hat{v} = l$ and there are no $v_i^<$. This does not impact the correctness. We now annotate $I$ on the right outgoing edge of every $v_i^<$, if $R(v_i^<) \notin \{v_{i+1}^<, \hat{v}\}$, i.e., if this is not an edge of the path between $l$ and $u$. Symmetrically, we annotate $I$ on the left outgoing edge of every $v_i^>$ if $L(v_i^>) \neq v_{i+1}^>$. This yields exactly the desired annotation.

For rotations, again see Figure 9.2. We only explain counter-clockwise rotation around $a$ — clockwise rotations works symmetrically. To ensure that the weak segment tree property still holds after the rotation, we must make sure that all search paths coming from the top and exiting the nodes $a$, $b$ and $c$ to the bottom have seen the same annotations after the rotation as they would have seen before. To achieve this, we push the annotations of $\vec{R}(a)$ down to $\vec{R}(c)$ and $\vec{L}(c)$ and clear the annotation at $\vec{R}(a)$. For an edge $e$, if the annotation before the push-down operation is $S(e)$, let the annotation after the push-down operation be $S'(e)$. Thus, we set $S'\big(\vec{L}(c)\big) = S\big(\vec{L}(c)\big) \cup S\big(\vec{R}(a)\big)$, $S'\big(\vec{R}(c)\big) = S\big(\vec{R}(c)\big) \cup S\big(\vec{R}(a)\big)$ and $S'\big(\vec{R}(a)\big) = \emptyset$. For all other edges, we set $S'(e) = S(e)$. The required operations, namely set union, set duplication and creation of an empty set, are efficient in the union-copy data structure. Let furthermore $S''(e)$ be the annotated set of an edge after the rotation has been performed. The only edges modified during the rotation are $\vec{R}(a)$ and $\vec{L}(c)$. We set $S''\big(\vec{R}(a)\big) = S'\big(\vec{L}(c)\big)$ and $S''\big(\vec{L}(c)\big) = S'\big(\vec{R}(a)\big)$, i.e., we swap the annotations of the two changed edges. For all other edges $e$, we set $S''(e) = S'(e)$.

We can now look at which search paths see which annotations before and after rotation. A search path exiting to ① picks up $S\big(\vec{L}(a)\big) \cup S\big(\vec{L}(b)\big)$ before rotation and $S''\big(\vec{L}(c)\big) \cup S''\big(\vec{L}(a)\big) \cup S''\big(\vec{L}(b)\big) = \emptyset \cup S\big(\vec{L}(a)\big) \cup S\big(\vec{L}(b)\big)$ after. Similar for ②: $S\big(\vec{L}(a)\big) \cup$

$S(\vec{R}(b))$ becomes $S''(\vec{L}(c)) \cup S''(\vec{L}(a)) \cup S''(\vec{R}(b)) = \emptyset \cup S(\vec{L}(a)) \cup S(\vec{R}(b))$. A path to ③ picks up $S(\vec{R}(a)) \cup S(\vec{L}(c))$ before and $S''(\vec{L}(c)) \cup S''(\vec{R}(a)) = \emptyset \cup (S(\vec{R}(a)) \cup S(\vec{L}(c)))$ after. Finally, a path to ④ picks up $S(\vec{R}(a)) \cup S(\vec{R}(c))$ before rotation and $S''(\vec{R}(c)) = S(\vec{R}(a)) \cup S(\vec{R}(c))$ after. This concludes the proof that after rotation, all search paths still see the same annotations as before.

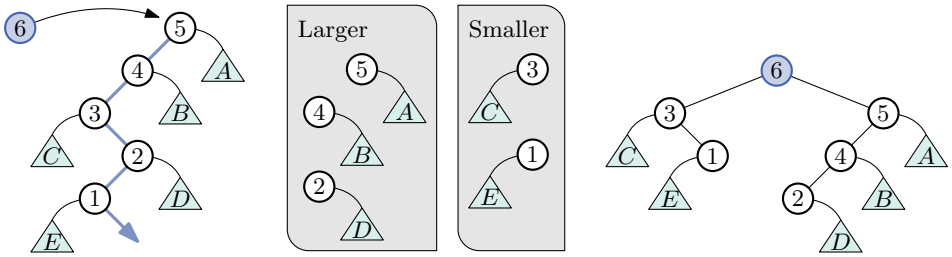
Deletion from a dynamic segment tree is handled elegantly by the union-copy data structure, by offering an operation that deletes an element from *all* sets. We start by using this operation to remove the deleted interval from all annotations at all edges. Then, we perform a normal red-black deletion operation on the two associated nodes. Red-black trees delete nodes as a sequence of up to three operations: First, if the node (call it $v$) to be deleted has two children, it is swapped with the next node in tree order. Call that node $w$. Note that before the swap, $w$ cannot have had a left child. Otherwise, there would have been a node smaller than $w$ in the right subtree of $v$, and that node would have been the successor of $v$ in tree order. If after a potential swap, $v$ has no children, it is just deleted. If it has one child, it is replaced by this child.

Fixing the annotations when deleting a leaf $v$ is easy in this case. The node $v$ represents an endpoint of an interval the annotations of which we have already all removed. It must hold that $S(\vec{L}(v)) = S(\vec{R}(v))$, since the decision whether a search path ends left or right of $v$ does not change the intervals it ends in anymore. We can therefore just promote one of these two annotations onto the edge from $v$'s parent to $v$ and delete $v$. A similar argument holds for the case that $v$ has exactly one child (say $R(v)$) and is replaced by this child. Again, the decision at $v$ does not distinguish between different sets of intervals anymore. In this case though, some of the annotations present in $S(\vec{L}(v))$ might have been pushed down into the subtree rooted in $R(v)$. Thus, we promote $S(\vec{R}(v))$ to the edge from $v$'s parent to $v$.

### 9.3.2 General Interval Borders

So far, we made two assumptions as to the nature of the intervals' borders: we did assume a total ordering on the keys of the nodes, i.e., no segment border may appear in two segments, and we assumed all intervals to be right-open. We now briefly show how to lift this restriction.

The important aspect is that a query path must see the nodes representing interval borders on the correct side. As an example, consider two intervals $I_1 = [a, b)$ and $I_2 = [b, c]$. A query for the value $b$ should return $I_2$ but not $I_1$. Thus, the node representing "$b$)" must lie to the left of the resulting search path, such that the query path does not end at a leaf between the nodes representing "[$a$" and "$b$)". That way, the annotation for $I_1$ will not be picked up. Conversely, the node representing "[$b$" must lie to the left of the query path, such that the query path ends in a leaf between the nodes representing "[$b$" and "$c$]".




**(a)** Before unzipping **(b)** Unzipped parts **(c)** After reassembly

**Figure 9.3:** Illustration of the process of unzipping a path in zip trees. Nodes' names are simultaneously their ranks. Node keys are not shown.

This dictates the ordering of nodes with the same numeric key $k$: First come the open upper borders, then the closed lower borders, then the closed upper borders, and finally the open lower borders. When querying for a value $k$, we descend left on nodes representing "$k$]" and "($k$", and descend right on nodes representing "[$k$" and "$k$)". That way, a search path for $k$ will end at a leaf after all closed lower borders and open upper borders, but before all closed upper borders and open lower borders. This yields the desired behavior.

## 9.4 Zipping Segment Trees

In Section 9.3 we have described a variant of the dynamic segment trees introduced by van Kreveld and Overmars [KO93]. These are built on top of a balancing binary search tree, for which van Kreveld and Overmars suggested using red-black trees. The presented technique is able to uphold the weak segment tree property during the operations necessary for red-black trees: node rotations, node swaps, leaf deletion and deletion of vertices of degree one. However, these are comparatively many operations that must be adapted to dynamic segment trees. Also, each repair of the weak segment tree property in each of these operations incurs a run time cost.

Thus it stands to reason to look at different underlying trees which either reduce the number of necessary balancing operations. One such data structure are *zip trees* introduced by Tarjan et al. [TLT19]. Instead of inserting nodes at the bottom of the tree and then rotating the tree as necessary to achieve balance, these trees determine the height of the node to be inserted before inserting it in a randomized fashion by drawing a *rank*. The zip tree then forms a *treap*, a combination of a search tree and a heap: While the key of $L(v)$ (resp. $R(v)$) must always be smaller or equal (resp. larger) to the key of $v$, the ranks of both $L(v)$ and $R(v)$ must also be smaller or equal to the rank of $v$. Thus, any search path always sees nodes' ranks in a monotonically

zip tree

rank

treap

decreasing sequence. The ranks are chosen randomly in such a way that we expect the result to be a balanced tree. In a balanced binary tree, half of the nodes will be leaves. Thus, we assign rank 0 with probability $1/2$. A fourth of the nodes in a balanced binary tree are in the second-to-bottom layer, thus we assign rank 1 with probability $1/4$. In general, we assign rank $k$ with probability $(1/2)^{k+1}$, i.e., the ranks follow a geometric distribution with mean 1. With this, Tarjan et al. show that the expected length of search paths is in $O(\log n)$, thus the tree is expected to be balanced.

zipping
unzipping

Zip trees do not insert nodes at the bottom or swap nodes down into a leaf before deletion. If nodes are to be inserted into or removed from the middle of a tree, other operations than rotations are necessary. For zip trees, these operations are *zipping* and *unzipping*. In the remainder of this section, we examine these two operations of zip trees separately and explain how to adapt them to preserve the weak segment tree property. For a more thorough description of the zip tree procedures, we refer the reader to [TLT19].

### 9.4.1 Insertion and Unzipping

Figure 9.3 illustrates the unzipping operation that is used when inserting a node. Note that we use the numbers 1 through 6 as nodes' names as well as their ranks in this example. The triangles labeled $A$ through $E$ represent further subtrees. The node to be inserted is ⑥, the fat blue path is its search path (i.e., its key is smaller than the keys of ⑤, ④ and ②, but larger than the keys of ③ and ①). Since ⑥ has the largest rank in this example, the new node needs to become the new root. To this end, we unzip the search path, splitting it into the parts that are — in terms of nodes' keys — larger than ⑥ and parts that are smaller than ⑥. In other words: We group the nodes on the search path by whether we exited them to the left (a larger node) or to the right (a smaller node). Algorithm 4, when ignoring the highlighted parts, provides pseudocode for the unzipping operation.

We remove all edges on the search path (Step 1 in Algorithm 4). The result is depicted in the two gray boxes in Figure 9.3b: several disconnected parts that are either larger or smaller than the newly inserted node. Taking the new node ⑥ as the new root, we now reassemble these parts below it. The smaller parts go into the left subtree of ⑥, stringed together as each others' right children (Step 3 in Algorithm 4). Note that all nodes in the "smaller" set must have an empty right subtree, because that is where the original search path exited them — just as nodes in the "larger" set have empty left subtrees. The larger parts go into the right subtree of ⑥, stringed together as each others' left children. This concludes the unzipping operation, yielding the result shown in Figure 9.3c. With careful implementation, the whole operation can be performed during a single traversal of the search path.

To insert a segment into a dynamic segment tree, we need to do two things: First, we must correctly update annotations whenever a segment is inserted. Second, we must ensure that the tree's unzipping operation preserves the weak segment tree property.

We will not go into detail on how to achieve step one. In fact, we add new segments in basically the same fashion as red-black-tree based DSTs do. We first insert the two nodes representing the segment's start and end. Take the path between the two new nodes. The nodes on this path are the nodes at which a static segment tree would carry the annotation of the new segment. Thus, annotating these nodes (resp. the appropriate edges) repairs the weak segment tree property for the new segment.

In the remainder of this section, we explain how to adapt the unzipping operations of zip trees to repair the weak segment property. Let the annotation of an edge $e$ before unzipping be $S(e)$, and let the annotation after unzipping be $S'(e)$. As an example how to fix the annotations after unzipping, consider in Figure 9.3 a search path that descends into subtree $D$ before unzipping. It picks up the annotations on the unzipped path from ⑤ up to ②, i.e., $S(\vec{L}(5))$, $S(\vec{L}(4))$, $S(\vec{R}(3))$, and on the edge going into $D$, i.e., $S(\vec{R}(2))$. After unzipping, it picks up the annotations on all edges that we newly inserted on the path from ⑥ to ② plus $S'(\vec{R}(2))$. We set the annotations on all newly inserted edges to $\emptyset$ after unzipping. Thus, we need to add the annotations before unzipping, i.e., $S(\vec{L}(5)) \cup S(\vec{L}(4)) \cup S(\vec{R}(3))$, to the edge going into $D$. We therefore set $S'(\vec{R}(2)) = S(\vec{R}(2)) \cup S(\vec{L}(5)) \cup S(\vec{L}(4)) \cup S(\vec{R}(3))$ after unzipping.

In Algorithm 4, the blue highlighted parts are responsible for repairing the annotations. While descending the search path to be unzipped, we incrementally collect all annotations we see on this search path (line 10), and at every visited node add the previously collected annotations to the other edge (line 9), i. e., the edge that is not on the search path. By setting the annotations of all newly created edges to the empty set (lines 23 and 26), we make sure that after reassembly, every search path descending into one of the subtrees attached to the reassembled parts picks up the same annotations on the edge into that subtree as it would have picked up on the path before disassembly.

### 9.4.2 Deletion and Zipping

Deleting segments again is a two-staged challenge: We need to remove the deleted segment from all annotations, and must make sure that the *zipping* operation employed for node deletion in zip trees upholds the weak segment tree property. Removing a segment from all annotations is trivial when using the union-copy data structure outlined in Section 9.2.1: The *deleteItem()* method does exactly this.

We now outline the *zipping* procedure and how it can be amended to repair the weak segment tree property. Zipping two paths in the tree works in reverse to unzipping. Pseudocode is given in Algorithm 5. Again, the pseudocode without the highlighted parts is the pseudocode for plain zipping, independent of any dynamic segment tree.

**Algorithm 4:** Unzipping routine. This inserts *new* into the tree at the position currently occupied by $v$ by first disassembling the search path below $v$, and then reassembling the different parts as left and right spines below *new*. The highlighted parts are used to repair the dynamic segment tree's annotations. Note that in an efficient implementation, one would interleave all four steps.

**Input:** *new*: Node to be inserted
**Input:** $v$: Node to be replaced by *new*

```
1  cur ← v;
2  oldParent ← P(v);
3  smaller ← newList(); larger ← newList();
4  collected ← createSet();
   /* Step 1: Remove edges along search path.                    */
5  while cur ≠ ⊥ do
6      if new < cur then
7          larger.append(cur);
8          next ← L(cur);
9          S(R⃗(cur)) ← unionSets((S(R⃗(cur)), collected);
10         collected ← unionSets(collected, S(L⃗(cur)));
11         next ← L(cur);
12         L(cur) ← ⊥;
13         cur ← next;
14     else
           /* Omitted, symmetric to the case new < cur.          */
   /* Step 2: Insert new.                                        */
15 if L(oldParent) = v then
16     L(oldParent) ← new;
17 else
18     R(oldParent) ← new;
   /* Step 3: Reassemble left spine from smaller parts.          */
19 parent ← new;
20 for n ∈ smaller do
21     if parent = new then
22         L(parent) ← n;
23         deleteSet(S(L⃗(parent))); S(L⃗(parent)) ← createSet();
24     else
25         R(parent) ← n;
26         deleteSet(S(R⃗(parent))); S(R⃗(parent)) ← createSet();
   /* Step 4: Reassemble right spine. This is symmetric to
      the left spine and thus omitted.                           */
```

Assume that in the situation Figure 9.3c, we want to remove ⑥, thus we want to arrive at the situation in Figure 9.3a. The zipping operation consist of walking down the left spine (consisting of ③ and ① in the example) and the right spine (consisting of ⑤, ④ and ② in the example) simultaneously and zipping both into a single path. This is done by the loop in line 7. At every point during the walk, we have a current node on both spines, call it the *current left* node $l$ and the *current right* node $r$. Also, there is a *current parent* $p$, which is the bottom of the new zipped path being built. In the beginning, the current parent is the parent of the node being removed.[3] In each step, we select the current node with the smaller rank, breaking ties arbitrarily (line 8). Without loss of generality, assume the current right node is chosen (the branch starting in line 20). We attach the chosen node to the bottom of the zipped path ($p$), and then $r$ itself becomes $p$. Also, we walk further down on the right spine.

Note that the choice whether to attach left or right to the bottom of the zipped path (made via *attachRight* in Algorithm 5) is made in such a way that the position in which we attach previously was part of one of the two spines being zipped. For example, if $p$ came from the right spine, we attach left to it. However, before zipping, $\vec{L}(p)$ was part of the right spine. This method of attaching nodes always upholds the search tree property: When we make a node from the right spine the new parent (line 29), we know that the new $p$ is currently the largest remaining nodes on the spines. We always attach left to that node (line 31). Since all other nodes on the spine are smaller than $p$, this is valid. The same argument holds for a node from the left spine.

We now explain how the edge annotations can be repaired during zipping so that the weak segment tree property is upheld. For this argument, assume that for an edge $e$, $S(e)$ is the annotation of $e$ before zipping, and $S'(e)$ is the annotation of $e$ after zipping. Again, we argue via the subtrees that search paths can descend into. A search path descending into a subtree on the right of a node on the right spine, e.g., subtree $B$ attached to ④ in Figure 9.3c, will — before zipping — always pick up the annotation on the right edge of the node being removed plus all annotations on the spine up to the respective node, e.g., $S\big(\vec{R}(6)\big) \cup S\big(\vec{L}(5)\big)$, before descending into the respective subtree ($B$ in the example). To preserve these picked up annotations, we again push them down onto the edge that actually leads away from the spine into the respective subtree.

Formally, during zipping, we keep two sets of annotations, one for the left and one for the right spine each. In Algorithm 5, these are $collected_l$ and $collected_r$, respectively. Let $n$ be the node to be removed. Initially, we set $collected_l = S\big(\vec{L}(n)\big)$ and $collected_r = S\big(\vec{R}(n)\big)$. Then, whenever we pick a node $c$ from the left (resp. right) spine as new parent, we set $S'\big(\vec{L}(c)\big) = S\big(\vec{L}(c)\big) \cup collected_l$ (resp. $S'\big(\vec{R}(c)\big) = S\big(\vec{R}(c)\big) \cup collected_r$). This pushes down everything we have collected so far onto the edge leading away

[3] If the root is being removed, pretend there is a pseudonode above the root.

**Algorithm 5:** Zipping routine. This removes $v$ from the tree, zipping the left and right spines of $v$. The highlighted parts are used to repair the dynamic segment tree's annotations.

**Input:** $n$: Node to be removed

```
1  l ← L(n);                          // Current node descending v's left spine
2  r ← R(n);                          // Current node descending v's right spine
3  p ← P(n);                          // Bottom of the partially zipped path
4  attachRight ← R(P(n)) = v;
5  collected_l ← copySet(S(L⃗(n)));
6  collected_r ← copySet(S(R⃗(n)));
7  while l ≠ ⊥ ∨ r ≠ ⊥ do
8      if (l ≠ ⊥) ∧ ((r = ⊥) ∨ (rank(l) > rank(r))) then
9          if attachRight then
10             R(p) ← l;
11             deleteSet(S(R⃗(p))); S(R⃗(p)) ← createSet();
12         else
13             L(p) ← l;
14             deleteSet(S(L⃗(p))); S(L⃗(p)) ← createSet();
15         S(L⃗(l)) ← unionSets(S(L⃗(l)), collected_l);
16         collected_l ← unionSets(collected_l, S(R⃗(l)));
17         p ← l;
18         l ← R(l);
19         attachRight ← true;
20     else
21         if attachRight then
22             R(p) ← r;
23             deleteSet(S(R⃗(p))); S(R⃗(p)) ← createSet();
24         else
25             L(p) ← r;
26             deleteSet(S(L⃗(p))); S(L⃗(p)) ← createSet();
27         S(R⃗(r)) ← unionSets(S(R⃗(r)), collected_r);
28         collected_r ← unionSets(collected_r, S(L⃗(r)));
29         p ← r;
30         r ← L(r);
31         attachRight ← false;
```

from the spine at $c$. Then, we set $collected_l = collected_l \cup S(\vec{R}(c))$ and $S'(\vec{R}(c)) = \emptyset$ (resp. $collected_r = collected_r \cup S(\vec{L}(c))$ and $S'(\vec{L}(c)) = \emptyset$).

This concludes the techniques necessary to use zip trees as a basis for dynamic segment trees, yielding *zipping segment trees*.

### 9.4.3 Numeric Annotations

Many applications of segment-storing data structures deal with weighted segments, i.e., each segment is associated with a number (or a vector of numbers). In such scenarios, one is often only interested in determining the aggregated weight of the segments overlapping at a certain point instead of the actual set of segments.

In a peak-demand scheduling heuristic such as the one presented in Chapter 8, the vector associated with every interval corresponds to the respective job's resource usage across the different resource types. If the dynamic segment tree holds intervals corresponding to a schedule, the aggregate value at a point in the tree then corresponds to the total demand in the schedule at that point.

This question can be answered without the need for a complicated union-copy data structure. In this case, we annotate each edge with a real number resp. with a vector. Instead of adding the actual segments to the $S(\cdot)$ sets at edges, we just add the associated weight of the segment. The copy operation is a simple duplication of a vector, a union is achieved by vector addition.

Deletion becomes a bit more complicated in this setting. Previously, we have exploited the convenient operation of deleting an item from all sets offered by the union-copy data structure. Now, say an interval associated with a weight vector $d \in \mathbb{R}^k$ is deleted from the dynamic segment tree, and the segment is represented by the two nodes $a$ and $b$. If we had just inserted the interval (and therefore $a$ and $b$), we would now add $d$ to the annotations on a certain set of edges (see above for a description of the insertion process). When deleting an interval, we annotate the same set of edges with $-1 \cdot d$. This exactly cancels out the annotations made when the interval was inserted.

### 9.4.4 Complexity

Zip trees are randomized data structures, therefore all bounds on run times are expected bounds. In [TLT19, Theorem 4], Tarjan et al. claim that the expected number of pointers changed during a zip or unzip is in $O(1)$. However, they actually even show the stronger claim that the number of nodes on the zipped (or unzipped) paths is in $O(1)$.[4] Observe that the loops in lines 5 and 20 of Algorithm 4 as well as line 7 of Algorithm 5 are executed at most once per node on the unzipping (resp. zipping) path. Inside each

[4]Note that nodes on these paths might not be changed — thus, the number of changed pointers could well be in $O(1)$ and the paths still be of non-constant length.

of the loops, a constant number of calls are made to each of the *copySet*, *createSet*, *deleteSet* and *unionSets* operations. Thus, the rebalancing operations incur expected constant effort plus a constant number of calls to the union-copy data structure.

When inserting a new segment, we add it to the sets annotated at every vertex along the path between the two nodes representing the segment borders. Since the depth of every node is expected logarithmic in $n$, this incurs expected $\ln(n)$ calls to *unionSets*. The deletion of a segment from all annotations costs exactly one call to *deleteItem*.

All operations but *deleteSet* and *deleteItem* are in $O(1)$ if the union-copy data structure is appropriately built. The analysis for the two deletion functions is more complicated and involves amortization. The rough idea here is that every non-deletion operation can increase the size of the union-copy's internal representation only by a limited amount. On the other hand, the two deletion operations each decrease the representation size proportionally to their run time.

The red-black-tree-based DSTs by van Kreveld and Overmars [KO93] also need $\Omega(\ln n)$ calls to *copySet* during the insertion operation, and at least a constant number of calls during tree rebalancing and deletion. Therefore, for every operation on zipping segment trees, the (expected) number of calls to the union-copy data structure's functions is no larger than the number of calls in the red-black-tree-based implementation and we achieve the same (but only expected) run time guarantees, which are $O(\log n)$ for insertion, $O(\log n \cdot a(i,n))$ for deletion[5] and $O(\log n + k)$ for stabbing queries, where $k$ is the number of reported segments.

### 9.4.5 Generating Ranks

Nodes' ranks play a major role in the rebalancing operations of zip trees. In Section 9.4 we already motivated why nodes' ranks should follow a geometric distribution with mean 1; it is the distribution of the node depths in a perfectly balanced tree.

A practical implementation needs to somehow generate these values. The obvious implementation would be to somehow generate a (pseudo-) random number and determine the position of the first 1 in its binary representation. The rank generated in this way is then stored at the respective node.

However, storing the rank at the node can be avoided if the rank is generated in a reproducible fashion. Tarjan et al. [TLT18] already point out that one can "compute it as a pseudo-random function of the node (or of its key) each time it is needed." In fact, the idea already appeared earlier in the work by Seidel and Aragon [SA96] on treaps. They suggest evaluating a degree 7 polynomial with randomly chosen coefficients at the (numerical representation of) the node's key. However, the 8-wise independence of the random variables generated by this technique is not sufficient to uphold the theoretical guarantees given by Tarjan et al. [TLT18].

[5] With $a(i,n)$ being the row-inverse of the Ackermann function, for some constant $i$.

However, without any theoretical guarantees, a simpler method for reproducible ranks can be achieved by employing simple hashing algorithms. Note that even if applying universal hashing, we do not get a guarantee regarding the probability distribution for the values of individual bits of the hash values. However, in practice, we expect it to yield results similar to true randomness. As a fast hashing method, we suggest using the $2/m$-almost-universal multiply-shift method from Dietzfelbinger et al. [Die+97]. Since we are interested in generating an entire machine word in which we then search for the first bit set to 1, we can skip the "shift" part, and the whole process collapses into a simple multiplication.

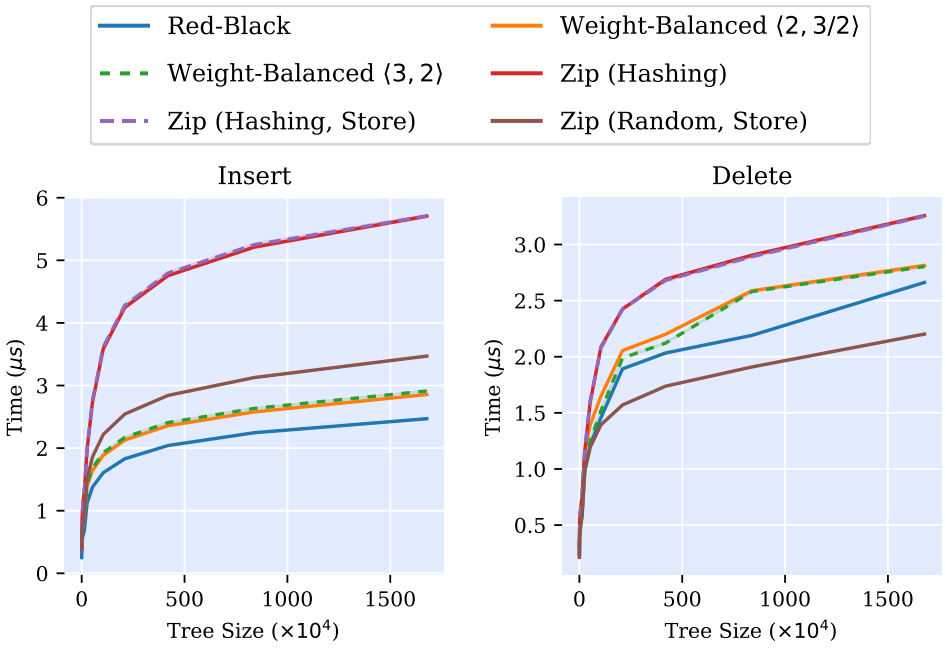
## 9.5 Experimental Evaluation of Dynamic Segment Trees Bases

In this section, we experimentally evaluate zipping segment trees as well as dynamic segment trees based on two of the most prominent rotation-based balanced binary search trees: red-black trees and weight-balanced trees. Weight-balanced trees require a parametrization of their rebalancing operation. In Chapter 10, we perform an in-depth engineering of weight-balanced trees. For this analysis of dynamic segment trees, we pick only the two most promising variants of weight-balanced trees: top-down weight-balanced trees with $\langle \Delta, \Gamma \rangle = \langle 3, 2 \rangle$ and top-down weight-balanced trees with $\langle \Delta, \Gamma \rangle = \langle 2, 3/2 \rangle$.

Note that since we are only interested in the performance effects of the trees underlying the DST, and not in the performance of an implementation of the complex union-copy data structure, we have implemented the simplified variant of DSTs outlined in Section 9.4.3. Evaluating the performance of the union-copy data structure is out of scope of this work.

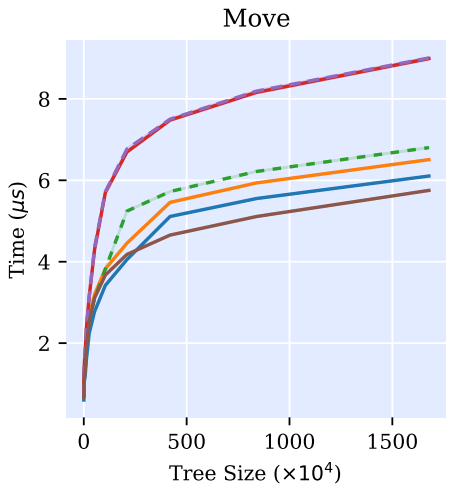
For the zip trees, we choose a total of three variants, based on the choices explained in Section 9.4.5: The first variant, denoted *Hashing*, generates nodes' ranks by applying the fast hashing scheme by Dietzfelbinger et al. [Die+97] to the nodes' memory addresses. In this variant, node ranks are not stored at the nodes but rather recomputed on the fly whenever they are needed. The second variant, denoted *Hashing, Store* also generates nodes' ranks from the same hashing scheme, but stores ranks at the nodes, increasing node sizes but removing the necessity of recomputing hashes. The last variant, denoted *Random, Store* generates nodes' ranks by a pseudo-random process independent of the nodes and stores the ranks at the nodes.

We first individually benchmark the two operations of inserting (resp. removing) a segment to (resp. from) the dynamic segment tree. Our benchmark works by first creating a base dynamic segment tree of a certain size, then inserting new segments (resp. removing segments) into that tree. The number of new (resp. removed) segments is chosen to be the minimum of $10^5$ and 5% of the base tree size. Segment borders are




**(a)** Timings for inserting a segment.

**(b)** Timings for deleting a segment.




**(c)** Timings for moving a segment.

**Figure 9.4:** Benchmark times for dynamic segment trees based on different balancing binary search trees. The $y$ axis indicates the measured time per operation, while the $x$ axis indicates the size of the tree that the operation is performed on. The lines indicate mean values. The standard deviation is all cases too small to be visible.

chosen by drawing twice from a uniform distribution. All segments are associated with a real-valued value, as explained in Section 9.4.3. We conduct our experiments on a machine equipped with 128 GB of RAM and an Intel® Xeon® E5-1630 CPU, which has 10 MB of level 3 cache. We compile using GCC 8.1, at optimization level "-O3 -ffast-math". We do not run experiments concurrently. To account for randomness effects, each experiment is repeated for 15 different seed values, and repeated five times for each seed value to account for measurement noise.

Figure 9.4a displays the results for the insert operation. We see that the red-black tree performs best for this operation, about a 30% faster ($\approx 2.5\mu s$ per operation at $1.5 \cdot 10^7$ nodes) than the fastest zip tree variant, which is the variant using random rank selection ($\approx 3.5\mu s$ per operation).

The two weight-balanced trees lie between the red-black tree and the randomness-based zip tree. Both hashing-based zip trees are considerably slower than all other trees.

For the deletion operation, shown in Figure 9.4b, the randomness-based zip tree is significantly faster than the next best competitor, the red-black tree. Again, the weight-balanced trees are slightly slower than the red-black tree, and the hashing-based zip trees fare the worst.

Since (randomness-based) zip trees are the fastest choice for deletion and red-black trees are the fastest for insertion, benchmarking the combination of both is obvious. Also, using an *dynamic* segment tree makes no sense if only the insertion operation is needed. Thus, we next benchmark a *move* operation, which consists of first removing a segment from the tree, changing its borders, and re-inserting it. The results are shown in Figure 9.4c. We see that the randomness-based zipping segment tree is the best-performing dynamic segment tree for trees with at least $2.5 \cdot 10^6$ segments.

The obvious measurement to explore why different trees perform differently is the trees' balance, i.e., the average depth of a node in the respective trees. We conduct this experiment as follows: For each of the trees under study, we create trees of various sizes with randomly generated segments. In a tree generated in such a way, we only see the effects of the *insert* operation, and not the *delete* operation. Thus, we continue by moving each segment once by removing it first, changing its interval borders and re-inserting it. This way, the effect of the *delete* operation on the tree balance is also accounted for. Since the weight-balanced trees were not competitive in our run time measurements, we perform this experiment only for the red-black and zip trees. We create each tree for 30 different seeds to account for randomness effects.

The somewhat surprising results can be found in Figure 9.5. We can see that zipping segment trees, whether based on randomness or hashing, are considerably less balanced than red-black-based DSTs. Also, whether ranks are generated from hashing or randomness does not impact balance.

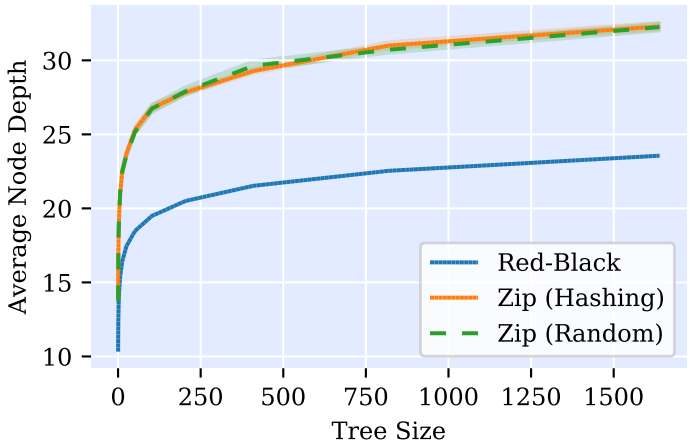



**Figure 9.5:** Average depths of the nodes in DSTs based on red-black trees and zip trees. The $x$ axis specifies the number of inserted segments. Shaded areas indicate the standard deviation.

Concluding the evaluation, we gain several insights. First, deletions in zipping segment trees are so much faster than for red-black-based DSTs that they more than make up for the slower insertion, and the fastest choice for moving segments are zipping segment trees with ranks generated randomly. Second, we see that this speed does not come from a better balance, but in spite of a worse balance. The speedup must therefore come from more efficient rebalancing operations. Third, and most surprising, the question of how ranks are generated does not influence tree balance, but has a significant impact on the performance of deletion and insertion. However, the hash function we chosen is very fast. Also, during deletion, no ranks should be (re-) generated for the variant that stores the ranks at the nodes. Thus, the performance difference can not be explained by the slowness of the hash function. Generating ranks with our chosen hash function must therefore introduce some disadvantageous structure into the tree that does not impact the average node depth.

**Code Publication.** We publish our C++17 implementation of all evaluated tree variants, including all code to replicate our benchmarks, at

`https://github.com/tinloaf/ygg`.

Note that this data structure library is still work in progress and might change after the publication of this work. The exact version used to produce the benchmarks shown in this chapter can be accessed at

`https://github.com/tinloaf/ygg/releases/tag/version_sea2020`.

## 9.6 Conclusion

We have presented zipping segment trees — a variation of dynamic segment trees, based on zip trees. The technique to maintain the necessary annotations for dynamic segment trees is comparatively simple, requiring only very little modification of zip trees' routines. In our experimental evaluation, we were able to show that zipping segment trees perform well in practice, and outperform red-black-tree or weight-balanced-tree based DSTs with regards to tree modifications.

However, we were not yet able to discover exactly why generating ranks from a (very simple) hash function does negatively impact performance. Exploring the adverse effects of this hash function and possibly finding a different hash function that avoids these effects remains future work. Another compelling future experiment would be to evaluate the performance when combined with the actual union-copy data structure by van Kreveld and Overmars.

All things considered, their relatively simple implementation and the superior performance when modifying segments makes zipping segment trees a good alternative to classical dynamic segment trees built upon rotation-based balancing binary trees.

# 10 Engineering Top-Down Weight-Balanced Trees

Weight-balanced trees are a popular form of self-balancing binary search trees. Their popularity is due to desirable guarantees, for example regarding the required work to balance annotated trees.

While usual weight-balanced trees perform their balancing operations in a bottom-up fashion after a modification to the tree is completed, there exists a top-down variant which performs these balancing operations during descend. This variant has so far received only little attention. We provide an in-depth analysis and engineering of these top-down weight-balanced trees, demonstrating their superior performance. We also gaining insights into how the balancing parameters necessary for a weight-balanced tree should be chosen — with the surprising observation that it is often beneficial to choose parameters which are not feasible in the sense of the correctness proofs for the rebalancing algorithm.

This chapter is based on joint work with Dorothea Wagner [BW20a].

## 10.1 Introduction

Weight-balanced trees (*WBT*s), originally introduced as *binary search trees of bounded balance* or *BB*[$\alpha$]-trees by Nievergelt and Reingold [NR73], later gained more attention through the seminal work by Knuth [Knu98], which also coined the name *weight-balanced trees* that is better known today. WBTs are balancing binary search trees. As many other flavours of balancing binary search trees, they employ rotations to correct imbalances caused by modifications to the tree. The specialty of weight-balanced trees is that the balancing is done based on the *weight* of subtrees, which is the number of nodes in the respective subtree.

This entails some interesting properties, such as the fact that it can be shown that rotations around *heavy* nodes, i. e., nodes that are roots of subtrees of a large weight, occur only rarely (see Mehlhorn [Meh84a]). Using this analysis, weight-balanced trees can serve as basis for augmented binary search trees, i. e., trees that carry additional annotations at every node. Usually, e.g. in the case of dynamic segment trees, these annotations depend on a node's children, thus the annotation must be repaired if the children are changed. If the effort to repair the annotation at a node after rotation correlates with the weight of the subtree rooted in that node, weight-balanced trees can be used to show amortized bounds on the necessary work. Annotated trees that require this property are often used in computational geometry, examples include Kurt Mehlhorn's Segment Trees ([Meh84b, Section VIII.5.1.3]) or Interval Trees ([Meh84b,

Section VIII.5.1.1]). Also, the weight annotation that every node in a weight-balanced tree carries can be used to efficiently implement order statistic trees (Cormen et al. [Cor+09, Chapter 15.1]).

These advantages of weight-balanced trees have led to them receiving ample attention throughout the literature. Adams [Ada93] gives a functional implementation of weight-balanced trees and claims they perform as well as red-black trees, however does not provide a practical evaluation. With weight-balanced trees, a set of *balancing parameters* (see Section 10.2.1) play a crucial role. While Nievergelt and Reingold introduced the technique and conjectured its correctness, the balancing technique does not work for the whole range of balance parameters they state in their paper. Later, Blum and Mehlhorn [BM80] not only point out this incorrectness, but also give a rigorous proof for a smaller space of the balancing parameters. Hirai and Yamamoto [HY11] use a computer-assisted proof system to discover the whole space of feasible balancing parameters. Cho and Sahni [CS00] present a variation of weight-balanced trees, which rotates subtrees even if they are not out of balance if the rotation reduces path lengths, thus reducing the expected average node depths within the tree. Roura presents two variations, one that uses logarithmic subtree sizes for balancing [Rou01] and one that uses the inverse of the Fibonacci function for balancing [Rou13].

This work focuses on analyzing the advantages of two variations in the weight-balanced trees: first, using a top-down balancing scheme, i. e., repairing the balance constraint while descending the tree for an insertion (resp. removal), instead of having a second bottom-up pass over the traversed tree path. Second, the effect that the choice of balancing parameters (especially "infeasible" parameters) has. The idea of top-down rebalancing has also been explored for other types of balanced binary search trees, such as red-black trees (Tarjan [Tar85]) or weak AVL trees (Haeupler et al. [HST15]). Rebalancing weight-balanced trees from the top down has an interesting history: While the original proposal (although incorrect, as Blum and Mehlhorn have shown) by Nievergelt and Reingold was already a top-down algorithm, the supplied proof by Blum and Mehlhorn only works for a bottom-up rebalancing. Later, Lai and Wood [LW93] have provided a top-down rebalancing algorithm and shown its correctness. This is the foundation for our contribution. However, the top-down variant of weight-balanced trees has received little attention so far. To our knowledge, no empirical analysis of top-down weight-balanced trees has been done yet.

**Our Contribution.** In this chapter, we provide a comprehensive experimental evaluation of top-down as well as bottom-up weight-balanced trees and the possible choices for the balancing parameters, resulting in recommendations when to use which tree variant based on the expected usage pattern. We gain the insight that top-down weight-balanced trees should be preferred over bottom-up weight-balanced trees, and most of the time they can compete with the performance of red-black trees. Moreover,

we gain the surprising insight that regarding the choice of balancing parameters, it often is beneficial to chose parameters that violate the theoretical guarantees in favor of a better empirical balance. We also publish thoroughly engineered implementations of all evaluated trees.

## 10.2 Top-Down Weight-Balanced Trees

In this section, we describe the top-down balancing approach for weight-balanced trees. We start by introducing notation and recapitulating the workings of bottom-up weight-balanced trees in Section 10.2.1.

### 10.2.1 Weight-Balanced Trees

weight-balanced tree

We denote a tree $T$ with node set $V$ and edge set $E$ as $T = (V, E)$. Every node $v$ can have a left (resp. right) child, which we denote by $L(v)$ (resp. $R(v)$), and say $L(v) = \perp$ (resp. $R(v) = \perp$) if $v$ has no left (resp. right) child. Additionally, in a weight-balanced tree, each node has an associated *weight*. Note that different notions as to what the weight of a node is are found throughout the literature. For us, the weight of $v$, denoted as $|v|$, is the number of nodes in the subtree rooted in $v$ plus one.[1] Thus, a leaf has weight 2. Also, since in the case $L(v) = \perp$ the left subtree has zero nodes, it results that $|L(v)| = 1$.

weight

The *balance criterion* for weight-balanced trees limits the relative difference between the weight of the left subtree and the right subtree at every node. The balance criterion and the balancing mechanism use two *balancing parameters*, $\langle \Delta, \Gamma \rangle$.[2] Balance is achieved at node $v$ if both

balance criterion

balancing parameter

$$|L(v)| \cdot \Delta \geq |R(v)| \text{ and} \tag{10.1}$$

$$|R(v)| \cdot \Delta \geq |L(v)|. \tag{10.2}$$

Note that the $\Gamma$ parameter is not directly relevant for the balance criterion. If the balance criterion is violated during a modification of the tree, the $\Gamma$ parameter is used to determine the correct balancing procedure. In [BM80], Blum and Mehlhorn show that if $\langle \Delta, \Gamma \rangle$ are chosen in a particular way, and rotations are applied as described in [NR73], this balance criterion is an invariant of the data structure at every node. The proof is technical and tedious, so we do not summarize it here.

**Insertion and Deletion in Bottom-Up Weight-Balanced Trees.** The first pass for insertion and deletion in *bottom-up* weight-balanced trees is performed as with

bottom-up

[1] Note that this corresponds to the number of $\perp$ entries in the subtree rooted in $v$.

[2] We are using the notation from Hirai and Yamamoto [HY11].

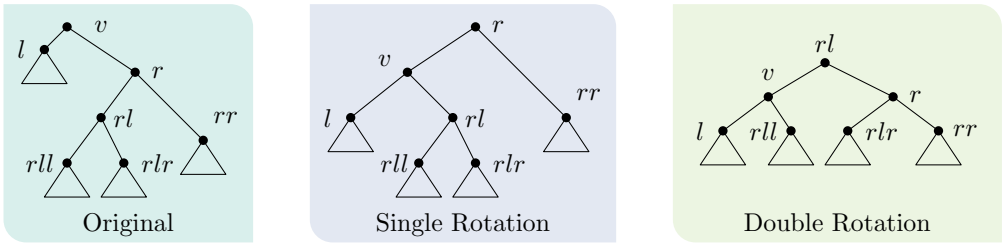



**Figure 10.1:** The result of a single left rotation around $v$ and a double rotation, first right around $r$, then left around $v$. Note that the node names differ from the notation in text to provide consistent labels before and after rotation. Also, the notation is reversed from the function notation to yield more natural node names. For example, $rll$ corresponds to $L(L(R(v)))$. Triangles indicate (possibly empty) subtrees that have been omitted.

unbalanced binary search trees. For an insertion, follow the search path for the new node until you walk out of a leaf. This is the position where to insert the new node. For a deletion of $v$, if $v$ is a leaf, just delete it. If it has only one child, replace $v$ by its only child, splicing the node out of the tree. Otherwise, find the largest node in $L(v)$ (resp. the smallest node in $R(v)$), and swap $v$ with that node. Now, $v$ has at most one child and we can proceed as above. The insertion procedure is also shown in Algorithm 6.

During the above, we do not pay attention to any balance criterion. Thus, after insertion and deletion, the balance might be violated at several nodes on the path from the tree's root to the position of insertion or deletion. In bottom-up weight-balanced trees, we repair the tree by traversing that path back up, repairing imbalances using single rotations and double rotations as necessary.

rotation

**Rebalancing Operation.** Whenever the balance criterion at a node $v$ is violated, a single or double *rotation* as depicted in Figure 10.1 is performed to reestablish balance. Since the process is symmetric for the right and left subtrees, we only discuss the case that the right subtree has become too heavy (because of an insertion into $R(v)$ or a deletion from $L(v)$).

Given a $\langle \Delta, \Gamma \rangle$ pair as defined above, the first decision at $v$ is whether to perform a rotation at all. A rotation is performed if (in the case of a possible right-overhang) $|L(v)| \cdot \Delta < |R(v)|$. A left rotation around $v$ will certainly reduce the weight of $v$'s right subtree, essentially removing $R(v)$ and the subtree rooted in $R(R(v))$ from below $v$. However if $L(R(v))$ is too heavy, after the rotation, the balance at the old $R(v)$ could be violated. Thus, if $|L(R(v))| > |R(R(v))| \cdot \Gamma$, we perform a double rotation as shown

in Figure 10.1.[3] This procedure has been shown to always reestablish balance at all involved nodes by Blum and Mehlhorn if $\langle \Delta, \Gamma \rangle$ is chosen appropriately.

**Balancing Parameter Space.** When talking about the balancing parameters $\langle \Delta, \Gamma \rangle$, we often call them *feasible* or *infeasible*. A parameter set $\langle \Delta, \Gamma \rangle$ is *feasible* (for bottom-up rebalancing resp. top-down rebalancing) if the respective balancing algorithm has been shown to be correct for $\langle \Delta, \Gamma \rangle$, i. e., if it is guaranteed that all nodes satisfy (10.1) and (10.2) after rebalancing. Otherwise, the parameter set is called *infeasible*. Note that an infeasible parameter set still yields a valid binary search tree.

Regarding the feasible values for $\langle \Delta, \Gamma \rangle$, the first thing to note is that the two correctness proofs from Blum and Mehlhorn as well as Lai and Wood [LW93] use a different notation than $\langle \Delta, \Gamma \rangle$. In these proofs, the balancing factor is $\alpha$, and the balancing criterion is

$$\alpha \leq \frac{|L(v)|}{|L(v)| + |R(v)|} \leq (1 - \alpha)$$

Looking at only one side of both types of balance constraints (the other side is symmetric), from $|L(v)|/(|L(v)| + |R(v)|) \geq \alpha$ and $\Delta|L(v)| \geq |R(v)|$, we get that $\Delta = (1 - \alpha)/\alpha$. In fact, using the upper bound on $\alpha$ given by Blum and Mehlhorn, $\alpha \leq 1 - \sqrt{2}/2$, this leads to $\Delta \geq 1 + \sqrt{2}$. Note that the larger the value for $\alpha$ (and the smaller the value for $\Delta$), the better we expect the tree to be balanced, i. e., we expect the smallest average node depths for these values. For their correctness proofs, both Blum and Mehlhorn as well as Lai and Wood fix the second balance parameter (the parameter deciding whether to use single or double rotation, call it $\gamma$) to $\gamma = 1/(2 - \alpha)$. Again, taking the two different forms of constraints for a double rotation, namely $|L(v)| > \Gamma|R(v)|$ and $|L(v)|/(|L(v)| + |R(v)|) > \gamma$, it follows that $\Gamma = \gamma/(1 - \gamma)$ and therefore $\Gamma = 1/(1 - \alpha)$. With this, for $\alpha = 1 - \sqrt{2}/2$, it follows that $\Gamma = \sqrt{2}$, and with that the most common (and maximally balanced) choice for $\langle \Delta, \Gamma \rangle = \langle 1 + \sqrt{2}, \sqrt{2} \rangle$.

However, Hirai and Yamamoto [HY11] have shown that in the bottom-up balancing case, the feasible space for $\langle \Delta, \Gamma \rangle$ is in fact a nonempty polytope, i. e., the linear dependency between $\Delta$ and $\Gamma$ (resp. $\alpha$ and $\gamma$) is not necessary. The only integral choice for $\langle \Delta, \Gamma \rangle$ within the polytope is $\langle 3, 2 \rangle$. Integral values for $\langle \Delta, \Gamma \rangle$ are interesting since (as Roura [Rou01] shows), using floating point arithmetic, or even worse, computing $\sqrt{2}$ during balancing, is a major factor slowing down weight-balanced trees. Note that with the relationship between $\Delta$ and $\Gamma$ (resp. $\alpha$ and $\gamma$) established by Blum and Mehlhorn, the $\Gamma$ value for $\Delta = 3$ would have been $\Gamma = 4/3$.

The correctness proof for top-down balancing from Lai and Wood holds only for $\alpha \leq 1/4$, meaning that we expect the best balanced top-down weight-balanced trees

[3] Note that the node names in the figure differ from the node names in text to allow for consistent names before and after rotation.

for $\alpha = 1/4$, which translates to $\langle \Delta, \Gamma \rangle = \langle 3, 4/3 \rangle$. Note that even though this means that $\Delta = 3$ is feasible for top-down balancing, the aforementioned $\langle 3, 2 \rangle$ possibly is not a feasible choice in the top-down case, since it is unclear how the feasible polytope looks like.

top-down

### 10.2.2 From Bottom-Up to Top-Down

Weight-balanced trees as described above perform two full traversals of the path from the tree's root to a leaf (resp. to-be-deleted node) for each insertion and deletion: One traversal down to perform the deletion or insertion, and one traversal up to check for and repair the balance. However, whenever we know that we will definitely delete a node (e.g., because we know that the value to be deleted is in the set represented by the tree), or that we will definitely insert a node (e.g., because we allow multiple nodes with the same value to be inserted), it is possible to perform necessary repair operations on the first traversal towards the leaves.

Algorithm 6 shows pseudocode for such an insertion. Note that while `Insert` descends the tree towards the insertion position for $n$, `RepairDuringInsertion` is called at every node, performing rotations as if $n$ was already inserted into the appropriate subtree, but without that subtree being rebalanced before. The pseudocode omits some technical details, such as correctly adjusting the weights of the nodes that become ancestors of $v$ because of a rotation, and correctly descending in case of a rotation. Consider as an example for a more complicated procedure the case that $n > v$, $n > R(v)$, that the insertion causes $|R(v)| > |L(v)| \cdot \Delta$ and that $|L(R(v))| > |R(R(v))| \cdot \Gamma$. Then, `RepairDuringInsertion` calls a double rotation (see Figure 10.1), after which $n$ should of course still be inserted below the old $R(R(v))$ ($rr$ in Figure 10.1). However, that node is not a descendant of $v$ anymore.

Note that this approach does not lead to the same trees as the bottom-up approach. In the bottom-up approach, during rebalancing at node $v$, balance is already established at $L(v)$, $R(v)$ and all nodes below. In the top-down approach, this balance can be violated by up to one node. For the top-down procedure, Lai and Wood show that even though the lower nodes cannot yet be assumed to be balanced, the above procedure balances all involved nodes, if $\langle \Delta, \Gamma \rangle$ is chosen appropriately.

The approach outlined here assumes that every insertion and removal always changes the tree. This is not necessarily the case, as a removal of a value that is not in the tree will fail, and so will insertion if the tree is used to implement a set (instead of a multiset) and the value is already in the tree. The case that the tree is not modified can naively be accommodated by having a second pass over the modified path in that case. Obviously, with this naive solution, top-down rebalancing is only a useful approach if the number of modifying insertions and removals is way higher than the number of non-modifying ones. However, careful analysis by Lai and Wood [LW93] shows that for a correct choice of rebalancing parameters, top-down rebalancing keeps the

**Algorithm 6:** Top-down insertion of a node $n$.

```
1  Function RepairDuringInsertion (n, v):
2      if n ≤ v then
3          if |L(v)| + 1 > |R(v)| · Δ then  /* +1 b/c we insert into left
             subtree */
4              if (n ≤ L(v) and |R(L(v))| > (|L(L(v))| + 1) · Γ) or
5                 (n > L(v) and |R(L(v))| + 1 > |L(L(v))| · Γ)) then
6                  doubleRotation(v);
7              else
8                  singleRotation(v);
9      else
           /* Omitted, symmetric to the case n ≤ v                */
10 Function Insert (n):
11     v ← root;
12     while true do
13         |v| ← |v| + 1;
14         RepairDuringInsertion (n, v);
15         if n ≤ v then
16             if L(v) = ⊥ then
17                 L(v) ← n;
18                 return;
19             else
20                 v ← L(v);
21         else
               /* Omitted, symmetric to the case n ≤ v            */
```

balancing criterion intact even if the algorithm aborts the operation during descend, either because a key to be deleted is not in the tree or because a key to be inserted is already in the tree.[4]

## 10.3 Evaluation

We now provide an in-depth experimental evaluation of the various flavours of weight-balanced trees. This evaluation encompasses multiple parts: First, we measure the time that operations such as inserting into and removing from the trees take in Section 10.3.1. Since the time necessary to search for a vertex in a tree is only dependent on the depth of the respective node, and measuring the average node depth is less noisy than

[4] Lai and Wood call this a *redundant* operation.

measuring the time a search takes, we use this measure instead of measuring search timings in Section 10.3.2. Also in that section we look at how much the balancing criterion is violated when one chooses balancing parameters outside the feasible space. All these analyses are done for different kinds of test data, resembling a broad spectrum of use cases. To study the various rebalancing schemes in even more realistic scenarios, we use sequences of tree operations captured during the execution of an optimization algorithm utilizing a balancing search tree in Section 10.3.3. Finally, we take a look at the total number and weight of rotated nodes in Section 10.3.4.

We implemented all trees in C++, our implementation including all the benchmarking code can be found at:

```
https://github.com/tinloaf/ygg/releases/tag/version_thesis
```

Additionally, we publish all raw results obtained from our experiments as a separate data publication [BW19b]. See Section C.1 in the appendix for further details.

All measurements are taken on a machine with 192 GBs of DDR4 memory and two eight-core Intel® Xeon® Gold 6144 CPUs, which have 32 KB of L1 data cache per core, 1 MB of L2 cache per core and a total of 25 MB of L3 cache per CPU. However, we did not run multiple benchmarks concurrently. The size of our trees' nodes is 40 bytes. In the following experiments, the largest tested trees usually have size $\approx 4 \times 10^6$, which leads to a memory footprint of around 150 MB, well above L3 cache sizes. We therefore expect to see the effects of caching for the larger tested trees, and little to no caching effects for trees of at most $6 \times 10^5 \approx 25$ MB/40 B nodes.

In the following evaluation, we compare the following balanced binary search trees: First, a (bottom-up) red-black tree as baseline, denoted *red-black*. Second, the basic version of a weight-balanced tree, with bottom-up balancing, denoted *bottom-up*. Third, the top-down weight-balanced tree, denoted *top-down*. For the top-down weight-balanced trees, we evaluate different choices for the balancing parameters $\langle \Delta, \Gamma \rangle$: First, the choices listed and explained in Section 10.2.1: $\langle 1+\sqrt{2}, \sqrt{2} \rangle$ (the original parameter set given by Blum and Mehlhorn [BM80]), $\langle 3, 2 \rangle$ (the integral parameters suggested by Hirai and Yamamoto [HY11]) and $\langle 3, 4/3 \rangle$ (the parameters from the top-down correctness proofs by Lai and Wood [LW93]). Note that even though the first two are not feasible in the sense of the top-down correctness proof by Lai and Wood, we still use them for the top-down balancing technique. Similarly, we try the additional choice of $\langle 2, 3/2 \rangle$. Even though $\Delta = 2$ is not a feasible choice for top-down *or* bottom-up balancing, we want to evaluate how this smaller $\Delta$ value (which we expect to lead to a better balance) performs in practice. For an even more extreme example, we also evaluate $\langle 3/2, 5/4 \rangle$.

### 10.3.1 Timing Operations

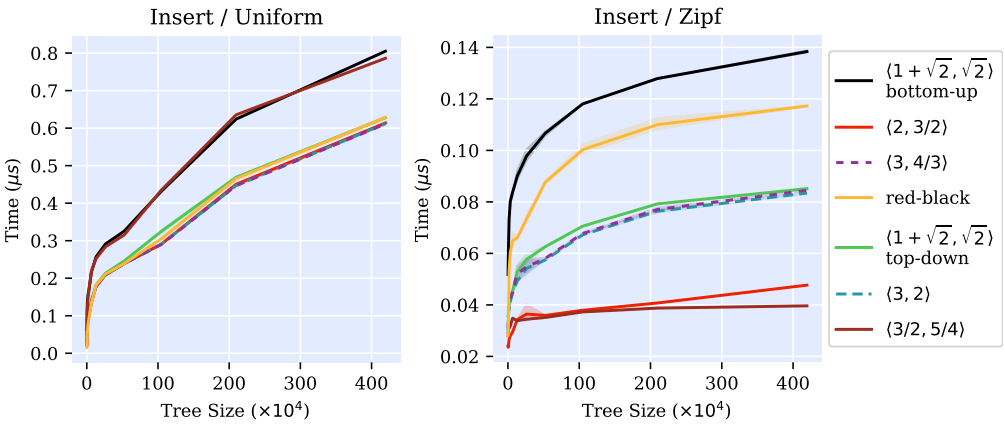
We first benchmark the two basic operations insertion and deletion. Our aim is to measure the time these operations take on trees of various sizes for different distributions of nodes' keys. Specifically, for each benchmark we first create a random tree of a certain base size (the *base tree*), and then remove five percent of the nodes resp. insert five percent new nodes. For all benchmarks, we employ four different methods to generate nodes' keys: First in the `uniform` case, we generate keys uniformly at random. Second, we assume that the search tree may be used to index data that pertains to physical or social sciences. In this case, Zipf's Law (see [Pow98]) states that this data, e. g. text corpora, often follow a Zipf distribution. We accommodate this fact with the `zipf` case, in which nodes' keys are picked using a Zipf distribution. Third, it seems prudent to study cases with a heavy concentration of the keys in one or two areas of the key space. For this, we use the `skewed` distribution suggested by Mäkinen [Mäk87] in his analysis of top-down splay trees. In this distribution, every third value is drawn from a uniform distribution over the whole key space, and the other two thirds are drawn from two uniform distributions each spanning only 10% of the key space. Finally, an obvious benchmark case for balancing search trees is partially pre-sorted data. In the `pre-sorted` case, we first take a sequence of sorted numbers, and then randomly permute half of them. For the deletion benchmark, the node to be deleted is picked uniformly at random in each case.

Note that we do not discuss each plot individually in this section, but only those from which interesting insights can be drawn. The plots that are not mentioned in the text can be found in Appendix C.2.

To account for randomness effects and measurement noise, we run each experiment for each base tree size on 10 different base trees, and in turn repeat the experiment itself on each base tree until the experiment ran for at least one second on each base tree.

Figure 10.2a shows the time (averaged over all the iterations explained above) it takes to insert 5% new nodes into the seven different trees of various base sizes, for the `uniform` case. We first see that the bottom-up variant (with $\langle 1+\sqrt{2}, \sqrt{2}\rangle$) is about 30% (resp. $0.2\mu s$) slower than the corresponding top-down variant. We then see that the red-black tree and the $\langle 2, 3/2\rangle$, $\langle 3, 4/3\rangle$ and the top-down $\langle 1+\sqrt{2}, \sqrt{2}\rangle$ variants all show almost the same performance. Interestingly, the variant with the tightest balancing parameter, $\langle 3/2, 5/4\rangle$ performs as bad as the bottom-up variant. Note that in this (and all following) plots, shaded areas indicate standard deviation. Where no shaded area is visible, the standard deviation is too small to be visible.

When performing the same experiment with node keys chosen from a Zipf distribution (shown in Figure 10.2b), results look very different: Here, the two strongly balanced variants ($\langle 2, 3/2\rangle$ and $\langle 3/2, 5/4\rangle$) outclass all other variants. Also, all but the top-down variant outclass the red-black tree, with a factor of 3 between the best




**(a)** Nodes' keys chosen from a uniform distribution.

**(b)** Nodes' keys chosen from a Zipf distribution.

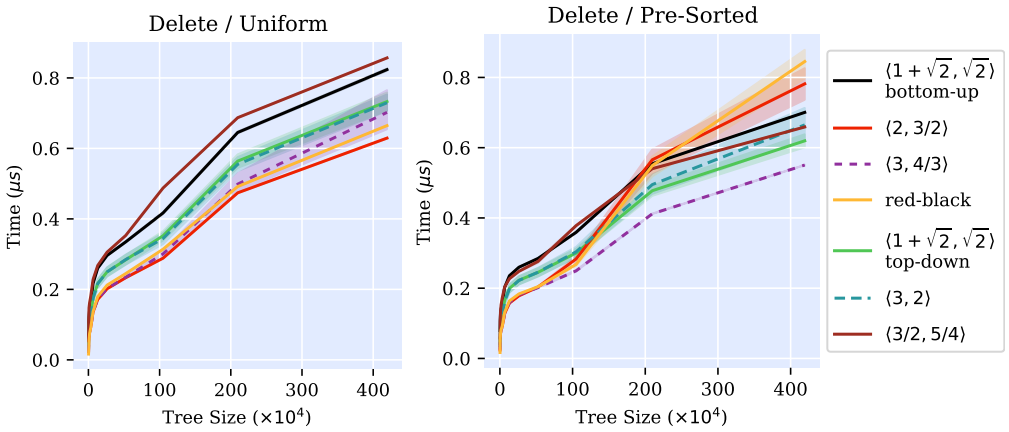
**Figure 10.2:** Times to insert 5% new nodes into trees of various sizes. The $x$ axis specifies the size of the base tree. The $y$ axis reports the time needed for a single insertion in microseconds. Shaded areas indicate standard deviation.

balanced weight-balanced tree and the red-black tree. Results for the `skewed` distribution (shown in Figure C.1a in Appendix C.2) are less pronounced, but similar. For the `pre-sorted` case, the results are very similar to the `uniform` case and can be found in Figure C.1b in Appendix C.2. Note that in absolute terms, insertion on Zipf-distributed trees is a lot faster than for example on the uniformly distributed ones. This can be explained with caching: In a Zipf-distributed tree, which is heavily skewed towards small keys, most operations access only very few search paths, namely the search paths to the nodes with small keys. Holding the nodes on these search paths in cache allows the tree to perform most work directly in cache.

Since we implemented all mentioned trees ourselves, the question of how efficient our implementations are as a whole comes to mind. For the insertion benchmark, we added the C++ STL's `std::multiset`[5] and Boost's `intrusive::multiset` to the comparison. The plot can be found in Appendix C.2, Figure C.3. As can be seen, all our trees perform slightly better than `std::multiset`, but slightly worse than `boost::intrusive::multiset`. We may therefore assume that our implementations are properly optimized.

Next, we look at the deletion operation. Figure 10.3a shows the `uniform` case. We see that the $\langle 2, 3/2\rangle$ variant has a slight advantage over the red-black tree and all other variants. For deletion, the `pre-sorted` case (shown in Figure 10.3b) is especially interesting: Here, all weight-balanced trees, but especially the $\langle 3, 2\rangle$ variant, clearly

[5]STL bundled with GCC 8.1, which implements std::multiset as a red-black tree.



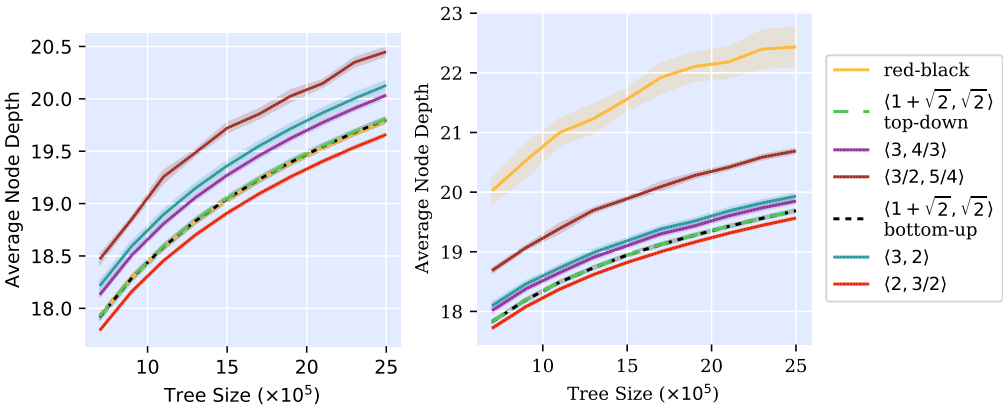

**(a)** Node keys generated from a uniform distribution.

**(b)** Node keys generated in the `pre-sorted` fashion.

**Figure 10.3:** Times to delete 5% nodes from trees of various sizes. The $x$ axis specifies the size of the base tree. The $y$ axis reports the time needed for a single deletion in microseconds. Shaded areas indicate standard deviation.

**Table 10.1:** Summary of the benchmark findings, specifying which weight-balanced tree variant was the best for each of our benchmark cases. Where two variants could virtually not be distinguished, we specify both. A checkmark signifies that in this case, the best weight-balanced tree outperformed the red-black tree, a cross the opposite.

| | Deletion | Insertion |
|---|---|---|
| `uniform` | $\langle 2, 3/2\rangle$ ✓ | $\langle 3, 4/3\rangle/\langle 2, 3/2\rangle$ ✓ |
| `skewed` | $\langle 3, 4/3\rangle$ × | $\langle 3, 4/3\rangle/\langle 2, 3/2\rangle$ × |
| `zipf` | $\langle 3, 2\rangle$ × | $\langle 3/2, 5/4\rangle$ ✓ |
| `pre-sorted` | $\langle 3, 4/3\rangle$ ✓ | $\langle 3, 4/3\rangle/\langle 2, 3/2\rangle$ ✓ |




**(a)** Node keys generated from a uniform distribution.

**(b)** Node keys generated from a Zipf distribution.

**Figure 10.4:** Average node depth for various trees. The $x$ axis specifies the size of the tree, the $y$ axis the average node depth. All nodes in every tree were randomly generated, removed once, had their key changed, and were reinserted. The solid lines indicate average values, the shaded areas the standard deviation.

outperform the red-black tree. On the other side of the spectrum, for the `skewed` and `zipf` cases (shown in Appendix C.2, Figure C.2), the red-black tree has a slight advantage over the weight-balanced trees.

Our benchmark findings are summarized in Table 10.1. From the results, we can deduce that one should always use the top-down variant, and should never use $\langle 1 + \sqrt{2}, \sqrt{2} \rangle$ as balancing parameter. Whether $\Delta = 2$ or $\Delta = 3$ is the wiser choice depends on the expected usage pattern. We can also see that the race between red-black trees and weight-balanced trees is a toss-up: While weight-balanced trees seem to be ahead in the `uniform` and `pre-sorted` cases, red-black trees exhibit better performance in the `skewed` and `zipf` cases.

### 10.3.2 Tree Balance

Aside from insertion and deletion times, an interesting metric is the average depth of a node. The average depth determines the expected length of the search path for that node, which not only influences insertion and removal speeds, but even more strongly the search performance. In fact, we do not benchmark runtimes for searches within the trees, since the average depth of the nodes should be the only influencing parameter, with everything else being measurement noise. To analyze the average node depth, we again create random trees of various sizes. Since just creating a tree does not involve the remove operation, and we also want to evaluate the effects of

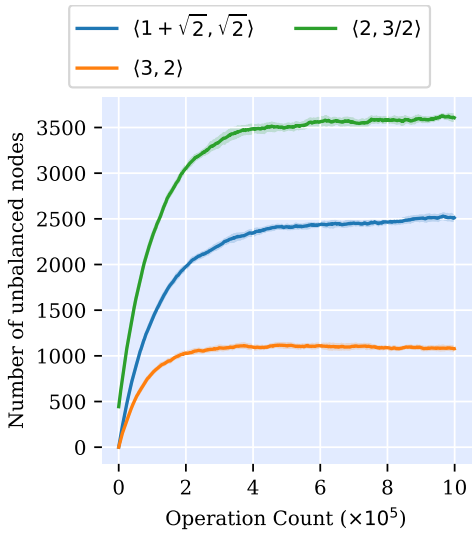



**Figure 10.5:** Number of unbalanced nodes on the $y$ axis versus number of remove / insert operations on random trees of size $10^6$ on the $x$ axis. The solid line reports the mean value, the shaded area indicates the standard deviation.

this operation, we iterate over all nodes after creating the tree, and first remove each node from the tree, change its key, and then reinsert it. After this, we compute the average depth of all nodes. Figure 10.4a shows the results for keys being drawn from a uniform distribution.

We see that red-black trees and weight-balanced trees using $\langle 1 + \sqrt{2}, \sqrt{2} \rangle$ as balance parameters are virtually equally well balanced. The weight-balanced tree using $\langle 2, 2/3 \rangle$ has a slight advantage over them — as we expected, since $\Delta = 2$ enforces a stricter balance than $\Delta = 1 + \sqrt{2}$. However, the $\langle 3/2, 5/4 \rangle$ variant is the worst in terms of balance, even though it is using the smallest $\Delta$. This suggests that choosing parameters that are too far outside of the space of feasible choices for $\langle \Delta, \Gamma \rangle$, the balancing criterion is violated too badly for the smaller $\Delta$ to make up for it.

Using a Zipf distribution instead of a uniform distribution for the nodes' keys (shown in Figure 10.4b) reveals that while the various weight-balanced trees are almost unaffected by the heavily skewed distribution, the red-black tree handles it a lot worse, with more than 10% difference between the best weight-balanced tree and the red-black tree. Interestingly, the `skewed` case, shown in Figure C.4 in Appendix C.2, shows results very similar to the `uniform` case.

The fact that in Figure 10.4, the values for the top-down weight-balanced tree with $\langle 1 + \sqrt{2}, \sqrt{2} \rangle$ do not differ much from the bottom-up weight-balanced tree with the same balancing parameters (which are infeasible for a top-down balancing approach), and that the (infeasible) parameter pair $\langle 2, 3/2 \rangle$ outperforms all other trees, hint at the

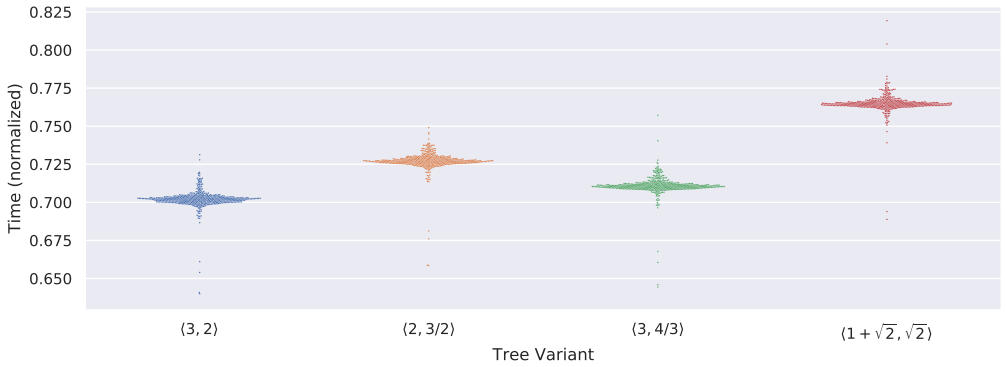



**Figure 10.6:** Times elapsed during the execution of each captured sequence, normalized to the time it took the bottom-up weight balanced tree with $\langle 1 + \sqrt{2}, \sqrt{2} \rangle$. Every dot is one sequence.

fact that even infeasible balancing parameters for a top-down balancing approach may produce little to no balance violations in practice. We examine this claim by continually counting the number of nodes at which balance is violated while repeatedly removing and re-inserting (with a changed value) random nodes from resp. into a random tree. Figure 10.5 shows the results for a tree of size $10^6$. We repeat the experiment with 10 different seeds, the line indicates the mean, the shaded areas indicate the standard deviation. We see that for all three[6] evaluated variants, the number of nodes at which balance is violated stabilizes after approximately $4 \times 10^5$ removals and insertions.[7] We also see that even for the worst of the parameter choices, $\langle 2, 3/2 \rangle$, only about 0.35% of all nodes are unbalanced after $10^6$ operations. This behavior can be explained by the fact that unbalanced nodes will likely be rebalanced by the next operation that passes over them. Thus, we also expect the unbalanced nodes to have large depths, since nodes close to the root are passed over very frequently.

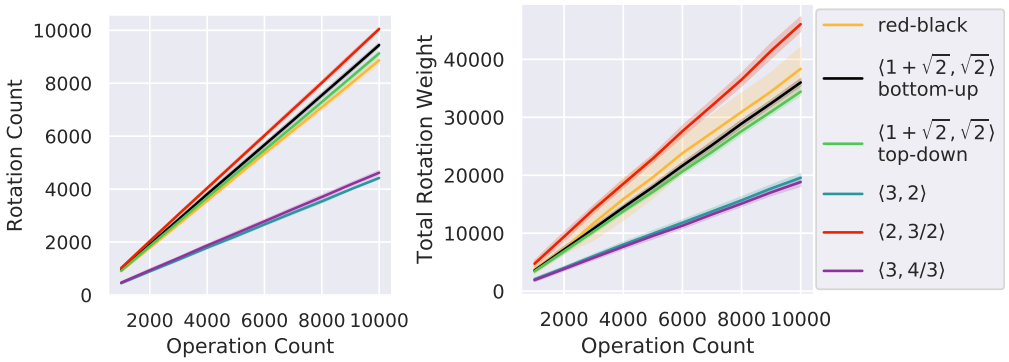
### 10.3.3 Real-Life Sequences

After the experiments on randomly generated data, we finally take a look at tree operations generated from an algorithm that heavily relies on balancing binary trees. To this end, we instrumented the SWAG algorithm presented in Chapter 8. This is a scheduling algorithm that in its innermost loop uses a dynamic segment tree, which is built on top of a balancing binary search tree. Note that the algorithm only deletes from and inserts into the tree and never performs any searches.[8] We collected a total of 514 sequences of tree operations. To benchmark our weight-balanced trees, we

[6] We excluded $\langle 3/2, 5/4 \rangle$ here, since it breaks the balancing so badly that it distorts the plot.

[7] One removal and one insertion count as one operation.

[8] While this might seem useless, the information needed by the scheduling algorithm is computed in an annotation at the root of the tree.




**(a)** Total number of of performed rotations (on the $y$ axis) after a number of operations (on the $x$ axis).

**(b)** Total node weight of rotated nodes (on the $y$ axis) after a number of operations (on the $x$ axis).

**Figure 10.7:** Rotation count and rotated node weight for several kinds of trees of size $10^6$ after various numbers of operations. Solid lines report the mean value, while shaded areas indicate the standard deviation.

replay each sequence ten times for every weight-balanced tree variant. Figure 10.6 shows the results, where every dot is the time (averaged over the ten iterations) it took the tree indicated by the $x$ axis to execute one sequence, normalized to the time it took the bottom-up $\langle 1 + \sqrt{2}, \sqrt{2} \rangle$ variant to execute the same sequence. We see that in this specific use case, the $\langle 3, 2 \rangle$ variant suggested by Hirai and Yamamoto (for the bottom-up variant) performs the best, being about 30% faster than the bottom-up variant — again, we see the best results for a parameter choice that is infeasible for top-down rebalancing. The top-down feasible variant $\langle 3, 4/3 \rangle$ performs slightly worse. But even the worst variant, $\langle 1+\sqrt{2}, \sqrt{2} \rangle$ still is more than 23% faster than the bottom-up variant. Note that we have excluded the $\langle 3/2, 5/4 \rangle$ variant here. It has its mean at approximately 1.15 and would distort the plot.

### 10.3.4 Rotated Node Weight

For the final evaluation step, we consider that weight-balanced trees are often chosen because the total weight of the nodes rotated around can be theoretically bounded. This is useful if rotations around larger nodes are expensive, for example because of annotations that need to be repaired. We explore their behavior in this regard by creating random trees of size $10^6$, performing a number of operations (where every operation consists of one node removal and reinsertion with changed key) on them and counting how many rotations occurred, and what the total weight of the rotated nodes is. Figure 10.7a shows how the number of rotations increases with increasing

number of operations, Figure 10.7b shows the same for the total weight of the rotated nodes. Note that we excluded the $\langle 3/2, 5/4 \rangle$ variant here, since $\Delta = 3/2$ is such a strong balancing requirement that the number of rotations is about 20 times larger than for all the other variants, thus including it would have distorted the plot. The most striking point is that both numbers are significantly smaller for the variants with $\Delta = 3$, usually roughly half the number of rotations (resp. total rotation weight) than for the other variants or the red-black tree. The less strict balancing requirement apparently drastically reduces the number of necessary rotations. Whether one uses top-down or bottom-up balancing does not seem to make a serious difference.

It is also notable that the red-black tree, although not possessing a similar theoretical guarantee, does not perform significantly worse in terms of rotation count or weight than the weight-balanced trees with $\Delta < 3$, although its total rotation weight has a much larger standard deviation. Consistent with the finding for $\Delta = 3$, the weight-balanced tree with $\Delta = 2$ performs the worst in terms of rotations.

## 10.4 Conclusion

In the chapter on hand, we evaluated and engineered top-down weight-balanced trees. A rigorous evaluation has shown that using a top-down balancing approach instead of a bottom-up approach in fact leads to a significant performance increase, if one chooses the correct balancing parameters. The correct choice of balancing parameters can even make weight-balanced trees more performant than red-black trees, which is surprising considering the fact that red-black trees are used widely, while weight-balanced trees have received little attention in practice. However, the balancing parameters should be chosen with the intended use for the weight-balanced tree in mind. If little modification and a lot of searches are expected, we recommend using $\langle 2, 3/2 \rangle$ because of its superior average node depth. Even stronger balanced choices such as $\langle 3/2, 5/4 \rangle$ do not look advisable. One should also consider the expected distribution of nodes' keys. For strongly skewed distributions, as for example the `zipf` case, smaller $\Delta$ values such as $\langle 2, 3/2 \rangle$ tend to be advantageous. Also, for these distributions, weight-balanced trees should be chosen over red-black trees, as our analysis of average node depth has shown.

In case that the weight-balanced tree is annotated and rotations, especially around large nodes, are costly, using $\langle 3, 2 \rangle$ or even larger $\Delta$ values is likely to be the best of the evaluated choice. In fact, our benchmark of insert and deletion suggests that $\langle 3, 2 \rangle$ and $\langle 3, 4/3 \rangle$ are overall fairly performant choices, even if their average path lengths might be slightly inferior. It never seems to be a good choice to use the classic $\langle 1 + \sqrt{2}, \sqrt{2} \rangle$ variant. Summarizing these recommendations, it is surprising that empirically, many times the best choice for balancing parameters are parameters for which the theoretical guarantees do not hold, especially in the top-down rebalancing case. Of course, these

parameters are only a viable choice if one does not have to worry about artificially crafted adversarial instances.

In the future, it would be interesting to determine the space of feasible balancing parameters for top-down weight-balanced trees similar to how Hirai and Yamamoto have done for bottom-up weight-balanced trees.

# 11 TCPSP is Fixed-Parameter Tractable in a Local Measure

Most project scheduling problems that occur in the real world are NP-hard. One way of coping with NP-hardness is to isolate aspects of the problem which are the main contributors to the complexity of the problem. Finding such a parameter can be a valuable tool in classifying the instances of the problem which might still be optimizable with reasonable effort.

In this chapter, we present such a parameter for the Time-Constrained Project Scheduling Problem. We present a simple enumerative exponential-time algorithm and show that if a certain parameter, which captures a measure of local complexity, is bounded, the algorithm solves the problem in pseudo-polynomial time.

## 11.1 Introduction

Project scheduling problems of various flavors play an important role in optimization of industrial processes. Project scheduling methods can also be used to optimize schedules of flexible electrical loads in smart grids and facilitate demand response, as we show extensively in Part I of this thesis. For smart grid scheduling, variants of the Time-Constrained Project Scheduling Problem (TCPSP) are of special interest. In this chapter, we present an analysis of the complexity of TCPSP based on a simple enumerative exponential-time algorithm. The analysis yields that if a certain parameter of the input instance is limited, optimization of that instance is possible in polynomial time.

This statement closely corresponds to one possible definition of the concept of *fixed-parameter tractability*. Intuitively, a problem being fixed-parameter tractable means that the instances of the problem have a certain parameter, and that the value of that parameter dominates the run time complexity.

In our case, the parameter is a measure of the complexity of the instance at every point in time — a local complexity, in a manner of speaking. Thus, the analysis allows us to draw the conclusion: The complexity of TCPSP is dominated by the maximal local complexity, not by the instance size itself. This result indicates that even very large instances with thousands of jobs might be solvable if one can manage to limit local complexity.

For a general overview over project scheduling techniques, we refer the reader to the survey by Węglarz [Węg99], which gives a comprehensive overview over applications and optimization techniques. Regarding fixed parameter tractability of

the problem, Yaw and Mumey [YM17] present a similar result based on a branch-and-bound algorithm. However, not only is our presented algorithm (and subsequent analysis) simpler, but it is also able to deal with dependencies between jobs. For machine scheduling models, Mnich and Wiese [MW15] present a list of fixed-parameter tractability results.

**Overview.** We start this chapter by introducing necessary notation and concepts in Section 11.2. Section 11.3 introduces *local configurations*, which are the key concept that the exact algorithm from Section 11.4 builds upon. In Section 11.5, we analyze the complexity of this algorithm and show that it is in fact fixed-parameter tractable.

## 11.2 Preliminaries

In this section, we introduce several concepts necessary for the analysis of TCPSP, as well as the formal definition of the problem we concern ourselves with. For basic notation surrounding computational complexity, we refer the reader to the definitive book by Garey and Johnson [GJ79].

### 11.2.1 Pseudo Fixed-Parameter-Tractability

The main contribution of this chapter is placing an upper bound on the run time complexity of optimizing TCPSP instances in terms of a parameter of the instances. For this, we need the concepts of *fixed-parameter tractability* and *pseudo-polynomiality*, which we combine to *pseudo fixed-parameter-tractability*.

fixed-parameter tractable

A problem is said to be *fixed-parameter tractable* in a parameter $k$, if the time needed to solve instance $I$ (which includes some parameter $k$) can be upper-bounded by a function of the form $f(k)\cdot O(poly(|I|))$. Here, $f$ can be any arbitrary function, especially an exponential function, while *poly* can be any polynomial function. Note that this means that the run time is polynomial in the instance size.

Another concept commonly used to specify run time complexities is that of *pseudo-polynomiality*. One usually talks of a problem being "polynomial" (or being in $\mathcal{P}$), if instances $I$ of the problem can be solved (on a deterministic Turing machine) in a time bounded in $O(poly(|I|))$. Here $|I|$ means the length of the encoded instance written on the Turing machine's tape. While the exact alphabet used to encode the instance (or the coding scheme) is not important, there is one crucial catch: Does the alphabet have more than one element? An alphabet with at least two elements allows one to encode a number $n$ in at most $\log_2(n)$ digits, while an alphabet of size 1 enforces a coding length of $n$. With this in mind, a problem is said to be *pseudo-polynomial* if the run time to optimize an instance $I$ can be bounded by $O(poly(|I|_1))$, where $|I|_1$ means the length of a unary encoding of $I$. We combine these two concepts to form pseudo fixed-parameter-tractability:

pseudo-polynomial

**Definition 7** (Pseudo Fixed-Parameter-Tractability)**.** *We say a problem is pseudo fixed-parameter-tractable with respect to a parameter $k$ if any instance $I$ (with parameter $k$) of the problem can be solved in running time bounded by $f(k) \cdot O(poly(|I|_1))$.*

pseudo fixed-parameter-tractable

### 11.2.2 Problem Definition

This work considers project scheduling problems that are variants of the common Time-Constrained Project Scheduling Problem (TCPSP). Since the machine scheduling problem known as Scheduling with Release Times and Deadlines on a Minimum Number of Machines (SRDM) is a special case of TCPSP, and in fact our variations of the TCPSP are compatible with SRDM, our results are also applicable to SRDM. We now formally define the problems and notation that we use throughout this chapter.

An *instance* of TCPSP consists of a set $J$ of *jobs*. We assume $n$ to be the number of jobs, i.e., $n = |J|$. Also, we assume $k$ resources to be given. In the job set $J = \{j_1, j_2, \dots j_n\}$, each job $j_i$ is a four-tuple: $j_i = (r_i, d_i, p_i, u_i) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{R}^k$. For job $j_i$, $r_i$ states the *release time* of $j_i$, i.e., the earliest time at which $j_i$ can be executed. In turn, $d_i$ states the *deadline*, i.e., the time at which $j_i$ must be finished. The *processing time* $p_i$ indicates how long $j_i$ must be executed without interruption. Finally, $u_i$ specifies the *usage* of $j_i$, i.e., how much resources $j_i$ requires. For job $i$, $u_{i,k}$ specifies the usage of resource $k$.

instance
jobs
release time
deadline
processing time
usage

Dependencies between jobs are captured in a *lag* (partial) function $L\colon \mathbb{N}^2 \nrightarrow \mathbb{N}_0$. For two jobs $i, j$, an entry $L(i, j)$ specifies the minimal amount of time that must pass after the start of $i$ before $j$ can start. If $(i, j) \notin L$, then there is no such dependency between $i$ and $j$.

lag

The objective is to assign start times to every job $j$ that respect $r_j$ and $d_j$ and any potential $L(\cdot, j)$ resp. $L(j, \cdot)$. We call such a set of start times a *schedule*. More formally, a schedule is a function $S\colon \{1, \dots n\} \to \mathbb{N}_0$, where $S(j)$ is the start time of job $j$. A schedule $S$ is *feasible* if and only if for all j, $S(j) \geq r_j$, $S(j) \leq d_j - p_j$ and $S(j) \geq S(i) + L(i, j)$ for all $(i, j) \in L$. The set of all feasible schedules is denoted as $\mathcal{S}$.

schedule

Finally, an objective function $Obj\colon \mathcal{S} \to \mathbb{R}$ is given, assigning a cost to any feasible schedule. The objective is to determine start times that minimize $Obj(S)$. We do not allow arbitrary objective functions. We defer the formal specification of the requirements for valid objective functions to Section 11.3.2, since more concepts need to be established first. On a high level, we require our objective function to be computable by iterating the schedule from left to right, with only a constant amount of memory. In Section 11.6, we show how two popular objective functions can be integrated into our approach.

In [HDD99], Herroelen et al. propose a classification scheme for project scheduling problems. If the objective function is chosen to reflect peak shaving, the characterization for the problem defined here according to that scheme is $va|min, \rho_j, \delta_j|av$.

However, many different variants can also be optimized by the presented approach. See Section 11.6 for details on how other objective functions can be optimized.

## 11.3 Local Configurations

local configurations

After introducing the necessary concepts, we now introduce *local configurations*, which are the foundation of this work.

Intuitively, a local configuration at time $t$ specifies for each job that could potentially be executing at time $t$ whether it has already finished, or else whether and when it was started. We will use it in Algorithm 8 to construct a schedule by iterating over all time points. The idea is that given a local configuration at time $t$, one can determine how the future of the schedule can look like. To do that, we need to know the start time of a job $j$ in the local configuration at $t$ under several circumstances:

- If $j$ is running during time $t$, we must know when it stops running.
- If $j$ has a dependee $k$ with time lag $l$ (i.e., $L(j,k) = l$), and $t - s_j < l$, we need to know that we cannot start $k$ yet.

Under all other circumstances, we just need to know whether $j$ has already completed (and all its dependees can be started already), or whether it has not been started yet.

max lag

We define our local configuration to reflect this information. We define a job $j$'s *max lag* $\hat{L}(j) = \max\{l \mid \exists k \in J\colon L(j,k) = l\} \cup \{0\}$ as the maximum over all its outgoing lags.

Then, given an instance of TCPSP and a schedule $S$ we define the *local configuration* at time step $t$, written $\mathrm{LC}_S(t)$, as

$$\mathrm{LC}_S(t) = \{\,(i, S(i)) \quad : S(i) \le t \wedge t - S(i) < \max\{p_i, \hat{L}(i)\}\} \tag{11.1}$$

$$\cup\ \{\,(i, \dagger) \quad : S(i) \le t \wedge t - S(i) \ge \max\{p_i, \hat{L}(i)\}\} \tag{11.2}$$

In this definition, (11.1) causes $(j, S(j))$ to be part of the local configuration for all jobs $j$ that were already started ($S(j) \le t$), and that are either not yet finished ($t - S(j) < p_j$) or that have dependees that can not be started yet because of them ($t - S(j) < \hat{L}(j)$). For other already started jobs $j$, (11.2) captures the fact that job $j$ is already finished (and all dependees can be started) as $(j, \dagger)$.

local configuration set

Building upon this, we define the *local configuration set* at a time step $t$, the set of all local configurations for all valid schedules. The *local configuration set* at time step $t$ is the set of all feasible local configurations at time step $t$, i.e.

$$\mathrm{LCS}(t) = \{\mathrm{LC}_S(t) : S \in \mathcal{S}\}$$

local configuration set size

We call its size the *local configuration set size* at $t$, denoted as $\mathrm{LCSS}(t) = |\mathrm{LCS}(t)|$.

We now determine an upper bound on LCSS($t$). Given a job $j$, which $(j, \cdot)$ pairs can appear in LCS($t$)? Certainly, this set is at most $\{(j, \dagger), (j, t - \max\{p_j, \hat{L}(j)\} + 1), (j, t - \max\{p_j, \hat{L}(j)\} + 2), \ldots (j, t)\}$, which is of size $\max\{p_j, \hat{L}(j)\}$. If $j$ was started more than $\max\{p_j, \hat{L}(j)\}$ time steps ago, its exact start time is irrelevant and it would be recorded as $(j, \dagger)$. The product of the sizes of the possible entries for all jobs yields an upper bound on LCSS($t$):

$$\text{LCSS}(t) \leq \prod_{j=1}^{n} \left( \max \left\{ p_j, \hat{L}(j) \right\} \right)$$

However, at point $t$, usually only a subset of jobs is relevant at all. For example any job $j$ with $r_j > t$ cannot have any entry in $\text{LC}_S(t)$. Similarly, any job $j$ with $d_j - p_j + \max\{p_j, \hat{L}(j)\} \leq t$ can only appear as $(j, \dagger)$ in $\text{LC}_S(t)$. Such a job $j$ has not only inevitably stopped running at $t$ (because of $d_j - p_j + p_j = d_j \leq t$), but also all of its dependees can be started (because of $d_j - p_j + \hat{L}(j) \leq t$; note that $d_j - p_j$ is the latest possible start time of $j$). This means that the information about when $j$ was started is irrelevant for the further schedule. Since these jobs contribute only a 1 to the product, we may discard them. We call the right-open interval $[r_j, d_j - p_j + \max\{p_j, \hat{L}(j)\})$ the *extended window* of job $j$. To place an upper bound on LCSS($t$), it suffices to consider jobs which have $t$ in their extended window because of the arguments laid out above. This allows us to reformulate the upper bound on LCSS($t$):

extended window

$$\text{LCSS}(t) \leq \prod_{i \in \left\{ j \in J \mid r_j \leq t < d_j - p_j + \max\left\{ p_j, \hat{L}(j) \right\} \right\}} \max \left\{ p_i, \hat{L}(i) \right\}$$

We call the maximum LCSS($t$) the *maximum local configuration set size*, denoted as MLCSS $= \max_t \text{LCSS}(t)$. This MLCSS captures the maximum local complexity of an instance. It is the parameter in which we show the fixed-parameter tractability of TCPSP.

maximum local configuration set size

Formally, we state as the main result of this chapter:

**Theorem 11.1** (TCPSP is Pseudo-FPT)**.** *Given an instance $\mathcal{I}$ of TCPSP, an optimal schedule can be computed in*

$$\text{MLCSS}^2 \cdot \mathcal{O}(|\mathcal{I}|_1^2)$$

*time, where $|\mathcal{I}|_1$ represents the length of an unary encoding of $\mathcal{I}$.*

### 11.3.1 Configuration Continuation

Working towards an algorithm to solve TCPSP, we now present a way of determining how a given local configuration can be completed to a feasible schedule.

Given a local configuration $l_a$ at $t$ and a local configuration $l_b$ at $t+1$, we say that $l_a$ can be *continued* to $l_b$ if and only if there is a schedule $S$ such that $\mathrm{LC}_S(t) = l_a$ and $\mathrm{LC}_S(t+1) = l_b$. Intuitively, this means that $l_a$ can be a predecessor of $l_b$ in a valid schedule. Instead of enumerating all possible schedules to check whether a given $l_b$ is a continuation of $l_a$, we can check this efficiently. We must check three things:

- Jobs the start time of which becomes irrelevant at time $t+1$ are correctly marked: For every $(j, t_j)$ in $l_a$ with $t_j + \max\{p_j, \hat{L}(j)\} = t+1$, $l_b$ must contain $(j, \dagger)$,
- all other items in $l_a$ are also in $l_b$,
- jobs that need to start are started: For every $i$ such that there is no $(i, \cdot)$ in $l_a$, and for which $d_i - p_i = t+1$, the item $(i, t+1)$ must be in $l_b$.

successor local configuration set

We call the set of continuations of a local configuration $l_a$ its *Successor Local Configuration Set*, written as $\mathrm{SLCS}(l_a)$.

We can use the characterization above to easily compute the successor local configuration set $\mathrm{SLCS}(l)$ for a local configuration $l$, as shown in Algorithm 7. Starting with an old local configuration $l$ at time $t$, lines 1 to 7 mark the jobs from $l$ the start time of which becomes irrelevant at $t+1$ as such. Lines 8 to 9 handle jobs that are not started in $l$ and for which $t+1$ is the latest possible start time. Line 10 generates the set of jobs that could potentially start at $t+1$, and lines 11 to 12 create all possible subsets of these.

### 11.3.2 Extensible Cost Functions

The algorithm presented in this chapter is able to solve several variants of the the TCPSP problem defined in Section 11.2.2. Which objective is optimized for for is determined by choosing the right objective function, of which we show examples in Section 11.6.

extensible cost function

During the execution of the algorithm, we replace the chosen objective function with a modified objective function, which we call an *extensible cost function*. Section 11.6 shows examples for such functions as well.

Note that we call it extensible *cost* function, not extensible *objective* function. In our terminology, costs do not only contain the objective value, but might also contain some auxiliary data needed for future computations. Intuitively, the extensible cost function should compute the objective value of a schedule if it is iteratively applied to all the local configurations of the schedule, left to right. At every local configuration, it is only allowed to utilize a fixed amount of information (the auxiliary data) from its earlier runs.

schedule prefix

To formally define extensible cost functions, we need the concept of a *schedule prefix*. Given a schedule, we can look at the schedule only up to a certain point and

**Algorithm 7:** Compute the Successor Local Configuration Set for a local configuration $l$.

**Input:** $t$ : previous time step
**Input:** $l$ : Local configuration at $t$
**Output:** The successor local configuration set of $l$

$Result \leftarrow \emptyset$;
$l' \leftarrow \emptyset$;
**for** $(i, t_i) \in l$ **do**
  // Mark irrelevant jobs as irrelevant
  **if** $t_i = t - \max\left\{p_i, \hat{L}(i)\right\}$ **then**
    $l' \leftarrow l' \cup \{(i, \dagger)\}$;
  **else**
    $l' \leftarrow l' \cup \{(i, t_i)\}$;
/* Start unstarted jobs for which this is the last possible start time */
**for** $i \in \{j \in J \mid d_j - p_j = t + 1 \wedge (j, \cdot) \notin l\}$ **do**
  $l' \leftarrow l' \cup \{(i, t + 1)\}$;
$StartCandidates \leftarrow \{j \in J \mid r_j \leq t + 1 \wedge (j, \cdot) \notin l'\}$;
**for** $c \in \{C \mid C \subseteq StartCandidates\}$ **do**
  $Result \leftarrow Result \cup \{l' \cup \{(i, t + 1) \mid i \in c\}\}$;

forget anything happening after that point. Formally, let $S$ be a schedule and $S[t]$ the schedule prefix of $S$ up to time point $t$. Then:

$$S[t]\colon \{i \mid S(i) \leq t\} \to \mathbb{N}_0$$
$$S[t](i) = S(i)$$

Note that $S[t]$ is most likely not a valid schedule, since it does not assign a start time to every job.

Formally, an extensible cost function is a function

$$c\colon \left\{S[t] \mid S \in \mathcal{S}, t \in \left\{\min_i r_i, \ldots \max_i d_i\right\}\right\} \times Q \to \mathbb{R} \times Q$$

where $Q$ is the set of all possible auxiliary information it can carry over. The first set is the set of all possible schedule prefixes of all feasible schedules. We denote the second component of $c(\cdot, \cdot)$ as $c(\cdot, \cdot)_2$. We treat the first component (denoted $c(\cdot, \cdot)_1$) of the output as a preliminary objective value. We require $c$ to be defined in such a way that for a schedule $S$ and a latest deadline $D = \max_i d_i$,

$$c(S[D], c(S[D-1], c(S[D-2], \ldots)_2)_2)_1$$

results in the objective value of $S$.

Intuitively, a cost function being extensible means that it can be used to compute the objective value of a schedule by looking at the schedule one time step (and local configuration at that time step) at a time. We also require an extensible cost function to be computable with time complexity $O(n)$. Note that this also means that the output, which includes a representation of the elements of $Q$, must be bounded by $O(n)$. Thus, $Q$ cannot be arbitrarily large.

## 11.4 An Exact Algorithm

We assume to be given an instance of TCPSP as defined in Section 11.2.2 with an objective function that can be expressed as an extensible cost function. We present an algorithm that iterates all time steps and, for each time step $t$, computes preliminary costs (using the extensible cost function) for each possible local configuration in LCS($t$). Let $D$ be the global deadline of the instance, i.e., $D = \max_i(d_i)$.

The algorithm (shown as pseudocode in Algorithm 8) builds a table $T$ of dimensions $D \times$ MLCSS. We denote by $T[a][b]$ the entry in row $a$, column $b$. We assume the rows to be indexed by the integers $\{1, \ldots, D$. Abusing notation, we assume the columns of $T[a]$ to be indexed by local configurations. Of course, the feasible local configurations for time steps $a$ and $b$ differ, thus this is not a "real" index. However, we know that there are at most MLCSS local configurations at every time step. We assume these to be somehow mapped to the set of integers $\{1, \ldots \text{MLCSS}\}$, thus resulting in a "valid" table index.

The algorithm works by iterating over all time steps in the loop started in line 2. For each time step $t$, it enumerates all local configurations at $t-1$ in line 3, generating each successor local configuration set in line 4. This yields all possible local configurations at time $t$. Note that the same local configuration at $t$ can be generated in multiple successor local configuration sets for different local configurations at time $t-1$. Thus, we must for each generated local configuration (line 5) not only compute its preliminary cost (line 6), but also check whether this is the best value we have so far seen for this specific local configuration (lines 7–8).

After the algorithm has finished, we can easily determine the objective value of an optimum solution by extending the table built in Algorithm 8 by one more time step. In this time step $D+1$, all jobs will have finished (or the input instance is infeasible). Thus, $T[D+1]$ will have exactly one local configuration, namely $\{(i, \dagger) \colon i \in \{1, \ldots n\}\}$. The value of that entry is the optimal solution quality of the input instance.

### 11.4.1 Schedule Reconstruction

To actually reconstruct the schedule, we need only to modify what we store in $T[t][\cdot]$: Instead of only storing the optimal partial costs, we also store which local configuration

**Algorithm 8:** An exponential-time algorithm to solve the TCPSP

**Input:** An instance of TCPSP
**Input:** $c$: An extensible cost function

```
1 Enumerate LCS(1) and compute costs for each schedule prefix in LCS(1);
2 for t ← 2 to D do
3 |   for l ∈ LCS(t − 1) do
4 |   |   SuccessorSet ← computeSLCS(l, t);
5 |   |   for l′ ∈ SuccessorSet do
6 |   |   |   cost ← c(l′, T[t − 1][l]);
7 |   |   |   if l′ ∉ T[t] or cost_1 < T[t][l′] then
8 |   |   |   |   T[t][l′] = c;
```

in $T[t-1]$ was continued to achieve this value. Using this information, we can later trace back a path from $T[D+1]$ to $T[1]$ during which we will see each job together with its start time at least once.

## 11.5 Complexity

We first analyze the complexity of Algorithm 7, which is being used in the TCPSP Solver (Algorithm 8). The loop in lines 3–7 is executed at most $n$ times, and takes $O(1)$ time per iteration. Similarly, the loop in lines 8–9 can be executed in $O(n)$ time. The set of jobs not started yet in line 10 can also be computed in $O(n)$ time. Interesting is the question how often the loop in line 11 executes. We know that in the end, every element in *Result* is a member of LCS$(t+1)$. We also know that every iteration of the loop adds exactly one element to *Result*, and that the size of LCS$(t+1)$ is bounded by MLCSS. Thus, the loop can be executed at most MLCSS times. The actual generation of an element of *Result* in line 12 can be done in $O(n)$ time. Thus, the loop in line 11 takes a total of MLCSS $\cdot\, O(n)$ time, which also dominates the run time of Algorithm 7.

We now analyze the complexity of Algorithm 8. Since our objective function is required to be an extensible cost function, and we may assume extensible cost functions to have time complexity $O(n)$ as per Section 11.3.2, lines 6 to 8 have time complexity $O(n)$. Since the size of the successor set is bounded by MLCSS, the loop in line 5 requires time in MLCSS $\cdot\, O(n)$. The same holds for line 4. Since the size of LCS$(t-1)$ is also bounded by MLCSS, we know that the loop in line 3 requires a total time in MLCSS$^2 \cdot O(n)$. Finally, the outer loop in line 2 executes once for every time step, leading to a total time complexity of MLCSS$^2 \cdot O(nD)$. Note that $D$ is not a size parameter of the input instance, but rather an input value — thus, $O(nD)$ is not polynomial in the size of the input, and thus MLCSS$^2 \cdot O(nD)$ does not fulfill the requirement for fixed-parameter tractability as defined in Section 11.2.1. However, the concept of pseudo-polynomiality

as defined in in the same section relates the run time complexity to the length of an unary encoding of the instance, $|\mathcal{I}|_1$, and clearly $\mathcal{O}(nD) \subseteq \mathcal{O}(n \cdot |\mathcal{I}_1|) \subseteq \mathcal{O}(|\mathcal{I}|_1^2)$. Thus we arrive at the time complexity from Theorem 11.1: $\text{MLCSS}^2 \cdot \mathcal{O}(|\mathcal{I}|_1^2)$. With this, upper-bounding MLCSS by a constant value makes Algorithm 8 pseudo-polynomial. We have therefore shown that Algorithm 8 is pseudo fixed-parameter-tractable in the parameter MLCSS.

## 11.6 Objective Functions

We have shown that the algorithm in Section 11.4 solves the Time-Constrained Project Scheduling Problem with any objective function that can be expressed in terms of an extensible cost function. We now first show how to express classical peak shaving and resource leveling problems as such an extensible cost function.

An extensible objective function receives two inputs: A local configuration $l$ at $t$, and the second component of the result of the extensible cost function on the local configuration that was the predecessor of $l$ — see Section 11.3.2 for a formal definition. It must produce the partial costs for $l$.

**Peak Shaving.** In a peak shaving setting (which in literature is also known as the Resource Acquirement Cost Problem), the objective is to minimize the maximum resource usage for each resource. Since we are not considering multiobjective optimization here, some weighting factor is given for every resource to combine the maximum usages for all resources into a single objective. Let $w_\rho$ be the weight factor for resource $\rho$, and let $R(S, \rho, t)$ be the usage of resource $\rho$ at time $t$ in schedule $S$, then a peak shaving objective can look like

$$Obj(S) = \sum_{\rho \in \{1,\dots k\}} w_\rho \cdot \max_t \left(R(S, \rho, t)\right)$$

The extensible cost function suggested to optimize for this objective does return as auxiliary data the maximum value seen for every of the $k$ resources up to the current point. Thus, $Q = \mathbb{R}^k$ is the set of possible auxiliary information. Note that in Section 11.3.2, we stated that $Q$'s size must be bounded in a certain way, which obviously is not the case here. However, only a distinct set of values, of size $2^n$, can be taken for the demand of each resource, because each job can either contribute or not. This is sufficient to find a small enough representation for the values of $Q$. We treat an element of this $Q$ as a vector and for $q \in Q$ denote by $q[i]$ the $i$'th element of the vector.

To formally define the extensible cost function, we need a function that computes the resource usages for a local configuration. Let $l$ be a local configuration at time $t$,

then $U(l,t)_\rho$ computes the usage of resource $\rho$ at time $t$ in a schedule corresponding to $l$:

$$U(l,t)_\rho = \sum_{\substack{i \in \{j \mid (j,t_j) \in l \\ \wedge\, t_j \neq \dagger \\ \wedge\, t_j + p_j > t\}}} u_{i,\rho}$$

The sum sums over all jobs that are executing during $l$, i.e., that have an entry in $l$ but are not yet finished. With this, the extensible cost function to achieve peak shaving is:

$$c(l,q) = \left( \max\left(q_1, \sum_{\rho=1}^{k} w_\rho \cdot \max\left(q_2[\rho], U(l,t)_\rho\right)\right), \begin{bmatrix} \max\left(q_2[1], U(l,t)_1\right) \\ \max\left(q_2[2], U(l,t)_2\right) \\ \vdots \\ \max\left(q_2[k], U(l,t)_k\right) \end{bmatrix} \right)$$

The first component is the maximum of the previous preliminary cost and the weighted sum over all so-far seen resource demand maxima. The second component is the updated vector of so-far seen resource demand maxima.

**Resource Leveling.** In a resource leveling problem, the objective function aims to minimize at every time step and for every resource the deviation of the usage of the respective resource from a predefined desired value. If only positive deviations are to be minimized, the desired value can be interpreted as an amount of a resource that is available for free at every time step, while all usage above that level must be paid for. Problems where positive as well as negative deviations are to be minimized arise for example in settings where machines must be always run close to a certain operating point for technical reasons.

Again, since we do not consider multiobjective optimization, we assume a weight factor $w_\rho$ to be given for every resource $\rho$. If $A(\rho,t)$ is the desired usage for resource $\rho$ at time step $t$, the objective can be written as

$$Obj(S) = \sum_{\rho=1}^{k} \sum_{t=1}^{D} w_\rho \cdot |R(S,\rho,t) - A(\rho,t)|$$

In this case, the only value needed from time step $t-1$ to compute the preliminary costs at $t$ are the preliminary costs itself, no auxiliary information is needed. We therefore

formally set $Q = \{\perp\}$, in which we treat $\perp$ as a dummy element. This extensible cost function is significantly easier:

$$c(l, q) = \left( q_1 + \sum_{\rho=1}^{k} w_\rho \cdot |U(l)_\rho - A_\rho| \, , \; \perp \right)$$

## 11.7 Conclusion

In the present chapter we have presented an analysis of the complexity of the Time-Constrained Project Scheduling Problem. Using an exponential-time algorithm that optimizes TCPSP in various flavors, e.g., the Resource Acquirement Cost Problem or the Resource Leveling Problem, we have shown these problems to be pseudo fixed-parameter-tractable in a parameter that measures local complexity. The result indicates that optimization even of large instances might be feasible if one is able to bound local complexity.

It is conceivable that the presented algorithm can be adapted to form a heuristic by generating only a subset of the possible local configurations at every time step. It would be interesting to see if this yields a useful algorithm. Another interesting question touching practical applicability is whether real-world instances, for example from smart grid scheduling problems as introduced in Part I, have high or low local complexity.

# 12 Conclusion

In this thesis, we investigated ways to exploit demand side flexibility in electrical grids with a strong focus on centralized demand response. As outlined in Chapter 1, demand response can be an important puzzle piece in the endeavor to ensure uninterrupted service and efficient use of generation assets in changing energy systems. We concentrated our research on algorithmic approaches to facilitate demand response.

**Summary.** In Part I, we focused on modeling demand side flexibility and performing optimization based on these models. We started by presenting a comprehensive modeling framework in Chapter 3, which unifies several models from literature and we practically evaluated the complexity of optimizing the resulting models. Taking it one step further, our interaction with the energy community lead us to believe that models requiring the specification of flexibility on a per-job basis are not very applicable in practice, since that information is not readily available even to operators of industrial plants. Therefore, we presented a new modeling technique in Chapter 4. The new modeling technique needs only a global limit on flexibility, and its result can be interpreted as a recommendation as to which processes might benefit the most from flexibilization. The mixed-integer linear models used throughout this part are only useful if they can be solved in a reasonable amount of time. Thus, we reworked parts of our models to make them more efficient in Chapter 5. We presented and evaluated a new order-based modeling technique. Comparison to two existing techniques showed the order-based approach to be superior. We complemented this first part in Section 6 by presenting a data set of benchmark scheduling instances derived from real-world production data, which can be used to evaluate scheduling approaches.

Since solving mixed-integer linear programs — or any other $\mathcal{NP}$-hard problem — is feasible only up to a size which does not cover realistic instance sizes of e.g. large industrial plants, heuristics must be developed to cope with larger instances. In Part II, we presented two such heuristics. The first heuristic, RUSH, presented in Chapter 7, deals with a relatively simple model. The heuristic is aimed at finding good solutions fast by employing a rapid iterative improvement process. In Chapter 8, the second heuristic, SWAG, uses a more elaborate approach of interpreting the problem as a graph problem, using graph augmentation as the scheduling mechanism. We evaluated both heuristics, both against mixed-integer linear programs and competitor heuristics. The results show that both heuristics are feasible approaches to large-scale scheduling instances, and that SWAG outperforms a competitor heuristic on instances derived from real-world data.

In Part III, we examined algorithmic foundations of scheduling, especially peak shaving. An algorithm can only be as fast as the basic operations it uses. Peak shaving heuristics such as SWAG commonly require quickly finding the peak demand in a current schedule. In Chapter 9, we presented a variant of dynamic segment trees that allow for very efficient lookup of the peak in a schedule. We engineered the dynamic segment trees by replacing the underlying red-black tree with a zip tree or a weight-balanced tree. The weight-balanced trees used can in turn profit from further engineering, which we presented in Chapter 10. We conducted an in-depth evaluation of top-down rebalancing strategies with various rebalancing parameters, and uncovered that parameter choices that are not advisable from a theoretic standpoint do in many cases exhibit the best performance in practice. Additionally, we presented a complexity analysis of the Time-Constrained Project Scheduling Problem by means of an exponential-time enumerative algorithm. The analysis shows that TCPSP becomes pseudo-polynomial if a certain local measure of complexity is bounded, making the problem pseudo-fixed-parameter-tractable.

**Conclusion.** The aim of this thesis was to provide insights into how the scheduling problems behind demand response can be facilitated to aid the Energiewende, and to investigate the algorithmic challenges presenting themselves in this endeavor. The scheduling approaches we presented in chapters 3, 4, 5, 7 and 8 each solve a variant (or several variants) of the problems associated with demand response. We have thoroughly evaluated each, in most cases based on data derived from real-world applications, and shown them each to be a well-suited technique in at least some cases. The additional algorithmic results presented in Part III can hopefully aid the algorithm engineering community in building faster algorithms for a multitude of problems. Summarizing this, it seems fair to say this thesis lives up to its original goals.

**Outlook.** One aspect that makes smart grid scheduling an exciting subject is that the well-researched field of scheduling algorithms meets the emerging very active, sometimes even chaotic, field of energy informatics. Thus, going forward will require a lot of work consolidating and summarizing the many papers in the field of energy informatics and bringing them together with the known results from scheduling. A more precise mathematical modeling of demand response potentials appearing in real-world applications would be particularly helpful. This brings us to another important point: field studies of demand response in real industrial contexts would be very helpful to validate the work being done. Ultimately, bridging the gap between the many clever algorithms being written on paper and real electricity consumers actually participating in demand response — preferably many and large ones — may well be the most important, but also the most demanding challenge.

# Bibliography


[AB00] Sankar Ashok and Rangan Banerjee. **Load-management applications for the industrial sector**. *Applied Energy* 66:2 (2000), pages 105–111. DOI: 10.1016/s0306-2619(99)00125-7.
Cited on pages 6, 20, 21.

[Ada93] Stephen Adams. **Functional Pearls Efficient sets — a balancing act**. en. *Journal of Functional Programming* 3:4 (1993), pages 553–561. DOI: 10.1017/S0956796800000885. (Visited on 08/16/2019).
Cited on page 158.

[Ali+15] Mahnoosh Alizadeh, Anna Scaglione, Andy Applebaum, George Kesidis, and Karl Levitt. **Reduced-Order Load Models for Large Populations of Flexible Appliances**. *IEEE Transactions on Power Systems* 30:4 (2015), pages 1758–1774. DOI: 10.1109/tpwrs.2014.2354345.
Cited on page 19.

[All+12] Florian Allerding, Marc Premm, Pradyumn Kumar Shukla, and Hartmut Schmeck. **Electrical load management in smart homes using evolutionary algorithms**. In *Proceedings of the 9th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP'12)*. Springer, 2012, pages 99–110. DOI: 10.1007/978-3-642-29124-1_9.
Cited on pages 6, 19, 21.

[Amb+18] Mirjam Ambrosius, Veronika Grimm, Christian Sölch, and Gregor Zöttl. **Investment incentives for flexible demand options under different market designs**. *Energy Policy* 118 (2018), pages 372–389. DOI: 10.1016/j.enpol.2018.01.059.
Cited on page 39.

[AMR03] Christian Artigues, Philippe Michelon, and Stéphane Reusser. **Insertion techniques for static and dynamic resource-constrained project scheduling**. *European Journal of Operational Research* 149 (2 2003), pages 249–267. DOI: 10.1016/S0377-2217(02)00758-0.
Cited on pages 5, 63.

[AMS14] Florian Allerding, Ingo Mauser, and Hartmut Schmeck. **Customizable energy management in smart buildings using evolutionary algorithms**. In *Proceedings of the 5th European Conference on the Applications of Evolutionary Computation (EvoApplications'14)*. Springer, 2014, pages 153–164. DOI: 10.1007/978-3-662-45523-4_13.
Cited on page 100.

[Ard+17] Omid Ardakanian, Ye Yuan, Roel Dobbe, Alexandra von Meier, Steven Low, and Claire Tomlin. **Event detection and localization in distribution grids with phasor measurement units**. In *Proceedings of the 15th IEEE Power & Energy Society General Meeting (PESGM'17)*. IEEE, 2017, pages 1–5. DOI: 10.1109/pesgm.2017.8273895.
Cited on page 39.

[Ash06] Sankar Ashok. **Peak-load management in steel plants**. *Applied Energy* 83:5 (2006), pages 413–424. DOI: 10.1016/j.apenergy.2005.05.002.
Cited on pages 6, 20, 21, 100.

[Ass17] Solar Energy Industries Association. **Solar Market Insight Report 2017 Year in Review**. 2017. URL: https://www.seia.org/research-resources/solar-market-insight-report-2017-year-review (visited on 02/17/2020).
Cited on page 2.

[Bal07] Francisco Ballestín. **A genetic algorithm for the resource renting problem with minimum and maximum time lags**. In *Proceedings of the 4th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP'07)*. Springer, 2007, pages 25–35. DOI: 10.1007/978-3-540-71615-0_3.
Cited on page 5.

[Bar+15] Antimo Barbato, Antonio Capone, Lin Chen, Fabio Martignon, and Stefano Paris. **A distributed demand-side management framework for the smart grid**. *Computer Communications* 57 (2015), pages 13–24. DOI: 10.1016/j.comcom.2014.11.001.
Cited on page 4.

[Bar+18a] Lukas Barth, Veit Hagenmeyer, Nicole Ludwig, and Dorothea Wagner. **Dataset accompanying "How much demand side flexibility do we need? - Analyzing where to exploit flexibility in industrial processes"**. KITOpen Repository. 2018. DOI: 10.5445/IR/1000082194.
Cited on page 215.

[Bar+18b] Lukas Barth, Veit Hagenmeyer, Nicole Ludwig, and Dorothea Wagner. **How much demand side flexibility do we need?** In *Proceedings of the 9th International Conference on Future Energy Systems (e-Energy'18)*. ACM, 2018, pages 43–62. DOI: 10.1145/3208903.3208909.
Cited on page 35.

[Bar+18c] Lukas Barth, Nicole Ludwig, Esther Mengelkamp, and Philipp Staudt. **A comprehensive modelling framework for demand side flexibility in smart grids**. In *Computer Science — Research and Development*. Volume 33. 1-2. Springer, 2018, pages 13–23. DOI: 10.1007/s00450-017-0343-x.
Cited on page 17.

[BDF17] Nina Boogen, Souvik Datta, and Massimo Filippini. **Demand-side management by electric utilities in Switzerland: Analyzing its impact on residential electricity demand**. *Energy Economics* 64 (2017), pages 402–414. DOI: 10.1016/j.eneco.2017.04.006.
Cited on page 79.

[Ben77] Jon Louis Bentley. **Algorithms for Klee's rectangle problems**. Tech. rep. Unpublished notes. Department of Computer Science, Carnegie-Mellon University, 1977.
Cited on page 135.

[Ber+08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. **Computational Geometry: Algorithms and Applications**. Springer, 2008. ISBN: 978-3-540-77973-5. DOI: 10.1007/978-3-540-77974-2.
Cited on page 138.

[Bis+18] Simon Bischof, Holger Trittenbach, Michael Vollmer, Dominik Werle, Thomas Blank, and Klemens Böhm. **HIPE: An Energy-Status-Data Set from Industrial Production**. In *Proceedings of the 9th International Conference on Future Energy Systems (e-Energy'18)*. e-Energy '18. Karlsruhe, Germany: ACM, 2018, pages 599–603. DOI: 10.1145/3208903.3210278.
Cited on pages 40, 80, 85, 124, 215.

[BK12] Peter Brucker and Sigrid Knust. **Complex Scheduling**. Second edition. Springer, 2012. ISBN: 978-3-642-23928-1. DOI: 10.1007/978-3-642-23929-8.
Cited on pages vii, 5, 12, 115.

[BKF15] Zdenek Bradac, Vaclav Kaczmarczyk, and Petr Fiedler. **Optimal scheduling of domestic appliances via MILP**. *Energies* 8:1 (2015), pages 217–232. DOI: 10.3390/en8010217.
Cited on page 6.

[BM80] Norbert Blum and Kurt Mehlhorn. **On the Average Number of Rebalancing Operations in Weight-Balanced Trees**. *Theoretical Computer Science* 11 (1980), pages 303–320. DOI: 10.1016/0304-3975(80)90018-3.
Cited on pages 158, 159, 164.

[Bun19] Bundesnetzagentur. *Bericht über die Mindesterzeugung 2019*. Oct. 7, 2019. URL: https://www.bundesnetzagentur.de/DE/Sachgebiete/ElektrizitaetundGas/Unternehmen_Institutionen/Versorgungssicherheit/Erzeugungskapazitaeten/Mindesterzeugung/Mindesterzeugung_node.html (visited on 02/17/2020).
Cited on page 3.

[BW18] Lukas Barth and Dorothea Wagner. **Exploiting flexibility in smart grids at scale**. In *Computer Science — Research and Development*. Volume 33. 1-2. Springer, 2018, pages 185–191. DOI: 10.1007/s00450-017-0357-4.
Cited on page 99.

[BW19a] Lukas Barth and Dorothea Wagner. **Dataset accompanying "Shaving Peaks by Augmenting the Dependency Graph"**. KITOpen Repository. 2019. DOI: 10.5445/IR/1000094106.
Cited on page 123.

[BW19b] Lukas Barth and Dorothea Wagner. **Dataset accompanying "Engineering Top-Down Weight-Balanced Trees"**. KITOpen Repository. 2019. DOI: 10.5445/IR/1000098852.
Cited on page 164.

[BW19c] Lukas Barth and Dorothea Wagner. **Dataset accompanying "Engineering Top-Down Weight-Balanced Trees"**. KITOpen Repository. 2019. DOI: 10.5445/IR/1000098852.
Cited on page 233.

[BW19d] Lukas Barth and Dorothea Wagner. **Shaving Peaks by Augmenting the Dependency Graph**. In *Proceedings of the 10th ACM International Conference on Future Energy Systems (e-Energy'19)*. ACM, 2019, pages 181–191. DOI: 10.1145/3307772.3328298.
Cited on page 111.

[BW20a] Lukas Barth and Dorothea Wagner. **Engineering Top-Down Weight-Balanced Trees**. In *Proceedings of the 22nd Workshop on Algorithm Engineering and Experiments (ALENEX'20)*. SIAM, 2020, pages 161–174. DOI: 10.1137/1.9781611976007.13.
Cited on page 157.

[BW20b] Lukas Barth and Dorothea Wagner. **Zipping Segment Trees**. In *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/LIPIcs.SEA.2020.25.
Cited on page 135.

[Chu+04] Julia Chuzhoy, Sudipto Guha, Sanjeev Khanna, and Joseph Seffi Naor. **Machine minimization for scheduling jobs with interval constraints**. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*. IEEE, 2004, pages 81–90. DOI: 10.1109/focs.2004.38.
Cited on page 101.

[Cie+04] Mark Cieliebak, Thomas Erlebach, Fabian Hennecke, Birgitta Weber, and Peter Widmayer. **Scheduling with release times and deadlines on a minimum number of machines**. In *Proceedings of the 3rd International Conference on Theoretical Computer Science (TCS'04)*. Ed. by Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell. Kluwer Academic Publishers, 2004, pages 209–222. DOI: 10.1007/1-4020-8141-3_18.
Cited on pages 12, 100, 113.

[CKL03] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. **Probabilistic discovery of time series motifs**. In *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'03)*. ACM, 2003, pages 493–498. DOI: 10.1145/956750.956808.
Cited on page 83.

[CL07] Yeim-Kuan Chang and Yung-Chieh Lin. **Dynamic segment trees for ranges and prefixes**. *IEEE Transactions on Computers* 56 (6 2007), pages 769–784. DOI: 10.1109/TC.2007.1037.
Cited on page 136.

[CMB02] Pedro M. Castro, Henrique Matos, and Ana P.F.D. Barbosa-Povoa. **Dynamic modelling and scheduling of an industrial batch system**. *Computers & Chemical Engineering* 26:4-5 (2002), pages 671–686. DOI: 10.1016/s0098-1354(01)00792-x.
Cited on page 21.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. **Introduction to algorithms**. MIT Press, 2009. ISBN: 978-0-262-53305-8.
Cited on pages 11, 158.

[CPW98] Bo Chen, Chris N. Potts, and Gerhard J. Woeginger. **A Review of Machine Scheduling: Complexity, Algorithms and Approximability**. In. *Handbook of Combinatorial Optimization*. Ed. by Panos M. Pardalos Ding-Zhu Du. Springer, 1998, pages 1493–1641. ISBN: 978-1-4613-7987-4. DOI: 10.1007/978-1-4613-0303-9_25.
Cited on page 113.

[CS00] Seonghun Cho and Sartaj Sahni. **A new weight balanced binary search tree**. *International Journal of Foundations of Computer Science* 11:3 (2000), pages 485–513. DOI: 10.1142/S0129054100000296.
Cited on page 158.

[Dem95] Erik Demeulemeester. **Minimizing resource availability costs in time-limited project networks**. *Management Science* 41:10 (1995), pages 1590–1598. DOI: 10.1287/mnsc.41.10.1590.
Cited on page 113.

[Den+15] Ruilong Deng, Zaiyue Yang, Mo-Yuen Chow, and Jiming Chen. **A survey on demand response in smart grids: Mathematical models and approaches**. *IEEE Transactions on Industrial Informatics* 11:3 (2015), pages 570–582. DOI: 10.1109/tii.2015.2414719.
Cited on pages 4, 113.

[den19] Deutsche Energie-Agentur GmbH (dena). **Energy Transition Trends 2019**. 2019. URL: https://www.dena.de/fileadmin/dena/Dokumente/Themen_und_Projekte/Internationales/China/CREO/Energy_transition_trends_2019_engl.pdf (visited on 01/13/2020).
Cited on page 1.

[DH02] Erik L. Demeulemeester and Willy S. Herroelen. **Project scheduling: A Research Handbook**. Ed. by Camille C. Price. Volume 49. International Series in Operations Research & Management Science. Kluwer Academic Publishers, 2002. ISBN: 978-1-4020-7051-8. DOI: 10.1007/b101924.
Cited on page 5.

[DH89] Richard F. Deckro and John E. Hebert. **Resource constrained project crashing**. *Omega* 17:1 (1989), pages 69–79. DOI: 10.1016/0305-0483(89)90022-4.
Cited on page 100.

[Dhu+15] Reinhilde D'hulst, Wouter Labeeuw, Bart Beusen, Sven Claessens, Geert Deconinck, and Koen Vanthournout. **Demand response flexibility and flexibility potential of residential smart appliances: Experiences from large pilot test in Belgium**. *Applied Energy* 155 (2015), pages 79–90. DOI: 10.1016/j.apenergy.2015.05.101.
Cited on page 39.

[Die+97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. **A reliable randomized algorithm for the closest-pair problem**. *Journal of Algorithms* 25:1 (1997), pages 19–51. DOI: 10.1006/jagm.1997.0873.
Cited on page 151.

[Die17] Reinhard Diestel. **Graph theory**. Fifth edition. Volume 173. Graduate Texts in Mathematics. Springer, 2017. ISBN: 978-3-662-53622-3. DOI: 10.1007/978-3-662-53622-3.
Cited on page 11.

[DL12] Pengwei Du and Ning Lu. **Appliance commitment for household load scheduling**. In *Proceedings of the 11th Transmission and Distribution Conference and Exposition (PES T&D'12)*. IEEE, 2012. DOI: 10.1109/tdc.2012.6281462.
Cited on page 19.

[Dür+16] Christoph Dürr, Sigrid Knust, Damien Prot, Rob van Stee, and Óskar C. Vásquez. **The scheduling zoo**. 2016. URL: http://schedulingzoo.lip6.fr/ (visited on 10/01/2019).
Cited on page 12.

[Eis11] Moustafa Mohammed Eissa. **Demand side management program evaluation based on industrial and commercial field data**. *Energy Policy* 39:10 (2011), pages 5961–5969. DOI: 10.1016/j.enpol.2011.06.057.
Cited on page 6.

[EJW05] Stefan Edelkamp, Shahid Jabbar, and Thomas Willhalm. **Geometric travel planning**. *IEEE Transactions on Intelligent Transportation Systems* 6:1 (2005), pages 5–16. DOI: 10.1109/TITS.2004.838182.
Cited on page 136.

[EKM09] Robert Earle, Edward P. Kahn, and Edo Macan. **Measuring the capacity impacts of demand response**. *The Electricity Journal* 22:6 (2009), pages 47–58. DOI: 10.1016/j.tej.2009.05.014.
Cited on page 100.

[Est+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. **A Density-Based Algorithm for Discovering Clusters a Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise**. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*. KDD'96. Portland, Oregon: AAAI Press, 1996, pages 226–231.
Cited on pages 84, 125.

[Fan+11] Xi Fang, Satyajayant Misra, Guoliang Xue, and Dejun Yang. **Smart grid — The new and improved power grid: A survey**. *IEEE Communications Surveys & Tutorials* 14:4 (2011), pages 944–980. DOI: 10.1109/surv.2011.101911.00087.
Cited on page 100.

[Fed19] Federal Ministry for Economic Affairs and Energy. *Zeitreihen zur Entwicklung der erneuerbaren Energien in Deutschland*. Berlin, Germany, 2019. URL: https://www.erneuerbare-energien.de/EE/Navigation/DE/Service/Erneuerbare_Energien_in_Zahlen/Zeitreihen/zeitreihen.html.
Cited on page 1.

[Feh+14] Daniel Fehrenbach, Erik Merkel, Russell McKenna, Ute Karl, and Wolf Fichtner. **On the economic potential for electric load management in the German residential heating sector — An optimising energy system model approach**. *Energy* 71 (2014), pages 263–276. DOI: 10.1016/j.energy.2014.04.061.
Cited on page 19.

[FF14] Paddy Finn and Colin Fitzpatrick. **Demand side management of industrial electricity consumption: promoting the use of renewable energy through real-time pricing**. *Applied Energy* 113 (2014), pages 11–21. DOI: 10.1016/j.apenergy.2013.07.003.
Cited on pages 6, 79.

[FHM14] Jirı Fink, Johann L. Hurink, and Albert Molderink. **Mathematical modelling of devices and flows in energy systems**. Tech. rep. Technical report, 2014. URL: https://kam.mff.cuni.cz/~fink/publications/flow.pdf.
Cited on page 21.

[FN16] Stefan Feuerriegel and Dirk Neumann. **Integration Scenarios of Demand Response into Electricity Markets: Load Shifting, Financial Savings and Policy Implications**. *Energy Policy* 96 (2016), pages 231–240. DOI: 10.1016/j.enpol.2016.05.050.
Cited on pages 35, 39.

[GEM17] Nicholas Good, Keith A. Ellis, and Pierluigi Mancarella. **Review and classification of barriers and enablers of demand response in the smart grid**. *Renewable and Sustainable Energy Reviews* 72 (2017), pages 57–72. DOI: 10.1016/j.rser.2017.01.043.
Cited on page 113.

[GFW16] Johannes Gärttner, Christoph M. Flath, and Christof Weinhardt. **Load Shifting, Interrupting or Both? Customer Portfolio Composition in Demand Side Management**. In *Proceedings of the 11th International Conference on Management Science (CMS'14)*. Springer, 2016, pages 9–15. DOI: 10.1007/978-3-319-20430-7_2.
Cited on page 20.

[GJ79] Michael R. Garey and David S. Johnson. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. W. H. Freeman, 1979. ISBN: 978-0-7167-1044-8.
Cited on page 176.

[GNR16] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. **Evaluation of labeling strategies for rotating maps**. *Journal of Experimental Algorithmics* 21 (1 2016), pages 1–21. DOI: 10.1145/2851493.
Cited on page 136.

[Gon+15] Yanmin Gong, Ying Cai, Yuanxiong Guo, and Yuguang Fang. **A privacy-preserving scheme for incentive-based demand response in the smart grid**. *IEEE Transactions on Smart Grid* 7:3 (2015), pages 1304–1313. DOI: 10.1109/tsg.2015.2412091.
Cited on page 39.

[Got+11] Sebastian Gottwalt, Wolfgang Ketter, Carsten Block, John Collins, and Christof Weinhardt. **Demand side management—A simulation of household behavior under variable prices**. *Energy Policy* 39:12 (2011), pages 8163–8174. DOI: 10.1016/j.enpol.2011.10.016.
Cited on pages 19, 80.

[Got+16] Sebastian Gottwalt, Johannes Gärttner, Hartmut Schmeck, and Christof Weinhardt. **Modeling and valuation of residential demand flexibility for renewable energy integration**. *IEEE Transactions on Smart Grid* 8:6 (2016), pages 2565–2574. DOI: 10.1109/tsg.2016.2529424.
Cited on pages 19, 21, 100.

[Gra+79] Ronald L. Graham, Eugene L. Lawler, Jan K. Lenstra, and Alexander H.G. Rinnooy Kan. "Optimization and approximation in deterministic sequencing and scheduling: a survey." In *Annals of Discrete Mathematics*. Ed. by Peter L. Hammer, Ellis L. Johnson, and Bernhard H. Korte. Volume 5. Elsevier, 1979, pages 287–326. DOI: 10.1016/S0167-5060(08)70356-X.
Cited on page 12.

[Gul+08] T.A. Guldemond, Johann L. Hurink, Jacob Jan Paulus, and Johannes M.J. Schutten. **Time-constrained project scheduling**. *Journal of Scheduling* 11:2 (2008), pages 137–148. DOI: 10.1007/s10951-008-0059-7.
Cited on pages 100, 113.

[Har92] George William Hart. **Nonintrusive appliance load monitoring**. *Proceedings of the IEEE* 80:12 (1992), pages 1870–1891. DOI: 10.1109/5.192069.
Cited on page 39.

[HDD99] Willy Herroelen, Erik Demeulemeester, and Bert De Reyck. "A classification scheme for project scheduling." In *Project scheduling*. Ed. by Jan Węglarz. Springer, 1999, pages 1–26. DOI: 10.1007/978-1-4615-5533-9_1.
Cited on pages 114, 177.

[He+13] Xian He, Nico Keyaerts, Isabel Azevedo, Leonardo Meeus, Leigh Hancher, and Jean-Michel Glachant. **How to engage consumers in demand response: A contract perspective**. *Utilities Policy* 27 (2013), pages 108–122. DOI: 10.1016/j.jup.2013.10.001.
Cited on page 19.

[HL01] Bente Halvorsen and Bodil M. Larsen. **The flexibility of household electricity demand over time**. *Resource and Energy Economics* 23:1 (2001), pages 1–18. DOI: 10.1016/s0928-7655(00)00035-x.
Cited on page 19.

[HLH14] Veit Hagenmeyer, Heinz Langner, and Werner Hartwig. "Eine Methode zur Bewertung der Energieversorgungssicherheit von komplexen Produktionsstätten." In *VGB-Fachtagung: Dampferzeuger, Wirbelschichtfeuerungen, Industrie- und Heizkraftwerke*. Weimar, 2014.
Cited on page 47.

[HS91] Yuan-Yih Hsu and Chung-Ching Su. **Dispatch of direct load control using dynamic programming**. *IEEE Transactions on Power Systems* 6:3 (1991), pages 1056–1061. DOI: 10.1109/59.119246.
Cited on page 113.

[HSE16] Haider Tarish Haider, Ong Hang See, and Wilfried Elmenreich. **A review of residential demand response of smart grid**. *Renewable and Sustainable Energy Reviews* 59 (2016), pages 166–178. DOI: 10.1016/j.rser.2016.01.016.
Cited on pages 4, 6.

[HST15] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. **Rank-Balanced Trees**. *ACM Transactions on Algorithms* 11:4 (2015), pages 1–26. DOI: 10.1145/2689412.
Cited on page 158.

[HY11] Yoichi Hirai and Kazuhiko Yamamoto. **Balancing weight-balanced trees**. *Journal of Functional Programming* 21:3 (2011), pages 287–307. DOI: 10.1017/S0956796811000104.
Cited on pages 158, 159, 161, 164.

[ISE20] Fraunhofer ISE. **Energy Charts**. 2020. URL: https://www.energy-charts.de/power_inst_de.htm (visited on 01/13/2020).
Cited on page 1.

[KJ11] J. Zico Kolter and Matthew J. Johnson. **REDD: A public data set for energy disaggregation research**. In *Proceedings of the Workshop on Data Mining Applications in Sustainability (SustKDD'11)*. 2011.
Cited on pages 36, 105.

[Kli+15] Sonja Klingert, Florian Niedermeier, Corentin Dupont, Giovanni Giuliani, Thomas Schulze, and Hermann de Meer. **Introducing flexibility into data centers for smart cities**. In *Proceedings of the 1st International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS'15)*. Springer, 2015, pages 128–145. DOI: 10.1007/978-3-319-27753-0_7.
Cited on page 39.

[Knu98] Donald E. Knuth. **The Art of Computer Programming: Sorting and Searching**. Second edition. Volume 3. Addison-Wesley, 1998. ISBN: 978-0-201-89685-5.
Cited on page 157.

[KO93] Marc J. van Kreveld and Mark H. Overmars. **Union-copy structures and dynamic segment trees**. *Journal of the ACM* 40:3 (1993), pages 635–652. DOI: 10.1145/174130.174140.
Cited on pages 121, 135, 137, 138, 139, 143, 150.

[Kol96] Rainer Kolisch. **Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation**. *European Journal of Operational Research* 90:2 (1996), pages 320–333. DOI: 10.1016/0377-2217(95)00357-6.
Cited on page 116.

[Kon+11] Oumar Koné, Christian Artigues, Pierre Lopez, and Marcel Mongeau. **Event-based MILP models for resource-constrained project scheduling problems**. *Computers & Operations Research* 38:1 (2011), pages 3–13. DOI: 10.1016/j.cor.2009.12.011.
Cited on pages 5, 63, 64, 68.

[KR13] Jungsuk Kwac and Ram Rajagopal. **Demand response targeting using big data analytics**. In *Proceedings of the 1st IEEE International Conference on Big Data (BigData'13)*. IEEE, 2013, pages 683–690. DOI: 10.1109/bigdata.2013.6691643.
Cited on page 35.

[KS97] Rainer Kolisch and Arno Sprecher. **PSPLIB — a project scheduling problem library**. *European Journal of Operational Research* 96:1 (1997), pages 205–216. DOI: 10.1016/S0377-2217(96)00170-1.
Cited on page 80.

[KSS99] Rainer Kolisch, Christoph Schwindt, and Arno Sprecher. "Benchmark instances for project scheduling problems." In *Project scheduling*. Ed. by Jan Węglarz. Springer, 1999, pages 197–212. DOI: 10.1007/978-1-4615-5533-9_9.
Cited on page 80.

[La 90] Johannes Antonius La Poutré. **New techniques for the union-find problem**. In *Proceedings of the 1st annual ACM-SIAM symposium on Discrete algorithms (SODA'90)*. SIAM, 1990, pages 54–63. ISBN: 978-0-89871-251-3.
Cited on page 137.

[Lar+17] Emil Mahler Larsen, Pierre Pinson, Fabian Leimgruber, and Florian Judex. **Demand response evaluation and forecasting — Methods and results from the EcoGrid EU experiment**. *Sustainable Energy, Grids and Networks* 10 (2017), pages 75–83. DOI: 10.1016/j.segan.2017.03.001.
Cited on page 4.

[Li+12] Ying Li, Boon Loong Ng, Mark Trayer, and Lingjia Liu. **Automated residential demand response: Algorithmic implications of pricing models**. *IEEE Transactions on Smart Grid* 3:4 (2012), pages 1712–1721. DOI: 10.1109/TSG.2012.2218262.
Cited on pages 80, 100, 105.

[Liu+13] Zhenhua Liu, Adam Wierman, Yuan Chen, Benjamin Razon, and Niangjun Chen. **Data Center Demand Response: Avoiding the Coincident Peak via Workload Shifting and Local Generation**. In *Proceedings of the 31st International Symposium on Computer Performance, Modeling, Measurements and Evaluation (Performance'13)*. Elsevier, 2013. DOI: 10.1016/j.peva.2013.08.014.
Cited on page 39.

[LSS12] Thillainathan Logenthiran, Dipti Srinivasan, and Tan Zong Shun. **Demand side management in smart grid using heuristic optimization**. *IEEE Transactions on Smart Grid* 3:3 (2012), pages 1244–1252. DOI: 10.1109/TSG.2012.2195686.
Cited on pages 80, 113.

[Lud+17] Nicole Ludwig, Simon Waczowicz, Ralf Mikut, and Veit Hagenmeyer. **Mining Flexibility Patterns in Energy Time - Series from Industrial Processes**. In *Proceedings of the 27th Workshop Computational Intelligence*. Ed. by Frank Hoffmann, Eyke Hüllermeier, and Ralf Mikut. KIT Scientific Publishing, 2017, pages 13–13. DOI: 10.5445/KSP/1000074341.
Cited on pages 41, 83, 124.

[Lud+19a] Nicole Ludwig, Lukas Barth, Dorothea Wagner, and Veit Hagenmeyer. **Benchmark Dataset for "Industrial Demand-Side Flexibility: A Benchmark Data Set"**. KITOpen Repository. 2019. DOI: 10.5445/IR/1000094324.
Cited on page 86.

[Lud+19b] Nicole Ludwig, Lukas Barth, Dorothea Wagner, and Veit Hagenmeyer. **Industrial Demand-Side Flexibility: A Benchmark Data Set**. In *Proceedings of the 10th ACM International Conference on Future Energy Systems (e-Energy'19)*. ACM, 2019, pages 460–473. DOI: 10.1145/3307772.3331021.
Cited on pages 79, 124.

[Luo+98] Zhonghui Luo, Ratnesh Kumar, Joseph Sottile, and Jon C. Yingling. **An MILP formulation for load-side demand control**. *Electric Machines & Power Systems* 26:9 (1998), pages 935–949. DOI: 10.1080/07313569808955868.
Cited on page 21.

[LW93] Tony W. Lai and Derick Wood. **A Top-Down Updating Algorithm for Weight-Balanced Trees**. *International Journal of Foundations of Computer Science* 4:4 (1993), pages 309–324. DOI: 10.1142/S0129054193000201.
Cited on pages 158, 161, 162, 164.

[Mah+16] Danish Mahmood, Nadeem Javaid, Nabil Alrajeh, Zahoor Khan, Umar Qasim, Imran Ahmed, and Manzoor Ilahi. **Realistic scheduling mechanism for smart homes**. *Energies* 9:3 (2016), page 202. DOI: 10.3390/en9030202.
Cited on page 6.

[Mäk87] Erkki Mäkinen. **On top-down splaying**. *BIT Numerical Mathematics* 27:3 (1987), pages 330–339. DOI: 10.1007/BF01933728.
Cited on page 165.

[Meh84a] Kurt Mehlhorn. **Data Structures and Algorithms 1: Sorting and Searching**. Ed. by Juraj Hromkovič and Mogens Nielsen. EATCS Monographs on Theoretical Computer Science. Springer, 1984. ISBN: 978-3-642-69674-9. DOI: 10.1007/978-3-642-69672-5.
Cited on page 157.

[Meh84b] Kurt Mehlhorn. **Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry**. Ed. by Juraj Hromkovič and Mogens Nielsen. EATCS Monographs on Theoretical Computer Science. Springer, 1984. ISBN: 978-3-642-69902-3. DOI: 10.1007/978-3-642-69900-9.
Cited on page 157.

[Mer+15] Lennart Merkert, Iiro Harjunkoski, Alf Isaksson, Simo Säynevirta, Antti Saarela, and Guido Sand. **Scheduling and energy — Industrial challenges and opportunities**. *Computers & Chemical Engineering* 72 (2015), pages 183–198. DOI: 10.1016/j.compchemeng.2014.05.024.
Cited on page 6.

[Mit+12] Sumit Mitra, Ignacio E. Grossmann, Jose M. Pinto, and Nikhil Arora. **Optimal production planning under time-sensitive electricity prices for continuous power-intensive processes**. *Computers & Chemical Engineering* 38 (2012), pages 171–184. DOI: 10.1016/j.compchemeng.2011.09.019.
Cited on pages 20, 21, 100.

[Möh84] Rolf H. Möhring. **Minimizing costs of resource requirements in project networks subject to a fixed completion time**. *Operations Research* 32:1 (1984), pages 89–120. DOI: 10.1287/opre.32.1.89.
Cited on pages 13, 113.

[MP14] Joon-Yung Moon and Jinwoo Park. **Smart production scheduling with time-dependent and machine-dependent electricity cost by considering distributed energy resources and energy storage**. *International Journal of Production Research* 52:13 (2014), pages 3922–3939. DOI: 10.1080/00207543.2013.860251.
Cited on pages 20, 21.

[MW15] Matthias Mnich and Andreas Wiese. **Scheduling and fixed-parameter tractability**. *Mathematical Programming* 154:1-2 (2015), pages 533–562. DOI: 10.1007/s10107-014-0830-9.
Cited on page 176.

[NHG17] Seyyed Mostafa Nosratabadi, Rahmat-Allah Hooshmand, and Eskandar Gholipour. **A comprehensive review on microgrid and virtual power plant concepts employed for distributed energy resources scheduling in power systems**. *Renewable and Sustainable Energy Reviews* 67 (2017), pages 341–363. DOI: 10.1016/j.rser.2016.09.025.
Cited on pages 5, 6.

[NK14] Anulark Naber and Rainer Kolisch. **MIP models for resource-constrained project scheduling with flexible resource profiles**. *European Journal of Operational Research* 239:2 (2014), pages 335–348. DOI: 10.1016/j.ejor.2014.05.036.
Cited on page 5.

[NR73] Jürg Nievergelt and Edward M. Reingold. **Binary Search Trees of Bounded Balance**. *SIAM Journal on Computing* 2:1 (1973), pages 33–43. DOI: 10.1137/0202005.
Cited on pages 157, 159.

[OCB07] Alexandre Oudalov, Rachid Cherkaoui, and Antoine Beguin. **Sizing and optimal operation of battery energy storage system for peak shaving application**. In *Proceedings of the 4th IEEE Power Tech (PowerTech'07)*. IEEE, 2007, pages 621–625. DOI: 10.1109/pct.2007.4538388.
Cited on pages 20, 21.

[OR15] Gearóid O'Brien and Ram Rajagopal. **Scheduling non-preemptive deferrable loads**. *IEEE Transactions on Power Systems* 31:2 (2015), pages 835–845. DOI: 10.1109/tpwrs.2015.2402198.
Cited on page 39.

[PB11] Moritz Paulus and Frieder Borggrefe. **The potential of demand-side management in energy-intensive industries for electricity markets in Germany**. *Applied Energy* 88:2 (2011), pages 432–441. DOI: 10.1016/j.apenergy.2010.03.017.
Cited on page 20.

[PD11] Peter Palensky and Dietmar Dietrich. **Demand Side Management: Demand Response, Intelligent Energy Systems, and Smart Loads**. *IEEE Transactions on Industrial Informatics* 7:3 (2011), pages 381–388. DOI: 10.1109/tii.2011.2158841.
Cited on page 19.

[Pet+13] Mette K. Petersen, Kristian Edlund, Lars Henrik Hansen, Jan Bendtsen, and Jakob Stoustrup. **A taxonomy for modeling flexibility and a computationally efficient algorithm for dispatch in smart grids**. In *Proceedings of the 31st American Control Conference (ACC'13)*. IEEE, 2013, pages 1150–1156. DOI: 10.1109/acc.2013.6579991.
Cited on pages 20, 21.

[Pet+14] Mette K. Petersen, Lars H. Hansen, Jan Bendtsen, Kristian Edlund, and Jakob Stoustrup. **Heuristic optimization for the discrete virtual power plant dispatch problem**. *IEEE Transactions on Smart Grid* 5:6 (2014), pages 2910–2918. DOI: 10.1109/TSG.2014.2336261.
Cited on pages ix, 6, 20, 21, 32, 80, 100, 105, 109, 112, 113, 122, 123.

[PH00] Kostantinos N. Plataniotis and Dimitris Hatzinakos. **Gaussian mixtures and their applications to signal processing**. In. *Advanced signal processing handbook: theory and implementation for radar, sonar, and medical imaging real time systems*. Ed. by Stergios Stergiopoulos. CRC Press, 2000. 3. DOI: 10.1201/9781420037395.
Cited on page 84.

[Pon+13] Jose Luis Ponz-Tienda, Víctor Yepes, Eugenio Pellicer, and Joaquin Moreno-Flores. **The resource leveling problem with multiple resources using an adaptive genetic algorithm**. *Automation in Construction* 29 (2013), pages 161–172. DOI: 10.1016/j.autcon.2012.10.003.
Cited on page 5.

[Pow98] David M.W. Powers. **Applications and explanations of Zipf's law**. In *Proceedings of the 1st Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning (NeMLaP3/CoNLL'98)*. Association for Computational Linguistics, 1998, pages 151–160. DOI: 10.3115/1603899.1603924.
Cited on page 165.

[Pra59] John W. Pratt. **Remarks on zeros and ties in the Wilcoxon signed rank procedures**. *Journal of the American Statistical Association* 54:287 (1959), pages 655–667. DOI: 10.1080/01621459.1959.10501526.
Cited on page 49.

[PS17] Danny Pudjianto and Goran Strbac. **Assessing the value and impact of demand side response using whole-system approach**. *Proceedings of the Institution of Mechanical Engineers, Part A: Journal of Power and Energy* 231:6 (2017), pages 498–507. DOI: 10.1177/0957650917722381.
Cited on pages 4, 39.

[PSM10] Michael Angelo A. Pedrasa, Ted D. Spooner, and Iain F. MacGill. **Coordinated scheduling of residential distributed energy resources to optimize smart home energy services**. *IEEE Transactions on Smart Grid* 1:2 (2010), pages 134–143. DOI: 10.1109/tsg.2010.2053053.
Cited on page 100.

[PWW69] A. Alan B. Pritsker, Lawrence J. Waiters, and Philip M. Wolfe. **Multiproject scheduling with limited resources: A zero-one programming approach**. *Management Science* 16:1 (1969), pages 93–108. DOI: 10.1287/mnsc.16.1.93.
Cited on pages 5, 18, 63.

[QGJ14] Faran A. Qureshi, Tomasz T. Gorecki, and Colin N. Jones. **Model predictive control for market-based demand response participation**. *IFAC Proceedings Volumes* 47:3 (2014), pages 11153–11158. DOI: 10.3182/20140824-6-za-1003.02395.
Cited on page 20.

[Ran13] Mohammad Ranjbar. **A path-relinking metaheuristic for the resource levelling problem**. *Journal of the Operational Research Society* 64:7 (2013), pages 1071–1078. DOI: 10.1057/jors.2012.119.
Cited on pages 5, 113.

[RB14] Sami Rollins and Nilanjan Banerjee. **Using rule mining to understand appliance energy consumption patterns**. In *Proceedings of the 12th IEEE International Conference on Pervasive Computing and Communications (PerCom'14)*. IEEE, 2014, pages 29–37. DOI: 10.1109/percom.2014.6813940.
Cited on page 39.

[Ren18] Center for Renewable Energy Development. *China Renewable Energy Outlook 2018*. Beijing, China, 2018. URL: http://boostre.cnrec.org.cn/index.php/2018/11/27/china-renewable-energy-outlook-2018/?lang=en (visited on 01/13/2020).
Cited on pages 1, 2.

[Rou01] Salvador Roura. **A New Method for Balancing Binary Search Trees**. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*. Ed. by Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen. Springer, 2001. DOI: 10.1007/3-540-48224-5_39.
Cited on pages 158, 161.

[Rou13] Salvador Roura. **Fibonacci BSTs: A new balancing method for binary search trees**. *Theoretical Computer Science* 482 (2013), pages 48–59. DOI: 10.1016/j.tcs.2012.11.027.
Cited on page 158.

[RZG12] Julia Rieck, Jürgen Zimmermann, and Thorsten Gather. **Mixed-integer linear programming for resource leveling problems**. *European Journal of Operational Research* 221:1 (2012), pages 27–37. DOI: 10.1016/j.ejor.2012.03.003.
Cited on page 5.

[SA96] Raimund Seidel and Cecilia R. Aragon. **Randomized search trees**. *Algorithmica* 16:4-5 (1996), pages 464–497. DOI: 10.1007/BF01940876.
Cited on page 150.

[Sch12] Ruggero Schleicher-Tappeser. **How renewables will change electricity markets in the next five years**. *Energy Policy* 48 (2012), pages 64–75. DOI: 10.1016/j.enpol.2012.04.042.
Cited on page 17.

[Sco+13] Paul Scott, Sylvie Thiébaux, Menkes van den Briel, and Pascal van Hentenryck. **Residential demand response under uncertainty**. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP'13)*. Springer, 2013, pages 645–660. DOI: 10.1007/978-3-642-40627-0_48.
Cited on page 19.

[SGA14] Ana Soares, Álvaro Gomes, and Carlos Henggeler Antunes. **Categorization of residential electricity consumption as a basis for the assessment of the impacts of demand response actions**. *Renewable and Sustainable Energy Reviews* 30 (2014), pages 490–503. DOI: 10.1016/j.rser.2013.10.019.
Cited on page 19.

[Sia14] Pierluigi Siano. **Demand response and smart grids—A survey**. *Renewable and Sustainable Energy Reviews* 30 (2014), pages 461–478. DOI: 10.1016/j.rser.2013.10.022.
Cited on pages vii, 4, 99, 111, 112.

[Sou+11] Kin Cheong Sou, James Weimer, Henrik Sandberg, and Karl Henrik Johansson. **Scheduling smart home appliances using mixed integer linear programming**. In *Proceedings of the 50th IEEE Conference on Decsion and Control and European Control Conference (CDC-ECC'11)*. IEEE, 2011. DOI: 10.1109/cdc.2011.6161081.
Cited on pages 19, 21.

[SP96] Gordian Schilling and Constantinos C. Pantelides. **A simple continuous-time process scheduling formulation and a novel solution algorithm**. *Computers & Chemical Engineering* 20 (1996), S1221–S1226. DOI: 10.1016/0098-1354(96)00211-6.
Cited on pages 20, 21.

[SS05] Raimund Seidel and Micha Sharir. **Top-down analysis of path compression**. *SIAM Journal on Computing* 34:3 (2005), pages 515–525. DOI: 10.1137/S0097539703439088.
Cited on page 137.

[Str+14] Philipp Ströhle, Enrico H. Gerding, Mathijs M. de Weerdt, Sebastian Stein, and Valentin Robu. **Online Mechanism Design for Scheduling Non-Preemptive Jobs under Uncertain Supply and Demand**. In *Proceedings of the 13th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'14)*. International Foundation for Autonomous Agents and Multiagent Systems, 2014, pages 437–444. ISBN: 978-1-4503-2738-1.
Cited on page 19.

[Str08] Goran Strbac. **Demand side management: Benefits and challenges**. *Energy Policy* 36:12 (2008), pages 4419–4426. DOI: 10.1016/j.enpol.2008.09.030.
Cited on pages 4, 17, 19, 39.

[SXZ14] Ditiro Setlhaolo, Xiaohua Xia, and Jiangfeng Zhang. **Optimal scheduling of household appliances for demand response**. *Electric Power Systems Research* 116 (2014), pages 24–28. DOI: 10.1016/j.epsr.2014.04.012.
Cited on page 19.

[Tan14] Jay Taneja. **Growth in renewable generation and its effect on demand-side management**. In *Proceedings of the 5th IEEE International Conference on Smart Grid Communications (SmartGridComm'14)*. IEEE, 2014, pages 614–619. DOI: 10.1109/smartgridcomm.2014.7007715.
Cited on pages 35, 39.

[Tar85] Robert E. Tarjan. **Efficient top-down updating of red-black trees**. Tech. rep. 1985. URL: ftp://ftp.cs.princeton.edu/techreports/1985/006.pdf.
Cited on page 158.

[TLT18] Robert E. Tarjan, Caleb C. Levy, and Stephen Timmel. **Zip Trees**. *CoRR* abs/1806.06726 (2018). arXiv: 1806.06726. URL: http://arxiv.org/abs/1806.06726v4.
Cited on page 150.

[TLT19] Robert E. Tarjan, Caleb C. Levy, and Stephen Timmel. **Zip trees**. In *Proceedings of the 16th Workshop on Algorithms and Data Structures (WADS'19)*. Springer, 2019, pages 566–577. DOI: 10.1007/978-3-030-24766-9_41.
Cited on pages 135, 136, 137, 143, 144, 149.

[Tra06] European Network of Transmission System Operators for Electricity. **Final Report — System Disturbance on 4 November 2006** (2006). URL: https://www.entsoe.eu/fileadmin/user_upload/_library/publications/ce/otherreports/Final-Report-20070130.pdf.
Cited on page 2.

[Tru+16] Ngoc Cuong Truong, Tim Baarslag, Sarvapali D. Ramchurn, and Long Tran-Thanh. **Interactive Scheduling of Appliance Usage in the Home**. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*. AAAI, 2016, pages 869–875. ISBN: 978-1-57735-770-4. URL: https://dl.acm.org/doi/abs/10.5555/3060621.3060742.
Cited on page 39.

[US 06] U.S. Department of Energy. **Benefits of demand response in electricity markets and recommendations for achieving them**. 2006. URL: https://www.energy.gov/oe/downloads/benefits-demand-response-electricity-markets-and-recommendations-achieving-them-report (visited on 02/03/2020).
Cited on pages 4, 99, 111.

[VZV15] John S. Vardakas, Nizar Zorba, and Christos V. Verikoukis. **A survey on demand response programs in smart grids: Pricing methods and optimization algorithms**. *IEEE Communications Surveys & Tutorials* 17:1 (2015), pages 152–178. DOI: 10.1109/comst.2014.2341586.
Cited on pages 4, 112.

[Węg99] Jan Węglarz. **Project scheduling: Recent models, Algorithms and Applications**. Volume 14. International Series in Operations Research & Management Science. Kluwer Academic Publishers, 1999. ISBN: 978-1-4613-7529-6. DOI: 10.1007/978-1-4615-5533-9.
Cited on pages vii, 5, 12, 40, 64, 113, 175.

[Wei+12] Anke Weidlich, Harald Vogt, Wolfgang Krauss, Patrik Spiess, Marek Jawurek, Martin Johns, and Stamatis Karnouskos. **Decentralized intelligence in energy efficient power systems**. In. *Handbook of Networks in Power Systems I*. Ed. by Alexey Sorokin, Steffen Rebennack, Panos M. Pardalos, Niko A. Iliadis, and Mario V. F. Pereira. Springer, 2012, pages 467–486. ISBN: 978-3-642-23192-6. DOI: 10.1007/978-3-642-23193-3_18.
Cited on page 17.

[Yaw+14] Sean Yaw, Brendan Mumey, Erin McDonald, and Jennifer Lemke. **Peak demand scheduling in the smart grid**. In *Proceedings of the 5th IEEE International Conference on Smart Grid Communications (SmartGridComm'14)*. IEEE, 2014, pages 770–775. DOI: 10.1109/SmartGridComm.2014.7007741.
Cited on pages 80, 105, 109, 113.

[YM17] Sean Yaw and Brendan Mumey. **Scheduling Non-Preemptible Jobs to Minimize Peak Demand**. *Algorithms* 10:4 (2017), page 122. DOI: 10.3390/a10040122.
Cited on page 176.

[YZ09] Guosong Yu and Guochuan Zhang. **Scheduling with a minimum number of machines**. *Operations Research Letters* 37:2 (2009), pages 97–101. DOI: 10.1016/j.orl.2009.01.008.
Cited on page 113.

[Zeh+17] M. Alparslan Zehir, M.H. Wevers, Alp Batman, Mustafa Bagriyanik, Johann L. Hurink, Unal Kucuk, Filipe J. Soares, and Aydoğan Ozdemir. **A novel incentive-based retail demand response program for collaborative participation of small customers**. In *Proceedings of the 9th IEEE Power Tech (PowerTech'17)*. IEEE, 2017, pages 1–6. DOI: 10.1109/PTC.2017.7981059.
Cited on page 39.

[ZK08] Audrey Zibelman and Edward N. Krapels. **Deployment of demand response as a real-time resource in organized markets**. *The Electricity Journal* 21:5 (2008), pages 51–56. DOI: 10.1016/j.tej.2008.05.011.
Cited on page 100.

[ZLC17] Shizhen Zhao, Xiaojun Lin, and Minghua Chen. **Robust Online Algorithms for Peak-Minimizing EV Charging Under Multistage Uncertainty**. *IEEE Transactions on Automatic Control* 62:11 (2017), pages 5739–5754. DOI: 10.1109/tac.2017.2699290.
Cited on page 39.

# Index

# A Appendix for: Exploring the Benefits of Flexibilization in Industrial Contexts

## A.1 Data Publication

We publish the instance sets from Section 4.4.2 at

`https://publikationen.bibliothek.kit.edu/1000082194`

This publication [Bar+18a] contains

- The PS-Nonuniform, PS-Uniform, PSG and OM instance sets,
- The computational results of our optimization,
- and information on how to repeat our experiments.

Additionally, we publish the software that we used for optimization at

`https://github.com/kit-algo/TCPSPSuite`

Note that the raw data from which we detected our motifs is also published as the HIPE data set [Bis+18].

## A.2 Omitted Figures and Tables

**Table A.1:** Parameter choices for the motif discovery algorithm. The alphabet size was varied between 2 and 10 words, resulting in largely the same results as presented above.

| Machine | Motifs | Alphabet Size | Wordlength |
|---|---|---|---|
| AVT 01 | A, D | 4 | 505 |
| AVT 02 | J, K | 4 | 403 |
| AVT 03 | B, M, N | 4 | 267 |
| AVT 04 | H | 4 | 433 |
| AVT 05 | C | 4 | 543 |
| AVT 06 | E | 4 | 406 |
| AVT 08 | F, L | 4 | 211 |
| AVT 09 | G | 4 | 500 |
| AVT 10 | I, O | 4 | 426 |

**Table A.2:** $p$-Values for the change of one parameter in the PS-Uniform set. Values highlighted in green indicate that changing $\hat{J}$ and $\Theta$, while keeping the other one constant, results in a statistically significant change in improvements. Values in blue are significant before Bonferroni correction.

| | 3 | → | 6 | → | 9 |
|---|---|---|---|---|---|
| 0.005 | | $< 10^{-5}$ | | 0.00089 | |
| ↓ | $< 10^{-5}$ | | $< 10^{-5}$ | | $< 10^{-5}$ |
| 0.01 | | $< 10^{-5}$ | | $< 10^{-5}$ | |
| ↓ | 0.00025 | | $< 10^{-5}$ | | 0.00033 |
| 0.02 | | $< 10^{-5}$ | | 0.00036 | |
| ↓ | 0.0008 | | 0.03 | | 0.041 |
| 0.03 | | $< 10^{-4}$ | | 0.086 | |
| ↓ | 0.00011 | | $< 10^{-5}$ | | 0.00022 |
| 0.04 | | 0.00033 | | 0.0018 | |

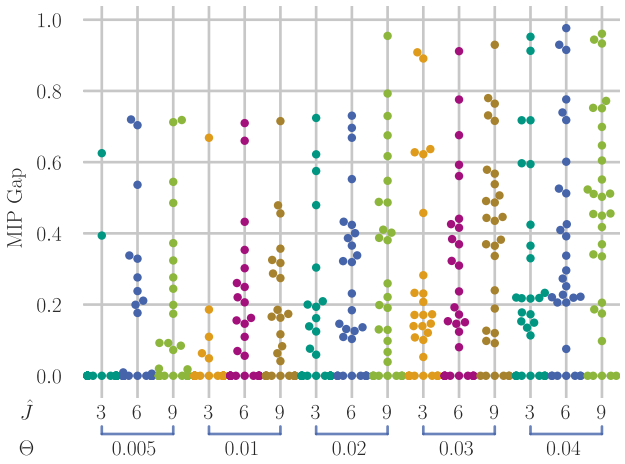



**Figure A.1:** MIP gaps for the various settings of $\hat{J}$ and $\Theta$ in the PS-Nonuniform instance set. Every dot corresponds to one instance. Colors are used to distinguish the columns.

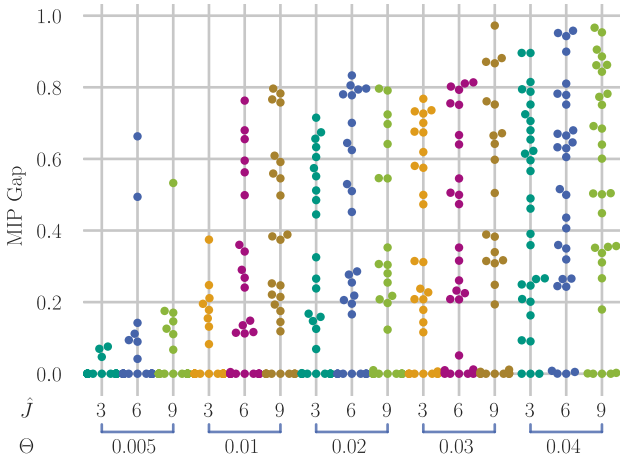



**Figure A.2:** MIP gaps for the various settings of $\hat{J}$ and $\Theta$ in the PSG instance set. Every dot corresponds to one instance. Colors are used to distinguish the columns.

**Table A.3:** $p$-Values for the change of one parameter in the PSG set. Values highlighted in green indicate that changing $\hat{J}$ and $\Theta$, while keeping the other one constant, results in a statistically significant change in improvements. Values in blue are significant before Bonferroni correction.

| | 3 | → | 6 | → | 9 |
|---|---|---|---|---|---|
| 0.005 | | $< 10^{-5}$ | | $< 10^{-4}$ | |
| ↓ | $< 10^{-5}$ | | $< 10^{-5}$ | | $< 10^{-5}$ |
| 0.01 | | $< 10^{-5}$ | | $< 10^{-5}$ | |
| ↓ | 0.00037 | | $< 10^{-4}$ | | $< 10^{-5}$ |
| 0.02 | | $< 10^{-5}$ | | $< 10^{-5}$ | |
| ↓ | 0.00021 | | 0.0051 | | 0.00017 |
| 0.03 | | $< 10^{-5}$ | | $< 10^{-5}$ | |
| ↓ | 0.0021 | | $< 10^{-5}$ | | $< 10^{-4}$ |
| 0.04 | | $< 10^{-5}$ | | $< 10^{-5}$ | |

**Table A.4:** $p$-Values for the change of one parameter in the OM set. Values highlighted in green indicate that changing $\hat{J}$ and $\Theta$, while keeping the other one constant, results in a statistically significant change in improvements. Values in blue are significant before Bonferroni correction.

| | 3 | → | 6 | → | 9 |
|---|---|---|---|---|---|
| 0.005 | | $< 10^{-4}$ | | 0.00045 | |
| ↓ | $< 10^{-5}$ | | $< 10^{-5}$ | | $< 10^{-5}$ |
| 0.01 | | 0.04 | | 0.059 | |
| ↓ | $< 10^{-4}$ | | $< 10^{-5}$ | | $< 10^{-5}$ |
| 0.02 | | $< 10^{-5}$ | | 0.00068 | |
| ↓ | $< 10^{-5}$ | | $< 10^{-5}$ | | $< 10^{-5}$ |
| 0.03 | | $< 10^{-4}$ | | 0.011 | |
| ↓ | $< 10^{-4}$ | | $< 10^{-5}$ | | $< 10^{-5}$ |
| 0.04 | | $< 10^{-5}$ | | $< 10^{-5}$ | |

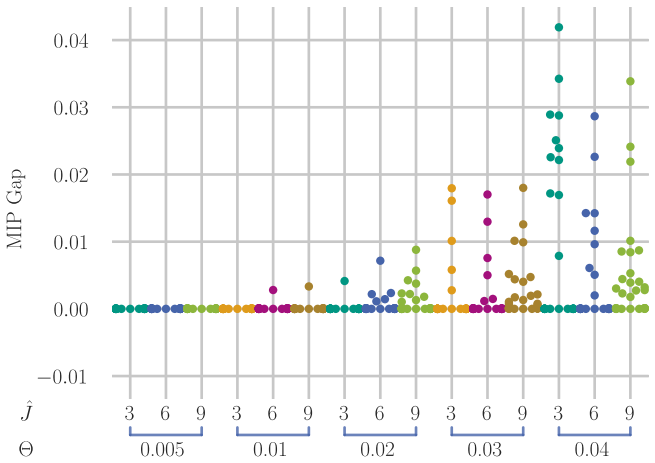



**Figure A.3:** MIP gaps for the various settings of $\hat{J}$ and $\Theta$ in the OM instance set. Every dot corresponds to one instance. Colors are used to distinguish the columns.

| $\Theta$ | 0.005 | | | 0.01 | | | 0.02 | | | 0.03 | | | 0.04 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\hat{J}$ | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 |
| Min | 0.57 | 0.54 | 0.57 | 0.56 | 0.5 | 0.49 | 0.52 | 0.44 | 0.41 | 0.52 | 0.44 | 0.38 | 0.52 | 0.44 | 0.38 |
| Max | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 |
| Mean | 0.84 | 0.83 | 0.83 | 0.81 | 0.78 | 0.78 | 0.8 | 0.77 | 0.76 | 0.8 | 0.77 | 0.75 | 0.8 | 0.77 | 0.75 |
| Median | 0.86 | 0.81 | 0.81 | 0.83 | 0.75 | 0.75 | 0.83 | 0.74 | 0.72 | 0.83 | 0.74 | 0.72 | 0.83 | 0.74 | 0.72 |
| Std. Dev. | 0.11 | 0.12 | 0.12 | 0.13 | 0.15 | 0.15 | 0.14 | 0.16 | 0.17 | 0.14 | 0.16 | 0.17 | 0.14 | 0.16 | 0.17 |

**Table A.5:** Statistics of the change in peak demand after optimization in the PS-Nonuniform set.

| $\Theta$ | 0.005 | | | 0.01 | | | 0.02 | | | 0.03 | | | 0.04 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\hat{J}$ | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 |
| Min | 0.84 | 0.8 | 0.8 | 0.83 | 0.77 | 0.75 | 0.81 | 0.73 | 0.7 | 0.8 | 0.72 | 0.7 | 0.8 | 0.71 | 0.69 |
| Max | 0.98 | 0.93 | 0.93 | 0.98 | 0.92 | 0.91 | 0.98 | 0.92 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| Mean | 0.9 | 0.88 | 0.88 | 0.89 | 0.84 | 0.83 | 0.88 | 0.81 | 0.79 | 0.93 | 0.88 | 0.88 | 0.98 | 0.98 | 0.97 |
| Median | 0.91 | 0.87 | 0.87 | 0.88 | 0.84 | 0.83 | 0.88 | 0.81 | 0.78 | 0.92 | 0.84 | 0.93 | 1.0 | 1.0 | 1.0 |
| Std. Dev. | 0.032 | 0.036 | 0.036 | 0.035 | 0.042 | 0.043 | 0.04 | 0.05 | 0.071 | 0.066 | 0.099 | 0.12 | 0.056 | 0.068 | 0.085 |

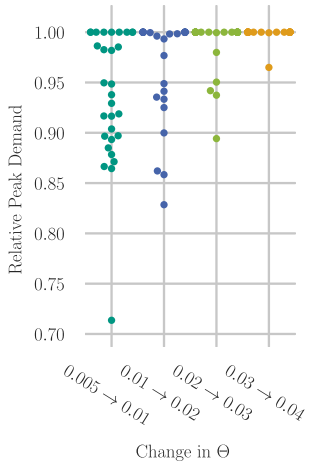
**Table A.6:** Statistics of the change in peak demand after optimization in the PS-Uniform set.

| $\Theta$ | 0.005 | | | 0.01 | | | 0.02 | | | 0.03 | | | 0.04 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\hat{J}$ | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 |
| Min | 0.51 | 0.48 | 0.48 | 0.47 | 0.4 | 0.37 | 0.47 | 0.34 | 0.27 | 0.47 | 0.29 | 0.25 | 0.47 | 0.28 | 0.24 |
| Max | 0.98 | 0.97 | 0.97 | 0.97 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 | 0.96 |
| Mean | 0.79 | 0.77 | 0.77 | 0.78 | 0.73 | 0.72 | 0.77 | 0.71 | 0.68 | 0.77 | 0.7 | 0.67 | 0.77 | 0.69 | 0.66 |
| Median | 0.85 | 0.81 | 0.8 | 0.83 | 0.79 | 0.75 | 0.82 | 0.76 | 0.72 | 0.82 | 0.76 | 0.72 | 0.82 | 0.76 | 0.72 |
| Std. Dev. | 0.14 | 0.15 | 0.15 | 0.14 | 0.17 | 0.18 | 0.14 | 0.19 | 0.21 | 0.14 | 0.19 | 0.22 | 0.14 | 0.2 | 0.22 |

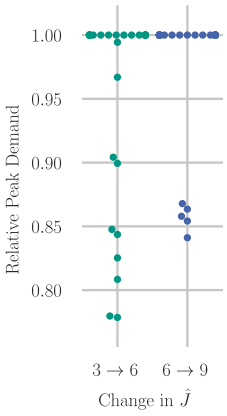
**Table A.7:** Statistics of the change in peak demand after optimization in the PSG set.

| $\Theta$ | 0.005 | | | 0.01 | | | 0.02 | | | 0.03 | | | 0.04 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\hat{J}$ | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 | 3 | 6 | 9 |
| Min | 0.97 | 0.97 | 0.97 | 0.52 | 0.94 | 0.94 | 0.89 | 0.89 | 0.88 | 0.84 | 0.85 | 0.85 | 0.82 | 0.81 | 0.81 |
| Max | 0.98 | 0.98 | 0.98 | 0.97 | 0.97 | 0.97 | 0.94 | 0.94 | 0.94 | 0.93 | 0.92 | 0.92 | 0.92 | 0.9 | 0.89 |
| Mean | 0.98 | 0.98 | 0.98 | 0.94 | 0.95 | 0.95 | 0.92 | 0.91 | 0.91 | 0.89 | 0.88 | 0.88 | 0.88 | 0.85 | 0.85 |
| Median | 0.98 | 0.98 | 0.98 | 0.95 | 0.95 | 0.95 | 0.92 | 0.91 | 0.91 | 0.89 | 0.88 | 0.88 | 0.88 | 0.85 | 0.85 |
| Std. Dev. | 0.0037 | 0.0037 | 0.0037 | 0.078 | 0.0069 | 0.0069 | 0.011 | 0.011 | 0.013 | 0.019 | 0.015 | 0.015 | 0.021 | 0.019 | 0.019 |

**Table A.8:** Statistics of the change in overshoot after optimization in the OM set.

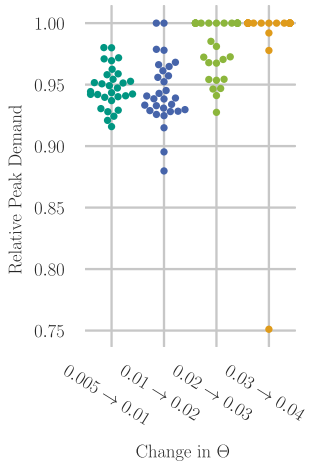



**(a)** Improvement in peak demand by increasing Θ while keeping $\hat{J}$ constant

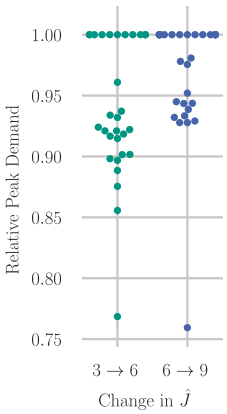



**(b)** Improvement in peak demand by increasing $\hat{J}$ while keeping Θ constant.

**Figure A.4:** Additional Results for set PS-Nonuniform, one point per instance. The columns are the different settings for $\hat{J}$ and Θ. The Y axis indicates the change in peak demand after optimization. Color indicates how well the instance could be optimized.

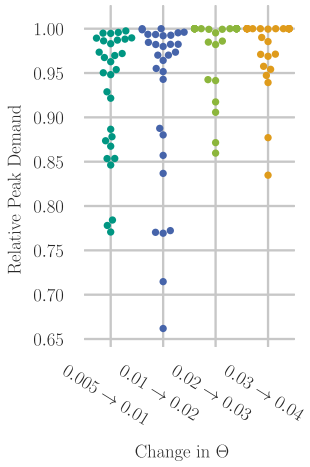



**(a)** Improvement in peak demand by increasing Θ while keeping $\hat{J}$ constant.

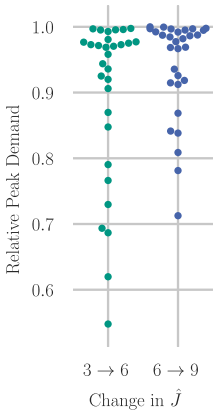



**(b)** Improvement in peak demand by increasing $\hat{J}$ while keeping Θ constant.

**Figure A.5:** Additional Results for set PS-Uniform, one point per instance. The columns are the different settings for $\hat{J}$ and Θ. The Y axis indicates the change in peak demand after optimization. Color indicates how well the instance could be optimized.

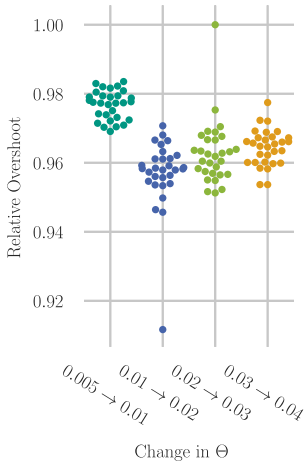



**(a)** Improvement in peak demand by increasing Θ while keeping $\hat{J}$ constant

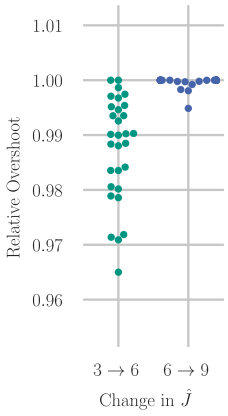



**(b)** Improvement in peak demand by increasing $\hat{J}$ while keeping Θ constant.

**Figure A.6:** Additional results for set PSG, one point per instance. The columns are the different settings for $\hat{J}$ and Θ. The Y axis indicates the change in peak demand after optimization. Color indicates how well the instance could be optimized.



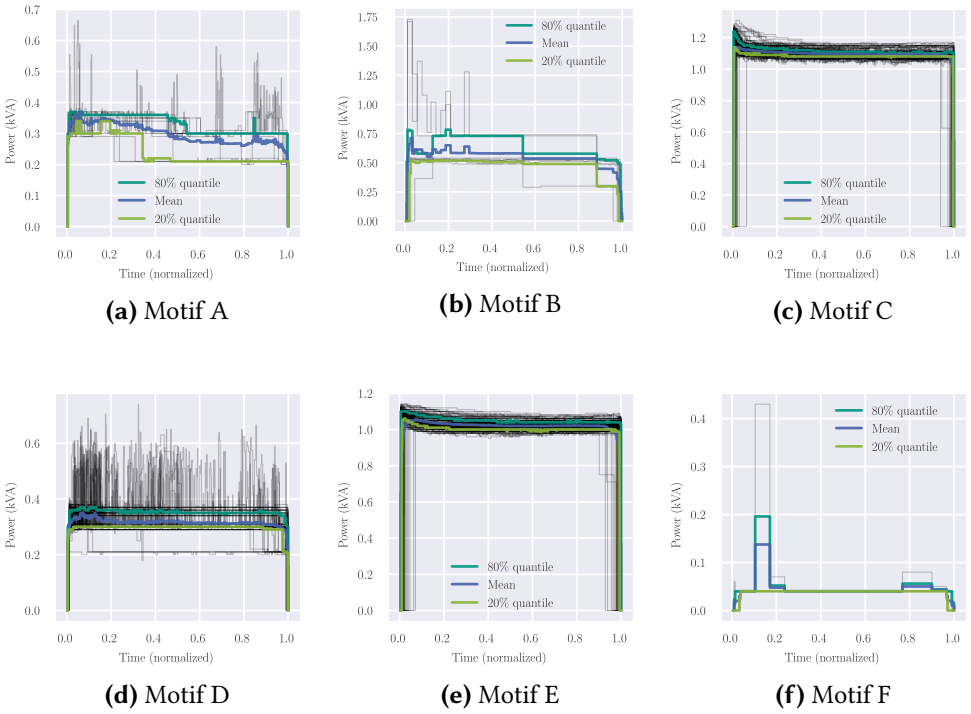

**(a)** Relative overshoot reduction by increasing Θ while keeping $\hat{J}$ constant




**(b)** Relative overshoot reduction by increasing $\hat{J}$ while keeping Θ constant.

**Figure A.7:** Additional results for set OM, one point per instance. The columns are the different settings for $\hat{J}$ and Θ. The Y axis indicates the change in overshoot after optimization. Color indicates how well the instance could be optimized.

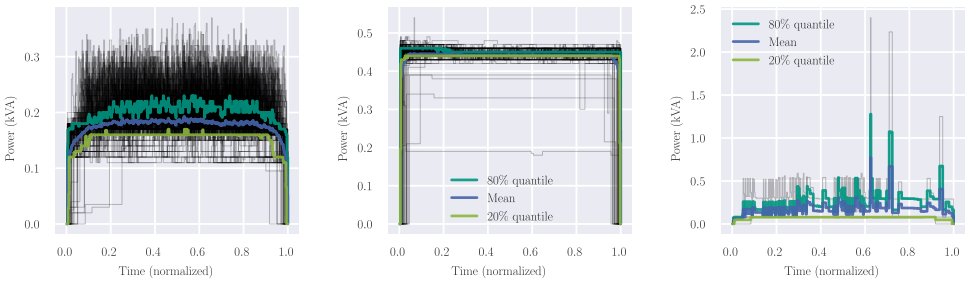
## A.3 Discovered Motifs



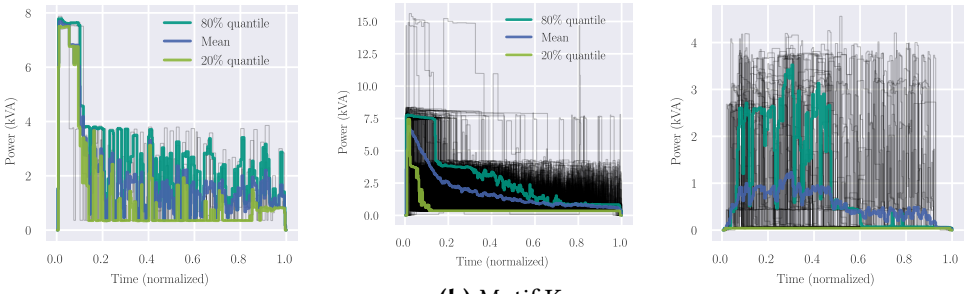

**(a)** Motif A **(b)** Motif B **(c)** Motif C

**(d)** Motif D **(e)** Motif E **(f)** Motif F

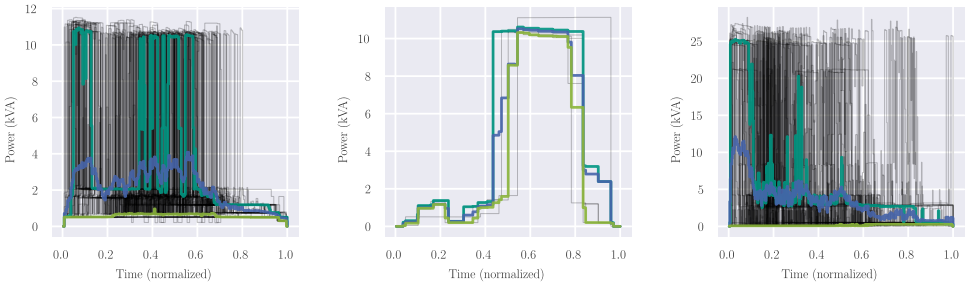
**Figure A.8:** All discovered motifs. Each black line indicates one occurrence of the respective motif.




**(g)** Motif G **(h)** Motif H **(i)** Motif I




**(j)** Motif J **(k)** Motif K **(l)** Motif L




**(m)** Motif M **(n)** Motif N **(o)** Motif O

**Figure A.8:** All discovered motifs. Each black line indicates one occurrence of the respective motif.

# B Appendix for: Industrial Demand Side Flexibility: A Benchmark Data Set

## B.1 Full Figures for Section 6.7.1

Here, we supply larger plots for the analysis performed in Section 6.7.1 and for all values for $k$ in $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20\}$. Note that in rare cases, the $\Delta(P_o, \tilde{P}_{o,k})$ value slightly increases with increasing $k$ for some occurrences. This is likely because the algorithm we used to optimize the block decomposition is not exact. We used $10^5$ iterations of sequential least-squares programming.

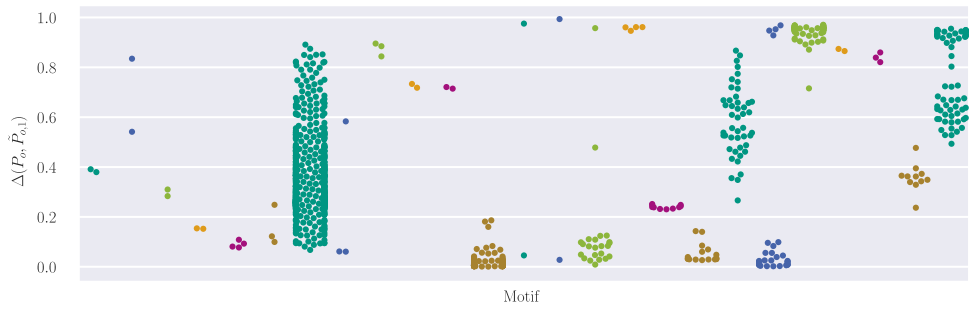



**Figure B.1:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 1$.

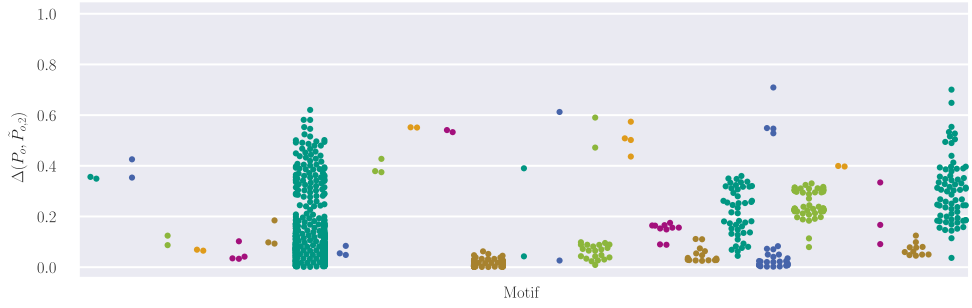



**Figure B.2:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 2$.

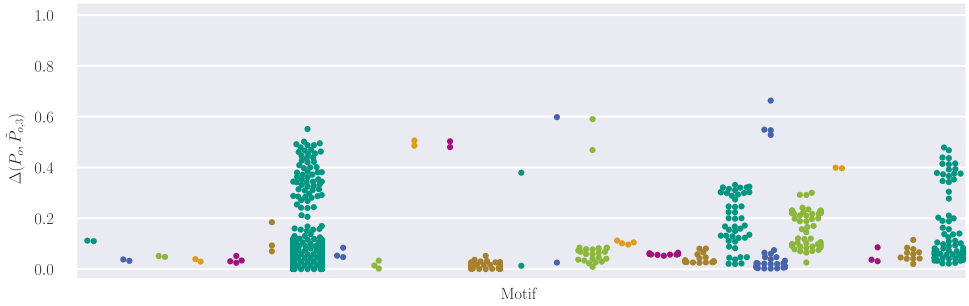



**Figure B.3:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 3$.

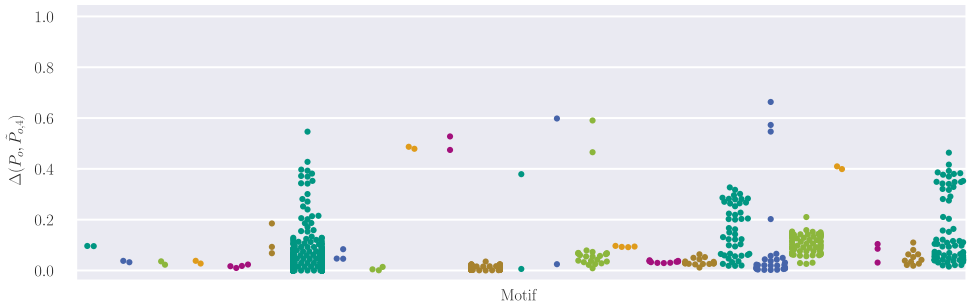



**Figure B.4:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 4$.

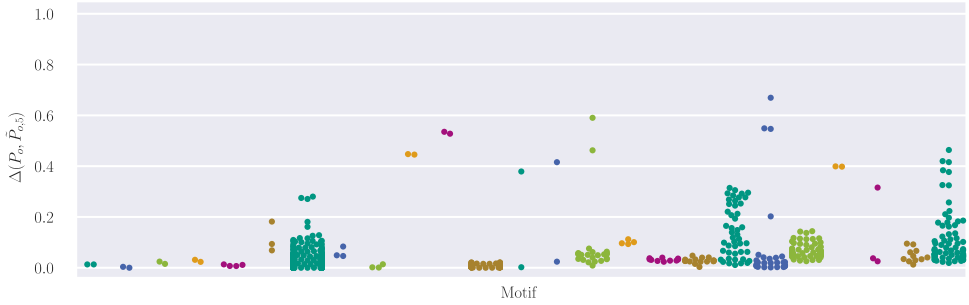



**Figure B.5:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 5$.

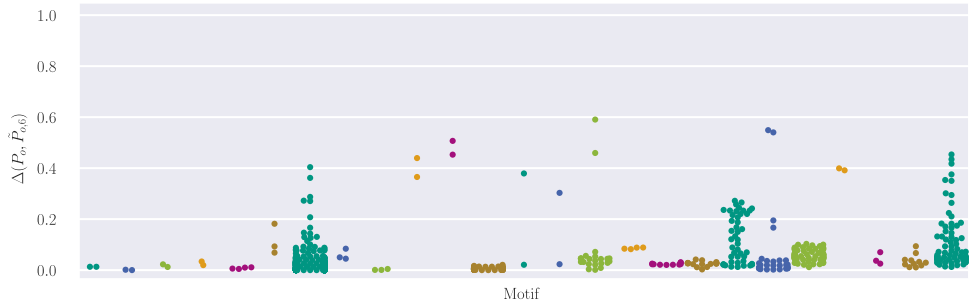



**Figure B.6:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 6$.

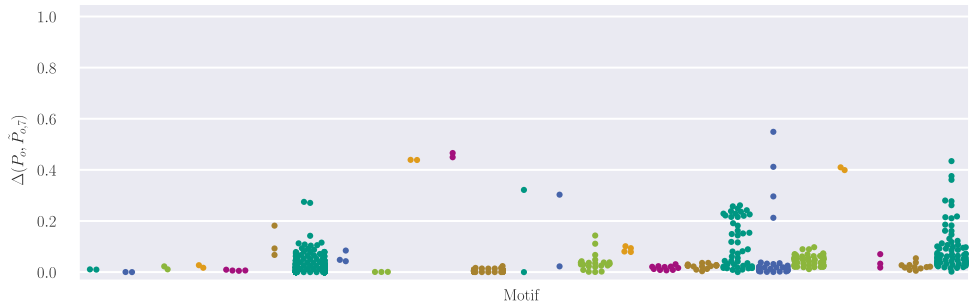



**Figure B.7:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 7$.

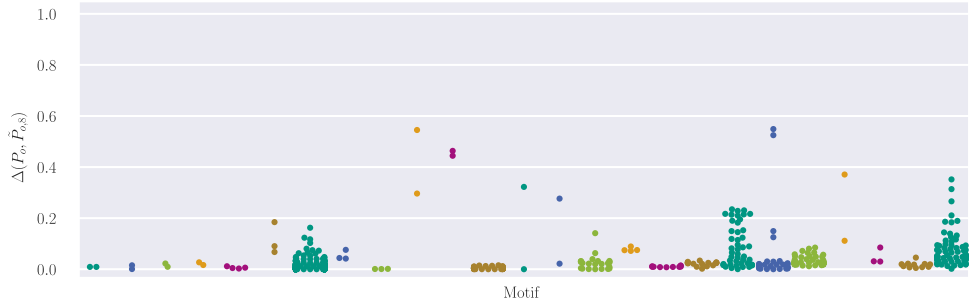



**Figure B.8:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 8$.

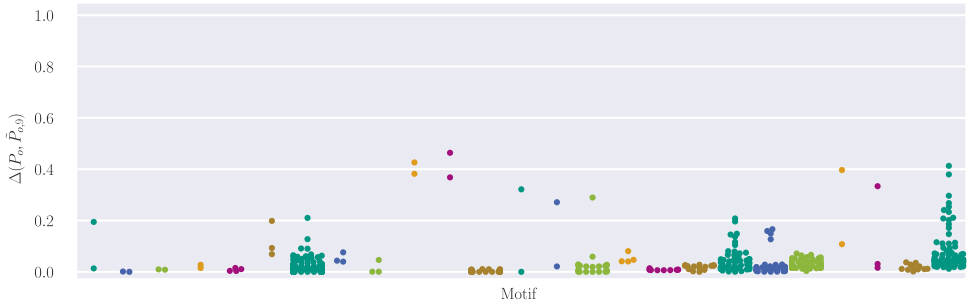



**Figure B.9:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 9$.

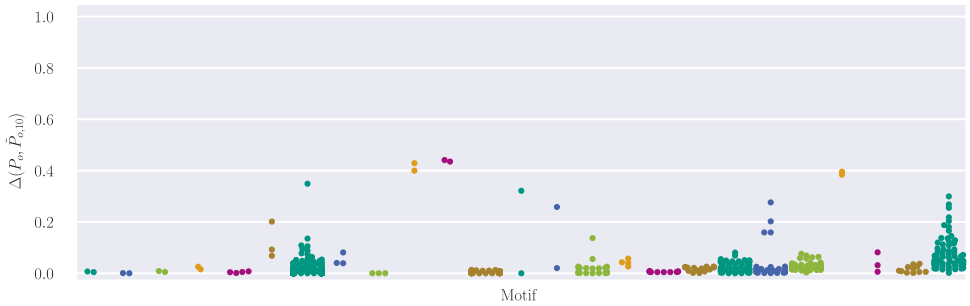



**Figure B.10:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 10$.

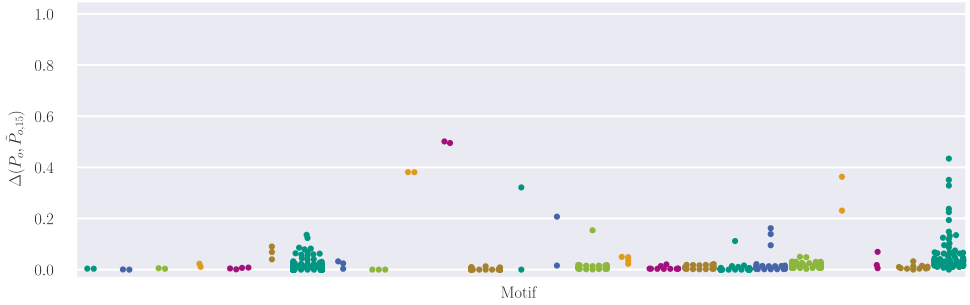



**Figure B.11:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 15$.

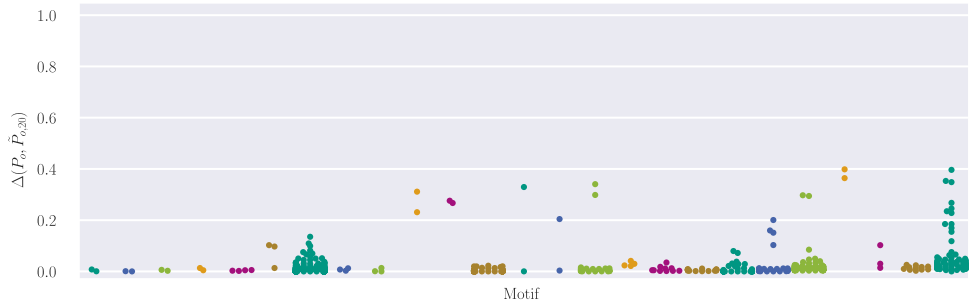



**Figure B.12:** $\Delta$ measures for each occurrence, ordered by motif (on the x axis), for $k = 20$.

# C Appendix for: Top-Down Weight-Balanced Trees

## C.1 Engineering Top-Down Weight-Balanced Trees: Code and Data Publication

All evaluated trees as well as all benchmarking code is implemented in C++17. We publish the code (including all benchmarking code) at

`https://github.com/tinloaf/ygg/`

Note that this is an ongoing project subject to changes. The exact code revision used in this chapter can be accessed at

`https://github.com/tinloaf/ygg/releases/tag/version_thesis`

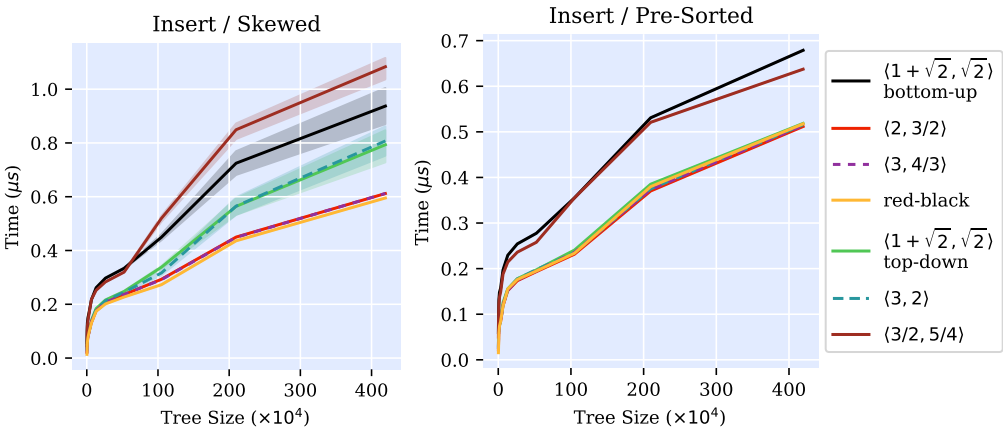
The code includes a file `README.md` with build instructions. After building, the directory "benchmark" contains all binaries necessary to reproduce our benchmarks. The file `BENCHMARKING.md` contains instructions on how to run the benchmarks.

We also publish all raw results we obtained from the benchmarks in a separate data publication [BW19c]. This publication can be accessed at

`https://publikationen.bibliothek.kit.edu/1000098852`

It also contains a detailed description of the data format output by the various benchmarking tools.

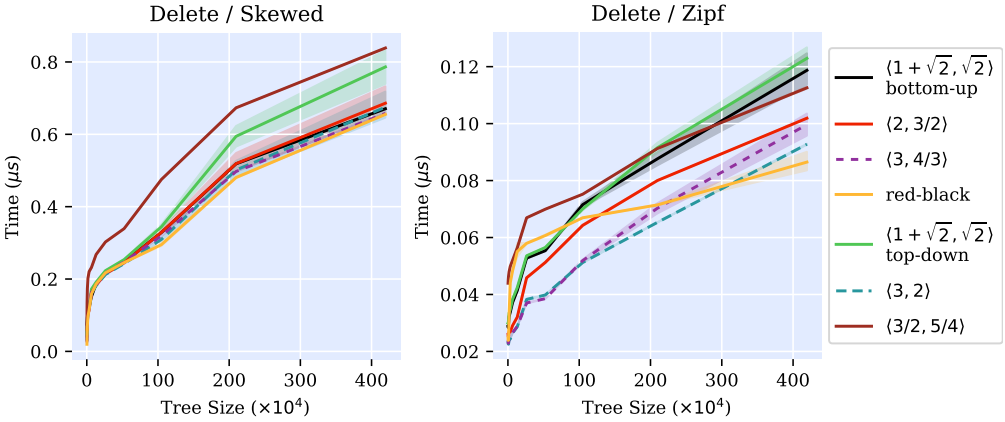
## C.2 Omitted Benchmark Plots




**(a)** Nodes' keys chosen as for the `skewed` case.

**(b)** Nodes' keys chosen as for the `pre-sorted` case.

**Figure C.1:** Times to insert 5% new nodes into trees of various sizes. The $x$ axis specifies the size of the base tree. The $y$ axis reports the time needed for a single insertion in microseconds. Shaded areas indicate standard deviation.




**(a)** Nodes' keys chosen as for the `skewed` case.

**(b)** Nodes' keys chosen as for the `zipf` case.

**Figure C.2:** Times to delete 5% nodes from trees of various sizes. The $x$ axis specifies the size of the base tree. The $y$ axis reports the time needed for a single deletion in microseconds. Shaded areas indicate standard deviation.

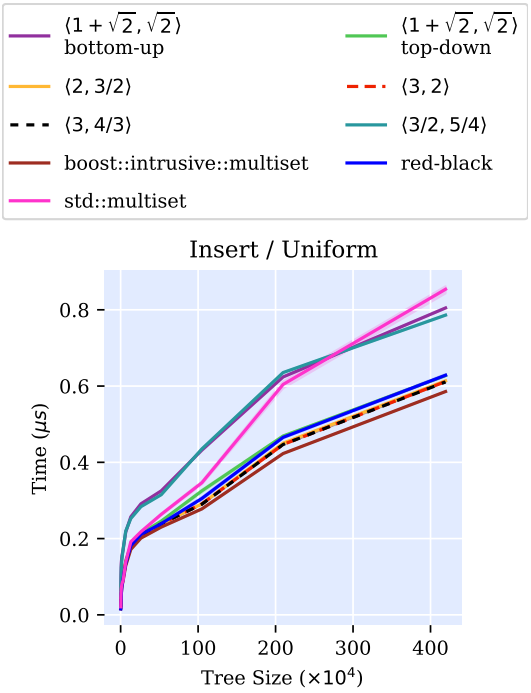



**Figure C.3:** Times to insert 5% new nodes into trees of various sizes. The $x$ axis specifies the size of the base tree. The $y$ axis reports the time needed for a single insertion in microseconds. Shaded areas indicate standard deviation. Node keys are chosen uniformly at random. Note that this plot includes `std::multiset` and `boost::intrusive::multiset`.

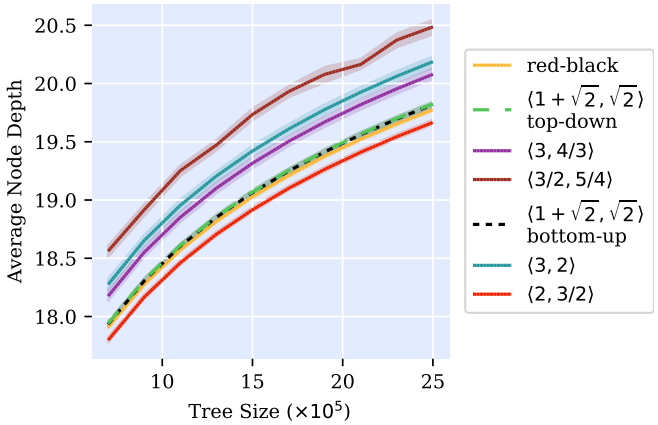



**Figure C.4:** Average node depth for trees of various sizes, with keys chosen for the `skewed` case. The $x$ axis specifies the size of the tree, the $y$ axis the average node depth. All nodes in every tree were randomly generated, removed once, had their key changed, and were reinserted. The solid lines indicate average values, the shaded areas the standard deviation.