Shaving Peaks by Augmenting the Dependency Graph

Lukas Barth
Institute of Theoretical Informatics,
Karlsruhe Institute of Technology
Karlsruhe, Germany
lukas.barth@kit.edu

Dorothea Wagner
Institute of Theoretical Informatics,
Karlsruhe Institute of Technology
Karlsruhe, Germany
dorothea.wagner@kit.edu

ABSTRACT

Demand Side Management (DSM) is an important building block for future energy systems, since it mitigates the non-dispatchable, fluctuating power generation of renewables. For centralized DSM 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.

CCS CONCEPTS

• **Mathematics of computing** → *Combinatorial optimization*; • **Theory of computation** → *Design and analysis of algorithms*; **Scheduling algorithms**; • **Applied computing** → Operations research; • **Hardware** → Smart grid.

KEYWORDS

scheduling, flexibility, smart grid, demand side management, demand response

ACM Reference Format:
Lukas Barth and Dorothea Wagner. 2019. Shaving Peaks by Augmenting the Dependency Graph. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems (e-Energy '19), June 25–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3307772.3328298

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 side management* (DSM). Demand side management 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 [22] defines DSM 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 wen system reliability is jeopardized."

One way to categorize DSM 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 (cf. Siano [21]). Another important dimension of DSM 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 cases 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 instances.

e-Energy '19, June 25–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6671-7/19/06...$15.00
https://doi.org/10.1145/3307772.3328298

1.1 Our contribution

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. [19] 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.

1.2 Related Work

The field of demand side management has received much attention lately. Among the numerous reviews of the field, a general survey is provided by Siano [21], while Vardakas et al. [24] provide a survey with focus on the methodology for implementing DSM. A survey with a stronger focus on the modelling of DSM problems is provided by Deng et al. [8]. A review by Good et al. [10] examines the challenges and enablers that DSM 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. [5]. 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. [6], 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 [27].

The problem considered in this paper 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 Weglarz [25]. The RACP has first been tackled by Möhring [18] and Demeulemeester [7]. Recently, Guldemond et al. [11] use an ILP to solve a variant of RACP, while Ranjbar [20] uses a metaheuristic based on path relinking.

There also is a considerable amount of work on scheduling with special regards to the smart grid. Barth et al. [1] provide an overview over various approaches based on mathematical programming. One of the earliest works is by Hsu and Su [13], 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. [19] use a metaheuristic to solve a problem similar to the one considered in this work. Yaw et al. [26] give two simple combinatoric algorithms for peak demand scheduling problems with certain constraints. Logenthiran et al. [15] use an evolutionary algorithm to schedule large amounts of loads in a simulated scenario.

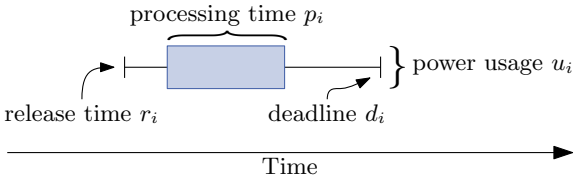

Figure 1: A graphical representation of a job in an S-RACP instance. With the release time, deadline and processing time as given, the box can be moved within the two whiskers, while the box's height represents the job's power demand.

2 PRELIMINARIES

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

2.1 The Problem

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

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, 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 power (the single resource of the project scheduling problem) j_i requires. Such a job is depicted in Figure 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.

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:

Problem 1 (Single-Resource Acquirement Cost Problem). *Given an Single-Resource Acquirement Cost Problem instance as J and $\mathcal{G}$ as defined above, find the optimal feasible schedule S^*.*

The S-RACP problem as defined above can be classified in the classification scheme by Herroelen et al. [12] as $1|cpm, \rho_j, \delta_j|av$.

2.2 Notation

We now introduce some notation used throughout this paper. 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.

Definition 1 (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 $j_i \in J$:

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

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 [4], 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 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., $E' \supseteq E$. 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'. □

3 SCHEDULING WITH AUGMENTED GRAPHS

In this section, we describe the SWAG algorithm using pseudo code 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 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 5.2 for how to choose them). Throughout this paper, we always typeset SWAG's parameters underlined. See Table 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 1. 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 [14]) 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 a dependency graph as representation: Given an S-RACP instance as J and $\mathcal{G} = (J, E)$ as defined in Section 2.2, we use a graph $G = (J, E')$ with $E' \supseteq D$ as representation. Then, we use the generation of a left-shifted schedule as described in Section 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 3.1, hiding much of the detail. We then present in sections 3.2 and 3.3 more detailed descriptions of several aspects as well as some insights into how to make the algorithm more efficient.

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 20. 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, $start$, the earliest possible starts for every job admitted by G, 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 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).

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 (cf. 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

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

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. We only ever insert feasible edge candidates, thereby making sure that the left-shifted schedule of G always stays feasible.

Edges are selected in line 9. Checking whether an edge is feasible is straightforward: since the *start* of u and the *latestFinish* of v are known, a useful edge candidate (j_u, j_v) is feasible if and only if $start[u] + p_u + p_v \leq latestFinish[v]$.

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).

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 cannot 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.

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) (cf. lines 9–12). If not, the algorithm tries to make a different edge candidate feasible, for up to deletionTrials trials.

Algorithm 1: The SWAG Algorithm

```
   Data: 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         end
16     else
17         e ← randomSelection(candidateEdges);
18         G.insert(e);
19     end
20 end
```

Algorithm 2: UnblockByDeletion

```
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         end
13     end
14     return Null;
15 end
```

The searches in lines 6 and 8 are equivalent, so we only describe *findDeletionEdgesForward*, 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*.

3.3 Optimizations

The heuristic as described so far should be 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 5.3.

Algorithm 3: FindDeletionEdgesForward

```
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 ← ⟨((⊥, 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));
10             end
11         end
           /* Pretend to remove edges to determine
              new latest finishs                        */
12         G.removeEdges(edgesToDelete);
13         newLF ←computeLatestFinish (startJob);
14         G.insertEdges(edgesToDelete);
15         movement ← newLF − latestFinish[startJob];
16         quality ← |edgesToDelete| + (undermovePenalty ·
            max(0, overlap − movement));
17         if quality < bestQuality then
18             bestSolution ← markedEdges;
19             bestQuality ← quality;
20         end
21     end
22     return bestSolution, bestMovement;
23 end
```

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.

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 Kreveld and Overmars [23] 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 [23], 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.

4 COMPETITOR ALGORITHM: GRASP

We implemented the scheduling heuristic by Petersen et al. [19] 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 paper.

The authors present and evaluate multiple variants of their algorithm. We also 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 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 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 (cf. Table IV in [19]). This could be explained by the fact that our instances are larger than the test instances in [19], 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 [19] seem to favor the weighted variant. The optimal parameters determined by our tuning are shown in Table 1. Consistent with the findings in [19] is that the smallest possible values are chosen for m and l, while rather high values are chosen for the number of hill climber iterations.[2] This means that effectively, the hill climber does the main part of the algorithm's work.

[2]In [19], the authors use a time limit instead of an iteration limit for the hill climber. We used an iteration limit for finer control.

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

Table 1: Chosen parameters for GRASP. Corresponds to Table IV in [19].

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 [19] 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.

5 EVALUATION

We perform three kinds of evaluation of the SWAG algorithm: First, we compare 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 investigate the influence that various SWAG parameters and properties of the instances have on the computed solution quality. Finally, we compare SWAG to the GRASP algorithm by Petersen et al. [19].

All computations 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 run time 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 [2]. 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 https://github.com/kit-algo/TCPSPSuite/.

5.1 Instance Sets

We use a total of three sets of instances to evaluate SWAG: A set of small instances, called I_{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_{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 generate a third set of instances with larger window sizes, called I_{window}.

All sets are generated based on the HIPE data set by Bischof et al. [3] that 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 [3]. From this factory data, we get 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 detect patterns (which we call *processes*) in this data using a technique adapted from Ludwig et al. [17]. The technique is very similar to the technique described by Ludwig et al. [16] 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) is fit to the duration and energy components of the process' point set. We use 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 are first clustered using the DBSCAN[3] algorithm [9]. Then, the 0.1 and 0.9 quantile of every cluster is determined and taken as the lower respective upper limit of a uniform distribution. This results in one uniform distribution per cluster. The uniform distributions are weighted by the number of points in the respective cluster.

To generate a job as defined in Section 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_{large}, we do everything as described above, and generate between 500 and 1500 jobs per instance, for a total of 300 jobs. For the window growth distribution, we use 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_{small}, we do everything as for I_{large}, only that we limit the number of jobs to between 50 and 150, and set the time horizon to 3 days. We generate a total of 200 instances.

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

5.2 Parameter Tuning

Both the SWAG and the GRASP algorithm need to be parameterised. To have a fair comparison, we determine 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 ρ_i, we select a set of l_i possible values $\Gamma_i = \{\gamma_{i,1}, \gamma_{i,2}, \dots \gamma_{i,l_i}\}$. The Γ_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 ρ_i falls outside of the range of values in Γ_i, one of the two extreme values at its ends should be chosen. In this case, we repeat the tuning with an adjusted Γ_i. All the Γ_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

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

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

Table 2: Chosen parameters for SWAG. We use separate parameter sets for the set of small instances and the sets of large instances.

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. Let P and Q be two configurations, then $dist(P, Q) = \sum_i |P_i - Q_i|$. 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

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

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

The results of tuning SWAG can be found in Table 2. Note that we use separate sets of parameters for small and large instances. The tuned parameters of GRASP can be found in Table 1. Since we use GRASP only on the large instances, there is only one set of parameters here.

5.3 SWAG analysis

This section presents a first analysis of SWAG, starting by comparing results for the I_{small} instance set computed by SWAG to results computed using a mixed-integer linear program (MIP) adapted from [1]. The MIP was optimized for 1800 seconds using 16 parallel threads, while SWAG was executed for 5 seconds in a single thread. The MIP finds optimal solutions for 97 of the 200 instances, and closes 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 2 shows a dot for every instances in I_{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_{small}, SWAG computes the optimal solution within 5 seconds on many instances, especially on the smaller ones. In total, 99 instances are solved to optimality by SWAG. Of the 100 instances with less that 100 jobs, 63 are solved to optimality.

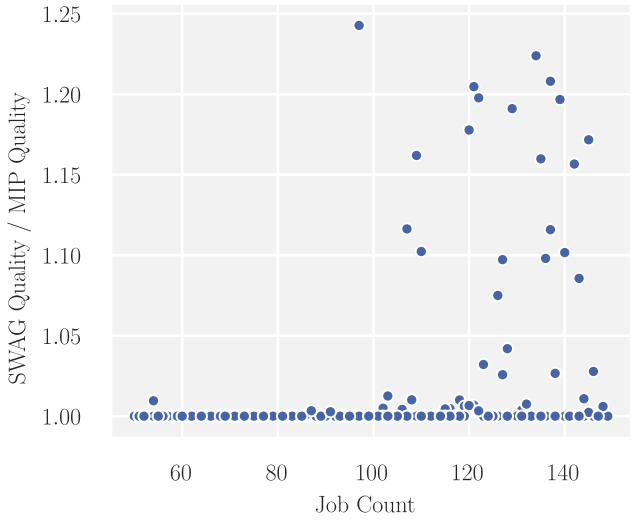

Figure 2: Quality computed by SWAG compared to MIP solution on I_{small}, ordered by number of jobs in the instance.

As the instances grow in size, more instances cannot be solved to optimality anymore by SWAG, with the factor between the MIP solution and the SWAG solution reaching 24% in the worst case.

5.3.1 Parameter Impact. We now investigate how the change of various parameters impacts the performance of SWAG. This analysis is done on the I_{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 completePropagationAfter parameter (cf. Table 2 and Section 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 (cf. 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.

5.3.2 Convergence Speed. We now analyze how fast the solutions found by SWAG converge. We do this analysis on the I_{large} instance set, using a maximum run time of 10 seconds. Every 0.1 seconds, we sample the currently found best solution. After the run is completed, we normalize the solution qualities of all samples taken during that run by the final solution quality of the respective run, i.e., a value

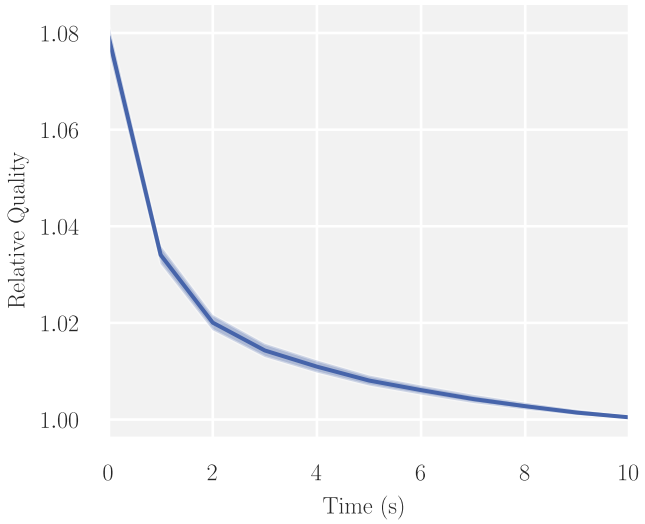

Figure 3: Convergence speed of SWAG. The y axis indicates the quality of the best found solution at a certain point in time relative to the best solution found after 10 seconds. The blue line is the median over all instances, the shaded area indicates the 0.1 / 0.9 quantiles.

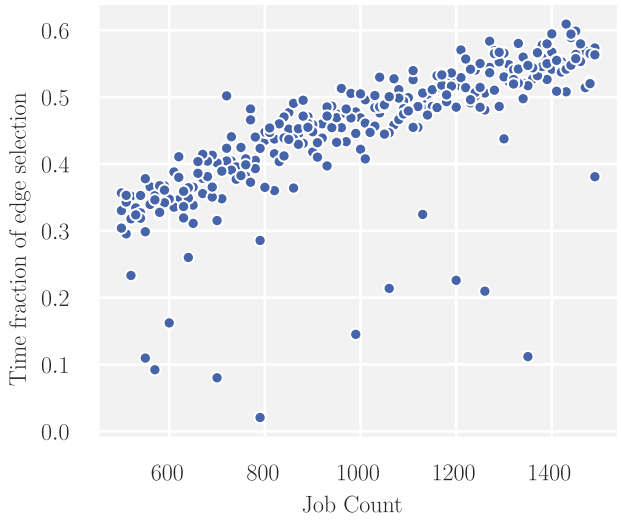

Figure 4: Fraction of run time spent on finding feasible edge candidates

of 1.1 would indicate a solution that is 10% worse than the solution found after 10 seconds.

Figure 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 traces lie above or below the blue band. We can see that already with the first samples after 0.1 seconds, we are 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 take a look at where the SWAG algorithm spends the majority of its time. Figure 4 shows the fraction of the total run time that SWAG spends 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 increases 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.

5.4 Comparison SWAG vs. GRASP

We now present how SWAG compares to GRASP on the instance set I_{large} with parameters chosen as shown in tables 1 and 2. Figure 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 performs slightly better than SWAG, in the best case by about 18%. For 268 of 300 instances, SWAG outperforms GRASP, by a factor of up to 3. Figure 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 computes 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 examine the behavior of SWAG on instances with larger window sizes on the instance set I_{window}, see Section 5.1 on how we enlarged the jobs' windows. The performance of SWAG compared to GRASP is depicted in Figure 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 6 and Figure 7, SWAG's better scalability in terms of instance size seems to be corroborated by instances with larger window sizes.

6 CONCLUSION

In this paper, 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 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

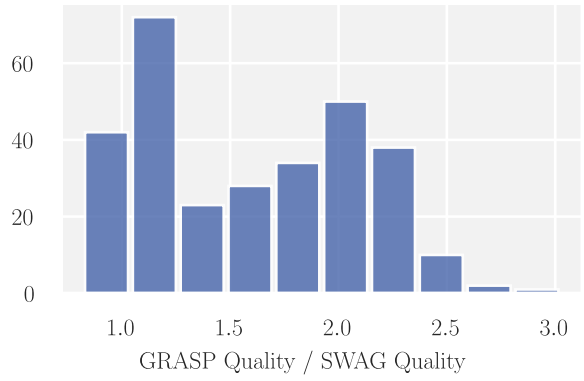

Figure 5: Histogram of the relative score of GRASP compared to SWAG on I_{large}.

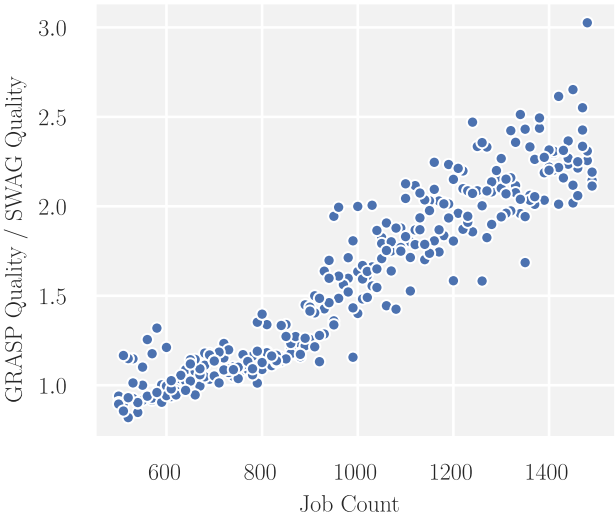

Figure 6: Comparison between SWAG and GRASP by job count, on I_{large}. Every dot is one instance. The y coordinate corresponds to the solution computed by GRASP divided by the solution computed by SWAG.

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

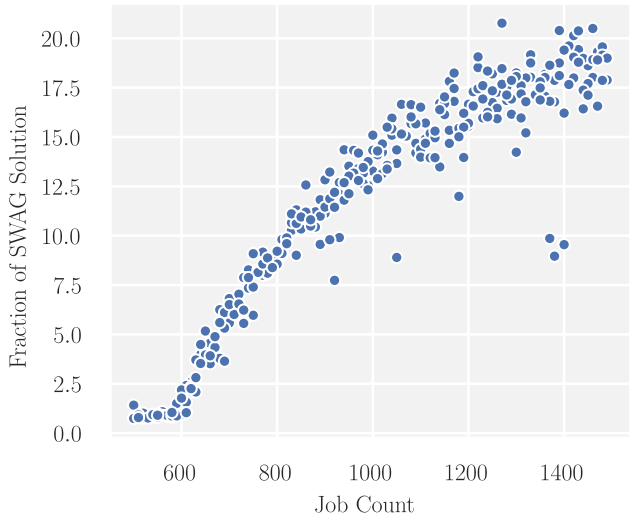

Figure 7: Comparison between SWAG and GRASP by job count, on I_{window}. Every dot is one instance. The y coordinate corresponds to the solution computed by GRASP divided by the solution computed by SWAG.

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.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) as part of the Research Training Group GRK 2153: Energy Status Data – Informatics Methods for its Collection, Analysis and Exploitation. The authors thank Nicole Ludwig for help with the generation of benchmark instances.

REFERENCES

[1] Lukas Barth, Nicole Ludwig, Esther Mengelkamp, and Philipp Staudt. 2018. A comprehensive modelling framework for demand side flexibility in smart grids. *Computer Science - Research and Development* 33, 1 (01 Feb 2018), 13–23. https://doi.org/10.1007/s00450-017-0343-x

[2] Lukas Barth and Dorothea Wagner. 2019. Dataset accompanying "Shaving Peaks by Augmenting the Dependency Graph". KITOpen Repository. https://doi.org/10.5445/IR/1000094106

[3] Simon Bischof, Holger Trittenbach, Michael Vollmer, Dominik Werle, Thomas Blank, and Klemens Böhm. 2018. HIPE – an Energy-Status-Data Set from Industrial Production. In *Proceedings of ACM e-Energy (e-Energy 2018)*. ACM, New York, NY, USA. https://doi.org/10.1145/3208903.3210278

[4] Peter Brucker and Sigrid Knust. 2011. *Complex Scheduling* (2nd ed.). Springer Publishing Company, Incorporated.

[5] Bo Chen, Chris N. Potts, and Gerhard J. Woeginger. 1998. *A Review of Machine Scheduling: Complexity, Algorithms and Approximability.* Springer US, Boston, MA, 1493–1641. https://doi.org/10.1007/978-1-4613-0303-9_25

[6] Mark Cieliebak, Thomas Erlebach, Fabian Hennecke, Birgitta Weber, and Peter Widmayer. 2004. Scheduling With Release Times and Deadlines on A Minimum Number of Machines. In *Exploring New Frontiers of Theoretical Informatics*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). Springer US, Boston, MA, 209–222.

[7] Erik Demeulemeester. 1995. Minimizing Resource Availability Costs in Time-Limited Project Networks. *Management Science* 41, 10 (1995), 1590–1598. http:

//www.jstor.org/stable/2632739
[8] Ruilong Deng, Zaiyue Yang, Mo-Yuen Chow, and Jiming Chen. 2015. A Survey on Demand Response in Smart Grids: Mathematical Models and Approaches. *IEEE Transactions on Industrial Informatics* 11, 3 (June 2015), 570–582. https://doi.org/10.1109/TII.2015.2414719
[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. AAAI Press, 226–231.
[10] Nicholas Good, Keith A. Ellis, and Pierluigi Mancarella. 2017. Review and classification of barriers and enablers of demand response in the smart grid. *Renewable and Sustainable Energy Reviews* 72 (2017), 57 – 72. https://doi.org/10.1016/j.rser.2017.01.043
[11] T. A. Guldemond, Johann L. Hurink, Jacob J. Paulus, and Marco J. Schutten. 2008. Time-constrained project scheduling. *Journal of Scheduling* 11, 2 (2008), 137–148. https://doi.org/10.1007/s10951-008-0059-7
[12] Willy Herroelen, Erik Demeulemeester, and Bert De Reyck. 1999. *A Classification Scheme for Project Scheduling*. Springer US, Boston, MA, 1–26. https://doi.org/10.1007/978-1-4615-5533-9_1
[13] Yuan-Yih Hsu and Chung-Ching Su. 1991. Dispatch of direct load control using dynamic programming. *IEEE Transactions on Power Systems* 6, 3 (Aug 1991), 1056–1061. https://doi.org/10.1109/59.119246
[14] Rainer Kolisch. 1996. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research* 90, 2 (1996), 320 – 333. https://doi.org/10.1016/0377-2217(95)00357-6
[15] Thillainathan Logenthiran, Dipti Srinivasan, and Tan Z. Shun. 2012. Demand Side Management in Smart Grid Using Heuristic Optimization. *IEEE Transactions on Smart Grid* 3, 3 (Sep. 2012), 1244–1252. https://doi.org/10.1109/TSG.2012.2195686
[16] Nicole Ludwig, Lukas Barth, Dorothea Wagner, and Veit Hagenmeyer. 2019. Industrial Demand-Side Flexibility: A Benchmark Data Set. In *Proceedings of ACM e-Energy (e-Energy 2019)*. ACM, New York, NY, USA. (to appear).
[17] Nicole Ludwig, Simon Waczowicz, Ralf Mikut, and Veit Hagenmeyer. 2017. Mining flexibility patterns in energy time series from industrial processes. In *Proceedings. 27. Workshop Computational Intelligence, Dortmund, 23.-24. November 2017*. KIT Scientific Publishing, 13.
[18] Rolf H. Möhring. 1984. Minimizing Costs of Resource Requirements in Project Networks Subject to a Fixed Completion Time. *Operations Research* 32, 1 (1984), 89–120. https://doi.org/10.1287/opre.32.1.89
[19] Mette K. Petersen, Lars H. Hansen, Jan Bendtsen, Kristian Edlund, and Jakob Stoustrup. 2014. Heuristic Optimization for the Discrete Virtual Power Plant Dispatch Problem. *IEEE Transactions on Smart Grid* 5, 6 (Nov 2014), 2910–2918. https://doi.org/10.1109/TSG.2014.2336261
[20] Mohammad Ranjbar. 2013. A path-relinking metaheuristic for the resource levelling problem. *Journal of the Operational Research Society* 64, 7 (01 Jul 2013), 1071–1078. https://doi.org/10.1057/jors.2012.119
[21] Pierluigi Siano. 2014. Demand response and smart grids—A survey. *Renewable and Sustainable Energy Reviews* 30 (2014), 461 – 478. https://doi.org/10.1016/j.rser.2013.10.022
[22] U.S. Department of Energy. 2006. Benefits of demand response in electricity markets and recommendations for achieving them. (2006).
[23] Marc J. van Kreveld and Mark H. Overmars. 1993. Union-copy Structures and Dynamic Segment Trees. *J. ACM* 40, 3 (July 1993), 635–652. https://doi.org/10.1145/174130.174140
[24] John S. Vardakas, Nizar Zorba, and Christos V. Verikoukis. 2015. A Survey on Demand Response Programs in Smart Grids: Pricing Methods and Optimization Algorithms. *IEEE Communications Surveys Tutorials* 17, 1 (Firstquarter 2015), 152–178. https://doi.org/10.1109/COMST.2014.2341586
[25] Jan Weglarz. 2012. *Project Scheduling: Recent Models, Algorithms and Applications*. Springer US. https://books.google.de/books?id=uFDtBwAAQBAJ
[26] Sean Yaw, Brendan Mumey, Erin McDonald, and Jennifer Lemke. 2014. Peak demand scheduling in the Smart Grid. In *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. 770–775. https://doi.org/10.1109/SmartGridComm.2014.7007741
[27] Guosong Yu and Guochuan Zhang. 2009. Scheduling with a minimum number of machines. *Operations Research Letters* 37, 2 (2009), 97 – 101. https://doi.org/10.1016/j.orl.2009.01.008