SCHEDULING FOR OPTIMIZED NETWORK RESOURCE UTILIZATION

#SMARTGRID #CLOUD

by

Sean Yaw

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April, 2017

©COPYRIGHT

by

Sean Yaw

2017

All Rights Reserved

TABLE OF CONTENTS

1. INTRODUCTION 1

Motivation 1
Contributions 2
Power Scheduling 2
Cloud Resource Optimization 4
Organization 5

2. PEAK DEMAND MINIMIZATION 6

Background 6
Related Work 8
Problem Formulation 10
Computational Complexity 11
Algorithms 12
An Optimal FPT Algorithm 12
Job Overlap Lists 12
Configuration Generation 14
Feasibility Testing 15
Schedule Building 16
Fixed Parameter Tractability 16
A Greedy Heuristic Algorithm 17
A Common Execution Window Algorithm 18
PDM-CW for Online Scheduling 23
A Non-nesting Execution Window Algorithm 24
Experimental Results 28
Staggered Execution Window Evaluation 28
Common Execution Window Evaluation 29
Conclusions 30

3. PEAK DEMAND MINIMIZATION WITH RENEWABLES 31

Background 31
Related Work 34
Power Job Scheduling 34
Renewable Energy Forecasting 35
Problem Formulation 35
Computational Complexity 38
Algorithms 38
An Optimal Algorithm 38

TABLE OF CONTENTS — CONTINUED

Configuration Lists 39
Configuration Trees 40
Dynamic Programming 40
Branch-and-Bound 42
A Randomized Rounding Algorithm 44
Integer Linear Programming Formulation 44
Randomized Rounding 46
A Greedy Heuristic Algorithm 49
Continuous to Discrete Transformation 49
Experimental Results 52
Conclusions 55

4. CLOUD RESOURCE ALLOCATION 56

Background 56
Related Work 59
Distributed Resource Management 59
Related Optimization Problems 61
Problem Formalization 62
Algorithms 65
A Greedy Algorithm 65
Group Churn Considerations 67
Multicast Considerations 68
Experimental Results 68
Group Churn Evaluation 71
Conclusions 73

5. TURNSTILE PLACEMENT PROBLEM 74

Background 74
Related Work 76
Problem Formulation 79
Turnstile Placement for a Single Commodity 80
Turnstile Problem Complexity 82
Algorithms 85
An Approximation Algorithm 85
A Maximum Spanning Tree Approach 89
A Multipath to Single Path Translation 90
Experimental Results 93

TABLE OF CONTENTS — CONTINUED

Number of Turnstiles Placed 94
Placement Effectiveness 97
Conclusions 99

REFERENCES CITED 101

LIST OF TABLES

Table Page

3.1 Algorithm performance summary 55

LIST OF FIGURES

Figure Page

2.1 Non-preemptive power jobs to be scheduled 11
2.2 Job 0 overlaps job 1 due to job 2's early arrival time 13
2.3 Compatible configurations 15
2.4 Example job schedule 18
2.5 Performance of PDM scheduling algorithms 29
3.1 Probabilistic wind energy forecast from [25] 36
3.2 Sample jobs for scheduling 39
3.3 Sample configuration trees with $c(l)$ and $b(m)$ pointers 41
3.4 Performance of PDM-R scheduling algorithms 54
4.1 Proxy network to decrease end-to-end latency for clients 57
4.2 DOMINATING-SET reduction to CGP 64
4.3 Performance of static and churn group instances 69
5.1 Sample TP instance with two commodities 80
5.2 Reduction of SET-COVER instance to TP instance 84
5.3 Path and pathlet decomposition 91
5.4 TP algorithm evaluation on small network instances 95
5.5 TP algorithm evaluation on large network instances 96
5.6 TP algorithm evaluation with variable number of paths 97
5.7 TP algorithm monitoring benefit evaluation 99

ABSTRACT

The performance of distributed applications is heavily dependent on the interplay between the applications and the underlying network. Disparity between the requirements of the applications and the capabilities of the network leads to degraded application performance, which in turn results in a drop in application usage or revenue. For example, many real-time interactive applications require lower latency than the public Internet provides, resulting in a poor experience for application users. At other times though, applications fail to effectively utilize all network capabilities. For example, conventional electrical appliances are currently unable to leverage the increased communication capabilities provided by the future smart power grid to decrease costs or modify consumption. Scheduling is an optimization technique to temporally and spatially allocate resources in such a way as to achieve some desired parameter optimization, such as minimized cost. In this dissertation, I study the use of scheduling techniques to counteract application performance degradation present due to the disparity between application requirements and network capabilities. I explore this disparity in both the smart grid and cloud networks, and propose novel algorithms that rely on numerous algorithmic techniques to realize application performance increases.

1

INTRODUCTION

Motivation

There is a disparity between the capabilities of modern networks and the requirements of the applications they support. On the one hand, the ongoing effort to add appliance-to-provider communication capabilities to the legacy power grid will enable unprecedented control over the generation and consumption of power. Conventional electrical appliances do not communicate with the energy providers, they just draw power as they need it. As such, the appliances are unable to leverage this newfound communication capability to reduce energy consumption or save cost. On the other hand, modern Internet applications require low latency communication to enable a responsive user experience. This latency requirement is challenging for the current Internet to provide due to factors outside of the application's control (e.g. public Internet congestion and circuitous forwarding routes). Failure to meet these latency requirements leads to degraded application user experience and an eventual loss of revenue as application usage drops.

Intelligently scheduling the utilization of resources can compensate for the application performance degradation due to the disparity between capabilities and requirements of networks and their applications. Scheduling is a broadly applicable resource and temporal optimization technique that lends itself to networked applications, as many resources are subject to competing interests and many network factors influencing application performance lie outside the reach of the applications. The scheduling of resources can increase the efficiency of applications, thereby reducing the need to upgrade infrastructure to achieve a boost in performance. Scheduling can also be used to leverage existing network capabilities not currently employed

by applications, resulting in increased application efficiency. In this dissertation, I explore various methods of scheduling for improving network efficiency in cloud networks and the smart grid. These methods leverage diverse algorithmic techniques and result in approximation algorithms that improve on previous theoretical results. I also introduce versatile heuristics that demonstrate good performance. All solutions presented are subjected to extensive simulations on real-world data and conditions to determine their viability.

Contributions

This section details the specific contributions made by the research presented in this dissertation. These contributions are generally categorized as relating to smart grid power scheduling or cloud resource optimization, and are outlined below.

Power Scheduling

- A reduction showing the *Peak Demand Minimization* (PDM) problem to be NP-hard to approximate within a ratio of 2 by reducing it from the *Scheduling with Release Times and Deadlines on a Minimum Number of Machines* (SRDM) problem.

- An optimal dynamic programming algorithm for the PDM problem that is fixed-parameter tractable, where the exponential parameter is the maximal number of overlapping jobs.

- A 4-approximation algorithm for the PDM problem that uses a geometric argument to establish its performance guarantee for the special case where each job has the same arrival time and deadline. Prior to these results, a 7.82-approximation algorithm was the best existing algorithm [91] for this case.

- A $O(\log \Delta)$-approximation algorithm for the PDM problem, where Δ is the ratio of the widest job execution window to the narrowest, for the case where job windows can be staggered, but are non-nesting.

- Formulation the *Peak Demand Minimization with Renewables* (PDM-R) problem and a proof that its complexity is at least as high as the related PDM problem.

- An optimal fixed-parameter tractable algorithm for the PDM-R problem with improved performance on PDM instances over the fixed-parameter tractable algorithm presented for the PDM problem. This performance improvement is achieved using a branch-and-bound technique to limit the search space of potential solutions.

- A linear programming relaxation scheme for the PDM-R problem. This scheme, in concert with Chernoff Bounds also provides the first approximation (ratio of $O\big(\frac{\log n}{\log \log n}\big)$) algorithm for the PDM problem.

- An online heuristic for the PDM-R problem that greedily schedules jobs so that they cost the least amount of non-renewable energy.

- An offline heuristic for both the PDM and PDM-R problems that considers the amount of flexibility each job has prior to scheduling greedily.

- Through evaluations of all of these solutions using real world test scenarios generated with current energy disaggregation research.

Cloud Resource Optimization

- Formulation of the *Cooperative Group Provisioning* (CGP) problem and a proof that is it NP-hard to approximate within a constant ratio via a reduction from the *Dominating Set* problem.

- A greedy algorithm for the CGP problem that sequentially adds group members to the solution by finding the connection path that contributes the least to the overall cost from the k shortest (by latency) paths.

- A greedy algorithm for the CGP problem made to handle client churn and multicast traffic by utilizing paths for entire connection lifespans (for churn) and by modifying the underlying graph representation to share link costs (for multicast).

- An evaluation of the CGP greedy algorithm with an extensive set of simulations on networks configured with the connectivity and pricing models of three major cloud service providers.

- A proof that the general *Turnstile Placement* (TP) problem is NP-hard to approximate within a bound of $o(\ln k)$.

- A polynomial time solution for the TP problem when there is only a single commodity by relating it to the cycle-transversal problem [97, 98].

- A $(\ln m + 1)(\ln k + 1)$-approximation algorithm for the TP problem, where m is the number of edges in the graph and k is the number of commodities.

- A spanning tree based algorithm for the general TP problem that leverages the relationship between the single commodity TP problem and the cycle-transversal problem.

- A method for translating multipath flow instances into single path instances that enable existing single path flow monitoring solutions to monitor multipath TP instances.

Organization

The rest of this proposal is organized as follows:

Chapter 2 introduces the *Peak Demand Minimization* problem for scheduling jobs so as to minimize the peak demand of the resulting schedule. It explores the complexity of this problem as well as details four different algorithms. An evaluation is presented that uses real-world, appliance specific, energy data to evaluate these algorithms against each other and against legacy solutions.

Chapter 3 introduces the *Peak Demand Minimization with Renewables* problem for scheduling jobs so as to minimize the peak non-renewable energy required to support the resulting schedule. Three different algorithms are detailed, including a linear programming relaxation scheme that provides the first approximation algorithm for the related *Peak Demand Minimization* problem.

Chapter 4 introduces the *Cooperative Group Provisioning* problem. The complexity of this problem is explored and a greedy algorithm that can handle common network conditions (group churn and multicast schemes) is detailed. An evaluation of the algorithm in a simulated multi-cloud environment using parameters gleamed for cloud service providers and ping measurements is also presented.

Chapter 5 explores the complexity and details various solutions for the *Turnstile Placement* problem. An optimal algorithm for a special case and approximation algorithm for the general case are introduced. For completeness, a simulation of these algorithms completed by a colleague is also included.

6

PEAK DEMAND MINIMIZATION

This chapter details the methods presented in [101, 102] of scheduling power jobs so as to minimize the peak cumulative power demand of a schedule in the smart grid. Smart grid technology has the opportunity to revolutionize our control over power consumption. Currently power-requesting jobs are scheduled in an on-demand fashion; power draw begins when the consumer requests power (turns on an appliance) and ends when the job is complete (appliance is turned off). Often such jobs have some flexibility in their starting times (e.g. a dishwasher or electric vehicle charger). This chapter aims to schedule power jobs so as to minimize peak demand of the resulting schedule. A general version of the problem is first introduced in which the job intervals can be staggered. While the problem is known to be NP-hard (shown in fact to be even NP-hard to approximate), this chapter introduces an optimal algorithm based on dynamic programming that is fixed-parameter tractable (FPT) and an effective greedy heuristic algorithm. For several important special cases, new constant-factor approximation algorithms that improve on previous results are also detailed. Simulation results using household power usage data show that peak power demand can be significantly reduced by intelligently leveraging the flexibility in job execution times.

Background

This chapter considers a variation on the classic job scheduling problem in which the goal is to minimize the peak demand of a schedule. An important scenario where this problem arises is in scheduling power jobs in the smart grid. It is advantageous for utility providers to keep aggregate power demand as flat as possible since this reduces the cost of generation and distribution. In addition, power generation may

be performed at a local level in which there is reduced benefit of averaging over a large set of users. Currently, power job requests in the home are not scheduled: power draw begins when the consumer turns on an appliance and ends when the appliance is turned off. This on-demand scheduling can lead to high peaks of power usage when consumers collectively issue many jobs, such as in the evening when numerous household appliances are running. Though many appliances need to be scheduled in an on-demand fashion to provide instant functionality (e.g. television, light bulbs), other appliances may have more flexibility and can be delayed and executed at a later time, when power draw is not so high (e.g. dishwasher, plug-in vehicle charging) [94]. This job flexibility provides the opportunity to flatten out the overall demand by intelligently delaying execution of some jobs.

There are several interesting variations on the peak demand scheduling problem: To begin with, some jobs may be known in advance while others arrive in a dynamic unpredictable fashion. Thus, the problem can be viewed in both offline and online settings. Another variation is whether jobs are preemptable, meaning they can be interrupted for some time, provided they are completed by the deadline. Some loads such as heating an electric water heater or charging a plug-in electric vehicle are preemptable but other loads such as running a dishwasher are not. Finally, some loads may be resizable in the sense that their power requirements may be adjustable between higher or lower power levels in conjunction with different execution times (e.g. battery charging). Each of these variations has an impact on the computational complexity of creating a schedule that minimizes the peak demand.

This chapter focuses on minimizing the peak demand of a schedule of non-preemptable, non-resizable jobs having flexible timelines. The contributions of this chapter are summarized below:

1. An optimal FPT algorithm based on dynamic programming.

2. An effective heuristic algorithm for the general case of non-preemptive peak demand minimization scheduling.

3. A 4-approximation algorithm for the case where jobs have the same arrival times and deadlines. Prior to this result, a 7.82-approximation algorithm was the best existing algorithm [91] for this case.

4. A $O(\log \Delta)$-approximation algorithm, where Δ is the ratio of the widest job execution window to the narrowest, for the case where job windows can be staggered, but are non-nesting.

5. An evaluation of the algorithms developed in this chapter using real world test scenarios that leverage current energy disaggregation research.

The rest of this chapter discusses related work, formalization of the PDM problem, algorithmic solutions, simulation results, and conclusions.

Related Work

Several variations on job scheduling for the smart grid have recently been explored. In [61] and [62], the authors define a job model, derive a lower bound on the optimal schedule cost for the offline, preemptive, non-resizable variation and also provide an optimal online algorithm for the preemptive, non-resizable case, assuming a statistical description of the input. In [70] and [38], the authors apply linear programming techniques to power job scheduling with an emphasis on pricing and price prediction models with the result of reducing peak loads. The main difference is that the work in this chapter considers specific jobs and aims to schedule them

within an acceptable window. These other works do not look at specific jobs, but instead aim to reduce overall usage with price incentives. In [91], the authors provide an approximation algorithm for peak minimization (MP) of non-preemptive jobs with the common execution window version of the problem addressed by the approximation algorithm on page 18. The MP algorithm has an approximation ratio of 7.82[1], while the one on page 18 is 4.

A problem closely related to peak demand scheduling is the much studied problem of job scheduling for a speed scalable CPU [99]. In this case, the tasks are compute jobs and the processing unit can vary the speed at which the current job executes; higher speed results in more energy use. While the preemptive version of the problem has been well-studied and good offline and online algorithms are known, only recently has the non-preemptive version been addressed [19], [21]. A version of the speed scaling problem that considers parallelizable jobs has also received recent attention [45]. Scheduling parallelizable compute jobs is similar for power jobs scheduling as both leverage concurrent execution.

Non-preemptive job scheduling is similar to rectangular strip packing [71], [20]. The main differences are that, in the general case, jobs are limited in where they can be placed in the strip and once jobs are scheduled, they do not need to remain as intact rectangles. Since job height represents the power required, each segment of a scheduled job will drop to lie on top of the job below it, instead of remaining as an intact rectangle in the strip.

[1]While not explicitly stated in [91], the best approximation ratio achieved for the MP algorithm results from minimizing $a + 2 + \frac{2a}{a-1}$, which occurs at $a = \sqrt{2} + 1$ and yields an approximation ratio of 7.82.

Problem Formulation

Job j is defined as a 4-tuple: (a_j, d_j, l_j, h_j), where a_j and d_j are the arrival time and deadline within the time interval $[0, T]$, l_j is the job length, and h_j is the job demand. Job j is scheduled by assigning it a start time, s_j, such that $[s_j, s_j + l_j] \subset [a_j, d_j]$. $[a_j, d_j]$ is called the *execution window* of j.

The *demand* at time t is,

$$H(t) = \sum_j h_j \chi_{[t \in [s_j, s_j + l_j]]}$$

where χ indicates if job j is being executed at time t:

$$\chi_{[t \in [s_j, s_j + l_j]]} = \begin{cases} 1, & \text{if } t \in [s_j, s_j + l_j] \\ 0, & \text{otherwise} \end{cases}$$

Then, the *peak demand*, H^{max}, of the schedule is the maximum demand of any of the timeslots in $[0, T]$:

$$H^{max} = \max_{t \in [0,T]} H(t)$$

Definition 2.1. *The **Peak Demand Minimization Problem (PDM)** is, given a set of n jobs, determine a set of start times, $\{s_j\}$, so as to minimize H^{max}.*

In general, each job can have any valid arrival time and deadline pair which leads to staggered execution windows as shown in Figure 2.1. This chapter considers both the general version of this problem as well as the case where jobs have common execution windows (same arrival times and deadlines).

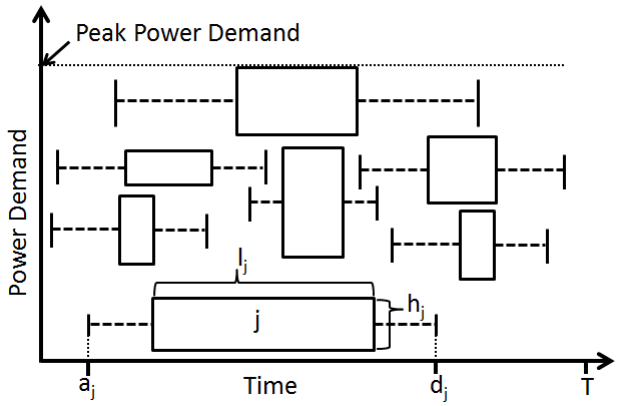


Figure 2.1: Non-preemptive power jobs to be scheduled

Computational Complexity

In [102], we showed PDM to be NP-hard to approximate within a factor of $\frac{3}{2} - \epsilon$ via a reduction from the *Bin-Packing* problem. It can be seen that PDM is even NP-hard to approximate within a ratio of 2 by reducing it from the *Scheduling with Release Times and Deadlines on a Minimum Number of Machines* (SRDM) problem.

Lemma 2.1. *For all $\epsilon > 0$, PDM is NP-hard to approximate within a factor of $2 - \epsilon$.*

Proof. The SRDM problem considers scheduling jobs that consist of release times, deadlines, and lengths in a way that minimizes the number of machines needed to construct a schedule. An instance of SRDM can be reduced to an instance of PDM by retaining each job's release time, deadline, length, and making each job's height to be 1.

Then, any solution to the PDM instance that results in a schedule of height p, will be a schedule for the SRDM instance requiring p machines due to the unit height of all the jobs. Likewise, any solution to the SRDM instance requiring p machines will be a schedule for the PDM instance of height p because that is the minimum number of concurrently executing jobs. It follows that any approximation algorithm

for PDM will provide the same approximation ratio for SRDM. Since SRDM cannot be approximated within a factor of $2 - \epsilon$, so the same hardness result applies to PDM [35]. □

Algorithms

This section introduces a simple heuristic algorithm for the general case of the PDM problem and then presents approximation algorithms for several important special cases.

An Optimal FPT Algorithm

The PDM-Exact algorithm detailed here employs dynamic programming techniques to minimize schedule height for the general, staggered execution window version of the problem. The algorithm searches for the minimum feasible peak power demand by testing whether or not the jobs can be scheduled under a predetermined power threshold τ. The threshold τ is then varied until the minimum value can be determined. Feasibility testing is performed by considering, for each job, every possible configuration of start times of that job and each previous job it overlaps (employing a non-trivial definition of overlap), and then seeing if a set of configurations for each job can be concurrently scheduled. Pseudocode is presented in Algorithm 2.1 and further explained below.

Job Overlap Lists Order the jobs by increasing deadline first and then by increasing arrival time. Then determine, for each job j, the set of jobs that j overlaps. Overlap is defined in a non-trivial manner: Job i is said to *overlap* job j if $i \leq j$ and $\min_{l \geq j}(a_l) < d_i$. I define *job overlap lists* $\mathcal{L}_j$, as follows:

On a set of jobs:

Sort jobs first by deadline and then by arrival time.

Compute $\mathcal{L}_j = \{i : \text{job } i \text{ overlaps job } j\}$.

Let $\tau_{\max} = 1$

while (**not** Is-Feasible($\tau_{\max}$))

$\tau_{\max} = 2 \cdot \tau_{\max}$

endwhile

Binary search for $\tau_{\min} = \arg\min_{\tau \in (\tau_{\max}/2, \tau_{\max}]}$ Is-Feasible(τ)

Select $c \in \mathcal{C}_{n-1}$

Set s_{n-1} according to c

for each job $j = n - 2$ to 0:

Let $c' \in h(c)$ (note: $c' \in \mathcal{C}_j$)

Set s_j according to c'

Let $c = c'$

endfor

Algorithm 2.1: PDM-Exact

$\mathcal{L}_j = \{i : \text{job } i \text{ overlaps job } j\}$. Figure 2.2 illustrates a non-trivial example of job overlaps. Note that $\mathcal{L}_0 = \{0\}$, $\mathcal{L}_1 = \{0, 1\}$ and $\mathcal{L}_2 = \{0, 1, 2\}$ in Figure 2.2.

Lemma 2.2. *For ordered jobs: $i, ..., k, ..., j$, if job i is in $\mathcal{L}_j$, then job i is also in $\mathcal{L}_k$.*

Proof. This is easy to confirm as $i \in \mathcal{L}_j$ implies that $\min_{l \geq j}(a_l) < d_i$. Since the jobs are ordered by deadline, $\min_{l \geq j}(a_l) < d_i \leq d_k$. □

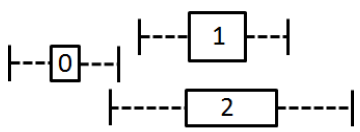


Figure 2.2: Job 0 overlaps job 1 due to job 2's early arrival time

On a potential power threshold τ:
  update $\mathcal{C}_j = \{\langle \ldots, s_k, \ldots \rangle : k \in \mathcal{L}_j, a_k \leq s_k \leq d_k - l_k,$
    $\text{maxHeight}(\langle \ldots, s_k, \ldots \rangle) \leq \tau\}$.
  for each job $j = 1$ to $n - 1$:
    for each $c' \in \mathcal{C}_j$:
      Let $h(c') = \{c \in \mathcal{C}_{j-1} : c \sim c'\}$
      if $h(c') = \emptyset$
        $\mathcal{C}_j$.remove(c)
      endif
    endfor
    if $\mathcal{C}_j = \emptyset$
      return false
    endif
  endfor
  return true

Algorithm 2.2: Is-Feasible(τ)

Configuration Generation A value for the power threshold τ is given. For each job j, generate every possible configuration $\langle \ldots, s_k, \ldots \rangle$ of start times of jobs k in its overlap list $\mathcal{L}_j$. Define $\text{maxHeight}(\langle \ldots, s_k, \ldots \rangle)$ as the peak demand of the jobs in $\mathcal{L}_j$ given these start times. Each configuration is a tuple consisting of a valid start time for each job k in $\mathcal{L}_j$, such that the peak demand of the configuration is at most τ. The configuration lists are formally defined as:

$$\mathcal{C}_j = \{\langle \ldots, s_k, \ldots \rangle : k \in \mathcal{L}_j, a_k \leq s_k \leq d_k - l_k, \text{maxHeight}(\langle \ldots, s_k, \ldots \rangle) \leq \tau\}$$

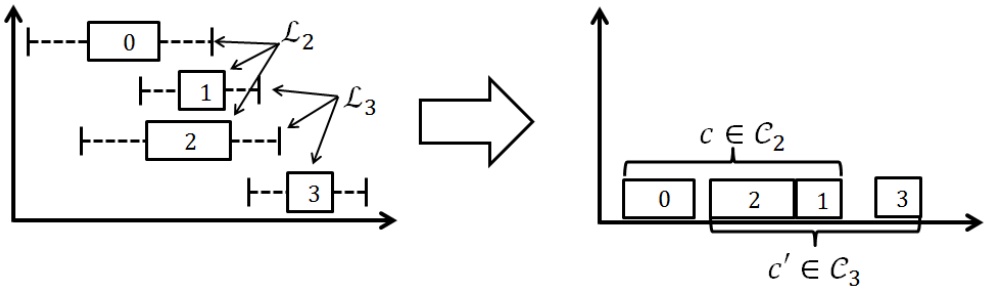


Figure 2.3: Compatible configurations

Definition 2.2. *Configuration c in $\mathcal{C}_i$ and c' in $\mathcal{C}_j$ are* ***compatible*** *(written $c \sim c'$) if they agree on the starting times of all jointly shared jobs. A configuration, c' in $\mathcal{C}_j$ has a* ***valid history*** *if $j = 0$, or there is a c in $\mathcal{C}_{j-1}$ such that $c \sim c'$ and c has a valid history.*

Figure 2.3 shows an example of compatible configurations. Note that since job 0 in Figure 2.3 has multiple valid start times that do not interfere with the other jobs, there are multiple configurations in $\mathcal{C}_2$ that are compatible with $c' \in \mathcal{C}_3$.

Feasibility Testing Lemma 2.2 implies that each overlap list $\mathcal{L}_j$ is consecutive run of jobs ending at job j. It follows that a configuration history $c_0 \sim c_1 \sim \cdots \sim c_j$ defines a specific start time for each job 0 to j. For every configuration $c \in \mathcal{C}_j$, ordered by j, $\mathcal{C}_{j-1}$ is searched for valid histories of c. As they are found, pointers are assigned to point from c to its valid history configurations $c' \in \mathcal{C}_{j-1}$. If a valid history for $c \in \mathcal{C}_j$ cannot be found then it is removed from $\mathcal{C}_j$ and cannot serve as part of a valid history for configurations in $\mathcal{C}_{j+1}$. Furthermore, if maxHeight$(c) > \tau$ then c is also removed from $\mathcal{C}_j$. If $\mathcal{C}_j$ is empty at the end, then the peak demand threshold τ is not feasible. If a valid history can be found for at least one configuration of the final job n, then a schedule can be generated with a peak demand at most τ.

Schedule Building A schedule can be built once the minimum τ is known by beginning with the last job and constructing the valid history in reverse. That configuration includes a pointer to a valid history configuration in the prior configuration list and so on. Proceed through the jobs in reverse, selecting a configuration for job $j-1$ that belongs to a valid history of the configuration that was chosen for job j.

Theorem 2.1. *The schedule found by PDM-Exact achieves the optimal peak power demand.*

Proof. Suppose $\mathcal{S}^*$ is an optimal PDM schedule for the given input jobs with peak demand τ^*. Let c_j^* be the configuration for job j that agrees with $\mathcal{S}^*$. Clearly, c_0^* will be an available configuration in $\mathcal{C}_0$ if the current test threshold $\tau \geq \tau^*$. By induction, c_j^* will also be an available configuration in $\mathcal{C}_j$ Thus, τ will reach τ^* and the schedule with the optimal peak demand will be found. □

Fixed Parameter Tractability

Definition 2.3. *An algorithm is said to be fixed parameter tractable (FPT) if there exists some parameters, $p_1, p_2, ...$, of the input or output such that the running time of the algorithm is a polynomial function of the input size times some function of the parameters: $f(p_1, p_2, ...) \cdot poly(n)$.*

The complexity of PDM-Exact is driven by the size of the configuration lists $\mathcal{C}_j$, which is dependent on the size of $\mathcal{L}_j$ and the number of possible start times for each job. Thus, a large number of jobs having limited overlap list sizes can still be scheduled efficiently. On the other hand, the worst case is if there is a job that spans the entire timescale, then every configuration of all jobs needs to be considered. If n represents the number of jobs, $m = \max_j(|\mathcal{L}_j|)$, and q is the maximum number of job

start times, i.e. $q = \max_j(d_j - a_j - l_j + 1) \leq T$, then the running time of PDM-Exact is $O(\lg \tau^* \cdot q^m \cdot n)$, where τ^* is the optimal peak demand value.

A Greedy Heuristic Algorithm

As noted above, the general PDM problem is NP-hard to approximate so we consider a heuristic approach that is shown to be effective in practice. The idea is to schedule jobs that have tight execution windows first and save jobs with more space in their execution windows to be scheduled later. To schedule a job, simply find the starting time that minimizes the peak demand of the schedule created so far. The complete algorithm is given below as Algorithm 2.3 (note that $w_j \in [0, 1]$ measures the tightness of job j; values closer to 1 indicate tighter jobs).

```
On a set of jobs:
    forall jobs j
        Set w_j = l_j / (d_j - a_j + 1)
    endforall
    Sort jobs by decreasing w_j values
    forall jobs j (in sorted order)
        s_j = a_j
        for Each start time t, in j's execution window
            if maxHeight(S ∪ t) < maxHeight(S ∪ s_j)
                s_j = t
            endif
        endfor
    endforall
```

Algorithm 2.3: PDM-Heuristic

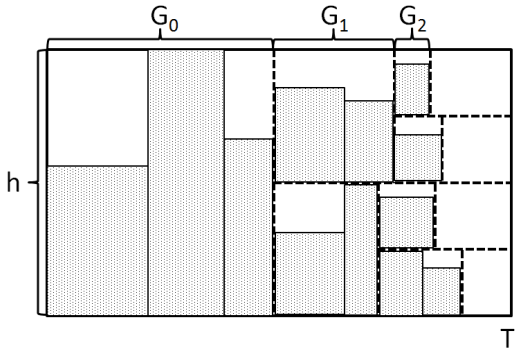


Figure 2.4: Example job schedule

A Common Execution Window Algorithm

This section details a 4-approximation algorithm that aims to minimize schedule height for the special case version where jobs have common execution windows. This algorithm compares favorably to the MP algorithm presented in [91], which achieves a 7.82-approximation ratio for the same problem.

The basic idea of this algorithm is to group jobs based on their height and then schedule each group in horizontal strips. Group i contains all jobs that have height between $\frac{h}{2^{i+1}}$ and $\frac{h}{2^i}$, where h is the largest job height in the input. Jobs within each group are ordered by non-increasing length and scheduling is done one group at a time beginning with G_0. At the beginning of each group being scheduled, the strips that already exist from the prior groups are each split in half horizontally. This results in two times the number of strips, with each being half of the previous strips' height. Jobs from the group being scheduled have access to the remaining space in the current strips, or a new strip is created if needed. Figure 2.4 illustrates an example of jobs being scheduled that do not require generation of any new strips. Details are given in the subsequent proofs.

Definition 2.4. *The* ***span*** *of a job is the number of strips that a job occupies. Initially, span(j) = 1 when the job is first scheduled and is doubled when each*

On a set of jobs:
    Sort jobs by non-increasing height
    Let h be the maximum job height
    Partition jobs into groups, G_i, based on job height such that:
        $G_i = \{jobs, j | \frac{h}{2^i} \geq h_j > \frac{h}{2^{i+1}}\}$
    Sort each group by non-increasing length
    Let $\mathcal{R}$ be the set of largest scheduled timeslots in each strip
    forall groups G_i, starting at $i = 0$
        forall $r \in \mathcal{R}$
            $\mathcal{R}$.add(r)
        endforall
        forall $j \in G_i$
            Find largest $r \in \mathcal{R}$ s.t. $r + l_j - 1 \leq T$
            if such an r exists
                Set $s_j = r$ and $\mathcal{R}$.remove(r)
            else
                Set $s_j = 0$ to begin a new strip
            endif
            $\mathcal{R}$.add($s_j + l_j$)
        endforall
    endforall

Algorithm 2.4: PDM-CW (Common Execution Windows)

subsequent group splits all strips in half.

Define the ***area*** *of a job, strip, schedule, and jobs in a schedule as follows:* $A_j = h_j \cdot l_j$ *(area of a specific job)*

$A_{\mathcal{S}} = \sum_{j \in \mathcal{S}} \frac{A_j}{span(j)}$ *(area of jobs in strip* $\mathcal{S}$*)*

$A_{alg} = H^{alg} \cdot T$ *(area of the algorithm's schedule)*

$A_{jobs} = \sum_{\mathcal{S} \in schedule} A_{\mathcal{S}}$ *(area of all jobs in a schedule)*

*Strip $\mathcal{S}$ is **filled** if $A_{\mathcal{S}} \geq \frac{1}{4} h_{\mathcal{S}} T$, where $h_{\mathcal{S}}$ is the height of $\mathcal{S}$.*

The proof that PDM-CW is a 4-approximation algorithm leverages the fact that a schedule's optimal height (H^{opt}) and actual height (H^{alg}) can be related through A_{alg} and A_{jobs} since,

$$A_{jobs} \leq A_{opt} \leq A_{alg}$$

then,

$$\begin{aligned} A_{alg} \leq \alpha A_{jobs} &\Rightarrow A_{alg} \leq \alpha A_{opt} \\ &\Rightarrow H^{alg} \cdot T \leq \alpha H^{opt} \cdot T \\ &\Rightarrow \frac{H^{alg}}{H^{opt}} \leq \alpha \end{aligned}$$

It is also important to identify the three ways that a strip can be created in the PDM-CW algorithm:

1. The *initial strip* of height h is created at the beginning of G_0's scheduling.

2. All existing strips are split in half at the beginning of each group's scheduling. These strips are called *children*.

3. If a job is too wide to fit into any existing strips, a *new strip* of that group's height is created and the job is scheduled to start at time 0. This strip is the *parent* of all its children created for subsequent groups.

Lemma 2.3. *A new, not filled, strip is created for G_i only if all preceding strips are filled. Furthermore, at most one new, not filled, strip can exist at the end of G_i's scheduling.*

Proof. First see that every unfilled strip has a length of less than $\frac{T}{2}$. By Definition 2.4, if strip $\mathcal{S}$ is unfilled,

$$\sum_{j \in \mathcal{S}} \frac{A_j}{span(j)} = A_{\mathcal{S}} < \frac{1}{4} h_{\mathcal{S}} T$$

Then, since each job's area is evenly spread amongst all strips it lies in, the area of job j allotted to each strip is greater equal $\frac{h_{\mathcal{S}}}{2} l_j$, where $h_{\mathcal{S}}$ is the height of the strip. This is because every group contains jobs with a height minimum of half the minimum height of the previous group. Thus,

$$\Rightarrow \sum_{j \in \mathcal{S}} \frac{h_{\mathcal{S}}}{2} l_j \leq \sum_{j \in \mathcal{S}} \frac{A_j}{span(j)} < \frac{1}{4} h_{\mathcal{S}} T$$
$$\Rightarrow \sum_{j \in \mathcal{S}} l_j < \frac{T}{2}$$

It follows easily that a new unfilled strip will not be created when there are existing unfilled strips, as the new strip would merely be scheduled in an existing unfilled strip. A similar argument shows that at most one new unfilled strip exists at the end of G_i's scheduling: If two strips existed, then their lengths are both less than $\frac{T}{2}$ and could be scheduled in the same strip. □

Lemma 2.4. *If all strips in a schedule are filled, then the maximum height of the entire schedule is at most four times the optimal value.*

Proof. Suppose that G_i was the last group to be scheduled and that there are n strips in the final schedule. Since G_i is the final group, each strip is of height $\frac{h}{2^i}$ and $H^{alg} \leq n \frac{h}{2^i}$. Further, since each strip in the schedule is filled, the area of jobs within

each strip is at least $\frac{1}{4}\frac{h}{2^i}T$. Then,

$$A_{alg} \leq n\frac{h}{2^i}T \leq 4n\frac{1}{4}\frac{h}{2^i}T = 4A_{jobs}$$

□

Lemma 2.5. *If there is at least one strip in the schedule that is not filled, then the maximum height of the entire schedule is at most four times the optimal value.*

Proof. Suppose the schedule contains a not filled strip and let q be its parent, created during G_i's scheduling. By Lemma 2.3, q is the parent of all not filled strips in the final schedule, since new strips created before and after q must be filled. This leads to two cases:

Case 1. q is not the initial strip. Let w_1 be the length of q and w_2 be the length of any other strip, at the end of G_i's scheduling. q was created because some job was too wide to fit in any existing strips, which means that $T \leq w_1 + w_2$. Since $\frac{h}{2^{i+1}}$ is the minimum height of jobs in each strip at this point,

$$\begin{aligned} T \leq w_1 + w_2 \Rightarrow \frac{h}{2^{i+1}}T &\leq \frac{h}{2^{i+1}}(w_1 + w_2) \\ \Rightarrow \frac{h}{2^{i+1}}T &\leq A_{jobs} \end{aligned}$$

Thus, the area of these two strips is:

$$A_{alg} \leq 2\frac{h}{2^i}T \leq 4\frac{h}{2^{i+1}}T = 4A_{jobs}$$

Since the height of q and any other strip at the end of G_i's scheduling is at most four times optimal, the schedule is too since all remaining strips are filled. The final

schedule is then at most four times optimal since all new rows on top of q must be filled, and thus accounted for through Lemma 2.4.

Case 2. q is the initial strip. By Lemma 2.3, all strips above q are filled and too wide to fit in q. We know that q has height h, since a job of height h must exist and be in G_0. Let $H = H^{alg} - h$, and let w_1 be the minimum length of children of q and w_2 be the minimum length of strips that are not children of q. Then, $T \leq w_1 + w_2$ and $w_2 > w_1$, or else q would be filled. Then,

$$\begin{aligned}
A_{alg} &\leq hT + HT = hT + \alpha hT, \text{ for some } \alpha > 0 \\
&\leq h(w_1 + w_2)(\alpha + 1), \text{ since } T \leq w_1 + w_2 \\
&= 2hw_1 + hw_1(\alpha - 1) + hw_2\alpha + hw_2 \\
&\leq 2hw_1 + hw_2(\alpha - 1) + hw_2\alpha + hw_2, \text{ since } w_2 > w_1 \\
&= 2hw_1 + 2hw_2\alpha \leq 4(\frac{h}{2}w_1 + \frac{\alpha h}{2}w_2) \leq 4A_{job}
\end{aligned}$$

□

Theorem 2.2. *PDM-CW is a 4-approximation algorithm and runs in $O(n^2)$ time.*

Proof. The fact that PDM-CW is a 4-approximation algorithm follows directly from Lemmas 2.4 and 2.5. Step 1 of PDM-CW is sorting and partitioning, which can be done in $O(n^2)$ time, and Step 2 can be completed in $O(n^2)$ because for every job, we need to search through the list of rightmost boundary points, and we cannot have more rightmost boundary points than we have jobs. □

PDM-CW for Online Scheduling To modify PDM-CW for online job scheduling, let group $i \in (-\infty, +\infty)$ contain all jobs that have a height between 2^{i+1} and 2^i. Unlike for PDM-CW, strips are only used by the group that created them. When a

new jobs arrives, it is scheduled in the strip for its group that provides the tightest fit, or a new strip is created if none are available.

Theorem 2.3. *The algorithm described above achieves a competitive ratio of* 8 *against an optimal offline algorithm.*

Proof. If all strips become filled, the height of the schedule is at most four times optimal (similar to Lemma 2.4). If no strips are filled, the height of the schedule is at most four times optimal since the group heights define a geometric series and the sum of which must be at most four times the height of the largest job. If there is a mixture of filled and unfilled strips, they can be grouped and sum to at most eight times the optimal height. □

A Non-nesting Execution Window Algorithm

Finally, consider the case in which the job execution windows can be staggered, but are *non-nesting*, meaning that if $a_i \leq a_j$ then $d_i \leq d_j$ for all jobs i, j. Since the jobs are non-nesting, they can be ordered first by increasing arrival times then by increasing deadlines, in case of ties. Partition the ordered job list into consecutive runs called *regions* such that if s is the first job in the region and t is the last job then for all $s \leq i \leq t$, we have $l_i \geq \frac{1}{2}L$, where $L = (d_t - a_s)$, the total region length.

I first describe a modification to the common execution window algorithm from page 18 that will be useful: The main difference is that we will reserve additional space for "remainder" jobs in each height category (these are the jobs that end up in unfilled strips). As noted, the maximum length of the remainder jobs for each height category is $T/2$. Let the maximum height of any job be h, as before. Reserve a $[0, T]$ strip of height h the remainder jobs with height category $(h/2, h]$; these jobs will be scheduled in $[0, T/2]$. Remainder jobs from height category $(h/4, h/2]$ will then be

scheduled in the $[T/2, T]$ portion of the strip. Next, a strip of height $h/4$ will be allocated for remainder jobs from the next two height categories and so on. The total height of the remainder strips is then $h + \frac{h}{4} + \frac{h}{16} + \ldots \leq \frac{4}{3}h$. By arguments analogous to those in Lemma 2.4, regular strips (as opposed to remainder strips) are at least $\frac{1}{4}$ full so the above construction leads to a $4 + \frac{4}{3} = 5\frac{1}{3}$-approximation algorithm for the common execution window case. A useful observation about this modified algorithm is that in each strip, no jobs overlap, i.e. no jobs are simultaneously scheduled within a strip.

The strategy we employ for the non-nesting execution window case is to use a further modification of the above algorithm to schedule jobs within each *region* from the job list: The modified algorithm proceeds by first determining the height category of each job in the region and then processing each height category separately from largest to smallest. Within each category, the jobs are ordered by decreasing length into a list J. Next, the algorithm creates a *schedule strip* by finding the maximum number of jobs from the beginning of J that can placed in a non-overlapping fashion in the schedule; this is done greedily: the jobs selected are ordered by their arrival times and the job with the earliest arrival time is scheduled at this time, then the job with the next earliest arrival time is scheduled as early as possible in such a way to not overlap the first job and so on. Once the maximum number of jobs from the head of J that can be scheduled in this way is found, these jobs are scheduled in a strip and removed from J. This process continues until all jobs are scheduled. If the total length of the jobs in the final strip is less than $L/4$, then these jobs are considered remainder jobs and they are scheduled in the reserve strips as above (each job execution window is is at least $L/2$ in width; it follows that the width of the scheduled jobs in each strip, except possibly the final strip from the height category will have total width at least $L/4$).

Lemma 2.6. *The above algorithm provides a $9\frac{1}{3}$-approximation for the region scheduling problem.*

Proof. Let H^{opt} be the peak demand of an optimal schedule for the region in question. Regular strips are at least $L/4$ full with non-overlapping jobs that differ in height by at most a factor of 2. This means that they occupy at least $\frac{1}{8}$ of the total area available within the strip. Again, by a similar area argument to the one used in the proof of Lemma 2.4, it follows that the optimal schedule for such jobs must have a peak demand at least $\frac{1}{8}$ the peak demand found by the algorithm; i.e. $H^{max} \leq 8H^{opt}$. As discussed, the remainder jobs can be slotted into strips whose total height is at most $\frac{4}{3}h \leq \frac{4}{3}H^{opt}$, where h is the largest job height present in the input. It follows that $H^{max} \leq 9\frac{1}{3}H^{opt}$ as claimed. □

Next, observe that the maximum number of regions that can overlap each other is $O(\log \Delta)$, where Δ is the ratio of the longest job interval to the shortest, i.e. $\Delta = \frac{\max_i d_i - a_i}{\min_i d_i - a_i}$.

Begin with a lemma that shows that only a constant number of regions of similar size can overlap. Define the *length class* of a region of length L to be $\lceil \log_2 L \rceil$, e.g. regions whose length falls in $(1, 2]$ have length class 1. Observe that the number of distinct region length classes in the input is at most $\lceil \log_2 \Delta \rceil$.

Lemma 2.7. *At most 24 regions of the same length class can overlap.*

Proof. Consider the case of length class 1; the proof is analogous for other values. Suppose that the arrival time of the first job in the first region is 0. The duration of any job from a class 1 region is at least 1/2 (since the total region length is is at least 1). Suppose that the left endpoint of subsequent regions stays in the interval $[0, 1/4]$. This can happen for at most 3 subsequent regions. To see this, note that the

right endpoints of all jobs in this new region must be greater than 1 (otherwise they would belong to the first region). Thus, these jobs must all be at least 3/4 in length. A similar argument shows that jobs in the next region must be at least 3/2 in length (the right endpoints must be greater than $1\frac{3}{4}$). At this point another new region of class 1 cannot be created if the left endpoint is to stay in the $[0, 1/4]$; its jobs would be of length at least 3 (doubling as before) which is too long for a class 1 region. This means that the next region's left endpoint must be located outside of $[0, 1/4]$. Thus, at most three different class 1 regions can have a left endpoint in $[0, 1/4]$. Since the maximum length of a class 1 region is 2, it follows that at most $3 \cdot 8 = 24$ class 1 regions can overlap. □

Theorem 2.4. *Applying the region-scheduling algorithm to each region in the input independently yields a $O(\log \Delta)$-approximation algorithm to the overall problem of power scheduling with staggered, non-nesting job intervals. The running time of the algorithm is $O(n^2)$.*

Proof. By the above observation, there are at most $\lceil \log_2 \Delta \rceil$ length classes. Together with Lemma 2.7, this implies that the maximum number of overlapping regions at any time point is $24\lceil \log_2 \Delta \rceil$. Within each of the each regions the schedule created is at most $9\frac{1}{3}$ times optimal. Since the schedules created for overlapping regions stack up, the maximum peak heights for each region may add up in the final schedule created by the algorithm (whereas the optimal schedule may find a way to avoid this). Thus, the overall solution found by the algorithm is at most a factor of $9\frac{1}{3} \cdot 24\lceil \log_2 \Delta \rceil = O(\log \Delta)$ from optimal. Suppose region i contains n_i jobs; the running time of the algorithm is dominated by the time to apply the modified version of Algorithm 2.4 to each region. The time required is $\sum_i O(n_i^2) = O(n^2)$, since $\sum_i n_i = n$. □

Since the approximation constant in the above algorithm is somewhat large, it was not implemented for testing.

Experimental Results

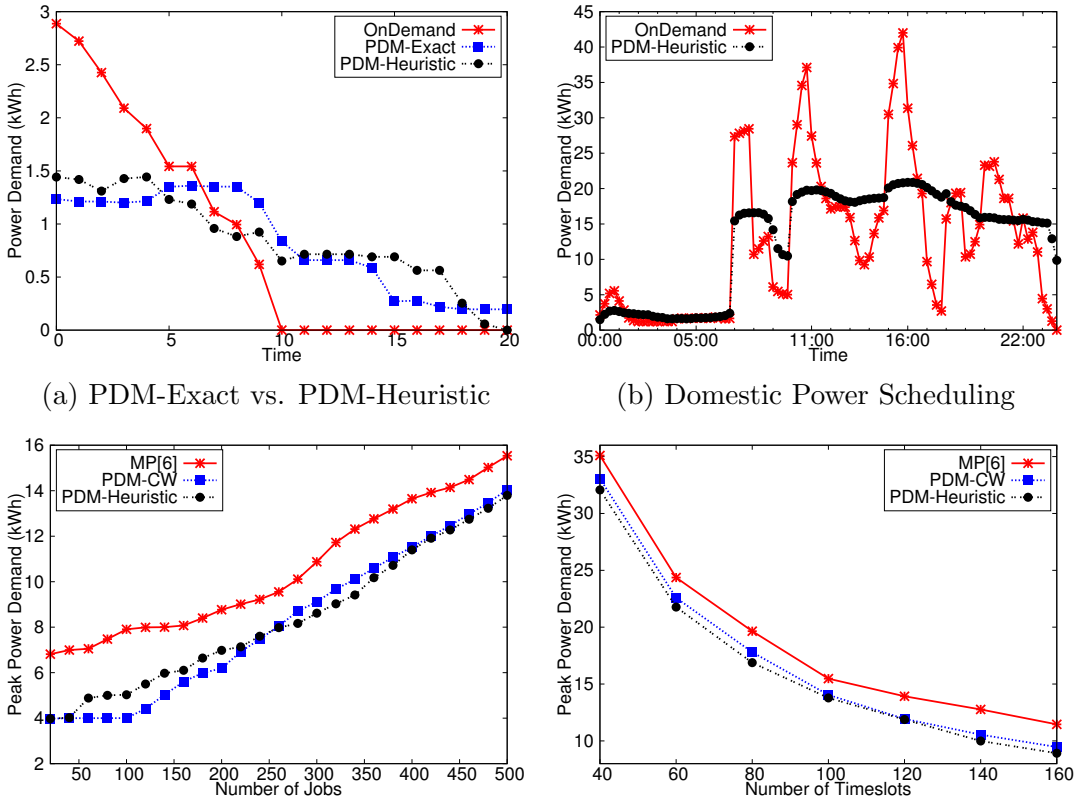
Simulations were conducted using *PDM-Exact*, *PDM-Heuristic*, *PDM-CW*, and two other algorithms: *MP [91]* is the 7.82-approximation algorithm presented in [91], *OnDemand* schedules jobs to start at their arrival time.

Jobs were created using appliance specific data from six residences [59]. Appliances likely to have flexible timelines (e.g. washing machine) were identified and their height, length, and arrival time distributions within a 24 hour period were determined. Deadlines were set to be uniformly distributed between the minimum deadline and four times the average length.

Staggered Execution Window Evaluation

Two different scenarios were considered to test the effectiveness of the algorithms. First, a simpler scenario was created in order to test the performance of the PDM-Exact algorithm. Instead of generating jobs as described above, jobs were randomly generated with arrival time 0 to simulate a single peak. Figure 2.5a shows the average results of running PDM-Exact, PDM-Heuristic and OnDemand on five iterations of five jobs each from this simplified data generation. The average peak power demand was 1.36 for PDM-Exact, 1.44 for PDM-Heuristic, and 2.89 for OnDemand scheduling. As can be seen in the figure, the PDM-Heuristic algorithm achieves a near-optimal schedule.

Next, Figure 2.5b shows the average of 1000 iterations of peak power demand versus time of day for both PDM-Heuristic and the OnDemand algorithm (PDM-Exact was unable to schedule a realistic number of these jobs) over a 24 hour period.



(a) PDM-Exact vs. PDM-Heuristic

(b) Domestic Power Scheduling

(c) Peak demand VS. jobs

(d) Peak demand VS. timeslots

Figure 2.5: Performance of PDM scheduling algorithms

The average peak power demand from OnDemand scheduling is 42.0 kWh while the average peak for the PDM-Heuristic is 20.8 kWh.

Common Execution Window Evaluation

The first test scenario for common window algorithms varied the number of jobs from 20 to 500 for 100 timeslots. The average of 1000 iterations of peak demand versus number of jobs is shown in Figure 2.5c. The OnDemand algorithm is not plotted, as all solutions present a very large improvement over every job being started at timeslot 0. PDM-CW reduces the peak by 50% compared to MP [91] for 100 jobs, but this

reduction decreases to 10% by 500 jobs. It is also apparent that PDM-CW performs similarly to PDM-Heuristic in the case of common windows.

The second scenario varies the number of timeslots from 40 to 160 for 500 jobs. The average of 1000 iterations of peak demand versus number of timeslots is shown in Figure 2.5d. The peak demand for all algorithms decreases as the number of timeslots increases, since there is more space for job packing. PDM-CW reduces the peak by 6% at 40 timeslots compared to MP [91], and 17% at 160 timeslots.

Conclusions

Emerging smart grid systems will allow for more control over the scheduling of power jobs which will enable cost savings. This chapter presented several new algorithms for the PDM problem: an FPT algorithm and a heuristic algorithm for the general (staggered execution window) problem, a 4-approximation algorithm for the common execution window special case, and another approximation algorithm for non-nesting jobs. Together, these algorithms advance the state-of-the-art for the PDM problem and offer the ability to effectively reduce peak loads in future smart grid deployments.

PEAK DEMAND MINIMIZATION WITH RENEWABLES

This chapter explores methods of scheduling power jobs under a renewable energy curve (paper containing these results in review). Power-using jobs are traditionally scheduled in an on-demand fashion leading to large peaks in energy usage that are proportionally more expensive than constant demand to generate and provision for. These peaks are exacerbated in local microgrids that do not benefit from the aggregate smoothing experienced by large grids. Demand-side scheduling can reduce these peaks by taking advantage of the fact that there is often flexibility in job start times. The availability of renewable energy provides an additional mechanism for controlling the non-renewable consumption. This chapter explores shifting the execution of some jobs to times of high renewable energy generation, so as to minimize the peak non-renewable energy required. The *Peak Demand Minimization with Renewables* (PDM-R) optimization problem is formalized and four algorithms provided: An optimal, fixed-parameter tractable algorithm, a linear programming relaxation scheme, and an offline and online heuristic (where jobs are unknown in advance). This linear programming relaxation also provides the first approximation algorithm for the well-studied *Peak Demand Minimization* (PDM) problem's general case, where jobs can have any release time or deadline. Simulation results using household consumption data and local wind power availability show that these algorithms are effective in reducing peak non-renewable power consumption.

Background

Power grids are forced to tailor the generation of energy to fulfill instantaneous consumer demand. The lack of coordination between generation and consumption leads to peaks in demand, which are proportionally more expensive to generate

and provision for. Large grids are able to realize the benefit of smoothing by the aggregation of many jobs, which naturally flatten peaks. Local microgrids, and even larger regional grids, do not see such a high degree of smoothing and are particularly sensitive to variability in power demand.

Direct consumer appliance to energy provider communication is a foundation of enhanced smart grid capabilities. Intra-grid communication enables coordination between consumers and suppliers, and provides unprecedented control over energy consumption.

Smoothing out demand can be accomplished in several ways, one of which is to schedule jobs such that peaks are minimized. Scheduling to reduce peaks is possible because many jobs have flexibility in when they can be executed. Some appliances require instant functionality (e.g. light bulbs) but many others have more flexibility in exactly when they operate (e.g. dishwasher). This flexibility can be leveraged to shift demand to off peak times. This problem is called the *Peak Demand Minimization* (PDM) problem and has been studied extensively [43, 54, 62, 65, 82, 91, 101, 102].

The challenges associated with smoothing out demand are complicated when renewable generation sources are integrated into the grid. Many renewable generation sources do not have the capability to dynamically scale output to the degree that traditional generation sources are capable of. Additionally, without more sophisticated storage mechanisms, failure to utilize renewable energy does not save it for later. This leads to the compounded issue of using volatile sources to generate energy for volatile demand.

This chapter proposes methods for scheduling non-preemptable jobs, which are jobs that must be executed without interruption (i.e. they cannot be paused during execution). These methods aim to reduce the peak load required of non-renewable generation sources by leveraging available renewable energy. This chapter formally

introduce the *Peak Demand Minimization with Renewables* (PDM-R) problem, an extension of the PDM problem, and describes three algorithms:

1. An optimal algorithm based on dynamic programming that is *fixed parameter tractable*.

2. An approximation algorithm, based on a linear programming rounding technique. For the PDM version of the problem, this provides a $O\left(\frac{\log n}{\log \log n}\right)$-approximation and is the first known approximation.

3. An effective greedy heuristic algorithm that can be used in either an online or offline fashion.

Each of these algorithms is tested using real power job data from energy disaggregation research and renewable energy generation data from the National Renewable Energy Laboratory [41].

A key assumption of the methods in this chapter is that the amount of renewable energy available is known in advance for the entire time period being scheduled. This is a large assumption for the general use of renewable sources. Forecasting techniques are detailed on page 35 that allow tuning the forecasted renewable energy availability to the risk one is willing to accept of the scheduled peak exceeding the actual peak. Future efforts to incorporate uncertainty directly into these algorithms is discussed on page 55.

The rest of this chapter is organized as follows. Related work is discussed on page 34. The problem is formulated on page 35 and examine its computational complexity. Algorithms are presented on page 38 and simulation results are presented on page 52. The chapter is concluded and future work discussed on page 55.

Related Work

In this section, this work is placed in the context current demand response research, and background on forecasting renewable power availability is provided.

Power Job Scheduling

Many variations on modifying demand to improve the efficiency of power grids (generically called *demand response*) have been explored. These approaches can generically be categorized as utility function optimization (commonly cost function minimization) and peak demand minimization.

Utility function optimization techniques use price incentives to encourage consumers to modify their behavior to achieve some aggregate objective (e.g. generation cost minimization) [28, 33, 66, 84, 90, 107]. Some of these efforts consider renewable energy when influencing load shape, which is the goal [33, 66, 84, 90, 107] of this chapter. However, generic cost function optimization is fundamentally different from peak demand minimization, since it does not provide explicit control over load shapes and does not schedule individual jobs.

PDM aims to directly schedule jobs to reduce peaks in power schedules instead of passively influencing changes with price incentives. PDM has been studied in the context of jobs that can be interrupted partway through execution [43, 62]. This chapter considers the scenario where jobs are unable to be interrupted. The PDM problem has been found to be NP-hard to approximate within a ratio of 2 in this scenario [101]. Three approaches have been taken to get around this complexity. Heuristics have been developed that provide no theoretical guarantee [54, 65], approximation algorithms have been developed for special cases [91, 102], and optimal algorithms have been introduced that work on a limited number of jobs [82, 101].

Additionally, none of these approaches address the integration of renewable resources as part of peak demand minimization. This work differs in that it is the first to provide a scalable algorithm with a theoretical guarantee for the general PDM problem for uninterruptable jobs, it provides an optimal algorithm achieving a 15x increase in the number of schedulable jobs over previous approaches, and it considers renewable energy availability when scheduling.

Renewable Energy Forecasting

The algorithms presented in this chapter require knowledge of the renewable energy available throughout the scheduling horizon (evaluated at 24 hours). Extensive research has been done with wind power forecasting (WPF) and modeling those forecast errors [75]. The research most relevant to this chapter is the research that aims to quantify the uncertainty of forecasts with a horizon of at least 24 hours [24, 25, 87, 108]. All of these efforts employ probabilistic forecasting to bound the prediction error and enable system operators to select a forecast based on a confidence level. A sample of this type of probabilistic forecast is taken from [25] and presented in Figure 3.1. This enables users to select the lower curve of a forecasted confidence interval based on the probability that they are willing to accept failing to meet (exceeding) the peak demand predicted by these algorithms.

Problem Formulation

The system model consists of a renewable generation source curve and jobs to be scheduled within some timescale. The renewable generation source curve is represented as a function, $R(t)$, mapping each time in the discrete or continuous timescale interval $[0, T]$ to the real value being generated by the renewable source at that time. This chapter assumes that $R(t)$ is known in advance for the entire interval

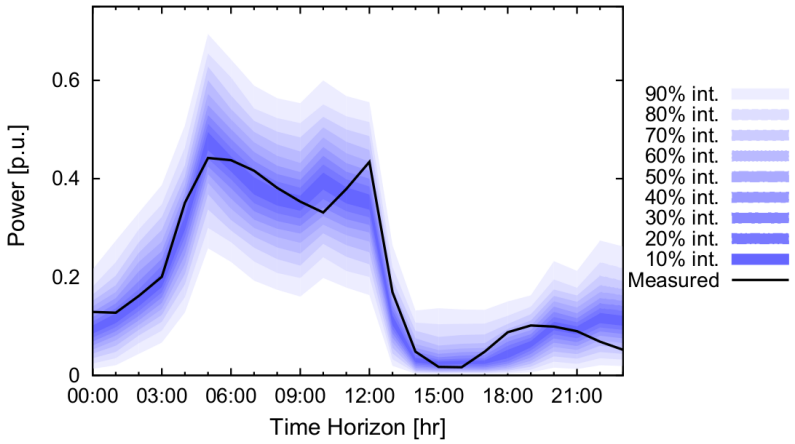


Figure 3.1: Probabilistic wind energy forecast from [25]

$[0, T]$. As discussed on page 31, this is a strong assumption for the integration of general renewable resources. On page 34, relevant forecast error modeling work was discussed to be used in conjunction with this chapter to help control for the uncertain nature of renewable energy generation.

Each job j is non-preemptable, meaning once it begins execution, it must run to completion without any interruptions. Each job j is characterized by four parameters: an arrival time (a_j), deadline (d_j), and length (l_j) that are real-valued for continuous timescales and integer-valued for discrete timescales, and a real-valued instantaneous demand (h_j) which is conceptually the height of the job. The arrival time and deadline are within the fixed time interval $[0, T]$ and form the *execution window* of the job. Job j is scheduled by assigning it a start time, s_j, which allows it to run in the closed interval $[s_j, s_j + l_j]$ such that $[s_j, s_j + l_j] \subseteq [a_j, d_j]$. The set of jobs to be scheduled can have a variety of parameter values, which leads to staggered execution windows.

The *non-renewable demand* at time t is the difference between the sum of the job heights that are scheduled to run during t and the amount of renewable energy

being generated at t:

$$N(t) = \sum_{j:t\in[s_j,s_j+l_j)} h_j - R(t).$$

Then, the *peak non-renewable demand*, $N^{\max}$, of the schedule is the maximum non-renewable demand for any time t in $[0,T]$,

$$N^{\max} = \max_{t\in[0,T]} N(t).$$

Definition 3.1. *The* ***Peak Demand Minimization with Renewables Problem (PDM-R)*** *is, given a set of n jobs, determine a job schedule, $\mathcal{S} = \langle s_j \rangle$, so as to minimize $N^{\max}$.*

If the renewable generation is uniformly zero, $R(t) = 0$ for all t, then the PDM-R problem simplifies to the PDM problem, as $N(t)$ becomes the sum of heights of jobs running during time t. The key nomenclature for the system model is summarized below.

n - Number of jobs.

a_j - Job arrival time.

d_j - Job deadline.

l_j - Job length (duration).

h_j - Job height (instantaneous demand).

$R(t) \in \mathbb{R}$ - Renewable energy generated at time t.

$N(t) \in \mathbb{R}$ - Nonrenewable energy required at time t.

Computational Complexity

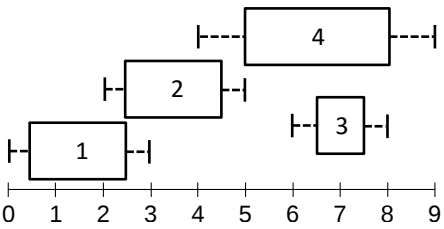
In [101], PDM was shown to be NP-hard to approximate within a ratio of 2 via a reduction from the *Scheduling with Release Times and Deadlines on a Minimum Number of Machines* (SRDM) problem. Since any instance of the PDM problem is merely a PDM-R instance with the renewable generation function of zero, PDM-R is also NP-hard to approximate within a ratio of 2.

Algorithms

This section introduces four algorithms for the PDM-R problem. The first is an optimal, fixed-parameter tractable algorithm, the second is a linear programming relaxation that is a $O\left(\frac{\log n}{\log \log n}\right)$-approximation algorithm for the general PDM problem when the renewable generation is zero, and the third and fourth are effective online and offline greedy heuristic algorithms. A renewable energy forecast curve is provided when running all of these algorithms. The user is assumed to have chosen a forecast curve, via the probabilistic forecasting techniques discussed on page 35, that will reflect the confidence that they require the actual peak nonrenewable consumption to be at most the peak of the algorithm's schedule.

An Optimal Algorithm

This section details a branch-and-bound, dynamic programming algorithm inspired by the algorithm presented in [101]. The algorithm in this section takes a similar approach of identifying groups of overlapping jobs, but differs from there by utilizing trees to represent and explore schedules. Tree representation of schedules reduces exploration of the search space, resulting in a 15x increase in the number



Job	a_j	d_j	l_j
1	0	3	2
2	2	5	2
3	6	8	1
4	4	9	3

Figure 3.2: Sample jobs for scheduling

of schedulable jobs over the algorithm presented in [101]. This algorithm is also generalized to handle the renewable aspect of the PDM-R problem.

At a high level, jobs are first ordered by increasing deadlines. For each job, consider every possible configuration of start times of that job and as well as other earlier jobs that could overlap any job in the future. Job configurations are represented using trees where each level in the tree represents the scheduling choices of one job. Keep track of which configurations of the current job are compatible with those of the previous job using pointers. Valid schedules can be built by traversing these trees backwards, beginning with the final job in the list. Branch-and-bound techniques are employed to curtail unnecessary schedule exploration if it is known that the schedule cannot result in a peak below the value of an already known schedule.

Configuration Lists Sort the jobs by increasing deadline first and then by increasing arrival time. For each job j, a list of preceding jobs $\mathcal{L}_j$, called the *configuration list*, is defined as the set of jobs whose execution windows either overlap j's execution window, or overlap the execution window of a job following j. Figure 3.2 illustrates a sample set of jobs along with their configuration lists: $\mathcal{L}_1 = \{1\}$, $\mathcal{L}_2 = \{1, 2\}$, $\mathcal{L}_3 = \{2, 3\}$, $\mathcal{L}_4 = \{2, 3, 4\}$. Observe that each job configuration list consists of a consecutive interval in the sorted list of jobs, i.e. $\mathcal{L}_j = [s(j), j]$, where the head $s(j)$ of each list has the earliest deadline (and arrival in case of ties).

Lemma 3.1. *The list heads are non-decreasing. (i.e.* $s(1) \leq s(2) \leq \ldots \leq s(n)$*).*

Proof. Suppose job $j \leq k$. Since $s(k) \in \mathcal{L}_k$, $\exists l \geq k$ *s.t.* $a_l < d_{s(k)}$. If $j \in \mathcal{L}_k$, then $s(k) \in L_j$ since $a_l < d_{s(k)} \leq d_j$ and $l \geq k > j$. Thus $s(j) \leq s(k)$. If $j \notin \mathcal{L}_k$, then $d_{s(j)} \leq d_j < d_{s(k)}$ and so $s(j) < s(k)$. □

Configuration Trees For each job j, represent each possible configuration of start times of all the jobs in its configuration list $\mathcal{L}_j = [s(j), j]$ using a *configuration tree*, $T(j)$. Consider the jobs in reverse order starting from j and create a branching tree where nodes at level k in the tree correspond to the possible starting times for job $j-(k-1)$. (Assume the root of the tree is at level 0). Thus, each leaf in $T(j)$ specifies start times for all jobs in $\mathcal{L}_j$. Let height(l) be the peak height above the renewable energy curve of this particular schedule of $\mathcal{L}_j$ job start times.

We need to keep track of compatible job configurations in order to build a consistent schedule for all jobs. This is done by maintaining a compatibility pointer, $c(l)$, from each leaf node l in $T(j)$ to a compatible node in $T(j-1)$ $(j > 1)$. By Lemma 3.1, $\mathcal{L}_j \cap \mathcal{L}_{j-1} = [s(j), j-1]$. Thus we can simply follow the same path in $T(j-1)$ that we took beginning with a node at level 2 in $T(j)$ to reach the compatible node in $T(j-1)$. If $[s(j), j-1] = \emptyset$, then simply make each leaf in $T(j)$ point to the root of $T(j-1)$.

Dynamic Programming Once configuration trees $T(j)$ are constructed and compatibility pointers $c(l)$ found, then we can compute the minimum overall schedule efficiently using dynamic programming as follows:

For each leaf node $l \in T(j)$, store $h(l)$ which is the peak height above the renewable energy curve of an optimal schedule that is compatible with l. For each node $m \in T(j)$, store a pointer, $b(m)$, to a leaf l in the subtree rooted at m with best

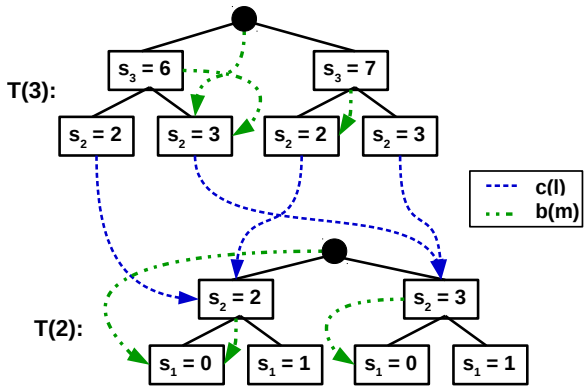


Figure 3.3: Sample configuration trees with $c(l)$ and $b(m)$ pointers

(i.e. lowest) peak height $h(l)$. The $b(m)$ pointer for a leaf node is to itself. The h values satisfy the following recurrence,

$$h(l) = \begin{cases} \text{height}(l) & \text{if } l \in T(1) \\ \max[h(b(c(l))), \text{height}(l)] & \text{if } l \in T(j), j > 1. \end{cases} \tag{3.1}$$

Note that once the $h(l)$ values are found for a given tree $T(j)$, it is easy to compute the $b(m)$ pointers for the internal nodes $m \in T(j)$ in a bottom-up fashion. Thus we can apply dynamic programming to find the h and b values for each job configuration tree in order, beginning with $T(0)$. An optimal schedule can be found by following alternating b and c pointers, starting with $b(\text{root}[T(m)])$. The peak height of this schedule will be given by $h(b(\text{root}[T(m)]))$. Figure 3.3 shows an example with two configuration trees, $c(l)$, and $b(m)$ pointers (with $b(m)$ pointers from leaves to themselves are omitted). Starting times in Figure 3.3 reflect the sample jobs from Figure 3.2. By following alternating b and c pointers, we see an optimal solution is $\langle s_3 = 6, s_2 = 3, s_1 = 0 \rangle$.

Branch-and-Bound To avoid creating large configuration trees that end up being unnecessary, a branch-and-bound technique can be employed to avoid explicitly building and storing configuration trees. The use of these tree structures and pointers serve as a *memoization* technique to eliminate the need to recompute optimal sub-schedules when following similar paths through the set of possible schedules.

In this process, the configuration lists are created for each job as described above. We then keep track of the peak demand above the renewable energy curve for the best schedule found so far (initialized to $+\infty$). Begin the recursive process of building and exploring paths through the configuration trees by starting with the last job in the list. It then proceeds in a depth first fashion, creating subtrees of the next job as compatibility pointers are followed. Once a viable path has been established from the last job through the first, the peak demand value is updated, and the method recurses to explore alternate paths. If at any point an alternate path realizes an $h(l)$ value above the current peak demand value, that subtree can be pruned and all compatibility nodes referencing it ignored.

Pseudocode for this recursive exploration is detailed in Algorithm 3.1, which uses the following definition: For node m in configuration tree T, define $height(m)$ to be the peak height above the renewable curve for the jobs scheduled by the path in T from $root(T)$ to m.

Fixed-Parameter Tractability. An algorithm is said to be fixed-parameter tractable if there exists some parameters, $p_1, p_2, ...,$ of the input or output such that the running time of the algorithm is a polynomial function of the input size times some function of the parameters: $f(p_1, p_2, ...) \cdot poly(n)$. The complexity of the algorithm is driven by the size of the configuration trees $T(j)$, that can be bounded by their branching factor and maximum depth. The maximum branching factor is $q = \max_j(d_j - a_j - l_j + 1)$ and the maximum depth is $r = \max_j(|\mathcal{L}_j|)$. Thus the

Let $best = +\infty$ be the peak demand above the renewable curve.
Let *lastJob* be the last job in the sorted input.
Call $explore(root(lastJob))$ where $explore(m)$ for node m is defined as follows:

```
if height(m) < best
    if m is an internal node
        sort children by increasing height
        forall child nodes
            explore(child)
        endforall
        b(m) = b(child*), for minimal h(b(child*)) of children of m.
    elseif m is a leaf node
        if c(m) is undefined
            construct c(m) in previous tree
            explore(c(m))
        endif
        compute h(m) using Equation 3.1
        b(m) = m
        if m ∈ T(lastJob) and h(m) < best
            best = h(m)
        endif
    endif
endif
return schedule found from b(root(lastJob)).
```

Algorithm 3.1: OptimalDP

maximum number of nodes in each configuration tree is $O(q^r)$. Since the dynamic programming time is easily seen to be linear in the size of the trees, the total running time is $O(q^r \cdot n)$.

A Randomized Rounding Algorithm

This section details a rounding scheme of a linear programming relaxation. When the renewable generation is zero, this algorithm schedules jobs with a peak height of at most $O\left(\frac{\log n}{\log \log n}\right)$ times the optimal peak schedule height, resulting in the first approximation algorithm for the general PDM problem. This approximation ratio is achieved via the use of Chernoff bounds [79] to bound the probability of the peak exceeding the ratio.

Integer Linear Programming Formulation The PDM-R problem can be easily formulated as an integer linear program (ILP) using the following notation and variables:

J - Set of jobs.

I_j - Set of all valid start/end time intervals for job $j \in J$. Interval is valid if $[s_j, s_j + l_j) \subseteq [a_j, d_j)$. The scenario where the timescale is continuous and I_j has infinite cardinality is discussed on page 49.

h_j - Height of job j.

L - Set of all left hand time points of intervals in $\bigcup_j I_j$.

$N^{\max} \in \mathbb{R}$ - Peak non-renewable energy required.

$R(t) \in \mathbb{R}$ - Renewable energy generated at time t.

$x_{i,j} \in \{0, 1\}$ - Indicates if interval $i \in I_j$ is scheduled.

The ILP then becomes:

objective:

$$\min N^{\max}$$

subject to the following constraints:

$$\sum_{i \in I_j} x_{i,j} = 1, \quad \forall j \tag{3.2}$$

$$\sum_{i \in \bigcup_j I_j : t \in i} h_j x_{i,j} \leq R(t) + N^{\max}, \quad \forall t \in L \tag{3.3}$$

Constraint 3.2 ensures that exactly one interval will be selected for each job, and constraint 3.3 ensures that $N^{\max}$ will denote the maximum amount of non-renewable energy required for any time in the entire schedule. Only elements of L need to be considered in constraint 3.3 as a consequence of Lemma 3.2. When the renewable generation is zero, $N^{\max}$ represents the peak height of the schedule.

Lemma 3.2. *The maximum height of any schedule must occur for some $t \in L$.*

Proof. Schedule height only increases at the moment a new job begins processing, thus the peak height of a schedule must be initiated at the start time of some job. Since L is the collection of all possible start times for all jobs, some point in L must correspond to the time of the peak height of the schedule. □

The preceding ILP is polynomial in size for the case of a discrete timescale, where each I_j is finitely large. When the timescale is continuous and each I_j is potentially infinitely large, the ILP has an infinite number of variables. A process to translate a continuous instance into a discrete instance with at most a factor two degradation from optimal is discussed on page 49. Proceed from here with the assumption that the ILP has been discretized.

Randomized Rounding The ILP presented above cannot be efficiently solved due to the restriction of $x_{i,j}$ to integer values. To address this, construct a relaxed LP in the exact same fashion as the ILP, except the $x_{i,j}$ variables may take on any values in the range $[0, 1]$. After the relaxed LP is solved, a specific interval is selected for each job j by randomly selecting interval i with probability $x_{i,j}$.

Theorem 3.1. *When $R(t) = 0$ for all values of t, the schedule generated by the relaxed LP has a height of at most $O\left(\frac{\log n}{\log \log n}\right)$ times the height of the optimal ILP's schedule, with probability at least $1 - O(1/n)$.*

Proof. Suppose that $R(t) = 0$ for all values of t. This means that the ILP from page 44 reduces to the general PDM problem as constraint 3.3 becomes:

$$\sum_{i \in \bigcup_j I_j : t \in i} h_j x_{i,j} \leq N^{\max}, \quad \forall t \in L$$

For each $t \in L$, define the random variables Z_t and Y_t as,

$$Z_t = \sum_{i \in \bigcup_j I_j : t \in i} h_j x_{i,j}$$

$$Y_t = \sum_{i \in \bigcup_j I_j : t \in i} \frac{h_j}{h} x_{i,j}, \text{ where } h = \max_j \{h_j\}$$

Bounding the probability that Y_t exceeds some factor α times the optimal height (N^{opt}) divided by h results in the same bound for the probability that Z_t exceeds α times the optimal schedule height:

$$\begin{aligned} \Pr[Z_t > \alpha N^{\text{opt}}] &= \Pr[hY_t > \alpha N^{\text{opt}}] \quad (\text{since } Z_t = hY_t, \forall t) \\ &= \Pr[Y_t > \alpha \frac{N^{\text{opt}}}{h}] \end{aligned} \tag{3.4}$$

I now use a Chernoff-type bound to bound the deviation of Y_t above $\frac{N^{\text{opt}}}{h}$. Note that $E[Y_t] \leq \frac{N^{\text{opt}}}{h}$ by (3.3). Since $x_{i,j}$ are independent Bernoulli trials, for any given t, with $E[x_{i,j}] = p_{i,j}$ and $0 < \frac{h_j}{h} \leq 1$ for all jobs, the following bound exists for all $\delta > 0$:[1]

$$\Pr[Y_t > (1+\delta)\frac{N^{\text{opt}}}{h}] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\frac{N^{\text{opt}}}{h}}$$

Substituting $\alpha = 1 + \delta$ and gathering terms on the right hand of the expression into one exponential results in,

$$\begin{aligned}
\Pr[Y_t > \alpha\frac{N^{\text{opt}}}{h}] &< \exp\left[-(\frac{N^{\text{opt}}}{h})(\alpha\ln\alpha - \alpha + 1)\right] \\
&\leq \exp\left[-\alpha\ln\alpha + \alpha - 1\right] \quad (\text{since } \frac{N^{\text{opt}}}{h} \geq 1) \\
&< \exp\left[-\alpha(\ln\alpha - 1)\right], \qquad (3.5)
\end{aligned}$$

for all $\alpha > 1$. Next, choose $\alpha = \frac{4\log n}{\log\log n}$ and consider the value of $\ln\alpha - 1$:

$$\begin{aligned}
\ln\alpha - 1 &= \ln\left(\frac{4\log n}{\log\log n}\right) - 1 \geq \ln\log n - \ln\log\log n \\
&= \frac{\log\log n}{\log e} - \frac{\log\log\log n}{\log e} \quad (\text{change of base}) \\
&\geq \frac{1}{\log e}(\log\log n - \frac{1}{2}\log\log n) = \frac{\log\log n}{2\log e}. \qquad (3.6)
\end{aligned}$$

[1] Theorem 1 in [78] proves that $\Pr[X > (1+\delta)\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}$ for $\mu = E[X]$, but it is easy to see that this holds for all $\mu \geq E[X]$.

Plugging (3.6) and $\alpha = \frac{4 \log n}{\log \log n}$ into (3.5) yields,

$$\begin{aligned}
\Pr\left[Y_t > \frac{4 \log n}{\log \log n} * \frac{N^{\text{opt}}}{h}\right] &< \exp\left[\frac{-4 \log n}{\log \log n} * \frac{\log \log n}{2 \log e}\right] \\
&= \exp\left[\frac{-4 \ln n \log e}{\log \log n} * \frac{\log \log n}{2 \log e}\right] \\
&= \exp\left[-2 \ln n\right] = 1/n^2. \qquad (3.7)
\end{aligned}$$

Finally, Equation 3.4, Equation 3.7, Lemma 3.2, and the bound on the size of L presented at the end of the section starting on page 49 can bound the probability that the height of rounded schedule, N^{rnd}, is greater than $\frac{4 \log n}{\log \log n} N^{\text{opt}}$:

$$\begin{aligned}
\Pr\left[N^{\text{rnd}} > \frac{4 \log n}{\log \log n} N^{\text{opt}}\right] &\leq \sum_{t \in L} Pr\left[Z_t > \frac{4 \log n}{\log \log n} N^{\text{opt}}\right] \\
&= \sum_{t \in L} \Pr\left[Y_t > \frac{4 \log n}{\log \log n} \frac{N^{\text{opt}}}{h}\right] \\
&< \sum_{t \in L} 1/n^2 \\
&\leq |L|/n^2 \\
&\leq 2n/n^2 = 2/n \in O(1/n).
\end{aligned}$$

So, with probability $1 - O(1/n)$, $N^{\text{rnd}} \leq \frac{4 \log n}{\log \log n} N^{\text{opt}}$. □

Theorem 3.1 combined with results on page 49 that show an optimal solution for the discrete case yields a 2-approximation to the continuous case, provides a randomized $O\big(\frac{\log n}{\log \log n}\big)$-approximation algorithm for the PDM problem.

A Greedy Heuristic Algorithm

Finally, this section considers offline and online heuristics that greedily schedule jobs to have a start time with the smallest contribution to the peak amount of non-renewable energy required by the schedule.

The online heuristic schedules each job by simply finding the starting time that minimizes the maximum value of the difference between the sum of demand already scheduled and the renewable energy available for each time in the new job's duration. The complete algorithm is presented as Algorithm 3.2.

The offline heuristic schedules individual jobs in the same fashion as the online one, but the jobs that have tight execution windows are scheduled first and jobs with more space in their execution windows are scheduled later. The complete algorithm is presented as Algorithm 3.3 (note that $w_j = \frac{l_j}{d_j - a_j} \in (0, 1]$ measures the execution window tightness of job j; values closer to 1 indicate tighter jobs).

Continuous to Discrete Transformation

The ILP detailed on page 44 is polynomial in size for the case of a discrete timescale, where each I_j is finite, but not for continuous case in which jobs can be scheduled anywhere in the arrival-deadline window (I_j may be infinite). Using a technique similar to the one introduced in [34] for the unit-height machine minimization problem, we show that any continuous instance can be transformed into a corresponding discrete instance such that an optimal solution to the discrete instance provides a solution to the continuous instance that is within a factor of two from optimal.

First, schedule a subset of the jobs, J' while concurrently identifying a set of time points, P, such that each interval of every remaining job contains at least one element of P. This is accomplished by scheduling as many jobs as possible horizontally,

Define $height(\mathcal{S}, j, s_j)$ as schedule $\mathcal{S}$'s height with job j scheduled to start at s_j:

peakHeight = $\mathcal{S}$.peakHeight
for time, $t = s_j, \ldots, s_j + l_j - 1$
newHeight = $\mathcal{S}$.heightAt(t) + $h_j - R(t)$
if peakHeight < newHeight
peakHeight = newHeight
endif
endfor
return peakHeight

Build schedule, $\mathcal{S}$, by determining start times as follows:

forall jobs, j (in the order that jobs arrive)
$s = a_j$
for time, $t = a_j + 1, \ldots, d_j - l_j + 1$
if $height(\mathcal{S}, j, t) < height(\mathcal{S}, j, s)$
$s = t$
endif
endfor
start$_j$ = s
endforall

Algorithm 3.2: MinFit-Online

Sort jobs by decreasing $w_j = \frac{l_j}{d_j - a_j}$ values.
Execute Algorithm 3.2 on this ordered list of jobs.

Algorithm 3.3: MinFit-Offline

without any overlaps. Start at the first time point and proceed in this fashion: As soon as no job is being serviced, schedule the job that will complete the earliest from all remaining jobs.

P can be constructed from this initial scheduling by adding the start and end times from all the scheduled intervals in J' and by adding the release times and deadlines for each remaining job in $J \setminus J'$. Thus, $|P| \leq 2n$. Then, each interval of every job in $J \setminus J'$ must contain at least one point of P. If some interval of job $j \in J \setminus J'$ did not contain a point of P, then either it is nested inside a scheduled interval and thus would have been scheduled since it ends sooner, or it is totally separated from any scheduled intervals and thus would also have been scheduled as well.

Next, expand each interval of each remaining job j by moving its left and right endpoints to the nearest points in P to the left and right of the original endpoints, respectively. This results in a finite number (at most $|P|$) of possible intervals for job j.

Consider an optimal solution to the continuous instance. When we transform this solution to the discrete instance by expanding the intervals as above, the overall height of the solution may increase, as two intervals that did not overlap now overlap. It is easy to see that these intervals must have originally overlapped two consecutive points from P. Thus, the height of the schedule can at most double after interval expansion. This implies that the height of an optimal schedule for the discrete instance is at most twice the optimal height of the continuous instance.

Finally, observe that in the discrete instance, we only need to look at the height at times $t \in L$ to determine the overall height of the schedule; this is because we can slide any time t a little bit to the left until it touches the left endpoint of some interval without changing the overall height of the schedule above t. By our construction, $|L| \leq |P| \leq 2n$.

Experimental Results

Simulations were conducted using the *OptimalDP, RoundLP, MinFit-Online*, and *MinFit-Offline* algorithms. For a baseline comparison we also compare to an on-demand algorithm, *OnDemand*, that schedules jobs to start on their arrival times.

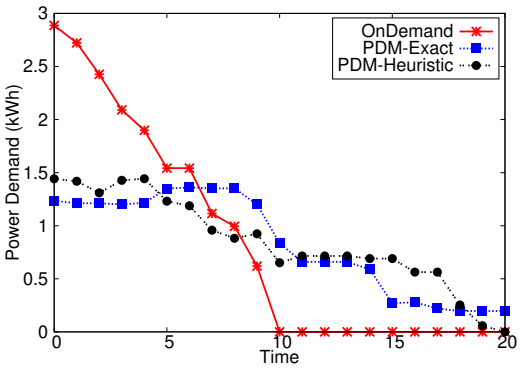
Jobs were created using appliance specific data from six residences [59]. We identified appliances (e.g. washing machine) likely to have flexible timelines and determined their height, length, and arrival time distributions within a 24 hour period. Deadlines were set to be uniformly distributed between the minimum possible deadline ($a_j + l_j - 1$) and the arrival time plus four times the average job length of that appliance.

Renewable energy generation curves were established using the National Renewable Energy Laboratory's Wind Integration National Dataset [41]. We sampled locations in the dataset nearest Bozeman, Montana and selected several days with representative wind generation curves.

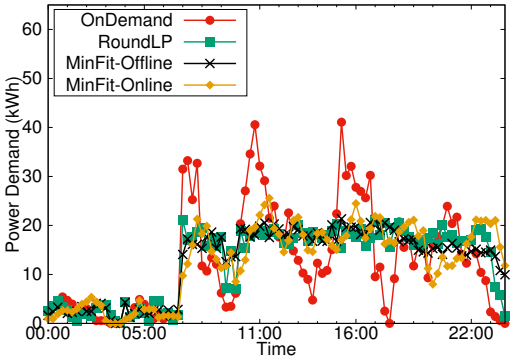
The first scenario we considered was a simple scenario meant to test the performance of the *OptimalDP* algorithm. Instead of generating jobs as described above, we randomly generated jobs with an arrival time of 0 to simulate a single peak. Figure 3.4a shows the average results of running *OptimalDP, RoundLP, MinFit-Online, MinFit-Offline*, and *OnDemand* on 20 iterations of a variable number of jobs, each from this simplified data generation method. *RoundLP, MinFit-Online*, and *MinFit-Offline* all produced near-optimal solutions at each job set size, in stark contrast to the solutions provided by *OnDemand*. The performance of *OptimalDP* is a great improvement over the related algorithm presented in [101] which was only able to schedule fewer than 10 jobs.

The second scenario we considered was the performance of the *RoundLP, MinFit-Online*, and *MinFit-Offline* algorithms with a larger number of realistically generated jobs, but no renewable energy available (i.e. the general PDM problem). Figure 3.4b shows the power demand versus time of day for the *OnDemand, RoundLP, MinFit-Online*, and *MinFit-Offline* algorithms scheduling 500 jobs (*OptimalDP* was unable to schedule more than about 150 of these jobs) over a 24 hour period. The peak demands in kWh were 41.1 for *OnDemand*, 21.0 for *RoundLP*, 21.3 for *MinFit-Offline*, and 25.6 for *MinFit-Online*.

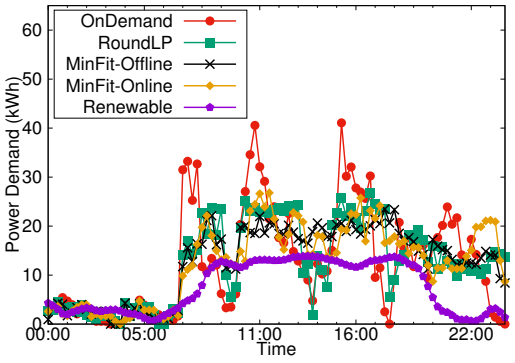
The final scenario we considered was the performance of the *RoundLP, MinFit-Online*, and *MinFit-Offline* algorithms with a large number of realistically generated jobs and the inclusion of a renewable energy generation curve. Figure 3.4c shows the power demand versus time of day for the *OnDemand, RoundLP, MinFit-Online*, and *MinFit-Offline* algorithms when scheduling 500 jobs over a 24 hour period, as well as the renewable generation curve. The same job instances were used for Figure 3.4b to enable comparison, hence the identical nature of the *OnDemand* schedules. The inclusion of a renewable energy source changes the resulting schedules as it enables the algorithms to shift demand and fill up the "free" energy that the renewable source provides. The raw schedule peaks here are not the metric of interest in this scenario. Instead, we are aiming to minimize the peaks above the renewable curve (i.e. the amount of non-renewable needed to support the schedule). The peak demands in kWh were 28.6 for *OnDemand*, 14.7 for *RoundLP*, 12.9 for *MinFit-Offline*, and 19.1 for *MinFit-Online*.



(a) Smaller test case



(b) Domestic power scheduling without renewables



(c) Domestic power scheduling with renewables

Figure 3.4: Performance of PDM-R scheduling algorithms

Table 3.1: Algorithm performance summary

	Strengths	Weaknesses
OnDemand	No processing needed	Poor performance
OptimalDP	Optimal schedules	Schedules < 200 jobs
RoundLP	PDM guarantee	No PDM-R guarantee
MinFit-Offline	Minimal computation	No guarantee
MinFit-Online	No day-ahead jobs	No guarantee

Conclusions

Integration of renewable energy into microgrids will require novel scheduling techniques to compensate for the variable nature of renewable generation. In this chapter, we introduced the PDM-R problem and presented a fixed-parameter tractable algorithm, a linear programming relaxation scheme that yields the first approximation algorithm for the general PDM problem (a $O\big(\frac{\log n}{\log \log n}\big)$-approximation algorithm), and online and offline heuristics. A summary of the strength and weaknesses of each approached presented is provided in Table 3.1. These approaches improve the generality of current PDM demand response scheduling algorithms.

The linear programming relaxation scheme presented on page 44 aims to minimize the deterministic peak of non-renewable energy consumed. Future efforts include incorporating renewable resource uncertainty directly into this algorithm by minimizing the expected value of the stochastic peak instead of minimizing the current deterministic peak. We are also pursuing a hybrid offline/online approach that determines optimal schedules for predicted scenarios [75, 108], and then makes adjustments in real-time based on actual conditions. Finally, it may be possible to generalize the proof of Theorem 3.1 to when the renewable generation curve is non-zero, thereby providing an approximation algorithm for the PDM-R problem.

CLOUD RESOURCE ALLOCATION

This chapter explores scheduling the allocation of cloud resources presented in [100]. Given a set of datacenters and groups of application clients, well-connected datacenters can be rented as traffic proxies to reduce client latency. Rental costs must be minimized while meeting the application specific latency needs. This leads to the formalization of the *Cooperative Group Provisioning* (CGP) problem which turns out to be NP-hard to approximate within a constant factor. A novel greedy approach is introduced that demonstrates promise through extensive simulation using real cloud network topology measurements and realistic client churn. Finally, multi-cloud deployments are found to dramatically increase the likelihood of meeting group latency thresholds with minimal cost increase compared to single-cloud deployments.

Background

Innovative Internet applications and services increasingly need low latency communication to deliver a responsive user experience. Augmented reality, online gaming, video chat, and real-time language translation are examples, where a small decline in responsiveness is noticeable to users and can eventually lead to a drop in usage, resulting in loss of application revenue [13, 22, 30, 50, 72, 92]. Many of these applications include a social component, which allows users to *cooperate* in a shared online experience. Recent work shows the impact of one user's *lag* on the Quality of Experience of all group members in a cooperative game and argues the need to meet latency requirements for all members [52].

Network latency, a significant component of overall lag, is difficult to control when it stems from conditions in the public Internet, such as congestion and circuitous forwarding routes [32, 48]. For example, public network switching overhead comprises

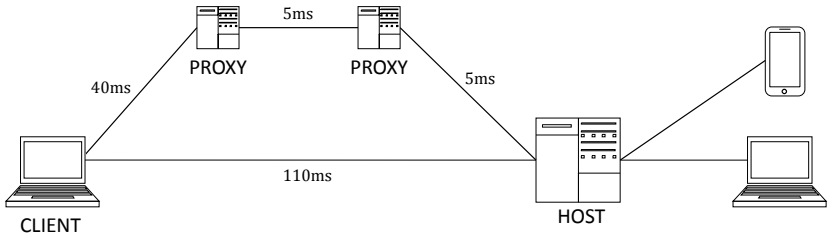


Figure 4.1: Proxy network to decrease end-to-end latency for clients

38% of end-to-end latency [48]. Large-scale application providers reduce the public Internet's latency for their users by distributing private resources (e.g. traffic proxies, VPNs) closer to users and forwarding application traffic within private networks [32]. By contrast, small-scale application providers do not have access to the same scale of private infrastructure and remain at a disadvantage. This disparity makes it challenging for small providers to deliver a comparatively responsive and affordable application experience for all their users.

Despite this disadvantage, small-scale providers can leverage the increasingly geographically dispersed public cloud datacenters to reduce user request delay. Cloud datacenters are interconnected with low-latency Internet paths and are often located with large population centers. Small-scale application providers can employ cloud resources in multi-cloud deployments to act as a proxy network, thereby leveraging its low-latency properties [57, 73]. These low latency paths introduce triangle inequality violations which can be used in place of default routing paths to reduce end-to-end delay. Figure 4.1 shows an example network, in which proxies reduce path latency. Previous research has demonstrated how public cloud proxy networks reduce latency between last-mile networks and distant application datacenters [48, 96]. Large cloud providers such as Amazon (AWS) and Microsoft (Azure) already offer proxy products, but only within their networks [2, 10].

This chapter addresses two challenges of cloud proxy networks:

First, can small-scale application providers deploy multi-cloud proxy networks in a cost effective manner? Larger geographical proxy distribution reduces average client path latency, but also increases cloud rental costs. Further, inter-datacenter traffic is more expensive when the two datacenters do not belong to the same cloud provider. Despite the potential for higher costs of widely distributed deployments, an attractive tradeoff between latency and cost is demonstrated, even in multi-cloud deployments.

Second, how can small-scale application providers allocate cloud resources to meet latency constraints of cooperative group applications? I propose a method to manage cloud resources (servers and communication links) to minimize their rental cost subject to a group latency threshold. The model considers link and forwarding latency as part of total latency. I formally define the Cooperative Group Provisioning (CGP) problem, show that it is NP-hard to approximate within a constant ratio, and propose an algorithm with good performance on multiple simulated cloud provider networks.

These results enable small-scale providers to incorporate multi-cloud proxy networks as one of the tools in the design of responsive distributed systems. Our algorithms offer developers a level of currently unavailable control over end-to-end network performance and will allow them to design cooperative group applications with explicit latency/cost budgets, or predictable performance and cost tradeoffs.

The remainder of this chapter discusses related work, a detailed system model, CGP problem formalization, algorithmic solutions, and an evaluation and conclusion.

Related Work

Multi-cloud deployments are a rapidly maturing research area. A growing number of tools make it possible for developers to build applications that simultaneously use geographically distributed resources from several cloud providers. I present the state of scheduling tools and algorithms in that space to help frame the CGP problem.

Distributed Resource Management

The continued expansion of cloud computing creates opportunities for widely distributed application deployments. At the same time, standardization of cloud interfaces and advances in multi-datacenter and multi-cloud application deployments make it increasingly feasible for small scale providers to build dynamic proxy networks that adjust to application demands and network conditions.

Overlay networks have been explored to manage Internet traffic, with the goal of decreased loss rate [18] and reduced latency [85]. The contributions of this chapter differ from traditional overlay networks in that the network proposed (public multi-clouds) incur rental costs, and resources are chosen based on aggregate utilization, not on a client-by-client basis. This chapter shows that the use of multiple public cloud providers has the potential to lower the latency between users and the host server for group applications. This chapter also proposes an approach to mitigate the extra costs of multicloud deployments, to make such deployments practical.

Large-scale providers developed tools (e.g. Borg, Mesos, Kubernetes, Omega) for application resource management within and across their datacenters [51, 67, 86]. Container technologies (e.g. Docker, ZeroVM) complement resource management frameworks to simplify the packaging of application code and OS dependencies to make code portable across heterogeneous cloud resources [7, 63]. These technologies

make it possible for small-scale application providers to achieve both agile vertical and horizontal resource scaling in multi-cloud deployments.

These solutions, however, do not advise developers on where and when to instantiate virtual resources. Conductor considers price, performance, as well as cloud service transfer costs to schedule MapReduce jobs [95]. Meryn considers computation time and throughput to meet job QoS constrains in platform-as-a-service (PaaS) clouds [40]. Quasar rectifies resource underutilization in Mesos and allows jobs to specify performance goals, notably in terms of latency [39]. Finally, Sparrow and alsched use a queue balancing method to schedule small jobs for fast completion [72]. These tools are not immediately suitable for multi-cloud deployments, because they do not take into account the cost and network performance between clients and datacenters. CloudTalk considers application traffic patterns and computes the cost of various placement strategies, but falls short of recommending a deployment strategy [80].

Recent multi-cloud deployment work does not address the costs and communication latency between clients and datacenters. Paraiso *et al.* describe a Multi-Cloud-PaaS (MCP) architecture to manage elasticity across multiple providers as a backstop to cloud outages [73]. Keahey *et al.* propose a system that provides similar elasticity for the software-as-a-service (SaaS) model in response to application-specific indicators [57]. Their model considers high network delay as an indicator of failure, but does not support latency-based deployment optimization. Work by Franceschelli *et al.* provides cost and performance estimates for multi-cloud deployments based on service layer objectives [46].

Finally, several works provide portals for multi-cloud resource management. Alrokayan and Buyya propose Cloud Web Portal (CWP) as a unified interface to manage VMs distributed across multiple clouds [16]. CWP scheduler takes network

latency into account, but only in the context of batch task distribution, for example rendering jobs. Similar approaches for management of multi-cloud resources include Delta Cloud, jClouds, and CloudFoundry [4, 6, 8]. Commercial services in that space include RightScale, enStratus, and Kaavo [5, 9, 11]. These architecture are extendible and could present a useful platform for the deployment of the algorithms presented in this chapter.

Related Optimization Problems

The systems solutions discussed above address the scheduling problem in the cloud, but do not take into account delay between clients and datacenters or group latency constraints. The problem of selecting cost effective, latency constrained paths through a network is similar to the well studied problem of Constrained Shortest Path (CSP).

In CSP, the single source/destination cheapest path, subject to a latency bound, is sought. Bernstein introduced a near linear time PTAS for undirected graphs that approximates both the path cost and latency [23]. Heuristics have been developed that have been found to be very efficient in practice [31]. CSP, however, does not have node weights to represent datacenter rental costs. Further, datacenter rental costs are shared by multiple clients, and CSP only considers single source/destination paths which doesn't provide the ability to capture shared costs.

The Constrained Steiner Tree (CST) problem, where the cheapest Steiner tree is found subject to a latency threshold, does share costs among multiple source nodes. Heuristics for the CST problem have been extensively studied since the early 1990's [60]. Modern approaches include genetic algorithms and ant colony optimization [42, 47]. CST also lacks node costs, restricts solutions to trees, and shares edge costs, which is not a realistic model for cloud networks.

Various efforts consider cloud resource allocation algorithms that respond to network conditions [14, 15]. These approaches do not include multiple groups of clients, nor do they consider application specific latency thresholds.

Problem Formalization

Consider a scenario where groups of clients look to connect to their other group members through a cooperative application (e.g. a group of friends playing an online game and a group of colleagues engaged in a video conference). Each group, and thus each client, has an application specific latency threshold, as well as a datacenter (DC) serving as the group's host. DCs exist in the network that can forward client traffic, in addition to application hosting.

These clients and DCs are all interconnected with links that have an associated usage cost and latency. Each DC has the capability to rent out virtual machines (VMs) that forward traffic. These VMs have an associated rental cost and a capacity, given in terms of the number of clients they are able to service simultaneously.

This network can be modeled as a graph, $G = (V, E)$, where the DCs and clients are nodes in the graph ($V = D \bigcup C$), and the links are edges. At each DC d, there is a VM rental cost $r(d)$, as well as a VM forwarding capacity $f(d)$. Each edge e has a rental cost $r(e)$ and latency $l(e)$, which includes the latency induced at the incident VMs. Each group (and thus, each client) has a hosting DC, $h_c \in D$, as well as a latency threshold, L_c. Many of these parameters can be time varying and this solution will use the current values at time of computation.

I make the following assumptions that reflect a broad range of network application architectures: 1) Each group's hosting DC, h_c, is determined ahead of time and

provided when the group is formed. 2) A VM can forward traffic for clients in different groups. 3) VM costs are accrued by number purchased (i.e. $r(d)$ will be charged to cover any number of clients less than $f(d)$). 4) Link costs are accrued by the number of clients using the link (i.e. $r(e)$ will be charged each time e belongs to any client's path).

For any path p_c from client c to its host h_c, the rental cost of c's connection is,

$$r(p_c) = \sum_{e \in p_c} r(e)$$

Likewise, the latency of c's connection via path p_c is,

$$l(p_c) = \sum_{e \in p_c} l(e) \tag{4.1}$$

Given a path p_c for each client c, the number of VMs that need to be rented at DC d is,

$$v_d = \left\lceil \frac{|\{c : d \in p_c\}|}{f(d)} \right\rceil$$

Finally, the total rental cost for deployment, given a path p_c for each client c, is the sum of the clients' connection costs plus the cost of each VM instance:

$$r = \sum_{c \in C} r(p_c) + \sum_{d \in D} v_d \cdot r(d) \tag{4.2}$$

Definition 4.1. *The input for the* ***Cooperative Group Provisioning (CGP)*** *problem is a set of clients nodes C, a set of DC nodes D (with cost and capacity values), links between nodes (with cost and latency values), an assignment of a host DC, h_c, for each client, as well as a latency threshold, L_c, for each client. The output is a path from each client to its host such that the latency of each path (equation 4.1) is below the latency threshold L_c and the total rental cost (equation 4.2) is minimized.*

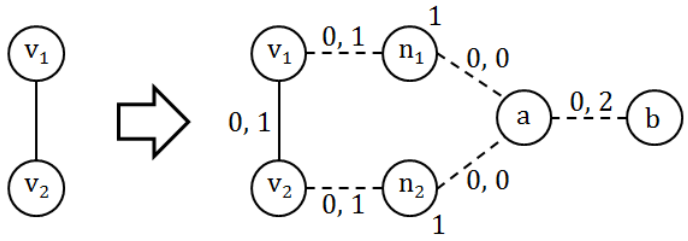


Figure 4.2: DOMINATING-SET reduction to CGP

Theorem 4.1. *There exists a constant,* $0 < \gamma < 1$*, such that the CGP problem has no* $\gamma \ln m$*-approximation, where* m *is the number of clients, unless* $P = NP$*.*

Proof. This result is via an approximation-preserving reduction from the DOMINATING-SET problem: Given a graph $G = (V, E)$, find a minimal $U \subseteq V$ such that for all $v \in V \setminus U$, there is a $u \in U$ such that $(u, v) \in E$.

Let $G = (V, E)$ be an instance of DOMINATING-SET where $|V| = m$. Reduce G to a CGP instance, G', as follows: Make each node in V have cost 0 and edge in E have weight = (cost, latency) of $(0, 1)$. Add new nodes a and b with cost 0 and connect them with an edge having weight of $(0, 2)$. Create a set of new nodes, N, by adding a new node n_i with cost 1 for each node $v_i \in V$. For each n_i, add an edge that connects to v_i with weight of $(0, 1)$, and add an edge that connects to a with weight of $(0, 0)$. Figure 4.2 shows the reduction from DOMINATING-SET to CGP.

Let the set of clients $C = V \cup b$, the latency threshold for all clients be 2, and every node in G' have an unlimited forwarding capacity. A solution to G' is a subset of N, rented to connect all members of C to a. Node a must be the host for any valid solution to G' because $b \in C$, and a is the only node reachable within the latency bound.

Suppose that $U = \{v_i, ..., v_j\}$ is a dominating set of G. Since a is the host, every element of U can be reached with a latency of 1. Since every other element in $V \setminus U$

can be reached in one more (latency of 1) hop, every client can be reached with at most a latency of 2. Thus, $\{n_i, ..., n_j\}$ are the only nodes from N needed to form a solution to G'.

Suppose there is a solution to G', $\{n_i, ..., n_j\} \subseteq N$. Since n_k corresponds to an element of V, the subset $\{n_i, ..., n_j\}$ represents a unique subset, $S = \{v_i, ..., v_j\} \subseteq V$. Since the latency of each path is at most 2, each element of V must be in S (latency of 1) or directly connected to a member of S (latency of 2), making S a dominating set.

Since elements of N and V are in one-to-one correspondence, the number of nodes selected from N is the same as the number of nodes in the dominating set, thereby making the costs equal and the reduction approximation-preserving. It was shown in [81] that there exists a constant, $0 < \gamma < 1$, such that the DOMINATING-SET cannot be approximated within a factor of $\gamma \ln m$ unless $P = NP$, thus this result holds for CGP as well. □

Algorithms

This section presents a greedy algorithm for the general CGP problem as well as modifications to handle two important real-world considerations.

A Greedy Algorithm

Due to the complexity of the CGP problem, a greedy strategy for the problem is examined. Our approach is to consider multiple paths from each client to its host and select the path that will incur the least additional cost, considering the current state of the network (i.e. what VMs are already rented).

The algorithm first generates the k shortest paths (based on latency) from each client to its host and discards any paths that are over the latency threshold. For

each path, the path's cost is determined to be the sum of the link costs plus the forwarding cost incurred at each intermediate DC. This forwarding cost depends on the number of forwarding VMs currently owned at each intermediate DC and the capacity remaining on each VM. The paths are then sorted based on the total cost. The path with the least total cost is selected and added to the solution. Any VM purchases that need to be made to support this path are recorded. The process then loops back and determines the path costs for the paths of the remaining clients. This algorithm is presented as Algorithm 4.1.

On a CGP instance:
- **forall** clients in CGP instance
  - Generate k shortest (by latency) paths for client
  - Add paths below client's latency threshold to $\mathcal{P}$
- **endforall**
- **while** $\mathcal{P}$ is not empty (i.e. client(s) still need to be scheduled)
  - **forall** paths, p, in $\mathcal{P}$
    - Let cost of p be sum of link costs and VM forwarding costs
  - **endforall**
  - Schedule the least cost path in $\mathcal{P}$
  - Remove all paths from $\mathcal{P}$ for scheduled client
  - Update VM forwarding numbers along the scheduled path
- **endwhile**

Algorithm 4.1: Greedy-k

A variety of methods could be used to generate the k shortest paths in Step 1 of the algorithm. For the evaluation on page 68, Yen's algorithm was used for finding the k shortest loopless paths [103]. Other shortest path algorithms could be used, or

determining paths based on parameters other than latency, such as total link cost, are possible.

Group Churn Considerations

Any functional solution to the CGP problem must address group churn. Instances will not remain static, as groups come online and go offline routinely. Algorithm 4.1 can be modified to account for churn. This algorithm relies on the same underlying principle as Algorithm 4.1 by scheduling the cheapest path each iteration, until all clients have an assigned path. Churn is handled by only scheduling paths to new clients. Legacy clients retain the initially assigned paths.

When a new group comes online, each client is assigned a path that it will use for the rest of its connection. As such, for each moment in time, only the new clients need to be scheduled and they are done so by Algorithm 4.1 with consideration given to the already rented VMs. This algorithm is presented as Algorithm 4.2.

On a CGP instance with groups joining and leaving over time:
    forall timeslots, t
        Let the solution for t start as the solution to $t-1$ (empty if $t=0$)
        Let $\mathcal{C}$ be the set of new clients for t
        Remove offline clients from the solution
        Execute Algorithm 4.1 on the set $\mathcal{C}$
        Add the solution from Algorithm 4.1 to the current solution
    endforall

Algorithm 4.2: Greedy-k Churn

Multicast Considerations

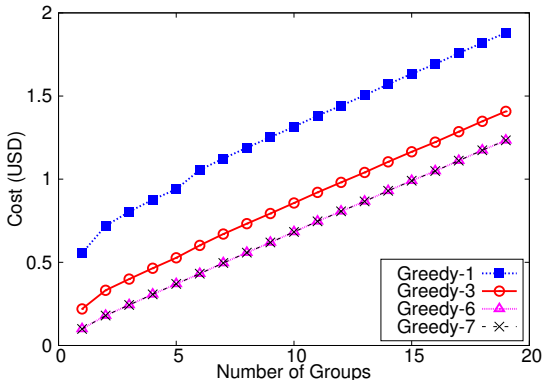
The algorithms described so far select paths for unicast traffic. Some applications, such as video conferencing, send identical traffic to all group members. It is cost effective to distribute this traffic in a multicast fashion. Algorithm 4.1 can be used to determine multicast traffic paths.

Algorithm 4.1 does not need to be modified to support multicast distribution. Instead, the input graph is modified. Existing nodes and edges between clients and DCs are not changed. New intermediate nodes are added to all inter-DC links. The cost of VMs on the new nodes is the cost of the link it is added to and the forwarding capacity is infinite. The link now split in two has no cost and one of the sublinks has the old latency value.

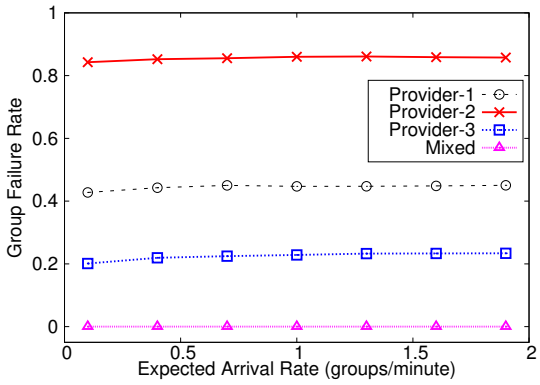
Since VM costs are shared by all clients being serviced, merely push the link cost to a VM on a new node. Then, when unicast paths are determined, they will leverage the existence of other traffic in the model (by way of already rented VMs). This enables link costs to be shared when identical data is transferred from host to clients.

Experimental Results

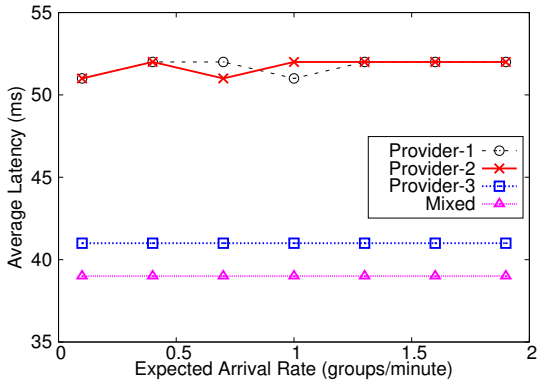
This section evaluates the effectiveness of the greedy algorithm from this chapter in meeting the QoS needs of cooperative group applications in multi-cloud deployments. The evaluation is based on an extensive set of simulations on networks configured with connectivity and pricing models of Amazon, Microsoft, and Rackspace cloud providers. To avoid comparisons, the identity of these services in the presented results are anonymized.



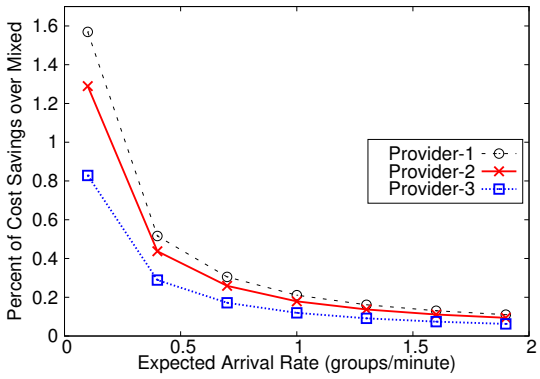
(a) k values: cost vs. num groups



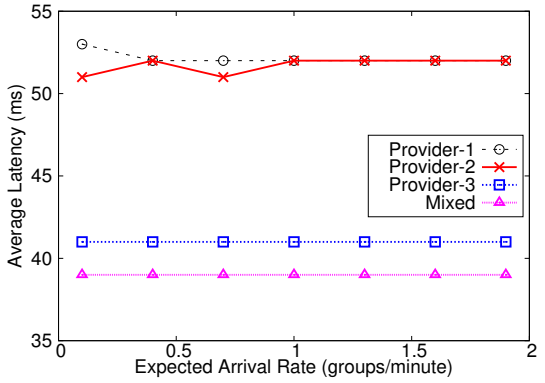
(b) Game churn: failure vs. arrival rate



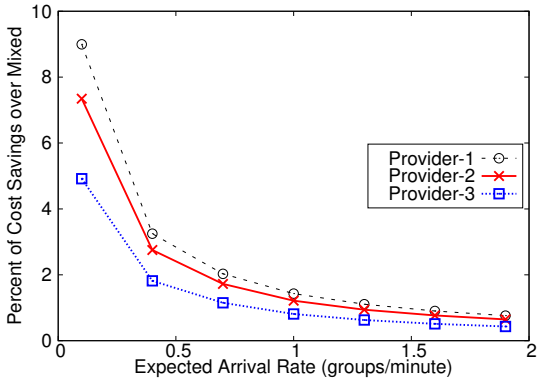
(c) Game churn: latency vs. arrival rate



(d) Game churn: cost vs. arrival rate



(e) Video churn: latency vs. arrival rate



(f) Video churn: cost vs. arrival rate

Figure 4.3: Performance of static and churn group instances

The simulation model used has been parameterized to reflect real-world connectivity between datacenters, as well as between datacenters and users in last-mile networks. The latency between datacenters is estimated as their great-circle (orthodrome) distance divided by half the speed of light to account for switching delay and coiled fiber in fiber optic conduits [93]. Latency between two datacenters that belong to the same cloud provider is assumed to be lower, because traffic traverses VPNs, rather than the public Internet. As such, the distance of those links is only divided by two-thirds the speed of light. I assume the throughput of inter-datacenter links to be infinite, as their capacity is orders of magnitude larger than simulated traffic flows.

Server and bandwidth costs were taken from the advertised parameters of three top cloud service providers in the US [17, 68, 69, 77]. The link pricing scheme in the models reflects the lower cost of traffic between two datacenters that belong to the same provider.

I randomly sample user locations from the locations and relative population sizes of the 100 most populous US cities. The latency between these clients and datacenters is based on distance plus last mile latency of access networks in a user's city based on FCC data [44]. These values contain the effect of low bandwidth and traffic congestion in access networks. All latency values include the latency induced by the VMs and were corroborated through direct ping measurements from cloud providers to random hosts (location verified with IP to location database) in last-mile networks of the 100 most populous US cities. To randomize the latency of different hosts from the same city, we assume the latency measurement represents the median latency in that city, which is then increased or decreased based on the distribution of FCC measurements of latency for different link access speeds.

Clients utilizing two types of applications as simulated. The first is a gaming application with a latency threshold of 80 ms, group size of 12, and fixed session duration of 10 minutes [3, 37]. The second is a video chat application with a latency threshold of 200 ms, group size of 10, and an exponentially distributed duration [1, 88]. To calculate VM proxy capacity for these applications, the number of sessions that can be proxied by a VM on each data center without queuing request traffic is empirically evaluated. To calculate the traffic costs, the mean volume of flows in these applications was is measured using pcap captures.

Finally, simulations were performed on gaming groups without any churn to determine the optimal number of paths per client for which to configure Algorithm 4.1. The simulation was the average of ten iterations on instances of between one and twenty groups. The number of paths considered was one, three, six, and seven. Results of this simulation are shown in Figure 4.3a. Using three paths results in an average cost savings of 40% compared to using only one path. Using six or seven paths resulted in an average cost savings of 50% over a single path, and diminishing marginal benefits over three paths. Churn simulations were run with the number of paths for Algorithm 4.1 set at three.

Group Churn Evaluation

This section details the performance of Algorithm 4.2 with varying degrees of group churn and different cloud service provider networks over a 24 hour period. Group churn is modeled as group arrival times fitting a Poisson process. The degree of churn is manipulated by the expected arrival rate of groups, in groups per minute. Group duration is subject to the parameters discussed earlier.

I considered four different cloud solutions for group provisioning. Provider-1, 2, and 3 are the three top providers in the US. When Provider-n is referenced, only that

provider's network can be used. The fourth solution, Mixed, represents a multi-cloud deployment on Providers 1, 2, and 3.

Figure 4.3b shows failure rate of groups meeting their latency threshold (80 ms for gaming) as a function of group arrival rate. The Mixed network is the only one that had a zero percent failure rate. Since the greedy algorithm determines shortest paths based on latency, it is guaranteed to find a path under the latency threshold, should one exist. Thus, the failure rate is not artificially high due to peculiarities of Algorithm 4.2. The decrease in failure rate is related to the number of DCs in the network. More geographically distributed multi-cloud deployments provide more opportunity for proxies to meet latency thresholds. Figure 4.3b also indicates the advantage to employing proxies. In the model, clients are able to connect to an application host in a cloud DC without using proxies. Thus, any failures seen when using proxies would also be failures without proxies.

Figure 4.3c shows the average client latency for increasing churn rates. As expected from the failure rate results, the networks that experience higher failure rates also have higher average latencies. The curves are mostly flat, suggesting that the average latency for each provider is more dependent on the location of datacenters than it is on the number of active groups. The independence of latency from group churn is due to the fact that even a small number of active groups leads to VMs being rented at most datacenter locations, thereby keeping the average latency stable.

Figure 4.3d shows the percent of cost savings of Provider-1, 2, and 3 compared to the Mixed network over a 24 hour period with increasing churn rates. The cost savings of using a single-cloud deployment starts at at most 1.6% and decreases with increasing churn rate, which points to profitability of multi-cloud deployments. The small variability of cost savings is due to pricing similarities between providers, geographic proximity of DCs from different networks, and consistent selection of similar

resources. This result suggests that for nearly the same cost, latency thresholds can be met with much greater reliability by utilizing multi-cloud deployments.

Results of churn simulations on video chat groups are similar to the results for game groups. Failure rate was zero for all networks due to the high latency threshold of 200 ms. Even without any latency failures on the single provider networks, Figure 4.3e shows that the average latency is still lower with a multi-cloud deployment. Further, Figure 4.3f shows that the cost savings of using a single-cloud deployment is low and decreases with the increasing rate of churn.

Conclusions

In this chapter, the CGP problem was introduced and it was shown to be NP-hard to approximate within a constant factor. A versatile greedy approach was presented that is applicable for static instances, realistic group churn, and applications that are able to leverage multicast traffic. This approach has shown to be efficient in realistic simulations and has the guarantee of finding feasible solutions if they exist.

Simulations on current US cloud providers indicate that multi-cloud deployments are cost effective in reducing latency. Larger numbers of DC locations lead to greater selection of resources to rent, which reduces average latency and group failure rate (tail latency). Simulations have shown that leveraging these larger cloud networks can be expected to not adversely affect the cost of application deployment.

TURNSTILE PLACEMENT PROBLEM

This chapter presents research in placing monitors in networks so as to determine flow volumes on each edge (paper containing these results in review). Network administrators and traffic management tools need up-to-date traffic metrics to monitor and balance traffic load. Recording and reporting flow information is costly in terms of control plane traffic and router memory. Prior work mitigates these costs by limiting the number of monitoring devices, sampling, or only reporting constraint violations. However, these techniques are limited to single-path routing, do not address multiple network flows, and do not guarantee network coverage. This chapter proposes strategies to minimize the cost of placing a sufficient number of logical monitors on edges (called *turnstiles*) so as to monitor all (possibly multipath) flows. The main result is a $(\ln m + 1)(\ln k + 1)$-approximation algorithm for the general *Turnstile Placement* problem, for a network with m edges and k flows (commodities). It is also shows to be NP-hard to achieve an $o(\ln k)$ approximation ratio. A simple heuristic algorithm for the problem is examined as well as a method to adapt existing single path solutions. Simulations show that the approximation algorithm achieves near optimal performance and outperforms existing approaches. The proposed methods reduce monitoring costs in multipath networks, enabling more agile and more accurate load balancing of network flows.

Background

With the increasing prevalence of Internet-connected devices, delivering data promptly continues to be an important challenge. Software defined networks (SDNs) reduce network management complexity through a simple, yet powerful control model. A logically centralized controller with a global view of the network state installs

forwarding rules on routers, which match and apply them to arriving packets. A crucial source of controller information are flow volumes collected by router hardware counters. Based on this information, the controller alters forwarding rules to balance network load.

As SDN controllers become integrated with application delivery controllers (ADCs) to meet application and even flow level quality of service (QoS) requirements, they require flow volume information of increasing spatial and temporal resolution [12,64]. These flow-specific metrics can be collected from routers configured to monitor flow traffic, hereon simply referred to as *monitors*. Such detailed network state, if recorded and reported naïvely, creates control plane congestion and limits the scalability of the centralized control model in dynamic network scenarios.

Optimizing the placement of monitors is necessary to observe all flows in a network, and will help reduce the amount of control plane reporting overhead. This cost reduction will allow for up-to-date, low-cost, and complete reporting of flow volumes to the SDN centralized controller. Accurate reporting of flow specific metrics will help traffic management tools make load balancing decisions to improve network performance through congestion reduction. Perhaps most importantly, less costly monitoring will enable SDNs to take on more load balancing functions and assure the scalability of centralized network control in dynamic networks.

This chapter updates existing theoretical work on minimum monitor placement to include the multipath flows present in modern networks. We call the monitors used *turnstiles*, and place them logically on edges in the network. Physically, these monitors are not additional hardware, but rather traffic recording on specific router interfaces. The ***Turnstile Placement (TP)*** problem looks for minimum cost turnstile placements to monitor multiple multipath flows in a network. The TP problem formulation supports general turnstile cost functions for customized

performance metric optimization (e.g. minimize the number of turnstiles, minimize the distance to controller). I establish a polynomial time solution to the TP problem when there is only a single commodity by relating it to the cycle-transversal problem [97, 98]. I show the general TP problem is NP-hard to approximate within a bound of $o(\ln k)$, and introduce a novel $(\ln m + 1)(\ln k + 1)$-approximation algorithm for the TP problem, where m is the number of edges in the graph and k is the number of commodities. I also present a spanning tree based algorithm for the general TP problem that leverages the relationship between the single commodity TP problem and the cycle-transversal problem. Finally, a method for adapting existing single path flow monitoring solutions to multipath TP instances is developed.

To demonstrate the efficacy of these approaches, an evaluation is included that was primarily conducted by a colleague where the monitor placement costs of these algorithms are compared on real Internet topologies with realistic traffic patterns. The simulations show that the approximation algorithm placements are within 1% of optimal, on scenarios for which the optimal solution could be computed. On larger examples, the approximation algorithm outperforms the other methods proposed.

The rest of this chapter is organized as follows. Related work is discussed on page 76. Formalizing the TP problem, detailing its relationship to other fundamental graph problems, and discussing its complexity begins on page 79. Algorithms are presented on page 85. Experimental results are presented on page 93 and conclusions are made on page 99.

Related Work

Previous efforts to reduce the overhead of control traffic in network monitoring can be classified into three categories: (1) sampling or stopping monitor placement

after covering a portion of the network, (2) utilizing heuristics, and (3) considering a simplified problem formulation in which flows have a single path.

The first category of related work deals with monitor placement using sampling techniques [56, 89, 104, 106]. Jackson *et al.* focus on capturing a high percentage of network traffic data by using a minimum number of monitors on autonomous system boundaries [56]. However, considering different autonomous systems as single entities results in a loss of flow monitoring precision at the router level. Another common approach to reduce monitoring load is to sample only a fraction of packets at a router [89, 104, 106]. Suh *et al.* , in one problem formulation, consider the problem of monitoring flows by maximizing the coverage without violating the deployment and operating costs of monitors, while optimizing sampling rates [89]. Cantieni *et al.* also optimize monitor placement and sampling strategies to achieve high monitoring coverage [26]. However, sampling approaches result in a probabilistic coverage of the network flows, which can lead to lower accuracy of flow volume estimation and increase the likelihood of missing smaller flows. Probabilistic and redundant coverage of sampling is also a drawback of monitoring tools available on routers such as NetFlow, or CMON [36, 55].

The second type of solution uses heuristics to place monitors [27, 53, 76, 104, 105]. Zang *et al.* propose greedy heuristics that monitors flows on each edge [104]. Huang *et al.* optimize monitor placement in dynamic routing scenarios using one MILP and two heuristics [53]. The authors later introduce a framework, LEISURE, that balances monitoring tasks equally between different monitors in the network [27]. Heuristic solutions, however, do not offer performance guarantees.

Third, previous work considers single path routing for each flow [26, 29, 89]. Chaudet *et al.* considers minimizing monitoring overhead with passive monitors by minimizing the number necessary to observe the network edges [29]. Assuming fixed

path routing allows the authors to show the problem's equivalence to Set Cover as well as allowing for a reduction from their problem to the Minimum Edge Cost Flow problem. In one formulation of the problem, Chaudet *et al.* consider the multipath version of the problem but only introduce an ILP with no additional complexity results. Suh *et al.* define several variations of network monitoring problems, without sampling, show them to be NP-Hard, and propose approximation algorithms and Integer Program solutions [89]. The shortcoming of single path routing is that they are not compatible with multipath forwarding of equal-cost multipath routing (ECMP), or multipath TCP (MPTCP).

This chapter takes two approaches to addressing these gaps. First, it provides an approximation algorithm for multipath flow instances that achieves a performance guarantee and a novel heuristic. Second, it provides a method to adapt existing single path flow monitoring solutions in a multipath context. These solutions also use a flexible cost function able to minimize the overhead of monitoring control traffic.

One final related work category monitors vertices in the network, rather then edges [49, 74, 76, 105]. Gupta *et al.* develop techniques to reduce overall monitoring overhead in wireless mesh networks [49]. One of the techniques that they utilize is minimizing the number of routers used as monitors with a modified vertex cover approximation algorithm that prioritizes vertices with a high degree. Park and Lee also maximize network coverage while reducing the number of monitors, placed on vertices, to detect DDoS attacks in networks [74]. Zeng *et al.* and Qin *et al.* focus on minimizing the number of vertex monitors in a network using heuristics [76, 105]. These approaches have the cost equivalent to monitoring traffic on all the edges at a vertex – a solution subsumed by the approach taken in this chapter and avoided in practice due to its high cost.

Problem Formulation

Consider a network, comprised of a set of vertices V and directed edges E. Each of the k flows in the network consists of a source s_k and destination t_k vertex, along with a set of edges, $E_k \subset E$, hosting the flow. Flow instances (s_k, t_k, E_k) are known, but traffic volumes at each edge are unknown. Additionally, a weight $c(e)$ is associated with each edge in the network, reflecting the cost of monitoring the traffic on that edge. This monitoring cost function enables optimization of various performance metrics (e.g. minimize the number of monitors placed, minimize distance between monitors and controllers).

By monitoring traffic patterns on a select set of edges, we seek to determine the flow volumes on all edges of the network. The network is subject to the normal conservation of flow assumption: The amount of flow into each vertex equals the amount out, except at sources and destinations. We aim to identify the amount of traffic traversing each edge in the network by utilizing *turnstile* monitors to disambiguate traffic volume on each edge. A turnstile is able to log all traffic that it encounters and is able to determine which flow the traffic belongs to. The placement of a turnstile incurs the cost of the edge weight associated with monitoring that edge. The goal is to place a minimum cost set of turnstiles so as to determine the traffic volume on every edge in the network. The Turnstile Placement problem is formally defined below and a sample instance with a valid turnstile placement is presented in Figure 5.1.

Definition 5.1. *Given a weighted, directed graph, $G = (V, E, c)$, where $c\colon E \to \mathbb{R}$ is a turnstile placement cost function, and a set of commodity flows $\{(s_k, t_k, E_k)\}$, where s_k is the source, t_k is the destination, and E_k are the edges used for the k^{th} commodity, the* ***Turnstile Placement (TP)*** *problem seeks a subset $M \subseteq E$ of*

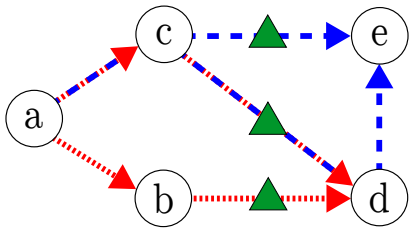


Figure 5.1: Sample TP instance with two commodities

minimum total cost, such that knowing the commodity flow volumes on the edges in M uniquely determines all commodity flows on all edges.

Turnstile Placement for a Single Commodity

The special case of the TP problem with a single commodity (one multipath flow) is called the ***Single Commodity TP (SC-TP)*** problem. Given an undirected graph, $G = (V, E)$, a *cycle transversal* of the graph is a set of edges, $S \subset E$, such that every cycle in the graph includes at least one edge from S [97, 98]. Establishing the equivalence between the SC-TP and the cycle transversal problem requires a slight modification to the SC-TP input. Complete a source-destination cycle by adding a back-edge from the destination to the source vertex. This modified graph is represented as $G = (V, E \cup \{(t, s)\}, c)$ for source s and destination t in V. With the back-edge in place, conservation-of-flow at each each vertex in G can be enforced, as the back-edge carries the total flow from s to t, and back to s.

Lemma 5.1. *Given an instance to the SC-TP problem, $G = (V, E \cup \{(t, s)\}, c)$ and a single flow, the set $T \subset E$ is a feasible solution to the SC-TP instance if and only if it is a cycle transversal for G with undirected edges.*

Proof. Suppose that $T \subset E$ is a feasible solution to an SC-TP problem instance. This means that the traffic values for each edge in E can be determined from the turnstiles on T, but the cost of T need not be optimal. Consider, for the sake of a

contradiction, a cycle in the undirected version of G that did not have an edge in T. If such a cycle existed, then an arbitrary amount of traffic r, could be added to each edge e of the cycle, where the flow on e is increased by r if the edge points in the direction of the cycle and $-r$ otherwise. Hence, edges can host unrestricted traffic values, and contradicts T being a solution to the SC-TP problem. Thus, T must provide a cycle transversal of the undirected version of G.

Suppose that $S \subset E$ is a cycle transversal for the undirected version of G. For the sake of a contradiction, further suppose that S is not a valid solution to the SC-TP problem. This means that there is some edge, (u, v), whose flow value cannot be determined from knowing the flows on the edges S. Since conservation of flow holds at each vertex, there must be at least one other edge incident to v whose flow is also not determined. Without loss of generality, say this is the edge from v to w. Again, there is an edge starting at w and going to some vertex other than v on which the flow is also undetermined. Note that this edge is necessarily not in S. Since G is finite, continuing to follow a path of undetermined edges in this manner must ultimately create a cycle without any edges in S. Existence of this cycle contradicts the assumption that S provides a cycle transversal. Thus, S must be a valid solution to the SC-TP problem. □

Since the same set of edges is used in an SC-TP solution as in a cycle transversal solution, the accumulated edge costs of these solutions are identical. Thus, a minimum cost cycle transversal solution equates to a minimum cost SC-TP solution, and vice versa. The equivalence of the SC-TP and cycle transversal problems provides a solution to SC-TP instances, as the minimum cycle transversal problem can be solved by determining a maximum cost spanning tree (easily computed using a standard minimum-weight spanning tree algorithm and negating all edge costs).

Lemma 5.2. *Given an undirected graph with edge costs, $G = (V, E, c)$, and a spanning tree, $T \subset E$, of maximum cost,the remaining edges, $E \setminus T$, form a minimum cost cycle transversal of G.*

Proof. Suppose that T is a maximum cost spanning tree of G. Since T is a tree, it cannot contain any cycles, and so any cycle in G must have at least one edge in $E \setminus T$, otherwise the entire cycle would be contained in T. Therefore, the set $E \setminus T$ forms a cycle transversal of G.

Since T is a spanning tree of maximum cost, the remaining edges in $E \setminus T$ form a minimum cost spanning tree complement, and therefore, a minimum cost cycle transversal of G. □

Since the back edges in the SC-TP instance are not actually present in the network, we wish to avoid selecting them for turnstile placement. If the back-edge costs are made larger than any real edge in the SC-TP instance, they will be included in the maximum cost spanning tree, thereby ensuring they will not be selected for turnstile placement.

Turnstile Problem Complexity

Lemma 5.1 and Lemma 5.2 show how the SC-TP problem can be solved optimally in polynomial time. However, The general TP problem is inapproximable within a bound of $w \ln k$ for some $w > 0$, where k is the number of commodities in the TP instance. This result is obtained via a reduction from the SET-COVER problem and details are included in Lemma 5.3. Due to the complexity of the TP problem, on page 85 an approximate solution is pursued in lieu of pursuing optimal approaches.

Lemma 5.3. *There exists a constant, $w > 0$, such that the TP problem has no polynomial time $w \ln k$-approximation algorithm, where k is the number of commodities, unless $P = NP$.*

Proof. This result is via an approximation-preserving reduction from the SET-COVER problem: Given a collection S of subsets of a finite sized universe, U, find a minimum sized $S' \subset S$ such that for all $u \in U$, u is contained in some set in S'. Let $S = \{S_1, ..., S_n\}$, where $S_i \subset U$ for all i, be an instance of SET-COVER. Reduce this instance to a TP instance as follows:

Create a graph component with a line topology having $n + 1$ vertices and n edges, where vertex j shares a directed edge to vertex $j + 1$. These n edges will each be sequentially associated with a unique member of S (e.g. edge from j to $j + 1$ corresponds to set S_j). Let the turnstile placement cost function be unit valued for all edges in the graph, thereby making the objective to minimize the total number of turnstiles placed.

Each element of the universe, U, will be represented by a single-path commodity flow. Do this by first creating, for each element in U, two new vertices to serve as that commodity's source and destination. For each commodity, i, consider the ordered list of sets, $S^i \subset S$ that its corresponding element of the universe is a member of. Create a source- to-destination path for each commodity by beginning at that commodity's source and making a directed edge to the vertex corresponding to the source vertex of its first element of S^i. So, if S_j is the first element of S^i, an edge will be made from commodity i's source to vertex j. Now, take the existing edge to that set's destination vertex and create a new directed edge to the source vertex of the next element of S^i. Continue this process until there are no remaining elements in S^i, and then terminate at that commodity's destination vertex. See Figure 5.2 for an

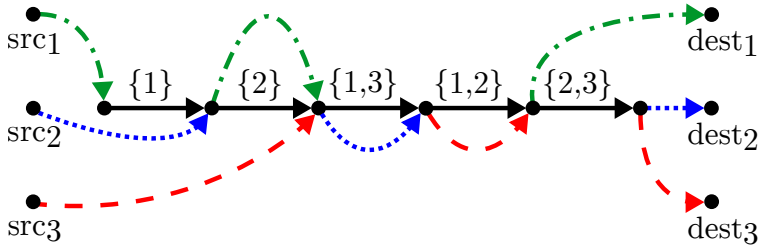


Figure 5.2: Reduction of SET-COVER instance to TP instance

example reduction where the SET-COVER instance has universe, $U = \{1, 2, 3\}$, and sets, $S = \{\{1\}, \{2\}, \{1, 3\}, \{1, 2\}, \{2, 3\}\}$.

A solution to the SET-COVER instance provides a solution to the TP instance of the same size since a subset of S that contains all elements of the universe will correspond to a set of edges that are cumulatively traversed by each commodity. Since each commodity is a single path, requiring a single turnstile to monitor its traffic, these edges provide a valid solution to the TP instance.

Given any solution to the TP instance, this solution is equivalent to one with turnstiles placed only on edges generated by elements of S, as opposed to the commodity specific edges. This solution corresponds to an equally sized solution to the SET-COVER instance, as a subset of S is selected that must cover all elements of the universe, since the turnstiles cover all commodities.

Since the turnstiles placed and the set cover elements are in one-to- one correspondence, the number of turnstiles placed for the TP instance is the same as the size of the set cover, thereby making the costs equal and the reduction approximation-preserving. It was shown in [81] that there exists a constant, $w > 0$, such that SET-COVER cannot be approximated within a factor of $w \ln n$, where n is the size of the universe U, unless $P = NP$. Therefore, TP cannot be approximated within a factor of $w \ln k$, where k is the number of commodities, unless $P = NP$.

□

Algorithms

This section provides three algorithms for the TP problem. The first is an approximation algorithm that greedily selects turnstile locations to minimize total solution cost. The second algorithm leverages the relationship between the SC-TP problem and the complement of maximum spanning trees detailed in Lemma 5.1 and Lemma 5.2. The third is a method that enables using existing single path solutions for a multipath TP instance. On page 93, performance of these solutions are demonstrated with a constant valued turnstile placement cost function, resulting in the objective of minimizing the number of turnstiles placed. However, the algorithms detailed in this section are able to handle any generalized turnstile placement cost function.

An Approximation Algorithm

This section introduces an approximation algorithm, TP-Approx, for the TP problem with a general turnstile placement cost function, which achieves an approximation ratio of $(\ln m + 1)(\ln k + 1)$, where k is the number of commodities and m is the number of edges in the network. (Theorem 5.1 below actually proves a slightly tighter bound.)

Each edge in a TP instance hosts some number of flows. A turnstile placed on some edge is said to *help* a commodity if that turnstile lies on at least one cycle in that commodity not already containing a turnstile. In this way, a proposed turnstile location only helps a commodity if that turnstile will reduce the number of additional turnstiles required for that commodity.

TP-Approx makes iterative turnstile placements, where the turnstile placed in each iteration is on the edge that minimizes the cost per commodity helped. This process continues until there are no cycles left in any of the commodity subgraphs. The turnstile cost function determines the cost of a candidate turnstile. The number of commodities helped by an edge (u, v) is calculated by checking, for each commodity, if any path exists from u to v besides the direct edge (u, v) in each commodity's subgraph. If a such a path exists, then (u, v) belongs to an uncovered cycle, and thus placing a turnstile on (u, v) helps that commodity. Once the best turnstile location edge e is determined, e is removed from all commodity graphs. This ensures that cycles discovered in subsequent iterations are not covered by any turnstile selected up to that point. An outline of TP-Approx is presented in Algorithm 5.1 and its running time is analyzed in Lemma 5.4.

Let $c(e)$ be the cost of placing a turnstile on edge e.
Let helps(e) be commodities helped by a turnstile on edge e.
On TP instance $\{(V, E, c), \{(s_k, t_k, E_k)\}\}$:
    while there exists $e \in E$ such that helps(e) $\neq \emptyset$
        Determine $e' = \arg\min_e \frac{c(e)}{|\text{helps}(e)|}$
        Place a turnstile on edge e'
        Remove e' from G
        Update helps(e) for remaining edges $e \in G$
    endwhile

Algorithm 5.1: TP-Approx

Lemma 5.4. *TP-Approx returns a set of turnstiles that resolves all traffic flows on each edge in the network and has a running time of $O(km^3)$, where k is the number of commodities and m is the number of edges in the network.*

Proof. Lemma 5.1 shows that a cycle transversal provides a valid turnstile solution for a single commodity. Since the algorithm runs until all intra-commodity cycles have at least one turnstile on them, all traffic for each commodity is monitored, and thus the turnstiles selected provide a valid solution for the entire network.

In each iteration of the algorithm, the number of commodities helped must be calculated for each edge. This amounts to, for each edge, searching for a cycle in each commodity's subgraph, which can be done with depth first search in a total of $O(km^2)$. The number of iterations needed is maximized when the algorithm places a turnstile on each edge. Therefore, the running time of Algorithm 5.1 is in $O(km^3)$. □

Theorem 5.1. *TP-Approx achieves a* $(\ln R_{\max} + 1)(\ln k + 1)$*-approximation ratio, where* k *is the number of commodities and* $R_{\max}$ *is the maximum number of turnstiles needed by any single commodity, considered in isolation as an SC-TP instance.* $R_{\max} = \max_i(m_i - n_i + 1)$ *where* n_i *is the number of vertices and* m_i *is the number of edges in the subgraph hosting commodity* i*.*

Proof. Let R_i^t be the number of remaining turnstiles needed for commodity K_i, just after TP-Approx places the t-th turnstile, T_t. Note that for each commodity K_i, $R_i^0 = m_i - n_i + 1$ since the spanning tree has exactly $n_i - 1$ edges and the complement of the spanning tree are the edges that break all cycles, as detailed in Lemma 5.2. Say that commodity K_i is *finished* when R_i^t reaches 0. Define the function *helps* on a turnstile T at iteration t as,

$$\text{helps}_t(T) = \{K_i : R_i^t = R_i^{t-1} - 1, \text{ if } T \text{ is chosen at } t\} \tag{5.1}$$

Let OPT $= \{T_1^*, \ldots, T_p^*\}$ be an optimal placement of turnstiles. Suppose the TP-Approx algorithm chooses turnstiles, ALG $= \{T_1, \ldots, T_a\}$, in this order. Consider $K_i \in \text{helps}(T_t)$. Just prior to TP-Approx selecting T_t, at least R_i^{t-1} turnstiles in OPT

that could help K_i have not yet been selected by ALG. If this were not the case, then by adding them to TP-Approx's solution, commodity K_i could be finished with fewer than R_i^{t-1} turnstiles, which contradicts Lemma 5.2. Note that the per-commodity cost of T_t is at most the per-commodity cost of the aforementioned OPT turnstiles (since TP-Approx makes a greedy choice). Pick any R_i^{t-1} of these OPT turnstiles and call the set of them $\text{OPT}(t,i)$. Observe that,

$$\frac{c(T_t)}{|\text{helps}_t(T_t)|} \leq \frac{1}{R_i^{t-1}} \sum_{T^* \in \text{OPT}_{(t,i)}} \frac{c(T^*)}{|\text{helps}_t(T^*)|}$$

We have,

$$\begin{aligned}
c(\text{ALG}) &= \sum_{t=1}^{a} c(T_t) \\
&= \sum_{t=1}^{a} \sum_{K_i \in \text{helps}_t(T_t)} \frac{c(T_t)}{|\text{helps}_t(T_t)|} \\
&\leq \sum_{t=1}^{a} \sum_{K_i \in \text{helps}_t(T_t)} \frac{1}{R_i^{t-1}} \sum_{T^* \in \text{OPT}_{(t,i)}} \frac{c(T^*)}{|\text{helps}_t(T^*)|} \\
&= \sum_{T^* \in \text{OPT}} c(T^*) \sum_{t=1}^{a} \sum_{\substack{K_i \in \text{helps}_t(T_t) \\ T^* \in \text{OPT}_{(t,i)}}} \frac{1}{R_i^{t-1} |\text{helps}_t(T^*)|}
\end{aligned} \tag{5.2}$$

After each iteration, t, some of the R_i^t decrease by 1 (those commodities helped by T_t). If R_i^t reaches 0 and $K_i \in \text{helps}_1(T^*)$, then $|\text{helps}_{t+1}(T^*)| < |\text{helps}_t(T^*)|$. Order the commodities in $\text{helps}_1(T^*)$ by increasing R_i^0 values; say the set is $K_{1^*}, \ldots K_{k_0^*}$. The sum over iterations, t, is maximized if TP-Approx helps the commodities in this order (so that the first commodity, K_{1^*}, is finished before any others are helped, etc.).

In this case, we have:

$$\begin{aligned}
&\sum_{t=1}^{a} \sum_{\substack{K_i \in \text{helps}_t(T_t) \\ T^* \in \text{OPT}(t,i)}} \frac{1}{R_i^{t-1} |\text{helps}_t(T^*)|} \\
&= \sum_{k=1}^{k_0} \frac{1}{k_0 - k + 1} \sum_{h=R_{k^*}^0}^{1} \frac{1}{h} \\
&\leq \sum_{k=1}^{k_0} \frac{1}{k_0 - k + 1} (\ln R_{k^*}^0 + 1) \\
&\leq \sum_{k=1}^{k_0} \frac{1}{k_0 - k + 1} (\ln R_{\max} + 1) \\
&= (\ln R_{\max} + 1) \sum_{k=1}^{k_0} \frac{1}{k_0 - k + 1} \\
&\leq (\ln R_{\max} + 1)(\ln k + 1), \qquad (5.3)
\end{aligned}$$

where $R_{\max} = \max_i R_i^0$. Substituting (5.3) into (5.2) yields,

$$\begin{aligned}
c(\text{ALG}) &\leq \sum_{T^* \in \text{OPT}} c(T^*)(\ln R_{\max} + 1)(\ln k + 1) \\
&= (\ln R_{\max} + 1)(\ln k + 1) c(\text{OPT}),
\end{aligned}$$

as was to be shown. □

Corollary 5.1. *Since $R_{\max} \leq m$, the TP-Approx algorithm trivially achieves an $(\ln m + 1)(\ln k + 1)$ approximation ratio.*

A Maximum Spanning Tree Approach

This section introduces another approach to solve the TP problem based on the relationship between the turnstile problem and the cycle transversal problem detailed on page 80. Lemma 5.2 showed that placing turnstiles on the complement of

a maximum spanning tree (MaxST) for a single commodity instance, SC-TP, provides an optimal turnstile placement for that instance.

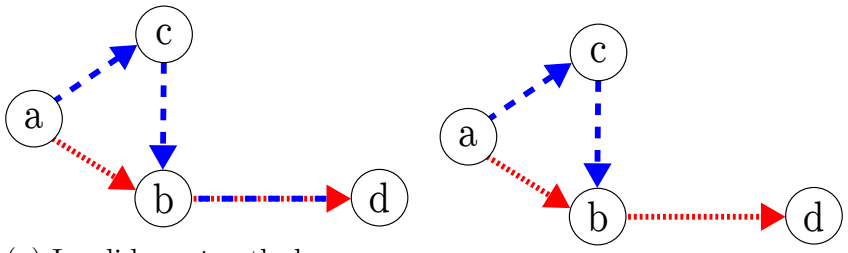
Given an instance of the TP problem, the algorithm TP-MaxST iterates through the set of commodities, finding a MaxST over the edge set for that commodity, and places turnstiles on the complement of the MaxST. After turnstiles are placed for a commodity, all the edges selected are removed from the graph. This will ensure that purchased turnstiles will break cycles in any future commodities sharing those edges, and will not be included in their MaxSTs. Since each commodity has turnstiles placed on the complement of a MaxST over its edge set, TP-MaxST will produce a valid solution for the full TP instance. TP-MaxST is detailed in Algorithm 5.2.

On TP instance $\{(V, E, c), \{(s_k, t_k, E_k)\}\}$:
    forall commodities (s_k, t_k, E_k) in TP instance
        Find a MaxST T_k that spans E_k
        Place turnstiles on $E_k \setminus T_k$
        Remove all edges $e \in E_k \setminus T_k$ from E
    endforall

Algorithm 5.2: TP-MaxST

A Multipath to Single Path Translation

This section details a process for adapting existing multicommodity, single path, flow monitoring approaches to multipath TP instances. One naïve approach would be, given an instance with a multipath commodity, decompose its flow into a collection of single path $s-t$ flows. Aggregate solutions to the single path flows could then be used as a solution to the multipath flow. This approach does not address ambiguities that can occur at intersections of those decomposed paths and is illustrated in Figure 5.3a.



(a) Invalid $s-t$ path decomposition

(b) Valid pathlet decomposition

Figure 5.3: Path and pathlet decomposition

In that example, the instance requires at least two turnstiles however, if the multipath flow were to be decomposed into $s-t$ paths, blue and red, a turnstile placed on edge (b, d) would cover the two single path flows, but not the original multipath flow with undirected cycle (a, b, c)

To successfully decompose a multipath TP instance, segment each multipath flow into a set of sub-paths (pathlets), such that a turnstile anywhere on each pathlet will provide a valid turnstile placement for the original multipath flow. These pathlets can then be treated as their own single path flows, and the same monitors placed by existing multicommodity single path monitoring approaches for the set of pathlets will suffice to monitor the original TP instance.

The translation from multipath TP instances into single path commodities involves decomposing each multipath flow into a collective set of pathlets. For each multipath flow from the TP instance, the decomposition begins by marking the source and sink vertices in the flow. Unmarked pathlets are then iteratively added to the set. An *unmarked* pathlet is a loopless path constructed between marked vertices such that only unmarked edges and vertices are traversed. One way to generate this is by using breadth first search from some marked vertex over unmarked edges until the first marked vertex is reached. The pathlet is added to the set of pathlets, and

each vertex and edge in the pathlet is marked. This process is continued until all edges in the flow are marked. Once a flow is decomposed into pathlets, all marks are disregarded and the next flow is decomposed. The set of pathlets resulting from the decomposition of all multipath flows in the TP instance is returned as a set of single path flows, ready for existing single path sensor placement algorithms. This translation is detailed in Algorithm 5.3 and shown in Figure 5.3b.

Let $\mathcal{P} = \varnothing$ be the set of pathlets
On TP instance $\{(V, E, c), \{(s_k, t_k, E_k)\}\}$:
    forall commodities (s_k, t_k, E_k) in TP instance
        Mark s_k and t_k
        while unmarked edges remain in E_k
            Find *unmarked* path between marked vertices
            Add path to $\mathcal{P}$ and mark all its vertices and edges
        endwhile
    endforall
    return set of pathlets, $\mathcal{P}$

Algorithm 5.3: Multipath Flows to Single Path Flows

Lemma 5.5. *A turnstile placed anywhere on each pathlet in the set of pathlets returned by Algorithm 5.3 will form a valid turnstile placement for the multipath TP input.*

Proof. By Lemma 5.1, a placement of turnstiles is valid if each cycle in the graph has a turnstile on at least one edge. I aim to show that each cycle in the graph must have one pathlet from $\mathcal{P}$ totally contained within that cycle. If that were the case, then any turnstile placement along that pathlet would sufficiently cover that cycle, thereby providing a valid placement as each cycle has such a pathlet.

Consider a cycle in the graph. Because each pathlet in $\mathcal{P}$ is loopless, this cycle must consist of multiple pathlets. Select some pathlet on the cycle that also contains edges off the cycle (if no such pathlet exists, then the pathlets are fully contained in the cycle and any turnstile placement will suffice). The vertices of that pathlet that lie on the cycle cannot be intermediate vertices of any other pathlet, since pathlet building ends as soon as a marked vertex is encountered. This means that those vertices must be endpoints for one or more pathlets. Thus, the remaining pathlets composing the rest of the cycle cannot all include edges off the cycle, or else some pathlet must include a vertex from another pathlet as an intermediate vertex. □

Experimental Results

This section is an evaluation of the algorithms presented in this chapter. This evaluation is part of a paper, currently in review, containing the results detailed in this chapter. The primary contributor to this evaluation, from implementing algorithms to running simulations to assembling figures, is Samuel Micka, but it is included in this dissertation to demonstrate the viability of the algorithms presented in this chapter. To compare the solutions from this chapter to existing single path solutions, we apply the multipath to single path translation described on page 90 and then employ a greedy Set-Cover motivated by the solution introduced in [29]. This hybrid algorithm is labeled the Greedy Single Path (GSP) algorithm. To minimize the total number of turnstiles placed, a constant valued turnstile placement cost function is employed.

To conduct our experiments, we use five network topologies from the Internet Topology Zoo [58]: GEANT, a European backbone network, UUNET, a United States backbone network, DFN, a German backbone network, Viatel, a European backbone

network, and Tinet, a global backbone network. We choose these topologies as representative examples of large real-world networks that support multipath flows.

Network traffic is generated on a per-commodity, per-instance, basis. We sample from a set of vertices in each topology to serve as sources and destinations. The Fast Network Simulation Setup (FNSS) is used to generate flow volumes [83]. Two methods are used for generating commodity edge sets. The first process uses a standard approach for determining multipath routing by having flows, volumes, and edge sets generated by solving an instance of the multicommodity flow problem with the linear program solver CPLEX. In the second method, edge sets are generated for flows by choosing edges in the k-shortest paths from source to destination where k was varied between one, three, and five. This method was used to reduce instance variability favoring one algorithm over another by explicitly controlling the number of paths accessible to each flow. Both methods result in a network with a defined topology and a set of commodities with multipath routes from their sources to destinations. The algorithms are tested on topologies containing four commodities and 100 commodities. The instances containing 100 commodities are the result of merging 25 instances of four commodities on the same topology. The volumes of these commodities make these instances representative of a topology with a 100 commodities that has the capacity to support four of them transmitting at full volume at any given time.

Number of Turnstiles Placed

The first simulation aims to determine the number of turnstiles placed by the algorithms on instances small enough for optimal turnstile placements to be computed. Figure 5.4 shows the results of these simulations. Four commodities are generated, per instance, using the technique described above on the five Internet Topology Zoo topologies using the CPLEX LP-Solver to generate the commodity

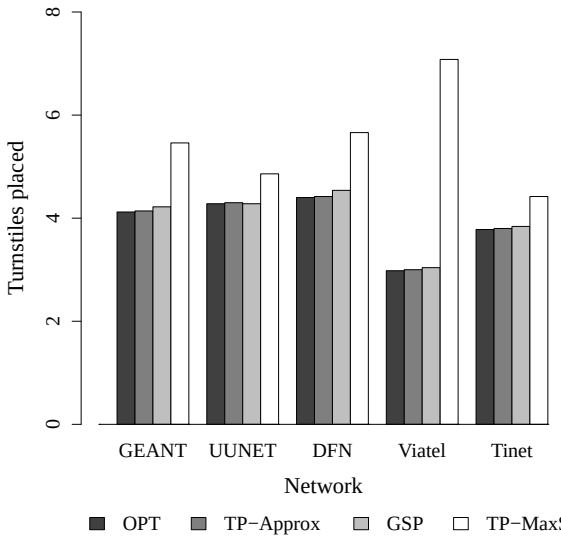


Figure 5.4: TP algorithm evaluation on small network instances

edge sets. The graph shows the average number of turnstiles needed to monitor each instance over 50 simulations, on each topology. OPT and TP-Approx average between three and four turnstiles for each topology with TP-Approx placing, on average, fewer than 1% more turnstiles than OPT. The number of turnstiles chosen by the optimal solution to be necessary for the topologies averages at 3.91. GSP and TP-MaxST place about 1% and 40% more turnstiles than OPT respectively. We suspect that the Viatel topology requires fewer turnstiles than the others because there are fewer distinct paths through the network than the others.

The second simulation aims to determine the number of turnstiles placed by the algorithms on the five Internet Topology Zoo topologies, with larger traffic patterns which contain too many flows to solve optimally. To generate an instance with 100 commodities, the instances are generated, four commodities at a time using the CPLEX LP-Solver to generate commodity edge sets, and merged into a single graph. Figure 5.5 shows the average number of turnstiles required by each algorithm over 50 iterations on each topology. TP-Approx places, on average across the five topologies,

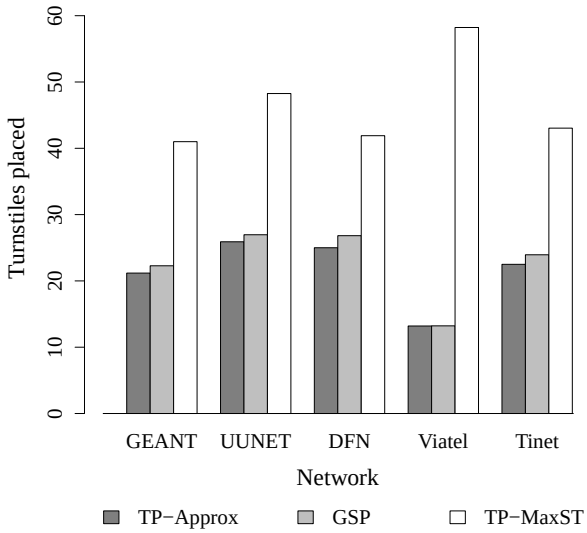


Figure 5.5: TP algorithm evaluation on large network instances

21.34 turnstiles in each simulation, whereas GSP places 22.6, and TP-MaxST places 45.04. As a result, TP-Approx places about 6% fewer turnstiles than GSP and about 53% fewer turnstiles than TP-MaxST. Increasing the number of commodities results in more of the underlying network being utilized, leading to a larger number of turnstiles being placed by all solutions. The relative number of turnstiles required between the various algorithms also increases due to TP-Approx and, to a lesser extent, GSP making better use of inter-commodity collaborative placements.

The third simulation aims to determine the number of turnstiles placed by the algorithms on the five Internet Topology Zoo topologies, with commodity edge sets determined using the k-shortest path algorithm on each source destination pairing. Like the previous simulation, 100 commodities are generated on each instance. To determine the commodity edge sets, we use k-shortest paths with k equal to one, three, and five, to explore the effectiveness of the different algorithms as the commodities contain more paths. Figure 5.6 shows the average number of turnstiles required by each algorithm over 50 iterations on the Tinet topology. TP-Approx places fewer

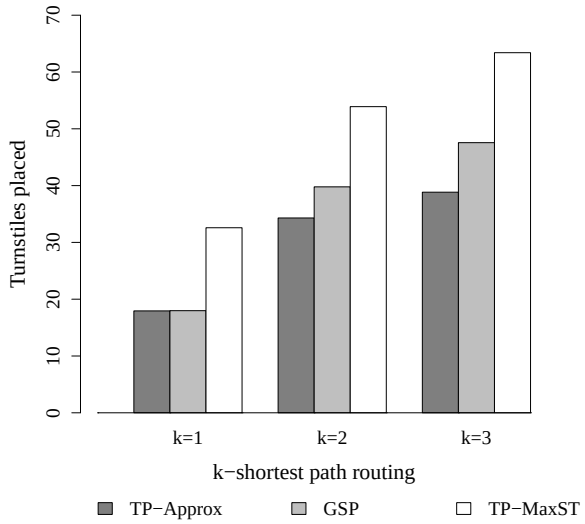


Figure 5.6: TP algorithm evaluation with variable number of paths

turnstiles in every instance, even when k is set to one. This result confirms that TP-Approx is a viable solution for the single path version of the problem, as well as the multipath formulation. Increasing the value of k highlights the reduced number of turnstiles placed by TP-Approx compared to the other solutions. TP-Approx places 34.3 turnstiles on average while GSP and TP-MaxST place 39.78 and 53.9 respectively when k is set to three. In this case, the GSP solution places about 16% more turnstiles than TP-Approx. TP-Approx places 38.84 turnstiles on average and GSP and TP-MaxST place 47.56 and 63.4 when k is set to five. This results in GSP placing about 22% more turnstiles than TP-Approx.

Placement Effectiveness

In certain scenarios (e.g. large topologies, restricted monitoring budget, loose monitoring requirements), placing fewer turnstiles than necessary makes sense, accepting that the entire network may not be covered. In this scenario, it is advantageous to employ an algorithm that provides the highest rate of return, in

terms of network coverage, per turnstile placed. The fourth simulation explores this scenario by investigating the rate at which the algorithms cover the network. Specifically, the Monitoring Benefit for placing a turnstile on an edge is the number of flows that have a cycle covered as a result of the placement.

Figure 5.7 shows the total number of turnstiles placed, in the order that they are placed, versus the number of commodity-required turnstiles covered. Note that the number of commodity-required turnstiles is just the sum of the number of turnstiles each individual commodity requires, $\sum_{i=0}^{k} m_i - n_i + 1$, where k is the number of commodities as detailed in the beginning of the proof to Theorem 5.1. Using the Tinet topology, with 100 commodities, we obtain the results shown in Figure 5.7. The commodity edge sets are generated using k-shortest paths with k set to three.

TP-Approx achieves 95% of the total number of commodity-required turnstiles after placing 26 turnstiles, whereas GSP requires 30 and TP-MaxST requires 36 to cover 95% of the commodity-required turnstiles. In this scenario, TP-Approx requires almost 14% fewer turnstiles than GSP and almost 28% fewer turnstiles than TP-MaxST for cover 95% of the commodity-required turnstiles. TP-Approx is not only able to place a fewer number of total turnstiles for full coverage, but is capable of placing turnstiles that benefit a large number of flows from the beginning. Figure 5.7 shows that this trend holds steady for any percent of the total number of commodity-required turnstiles. Interestingly, initial turnstiles placed by TP-MaxST and GSP are close to benefiting the same as TP-Approx, but that TP-MaxST diverges into less valuable placements at about 25% coverage of the commodity-required turnstiles.

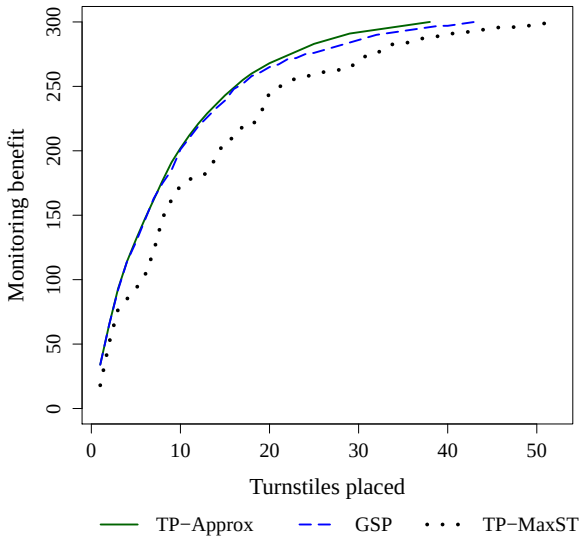


Figure 5.7: TP algorithm monitoring benefit evaluation

Conclusions

The turnstile placement techniques introduced in this chapter provide a new and efficient way to monitor multipath traffic flows, in settings such as software defined networks, to improve load balancing and network efficiency. Minimizing the number of turnstiles reduces the cost of monitoring; I have developed several efficient algorithms that minimize the number of turnstiles needed to monitor traffic flows. In particular, the TP-Approx algorithm achieves near optimal performance in simulations on real Internet topologies. There are several interesting open questions regarding flow monitoring with turnstiles: First, besides the cost of turnstile placement, there may be additional costs related to the volume of the flows monitored. Although flow volumes are initially unknown, once turnstiles are placed and edge-flow volumes calculated, the turnstile costs could be updated to reflect current flow volumes. Second, the network is dynamic and new flows will be added while old flows are removed. If the existing turnstiles are insufficient (or redundant), then a new turnstile solution should

be found, that accrues the least changeover cost. Finally, turnstile monitoring may be of use for flow monitoring in other types of networks, e.g. transportation networks, social networks, etc.

REFERENCES CITED

[1] ITU-T G.114. http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-G.114-200305-I!!PDF-E, May 2003.

[2] AWS Direct Connect. http://aws.amazon.com/directconnect/, Dec. 2014.

[3] Call of Duty: Ghosts - Multiplayer. http://www.ign.com/wikis/call-of-duty-ghosts/Multiplayer, Apr. 2014.

[4] Cloud Foundry. http://cloudfoundry.org/, Dec. 2014.

[5] Dell Cloud Manager. http://enstratus.com/, Dec. 2014.

[6] δ-cloud. http://deltacloud.apache.org/, Dec. 2014.

[7] Docker. http://www.docker.com/, Dec. 2014.

[8] jClouds. http://jclouds.apache.org/, Dec. 2014.

[9] Kaavo. http://kaavo.com/, Dec. 2014.

[10] Microsoft Azure ExpressRoute. http://azure.microsoft.com/en-us/services/expressroute/, Dec. 2014.

[11] RightScale. http://rightscale.com/, Dec. 2014.

[12] KEMP SDN Adaptive, powered by the HP VAN SDN Controller. https://kemptechnologies.com/sdn-adaptive-load-balancing/, Accessed Jul 2016.

[13] M. Abrash. Latency the sine qua non of AR and VR. http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/, Dec. 2012.

[14] M. Alicherry and T. V. Lakshman. Network aware resource allocation in distributed clouds. In *IEEE Infocom*, Mar. 2012.

[15] F. AlQayedi, K. Salah, and M. Zemerly. Network-aware resource allocation for cloud elastic applications. In *Electronics, Circuits, and Systems (ICECS)*, Dec. 2013.

[16] M. Alrokayan and R. Buyya. A Web portal for management of Aneka-based MultiCloud environments. In *Australasian Symposium on Parallel and Distributed Computing*, Jan. 2013.

[17] Amazon. Amazon EC2 pricing. http://aws.amazon.com/ec2/pricing/, July 2014.

[18] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *ACM SOSP*, Oct. 2001.

[19] A. Antoniadis and C.-C. Huang. Non-preemptive speed scaling. In *Scandinavian Workshop on Algorithms Theory*, pages 249–260. Springer, 2012.

[20] B. S. Baker and J. S. Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12(3):508–525, 1983.

[21] E. Bampis, G. Lucarelli, and I. Nemparis. Improved approximation algorithms for the non-preemptive speed-scaling problem. *arXiv preprint arXiv:1209.6481*, 2012.

[22] N. Beighton. The human cloud: Wearable technologys impact on society. `http://www.rackspace.com/blog/the-human-cloud-wearable-technologys-impact-on-society/`, June 2014.

[23] A. Bernstein. Near linear time $(1 + \epsilon)$-approximation for restricted shortest paths in undirected graphs. In *ACM-SIAM SODA*, Jan. 2012.

[24] R. J. Bessa, J. Mendes, V. Miranda, A. Botterud, J. Wang, and Z. Zhou. Quantile-copula density forecast for wind power uncertainty modeling. In *PowerTech, 2011 IEEE Trondheim*, pages 1–8, June 2011.

[25] R. J. Bessa, V. Miranda, A. Botterud, Z. Zhou, and J. Wang. Time-adaptive quantile-copula for wind power probabilistic forecasting. *Renewable Energy*, 40(1):29 – 39, 2012.

[26] G. R. Cantieni, G. Iannaccone, C. Barakat, C. Diot, and P. Thiran. Reformulating the monitor placement problem: Optimal network-wide sampling. In *ACM CoNEXT*, Dec 2006.

[27] C.-W. Chang, G. Huang, B. Lin, and C.-N. Chuah. LEISURE: Load-balanced network-wide traffic measurement and monitor placement. *Parallel and Distributed Systems*, 26(4):1059–1070, Apr 2015.

[28] T. H. Chang, M. Alizadeh, and A. Scaglione. Real-time power balancing via decentralized coordinated home energy scheduling. *IEEE Transactions on Smart Grid*, 4(3):1490–1504, Sept 2013.

[29] C. Chaudet, E. Fleury, I. G. Lassous, H. Rivano, and M.-E. Voge. Optimal positioning of active and passive monitoring devices. In *ACM conference on Emerging network experiment and technology*, Oct 2005.

[30] K.-T. Chen, P. Huang, and C.-L. Lei. Effect of network quality on player departure behavior in online games. *Parallel Distributed Systems*, 20:593–606, May 2009.

[31] S. Chen, M. Song, and S. Sahni. Two techniques for fast computation of constrained shortest paths. *IEEE/ACM ToN*, 16(1):105–115, Feb. 2008.

[32] Y. Chen, S. Jain, V. K. Adhikari, and Z.-L. Zhang. Characterizing roles of front-end servers in end-to-end performance of dynamic content distribution. In *ACM IMC*, Nov. 2011.

[33] S. Choi, S. Park, D. J. Kang, S. j. Han, and H. M. Kim. A microgrid energy management system for inducing optimal demand response. In *Smart Grid Communications (SmartGridComm), 2011 IEEE International Conference on*, pages 19–24, Oct 2011.

[34] J. Chuzhoy, S. Guha, S. Khanna, and J. Naor. Machine minimization for scheduling jobs with interval constraints. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 81–90, 2004.

[35] M. Cieliebak, T. Erlebach, F. Hennecke, B. Weber, and P. Widmayer. Scheduling with release times and deadlines on a minimum number of machines. In J.-J. Levy, E. Mayr, and J. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, volume 155 of *IFIP International Federation for Information Processing*, pages 209–222. Springer US, 2004.

[36] B. Claise. Cisco systems netflow services export version 9. 2004.

[37] M. Claypool and K. Claypool. Latency and player actions in online games. *CACM*, 49(11):40–45, Nov. 2006.

[38] A. J. Conejo, J. M. Morales, and L. Baringo. Real-time demand response model. *IEEE Transactions on Smart Grid*, 1(3):236–242, 2010.

[39] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *ASPLOS*, Mar. 2014.

[40] D. Dib, N. Parlavantzas, and C. Morin. Meryn: Open, SLA-driven, Cloud Bursting PaaS. In *ACM Workshop on Optimization Techniques for Resources Management in Clouds*, June 2013.

[41] C. Draxl, A. Clifton, B.-M. Hodge, and J. McCaa. The wind integration national dataset (wind) toolkit. *Applied Energy*, 151:355 – 366, 2015.

[42] R. Fabregat, Y. Donoso, F. Solano, and J. Marzo. Multitree routing for multicast flows: A genetic algorithm approach. In *Catalan Conference on Artificial Intelligence*, Oct. 2004.

[43] M. Fathi and H. Bevrani. Adaptive energy consumption scheduling for connected microgrids under demand uncertainty. *IEEE Transactions on Power Delivery*, 28(3):1576–1583, July 2013.

[44] FCC. Measuring Broadband America - 2014. `http://www.fcc.gov/reports/measuring-broadband-america-2014`, June 2014.

[45] K. Fox, S. Im, and B. Moseley. Energy efficient scheduling of parallelizable jobs. In *Symposium on Discrete Algorithms*, pages 948–957, 2013.

[46] D. Franceschelli, D. Ardagna, M. Ciavotta, and E. Di Nitto. Space4cloud: A tool for system performance and costevaluation of cloud systems. In *Workshop on Multi-cloud Applications and Federated Clouds*, Apr. 2013.

[47] K. Ghoseiri and B. Nadjari. An ant colony optimization algorithm for the bi-objective shortest path problem. *Applied Soft Computing*, 10(4):1237–1246, 2010.

[48] I. Grigorik. Latency: The new Web performance bottleneck. `http://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/`, July 2012.

[49] D. Gupta, P. Mohapatra, and C.-N. Chuah. Efficient monitoring in wireless mesh networks: Overheads and accuracy trade-offs. In *IEEE Mobile Ad Hoc and Sensor Systems (MASS)*, Sep 2008.

[50] O. Hansen. The biggest problem in Augmented Reality: Latency. `https://identifeye.wordpress.com/2013/01/03/the-biggest-problem-in-augmented-reality-latency/`, Jan. 2013.

[51] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI*, Apr. 2011.

[52] E. Howard, C. Cooper, M. P. Wittie, S. Swinford, and Q. Yang. Cascading impact of lag on user experience in multiplayer games. In *ACM NetGames*, Dec. 2014.

[53] G. Huang, C.-W. Chang, C.-N. Chuah, and B. Lin. Measurement-aware monitor placement and routing: a joint optimization approach for network-wide measurements. *IEEE Transactions on Network and Service Management*, 9(1):48–59, Mar 2012.

[54] Q. Huang, X. Li, J. Zhao, D. Wu, and X. Y. Li. Social networking reduces peak power consumption in smart grid. *IEEE Transactions on Smart Grid*, 6(3):1403–1413, May 2015.

[55] G. Iannaccone, S. Bhattacharrya, N. Taft, and C. Diot. Always-on monitoring of IP backbones: requirements and design challenges. Technical report, Sprint ATL, July 2013.

[56] A. W. Jackson, W. Milliken, C. A. Santiváñez, M. Condell, and W. T. Strayer. A topological analysis of monitor placement. In *IEEE Network Computing and Applications*, Jul 2007.

[57] K. Keahey, P. Armstrong, J. Bresnahan, D. LaBissoniere, and P. Riteau. Infrastructure outsourcing in multi-cloud environment. In *Workshop on Multi-cloud Applications and Federated Clouds*, Sept. 2012.

[58] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, Oct 2011.

[59] J. Z. Kolter and M. J. Johnson. Redd: A public data set for energy disaggregation research. In *SustKDD workshop on Data Mining Applications in Sustainability*, 2011.

[60] V. Kompella, J. Pasquale, and G. Polyzos. Multicast routing for multimedia communication. *IEEE/ACM ToN*, 1(3):286–292, Jun 1993.

[61] I. Koutsopoulos and L. Tassiulas. Control and optimization meet the smart power grid: scheduling of power demands for optimal energy management. In *International Conference on Energy-Efficient Computing and Networking*, pages 41–50. ACM, 2011.

[62] I. Koutsopoulos and L. Tassiulas. Optimal control policies for power demand scheduling in the smart grid. *IEEE Journal on Selected Areas in Communications*, 30(6):1049–1060, 2012.

[63] V. Lindberg. ZeroVM: smaller, lighter, faster. `http://www.rackspace.com/blog/zerovm-smaller-lighter-faster/`, Oct. 2013.

[64] R. G. Little. F5 SDN: Why ADCs are relevant in a programmable world. `http://searchsdn.techtarget.com/news/2240226557/F5-SDN-Why-ADCs-are-relevant-in-a-programmable-world`, Aug. 2014.

[65] T. Logenthiran, D. Srinivasan, and T. Z. Shun. Demand side management in smart grid using heuristic optimization. *IEEE Transactions on Smart Grid*, 3(3):1244–1252, Sept 2012.

[66] M. Mazidi, A. Zakariazadeh, S. Jadid, and P. Siano. Integrated scheduling of renewable generation and demand response programs in a microgrid. *Energy Conversion and Management*, 86:1118 – 1127, 2014.

[67] C. Metz. Google open sources its secret weapon in cloud computing. `http://www.wired.com/2014/06/google-kubernetes/`, June 2014.

[68] Microsoft Azure. Data transfers pricing details. `http://azure.microsoft.com/en-us/pricing/details/data-transfers/`, July 2014.

[69] Microsoft Azure. Virtual machines pricing details. `http://azure.microsoft.com/en-us/pricing/details/virtual-machines/`, July 2014.

[70] A.-H. Mohsenian-Rad and A. Leon-Garcia. Optimal residential load control with price prediction in real-time electricity pricing environments. *IEEE Transactions on Smart Grid*, 1(2):120–133, 2010.

[71] F. G. Ortmann, N. Ntene, and J. H. van Vuuren. New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems. *European Journal of Operational Research*, 203(2):306–315, 2010.

[72] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *ACM SOSP*, Nov. 2013.

[73] F. Paraiso, P. Merle, and L. Seinturier. Managing elasticity across multiple cloud providers. In *Workshop on Multi-cloud Applications and Federated Clouds*, Apr. 2013.

[74] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internets. In *SIGCOMM CCR*, volume 31, pages 15–26, Aug 2001.

[75] P. Pinson et al. Wind energy: Forecasting challenges for its operational management. *Statistical Science*, 28(4):564–585, 2013.

[76] Y. Qin, D. G. Feng, K. Chen, and Y. F. Lian. Research on monitor position in network situation assessment. In *IEEE Industrial Control and Electronics Engineering (ICICEE)*, Aug 2012.

[77] Rackspace. Managed VMs and bare-metal servers in the cloud. `http://www.rackspace.com/cloud/servers/`, July 2014.

[78] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 10–18. IEEE, 1986.

[79] P. Raghavan and C. D. Tompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[80] C. Raiciu, M. Ionescu, and D. Niculescu. Opening up black box networks with CloudTalk. In *USENIX Hot Topics in Cloud Computing (HotCloud)*, June 2012.

[81] R. Raz and S. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *ACM Symposium on Theory of Computing (STOC)*, May 1997.

[82] H. T. Roh and J. W. Lee. Residential demand response scheduling with multiclass appliances in the smart grid. *IEEE Transactions on Smart Grid*, 7(1):94–104, Jan 2016.

[83] L. Saino, C. Cocora, and G. Pavlou. A toolchain for simplifying network simulation setup. In *International Conference on Simulation Tools and Techniques (ICST), SIMUTOOLS*, Mar 2013.

[84] S. Salinas, M. Li, P. Li, and Y. Fu. Dynamic energy management for the smart grid with distributed energy resources. *IEEE Transactions on Smart Grid*, 4(4):2139–2151, Dec 2013.

[85] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: informed internet routing and transport. *Micro, IEEE*, 19(1):50–59, Jan 1999.

[86] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *ACM EuroSys*, Apr. 2013.

[87] G. Sideratos and N. D. Hatziargyriou. Probabilistic wind power forecasting using radial basis function neural networks. *IEEE Transactions on Power Systems*, 27(4):1788–1796, Nov 2012.

[88] Skype. How much bandwidth does Skype need? `https://support.skype.com/en/faq/FA1417/how-much-bandwidth-does-skype-need`, June 2014.

[89] K. Suh, Y. Guo, J. Kurose, and D. Towsley. Locating network monitors: Complexity, heuristics, and coverage. *Computer Communications*, 29(10):1564–1577, Jun 2006.

[90] S. Sun, M. Dong, and B. Liang. Distributed real-time power balancing in renewable-integrated power grids with storage and flexible loads. *IEEE Transactions on Smart Grid*, 7(5):2337–2349, Sept 2016.

[91] S. Tang, Q. Huang, X.-Y. Li, and D. Wu. Smoothing the energy consumption: Peak demand reduction in smart grid. In *INFOCOM, 2013 Proceedings IEEE*, pages 1133–1141, 2013.

[92] P. Tarng, K. Chen, and P. Huang. On prophesying online gamer departure. In *ACM NetGames*, Nov. 2009.

[93] I. Vazquez-Abrams. Stackexchange superuser. `http://superuser.com/questions/289978`, May 2011.

[94] P. Vytelingum, S. D. Ramchurn, T. D. Voice, A. Rogers, and N. R. Jennings. Trading agents for the smart electricity grid. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 897–904, 2010.

[95] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with Conductor. In *USENIX NSDI*, April 2012.

[96] M. P. Wittie, V. Pejovic, L. Deek, K. C. Almeroth, and B. Y. Zhao. Exploiting locality of interest in online social networks. In *ACM CoNEXT*, Nov. 2010.

[97] G. Xia and Y. Zhang. On the small cycle transversal of planar graphs. *Theoretical Computer Science*, 412(29):3501 – 3509, Jul 2011.

[98] G. Xia and Y. Zhang. Kernelization for cycle transversal problems. *Discrete Applied Mathematics*, 160(7):1224–1231, May 2012.

[99] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[100] S. Yaw, E. Howard, B. Mumey, and M. P. Wittie. Cooperative group provisioning with latency guarantees in multi-cloud deployments. *SIGCOMM Comput. Commun. Rev.*, 45(3):4–11, July 2015.

[101] S. Yaw and B. Mumey. An exact algorithm for non-preemptive peak demand job scheduling. In *Combinatorial Optimization and Applications*, volume 8881 of *Lecture Notes in Computer Science*, pages 3–12. Springer International Publishing, 2014.

[102] S. Yaw, B. Mumey, E. McDonald, and J. Lemke. Peak demand scheduling in the smart grid. In *Smart Grid Communications (SmartGridComm), 2014 IEEE International Conference on*, pages 770–775, 2014.

[103] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11), July 1971.

[104] H. Zang and A. Nucci. Traffic monitor deployment in IP networks. *Computer Networks*, 53(14):2491–2501, Sep 2009.

[105] Y. Zeng, D. Wang, W. Liu, and A. Xiong. An approximation algorithm for weak vertex cover problem in IP network traffic measurement. In *IEEE Network Infrastructure and Digital Content (IC-NIDC)*, Nov 2009.

[106] Y. Zhang. An adaptive flow counting method for anomaly detection in SDN. In *ACM CoNEXT*, Dec 2013.

[107] Y. Zhang, N. Gatsis, and G. B. Giannakis. Robust energy management for microgrids with high-penetration renewables. *IEEE Transactions on Sustainable Energy*, 4(4):944–953, Oct 2013.

[108] Z. Zhou, A. Botterud, J. Wang, R. Bessa, H. Keko, J. Sumaili, and V. Miranda. Application of probabilistic wind power forecasting in electricity markets. *Wind Energy*, 16(3):321–338, 2013.