The cost of being restricted by time, knowledge or fairness in scheduling and allocation problems

or: How to run an Asian restaurant as a mathematician

DISSERTATION
zur Erlangung des Grades
eines Doktors der Naturwissenschaften
der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen

vorgelegt von
Felix Höhne, M. Sc. (Mathematik)

Erstgutachter/in: Prof. Rob van Stee
Zweitgutachter/in: Prof. Lene Monrad Favrholdt
Datum der Disputation: 23.11.2022

Abstract

In this thesis, we consider a variety of problems where solutions must be found while adhering to some kind of restriction. We measure the performance of our algorithms by comparing them to optimal solutions.

Online algorithms face an optimization problem and must find a solution with incomplete knowledge of the input. Their performance is measured using the *competitive ratio*. For a minimization problem this is the maximum (or supremum) of the value achieved by the algorithm divided by the optimal value over all instances.

Offline approximation algorithms must find a solution in polynomial time and are compared to an optimal algorithm using the *approximation ratio* that is defined analogously to the competitive ratio.

Finally, when allocating goods or chores we can define the *price of fairness* which compares the welfare of an allocation that adheres to some notion of fairness to the welfare of an optimal allocation.

In the buffer minimization problem with conflicts we are given a graph where the nodes represent processors and the edges represent conflicts. At any time tasks may arrive on a processor adding to this processor's workload. Each processor stores its workload in a separate input buffer. The goal is to find a scheduling strategy that minimizes the maximum used buffer size of all processors. We introduce a new model called the flow model, where load may not arrive in blocks but instead arrives at a fixed rate.

We discuss a variety of results, including tight or almost tight competitive ratios for both the original and the flow model on the path with up to five machines. We also slightly improve the lower bound on the path with m machines and discuss algorithms with resource augmentation that achieve good results on general graphs.

In the discrete bamboo garden trimming problem, we are given a set of n bamboo that grow at rates $v_1, \dots, v_n$ per day. Initially, the height of all plants is zero. Each day a robotic gardener cuts down one bamboo to height zero. The goal is to design a trimming schedule such that the height of the tallest bamboo that is ever achieved is minimized. For this problem we improve the approximation ratio to $\frac{7}{5}$ using a reduction to the pinwheel-scheduling problem. We also consider the continuous version of the problem, where the gardener travels between plants in a metric space, and find algorithms with constant approximation ratios for the case where this metric is a star graph.

Previous work has discussed the fair allocation of indivisible and divisible goods as well as divisible chores and we fill a gap in the literature by considering the problem of allocating contiguous blocks of indivisible chores fairly. We show the existence of certain types of fair allocations and find the prices of fairness for the commonly used notions of fairness.

Zusammenfassung

In dieser Dissertation untersuchen wir Probleme, deren Lösungen unter gewissen Einschränkungen gefunden werden müssen. Die Qualität unserer Algorithmen messen wir durch den Vergleich mit der optimalen Lösung.

Online-Algorithmen sind mit einem Optimierungsproblem konfrontiert und müssen Entscheidungen treffen, ohne vollständige Kenntnis der Eingabe zu haben. Die Informationen werden dem Algorithmus erst nach und nach übermittelt. Wir messen die Qualität eines Online-Algorithmus mithilfe des *Kompetitivitätsverhältnisses*. Dies ist das Maximum (oder Supremum) des Verhältnisses zwischen dem Wert des Algorithmus und dem optimalen Wert über allen Eingaben.

Offline-Approximations-Algorithmen müssen eine Lösung in polynomieller Zeit finden und werden ebenfalls mit einem optimalen Algorithmus verglichen. Dieser Vergleich wird *Approximationsverhältnis* genannt und ist analog zum Kompetitivitätsverhältnis definiert.

Wenn wir Güter oder Aufgaben verteilen, dann definieren wir den *Preis der Fairness*. Dieser vergleicht das von einer fairen Verteilung erzielte Allgemeinwohl mit dem von einer optimalen Verteilung erreichten.

Im Buffer-Minimierungsproblem mit Konflikten ist ein Graph gegeben, in dem die Knoten Prozessoren und die Kanten Konflikte darstellen. Jederzeit kann Last auf den Knoten ankommen, die verarbeitet werden muss. Die Prozessoren speichern diese Last in einem Buffer. Ziel ist es, die Last zu verarbeiten und dabei die maximal notwendige Buffergröße klein zu halten.

Wir führen hier ein neues Modell, Flow Model genannt, ein und bestimmen das Kompetitivitätsverhältnis auf einem Pfad mit bis zu 5 Maschinen. Zusätzlich betrachten wir Algorithmen mit Ressourcenaugmentation auf allgemeinen Graphen.

Im diskreten *bamboo garden trimming problem* wachsen n Bambusse mit Geschwindigkeit $v_1, \dots, v_n$ pro Tag. Jeden Tag wird ein Bambus auf Höhe null geschnitten. Ziel ist es, die Höhe des größten Bambus gering zu halten. Wir verbessern das Approximationsverhältnis auf den Wert 7/5, indem wir das Problem auf das *pinwheel-scheduling problem* zurückführen. Darüber hinaus betrachten wir die kontinuierliche Version, in der sich der Gärtner zwischen den Pflanzen bewegt, und finden konstante Approximationsalgorithmen für den Fall, dass die Pflanzen auf einem Sterngraph angeordnet sind.

In bisherigen Arbeiten wurde die faire Verteilung von teilbaren und unteilbaren Gütern sowie von teilbaren Aufgaben behandelt. Wir widmen uns der Verteilung von unteilbaren Aufgaben und zeigen, wann bestimmte faire Verteilungen existieren und bestimmen die Preise der Fairness für dieses Problem.

Acknowledgements

First of all, I would like to thank my supervisor Rob van Stee for his helpful advice and great support, as well as the other members of our research group. I have enjoyed my time at the University of Siegen very much.

I also want to thank my parents for their encouragement, and the same goes for my friends, be they in my Gloomhaven group, in Minnesota or on a frisbee field.

Contents

Chapter 1

Introduction

Let us assume we are running a restaurant. It is a quiet and cozy place at the edge of town set in a beautiful bamboo garden. Inside we might imagine a set of pinewood tables, separated by woven bamboo panels, some soft music and an aquarium in the background. Today's menu consists of a few Asian specialties, we particularly recommend the mango duck.

Now that we have set the scene, let us explore some of the mathematical problems that naturally arise in such a setting.

A group of guests arrives, perhaps a conference of mathematicians, and we want to serve them food. After going around the table and taking orders we return to the kitchen and are now faced with the problem of distributing the orders among the chefs in our kitchen. Some dishes are easy to prepare while others take more time, and we want to avoid a guest having to wait a long time for their food.

This is a classic problem in optimization. We need to find a schedule that minimizes our objective function which in this case is the time when the last meal finishes being prepared. This problem is also called the job scheduling problem and in more formal language we describe it as follows.

Given m chefs and n orders with different processing times, allocate the orders among the chefs in such a way that the makespan, i.e. the time when the last chef finishes processing their workload, is minimized.

This problem like many other problems considered in optimization, is NP-hard. This means (assuming $P \neq NP$) that it takes a very long time to calculate the optimal solution. For this reason there has been great interest in approximation algorithms that provide an approximate solution with short computation time, i.e. with time polynomial in the size of the input.

We measure the quality of an approximation algorithm using the approximation ratio. For a minimization problem this is the maximum (or supremum) of the value achieved by the algorithm divided by the optimal value over all instances. For example, this means that a 2-approximation algorithm is always within a factor of 2 of the optimal value.

A simple approximation algorithm for the job scheduling problem goes through the jobs one by one and assigns each job to the chef that currently has the lowest workload. It turns out that this algorithm is a $(2 - \frac{1}{m})$-approximation. Thus the approximation ratio grows arbitrarily close to 2 as the number of chefs increases.

One might ask the question how the problem changes if the input, in this case the set of jobs, is not known to the algorithm in advance, and instead the jobs are revealed one by one and each job must be assigned immediately after it is revealed. Here the algorithm needs to make decisions without knowledge of future events that might affect the outcome.

We call this class of algorithms, that have to deal with incomplete information about the future, online algorithms. We also say that approximation algorithms that do have complete information about the input are offline.

When talking about online algorithms we are only interested in the effect of incomplete information. This means there is no requirement that a solution must be computed in polynomial time.

We measure the quality of an online algorithm using the competitive ratio. This is again the maximum (or supremum) of the value achieved by the online algorithm compared to the optimal value over all instances. The definition is identical to that of the approximation ratio. The difference is that online algorithms are differently restricted.

Fortunately, for the case of job scheduling the algorithm presented above also works for the online variant of the problem and the approximation ratio and competitive ratio are the same.

We can partition the world of optimization problems into online and offline problems where in some cases, such as the scheduling problem, both the online as well as the offline version are meaningful while other problems are intrinsically online or offline. For example it might be natural to consider most instances of linear programming to be offline while problems like paging or the secretary problem are fully online.

Now that we have figured out the order in which we need to prepare the food we instruct our waitress to set the table by placing a pair of chopsticks with each guest. Unfortunately we are short on chopsticks, however our waitress, also a mathematician, still manages to follow our instruction precisely by placing one chopstick to the left and right of each guest, one between each pair of guests.

This of course means that, whenever a guest picks up two chop sticks and eats, his neighbors cannot since at least one of their chop sticks is in use. Sanitary reservations against the sharing of chopsticks aside, our guests are now confronted with their very own optimization problem.

As the waiters relentlessly serve plate after plate they must figure out a schedule that allows them to eat their meals without any guest building up too large a backlog of uneaten food.

This optimization problem is called the *buffer minimization problem with conflicts*. In more formal terms it is described as follows. We are given a graph where the nodes represent machines and the edges represent conflicts. Each machine is equipped with a buffer, and at any time workload may arrive in that buffer that must be processed. However, two adjacent machines on the graph are not allowed to work at the same time. The goal is to minimize the maximum buffer size that is needed on any machine to store the unprocessed workload.

In this work we are mainly interested in the case where the graph is a line and present tight bounds on the competitive ratio in small instances of the problem. We also give a variety of results on larger instances and more general graphs.

After apologizing for the inconvenience and presenting the guests with our findings, which hopefully allow them to fully enjoy their dinner, we may take a break in the bamboo garden outside the restaurant. This garden is home to a robotic panda gardener, and programming this gardener is another optimization problem.

In the *bamboo garden trimming problem* we are given a set of bamboo that grow at a certain rate each day. Each day our gardener may choose one of the bamboo and cut it down to the ground. Our goal is to find a schedule that keeps the maximum height that any bamboo reaches as low as possible.

We improve the approximation ratio for this problem. We also consider the continuous variation of the problem where the gardener travels between the bamboo and cuts down a plant whenever it reaches it. We find algorithms with constant approximation ratios for the case where the bamboo are planted in a star shape, meaning the metric space the gardener travels in is a star graph.

Finally the day comes to an end and after all the guests have left, we have to clean up the place for the next day. This means that a variety of chores need to be assigned among our workers. This time we are less concerned with the time these tasks take or optimizing the results. Instead we care about the happiness of our workers. The workers may have preferences for different tasks and for example might prefer washing dishes to cleaning the floor or vice versa. Some are also more motivated than others.

Our first approach might be to simply maximize overall welfare, which means minimizing the total disutility caused by the chores. However, this might mean allocating all chores to our most motivated workers while others do nothing and although this allocation is optimal, it might be criticized as extremely unfair.

To solve this dilemma we consider different commonly used notions of fairness such as envy-freeness, equitability and proportionality. The trade-off between optimizing welfare and fairness is called the price of fairness. For a given problem and in particular when assigning chores, the price of fairness is calculated as the disutility of an optimal fair allocation divided by the disutility of an optimal allocation. If we were allocating goods instead, we would consider welfare and use the inverse of this ratio.

Many variations of this problem have already been considered in previous works and we fill a gap in the literature by considering the specific case where we assume the chores are positioned in some order on a line and we are to assign *contiguous blocks of indivisible chores*. In our setting the contiguity requirement might be explained by locality. For example, we might not want to assign cleaning in the kitchen to the same person that goes on a shopping trip.

We prove the existence of fair allocations for this problem and calculate the price of fairness for the different notions of fairness.

After considering all of these topics which are discussed in the following chapters we should be well equipped to handle all challenges thrown at us by the food service industry. And if that is not the case, at least we hopefully have learned something interesting about optimization algorithms.

1.1 Preliminaries

An optimization problem $\mathcal{P}$ consists of a set of inputs $\mathcal{I}$ and a cost function C. Associated with an input $I \in \mathcal{I}$ is a set of feasible solutions $F(I)$ and each solution $J \in F(I)$ is assigned a positive real number $C(I, J)$ called its objective function value.

There are maximization and minimization problems, however in this work we mostly consider minimization problems. Then the objective function value is the *cost* associated with the solution.

Given any input I, an algorithm $\textsc{Alg}$ produces a solution $\textsc{Alg}[I] \in F(I)$ and the cost of this solution is denoted as $\textsc{Alg}(I) = C(I, \textsc{Alg}[I])$. An optimal algorithm $\textsc{Opt}$ finds the optimal solution for any input, i.e. for all feasible inputs I we have

$$\textsc{Opt}(I) = \min_{J \in F(I)} C(I, J).$$

An algorithm is called a c-approximation algorithm for a minimization problem $\mathcal{P}$ if for all feasible inputs I we have

$$\textsc{Alg}(I) \leq c \cdot \textsc{Opt}.$$

Analogously, for a maximization problem we require $\textsc{Alg}(I) \geq c \cdot \textsc{Opt}$. In both cases the approximation factor is greater than or equal to 1 with lower values corresponding to a better approximation.

We differentiate between online and offline algorithms. Offline algorithms are given full information about the input before determining a solution.

When considering offline algorithms, we are primarily concerned with problems where calculating the optimal solution is NP-hard. We require the offline approximation algorithm to calculate its solution in time polynomial to the size of the input.

Online algorithms, on the other hand, do not have full access to the input in the beginning. Instead, the input is revealed to the algorithm over time (or in a list). This means the algorithm has to make decisions with incomplete information.

An online algorithm is called c-competitive for a minimization problem $\mathcal{P}$ if for all inputs I we (again) have

$$\textsc{Alg}(I) \leq c \cdot \textsc{Opt}$$

For online algorithms we do not impose any requirements on computation efficiency. This definition is the same as for approximation algorithms but the algorithms are differently restrained.

We now formally define the three problems considered in this work.

Buffer management with conflicts In the buffer minimization problem with conflicts we are given a graph $G = (V, E)$ where the nodes V represent processors and the edges E represent conflicts. At certain times tasks arrive that need to be processed, and each processor stores this workload in an input buffer. By numbering the processors from 1 to m, the state of the system can then be described by a load vector $a = (a_1, \dots, a_m)$. Load vectors are never negative.

Time runs continuously and tasks may arrive at any time. The speed of a machine describes how fast the machine can reduce its workload. A machine that runs at speed s for a duration of length T will reduce its workload by sT if no load arrives during that time period. At any time the algorithm may select an independent set of vertices in the graph and run the machines on these vertices at speed 1. In other words, two adjacent machines may not run at the same time.

We also consider a variation where machines may run at fractional speeds as long as two adjacent machines run at a combined speed of at most 1. This means two adjacent machines may share processing power and both work at speed $\frac{1}{2}$ instead of just one machine working at speed 1 at any given time.

For bipartite graphs, and in particular the line graph, these two models are equivalent. We can achieve fractional speeds in the original model by rapidly switching between different independent sets.

We also introduce a variation of task arrivals called the flow model. In the flow model, load arrives on machines at a certain *rate*. When load arrives on a machine at a rate of r for a period of length T and that machine is not running, its workload increases by rT. Load arrives at a rate of at most 1 in the flow model.

Bamboo garden trimming In the discrete bamboo garden trimming problem (BGT), first introduced by Gasieniec et al. [51], we are given a set of n bamboo that grow at rates $v_1, \dots, v_n$ per day. We assume that these growth rates are arranged such that $v_1 \geq v_2 \geq \cdots \geq v_n$. Initially, the height of all plants is zero. Each day a robotic gardener cuts down one bamboo to height zero. The goal is to design a trimming schedule such that the height of the tallest bamboo that is ever achieved is minimized.

In the continuous version of the problem the bamboo are distributed in some metric space and the gardener needs to travel between the bamboo to cut them. Cutting is done instantly and the goal is to find a route that minimizes the maximum height of the bamboos.

Fair allocation of contiguous blocks of indivisible chores Let $N = \{1, .., n\}$ be a set of agents and $M = \{1, ..., m\}$ a set of chores. We assume the chores lie on a line in this order. Each agent i has nonnegative disutility $d_i(j)$ for chore j.

We assume that the disutilities are *additive*, which means that $d_i(M') = \sum_{j \in M'} d_i(j)$ for any agent i and a subset $M' \subseteq M$.

An *allocation* $M = (M_1, ..., M_n)$ is a partition of all chores into n bundles so that agent i receives bundle M_i. The allocation is *contiguous* if each M_i forms a contiguous block of items on the line.

The *utilitarian social cost* of an allocation is defined as $\sum_{i \in N} d_i(M_i)$ and the *egalitarian social cost* is $\max_{i \in N} d_i(M_i)$.

We now define the notions of fairness that we consider in this thesis. An allocation $M = (M_1, ..., M_n)$ is

- *proportional* if $d_i(M_i) \leq \frac{1}{n} d_i(M)$ for all $i \in N$.
- *equitable* if $d_i(M_i) = d_j(M_j)$ for all $i, j \in N$.
- *envy-free* if $d_i(M_i) \leq d_i(M_j)$ for all $i, j \in N$.

The price of fairness (i.e., the price of proportionality, respectively envy-freeness, equitability) is the minimal social cost achievable in fair (proportional, respectively envy-free, equitable) allocations divided by the minimal social cost achievable in arbitrary allocations. If both are 0, we define the price of fairness to be equal to 1. The price of fairness is only defined if a fair allocation exists, therefore we only consider such instances.

1.2 Structure and publications

The following publications are the foundation of this thesis:

[58] Felix Höhne, Sören Schmitt, and Rob van Stee. SIGACT news online algorithms column 35: 2019 in review, *SIGACT News 50*, 2019

[59] Felix Höhne, Sören Schmitt, and Rob van Stee. SIGACT news online algorithms column 35: 2020 in review, *SIGACT News 51*, 2020

[60] Felix Höhne, Sören Schmitt, and Rob van Stee. SIGACT news online algorithms column 35: 2021 in review, *SIGACT News 52*, 2021

[61] Felix Höhne and Rob van Stee. Buffer minimization with conflicts on a line, *FAW*, 2020

[62] Felix Höhne and Rob van Stee. Allocating contiguous blocks of indivisible chores fairly, *Information and Computation*, 2021

[63] Felix Höhne and Rob van Stee. Buffer minimization with conflicts on a line, *Theoretical Computer Science*, 2021

In chapter 2 we begin with a survey of contemporary results in the field of online algorithms. Some of these results are closely related to the topics discussed in the later chapters, but we also include some themes that have been of great interest but are not immediately related to the following chapters. The contents of this chapter is based upon the SIGACT news column from the years 2019 to 2021 [58, 59, 60] that presents interesting papers taken from major conferences on a variety of online problems. These surveys have been created together with Rob van Stee and Sören Schmidt.

Chapter 3 and 4 are devoted to the buffer minimization problem with conflicts. Chapter 3 is based on a series of papers published in the proceedings of the 14-th FAW conference and *Theoretical Computer Science* [61, 63]. The first paper provides results on the competitive ratio of the problem on certain line segments while the second version of the paper considers similar instances but introduces a new variation of the problem and thus gives results from a different perspective. Chapter 3 combines both works and presents them in a cohesive manner. An additional section is added that discusses results that consider more general graphs than the line graph.

A goal to strive for when considering the buffer minimization problem with conflicts is a sublinear upper bound on the path with m machines. Chapter 4 considers a number of candidate algorithms that fell short on this goal, and presents linear lower bounds for these algorithms.

In chapter 5, we adress a different problem, the bamboo garden trimming problem. There are two versions of the problem, a discrete and a continuous one. We improve the approximation ratio in the discrete case and present results on the star graph in the continuous case. This paper is currently submitted to WAOA.

Finally, in chapter 6 we discuss the fair allocation of contiguous blocks of indivisible chores and develop the prices of fairness for this problem under different notions of fairness. This paper has been published in the journal *Information and Computation* [62].

All of these papers have been created working together with my advisor Rob van Stee.

Chapter 2

A survey of online algorithms

In online optimization an algorithm faces an uncertain future and must irrevocably make decisions with incomplete information about the input. This is a wide area of research that contains many subcategories and problems such as scheduling, matching, bin packing and many more.

Research on each of these problems is very active and many interesting papers are published every year.

In this chapter we present a selection of papers from this field that were written in the years 2019 to 2021. Since research has been very active, we have made a selection by choosing papers that have been published in major conferences and highlighting works that are closely related to our work.

In this work we focus mainly on problems in the realm of scheduling, but of course there are many other active and interesting topics to be found in online optimization. The papers discussed here and similar presentations on different topics can be found in the yearly SIGACT News column on online optimization.

We begin by examining a topic that has recently seen a lot of progress and is also closely related to our research.

2.1 The p-processor cup game

The p-processor cup game is a problem that naturally arises in the context of processor scheduling. In this game, a filler distributes up to p units of water among n cups. Then an emptier selects p cups and removes up to one unit of water from each. The goal is to minimize the amount of water in the fullest cup which is called the backlog. We can also think of p processors working on one of n tasks each while p new units of work may arrive during each slice of time.

For the single processor cup game the GREEDY algorithm (that always chooses the highest cup) is $O(\log n)$ competitive. Positive results for the problem mostly considered a variation of the problem which uses resource augmentation where the filler is restricted to distributing $(1-\varepsilon)p$ units of water in each step with at most $1-\delta$ of the water going into a single cup.

In STOC 2019, Bender et al. [14] consider this version of the problem. They introduce a randomized algorithm called the smoothed greedy algorithm.

This algorithm first inserts a random quantity $r_j \in (0,1)$ into each cup j. Afterwards the algorithm behaves like GREEDY with one exception. If any of the the highest cups contains less than one unit of load, the algorithm does not remove any load from these cups. This ensures that the randomized quantities that differentiate cups of similar loads remain intact.

This small amount of randomization greatly improves the performance of the algorithm on the single processor cup game and achieves backlog at most $O(k)$ for any $k \geq \Omega(\log \varepsilon^{-1})$ with probability at most $1 - O(2^{-2^k})$. They then present a simple analysis of the deterministic greedy algorithm for the multi-processor cup game which establishes a backlog of $O(\varepsilon^{-1} \log n)$ as long as $d > 1/\text{poly}(n)$.

Using a more intricate randomized algorithm they achieve backlog $O(\frac{1}{\varepsilon} \log \log n)$ with probability $1 - \frac{1}{\text{poly}(n)}$ as long as ε and δ satisfy reasonable constraints. Additionally, they make the interesting observation that the backlog drops to a constant with high probability if p is large enough. The algorithm maintains backlog 3 after any given step with probability $O(1 - O(\exp(\varepsilon^{2p})))$. This means if ε is constant this brings the probability of the backlog being super-constant down to exponentially small in p, whereas in the single processor cup game no bounds on constant backlog can be achieved with better than constant probability.

In SODA 2020, William Kuszmaul [82] considers the version of the problem without resource augmentation. He shows that GREEDY (the algorithm that always chooses the highest p cups) is $O(\log n)$ competitive and this is optimal for $n \geq 2p$. This is shown using an intricate system of invariants for the p-processor cup game.

Then he again considers the smoothed greedy algorithm and shows that this algorithm achieves backlog $O(\log p + \log \log n)$ with probability $1 - 2^{-\text{polylog}(n)}$ however, in the case of the vanilla cup game this only holds for $2^{\text{polylog}(n)}$ steps. For fixed p and n large enough this becomes $O(\log \log n)$ which is asymptotically optimal. This result doubles as a smoothed analysis of the deterministic greedy algorithm.

In SODA 2021, Michael Bender and William Kuszmaul [16] follow up on these results once more and show that randomized algorithms can still achieve a backlog of $O(\log \log n)$ even if the adversary is not completely oblivious. It is known that this is not possible against a fully adaptive adversary. The adaptive adversary considered in this paper knows the fills of every cup which contains more than three units of water but not of the cups containing less than three units. The algorithm dynamically switches between a "greedy" mode that prevents the adversary from quickly building up the fill and a prescheduled mode that slowly leaks water from the cups while obscuring the fills from the adversary. The second mode prevents the filler from building up backlog over a long period of time.

In ITCS 2021, William Kuszmaul and Alek Westover [84] introduce a variation of this problem where p changes each round. They show that this version of the problem is significantly more difficult. The optimal backlog is now $\theta(n)$. There is a filling strategy that yields backlog $\Omega(n^{1-\varepsilon})$ in quasipolynomial time and even straightforward randomization with a "greedy-like" algorithm does not improve this result. A "greedy-like"-algorithm chooses the fullest cup whenever possible, but uses randomization to break ties.

Furthermore, in STOC 2021 William Kuszmaul [83] considers an alternate objective function called tail size, which is the number of cups with fill more than 2. They present a randomized algorithm that achieves tailsize $\tilde{O}(\log n + p)$ with high probability for poly n steps. The algorithm introduces a randomized offset on the fill of the cups as well as a random priority on the cups. If all cups have low fill, the algorithm chooses cups based on priority while maintaining the offset. If there are cups with high fill then the cups are chosen greedily. They also consider games with more than poly n steps and show that no monotone memoryless emptying algorithm can achieve a good guarantee in this case. However, a small resource augmentation of $1/\text{poly } n$ is sufficient to overcome this barrier.

2.2 Scheduling

While the problems presented in the previous chapter were closely related to our research, there are many more interesting topics that fall into the area of online scheduling. In this chapter we present a selection of recent papers that fall into this category.

One concept that naturally arises in the area of scheduling is the idea of deadlines. For example, if we aim to maintain a certain height among the bamboo in a garden then whenever we cut a bamboo we are given a deadline and we need to cut the bamboo again before this deadline expires. So in this context we are dealing with hard deadlines that must be adhered to. In other problems it is also possible to miss a deadline as long as the value gained by missing this deadline offsets the cost. We begin the survey of scheduling with two problems that revolve around this idea.

In the online packet scheduling problem with deadlines, packets arrive over time on a network switch and need to be sent across a link. Each packet comes with a deadline and a weight. Since only one packet can be transmitted per timeslot, some of the packets will inevitably expire and be lost. The goal is to find a schedule that maximizes the weights of packets sent across the link. The problem has been known to have a lower bound of $\phi \approx 1.618$ on the competitive ratio. In SODA 2019, Vesely et al. [110] give an algorithm that matches the lower bound. This ϕ-competitive algorithm is based on the concept of the plan, which at any given time t, is the maximum-weight subset of pending packets that can be feasibly scheduled in the future (if no other packets arrive). For each packet p in the plan P at time t there is a substitute packet $sub(P, p)$ which gets scheduled in the plan, if p gets scheduled at time t. Greedily scheduling the heaviest pending packet may not be optimal, since its substitute may be small. Both the weight of the packet and its substitute should be considered. The algorithm schedules p at time t such that $w_p + \phi w(sub(P, p))$ is maximized. This guarantees ϕ-competitiveness, but only in certain instances in which a property called slot-monotonicity holds.

Let $minwt(P, \tau)$ be the minimum weight of a packet that can be scheduled in some slot between the current time and a time slot τ. Slot monotonicity holds if for a fixed τ, $minwt(\tau)$ is monotone increasing as t goes from 0 to τ. The algorithm then increases weights and decreases deadlines forcing this property to hold. These changes get accounted for in the (very technical) analysis.

In SPAA 2020, Jamalabadi et al. [72] considered an admission control problem on parallel machines. Jobs with deadlines arrive online and need to be assigned to a machine (potentially starting only later) or discarded immediately. The goal is to maximize the total size of the accepted jobs. Of course, the schedule must be such that all accepted jobs complete by their deadlines.

The authors show that if each job has a small amount of slack, it is possible to design a nearly optimal deterministic online algorithm. Here a slack of ε means that for a job with size p, deadline d and release date r, we always have $d \geq (1+\varepsilon)p + r$. The paper first gives a lower bound. The design of the algorithm is then based on this lower bound construction. This is the first such result which holds for parallel machines and does not require job preemption or migration.

It is sometimes possible to handle a variety of seemingly disparate problems with the same algorithm. In ICALP 2021, Marcin Bienkowski et al. [19] present an algorithm for minimizing average completion time in various online scheduling problems. Their algorithm executes a routine $\textsc{Mimic}(\gamma, \omega)$, where γ is a parameter characteristic to the scheduling problem and ω controls an initial delay.

In the deterministic version there is no initial delay, so $\omega = 0$. For the randomized version, ω is first chosen uniformly at random from the range $(-1, 0]$ and then $\textsc{Mimic}(\gamma, \omega)$ is executed. Their routine works by computing and later executing auxiliary schedules, each optimizing a certain function on an already seen prefix of the input.

They classify scheduling problems as γ-resettable if they obey three restrictions. These problems have to allow delayed execution, so if the executor at any time t is in the initial state, it can execute an arbitrary auxiliary τ-schedule (so a schedule of length τ) in the time interval $[t, t+\tau)$. The second restriction is that if the executor executed a τ-schedule from the initial state, then it must be possible to reset the executor using at most extra $\gamma \cdot \tau$ time. Lastly, the value of $\min(\mathcal{I})$ (the earliest time at which $\textsc{Opt}$ may complete some job), is required to be learned by an online algorithm at or before time $\min(\mathcal{I})$ and additionally they require $\min(\mathcal{I}) > 0$.

The authors then show that $\textsc{Mimic}$ achieves a deterministic competitive ratio of $3 + \gamma$ and a randomized one of $1 + (1+\gamma)/\ln(2+\gamma)$ for γ-resettable scheduling problems.

Then they observe that three well-known problems are γ-resettable. In the traveling repairperson problem (TRP), where requests arrive in time at points of a metric space and need to be serviced by a server that moves at constant speed, the goal is to minimize the sum of all completion times. For the dial-a-ride problem (DARP), where the requests consist of a source and a destination, the goal is to transport an object between these two points. Here the server may carry a fixed number of objects simultaneously. The authors observe that TRP and DARP are 1-resettable and hence, $\textsc{Mimic}$ improves the best known deterministic upper bounds for TRP and DARP from 5.14 to 4 and the randomized upper bounds from 3.641 to 2.821. For the scheduling problem on m unrelated machines, where weighted jobs arrive online, the goal is to minimize the weighted sum of completion times. This problem is 0-resettable. MIMIC implies a deterministic upper bound of 3 (previously best known was 4) and a randomized upper bound of 2.443 (previously best known was 2.886). It is worthy to note that their results apply to many variations of these mentioned problems. For example, for the scheduling problem on unrelated machines their algorithm is still applicable if preemption is allowed or if there exist precedence constraints.

The next papers picks up on the notion of precedence constraint scheduling. Jobs arrive online, each with processing time and weight, and need to be scheduled on one of m machines. In the case of unrelated machines, the processing time of a job may be different for each machine.

Precedence constraints are given by a partial order $\prec$ between jobs. For jobs j and j' with $j \prec j'$ job j' may not be started until j is finished. Naturally, this partial order should respect release dates.

In ICALP 2019, Naveen Garg et al. [50] consider the online problem of scheduling jobs on identical machines with precedence constraints. In the case of identical machines, the processing time of a job is the same for every machine. Furthermore, the algorithm is non-clairvoyant, meaning it does not know the processing time of a job before it is finished. Garg et al. give a 10-competitive algorithm for minimizing the total weighted completion time.

Let I_t be the arrived jobs that can be scheduled and J_t all arrived jobs including the ones that can not be scheduled due to precedence constraints. This means $I_t \subseteq J_t$. The processing time among the jobs in I_t is determined using a convex program that maximizes the weighted Nash welfare $\sum_{J_t} w_j \log R_j$ where R_j is the virtual rate of a job regardless whether it can currently be run. Jobs can then "donate" their allocated time to some preceding job in I_t. This can be solved using a Eisenberg-Gale-type convex program.

They then extend this result to an $O(1/\varepsilon)$-competitive algorithm for minimizing the average weighted flow time. This requires a $(1+\varepsilon)$-speedup and no-surprises assumption, meaning when a job j is released, all jobs having a precedence relationship to j are also released at the same time. That is, all jobs in each connected component of precedence constraints have the same release date. This assumption is shown to be necessary.

We conclude this section by reviewing two more scheduling papers. The first one considers flow time optimization on a single machine and the second considers the makespan on two-dimensional machines.

In the flow time scheduling problem, jobs arrive on a single machine and the goal is to minimize the weighted sum over all jobs of the difference between release and completion time. Preemption is possible. The best known algorithms for this problem are $O(\log W)$, $O(\log P)$ and $O(\log D)$ competitive where P, D, W are the maximum ratios of processing time, weights and density which is processing time divided by weight. These algorithms immediately degrade to a polynomial factor if the processing times are distorted by a factor up to μ even if $\mu = 1 + \varepsilon$. Yossi Azar et al. [6] in STOC 2021 introduce algorithms that are robust when facing such distortion.

In ICALP 2020, Cohen et al. [35] consider online 2D load balancing. Two-dimensional vectors arrive and need to be assigned to two-dimensional machines. The goal is to minimize the makespan, which is defined as the maximum load of any machine in any dimension. (The load of a machine in dimension i is the sum of the i-th entries of the vectors assigned to this machine.) This can of course be seen as a special case of vector scheduling, which has already been (asymptotically) resolved. However, it is also a generalization of the widely studied online load balancing problem. The authors argue that this is an important case to study as it models the case where resources of two types are available and we try to minimize their usage.

First, they show that (a straightforward generalization of) the classical load balancing algorithm has a competitive ratio of exactly 8/3. This is done by analyzing reachable states for priority algorithms [26], which allows us to essentially restrict our attention to two machines. In the analysis, the pair of machines considered needs to be selected carefully, and it is therefore not simply a matter of analyzing an algorithm which only has two machines in the first place.

For the case where OPT is known, the authors give a best fit-type algorithm with competitive ratio exactly 9/4. (For one dimension, the best known result for this problem is 3/2.) This algorithm works by balancing between the two dimensions and minimizing the difference in their loads subject to the constraint that the goal competitive ratio of 2.25 is maintained. They further analyze First Fit and show that it has competitive ratio 2.5 for known OPT and 2.89 for unknown OPT (by maintaining a guess for OPT), beating the simple upper bound of 3 achieved by a greedy algorithm.

Finally, they give an existence result for fixed dimension d, showing that it is (in theory) possible to give a deterministic online algorithm with competitive ratio arbitrarily close to c_d, the optimal competitive ratio against the fractional optimal solution. It is interesting that this can be achieved, especially given that c_d is not actually known. As the authors stress, actually constructing the algorithm (or to be precise, constructing the decision tree that it needs) requires extremely large, though polynomial for fixed d and ε, running time.

2.3 Advice and predictions

Recently, increased attention has been given to the augmentation of online algorithms with advice. This advice may come in the form of machine learned predictions. In many cases these predictions allow an algorithm to perform much better than an algorithm that does not have access to such predictors, and in particular it may allow such algorithms to break strong lower bounds in a variety of problems.

The performance of an algorithm that has access to predictions is impacted by the quality of the prediction. When given good but not perfect predictions, an algorithm with advice may outperform any algorithm that does not have access to any advice. However, it is also important to consider how fast the performance of an algorithm degrades when given bad or even adversarial advice. In particular it may be possible that an algorithm starts to perform worse than a regular algorithm if the quality of the advice is bad enough.

We begin the survey of online algorithms with advice by presenting two papers that particularly emphasize how the latter phenomenon may be avoided by designing algorithms that are robust against prediction errors.

In SPAA 2021, Sungjin Im et al. [69] consider non-clairvoyant scheduling with predictions that may be erroneous and even adversarial. Non-clairvoyance means that the size of a job is only known when it completes. This of course makes things very difficult for an online algorithm. Previous results on this problem by Manesh Purohit et al. [95] were rather pessimistic. The authors show that a new measure of prediction quality can be used to obtain better results. Prediction errors that satisfy monotonicity (if more job sizes predicted are correct, the error decreases) and a Lipschitz condition (predictions can only be considered good if their associated optimal schedule is close to the true optimal schedule) give better guarantees. It is now possible to obtain an algorithm that is $O(1/\varepsilon)$-robust and $(1+\varepsilon)$-consistent in expectation. Consistency means that the performance of the algorithm improves with good predictions and robustness means that the algorithm gracefully handles bad predictions.

The algorithm works by repeatedly sampling a small subset of the remaining jobs and running *half* of them to completion (it may take too long to wait until all jobs complete). This is used to obtain an estimate of the median job size. Then, it samples some more jobs in order to estimate the error of the predictions, but caps their running time to avoid spending too long here.

In ITCS 2020, Spyros Angelopolous et al. [2] introduced a framework for online computation with untrusted advice. The next year in ITCS 2021, Spyros Angelopolous [1] adapts this framework to the online search or cow-path-problem. In this version of the problem, the cow is given a hint about the location of the destination. There are two competitive ratios to consider, one for the case where the hint is true, called consistency, and one for the case where the hint is false, called robustness. The goal is to design an algorithm with a good tradeoff between these two competitive ratios which means looking for Pareto-optimal algorithms.

In a geometric strategy with base b the algorithm walks distance b^i in a direction along the line, alternating between the left and right side of the point of origin. Given a hint that is the direction of the destination from the point of origin, the distance travelled away from the direction of the hint is scaled with a parameter $d \in [0, 1)$ and this is a Pareto-optimal strategy with consistency $1 + 2(\frac{b^2}{b^2-1} + \delta\frac{b^3}{b^2-1})$ and robustness $1 + 2(\frac{b^2}{b^2-1} + \frac{1}{\delta}\frac{b^3}{b^2-1})$. Another possibility is a hint that gives away the exact location. In this case, the algorithm chooses a geometric strategy with parameter b_r such that it is r-competitive in the case of the hint being wrong. This strategy is scaled such that the destination lies on a point where the algorithm turns around. This avoids wasted movement and allows for a $\frac{b_r+1}{b_r-1}$-competitive ratio in the case of the hint being correct.

Finally, they give upper and lower bounds in the case that a k-bit string is given as advice.

The paper from 2020 contains similar results on different problems such as ski-rental, online bidding, bin packing and list update.

After considering how to deal with adversarial advice we now present a series of papers that show how machine-learned predictions with only small errors may be used to overcome established lower bounds in classic online problems.

We begin with Silvio Lattanzi et al. [85] who, in SODA 2020, consider online scheduling with machine learned predictions. Specifically, they consider makespan minimization under restricted assignment. This is a special case of unrelated machines. Jobs arrive one at a time, each annotated with its size and a subset of m machines it can run on.

In the classical online setting, there is an $\Omega(\log m)$ lower bound for the competitive ratio. Lattanzi et al. assign weights to each machine, and fractionally assign jobs according to these weights. Given predictions of these weights with error η, they construct a $O(\log \eta)$-competitive fractional assignment. The second step is an online rounding algorithm that rounds any fractional assignment into an integral schedule, losing an $O((\log \log m)^3)$ factor in the makespan. This is nearly optimal, since they also give an $\Omega(\log \log m)$ lower bound for online rounding algorithms. Combining both steps yields a competitive ratio of $O(\log \eta (\log \log m)^3)$.

We continue with a series of papers that examine the online caching problem. In the online caching problem, a sequence of items arrives one by one, and we have a cache of fixed size. The goal is to minimize the number of page faults. For the traditional online model, there is a lower bound of $\Omega(\log k)$ on the competitive ratio of randomized algorithms.

We can get around this lower bound by changing the model and giving the algorithm access to more information in the form of machine learned advice. In the case of online caching, an oracle predicts when an element will arrive again. Lykouris and Vassilvitskii [87] showed that there is a prediction augmented caching algorithm with a competitive ratio of $O(1+\min(\sqrt{\eta/\text{OPT}}, \log k))$ where the prediction error is bounded by η. The dependence on k in the competitive ratio is optimal but the dependence on η/OPT left room for improvement. Dhruv Rohatgi in SODA 2020 [99] makes progress in closing this gap by providing an improved algorithm with competitive ratio $O(1+\min(\frac{\log k}{k}\frac{\eta}{\text{OPT}}, \log k))$ and a lower bound of $\Omega(\log \min(\frac{\eta/\text{OPT}}{k \log k}, k))$.

The algorithm in [87] is a marking algorithm that considers eviction chains and trusts the oracle until the chain becomes too long and then ignores the oracle. A marking algorithm works in phases that start with all elements being unmarked. When a cache hit occurs, the corresponding element is marked. When a cache miss occurs, some unmarked element is evicted, and the new element is inserted and marked. If all elements in the cache are marked and a cache miss occurs, all elements are unmarked, and a new phase begins. Rohatgi shows that the simple improvement of only trusting the oracle once at the start of each eviction chain already leads to an improved ratio of $O(1 + \min(\frac{\log \eta}{\text{OPT}}, \log k))$. Further improvement of the algorithm requires deviating from the marking based framework by sometimes evicting marked elements.

Alexander Wei [112] in APPROX 2020 directly follows up on this work. They give a substantially simpler algorithm that gives an improved competitive ratio of $O(1 + \min(\frac{1}{k}\frac{\eta}{\text{OPT}}, \log k))$. This algorithm compares LRU and BLINDORACLE, the algorithm that blindly follows the oracle, and follows the one that has performed better so far. This algorithm is optimal among deterministic algorithms, which means randomization is needed to approach Ruhatgi's lower bound.

Zhihao Jiang et al. in ICALP 2020 [73] continue this line of work by considering *weighted* paging (caching) with predictions. As shown by Rohatgi [99], predicting the next arrival of an element with perfect accuracy gives a competitive ratio of $O(1)$ for unweighted paging. Furthermore it is also possible to obtain a constant approximation when given a lookahead of $l \geq k-2$ distinct pages. However, neither of these models are sufficient to beat the existing lower bounds for online weighted caching, which are $\Omega(k)$ for deterministic and $\Omega(\log k)$ for randomized algorithms. Jiang et al. show that combining both models allows for a 2-competitive algorithm. In the combined model, when an item p arrives the algorithm is given the next timestep when p will arrive again and additionally all page requests until that request.

In ITCS 2021, Yuval Emek et al. [42] consider a variant of the online paging problem where the online algorithm has access to multiple predictors that produce a sequence of predictions for next page arrival times. At least one of those predictors makes a sublinear number of errors in total. They show that this assumption suffices to design randomized algorithms whose regret compared to the optimal algorithm tends to zero as time goes to infinity.

When given only a single predictor with a sublinear number of errors, simulating the optimal offline algorithm longest-distance-forward based on the predictions yields a vanishing regret. In the full-access model the algorithm is given multiple predictors and an algorithm from Avrim Blum and Carl Burch [23] is used that, given multiple online algorithms as subroutines, performs almost as good as the best subroutine in hindsight. In the bandit access model only one predictor may be queried each round and they show that the good predictor can be "chased" by the online algorithm and vanishing regret can be achieved.

The final two papers considered in this section explore different approaches of giving advice to an algorithm. The first paper again considers the paging problem, while the second explores their model in the network flow allocation problem.

Ravi Kumar et al. in SODA 2020 [79] consider a different way of giving the algorithm information about the request sequence. They consider a semi-online model where request sequences are generated by walks on a directed graph called the access graph. The algorithm knows the access graph but not the actual request sequence. Kumar et al. extend this model by creating several individual request sequences using access graphs. The final sequence given to the algorithm is then obtained by interleaving the individual sequences. This approach is motivated by multitasking systems that support a number of tasks at the same time with a limited cache.

For the case of a single access graph, they give a tight competitive ratio of $\Theta(b)$ where b is a structural parameter of the graph called branching depth. For the interleaved model, they give a tight competitive ratio of $\Theta(b + \log t)$ where t is the maximum number of interleaved access graphs. For the case of interleaved cashing, they reduce the problem to a coin hunting game where a hider hides a coin in one of t bins and a seeker, that only knows the probability distribution used by the hider, tries to find that coin. Interestingly, they show that this coin hunting game has other applications besides the interleaved caching model. In particular, it can be used to yield an alternative analysis of the classical random marking algorithm for the standard caching problem.

In ESA 2021, Thomas Lavistate et al. [86] propose a new model for augmenting algorithms with predictions, by requiring that these predictions are learnable using only a reasonable amount of past data as well as robust to changes in the input. They build algorithms that fulfill these criteria for the network flow allocation problem. Consider a directed acyclic graph with capacities on each node that is not a sink or source node (nodes with only incoming or only outgoing edges). The source nodes arrive one by one and flow has to be fractionally directed towards a single sink node without exceeding capacities. The goal is to maximize the flow.

They provide an offline $(1 - \varepsilon)$-approximation algorithm that uses weights to determine the flow. These weights can be learned using only a polynomial sample complexity. However, future inputs may not follow the same distribution. The competitive ratio of the algorithm linearly degrades as the distance between the inputs grows, but never performs worse than the best worst-case algorithm even if the prediction error is large. They provide similar results for the problem of restricted assignment makespan minimization.

2.4 Online matching

The most commonly studied version of online matching is bipartite online matching, which was introduced by Karp et al. [75] in STOC 1990. In this version, one set L of vertices is given in advance (the offline vertices) and the other set R arrives online. There is a variety of other models, many of which have seen a lot of progress in the last years. In this section, we present a selection of papers covering a variety of models, beginning with a very strong result on online matching on the line.

In 2021, the famous online matching on the line problem was nearly resolved. In this problem, there are n servers on given positions on the real line. Requests (also on the line) arrive and need to be matched to the servers. A few years ago, Krati Nayyar and Sharath Raghvendra [92] presented an algorithm called Robust Matching which Raghvendra [97] later showed to be $O(\log n)$-competitive. In ICALP 2021, Enoch Peserico and Michele Scquizzato [94] give a remarkably straightforward lower bound for this problem, showing that no randomized algorithm can be $o(\sqrt{\log n})$-competitive for this problem. The construction is roughly as follows (omitting some technical issues): For $n = 2^k$, partition a line segment of length n into k subintervals. The servers are distributed evenly over these subintervals, and for each subinterval, one request arrives in it at a uniformly random location. Then, in k rounds, this process is repeated, doubling the length of the subintervals in each round. In total, n requests arrive, the expected online cost for each round is $\Omega(n)$ and the expected offline cost for each request is $O(\sqrt{\log n})$.

One year earlier in APPROX 2020, Nicole Megow and Lukas Nölke [88] also consider the problem of online matching on the line. They consider the situation where previous matching decisions may be changed to some extent (the algorithm has *recourse*) and they give an $O(1)$-competitive algorithm using $O(\log n)$ recourse. The algorithm interpolates between an $O(\log n)$-competitive online solution without recourse and an $O(1)$-approximate offline solution, i.e., with very large recourse. They also consider the special case of alternating instances, in which no more than one request arrives between two adjacent servers, and give a $(1+\varepsilon)$-competitive algorithm that reassigns each request $O(\varepsilon^{-1.001})$ times. For this special case, the lower bound of $\Omega(\log n)$ without recourse still holds.

In the same workshop, Varun Gupta et al. [56] also give an $O(\log k)$-competitive algorithm for online matching in general metrics, using $O(\log k)$ recourse per client, as well as a 3-competitive algorithm on the line using $O(\log k)$ amortized recourse. They also consider the dynamic setting in which clients and servers may arrive or depart. For this model, they present a simple randomized $O(\log n)$-competitive algorithm with $O(\log \Delta)$ recourse, where Δ is the aspect ratio of the underlying metric. Without recourse, it is not possible to achieve any nontrivial result in this setting.

In STOC 2018, Zhiyi Huang et al. [65] introduced a fully online model in which all vertices are online. Once a vertex arrives, its incident edges to previously arrived vertices are revealed. This is called an edge-arrival model. Each vertex has a deadline that is after all its neighbors' arrivals. If a vertex is unmatched until its deadline, it must then irrevocably either be matched to another unmatched vertex or be left unmatched. Huang et al. showed that for bipartite graphs the competitive ratio of RANKING is between 0.5541 and 0.5671 and for general graph the algorithm is 0.5211 competitive.

In SODA 2019, Zhiyi Huang et al. [66] (an overlapping set of authors) improve this result to a tight bound of 0.5671 on bipartite graphs. A second result is the following: If fractional matchings are allowed then the algorithm WATER FILLING, which at each vertex's deadline matches its unmatched portion fractionally to all neighbors with smallest matched portion, achieves a tight competitive ratio of $2-\sqrt{2} = 0.585$. This hardness result applies to arbitrary algorithms in edge-arrival models, even when preemption is allowed, improving the upper bound of $\frac{1}{1+\ln 2} = 0.5906$ [43].

In FOCS 2020, Zhiyi Huang et al. [68] follow up on these results once more. Ideas from WATER FILLING and RANKING are used to give a 0.569-competitive algorithm for integral matching using the online primal dual framework. Essentially, the authors find a way to (in some cases) correct the decisions by RANKING using the decisions by WATER FILLING (even though that is an algorithm for fractional matching). They also give an algorithm for fractional matching on general graphs called EAGER WATER-FILLING which sometimes matches vertices immediately upon arrival instead of always waiting until the deadline. This algorithm achieves a competitive ratio of 0.592.

Ashlagi et al. [3] in EC consider a variation of this problem where the edges are weighted. They are motivated by ride-sharing and formulate their problem as follows: At every timestep passengers arrive as vertices of a graph. Each edge has a nonnegative weight, the utility from matching two passenger-vertices. Passengers can only be matched within a certain window of length d after which the passenger must be assigned a single ride and the vertex departs. The goal is to find a weighted matching with a large total weight in an online manner.

They first study the case where vertices arrive adversarially. The naive greedy algorithm randomly assigns each vertex to be either a seller or a buyer. Ashlagi et al. present an $\frac{1}{4}$-competitive algorithm *Postponed Greedy*, which postpones this decision for as long as possible in order to gather more information about the graph structure. They extend the model to the case where the departure of vertices is determined stochastically. Postponed Greedy can be adapted to be $\frac{1}{8}$-competitive, when the departure distribution is memoryless and realized departure times are revealed to the algorithm just when the vertex departs.

They also consider the setting where vertices arrive in a random order. A batching algorithm which, every $d+1$ steps, computes a maximum weighted matching among the last $d+1$ arrivals, achieves a competitive ratio of 0.279.

In FOCS 2019, Buddhima Gamlath et al. [48] consider the question whether randomization helps in online matching, in particular if beating the lower bound of $\frac{1}{2}$ by Karp is possible. For bipartite matching, this has been answered by Karp and Vazirani [75]. Gamlath et al. show that for general vertex arrival models, which are similar to the fully online model above but the decision to match a vertex must be made on arrival of the vertex, there exists a random $\frac{1}{2}+\Omega(1)$-competitive algorithm. This algorithm works by skillfully rounding a slightly better than $\frac{1}{2}$-fractional matching, without incurring too high a loss. This fractional online matching algorithm was originally created by Wang and Wong in ICALP 2015 [111]. They also consider the edge arrival model where edges are revealed one by one and the algorithm must immediately and irrevocably decide to include the edge in the matching or not. They show that in the edge arrival model randomization does not help and no randomized algorithm is better than $\frac{1}{2}$.

We continue with another paper that considers edge arrivals. In the bipartite matching problem with edge arrivals, the straightforward greedy algorithm, that adds any arriving edge to the matching whenever possible, achieves an optimal competitive ratio of 0.5. In ISAAC 2021, Nick Gravin et al. [54] consider the stochastic version of the problem where edges arrive with a certain probability. They propose a class of algorithms called prune&greedy that first prune the graph by decreasing the probabilities of some edges and then proceed with the greedy algorithm. This yields a 0.503-competitive ratio on arbitrary stochastic bipartite graphs. In the special case where the graph can be pruned to a log-normalized 2-regular stochastic graph, greedy is 0.552-competitive. These graphs extend the concept of c-regularity to the stochastic setting. Instead of summing up the amount of edges incident to a node, a sum of weights is considered. These weights depend on the probability with which the edge is realized. Additionally, they show that no online-algorithm can be better than $2/3$ in this setting.

There are still many more variations of the matching problem that can be obtained by changing the way nodes and edges arrive as is illustrated by the following papers.

Kumar et al. [78] introduce a *semi-online* model in ITCS, which separates the unknown future into an adversarial part and a predicted part. The chosen example is bipartite matching, with an offline side that is known in advance and an online side whose nodes and incident edges are revealed one at a time. The online side is partitioned into a predicted set of size $n - d$ and an adversarial set of size d. The algorithm knows the incidents of all predicted nodes, but nothing about the adversarial ones. It is furthermore assumed that the optimal solution is a perfect matching. They then give an iterative sampling algorithm, that reserves a set of offline nodes to be matched to the adversarial nodes by repeatedly selecting a random offline node that is unnecessary for matching the predictable nodes. This gives a $(1 - \delta + \frac{\delta^2}{2}(1 - 1/\varepsilon))^2$ competitive algorithm, where $\delta = d/n$ is the fraction of adversarial nodes. An improved algorithm samples a set of nodes that in expectation has a large overlap with the set matched to adversarial nodes in the optimum solution. This yields a $(1 - \delta + \delta^2(1 - 1/\varepsilon))$-competitive algorithm.

These algorithms are complimented by a lower bound of $(1 - \delta\varepsilon^{-\delta})$, showing that the algorithms are near optimal. This lower bound coincides with the best offline bound of 1 for $\delta = 0$ and the best online bound of $(1 - 1/\varepsilon)$ for $\delta = 1$.

In ITCS 2021, Yiding Feng and Rad Niazadeh [46] consider the online bipartite matching problem but instead of nodes being revealed one by one, they arrive in K batches of nodes. For the original problem where nodes arrive one by one, the algorithm BALANCE achieves a competitive ratio of $(1 - 1/e)$. This algorithm balances a greedy approach with the robustness against nodes arriving in the future.

This algorithm is generalized to the batch arrival problem. In each step, the algorithm solves a convex program that maximizes the total weight but also subtracts a regularization term. This regularization term is based on a set of polynomials whose degree decreases with each step. This means at the beginning there is more regularization applied while the last step is entirely greedy. This approach yields a $1 - (1 - 1/K)^K$-competitive algorithm. This result can be extended to further variants of the problem like vertex weighted bipartite b-matching and AdWords [90].

In STOC 2021, Zhiyi Huang and Xinkai Shu [67] consider online stochastic bipartite matching. In this problem, n online nodes are drawn from a set of types by some distribution. This means a node of a certain type can be drawn multiple times. This can be compared to the random-order model where the type of each node is set in advance and only the order is randomized. They improve the competitive ratio from 0.706 to 0.711 in the unweighted case, and from 0.662 to 0.7009 in the weighted case. In the weighted case, this is the first improvement over the weaker random-order model, and it is achieved by designing an algorithmic amortization that replaces the analytic one in previous works. They introduce the Poisson arrival model where each type of node arrives following a Poisson process and the overall number of online nodes is not fixed but drawn from a Poisson distribution. This model is easier to analyze, and the model is asymptotically equivalent to online stochastic matching. They use a natural linear program to lower bound the optimal solution, and previously used linear programs are special cases of this natural linear program.

In STOC 2020, Zhiyi Huang and Qiankun Zhang [89] consider online matching with stochastic rewards. Each edge has a success probability and the matching along this edge succeeds with that probability. This models online advertising, where it is not certain that a user will click on an ad. This model was introduced by Mehta and Panigrahi in FOCS 2012 [89], who focused on the case where every edge has the same success probability p. They gave results for general p and for the case where p tends to 0 (vanishing probabilities).

Huang and Zhang apply the randomized online primal dual framework using the configuration LP, and achieve a competitive ratio of 0.572 (resp. 0.576) for the case of vanishing and unequal (resp. unequal) probabilities. The previous best results were 0.534 due to Mehta et al. [91] and 0.567 [89].

After examining a variety of different arrival models, we return to the bipartite matching problem with regular arrivals but add a twist to the objective funktion.

In FOCS 2020, Matthew Fahrbach et al. [45] consider the less studied edge-weighted version of the online bipartite matching problem. A simple example with three nodes and two edges, where the first arriving edge has weight 1 and the second has weight either 0 or W for some large value W shows that it is not possible to be competitive in this model without additional assumptions. Fahrbach et al. consider the free disposal model, in which previously matched edges may be disposed of for free, a widely-accepted assumption in display advertising. For this model, the authors give the first algorithm which beats the ratio of $1/2$ by providing an algorithm with competitive ratio 0.5086.

An important idea in the algorithm is to use a subroutine which the authors call online correlated selection. It ensures that decisions across different pairs of vertices are negatively correlated, and ultimately guarantees that a vertex appearing in k pairs is selected at least once with probability greater than $1 - 2^{-k}$. Essentially, for neighbors of an online vertex, we select the least matched ones. For the vertex-weighted problem, this would mean those vertices that have appeared in the least number of pairs. For the edge-weighted problem, the authors use the online primal-dual framework of Devanur et al. [39] and a formulation of the problem due to Devanur et al. [38] to design a “least-matched” notion.

This routine may have uses in other online problems as well and thus the question arises whether it can be extended to choices between three or more vertices. This question is positively answered by Yongho Shin and Hyung-Chan An [103] in ISAAC 2021. They consider a subroutine for three nodes that simply runs two instances of the two-way-OCS and uses the output of the first instance as well as the third vertex as input for the second instance. The resulting probability distribution is not easy to analyze but results in an improved competitive ratio for edge-weighted online bipartite matching of 0.5093.

To end the section on matching, we explore two papers that leave graphs behind and consider matching problems on general metrics as well as a paper that connects weighted bipartite matching with thruthful mechanisms.

The minimum-cost metric matching problem was already mentioned in SIGACT 2017. There are n servers in metric space. Requests arrive online and need to be matched to these servers. Nayyar and Raghvendra in APPROX 2016 [96] gave a tight $O(\log n)$-competitive algorithm for the random-arrival model. Of interest is also the case where the metric is a line or tree. Gupta et al. [55] in ICALP 2019 consider the case where the requests arrive randomly and are drawn independently from a known probability distribution. They show that this case allows for a substantially better algorithm. They give an $O((\log\log\log n)^2)$-competitive algorithm for this model, that is furthermore 9-competitive for tree and line metrics. In each round, the algorithm computes an optimal fractional matching between remaining free servers and remaining requests (in expectation). New arriving requests then get matched with probabilities according to this matching.

In SODA 2021, Yossi Azar et al. [7] consider online matching with delays. Requests arrive over time, possibly in different locations, and the goal is to minimize the sum of the distances between matched elements plus the total delay, which is the time that requests are waiting to be served. The authors consider this general case as well as the bichromatic case, where requests are in two groups (for instance, Uber drivers and customers). The paper focuses on concave delay functions. That is, it is not the actual delays that go into the objective function but rather the valuations that the participants have for these delays. It is reasonable to consider concave delays since once you have been waiting for half an hour, an additional five minutes is not as bad as at the start.

If delays are linear and requests all arrive at one location, a simple greedy matching is optimal, but with concave delays you need to be much more careful. The paper first describes how concave functions can be approximated using piecewise linear functions and then uses careful bookkeeping to match requests in a suitable manner. In particular, a pair of requests that are currently experiencing the same linear delay function are always matched.

This gives an $O(1)$-competitive algorithm. For multiple locations, an intricate argument using hierarchical spanning trees shows $O(\log n)$-competitiveness; this is not "simply" a random embedding of a general metric into an HST, but a more involved construction.

The same results are then shown to hold for the bichromatic case using appropriately adapted algorithms.

Thomas Kesselheim et al. [76] presented an optimal e-competitive algorithm for weighted bipartite matching back in ESA 2013. Assuming bidders arrive in a random order, this problem can be understood in terms of the secretary problem with bidders bidding for items. As is common for the secretary problem, this algorithm starts with a sampling phase and then assigns items to the arriving bidders based on a local optima that considers all known bidders and items.

In SODA 2019, Rebecca Reiffenhauser [98] extends this approach to a truthful mechanism. This mechanism considers only unassigned items and instead of using the valuation of the bidder, prices are determined using a Vickrey-Clarke-Groves mechanism, which is designed to ensure truthfulness.

2.5 Online coloring

In the vertex coloring problem, the vertices of a graph must be colored in such a way that no two adjacent nodes have the same color. Similarly, in the edge coloring problem, adjacent edges must be colored with different colors. Trying to minimize the amount of colors used is quite challenging.

There is an important connection between vertex coloring and the buffer management problem with conflicts which is explained below.

The weighted fractional chromatic number in a graph G is the optimal objective value of the following linear program:

$$\begin{aligned}
\text{minimize} \quad & \chi(G,h) = \sum_{I} x_I \\
\text{subject to} \quad & \sum_{I \ni i} x_I = h_i \\
& x_I \geq 0
\end{aligned}$$

Here I denotes an independent set in the graph and h is a vector of weights with one weight given to each node of the graph. By setting all weights to one we get the fractional chromatic number, and if we additionally require x_I to be integral we get the chromatic number which is the minimum number of colors needed to color the vertices of the graph.

The fractional weighted chromatic number is also the minimum time required to remove load h_i from each node in the graph. Here x_I can be seen as the time independent set I runs for in the schedule.

In the online version of vertex coloring, the vertices arrive one by one (together with all the edges adjacent to the already presented vertices) and must be assigned a color immediately and irrevocably. For the general vertex coloring problem there is no hope for any algorithm to be constant competitive as no algorithm can be better than $o(n/\log^2 n)$-competitive. Therefore it is common to study particular graph classes, and we begin this section on online coloring with a result on unit interval graphs.

We then proceed with a variety of results on edge coloring which has been the more active area of research.

Joanna Chybowska-Sokól et al. [34] work on the online graph coloring problems on the intersection graphs of intervals. Here no two overlapping intervals can be of the same color. If all intervals have length between 1 and σ, the problem is called σ-interval coloring. The authors show that σ-interval coloring is strictly easier than interval coloring, and σ-interval coloring is strictly harder than unit interval coloring (where σ is 1).

They present an algorithm that for every $\sigma \in$ with $\sigma \geq 1$ is $1+\sigma$-competitive regarding the asymptotic competitive ratio. Their algorithm constructs small and large blocks (intervals) and then uses counters for these blocks to determine the color for an incoming interval. For $\sigma < 2$, the competitive ratio is less than 3, whereas Kierstead and Trotter [77] showed that there cannot exist a less-than-3-competitive algorithm for (general) interval coloring.

For different values of σ, the authors present corresponding lower bounds. They begin with showing that for every $\sigma > 1$ there is no online algorithm for σ-interval coloring with an asymptotic competitive ratio smaller than $5/3$. Then they improve the lower bound to $7/4$ for every $\sigma > 2$. Their main result regarding lower bounds is that for every $\varepsilon > 0$ there is $\sigma \geq 1$ such that there is no online algorithm with an asymptotic competitive ratio $5/2 - \varepsilon$. This implies that there is no 2-competitive algorithm for σ-interval coloring (for $\sigma > 2^{78}$). For unit interval graphs, the simple greedy FIRSTFIT algorithm is 2-competitive [44].

Vizing's famous theorem on edge coloring states that any graph of maximum degree Δ is $\Delta + 1$-colorable. Furthermore, Bar-Noy, Naor and Motwani [11] showed that the trivial greedy algorithm which uses $2\Delta - 1$ colors is optimal among online algorithms for $\Delta = O(\log n)$. This leaves a conjecture that there is an $(1 + o(1))\Delta$-edge-coloring algorithm for $\Delta = \omega(\log n)$. Cohen et al. in FOCS 2019 [36] resolve this conjecture for one-sided adversarial vertex arrivals in bipartite graphs. The classic fractional relaxation asks to minimize $\sum_M x_M$ subject to $\sum_{e \in M} x_e = 1$ for every edge e and $x_M \geq 0$ for every matching M. This relaxation fractionally uses integral matchings to cover each edge. Cohen et al. deploy a novel relaxation that integrally uses fractional matchings to cover each edge, i.e., $\sum_c x_{e,c} = 1$ and the goal is to minimize the number of non-zero fractional matchings used. This relaxation is a lot more useful in the online setting. They then provide an optimal online fractional edge coloring algorithm. If Δ is known beforehand, setting $x_{e,c} = 1/\Delta$ provides a trivial 1-competitive solution. For unknown Δ, they use an adaptation of the greedy "water-filling" algorithm. They use an asymmetric approach where colors are picked only based on the load of the offline vertex. Furthermore, they introduce a constraint on the value of each edge-color pair. This results in a more balanced allocation compared to the base water-filling-algorithm. Finally, they present a near-lossless online rounding scheme. Given an α-competitive fractional online algorithm, a natural approach would be to repeatedly round the $\alpha\Delta$ fractional matchings online to obtain $\alpha\Delta$ integral matchings. The remaining uncolored edges require further $(o(\Delta))$ colors. The problem with this approach is that maximum-degree vertices will be matched $\approx \Delta$ times, however, there is a chance that these matches are along previously colored edges. Instead, Cohen et al. repeatedly round subsets of multiple fractional matchings. This subset has to be small enough not to decrease the chance of a node being matched along an uncolored edge (due to rematches), but big enough to argue that the degree of all high degree vertices decreases at a rate of $\approx 1/\alpha$ per color used. This approach yields the desired result of a $(1 + o(1))\Delta$-edge-coloring algorithm for $\Delta = \omega(\log n)$ in bipartite graphs. This is matched by a tight lower bound.

The result by Cohen et al. leaves open the question whether the competitive ratio of 2 of the greedy algorithm is optimal under general vertex arrivals, in general graphs and in ICALP 2021, Amin Saberi and David Wajc [100] pick up the problem again. For an input of an (unknown) bipartite graph G and a fractional matching x in G, they present a rounding scheme algorithm (inspired by results in [93]) that outputs a random matching M in which each edge $e \in E$ belongs to the matching with probability at least $0.527 \cdot x_e$.

One result in [36] implies that α-competitive edge coloring can be reduced to online matching under the following conditions: If an online matching algorithm for any (bipartite) graph of max degree $\Delta \leq \Delta'$ under vertex arrivals can match each edge with high enough probability (at least $1/(\alpha\Delta')$), then there exists a $(\alpha + o(1))$-competitive (w.h.p) edge coloring algorithm for (bipartite) graphs of max degree $\Delta = \omega(\log n)$ under vertex arrivals. Their proposed rounding scheme satisfies the condition with $\alpha = 1/0.527 \leq 1.9$. They then combine this result with a result implied by Howard Karloff and David Shmoys [74]. If one can α-competitively and with high probability color edges online on (bipartite) graphs of max degree $\Delta = \omega(\log n)$ under interleaved vertex arrivals, then one can also $(\alpha + o(1))$-competitively with high probability color edges online on general graphs of maximum degree $\Delta = \omega(\log n)$ under vertex arrival.

Putting all this together, the authors present a $(1.9 + o(1))$-competitive online edge coloring algorithm for general graphs of maximum degree $\Delta = \omega(\log n)$ under vertex arrival (the term $o(1)$ is sufficiently small to imply a competitive ratio better than 2). Hence they resolve the long standing conjecture in the affirmative and show that the greedy algorithm is indeed not optimal for general online edge coloring.

In SODA 2021, Sayan Bhattacharya et al. [18] consider online edge coloring in the random order model and with recourse. In both models, they present algorithms that use at most $(1 + o(1))\Delta$ colors, where Δ is the maximum degree of the graph, which is presumed to be known in advance. The algorithm is based on a distributed offline algorithm called the Nibble method. Nibble selects a random ε fraction of the incident edges of each vertex in each round. Each edge is given a tentative color uniformly at random from the set of colors which is not yet used by incident edges. If an edge receives the same tentative color as an incident edge, it is marked failed and left uncovered in this round. Since only few edges tend to fail in each round, after some number of rounds the remaining edges can be colored greedily. The authors adapt this algorithm by not attempting to recolor edges that fail (because an incident edge received the same color), removing a color from the set of allowed colors of incident edges even if a coloring fails and sampling each edge independently in each round. These changes allow for a simplified analysis and also enable the algorithm to be adapted to the online settings considered in this paper.

Chapter 3

Buffer management with conflicts

3.1 Introduction

Scheduling is a fundamental problem in computer science. It played an important role in the development of approximation and online algorithms. One of the earliest online algorithms was designed for makespan scheduling in the 1960s [53]. Many objective functions for online scheduling have been studied over the years. In the well-known area of throughput scheduling, the goal is to maximize the weighted or unweighted number of completed jobs (the throughput). Each job has a deadline, and we only count the jobs that complete by their deadlines [102].

This chapter is based on a series of two papers [61, 63]. The first was presented 2020 in the FAW-conference. In this paper we consider a variation of the scheduling problem in which jobs do not have deadlines. Instead, jobs that arrive are stored in a buffer of some size until they can be processed. The buffer minimization in multiprocessor systems with conflicts or simply *buffer minimization problem* was first introduced by Chrobak et al. [33]. A sequence of tasks needs to be scheduled in a multi-processor system with conflicts. A conflict occurs if two processors share a common resource that they cannot both access at the same time. The multi-processor system is modelled as an undirected graph where the processors are the nodes of the graph. They are in conflict if they are connected by an edge, i.e. adjacent nodes on the graph may not run at the same time. Without loss of generality, the conflict graph is connected.

In this paper we consider the special case where this graph is a path. At any time tasks may arrive on a processor adding to this processor's workload. Each processor stores its workload in a separate input buffer. The goal is to find a scheduling strategy that minimizes the maximum used buffer size of all processors.

An online algorithm processes the workload without knowledge of future tasks. We measure the performance of such algorithms using the competitive ratio. For the present problem, that is the ratio between the maximum used buffer size of an online algorithm and the buffer size of an optimal offline algorithm in the worst case. Chrobak et al. [33] provide results for several kinds of graphs, including K_n and trees as well as graphs with up to four vertices. In particular, they show that GREEDY is $\frac{5}{2}$-competitive on the graph that has four vertices on a path, and they give a lower bound of 2 for the path with three vertices. In each time instant, GREEDY iteratively picks the highest available machine and runs that machine (making its neighbors unavailable). In the case of two adjacent machines having the same load GREEDY rapidly switches between them. This can also be seen as both machines running at the same time at half speed.

This paper is followed up in *Theoretical Computer Science* in 2021. In this second paper we introduce the flow model. In the original model, blocks of load can arrive at any time, with size up to the offline buffer size. In contrast, in the flow model load only arrives incrementally at a rate of at most 1. Clearly, any algorithm for the original model can also be used in the flow model. However, in many cases it is possible to do better. We provide modified algorithms which are optimal for the path with up to five machines.

We assume that the size of the offline buffer is known and we scale it to 1. Chrobak et al. [33] also consider the case where the offline buffer is not known and show that the competitive ratio in that case is at most a factor of 4 higher.

We give an example using GREEDY in the flow model. Assume there are three machines 1, 2 and 3 which are empty at time $t = 0$. For one unit of time, load arrives at a rate of 1 on machines 1 and 2. GREEDY runs both machines at half speed. At time $t = 1$, GREEDY has load $\frac{1}{2}$ on both machines 1 and 2, and no load on machine 3. Assume the optimal offline algorithm runs machine 2 at full speed, that is at a processing rate of 1, for that unit of time. Then at time $t = 1$ the optimal offline algorithm has one unit of load on machine 1, and its other machines are empty.

For the next unit of time, load arrives at a rate of 1 on machine 3. Since the highest available machines are 1 and 2 and they have the same load, GREEDY initially runs all machines at speed $\frac{1}{2}$. This means the loads on machines 1 and 2 decrease at a rate of $\frac{1}{2}$ while the load on machine 3 *increases* at a rate of $\frac{1}{2}$. After $\frac{1}{2}$ unit of time, all machines of GREEDY have load $\frac{1}{4}$. Since machine 3 is still receiving load, GREEDY runs machines 1 and 3 at full speed for the remainder of this timestep. At time $t = 2$, GREEDY has no load on machine 1 and $\frac{1}{4}$ load on machine 2 and 3. The optimal offline algorithm can run machines 1 and 3 at full speed for the entire unit of time, therefore its machines are empty at time $t = 2$.

The goal of this chapter is to coalesce the results for both versions of the problem. Thus for each subsection we first give the results for the original model and then add the results for the flow model. For the path with 2 and 3 machines, more emphasis is given to the flow model since the results for the original model were already known in these cases. For paths with more machines we emphasize the original model and then explain the changes made for the flow model.

3.1.1 Our results

We provide lower and upper bounds on the competitive ratios on the paths with four and five machines for both the original model as well as the flow model. For the path with four machines the ratio $\frac{9}{4}$ is optimal in the original model, while in the flow model we find optimal ratios for both the path with four and five machines and these are $\frac{4}{3}$ and $\frac{3}{2}$.

We also show that in the flow model GREEDY is 1-competitive on two machines and give a 1-competitive algorithm on three machines. Finally, for large m, we show that no algorithm can be better than $\frac{12}{5}$-competitive on m machines in the original model. The following table sums up our and previous bounds on the competitive ratio for different path lengths for both versions of the problem.

Number of machines	2	3	4	5	$m > 5$
Lower bound	3/2 [33]	2 [33]	9/4	16/7	12/5
Upper bound	3/2 [33]	2 [33]	9/4	5/2	$m/2 + 1/4$
Lower bound flow	1	1	4/3	3/2	3/2
Upper bound flow	1	1	4/3	3/2	$m/2 - 2/3$

Before this paper, the best known lower bound on general connected graphs (of any size) was only 2, whereas the best known upper bound is linear in the diameter of the graph (the length of the path, in our case). This upper bound (with a minor technical change) also applies to the flow model.

Additionally we consider a few results that go beyond the line graph. The first is an algorithm with a competitive ratio linear in the size of the graph that works on bipartite graphs.

Given resource-augmented machines that work at speed $1 + \chi\varepsilon$ there is a $\frac{1}{\varepsilon} + \chi + 2$-competitive algorithm on general graphs. Here χ is the fractional chromatic number of the graph.

Finally we consider the case where adjacent machines may work at a combined speed of 1. This type of processor sharing is always allowed on the path, since it can be achieved by rapidly switching between the machines. Whenever this type of processor sharing is allowed we get an $\frac{1}{\varepsilon} + 1$-competitive algorithm with machines working at speed $1 + 2\varepsilon$.

3.1.2 Related work

The problem was first introduced by Chrobak et al. [33] They show that the competitive ratio is finite for all graphs. Furthermore, they show that the competitive ratio for the complete graph K_n is exactly H_n, the nth harmonic number. For trees they achieve a competitive ratio of at most $\Delta/2 + 1$, where Δ is the tree diameter. They also give competitive ratios of GREEDY on small graphs including a competitive ratio of $\frac{5}{2}$ on the graph with 4 machines in a line.

A similar model is studied by Irani and Leung [70, 71], who also introduce a conflict graph. However, they study the conflict between jobs, not processors, and are minimizing the makespan. They show that even for paths no algorithm is better than $\Omega(n)$-competitive.

Bodlaender et al. [24] study a similar scenario in form of a two player game in which one player wants to keep a water-bucket system stable, whereas the other player wants to cause overflows. In their case the restriction is not that the player can only remove water from an independent set of buckets, instead the restriction is that the player can only empty subsets of consecutive buckets.

More recently, Gasieniec et al. [51] introduced the Bamboo Garden Trimming Problem, which is similar to our problem, but does not feature any conflicts. A robot may trim only one bamboo at the end of each day, while the other bamboos grow at some rate. The goal is to find a schedule that maintains the elevation of the garden as low as possible.

3.2 Definitions

We are given m machines on a line numbered from 1 to m. We denote the workload of a machine by a_i, thus the state of an online algorithm ALG at a given time t can be described by the vector $a = (a_1, ..., a_m)$. We also call this the configuration of the online algorithm at time t. We denote the state or configuration of the optimal offline algorithm OPT with the vector $z = (z_1, ..., z_m)$. Load vectors are never negative. We define the delay of a machine as $d_i := a_i - z_i$. Initially $a_i = z_i = 0$ for all i. Delays can be negative. Additionally to the workloads of machines, we also refer to arriving jobs as load, i.e. when a job arrives and the workload of that machine increases we may say load arrives on that machine.

Time runs continuously. In the original model, tasks may arrive at any time. The speed of a machine describes how fast the machine can reduce its workload. A machine that runs at speed s for a duration of length T will reduce its workload by sT if no load arrives during that timeperiod. The combined processing rate of two adjacent machines cannot exceed 1. This means two adjacent machines can share processing power both working at speed $\frac{1}{2}$ instead of just one of the machines working at speed 1 at any given time.

As discussed above, we introduce the flow model. In the flow model, load arrives on machines at a certain rate. When load arrives on a machine at a rate of r for a period of length T and that machine is not running, its workload increases by rT. Load arrives at a rate of at most 1.

The algorithms in this paper use the concept of groups of machines. A set S of adjacent machines is called a *group* if all machines in S have load and no machine adjacent to S has load.

We also consider general graphs. In this setting processor sharing is not always allowed. Instead machines run at speed 1 and two adjacent machines may not run at the same time. In the graph setting a *group* is the induced subgraph of a maximal contiguous set of machines that all have load.

3.2.1 Comparison between the flow model and the original model

As noted above, any algorithm for the original model can also be used in the flow model.

Any algorithm for the flow model can be used as a black box for the original model by using additional buffer space. This works as follows. We scale the input (here and in the rest of the thesis) so that the offline buffer has size 1. Incoming load is stored in reserved buffer space of size 1. Whenever there is load in reserved buffer space, this load is transferred at a rate of 1 to the rest of the buffer. The flow algorithm only sees and works on the load outside the reserved buffer space. This gives us an algorithm for the original model at an additive loss of 1 in the competitive ratio. However, as can be seen from the table, in several cases the difference between the flow model and the original model is strictly less than 1 in terms of the competitive ratio that can be achieved.

The flow model has the advantage that it leads to a simpler analysis in many cases, and also allows for more tight results.

3.3 The path with two and three machines

For the original model, the exact competitive ratio on the paths with 2 and 3 machines were given by Chrobak et al. For the path with two machines the ratio is 3/2 and for the path with three machines it is 2.

For the flow model we show that GREEDY is 1-competitive on the path with two machines and give an algorithm on three machines that is also 1-competitive.

GREEDY always runs the machine with more load. If the machines have the same load, it runs both at half speed.

Theorem 1. GREEDY *is* 1*-competitive on the path with two machines in the flow model.*

Proof. In the flow model it is seen easily that both machines of GREEDY have the same load at all times. If load arrives only on one machine, the algorithm runs that machine at full speed and the loads stay equal. Otherwise both machines run at half speed if they have load and decrease or increase at the same rate.

Furthermore, GREEDY never has more overall load than OPT, because the only case in which OPT could process more load than GREEDY is if GREEDY's machines are empty. The overall load of OPT and thus GREEDY is at most 2, in which case both machines of GREEDY have load 1. □

Theorem 2. *Algorithm 1 is 1-competitive on the path with three machines in the flow model.*

Proof. We show that $d_i + d_{i+1} \leq 0$ always holds for $i = 1, 2$. Whenever there is a running machine in a pair, the invariant is maintained. The only situation

Algorithm 1 (for three machines)

1. Run any group of size 1 at full speed.
2. If there is a group of size 2, run the middle machine, unless the other machine with load has load 1 and is receiving load.
3. If all machines have load, run the two outer machines, unless the middle one has load 1 and is receiving load.

in which $d_i + d_{i+1}$ (w.l.o.g. $i = 2$) can increase is if machines 1 and 2 have load, machine 2 is receiving load but not running and machine 3 is empty. In that case, machine 1 is running, which means machine 1 has load 1 and is receiving load. Thus $d_1 \geq 0$ and because the invariant holds on machines 1 and 2, $d_2 \leq 0$. This means the invariant also holds on machines 2 and 3. The only cases in which the algorithm does not run a machine are if either the machine has load less than 1 or the machine has a neighbor with load 1 that is running. In the second case, that machine has delay at most 0 due to the invariants and therefore load at most 1. □

3.4 The competitive ratio of the path with four machines

3.4.1 ...for the original model

In this section we determine the competitive ratio of the path with four machines. We begin by presenting a $\frac{9}{4}$-competitive algorithm:

Lemma 3. *For Algorithm 2, the following invariants hold at all times.*

(I1) $d_1 + d_2 \leq \frac{3}{4}$, (I2) $d_2 + d_3 \leq \frac{1}{2}$, (I3) $d_3 + d_4 \leq \frac{3}{4}$, (I4) $d_1 + d_2 + d_3 + d_4 \leq 1$

Proof. Whenever all machines are empty, all invariants hold. When load arrives the invariants are obviously preserved. We consider task execution.

Observation 1. *If* ALG *(any algorithm) runs a machine at full speed, invariants (I1) to (I3) involving this machine are maintained.*

ALG reduces the load on that machine by ε during the ε-step, while OPT reduces the load of any two adjacent machines by at most ε, thus the invariant is maintained.

Observation 2. *Whenever any two machines are running, (I4) is maintained.*

ALG reduces the overall load by 2ε, while OPT can increase the overall delay by at most 2ε, therefore (I4) is maintained.

Algorithm 2

1. Run any group of size 1 at full speed.
2. If there is a group of size 2 in the middle, meaning $a_1 = a_4 = 0$, run the machine with higher load.
3. Else if there is a group of size 2 at the edge, run the more central machine, unless the other machine has load more than $\frac{5}{4}$. In that case run the other one.
4. If there is a group of size 3, run two machines unless the middle one has load more than $\frac{5}{4}$. In that case run the middle one.
5. If all machines have load, run the best parity (the parity that includes the machine with highest load), unless both outer machines have load more than $\frac{5}{4}$. In that case run the outer machines.

We show that when executing tasks, all invariants are maintained at all times. We consider all possible configurations of the load vector. In every situation some invariants are maintained because of the observations above. Given these invariants we then show that the remaining invariants also hold in that situation and are therefore also maintained.

If only one machine i of Algorithm 2 has load, Algorithm 2 runs machine i at full speed and invariants (I1) to (I3) that involve machine i are maintained by Observation 1. The invariants that do not involve machine i hold, because all other machines are empty, and therefore the delay on those machines is at most 0. Additionally, since (I1) to (I3) hold, it follows that if only machine i has load, we have $d_i \leq \frac{3}{4}$. The other machines are empty, therefore $d_j \leq 0$ for $i \neq j$. Thus (I4) holds in the case of only one machine having load, due to (I1) to (I3) being maintained in this case.

Suppose two machines have load. If there are two groups of size 1, (I4) is maintained by Observation 2. For the other invariants (I1) to (I3) either one of the relevant machines is running, and Observation 1 applies, or both machines are empty, which means the delay on these machines is at most 0. Thus all invariants are maintained.

We next show that (I1) to (I3) are maintained whenever there are groups of size 2.

Suppose $a_1 > 0$ and $a_2 > 0$ (and $a_3 = 0$). If $a_1 \leq \frac{5}{4}$, machine 2 is running and (I1) and (I2) are maintained by Observation 1. If $a_1 > \frac{5}{4}$, this means $d_1 \geq \frac{1}{4}$. Since machine 1 is running, (I1) is maintained by Observation 1. With (I1) it follows that $d_2 \leq \frac{1}{2}$. Since $a_3 = 0$ and therefore $d_3 \leq 0$ we get $d_2 + d_3 \leq \frac{1}{2}$. Thus (I2) also holds. (I3) holds, because either $a_3 = a_4 = 0$ and thus $d_3 + d_4 \leq 0$ or $a_3 = 0$ and $a_4 > 0$ and (I3) is maintained because machine 4 is running and Observation 1 applies. The case $a_3 > 0$ and $a_4 > 0$ is symmetrical to the one above.

Suppose there is a group of size 2 in the middle. This means $a_2 > 0$ and $a_3 > 0$. Algorithm 2 runs the machine with more load. If machine 2 is running, (I1) and (I2) are maintained by Observation 1. If machine 3 is running, (I2) and (I3) are maintained by Observation 1. To prove that (I1) holds if machine 3 is running, assume $d_1 + d_2 > \frac{3}{4}$. Since $d_1 \leq 0$, $d_2 > \frac{3}{4}$. With (I2) $d_3 < -\frac{1}{4}$ follows. However from $d_2 > \frac{3}{4}$ we also get $a_2 > \frac{3}{4}$ and since $a_2 \leq a_3$, $a_3 > \frac{3}{4}$ and thus $d_3 > -\frac{1}{4}$. This is a contradiction to $d_3 < -\frac{1}{4}$. This means $d_1 + d_2 \leq \frac{3}{4}$ and (I1) holds. (I3) holds if machine 2 is running, because of symmetry.

Since (I1) to (I3) hold for groups of size 2, the combined delay of the machines in that group is at most $\frac{3}{4}$. If the other machines are empty, their delay is at most 0 and (I4) holds as well. If there is a group of size 1 and a group of size 2, all invariants are maintained by Observation 1 and Observation 2.

Suppose there is a group of size 3 and $a_4 = 0$. Either machines 1 and 3 or machine 2 are running, therefore (I1) and (I2) are maintained by Observation 1. If machines 1 and 3 are running (I3) and (I4) are maintained as well by Observation 1 and Observation 2. If machine 2 is running, this means $a_2 > \frac{5}{4}$, therefore $d_2 \geq \frac{1}{4}$. Adding (I1) and (I2) yields $d_1 + 2d_2 + d_3 \leq \frac{5}{4}$. Since $d_2 \geq \frac{1}{4}$ and $d_4 \leq 0$ we get $d_1 + d_2 + d_3 + d_4 \leq 1$. Therefore (I4) is also maintained, if machine 2 is running. It remains to show that (I3) is maintained for machine 2 running. Since $a_2 > \frac{5}{4}$, it follows that $d_2 \geq \frac{1}{4}$. Since (I2) is maintained, $d_3 \leq \frac{1}{4}$. Machine 1 is empty, therefore $d_4 \leq 0$ and (I3) holds. The case $a_1 = 0$ is symmetrical to the previous one.

Finally, suppose all machines have load. In all cases there are two machines running, therefore (I4) is maintained by Observation 2. If machines 1 and 4 are not running at the same time, two machines of the same parity are running and all invariants are maintained by Observation 1. Machines 1 and 4 run at the same time if $a_1 > \frac{5}{4}$ and $a_4 > \frac{5}{4}$. This means $d_1 \geq \frac{1}{4}$ and $d_4 \geq \frac{1}{4}$. From (I4) we get $d_2 + d_3 \leq \frac{1}{2}$, thus (I2) holds. This means all invariants hold at all times. □

Theorem 4. *Algorithm 2 is $\frac{9}{4}$-competitive on the path with four machines.*

Proof. We first show that $d_1 < \frac{5}{4}$ holds at all times. Whenever $a_1 < \frac{5}{4}$, $d_1 < \frac{5}{4}$ holds. If $a_1 > \frac{5}{4}$ and machine 1 is running, d_1 does not increase. We only need to consider the case where $a_1 > \frac{5}{4}$ and machine 1 is not running. In this case $a_2 > \frac{5}{4}$ (Rules 4 and 5 of Algorithm 2). It follows that $d_2 > \frac{1}{4}$ and therefore $d_1 < \frac{1}{2}$ by (I1). This means $d_1 < \frac{5}{4}$ holds at all times.

We now show that $d_2 < \frac{5}{4}$. Again we only need to consider the case where $a_2 > \frac{5}{4}$, but machine 2 is not running. In this case $a_1 > \frac{5}{4}$ or $a_3 > \frac{5}{4}$ (Rule 3 and 5 of Algorithm 2). It follows that $d_1 > \frac{1}{4}$ and therefore $d_2 < \frac{1}{2}$ by (I1) or $d_3 > \frac{1}{4}$ and therefore $d_2 < \frac{1}{4}$ by (I2). This means $d_2 < \frac{5}{4}$ holds at all times.

By symmetry the delay on each machine i is at most $d_i = a_i - z_i \leq \frac{5}{4}$. Adding $z_i - 1 \leq 0$ yields $a_i \leq \frac{9}{4}$. Therefore the algorithm is $\frac{9}{4}$-competitive on the path with four machines. □

We now present a matching lower bound of $\frac{9}{4}$ on the competitive ratio of the path with four machines.

Lemma 5. *Let the competitive ratio of* ALG *be equal to* $2 + x$. *Whenever* $z_i = 0$ *is possible we have* $a_i \leq 1 + x$.

Proof. Otherwise one unit of load arriving on machine i would yield a ratio higher than $2 + x$. □

Lemma 6. *If* ALG *has a delay of x on two adjacent machines, we can reach a state where* ALG *has load at least x on those machines and all machines of* OPT *are empty.*

Proof. Let the two adjacent machines be i and $i + 1$. No load arrives until OPT has removed its load on machines i and $i + 1$. Since there was a delay of x on those machines, there is still at least x load left once OPT's machines i and $i + 1$ are empty. Then load arrives on machine i and OPT runs all machines that have the same parity as i. After one timestep, this parity is empty for OPT, as it has load at most 1 on each machine at all times, and the total load of ALG on i and $i + 1$ has not changed. Repeat this for the other parity. □

Theorem 7. *No algorithm can be better than $\frac{9}{4}$-competitive on the path with four machines.*

Proof. Let the competitive ratio of ALG be equal to $2 + x$. Suppose that $x < \frac{1}{4}$. We start with both OPT's and ALG's machines empty. We present two phases, consisting of repeating sets of similar inputs. The first phase creates an initial delay on the middle two machines. Based on that, the second phase forces a competitive ratio that is unbounded for $x < \frac{1}{4}$, which contradicts the assumption.

Phase 1: Repeating the following set of inputs forces a total load of (almost) $1 - 2x$ on the middle two machines of ALG in the limit, with OPT's machines being empty.

Let $L = a_2 + a_3$ be the combined load of the middle two machines before the first time step. Load arrives after every timestep i. We denote the time right before load arrives with i^- and the time right after with i.

Time 0: At time 0, one unit of load arrive on machines 2 and 3 each. Thus a_2 and a_3 increase by 1 each, while during the first timestep $a_2 + a_3$ decreases by at most 1. This means at time 1^-, $a_2 \geq \frac{L+1}{2}$ or $a_3 \geq \frac{L+1}{2}$. W.l.o.g., let $a_2 \geq \frac{L+1}{2}$. During this timestep we allow ALG to process all existing loads on machines 1 and 4 from previous iterations (which only makes it easier for ALG).

Time 1: One unit of load arrives on machine 1. At time 2^-, $a_1 + a_2 \geq \frac{L+1}{2}$ and OPT has configuration $(0, 0, 0, 0)$.

Time 2: One unit of load arrives on machines 1 and 2 each, while $a_1 + a_2$ decreases by at most 1 during the following timestep. This means $a_1 + a_2 \geq 1 + \frac{L+1}{2}$ at time 3^-.

Since a possible configuration of OPT at this time is $(0, 1, 0, 0)$, $a_1 \leq 1 + x$ by Lemma 5. This implies $a_2 \geq \frac{L+1}{2} - x$ at time 3^-.

TABLE 3.1: Phase 1: The notation i^- refers to the state at time i just before new jobs arrive, i refers to the state after the job arrivals. The boxed entries are lower bounds for two adjacent machines. Those marked with $\leq$ are upper bounds.

	Loads of ALG				Loads of OPT				Next input
Time\Machines	1	2	3	4	1	2	3	4	
0^-	...	$\boxed{L}$		...	0	0	0	0	0110
0	...	$\boxed{L+2}$		...	0	1	1	0	
1^-	0	$\boxed{L+1}$		0	0	0	1	0	1000
1	1	$\boxed{L+1}$		0	1	0	1	0	
2^-	$\boxed{\frac{L+1}{2}}$		...	0	0	0	0	0	1100
2	$\boxed{\frac{L+5}{2}}$		...	0	1	1	0	0	
3^-	$\leq 1+x$	$\boxed{\frac{L+1}{2}-x}$		0	1	0	0	0	0010
3	$\leq 1+x$	$\boxed{\frac{L+3}{2}-x}$		0	1	0	1	0	
4^-	x	$\boxed{\frac{L+1}{2}-x}$		0	0	0	0	0	0110

Time 3: One unit of load arrives on machine 3. Then at time 4^-, $a_2 + a_3 \geq \frac{L+1}{2} - x$ while the machines of OPT are again empty. We can now repeat the input if desired.

Repeating these four timesteps, we get a sequence $(L_n)_n$ with $L_{n+1} \geq \frac{L_n+1}{2} - x$.

As long as $L_n \leq 1 - 2x$, we have

$$1 - 2x - L_{n+1} \leq \frac{1}{2}(1 - 2x - L_n)$$

Since $L_0 = 0 \leq 1 - 2x$ we find $1 - 2x - L_{n+1} \leq \frac{1}{2^n}(1 - 2x)$, so $L_{n+1} \geq 1 - 2x - \frac{1}{2^n}(1 - 2x)$.

Phase 1 continues until $L_n > 6x - 1$. This is possible because $x < \frac{1}{4}$ and therefore $1 - 2x > 6x - 1$.

Phase 2: Again let L be the load on the middle two machines. Let B be the load on all machines. We reset time to 0. This means after phase 1 we start phase 2 with a load of $L_0 > 6x - 1$.

Time 0: At time 0, one unit of load arrives on machines 2 and 3 each. During this timestep ALG can reduce $a_2 + a_3$ by 1. We again allow ALG to process all existing loads on machines 1 and 4 from previous iterations. For the total load at time 1^- (before new jobs arrive) we have

$$B \geq L + 1$$

with one of the machines (w.l.o.g. machine 2) having at least load $\frac{L+1}{2}$.

TABLE 3.2: Phase 2: The entries for machines 2 and 3 of ALG and the total load are lower bounds. Those marked with $\leq$ are upper bounds.

	Loads of ALG			Loads of OPT				Total load	Next input
Time \Machines	1	2 and 3	4	1	2	3	4		
0^-	...	L	...	0	0	0	0	L	0110
0	...	$L+2$	...	0	1	1	0	$L+2$	
1^-	0	$L+1$	0	0	0	1	0	$L+1$	1000
1	1	$L+1$	0	1	0	1	0	$L+2$	
2^-				0	0	0	0		1110
2				1	1	1	0		
3^-				0	1	0	0	$\frac{3}{2}L-x+\frac{3}{2}$	1011
3				1	1	1	1	$\frac{3}{2}L-x+\frac{9}{2}$	
4^-				$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{3}{2}L-x+\frac{5}{2}$	
4									1 and 4 continuous
5^-	$\leq 1+x$	$\frac{3}{2}L-3x+\frac{1}{2}$	$\leq 1+x$	1	0	0	1	$\frac{3}{2}L-x+\frac{5}{2}$	0110
...	...	$\frac{3}{2}L-3x+\frac{1}{2}$	...	0	0	0	0		0110

Time 1 and 2: At time 1, one unit of load arrives on machine 1. After this step OPT's machines are empty. Then load arrives on machines 1, 2 and 3. At time 3^-, OPT has configuration (0,1,0,0), however (1,0,1,0) is also a possible configuration. Because (1,0,1,0) is a possibility, $a_2 \leq 1+x$ at time 3 by Lemma 5. During these two timesteps, ALG therefore has to run machine 2 (alone, since machine 4 is empty) for a duration of at least $\frac{L+1}{2}+1-(1+x)=\frac{L+1}{2}-x$. ALG can then run machines 1 and 3 for a duration of at most $2-(\frac{L+1}{2}-x)=-\frac{1}{2}L+x+\frac{3}{2}$. This decreases the overall load by at most twice that amount. (It is less if machine 3 becomes empty.) For the total load at time 3^- we have $B \geq L+5-(\frac{L+1}{2}-x)-2\cdot(-\frac{1}{2}L+x+\frac{3}{2})=\frac{3}{2}L-x+\frac{3}{2}$.

Time 3: At time 3, one unit of load arrives on machines 1, 3 and 4 each. This means all machines of OPT have load 1. OPT then uses processor sharing to equally decrease the load on all machines, ending in state $(\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2})$ at time 4^-. However it is also possible for OPT to be in state (0,1,1,0) at time 4^-. At time 3, three units of load arrive, but ALG can only remove two in one time unit. This means

$$B \geq \frac{3}{2}L - x + \frac{5}{2}. \tag{3.1}$$

at time 4^-.

Time 4: Then, *during* timestep 5, one unit of load arrives on machines 1 and 4. This happens continuously at a constant speed of 1 over the whole timestep. OPT continues to use processor sharing, ending in the state (1,0,0,1). However, ALG needs to take into account that if OPT was in state (0,1,1,0) at time 4^-, OPT could have continuously run machines 1 and 4 during timestep 5. In this case it would still be in state (0,1,1,0) at time 5^-.

The overall load added during this timestep is 2. This means at time 5^- we still have (3.1). Because the state (0,1,1,0) is possible for OPT, $a_1 \leq 1+x$

and $a_4 \leq 1 + x$ by Lemma 5. The rest of the load is on the middle two machines. Given our bound on B, the middle two machines have a load of at least

$$B - 2 \cdot (1 + x) \geq \frac{3}{2}L - 3x + \frac{1}{2}. \tag{3.2}$$

By Lemma 6 we can now reach a state where the middle two machines have load at least $\frac{3}{2}L - 3x + \frac{1}{2}$ and all of OPT's machines are empty.

We can now repeat this phase if desired.

Let $\varepsilon_n = L_n - (6x - 1)$. We show that $\varepsilon_{n+1} \geq \frac{3}{2}\varepsilon_n$. Since $L_{n+1} \geq \frac{3}{2}L_n - 3x + \frac{1}{2}$, we have

$$\varepsilon_{n+1} - \varepsilon_n = L_{n+1} - L_n = \frac{1}{2}L_n - 3x + \frac{1}{2} = \frac{1}{2}(6x - 1 + \varepsilon_n) - 3x + \frac{1}{2} = \frac{1}{2}\varepsilon_n,$$

hence $\varepsilon_{n+1} \geq \frac{3}{2}\varepsilon_n$. It follows that $L_n \geq L_0 + (\frac{3}{2})^n\varepsilon_0$. Since $\varepsilon_0 > 0$, this implies $(L_n)_n$ grows without bound, contradicting the assumption that the competitive ratio is $2 + x$. □

3.4.2 ...for the flow model

In this section we determine the competitive ratio of the path with four machines for the flow model.

Algorithm 3 (for four machines)

1. Run any group of size 1 at full speed.
2. If $a_1 = a_4 = 0$, run the machine with higher load.
3. Else if there is a group of size 2 at the edge, run the more central machine, unless the outside machine has load (at least) $\frac{4}{3}$ and is receiving load. In that case run the outside machine.
4. If there is a group of size 3, run the two outer machines of the group unless the middle one has load $\frac{4}{3}$ and is receiving load. In that case run the middle one.
5. If all machines have load, run the best parity (even or odd), that is the parity that includes the machine with highest load, unless both outer machines have load $\frac{4}{3}$. In that case run the outer machines.

Note that this algorithm is essentially the same as the one for the original model. We are simply changing the target value and rephrasing slightly to account for load arriving as flow. The same is true for the invariants. The values change but otherwise the proof remains the same.

Lemma 8. *For Algorithm 3, the following invariants hold at all times.*

(I1) $d_1 + d_2 \leq \frac{2}{3}$, (I2) $d_2 + d_3 \leq \frac{1}{3}$, (I3) $d_3 + d_4 \leq \frac{2}{3}$, (I4) $d_1 + d_2 + d_3 + d_4 \leq 1$

Proof. Whenever all machines are empty, all invariants hold. When load arrives the invariants are obviously preserved. We consider all possible configurations of the load vector. In every situation some invariants are maintained because of the observations above. Given these invariants we then show that the remaining invariants also hold in that situation and are therefore also maintained.

If only one machine i of Algorithm 3 has load, Algorithm 3 runs machine i at full speed and invariants (I1) to (I3) that involve machine i are maintained by Observation 1. The invariants that do not involve machine i hold, because all other machines are empty, and therefore the delay on those machines is at most 0. Additionally, since (I1) to (I3) hold, it follows that if only machine i has load, we have $d_i \leq \frac{2}{3}$. The other machines are empty, therefore $d_j \leq 0$ for $i \neq j$. Thus (I4) holds in the case of only one machine having load, due to (I1) to (I3) being maintained in this case.

Suppose two machines have load. If there are two groups of size 1, (I4) is maintained by Observation 2. For the other invariants (I1) to (I3) either one of the relevant machines is running, and Observation 1 applies, or both machines are empty, which means the delay on these machines is at most 0. Thus all invariants are maintained.

We next show that (I1) to (I3) are maintained whenever there are groups of size 2.

Suppose $a_1 > 0$ and $a_2 > 0$ (and $a_3 = 0$). If $a_1 \leq \frac{4}{3}$ and is not receiving load, machine 2 is running and (I1) and (I2) are maintained by Observation 1. If $a_1 = \frac{4}{3}$ and is receiving load, this means $d_1 \geq \frac{1}{3}$. Since machine 1 is running, (I1) is maintained by Observation 1. With (I1) it follows that $d_2 \leq \frac{1}{3}$. Since $a_3 = 0$ and therefore $d_3 \leq 0$ we get $d_2 + d_3 \leq \frac{1}{3}$. Thus (I2) also holds. (I3) holds, because either $a_3 = a_4 = 0$ and thus $d_3 + d_4 \leq 0$ or $a_3 = 0$ and $a_4 > 0$ and (I3) is maintained because machine 4 is running and Observation 1 applies. The case $a_3 > 0$ and $a_4 > 0$ is symmetrical to the one above.

Suppose there is a group of size 2 in the middle. This means $a_2 > 0$ and $a_3 > 0$. Algorithm 3 runs the machine with more load. If machine 2 is running, (I1) and (I2) are maintained by Observation 1. If machine 3 is running, (I2) and (I3) are maintained by Observation 1. To prove that (I1) holds, assume $d_1 + d_2 > \frac{2}{3}$. Since $d_1 \leq 0$, $d_2 > \frac{2}{3}$. With (I2) $d_3 < -\frac{1}{3}$ follows. However, from $d_2 > \frac{2}{3}$ we also get $a_2 > \frac{2}{3}$ and since $a_2 \leq a_3$, $a_3 > \frac{2}{3}$ and thus $d_3 > -\frac{1}{3}$. This is a contradiction to $d_3 < -\frac{1}{3}$. This means $d_1 + d_2 \leq \frac{2}{3}$ and (I1) holds. (I3) holds if machine 2 is running, because of symmetry.

Since (I1) to (I3) hold for groups of size 2, the combined delay of the machines in that group is at most $\frac{2}{3}$. If the other machines are empty, their delay is at most 0 and (I4) holds as well. If there is a group of size 1 and a group of size 2, all invariants are maintained by Observation 1 and Observation 2.

Suppose there is a group of size 3 and $a_4 = 0$. Either machines 1 and 3 or machine 2 are running, therefore (I1) and (I2) are maintained by Observation 1. If machines 1 and 3 are running (I3) and (I4) are maintained as well by Observation 1 and Observation 2. If machine 2 is running, this means $a_2 = \frac{4}{3}$, therefore $d_2 \geq \frac{1}{3}$. Adding (I1) and (I2) yields $d_1 + 2d_2 + d_3 \leq 1$. Since $d_2 \geq \frac{1}{3}$

and $d_4 \leq 0$ we get $d_1 + d_2 + d_3 + d_4 \leq \frac{2}{3}$. Therefore (I4) is also maintained, if machine 2 is running. It remains to show that (I3) is maintained for machine 2 running. Since $a_2 = \frac{4}{3}$, it follows that $d_2 \geq \frac{1}{3}$. Since (I2) is maintained, $d_3 \leq 0$. Machine 1 is empty, therefore $d_4 \leq 0$ and (I3) holds. The case $a_1 = 0$ is symmetrical to the previous one.

Finally, suppose all machines have load. In all cases there are two machines running, therefore (I4) is maintained by Observation 2. If machines 1 and 4 are not running at the same time, two machines of the same parity are running and all invariants are maintained by Observation 1. Machines 1 and 4 run at the same time if $a_1 = \frac{4}{3}$ and $a_4 = \frac{4}{3}$ and both are receiving flow. This means $d_1 \geq \frac{1}{3}$ and $d_4 \geq \frac{1}{3}$. From (I4) we get $d_2 + d_3 \leq \frac{1}{3}$, thus (I2) holds. This means all invariants hold at all times. □

Theorem 9. *Algorithm 3 is $\frac{4}{3}$-competitive on the path with four machines*

Proof. There are two cases in which a machine i that has load does not get run by the algorithm. In the first case, the load of the machine is below $\frac{4}{3}$. In the second case, machine $i-1$ or $i+1$ has load $\frac{4}{3}$. Then that machine has delay at least $\frac{1}{3}$. Given the invariants, machine i then has delay at most $\frac{1}{3}$ in this case and thus load at most $\frac{4}{3}$. □

We now present a matching lower bound.

Theorem 10. *No algorithm can be better than $\frac{4}{3}$-competitive on the path with four machines.*

Proof. Let $R < \frac{4}{3}$.

Consider an R-competitive algorithm for four machines. Denote the load on machines 2 and 3 by L. At the beginning, $L = 0$.

We consider the following input. Load arrives for one time unit on machines 2 and 3. ALG then has at least $\frac{L+1}{2}$ load on one of those machines (w.l.o.g. it is machine 2), while OPT can be empty on machines 1 and 2. Then load arrives on machine 1 for one unit of time. After this, ALG has at least $\frac{L+1}{2}$ load on machines 1 and 2 and OPT's machines are empty. Load arrives on machines 1 and 2 for one unit of time and the overall load on that pair becomes $\frac{L+1}{2} + 1$. At most R of that load is on machine 1 and therefore at least $\frac{L+1}{2} + 1 - R$ is on machine 2. Finally, load arrives on machine 3 for one unit of time. After this, OPT's machines are empty, while ALG has at least $\frac{L+1}{2} + 1 - R$ load on machines 2 and 3. This means the load in the middle has become larger after processing this input as long as $L < \frac{L+1}{2} + 1 - R$. Since $R < \frac{4}{3}$ we can achieve $L > \frac{1}{3}$ given enough iterations.

We next consider the following input. Again we begin by forcing a load of $\frac{L+1}{2}$ on machine 2. Then load arrives on machines 1 and 2 for two units of time, forcing a load of at least $(\frac{L+1}{2} + 2)/2$ on either machine 1 or 2. Thus $R \geq (\frac{L+1}{2} + 2)/2$. With $L > \frac{1}{3}$ we have $R > \frac{4}{3}$ which is a contradiction. □

3.5 The competitive ratio of the path with five machines

3.5.1 ...for the original model

In this section we first present an algorithm that achieves a competitive ratio of $\frac{5}{2}$ and follow up with a lower bound of $\frac{16}{7}$.

Algorithm 4

1. Run any group of size 1 at full speed.
2. If there is a group of size 2 at the edge, run the more central machine, unless the other machine has load more than $\frac{3}{2}$.
3. Otherwise, if there is a group of size 2 run the more central machine, unless the other machine has load more than 1.
4. If there is a group of size 3, run two machines unless the middle one has load more than $\frac{3}{2}$.
5. We consider a group of size 4, e.g. machines 1 to 4. We run machines 2 and 4 unless $a_1 > \frac{3}{2}$ or $a_3 > \frac{3}{2}$. In that case we run machines 1 and 3.

   Additionally, if $a_1 > \frac{3}{2}$ and $a_4 > \frac{3}{2}$ we run machines 1 and 4. Machines 2 to 5 are symmetrical to machines 1 to 4.
6. If all machines have load, we run the odd parity, unless $a_2 > \frac{3}{2}$ or $a_4 > \frac{3}{2}$. If $a_2 > \frac{3}{2}$ we run the even parity as long as $a_5 \leq \frac{3}{2}$. If $a_2 > \frac{3}{2}$ and $a_5 > \frac{3}{2}$ we run machines 2 and 5. The case of $a_4 > \frac{3}{2}$ is symmetrical to $a_2 > \frac{3}{2}$.

Lemma 11. *Algorithm 4 maintains the following invariants at all times.*

1. $d_i \leq \frac{3}{2}$
2. $d_i + d_{i+1} \leq 1$ *for* $i = 1, 4$
3. $d_i + d_{i+1} \leq \frac{1}{2}$ *for* $i = 2, 3$
4. $d_i + d_{i+1} + d_{i+2} + d_{i+3} \leq 1$

Proof. Group of size 1: If there is a group of size 1, this machine is running and all invariants are maintained.

Group of size 2: Let there be a group of size 2 at the edge. W.l.o.g. we consider machines 1 and 2. Algorithm 4 runs the more central machine, unless the other one has load more than $\frac{3}{2}$. Thus since $a_1 \leq \frac{3}{2}$ and machine 2 is running, invariant 1 is maintained. Invariant 2 and 3 are also maintained, since every pair is either empty or has a running machine. Consider 4 adjacent machines, containing a group of size 2. Either there is a group of size 2 and 2 empty machines, in that case invariant 4 follows from invariant 2 or 3, or there is a group of size 2 and a group of size 1. In that case, two machines are running and invariant 4 is maintained.

If $a_1 > \frac{3}{2}$, we have $d_1 > \frac{1}{2}$ and thus $d_2 \leq \frac{1}{2}$. Since $d_3 \leq 0$ invariant 3 holds on machines 2 and 3. All other invariants clearly hold as well.

Otherwise let there be a group of size 2 not at the edge. W.l.o.g. these machines are machines 2 and 3. Algorithm 4 runs machine 3 if $a_2 \leq 1$. All invariants hold and in particular invariant 2 holds on machines 1 and 2, since $d_2 \leq 1$. If $a_2 > 1$ machine 2 runs instead. We have $d_2 > 0$ and thus $d_3 < \frac{1}{2}$ from invariant 3 on machines 2 and 3. Therefore invariant 3 als holds on machines 3 and 4. The other invariants clearly hold as well, in particular invariant 1 holds on machine 3, since $d_3 < \frac{1}{2}$.

Group of size 3: Let there be a group of size 3. Let the middle machine be numbered i. If $a_i \leq \frac{3}{2}$, Algorithm 4 runs machines $i-1$ and $i+1$. Since $d_i \leq \frac{3}{2}$ invariant 1 is maintained. Invariants 2 and 3 are clearly maintained as well. Invariant 4 is maintained, because 4 adjacent machines either contain 2 running machines, or 2 machines with load and a running machine and 2 empty machines. In the second case invariant 4 follows from invariant 3. If $a_i > \frac{3}{2}$, Algorithm 4 runs machine i instead. We have $d_i > \frac{1}{2}$. If $i-1$ is on the edge, we get $d_{i-1} \leq \frac{1}{2}$ from invariant 2 and invariant 1 is maintained. If $i-1$ is not on the edge, we get $d_{i-1} \leq 0$ from invariant 3. In that case invariant 1 is maintained, as well as invariant 3 on machines $i-1$ and $i-2$ (since $i-2$ is empty). Invariant 4 clearly holds as well. The same can be done for $i+1$. Thus all invariants are maintained.

Group of size 4: Let machines 1 to 4 be a group of size 4. Algorithm 4 runs the even parity if there is no machine with load more than $\frac{3}{2}$ that is not running. All invariants are maintained.

If $a_3 > \frac{3}{2}$, Algorithm 4 runs the uneven parity. We need to consider invariant 1 on machines 2 and 4, invariant 3 on machines 4 to 5 as well as invariant 4 on machines 2 to 5. We have $d_3 > \frac{1}{2}$, thus $d_2 \leq 0$ and $d_4 \leq 0$. Invariant 1 is maintained on machines 2 and 4 and invariant 3 is maintained on machines 4 and 5. Invariant 4 is also maintained, because we have $d_2 + d_3 \leq \frac{1}{2}$ from invariant 1 as well as $d_4 \leq 0$ and $d_5 \leq 0$. Therefore $d_2 + ... + d_5 \leq \frac{1}{2}$ and invariant 4 holds as well.

If $a_1 > \frac{3}{2}$ Algorithm 4 runs the uneven parity as long as $a_4 \leq 1$. We get $d_2 \leq \frac{1}{2}$. Invariants 1,2 and 3 are clearly maintained. We need to consider invariant 4 on machines 2 to 5. Invariant 4 is maintained on machines 1 to 4, because machines 1 and 3 are running, which means $d_1 + ... + d_4 \leq 1$. With $d_1 > \frac{1}{2}$ and $d_5 \leq 0$ (since machine 5 is empty) we get $d_2 + ... + d_5 \leq \frac{1}{2}$.This means invariant 4 holds on machines 2 to 5 as well.

If both $a_1 > \frac{3}{2}$ and $a_4 > 1$ we have $d_1 > \frac{1}{2}$ and $d_4 > 0$ and get $d_1 \leq \frac{1}{2}$ and $d_3 \leq 0$. We need to consider invariant 3 on machines 2 and 3 and invariant 4 on machines 2 to 5. Because of $d_1 + d_4 > \frac{1}{2}$ and invariant 4 on machines 1 to 4, we get $d_2 + d_3 \leq \frac{1}{2}$ and invariant 3 holds. Invariant 4 on machines 2 to 5 holds for the same reason as above.

Machines 2 to 5 are symmetrical to machines 1 to 4.

Group of size 5: Let all machines have load. Algorithm 4 runs the uneven parity if there is no machine with load more than $\frac{3}{2}$ that is not running. All invariants are maintained.

Assume $a_2 > \frac{3}{2}$. Algorithm 4 runs the even parity as long as $a_5 \leq \frac{3}{2}$. We need to consider invariant 1 on machines 1 and 3. We have $d_2 > \frac{1}{2}$ and thus $d_1 \leq \frac{1}{2}$ and $d_3 \leq 0$. Thus all invariants are maintained.

Assume $a_2 > \frac{3}{2}$ and $a_5 > \frac{3}{2}$. Algorithm 4 runs those two machines. Invariant 1 is maintained on machines 1, 3 and 4 for similar reasons as above. We need to consider invariant 3 on machines 3 and 4 as well as invariant 4 on machines 1 to 4. Since $d_2 > \frac{1}{2}$ and $d_5 > \frac{1}{2}$ we get $d_3 \leq 0$ and $d_4 \leq \frac{1}{2}$. Invariant 3 is maintained. We have $d_2 + ... + d_5 \leq 1$. With $d_5 > \frac{1}{2}$ and $d_1 < \frac{1}{2}$ we get $d_1 + ... + d_4 < 1$. Invariant 4 is maintained.

The case $a_4 > \frac{3}{2}$ is symmetrical to the case $a_2 > \frac{3}{2}$. □

Theorem 12. *Algorithm 4 is $\frac{5}{2}$-competitive on P_5.*

Proof. Follows directly from invariant 1 in Lemma 11 above. □

We now present the lower bound. The construction for four machines forces a load of $1 - 2x$ on the middle two machines in phase 1. For five machines we can force a load of $1 - x$ on machine 3. Applying phase 2 of the construction for four machines with a higher starting value yields a higher lower bound.

Lemma 13. *For any competitive algorithm and any $\varepsilon \in [0, 1]$, there exists a set of inputs on P_5 and $n \in \mathbb{N}$, so that at time n, $a_2 \geq 1 - \varepsilon$ or $a_4 \geq 1 - \varepsilon$ and* OPT *is in configuration* $(0, 0, 1, 0, 0)$.

Proof. We present a repeating set of inputs and show that, if the algorithm never allows a load of at least $1 - \varepsilon$ on machines 2 or 4, the load on machine 3 grows without bound. The construction is essentially one from Chrobak et al. (Theorem 5.4 in [33]) Load arrives after every timestep i. We denote the time right before load arrives with i^- and the time right after with i.

Let $a_2 + a_3 = L$ and OPT'S machines be empty at time t^-.

Time t: One unit of load arrives on machines 2 and 3 each, therefore $a_2 + a_3 = L + 2$. At time $(t + 1)^-$, we have $a_2 + a_3 = L + 1$, with $a_3 \geq L + \varepsilon$, since $a_2 \leq 1 - \varepsilon$. OPT can be in the configuration $(0, 1, 0, 0, 0)$.

Time t+1: One unit of load arrives on machine 4. At time $(t + 2)^-$, we have $a_3 + a_4 = L + \varepsilon$ and OPT'S machines are empty.

Because of symmetry, we can now repeat these inputs starting from an initial load of $L' = L + \varepsilon$ on machines 3 and 4. After every repetition the initial load increases by ε and therefore grows without bound. Because the load on machines 2 and 4 is bounded by $1 - \varepsilon$ this means the load on machine 3 grows without bound. Therefore the algorithm is not competitive if it never allows a load of at least $1 - \varepsilon$ on machines 2 and 4.

This means any competitive algorithm has to allow a load of at least $1 - \varepsilon$ on machines 2 or 4 at some point. W.l.o.g. this happens at machine 2 at time t^-. OPT can then be in configuration $(0, 0, 1, 0, 0)$ instead of $(0, 1, 0, 0, 0)$. This concludes the proof. □

Theorem 14. *No Algorithm can be better than $\frac{16}{7}$-competitive on P_5.*

Proof. Let the competitive ratio of ALG be equal to $2 + x$. Suppose that $x < \frac{2}{7}$.

W.l.o.g. let $a_2 \geq 1 - \varepsilon$ with OPT being in the state $(0, 0, 1, 0, 0)$. We can reach this state for any competitive algorithm because of Lemma 13. Let the time be 0^-.

The following set of inputs forces a combined load of at least $1 - x - \varepsilon$ on machines 2 and 3, while all machines of OPT are empty.

Time 0: One unit of load arrives on machine 1. At time 1^-, $a_1 + a_2 \geq 1 - \varepsilon$. OPT'S machines are empty.

Time 1: One unit of load arrives on machine 1 and 2 each. At time 2^-, $a_1 + a_2 \geq 2 - \varepsilon$. OPT is in configuration $(1, 0, 0, 0, 0)$. Additionally, $a_1 \leq 1 + x$ because of Lemma 5. This means $a_2 \geq 1 - x - \varepsilon$

Time 2: One unit of load arrives on machine 3. At time 3^-, $a_2 + a_3 \geq 1 - x - \varepsilon$. All machines of OPT are empty.

We use the achieved configuration as a starting configuration and run phase 2 (from the proof of the lower bound on P_4) on machines 1 to 4. During phase 2 the load on machines 2 and 3 grows without bound, as long as the initial load is more than $6x - 1$ and the side loads remain below $1 + x$. This is the case, because the initial load is arbitrarily close to $1 - x$ and $x < \frac{2}{7}$. This means no algorithm can achieve a competitive ratio better than $\frac{16}{7}$. □

3.5.2 ...for the flow model

For the path with five machines, we use the same algorithm as in the original model and achieve a competitive ratio of $\frac{3}{2}$. We also give a matching lower bound showing that the algorithm is optimal for the flow model.

Algorithm 5 (for 5 machines)

1. Run any group of size 1 at full speed.
2. If there is a group of size 2 at the edge, run the more central machine, unless the other machine has load $\frac{3}{2}$ and is receiving load.
3. Otherwise if there is a group of size 2, run the more central machine, unless the other machine has load 1 and is receiving load.
4. If there is a group of size 3, run the two outer machines unless the middle one has load $\frac{3}{2}$ and is receiving load.
5. We consider a group of size 4, e.g. machines 1 to 4. We run machines 2 and 4 unless $a_1 > \frac{3}{2}$ or $a_3 > \frac{3}{2}$. In that case run machines 1 and 3. Unless $a_1 > \frac{3}{2}$ and $a_4 > \frac{3}{2}$, in which case we run machines 1 and 4. Machines 2 to 5 are symmetrical to machines 1 to 4.
6. If all machines have load, we run the odd parity unless $a_2 > \frac{3}{2}$ or $a_4 > \frac{3}{2}$. If $a_2 > \frac{3}{2}$, we run the even parity as long as $a_5 \leq \frac{3}{2}$. If $a_2 > \frac{3}{2}$ and $a_5 > \frac{3}{2}$, we run machines 2 and 5. The case of $a_4 > \frac{3}{2}$ is symmetrical to $a_2 > \frac{3}{2}$.

Again the algorithm as well as the proof are the same as in the original model and in this case even the invariants remain the same.

Lemma 15. *Algorithm 5 maintains the following invariants at all times.*

1. $d_i \leq \frac{3}{2}$
2. $d_i + d_{i+1} \leq 1$ *for* $i = 1, 4$
3. $d_i + d_{i+1} \leq \frac{1}{2}$ *for* $i = 2, 3$
4. $d_i + d_{i+1} + d_{i+2} + d_{i+3} \leq 1$

Proof. The proof is the same as for the original model. □

Theorem 16. *Algorithm 5 is $\frac{3}{2}$-competitive on the path with five machines.*

Proof. There are two cases in which a machine that has load does not get run by the algorithm. In the first case the load of the machine is below $\frac{3}{2}$. In the second case the machine has a neighbor with load $\frac{3}{2}$. Then the neighbour has delay at least $\frac{1}{2}$. Given the invariants, the original machine then has delay at most $\frac{1}{2}$ and thus load at most $\frac{3}{2}$. □

We continue by showing a matching lower bound.

Lemma 17. *On the path with three machines there is an input that forces a load of 1 on a machine while* OPT*'s load is 0 on that machine (and 1 on one other machine).*

Proof. We use a simple version of the input used by Chrobak et al. Let there be a load of L on a pair (initially $L = 0$). Load arrives on both machines of the pair for one unit of time. Either ALG allows a load of 1 on the side machine, in which case the proof is finished, or there is at least load $L + \varepsilon$ on the middle machine, with load at most $1 - \varepsilon$ on the side machine. For one unit of time load arrives on both machines of the other pair. The load on the other pair is at least $L + 1 + \varepsilon$ and OPT can be in state $(1, 0, 1)$. After one unit of time OPT's machines are empty and there is a pair with load at least $L + \varepsilon$. We can repeat this input and the load on the pairs grows without bound unless ALG allows load 1 on a side machine at some point. □

Theorem 18. *No algorithm can be better than $\frac{3}{2}$-competitive on the path with five machines.*

Proof. By Lemma 17 we can reach a state where ALG has load 1 on machine 2, 3 or 4 and OPT has load 1 on another machine. This means there is a pair where ALG has load 1 and OPT's machines are empty. Load arrives on both machines for two units of time. This forces a load of at least $(1 + 2)/2 = 3/2$ on one of those machines. □

3.6 Results on m machines

In this section we provide some results for m machines. The first result is an $(\frac{1}{\varepsilon} + 2)$-competitive algorithm for the flow model, with machines running at speed $1 + 2\varepsilon$.

Lemma 19. *In any interval of length T, at most T+2 load arrives on any two adjacent machines and at most $T + 1$ load arrives on any single machine.*

Proof. Even if the buffers of OPT are empty at the start of this interval, it can process only T load during the interval, therefore more than $T + 2$ load on a pair or $T + 1$ load on an individual machine would make OPT's buffer overflow. □

Algorithm 6 (for machines of speed $1 + 2\varepsilon$)

Use phases of length $\frac{1}{\varepsilon}$.

- In phase 0 do nothing.
- In phases > 0 let L_i be the load that arrived on each *odd* machine i. Process this load during the first $\frac{L_i}{1+2\varepsilon}$ time in the next phase.
- Run each *even* machine whenever it can be run. This means it gets run at the end of each phase, after both of its neighbours have stopped processing.

Theorem 20. *Algorithm 6, whose machines run at $(1 + 2\varepsilon)$-speed, is $(\frac{1}{\varepsilon} + 2)$-competitive.*

Proof. W.l.o.g. consider machines 1 to 3 and let the current phase be j. In total the algorithm can process $\frac{1+2\varepsilon}{\varepsilon} = \frac{1}{\varepsilon} + 2$ load on machines 1 and 2, which means it can process everything that could possibly arrive during phase $j-1$ by Lemma 19.

The load on odd machines is highest at the start of the phase and is at most $\frac{1}{\varepsilon}$, because no more load can arrive during phase $j-1$ and all the load that arrived before phase $j-1$ gets processed before phase j.

For the load on even machines let L_i be the load that arrived during the previous phase on machine i and consider machine 2. W.l.o.g. $L_1 > L_3$. At the end of a phase, its load is at most $\frac{1}{\varepsilon} + 2 - L_1$. By the time it starts processing, machine 1 has run for $\frac{L_1}{1+2\varepsilon}$, which means $\frac{L_1}{1+2\varepsilon}$ additional load can arrive on machine 2. Thus when machine 2 starts processing, the load is at most $\frac{1}{\varepsilon} + 2 - L_1 + \frac{L_1}{1+2\varepsilon} = \frac{1}{\varepsilon} + 2 - \frac{2\varepsilon L_1}{1+2\varepsilon} \leq \frac{1}{\varepsilon} + 2$. □

We use an algorithm called greedy extension from Chrobak et al [33] to achieve the following upper bounds on $m > 5$ machines.

Theorem 21. *On the path with $m > 5$ machines there is a $\frac{m}{2} + \frac{1}{4}$-competitive algorithm for the original model and a $\frac{m}{2} - \frac{2}{3}$-competitive algorithm for the flow model.*

Proof. Chrobak et al [33] use an algorithm called greedy extension to achieve the following result. If there is an algorithm that achieves ratio R on the path with k machines, then extending this path by one machine to the left and right increases the competitive ratio by 1 (Lemma 6.1 in their paper). Thus we get an upper bound of $R + \frac{m-k}{2}$ for paths with $m > k$ machines with m being of the same parity as k.

In the original model there are algorithms with competitive ratio $\frac{9}{4}$ on 4 machines and $\frac{5}{2}$ on 5 machines. This gives upper bounds of $\frac{m}{2} + \frac{1}{4}$ on paths with an even number of machines and $\frac{m}{2}$ for an odd number of machines.

In the flow model there are algorithms with competitive ratio $\frac{4}{3}$ on 4 machines and $\frac{3}{2}$ on 5 machines. This gives upper bounds of $\frac{m}{2} - \frac{2}{3}$ on paths with an even number of machines and $\frac{m}{2} - 1$ for an odd number of machines. □

Finally, we present a lower bound of $\frac{12}{5}$ on m machines for the original model.

Theorem 22. *No algorithm can be better than $\frac{12}{5}$-competitive on m machines for m large enough in the original model.*

Proof. Let $x < \frac{2}{5}$.

We consider 4 adjacent machines numbered 1 to 4. We start with a load of L on machines 2 and 3, while all machines of OPT are empty. There is a set of inputs that forces a total load of at least $\frac{5}{2} - x + \frac{3}{2}L_n > \frac{21}{10} + \frac{3}{2}L_n$ on all machines combined, while OPT can be in state $(1,0,0,1)$ as well as $(0,1,1,0)$ (phase 2 of the proof of the lower bound on P_4). If we reach a state where ALG has a delay of L_{n+1} on two adjacent machines, we can reach a state where ALG has a load of L_{n+1} on these machines and all machines of OPT are empty by Lemma 6.

This means if, after the set of inputs, there is a delay of $L_{n+1} > L_n$ on either machine 1, machines 2 and 3, or machine 4 we can reach a state with a higher starting load L on two adjacent machines and can repeat the set of inputs. This is clearly true if this load is on machines 2 and 3. If the load is on machine 1 or machine 4, then we shift the set of considered machines to the left or right. This is possible, because we assume the initial set of four machines to be in the middle of the set of m machines. This means for m large enough, there can always be empty machines to the left or right of the considered set of four machines. Thus if there is a delay of $d_1 = L_{n+1}$ on machine 1 and OPT is in state $(0, 1, 1, 0)$, we remove machines 3 and 4 from the considered machines and add 2 machines to the left of machine 1. Then we update the numbering accordingly. We can then reach a state where the middle two machines have load L_{n+1} and all machines of OPT are empty.

Next, we choose the highest load out of a_1, $a_2 + a_3$ and a_4 as our next load L_{n+1}. Since the total load is at least $\frac{21}{10} + \frac{3}{2}L_n$ we have $L_{n+1} \geq (\frac{21}{10} + \frac{3}{2}L_n)/3 = \frac{7}{10} + \frac{1}{2}L_n$. Starting from load $L_0 = 0$ this recurrence can be expressed as $L_n = \frac{7}{5} - (\frac{1}{2})^n \frac{7}{5}$. Thus for large enough n (which may require many iterations and shifts and thus a large m) we get a load that is arbitrarily close to $\frac{7}{5}$ on the middle pair of a set of four machine, while all machines of OPT are empty. We can now apply phase 2 of the lower bound for four machines with a starting load of $\frac{7}{5}$. Since $x < \frac{2}{5}$, we have $\frac{7}{5} > 6x - 1$ and the load on the middle two machines grows without bound as long as the side loads remain below $1 + x$. This concludes the proof. □

This construction only works if one unit of load can arrive instantly. Thus this lower bound only holds for the original model and not the flow model.

3.7 Straying from the path

In this section we explore some graphs other than the path. Chrobak et al. already provide a few results in this direction. For one, the greedy extension algorithm provides a linear result on tree graphs. Furthermore, they are able to guarantee a finite competitive ratio on general graphs, however, this ratio may be very large.

We begin with an algorithm that works on bipartite graphs, which includes trees as well as even cycles. This algorithm maintains a competitive ratio that is linear in the size of the graph.

We then provide an algorithm with speed-up that works on general graphs, and finally we explore a relaxation of the problem that allows for more results on general graphs while preserving the difficulty on the path.

We start off with a few definitions that are useful in the graph setting.

In the graph setting a *group* is the induced subgraph of a maximal contiguous set of machines that all have load. Let d be the largest diameter among all possible groups in the graph.

In other words, it is the length of the longest path where there is no shortcut that uses a single edge. This is because that edge would connect two nodes within the group. There may be a shortcut that uses two or more (consecutive) edges but this shortcut would travel over a node outside the group.

The algorithm aims to maintain delay $d+2$ on every pair. While this holds, the delay on any individual machine i is at most $(d+2)-(a_j-1)$ for each machine j adjacent to i. This is because of the invariant and the fact that j has load a_j and therefore at least delay $a_j - 1$. The delay on machine i can also not be more than its load a_i. We define the *potential delay* on a machine as

$$p_i := \min_{j \in N}((d+2)-(a_j-1), a_i)$$

where N is the set of neighbors of i. This is an upper bound on the delay of i.

Algorithm 7 for bipartite graphs

1. Sort machines by nonincreasing load and consider them in this order. If a neighbor of i is running, i is skipped. Else, machine i is selected to run if $a_i > d+2$.
2. Consider machine i if it has distance 2 to a machine k that is already selected to run and if it has not already been considered during step 2. Let j be the machine between i and k. If $p_i + p_j > d+2$ run machine i. Repeat this step until there are no more machines that fit the criteria.
3. Run the remaining machines without running neighbors at speed $1/2$.

Rule 3 of the algorithm uses processor sharing between adjacent machines and potentially runs two adjacent machines at half speed. This can be achieved by rapidly switching between two independent sets. This is possible because the graph is bipartite.

Lemma 23. *On a bipartite graph, Algorithm 7 maintains a delay of $d+2$ on any pair of adjacent machines.*

Proof. Consider a group of machines. If no machine with load more than $d+2$ exists, then the algorithm does not select any machine during step 1. Since the criterion in step 2 requires a running machine, the algorithm also does not select any machine during step 2. This means the algorithm runs all machines at speed $1/2$. The invariant is maintained for any pair that is running at full speed, including if it contains two machines running at speed $1/2$. The only pairs that are not running at full speed are either empty or consist of an empty machine and another machine that has load at most $d+2$ under the assumption. The invariant holds on both types of pairs.

Now assume there are machines with load more than $d+2$. As long as such machines are available, the algorithm selects them during step 1 to run at full speed.

Once the algorithm has considered all of these machines, it continues to select machines based on the criterion in step 2. When using this criterion, the algorithm only selects machines that have distance 2 to a previously selected machine.

In particular this means that for any machine that is running at full speed, we can find a path of odd length that ends with this machine and begins with a machine that is running at full speed and has load more than $d + 2$ and every second machine on this path (the odd parity) is running at full speed. For machines with load more than $d + 2$ that were selected during step 1 this path only consists of the machine itself.

Now consider three sequential machines i, $i + 1$ and $i + 2$ on this path. Machine i is running at full speed and has load a_i. Then $p_{i+1} \leq (d+2) - (a_i - 1)$. The algorithm only runs machine $i + 2$ at full speed if $p_{i+1} + p_{i+2} > d + 2$ which means $p_{i+2} > a_i - 1$. In other words, whenever the algorithm selects a machine during step 2 there exists another running machine within distance 2 and the load of the two machine differs by at most 1.

Combining these two observations it follows that if a machine i has (even) distance k to a machine with load $d + 2$ that is running, then machine i has load at least $d + 2 - k/2$ if it is running at full speed. Since the distance between two machines is at most d in any group, any machine that is running at full speed therefore has load at least $\frac{d}{2} + 2$.

Any machine that is not running is either empty or adjacent to a machine that is running at full speed and has load at least $\frac{d}{2} + 2$. This means the delay on a machine that is not running is at most $\frac{d}{2} + 1$ by the invariant. Therefore the invariant holds on pairs that consist of two machines that are not running.

If there is a pair (i, j) of adjacent machines where machine i is running at speed $1/2$ and machine j is not running, then $p_i + p_j \leq d + 2$ otherwise the algorithm would run machine i at full speed and the invariant holds on this pair. □

The weighted fractional chromatic number in a graph G is the optimal objective value of the following linear program:

$$\begin{aligned} \text{minimize} \quad & \chi(G,h) = \sum_I x_I \\ \text{subject to} \quad & \sum_{I \ni i} x_I = h_i \\ & x_I \geq 0 \end{aligned}$$

Here I denotes an independent set in the graph and h is a vector of weights with one weight given to each node of the graph. The fractional weighted chromatic number is also the minimum time required to remove load h_i from each node in the graph. x_I is the time independent set I runs for in that schedule.

The fractional chromatic number χ is defined as $\chi(G, \mathbf{1})$ and is equal to the minimum time required to remove one unit of load from each node in the graph.

The following algorithm uses machines of speed $1 + \chi\varepsilon$ and is $\frac{1}{\varepsilon} + 2 + \chi$-competitive on any graph.

Algorithm 8 for machines of speed $1 + \chi\varepsilon$

Use phases of length $T = \frac{1+\chi\varepsilon}{\varepsilon}$

- In phase 0 do nothing.
- In phases > 0 consider only load that arrived in the previous phase. First execute the fastest schedule that leads to a state where no machine has load more than 1. Then execute the fastest schedule that removes the remaining load.

Lemma 24. *Algorithm 8 is* $\max\left(\frac{1}{\varepsilon} + 1 + \chi + \frac{1}{1+\chi\varepsilon}, \frac{1}{\varepsilon} + 2 + \chi - \frac{1}{1+\chi\varepsilon}\right)$*-competitive on a graph with fractional chromatic number* χ.

Proof. Consider phase $j > 1$. We first show that during phase j the algorithm removes any load that arrived in phase $j - 1$.

Let L_i be the load that arrived on machine i during phase $j - 1$ and let h_i be the load above 1 that arrived during phase $j - 1$. That is $h_i := \max(L_i - 1, 0)$. Since OPT's buffersize is 1, OPT must run machine i for at least time h_i during phase $j - 1$. This means there is a schedule that removes the loadvector $h = (h_1, \ldots, h_{|V|})$ in time T. This schedule (or a more efficient one) can be found by computing the weighted fractional chromatic number $\chi(h)$.

Since ALG's machines run at speed $1 + \chi\varepsilon$, this schedule can be executed in time $\frac{T}{1+\chi\varepsilon}$ by the algorithm. This means after time $\frac{T}{1+\chi\varepsilon}$ the algorithm can be in a state where on each machine there is at most one unit of load that arrived during phase $j - 1$ and there is time $T - \frac{T}{1+\chi\varepsilon}$ remaining until the end of phase j. We have

$$\begin{aligned} T - \frac{T}{1+\chi\varepsilon} &= \frac{1+\chi\varepsilon}{\varepsilon} - \frac{1+\chi\varepsilon}{\varepsilon}\frac{1}{1+\chi\varepsilon} \\ &= \frac{1+\chi\varepsilon}{\varepsilon} - \frac{1}{\varepsilon} \\ &= \chi \end{aligned}$$

Since the algorithm is in a state where there is at most one unit of load that arrived during phase $j - 1$ on any node and χ, the fractional chromatic number, is the time required to remove one unit of load from each node, the algorithm can clear the remaining load that arrived during phase $j - 1$ in time at most χ and all load that arrived during phase $j - 1$ is cleared before the end of phase j.

We now determine the maximum load that any machine can reach. Assume load $y + h_i$ arrives on machine i during phase $j - 1$ with $0 \le y \le 1$ load remaining in the buffer of OPT while load $h_i \le T$ is cleared by OPT.

We consider two time intervals within phase j where the load on machine i may increase and calculate the maximum value reached for each of these intervals. These intervals are $\left[0, \frac{T-(y+h_i-1)}{1+\chi\varepsilon}\right]$ and $\left[\frac{T}{1+\chi\varepsilon}, T - \frac{1}{1+\chi\varepsilon}\right]$ where 0 is the beginning of phase j.

First consider the time interval $\left[0, \frac{T}{1+\chi\varepsilon}\right]$. The algorithm clears load $y + h_i - 1$ before time $\frac{T}{1+\chi\varepsilon}$ and this takes time $\frac{y+h_i-1}{1+\chi\varepsilon}$. In the worst case this happens at the end of the interval which means load may arrive for time $\frac{T-(y+h_i-1)}{1+\chi\varepsilon}$ and all of this load can be cleared by OPT. Then an additional $1 - y$ load may arrive which fills the buffer of OPT. Then at time $\frac{T-(y+h_i-1)}{1+\chi\varepsilon}$ machine i can have load

$$y + h_i + \frac{T - (y + h_i - 1)}{1 + \chi\varepsilon} + (1 - y).$$

This is maximized for $y = 0$ and we get

$$\begin{aligned} 1 + h_i + \frac{T - (h_i - 1)}{1 + \chi\varepsilon} &= 1 + h_i + \frac{1}{\varepsilon} + \frac{h_i - 1}{1 + \chi\varepsilon} \\ &= \frac{1}{\varepsilon} + 1 + \frac{(1 + \chi\varepsilon)h_i - h_i + 1}{1 + \chi\varepsilon} \\ &= \frac{1}{\varepsilon} + 1 + \frac{\chi\varepsilon h_i + 1}{1 + \chi\varepsilon} \end{aligned}$$

This is maximized for $h_i = T$ and we get a load of at most

$$\frac{1}{\varepsilon} + 1 + \chi + \frac{1}{1 + \chi\varepsilon}$$

on machine i.

At time $\frac{T-(y+h_i-1)}{1+\chi\varepsilon}$ ALG starts running machine i and the load decreases until time $\frac{T}{1+\chi\varepsilon}$. At this point at most one unit of load that arrived during phase $j - 1$ is left on machine i. The latest time the algorithm can start processing this unit of load is $T - \frac{1}{1+\chi\varepsilon}$. At this point at most load $(1 - y) + T - \frac{1}{1+\chi\varepsilon} \leq 1 + T - \frac{1}{1+\chi\varepsilon}$ can have arrived on machine i during phase j. This means the load at time $T - \frac{1}{1+\chi\varepsilon}$ is at most

$$2 + T - \frac{1}{1 + \chi\varepsilon} = 2 + \frac{1 + \chi\varepsilon}{\varepsilon} - \frac{1}{1 + \chi\varepsilon} = \frac{1}{\varepsilon} + 2 + \chi - \frac{1}{1 + \chi\varepsilon}$$

It follows that no machine has load more than

$$\max\left(\frac{1}{\varepsilon} + 1 + \chi + \frac{1}{1 + \chi\varepsilon}, \frac{1}{\varepsilon} + 2 + \chi - \frac{1}{1 + \chi\varepsilon}\right)$$

□

Buffer management with processor sharing Consider the following variant of the buffer management problem. Machines may run at partial speeds as long as two adjacent machines do not run at a combined speed of more than 1. This gives the algorithm a lot more power since for example all machines in a clique may run at speed $\frac{1}{2}$. Note that this type of processor sharing is already possible on bipartite graphs and thus this relaxation does not change the results on the path.

However, given this relaxation, Algorithm 7 can be used on any graph. This means Algorithm 7 has a linear competitive ratio on any graph in this new model.

Furthermore, this model allows us to generalize the algorithm with resource augmentation from paths to general graphs. With resource augmentation, the algorithm may run two adjacent machines at a combined speed of $1 + 2\varepsilon$. With a slightly improved analysis we even get a slightly improved competitive ratio this time.

Algorithm 9 for machines of combined speed $1 + 2\varepsilon$ on pairs

```
Use phases of length $\frac{1}{\varepsilon}$
    • In phase 0 do nothing.
    • In phases $> 0$ let $L_i$ be the load that arrived on machine $i$ in the previous
      phase. Run machine $i$ at speed $L_i\varepsilon$ for the entire phase.
```

Lemma 25. *Algorithm 9 is $\frac{1}{\varepsilon} + 1$ competitive on any graph if processor sharing is allowed.*

Proof. Let the current phase be $j > 0$. We first show that the algorithm does not run any pair at speed more than $1 + 2\varepsilon$. At the beginning of phase $j - 1$, one unit of load may arrive on each machine of the pair, and since OPT is running the machines of the pair at a combined speed of at most 1 during phase $j - 1$, an additional $\frac{1}{\varepsilon}$ load may arrive. This means at most $\frac{1}{\varepsilon} + 2$ load arrives on a pair during phase $j - 1$. Therefore the combined speed of the machines in a pair is at most $(\frac{1}{\varepsilon} + 2)\varepsilon = 1 + 2\varepsilon$.

During phase j the algorithm removes load $\frac{1}{\varepsilon}L_i\varepsilon = L_i$ from machine i which means it removes all load that arrived in phase $j - 1$.

We now show that there is never more than $\frac{1}{\varepsilon} + 1$ load on any machine.

First assume machine i runs at speed at least 1 during phase j and assume time t has passed since the beginning of phase j. Since the beginning of phase $j - 1$ up to $\frac{1}{\varepsilon} + 1 + t$ load can have arrived on machine i and all load from phases before $j - 1$, has already been processed. Since the machine runs at speed at least 1 at least t load has been processed during phase j. This means the load on the machine is at most $\frac{1}{\varepsilon} + 1$.

Now assume machine i runs at speed less than 1 during phase j. At the beginning of the phase the load on machine i is L_i. At most $\frac{1}{\varepsilon} + 1$ load can arrive on machine i during phase j. The first unit of load may arrive at the beginning of the phase and then load may arrive at a rate of 1 for the entire phase while machine i runs at speed less than 1. This means the load is maximized at the end of the phase. At this point load L_i has been processed and the machine has load at most $\frac{1}{\varepsilon} + 1$. $\square$

Chapter 4

Lower bounds on algorithms for the buffer management problem

Despite the large gap between the lower bound of 12/5 and the upper bound of $(m+1)/2$ an algorithm with a sublinear competitive ratio has so far been elusive. Of course quite a few candidates have been considered. In this section we present a series of algorithms and give lower bounds or examples to explain why the algorithms do not perform as well as desired.

We start out with the classic algorithm GREEDY and the interesting result, that this algorithm is not competitive on the cycle with 6 machines. This result can then be used to find a linear lower bound on m machines, where adding 12 machines increases the lower bound by $\frac{1}{4}$.

Next, we consider the greedy extension and not only confirm the linear lower bound matching that matches the result of Chrobak et al. [33] when we extend by one machine, but we also show that we still get linear lower bounds when we extend by two or three machines.

GREEDY sometimes runs machines that are a distance of two apart from each other, which leads to suboptimal throughput overall.

Another simple idea that avoids this problem is to always run a parity in any given group. We choose the parity that contains the machine with higher load, and if there is a tie, we run the odd parity. We find a linear lower bound where adding six machines increases the lower bound by one.

In the last section of this chapter we consider throughput maximizing algorithms. In particular we consider a throughput maximizing version of Algorithm 7, the algorithm from chapter 3.7, that achieves a linear competitive ratio on bipartite graphs, and we show that this algorithm is not better than linear competitive on the line.

4.1 Greedy

We begin with the most straightforward algorithm GREEDY. This algorithm in each step chooses the machine with the most load from the set of available machines, resolving ties arbitrarily. Whenever a machine is chosen, this machine and its neighbors are removed from the set of available machines.

First we consider the following useful lemma:

Lemma 26. *Given delay d on a pair we can reach delay $\frac{d}{2}+\frac{1}{2}$ on an individual machine.*

Proof. Assume ALG has delay d on a pair and load $d+r$ with $r \leq 2$. Load arrives continuously on both machines for time $2-r$. OPT runs one machine on the pair the whole time. Afterwards the load of ALG on that pair is at least $d+2$ and OPT has one unit of load on both machines. After one more unit of time ALG's load on the pair is still at least $d+1$ with at least $\frac{1}{2}d+\frac{1}{2}$ of that load being on one machine. OPT can be empty on this machine and thus this machine has delay $\frac{1}{2}d+\frac{1}{2}$. □

We proceed with a lower bound on the competitive ratio of GREEDY on the path with three machines.

Lemma 27. *On the path with three machines, we can reach a state where* GREEDY *has load arbitrarily close to* $\frac{1}{2}$ *on every machine and* OPT'S *machines are empty.*

Proof. We use the same input as in the first phase for the lower bound on four machines, however, since GREEDY distributes the load on the middle two machines evenly in the first step, a fourth machine is not necessary.

Repeating the following sets of inputs forces a load of $\frac{1}{2}$ on machines 1 and 2 each in the limit, while OPT'S machines are empty. Let $L = a_1 = a_2$ be the load on the first two machines. In the beginning $L = 0$.

Time 0: Before each iteration we allow GREEDY to process all load on machine 3. Thus at time 0 GREEDY is in state $(L, L, 0)$ while OPT is in state $(0, 0, 0)$. For one unit of time load arrives on machines 1 and 2. After the timestep, GREEDY is in state $(L+\frac{1}{2}, L+\frac{1}{2}, 0)$ and OPT in state $(1, 0, 0)$.

Time 1: For one unit of time load arrives on machine 3. We may assume $L+\frac{1}{2} < 1$ otherwise we would have had $L > \frac{1}{2}$ in the beginning. Then all machines run at half speed for time $L+\frac{1}{2}$ and then every machine has load $\frac{1}{2}L+\frac{1}{4}$. For the remaining time of $\frac{1}{2}-L$, machines 1 and 3 run at full speed. At the end of the timestep, GREEDY ends up in state $(\frac{3}{2}L-\frac{1}{4}, \frac{1}{2}L+\frac{1}{4}, \frac{1}{2}L+\frac{1}{4})$ while OPT is in state $(0, 0, 0)$, since $\frac{1}{2}L+\frac{1}{4}-(\frac{1}{2}-L) = \frac{3}{2}L-\frac{1}{4}$.

We may now repeat this input with load $L' = \frac{1}{2}L+\frac{1}{4}$. The load L increases as long as $L < \frac{1}{2}$ and in the limit L gets arbitrarily close to $\frac{1}{2}$. Additionally, since the load on the third (or first) machine is $\frac{3}{2}L-\frac{1}{4}$, this machine also approaches load $\frac{1}{2}$ in the limit. This means we get a load of (almost) $\frac{1}{2}$ on all three machines while OPT'S machines are empty. □

This lemma holds for both the flow and the original model. We reach a delay of 1 on two machines. Because of Lemma 26 we can then reach a delay of 1 on an individual machine as well which results in a lower bound of 2 on the competitive ratio of GREEDY in the original model.

After letting load arrive on a pair of machines with delay 1 for two units of time, we reach a combined load of 3 in the flow model and an individual machine with load $\frac{3}{2}$.

Chrobak et al. [33] have shown a lower bound of $\frac{5}{2}$ on GREEDY in the original model on the path with four machines.

Theorem 28. *In the original model the algorithm* GREEDY *is not better than* 2*-competitive on the path with three machines and* 2.5*-competitive on the path with four machines.*

Theorem 29. *In the flow model, the algorithm* GREEDY *is not better than $\frac{3}{2}$-competitive on the path with three machines and four machines.*

We now take a look at a cycle of 6 machines and show that, surprisingly, GREEDY is not competitive at all in this case.

Theorem 30. GREEDY *is not competitive on the cycle with 6 machines.*

Proof. By Lemma 27 we can get load $\frac{1}{2}$ on three adjacent machines. Then we repeat this construction on the second triple while load arrives on the middle machine of the first triple. This means we can get arbitrarily close to $\frac{1}{2}$ on all machines while OPT's machines are empty.

Now let $x > \frac{1}{4}$ and consider the state (x, x, x, x, x, x) for ALG.

	Loads of ALG						Loads of OPT					
Time	1	2	3	4	5	6	1	2	3	4	5	6
0^-	x	x	x	x	x	x	0	0	0	0	0	0
0	$x+1$	$x+1$	$x+1$	x	$x+1$	x	1	1	1	0	1	0
1^-	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+\frac{1}{2}$	x	x	x	0	1	0	0	0	0
1	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+1$	x	$x+1$	0	1	0	1	0	1
1.5	$x+\frac{1}{2}$	x	$x+\frac{1}{2}$	$x+\frac{1}{2}$	x	$x+\frac{1}{2}$						
2^-	$x+\frac{1}{4}$	$x-\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x-\frac{1}{4}$	$x+\frac{1}{4}$	0	0	0	0	0	0
2	$x+\frac{1}{4}$	$x+\frac{3}{4}$	$x+\frac{5}{4}$	$x+\frac{5}{4}$	$x-\frac{1}{4}$	$x+\frac{5}{4}$	0	1	1	1	0	1
3^-	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{3}{4}$	$x+\frac{3}{4}$	$x-\frac{1}{4}$	$x+\frac{1}{4}$	0	0	1	0	0	0
3	$x+\frac{5}{4}$	$x+\frac{1}{4}$	$x+\frac{3}{4}$	$x+\frac{3}{4}$	$x+\frac{3}{4}$	$x+\frac{1}{4}$	1	0	1	0	1	0
4^-	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{1}{4}$	0	0	0	0	0	0

Time 0: GREEDY runs machine 5 for the entire timestep and thus the load on machines 4 and 6 does not change. Meanwhile, machines 1 to 3 all run at speed $\frac{1}{2}$.

Time 1: For the first half of the timestep, machines 4 and 6 have the most load and are running, and then machine 2 may also run.

For the second half of the timestep, machines 1 and 6 and 3 and 4 share processing time, i.e. run at half speed. This means GREEDY may run all machines at half speed.

Time 2: Machine 6 runs for the entire timestep and thus the load on machines 5 and 1 does not change. Meanwhile, machines 2 to 4 all run at speed $\frac{1}{2}$.

Time 3: Machine 1 runs for the entire timestep and thus the load on machines 6 and 2 does not change. Meanwhile, machines 3 to 5 all run at speed $\frac{1}{2}$.

We can repeat this construction and each time the load on all machines grows by $\frac{1}{4}$. Therefore GREEDY is not competitive on the cycle with 6 machines. □

This result on the cycle can help us find a linear lower bound for GREEDY on the path.

Theorem 31. *There is a linear lower bound on the competitive ratio of* GREEDY *on the path with m machines.*

Proof. Consider two adjacent groups of 6 machines with load x on every machine and run the construction for the cycle on both groups. We can verify that for the first iteration, the behavior on the rightmost machine of the left group, that is machine 6, and the leftmost machine of the right group, that is machine 7, is the same as the behavior on the corresponding machines in the cycle (machines 6 and 1).

	Loads of ALG											
Time	1	2	3	4	5	6	7	8	9	10	11	12
0^-	x	x	x	x	x	x	x	x	x	x	x	x
0	$x+1$	$x+1$	$x+1$	x	$x+1$	x	$x+1$	$x+1$	$x+1$	x	$x+1$	x
1^-	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+\frac{1}{2}$	x	x	x	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+\frac{1}{2}$	x	x	x
1	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+1$	x	$x+1$	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+\frac{1}{2}$	x	x	x
1.5	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{1}{2}$	$x+\frac{1}{2}$	x	$x+\frac{1}{2}$	$x+\frac{1}{2}$	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x-\frac{1}{4}$	$x-\frac{1}{4}$	$x-\frac{1}{4}$
2^-	x	x	$x+\frac{1}{4}$	$x+\frac{1}{4}$	$x-\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{1}{4}$	x	x	$x-\frac{1}{2}$	$x-\frac{1}{2}$	$x-\frac{1}{2}$
2					$x-\frac{1}{4}$	$x+\frac{5}{4}$	$x+\frac{1}{4}$	x				
3^-					$x-\frac{1}{4}$	$x+\frac{1}{4}$	$x+\frac{1}{4}$					
4						$x+\frac{1}{4}$	$x+\frac{5}{4}$					
4^-						$x+\frac{1}{4}$	$x+\frac{1}{4}$					

Now consider three adjacent groups of 6 machines with load x on every machine and run the construction on each group. The observation above implies that the behavior on the middle group is exactly the same as the behavior on the cycle for one iteration of the input. This means after one iteration all machines in the middle group have load $x + \frac{1}{4}$.

Now consider 5 adjacent groups of 6 machines with load x on every machine and run the construction on each group. We number the groups from left to right. After one iteration, groups 2, 3 and 4 have load $x + \frac{1}{4}$ on all of their machines, since they are each adjacent to another group on both sides and we can apply the argument for three adjacent groups.

We repeat the input again on groups 2, 3 and 4, and after this iteration the load on all machines of group 3 is $x + \frac{1}{2}$. This is because we can again apply the argument for three adjacent groups.

Extending a group of 6 machines by k other groups on each side allows us to simulate the behavior on the cycle on the middle group for k timesteps. Therefore adding 12 machines, six to each side, increases the lower bound by $\frac{1}{4}$. □

4.2 The Greedy extension

In this section we consider the algorithm that provides the current upper bounds on the buffer problem on the line, the GREEDY EXTENSION.

Let there be an algorithm that works on a segment of the line. We call this line segment the core and the algorithm the core algorithm. We extend the algorithm by adding one or more machines on both sides of the core.

At any point in time the algorithm first selects machines based on the choices of the core algorithm. Then the algorithm chooses a maximal independent set among the available machines of the extension, by greedily selecting machines with the highest load.

Chrobak et al. show that adding one machine to the left and right increases the competitive ratio by one. Repeatedly extending by one machine from a core on a smaller line segment provides the current upper bounds on the line with m machines.

We now examine a few variations of extending a core, not only reviewing the lower bound on the extension with one machine but also on extensions with up to three machines.

Lemma 32. *Let machine 1 be the rightmost machine of the core that is extended to the right by machine* 2*. If we can reach a delay of d on machine* 1*, then we can reach a delay of* $d+1$ *on machine* 2*.*

Proof. Since we can reach delay d on machine 1 we can get into state $(d+1, 0)$ on machines 1 and 2 for ALG while OPT is in state $(1, 0)$. Load arrives continuously on machine 2 while no load arrives on the core, and we wait for the algorithm on the core to clear the load on machine 1.

We can assume that the core algorithm runs machine 1 the whole time, because whenever this is not the case ALG and OPT both run machine 2 and the situation on the extension does not change until the core algorithm resumes processing machine 1.

Since ALG cannot run machine 2 while machine 1 is running, we end up in state $(0, d+1)$ while OPT is still in state $(1, 0)$. □

In the next version of the GREEDY EXTENSION we extent the core by two machines on each side.

Lemma 33. *Let machine 1 be the rightmost machine of the core that is greedily extended to the right by machines 2 and 3. If we can reach a delay of d on machine 1, then we can asymptotically reach a delay of* $2(d+1)$ *on machine 2 and* $d+\frac{3}{2}$ *on machine 3.*

Proof. Assume ALG is in state $(0, L, L)$ on machines 1 to 3 while OPT'S machines are empty. In the beginning $L = 0$. Since we can reach delay d on machine 1, we can eventually reach state $(d+1, L, L)$ with OPT being in state $(1, 0, 0)$. This is achieved by continuously having load arrive on machine 3 which causes the loads on the extension to remain the same while the delay on the core grows.

Now load arrives continuously on machine 2 and we wait till the core algorithm has cleared the load on machine 1. Again we can assume that machine 1 simply runs the whole time until it is empty, because whenever this is not the case load may arrive on whatever machine is running in the extension and the state on the three machines remains the same.

ALG ends up in state $(0, L+d+1, L-d-1)$ if $L > d+1$ and in state $(0, L+d+1, 0)$ if $L \leq d+1$ while OPT is still in state $(1, 0, 0)$. Consider the second case where ALG is in state $(0, L+d+1, 0)$. Load arrives continuously on machine 3 and after time $(L+d+1)/2$ we are in state $(0, \frac{L+d+1}{2}, \frac{L+d+1}{2})$. OPT can clear its load on machine 1 during this time.

We can repeat this process with load $L' = \frac{L+d+1}{2}$ and since $\frac{L+d+1}{2} > L$ for $L < d+1$ we reach state $(0, d+1, d+1)$ in the limit with OPT'S machines being empty. This also means that the first case does not occur.

With ALG being arbitrarily close to state $(0, d+1, d+1)$ let one unit of load arrive on machines 2 and 3 each. We end up in (almost) state $(0, d+2, d+2)$. For one unit of time ALG runs both machines at the same speed while OPT runs machine 3. This leads to (almost) state $(0, d+\frac{3}{2}, d+\frac{3}{2})$ on ALG versus $(0,1,0)$ on OPT. We have reached delay (almost) $d+\frac{3}{2}$ on machine 3.

Since L gets arbitrarily close to $d+1$, and during the construction we reach load $L+d+1$ on machine 2 while OPT is empty on this machine it follows that we can reach a delay arbitrarily close to $2(d+1)$ on machine 2. □

The extension algorithm above chooses greedily among the machines of the extension if given the choice. It turns out that other strategies do not help and no extension algorithm that extends by two machines (numbered 2 and 3) can avoid delay $d+1$ on machine 3.

Lemma 34. *Let machine 1 be the rightmost machine of the core that is extended to the right by machines 2 and 3. If we can reach delay d on machine 1 then no algorithm can avoid delay $d+1$ on machine 3.*

Proof. Assume the algorithm allows at most load $d+1-\varepsilon$ on machine 3. We can reach a state where ALG has load $d+1$ on machine 1 and let L be the combined load of machine 2 and 3. In the beginning $L=0$.

Load arrives on machine 2 while the core algorithm processes the load on machine 1. Again we only consider the time where machine 1 is running. During this process load $d+1$ arrives on machines 2 and 3 but at most $d+1-\varepsilon$ is processed, because the algorithm can only process load on machine 3. The combined load of machines 2 and 3 is now $L+\varepsilon$.

Then load arrives on machine 3 until the core algorithm has again load $d+1$ on machine 1. We can repeat the process with a combined load of $L' = L+\varepsilon$ on machines 2 and 3.

The load on the extension increases by ε each iteration and therefore grows without bound. □

Finally we consider greedily extending by three machines.

Lemma 35. *Let machine 1 be the rightmost machine of the core that is greedily extended to the right by machines 2, 3 and 4. If we can reach a delay of d on machine 1 then we can asymptotically reach a delay of $3(d+1)$ on machine 2, $2d+\frac{5}{2}$ on machine 3 and $d+\frac{7}{4}$ on machine 4.*

Proof. Assume ALG is in state $(0, L, L, 0)$ in the beginning, with OPT'S machines being empty. As before we can reach state $(d+1, L, L, 0)$. While machine 1 is being processed, load arrives on machines 2 and 4. Assuming $L < 2(d+1)$ we reach the state $(0, L+d+1, \frac{L}{2}, \frac{L}{2})$.

Now load arrives on machine 3 until we reach state $(0, \frac{3}{4}L + \frac{d}{2} + \frac{1}{2}, \frac{3}{4}L + \frac{d}{2} + \frac{1}{2}, 0)$. This is because $\frac{1}{2}(L + d + 1 + \frac{L}{2}) = \frac{3}{4}L + \frac{d}{2} + \frac{1}{2}$.

We repeat this process. Since $L < \frac{3}{4}L + \frac{d}{2} + \frac{1}{2}$ for $L < 2(d+1)$ we reach state $(0, 2(d+1), 2(d+1), 0)$ in the limit. OPT'S machines are empty. Since L gets arbitrarily close to $2(d+1)$ and during the process machine 2 reaches load $L + d + 1$ this also means we asymptotically reach a delay of $3(d+1)$ on machine 2.

By having one unit of load arrive on both machines 2 and 3 and having OPT process the load on machine 3, we may reach a delay of (almost) $2d + \frac{5}{2}$ on machine 3.

We can then reach a state where OPT has no load on machines 3 and 4 while ALG has (almost) load $2d + \frac{5}{2}$ on machine 3. Load arrives continuously on machine 4 until both machines have load $d + \frac{5}{4}$. Then one unit of load arrives on both machines and after one more timestep OPT can have one unit of load on machine 3 and no load on machine 4 while ALG has load arbitrarily close to $d + \frac{7}{4}$ on machine 4. □

Finally we show that even with strategies other than greedy, no algorithm that extends by three machines can avoid a delay of $d + 1$ on machine 4.

Lemma 36. *Let machine 1 be the rightmost machine of the core that is extended to the right by machines 2, 3 and 4. If we can reach delay d on machine 1 then no algorithm can avoid delay $d + 1$ on machine 4.*

Proof. Assume the algorithm allows at most load $d + 1 - \varepsilon$ on machine 4, and we have load $d + 1$ on machine 1. Furthermore, let L_2 be the load of machine 2 and $L_{3,4}$ be the combined load of machines 3 and 4.

Load arrives continuously on machines 2 and 4 while the load on machine 1 is cleared by the algorithm. After this process the load on machine 2 is $L_2 + d + 1$ and the combined load of machines 3 and 4 is still at least $L_{3,4}$.

We now wait while the load of $d + 1$ is restored on machine 1. During the entirety of this process load arrives continuously on machine 3.

We consider two cases. First assume the algorithm never runs machine 2 for time more than $d + 1 - \delta$ with $\delta < \varepsilon$ during this process. Then after this process the load on machine 2 is at least $L_2 + \delta$ and the load on machines 3 and 4 is still at least $L_{3,4}$. This is the initial situation with more load on machine 2.

We can repeat this construction and if the algorithm never runs machine 2 for more than time $d + 1 - \delta$ during an iteration of the construction then the load on machine 2 grows without bound.

Now assume that at some point the algorithm runs machine 2 for more than time $d + 1 - \delta$. This means the load on machine 3 grows by $d + 1 - \delta$ and since the algorithm can run machine 4 for time at most $d + 1 - \varepsilon$ the combined load of machines 3 and 4 increases by $\varepsilon - \delta > 0$. We can then repeat the construction with a higher load on machines 3 and 4.

It follows that the load on machines 2 or 3 grows without bound if the algorithm never allows a load of $d + 1$ on machine 4. □

4.3 Parity algorithms

In the construction against GREEDY on the cycle with 6 machines we observe that the algorithm repeatedly runs only 2 out of 6 machines, despite all machines having load. It makes a lot of sense to always run a parity whenever we are given a group. We still follow a greedy approach by choosing the parity that contains the machine with the highest load. But sometimes there may be a tie between the two parities. In this section we present a lower bound to a parity algorithm that, when presented with a tie, prefers to run the odd side.

Algorithm 10 Parity algorithm: prefer odd

1. Round the load on each machine down to the nearest integer
2. Starting from the left, repeatedly select a maximal contiguous subset of machines that have nonzero rounded load. These subsets are called groups.
3. In each group, run the even or the odd machines for one time unit, depending on which machine has the highest rounded load (we run the parity which contains the machine with highest load). In case of tied rounded loads, run the odd machines for one time unit.

Lemma 37. *There is a linear lower bound on the competitive ratio of the parity algorithm that prefers the odd parity.*

Proof. We show that there is a construction that, given $n = 2 + 6(k-1)$ machines, achieves delay k on the rightmost (even) machine. Furthermore the second last machine $n-1$ is empty for ALG but OPT has one unit of load there. In other words, there is a construction on $n = 2 + 6(k-1)$ machines that ends in state $(\dots, 0, k)$ for ALG and $(\dots, 1, 0)$ for OPT. We show this using induction.

For $k = 1$, consider two machines 1 and 2 and let one unit of load arrive on both machines. After one timestep, ALG has one unit of load on machine 2, while OPT can be empty on that machine. We reach state $(0, 1)$ for ALG and $(1, 0)$ for OPT.

Now assume we reach state $(\dots, 0, k)$ for ALG and $(\dots, 1, 0)$ for OPT using $n = 2 + 6(k-1)$ machines. We show that we can reach a delay of $k+1$ on machine $n+2$ while there is a delay of -1 on machine $n+1$ und this construction uses 4 additional machines to the left of machine 1.

We present a construction in two parts. In the first part we construct an input that moves the load on machine n first to machine $n+1$ and then to machine $n+2$. The purpose of this part of the construction is to remove the load OPT has on the second to last machine. The second part of the construction creates a delay of $k+1$ on machine $n+2$.

Construction part 1, moving load: Let ALG have a combined load of k on machines n and $n+1$ while OPT has no load on these machines. Load arrives continuously on machine $n+1$ until the load on all machines to the left of n is cleared (for both Alg and Opt). The load on pair $(n, n+1)$ is still k.

Due to the inductive hypothesis, we can use the initial construction again to create load k on machine $n-2$. Alg is now in state $(0, k, 0, a_n, a_{n+1})$ on machines $n-3$ to $n+1$ with $a_n + a_{n+1} = k$ and Opt is in state $(1, 0, 0, 0, 0)$.

Now one unit of load arrives on machines $n-2, n-1$ and $n+1$ and we reach state $(0, k+1, 1, a_n, a_{n+1}+1)$. Alg runs the even parity since it contains the machine $n-2$ with the highest load of $k+1$ and ends up in state $(0, k, 1, a_n - 1, a_{n+1}+1)$. The combined delay on machines n and $n+1$ is still k. Now load arrives continuously on machine $n+1$ until all machines beside the pair $(n, n+1)$ are cleared.

We repeat this procedure and after each step the load on machine $n+1$ increases by one until eventually all of the load on machines n and $n+1$ is on machine $n+1$ and thus we have reached delay k on the odd machine $n+1$ while machine n is empty for Alg and both machines are empty for Opt. We now have a construction that given $n+3$ machines (machines -1 to $n+1$ in the construction above) achieves a delay of k on the rightmost odd machine.

The same construction can be used again to move the load from machine $n+1$ to machine $n+2$. Alg is now in state $(\dots, 0, k)$ while Opt can be empty on all machines. Overall, this construction uses the original n machines as well as two additional machines to each side.

Construction part 2, increasing delay: Now load continues to arrive on machine $n+2$ until the loads on all other machines are cleared. We use the initial construction to create load k on machine $n-2$ and then, using part 1 of the construction again, move this load to the odd machine $n-1$. (This requires an additional two machines to the left.) Meanwhile, load arrives continuously on machine $n+1$. This means we can reach state $(k, 0, 0, k)$ on machines $n-1$ to $n+2$ while Opt is empty on all machines. The adjacent machines are empty for Alg. Overall, we use four additional machines to the left of the original n machines, and two machines to the right.

The following input on machines $n-1$ to $n+2$ achieves delay $k+1$ on the even machine $n+2$.

Loads of Alg				Loads of Opt			
O	E	O	E	O	E	O	E
k	0	0	k	0	0	0	0
$k+1$	1	1	$k+1$	1	1	1	1
k	1	0	$k+1$	1	0	1	0

We are now in state $(\dots, 0, k+1)$ while Opt is in state $(\dots, 1, 0)$ and to reach this state $n+6$ machines are required. It follows that adding 6 machines to the path increases the lower bound by 1. □

The same lower bound construction also works for an algorithm that instead of preferring the odd or even parity, prefers the right or left parity in a group.

4.4 Throughput maximization

One thing all the algorithms studied so far have in common is that they always try to run the machine with the highest load. Another promising approach is to maximize the overall throughput instead, which means trying to run as many machines as possible.

If a group consists of an odd number of machines with load, it is obvious how to achieve this goal: We simply run the parity that contains more machines. For even groups, however, there are more options. Perhaps the simplest option is to treat all machines equally, that is run them all at half speed.

Algorithm 11 Throughput maximization: Equal treatment

1. In all groups with an odd number of machines, maximize throughput. This means running the parity that contains more machines.

2. In even groups run all machines at half speed.

Lemma 38. *Algorithm 11 is not competitive even on three machines.*

Proof. Let ALG be in state $(0, x, 0)$ while OPT has no load. In the beginning $x = 0$. Load arrives on machines 2 and 3 and we are in state $(0, x+1, 1)$. The algorithm treats both machines the same and thus after one timestep ALG is in state $(0, x+\frac{1}{2}, \frac{1}{2})$ while OPT can be in state $(0, 0, 1)$. Then load arrives on machine 1. ALG is in state $(1, x+\frac{1}{2}, \frac{1}{2})$. After half a timestep we end up in $(\frac{1}{2}, x+\frac{1}{2}, 0)$. For the remaining half of the timestep, ALG runs both machines 1 and 2 at half speed and we end up in state $(\frac{1}{4}, x+\frac{1}{4}, 0)$ and OPT is empty. Finally load arrives continuously on machine 2 until ALG is ins state $(0, x+\frac{1}{2}, 0)$. We can now repeat this input and the load grows without bound. □

As we can see, this algorithm is not competitive at all, and this is mostly because of its decision making on even groups.

Another idea would be to run either the odd or even parity when given an even group. This approach is very similar to the one found in the parity algorithms of the previous chapter. In fact the same lower bound still holds. This is because the construction used in the lower bound for the parity algorithm only considers even groups and the parity algorithms do maximize throughput in this case. This means the behavior of a throughput maximizing algorithm that chooses a parity on even groups would be the same as the behavior of the corresponding parity algorithm on these constructions. (The parity algorithm itself does not maximize throughput since it may sometimes run the parity with less machines on odd groups.)

Clearly a throughput maximizing algorithm needs to choose carefully which pair to skip when given an even group.

We use this opportunity to combine the idea of throughput maximization with the strategy of algorithm 7 that achieves a linear upper bound on bipartite graphs in chapter 3.7.

This strategy aims to maintain a certain delay b for each pair, and intervenes whenever there is a pair where this delay is in danger. This means the algorithm looks for pairs with a potential delay of more than b. The potential delay of a machine i on the line graph with respect to delay b is defined as

$$p_i := \min_{j \in N}(b - (a_j - 1), a_i)$$

where N is the set of neighbors of i.

Whenever this is no pair with $p_i + p_j > b$, the algorithm behaves like the simple throughput maximizing algorithm presented above. Since the delay cannot increase when a pair is running at full speed, the algorithm pays special attention to the state of the side machines.

Algorithm 12 Throughput maximization: Maintain delay b on pairs.

1. In all groups with an odd number of machines, maximize throughput. This means running the parity that contains more machines.
2. If no side machine has load more than b, run all machines at speed $\frac{1}{2}$.
3. If one or both side machine has load more than b, run these side machines. Then consider machines i if it has distance two to a machine k that is already selected to run and let j be the machine between i and k. If $p_i + p_j > b$ run machine i and repeat this step.

   Otherwise if $p_i + p_j \leq b$ skip the pair (i, j) and maximize throughput on the remaining machines.

Lemma 39. *Let n be odd. On a line segment of length n we can reach a state where Algorithm 12 has load arbitrarily close to b on every even machine and* OPT*'s machines are empty.*

Proof. Using the construction in Lemma 38 we can reach a state where machine 2 has load $b - 1$. This is because during the construction no machine ever reaches a load above b and the algorithm behaves like Algorithm 11.

We now show that repeating the same construction after reaching load $b - 1$ leads to a load arbitrarily close to b on machine 2. Let ALG be in state $(0, b - \varepsilon, 0)$ for $\varepsilon \leq 1$ while OPT has no load. One unit of load arrives on machines 2 and 3 and we are in state $(0, b - \varepsilon + 1, 1)$. The algorithm prefers machine 2 for time $(1 - \varepsilon)$ and then treats both machines the same. After one timestep, ALG is in $(0, b - \frac{\varepsilon}{2}, 1 - \frac{\varepsilon}{2})$ while OPT can be in state $(0, 0, 1)$. Then load arrives on machine 1 and ALG is in state $(1, b - \frac{\varepsilon}{2}, 1 - \frac{\varepsilon}{2})$. After time $1 - \frac{\varepsilon}{2}$ we end up in $(\frac{\varepsilon}{2}, b - \frac{\varepsilon}{2}, 0)$. For the remaining time $\frac{\varepsilon}{2}$ of the timestep, ALG runs both machines 1 and 2 at half speed and we end up in state $(\frac{\varepsilon}{4}, b - \frac{3\varepsilon}{4}, 0)$ and

OPT is empty. Finally load arrives continuously on machine 2 until ALG is ins state $(0, b - \frac{\varepsilon}{2}, 0)$. This process can be repeated and the load on machine 2 gets arbitrarily close to b.

Next we can apply an extended version of this input to a line segment of length 5 such that the load on machine 4 gets arbitrarily close to b while the load on machine 2 remains the same. The following table illustrates this input. Machines 1, 4 and 5 behave just as the triple described above, while machines 2 and 3 extend the construction.

Loads of ALG					Loads of OPT				
0	b	0	$b-\varepsilon$	0	0	0	0	0	0
0	$b+1$	0	$b-\varepsilon+1$	1	0	1	0	1	1
0	b	0	$b-\frac{\varepsilon}{2}$	$1-\frac{\varepsilon}{2}$	0	0	0	0	1
1	b	1	$b-\frac{\varepsilon}{2}$	$1-\frac{\varepsilon}{2}$	1	0	1	0	1
$\frac{\varepsilon}{2}$	b	$\frac{\varepsilon}{2}$	$b-\frac{\varepsilon}{2}$	0	$\frac{\varepsilon}{2}$	0	$\frac{\varepsilon}{2}$	0	$\frac{\varepsilon}{2}$
$\frac{\varepsilon}{4}$	$b-\frac{\varepsilon}{4}$	$\frac{\varepsilon}{4}$	$b-\frac{3\varepsilon}{4}$	0	0	0	0	0	0
Load arrives continuously on machines 2 and 4									
0	b	0	$b-\frac{\varepsilon}{2}$	0	0	0	0	0	0

This can be repeated for all even machines and ALG ends up with (almost) load b on every even machine while OPT's machines are empty. □

Lemma 40. *Let $n = 2(b+1)+3$. On a line of length n there is an input that forces delay $b+1$ on a pair against Algorithm 12.*

Proof. Due to Lemma 39 we can reach a state such as $(0, b, 0, b, 0, \dots, b, 0, b, 0)$ for ALG while OPT's machines are empty. For example if $b = 5$ we can get

$$(0, 5, 0, 5, 0, 5, 0, 5, 0, 5, 0, 5, 0, 5, 0)$$

Assume b is odd and let time be 0. For the next $2b$ timesteps load arrives as follows. Machine 1 continuously receives load starting from time 0. Since the algorithm runs machines 1 and 2 both at half speed this shifts the load towards machine 1. After two timesteps, machines 1 and 2 are in state $(1, 4)$ and after $2b$ timesteps all load has been shifted and the machines are in state $(5, 0)$

Before time 2 machine 4 and after time 2 machine 3 continuously receive load. This does not change the behavior on machines 1 and 2. For the first two timesteps the state of machines 3 and 4 does not change, then load get shifted to the left similar to the first pair. After $2b$ timesteps machines 3 and 4 are in state $(1, 4)$. Not all of the load is shifted as the process started later.

This pattern continues, with another pair beginning load shifting after every two time steps. There is one exception to this pattern. Machine $b+2$ gets added to the pattern at the same time as machine $b+4$.

The following example illustrates the development. Machines that are receiving load are marked with bold letters.

$$\begin{gathered}
(\mathbf{0},5,0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0) \\
(\mathbf{1},4,\mathbf{0},5,0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0) \\
(\mathbf{2},3,\mathbf{1},4,\mathbf{0},5,0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0,\mathbf{5},0) \\
(\mathbf{3},2,\mathbf{2},3,\mathbf{1},4,\mathbf{0},5,\mathbf{0},5,0,\mathbf{5},0,\mathbf{5},0) \\
(\mathbf{4},1,\mathbf{3},2,\mathbf{2},3,\mathbf{1},4,\mathbf{1},4,\mathbf{0},5,0,\mathbf{5},0) \\
(5,0,4,1,3,2,2,3,2,3,1,4,0,5,0)
\end{gathered}$$

Consider machines $b+1$ to $b+4$. These machines are in the middle of the group of machines with load and we will refer to these as the middle four machines.

Note that the middle four machines are in state $(\lfloor\frac{b}{2}\rfloor, \lfloor\frac{b}{2}\rfloor, \lceil\frac{b}{2}\rceil, \lfloor\frac{b}{2}\rfloor)$. Now load arrives on every machine except the third machine of the middle four machines which has load $\lceil\frac{b}{2}\rceil$. In the example we reach state

$$(6,1,5,2,4,3,3,3,3,4,2,5,1,6,0)$$

The middle four machines now have load $(\lceil\frac{b}{2}\rceil, \lceil\frac{b}{2}\rceil, \lceil\frac{b}{2}\rceil, \lceil\frac{b}{2}\rceil)$ while OPT is in state $(1,0,1,1)$ on those machines.

$p_i + p_{i+1} > b$ is true on every pair considered by the algorithm. In this situation the algorithm arbitrarily chooses a pair to skip and therefore might not run the left pair of the middle four machines. OPT can remove its load on that pair and this means this pair reaches delay $\lceil\frac{b}{2}\rceil + \lceil\frac{b}{2}\rceil = b+1$. A similar result can be achieved if b is even. □

This shows that there is a linear lower bound to Algorithm 7 presented for bipartite graphs in chapter 3. However, we can see that most of the delay is caused by exploiting the weaknesses of the underlying simple algorithm. The algorithm starts to carefully pick a good pair to skip only after it is already too late and all pairs have high load. A promising approach might be to try more consistently to identify good pairs. The algorithm could for example always choose the pair with lowest load in an even group.

Chapter 5

The bamboo garden trimming problem

5.1 Introduction

In the discrete bamboo garden trimming problem (BGT), first introduced by Gasieniec et al. [51], we are given a set of n bamboo that grow at rates $v_1, \ldots, v_n$ per day. We assume that these growth rates are arranged such that $v_1 \geq v_2 \geq \cdots \geq v_n$. Initially the height is set to zero. Each day a robotic gardener cuts down one bamboo to height zero. The goal is to design a trimming schedule such that the height of the tallest bamboo is minimized. Gasieniec et al. gave a 2-approximation for discrete BGT which has been improved by van Ee [109] to a $\frac{12}{7}$-approximation.

Both results are obtained by reducing BGT to the pinwheel scheduling problem, a different, but closely related problem.

The pinwheel scheduling problem is motivated by the communication between a satellite and its ground station. The ground station wants to receive messages from n satellites. Time is slotted and a satellite i sends a message for p_i consecutive timeslots before switching to a different message. Each timeslot the ground station receives a message from a single satellite. This means in order to guarantee that no message is missed we need to find a schedule that allocates at least one timeslot to satellite i in any interval of p_i units of time.

We represent an instance of the pinwheel scheduling problem by the vector $p = (p_1, \ldots, p_n)$ of periods with which each satellite should be dialed in on. The density of a pinwheel scheduling instance is $d(p) = \sum_{i=1}^{n} \frac{1}{p_i}$.

Then for example the instance $(2, 3)$ has density $\frac{5}{6}$ and can be scheduled by repeating the sequence 12. However it turns out that the instance $(2, 3, p_3)$ can not be scheduled regardless of the value of p_3.

We may permute the elements of the pinwheel instance to be in nondecreasing order and aside from the instance (1) an instance can only be scheduled if $p_1 \geq 2$. Consequently, we assume $2 \leq p_1 \leq p_2 \leq \cdots \leq p_n$. Another necessary condition for schedulability is $d(p) \leq 1$.

We say that b is a density guarantee for the pinwheel problem if all instances p with $d(p) \leq b$ can be scheduled. Fishburn et al. [47] show that any instance with density at most 0.75 can be scheduled and for the special case of $p_1 = 2$ they give a guarantee of $5/6$. Chan and Chin [32] conjecture that any pinwheel problem with density at most $\frac{5}{6}$ can be scheduled. The example $(2, 3, p_3)$ shows that we cannot hope for a density guarantee above $\frac{5}{6}$ for general pinwheel instances, since this instance cannot be scheduled and may have a density arbitrarily close to $\frac{5}{6}$.

If an algorithm maintains a height of K then each bamboo i must be visited at least once in a period of $p_i^* = \lfloor \frac{K}{v_i} \rfloor$ time steps. This means there is a BGT-schedule that maintains height K if and only if the pinwheel problem $(p_1^*, \ldots, p_n^*)$ is schedulable.

An important consideration for pinwheel scheduling and BGT is the representation of a solution. This is because in general writing out an explicit representation of the schedule may take exponential time and space. The solutions for pinwheel scheduling provided by Fishburn as well as Chan and Chin come in the form of *fast online schedulers*. These are algorithms that can decide whether they can schedule an instance in polynomial time and then generate each symbol of the schedule in constant time.

In the first part of this paper we present an algorithm based on a pinweel reduction that improves the approximation ratio for discrete BGT to $\frac{10}{7}$.

In the second part, we consider the continuous version of the BGT problem, also introduced by Gasieniec et al. [51]. In this version the bamboo are distributed in some metric space and the gardener needs to travel between the bamboo to cut them. Cutting is done instantly and the goal is to find a route that minimizes the maximum height of the bamboos. Gasieniec et al. give a $O(\log n)$-approximation algorithm that works in any metric space.

We are considering the case where the gardener travels on a star graph. In this context we study a set of simple algorithms (compared to the relative complicatedness of a pinwheel schedule) that achieve constant approximation ratios for this case.

The *Deadline-Driven Strategy* always cuts the bamboo with the earliest deadline provided that the height of this bamboo has reached a certain threshold. The deadline of a bamboo is the time it reaches the height the algorithm wants to maintain. This algorithm has already been considered for discrete BGT and there it is a 2-approximation as shown by J. Kuszmaul [81].

A second simple algorithm is *Reduce-Fastest* which is a 2.62-approximation for discrete BGT as shown by Bilò et al. [20]. This algorithm always cuts the fastest plant provided that the height of this bamboo has reached a certain threshold.

5.1.1 Our results

For discrete BGT we propose a $10/7$-approximation algorithm that is based on a reduction to the pinwheel-problem, but using some new techniques that improve the approximation ratio. The result can be improved to $\frac{7}{5}$ using a computer-assisted proof.

In the continuous version of the problem we consider a star graph. We show that the 2.62-approximate algorithm Reduce-Fastest for discrete BGT, which can be seen as a special instance of a star graph, still works on arbitrary star graphs with one bamboo on each branch and we show that it is a 6-approximation in this case.

Furthermore, the deadline driven algorithm gives us a $(2+\sqrt{3})$-approximation on the star. This result can be extended to the case where there are multiple bamboo on each branch of the star. Here we pay a price in the approximation ratio and achieve a $(5+\sqrt{21})$-approximation.

5.1.2 Related work

Both the discrete and continuous version of BGT were first introduced by Gasieniec et al. [51] They provide a variety of results. The first algorithm they present is Reduce-Fastest, which in each step cuts the fastest growing bamboo above a certain height treshhold. This algorithm is a 2.62-approximation as shown by Bilò el al. [20]. A similar algorithm is Reduce-Max that always cuts the highest bamboo and is a 4-approximation. Both of these algorithms are online algorithms based on simple queries. This means they are flexible and can easily adapt to changes in the input (the set of growth rates). The final algorithm in this set of algorithms based on simple queries is the Deadline-Driven Strategy which is a 2-approximation. The Deadline-Driven Strategy always selects the plant that soonest reaches a certain height. Both results for the last 2 algorithms are from J. Kuszmaul [81].

Gasieniec et al. also give a fully offline 2-approximation algorithm that preprocesses the input and reduces the problem to a pinwheel-instance. This approach has been improved by van Ee [109] to a $\frac{12}{7}$-approximation.

The closely related pinwheel-problem was first introduced by Holte et al. [64] and then picked up by Chin and Chan [32]. They conjecture that any pinwheel instance with density up to $5/6$ can be scheduled. This conjecture is supported by a variety of works, including Fishburn et al. [47] who show that any instance with density at most 0.75 as well as instances with $p_1 = 2$ and density up to $5/6$ can be scheduled. Dei Wing [40] shows that the claim also holds for low-dimensional vectors with dimension up to 5, and more recently Gasieniec et al. [52] improve this result further by proving the claim for instances with up to 12 elements and additionally they give a set of schedules that solve all schedulable instances with at most 5 tasks.

There are other problems where the goal is to minimize the maximum height or backlog reached on a machine. One example is the Minimum Backlog Problem [24, 15, 82] where an adversary distributes water among a set of cups and the player may empty one or more cups on their turn.

In the problem of Buffer minimization with conflicts [33] there is a set of machines on a graph and load may arrive on these machines at any time. The algorithm may run machines to decrease their load but machines that are adjacent to each other on the graph may not run at the same time.

```
Algorithm 13
Let $H = \sum_{i=1}^{n} v_i$ and $R = \frac{10}{7}$. Using binary search in the interval $[H, 2H]$ find
the smallest $K$ such that the following procedure returns a valid schedule and
return this schedule.

  1. Given $K$ define $p_i^* = \lfloor \frac{K}{v_i} \rfloor$ and $p_i = \lfloor R p_i^* \rfloor$.

  2. Solve the pinwheel problem $(p_1, \dots, p_n)$.
```

5.2 A 10/7-approximation for discrete BGT

Runtime We now explain that any pinwheel instance with density up to 0.75 can be solved in time $O(n^3)$ which means Algorithm 13 runs in polynomial time. Note that solving or scheduling a pinwheel instance for our purposes means finding a fast online scheduler but does not include explicitly writing out the schedule. Chin and Chan give an algorithm that runs in time $O(n^3)$ and schedules any instance with density at most $\frac{7}{10}$. More precisely, their algorithm achieves the following guarantees based on the value of p_1:

p_1	guarantuee
2	0.75
3	0.70
4	0.70833
5	0.72196
6	0.73359
7	0.73807
8	0.74470
≥ 9	> 0.75

In particular this means any instance with $p_1 \geq 9$ and density at most 0.75 can be scheduled in time $O(n^3)$. Fishburn et al. find schedules for the remaining cases with small values for p_1 and density up to 0.75 based on the following idea.

They find a *porous* schedule, that is a schedule for the first few elements which contains a pattern of holes, and they extend such a schedule using the results from Chin and Chan to find a schedule for the remaining machines that fits into the holes. For example for the instance $(3, 4, p_3, \dots)$ we can create a porous schedule 120120 where the 0s are the holes in the schedule. The remaining machines after you take away machine 1 and 2 have density at most $\frac{1}{6}$, thus $p_i \geq 6$ for $i = 3, \dots$. For this pattern of holes, we divide the remaining periods by 3 and solve the resulting pinwheel-instance. For $p_i = 6, 7, 8$ this increases the density by a factor of not more than 4.5, thus the density of the resulting instance is at most $\frac{3}{4}$ with the first element of the new instance being 2. By the table of Chin and Chan this is schedulable.

A similar argument holds for $p_i \geq 9$ with the density of the new schedule being not more than 0.7. They further refine this approach in the paper but when considering the runtime we can see that we need to find a porous schedule for the cases not covered by the table of Chin and Chan and a schedule for the remaining machines. The porous schedules are given in the paper of Fishburn et al. (in particular see table 1 of [47]) and are thus available in $O(1)$. Then the algorithm in [32] with runtime $O(n^3)$ is used to schedule the remaining machines. This means pinwheel-instances with density at most 0.75 can be solved in time $O(n^3)$. In addition, Fishburn et al. show that instances with $p_1 = 2$ and density up to $\frac{5}{6}$ can also be solved in time $O(n^3)$.

Approximation ratio In order to analyze Algorithm 13 we need to show that the algorithm can solve the pinwheel problem it creates. To that end we first declare some notation and consider a few helpful lemmas.

$H = \sum_{i=1}^{n} v_i$ is a lower bound on the optimal value and it is known that there is an algorithm that produces a schedule of height $2H$ [51]. H is a lower bound because as long as all bamboo have height at most $H' < H$ the sum of all bamboo increases by at least $H - H' > 0$ each step until it exceeds nH'. Then there must be a bamboo with height more than H'. Thus the optimal value is somewhere in the interval $[H, 2H]$.

Let $A \subseteq \{1, \ldots, n\}$. We create a porous schedule as described in [47] for $\{p_i | i \notin A\}$ and define h_i for $i \in A$ as the minimum number of holes in p_i consecutive positions of the porous schedule. Let $D_A = 1 - \sum_{k \notin A} \frac{1}{p_k^*}$ be the maximum possible offline density of the unscheduled machines.

Lemma 41. *If $\frac{h_i}{p_i} \geq D_A$ for each $i \in A$ in a porous schedule for $\{p_k | k \notin A\}$ as well as $\frac{1}{p_i} \leq \frac{3}{4p_i^*}$ for all $i \in A$, then $(p_1, \ldots, p_n)$ is schedulable for* ALG.

Proof. If $\frac{h_i}{p_i} \geq D_A$ then

$$\sum_{i \in A} \frac{1}{h_i} \leq \frac{1}{D_A} \sum_{i \in A} \frac{1}{p_i} \leq \frac{1}{D_A} \frac{3}{4} D_A = \frac{3}{4}.$$

Thus the instance is schedulable by Lemma 2 of [47]. □

This Lemma allows us to schedule subsets of the machines where the ratio p^*/p is too high and check whether the rest can be scheduled.

Lemma 42. *If there is a schedule of height K then the procedure in Algorithm 13 finds a schedule of height at most $\frac{10}{7}K$.*

Proof. Let $p_i^* = \lfloor \frac{K}{v_i} \rfloor$. Assume there is a schedule of height K. Then there is a solution to the pinwheel problem $p^* = (p_1^*, ..., p_n^*)$ and thus $\sum_{i=1}^{n} \frac{1}{p_i^*} \leq 1$. If $p_1^* = 1$ then there is only one plant, and the schedule is trivial. Thus we consider $p_1^* \geq 2$. The algorithm calculates the periods $p_i = \lfloor \frac{10 p_i^*}{7} \rfloor$ and solves the resulting pinwheel problem. Because of the definition of p_i^* the resulting schedule has height at most $\frac{10K}{7}$.

We show that there is always a schedule for the pinwheel problem of ALG. If $p_i^* \geq 11$ then $\frac{1}{p_i}$ is smaller than $\frac{1}{p_i^*}$ by a factor of at least $\frac{3}{4}$ since $\frac{1}{p_i} = \frac{1}{\lfloor 10/7 p_i^* \rfloor} \leq \frac{1}{(10/7 p_i^* - 1)} \leq \frac{3}{4} \frac{1}{p_i^*}$ for $p_i^* \geq 11$.

The same is true for every other p_i^* except $p_i^* = 2$ and $p_i^* = 4$ which can easily be verified. Thus if there is no p_i^* with value 2 or 4 the density of the pinwheel-problem is at most $3/4$ and it can be scheduled. We now consider a variety of cases where these values occur and use Lemma 41 to show that the pinwheel-problem can be solved.

We denote these cases by listing the initial values of the instance p^*. Then for example the notation "4,4,5" means that we are considering the case where $p_1^* = p_2^* = 4$ and $p_3^* = 5$. This does not mean that the instance consists of only three periods but that these are the initial periods that we are looking to create a porous schedule for.

For each case, after creating a porous schedule, we need to show that $\frac{h}{p} \geq D_a$ as well as $\frac{1}{p} \leq \frac{3}{4p^*}$ holds for all periods that are not yet scheduled by the porous schedule.

2 We have $p/p^* \geq 6/5$ for all p. This can easily be verified for $p^* = 2$ and $p^* = 4$ and it holds for all other periods because for those periods we have $p/p^* \geq 4/3$. This means the density in the pinwheel-problem of ALG is at most $5/6$. Since $p_1 = 2$ this pinwheel-problem can be solved.

3,3,4 and 3,4,4 In this case there are only three plants.

$3, 4, p_3^* \geq 5$ Then $p = (4, 5, p_3 \geq 7, \dots)$. ALG schedules plants 1 and 2 using the schedule 1020. Then $D_A = 5/12$ and $\frac{h}{p} = \frac{\lfloor \frac{1}{2} \lfloor \frac{10}{7} p^* \rfloor \rfloor}{\lfloor \frac{10}{7} p^* \rfloor}$. Let $p^* = 7a + b$. Then $\frac{h}{p} = \frac{5a + \lfloor \frac{1}{2} \lfloor \frac{10}{7} b \rfloor \rfloor}{10a + \lfloor \frac{10}{7} b \rfloor}$.

If $\frac{5a + \lfloor \frac{1}{2} \lfloor \frac{10}{7} b \rfloor \rfloor}{10a + \lfloor \frac{10}{7} b \rfloor} \geq D_A$ then $\frac{5(a+1) + \lfloor \frac{1}{2} \lfloor \frac{10}{7} b \rfloor \rfloor}{10(a+1) + \lfloor \frac{10}{7} b \rfloor} \geq D_A$ since $\frac{5}{10} > D_A$.

Therefore if the inequality holds for a particular p^*, then it also holds for all subsequent p^* that have the same remainder after division by 7. In particular, this means that if there are 7 consecutive periods p^* with $h/p \geq D_A$ then $h/p \geq D_A$ also holds for all subsequent periods p^*.

We can determine that $h/p \geq D_A$ holds for $p^* \geq 5$ by looking at enough periods (in this case periods 5 to 11). This argument is repeated in many of the subsequent cases, and appendix A contains tables that show the values for each case.

Furthermore $\frac{1}{p_i} \leq \frac{3}{4p_i^*}$ holds for $p^* \geq 5$. This means we can schedule this case.

4,4 Then $p = (5, 5, \dots)$. ALG schedules plants 1 and 2 using the schedule 10200. Then $D_A = 1/2$. For $p^* = 7a + b$ we get $\frac{h}{p} = \frac{6a + h(\lfloor \frac{10}{7} b \rfloor)}{10a + \lfloor \frac{10}{7} b \rfloor}$. By looking at 7 periods we can determine that $h/p \geq D_A$ holds for $p^* \geq 4$. Since $\frac{1}{p_i} \leq \frac{3}{4p_i^*}$ holds for $p^* \geq 5$ but not $p^* = 4$ we still need to consider the case where $p_1^* = p_2^* = p_3^* = 4$ but we can schedule this case for $p_3^* \geq 5$.

4,4,4 Then $p = (5,5,5,\dots)$. ALG schedules plants 1 to 3 using the schedule 12300. Then $D_A = 1/4$. Using the same approach as in the previous case we can determine that $h/p \geq D_A$ holds for $p^* \geq 4$. As in the previous case this means we can schedule all cases except the case where $p_1^* = \dots = p_4^* = 4$ since in this case $\frac{1}{p_i} \leq \frac{3}{4p_i^*}$ is not guaranteed. However, in this case there are only four plants and therefore this case can be scheduled as well.

In all subsequent cases, all periods except the first are greater than 4. This means $\frac{1}{p_i} \leq \frac{3}{4p_i^*}$ holds for all periods after the first and we only need to verify $h/p \geq D_A$.

4,5 Then $p = (5,7,\dots)$. ALG again uses the schedule 10200 but we now have $D_A = 11/20$. In this case $h/p \geq D_A$ holds for $p^* \geq 9$

4,5,6 Then $p = (5,7,8,\dots)$ In this case ALG schedules the first three plants with schedule 31002 01300 21003 01200 and $D_A = 23/60$. Choosing $p^* = 14a + b$ we get $\frac{h}{p} = \frac{10a+h(\lfloor \frac{10}{7}b \rfloor)}{20a+\lfloor \frac{10}{7}b \rfloor}$. We can verify that $h/p \geq D_A$ for $p^* \geq 7$

4,5,6,6 Then $p = (5,7,8,8)$. This case follows from $p_1^* = 3, p_2^* = 4, p_3^* = 5$.

4,5,7 Then $p = (5,7,10,\dots)$. Follows from the case below with the same schedule and smaller D_A.

4,5,8 Then $p = (5,7,11,\dots)$ ALG uses the schedule 21030 01200 01032 01000 and $D_A = 17/40$. Choosing $p^* = 14a + b$ we get $\frac{h}{p} = \frac{11a+h(\lfloor \frac{10}{7}b \rfloor)}{20a+\lfloor \frac{10}{7}b \rfloor}$. $h/p \geq D_A$ holds for $p^* \geq 7$.

4,6 Then $p = (5,8,\dots)$. ALG uses the schedule 21000 01020 01000 and $D_A = 7/12$. Choosing $p^* = 21a + b$ we get $\frac{h}{p} = \frac{20a+h(\lfloor \frac{10}{7}b \rfloor)}{30a+\lfloor \frac{10}{7}b \rfloor}$. $h/p \geq D_A$ holds for $p^* \geq 9$.

4,6,6 Then $p = (5,8,8,\dots)$. Follows from the case below with the same schedule and smaller D_A.

4,6,7 Then $p = (5,8,10\dots)$. ALG uses the schedule 21003 01020 01300 and $D_A = \frac{37}{84}$. Choosing $p^* = 21a + b$ we get $\frac{h}{p} = \frac{16a+h(\lfloor \frac{10}{7}b \rfloor)}{30a+\lfloor \frac{10}{7}b \rfloor}$. $h/p \geq D_A$ holds for $p^* \geq 6$.

4,6,8 Then $p = (5,8,8,\dots)$. ALG uses the schedule 21003 01020 01003 21000 01023 01000 and $D_A = \frac{11}{24}$. Choosing $p^* = 21a + b$ we get $\frac{h}{p} = \frac{17a+h(\lfloor \frac{10}{7}b \rfloor)}{30a+\lfloor \frac{10}{7}b \rfloor}$. $h/p \geq D_A$ holds for $p^* \geq 9$.

4,6,8,8 Then $p = (5,8,11,11,\dots)$. ALG uses the schedule 21003 01024 01003 21004 01023 01004 and $D_A = \frac{1}{3}$. Choosing $p^* = 21a + b$ we get $\frac{h}{p} = \frac{14a+h(\lfloor \frac{10}{7}b \rfloor)}{30a+\lfloor \frac{10}{7}b \rfloor}$. $h/p \geq D_A$ holds for $p^* \geq 4$.

4,7 Then $p = (5, 10, \dots)$. Follows from the case below with the same schedule and smaller D_A.

4,8 Then $p = (5, 11, \dots)$. ALG uses the schedule 10200 10000 and $D_A = 5/8$. Choosing $p^* = 7a + b$ we get $\frac{h}{p} = \frac{7a + h(\lfloor \frac{10}{7} b \rfloor)}{10a + \lfloor \frac{10}{7} b \rfloor}$. In this case $h/p \geq D_A$ holds for $p^* \geq 6$.

4, $\mathbf{p_2^*} \geq$ 9 Then $p = (5, p_2 \geq 12, \dots)$. ALG schedules plant 1 with the simple schedule 10000. Then $D_A = \frac{3}{4}$ and we get $\frac{h}{p} = \frac{\lfloor \frac{4}{5} \lfloor \frac{10}{7} p^* \rfloor \rfloor}{\lfloor \frac{10}{7} p^* \rfloor}$. Setting $p^* = 7a + b$ with $a \in \mathbb{N}, b \leq 6$ we get $\frac{h}{p} = \frac{8a + \lfloor \frac{4}{5} \lfloor \frac{10}{7} b \rfloor \rfloor}{10a + \lfloor \frac{10}{7} b \rfloor}$.

$h/p \geq D_A$ holds for $p^* \geq 9$.

□

Theorem 43. *Algorithm 13 is a $\frac{10}{7}$-approximation.*

Proof. This follows directly from Lemma 42. Since any height K that is at least the optimal height is schedulable, the procedure in Algorithm 13 finds a valid schedule of height $\frac{10}{7} K$ for any height that is at least OPT. This means the binary search settles on a value that is at most OPT and the algorithm returns a schedule of height at most $\frac{10}{7}$OPT. □

It is possible to achieve a better approximation ratio than $\frac{10}{7}$ by setting R in Algorithm 13 to a lower value but this requires a much larger case analysis. This is because the periods p_i will be smaller. In the next section we present a case analysis that, using some amount of computer assistance, achieves a ratio of $\frac{7}{5}$. However, the length and complexity of this proof encourages finding new ideas for further improvements.

5.3 A 7/5-approximation for discrete BGT

By using $R = \frac{7}{5}$ instead of $R = \frac{10}{7}$ in Algorithm 13 we can improve the approximation ratio to $\frac{7}{5}$. The proof remains mostly the same but we will use some computer assistance to verify the individual cases. The following lemma will help determine how long to run our program.

Lemma 44. *Given a porous schedule of length λ, whenever there are 5λ consecutive periods with $\frac{h}{p} \geq D_A$, then $h_p \geq D_A$ also holds for all subsequent periods.*

Proof. Let $p^* = 5\lambda a + b$. Then $\frac{h}{p} = \frac{7h(\lambda)a + h(\lfloor \frac{10}{7} b \rfloor)}{7\lambda a + \lfloor \frac{10}{7} b \rfloor}$.

Assume $\frac{h(\lambda)}{\lambda} \geq D_A$. In this case if $\frac{h}{p} = \frac{7h(\lambda)a + h(\lfloor \frac{10}{7} b \rfloor)}{7\lambda a + \lfloor \frac{10}{7} b \rfloor} \geq D_A$ then we also have $\frac{h}{p} = \frac{7h(\lambda)(a+1) + h(\lfloor \frac{10}{7} b \rfloor)}{7\lambda(a+1) + \lfloor \frac{10}{7} b \rfloor} \geq D_A$ and the result follows.

On the other hand, if $\frac{h(\lambda)}{\lambda} < D_A$, then by setting $b = 0$ we also have $\frac{7h(\lambda)a}{7\lambda a} < D_A$ for any $a \in \mathbb{N}$, which means there can never be 5λ consecutive periods with $\frac{h}{p} \geq D_A$. $\square$

An overview of the program used to assist the following proof can be found in Appendix B. Furthermore, the program is also available under the following link: https://github.com/Felixhhne/bamboo.

Theorem 45. *Algorithm 13 with $R = \frac{7}{5}$ is a $\frac{7}{5}$-approximation.*

Proof. Let $p_i^* = \lfloor \frac{K}{v_i} \rfloor$ and $p_i = \lfloor \frac{7p_i^*}{5} \rfloor$. Assume there is a schedule of height K. Then there is a solution to the pinwheel problem $p^* = (p_1^*, ..., p_n^*)$ and thus $\sum_{i=1}^{n} \frac{1}{p_i^*} \leq 1$.

We need to show that there is always a schedule for the pinwheel problem $p = (p_1, \ldots, p_n)$. In this case $\frac{1}{p_i} \leq \frac{3}{4p_i^*}$ holds for all periods except 2, 4 and 7.

We consider all cases that involve these periods by presenting porous schedules and determining for which periods p^* we have $\frac{h}{p} \geq D_A$.

Verifying these cases involves looking at a large number of periods with the help of the program in appendix B.

To keep the cases short we simply state the schedule and a value p^* such that $\frac{h}{p} \geq D_A$ holds for this p^* and all subsequent periods. We may also omit the value of p^* if a schedule completely solves a case and no subcases need to be considered.

2 We have $p/p^* \geq 6/5$ for all p. This means the density in the pinwheel-problem of ALG is at most $5/6$. Since $p_1 = 2$ this pinwheel-problem can be solved.

3,3,4 and 3,4,4 In these cases there are only three plants.

$3, 4, p_3^* \geq 5$ we schedule plants 1 and 2 using the schedule 1020. $\frac{h}{p} \geq D_A$ holds for $p^* \geq 5$. This solves all cases beginning with 3 and 4 that do not include 7 but we still need to consider cases that do include 7.

3,4,5,7 In this case there are only 4 plants. We give a short argument: Consider 7 consecutive positions and assume plant 5 is scheduled in position 4. Clearly plants 1 and 2 must each occur twice, once to the left and right of plant 5. Then plants 3 and 4 can each only occur once in these 7 positions. However, if plant 3 is to occur only once it must be in position 3 or 5 (w.l.o.g position 3). Then plant 1 must occur in positions 2 and 5 which leaves not enough room to schedule plant 2.

3,4,6 and 3,4,7 using the schedule 13201020 we get $p^* \geq 9$.

3,4,6,6 to 3,4,7,8 using the schedule 13201420 we get $p^* \geq 6$. Any additional periods of value 7 would lead to a negative density.

3,7 using the schedule 1020 1000 we get $p^* \geq 5$. (This also solves the cases 3,5,7 and 3,6,7.)

3,7,7 using the schedule 1020 1030 we get $p^* \geq 2$.

3,7,7,7 using the schedule 1024 1030 we get $p^* \geq 5$.

3,7,7,7,7 using the schedule 1024 1035 we get $p^* \geq 5$. Any additional periods of value 7 would lead to a negative density.

4 using the schedule 10000 we get $p^* \geq 9$.

4,4 follows from 2.

4,5 using the schedule 12000 10020 10002 10000 we get $p^* \geq 9$.

4,5,5 using the schedule 12003 10020 13002 10300 we get $p^* \geq 4$.

4,5,5,7 using the schedule 12043 10020 13402 10304 12003 10420 13002 14300 we get $p^* \geq 5$. This also solves any case where there are additional periods 5 or 6 before the 7.

4,5,5,7,7 using the schedule 12043 15020 13452 10304 12503 10425 13002 14350 we get $p^* \geq 5$. Any additional periods of value 7 would lead to a negative density.

4,5,6 using the schedule 31002 01300 21003 01200 we get $p^* \geq 7$.

4,5,6,6 follows from $p_1^* = 3, p_2^* = 4, p_3^* = 5$.

4,5,6,7 using the schedule 12043 10020 13402 10034 12000 13420 10032 14030 we get $p^* \geq 8$.

4,5,6,7,7 using the schedule 12043 10520 13402 15034 12005 13420 10532 14035 we get $p^* \geq 5$. Any additional periods of value 7 would lead to a negative density.

4,5,7 using the schedule 12030 10020 10302 10003 12000 10320 10002 13000 we get $p^* \geq 7$.

4,5,7,7 using the schedule 12430 10020 14302 10043 12000 14320 10042 13000 we get $p^* \geq 10$.

4,5,7,7,7 to 4,5,7,7,10 using the schedule 12430 15020 14302 15043 12005 14320 10542 13005 we get $p^* \geq 7$. Any additional periods of value 7 would lead to a negative density.

4,5,8 using the schedule 21030 01200 01032 01000 we get $p^* \geq 7$.

4,6 using the schedule 21000 01020 01000 we get $p^* \geq 9$.

4,6,6 follows from $3, 4$.

4,6,7 using the schedule 1020 1030 we get $p^* \geq 6$.

4,6,7,7 using the schedule 10304 10200 13042 10003 12400 10032 14002 we get $p^* \geq 7$. Any other cases involving 4, 6 and multiple periods 7 follow from the cases with 4 and multiple periods 7 below.

4,6,8 using the schedule 21030 01020 01003 21000 01023 01000 we get $p^* \geq 9$.

4,6,8,8 using the schedule 21003 01024 01003 21004 01023 01004 we get $p^* \geq 4$.

4,7 using the schedule 10200 10000 12000 10002 10000 10020 10000 we get $p^* \geq 10$.

4,7,7 and 4,7,8 using the schedule 10203 10000 12030 10002 10300 10020 13000 we get $p^* \geq 9$.

4,7,7,7 to 4,7,8,8 using the schedule 10204 10300 12040 13002 10403 10020 14030 we get $p^* \geq 6$.

4,7,7,7,7 using the schedule 15204 10305 12040 13052 10403 10520 14030 we get $p^* \geq 3$.

4,7,7,7,7,7 using the schedule 15204 10365 12040 13652 10403 16520 14036 we get $p^* \geq 5$. Any additional periods of value 7 would lead to a negative density.

4,7,9 using the schedule 10203 10000 12000 13002 10000 10320 10000 we get $p^* \geq 9$.

4,8 using the schedule 10200 10000 we get $p^* \geq 6$.

5,7 using the schedule 100200 we get $p^* \geq 9$.

5,5,7 and 5,6,7 using the schedule 1020300 we get $p^* \geq 5$. This also solves any case where there are additional periods 5 or 6 before the 7.

5,7,7 follows from case $7,7$ below. The same is true for any other case involving 5 and multiple periods 7 except the following case.

5,7,7,7,7 using the schedule 503012040 513002041 we get $p^* \geq 5$.

6,7 using the schedule 10002000 we get $p^* \geq 8$.

6,6,7 follows from $3,7$.

6,7,7 follows from case $7,7$ below. The same is true for all other cases involving 6 and multiple periods of 7.

7 using the schedule 100000000 we get $p^* \geq 15$.

7,7 using the schedule 100002000 we get $p^* \geq 5$. Any other case involving multiple periods of 7 can be solved with analogous simple schedules. This also solves any case with additional periods 5 or 6 before the 7s.

7,8 using the schedule 120000000 100200000 100002000 100000020 100000002 100000000 we get $p^* \geq 17$.

7,8,8 follows from $4,7$.

7,8,9 using the schedule 100020030 100000200 130020000 100030200 we get $p^* \geq 12$.

7,8,9,9 to 7,8,9,11 using the schedule 140020030 100040200 130020040 100030200 we get $p^* \geq 7$.

7,8,10 using the schedule 100020030 100000200 130020000 100030200 we get $p^* \geq 15$.

7,8,10,10 using the schedule 140020030 100040200 130020040 100030200 we get $p^* \geq 7$.

7,8,10,11 using the schedule 140020030 100040200 130020040 100030200 we get $p^* \geq 12$.

7,8,10,11,11 using the schedule 140025030 100040205 130020040 105030200 we get $p^* \geq 13$.

7,8,10,11,11,11 and 7,8,10,11,11,12 using the schedule 140025030 106040205 130026040 105030206 we get $p^* \geq 9$.

7,8,10,12 using the schedule 103040200 100000230 140000200 100300240 100003200 we get $p^* \geq 17$.

7,8,10,12,12 and 7,8,10,12,13 solved using the schedule 120000003 150200400 100032005 104000023 100000502 140030000 120500043 100200000 150432000 100000523 140000002 100530004.

7,8,10,12,14 using the schedule 120000053 100200400 100032005 104000023 100000052 140030000 120000543 100200000 100432050 100000023 140000052 100030004 we get $p^* \geq 16$.

7,8,10,12,14,14 and 7,8,10,12,14,15 using the schedule 120000653 100200400 100032605 104000023 100006052 140030000 120006543 100200000 100432650 100000023 140000652 100030004 we get $p^* \geq 12$.

7,8,10,12,15 and 7,8,10,12,16 solved using the same schedules as $7, 8, 10, 12, 14$.

7,8,10,13 and 7,8,10,14 using the schedule 102004003 102000000 102034000 102000003 102004000 102030000 we get $p^* \geq 12$.

7,8,11 using the schedule 100000003 120000000 100203000 100002000 103000020 100000023 100000000 120003000 100200000 103002000 100000023 100000002 100003000 120000000 103200000 100002003 100000020 100003002 100000000 123000000 100200003 100002000 100003020 100000002 103000000 120000003 100200000 100023000 100000200 103000002 we get $p^* \geq 17$.

7,8,11,11 using the schedule 124000003 100200004 100023000 100004200 103000002 104000003 120000004 100203000 100024000 103000200 104000023 100000024 100003002 100004002 103000000 we get $p^* \geq 10$.

7,8,11,12 and 7,8,11,13 solved using the schedule 124000003 100200004 100023000 100004200 103000002 104000003 120000004 100203000 100024000 103000200 104000023 100000024 100003002 100004002 103000000.

7,8,11,14 to 7,8,11,16 using the schedule 100040003 120000000 102043000 100200000 103024000 100020003 100042000 100003200 100040020 103000002 we get $p^* \geq 13$.

7,8,12 using the same schedule as $7, 8, 11$ we get $p^* \geq 17$.

7,8,12,12 using the same schedule as $7, 8, 11, 11$ we get $p^* \geq 18$.

7,8,12,12,12 using the schedule 124050003 100200004 150023000 100004250 103000002 104050003 120000004 150203000 100024050 103000200 104050023 100000024 150003002 100004052 103000000 we get $p^* \geq 18$.

7,8,12,12,12,12 using the schedule 124050003 100260004 150023000 160004250 103000062 104050003 120060004 150203000 160024050 103000260 104050023 100060024 150003002 160004052 103000060 we get $p^* \geq 9$.

7,8,12,12,12,13 to 7,8,12,12,12,17 solved using the schedules for $7, 8, 13$ to $7, 8, 17$.

7,8,12,12,13 to 7,8,12,12,17 solved using the schedules for $7, 8, 13$ to $7, 8, 17$.

7,8,12,13 to 7,8,12,16 solved using the schedules for $7, 8, 13$ to $7, 8, 17$.

7,8,13 to 7,8,15 using the schedule 120000000 100203000 100002000 100003020 100000002 100003000 we get $p^* \geq 11$

7,8,16 and 7,8,17 using the same schedule as above we get $p^* \geq 12$.

7,9 using the schedule 100002000 100000002 100000000 102000000 we get $p^* \geq 15$.

7,9,9 using the schedule 100002000 103000002 100003000 102000003 we get $p^* \geq 7$.

7,9,10 to 7,9,12 using the same schedule as above we get $p^* \geq 11$.

7,9,10,10 using the schedule 140002000 103040002 100003040 102000003 we get $p^* \geq 12$.

7,9,10,10,10 and 7,9,10,10,11 using the schedule 140052000 103040052 100003040 152000003 we get $p^* \geq 11$.

7,9,10,10,10,10 using the schedule 140052060 103040052 160003040 152060003 we get $p^* \geq 8$.

7,9,13 and 7,9,14 using the schedule 120003000 100020000 100003020 100000000 we get $p^* \geq 12$.

7,10 using the schedule 100000002 100000000 100020000 we get $p^* \geq 15$.

7,10,10 and 7,10,11 using the schedule 100030002 100000003 100020000 we get $p^* \geq 9$

7,10,12 using the schedule 100000023 100000000 100200300 100000020 100030000 100200000 103000020 100000003 100200000 100000320 100000000 100230000 100000020 103000000 100200003 100000020 100000300 100200000 100030020 100000000 103200000 we get $p^* \geq 16$.

7,10,12,12 follows from $p_1 = 6$, $p_2 = 7$ and $p_3 = 10$.

7,10,12,13 and 7,10,12,14 using the schedule 100004023 100000000 100204300 100000020 100034000 100200000 103004020 100000003 100204000 100000320 100004000 100230000 100004020 103000000 100204003 100000020 100004300 100200000 100034020 100000000 103204000 100000023 100004000 100200300 100004020 100030000 100204000 103000020 100004003 100200000 100004320 100000000 100234000 100000020 103004000 100200003 100004020 100000300 100204000 100030020 100004000 103200000 we get $p^* \geq 15$.

7,10,12,13,13 to 7,10,12,14,14 using the schedule 100004023 100005000 100204300 100005020 100034000 100205000 103004020 100005003 100204000 100005320 100004000 100235000 100004020 103005000 100204003 100005020 100004300 100205000 100034020 100005000 103204000 100005023 100004000 100205300 100004020 100035000 100204000 103005020 100004003 100205000 100004320 100005000 100234000 100005020 103004000 100205003 100004020 100005300 100204000 100035020 100004000 103205000 we get $p^* \geq 15$.

7,10,12,13,13,13 to 7,10,12,14,14,14 using the schedule 100004023 160005000 100204300 160005020 100034000 160205000 103004020 160005003 100204000 160005320 100004000 160235000 100004020 163005000 100204003 160005020 100004300 160205000 100034020 160005000 103204000 160005023 100004000 160205300 100004020 160035000 100204000 163005020 100004003 160205000 100004320 160005000 100234000 160005020 103004000 160205003 100004020 160005300 100204000 160035020 100004000 163205000 we get $p^* \geq 9$.

7,10,13 solved using the schedule for $7, 13$.

7,10,14 using the schedule 100203000 100000020 100003000 100200000 100003002 100000000 we get $p^* \geq 15$.

7,10,14,14 using the schedule 100203000 100004020 100003000 100204000 100003002 100004000 we get $p^* \geq 6$.

7,11 using the schedule 100002000 100000000 102000000 100000002 100000000 we get $p^* \geq 16$.

7,11,11 using the schedule 103002000 100000003 102000000 100003002 100000000 we get $p^* \geq 16$.

7,11,11,11 using the schedule 103002040 100000003 102040000 100003002 140000000 we get $p^* \geq 16$.

7,11,11,11,11 using the schedule 103002045 100000003 102045000 100003002 145000000 we get $p^* \geq 9$.

7,11,11,11,12 using the same schedule as $7, 11, 11, 11, 11$ we get $p^* \geq 17$.

7,11,11,11,12,12 using the schedule 103002045 160000003 102045060 100003002 145060000 we get $p^* \geq 10$.

7,11,11,11,12,13 to 7,11,12,12,12,15 using the schedule 103002045 100600003 102045000 100603002 145000000 103602045 100000003 102645000 100003002 145600000 we get $p^* \geq 10$.

7,11,11,11,12,16 using the schedule 103002045 100600003 102045000 100603002 145000000 103602045 100000003 102645000 100003002 145600000 we get $p^* \geq 18$.

7,11,11,11,12,16,16 and 7,11,11,11,12,16,17 using the schedule 103702045 100600003 102745000 100603002 145700000 103602045 100700003 102645000 100703002 145600000 we get $p^* \geq 9$.

7,11,11,11,13 using the schedule 103002040 100500003 102040000 100503002 140000000 103502040 100000003 102540000 100003002 140500000 we get $p^* \geq 10$.

7,11,11,11,14 and 7,11,11,11,15 using the same schedule as $7, 11, 11, 11, 13$ we get $p^* \geq 10$.

7,11,11,11,16 using the same schedule as $7, 11, 11, 11, 13$ we get $p^* \geq 18$

7,11,11,11,16,16 using the schedule 103602040 100500003 102640000 100503002 140600000 103502040 100600003 102540000 100603002 140500000 we get $p^* \geq 9$.

7,11,11,12 using the same schedule as $7, 11, 11, 11$ we get $p^* \geq 17$.

7,11,11,12,12 follows from $6, 7, 11$.

7,11,11,12,13 and 7,11,11,12,14 using the schedule for $7, 11, 11, 11, 13$ we get $p^* \geq 10$.

7,11,11,12,15 and 7,11,11,12,16 The first case is solved using the same schedule as $7, 11, 11, 11, 13$ in the second case we get $p^* \geq 18$.

7,11,11,12,16,16 and 7,11,11,12,16,17 using the same schedule as $7, 11, 11, 11, 16, 16$ we get $p^* \geq 9$.

7,11,11,13 using the schedule for case $7, 13$ we get $p^* \geq 10$ which solves this case.

7,11,11,14 and 7,11,11,15 using the schedule 103042000 100000003 102040000 100003002 100040000 103002000 100040003 102000000 100043002 100000000 we get $p^* \geq 10$.

7,11,12 using the same schedule as $7, 11, 11$ we get $p^* \geq 17$.

7,11,12,12 follows from $6, 7, 11$.

7,11,12,13 using the schedule for case $7, 13$ we get $p^* \geq 10$ which solves this case.

7,11,12,14 to 7,11,12,16 using the schedule 103042000 100000003 102040000 100003002 100040000 103002000 100040003 102000000 100043002 100000000 we get $p^* \geq 18$.

7,11,12,14,14 to 7,11,12,16,17 using the schedule 103042000 100500003 102040000 100503002 100040000 103502000 100040003 102500000 100043002 100500000 we solve cases $7, 11, 12, 14, 14$ and $7, 11, 12, 14, 15$ and in the other cases we get $p^* \geq 17$.

7,11,12,14,16,16 to 7,11,12,16,16,16 These cases can be solved using the schedule 103042000 100500603 102040000 100503602 100040000 103502600 100040003 102500600 100043002 100500600.

7,11,13 using the schedule for case $7, 13$ we get $p^* \geq 10$ which solves this case.

7,11,14 and 7,11,15 using the schedule 100302000 100000000 102300000 100000002 100300000 100002000 100300000 102000000 100300002 100000000 we get $p^* \geq 17$.

7,11,14,14 to 7,11,15,16 using the schedule 100302000 100400000 102300000 100400002 100300000 100402000 100300000 102400000 100300002 100400000 solves all cases except $7, 11, 15, 16$ where we get $p^* \geq 17$.

7,11,15,16,16 using the schedule 100302050 100400000 102300050 100400002 100300050 100402000 100300050 102400000 100300052 100400000 we get $p^* \geq 16$.

7,12 using the schedule 100000002 100000000 100000200 100000000 100020000 100000000 102000000 we get $p^* \geq 17$.

7,12,12 follows from $6, 7$.

7,12,13 using the schedule for case $7, 13$ we get $p^* \geq 10$ which solves this case.

7,12,14 using the schedule 100003002 100000000 100003200 100000000 100023000 100000000 102003000 100000002 100003000 100000200 100003000 100020000 100003000 102000000 we get $p^* \geq 18$.

7,12,14,14 to 7,12,14,17 solved using the schedule 100003002 100004000 100003200 100004000 100023000 100004000 102003000 100004002 100003000 100004200 100003000 100024000 100003000 102004000.

7,12,15 and 7,12,16 using the schedule 100030002 100000000 100000230 100000000 100020000 130000000 102000000 we get $p^* \geq 18$.

7,12,15,15 to 7,12,16,17 using the schedule 100030002 100040000 100000230 100000040 100020000 130000000 142000000 we solve all cases except $7, 12, 16, 17$ where we get $p^* \geq 18$

7,12,16,17,17 using the schedule 100030002 105040000 100000230 100005040 100020000 130000005 142000000 we get $p^* \geq 17$.

7,13 using the schedule 100002000 100000000 we get $p^* \geq 10$.

7,14 using the same schedule as above we get $p^* \geq 18$.

7,14,14 to 7,14,17 solved using the schedule 100002000 100003000.

□

5.4 Continuous BGT on a star

We now examine the continuous case of the BGT-problem. For this problem Gasieniec et al. propose a $O(\log n)$-approximation algorithm that works in any metric space.

In this chapter we consider a metric space that is the star graph and show that many of the constant approximation factor algorithms for discrete BGT can be extended to the star graph with only small losses in the approximation ratio.

5.4.1 Notation

Consider a star with n branches and k_i bamboo on branch i. We number the plants on each branch starting with the closest plant from the center. This means plant 1 on branch i is closer to the center than plant 2 and so on.

Let h_{ij} be the speed of the j-th plant on branch i, and δ_{ij} be twice the distance of the j-th plant on branch i to the $j-1$-th plant or to the center for $j = 1$ while d_{ij} is twice the distance to the center itself.

We always consider twice the distance because on the star graph the algorithm always returns to the center which means edges are traversed in both directions every time. In the remainder of the chapter we often simply use the term distance when referring to d_{ij} or δ_{ij}.

We assume that the speeds are decreasing on each branch. (If there is a plant a that is further away than plant b on a branch and has the same speed or more, then plant b gets visited whenever plant a is visited.) When talking about the star with only one plant on each branch we may simply write δ_i or d_i for the distances and h_i for the speed of the plant on branch i.

5.4.2 Fractional lower bounds

We now present a lower bound on the maximum height of a plant for any algorithm. To find this lower bound we allow the algorithm to traverse the edges at fractional periods. We assume that on average we visit one branch each round (one round means travelling into a branch, visiting nodes and then returning to the center). Furthermore, if a plant a is behind a plant b on a branch, the edge towards a cannot be traversed at a shorter period than b. Whenever an edge towards a plant is traversed, that plant is cut.

An algorithm in this setting has a big advantage compared to an algorithm in the regular setting. Thus a lower bound on this setting is also a lower bound in the regular setting.

We illustrate the concept on a small example. Let there be a star with three branches. On branch 1 we have first a plant of speed $\frac{1}{2}$ at distance 1 and then a plant of speed $\frac{1}{3}$ at distance 2. On branch 2 we have a plant of speed $\frac{1}{8}$ at distance 1. On branch 3 we have a plant of speed $\frac{3}{8}$ at distance 1.

We now set the periods that an edge is traversed with to be equal to the inverse of the speed of the plant. This means the first plant on the first branch gets visited every second round. The second plant on the first branch gets visited every third round (despite this not being feasible in the original model since 3 is not divisible by 2.) The plant on the second branch gets visited every 8 rounds and the plant on branch 3 gets visited every $\frac{8}{3}$ rounds.

How high does the first plant on the first branch grow? Let us call this plant P. Since at some point we visit the second plant on this branch and traverse the full length of 1 of the edge towards it and back, the height of P is at least $\frac{1}{2}$.

However, in this example the height is determined by the time the algorithm spends cutting plants on other branches. This means we consider the time in between cuts of P when the server moves back to the center. Since we visit P every two rounds and the plant on the second branch every 8 rounds, we can (fractionally) fit in $\frac{2}{8} = \frac{1}{4}$ visits towards the plant on the second branch. Similarly we traverse the distance towards the plant on the third branch $\frac{2}{8/3} = \frac{3}{4}$ times between cuts of P. Thus plant P grows to a height of $\frac{1}{2}(1 + \frac{1}{4} + \frac{3}{4}) = 1$ in this model.

Let edge i be traversed at a (fractional) period of f_i. Consider some plant p with speed h_p and let A be the set of plants that are behind p on the branch, while B is the set of plants towards the center on the same branch and on other branches. Since we need to travel to the end of the branch at some point, the height of p is at least $h_p \sum_{i\in A} \delta_i$.

Each plant in the set B and the edge leading up to it may be visited $\frac{f_p}{f_i}$ times between cuts of p. Additionally, the edge towards p with distance d_p gets travelled once. Therefore p grows to a height of at least $h_p(d_p + \sum_{i\in B} \frac{f_p}{f_i}\delta_i)$.

Consider the star graph with one plant on each branch. Set the periods $f_i := \frac{1}{h_i}$. Consider a plant p. All other plants are in set B. Then plant p grows to a height of $h_p(d_p + \sum_{i\in B} \frac{f_p}{f_i} d_i) = h_p(d_p + \sum_{i\in B} \frac{h_i}{h_p} d_i) = h_p d_p + \sum_{i\in B} h_i d_i = \sum_{i=1}^{n} h_i d_i := R$. For the star with only one bamboo on each branch we may also write $R = \sum_{i=1}^{n} h_i \delta_i$ and this is also the definition we want to use for the star with multiple plants on each branch.

Now consider a star with multiple plants on each branch. Again set the periods to $f_i := \frac{1}{h_i}$. Consider a plant p and let the maximum height it reaches be H_p. Since we have to travel to the end of the branch at some point, we have $H_p \geq h_P \sum_{i \in A} \delta_i \geq \sum_{i \in A} h_i \delta_i$. The second inequality holds because the speeds along a branch are nonincreasing. Furthermore we get $H_p \geq h_p(\delta_p + \sum_{i \in B} \frac{f_p}{f_i} \delta_i) = h_p(d_p + \sum_{i \in B} \frac{h_i}{h_p} \delta_i) = h_p \delta_p + \sum_{i \in B} h_i \delta_i$

It follows that $2H_p \geq \sum_{i \in A} h_i \delta_i + h_p d_p + \sum_{i \in B} h_i \delta_i = R$ and thus $H_p \geq \frac{1}{2}R$.

For the periods $f_i := \frac{1}{h_i}$ all plants have height at least $\frac{1}{2}R$. It is not possible to achieve a lower maximum height because adjusting the periods such that one plant grows to a smaller height increases the height of all other plants. It follows that $\frac{1}{2}R$ is a lower bound on the star with multiple plants on each plant and for the same reasons R is a lower bound on the star graph.

The lower bound also does not improve if the algorithm changes periods in between rounds, because in that case we can instead consider the average periods and compute the average maximum heights which are a lower bound for the maximum height.

5.4.3 Algorithms on the star

We have $R = \sum_{i=1}^{n} d_i h_i$. Define $L = \max(R, Dh_1)$, where D is the diameter of the star and h_1 is the growth rate of the fastest bamboo. This is a lower bound on OPT.

The Deadline-Driven algorithm always cuts the bamboo with the earliest deadline among those with height at least L where the deadline is determined by some height the algorithm wants to maintain. This means whenever the algorithm visits the center it chooses the plant with the earliest deadline, travels towards this plant and cuts it before returning to the center. The algorithm does not turn around when a more urgent plant reaches height L.

We say a plant is *requested* when it reaches height L and before that this request is *initialized* at the time the plant has height 0. This happens either after a cut or at the beginning of the process. We choose the *deadline* as the time a bamboo reaches height $(2 + \sqrt{3})L$.

This means each request to cut a plant consists of an initialization time, a request time and a deadline.

Lemma 46. *The Deadline-Driven algorithm maintains a height of* $(2 + \sqrt{3})L$.

Proof. Assume plant i reaches height $(2 + \sqrt{3})L$ at time T and is not cut. We scale the time (and distance) such that $L/h_i = 1$ which means plant i grows by L in one timestep. This is possible because whenever we scale the length of all edges by a factor then the heights of all plants reached in any schedule is scaled by the same factor. This holds for both ALG and OPT which means the approximation ratio is unaffected. We can also see this as changing the timescale using L/h_i as our unit of time.

Let t_1 be the last time the algorithm is idle or busy processing a request with deadline after T. Let time 0 be the most recent time before t_1 where plant i has height 0.

This means in between time t_1 and T the algorithm is only processing requests with deadlines at or before T and it is not idle. Let the set of these requests be S and let v be the earliest request time among all request in S. We have $i \in S$, so $v \leq 1$ and $t_1 \geq v$.

We further divide the set S into old requests S_0 which are initialised before v and new requests S_1 which are initialised after v. The time required to process S_0 is $\sum_{j \in S_0} d_j$ because any bamboo with an old request must be visited once to fulfill the request. Afterwards the bamboo either has a deadline after T and is not visited again or becomes part of the bamboo with new requests. We next show that all bamboo with new requests also have an old request. Consider the earliest new request for a bamboo. This new request gets initialized between v and T. This means there was a previous request for that bamboo with a deadline between v and T. This request is an old request.

The time required to process S_1 is $\sum_{j \in S_1} m_j d_j$ where m_j is the number of new requests of bamboo j which are requests with initialization time after v and a deadline before or at T. (Here $j \in S$ means there is a request for bamboo j in the set of requests S. We may also see S as a multi-set of bamboo instead.)

It is possible that just before v a request with a deadline after T arrives and is processed by the algorithm. The algorithm traverses a distance of r to serve this request if it exists, otherwise $r = 0$. This means $t_1 = v + r$.

We now show that $\sum_{j \in S_0} d_j + \sum_{j \in S_1} m_j d_j + v + r < T$ which contradicts the assumption that plant i is not cut before time T.

We begin by finding upper bounds on the time required to process the requests in S.

A bamboo with an old request must grow by at least $(1 + \sqrt{3})L$ in time at most $T - v$ to have a deadline before T which means $h_j(T - v) \geq (1 + \sqrt{3})L$. Meanwhile plant i grows by $(T - v)L$ in time $(T - v)$, that is $h_i(T - v) = (T - v)L$. It follows that

$$\frac{h_j}{h_i} \geq \frac{1 + \sqrt{3}}{T - v} \text{ for } j \in S_0 \tag{5.1}$$

A bamboo with new requests must grow by $(2 + \sqrt{3})L$ exactly m_j times in time at most $T - v$ in order to have m_j deadlines before T. Then $h_j(T - v) \geq m_j(2 + \sqrt{3})$. It follows that

$$\frac{h_j}{h_i} \geq m_j \frac{2 + \sqrt{3}}{T - v} \text{ for } j \in S_1 \tag{5.2}$$

We can now find upper bounds for the processing times of the requests in S. We first get

$$\sum_{j \in S_0} d_j \overset{(5.1)}{\leq} \frac{1}{h_i} \frac{T - v}{1 + \sqrt{3}} \sum_{j \in S_0} d_j h_j < \frac{T - v}{1 + \sqrt{3}} \frac{R}{h_i}$$

and then

$$\sum_{j\in S_1} m_j d_j \overset{(5.2)}{\leq} \frac{1}{h_i}\frac{T-v}{2+\sqrt{3}}\sum_{j\in S_0} d_j h_j < \frac{T-v}{2+\sqrt{3}}\frac{R}{h_i}$$

Given the timescale and because $R \leq L$ it takes time less than $\frac{T-v}{1+\sqrt{3}}$ to process the old requests and less than $\frac{T-v}{2+\sqrt{3}}$ to process the new requests.

It remains to show $\frac{T-v}{1+\sqrt{3}} + \frac{T-v}{2+\sqrt{3}} + v + r \leq T$.

We have $r \leq L/h_1 \leq L/h_i = 1$ where the first inequality holds because otherwise plant 1 would grow by more than L in the time it takes to travel distance r, and the second follows from the ordering of the plants by their growth rate. Furthermore, since $v < 1$ and the earliest deadline of i is $2+\sqrt{3}$ we have $T - v \geq (1+\sqrt{3})$.

It follows

$$r \leq 1 = \left(\frac{1}{1+\sqrt{3}}\right)\left(1+\sqrt{3}\right) \leq \left(\frac{1}{1+\sqrt{3}}\right)(T-v)$$

and then

$$\begin{aligned}
& r \leq \left(\frac{1}{1+\sqrt{3}}\right)(T-v) \\
\Rightarrow & \left(\frac{1}{1+\sqrt{3}}\right)v + r \leq \left(\frac{1}{1+\sqrt{3}}\right)T \\
\Rightarrow & \left(\frac{1}{1+\sqrt{3}}\right)v + r \leq \left(\frac{1}{1+\sqrt{3}}\right)T + \left(1-\frac{1}{1+\sqrt{3}}\right)T - \left(1-\frac{1}{1+\sqrt{3}}\right)T \\
\Rightarrow & \left(1-\frac{1}{1+\sqrt{3}}\right)T + \left(\frac{1}{1+\sqrt{3}}\right)v + r \leq T \\
\Rightarrow & \left(1-\frac{1}{1+\sqrt{3}}\right)T - \left(1-\frac{1}{1+\sqrt{3}}\right)v + v + r \leq T \\
\Rightarrow & \frac{T-v}{1+\sqrt{3}} + \frac{T-v}{2+\sqrt{3}} + v + r \leq T
\end{aligned}$$

This means we can process all requests (including the request for b_i with deadline at time T) before time T and plant i does not grow above height $(2+\sqrt{3})L$. □

For the star with multiple plants on each branch we use the Deadline-Driven algorithm with a small modification. Whenever the algorithm visits a bamboo b_j on a branch it walks double the distance and cuts all additional bamboo it encounters. These bamboo can be removed from the graph since they are always cut together with b_j. Then the distance between b_j and the next bamboo b_i (i.e δ_i) on the branch is at least as much as the distance from b_j to the center (i.e. $d_i - \delta_i$). In particular it follows that $\delta_i \geq d_i - \delta_i$ and thus $d_i \leq 2\delta_i$.

Furthermore, for the bamboo with multiple plants R is not a lower bound, but $\frac{1}{2}R$ is, and therefore we define $L = \max(\frac{1}{2}R, Dh_1)$ to get a lower bound on OPT.

For the star with multiple plants on each branch, the deadline is the point in time a plant reaches height $(5+\sqrt{21})L$. The request time and the initialization time remain the same.

Lemma 47. *The modified deadline driven algorithm maintains a height of* $(5+\sqrt{21})L$

Proof. Assume plant i reaches height $(5+\sqrt{21})L$ at time T. We scale the time (and distance) such that $L/h_i = 1$ which means plant i grows by L in one time step. The old and new requests S_0 and S_1 as well as v and r are defined analogously to the proof for the star.

The time required to process S_0 is at most $\sum_{j\in S_0} d_j$ because any bamboo with an old request must be visited once to fulfill the request, and in the worst case we start from the center traversing distance d_j to fulfill the request. It is possible that the algorithm is more efficient whenever plants with consecutive deadlines lie behind each other on the same branch.

Similarly, the time required to process S_1 is at most $\sum_{j\in S_1} m_j d_j$ where m_j is the number of new requests on plant b_j.

Therefore in order to arrive at a contradiction we again need to show $\sum_{j\in S_0} d_j + \sum_{j\in S_1} m_j d_j + v + r < T$ which means plant i can be cut before time T.

Again we find upper bounds on the time required to process the old and new requests. These bounds are slightly different due to the changes in the algorithm as well as the lower bound.

A bamboo with an old request must grow by at least $(4+\sqrt{21})L$ in time at most $T-v$ to have a deadline before T which means $h_j(T-v) \geq (4+\sqrt{21})L$. Meanwhile plant i grows by $(T-v)L$ in time $(T-v)$, that is $h_i(T-v) = (T-v)L$. It follows that

$$\frac{h_j}{h_i} \geq \frac{4+\sqrt{21}}{T-v} \text{ for } j \in S_0 \tag{5.3}$$

A bamboo with new requests must grow by $(5+\sqrt{21})L$ exactly m_j times in time at most $T-v$ in order to have m_j deadlines before T. Then $h_j(T-v) \geq m_j(5+\sqrt{21})$. It follows that

$$\frac{h_j}{h_i} \geq m_j \frac{5+\sqrt{21}}{T-v} \text{ for } j \in S_1 \tag{5.4}$$

We now get

$$\sum_{j\in S_0} d_j \overset{(5.3)}{\leq} \frac{1}{h_i}\frac{T-v}{4+\sqrt{21}} \sum_{j\in S_0} d_j h_j \leq \frac{1}{h_i}\frac{T-v}{4+\sqrt{21}} \sum_{j\in S_0} 2\delta_j h_j < \frac{T-v}{4+\sqrt{21}}\frac{2R}{h_i} \leq \frac{T-v}{4+\sqrt{21}}\frac{4L}{h_i}$$

where we are using $d_i \leq 2\delta_i$ in the second inequality and $R \leq 2L$ in the last inequality.

Analogously the following holds for the time required to process the new requests

$$\sum_{j\in S_1} m_j d_j \overset{(5.4)}{\leq} \frac{1}{h_i}\frac{T-v}{5+\sqrt{21}}\sum_{j\in S_0} d_j h_j \leq \frac{1}{h_i}\frac{T-v}{5+\sqrt{21}}\sum_{j\in S_0} 2\delta_j h_j < \frac{T-v}{5+\sqrt{3}}\frac{2R}{h_i} \leq \frac{T-v}{5+\sqrt{21}}\frac{4L}{h_i}$$

Given the timescale this means it takes time at most $4\frac{T-v}{4+\sqrt{21}}$ to process the old requests and at most $4\frac{T-v}{5+\sqrt{21}}$ to process the new requests. Furthermore, since $v < 1$ and the earliest deadline of i is $5+\sqrt{21}$ we have $T-v \geq (4+\sqrt{21})$. It follows

$$\begin{aligned}
&r \leq 1 = \left(\frac{1}{4+\sqrt{21}}\right)(4+\sqrt{21}) \leq \left(\frac{1}{4+\sqrt{21}}\right)(T-v)\\
&\Rightarrow \left(\frac{1}{4+\sqrt{21}}\right)v + r \leq \left(\frac{1}{4+\sqrt{21}}\right)T\\
&\Rightarrow \left(\frac{1}{4+\sqrt{21}}\right)v + r \leq \left(\frac{1}{4+\sqrt{21}}\right)T + \left(1-\frac{1}{4+\sqrt{21}}\right)T - \left(1-\frac{1}{4+\sqrt{21}}\right)T\\
&\Rightarrow \left(1-\frac{1}{4+\sqrt{21}}\right)T + \left(\frac{1}{4+\sqrt{21}}\right)v + r \leq T\\
&\Rightarrow \left(1-\frac{1}{4+\sqrt{21}}\right)T - \left(1-\frac{1}{4+\sqrt{21}}\right)v + v + r \leq T\\
&\Rightarrow 4\frac{T-v}{4+\sqrt{21}} + 4\frac{T-v}{5+\sqrt{21}} + v + r \leq T
\end{aligned}$$

This means we can process all requests (including the request for b_i with deadline at time T) before time T and plant i does not grow above height $(5+\sqrt{21})L$. □

The algorithm Reduce-Fastest(x), introduced by Gasieniec et al. [51], cuts the next fastest growing bamboo among those with height at least $x \cdot R$. The following proof is structured in a way similar to the proof by J. Kuszmaul for discrete BGT [81].

Lemma 48. *Reduce-Fastest*$(2(R+Dh_{max}))$ *is a 6-approximation on the star.*

Proof. Assume that at some point there is a bamboo b_i that reaches height $3(R+Dh_{max})$. Let t_1 be the most recent time b_i reaches height $2(R+Dh_{max})$ and t_3 the time it reaches height $3(R+Dh_{max})$. Furthermore let t_2 be the first time in between t_1 and t_2 where the gardener visits the center. Since it takes at most distance D to visit a plant and return to the center, the height of b_i is at most $2R+3Dh_{max}$ at this time. We consider the set S of bamboo that are cut at least once during the time interval $[t_2, t_3)$. For bamboo $j \in S$, let m_j be the number of times the bamboo is cut in the interval $[t_2, t_3)$.

Since b_i already has height at least $2(R+Dh_{max})$ when the algorithm decides to cut j we have $h_j \geq h_i$ for all $j \in S$.

For $m_j \geq 2$ we have $h_j(t_3 - t_2) > 2(R + Dh_{max})(m_j - 1)$ or $\frac{h_j(t_3-t_2)}{2(m_j-1)} > R + Dh_{max}$ because bamboo b_j needs to grow to height $2(R + Dh_{max})$ at least $m_j - 1$ times in the interval in order to be cut m_j times. Meanwhile, bamboo b_i grows by at most $R + Dh_{max}$ during this interval and therefore $R + Dh_{max} > h_i(t_3 - t_2)$. It follows that $h_j > 2(m_j - 1)h_i \geq m_j h_i$.

During the interval the algorithm visits each plant $b_j \in S$ a total of m_j times and travels distance d_j each time. Additionally, distance $\frac{d_i}{2}$ is traversed to reach plant b_i.

$$\begin{aligned}
t_3 - t_2 &= \sum_{b_j \in S} m_j d_j + \frac{d_i}{2} \\
&< \sum_{b_j \in S} \frac{h_j d_j}{h_i} + \frac{d_i}{2} \\
&= \frac{1}{h_i} \sum_{b_j \in S} h_j d_j + \frac{d_i}{2} \\
&\leq \frac{1}{h_i}(R - h_i d_i) + \frac{d_i}{2} \\
&= \frac{R}{h_i} - \frac{d_i}{2}
\end{aligned}$$

This however means that the interval is too short for plant b_i to grow by height R since this takes time $\frac{R}{h_i}$. This is a contradiction and thus no plant can reach height $3(R + Dh_{max}) \leq 6L$. $\square$

Chapter 6

Allocating contiguous blocks of indivisible chores fairly

6.1 Introduction

Fair division is a classic problem in economics, that deals with the question of how we can divide a set of resources between a number of different people with varying likes and dislikes. This is a very relevant problem, which occurs often in everyday social interaction and it has motivated a lot of research on different variations of the problem.

A classical problem in this area is the so called *cake cutting* problem [104]. In this problem we need to divide a cake with different toppings between a number of players who value these toppings differently. Our goal here is to maximize the happiness of each player, which is determined by the fraction of cake that she receives.

In this problem the players get a positive utility from their fraction of cake, so the more cake a player receives the happier she is. We can also consider a minimization problem, where the cake represents work that needs to be performed by the agents. This leads to the problem of *chore division* [49] where the goal is to minimize the discontent every player gets from his share of the work.

In order to evaluate the allocations we use two different kinds of welfare, *utilitarian welfare* and *egalitarian welfare*. To calculate the utilitarian welfare of an allocation, we add the utilities of all players in that allocation together. To calculate the egalitarian welfare we take the lowest utility or in the case of chore division the highest disutility as our welfare.

The resource that has to be allocated can be *divisible* or *indivisible*. In the first case, we can divide the resource into arbitrarily many pieces before allocating them to the agents. In the second case resources are divided into a set of distinct items, which cannot be divided further. Each item has to be fully allocated to one agent. When talking about allocating indivisible items, we assume these items to be lying in some order on a line.

When considering these problems of fair division, it is necessary to define some notion of fairness. In many earlier papers [4, 30, 107] the criteria for fairness are *proportionality*, *equitability* and *envy-freeness*. These are also used in this paper.

An allocation is proportional if every agent receives at least a fraction of $1/n$ of the utility that she would get if all of the resource gets allocated only to her (at most $1/n$ for chores). Here n is the number of players. An allocation is equitable if all agents get the same utility from their allocated share of the resource. Finally, an allocation is envy-free if every agent values the share she receives at least as much as the share any other agent receives.

As stated before, our goal is then to optimize the welfare of the players, while maintaining some notion of fairness. Since these goals do not necessarily go hand in hand, the natural question arises what the trade-off between these goals is. To this end, Caragiannis et al. [30] and independently Bertsimas et al. [17] initiated the concept of the *price of fairness*, which measures this trade-off, similarly to the related and well-known concepts of price of anarchy and price of stability. Caragiannis et al. [30] found upper and lower bounds for the problem of cake cutting as well as chore division in the divisible and indivisible case. However, their solutions did not limit the amount of pieces each player gets, leading to situations in which players might receive a high number of small pieces or cake crumbs. This situation can be undesirable in practice. This prompted Aumann and Dombb [4] to consider the case in which each player must be assigned a connected piece.

To calculate the *price of fairness*, we divide the welfare of an optimal allocation by the welfare of an optimal fair allocation. For chore division we use the inverse of that value to allow for easy comparisons (the optimal welfare is now never more than that of an optimal fair allocation). We speak of a *contiguous* allocation if each player gets a contiguous block of items on the line in the case of indivisible items or a connected piece in the case of a divisible resource.

Following the work of Aumann and Dombb [4], two different variations of this problem have been considered. This paper fills the gap between those works by considering both at once. Since Aumann and Dombb [4] only considered the problem of cake cutting, we can instead look at the allocation of indivisible goods, while still holding onto the requirement of the allocations being contiguous. This problem of fair division of contiguous blocks of indivisible items was studied by Suksompong [108].

The other variation keeps the goods divisible and instead considers again chores instead of goods. This problem of dividing connected chores fairly was studied by Heydrich and van Stee [57].

This paper now combines these variations and fills the gap between the papers from Suksompong and Heydrich and van Stee, considering the price of fairness for the fair allocation of contiguous blocks of indivisible chores. An example where this type of problem occurs could be the allocation of work shifts, which can not be further divided. A more specific example might be the division of routes between train drivers. This paper is influenced by both works; in fact many of the employed techniques are adaptations of techniques used in the one paper or the other. This is also mirrored in the results.

6.1.1 Our results

The price of proportionality is very similar between all three problems, being 1 in all cases for egalitarian welfare and n, n and $n - 1 + \frac{1}{n}$ for the combination of indivisibility and chores, divisible chores and indivisible goods respectively in the case of utilitarian welfare.

Furthermore, if we make the change from goods to chores the price of envy-freeness changed from being bounded to being unbounded. Also, if we change to indivisible items, the price of equitability suddenly becomes unbounded. In this paper both changes are combined and we can observe that now both prices of fairness are unbounded.

TABLE 6.1: Results of this work compared to [57] and [108]. (Contiguous blocks or pieces are allocated, some results only hold for $n > 2$)

Chores	indivisible (this work)		divisible [57]		
	utilitarian	egalitarian	utilitarian		egalitarian
			lower	upper	
proportionality	n	1	$\frac{n}{2}$	n	1
equitability	∞		n		1
envy-freeness	∞		∞		

Goods/Cakes	indivisible [108]			divisible [4]		
	utilitarian		egalitarian	utilitarian		egalitarian
	lower	upper		lower	upper	
proportionality	$n - 1 + \frac{1}{n}$		1	$\frac{\sqrt{n}}{2}$	$\frac{\lfloor\sqrt{n}\rfloor}{2} + 1 - o(1)$	1
equitability	∞			$n - 1 + \frac{1}{n}$	n	1
envy-freeness	$\frac{\lfloor\sqrt{n}\rfloor}{2}$	$\frac{\lfloor\sqrt{n}\rfloor}{2} + 1 - o(1)$	$\frac{n}{2}$	$\frac{\sqrt{n}}{2}$	$\frac{\lfloor\sqrt{n}\rfloor}{2} + 1 - o(1)$	$\frac{n}{2}$

It should be mentioned that [108] also deals with the existence of allocations of indivisible goods, which satisfy approximate versions of the three notions of fairness. It is possible to obtain analogous results for indivisible chores with only slight modifications to the proofs in [108].

In the case of equitability we do not change the definition when switching to chores, therefore the same proof that applies to goods also applies to chores. The proof for the existence of an allocation of indivisible goods that satisfies the approximate version of envy-freeness relies on the existence of an envy-free allocation for divisible goods. Su showed that such an allocation also always exists for divisible chores [106]. This means the proof also holds for chores. In the case of proportionality the proof needs to be altered slightly. However the structure stays the same.

6.1.2 Related work

Modern mathematicians started working on the topic of fair division in the 1940's with Banach, Steinhaus und Knaster giving the "Last Diminisher" mechanism for proportional divisions with n players [104]. In the following years researchers focussed on finding algorithms to achieve fair divisions [28, 29, 41, 105]. In particular Dubins and Spanier [41] gave an algorithm that guarantees the existence of a contiguous proportional allocation, Cechlárová et al. [31] showed the existence of certain contiguous equitable allocations and Stromquist [105] proved the existence of a contiguous envy-free allocation.

The problem of fair division of chores was first mentioned by Gardner [49]. The result of Steinhaus [104] can be applied to chores, thus a proportional division always exists. Su [106] proves the existence of envy-free divisions of chores with connected pieces, while Heydrich and van Stee [57] show the existence of equitable divisions with connected pieces.

The problem of the efficiency of fair divisions was first adressed by Caragiannis et al. [30]. They gave bounds for the price of fairness for utilitarian welfare, considering the three notions of fairness as well as divisible and indivisible cakes and chores. Aumann and Dombb [4] then restricted the problem, only allowing connected pieces. This way players do not get unions of very small pieces (crumbs) assigned. They considered the problem with cakes, Heydrich and van Stee [57] then extended this work to chores.

Bouveret et al. [27] considered the allocation of contiguous blocks of indivisible items on a line and showed that determining whether a contiguous fair allocation exists is NP-hard for proportionality and envy-freeness. Aumann et al. [5] considered the problem of finding a contiguous allocation that maximizes welfare for both divisible and indivisible items, showing that it is NP-hard to find the optimal solution, but there exists an efficient constant factor approximation algorithm. Bei et al. [12] and Cohler et al. [37] also considered this problem of maximizing welfare with additionally the constraint of respectively proportionality and envy-freeness. Suksompong [108] considered the problem of finding contiguous allocations of indivisible items on a line with respect to fairness constraints and shows the existence of certain contiguous allocations with respect to approximate versions of the usual notions of fairness, as well as giving bounds on the price of fairness with respect to exact proportionality, equitability and envy-freeness.

Recently, more work on the division of indivisible items and chores has been done. Kurokawa et al. [80] consider an alternative notion of fairness for indivisible items, the maximin share guarantee, and show that this fairness notion cannot be guaranteed even if the number of goods is small. Aziz et al. [9] add to this by considering indivisible chores with a weighted generalisation of maximin share. They provide a polynomial-time-constant-approximation algorithm for their problem.

Bilo et al. [21] study the allocations of indivisible items that are envy-free up to one good. While we assume that the items lie on a line, they generalize this to items that lie on the vertices of a graph and allocate bundles of items that are connected on the underlying item graph.

Other methods of dividing goods are the egalitarian equivalent and the competitive equilibrium with equal incomes rules, which have been compared by Bogolmonaia et al [25]. For goods the competitive rule fares better, however for chores this rule shows some disadvantages that the egalitarian rule does not have.

Suksompong at al. [13] give a truthful envy-free mechanism for cake cutting and chore division for two agents with piecewise uniform valuations without free disposal. This means the whole cake has to get allocated. Such a mechanism does not exist with some additional assumption, including the assumption that connected pieces are allocated. For more agents the class of valuations has to be restricted.

A new approach is to consider mixed cakes, where there are both good and bad pieces. Segal-Halevi [101] considers the case of mixed cakes for envy freeness and connected pieces and shows that a devision exists for 3 agents. Cariagannis et al. [8] consider this generalization for indivisible goods and chores.

6.2 Definitions

In this section we formally define the fair division problem with indivisible chores, the notions of fairness and social welfare, as well as the *price of fairness*, which measures the trade-off between fairness and social welfare. Since we are in the context of chores we will refer to social welfare as social cost.

Let $N = \{1, .., n\}$ be the set of agents and $M = \{1, ..., m\}$ the set of chores. We assume the chores lie on a line in this order. Each agent i has a nonnegative disutility $d_i(j)$ for chore j. We assume that the disutilities are *additive*, which means that $d_i(M') = \sum_{j \in M'} d_i(j)$ for any agent i and a subset $M' \subseteq M$.

An *allocation* $M = (M_1, ..., M_n)$ is a partition of all chores into n bundles so that agent i receives bundle M_i. The allocation is *contiguous* if each M_i forms a contiguous block of items on the line.

The *utilitarian social cost* of an allocation is defined as $\sum_{i \in N} d_i(M_i)$ and the *egalitarian social cost* is $\max_{i \in N} d_i(M_i)$.

We now define the notions of fairness that we consider in this paper. An allocation $M = (M_1, ..., M_n)$ is

- *proportional* if $d_i(M_i) \leq \frac{1}{n} d_i(M)$ for all $i \in N$.
- *equitable* if $d_i(M_i) = d_j(M_j)$ for all $i, j \in N$.
- *envy-free* if $d_i(M_i) \leq d_i(M_j)$ for all $i, j \in N$.

Since fairness cannot be guaranteed in all cases we also consider approximate versions of these notions. An allocation $M = (M_1, ..., M_n)$ is

- *ε-proportional* if $d_i(M_i) \leq \frac{1}{n} d_i(M) + \varepsilon$ for all $i \in N$.
- *ε-equitable* if $\left| d_i(M_i) - d_j(M_j) \right| \leq \varepsilon$ for all $i, j \in N$.
- *ε-envy-free* if $d_i(M_i) \leq d_i(M_j) + \varepsilon$ for all $i, j \in N$.

The price of fairness (price of proportionality, respectively envy-freeness, equitability) is the minimal social cost achievable in fair (proportional, respectively envy-free, equitable) allocations divided by the minimal social cost achievable in arbitrary allocations. If both are 0 we define the price of fairness to be equal to 1. The price of fairness is only defined if a fair allocation exists, therefore we only consider such instances. Following Caragiannis et al. [30], we assume the normalization $d_i(M) = 1$.

6.3 Existence of fair allocations

We now consider the existence of fair contiguous allocations of indivisible chores. Since achieving a completely fair allocation can not always be guaranteed we settle for approximate versions of the three notions of fairness.

Proportionality We begin with an algorithm that finds a $\frac{n-1}{n}d_{i,max}$-proportional allocation. Suksompong gives an analogous result for the allocation of goods and our proof follows their structure while making straightforward changes to adapt the problem to the allocation of chores instead of goods.

Lemma 49. *For any instance there exists a contiguous allocation M with $d_i(M_i) \leq \frac{1}{n}d_i(M) + \frac{n-1}{n}d_{i,max}$ for all agents $i \in \mathbb{N}$.*

Proof. Process items from left to right using the following algorithm:

1. Set the current block to the empty block
2. Add items to the current block until $d_i(M_i) > \frac{1}{n}d_i(M) + \frac{n-1}{n}d_{i,max}$ for all i.
3. Remove the last item and assign the current block to an agent who values it with at most $\frac{1}{n}d_i(M) + \frac{n-1}{n}d_{i,max}$.
4. If there is only one agent left, assign all remaining items, otherwise return to 1.

The process also ends if there are no more items to assign, but more than one agent is left. In this case we have found an allocation where every agent values its block with at most $\frac{1}{n}d_i(M) + \frac{n-1}{n}d_{i,max}$.

Now assume that the last agent does get assigned some items. Clearly all agents but the last one get assigned a block that they value with at most $\frac{1}{n}d_i(M) + \frac{n-1}{n}d_{i,max}$. It remains to show that the last agent values her assigned items with at most $\frac{1}{n}d_i(M) + \frac{n-1}{n}d_{i,max}$.

This is done by backwards induction. We show that when there are k agents remaining the last agent values all remaining items with at most $\frac{k}{n}d_i(M) + \frac{n-k}{n}d_{i,max}$.

Clearly this holds for $n = k$. Now assume there are $k+1$ agents remaining and the last agent evaluates the currently remaining items with at most $\frac{k+1}{n}d_i(M) + \frac{n-k-1}{n}d_{i,max}$.

After step 2 of the algorithm, the last agent evaluates the current block with at least $\frac{1}{n}d_i(M) + \frac{n-1}{n}d_{i,max}$. Then during step 3 the block that gets assigned this round is evaluated by the last agent with at least $\frac{1}{n}d_i(M) - \frac{1}{n}d_{i,max}$. It follows that after this block is assigned all remaining items are evaluated with at most $\frac{k}{n}d_i(M) + \frac{n-k}{n}d_{i,max}$. □

The following simple example shows that we cannot hope to improve the additive factor in this guarantee. Assume there are n agents but only one item. All agents value this item with disutility 1. Then the proportional share of each agent is $\frac{1}{n}$, but since we need to assign the entire item to a single agent, there is an agent with disutility $1 = \frac{1}{n} + \frac{n-1}{n}$.

Sometimes we may want to choose the ordering in which chores are assigned to the agent. The following example shows that we cannot always find a d_{max}-proportional allocation. In fact we show that there is not even a kd_{max}-proportional allocation for any constant k.

Let there be $m > 6k$ items and let m be even. Agent 1 values every item with disutility 1 and agent 2 values only the rightmost $\frac{m}{2} - k$ items with disutility 1. In a kd_{max}-proportional allocation we are only allowed to assign $\frac{m}{2} + k$ items to agent 1. If agent 1 is to be assigned a leftside block, then agent 2 will be assigned all $\frac{m}{2} - k$ items on the right side that she values at disutility 1. Since $m > 6k$ we get $\frac{m}{4} > \frac{3}{2}k$ and thus $\frac{m}{2} - k > \frac{m}{4} + \frac{1}{2}k = \frac{1}{2}(\frac{m}{2} - k) + k$. This means agent 2 gets assigned more than her proportional share.

Equitability The definition of ε-equitability does not change when going from goods to chores, which means many results for chores directly apply to goods as well, including the following result by Suksompong.

Lemma 50. *Given any instance and any ordering of the agents, there exists a contiguous d_{max}-equitable allocation in which agents are allocated blocks of items on the line according to the ordering.*

The algorithm that achieves this result first finds the block of the agent who currently has the highest disutility and then finds the block of the agent that currently has the lowest disutility. Should these blocks be adjacent to each other an item is moved from the agent with highest disutility to the one with lowest disutility. Otherwise an agent is found that sits between the agents with highest and lowest disutility such that an item can be moved from this agent to the agent with lowest disutility. This process gradually lowers the gap between the highest and lowest disutility and the algorithm ends in a d_{max}-equitable allocation.

This result only provides us with an approximately equitable solution, however it does not give any guarantee about the quality of this solution. There may be instances where this algorithm gives very high disutility for all agents, even though there is another fair solution with lower disutility for each agent. However when given the opportunity to choose the ordering of the agents it is possible to find an allocation with a much better welfare guarantee.

There is a d_{max}-equitable allocation where the egalitarian social cost is optimal. The following lemma explains how an optimal solution can be transformed into a solution that is also equitable. Unfortunately this does not give us an efficient algorithm to calculate this solution.

Lemma 51. *Given any instance, there exists a contiguous d_{max}-equitable allocation with an egalitarian social cost that equals the lowest egalitarian social cost that can be achieved by any contiguous allocation.*

Proof. Let w be the lowest egalitarian social cost among all allocations of the instance and let $M = (M_1, \ldots, M_n)$ be the allocation that achieves this social cost. There exists a contiguous allocation where all agents i receive disutility in the range $[w - d_{max}, w]$. This can be shown by induction over the number of agents. Clearly the claim holds for $n = 1$ agent.

Now assume the claim holds for $n - 1$ agent. We now increase the size of the first agents block, by taking items from his right side neighbor, until the first agents disutility is in the interval $[w - d_{max}, w]$. This is possible because the agent values no single item with more than d_{max} and therefore we do not overshoot the interval when adding an item. We also do not undershoot the interval, because if we run out of items before reaching a value within the interval, then we have found a solution with less social cost than w.

We repeat this process with the next agents in numerical order. The process ends if we have either run out of items to give to the current agent or all agents up to agent $n - 1$ receive a disutility with a value in $[w - d_{max}, w]$.

If after this process all agents receive disutility in the range $[w - d_{max}, w]$ we are done. Otherwise agent n receives disutility less than $w - d_{max}$. If that is the case we now increase the size of agent n's block by giving him items from his left side neighbors until agent n receives disutility in the interval $[w - d_{max}, w)$. We do not undershoot the interval for the same reason as before with agent 1. We also do not overshoot the interval because the agent values no single item with more than d_{max} and in the beginning agent n's disutility is strictly less than $w - d_{max}$. Note in particular that the disutility of agent n, after receiving these items, is strictly less than w.

Now remove agent n and its block. There is an allocation that assigns the remaining items among agents 1 to $n - 1$ with egalitarian social cost w and no allocation with lower egalitarian social cost. This is because the existence of an allocation with lower egalitarian social cost than w for the partial instance would imply an allocation with lower egalitarian social cost than w for the entire instance, which is reached by adding the block of n back to this allocation.

By the inductive hypothesis there is an allocation on the partial instance where agents 1 to $n - 1$ receive disutility in the interval $[w - d_{max}, w]$. Agent n also receives disutility in the interval $[w - d_{max}, w]$ which proves the claim. □

Envy-freeness We now consider the existence of envy-free allocation. It is known that for divisible chores a contiguous envy-free allocation always exists. We can round this allocation to a $2d_{max}$-envy-free allocation. This result is completely analogous to the result for goods.

Lemma 52. *Given any instance, there exists a $2d_{max}$-envy-free allocation.*

Proof. Let the set of chores be represented by the interval $[0, m]$. For $j \in M$, agent i has uniform disutility $d_i(j)$ for the interval $[j-1, j]$. Consider an envy-free allocation of this set of chores. We round this allocation as follows. For each item j if point j is in the interior of a piece, allocate the full item to the agent who owns that piece. Otherwise, if point j is at the boundary of two pieces belonging to different agents, assign item j to the left agent.

Agent i does not loose more than $d_{i,max}$ during the rounding. Also no agent gains more than $d_{i,max}$ from the perspective of agent i. Therefore agent i has envy at most $2d_{i,max}$ towards any other agent. □

We can get a slightly better result if there are only two agents. Due to Lemma 49 there is an allocation where agent 1 receives at most disutility $\frac{1}{2}d_1(M) + \frac{1}{2}d_{1,max}$. Then agent 2 receives at least disutility $\frac{1}{2}d_1(M) - \frac{1}{2}d_{1,max}$ from the perspective of agent 1. Because of symmetry this implies a d_{max}-envy-free allocation for two agents.

6.4 The price of proportionality

We begin with the price of proportionality. We first consider the utilitarian version, then the egalitarian one.

Theorem 53. *The utilitarian price of proportionality for contiguous allocations of indivisible chores is n.*

Proof. Upper bound: If the contiguous allocation with minimum utilitarian social cost is also proportional, then the price of proportionality is 1.

If the contiguous allocation with minimum utilitarian social cost is not proportional, then at least one agent has disutility more than $\frac{1}{n}$. Therefore, the utilitarian social cost of this allocation is more than $\frac{1}{n}$. A proportional allocation has at most a utilitarian social cost of 1, since all agents have disutility at most $\frac{1}{n}$. Therefore, the price of proportionality in this case is upper-bounded by $1/\frac{1}{n} = n$.

Lower bound for $n > 2$: Let $m = n$ and $0 < \varepsilon < 1$. In this proof we denote the n players and chores as $\{0, .., n-1\}$ instead of $\{1, .., n\}$. Choose the following disutilities. For $i = 0, ..., n-2$:

$$d_i(j) = \begin{cases} \frac{1}{n} - \varepsilon & j = i \\ 2\varepsilon & j = i+1 \pmod n \\ \frac{2}{n} - \varepsilon & j = i+2 \pmod n \\ \frac{1}{n} & \text{otherwise} \end{cases}$$

Finally,

$$d_{n-1}(j) = \begin{cases} \frac{1}{n} + \varepsilon & j = 0 \\ \frac{1}{n} - \varepsilon & j = n-1 \\ \frac{1}{n} & j = 1, ..., n-2. \end{cases}$$

For example $n = 4$:

$$\begin{pmatrix} \frac{1}{4} - \varepsilon & 2\varepsilon & \frac{2}{4} - \varepsilon & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} - \varepsilon & 2\varepsilon & \frac{2}{4} - \varepsilon \\ \frac{2}{4} - \varepsilon & \frac{1}{4} & \frac{1}{4} - \varepsilon & 2\varepsilon \\ \frac{1}{4} + \varepsilon & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} - \varepsilon \end{pmatrix}$$

The best proportional allocation allocates one chore to each player corresponding to the main diagonal of the matrix, giving $\frac{1}{n} - \varepsilon$ to every agent, thus the utilitarian social cost is $n\left(\frac{1}{n} - \varepsilon\right) = 1 - n\varepsilon$.

There exists a contiguous allocation which assigns $\frac{1}{n} + \varepsilon$ to agent $n-1$ and 2ε to all other agents, giving utilitarian social cost $\frac{1}{n} + (2n-1)\varepsilon$. This is achieved by giving item i to player $i - 1 \pmod{n}$ for $i = 0, \dots, n-1$. (For ε sufficiently small, it is easy to check that this is in fact the optimal contiguous allocation.)

Therefore the price of proportionality is lower bounded by $\frac{1-n\varepsilon}{\frac{1}{n}+(2n-1)\varepsilon}$ which is arbitrarily close to n for sufficiently small ε.

Lower bound for $n = 2$: For $n = 2$ consider the following example: Let $m = 4$.

$d_1(1) = d_1(3) = d_2(2) = d_2(4) = \frac{1}{2} - \varepsilon$,
$d_1(2) = d_1(4) = d_2(1) = d_2(3) = \varepsilon$.

A proportional allocation has to give both agents two pieces with each agent receiving $\frac{1}{2}$ giving a total disutility of 1. There is however a nonproportional allocation that gives agent 1 pieces 2, 3 and 4 yielding disutility $\frac{1}{2} + \epsilon$ and agent 2 piece 1 yielding disutility ϵ giving a total disutility of $\frac{1}{2} + 2\epsilon$. For small enough ϵ the price of fairness is arbitrarily close to 2. □

Theorem 54. *The egalitarian price of proportionality for contiguous allocations of indivisible chores is 1.*

Proof. If a proportional contiguous allocation exists, it follows that the optimal contiguous allocation is also proportional.

This can be shown via contraposition. If the optimal contiguous allocation is not proportional, it follows that the egalitarian social cost of any allocation is more than $\frac{1}{n}$. Then a proportional allocation does not exist.

This means that the best proportional contiguous allocation and the optimal contiguous allocation are the same and the price of proportionality is 1.
□

6.5 The price of equitability

We now turn to the price of equitability. The results differ when $n = 2$ compared to $n > 2$.

Theorem 55. *The utilitarian price of equitability for contiguous allocations of indivisible chores is 2 for $n = 2$ and unbounded for $n > 2$.*

Proof. We consider two cases $n = 2$ and $n > 2$.

Case 1: $n = 2$ In an equitable contiguous allocation with minimum utilitarian social cost, both agents have disutility $x \leq \frac{1}{2}$, otherwise they can switch their bundles. Assume without loss of generality that the first agent gets the leftmost chore in this optimal equitable allocation.

Consider an arbitrary allocation. If the first agent gets the leftmost chore, at least one of the agents gets at least disutility x, because at least one of the agents gets a block that is at least as big as the block he gets in the equitable optimal allocation. On the other hand, if the second agent gets the leftmost chore, at least one of the agents gets at least disutility $1 - x \geq x$.

Therefore, the egalitarian social cost of any contiguous allocation is at least x. The ratio of the equitable contiguous allocation with minimum utilitarian social cost to the contiguous allocation with minimum utilitarian social cost is then at most $2x/x = 2$.

The following example shows that the bound is tight. Let $m = 4$ and $d_1(1) = d_1(3) = d_2(2) = d_2(4) = \frac{1}{2}$ and $d_i(j) = 0$ otherwise. The minimum utilitarian social cost of a contiguous allocation is $\frac{1}{2}$ (e.g., by giving piece 1 to player 2 and the other pieces to player 1), whereas that of an equitable contiguous allocation is 1, as both players must receive the same valuation.

Case 2: $n > 2$ Let $m = n$.

The disutilities are assigned as follows: $d_i(i) = 1 - \varepsilon$. The remaining disutilities are chosen such that

- the remaining disutilities sum up to ε for each agent.
- the remaining disutilities are all pairwise distinct.
- $d_1(2) + d_2(3) ... + d_{n-2}(n-1) + d_{n-1}(n) + d_n(1) = \varepsilon$

$$\begin{pmatrix} 1-\varepsilon & d_1(2) & \dots & \dots & d_1(n) \\ d_2(1) & 1-\varepsilon & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & 1-\varepsilon & d_{n-1}(n) \\ d_n(1) & \dots & \dots & d_n(n-1) & 1-\varepsilon \end{pmatrix}$$

There exists an allocation with disutility ε (We assign chore $i + 1 \bmod n$ to player i). Thus the utilitarian social cost of the optimal contiguous allocation is at most ε. The only possible equitable allocation is to assign $1-\varepsilon$ to every agent, giving the utilitarian social cost $n - n\varepsilon$.

As ε goes to 0, the utilitarian social cost of the equitable allocation approaches n, while the utilitarian social cost of the optimal allocation approaches 0. Therefore, the price of equitability becomes arbitrarily large as ε approaches 0.

□

Theorem 56. *The egalitarian price of equitability for contiguous allocations of indivisible chores is 1 for $n = 2$ and unbounded for $n > 2$.*

Proof. We again consider $n = 2$ and $n > 2$.

Case 1: $n = 2$ Similarly to Theorem 55, the following holds. In an equitable contiguous allocation with minimum egalitarian social cost, both agents have utility $x \leq \frac{1}{2}$, otherwise they can switch their bundles. Assume without loss of generality that the first agent gets the leftmost chore in this optimal allocation. Consider any contiguous allocation. If the first agent gets the leftmost chore, at least one of the agents gets at least disutility x, because at least one of the agents gets a block that is at least as big as the block he gets in the optimal equitable allocation. On the other hand, if the second agent gets the leftmost chore, at least one of the agents gets at least disutility $1 - x \geq x$. Therefore, the egalitarian social cost of any contiguous allocation is at least x. This means the equitable contiguous allocation is optimal for the egalitarian social cost, so the price of equitability is 1.

Case 2: $n > 2$ We consider the same example as in Theorem 55. In this case the egalitarian social cost of the (only) equitable contiguous allocation is $\frac{1}{n} - \varepsilon$. The optimal contiguous allocation has egalitarian social cost less than ε. Thus as ε approaches 0, the price of equitability becomes arbitrarily high. □

6.6 The price of envy-freeness

Finally we consider the price of envy-freeness. Since for $n = 2$ envy-freeness is equivalent to proportionality, we only give results for $n > 2$.

Theorem 57. *The price of envy-freeness for contiguous allocations of indivisible chores is unbounded for both utilitarian and egalitarian social cost for $n > 2$.*

Proof. Let $0 < \varepsilon < \frac{1}{2(n-1)^2}$ and $m = (n-1)^2 + 1$. First consider a situation with $(n-1)^2$ pieces. We divide the chores into $n-1$ groups of $n-1$ pieces each, and each group contains one piece for every player $\{1, ..., n-1\}$. The first piece of the first group is associated with player 1, the second with player 2 and so on, until the last piece of the first group is associated with player $n-1$. The pieces of the second group are then associated with players $2, 3, ..., n-1, 1$ (in this order). Generally, the pieces of the i-th group are associated with players $i, i+1, ..., n-1, 1, ..., i-1$.

We call the first piece of each group a "Type A"-piece and the other pieces "Type B"-pieces. Every player values the only "type A"-piece associated with her as $1-(n-2)\varepsilon$ and the "Type B"-pieces associated with her as ε. All other valuations are 0.

We now divide the very last piece with index $(n-1)^2$ and replace it with 2 pieces, dividing the disutilities evenly between them. This means that the player that this piece is associated with (player $n-2$) values these pieces as $\frac{\varepsilon}{2}$ and the other players value them as 0. Finally player n values every piece but the last two as $\frac{1}{(n-1)^2}$. The last two are valued as $\frac{1}{2(n-1)^2}$.

Consider the following contiguous allocation, which allocates pieces 1 to $n-2$ to player $n-1$ and the pieces $i(n-1)$ to $i(n-1)+(n-2)$ to player i for $i=1,...,n-2$. This leaves pieces $(n-1)^2$ and $(n-1)^2+1$ which are also given to player $n-2$. In this allocation only player $n-2$ receives disutility ε, giving a utilitarian and egalitarian social cost of ε.

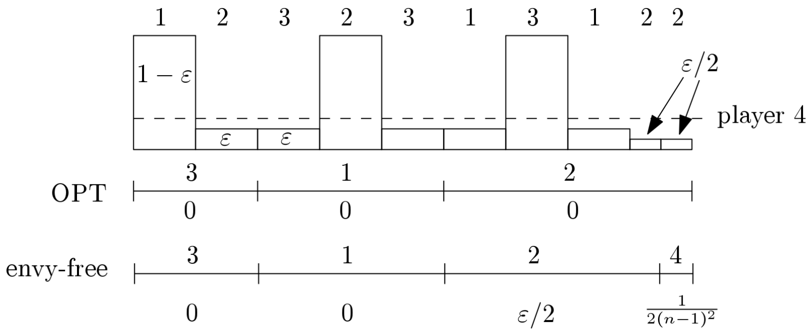

FIGURE 6.1: Example with $n=4$. The numbers above the pieces indicate the players associated with these pieces; all other players (except player n) have valuation 0 for these pieces. The valuation for player n (here 4) for all pieces is identical (apart from the last two items) and is indicated by the dotted line. Below the pieces are the optimal and the best possible envy-free allocation. The numbers above the lines indicate players, and the numbers below the lines indicate their valuation for their assigned block. (Player n receives nothing in the optimal allocation.)

We now show that this allocation is optimal by first proving the following lemma.

Lemma 58. *For the instance described above, there is no contiguous allocation where all players have disutility 0.*

Proof. Assume there is such an allocation. Clearly, player n does not receive any chore in this allocation, as it has nonzero disutility for every chore. This means that the allocation assigns all pieces to the first $n-1$ players with disutility 0 for each player.

Starting from piece 1, there is no player who can cover all of group 1 while getting disutility 0, because group 1 contains a piece for which this player receives disutility more than 0. We can only assign the pieces up to the first disliked piece for the player. Choose any player and assign as many pieces as possible to her, starting from piece 1.

Next, we have to assign pieces starting from the first disliked piece of the player that we have previously assigned pieces to. Again there is no player who can cover all of group 2 while getting disutility 0, because group 2 contains a disliked piece for that player. Repeating this argument for all $n-1$ groups, it follows that the pieces starting from the last disliked piece of the last player that got pieces assigned to her stay unassigned. This is a contradiction, as all pieces must be assigned. This proves Lemma 58. □

If we assign any piece to player n we get at least disutility $\frac{1}{2(n-1)^2} > \varepsilon$. The only other possibility to achieve social cost less than ε is to assign one of the last two pieces to player $n-2$ and try to cover all the groups with the remaining players. Using similar argumentation to the proof of Lemma 58, we can show that there is no allocation that assigns only one of the last two pieces to player $n-2$ and also covers the rest of the pieces with disutility 0.

Case 1: player $n-2$ gets the last piece. Again we try to cover the groups from left to right with the remaining $n-2$ players. (Since player n does not receive anything.) We can only cover the pieces up the second last disliked piece in group $n-2$ of the last player that we assign pieces to, leaving unassigned pieces in group $n-2$ and $n-1$.

Case 2: We assign pieces $(n-3)(n-1)+2$ (second piece in group $n-2$) to $(n-1)^2$ to player $n-2$. (This consists of the second last piece and all contiguous pieces before that for which player $n-2$ has disutility 0.)

We then need to assign the last piece to another player, leaving us in a situation in which we have to cover groups 1 to $n-3$ with $n-3$ players. Since every player can cover each group only up to her disliked piece we have unassigned pieces in group $n-3$ remaining.

Hence, the given allocation with egalitarian and utilitarian social cost ε is optimal. The optimal allocation is however not envy-free since player $n-2$ envies player n, because the empty piece is preferred by every player.

Lemma 58 states that it is impossible to divide the chores between the first $n-1$ players without giving player n a piece and without giving anyone disutility. To achieve an envy-free contiguous allocation we therefore have to give player n a piece. The smallest piece we can give player n is the last piece, valued as $\frac{1}{2(n-1)^2}$.

Therefore to get the optimal contiguous allocation which is also envy-free, we start with the optimal allocation and give player n the last piece instead of player $n-2$. (This is the smallest possible change to the optimal allocation that yields envy-freeness. Thus the resulting allocation is the optimal envy-free allocation.) Player $n-2$ does not envy n, since he values both allocations as $\frac{\varepsilon}{2}$. The other players' allocations each contain at least one piece that player $n-2$ dislikes, therefore she also does not envy them.

Player n envies no one, because she got the piece she dislikes the least allocated to her, while all other players also have at least one piece player n dislikes allocated to them. The other players get disutility 0, therefore they do not envy anyone. The best contiguous envy-free allocation then achieves utilitarian social cost $\frac{\varepsilon}{2} + \frac{1}{2(n-1)^2}$ and egalitarian social cost $\frac{1}{2(n-1)^2}$.

In conclusion, we have a utilitarian price of envy-freeness $\frac{1}{2} + \frac{1}{2(n-1)^2\varepsilon}$ and egalitarian price of envy-freeness $\frac{1}{2(n-1)^2\varepsilon}$. As ε goes to 0, these values grow without bound. $\square$

Chapter 7

Discussion and open problems

In this thesis we explored a variety of topics in the realm of offline and online scheduling as well as allocation of chores. We now summarize our most important results and pose some open questions that hopefully inspire further research.

We begin with the buffer management problem with conflicts presented in chapter 3. Our results on the line graph are depicted in the following table and include tight or almost tight bound for small line segments. In particular, the introduction of the flow model allows for tight results on the path with 5 machines.

Number of machines	2	3	4	5	$m > 5$
Lower bound	3/2 [33]	2 [33]	9/4	16/7	12/5
Upper bound	3/2 [33]	2 [33]	9/4	5/2	$m/2 + 1/4$
Lower bound flow	1	1	4/3	3/2	3/2
Upper bound flow	1	1	4/3	3/2	$m/2 - 2/3$

The situation is much different when considering m machines where we slightly improve the lower bound from 2 to 12/5 but still have a very big gap between the constant lower bound and linear upper bound. This leads to perhaps the most interesting open question of this thesis:

- What is the competitive ratio of the buffer management problem on the path with m machines?

In chapter 4, we provide linear lower bounds to a variety of algorithms for the buffer management problem. Nevertheless, there are still plenty of algorithms that can be tried and we hope to eventually find an algorithm with a sublinear competitive ratio.

We also consider a few results on graphs that are more general than the line graph. The first of these results is a linear competitive algorithm that works on bipartite graphs. We then discuss a $\frac{1}{\varepsilon} + 2 + \chi$-competitive algorithm that works on any graph and uses machines of speed $1 + \chi\varepsilon$.

Finally, we examine a relaxation of the problem where the algorithm is allowed to freely share processing power, i.e. two adjacent machines may run at a combined speed of 1. In this context we find a $\frac{1}{\varepsilon} + 1$-competitive algorithm with machines working at speed $1 + 2\varepsilon$. This result also holds on the line graph, since this type of processor sharing is already possible on these graphs.

It would be interesting to find good algorithms for even more classes of graphs. A good starting point might be cycles with an odd number of machines.

- Is there a linear-competitive algorithm for cycles with an odd number of machines?

In the discrete bamboo garden trimming problem presented in chapter 5 we find a 10/7-approximation algorithm that is based on a reduction to the pinwheel problem. This result can be improved to a 7/5-approximation after going through an extensive case analysis. Here we are of course interested in finding the true value for the approximation ratio of the problem. We conjecture that it should be possible to reach an approximation ratio arbitrarily close to 4/3 but new ideas will be required to work around the large case analysis. Additionally, resolving the 5/6-conjecture for pinwheel would also directly improve our algorithm.

- Can we further improve the approximation ratio for discrete BGT?
- Can we resolve the 5/6-conjecture for pinwheel?

In the continuous version of the problem we show that the deadline-driven strategy is a $(2+\sqrt{3})$-approximation on the star graph and a $(5+\sqrt{21})$-approximation on a star graph with multiple plants. Additionally, the algorithm Reduce-fastest$(2R+2Dh_{max})$ is a 6-approximation. It would be interesting to answer the following questions:

- What is the approximation ratio of reduce-max on the star?
- Can we extend these results to larger classes of graphs such as tree graphs?

For the problem of allocating goods or chores on the line presented in chapter 6, prices of fairness have previously been considered for indivisible as well as divisible goods and additionally divisible chores.

We fill a gap in the literature by providing the prices of fairness for indivisible chores and all of these results are presented in the table below.

Chores	indivisible (this work)		divisible [57]		
	utilitarian	egalitarian	utilitarian		egali-tarian
			lower	upper	
proportionality	n	1	$\frac{n}{2}$	n	1
equitability	∞		n		1
envy-freeness	∞		∞		

Goods/Cakes	indivisible [108]			divisible [4]		
	utilitarian		egali-tarian	utilitarian		egali-tarian
	lower	upper		lower	upper	
proportionality	$n-1+\frac{1}{n}$		1	$\frac{\sqrt{n}}{2}$	$\frac{\lfloor\sqrt{n}\rfloor}{2}+1-o(1)$	1
equitability	∞			$n-1+\frac{1}{n}$	n	1
envy-freeness	$\frac{\lfloor\sqrt{n}\rfloor}{2}$	$\frac{\lfloor\sqrt{n}\rfloor}{2}+1-o(1)$	$\frac{n}{2}$	$\frac{\sqrt{n}}{2}$	$\frac{\lfloor\sqrt{n}\rfloor}{2}+1-o(1)$	$\frac{n}{2}$

Besides the prices of fairness we also consider the existence of approximate versions of the notions of fairness and here we find an interesting question that is still open.

- Is there an efficient algorithm that computes a d_{max}-equitable allocation with a good welfare guarantee?

This question has also been asked in the context of goods. We can always transform an optimal allocation into a d_{max}-equitable one, but this does not lead to a constructive algorithm.

Appendix A

Tables for discrete BGT

Case $3, 4, p_3^* \geq 5$ with schedule 1020:

p^*	p	h	h/p	D_A
5	7	3	0.429	0.417
6	8	4	0.5	
7	10	5	0.5	
8	11	5	0.45	
9	12	6	0.5	
10	14	7	0.5	
11	15	7	0.47	
12	17	8	0.47	

Case $4, 4$ with schedule 10200

p^*	p	h	h/p	D_A
4	5	3	0.6	0.5
5	7	4	0.57	
6	8	4	0.5	
7	10	6	0.6	
8	11	6	0.54	
9	12	7	0.58	
10	14	8	0.57	
11	15	9	0.6	

Case $4, 4, 4$ with schedule 12300

p^*	p	h	h/p	D_A
4	5	2	0.4	0.25
5	7	2	0.28	
6	8	2	0.25	
7	10	4	0.4	
8	11	4	0.36	
9	12	4	0.33	
10	14	5	0.36	
11	15	6	0.4	

Case $4, 5$ with schedule 10200

p^*	p	h	h/p	D_A
9	12	7	0.5833333333	0.55
10	14	8	0.5714285714	
11	15	9	0.6	
12	17	10	0.5882352941	
13	18	10	0.5555555556	
14	20	12	0.6	
15	21	12	0.5714285714	
16	22	13	0.5909090909	

Case $4, 5, 6$ with schedule 31002 01300 21003 01200

p^*	p	h	h/p	D_A
7	10	5	0.5	0.38
8	11	5	0.45	
9	12	5	0.42	
10	14	6	0.43	
11	15	7	0.47	
12	17	8	0.47	
13	18	8	0.44	
14	20	10	0.5	
15	21	10	0.48	
16	22	10	0.45	
17	24	11	0.45	
18	25	12	0.48	
19	27	13	0.48	
20	28	13	0.46	
21	30	15	0.5	

Case $4, 5, 8$ with schedule 21030 01200 01032 01000

p^*	p	h	h/p	D_A
7	10	5	0.5	0.425
8	11	5	0.45	
9	12	6	0.5	
10	14	7	0.5	
11	15	7	0.47	
12	17	8	0.47	
13	18	9	0.5	
14	20	11	0.55	
15	21	11	0.52	
16	22	11	0.5	
17	24	12	0.5	
18	25	13	0.52	
19	27	14	0.525	
20	28	14	0.5	
21	30	16	0.53	

Case $4, 6$ with schedule 21000 01020 01000

p^*	p	h	h/p	D_A
9	12	7	0.58	0.583
10	14	9	0.64	
11	15	10	0.67	
12	17	10	0.59	
13	18	11	0.61	
14	20	13	0.65	
15	21	13	0.62	
16	22	14	0.64	
17	24	15	0.63	
18	25	16	0,64	
19	27	17	0.63	
20	28	18	0.64	
21	30	20	0.67	
22	31	20	0.65	
23	32	20	0.63	
24	34	22	0.65	
25	35	23	0.66	
26	37	24	0.65	
27	38	25	0.66	
28	40	26	0.65	
29	41	26	0.63	
30	42	27	0.649	

Case $4, 6, 8$ with schedule 21003 01020 01003 21000 01023 01000

p^*	p	h	h/p	D_A
9	12	6	0.5	0.458
10	14	7	0.5	
11	15	8	0.53	
12	17	8	0.47	
13	18	9	0.5	
14	20	11	0.55	
15	21	11	0.52	
16	22	12	0.55	
17	24	12	0.5	
18	25	13	0.52	
19	27	14	0.52	
20	28	15	0.54	
21	30	17	0.57	
22	31	17	0.55	
23	32	17	0.53	
24	34	18	0.53	
25	35	19	0.54	
26	37	20	0.54	
27	38	21	0.55	
28	40	22	0.55	
29	41	22	0.53	
30	42	23	0.546	

Case $4, 6, 8, 8$ with schedule 21003 01024 01003 21004 01023 01004

p^*	p	h	h/p	D_A
4	5	2	0.4	0.333
5	7	3	0.43	
6	8	3	0.375	
7	10	4	0.4	
8	11	4	0.36	
9	12	5	0.42	
10	14	6	0.43	
11	15	7	0.47	
12	17	7	0.41	
13	18	7	0.39	
14	20	9	0.45	
15	21	9	0.43	
16	22	10	0.45	
17	24	10	0.42	
18	25	11	0.44	
19	27	12	0.44	
20	28	12	0.43	
21	30	14	0.47	
22	31	14	0.45	
23	32	14	0.44	
24	34	15	0.44	
25	35	16	0.46	

Case $4, 8$ with schedule 10200 10000

p^*	p	h	h/p	D_A
6	8	5	0.625	0.625
7	10	7	0.7	
8	11	7	0.64	
9	12	8	0.67	
10	14	9	0.64	
11	15	10	0.67	
12	17	11	0.65	
13	18	12	0.67	

Appendix B

A case verification program for discrete BGT

In this section we give an overview over the program used in the proof of the $\frac{7}{5}$-approximation for discrete BGT, beginning with the input.

```
#include <iostream>
#include <math.h>
#include <list>
#include <vector>
#include <string>
using namespace std;


//INPUT:
//

float factor = 7.0 / 5.0;
int periodarray[] = { 3,7,7 };
string schedulestring = "1020␣1030";

//
/////////////////////////
```

After the input section there are two assisting functions. The first one tests the validity of a schedule and the second converts the schedule from a string into a vector.

```
void testschedule(vector<int> schedulevector, vector<
   int> periodvector){
        vector<int> dists;
        for (int j = 0; j < periodvector.size(); j++)
           {
                dists.push_back(0);
                float f = factor*periodvector[j];
                periodvector[j] = floorf(f);
        }
        int counter = 0;
        for (int i = 0; i < 2 * schedulevector.size();
            i++) {   //We go through the schedule
           twice and measure the distance inbetween
           occurences of each period
                for (int j = 0; j < periodvector.size
                   (); j++) {
                        dists[j]++;
                        if (schedulevector[counter] ==
                            j+1) {
                                dists[j] = 0;
                        }
                        if (dists[j] >= periodvector[j
                           ]) {
                                cout << "Invalid␣
                                   schedule!" << j <<
                                   "␣";
                                cin.get();
                        }
                }
                counter = counter+1;
                counter = counter % schedulevector.
                   size();
        }
}


void convert(vector<int> * emptyvector) {
        for (int i = 0; i < schedulestring.length(); i
           ++) {
                if (schedulestring[i] != '␣') {
                        emptyvector->push_back(stoi(
                           schedulestring.substr(i, 1)
                           ));
                }
        }
}
```

The next function determines the minimum number of holes in p consecutive positions in the schedule, i.e determines h for a given p.

```
int countholes(vector<int> schedulevector, int p) {
        //Determine the overall number of holes in the
            schedule
        int overallholes = 0;
        for (int i = 0; i < schedulevector.size(); i
           ++) {
                if (schedulevector[i] == 0) {
                        overallholes++;
                }
        }
        int c = p / schedulevector.size();
        int d = p % schedulevector.size();
        //p=c*schedulesize+d
        //Determine the minimum number of holes in d
           consecutive positions
        int start = 0;
        int min = schedulevector.size();
        while (start < schedulevector.size()) {
                int holesinsection = 0;
                int counter = start;
                for (int i = 0; i < d; i++) {
                        counter = counter+1;
                        counter = counter %
                           schedulevector.size();
                        if (schedulevector[counter] ==
                            0) {
                                holesinsection ++;
                        }
                }
                if (holesinsection < min) {
                        min = holesinsection;
                }
                start++;
        }
        return c*overallholes+min;
}
```

In the main part we iterate over p^*, trying to find a sufficiently large set of consecutive periods where $\frac{h}{p} \geq D_a$ holds.

```
int main() {
        //Converting the input into vectors
        vector<int> schedulevector;
        convert(&schedulevector);
        vector<int> periodvector(std::begin(
           periodarray), std::end(periodarray));

        //Is this schedule valid?
        testschedule(schedulevector, periodvector);

        //Calculate D_A
        float DA = 1.0;
        for (int i = 0; i < periodvector.size(); i++)
           {
                DA = DA - 1.0 / periodvector[i];
        }
        cout << "DA␣" << DA << "␣";
        cin.get();

        //Determine a suitable amount of iterations
           and check h/p \geq D_A for these periods
        int iterations = schedulevector.size() * 5 +
           100;
        int pstar = 0;
        for (int i = 1; i < iterations; i++) {
                float f = factor*i;
                int p = floorf(f);
                int h = countholes(schedulevector, p);
                float temp = h;
                temp = temp / p;
                temp = temp - DA;
                if (temp < 0) {
                        pstar = i;
                }
                cout << "Iteration:␣" << i << endl;
        }
        cout << "Result:␣p*␣>=␣" << pstar+1;
        cin.get();
}
```

Bibliography

[1] Spyros Angelopoulos. Online search with a hint. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, volume 185 of *LIPIcs*, pages 51:1–51:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[2] Spyros Angelopoulos, Christoph Dürr, Shendan Jin, Shahin Kamali, and Marc P. Renault. Online computation with untrusted advice. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 52:1–52:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[3] Itai Ashlagi, Maximilien Burq, Chinmoy Dutta, Patrick Jaillet, Amin Saberi, and Chris Sholley. Edge weighted online windowed matching. In *Proceedings of the 2019 ACM Conference on Economics and Computation, EC 2019, Phoenix, AZ, USA, June 24-28, 2019.*, pages 729–742, 2019.

[4] Yonatan Aumann and Yair Dombb. The efficiency of fair division with connected pieces. *ACM Trans. Economics and Comput.*, 3(4):23:1–23:16, 2015.

[5] Yonatan Aumann, Yair Dombb, and Avinatan Hassidim. Computing socially-efficient cake divisions. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13, Saint Paul, MN, USA, May 6-10, 2013*, pages 343–350. IFAAMAS, 2013.

[6] Yossi Azar, Stefano Leonardi, and Noam Touitou. Flow time scheduling with uncertain processing time. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1070–1080. ACM, 2021.

[7] Yossi Azar, Runtian Ren, and Danny Vainstein. The min-cost matching with concave delays problem. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 301–320. SIAM, 2021.

[8] Haris Aziz, Ioannis Caragiannis, Ayumi Igarashi, and Toby Walsh. Fair allocation of indivisible goods and chores. *Auton. Agents Multi Agent Syst.*, 36(1):3, 2022.

[9] Haris Aziz, Hau Chan, and Bo Li. Weighted maxmin fair share allocation of indivisible chores. In Sarit Kraus, editor, *Proceedings of the*

Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019, pages 46–52. ijcai.org, 2019.

[10] Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors. *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[11] Amotz Bar-Noy, Rajeev Motwani, and Joseph Naor. The greedy algorithm is optimal for on-line edge coloring. *Inf. Process. Lett.*, 44(5):251–253, 1992.

[12] Xiaohui Bei, Ning Chen, Xia Hua, Biaoshuai Tao, and Endong Yang. Optimal proportional cake cutting with connected pieces. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press, 2012.

[13] Xiaohui Bei, Guangda Huzhang, and Warut Suksompong. Truthful fair division without free disposal. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 63–69. ijcai.org, 2018.

[14] Michael A. Bender, Martin Farach-Colton, and William Kuszmaul. Achieving optimal backlog in multi-processor cup games. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 1148–1157. ACM, 2019.

[15] Michael A. Bender, Sándor P. Fekete, Alexander Kröller, Vincenzo Liberatore, Joseph S. B. Mitchell, Valentin Polishchuk, and Jukka Suomela. The minimum backlog problem. *Theor. Comput. Sci.*, 605:51–61, 2015.

[16] Michael A. Bender and William Kuszmaul. Randomized cup game algorithms against strong adversaries. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2059–2077. SIAM, 2021.

[17] Dimitris Bertsimas, Vivek F. Farias, and Nikolaos Trichakis. The price of fairness. *Operations Research*, 59(1):17–31, 2011.

[18] Sayan Bhattacharya, Fabrizio Grandoni, and David Wajc. Online edge coloring algorithms via the nibble method. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2830–2842. SIAM, 2021.

[19] Marcin Bienkowski, Artur Kraska, and Hsiang-Hsuan Liu. Traveling repairperson, unrelated machines, and other stories about average

completion times. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[20] Davide Bilò, Luciano Gualà, Stefano Leucci, Guido Proietti, and Giacomo Scornavacca. Cutting bamboo down to size. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *10th International Conference on Fun with Algorithms, FUN 2021, May 30 to June 1, 2021, Favignana Island, Sicily, Italy*, volume 157 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[21] Vittorio Bilò, Ioannis Caragiannis, Michele Flammini, Ayumi Igarashi, Gianpiero Monaco, Dominik Peters, Cosimo Vinci, and William S. Zwicker. Almost envy-free allocations with connected bundles. In Blum [22], pages 14:1–14:21.

[22] Avrim Blum, editor. *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, volume 124 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[23] Avrim Blum and Carl Burch. On-line learning and the metrical task system problem. *Mach. Learn.*, 39(1):35–58, 2000.

[24] Marijke H. L. Bodlaender, Cor A. J. Hurkens, Vincent J. J. Kusters, Frank Staals, Gerhard J. Woeginger, and Hans Zantema. Cinderella versus the wicked stepmother. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings*, volume 7604 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2012.

[25] Anna Bogomolnaia, Hervé Moulin, Fedor Sandomirskiy, and Elena Yanovskaya. Dividing goods or bads under additive utilities. *CoRR*, abs/1608.01540, 2016.

[26] Allan Borodin, Morten N. Nielsen, and Charles Rackoff. (incremental) priority algorithms. *Algorithmica*, 37(4):295–326, 2003.

[27] Sylvain Bouveret, Katarína Cechlárová, Edith Elkind, Ayumi Igarashi, and Dominik Peters. Fair division of a graph. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 135–141. ijcai.org, 2017.

[28] Steven J. Brams and Alan D. Taylor. An envy-free cake division protocol. American Mathematical Monthly, 102(1):9–18, 1995.

[29] Steven J. Brams, Alan D. Taylor, and William Zwicker. A moving-knife solution to the four-person envy-free cake division. In Proceedings of the American Mathematical Society, volume 125, pages 547–554, 1997.

[30] Ioannis Caragiannis, Christos Kaklamanis, Panagiotis Kanellopoulos, and Maria Kyropoulou. The efficiency of fair division. *Theory Comput. Syst.*, 50(4):589–610, 2012.

[31] Katarína Cechlárová, Jozef Dobos, and Eva Pillárová. On the existence of equitable cake divisions. *Inf. Sci.*, 228:239–245, 2013.

[32] Mee Yee Chan and Francis Y. L. Chin. General schedulers for the pinwheel problem based on double-integer reduction. *IEEE Trans. Computers*, 41(6):755–768, 1992.

[33] Marek Chrobak, János Csirik, Csanád Imreh, John Noga, Jirí Sgall, and Gerhard J. Woeginger. The buffer minimization problem for multiprocessor scheduling with conflicts. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 862–874. Springer, 2001.

[34] Joanna Chybowska-Sokól, Grzegorz Gutowski, Konstanty Junosza-Szaniawski, Patryk Mikos, and Adam Polak. Online coloring of short intervals. In Jaroslaw Byrka and Raghu Meka, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020, August 17-19, 2020, Virtual Conference*, volume 176 of *LIPIcs*, pages 52:1–52:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[35] Ilan Reuven Cohen, Sungjin Im, and Debmalya Panigrahi. Online two-dimensional load balancing. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[36] Ilan Reuven Cohen, Binghui Peng, and David Wajc. Tight bounds for online edge coloring. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1–25. IEEE Computer Society, 2019.

[37] Yuga J. Cohler, John K. Lai, David C. Parkes, and Ariel D. Procaccia. Optimal envy-free cake cutting. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press, 2011.

[38] Nikhil R. Devanur, Zhiyi Huang, Nitish Korula, Vahab S. Mirrokni, and Qiqi Yan. Whole-page optimization and submodular welfare maximization with online bidders. *ACM Trans. Economics and Comput.*, 4(3):14:1–14:20, 2016.

[39] Nikhil R. Devanur, Kamal Jain, and Robert D. Kleinberg. Randomized primal-dual analysis of RANKING for online bipartite matching. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 101–107. SIAM, 2013.

[40] Wei Ding. A branch-and-cut approach to examining the maximum density guarantee for pinwheel schedulability of low-dimensional vectors. *Real Time Syst.*, 56(3):293–314, 2020.

[41] L. E. Dubins and E. H. Spanier. How to cut a cake fairly. American Mathematical Monthly, 68(1):1–17, 1961.

[42] Yuval Emek, Shay Kutten, and Yangguang Shi. Online paging with a vanishing regret. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, volume 185 of *LIPIcs*, pages 67:1–67:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[43] Leah Epstein, Asaf Levin, Danny Segev, and Oren Weimann. Improved bounds for online preemptive matching. In Natacha Portier and Thomas Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*, pages 389–399. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[44] Leah Epstein and Meital Levy. Online interval coloring and variants. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 602–613. Springer, 2005.

[45] Matthew Fahrbach, Zhiyi Huang, Runzhou Tao, and Morteza Zadimoghaddam. Edge-weighted online bipartite matching. *CoRR*, abs/2005.01929, 2020.

[46] Yiding Feng and Rad Niazadeh. Batching and optimal multi-stage bipartite allocations (extended abstract). In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, volume 185 of *LIPIcs*, pages 88:1–88:1. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[47] Peter C. Fishburn and J. C. Lagarias. Pinwheel scheduling: Achievable densities. *Algorithmica*, 34(1):14–38, 2002.

[48] Buddhima Gamlath, Michael Kapralov, Andreas Maggiori, Ola Svensson, and David Wajc. Online matching with general arrivals. *CoRR*, abs/1904.08255, 2019.

[49] Martin Gardner. Aha! insight. W.F. Freeman and Co, 1978.

[50] Naveen Garg, Anupam Gupta, Amit Kumar, and Sahil Singla. Non-clairvoyant precedence constrained scheduling. In Baier et al. [10], pages 63:1–63:14.

[51] Leszek Gasieniec, Ralf Klasing, Christos Levcopoulos, Andrzej Lingas, Jie Min, and Tomasz Radzik. Bamboo garden trimming problem (perpetual maintenance of machines with different attendance urgency factors). In Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria, editors, *SOFSEM 2017: Theory and Practice of Computer Science - 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Limerick, Ireland, January 16-20, 2017, Proceedings*, volume 10139 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2017.

[52] Leszek Gasieniec, Benjamin Smith, and Sebastian Wild. Towards the 5/6-density conjecture of pinwheel scheduling. In Cynthia A. Phillips and Bettina Speckmann, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*, pages 91–103. SIAM, 2022.

[53] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[54] Nick Gravin, Zhihao Gavin Tang, and Kangning Wang. Online stochastic matching with edge arrivals. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 74:1–74:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[55] Anupam Gupta, Guru Guruganesh, Binghui Peng, and David Wajc. Stochastic online metric matching. In Baier et al. [10], pages 67:1–67:14.

[56] Varun Gupta, Ravishankar Krishnaswamy, and Sai Sandeep. Permutation strikes back: The power of recourse in online metric matching. In Jaroslaw Byrka and Raghu Meka, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020, August 17-19, 2020, Virtual Conference*, volume 176 of *LIPIcs*, pages 40:1–40:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[57] Sandy Heydrich and Rob van Stee. Dividing connected chores fairly. *Theor. Comput. Sci.*, 593:51–61, 2015.

[58] Felix Höhne, Sören Schmitt, and Rob van Stee. SIGACT news online algorithms column 35: 2019 in review. *SIGACT News*, 50(4):77–92, 2019.

[59] Felix Höhne, Sören Schmitt, and Rob van Stee. SIGACT news online algorithms column 36: 2020 in review. *SIGACT News*, 51(4):89–107, 2020.

[60] Felix Höhne, Sören Schmitt, and Rob van Stee. SIGACT news online algorithms column 38: 2021 in review. *SIGACT News*, 52(4):80–96, 2021.

[61] Felix Höhne and Rob van Stee. Buffer minimization with conflicts on a line. In Minming Li, editor, *Frontiers in Algorithmics - 14th International Workshop, FAW 2020, Haikou, China, October 19-21, 2020, Proceedings*, volume 12340 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 2020.

[62] Felix Höhne and Rob van Stee. Allocating contiguous blocks of indivisible chores fairly. *Inf. Comput.*, 281:104739, 2021.

[63] Felix Höhne and Rob van Stee. Buffer minimization with conflicts on a line. *Theor. Comput. Sci.*, 876:25–33, 2021.

[64] Robert Holte, Louis E. Rosier, Igor Tulchinsky, and Donald A. Varvel. Pinwheel scheduling with two distinct numbers. *Theor. Comput. Sci.*, 100(1):105–135, 1992.

[65] Zhiyi Huang, Ning Kang, Zhihao Gavin Tang, Xiaowei Wu, Yuhao Zhang, and Xue Zhu. How to match when all vertices arrive online. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 17–29, 2018.

[66] Zhiyi Huang, Binghui Peng, Zhihao Gavin Tang, Runzhou Tao, Xiaowei Wu, and Yuhao Zhang. Tight competitive ratios of classic matching algorithms in the fully online model. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2875–2886, 2019.

[67] Zhiyi Huang and Xinkai Shu. Online stochastic matching, Poisson arrivals, and the natural linear program. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 682–693. ACM, 2021.

[68] Zhiyi Huang, Zhihao Gavin Tang, Xiaowei Wu, and Yuhao Zhang. Fully online matching II: beating ranking and water-filling. *CoRR*, abs/2005.06311, 2020.

[69] Sungjin Im, Ravi Kumar, Mahshid Montazer Qaem, and Manish Purohit. Non-clairvoyant scheduling with predictions. In Kunal Agrawal and Yossi Azar, editors, *SPAA '21: 33rd ACM Symposium on Parallelism*

in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021, pages 285–294. ACM, 2021.

[70] Sandy Irani and Vitus J. Leung. Scheduling with conflicts, and applications to traffic signal control. In Éva Tardos, editor, *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, 28-30 January 1996, Atlanta, Georgia, USA*, pages 85–94. ACM/SIAM, 1996.

[71] Sandy Irani and Vitus J. Leung. Probabilistic analysis for scheduling with conflicts. In Michael E. Saks, editor, *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*, pages 286–295. ACM/SIAM, 1997.

[72] Samin Jamalabadi, Chris Schwiegelshohn, and Uwe Schwiegelshohn. Commitment and slack for online load maximization. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 339–348. ACM, 2020.

[73] Zhihao Jiang, Debmalya Panigrahi, and Kevin Sun. Online algorithms for weighted paging with predictions. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 69:1–69:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[74] Howard J. Karloff and David B. Shmoys. Efficient parallel algorithms for edge coloring problems. *J. Algorithms*, 8(1):39–52, 1987.

[75] Richard M. Karp, Umesh V. Vazirani, and Vijay V. Vazirani. An optimal algorithm for on-line bipartite matching. In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 352–358. ACM, 1990.

[76] Thomas Kesselheim, Klaus Radke, Andreas Tönnis, and Berthold Vöcking. An optimal online algorithm for weighted bipartite matching and extensions to combinatorial auctions. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*, pages 589–600. Springer, 2013.

[77] Henry A Kierstead and William T Trotter. An extremal problem in recursive combinatorics. In *Twelfth Southeastern Conference on Combinatorics, Graph Theory and Computing, Vol. II, Baton Rouge, LA, USA, March 1981*, volume 33 of *Congressus Numerantium*, pages 143–153, 1981.

[78] Ravi Kumar, Manish Purohit, Aaron Schild, Zoya Svitkina, and Erik Vee. Semi-online bipartite matching. In Blum [22], pages 50:1–50:20.

[79] Ravi Kumar, Manish Purohit, Zoya Svitkina, and Erik Vee. Interleaved caching with access graphs. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1846–1858. SIAM, 2020.

[80] David Kurokawa, Ariel D. Procaccia, and Junxing Wang. When can the maximin share guarantee be guaranteed? In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 523–529. AAAI Press, 2016.

[81] John Kuszmaul. Bamboo trimming revisited: Simple algorithms can do well too. *CoRR*, abs/2201.07350, 2022.

[82] William Kuszmaul. Achieving optimal backlog in the vanilla multi-processor cup game. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1558–1577. SIAM, 2020.

[83] William Kuszmaul. How asymmetry helps buffer management: achieving optimal tail size in cup games. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1248–1261. ACM, 2021.

[84] William Kuszmaul and Alek Westover. The variable-processor cup game. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, volume 185 of *LIPIcs*, pages 16:1–16:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[85] Silvio Lattanzi, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Online scheduling via learned weights. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1859–1877. SIAM, 2020.

[86] Thomas Lavastida, Benjamin Moseley, R. Ravi, and Chenyang Xu. Learnable and instance-robust predictions for online matching, flows and load balancing. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 59:1–59:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[87] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 3302–3311. PMLR, 2018.

[88] Nicole Megow and Lukas Nölke. Online minimum cost matching with recourse on the line. In Jaroslaw Byrka and Raghu Meka, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020, August 17-19, 2020, Virtual Conference*, volume 176 of *LIPIcs*, pages 37:1–37:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[89] Aranyak Mehta and Debmalya Panigrahi. Online matching with stochastic rewards. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 728–737. IEEE Computer Society, 2012.

[90] Aranyak Mehta, Amin Saberi, Umesh V. Vazirani, and Vijay V. Vazirani. Adwords and generalized online matching. *J. ACM*, 54(5):22, 2007.

[91] Aranyak Mehta, Bo Waggoner, and Morteza Zadimoghaddam. Online stochastic matching with unequal probabilities. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1388–1404. SIAM, 2015.

[92] Krati Nayyar and Sharath Raghvendra. An input sensitive online algorithm for the metric bipartite matching problem. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 505–515. IEEE Computer Society, 2017.

[93] Christos H. Papadimitriou, Tristan Pollner, Amin Saberi, and David Wajc. Online stochastic max-weight bipartite matching: Beyond prophet inequalities. In Péter Biró, Shuchi Chawla, and Federico Echenique, editors, *EC '21: The 22nd ACM Conference on Economics and Computation, Budapest, Hungary, July 18-23, 2021*, pages 763–764. ACM, 2021.

[94] Enoch Peserico and Michele Scquizzato. Matching on the line admits no o($\sqrt{\text{log n}}$)-competitive algorithm. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 103:1–103:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[95] Manish Purohit, Zoya Svitkina, and Ravi Kumar. Improving online algorithms via ML predictions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[96] Sharath Raghvendra. A robust and optimal online algorithm for minimum metric bipartite matching. In Klaus Jansen, Claire Mathieu, José

D. P. Rolim, and Chris Umans, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX-/RANDOM 2016, September 7-9, 2016, Paris, France*, volume 60 of *LIPIcs*, pages 18:1–18:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[97] Sharath Raghvendra. Optimal analysis of an online algorithm for the bipartite matching problem on a line. In Bettina Speckmann and Csaba D. Tóth, editors, *34th International Symposium on Computational Geometry, SoCG 2018, June 11-14, 2018, Budapest, Hungary*, volume 99 of *LIPIcs*, pages 67:1–67:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[98] Rebecca Reiffenhäuser. An optimal truthful mechanism for the online weighted bipartite matching problem. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1982–1993, 2019.

[99] Dhruv Rohatgi. Near-optimal bounds for online caching with machine learned advice. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1834–1845. SIAM, 2020.

[100] Amin Saberi and David Wajc. The greedy algorithm is not optimal for on-line edge coloring. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 109:1–109:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[101] Erel Segal-Halevi. Fairly dividing a cake after some parts were burnt in the oven. In Elisabeth André, Sven Koenig, Mehdi Dastani, and Gita Sukthankar, editors, *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 1276–1284. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018.

[102] Jirí Sgall. Open problems in throughput scheduling. In Leah Epstein and Paolo Ferragina, editors, *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, volume 7501 of *Lecture Notes in Computer Science*, pages 2–11. Springer, 2012.

[103] Yongho Shin and Hyung-Chan An. Making three out of two: Three-way online correlated selection. *CoRR*, abs/2107.02605, 2021. To appear in ISAAC 2021.

[104] Hugo Steinhaus. The problem of fair division. Econometrica, 16:101-104, 1948.

[105] Walter Stromquist. How to cut a cake fairly. American Mathematical Monthly, 87(8):640–644, 1980.

[106] Francis E. Su. Rental harmony: Sperner's lemma in fair division. American Mathematical Monthly, 106(10):930–942, 1999.

[107] Francis Edward Su. Cake-cutting algorithms: Be fair if you can. by Jack Robertson; William Webb. *The American Mathematical Monthly*, 107(2):185–188, 2000.

[108] Warut Suksompong. Fairly allocating contiguous blocks of indivisible items. In *Algorithmic Game Theory - 10th International Symposium, SAGT 2017, L'Aquila, Italy, September 12-14, 2017, Proceedings*, volume 10504 of *Lecture Notes in Computer Science*, pages 333–344. Springer, 2017.

[109] Martijn van Ee. A 12/7-approximation algorithm for the discrete bamboo garden trimming problem. *CoRR*, abs/2004.11731, 2020.

[110] Pavel Veselý, Marek Chrobak, Lukasz Jez, and Jirí Sgall. A ϕ-competitive algorithm for scheduling packets with deadlines. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 123–142. SIAM, 2019.

[111] Yajun Wang and Sam Chiu-wai Wong. Two-sided online bipartite matching and vertex cover: Beating the greedy algorithm. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 1070–1081. Springer, 2015.

[112] Alexander Wei. Better and simpler learning-augmented online caching. In Jaroslaw Byrka and Raghu Meka, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020, August 17-19, 2020, Virtual Conference*, volume 176 of *LIPIcs*, pages 60:1–60:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.