



UNIVERSITY OF

LIVERPOOL

Periodic Scheduling
Pinwheels, Pareto Surfaces, and Polycules
(and Powersort)

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy
by

Benjamin Mark Smith

August 2025

Acknowledgments I am profoundly indebted to my primary supervisor Sebastian Wild for his tireless guidance and support throughout this process. His intellectual rigour, keen skepticism, and powerful love of the game fundamentally shaped me as an academic. Thank you for teaching me, pushing me, and getting in the trenches with me to get this done.

I am deeply grateful to my secondary supervisor Leszek Gaşieniec for introducing me to Periodic Scheduling, which was the seed crystal of much of the following. Your calm encouragement and sage advice were invaluable.

To my other co-authors: Yuriy Biktairov, Wanchote Po Jiamjitrak, Namrata, William Cawley Gelling; thank you for the many ways you've contributed to this journey. Your enthusiasm, grit, and excellent ideas were instrumental to our shared work.

I'd also like to thank my examiners: Tomasz Radzik for your detailed feedback, and Viktor Zamaraev; both for your useful comments and for pointing Polyamorous Scheduling in the right direction at a very early stage.

To those who proofread my manuscript: My parents, Michael and Judi Smith, and friend Oliver Kim – thank you for keeping me relatively sane and this thesis somewhat readable.

Finally, to everyone else who has encouraged, distracted, or inspired me over the years; thank you. Academia is a collective endeavour, and this work exists because I did not have to pursue it alone.

Contents

Introduction	7
1 Introduction	8
1.1 Periodic Scheduling	8
1.1.1 Pinwheel Scheduling	8
1.1.2 Polyamorous Scheduling	10
1.2 Multiway Powersort	13
1.3 Contributions	14
I Periodic Scheduling	16
2 Related Work	17
2.1 The Periodic Scheduling Family	17
2.1.1 Density	20
2.1.2 Pinwheel Scheduling	21
2.2 Optimisation Periodic Scheduling Problems	23
2.2.1 Bamboo Garden Trimming	24
2.3 Polyamorous Scheduling	25
2.4 Related Problems	26
3 Preliminaries	28
3.1 Single Agent Problems	28
3.1.1 Pinwheel Scheduling	28
3.1.2 Bamboo Garden Trimming	31
3.2 Multi-Agent Problems	32
3.2.1 Decision Polyamorous Scheduling	32

3.2.2	Optimisation Polyamorous Scheduling	32
3.2.3	Additional Preliminaries	33
4	Towards the 5/6-Density Conjecture of Pinwheel Scheduling	36
4.1	The Pareto Surface	36
4.1.1	Small Frequency Conjecture	39
4.1.2	The Pareto Trie	40
4.1.3	Uniqueness	42
4.2	Engineering Pinwheel Scheduling	43
4.2.1	The Naïve Algorithm	43
4.2.2	The Optimised Algorithm	45
4.2.3	The Foresight Algorithm	49
4.2.4	Deciding Tight Feasibility	52
4.3	Engineering the 5/6 Surfaces	52
4.3.1	Core Algorithm	52
4.3.2	Constructing the Pareto Surface	53
4.3.3	Searching the Pareto Surface	55
4.4	Performance Evaluation	56
4.4.1	Pinwheel Schedulers	56
4.4.2	Constructing the 5/6 Pareto Surface	58
4.4.3	Searching the Pareto Surface	60
4.5	Conclusion	60
5	Polyamorous Scheduling	61
5.1	Results	61
5.2	Computational Complexity	64
5.3	Unweighted Polyamorous Scheduling & Edge Coloring	65
5.4	Approximation Algorithms	67
5.4.1	Lower Bounds	67
5.4.2	Approximation for Almost Equal Growth Rates	68
5.4.3	Layering Algorithm	69
5.5	Fractional Polyamorous Scheduling	70
5.5.1	Linear Programs for Polyamorous Scheduling	70
5.5.2	Poly Density	73

6	Simple Approximation Algorithms for Polyamorous Scheduling	75
6.1	Results	76
6.2	Approximating OPS	79
6.2.1	Lower Bounds	79
6.2.2	Reduce-Fastest(x)	82
6.2.3	Length of Schedules	87
6.3	Poly Density	89
6.3.1	Bounding Poly Density for OPS polycules	89
6.3.2	Poly Density of DPS Polycules	91
6.4	Inapproximability	95
6.4.1	Overview of Proof	95
6.4.2	Overview of Reduction	97
6.4.3	The True Clock & Colour Slots	98
6.4.4	Variables	99
6.4.5	Flippers	100
6.4.6	Duplication of Variables and Constants	102
6.4.7	Clauses	107
6.4.8	Tension	109
6.4.9	Proof of Reduction	110

II Multiway Powersort 112

7	Multiway Powersort	113
7.1	Introduction	114
7.1.1	Results	116
7.1.2	Related Work	117
7.2	Preliminaries	118
7.2.1	Lower Bound	118
7.2.2	Merging and its Memory-transfer Cost	118
7.2.3	Run-adaptive Mergesort	119
7.3	Multiway Powersort	120
7.3.1	Intuition and the Merge Tree	120
7.3.2	k-way Powersort	124
7.3.3	Analysis	124

7.3.4	k-way Peeksort	127
7.4	Results	130
7.4.1	Implementations and Compilation	130
7.4.2	Experimental Setup	131
7.4.3	Hypothesis 1: 4-way Powersort can yield significant performance im- provements	132
7.4.4	Hypothesis 2: Scanned elements explain the speedups	135
7.4.5	Hypothesis 3: 4-way Powersort halves merge cost	137
7.5	Conclusions	137
Conclusion		140
8	Conclusion	140
8.1	Future Work	141
8.1.1	Periodic Scheduling	141
8.1.2	Pinwheel Scheduling	142
8.1.3	Polyamorous Scheduling	142
8.1.4	Multiway Powersort	145
III Appendices		154
A	Original Inapproximability Proof	155
A.1	Overview of Proof	155
A.2	Reduction Overview	156
A.3	The True Clock & Colour Slots	159
A.4	Variables	160
A.5	Duplication of Variables and Constants	161
A.5.1	3-Duplicators	161
A.5.2	6-Duplicators	163
A.6	Clauses	165
A.7	Sorting Networks	167
A.7.1	12-Edge Duplicator	168
A.7.2	OR2 Gadget	170
A.7.3	AND2 Gadget	171

A.7.4	Slot Splitting Gadgets	173
A.7.5	SWAP Gadgets	174
A.7.6	Sorting Network Schedules	176
A.8	Tension	177
A.9	Correctness Proof of Reduction	178
B	Powersort – C++ Code	180
B.1	4-way merge with sentinels	180
B.2	3-way merge and 2-way merge	181
B.3	4-way Powersort	182
B.4	Sentinel-free 2-way merge	186
B.5	Sentinel-free 4-way merge (merging by stages)	187

Introduction

Periodic Scheduling

Pinwheels, Pareto Surfaces, and Polycules

Benjamin Mark Smith

Abstract *Pinwheel Scheduling* aims to find a perpetual schedule for unit-length tasks on a single machine subject to periodic deadlines or *frequencies*: maximal time spans between consecutive executions of each task. The density of a Pinwheel Scheduling instance is the sum of the inverses of these frequencies; in 1993, Chan and Chin conjectured that any Pinwheel Scheduling instance with a density of at most $5/6$ is schedulable – this conjecture stood for 31 years before being proved by Akitoshi Kawamura in 2024. We provided key elements of this proof that will be reproduced here: a formalized notion of *Pareto surfaces* (finite sets of solutions capable of solving infinite classes of Pinwheel Scheduling instances), the state-of-the-art oracle for arbitrary Pinwheel Scheduling instances, and 23 schedules which solve all Pinwheel Scheduling instances with at most 5 tasks.

Polyamorous Scheduling is the related problem of scheduling pairwise meetings between members of a complex social group, represented as an edge-weighted graph. The goal of Polyamorous Scheduling is to find a schedule that minimises the maximal weighted waiting time between consecutive occurrences of the same edge. We introduce this problem, giving both a 5.24-approximation and the first hardness-of-approximation result presented for a Periodic Scheduling problem. We also introduce *poly density* – a generalization of the density of Pinwheel Scheduling that serves to define both upper and lower bounds for Polyamorous Scheduling.

Our final topic is unrelated: *Multiway Powersort*, a stable mergesort variant that exploits existing runs and finds nearly-optimal merging orders for *k-way* merges with negligible overhead. This builds on Powersort (Munro & Wild, 2018), which has recently been adopted by the CPython reference implementation of Python, as well as by PyPy and other libraries. Multiway Powersort reduces the number of memory transfers, which increasingly determine the cost of internal sorting. We demonstrate that our 4-way Powersort implementation can achieve substantial speedups over 2-way Powersort and other stable sorting methods without compromising the optimally run-adaptive performance of Powersort.

Chapter 1

Introduction

This chapter will introduce the three key problems of this thesis: Pinwheel Scheduling, Polyamorous Scheduling, and Multiway Powersort.

1.1 Periodic Scheduling

Since the beginning of human civilization, people of all cultures have organised their lives as a series of circles within circles. Of the many, many tasks that each of us must perform, most must be performed indefinitely many times: we cook food, eat it, and must then prepare more; we clean the dog, walk her, and soon find ourselves back at the tub; we process a message from a satellite geostationary communications satellite, but what do you know – it soon transmits again.

There are many formalizations of Periodic Scheduling, each with its own quirks and results. These methods have many applications, from the mundane process of managing our lives to communications, periodic maintenance, automated manufacturing, and workforce management.

We will focus on two Periodic Scheduling problems: Pinwheel Scheduling, where the theory of Periodic Scheduling really began, and Polyamorous Scheduling, a new problem of my own design.

1.1.1 Pinwheel Scheduling

Pinwheel Scheduling (or PWS) is the problem of indefinitely repeating a set of tasks with periodic deadlines while limited to performing a single task each day. For example, consider

$\mathcal{P} = [3, 4, 5, 8]$ – a satisfying schedule must schedule $f_1 = 3$ at least once every 3 days, $f_2 = 4$ at least once every 4 days, $f_3 = 5$ at least once every 5 days, and $f_4 = 8$ at least once in every 8-day period. A solution for this instance is shown in Figure 1.1:

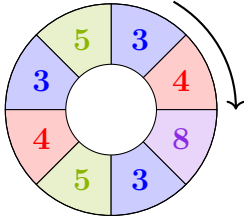


Figure 1.1: A schedule for the $[3,4,5,8]$ instance of Pinwheel Scheduling.

The key difficulty of Pinwheel Scheduling is the collisions which are unavoidable when scheduling many instances. As an example, consider $\mathcal{P} = [2, 3, *]$. As can be seen in Figure 1.2, any attempt to schedule this instance is doomed by the collisions between $f_1 = 2$ and $f_2 = 3$, which fill any gap that the last task ($f_3 = *$) could use, whatever its frequency.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
2		2		2		2		2		2		2		2		...
2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	...

Figure 1.2: A demonstration that the $[2,3,*]$ instance of Pinwheel Scheduling has no valid schedule: first schedule $f_1 = 2$, then $f_2 = 3$; observe that no space remains for the third task.

Pinwheel Scheduling is NP-hard[JL14], intuitively due to the potential snowballing effect of even the smallest change to a partial schedule. Even if we successfully schedule most tasks in some instance \mathcal{P} , resolving a single collision by scheduling a single task f a single day early could have significant consequences. Scheduling f early in one location moves its deadline one day earlier forever; if our partial schedule lacks the slack to deal with this, we will end up having to move other tasks, potentially sending us all the way back to the drawing board. This domino effect means that Pinwheel schedules are often long, intricate, and messy constructions.

Crucial to the study of Pinwheel Scheduling is the notion of *density* – a relaxation in which we forget about collisions momentarily and consider what fraction of our schedule could be spent resting if we did not have to worry about them. Consider again $\mathcal{P} = [2, 3, *]$:

in this relaxation, we must spend $\frac{1}{2}$ of our schedule on $f_1 = 2$, $\frac{1}{3}$ on $f_2 = 3$, and a small ε on $f_3 = *$. However, as we show in Figure 1.2, this instance has no solution – density $d = \sum_{i=1}^k \frac{1}{f_i}$ must be at most 1, but this is not sufficient for an instance to be solvable.

In their 1993 paper [CC93], Chan and Chin conjectured that any instance less dense than this has a schedule:

Question 1.1.1 5/6 Density Conjecture.

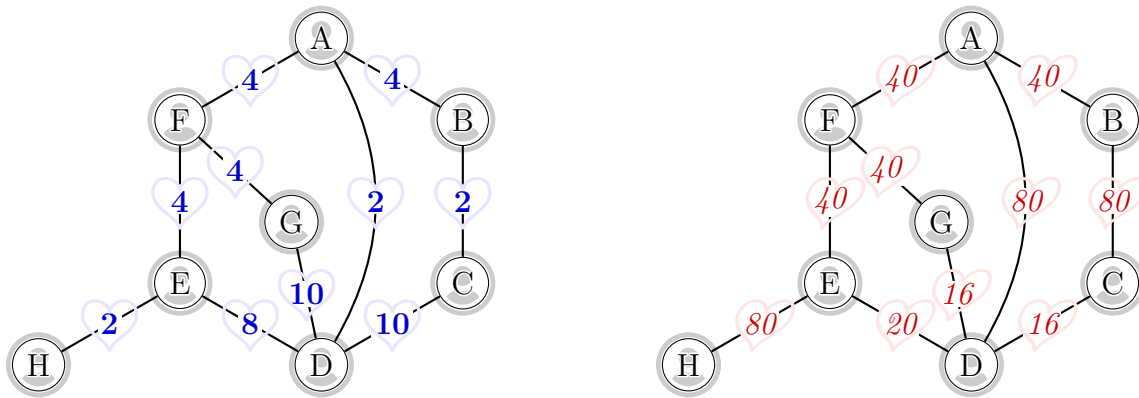
Every Pinwheel Scheduling instance with density $d \leq \frac{5}{6}$ is schedulable.

Chapter 4 introduces methods and results from our 2022 paper [GSW22] which provided support for this conjecture in several ways. We formalized the notion of *Pareto Surfaces* for Pinwheel Scheduling, proving that each member of several infinite classes of Pinwheel Scheduling instances can be solved by a schedule selected from a finite *Pareto Trie*. We then presented two highly engineered depth-first search algorithms: an algorithm which generates this Pareto Trie and the state-of-the-art oracle for arbitrary Pinwheel Scheduling problems. Finally, we used these algorithms to provide schedules for all Pinwheel Scheduling instances with at most five tasks and to prove Question 1.1.1 for all Pinwheel Scheduling instances with at most twelve tasks.

Question 1.1.1 went unsolved for 31 years before Akitoshi Kawamura proved it in 2024 [Kaw24], using a somewhat similar approach. This breakthrough paper generalized Pinwheel Scheduling to allow non-integer periods, showed that each Pinwheel Scheduling instance with a density $d \leq \frac{5}{6}$ can be solved by modifying a schedule for one member of a finite set of Pinwheel Scheduling instances, then scheduled these instances computationally (aided by some of our techniques).

1.1.2 Polyamorous Scheduling

In Chapters 5 and 6, we turn our attention to a natural Periodic Scheduling problem faced by groups of regularity-loving polyamorous people: Consider a set of persons connected by a set of pairwise relationships, with a value representing the neediness, importance, or emotional weight of each pairing. Find a periodic schedule of pairwise meetings between couples that minimizes the maximal weighted waiting time between such meetings, given that each person can meet with at most one of their partners on any particular day.



Day	A-B	A-D	A-F	B-C	C-D	D-E	D-G	E-F	E-H	F-G
0		♥		♥					♥	♥
1	♥				♥			♥		
2		♥		♥					♥	
3			♥				♥			
4		♥		♥					♥	♥
5	♥							♥		
6		♥		♥					♥	
7			♥			♥				

Figure 1.3: An example Polyamorous Scheduling instance with 8 persons: Adam, Brady, Charlie, Daisy, Eli, Frannie, Grace, and Holly. **Top left:** a Graph representation of the decision version with **edge labels** showing the meeting frequency required by each relationship. **Top right:** An optimization version of the same instance with **edge labels** showing the weight (desire growth rates) of each pairwise relationship. **Bottom:** A periodic schedule which solves the decision version and optimally solves the optimization version. The set of meetings scheduled each day are indicated by ♥s.

Before formally defining Optimisation Polyamorous Scheduling (OPS) and Decision Polyamorous Scheduling (DPS), we will illustrate some of their features using an example. Figure 1.3 shows an instance (or *polycule*) using the natural graph-based representation: We represent each person as a vertex and each relationship as an undirected weighted edge.

The optimization version is shown on the top-right, with an analogous decision version on the top-left and a schedule for both on the bottom. It is easy to check that this schedule never assigns more than one daily meeting to any of the eight persons in the group; in the graph representation, this means that the set of meetings for each day must form a matching in the graph.

We begin by considering *Optimisation Polyamorous Scheduling*. Each day, every couple’s mutual desire to meet grows by the weight or *desire growth rate* of that relationship¹ – that is, until a meeting occurs, satisfying this desire and returning it to zero. We will refer to the highest desire ever felt by any pair when following a given schedule as the *heat* of that schedule. The heat of the schedule in Figure 1.3 is 160; as the reader can verify, no pair ever feels a desire greater than 160 at any point. Desire 160 is also reached; e.g., Adam and Daisy are scheduled to see each other every other day, but over the period of 2 days between subsequent meetings, their desire grows to $2 \cdot 80 = 160$.

For the instance in Figure 1.3, it is also easy to show that no schedule with heat < 160 can exist. For that, we first convert from desire growth rates to required *frequencies*; under a heat-160 schedule, a pair with desire growth rate g must meet at least every $\lfloor 160/g \rfloor$ days. The top-left part of Figure 1.3 shows the resulting *Decision Polyamorous Scheduling* polycule. It is easy to check that the given schedule indeed achieves these frequencies. However, any further reduction of the desired heat to $160 - \varepsilon$ would leave, e.g., Adam hopelessly overcommitted: the relationship with Daisy would need a frequency of $\lfloor (160 - \varepsilon)/80 \rfloor = 1$, forcing them to meet *every* day; but then Brady and Frankie, each with frequency $\lfloor (160 - \varepsilon)/40 \rfloor \leq 3$ cannot meet with Adam at all.

While local arguments suffice for our small example, we will show that Polyamorous Scheduling is, in general, NP-hard. We will, therefore, focus on approximation algorithms and inapproximability results. Chapter 5 will present the results of our first paper concerning Polyamorous Scheduling [GSW24], while Chapter 6 contains the findings of our second such paper [Bik+25].

Note on Conventions and Style As we will show in Section 2.1, there are many Periodic Scheduling problems with issues regarding their naming. I chose the name "Polyamorous Scheduling" because I believe that problems should have persistent, consistent, eye-catching names that clearly distinguish them from similar problems; but also for one other reason: Polyamorous Scheduling [GSW24] was originally introduced in the 12th International Conference on Fun with Algorithms (FUN2024). I have retained and expanded on the humorous style called for by this venue.

¹“Remember, absence makes the heart grow fonder” [CAG73].
(<https://getyarn.io/yarn-clip/ae628721-c1d1-49d1-bd7c-78cbffceabf0>)



1.2 Multiway Powersort

Sorting lists of objects is another foundational problem with an extremely long history [Hol89; Hol17]. To the simplest of analyses, “standard” sorting is a solved problem: comparison-based internal sorting of simple objects using a single thread on a single machine requires $\mathcal{O}(n \log(n))$ comparisons, and there are multiple methods which are best-possible from this perspective. Versions of Quicksort and Mergesort are both commonly used: the former for its average case performance, the latter for its stability and its worst-case guarantees. If we look a little deeper, however, there is more to be gained than mere constant-factor performance improvements.

In the worst case, elements are initially in either a random order or an order specifically designed to foil our algorithm. *Adaptive* sorting methods look beyond these adversarial inputs; many practical applications involve sorting data which already has some exploitable structure, and to that we turn now. There are multiple measures of presortedness, including the number of inversions in the input, the number of elements outside of their final positions, and the number of *runs*: regions of the input which are already in sorted order.

Part II introduces *Multiway Powersort*, an adaptive sorting method first described in our 2023 paper of the same name [Gel+23]. Powersort is a version of mergesort which adapts to runs in the input, but doing so raises a new problem: in what order should we merge the runs that we find? The traditional bottom-up mergesort begins by treating each element as a run of length 1, then repeatedly passes through the input from left to right, merging each run with its neighbour as it passes. Because each initial run has the same length, the resultant merge tree is a complete and balanced binary tree. Figure 1.4 shows how two features of traditional merge sort separate when we begin with runs: on the left, we see that naively merging each run with its neighbor results in a long run being merged $\mathcal{O}(\lg(n))$ times for a merge cost of $\mathcal{O}(n \lg(n))$; on the right, we see that the optimal merge order only touches this long run once, for an $\mathcal{O}(n)$ merge cost. Powersort [MW18] takes the latter strategy, finding a nearly-optimal merge order based on nearly-optimal binary search trees with only minimal overheads.

The true cost of sorting is, in general, more complicated than counting the number of comparisons used: every time two runs are sent to the processor to be merged, they must be moved from the RAM to the processor cache and back again. Multiway merging avoids this: by increasing the number of runs merged at once, we reduce the number of times each element takes that round trip. As well as introducing Multiway Powersort, Chapter 7 will

show that 4-way Powersort halves the merge cost of 2-way Powersort and can yield significant performance improvements. It will also show that abstract cost measures like merge cost explain this speedup.

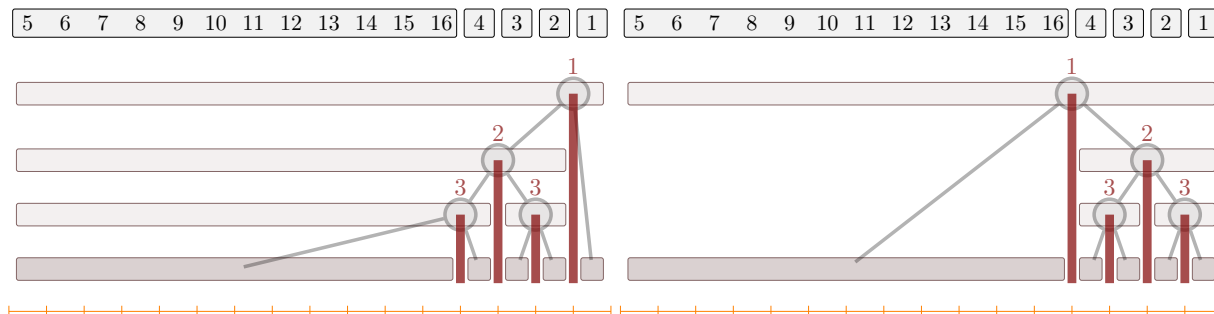


Figure 1.4: Example merge trees using a naive merge order (left) and the optimal merge order (right) for an input of length $n = 16$ with one run of length $n - \sqrt{n}$ followed by \sqrt{n} runs of length 1. Runs, represented by grey bars, are merged in grey circles from bottom to top. The naive merge order merges the long run $\mathcal{O}(\lg(n))$ times for a total cost of $\mathcal{O}(n \lg(n))$ or 46 merged elements; the optimal merge order merges the long run once, for a total cost of $\mathcal{O}(n)$ or 24 merged elements.

1.3 Contributions

The majority of this thesis has been published in four papers:

1. Leszek Gąsieniec, Benjamin Smith and Sebastian Wild. ‘Towards the 5/6-Density Conjecture of Pinwheel Scheduling’. In: *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. ed. by C. A. Phillips and B. Speckmann. SIAM, Jan. 2022, pp. 91–103
2. Leszek Gąsieniec, Benjamin Smith and Sebastian Wild. ‘Polyamorous Scheduling’. In: *12th International Conference on Fun with Algorithms (FUN 2024)*. Schloss Dagstuhl—Leibniz Center for Informatics. 2024
3. Yuriy Biktairov, Leszek Gąsieniec, Wanchote Po Jiamjitrak, Namrata, Benjamin Smith and Sebastian Wild. ‘Simple approximation algorithms for Polyamorous Scheduling’. In: *2025 Symposium on Simplicity in Algorithms (SOSA)*. SIAM. 2025, pp. 290–314

4. William Cawley Gelling, Markus E. Nebel, Benjamin Smith and Sebastian Wild. ‘Multiway Powersort’. In: *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)* (2023), pp. 190–200

“**Towards the 5/6-Density Conjecture of Pinwheel Scheduling**” was my first publication. I contributed the key ideas of this paper: I designed an algorithmic oracle for arbitrary Pinwheel Scheduling problems, developed the Pareto trie and Pareto surface, and proposed the 2^{k-1} conjecture (Conjecture 4.1.4). I also implemented both this oracle and algorithms designed to generate the Pareto trie, evaluated their performance experimentally, was the primary author of the text, and presented the paper at the conference.

“**Polyamorous Scheduling**” introduced two novel scheduling problems: Decision Polyamorous Scheduling and Optimisation Polyamorous Scheduling – both were my original ideas. Aside from this, I introduced the first nontrivial hardness-of-approximation result ever demonstrated for a Periodic Scheduling problem (Theorem 5.1.1). I also wrote significant portions of the paper, finding example problems such as Figures 5.2 and 5.3, and introducing the Secure and Fungible variants as open problems (Definitions 8.1.5 and 8.1.4).

My primary contribution to “**Simple approximation algorithms for Polyamorous Scheduling**” was the extension of Theorem 5.1.1 to bipartite graphs (Theorem 6.1.3), but I also contributed the final length of schedules proof (Section 6.2.3) and heavily edited the final paper.

In addition to presenting “**Multiway Powersort**” at ALENEX in 2023[Gel+23], I also aided in the generalization from 2-way Powersort to 4-way Powersort and extended this to the version of k -way Powersort that was finally published, as well as performing preliminary experiments and proposing the use of virtual sentinels (Section 8.1.4).

* * * * *

Part I

Periodic Scheduling

Chapter 2

Related Work

This chapter begins by introducing several Standard Periodic Scheduling problems, surveying known hardness and density results. We then give a more detailed review of the literature concerning Pinwheel Scheduling before turning our attention to Bamboo Garden Trimming and other related problems.

2.1 The Periodic Scheduling Family

Pinwheel Scheduling is at the heart of a cluster of closely related Periodic Scheduling problems which often share features and even names, rendering their taxonomy noteworthy and challenging [JL14; Kaw24]. Building on the discussion of problem variants by Jacobs and Longo [JL14], I define *Standard* Periodic Scheduling problems as those which have a set of periodic tasks with associated integer frequencies $(f) = f_1 \leq f_2 \leq \dots \leq f_k$ and are described using only those choices laid out in Table 2.1; also defining *Exotic* Periodic Scheduling problems as those requiring additional concepts (though many of these features are still useful when describing such problems). As defined here, Standard Periodic Scheduling problems are decision problems; Section 2.2 examines a group of optimisation problems which reduce each to a collection of standard Periodic Scheduling problem instances by replacing fixed deadlines/budgets with cost functions.

This typology uses several terms that we will formally introduce later on:

Standard Density d and Daily Work Capacity W are introduced in Section 2.1.1, along with a broad discussion of density. Note that all Dense Standard Periodic Scheduling problems are necessarily Exact: we can consider Exact problems in general, Dense problems in particular, or Inexact problems – there are no Dense Inexact problems.

Schedules here are finite sequences of assignments, i.e. mappings from the set of agents to the set of tasks; a valid schedule for a given instance is one that obeys the frequency, task duration, and workload constraints of that instance. Section 3.1 formally defines schedules for Pinwheel Scheduling and Bamboo Garden Trimming.

While schedules are certificates, alternative certificates include compressed schedules, algorithms proven to produce valid schedules for certain problems, or other proofs that an instance permits a valid schedule. Note that the existence of an infinite valid schedule implies the existence of a finite valid schedule, but not that such a schedule is easy to identify.

We will follow the convention of [BL03] and others by calling schedules which solve the exact version of a Periodic Scheduling problem *Exact* or *Perfect* schedules.

	Decision	Selection	Term
1	In every f_i day period, task i must be performed	exactly once	Exact
		at least once	Inexact
2	Task i takes	unit time	Unit Tasks
		non-unit time	Non-Unit Tasks
3	Tasks are assigned to	one agent	Single-Agent
		agent $a_j \in A$	Multi-Agent
4	In each day, agent a_j can perform	$b_j = 1$ task	Unit Workload(s)
		$b_j \geq 1$ tasks ¹	Busy Workload(s)
5	Standard Density d	equals W	Dense
		is unbounded by W	Non-Dense
6	$f_1, f_2,$ and f_3 have the same frequency, so	$f = (f_1, f_2, f_3)$	Standard Representation
		$f = (f_1, 3)$	Compact Representation
7	The output for a solvable instance is	a schedule	Standard Version
		any other certificate	Decision Version

¹note that several sources describe the case where one agent can do multiple tasks per day as multiple agents who can freely share tasks – there is no mathematical difference.

Table 2.1: A taxonomy of the features of Standard Periodic Scheduling problems.

Standard Periodic Scheduling Problems A wide variety of problems in the literature fit into the taxonomy of Standard Periodic Scheduling laid out in Table 2.1, including:

- *Pinwheel Scheduling* [Hol+89]: Inexact Periodic Scheduling with unit tasks and a single agent with a unit workload.

- *Dense Pinwheel Scheduling* [Hol+89]: Dense Periodic Scheduling with unit tasks and a single agent with a unit workload.
- *Exact Pinwheel Scheduling*¹ [WL83; Bar+02]: Exact Periodic Scheduling with unit tasks and a single agent with a unit workload.
- *Windows Scheduling*² [BL03]: Inexact Periodic Scheduling with unit tasks and a single agent with a busy workload.
- *Exact Windows Scheduling*³ [WL83; BL03]: Exact Periodic Scheduling with unit tasks and a single agent with a busy workload.
- *Bin Scheduling*⁴ [BL03; BLT07]: Inexact Periodic Scheduling with unit tasks and multiple agents with unit workloads.
- *Exact Bin Scheduling*⁵ [BL03; BLT07]: Exact Periodic Scheduling with unit tasks and multiple agents with unit workloads.
- *Generalized Pinwheel Scheduling* [FC05]: Inexact Periodic Scheduling with non-unit tasks and a single agent with a unit workload.

Note that while the nature of the deadlines, tasks, and agents define the above problems, the representation and version only need to be stated for some technical results, such as those concerning hardness.

Hardness of Periodic Scheduling Every Standard Periodic Scheduling problem named above has been shown, in some respect, to be NP-hard: Exact Pinwheel Scheduling is NP-complete [Bar+02], and is a special case of both Exact Windows Scheduling and Exact Bin Scheduling. Bar-Noy et al. [BLT07] show that the hardness of Exact Windows scheduling implies the hardness of Dense Windows Scheduling, which is a subset of Windows scheduling;

¹Studied as the one-server case of the Machine Maintenance problem by [WL83] and the *Periodic Maintenance Scheduling Problem* by [Bar+02]

²Various sources use Windows Scheduling to refer to multiple problems in this class or to other specific Periodic Scheduling problems

³Called the *Machine Maintenance* problem by [WL83], and *Perfect Schedules for Windows Scheduling* by [BL03]

⁴Called *Windows Scheduling without migration* [BLT07], and distinguished from what we call Windows Scheduling by the first remark in [BL03]

⁵Studied as a variant of Windows Scheduling by [BL03], and as *Windows Scheduling with exact gaps and without migration* by [BLT07]

Jacobs and Longo [JL14] noticed that this proof also applies to Exact Pinwheel Scheduling, showing Pinwheel Scheduling and Dense Pinwheel Scheduling to be NP-hard in compact representation. Finally, both Bin Scheduling and Generalized Pinwheel Scheduling contain Pinwheel Scheduling as a special case. Fienberg and Curry [FC05] go even further, showing Generalized Pinwheel Scheduling to be strongly NP-hard, even in standard representation.

2.1.1 Density

Most Periodic Scheduling problems have one or more definable *densities*: relaxations that eliminate the need for a collision-free schedule while retaining work capacity limitations. Some problems allow density relaxations which also capture other features. Densities easily identify classes of unsolvable instances for multiple problems, but can also identify classes of solvable instances, usually by producing algorithms which can schedule all instances below a given density threshold.

In Pinwheel Scheduling, a task with frequency f_i must constitute at least $\frac{1}{f_i}$ of any valid schedule, at most 100% of which can be assigned to the set of all tasks: thus the density of a Pinwheel Scheduling instance $\mathcal{P} = (f)$ is given by $d = \sum_{i=1}^k \frac{1}{f_i}$, and $d \leq 1$ is a necessary condition for \mathcal{P} to be schedulable. This is not sufficient, as shown by the infeasible instance $(2, 3, M)$ with $d = \frac{5}{6} + 1/M$ for any $M \in \mathbb{N}$. Multiple authors have developed the usefulness of the density of Pinwheel Scheduling as a lower bound, which will be discussed in Section 2.1.2.

While introducing Windows Scheduling, Bar-Noy and Ladner [BL03] used the same measure of density $d = \sum_{i=1}^k \frac{1}{f_i}$, showing that at least $\lceil d \rceil$ agents⁶ are needed for an instance to be schedulable. They also introduced an algorithm which produces an exact schedule for any input using at most $d + e \ln(d) + 7.3595$ agents provided $d > 1$. Bar-Noy, Ladner, and Tamir [BLT07] later showed that the *Unit Fractions Bin Packing* problem, in which items of size $\frac{1}{f_i}$ must be packed into unit size bins, is another such relaxation for Bin Scheduling; at least as many agents will be needed to schedule $(f) = f_1, \dots, f_k$ as bins will be needed to pack $\frac{1}{f_1}, \dots, \frac{1}{f_k}$. Packing bins only up to density $5/6$ guarantees a Pinwheel schedule [Kaw24], at the expense of a $6/5$ factor increase in the agents needed, and even better exploitations of this result are probably possible.

Han and Lin [HL92] use an expanded definition of density which covers Periodic Scheduling problems with non-unit tasks: if task i has duration l_i , then the density becomes $d = \sum_{i=1}^k \frac{l_i}{f_i}$, inheriting the bound from Windows Scheduling that at least $\lceil d \rceil$ agents are

⁶often called channels

necessary for an instance to be schedulable. They present an algorithm which can schedule Generalized Pinwheel Scheduling problem instances with k tasks and density $d \leq k(2^{1/k} - 1)$. I define this the *Standard Density* of Periodic Scheduling problems:

Definition 2.1.1 Standard Density.

Any Standard Periodic Scheduling problem with tasks $i \in [k]$, task lengths l_1, \dots, l_k , and task frequencies $f_1 \leq f_2 \leq \dots \leq f_k$ has standard density $d = \sum_{i=1}^k \frac{l_i}{f_i}$.

Each Standard Periodic Scheduling problem also has a *Daily Work Capacity*:

Definition 2.1.2 Daily Work Capacity.

Any Standard Periodic Scheduling problem with agents $a_j \in A$, who can each perform b_j tasks per day has a Daily Work Capacity of $W = \sum_{j=1}^{|A|} b_j$.

In Chapter 5, we introduce *poly density* and show it to be an upper bound for Polyamorous Scheduling (introduced in Section 3.2); Chapter 6 shows this to be a generalization of the Standard Density for Pinwheel Scheduling and uses it to generate lower bounds.

2.1.2 Pinwheel Scheduling

The Pinwheel Scheduling problem (hence PWS) was originally proposed by Holte et al. in 1989 [Hol+89], with the intention of assigning receiver time slots to satellites with varying bandwidth requirements but a single shared ground station. Each Pinwheel Scheduling instance \mathcal{P} consists of k positive integer frequencies $f_1 \leq f_2 \leq \dots \leq f_k$. Schedulable instances are those which permit a valid Pinwheel Schedule: a finite sequence of tasks $1, \dots, k$ which can be repeated infinitely such that any contiguous time window of length f_i will contain at least one occurrence of task i , for all tasks. In Pinwheel Scheduling, our goal is to either find or report the non-existence of such a schedule.

Hardness

The complexity status of Pinwheel Scheduling has gained some notoriety in the literature. When they introduced Pinwheel Scheduling, Holte et al. [Hol+89] showed that Decision Pinwheel Scheduling in standard representation is in PSPACE. As introduced in Table 2.1, problems in standard representation are composed of a single multiset of positive integer frequencies $\mathcal{P}_=(f)$, where $f = f_1 \leq f_2 \leq \dots \leq f_k$. In compact representation, f instead becomes a set, and the instance is expressed as $\mathcal{P}_=(f, x)$, where each element $x_i \in x$ corresponds to some frequency $f_i \in f$ and represents the number of tasks with that frequency.

Holte et al. also introduced *Dense Pinwheel Scheduling*, whose density $d = 1$, showed that Dense Decision Pinwheel Scheduling in standard representation is in NP, and claimed that Dense Decision Pinwheel Scheduling in compact representation is NP-hard; however, it seems that their proof of this was never published. Jacobs and Longo [JL14]⁷ showed that PWS is NP-hard in the compact representation, using two earlier results: the proof by Bar-Noy et al. [Bar+02] that Exact Pinwheel Scheduling in standard representation is NP-complete (via a reduction from graph coloring), and the proof by Bar-Noy et al. in [BLT07] that the NP-hardness of Exact Windows Scheduling in compact representation implies that Windows Scheduling in compact representation is also NP-hard (using dummy tasks, with a proof that also applies to the single-agent case).

Jacobs and Longo [JL14] also showed that PWS does not even admit a pseudo-polynomial time algorithm unless SAT can be solved by a randomized method in expected $n^{\mathcal{O}(\log n \log \log n)}$ time. However, my co-authors and I demonstrated in [GSW22] that PWS is fixed parameter tractable with respect to the number of tasks k (Corollary 4.1.2).

Whether PWS is in NP in either representation remains open, as schedules can be exponentially long, and it is not known whether the most general case allows shorter certificates.

Density

Since the introduction of Pinwheel Scheduling, it has been known that there is a threshold d^* such that $d \leq d^*$ implies schedulability and for each $\epsilon > 0$ there is an instance with density $d = d^* + \epsilon$ which cannot be scheduled. Whenever $d \leq \frac{1}{2}$, we can replace each frequency f_i by $2^{\lfloor \lg(f_i) \rfloor}$ without increasing d above 1, giving a periodic schedule which uses the largest frequency as the period. In their 1992 paper [CC93], Chan and Chin introduced the “5/6 Conjecture”, Question 1.1.1, which has motivated several papers improving the bounds on d^* : from 1/2 [Hol+89], to 2/3 [CC93], 7/10 [CC92], 3/4 [FL02], to its final value of 5/6 by Kawamura in 2024 [Kaw24].

Question 1.1.1 (restated). 5/6 Density Conjecture.

Every Pinwheel Scheduling instance with density $d \leq \frac{5}{6}$ is schedulable.

Several papers built up to Kawamura’s proof of the 5/6 density conjecture, with independently interesting results. In his 2020 paper, Ding [Din20] manually applied exhaustive case distinction to produce a finite set of schedules which solve all Pinwheel Scheduling problems with $d \leq \frac{5}{6}$ and $k \leq 5$ tasks, proving the 5/6 conjecture up to $k = 5$. Our 2022

⁷Note that this paper is an arxiv preprint and has not, to my knowledge, been peer reviewed.

paper [GSW22] formalizes the notion of Pareto surfaces for PWS, proving that $\forall k \in \mathbb{N}$ there exists a finite set of schedules which solve all schedulable PWS instances with at most k tasks. We also implemented the best known oracle for arbitrary PWS instances and general algorithms to efficiently automate the generation of Pareto surfaces. These algorithms substantially reduce the effort to verify the completeness of a Pareto surface: for $k = 11$, they reduced the number of calls to an oracle for this NP-hard problem from tens of thousands down to just 37. Using these methods, we published a set of 23 schedules that collectively solve all schedulable PWS instances with $k \leq 5$ and proved the 5/6 conjecture for $k \leq 12$. To do this, we exploited several novel insights into PWS, including methods like “folding” large PWS instances to reduce them to smaller instances and “unfolding” solved PWS instances to apply their schedules to larger instances. These methods and results are presented in detail in Chapter 4.

In his 2024 paper [Kaw24], Kawamura extended the idea of folding using fractional PWS tasks to generate a finite set of PWS instances whose schedulability demonstrates the truth of the 5/6 conjecture; he then showed that these instances are schedulable using exhaustive methods incorporating some of our techniques, proving this 32-year-old conjecture.

Key Results

An orthogonal line of work considered all Pinwheel Scheduling instances with a fixed number of *distinct* values for frequencies (but an arbitrary number of tasks), first for 2 distinct frequencies [Hol+89; Hol+92] and later for 3 [LL97].

Efficient algorithms for computing schedules are sometimes also considered; here the potential necessity of exponentially long periodic schedules motivated the introduction of “fast online schedulers”: simple programs that produce schedules from beginning to end, avoiding the need to retain the full schedule in memory [Hol+89; HL98].

2.2 Optimisation Periodic Scheduling Problems

Many Periodic Scheduling problems have optimisation variants, substituting periodic deadlines and pass-fail checks for cost functions. One of the earliest and most influential of these is the *Optimal Windows Scheduling* problem [BL03; BLT07], which asks for the smallest number of agents with unit workloads who can solve a particular set of unit tasks.

Another way to formulate optimisation variants of Standard Periodic Scheduling problems is to replace the deadline of each task with a "badness" that we will call height, h_i , and a growth rate g_i . When applied to Pinwheel Scheduling, this gives Bamboo Garden Trimming (BGT), which we will address in the next section, but broadly these problems create a series of decision problems and ask which are schedulable. Windows Scheduling has an optimisation version called *Multi-Processor Bamboo Garden Trimming* [Kus22], though decent algorithms for the multi-processor case can be derived from any algorithm for Bamboo Garden Trimming, to which we turn now.

2.2.1 Bamboo Garden Trimming

The *Bamboo Garden Trimming Problem* (hence BGT) is an optimisation variant of Pinwheel Scheduling introduced in 2017 [Gaş+17]. BGT replaces frequencies $f_1 \leq f_2 \leq \dots \leq f_k$ with daily *growth rates* $g_1 \leq g_2 \leq \dots \leq g_k$ (of k bamboo plants $1, \dots, k$). Each day, a single gardener selects a plant to cut (reducing its height to zero) – the task in BGT is to find a perpetual schedule of cuts that minimizes the maximum *height* ever reached by any of these plants⁸. Analogously to the density of PWS, the height h_{\max} is lower bounded by the total growth volume⁹ $G = \sum_{i=1}^k g_i$.

BGT permits efficient constant-factor approximations whose approximation factor has seen a lively race of successive improvements over the last few years: from 2 [Gaş+17], $\frac{32000}{16947} \approx 1.89$ [Del20], $\frac{12}{7} \approx 1.71$ [Ee21], 1.6 [Gaş+24], and $\frac{10}{7} \approx 1.43$ [HS23], down to the current record, $\frac{4}{3} \approx 1.33$, again by Kawamura [Kaw24].

Various online greedy approaches have been applied to BGT: As well as introducing BGT, Gaşieniec et al. [Gaş+17] introduced *Reduce-Max*, in which the tallest plant is trimmed every day, and *Reduce-Fastest*(x), which trims the fastest growing plant whose height is above a threshold x . Kuszmaul [Kus22] introduced *Deadline-Driven* scheduling, which uses two thresholds and trims the plant above the lower threshold which will be the first to reach the upper threshold.

When introduced, Reduce-Max was shown to have an approximation ratio of $\mathcal{O}(\log n)$ [Gaş+17], but this paper also speculated that Reduce-Max may keep the maximum bamboo height within $\mathcal{O}(G)$, i.e. have an approximation ratio of $\mathcal{O}(1)$. Bilò et al. [Bil+22] confirmed this, proving a bound of $9G$ on the maximum bamboo height under the Reduce-Max algorithm; this was later improved to $4G$ by Kuszmaul [Kus22] – the best bound currently

⁸Called *backlog* by [Kus22] and *makespan* by [Bil+22]

⁹Various sources, including [Gaş+17] use H for the total growth volume.

known. In their later journal paper, Ga̧sieniec et al. [Ga̧s+24] constructed a BGT instance with which they proved that the approximation ratio of Reduce-Max is at least $\frac{9}{8}$.

The bounds known for Reduce-Fastest(x) are tighter than those demonstrated for Reduce-Max. When introduced [Ga̧s+17], it was shown that Reduce-Fastest(2) is a 4-approximation. Subsequently, Bilò et al. [Bil+22] proved that the the maximum height reached when $x = 1 + 1/\sqrt{5} \approx 1.45$, is $\frac{(3+\sqrt{5})}{2}G \approx 2.62G$. Kuzmaul [Kus22] recently showed that for any $x \geq 2$, Reduce-Fastest(x) achieves a precise bound of $(x + 1)G$ and also that Reduce-Fastest(1) cannot achieve a bound of $2G$.

This paper also introduced the Deadline-Driven algorithm and showed that it bounds the maximum height of plants to $2G$ when the lower threshold is set to G and the upper threshold to $2G$.

2.3 Polyamorous Scheduling

Our publication of [GSW24] originally introduced Polyamorous Scheduling, which we followed with [Bik+25]; as far as we know, this problem has not been addressed by earlier publications.

Decision Polyamorous Scheduling(DPS) contains Pinwheel Scheduling as a special case: the case where the underlying graph is a *star*, i.e., a centre connected to k pendant vertices with edges of frequencies $f_1 \leq f_2 \leq \dots \leq f_k$. This causes Decision Polyamorous Scheduling to inherit hardness results from Pinwheel Scheduling: there are representations in which it is NP-hard, but it is not known to be in NP. Note that it is not generally possible to obtain a polyamorous schedule by combining the local schedule of each person¹⁰; see for example a triangle with edge frequencies of 2: In the DPS instance $(\{A, B, C\}, \{A-B, B-C, A-C\}, f)$ with $f(e) = 2$ for all edges, the local problem for each person is feasible by alternating between their two partners, but the global DPS instance has no solution. This example also shows that the simple strategy of replacing f_i by $2^{\lceil \lg(f_i) \rceil}$ is not sufficient to solve DPS instances with low density, as it is for Pinwheel Scheduling.

As reviewed above, no Optimisation Polyamorous Scheduling(OPS) problems have previously been shown to have hardness of approximation results; we changed that by ruling out the existence of a PTAS (polynomial-time approximation scheme) for Optimisation Polyamorous Scheduling: Theorem 5.1.1. It remains open whether it is possible to obtain a PTAS for the Bamboo Garden Trimming Problem or the Optimal Windows Scheduling

¹⁰The current state-of-the-art approach in practice, usually via Google Calendar.

Problem. As with Pinwheel Scheduling and DPS, Bamboo Garden Trimming is the special case of OPS on star graphs.

By applying the logic of density to Bamboo Garden Trimming, we can form a fractional relaxation where each day is divided between the k tasks $g_1 \leq g_2 \leq \dots \leq g_k$; the optimal fractional schedule achieves a height of G by spending a fraction g_i/G of each day on task i . While this is sufficient to establish G as a lower bound on h_{\max} , many instances have a tighter bound from the density of the underlying PWS instances: if $\sum_{i=1}^k \lfloor \frac{G}{g_i} \rfloor > 1$, h_{\max} must be raised until the corresponding PWS instance has density $d \leq 1$. For example, consider $\mathcal{B} = (1, 2, 4)$ with $G = 7$: the PWS instance $\mathcal{P}(\mathcal{B}, G) = (1, 3, 7)$ has a density of $\frac{31}{21}$, while $\mathcal{P}(\mathcal{B}, 8) = (2, 4, 8)$ is trivially schedulable. Versions of these relaxations can be defined for OPS, and a similar but far richer fractional problem is introduced in Section 5.5.

2.4 Related Problems

In addition to introducing Bamboo Garden Trimming, Gąsieniec et al. [Gąs+17] introduced *Continuous Bamboo Garden Trimming*, a patrolling problem in which tasks are located on a labeled graph which the agent must traverse to access them. This problem contains BGT as a special case where the graph is complete and the edges are all labelled with the same travel time. Other patrolling problems also involve periodic schedules, such as some *Multi-Robot Patrolling Problems* [KS20; Afs+22], each of which tasks a group of robots with covering a given area, using various metrics to measure success. The *Point Patrolling Problem* [KS20] is particularly relevant and can be seen as a “covering version” of Pinwheel Scheduling; it has a single task which must be done daily by one of n workers, and requires that worker i rests for a_i days between each day of work. Another group of patrolling problems is *Replenishment Problems with Fixed Turnover Times* [Bos+22], where vertices in a graph must be visited with given frequencies, but instead of restricting the number of vertices that can be visited per day, we must minimize the length of a tour that visits all of them (starting at a depot node).

Cup Games are closely related to the Bamboo Garden Trimming problem, sharing the mechanics of growth and cutting, as well as similar applications [BFK19; Kus20]. Single processor cup games follow a similar daily routine in which an adversarial filler distributes 1 unit of water freely between a set of cups, after which an emptyer selects a single cup and removes up to 1 unit of water from it. Multi-processor versions are also of interest, relating to the multi-gardener version of Bamboo Garden Trimming [Kus22]. As with Bamboo Garden

Trimming, greedy algorithms perform well. *Cup Emptying Games* [BK21] are one step closer to Bamboo Garden Trimming; they retain the adversarial filler, but allow the emptyer to remove all of the water from a single cup, rather than at most 1 unit.

In the *Fair Hitting Sequence Problem* [Cic+19], we are given a collection of sets $\mathbf{S} = \{\mathcal{S}_1, \dots, \mathcal{S}_m\}$, each consisting of a subset of the set of elements $\mathcal{V} = \{v_1, \dots, v_n\}$. Each set \mathcal{S}_j has an urgency factor g_j , which is comparable to the growth rates in BGT instances but with one key difference: We say that a set \mathcal{S}_j is visited whenever any $v_i \in \mathcal{S}_j$ is scheduled. The goal is again similar to BGT: to find a perpetual schedule of elements $v_i \in \mathcal{V}$ that minimizes the time between visits to each set \mathcal{S}_j , weighted by g_j . There is also a decision variant, similar to Pinwheel Scheduling in that growth rates are replaced by frequencies. We use a similar layering technique in our approximation algorithm (Section 5.4) as the $O(\log^2 n)$ -approximation from [Cic+19], but we obtain a better approximation ratio for OPS. Their $O(\log n)$ -approximation based on randomized rounding does not extend to OPS since the linear program it uses would have exponentially many variables in this context (Section 5.5).

Optimal Job Scheduling problems are a family of problems broadly comparable to Periodic Scheduling problems. Graham et al. introduce a similar taxonomy in their 1979 paper [Gra+79], detailing a wide variety of intricately differentiated non-Periodic Scheduling problems. Optimal Job Scheduling problems have many direct applications, and are easy to tailor to both computational settings and real-world settings like manufacturing.

* * * * *

Chapter 3

Preliminaries

Outline In this chapter, we will formally define the four Periodic Scheduling problems that the rest of this part will focus on: Pinwheel Scheduling in Section 3.1.1, Bamboo Garden Trimming in Section 3.1.2, Decision Polyamorous Scheduling in Section 3.2.1, and Optimisation Polyamorous Scheduling in Section 3.2.2. Key ideas will also be introduced and defined: the state graph, dominance, and methods for converting between decision and optimisation versions of each problem.

3.1 Single Agent Problems

3.1.1 Pinwheel Scheduling

Introduced by [Hol+89], Pinwheel Scheduling examines a set of tasks which must be repeated indefinitely subject to a set of periodic deadlines:

Definition 3.1.1 Pinwheel Scheduling (PWS).

A Pinwheel Scheduling instance $\mathcal{P} = (f)$ consists of a set of k tasks $[k] = \{1, \dots, k\}$ with ascending positive integer frequencies $f = f_1, f_2, \dots, f_k : f_1 \leq f_2 \leq \dots \leq f_k$.

The goal of Pinwheel Scheduling is to either find or report the nonexistence of a valid Pinwheel Schedule \mathcal{S} , a finite sequence which can be repeated infinitely to map \mathbb{N}_0 to k . A Pinwheel Schedule is valid iff, for every task i and every time t , we have $i \in \{\mathcal{S}(t), \mathcal{S}(t+1), \dots, \mathcal{S}(t+f_i-1)\}$.

If a valid Pinwheel Schedule for \mathcal{P} exists, \mathcal{P} is called *feasible*; if no valid Pinwheel Schedule exists, it is called *infeasible*.

If a Pinwheel Scheduling instance $\mathcal{P} = (f_1 \leq f_2 \leq \dots \leq f_k)$ satisfies $f_k = f_{k-1} = \dots = f_{k-\ell+1} \neq f_{k-\ell}$, we call ℓ , the number of tasks of equal maximal frequency, the *symmetry* of \mathcal{P} .

States & the State Graph

Although the goal of periodic scheduling is to find sequences which can indefinitely satisfy a set of constraints, we define schedules as finite sequences; one reason for this is that periodic scheduling problems have finite *State Spaces*. Let $\mathcal{P} = (f_1 \leq f_2 \leq \dots \leq f_k)$ be a Pinwheel Scheduling instance with valid infinite schedule \mathcal{S}_∞ . For any day t in \mathcal{S}_∞ , the *state* $\mathcal{X} = \mathcal{X}(\mathcal{S}_\infty, t)$ of a Pinwheel Scheduling instance is a vector $\mathcal{X} = (x_1, \dots, x_k)$, where x_i is the number of days since the last occurrence of i in the schedule ($x_i = t - \max(0, t' \leq t : \mathcal{S}(t') = i)$). Note that since \mathcal{S}_∞ is valid, all states it reaches must also be *valid* ($0 \leq x_i < f_i$, for $i \in [k]$). This condition also implies that there are only finitely many valid states, and therefore that some subsequence of \mathcal{S}_∞ must be a circuit that starts and ends at the same state; a finite schedule \mathcal{S} .

This notion of states allows us to cast Pinwheel Scheduling to a graph problem. Define the *state graph* $\mathbf{G}_\mathcal{P} = (\mathbf{V}_\mathcal{P}, \mathbf{E}_\mathcal{P})$ as a directed graph with all possible states as vertices, i.e., $\mathbf{V}_\mathcal{P} = \{(x_1, \dots, x_k) : \forall i \in [k], 0 \leq x_i < f_i\}$. Edges $(\mathcal{X}, \mathcal{X}') \in \mathbf{E}_\mathcal{P}$ are either:

- (a) (*task edges*) $\exists j \in [k] : (x'_j = 0 \wedge \forall i \in [k] \setminus \{j\} : x'_i = x_i + 1)$, or
- (b) (*gap edges*) $\forall i \in [k] : x'_i = x_i + 1$.

\mathcal{P} is then feasible if and only if $\mathbf{G}_\mathcal{P}$ contains an infinite walk starting at $\mathcal{X}_0 = (0, \dots, 0)$. Since $\mathbf{G}_\mathcal{P}$ is finite and any valid schedule must be valid from \mathcal{X}_0 , \mathcal{P} is feasible if and only if $\mathbf{G}_\mathcal{P}$ contains a directed cycle¹. We call a state *sustainable* if it can be revisited infinitely often by some valid schedule (i.e., when it is part of a directed cycle in $\mathbf{G}_\mathcal{P}$).

It follows from the state-graph representation that if \mathcal{P} is feasible, it is so by a *periodic* schedule, i.e., there is a schedule $\mathcal{S} = s_1, s_2, \dots$ and an integer p , so that for all t we have $s_t = s_{t+p}$. Unless explicitly mentioned in the following we assume schedules to be periodic and represent them by the finite periodic part $\mathcal{S} = s_1 \dots s_p$. Since p corresponds to the length of a cycle in $\mathbf{G}_\mathcal{P}$, we can always find \mathcal{S} with $p = |\mathcal{S}| \leq |\mathbf{V}_\mathcal{P}| = \prod_{i=1}^k f_i$, so long as \mathcal{P} is feasible [Hol+89].

In Section 4.2.3, conventional states $\mathcal{X}(\mathcal{S}, t)$ are transformed into urgency states $\mathcal{U}(\mathcal{S}, t) = (u_1, \dots, u_k)$ using the following equation:

¹See Lemma 4.2.1

$$\forall i, t : u_i(t) = f_i - x_i(t) - 1 \quad (3.1)$$

Task urgency u_i counts down if unattended and is reset to $f_i - 1$ when a task is executed. This is a relabeling containing exactly the same information; the same state graph connects urgency states and regular states, but urgency states do a better job of capturing the ticking clock inherent to Pinwheel Scheduling. Further features of urgency states are introduced and exploited in Section 4.2.3.

Holidays, Loose Feasibility, & Dominance

While we have defined Pinwheel Scheduling in the typical way, with schedules assigning some task to each day in the finite repeat unit of an infinite period, we will sometimes allow an additional possibility - *holidays*, gaps in a schedule on which no task is performed. Such days are represented by a special symbol: “-”.

We call a feasible Pinwheel Scheduling instance \mathcal{P} *loosely feasible* if it admits a periodic schedule with a gap (i.e., when $\mathbf{G}_{\mathcal{P}}$ has a cycle containing a gap edge); otherwise \mathcal{P} is *tight* or *tightly feasible*. For example, $(2, 4)$ is loosely feasible by $(1, 2, 1, -)$, whereas $(2, 3)$ is tightly feasible despite having density $d = 5/6 < 1$. Recall that $(2, 3, *)$ is not feasible for any value of $*$; this extends to all tightly feasible instances – none allow for the addition of an extra task.

Clearly, any holidays in a valid schedule could either be removed or filled with an arbitrary task without affecting the validity of the schedule, but the distinction between tight and loose feasibility will prove useful in Chapter 4.

Proposition 3.1.2.

Given a Pinwheel Scheduling instance $\mathcal{P} = (f_1 \leq f_2 \leq \dots \leq f_k)$, it can be decided whether \mathcal{P} is infeasible, tightly feasible or loosely feasible using $O(k \prod_{i=1}^k f_i)$ time and space. Moreover, if it exists, a corresponding schedule can be computed with the same complexity and has length at most $\prod_{i=1}^k f_i$.

Proof. We construct the state graph $\mathbf{G}_{\mathcal{P}}$ and compute the strongly connected components of $\mathbf{G}_{\mathcal{P}}$. Note that $\mathbf{G}_{\mathcal{P}}$ contains a directed cycle iff there is a strong component containing at least two vertices; moreover, $\mathbf{G}_{\mathcal{P}}$ contains a directed cycle containing a gap edge iff there is a gap edge with both endpoints in the same strong component. Both the computation of strong components and the testing of these two conditions can be done in linear time with respect to the size of the state graph $\mathbf{G}_{\mathcal{P}}$. We have $|\mathbf{V}_{\mathcal{P}}| = \prod_{i=1}^k f_i$ vertices in $\mathbf{G}_{\mathcal{P}}$. Since each

vertex in $\mathbf{G}_{\mathcal{P}}$ has at most $k + 1$ outgoing edges, a schedule can be found using a depth-first search. \square

The algorithm sketched in this proof is mostly of theoretical interest due to its prohibitive space cost; we present several alternatives in Section 4.2.

Given two Pinwheel Scheduling instances with k tasks each, $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ and $B = (b_1 \leq b_2 \leq \dots \leq b_k)$, we say that A *dominates* B , written $A \leq B$, if $\forall i \in [k] : a_i \leq b_i$. Obviously, any schedule that is valid for A is also valid for B . Moreover, $A \leq B$ implies that density $d(A) \geq d(B)$.

These ideas are used mainly in Section 4.1, where they allow us to construct *Pareto Surfaces*: classes of solved Pinwheel Scheduling problems that collectively solve much larger or even infinite classes of Pinwheel Scheduling problems.

3.1.2 Bamboo Garden Trimming

Bamboo Garden Trimming is the natural optimisation version of Pinwheel Scheduling, introduced in [Gas+17]. Much of this thesis either supports future study of Bamboo Garden Trimming (Chapter 4), extends Bamboo Garden Trimming (Chapter 5), or exploits past work studying Bamboo Garden Trimming (Chapter 6).

Definition 3.1.3 Bamboo Garden Trimming (BGT).

A *Bamboo Garden Trimming instance* $\mathcal{B} = (g)$ consists of a set of k tasks $[k] = \{1, \dots, k\}$ with descending positive growth rates $g = g_1, g_2, \dots, g_k : g_1 \geq g_2 \geq \dots \geq g_k$.

On each day $t \in \mathbb{N}_0$ of a Bamboo Schedule $\mathcal{S} : \mathbb{N}_0 \rightarrow k$, each task $i \in [k]$ has height

$$h_i(\mathcal{S}, t) = \begin{cases} g_i \cdot (t - t' + 1) & \exists t' < t \in \mathbb{N}_0 : \mathcal{S}(t') = i, i \notin \{\mathcal{S}(t' + 1), \mathcal{S}(t' + 2), \dots, \mathcal{S}(t)\}; \\ g_i \cdot (t + 1) & \text{otherwise.} \end{cases}$$

The goal of *Bamboo Garden Trimming* is to find a *Bamboo Schedule* which minimises the maximum height $h_{\max} = \max_{i \in [k], t \in \mathbb{N}_0} h_i(\mathcal{S}, t)$.

Note the similarity between the height vector $\mathcal{H}(\mathcal{S}, t) = (h_1, \dots, h_k)$ and the state vector of Pinwheel Scheduling $\mathcal{X}(\mathcal{S}, t) = (x_1, \dots, x_k)$. In general, \mathcal{H} is unbounded, but if we impose some maximum height h_{\max} we arrive at a PWS instance with $f_i = \lfloor h_{\max}/g_i \rfloor$ and $x_i = h_i/g_i - 1$.

3.2 Multi-Agent Problems

3.2.1 Decision Polyamorous Scheduling

We begin by defining a decision version of Polyamorous Scheduling: In the *Decision Polyamorous Scheduling* problem, we are given a set of people and pairwise relationships with “attendance frequencies” $f_{i,j}$, and we are trying to find a daily schedule of two-person meetings such that each couple $\{i, j\}$ meets at least every $f_{i,j}$ days. The only constraint on the number of meetings that can occur on any given day is that each person can only participate in at most one of them. A Decision Polyamorous Scheduling instance can naturally be modelled as a graph of people with the edges representing their relationships. Because each person can participate in at most one meeting per day, the edges scheduled on any given day must form a matching in this graph.

Definition 3.2.1 Decision Polyamorous Scheduling (DPS).

A *Decision Polyamorous Scheduling instance* $\mathcal{D} = (P, R, f)$ (a “decision polycule”) consists of an undirected graph (P, R) where the vertices $P = \{p_1, \dots, p_n\}$ are n persons and the edges R are pairwise relationships between them, with integer frequencies $f : R \rightarrow \mathbb{N}$ for each relationship $e \in R$.

The goal of *Decision Polyamorous Scheduling* is to either find or report the nonexistence of a valid schedule \mathcal{S} , a finite sequence which maps \mathbb{N}_0 to 2^R . A schedule is valid if:

- (a) (no conflicts) for all days $t \in \mathbb{N}_0$, $\mathcal{S}(t)$ is a matching in (P, R) , and
- (b) (frequencies) for all $e \in R$ and $t \in \mathbb{N}_0$, we have $e \in \mathcal{S}(t) \cup \mathcal{S}(t+1) \cup \dots \cup \mathcal{S}(t+f(e)-1)$.

As with PWS, \mathcal{D} is *feasible* if it has a valid schedule, and is *infeasible* if it does not.

We write $f_{i,j}$ and f_e as shorthands for $f(\{p_i, p_j\})$ and $f(e)$. An infinite schedule exists if and only if a *periodic* schedule exists, i.e., a schedule where there is a period $T \in \mathbb{N}$ such that for all t , we have $\mathcal{S}(t) = \mathcal{S}(t+T)$: any feasible schedule corresponds to an infinite walk in the finite configuration graph of the problem (see Section 5.2), implying the existence of a finite cycle. A periodic schedule can be finitely described by listing $\mathcal{S}(0), \mathcal{S}(1), \dots, \mathcal{S}(T-1)$.

3.2.2 Optimisation Polyamorous Scheduling

By relaxing the hard maximum frequencies of meetings between couples to “desire growth rates”, we obtain the *Optimisation Polyamorous Scheduling* problem. Our objective is to

find a schedule that minimizes the “heat”, i.e., the *worst pain of separation* ever felt by any couple in the polycule.

Definition 3.2.2 Optimisation Polyamorous Scheduling (OPS).

An *Optimisation Polyamorous Scheduling instance* (or “*optimisation polycule*”) $\mathcal{O} = (P, R, g)$ consists of an undirected graph (P, R) along with a desire growth rate $g : R \rightarrow \mathbb{R}_{>0}$ for each relationship $e \in R$. An infinite schedule $\mathcal{S} : \mathbb{N}_0 \rightarrow 2^R$ is valid if, for all days, $t \in \mathbb{N}_0$, $\mathcal{S}(t)$ is a matching in \mathcal{O} .

On each day $t \in \mathbb{N}_0$ in a schedule \mathcal{S} , the heat $h_e(\mathcal{S}, t)$ of each relationship $e \in R$ is:

$$h_e(\mathcal{S}, t) = \begin{cases} g_e \cdot (t - t' + 1) & \exists t' \in \mathbb{N}_0 : e \in \mathcal{S}(t'), e \notin \mathcal{S}(t' + 1) \cup \mathcal{S}(t' + 2) \cup \dots \cup \mathcal{S}(t); \\ g_e \cdot (t + 1) & \text{otherwise.} \end{cases}$$

The goal of *Optimisation Polyamorous Scheduling* is to find a valid schedule that minimises the maximum heat $h_{\max} = \max_{e \in R, t \in \mathbb{N}_0} h_e(\mathcal{S}, t)$.

As for DPS, \mathcal{S} can be assumed to be periodic without loss of generality, meaning that \mathcal{S} is finitely representable.

It will be convenient for much of the following to use *Normalised OPS* instances, which we define in the following way: Considering an OPS polycule $\mathcal{O} = (P, R, g)$, define the personal growth rate G_v for person $v \in P$ as $G_v := \sum_{e \in R: v \in e} g(e)$ and the maximum personal growth rate across all $v \in P$ as $G^* := \max_{v \in P} G_v$. \mathcal{O} is normalised, or in normal form, if $G^* = 1$. Any OPS polycule \mathcal{O} can be trivially normalised by dividing each growth rate $g_e \in g$ by G^* , and restored by inverting this.

3.2.3 Additional Preliminaries

In this section, we introduce some general notation (mostly concerning graph theory) and collect a few simple facts about Polyamorous Scheduling which will be used later.

We write $[n..m]$ for $\{n, n+1, \dots, m\}$ and $[n]$ for $[1..n]$. For a set A , we denote its powerset by 2^A . We also use \lg for \log_2 universally. All graphs $\mathcal{G} = (V, E)$ are simple and undirected unless indicated otherwise. We represent the set of vertices in the neighbourhood of $v \in V$ by $\mathcal{N}(v)$. We denote by $\mathbf{M} = \mathbf{M}(\mathcal{G})$ the set of *inclusion-maximal matchings* in graph \mathcal{G} , where matching has the usual meaning of an edge set with no two edges incident to the same vertex. By $\Delta = \Delta(\mathcal{G})$, we denote the *maximum degree* in $\mathcal{G} = (V, E)$. A *pendant vertex* is a vertex with degree 1. The *chromatic index* $\chi_1 = \chi_1(\mathcal{G})$ is the smallest number of “colours”

in a proper edge colouring of $\mathcal{G} = (V, E)$ (i.e., the number of disjoint matchings required to cover E); by Vizing's Theorem [Viz65], we have $\Delta \leq \chi_1 \leq \Delta + 1$ for every graph. Misra and Gries provide a polynomial-time algorithm for edge colouring any graph using at most $\Delta + 1$ colours [MG92].

Given a schedule $\mathcal{S} : \mathbb{N}_0 \rightarrow 2^R$ and an edge $e \in R$, we define the (*maximal*) *recurrence time* $r(e) = r_{\mathcal{S}}(e)$ of e in \mathcal{S} as the maximal time between consecutive occurrences of e in \mathcal{S} , formally:

$$r_{\mathcal{S}}(e) = \sup_{d \in \mathbb{N}} \begin{cases} d + 1 & \exists t \in \mathbb{N}_0 : e \notin \mathcal{S}(t) \cup \mathcal{S}(t+1) \cup \dots \cup \mathcal{S}(t+d-1); \\ \infty & \text{otherwise,} \end{cases}$$

where t here is the first day when e was not scheduled and d is the number of days on which e was not scheduled.

If an infinite schedule \mathcal{S} contains $\mathcal{O}(1)$ instances of e , then $r_{\mathcal{S}}(e) = \infty$, but all schedules in the following will have finite recurrence times for all edges. Note that recurrence times can be used to find the hardest DPS or PWS instance solved by a valid schedule for either problem.

Using recurrence time, the heat $h = h(\mathcal{S})$ of a schedule \mathcal{S} in an OPS instance (P, R, g) is $h(\mathcal{S}) = \max_{e \in R} g(e) \cdot r(e)$. Clearly, for any schedule $\mathcal{S} : \mathbb{N}_0 \rightarrow 2^R$, we can obtain $\mathcal{S}' : \mathbb{N}_0 \rightarrow \mathbf{M}$ by adding edges to $\mathcal{S}(t)$ until we have a maximal matching $\mathcal{S}'(t) \supseteq \mathcal{S}(t)$; then $r_{\mathcal{S}'}(e) \leq r_{\mathcal{S}}(e)$ for all $e \in R$ and hence \mathcal{S}' is a valid schedule for any DPS instance for which \mathcal{S} is valid, and if \mathcal{S} schedules an OPS instance with heat $h(\mathcal{S})$ then \mathcal{S}' does too, with $h(\mathcal{S}') \leq h(\mathcal{S})$.

Manipulating Polycules We use Lemma 3.2.3 to reduce OPS to DPS, and Lemma 3.2.4 to formalize how DPS solves OPS:

Lemma 3.2.3 OPS to DPS.

For every combination of OPS instance $\mathcal{O} = (P, R, g)$ and heat value h , there exists a DPS instance $\mathcal{D} = (P, R, f)$ such that

- (a)** *any feasible schedule $\mathcal{S} : \mathbb{N}_0 \rightarrow 2^R$ for \mathcal{D} is a schedule for \mathcal{O} with heat $\leq h$, and*
- (b)** *any schedule \mathcal{S}' for \mathcal{O} with heat $h' > h$ is not feasible for \mathcal{D} .*

Proof. Consider an OPS polycule $\mathcal{O} = (P, R, g)$; we set $\mathcal{D} = (P, R, f)$ where $f(e) = \lfloor \frac{h}{g(e)} \rfloor$ for all $e \in R$. Schedules satisfying \mathcal{D} when applied to \mathcal{O} will allow heat of at most $\max_{e \in R} g(e) \cdot f(e) = \max_{e \in R} g(e) \lfloor \frac{h}{g(e)} \rfloor \leq h$.

Now consider a schedule \mathcal{S}' for \mathcal{O} with heat $h' > h$. By definition, $h' = \max_{e \in R} r_{\mathcal{S}'}(e) \cdot g(e)$, where $r(e) = r_{\mathcal{S}'}(e)$ is the recurrence time of e in \mathcal{S}' . Assume towards a contradiction that $r(e) \leq f(e)$ for all $e \in R$. This implies that $h' = \max_{e \in R} r(e) \cdot g(e) \leq \max_{e \in R} \lfloor \frac{h}{g(e)} \rfloor \cdot g(e) \leq h$, a contradiction to the assumption. \square

Lemma 3.2.4 DPS to OPS.

Let $\mathcal{D} = (P, R, f)$ be a DPS instance. Set $F = \max_{e \in R} f(e)$. There is an OPS instance $\mathcal{O} = (P, R, g)$ such that the following holds:

(a) If \mathcal{D} is feasible, then \mathcal{O} admits a schedule of height $h \leq 1$.

(b) If \mathcal{D} is infeasible, then the optimal heat h^* of \mathcal{O} satisfies $h^* \geq \frac{F+1}{F}$.

Proof. Consider a DPS instance $\mathcal{D} = (P, R, f)$; we set $\mathcal{O} = (P, R, g)$ with $g(e) = 1/f(e)$ for $e \in R$. By definition, any feasible schedule \mathcal{S} for \mathcal{D} has recurrence time $r_e = r_{\mathcal{S}}(e) \leq f(e)$ for all $e \in R$, so its heat in \mathcal{O} is given by $h(\mathcal{S}) = \max_{e \in R} r(e) \cdot g(e) = \max_{e \in R} \frac{r(e)}{f(e)} \leq 1$. Conversely, if \mathcal{D} is infeasible, then for every $\mathcal{S} : \mathbb{N}_0 \rightarrow 2^R$ there exists an edge $e' \in R$ where $r(e') > f(e')$, i.e., $r(e') \geq f(e') + 1$. In \mathcal{O} , the heat $h(\mathcal{S})$ must then be $h(\mathcal{S}) = \max_{e \in R} r(e) \cdot g(e) \geq r(e') \cdot g(e') \geq \frac{f(e')+1}{f(e')} \geq \frac{F+1}{F}$. \square

It is also often convenient to rescale OPS instances to set $h^* = 1$; this can be assumed without loss of generality but is not generally useful for algorithms unless h^* is known.

Lemma 3.2.5 Unit Optimal Heat for OPS.

For every OPS instance $\mathcal{O} = (P, R, g)$, there is an equivalent OPS instance $\mathcal{O}' = (P, R, g')$ with optimal heat 1 where $g' : R \rightarrow \mathbf{U}$ for $\mathbf{U} = \{1/m : m \in \mathbb{N}_{\geq 1}\}$, i.e., the set of unit fractions. More precisely, for every schedule $\mathcal{S} : \mathbb{N}_0 \rightarrow 2^R$ holds: \mathcal{S} has optimal heat h^* in \mathcal{O} iff \mathcal{S} has heat 1 in \mathcal{O}' . That is, any optimal schedule \mathcal{S}^* for either problem is also optimal for the other problem.

Proof. Let (P, R, g) be an arbitrary OPS instance with optimal heat h^* . Setting $\hat{g}(e) = g(e)/h^*$ yields OPS instance (P, R, \hat{g}) with optimal heat 1. We now start by setting $g'(e) = \hat{g}(e)$ for all $e \in R$. Consider a particular optimal schedule \mathcal{S}^* . Suppose that for some edge $e \in R$, we have $g'(e) \notin \mathbf{U}$. In \mathcal{S} , there is a maximal separation $r(e) = q \in \mathbb{N}$ between consecutive occurrences of e with $q \cdot g(e) \leq h^*$. But then, increasing $g(e)$ to h^*/q would not affect the heat of \mathcal{S} . We can thus set $g'(e) = 1/q$. By induction, we thus obtain $g' : R \rightarrow \mathbf{U}$ without affecting the heat of \mathcal{S} . \square

* * * * *

Chapter 4

Towards the 5/6-Density Conjecture of Pinwheel Scheduling

Outline This chapter presents results originating in our 2022 paper of the same name [GSW22]. It begins with our main theoretical results in Section 4.1, after which Section 4.2 introduces our algorithms which determine the feasibility of and find schedules for single Pinwheel Scheduling instances; Section 4.3 then describes our algorithms for computing the Pareto surface. In Section 4.4, we report on a running-time study, comparing our algorithms and analyzing their efficiency. We conclude in Section 4.5 with a brief summary.

4.1 The Pareto Surface

We begin by deriving our main new structural tool for analyzing Pinwheel Scheduling: Pareto Surfaces. A special case of such a Pareto Surface is implicitly used in [Din20], but without developing its general applicability as we will now do.

To this end, we need some more vocabulary. It is often helpful to reduce Pinwheel schedules to their *recurrence vectors*: the hardest Pinwheel Scheduling instance solved by the schedule, i.e. the one which minimises f_i for all i . To this end, we call a tuple consisting of a schedule and its recurrence vector a *scheduled Pinwheel Scheduling problem*. Let \mathbf{P}_a be a (finite or infinite) set of Pinwheel Scheduling instances. We say that a (finite or infinite) set of Pinwheel Schedules \mathbf{S}_a *solves* \mathbf{P}_a if, for every problem $\mathcal{P} \in \mathbf{P}_a$, there is some $\mathcal{S} \in \mathbf{S}_a$ such that \mathcal{S} is a valid schedule for \mathcal{P} . This is equivalent to saying that every problem in \mathbf{P}_a is dominated by the recurrence vector of some problem in \mathbf{S}_a .

A *Pareto Surface* $\mathbf{C}_a = \mathbf{C}(\mathbf{P}_a)$ for a set of Pinwheel Scheduling instances \mathbf{P}_a is an inclusion-minimal set of scheduled Pinwheel Scheduling problems $\mathcal{C} = (\mathcal{P}, \mathcal{S})$ that solves \mathbf{P}_a , i.e., for every feasible $\mathcal{P}_i \in \mathbf{P}_a$ there is some $\mathcal{C} \in \mathbf{C}_a$ with $\mathcal{P}(\mathcal{C}) \leq \mathcal{P}_i$. We use *inclusion-minimal* to mean that no member of a set can be removed from that set without violating its defining property, i.e., for every \mathcal{C} in \mathbf{C}_a there must be some \mathcal{P} in \mathbf{P}_a that is solved by \mathcal{C} but not by any other member of \mathbf{C}_a . Note that while we will only consider finite \mathbf{C}_a surfaces, \mathbf{P}_a does not need to be finite, per Theorem 4.1.1.

The Pareto Surfaces of two families of sets of Pinwheel Scheduling instances are of particular interest: \mathbf{P}_k , by which we denote the class of all Pinwheel Scheduling instances with k tasks, and $\mathbf{P}_{k,d}$, by which we denote the Pinwheel Scheduling instances with k tasks and density at most d . For Pareto Surfaces solving \mathbf{P}_k for $1 \leq k \leq 5$, see Table 4.1.

The main result of this section is the following theorem:

Theorem 4.1.1 Finite Pareto Surfaces.

For every $k \in \mathbb{N}$, there is a finite set of periodic schedules such that every Pinwheel Scheduling instance with k tasks has a solution if and only if it has a solution in that set. Moreover, there is a unique inclusion-minimal such set \mathbf{S}_k .

Before proceeding with the proof of this theorem in the following two subsections, let us note a complexity-theoretic consequence of this result:

Corollary 4.1.2 Pinwheel is FPT.

Pinwheel Scheduling is fixed-parameter tractable with respect to the number of tasks k .

Proof. We give an algorithm deciding any instance $\mathcal{P} \in \mathbf{P}_k$ in time $\mathcal{O}(N + f(k))$ where N is the encoding length of \mathcal{P} and f is some computable function; this implies the claim.

By Theorem 4.1.1, all of \mathbf{P}_k is solved by \mathbf{S}_k . Let $m(k)$ be the maximum over all distances between consecutive occurrences of all tasks in any solution $\mathcal{S} \in \mathbf{S}_k$. We first compute \mathbf{S}_k and $m(k)$; as \mathbf{S}_k only depends on k , the cost to do so is bounded by some function $g(k)$. Read the input $\mathcal{P} = (f_1 \leq f_2 \leq \dots \leq f_k)$ for cost $\mathcal{O}(N)$, then replace any frequency $f_i > m(k)$ with $m(k)$, producing a new Pinwheel Scheduling instance $\mathcal{P}' = \kappa(\mathcal{P})$ of (encoding) size $N' = \mathcal{O}(k \lg m(k))$. Now \mathcal{P}' is feasible iff \mathcal{P} is feasible, because any $\mathcal{S} \in \mathbf{S}_k$ solves \mathcal{P}' iff it solves \mathcal{P} . Comparing \mathcal{P}' with all $\mathcal{S} \in \mathbf{S}_k$ for some cost $h(k)$ determines whether there exists a schedule to \mathcal{P} , so the schedulability of any input \mathcal{P} can be determined for a cost $\mathcal{O}(g(k) + N + h(k))$. \square

Instance	Schedule
(1)	(1)
(2,2)	(1,2)
(2, 4, 4)	(1, 2, 1, 3)
(3, 3, 3)	(1, 2, 3)
(2, 4, 8, 8)	(1, 2, 1, 3, 1, 2, 1, 4)
(2, 6, 6, 6)	(1, 2, 1, 3, 1, 4)
(3, 3, 6, 6)	(1, 2, 3, 1, 2, 4)
(3, 4, 5, 8)	(1, 2, 4, 1, 3, 2, 1, 3)
(3, 5, 5, 5)	(1, 2, 3, 1, 4, 2, 1, 3, 4)
(4, 4, 4, 4)	(1, 2, 3, 4)
(2, 4, 8, 16, 16)	(1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5)
(2, 4, 12, 12, 12)	(1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 5)
(2, 6, 6, 12, 12)	(1, 2, 1, 3, 1, 4, 1, 2, 1, 3, 1, 5)
(2, 6, 8, 10, 16)	(1, 2, 1, 3, 1, 5, 1, 2, 1, 4, 1, 3, 1, 2, 1, 4)
(2, 6, 10, 10, 10)	(1, 2, 1, 3, 1, 4, 1, 2, 1, 5, 1, 3, 1, 2, 1, 4, 1, 5)
(2, 8, 8, 8, 8)	(1, 2, 1, 3, 1, 4, 1, 5)
(3, 3, 6, 12, 12)	(1, 2, 3, 1, 2, 4, 1, 2, 3, 1, 2, 5)
(3, 3, 9, 9, 9)	(1, 2, 3, 1, 2, 4, 1, 2, 5)
(3, 4, 5, 14, 14)	(1, 2, 3, 1, 4, 2, 1, 3, 1, 2, 5, 1, 3, 2)
(3, 4, 6, 10, 16)	(1, 2, 3, 1, 4, 2, 1, 3, 1, 2, 4, 1, 3, 2, 1, 5)
(3, 4, 6, 11, 11)	(1, 2, 3, 1, 5, 2, 1, 3, 2, 1, 4)
(3, 4, 8, 8, 8)	(1, 2, 4, 1, 5, 2, 1, 3)
(3, 5, 5, 9, 9)	(1, 2, 5, 1, 3, 2, 1, 4, 3)
(3, 5, 6, 7, 12)	(1, 2, 4, 1, 3, 2, 1, 4, 2, 1, 3, 5)
(3, 5, 7, 7, 9)	(1, 2, 3, 1, 4, 2, 1, 5, 3, 1, 2, 4, 1, 3, 2, 1, 5, 4)
(3, 5, 7, 8, 8)	(1, 2, 3, 1, 4, 2, 1, 5, 1, 3, 2, 1, 4, 5)
(3, 6, 6, 6, 6)	(1, 2, 3, 1, 4, 5)
(4, 4, 4, 8, 8)	(1, 2, 3, 4, 1, 2, 3, 5)
(4, 4, 5, 7, 12)	(1, 2, 3, 4, 1, 2, 5, 3, 1, 2, 4, 3)
(4, 4, 6, 6, 6)	(1, 3, 2, 4, 1, 5, 2, 3, 1, 4, 2, 5)
(4, 5, 5, 6, 10)	(1, 2, 3, 5, 1, 4, 2, 3, 1, 4)
(4, 5, 5, 7, 7)	(1, 2, 5, 3, 1, 4, 2, 1, 3, 5, 2, 1, 4, 3)
(5, 5, 5, 5, 5)	(1, 2, 3, 4, 5)

Table 4.1: The Pareto Surfaces C_1 , C_2 , C_3 , C_4 , and C_5 .

Note that the *input size* N of a Pinwheel Scheduling instance can be substantially larger than k since it has to encode the frequencies (say, in binary); frequencies of 2^{k-1} are needed to form \mathbf{C}_{1-5} in Table 4.1 as well as the density-limited $\mathbf{C}_{k,d}$ surfaces generated for other parts of this work, and it remains open whether even larger frequencies are also needed for some \mathbf{C}_k . In general, N has not been shown to be bounded in terms of k ; though we conjecture that it can be.

4.1.1 Small Frequency Conjecture

We propose two new conjectures about the structure of feasible Pinwheel Scheduling instances. These arose from observations made while engineering our algorithms, but are of independent interest. We list evidence in their support in Section 4.3.2, with further arguments in Section 8.1.2.

Conjecture 4.1.3 2^k Conjecture.

Let $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ be a loosely feasible Pinwheel Scheduling instance. Then A admits a schedule \mathcal{S} with a holiday at least every 2^k days.

Conjecture 4.1.4 Kernel Conjecture.

Let $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ be a feasible Pinwheel Scheduling instance. Then there exists another Pinwheel Scheduling instance $B = (b_1 \leq b_2 \leq \dots \leq b_k)$ such that:

- (a) B is also feasible,
- (b) B dominates A , $B \leq A$, and
- (c) $b_k \leq 2^{k-1}$.

We show that these two conjectures are indeed *equivalent*.

Proposition 4.1.5 Equivalent conjectures.

Conjecture 4.1.3 and Conjecture 4.1.4 are equivalent.

Proof. First, assume that Conjecture 4.1.3 holds true. Let $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ be an arbitrary instance with schedule \mathcal{S}_A . If $a_k \leq 2^{k-1}$, we can satisfy Conjecture 4.1.4 by setting $B = A$; instead, assume that the last $\ell \geq 1$ frequencies obey $2^{k-1} < a_{k-\ell+1} \leq a_{k+1} \leq \dots \leq a_k$. Define C to exclude these frequencies: $C = (a_1 \leq a_2 \leq \dots \leq a_{k-\ell})$. C is loosely feasible since we can replace all occurrences of $i > k - \ell$ in \mathcal{S}_A by “–”. By Conjecture 4.1.3, C then

admits a gapped schedule \mathcal{S}_C with at least one gap every $2^{k-\ell}$ days. We claim that we can now set $B = (a_1 \leq \dots \leq a_{k-\ell}, 2^{k-1}, \dots, 2^{k-1})$, i.e., truncate all frequencies exceeding 2^{k-1} at 2^{k-1} , and remain feasible. We make a schedule \mathcal{S}_B for B by using ℓ repetitions of \mathcal{S}_C and assigning the ℓ tasks $k - \ell + 1, \dots, k$ in a Round-Robin fashion to the gaps. This achieves frequency at most $\ell \cdot 2^{k-\ell}$, for each of these tasks. We check indeed $\ell \cdot 2^{k-\ell} \leq 2^{k-1}$, for all k and $\ell \in \mathbb{N}$.

Now, conversely, assume that Conjecture 4.1.4 holds true. Let $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ be loosely feasible, and let \mathcal{S}_A be a gapped schedule for A , setting g to the frequency of the gap in \mathcal{S}_A . Define $B = (b_1 \leq b_2 \leq \dots \leq b_{k+1})$ with $b_i = a_i$ for $i \in [k]$ and $b_{k+1} = \max\{g, a_k\}$. B is feasible since we can replace the gap in \mathcal{S}_A by $k + 1$ and obtain a schedule for B . Hence by Conjecture 4.1.4, there is a feasible instance $C = (c_1 \leq c_2 \leq \dots \leq c_{k+1})$ with $c_i \leq b_i$ and $c_i \leq 2^k$. Let \mathcal{S}_C be a schedule for C . By replacing $k + 1$ in \mathcal{S}_C by “–”, we obtain a valid gapped schedule for A and since $c_{k+1} \leq 2^k$, the gap frequency in this schedule is at most 2^k . \square

In light of this, the evidence in support of Conjecture 4.1.4 that we provide in Section 4.3.2 equally supports Conjecture 4.1.3.

Remark 4.1.6 Kernel size.

The construction used to prove Corollary 4.1.2 shows that Pinwheel Scheduling has an FPT-kernel of size $\mathcal{O}(k \lg m(k))$; assuming Conjecture 4.1.4, this reduces to $\mathcal{O}(k^2)$.

4.1.2 The Pareto Trie

Towards the proof of the first claim of Theorem 4.1.1, we describe an algorithm to compute the Pareto Surface \mathbf{C}_k for a given k , based on an oracle for deciding whether a given Pinwheel Scheduling instance is infeasible, tightly feasible or loosely feasible (cf. Proposition 3.1.2). We describe our implementation of such an oracle in Section 4.2.

This algorithm conceptually explores an (infinite) trie \mathbf{T}_k for \mathbf{P}_k , each node of which is a Pinwheel Scheduling instance that is a prefix of all of its descendants. The root of \mathbf{T}_k is the empty instance $\mathcal{P}_0 = ()$, which has no tasks but infinitely many children, reached through the edges labelled $1, 2, 3, \dots$. In general, every node v at depths between 0 and k has infinitely many children; if v is reached from its parent by an edge labeled a , v has children $a, a + 1, a + 2, \dots$. We identify a node v in the trie with the sequence of edge labels on the path from the root to v . In this way, each node v at depth ℓ corresponds to a Pinwheel Scheduling instance with ℓ tasks.

Our algorithm explores \mathbf{T}_k using a depth-first search. Since \mathbf{T}_k has depth k , we can only descend in the tree k times; however, we have to show that we only need to explore finitely many children of any node. Suppose we are currently visiting a node v at depth $\ell < k$, corresponding to a Pinwheel Scheduling instance $A = (a_1 \leq a_2 \leq \dots \leq a_\ell)$. If A is infeasible or tightly feasible, all of v 's descendants are infeasible; in particular, none of the descendants at depth k – corresponding to extensions of A to instances in \mathbf{P}_k – will be feasible. We, therefore, need not visit any of them.

If A is loosely feasible, some descendant at depth k is guaranteed to be feasible: Let \mathcal{S}_A be a gapped schedule for A with some gap frequency g . Then $B = (a_1 \leq a_2 \leq \dots \leq a_\ell \leq (k-\ell)g, \dots, (k-\ell)g)$ – i.e., A with $k-\ell$ copies of $(k-\ell)g$ appended – is solved by a schedule \mathcal{S}_B consisting of $k-\ell$ repetitions of \mathcal{S}_A whose gaps are replaced by $\ell+1, \dots, k$, respectively. Hence, we need not visit any child of v with label larger than $(k-\ell)g$ (they are all feasible and dominated by B), and we therefore only have to visit finitely many children of v . For later reference, we call the smallest frequency f so that $(a_1 \leq a_2 \leq \dots \leq a_\ell < f, \dots, f) \in \mathbf{P}_k$ is feasible the *Round Robin frequency* of A with respect to k . The root can be treated as a loosely feasible node with a gap frequency $g = 1$, depth $\ell = 0$, and Round Robin frequency k . An example for the Pareto trie for $k = 4$ is shown in Figure 4.1.

As the Pareto trie is searched, an inclusion-minimal subset of its leaves, along with their solutions, are maintained – after the search is complete, these will form \mathbf{C}_k .

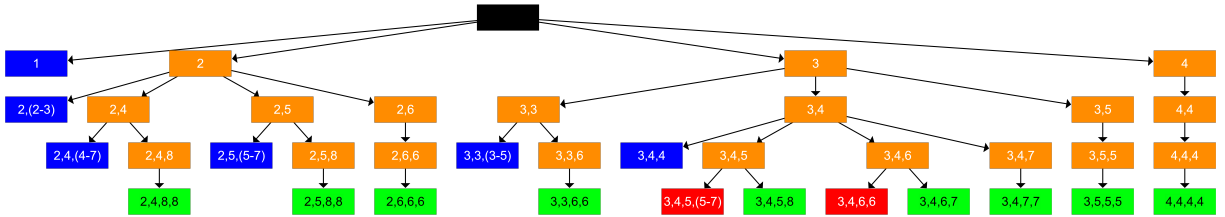


Figure 4.1: The portion of the Pareto trie for $k = 4$ that is explored by the traversal from Section 4.1.2. Nodes show one or more Pinwheel Scheduling instances: The black root node is the empty Pinwheel Scheduling problem; blue nodes are tightly feasible; orange nodes are loosely feasible; red nodes are infeasible and green nodes are feasible, complete and form the Pareto Surface. Solutions to the green nodes are shown in Table 4.1 (page 38).

Remark 4.1.7 Periodic solution length.

We can prove a bound on $m(k)$, the largest frequency in any instance of \mathbf{S}_k , using the trie and Proposition 3.1.2. For a loosely feasible instance $\mathcal{P} = (f_1 \leq f_2 \leq \dots \leq f_\ell)$, we can always find a schedule of length $\leq \prod_{i=1}^\ell f_i$ with at least one gap, so $g \leq \prod_{i=1}^\ell f_i$. Hence,

\mathcal{P} 's Round Robin frequency f is $f \leq (k - \ell) \prod_{i=1}^{\ell} f_i$, which gives an upper bound for all expansions $f_{\ell+1}$ to \mathcal{P} that can occur in \mathbf{S}_k . We hence always have $m(k) \leq m_k$, where $m_1 = 1$, $m_{\ell+1} = (k - \ell) \prod_{i=1}^{\ell} m_i$. Since $m_k = \Omega(2^{2^k})$, this proven upper bound for $m(k)$ is doubly exponential in k , whereas Conjecture 4.1.3 suggests the singly exponential bound $m(k) \leq 2^{k-1}$ is sufficient. As $m(k) \geq 2^{k-1}$, assuming Conjecture 4.1.3 would completely settle the question of the worst-case periodic solution length.

4.1.3 Uniqueness

Next we prove that \mathbf{C}_k is unique.

Lemma 4.1.8 Characterization Pareto Surface.

A Pinwheel Scheduling instance A can exist in some Pareto Surface \mathbf{C}_k iff A is feasible and decreasing any component of A by 1 makes it infeasible.

Proof. Consider some $A \in \mathbf{C}_k$, which is feasible by the definition of \mathbf{C}_k . Define $A_{i-} = (a_1 \leq \dots \leq a_{i-1} \leq a_i - 1 \leq a_{i+1} \leq \dots \leq a_k)$. Assume towards a contradiction that, for some value of $i \in [k]$, A_{i-} is also feasible. Since A_{i-} is feasible there must be some $B \in \mathbf{C}_k$ such that $B \leq A_{i-}$ and B is feasible. This means that $B \leq A_{i-} \leq A$, so anything dominated by A is also dominated by B , and we can remove A from the Pareto Surface; a contradiction.

Conversely, let A be feasible and let A_{i-} be infeasible for all $i \in [k]$. By the definition of \mathbf{C}_k , there is a feasible instance $B = (b_1 \leq b_2 \leq \dots \leq b_k) \in \mathbf{C}_k$ with $B \leq A$. Assume towards a contradiction that there is an index $i \in [k]$ with $b_i < a_i$; then we have $B \leq A_{i-}$. Since A_{i-} is already infeasible, so must B be; a contradiction. \square

The uniqueness claim from Theorem 4.1.1 now follows immediately from Lemma 4.1.8: \mathbf{C}_k consists of exactly those instances that satisfy the condition of that lemma.

Note that $\mathbf{P}_{k,d}$ does not in general have a unique Pareto Surface; for example, $\mathbf{P}_{2,2/3}$ has Pareto Surfaces $\{(2, 2)\}$ and $\{(2, 6), (3, 3)\}$. Both dominate all instances with density at most $2/3$ but fail to do so after deleting any one element of the sets, meeting the definition for $\mathbf{C}(\mathbf{P}_{k,d})$. In this instance, the former (\mathbf{C}_2) dominates the latter but that does not disqualify it as a Pareto Surface. Clearly \mathbf{P}_k is also a Pareto Surface for $\mathbf{P}_{k,d}$, so a finite $\mathbf{C}(\mathbf{P}_{k,d})$ always exists.

4.2 Engineering Pinwheel Scheduling

In this section, we introduce our backtracking algorithm for general Pinwheel Scheduling instances, with three consecutive stages building on each other: the Naïve algorithm, the Optimised algorithm, and finally the Foresight algorithm. The effects of each optimisation are discussed in Section 4.4.

4.2.1 The Naïve Algorithm

Each of the three algorithms presented here uses the same backtracking procedure to assess the schedulability of Pinwheel Scheduling instances. They form all possible solutions into a trie of *candidate solution prefixes* (\mathcal{S}_c), which they explore using four basic operations:

1. Push, which appends the next unexplored letter $i \in k$ to \mathcal{S}_c .
2. Pop, which deletes the last letter from \mathcal{S}_c .
3. Failure testing, which tests whether the current state is valid.
4. Success testing, which tests whether the current state is known to be sustainable.

Whenever a node is reached, the process tests for failure first, then for success. If a node is invalid, the pop operation is employed until there is an unexplored letter to push. Nodes pass the success test if their state is the same as some ancestral node – the path from that node to this node is a solution \mathcal{S} . If a node is valid but not known to be sustainable, the push operation is again employed. A flow chart depicting this procedure is shown in Figure 4.2.

Failure testing has two parts: conditions 1 and 2, called the safety conditions. Condition 3 is used for success testing. For a state \mathcal{X} in candidate solution prefix \mathcal{S}_c :

1. \mathcal{S}_c is feasible ($\forall t, i : x_i(t) \leq f_i$),
2. \mathcal{S}_c is not known to fail in the future (it has unexamined extensions that may succeed),
and
3. \mathcal{S} eventually succeeds ($\mathcal{X}(\mathcal{S}, t) = \mathcal{X}(\mathcal{S}, t + p)$, for some t and some $p > 0$).

To show that an instance \mathcal{P} is unsolvable, we must eliminate all candidate solution prefixes by showing that they fail either condition one or condition two. The third condition invokes periodicity – any path that returns to somewhere it has been can be followed indefinitely,

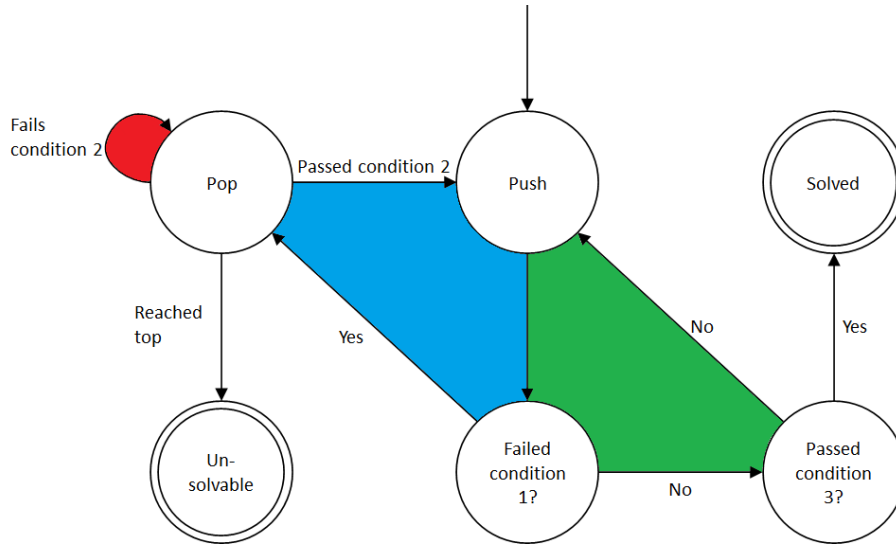


Figure 4.2: The backtracking procedure used to solve Pinwheel Scheduling problems. The left hand (red) loop climbs the tree when nodes have no potentially valid descendants, shortening \mathcal{S}_c . The central (blue) loop shows letters being tried until one is found which is valid, moving horizontally. The right hand (green) loop shows the extension of the tree when the safety conditions are passed.

returning to that location after each loop. Thus, if \mathcal{S}_c has this property, it can be safely repeated indefinitely, making \mathcal{S}_c a solution to \mathcal{P} .

In the naïve algorithm, tasks are pushed in descending frequency order. In feasible cases, this reduces the observed length of failed candidate solutions attempted before finding a viable schedule, thus reducing success testing cost. This seemed to reduce overall cost in many cases, probably because success testing is $O(n^2)$ (where n is the length of the testable solution fragment – naively $|(\mathcal{S}_c)|$, but optimised in Section 4.2.2) and failure testing is $O(k)$. An example of this difference is described in Figure 4.4.

Note that the first move can be freely chosen as all tasks must be a part of the final schedule.

Worked Examples

Examples of the trees generated by the Naïve method are shown in Figure 4.3 (infeasible) and Figure 4.4 (feasible). These examples follow conventions from the implemented code, which are slightly different from those used elsewhere:

1. states are 0 indexed, rather than 1 indexed ($x_i = 0$ after execution of task i), and
2. Pinwheel Scheduling instances are listed by weakly decreasing frequency, rather than weakly increasing frequency.

4.2.2 The Optimised Algorithm

The Optimised algorithm expands the Naïve algorithm described above with three improvements that remove repetitive and symmetric sections of the search space and reduce the cost of success testing.

Repetition

We first establish two simple properties when comparing different states. If we consider two states of the system, \mathcal{X} and \mathcal{X}' , then \mathcal{X} is *worse* than \mathcal{X}' if no $x_i \in \mathcal{X}$ is less than the corresponding $x'_i \in \mathcal{X}'$ and some x_i is greater than the corresponding x'_i . Formally: $\forall i : x_i \geq x'_i$ and $\exists i$ such that $x_i > x'_i$. \mathcal{X}' is then considered to be *better* than \mathcal{X} .

Lemma 4.2.1.

Any schedule \mathcal{S} which is valid from state \mathcal{X} in Pinwheel Scheduling instance \mathcal{P} is also valid from any better state \mathcal{X}' .

Proof. Let \mathcal{P} be a Pinwheel Scheduling instance for which \mathcal{S} is a valid schedule starting at state \mathcal{X} . Execute \mathcal{S} twice in parallel from two initial states: \mathcal{X} , and some better state \mathcal{X}' . At time t , let $\mathcal{X}(t)$ be the state of the former and $\mathcal{X}'(t)$ be the state of the latter. Proceed inductively:

The starting state must be valid in both executions: $\mathcal{X}(0)$ is part of a valid schedule and $\mathcal{X}'(0)$ is better, so $\forall i : x'_i(0) \leq x_i(0) \leq f_i$.

On an arbitrary day t , the schedule will select the same task $j = \mathcal{S}(t)$ for both executions. \mathcal{S} is valid from starting state \mathcal{X} , so for this execution $\forall i : x_i(t) \leq f_i$ before and after day t . After executing task j , $x_j(t) = 0$ and $\forall i \neq j : x_i(t) = x_i(t-1) + 1$. If $\mathcal{X}'(t-1)$ is better than or equal to $\mathcal{X}(t-1)$, then $\forall i : x'_i(t-1) \leq x_i(t-1) \leq f_i$ before day t and $x'_j(t) = x_j(t) = 0$ and $\forall i \neq j : x'_i(t) = x'_i(t-1) + 1 \leq x_i(t)$ after day t .

Thus, by induction, every state reached by both executions is valid, and \mathcal{S} is a valid schedule for \mathcal{P} from either starting state. □

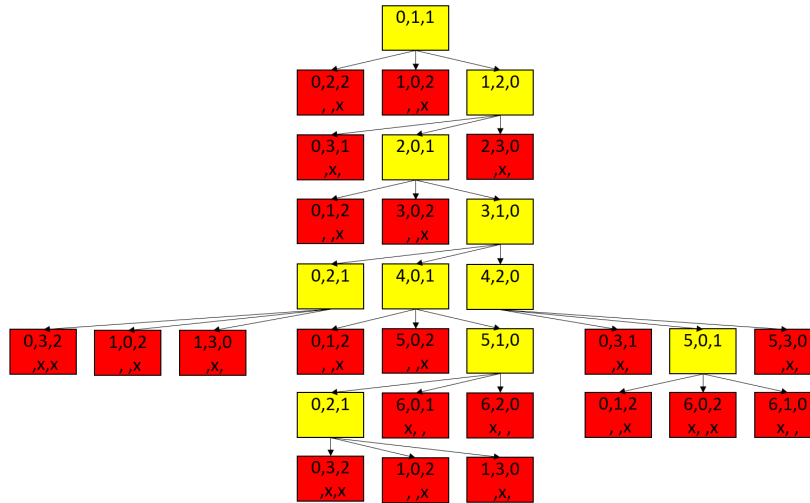


Figure 4.3: The trie produced by the Naïve procedure when applied to the input $(6, 3, 2)$. States which pass condition 1 are shown in yellow, while states which fail it are shown in red. As the tree is closed, success testing does not find a repeated condition, but success testing would compare all yellow elements to all of their ancestor elements for a total cost of 34 comparisons.

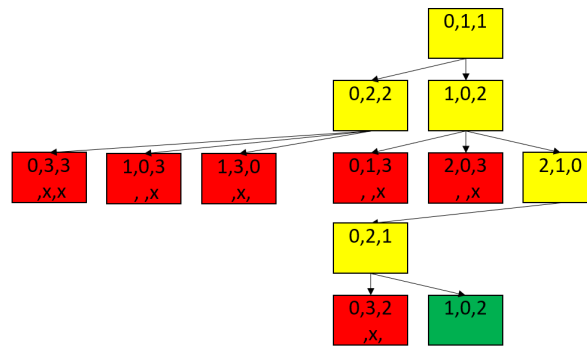


Figure 4.4: The trie produced by the Naïve procedure when applied to the input $(6, 3, 3)$. States which pass condition 1 are shown in yellow, states which fail it are shown in red and the eventual successful state is shown in green. Every state with descendants examines them from left to right, though several states have fewer than three descendants as the tree is open and is therefore only partially explored. Every yellow state was compared to all of its ancestor elements, while the green case was only compared until an identical state was found, for a total cost of 9 comparisons. If this example had used the most urgent to least urgent order described in Section 4.2.1, it would have examined 27 nodes for a testing cost of 104 comparisons. The solution found is $[1, 2, 0]$.

Lemma 4.2.2.

If Pinwheel Scheduling instance \mathcal{P} permits no valid schedule starting at some state \mathcal{X}' , then it permits no valid schedule starting at any worse state \mathcal{X} .

Proof. Consider a Pinwheel Scheduling system \mathcal{P} which permits no valid schedule from some state \mathcal{X}' . Suppose towards a contradiction that \mathcal{P} permits a valid schedule \mathcal{S} starting at some worse state \mathcal{X} . Per the previous Lemma, any schedule which is valid from state \mathcal{X} must also be valid from any better state \mathcal{X}' – a contradiction. \square

Avoiding immediate repetitions which are not themselves solutions shrinks the search space without changing the schedulability of Pinwheel Scheduling problems. This optimisation is based on the following observation.

Proposition 4.2.3 Repetition.

If a Pinwheel Scheduling instance \mathcal{P} admits a valid repetitive schedule $\mathcal{S}_r = \dots, r, r, \dots$ and r is not a solution to \mathcal{P} , then $\mathcal{S}_{1r} = \dots, r, \dots$ is also a valid schedule for \mathcal{P} .

Proof. Let \mathcal{S}_r be a solution to a Pinwheel Scheduling problem \mathcal{P} , and let r be a repeated phrase within \mathcal{S}_r , i.e., \mathcal{S}_r takes the form $\mathcal{S}_r = S_1, r, r, S_2$. Repeat \mathcal{S}_r until the same state is reached after the execution of each subsequence; let the state following S_1, r, r be called \mathcal{X} and the state following S_1, r be called \mathcal{X}' .

The state component x_i is the time since task i was last performed. For any tasks $i \in r$, it follows that $x_i = x'_i < |r|$ – likewise, For any task $j \notin r$, it follows that $x'_j \geq |r|$ and $x_j \geq 2|r|$. This means that state \mathcal{X} is worse than state \mathcal{X}' and, per Lemma 4.2.1, that any schedule valid from \mathcal{X} is valid from \mathcal{X}' . S_2, S_1, r is such a schedule, and is a cyclic permutation of \mathcal{S}_{1r} . \square

The simplest exploitation of Proposition 4.2.3 considers the simplest possible repeated subsequence – single character repetitions. Forbidding these has a small benefit in infeasible instances, namely reducing the effective alphabet size by 1 as the last letter played cannot be repeated.

In feasible instances the observed effect was larger, because of the order in which the trie is explored. Due to fixed order conventions, immediately repetitive additions to candidate solution prefixes are often the first to be tried. If we divide the search space around the first schedule found (\mathcal{S}_1), repetitive candidate solution prefixes are over-represented before \mathcal{S}_1 and thus removing them has a stronger effect on feasible instances.

Frequency Duplication

We call tasks in a Pinwheel Scheduling instance \mathcal{P} with the same values of f_i *duplicates*, as they are indistinguishable until either task is performed (after this point they can be distinguished by their x_i values, which can never again be identical). Naïvely, duplicate tasks are distinguished by their order of appearance in \mathcal{P} , but an alternative method exists which exploits frequency duplication by pruning identical subtrees.

Proposition 4.2.4 Duplicates.

If a Pinwheel Scheduling instance contains two tasks $i \neq i'$ with $f_i = f_{i'}$, then it is feasible when i is performed before i' iff it is feasible when i' is performed before i .

Proof. Consider a Pinwheel Scheduling system \mathcal{P}_s containing two symmetric tasks (i and i' such that $f_i = f_{i'}$). Let \mathcal{S}_c be an arbitrary sequence in \mathcal{P}_s where f_i is performed before $f_{i'}$. Then let \mathcal{S}'_c be an arbitrary sequence identical to \mathcal{S}_c , save that every occurrence of i is replaced with an occurrence of i' and vice versa. As $f_i = f_{i'}$, \mathcal{S}'_c exists iff \mathcal{S}_c exists. Therefore, if an infinite \mathcal{S}_c exists, an infinite \mathcal{S}'_c exists and vice versa. Also, if no infinite \mathcal{S}_c exists then no infinite \mathcal{S}'_c exists and vice versa. Therefore \mathcal{P}_s is feasible when i is performed before i' iff \mathcal{P}_s is feasible when i' is performed before i . \square

As having the same frequency is a transitive relation, Proposition 4.2.4 clearly applies to instances with more than two duplicate tasks. The algorithm can be optimised by choosing one ordering of all duplicate tasks, instead of naively exploring all orderings. While this effect is limited in scope (many Pinwheel Scheduling instances have no duplicates), it has a large effect on instances which have many duplicates: If the frequencies are sorted into bins according to their exact values, with b_j members per bin, then the total size of the graph is reduced by a factor of $B = \prod_{j=1}^{j^{max}} b_j!$ by this optimisation.

Minimum Solution Length

A valid schedule \mathcal{S} for Pinwheel Scheduling instance \mathcal{P} must contain all tasks $i \in [k]$, but must also follow the logic of density: each task i with frequency f_i must be at least $\frac{1}{f_i}$ of \mathcal{S} . To account for this, consider the *composition formula* R of candidate solution prefix \mathcal{S}_c . R has k components, with component r_i representing the number of instances of i in \mathcal{S}_c . If L is the length of the candidate solution prefix, then L is given by $L = \sum_{i=1}^k r_i$. For a candidate solution prefix to be sustainable, each letter i must appear at least every f_i letters, so over the whole solution $r_i \geq \lceil \frac{L}{f_i} \rceil$. To calculate the minimum value of L , L_{min} , we start by setting

$r_i = 1$ for all i , then increment r_i for each i value until this condition is met simultaneously for all i .

L_{min} can be used to avoid unnecessary comparisons in success testing in 2 ways:

1. Only comparing states which are L_{min} apart, because no closer states can be identical.
2. Only performing success testing when $|\mathcal{S}| \geq L_{min}$.

The former reduces the cost of success testing each node, while the latter reduces the number of nodes which perform success testing. This effect is significant because the cost of testing for success grows as $|\mathcal{S}|^2$ while all other costs remain constant over $|\mathcal{S}|$. For a discussion of minimum solution lengths in Pinwheel Scheduling instances with two distinct numbers, including several minimum solution length algorithms, see [Hol+92].

Worked Examples

Worked examples for the Pinwheel Scheduling instances shown in Section 4.2.1 are repeated using the Optimised algorithm in Figures 4.5 and 4.6.

4.2.3 The Foresight Algorithm

This optimisation modifies the Naïve failure testing process to gain more information from a similar amount of work. Instead of tracking conventional states $\mathcal{X}(t)$, recall *urgency* states $\mathcal{U}(t)$:

$$\forall i, t : u_i(t) = f_i - x_i(t) - 1 \tag{4.1}$$

This requires a different procedure when a task is performed (to perform task i , set $u_i = f_i - 1$) and a different growing procedure (to grow V , set $u_i(t + 1) = u_i(t) - 1$ for all i). The Naïve failure testing procedure would test that $\forall i : u_i \geq 0$ but an alternative failure testing procedure is now possible if the urgency values of tasks are stored in ascending order ($\forall i : u_i \leq u_{i+1}$):

Proposition 4.2.5 Urgency.

If an urgency state \mathcal{U} is feasible, then for every i we have $u_i \geq i$.

Proof. Consider a Pinwheel Scheduling instance \mathcal{P} at urgency state \mathcal{U} such that \mathcal{P} is feasible from \mathcal{U} . Proceed by induction over i .

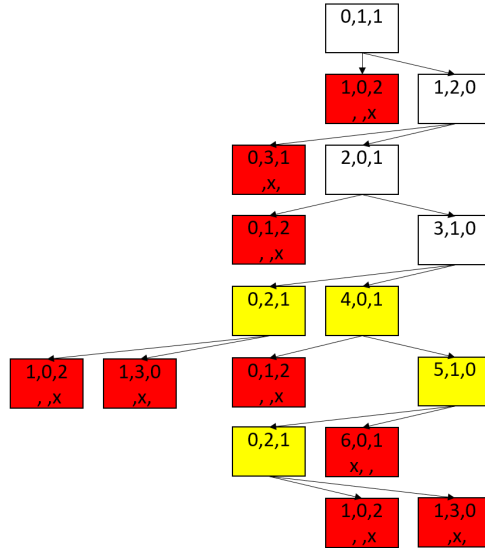


Figure 4.5: The trie produced by the Optimised algorithm when applied to the input $(6, 3, 2)$. States which pass condition 1 are shown in yellow or white, while states which fail it are shown in red. As opposed to the Naïve method for solving the same input (Figure 4.3), tasks are only scheduled if they were not performed the previous day. The minimum solution length for this input is 4, so only nodes deeper than 4 (coloured yellow) require success testing. Each such node is compared with all elements at least 4 days older than itself for a total of 7 comparisons and 17 nodes (while the Naïve algorithm uses 34 comparisons and 31 nodes for this instance).

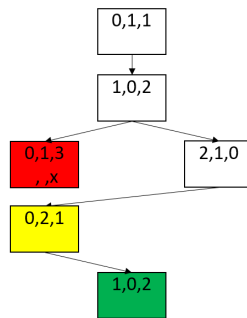


Figure 4.6: The trie produced by the Optimised algorithm when applied to the input $(6, 3, 3)$. States which pass condition 1 are shown in yellow, white or green; while states which fail it are shown in red. As opposed to the Naïve method for solving the same input (Figure 4.4), only tasks which were not performed the previous day are executed. The minimum solution length for this input is 3, so only nodes deeper than 3 (coloured yellow or green) require any success testing. Each such node is compared with all elements at least 3 days older than itself for a total of 3 comparisons and 6 nodes (while the Naïve algorithm uses 9 comparisons and 12 nodes for this instance).

- Base case: as \mathcal{U} is feasible, $\forall i : u_i \geq 0$ so $u_0 \geq 0$.
- Inductive hypothesis: $u_i \geq i$.
- Inductive step: \mathcal{U} is ordered, so $u_{i+1} \geq u_i$, hence $u_{i+1} \geq i$. If $u_{i+1} > u_i$ then $u_{i+1} \geq i+1$. Alternatively, if $u_i = u_{i+1}$ then either $u_i > i$ and $u_{i+1} = u_i > i$ or $u_{i+1} = u_i = i$.

Assume that $u_{i+1} = u_i = i$ towards a contradiction.

As \mathcal{U} is feasible, some infinite \mathcal{S} exists such that $\forall i, t : u_i(t) \geq 0$.

This schedule must execute u_i before $i + 1$ days have passed or it will have urgency $u_i \leq i - (i + 1)$ and hence $u_i < 0$ before being executed.

Likewise for u_{i+1} as $u_i = u_{i+1}$. As \mathcal{U} is ordered, all elements before u_i are less than or equal to u_i and hence less than or equal to i . Therefore they must also be executed before $i + 1$ days have passed, by the same reasoning. Therefore at least $i + 1$ tasks must be executed in the first i days, which contradicts the rule that only one element may be performed daily.

Therefore, $u_{i+1} \geq i + 1$.

Therefore in all feasible states $\forall i : u_i \geq i$. □

This can be used to detect failure up to to k days in advance for little additional cost over the Naïve failure testing method, reducing tree height. It can also be used to *force* the execution of certain tasks on certain days:

Proposition 4.2.6 Forcing.

If a feasible urgency state \mathcal{U} exists such that $\exists i' : u_{i'} = i'$, then the task at position i' and all preceding tasks must be executed in the next i' days.

Proof. Let \mathcal{U} be an urgency state of a Pinwheel Scheduling system \mathcal{P} , and let i' be a task in this system such that $u_{i'} = i'$. Let \mathcal{S} be an arbitrary schedule for \mathcal{P} from \mathcal{U} . As \mathcal{U} is ordered, $\forall i : u_i \leq u_{i+1}$, so for all elements preceding i' it holds that $u_{i < i'} \leq u_{i'}$ and hence $u_{i < i'} \leq i$. Assume towards a contradiction that there exists an $i'' \leq i'$ such that the task at i'' is executed after i' days. Initially, $u_{i''}(0) \leq i$. $u_{i''}$ decreases by 1 each day, so if it is executed on day t , before it is executed $u_{i''}(t \geq i + 1) \leq i - (i + 1) < 0$. This contradicts the safety condition that $\forall i, t : u_i(t) \geq 0$.

Therefore no task at positions $\leq i'$ can be executed later than day i' . □

This can be used to greatly restrict branching and hence tree breadth; if there exists some i' such that $u_{i'} = i'$ then the next move must have $i \leq i'$. This optimisation is compatible with all three changes described in Section 4.2.2, and all three are included in the final Foresight implementation.

4.2.4 Deciding Tight Feasibility

As introduced in Chapter 3, the tightness of Pinwheel Scheduling instances can be determined by testing for the existence of a schedule containing at least one gap. This was implemented using the Optimised algorithm in Section 4.2.2, by making the default action from every position a holiday and adding an extra testing step after a sustainable state was found – searching the schedule that produced this state for a holiday. If no gap was found in that schedule, the search continued until a loose schedule was found, or it was demonstrated that no loose schedule could exist. In principle, it would be possible to implement the Foresight algorithm from Section 4.2.3 with gaps, but this would have required a full re-implementation of Foresight – a substantial time investment.

4.3 Engineering the 5/6 Surfaces

Our principal application of the algorithms from Section 4.2 is the investigation of the $\frac{5}{6}$ conjecture for low k values. This section describes our algorithm for computing a Pareto Surface for $\mathbf{C}(\mathbf{P}_{k,5/6})$, code for which is available online [Smi21].

4.3.1 Core Algorithm

We search the trie of Pinwheel Scheduling problems introduced in Section 4.1.2 using the depth first search procedure outlined in that section. The search from a node at depth h begins by creating a child with the smallest possible added frequency, then proceeds until the subtree of the new node is fully explored. If a node has density $d = \frac{5}{6}$, it can have no descendants and is fully explored – otherwise the depth first search proceeds by fully exploring all children until each has a descendent with a symmetry of $k + 1 - h$. This descendent necessarily dominates all siblings seen after it in a depth first search. As only nodes with a density $d \leq \frac{5}{6}$ need be considered, denser nodes are ignored by this process.

We will outline several optimisations which introduce denser problems that may be used to dominate problems found by this search. To show that the $\frac{5}{6}$ conjecture is true for a certain value of k , we need to show that no infeasible Pinwheel Scheduling systems with a density $\leq \frac{5}{6}$ exist for that value of k . That is, we need to show that the set of all infeasible Pinwheel Scheduling systems with $d \leq \frac{5}{6}$ found when constructing $\mathbf{C}(\mathbf{P}_{k,5/6})$ is the empty set.

4.3.2 Constructing the Pareto Surface

We could consider *the* density restricted Pareto Surface comprised exclusively of members of $\mathbf{P}_{k,5/6}$, but this would prevent many useful optimisations. Instead, we require *any* density restricted Pareto Surface, consisting of a set of solutions which solve all Pinwheel Scheduling instances with a density $\leq \frac{5}{6}$ – that is, we allow our surface to contain denser instances, so long as it remains complete, inclusion-minimal, and no infeasible Pinwheel Scheduling instances with density $\leq \frac{5}{6}$ are found. This allows for optimisations which use easily schedulable feasible Pinwheel Scheduling instances to dominate large classes of feasible Pinwheel Scheduling instances with density below $\frac{5}{6}$ that are harder to schedule.

Frequency Capping

This optimisation builds on Conjecture 4.1.4, which we eagerly assume to be true here but then immediately check the validity of for each instance generated. We cap the maximum f_i value of considered Pinwheel Scheduling instances at 2^{k-1} . This only lowers frequencies, so the capped Pinwheel Scheduling instance dominates both the instance it was created from and often many similar instances – particularly when multiple frequencies are capped.

Because capping reduces frequencies, it raises densities. To avoid a potential issue where the density of a problem is below $\frac{5}{6}$ before capping but above $\frac{5}{6}$ after capping, we replace $\mathbf{P}_{k,5/6}$ with the similar and dominant $\mathbf{P}'_{k,5/6}$. This set includes all problems where either $d \leq \frac{5}{6}$ and $\forall i : f_i < 2^{k-1}$ or which consist of a prefix with density $d \leq \frac{5}{6}$ and a suffix where $\forall i : f_i = 2^{k-1}$.

Any infeasible members of $\mathbf{P}'_{k,5/6}$ would be counterexamples to either the 5/6 Conjecture (Question 1.1.1), or the Kernel Conjecture (Conjecture 4.1.4); which one would require future investigation, but both conjectures have proved true in all instances we have tested.

k	Foresight	Optimised	Naïve	Opt/FS	Naïve/Opt	Surface Size
6	2.36 ± 0.02	3.30 ± 0.07	4.97 ± 0.06	1.40 ± 0.03	1.51 ± 0.04	23
7	6.65 ± 0.04	18.80 ± 0.07	39.5 ± 0.3	2.83 ± 0.02	2.10 ± 0.02	78
8	16.3 ± 0.2	487 ± 3	895 ± 6	29.8 ± 0.3	1.84 ± 0.02	214
9	105.0 ± 0.8	367 ± 3	645 ± 3	3.50 ± 0.04	1.76 ± 0.02	638
10	869 ± 5	944 ± 2	3130 ± 30	1.086 ± 0.006	3.31 ± 0.04	5347
11	4300 ± 20	4670 ± 10	9860 ± 60	1.015 ± 0.006	2.26 ± 0.02	15265

Table 4.2: Total time in seconds and relative speedup to generate the 5/6 Pareto Surface using three Pinwheel Scheduling oracles and all optimisations from Section 4.3.2; the last column shows the size of the Pareto Surface found by Foresight, taken from a representative run because errors are too small to adequately estimate. Data for $k = 12$ is not shown: this surface took substantially longer to generate, requiring both the Foresight oracle and a more powerful computer than was used to generate the data shown here.

Folding

This optimisation uses a pair of simple operations on Pinwheel Scheduling instances: folding and unfolding. The c -task *folding* of a Pinwheel Scheduling instance $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ is the instance $B = (b_1 \leq b_2 \leq \dots \leq b_{k-c+1})$ with $k - c$ tasks $a_1 \leq a_2 \leq \dots \leq a_{k-c}$ and one task of frequency $\lfloor a_{k-c+1}/c \rfloor$, i.e., B is obtained from A by replacing the last c tasks for a single one with frequency $\lfloor a_{k-c+1}/c \rfloor$. The c -wise *unfolding* of a Pinwheel Scheduling instance $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ at task i is the instance $B = (b_1 \leq b_2 \leq \dots \leq b_{k+c-1})$, where B has $k - 1$ tasks of frequencies $a_1 \leq \dots \leq a_{i-1}, a_{i+1} \leq \dots \leq a_k$ plus c tasks each with frequency $c \cdot a_i$, i.e., B is obtained from A by replacing task i with c tasks with frequency $c \cdot a_i$.

Note that unfolding does not alter density, whereas folding never *lowers* density (but can substantially increase it if the folded tasks have different frequencies). Moreover, any schedule for A can be turned into a schedule for a c -wise unfolding of A by repeating the schedule c times, replacing each occurrence of i in B with the next of the c new tasks. Likewise, any schedule for a c -task folding of A can be used to generate a schedule for A itself, using the same process.

We use this as follows. Whenever a Pinwheel Scheduling instance \mathcal{P} needs to be solved, we try to find schedules for all viable foldings of that instance in parallel. If any c -task folding is feasible, we find the strictest Pinwheel Scheduling instance solved by its schedule and unfold the folded task back into c tasks. If $c > 1$, this unfolded instance will dominate the original instance \mathcal{P} but will frequently have higher symmetry than \mathcal{P} , and hence reduce

the amount of the Pareto Trie which must be explored. These folded problems will usually be faster to solve than \mathcal{P} because they have fewer tasks, smaller task separations, and exploit the frequency duplication optimization.

No challenge to the $\frac{5}{6}$ conjecture has been found unless the original instance \mathcal{P} is infeasible, so all foldings can be considered in parallel and any infeasible instances with $c > 1$ discarded. As some Pinwheel Scheduling instances can be dramatically more challenging to solve than others (for our tools), a very substantial speedup was achieved by running all foldings in parallel and terminating all threads as soon as the first feasible folding was found.

Initializing the Pareto Surface

While the core algorithm constructs $\mathbf{C}(\mathbf{P}_{k,5/6})$ from scratch, we can speed this up substantially by starting with some scheduled Pinwheel Scheduling problems, which may then be used to dominate problems in need of a solution. Unfolding a feasible smaller instance to k tasks is an easy way to generate many scheduled Pinwheel Scheduling problems.

We pump prime our computations by using an *unfolding surface*: Consider the example of the one-task instance (1). We recursively unfold this problem in each way that obtains k tasks. All resulting instances are both dense and feasible. If we want instances of $k = 3$ tasks, we first unfold (1) to (3, 3, 3) directly, then to (2, 2) and unfold that to (2, 4, 4).

The first version of this optimisation ($P(1)$) uses this unfolding of the single task instance (1) as sketched above; the second ($P(5)$) unfolds \mathbf{C}_5 , the Pareto Surface for $k = 5$ and the third ($P(k - 1)$) adds all elements of the *previous* Pareto trie. These developmental stages are evaluated in Section 4.4.2.

4.3.3 Searching the Pareto Surface

With the above approximations, most Pinwheel Scheduling problems considered at any k value have known solutions at any time (99.8% of problems in the Pareto Surface for $k = 11$ were solved by unfolding the $k = 10$ surface). In this case, the challenge is to search for a viable schedule in the known portion of the Pareto Surface. This is large at high k values (per Table 4.2), so it is crucial to search this surface efficiently. This task is effectively a dominance query over a dynamic set of points in \mathbb{R}^k .

While we could employ a standard data structure for orthogonal range searching here, the specific structure of our point set (the Pinwheel instances) suggests a bespoke trie-based solution: We maintain all members of the known portion of the Pareto Surface in a set of

tries, separated according to their symmetries (each a subset of the Pareto trie, and using the same ordering conventions). These tries are searched in descending order of symmetry with depth first dominance queries, to find the highest symmetry solution to each solved problem. This data structure minimises repeated comparisons of the same value, but also exploits the structure of solvable Pinwheel Scheduling problems.

When searching for a problem with a low value of f_0 , high f_0 problems are immediately excluded; when searching for a problem with a higher f_0 value, low f_0 problems are initially considered but quickly eliminated, because density constraints force them to escalate rapidly. The latter parts of problems are far less predictable, while also having a much larger range of possible values, and thus the dimensions which must be searched are highly asymmetric.

4.4 Performance Evaluation

This section reports on an extensive running-time study for various aspects of our tools.

4.4.1 Pinwheel Schedulers

We begin by evaluating the relative performance of an implementation of the state-graph based algorithm (Graph) and our backtracking algorithms (Naïve, Optimised and Foresight) using synthetic data. We then further evaluate the ability of the Naïve, Optimised and Foresight algorithms to compute Pareto Surfaces.

Randomly Generated Data

The four schedulers introduced in Sections 3.1.1 and 4.2 were evaluated using Pinwheel Scheduling instances generated using the following random process: Let a real number b be a density budget, initially 1. Generate a random real number $0 < r \leq b$, and from it a candidate task $f_c = \lfloor \frac{1}{r} \rfloor$. While $b - \frac{1}{f_c} > 0$, continue adding new tasks to an initially empty Pinwheel Scheduling problem \mathcal{P} , updating b each time ($b \rightarrow b - \frac{1}{f_c}$). Eventually, a task will be proposed which would raise the density of \mathcal{P} above 1; this task will be rejected. If $b \neq 0$, we add a final task $f_k = \lceil \frac{1}{b} \rceil$. Finally, we sort \mathcal{P} and replace any tasks where $f_i > 2^{k-1}$ with $f_i = 2^{k-1}$.

Problems generated this way have high densities, a variety of k values and a mixture of low and high f_i values, which makes them challenging to schedule. Capping by 2^{k-1} is done in light of Conjecture 4.1.3 to make instances more representative of the instances for which our algorithms were designed.

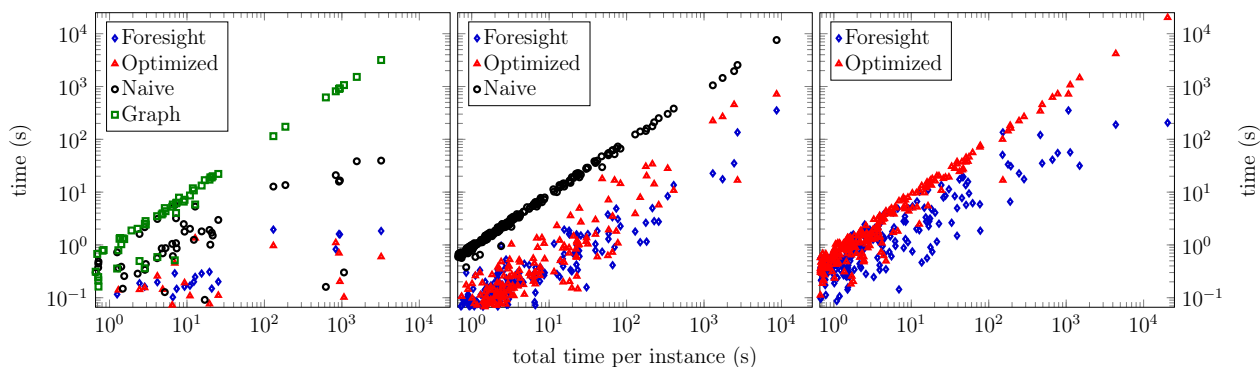


Figure 4.7: Results of a tournament between the four Pinwheel Scheduling solvers introduced in Chapter 3 and Section 4.2, using the randomly generated Pinwheel Scheduling problems introduced in Section 4.4.1. After each round, the slowest method was eliminated: first the Graph method, then the Naïve method, and finally the Optimised method. The x -axes show the total time all methods took to solve each instance, a measure of overall complexity due to the apparent correlation between solve times for different methods. The y -axis shows the individual running time of the compared algorithms.

A running-time study using these problems is shown in Figure 4.7, which demonstrates that each algorithm improves on its predecessor. Here, we repeatedly draw instances from the above distribution, but if an instance had already been drawn earlier, it is rejected and replaced with a newly drawn instance. Since instances with few tasks are more likely to arise in the above random process, non-rejected instances tend to get increasingly challenging over time. The correlation between instance difficulty for different algorithms is noteworthy. This suggests the existence of an intrinsic “difficulty” for Pinwheel Scheduling instances, at least w.r.t. the algorithms we studied. We leave a further exploration of this observation for future work.

The 5/6 Surface

A secondary evaluation used the time to solve $\mathbf{C}(\mathbf{P}_{k,5/6})$ with each method (see Table 4.2). This evaluation showed more variability between the performance of the Optimised and Foresight algorithms than seen in the previous section – with the Optimised algorithm taking

29.8 ± 0.3 times as long at $k = 8$ but 1.015 ± 0.006 times as long at $k = 11$ as the Foresight algorithm. This is likely due to the dominance of *search time* at high k values – at $k = 11$, Foresight and Optimised respectively spent $96.2 \pm 0.6\%$ resp. $87 \pm 4\%$ of their time matching problems with known solutions.

As such, the size of the $P(k-1)$ approximation of the Pareto Surface was the determining factor in these times – a complex effect of the properties of the specific solutions found by each method. Future algorithms will aim to produce solutions more capable of dominating many problems and less costly search procedures for the Pareto Surface.

4.4.2 Constructing the 5/6 Pareto Surface

This section evaluates the methods used to generate $\mathbf{C}(\mathbf{P}_{k,5/6})$, which were introduced in Section 4.3.2.

Frequency Capping

The Kernel optimisation improved performance in two key ways: Firstly, it increased the symmetry of problems, thus reducing the number of problems that needed to be considered. Secondly, it reduced the largest f_i values, which was particularly helpful for the deadline-driven Foresight algorithm, as it often ignores tasks with large f_i values for long periods of time. Reducing the maximum f_i value combated this, but our ongoing work will produce a version of Foresight more capable of handling arbitrarily large f_i values.

Folding

While folding had several benefits, the principal one was in exploiting the large variance between the cost of solving different Pinwheel Scheduling problems. Feasible problems, problems with smaller k values, and problems with smaller maximum f_i values are substantially faster than the converse – solving only the fastest of a set of problems that differ in these respects thus saves very considerable amounts of time. While starting all problems in the folded set increases overall work, these problems are solved in parallel, so this does not translate to a substantial additional time cost.

In addition to often being faster to solve, problems which have been folded, solved and then unfolded usually have higher symmetries and lower f_i values than the problems used to generate them, making them better at dominating other instances.

Initializing the Pareto Surface

Approximating the Pareto Surface had significant effects on solve times. In addition to being very cheap to generate, schedules produced by approximation tend to solve problems with very high densities (as all unfoldings of a problem share the density of that problem) and high symmetries – they are thus ideal for dominating problems and reducing the size of the search space.

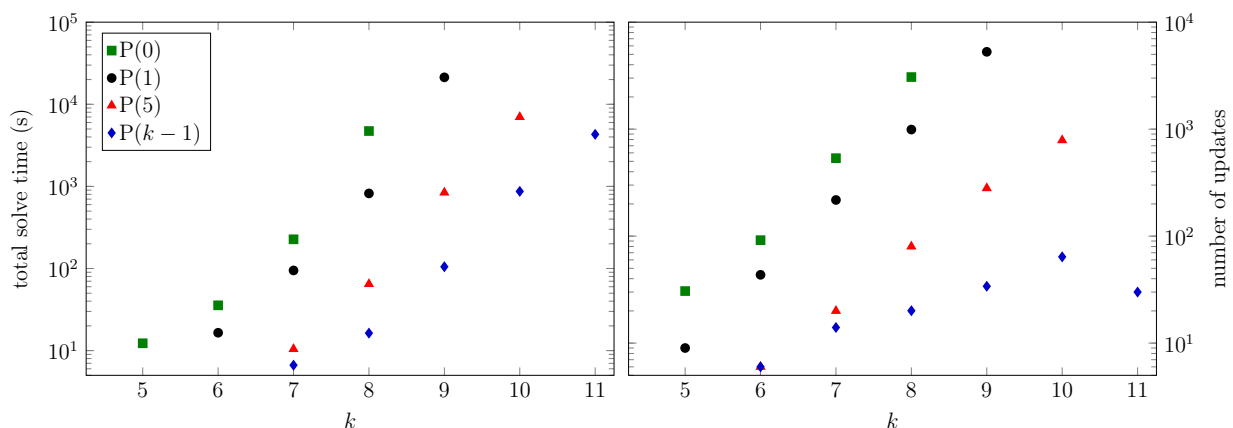


Figure 4.8: Running time (left) and total updates to each approximation required to generate a complete density-restricted Pareto Surface (right) with no approximation, and with all three approximations introduced in Section 4.3.2. The left hand figure demonstrates that while all costs are exponential with respect to k , $P(k-1)$ improves on $P(5)$, which improves on $P(1)$, which improves on the algorithm using no approximation. The right hand figure shows why – the approximation of the Pareto Surface is significantly better for the later methods.

As shown in Figure 4.8, each of the three approximations we considered substantially improves on its predecessor. With the introduction of the $P(k-1)$ approximation, the cost of searching the partial Pareto Surface began to dominate the cost of producing useful schedules, limiting the observed performance of $P(k-1)$ – though we plan to address this with our future work.

4.4.3 Searching the Pareto Surface

Both trie-based searching and Naïve searching were implemented, with trie-based searching running 8.80 ± 0.05 times faster at $k = 11$, a substantial speed increase (though the performance gain was less substantial at smaller k values, probably due to their smaller Pareto Surfaces and the overheads inherent in a more complex data structure).

4.5 Conclusion

At time of publication, the paper on which this chapter was based presented new evidence for the 5/6-density conjecture in Pinwheel Scheduling (Question 1.1.1) by engineering algorithms to compute a finite set of schedules that solves any of the infinitely many solvable instances with at most 12 tasks and $d \leq \frac{5}{6}$. This substantially strengthened the confidence in the conjecture and has led to new tools (theoretical and software) of independent interest for studying Pinwheel Scheduling.

As discussed in Section 2.1.2, several of these tools contributed to the eventual resolution of the 5/6 density conjecture[Kaw24].

Moreover, we have constructed the full Pareto Surfaces of Pinwheel Scheduling problems for $k \leq 5$, shown in Table 4.1, i.e., any Pinwheel Scheduling instance with at most 5 tasks is feasible if and only if one of the schedules listed in Table 4.1 is valid for it.

* * * * *

Chapter 5

Polyamorous Scheduling

Outline Finding schedules for pairwise meetings between the members of a complex social group without creating interpersonal conflict is challenging, especially when different relationships have different needs. This chapter presents our initial study of the underlying optimisation problem presented at FUN 2024: “Polyamorous Scheduling” [GSW24]. In Polyamorous Scheduling, we are given an edge-weighted graph and try to find a periodic schedule of matchings in this graph such that the maximal weighted waiting time between consecutive occurrences of the same edge is minimised.

We begin in Section 5.1 by formally introducing key results. We continue in Section 5.2 by showing that Decision Polyamorous Scheduling (DPS) is in PSPACE and is NP-hard. In Section 5.3, we examine the unweighted version of Optimisation Polyamorous Scheduling (OPS), showing that OPS does not allow polynomial time approximation algorithms with any ratio less than $\frac{4}{3}$ unless $P = NP$, with or without weights. On the other hand, Section 5.4 presents an $\mathcal{O}(\log n)$ -approximation algorithm (indeed, an $\mathcal{O}(\log \Delta)$ -approximation where Δ is the maximum degree of the graph). Finally, Section 5.5 defines a generalisation of density from the Pinwheel Scheduling Problem, “poly density”, and asks whether there exists a poly density threshold similar to the 5/6-density threshold for Pinwheel Scheduling [Kaw24].

5.1 Results

Despite the flurry of recent results for Periodic Scheduling problems, Polyamorous Scheduling seems not to have been studied before we introduced it in 2024 [GSW24]. Apart from its immediate practical applications, some quirks make Polyamorous Scheduling an interesting combinatorial optimization problem in its own right.

The first version of this manuscript used a direct reduction from 3SAT to introduce the following hardness-of-approximation result, which rules out the existence of a PTAS (polynomial-time approximation scheme) for Optimisation Polyamorous Scheduling.

Theorem 5.1.1 SAT Hardness of approximation.

Unless $P = NP$, there is no polynomial-time $(1 + \delta)$ -approximation algorithm for the Optimisation Polyamorous Scheduling problem for any $\delta < \frac{1}{12}$.

We retain this original proof in Appendix A, both for the record and because we expect future works to expand on the methods it develops. We have, however, since found a substantially simpler and stronger hardness-of-approximation result, Theorem 5.1.2, by containing the 3-Regular Chromatic Index Problem as a special case.

Theorem 5.1.2 Hardness of approximation.

Unless $P = NP$, there is no polynomial-time $(1 + \delta)$ -approximation algorithm for the Optimisation Polyamorous Scheduling problem for any $\delta < \frac{1}{3}$.

Though the current form of Theorem 5.1.1 follows from Theorem 5.1.2, the direct 3SAT reduction is significantly more versatile and we hope to improve the lower bound on the approximation ratio in future work. The core idea of the reduction in Theorem 5.1.1 is to force any valid schedule to have a periodic structure with a 3-day period, where edges scheduled on days t with $t \equiv 0 \pmod{3}$ represent the value *True* and edges scheduled on days with $t \equiv 1 \pmod{3}$ represent *False*; the remaining slots, $t \equiv 2 \pmod{3}$, are required to enforce correct propagation along logic gadgets. We return to this proof in Chapter 6.

Appendix A gives the detailed gadget constructions and proofs; Figure A.1 (page 158) shows the resulting DPS instance corresponding to an example 3-CNF formula.

Theorems 5.1.1 and 5.1.2 of course imply the NP-hardness of Polyamorous Scheduling; overall, we have 3 independent reductions establishing this. Section 5.2 surveys these and shows that the best-known upper bound for the complexity of Polyamorous Scheduling is PSPACE. We could thus call Polyamorous Scheduling *very* NP-hard; yet, efficient approximation algorithms are possible. Finding an edge colouring then using a simple round-robin schedule of its colours yields a good approximation if *both* the maximum degree and the ratio between the smallest and the largest desire growth rates are small:

Theorem 5.1.3 Colouring approximation.

For an Optimisation Polyamorous Scheduling instance $\mathcal{O} = (P, R, g)$ set $g_{\min} = \min_{e \in R} g(e)$, $g_{\max} = \max_{e \in R} g(e)$. Also, let Δ be the maximum degree in (P, R) and h^* be the heat of an optimal schedule. There is an algorithm that computes in polynomial time a schedule \mathcal{S} of heat h with $\frac{h}{h^*} \leq \min\left\{\frac{\Delta+1}{\Delta} \cdot \frac{g_{\max}}{g_{\min}}, \Delta + 1\right\}$.

A fully general approximation seems only possible with much weaker ratios; we provide an $\mathcal{O}(\log \Delta)$ -approximation by applying Theorem 5.1.3 to groups with similar weight and interleaving the resulting schedules:

Theorem 5.1.4 Layering approximation.

For an Optimisation Polyamorous Scheduling instance $\mathcal{O} = (P, R, g)$, let Δ be the maximum degree in (P, R) and h^* be the heat of an optimal schedule. There is an algorithm that computes in polynomial time a schedule \mathcal{S} of heat h with $\frac{h}{h^*} \leq 3\lceil \lg(\Delta + 1) \rceil = \mathcal{O}(\log n)$, where $n = |P|$.

Finally, we generalize the notion of density to Polyamorous Scheduling. As discussed above, density has proven instrumental in understanding the structure of Pinwheel Scheduling and in devising better approximation algorithms, by providing a simple, instance-specific lower bound. For Polyamorous Scheduling, the fractional problem is much richer, and indeed remains nontrivial to solve. We devise a generalization of density¹ for Polyamorous Scheduling from the dual of the Linear Program (LP) corresponding to a fractional variant of Polyamorous Scheduling, which gives the following instance-specific lower bound:

Theorem 5.1.5 Fractional lower bound.

Let $\mathcal{O} = (P, R, g)$ be an OPS instance with optimal heat h^* . For any set of values $z_e \in [0, 1]$, for $e \in R$, with $\sum_{e \in R} z_e = 1$, we have

$$h^* \geq \bar{h}(z) = \frac{1}{\max_{\mathbf{M} \in \mathbf{M}} \sum_{e \in \mathcal{M}} \frac{z_e}{g(e)}}$$

where \mathbf{M} is the set of all inclusion-maximal matchings in (P, R) . The poly density of \mathcal{O} is \bar{h}^* , the largest value of $\bar{h}(z)$ over all feasible z .

¹Note that poly density describes how tightly the polycule packs meetings together and not how dense each member personally is.

This bound formally establishes simple ad hoc bounds such as the following, which corresponds to the lower bound of G on the height in Bamboo Garden Trimming (setting $z_e = g(e)/G$):

Corollary 5.1.6 Total growth bound.

Given an OPS instance $\mathcal{O} = (P, R, g)$ with optimal heat h^* , let $G = \sum_{e \in R} g(e)$ and m be the size of a maximum matching in (P, R) ; then $h^* \geq G/m$.

More importantly though, Theorem 5.1.5 allows us to define a *poly density* similarly to the Pinwheel Scheduling Problem, and allows us to formulate the most interesting open problem about Polyamorous Scheduling:

Question 5.1.7 Is there a Poly Density Threshold?.

Is there a constant c such that every Decision Polyamorous Scheduling instance $\mathcal{D} = (P, R, f)$ with poly density $\bar{h}^*(\mathcal{D}) \leq c$ admits a valid schedule?

We will answer this question in the next chapter by showing that $c = \frac{1}{4}$ is such a constant.

For a DPS instance $\mathcal{D} = (P, R, f)$, define the poly density of \mathcal{D} , $\bar{h}^*(\mathcal{D})$, as the poly density of the OPS instance $\mathcal{O} = (P, R, 1/f)$ (see also Lemma 3.2.3).

5.2 Computational Complexity

One proof of the NP-hardness of the Decision Polyamorous Scheduling (DPS) Problem is that it contains Pinwheel Scheduling as a special case, an NP-hard problem [JL14]. We show in Section 5.3 that OPS also contains the Chromatic Index problem as a special case, which gives another proof of the NP-hardness of DPS using the conversion in Lemma 3.2.3. Since all good things come in threes, our inapproximability result in Appendix A gives a third independent proof of NP-hardness by reducing 3SAT to DPS.

Upper bounds on the complexity of DPS are much less clear. Similar to other Periodic Scheduling problems, characterizing the computational complexity of Polyamorous Scheduling is complicated by the fact that there are feasible instances that require an exponentially large schedule. It is, therefore, unclear whether Decision Polyamorous Scheduling is in NP since no succinct Yes-certificates are known; this is unknown even for the more restricted Pinwheel Scheduling Problem [Kaw24].

The following simple algorithm shows that DPS is at least in PSPACE (see also [KS20; GSW22]): Given the polycule $\mathcal{D} = (P, R, f)$ with $|P| = n$ and $|R| = m$, construct the *configuration graph* $\mathcal{G}_c = (V, E)$, where V consists of “countdown vectors” listing for each edge e how many days remain before e has to be scheduled again. $v \in V$ has an outgoing edge for every maximal matching \mathcal{M} in $\mathbf{M}(P, R)$, and leads to a successor configuration where all $e \in \mathcal{M}$ have their urgency reset to $f(e)$ and all $e \notin \mathcal{M}$ have their countdown decremented. Feasible schedules for \mathcal{D} correspond to infinite walks in the finite \mathcal{G}_c , and hence must contain a cycle. Conversely, any cycle forms a valid periodic schedule. Our algorithm for DPS thus checks whether \mathcal{G}_c contains a cycle in time $O(|V| + |E|)$.

The configuration graph \mathcal{G}_c has single exponential size: $V = \{(u_e)_{e \in R} : u_e \in [0..f(e)]\}$ and E has an edge for every matching in (P, R) . So $|E| \leq |V| \cdot 2^m$ (since we have at most $2^{|R|}$ matchings) and $|V| \leq \prod_{e \in R} f(e)$. To further bound this, we use the fact that all $f(e)$ need to be encoded explicitly in binary in the input. $\prod_{e \in R} f(e) \leq \prod_{e \in R} 2^{|f_e|} = 2^{\sum |f_e|} \leq 2^N$, where N is the size of the encoding of the input.

To obtain a PSPACE algorithm, we use the polylog-space s - t -connectivity algorithm (using Savich’s Theorem on the NL-algorithm that guesses the next vertex in the path) on \mathcal{G}_c , computing the required part of the graph on-the-fly when queried; this yields overall polynomial space.

5.3 Unweighted Polyamorous Scheduling & Edge Coloring

Given an OPS instance $\mathcal{O} = (P, R, g)$, one can always obtain a feasible schedule from a proper *edge colouring* $c : E \rightarrow [C]$ of the graph (P, R) : any round-robin schedule of the colours is a valid schedule for \mathcal{O} , and the number of colours becomes the separation between visits. More formally, we can define a schedule \mathcal{S} via $\mathcal{S}(t) = \{e \in R : c(e) \equiv t \pmod{C}\}$. An example is shown in Figure 5.1.

Such a schedule can yield an arbitrarily bad solution to general instances of \mathcal{O} , but it gives optimal solutions for a special case: The non-hierarchical polycule \mathcal{O}_u , which is an OPS polycule where all growth rates are $g_{i,j} = 1$ (i.e., an unweighted graph). Recall that any graph with maximal degree Δ can be edge-coloured with at most $\Delta + 1$ colours and clearly needs at least Δ colours.

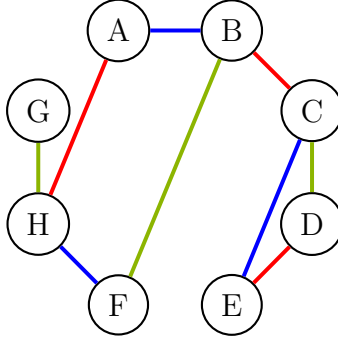


Figure 5.1: An unweighted polyamorous scheduling instance (that is, an OPS instance where all edges have growth rate 1). Edge colours show one optimal schedule, where every edge is visited exactly every three days: $[3, 3, 3]$, i.e., all red edges are scheduled on days t with $t \equiv 0 \pmod{3}$, all blue edges when $t \equiv 1 \pmod{3}$ and green edges for $t \equiv 2 \pmod{3}$.

Proposition 5.3.1 Unweighted OPS = edge coloring.

An unweighted OPS problem admits a schedule with heat h if and only if the corresponding graph is h -edge-colourable.

Proof. First note that any k -edge-colouring immediately corresponds to a schedule that visits each edge every k days, since we can schedule all edges e with $c(e) = i$ on days $t \equiv i \pmod{C}$. Moreover, any schedule with height h must visit every edge at least once within the first h days (otherwise it would grow to desire $> h \cdot 1$). We can therefore assign h colours according to these first h days of the schedule; some edges might receive more than one colour, but we can use any of these and retain a valid colouring using h colours. \square

Since it is NP-complete to decide whether a graph has chromatic index $\chi_1 = \Delta$ (even when the graph is 3-regular [Hol81]) unweighted Polyamorous Scheduling is NP-hard. This provides a second restricted special case of the problem that is NP-hard, which also gives us the inapproximability result stated in Theorem 5.1.2:

Theorem 5.1.2 (restated). Hardness of approximation.

Unless $P = NP$, there is no polynomial-time $(1 + \delta)$ -approximation algorithm for the Optimisation Polyamorous Scheduling problem for any $\delta < \frac{1}{3}$.

Proof of Theorem 5.1.2. Assume that there is a polynomial-time algorithm A that achieves an approximation ratio of $\frac{4}{3} - \varepsilon$ for some $\varepsilon > 0$. Given an input $\mathcal{G} = (V, E)$ to the 3-Regular Chromatic Index Problem (i.e., given a 3-regular graph, decide whether $\chi_1(\mathcal{G}) = \Delta = 3$ or $\chi_1(\mathcal{G}) = \Delta + 1 = 4$), we can apply A to (V, E, g) , setting $g(e) = 1$ for all $e \in E$. By

Proposition 5.3.1, A finds an edge colouring with $c \leq (\frac{4}{3} - \varepsilon) \cdot \chi_1(\mathcal{G})$ colours. If $\chi_1(\mathcal{G}) = \Delta = 3$, then $c \leq 4 - 3\varepsilon < 4$, so $c = 3$; if $\chi_1(\mathcal{G}) = 4$, then $c \geq 4$. Comparing c to Δ thus determines $\chi_1(\mathcal{G})$ exactly in polynomial time; in particular, for every 3-regular graph, this decides whether $\chi_1(\mathcal{G}) = 3$. Since 3-Regular Chromatic Index is NP-complete [Hol81], it follows that $P = NP$. \square

We close this section with the remark that there are weighted DPS instances where any feasible schedule must “multi-colour” some edges, including the polycule shown in Figure 5.2. For the general problem, we thus cannot restrict our attention to edge colourings (though they may be a valuable tool for future work).

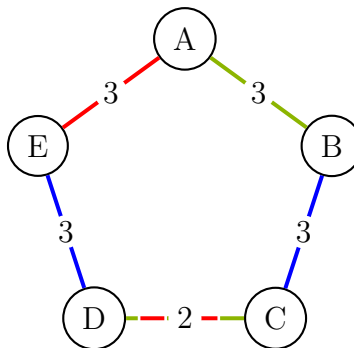


Figure 5.2: A discrete polyamorous scheduling instance which is solvable only by assigning multiple colours to the CD edge.

5.4 Approximation Algorithms

In this section, we present two efficient polynomial-time approximation algorithms for Optimisation Polyamorous Scheduling, thereby proving Theorems 5.1.3 and 5.1.4. Throughout this section, we assume a fixed instance $\mathcal{O} = (P, R, g)$ of Optimisation Polyamorous Scheduling (OPS) is given.

5.4.1 Lower Bounds

We first collect a few simple lower bounds used in the analysis later; note that Section 5.5 has further lower bounds.

Lemma 5.4.1 Simple lower bound.

Given an OPS instance $\mathcal{O} = (P, R, g)$, set $g_{\min} = \min_{e \in R} g(e)$, $g_{\max} = \max_{e \in R} g(e)$, and $\Delta = \max_{p \in P} \deg(p)$. Any periodic schedule for \mathcal{O} has heat $h \geq \max\{\Delta \cdot g_{\min}, g_{\max}\}$.

Proof. The chromatic number χ_1 of the unweighted graph (P, R) is $\chi_1 \in \{\Delta, \Delta + 1\}$. This means that under any periodic schedule, some edge desires will grow to at least to $\chi_1 \cdot g_{\min} \geq \Delta \cdot g_{\min}$, since we cannot schedule any two edges incident to a degree- Δ node on the same day. Moreover, we cannot prevent the weight- g_{\max} edge from growing to heat g_{\max} . \square

A second observation is that the lower bound for any subset of the problem is also a lower bound for the problem as a whole:

Lemma 5.4.2 Subset bound.

Given two OPS instances $\mathcal{O} = (P, R, g)$ and $\mathcal{O}' = (P, R', g')$ with $R' \subseteq R$ and $g(e) = g'(e)$ for all $e \in R'$, i.e., \mathcal{O}' results from \mathcal{O} by dropping some edges. Assume further that any schedule for \mathcal{O}' has heat at least h^* . Then, any schedule for \mathcal{O} also has heat at least h^* .

Proof. Suppose there is a schedule \mathcal{S} for \mathcal{O} of heat $h' < h$. We obtain a schedule \mathcal{S}' for \mathcal{O}' by dropping all edges $e \notin R'$. (The resulting schedule may have empty days.) By construction, when using \mathcal{S}' to schedule \mathcal{O}' , all edges in R' will grow to the same heat as in \mathcal{O} under \mathcal{S} , and hence also to heat $h' < h$. \square

5.4.2 Approximation for Almost Equal Growth Rates

We first focus on a special case of OPS instances with “almost equal weights”, which is used as base for our main algorithm. Let the edge weights satisfy $g_{\min} \leq g(e) \leq g_{\max}$ for all $e \in R$. We will show that scheduling a proper edge colouring round-robin gives a $\frac{\Delta+1}{\Delta} \cdot \frac{g_{\max}}{g_{\min}}$ approximation algorithm, establishing Theorem 5.1.3:

Theorem 5.1.3 (restated). Colouring approximation.

For an Optimisation Polyamorous Scheduling instance $\mathcal{O} = (P, R, g)$ set $g_{\min} = \min_{e \in R} g(e)$, $g_{\max} = \max_{e \in R} g(e)$. Also, let Δ be the maximum degree in (P, R) and h^* be the heat of an optimal schedule. There is an algorithm that computes in polynomial time a schedule \mathcal{S} of heat h with $\frac{h}{h^*} \leq \min\left\{\frac{\Delta+1}{\Delta} \cdot \frac{g_{\max}}{g_{\min}}, \Delta + 1\right\}$.

Proof of Theorem 5.1.3. We compute a proper edge colouring for (P, R) with $\Delta + 1$ colours using the algorithm from [MG92] and schedule these $\Delta + 1$ matchings in a round-robin schedule. No edge desire will grow higher than $(\Delta + 1) \cdot g_{\max}$ in this schedule. Lemma 5.4.1

shows that $\text{OPT} \geq \max\{\Delta \cdot g_{\min}, g_{\max}\}$. The edge-colouring schedule is thus never more than a $\min\{\frac{\Delta+1}{\Delta} \cdot \frac{g_{\max}}{g_{\min}}, \Delta + 1\}$ factor worse than OPT. \square

5.4.3 Layering Algorithm

The colouring-based algorithm from Theorem 5.1.3 can be arbitrarily bad if desire growth rates are vastly different and Δ is large. For these cases, a more sophisticated algorithm achieves a much better guarantee (Theorem 5.1.4). The algorithm consists of 3 steps:

1. breaking the graph into layers (by edge growth rates),
2. solving each layer using Theorem 5.1.3, and
3. interleaving the layer schedules into an overall schedule.

Let L be a parameter to be chosen later. We define layers of $\mathcal{O} = (P, R, g)$ as follows. For $i = 0, \dots, L - 1$, set $\mathcal{O}_i = (P, R_i, g)$ where

$$R_i = \left\{ e \in R : \frac{g_{\max}}{2^{i+1}} < g(e) \leq \frac{g_{\max}}{2^i} \right\}.$$

Moreover, $\mathcal{O}_L = (P, R_L, g)$ with $R_L = \left\{ e \in R : g(e) \leq \frac{g_{\max}}{2^L} \right\}$.

Denote by Δ_i , for $i = 0, \dots, L$, the maximal degree in (P, R_i) . Let \mathcal{S}_i be the round-robin- $(\Delta_i + 1)$ -colouring schedule from Theorem 5.1.3 applied on the OPS instance \mathcal{O}_i . If run in isolation on \mathcal{O}_i , schedule \mathcal{S}_i has heat $h_i \leq (\Delta_i + 1)g_{\max}/2^i \leq (\Delta + 1)g_{\max}/2^i$ by the same argument as in Section 5.4.2. Moreover, for $i < L$, \mathcal{S}_i is a $2^{\frac{\Delta_i+1}{\Delta_i}}$ -approximation (on \mathcal{O}_i in isolation); for $i = L$, we can only guarantee a $(\Delta_L + 1)$ -approximation.

To obtain an overall schedule \mathcal{S} for \mathcal{O} , we schedule the $L + 1$ layers in round-robin fashion, and within each layer's allocated days, we advance through its schedule as before, i.e., $\mathcal{S}(t) = \mathcal{S}_{(t \bmod (L+1))}(\lfloor t/(L+1) \rfloor)$. Any advance in layer i is now delayed by a factor $(L + 1)$. Hence \mathcal{S} achieves heat at most

$$\bar{h} = \max_{i \in [0..L]} (L + 1) \cdot h_i \leq \max_{i \in [0..L]} (L + 1)(\Delta_i + 1) \cdot \frac{g_{\max}}{2^i}$$

Using Lemma 5.4.2 on the layers and Lemma 5.4.1, we obtain a lower bound for OPT of

$$\underline{h} = \max \left\{ \max_{i \in [0..L-1]} \Delta_i \cdot \frac{g_{\max}}{2^{i+1}}, g_{\max} \right\}$$

We now distinguish two cases for whether the maximum in \bar{h} is attained for an $i < L$ or for $i = L$. First suppose $\bar{h} = (L + 1)(\Delta_i + 1)g_{\max}/2^i$ for some $i < L$. Since we also have $\underline{h} \geq \Delta_i \cdot g_{\max}/2^{i+1}$, we obtain an approximation ratio of $2(L + 1)\frac{\Delta_i+1}{\Delta_i} \leq 3(L + 1)$ overall in this case. Here, we assume that $\Delta_i \geq 2$; otherwise we have only monogamous couples in this layer and scheduling is trivial, giving $h_i = \Delta_i \cdot g_{\max}/2^i$.

For the other case, namely $\bar{h} = (L + 1)(\Delta_L + 1)g_{\max}/2^L > (L + 1) \cdot (\Delta_i + 1)g_{\max}/2^i$ for all $i < L$, we do not have lower bounds on the edge growth rates. But we still know $\underline{h} \geq g_{\max}$, so we obtain a $(L + 1)(\Delta_L + 1)/2^L$ -approximation overall in this case.

Equating the two approximation ratios suggests to choose L such that $L \approx \lg(\Delta_L + 1) - \lg 3$; with $L = \lceil \lg(\Delta + 1) - \lg 3 \rceil$ and using $\Delta_L \leq \Delta$, we obtain an overall approximation ratio of at most $3(L + 1) \leq 3\lceil \lg(\Delta + 1) \rceil \leq 3\lceil \lg n \rceil$. This concludes the proof of Theorem 5.1.4.

5.5 Fractional Polyamorous Scheduling

In this section, we generalize the notion of density from Pinwheel Scheduling for the Polyamorous Scheduling Problem. For that, we consider the dual of the linear program corresponding to a fractional variant of Polyamorous Scheduling.

5.5.1 Linear Programs for Polyamorous Scheduling

In the fractional Polyamorous Scheduling problem, instead of committing to a single matching \mathcal{M} in (P, R) each day, we are allowed to devote an arbitrary *fraction* $y_{\mathcal{M}} \in [0, 1]$ of our day to \mathcal{M} , but then switch to other matchings without cost or delay for the rest of the day (a simple form of scheduling with preemption). The heat of a fractional schedule is again defined as $\max_{e \in R} r(e)g(e)$, but the recurrence time $r(e)$ now is the maximal time in \mathcal{S} before the fraction of days devoted to matchings containing e sum to at least 1. (For a non-preemptive schedule with one matching per day, this coincides with the definition from Section 3.2.3.)

Schedules for the fractional problem are substantially easier because there is no need to have different fractions $y_{\mathcal{M}}$ for different days: the schedule obtained by always using the average fraction of time spent on each matching yields the same recurrence times. We can therefore assume without loss of generality that our schedule is given by $\mathcal{S} = \mathcal{S}(\{y_{\mathcal{M}}\}_{\mathcal{M} \in \mathbf{M}})$, with $y_{\mathcal{M}} \in [0, 1]$ and $\sum_{\mathcal{M} \in \mathbf{M}} y_{\mathcal{M}} = 1$. \mathcal{S} schedules the matchings in some arbitrary fixed order, each day devoting the same $y_{\mathcal{M}}$ fraction of the day to \mathcal{M} . Then, recurrence times are

simply given by $r_S(e) = 1 / \sum_{\mathcal{M} \in \mathbf{M}: e \in \mathcal{M}} y_{\mathcal{M}}$.

With these simplifications, we can state the fractional relaxation of Optimisation Polyamorous Scheduling instance $\mathcal{O} = (P, R, g)$ as the following optimisation problem:

$$\min \quad \bar{h} \tag{5.1}$$

$$\text{s. t.} \quad \sum_{\mathcal{M} \in \mathbf{M}} y_{\mathcal{M}} \leq 1 \tag{5.2}$$

$$\frac{1}{\sum_{\mathcal{M} \in \mathbf{M}: e \in \mathcal{M}} y_{\mathcal{M}}} \cdot g_e \leq \bar{h} \quad \forall e \in R \tag{5.3}$$

$$y_{\mathcal{M}} \in [0, 1] \quad \forall \mathcal{M} \in \mathbf{M} \tag{5.4}$$

Substituting $\bar{h} = 1/\ell$, this is equivalent to the following linear program (LP):

$$\max \quad \ell \tag{5.5}$$

$$\text{s. t.} \quad \sum_{\mathcal{M} \in \mathbf{M}} y_{\mathcal{M}} \leq 1 \tag{5.6}$$

$$\frac{1}{g_e} \sum_{\mathcal{M} \in \mathbf{M}: e \in \mathcal{M}} y_{\mathcal{M}} \geq \ell \quad \forall e \in R \tag{5.7}$$

$$y_{\mathcal{M}} \geq 0 \quad \forall \mathcal{M} \in \mathbf{M} \tag{5.8}$$

The optimal objective value ℓ^* of this LP gives $\bar{h}^* = 1/\ell^*$, the optimal fractional heat.

Lemma 5.5.1 Fractional lower bound.

Consider an OPS instance $\mathcal{O} = (P, R, g)$ with optimal heat h^* and let $\bar{h}^* = 1/\ell^*$ where ℓ^* is the optimal objective value of the fractional-problem LP from Equations (5.5 – 5.8). Then $\bar{h}^* \leq h^*$.

Proof. We use the same approach as in [Cic+19, §3]: For any schedule \mathcal{S} , $h(\mathcal{S})$ is at least the heat $h_T(\mathcal{S})$ obtained during the first T days only, which in turn is at least $\max_e g(e) \cdot \bar{r}(e)$ for $\bar{r}(e)$ the *average* recurrence time of edge e during the first T days. A basic calculation shows that for the fractions $y_{\mathcal{M}}$ of time spent on matching \mathcal{M} during the first T days there exists a value $1/\ell = h(\mathcal{S})(1 - o(T))$, so that we obtain a feasible solution of the LP (5.5). Hence $1/\ell^* \leq 1/\ell = h(\mathcal{S})(1 - o(T))$. Since these inequalities hold simultaneously for all T , taking the limit as $T \rightarrow \infty$, we obtain $1/\ell^* = \bar{h}^* \leq h(\mathcal{S})$. \square

The immediate usefulness of Lemma 5.5.1 is limited since the number of matchings can be exponential in n .

Remark 5.5.2 Randomized-rounding approximation?

One could try to use this LP as the basis of a randomized-rounding approximation algorithm, but since it is not clear how to obtain an efficient algorithm from that, we do not pursue this route here. The simple route taken in [Cic+19] cannot achieve an approximation ratio better than $O(\log n)$, so Theorem 5.1.4 already provides an equally good deterministic algorithm.

We therefore proceed to the dual LP of Equations (5.5) to (5.8):

$$\min \quad x \tag{5.9}$$

$$\text{s. t.} \quad \sum_{e \in R} z_e \geq 1 \tag{5.10}$$

$$\sum_{e \in \mathcal{M}} \frac{z_e}{g_e} \leq x \quad \forall \mathcal{M} \in \mathbf{M} \tag{5.11}$$

$$z_e \geq 0 \quad \forall e \in R \tag{5.12}$$

While still exponentially large and thus not easy to solve exactly, the dual LP yields the versatile result from Theorem 5.1.5:

Theorem 5.1.5 (restated). Fractional lower bound.

Let $\mathcal{O} = (P, R, g)$ be an OPS instance with optimal heat h^ . For any set of values $z_e \in [0, 1]$, for $e \in R$, with $\sum_{e \in R} z_e = 1$, we have*

$$h^* \geq \bar{h}(z) = \frac{1}{\max_{\mathcal{M} \in \mathbf{M}} \sum_{e \in \mathcal{M}} \frac{z_e}{g(e)}}$$

where \mathbf{M} is the set of all inclusion-maximal matchings in (P, R) . The poly density of \mathcal{O} is \bar{h}^ , the largest value of $\bar{h}(z)$ over all feasible z .*

Proof of Theorem 5.1.5. Using the given z_e and $x = \max_{\mathcal{M} \in \mathbf{M}} \sum_{e \in \mathcal{M}} \frac{z_e}{g(e)}$, we fulfil all constraints of (5.9). The optimal objective value x^* is hence $x^* \leq x$. By the duality of LPs, we have $x^* \geq \ell^*$ for ℓ^* the optimal objective value of (5.5). Together with Lemma 5.5.1, this means $h^* \geq \bar{h}^* = 1/\ell^* \geq 1/x^* \geq 1/x$. \square

5.5.2 Poly Density

Theorem 5.1.5 gives a more explicit way to compute the *poly density* \bar{h}^* than the primal LP, but it is unclear whether it can be computed exactly in polynomial time. Given the more intricate global structure of Polyamorous Scheduling, \bar{h}^* is necessarily more complicated than the density of Pinwheel Scheduling. A particularly interesting open problem for Polyamorous Scheduling is whether a sufficiently low poly density implies the existence of a valid (integral) schedule.

Specific choices for z_e in Theorem 5.1.5 yield several known bounds:

- Setting $z_e = g_e/G$ for $G = \sum_{e \in R} g_e$ yields Corollary 5.1.6.
- Fix any subset $R' \subseteq R$. Now set $z_e = g_e/C$ if $e \in R'$ and 0 otherwise, where $C = \sum_{e \in R'} g_e$. The maximum from Theorem 5.1.5 then simplifies to $\frac{1}{C} \max_{\mathcal{M} \in \mathbf{M}} |\mathcal{M} \cap R'|$, so

$$h^* \geq \frac{\sum_{e \in R'} g_e}{\max_{\mathcal{M} \in \mathbf{M}} |\mathcal{M} \cap R'|}.$$

- An immediate application of that observation with R' being all edges incident at a person $p \in P$ yields the BGT bound:

Corollary 5.5.3 Bamboo lower bound.

Given an OPS instance (P, R, g) and $p \in P$ with $g_1 \geq \dots \geq g_d$ the desire growth rates for edges incident at p . Set $G_p = g_1 + \dots + g_d$. Any periodic schedule for (P, R, g) has heat at least G_p .

Remark 5.5.4 Better general bounds?.

For the general case, it seems challenging to obtain other such simple bounds. The bound of G/m is easy to justify without the linear programs by a “preservation-of-mass argument”: Assume a schedule \mathcal{S} could achieve a heat $h < G/m$. Every day, the overall polycule’s desire grows by G , and \mathcal{S} can schedule at most m pairs to meet, whose desire is reset to 0 from some value $\leq h$. Every day, \mathcal{S} thus removes only a total of $\leq mh < G$ desire units from the polycule, whereas the overall growth is G , a contradiction to the heat remaining bounded.

Note that the bound of G/m is tight for some instances, so we cannot hope for a strict lower bound. On the other hand, the example from Figure 5.3 demonstrates that it can also be arbitrarily far from h^ .*

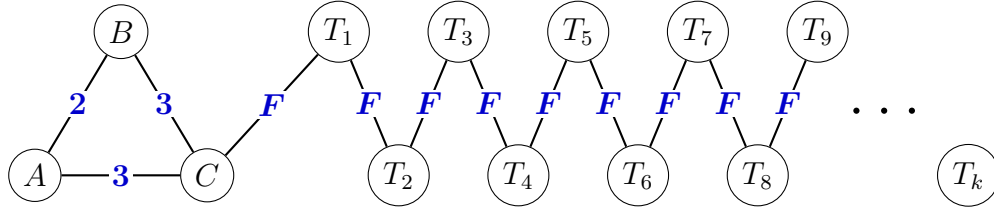


Figure 5.3: The tadpole family of DPS instances, defined for parameters $k \geq 0$ (tail length) and $F \geq 3$ (tail frequency). The total growth rate is $G = \frac{1}{2} + \frac{2}{3} + k \cdot 1/F = \frac{7}{6} + \frac{k}{F}$ and the size of a maximum matching is $m = 1 + \lfloor (k + 1)/2 \rfloor$.

Figure 5.3 shows the tadpole family of instances demonstrating the power of the dual-LP approach and Theorem 5.1.5. All DPS tadpoles (as shown in the figure) are infeasible since already the triangle $A-B-C$ does not admit a schedule obeying the given frequencies. The corresponding OPS instances (as given by Lemma 3.2.4) with $g(e) = 1/f(e)$ thus have $h^* > 1$; indeed $h^* = 4/3$ if $F \geq 2$. However, the simple lower bounds or local arguments do not detect this: (a) All local Pinwheel Scheduling instances (any person plus their neighbours) are feasible. (b) The mass-preservation bound (Corollary 5.1.6) is $G/m < 1$ for $k \geq 1$. Indeed, setting $F = k$ and letting $k \rightarrow \infty$, $G/m = O(1/k)$, gives an arbitrarily large gap to h^* . By contrast, consider the LP fractional lower bound. One can show that $\bar{h}^* = \frac{7}{6} > 1$ for any $k \geq 1$ and $F \geq 2$, so Theorem 5.1.5 correctly detects the infeasibility in this example.

Remark 5.5.5 Better Pinwheel density via dual LPs?.

Since Polyamorous Scheduling is a generalization of Pinwheel Scheduling resp. Bamboo Garden Trimming, we can apply Theorem 5.1.5 also to these problems. However, for this special case, the optimal objective value of the dual LPs is always $x^ = \ell^* = 1/G$ for G the sum of the growth rates, so we only obtain the trivial “biomass” lower bound of G for Bamboo Garden Trimming resp. the density ≤ 1 necessary condition for Pinwheel Scheduling. The more complicated structure of matchings in non-star graphs makes fractional lower bounds in Polyamorous Scheduling much richer and more powerful.*

* * * * *

Chapter 6

Simple Approximation Algorithms for Polyamorous Scheduling

Outline This chapter lays out the findings of our second foray into the study of Polyamorous Scheduling: “Simple Approximation Algorithms for Polyamorous Scheduling” [Bik+25]. As before, we begin by formally defining key results in Section 6.1, before addressing them in detail in the following three sections:

Section 6.2 studies Optimisation Polyamorous Scheduling, showing that the “layering algorithm” from the previous chapter is best possible up to constant factors in the class of methods that schedule a series of *disjoint* matchings. We also show that constant-factor approximations are possible outside of this class by introducing a simple generalization of the *Reduce-Fastest*(x) strategy from Bamboo Garden Trimming which achieves a 5.24-approximation. Finally, we introduce a technique for constructing polynomial schedules for both Optimisation Polyamorous Scheduling and Bamboo Garden Trimming.

Section 6.3 revisits “poly density”: providing improved upper and lower bounds, showing that poly density is a true generalization of the density of Pinwheel Scheduling, and proving that Optimisation Polyamorous Scheduling instances with $d \leq \frac{1}{4}$ are schedulable.

Section 6.4 returns to inapproximability: showing that a $\frac{13}{12}$ approximation for Bipartite Decision Polyamorous Scheduling is NP-hard via a reduction from 3SAT.

6.1 Results

Despite being an interesting combinatorial optimisation problem, Polyamorous Scheduling has only been formally studied in [GSW24]. This paper introduced several approximations for Optimisation Polyamorous Scheduling, the notion of poly density, and the first inapproximability result for any Periodic Scheduling problem; in this chapter we introduce better approximations, bound and exploit poly density, and extend this inapproximability result to *Bipartite Decision Polyamorous Scheduling*.

Approximating OPS An Optimisation Polyamorous Scheduling instance $\mathcal{O} = (P, R, g)$ consists of an undirected graph (P, R) of people P connected by relationships R , along with a desire growth rate $g_e : R \mapsto \mathbb{R}_{>0}$ for each relationship $e \in R$. The *current heat* of an edge is the total desire grown since the last time it was scheduled; every day, the scheduler must choose a set of all-day meetings such that each person $v \in P$ is not double-booked, with the goal of minimising the highest heat ever experienced by any couple. Optimisation Polyamorous Scheduling is formally defined in Section 3.2.2.

The previous chapter introduced the “layering algorithm”, a $3 \lg(\Delta + 1)$ -approximation for Optimisation Polyamorous Scheduling, where Δ is the highest number of partners of any person in the input. In Section 6.2.1, we will show that this layering algorithm is best possible up to constant factors in the class of methods that work by scheduling relationships as series of *disjoint* matchings.

In spite of this, constant-factor approximations remain possible; in Section 6.2.2, we define a simple generalization of the Reduce-Fastest(x) algorithm introduced by [Gas+17] and show that it achieves a 5.24-approximation. Broadly speaking, Reduce-Fastest(x) is an online scheduling method which chooses a matching to schedule each day: the matching with the fastest-growing edges whose current heat is above some user-selected threshold x . Section 6.2.2 presents a simple analysis showing that Reduce-Fastest(4) gives a 6-approximation, then a more in-depth analysis showing that Reduce-Fastest($2 + \frac{2\sqrt{5}}{5} \approx 2.89$) gives a $3 + \sqrt{5} \approx 5.24$ approximation. These both substantially improve on the $O(\log \Delta)$ approximation ratio in [GSW24]. Reduce-Fastest(x) for Optimisation Polyamorous Scheduling is formally defined in Section 6.2.

It is well established that schedules with exponentially long periods are often necessary when solving certain Optimisation Polyamorous Scheduling instances exactly; in fact, whether any feasible instance of Pinwheel Scheduling admits a succinctly encodable schedule remains an open problem. We show that the same is not true for *approximate* solutions. In Section 6.2.3, we introduce a method that takes an arbitrary Optimisation Polyamorous Scheduling schedule and constructs a new schedule with polynomial length and a maximum heat at most four times that of the original schedule, an alternative to accepting superpolynomial schedules for constant-factor approximations.

Poly Density The previous chapter introduced the notion of “poly density” for both OPS and DPS but this, as originally defined, is based on a linear program of exponential size and the task of computing the poly density of an instance is left unsolved: it therefore fails to provide an efficiently testable sufficient criterion.

In Section 6.3.1, we show that the poly density for OPS is narrowly sandwiched by the maximum personal growth rate of any person in the polycule: $G^* = \max_{v \in P} \sum_{e \in R: v \in e} g(e)$.

Theorem 6.1.1 Poly Density Approximation.

For an OPS instance with poly density \bar{h}^ and maximum personal growth rate G^* , we have $G^* \leq \bar{h}^* < \frac{3}{2}G^*$.*

This shows that, from the perspective of constant-factor approximations, the cost of any OPS instance is dominated by a single BGT instance embedded in it – the single busiest person in the polycule¹. This insight was the key ingredient to designing and analyzing a generalization of the Reduce-Fastest(x) approximation [Gas+17] for Polyamorous scheduling.

We then turn our attention to the poly density of Decision Polyamorous Scheduling. Each instance of DPS, or *DPS polycule* $\mathcal{D} = (P, R, f)$ consists of a set P of people, a set R of relationships, and a set f of “meetup frequencies” f_e . The goal is to either find a schedule of pairwise meetings such that each couple $e = \{i, j\}$ meets at least every f_e days or to report that no such schedule exists (subject to the constraint that each person can attend at most one meeting per day). A DPS polycule can naturally be represented as a graph of people with the edges representing their relationships; since each person can attend at most one meeting per day, the edges scheduled on any given day must form a matching in this graph. DPS is formally defined in Section 3.2.1.

¹A simple rule of thumb for practical applications: if the identification of this individual is non-trivial, it’s probably you.

As discussed in Section 2.3, Pinwheel Scheduling is a special case of DPS where the polycule is a star graph. In Section 6.3.2, we show that the poly density of DPS for star graphs is equivalent to the density of Pinwheel Scheduling, and hence that poly density generalizes the density of Pinwheel Scheduling. The recent proof of the long-standing $\frac{5}{6}$ -density conjecture of Pinwheel Scheduling by [Kaw24] admits a strong sufficient criterion to certify feasibility for this class of polycules; we produce a similar result by proving Theorem 6.1.2, affirmatively answering Question 5.1.7 from the previous chapter.

Theorem 6.1.2 Density Threshold.

Any DPS instance $\mathcal{D} = (P, R, f)$ with poly density $d \leq \frac{1}{4}$ is schedulable.

Inapproximability A well-motivated special case of Polyamorous Scheduling considers a *bipartite* graph of relationships (Bipartite Polyamorous Scheduling). In the original motivation for scheduling meetings of polyamorous people, bipartite polycules correspond to a group of heterosexuals; it also models Periodic Scheduling scenarios with two explicit classes or hierarchies, such as the members of a sports team who each need to regularly train on a shared set of machines or with a shared set of trainers. Apart from applications, there are strong indications in Chapter 5 that bipartite instances may be more well-behaved: every presented example of a decision polycule that was non-trivial to show to be infeasible had an odd cycle. Moreover, the hardness-of-approximation proof in Section 5.3 reduces the Chromatic Index problem to unweighted DPS instances; the latter is trivial on bipartite graphs, since they can always be Δ -edge coloured. We show that bipartiteness does indeed *not* make the problem easier: it remains NP-hard to find even approximate solutions to Bipartite Polyamorous scheduling. As a consequence, the hardness of Polyamorous Scheduling is intrinsically linked to the frequencies resp. weights of edges, not simply the embedded edge colouring problem.

In Section 6.4 we strengthen the inapproximability result introduced by Section 5.3 by proving that it holds even in the bipartite case:

Theorem 6.1.3 SAT Hardness of Approximation.

Unless $P = NP$, there is no polynomial-time $(1+\delta)$ -approximation algorithm for the Bipartite Optimisation Polyamorous scheduling problem for any $\delta < \frac{1}{12}$.

We prove this by constructing a bipartite DPS polycule \mathcal{D}_φ from an arbitrary 3SAT instance φ such that \mathcal{D}_φ is schedulable iff φ is satisfiable.

6.2 Approximating OPS

This section will examine approximation algorithms for OPS in 3 ways: Section 6.2.1 considers several lower bounds on OPS, Section 6.2.2 evaluates Reduce-Fastest(x) as an approximation algorithm for OPS, and Section 6.2.3 introduces a method for constructing polynomial schedules for OPS and BGT.

As OPS generalises BGT, with instances on star-graphs being fully equivalent, these problems share both features and algorithms. Particularly of interest here is the *Reduce-Fastest(x)* heuristic, which we generalise here and will use in multiple parts of this section:

Definition 6.2.1 Reduce-Fastest(x) for OPS.

Begin with an OPS instance $\mathcal{O} = (P, R, g)$ and relabel to sort edges $e \in R$ by decreasing growth rates $g(e)$, breaking ties arbitrarily. Construct an online schedule using the fixed parameter x by scheduling a greedy matching \mathcal{M} each day, built the following way:

For all edges $e = \{u, v\} \in R$ in order of decreasing growth rate, add e to \mathcal{M} iff:

- *the current heat of e is at least $x \cdot G^*$, and*
- *no edge incident at u or v is already in \mathcal{M} .*

Note that on a star graph, each matching \mathcal{M} will contain a single edge, so our OPS Reduce-Fastest(x) behaves like the BGT Reduce-Fastest(x) heuristic introduced by [Gås+17] on such instances.

6.2.1 Lower Bounds

We will begin by defining Disjoint Matching Algorithms and demonstrating them to be a dead end, then show a simple but general lower bound from the local BGT instance, then a lower bound for the Reduce-Fastest(x) algorithm.

Disjoint Matching Algorithms

In addition to introducing OPS, the previous chapter presented multiple proofs that OPS is NP-hard, several lower bounds for it, and the *Layering Algorithm* – an $\mathcal{O}(\log \Delta)$ -approximation for OPS.

This algorithm partitions edges into layers based on their growth rates, uses a fixed round-robin schedule for each layer, and then interleaves these schedules into another fixed round-robin schedule. Further details are not needed here, only that each edge only appears in a

single matching, and thus, all matchings in schedules produced by the Layering Algorithm are *disjoint*.

We define Disjoint Matching Algorithms as algorithms whose schedules contain only disjoint matchings, and will show that such algorithms obey the following:

Theorem 6.2.2.

The approximation ratio of any Disjoint Matching Algorithm for Optimisation Polyamorous Scheduling is $\Omega(\log n)$.

Proof. Consider an OPS instance $\mathcal{O}^* = (P, R, g)$ consisting of d disjoint stars $*_1, \dots, *_i, \dots, *_d$, such that star i has i edges and $g = \frac{1}{i}$ for all edges in star i . The first four such stars are shown in Figure 6.1. The optimal heat for \mathcal{O}^* is 1, and can be obtained by scheduling each star independently in a round-robin fashion.

Any set of disjoint matchings \mathbf{M} which covers all edges $e \in R$ will consist of at least d matchings due to the largest star $*_d$. Let g_M^* be the maximum growth rate of any edge in matching $\mathcal{M}_i \in \mathbf{M}$, i.e. $g_i^* := \max_{e \in \mathcal{M}_i} g(e)$ and $g^* = \max_{i \in \mathbf{M}} g_i^*$. In the given disjoint stars, there is at least one matching with $g_i^* = 1$, at least two matchings with $g_i^* \geq 1/2$, at least three with $g_i^* \geq 1/3$, and so on. Once we commit to scheduling the matchings as atomic objects, the problem has become equivalent to a BGT instance with d bamboos of growth rates $g_1^* \leq g_2^* \leq \dots \leq g_d^*$. The optimal height for any schedule of this instance is at least $\sum_{i=1}^d g_i^* \geq \sum_{i=1}^{\Delta} 1/i \sim \ln \Delta$. Since $n = \sum_{i=1}^{\Delta} (2i - 1) = \Delta^2$, any such schedule for \mathcal{O}^* thus has heat $\Omega(\log \Delta) = \Omega(\log n)$, compared to the optimal heat of 1. \square

This means that, in its “league” of algorithms that first divide the graph into a set of disjoint matchings and then schedule these matchings as atomic objects, the Layering Algorithm’s approximation ratio is best possible up to constant factors. To do much better than this, different kinds of approach will be needed.

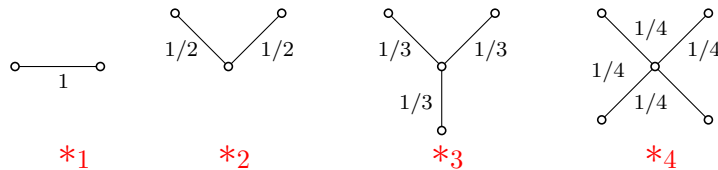


Figure 6.1: The first four disjoint stars in the OPS instance \mathcal{O}^* used to prove the worst-case lower bound of $\Omega(\log n)$ for the maximum heat reached by any Disjoint Matching Algorithm for OPS instances.

Lower Bound due to the Local BGT Instance

Recall that the schedule for every person in an OPS polycule must solve the BGT instance consisting of all edges $e \in R$ connected to that person. These BGT instances can be used to obtain a natural lower bound for OPS. To formalise this, consider a vertex $v \in P$, recalling that the personal growth rate G_v for person $v \in P$ is $G_v := \sum_{e \in R: v \in e} g(e)$ and that the maximum personal growth rate across all $v \in P$ is $G^* := \max_{v \in P} G_v$. We claim that G^* is a natural lower bound on the maximum heat of any schedule of an OPS instance:

Theorem 6.2.3.

For any OPS instance $\mathcal{O} = (P, R, g)$, G^ is a lower bound on the maximum heat of any schedule for \mathcal{O} .*

Proof. This can be easily proved by contradiction. Let $v^* = \arg \max_{v \in P} \sum_{e: v \in e} g(e)$ be the vertex that corresponds to G^* . Assume that there exists a periodic schedule that keeps the heat of all edges incident on v^* to some heat $h < G^*$. In the neighbourhood of v^* , $\mathcal{N}(v^*)$, we can only schedule one edge on any given day. After $\left\lfloor \frac{|\mathcal{N}(v^*)| \cdot h}{G^* - h} \right\rfloor + 1$ days, the heat of at least one edge must be greater than h , a contradiction. \square

Lower Bound for Reduce-Fastest(x)

The next section will analyse the approximation ratio of the Reduce-Fastest(x) heuristic, but first, we will demonstrate a lower bound for this heuristic. The proof concerns a fully connected OPS polycule which is fair in the sense that all growth rates are the same, but whose edges are ordered adversarially with respect to a single unfortunate individual.

Theorem 6.2.4.

When applied to normalised OPS instances $\mathcal{O} = (P, R, g)$, Reduce-Fastest(x) with $x \geq 2$ can produce schedules with heat $h \geq x + 2 - \frac{1}{n-1}$.

Proof. For some odd n , consider a normalised OPS instance $\mathcal{O} = (P, R, g)$ where (P, R) is the complete graph K_n on n vertices, and has uniform growth rates of $\frac{1}{n-1}$. Reduce-Fastest(x) will not schedule edges until they reach the threshold heat x , and will then schedule edges according to growth rate (all tied), then to an arbitrary choice of their ordering to break ties. To construct an adversarial ordering of edges, we will consider two subgraphs: the arbitrary induced subgraph K_{n-1} of P , and the subgraph of edges in K_n , but not K_{n-1} , called $K_n - R(K_{n-1})$.

Since n is odd, $n - 1$ is even, and K_{n-1} is 1-factorizable, i.e. edges of K_{n-1} can be decomposed into $n - 2$ disjoint matchings, each covering all vertices in K_{n-1} . As only one vertex appears in K_n but not K_{n-1} , $K_n - R(K_{n-1})$ requires a further $n - 1$ matchings to cover, as each matching can only contain a single edge from $K_n - R(K_{n-1})$.

So, for some initial ordering of edges $e \in R$, Reduce-Fastest(x) will first wait for all edges to grow to heat x , then schedule matchings in K_{n-1} , then matchings covering $K_n - R(K_{n-1})$. This means that the last edge is scheduled $(n - 2 + n - 1) = 2n - 3$ days later. This last edge thus starts with heat at least x and grows a further heat of $(2n - 3) \cdot \frac{1}{n-1} = 2 - \frac{1}{n-1}$ before being scheduled.

Note that in this $2n - 3$ day period, the heat of each edge grows by $\frac{1}{n-1}$ per day, so no previously scheduled edge will reach the threshold of $x \geq 2$, and thus no edge will interfere with the scheduling sequence. \square

Remark 6.2.5.

Note that if Reduce-Fastest(x) is applied to the above graph with $x < 2$, the maximum heat will tend to infinity for an appropriately large n . This is because the edges which are scheduled first will cross the threshold x for a second time before the last edge is scheduled, causing this last edge to grow indefinitely.

6.2.2 Reduce-Fastest(x)

This section gives two analyses for the approximation ratio of the OPS version of Reduce-Fastest (x), defined by Definition 6.2.1.

Both analyses use a notion of *blocking*: Each day, Reduce-Fastest(x) constructs a matching \mathcal{M} by greedily selecting edges with heat at least G^*x according to their growth rates (with ties broken by some initial ordering). This means that for some edge e with heat $h \geq G^*x$ not to be added to \mathcal{M} , a higher priority edge containing one of the vertices in e must block e by being added to \mathcal{M} before e was considered.

6-Approximation Algorithm

In this section, we prove that Reduce-Fastest(4) is a 6-approximation for OPS by proving Theorem 6.2.6. This analysis is inspired by the similar proof of an $(x + 1)$ -approximation algorithm for the BGT problem in [Kus22].

Theorem 6.2.6.

Given a normalised OPS instance $\mathcal{O} = (P, R, g)$; for all $x \geq 4$, the maximum heat achieved by an edge in $\text{Reduce-Fastest}(x)$ is strictly less than $x + 2$.

Proof. Suppose for the sake of contradiction that some edge $e_i \in R$ achieves heat of at least $x + 2$ at time t_2 after most recently reaching heat x at time t_1 . This implies that on every day in the interval $[t_1, t_2)$, e_i was blocked by a higher-priority adjacent edge. Consider the sequence of these $t_2 - t_1$ blocking edges and denote the set of distinct edges in this sequence by \mathcal{S} . e_i could be blocked by two separate higher-priority edges at the same time – in this case, arbitrarily select one to include in the sequence. For all $e_j \in \mathcal{S}$, we denote by m_j the number of times that e_j appears in the sequence. For an edge to appear m_j times in our sequence, its growth rate must follow:

Claim 6.2.7.

For all $e_j \in \mathcal{S}$, we have $g(e_j) > m_j \cdot g(e_i)$.

Proof. Begin by considering the case of $m_j = 1$, which simply implies that $g(e_j) \geq g(e_i)$. This follows from the definition of $\text{Reduce-Fastest}(x)$ (Definition 6.2.1) – the edge with faster heat growth rate is scheduled earlier and hence blocks the edge e_i .

Given that edge e_j can have any initial heat, it could be scheduled as early as t_1 . After this, it must grow to heat x before each time it is scheduled again, $m_j - 1$ times total. Thus, to achieve $m_j \geq 2$, edge e_j has to grow a total heat of at least $x(m_j - 1)$ in the interval $\mathcal{T} = (t_1, t_2)$. In this time interval, e_i must grow less than 2 units of heat, as it starts with heat x and does not reach a heat of $x + 2$ until t_2 . Thus

$$\frac{2}{g(e_i)} > \mathcal{T} \geq \frac{x(m_j - 1)}{g(e_j)}$$

and

$$g(e_j) > x(m_j - 1) \frac{g(e_i)}{2} > 2(m_j - 1)g(e_i) > m_j g(e_i),$$

for $x \geq 4$ and $m_j \geq 2$. This completes the proof of the claim. □

Using Claim 6.2.7, we can argue from the number of blockers:

$$t_2 - t_1 \leq \sum_{e_j \in \mathcal{S}} m_j < \sum_{e_j \in \mathcal{S}} \frac{g(e_j)}{g(e_i)} = \frac{1}{g(e_i)} \sum_{e_j \in \mathcal{S}} g(e_j) \leq \frac{2}{g(e_i)}(1 - g(e_i)),$$

where the last inequality is a product of normalisation – since $G^* = 1$, for each edge $e_i = \{u, v\}$, we have $G_u \leq 1$ and $G_v \leq 1$.

Since the heat of e_i is less than $x + g(e_i)$ at t_1 , the heat level reached by t_2 is strictly less than

$$x + g(e_i) + (t_2 - t_1)g(e_i) \leq x + g(e_i) \left(1 + \frac{2}{g(e_i)} (1 - g(e_i)) \right) = x + 2 - g(e_i) < x + 2,$$

which is a contradiction. □

Corollary 6.2.8.

Reduce-Fastest(4) is a 6-approximation of OPS.

From Theorems 6.2.4 and 6.2.6, the following simple corollary follows:

Corollary 6.2.9.

Given an OPS instance $\mathcal{O} = (P, R, g)$, the $x + 2$ bound on the maximum heat achieved by Reduce-Fastest(x) is tight for $x \geq 4$ and large n .

5.24-Approximation Algorithm

In this section, we prove our best bound on Reduce-Fastest(x): Reduce-Fastest(2.89) is a 5.24 approximation for OPS. The analysis follows the related proof of approximation algorithm for the BGT problem in [Bil+22].

Theorem 6.2.10.

Given a normalised OPS instance $\mathcal{O} = (P, R, g)$; for all $x > 2$, the maximum heat achieved by an edge in Reduce-Fastest(x) is bounded by

$$\max \left\{ x + \frac{x^2}{4(x-2)}, x + \frac{1}{2} + \frac{x^2}{4(x-1)} \right\}.$$

Proof. Let \mathcal{S}_{rf} be the schedule produced when Reduce-Fastest(x) is applied to \mathcal{O} , $h = h(\mathcal{S}_{\text{rf}})$ be the heat of this schedule, and $e_i = \{u, v\}$ be an edge which reaches heat h on day T . Before reaching day T , the heat of e_j has grown for some period without being scheduled, reaching threshold x on day 0 in this period. As e_i was not scheduled in the interval $[0, T-1]$, for each day in this interval, there must have been some higher-priority edge adjacent to e_i which was scheduled instead; construct the *blocking sequence* \mathcal{S}_b from these T blocking edges.

On any day which has two separate blocking edges, arbitrarily choose one to add to \mathcal{S}_b and add the other to the *suppressed sequence* Q . Let N be the number of distinct edges in \mathcal{S}_b .

Let the *volume* V be the overall growth of e_i and its adjacent edges in the days of the interval $[0, T - 1]$. Because of normalisation, $G_u \leq 1$ and $G_v \leq 1$, so the heat of this set of edges grows by at most $2 - g(e_i)$ per day, and $V \leq T(2 - g(e_i))$.

For each edge $e_j \in \mathcal{S}_b$, call the first time it was scheduled in \mathcal{S}_b the *first-time scheduling* and further times *repeated schedulings*. The volume of heat associated with edge e_j is either removed by the first-time scheduling, removed by repeated scheduling, or is *leftover volume*: heat that remains after day $T - 1$.

In order to bound the amount of volume V_r removed by repeated schedulings, we first calculate the volume V_f that is accumulated before the first time each edge is scheduled in \mathcal{S}_b , then the volume V_l which is leftover after the last time each edge is scheduled in \mathcal{S}_b . Note that $g(e_j) \geq g(e_i)$ for each edge $e_j \in \mathcal{S}_b$. If e_j appears in \mathcal{S}_b for the first time on day d , then the volume removed on day d in addition to the total volume removed by the preceding schedulings of e_j in Q will be at least $(d + 1)g(e_j) \geq (d + 1)g(e_i)$. This is because $(d + 1)g(e_j)$ is the total growth of e_j up to and including the day d , all of which will be removed when e_j is scheduled. It follows that the total volume removed by all first-time schedulings in addition to the preceding schedulings of the corresponding edges in the suppressed sequence Q will be given by $V_f \geq \sum_{j=1}^N jg(e_i) = \frac{N(N+1)}{2} \cdot g(e_i)$, since exactly one edge is added to the sequence \mathcal{S}_b per day.

Moreover, if e_j appears in the sequence \mathcal{S}_b for its last time at day $T - 1 - d$, then the leftover volume of e_j in addition to the volume removed by the following schedulings of e_j in Q is at least $d' \cdot g(e_i)$. Therefore, for all N edges, the total volume left after the last scheduling of e_j in \mathcal{S}_b will be $V_l = \sum_{j=1}^N (j - 1)g(e_i) = \frac{N(N-1)}{2} \cdot g(e_i)$. Finally, the edge e_i is not scheduled in the interval $[0, T - 1]$ but does grow by exactly $Tg(e_i)$ during this time.

We can then bound the total volume removed by repeated schedulings of edges in \mathcal{S}_b as

$$\begin{aligned} V_r &\leq V - \left(\frac{N(N+1)}{2} + \frac{N(N-1)}{2} \right) g(e_i) - Tg(e_i) \\ &= V - N^2 \cdot g(e_i) - Tg(e_i) \\ &\leq T(2 - g(e_i)) - N^2g(e_i) - Tg(e_i) \\ &= 2T(1 - g(e_i)) - N^2g(e_i). \end{aligned}$$

Each of these removes heat at least x , so the number N_r of repeated schedulings is upper

bounded by $\frac{V_r}{x} = \frac{2T(1-g(e_i))-N^2g(e_i)}{x}$, which gives us the following bound on the total number of elements in the sequence:

$$|\mathcal{S}_b| = T = N + N_r \leq N + \frac{2T(1-g(e_i)) - N^2g(e_i)}{x},$$

which, with $x \geq 2$, implies

$$T \leq \frac{Nx - N^2g(e_i)}{2g(e_i) + x - 2}$$

Consider the minimum value of the right-hand side; as $g(e_i)$ and x are fixed, we can view it as a function of N : $R(N) = (Nx - N^2g(e_i))/(2g(e_i) + x - 2)$. The function $R(N)$ is a concave downward parabola that attains its maximum at $N = \frac{x}{2g(e_i)}$, giving us

$$T \leq R\left(\frac{x}{2g(e_i)}\right) = \frac{x^2}{4g(e_i)(2g(e_i) + x - 2)}$$

Using this upper bound to T , we now bound the overall growth of the edge e_i , i.e. the maximum heat $h(\mathcal{S}_{\text{rf}})$. At day $d = 0$, e_i has heat at most $x + g(e_i)$ by our choice of the time interval, and in the next T days it grows by $Tg(e_i)$. Hence:

$$h(\mathcal{S}_{\text{rf}}) \leq x + g(e_i) + Tg(e_i) \leq x + g(e_i) + \frac{x^2}{4(2g(e_i) + x - 2)}.$$

We now show that the claimed bound follows. We distinguish two cases; first assume $g(e_i) \leq \frac{1}{2}$. Viewed as a function of $g(e_i) \in [0, \frac{1}{2}]$, the upper bound on $h(\mathcal{S}_{\text{rf}})$ is a convex function since its second derivative w.r.t. $g(e_i)$ is

$$\frac{2x^2}{(2g(e_i) + x - 2)^3} > 0$$

(for $x > 2$). The maximum (worst upper bound) is thus attained at the extreme points, either $g(e_i) = 0$ or $g(e_i) = \frac{1}{2}$:

$$h(\mathcal{S}_{\text{rf}}) \leq \max\left\{x + \frac{x^2}{4(x-2)}, x + \frac{1}{2} + \frac{x^2}{4(x-1)}\right\}. \quad (6.1)$$

The second case, $g(e_i) > \frac{1}{2}$, is trivial because then there are no possible blocking edges, as both G_u and G_v are at most 1. The edge e_i is then always scheduled immediately upon reaching x , and, thus, $h(\mathcal{S}_{\text{rf}}) \leq x + 1 \leq x + \frac{1}{2} + \frac{x^2}{4(x-1)}$ for $x \geq 2$. \square

This bound on $h(\mathcal{S}_{\text{rf}})$ gives us the following corollaries:

Corollary 6.2.11.

For *Reduce-Fastest*(4), we have $h(\mathcal{S}_{\text{rf}}) \leq 6$.

The above corollary gives the upper bound on the maximum heat achieved by *Reduce-Fastest*(4), which also meets the bound from Theorem 6.2.6. Optimizing the bound in (6.1) allows to slightly improve upon this by changing to $x = 2 + \frac{2\sqrt{5}}{5}$.

Corollary 6.2.12.

Reduce-Fastest($2 + \frac{2\sqrt{5}}{5} \approx 2.89$) gives $h(\mathcal{S}_{\text{rf}}) \leq 3 + \sqrt{5} < 5.24$.

6.2.3 Length of Schedules

In this section, we will show how an arbitrary schedule for an OPS problem can be modified to produce a periodic schedule with a polynomial-sized period and a maximum heat of at most 4 times the maximum heat of the original schedule.

This proof uses an overloading of the definition of heat h such that $h(\mathcal{S}, R')$ is the heat achieved by applying some cyclic schedule \mathcal{S} to an OPS instance $\mathcal{O}' = (P, R', g)$ where $R' \subseteq R$, and $h(\mathcal{S}, e, d)$ is the heat of a single edge $e \in R$ that results from applying the same schedule \mathcal{S} such that d is the duration since e was last scheduled.

Theorem 6.2.13.

Consider an OPS instance $\mathcal{O} = (P, R, g)$. Given an **Arbitrary** Schedule \mathcal{S}_A for \mathcal{O} , we can construct a **Polynomial** Schedule \mathcal{S}_P with period $2\chi_1(P, R)$ and heat $h(\mathcal{S}_P) \leq 4h(\mathcal{S}_A)$.

Proof. Recall that $r_{\mathcal{S}}(e)$ is the recurrence time of an edge e in any schedule \mathcal{S} . First, consider a **Truncated** Schedule \mathcal{S}_T which consists of the first χ_1 characters of \mathcal{S}_A . Let the subset of edges $R_T \subseteq R$ consist of the edges $e \in R_T$ that appear in \mathcal{S}_T . Using these, prove the following claim:

Claim 6.2.14.

Given the schedules \mathcal{S}_A and \mathcal{S}_T , defined above, it holds that $h(\mathcal{S}_T, R_T) \leq 2h(\mathcal{S}_A)$.

Proof. For the first χ_1 days, \mathcal{S}_A and \mathcal{S}_T are identical, so both have heat at most $h(\mathcal{S}_A)$ within that period. Any edge $e \in R_T$ will have a heat $h(\mathcal{S}_T, e, d) = g(e) \cdot d$, where d is again the duration since e was last scheduled. The set of such durations D includes the duration d_b before the first meeting, the duration d_a after the last meeting, and the durations

$D_i = d_1, d_2 \dots d_i$ between meetings, so $D = \{d_b\} \cup \{d_a\} \cup D_i$. Note that the heats of edges grow on days when they are scheduled, so the sum of all such durations $\sum_{d \in D} d$ is the period χ_1 of \mathcal{S}_T .

For the first χ_1 days, \mathcal{S}_T does not repeat, so it holds that:

$$h(\mathcal{S}_A) \geq \max_{d \in D} (h(\mathcal{S}_T, e, d)) = g(e) \cdot \arg \max(\{d_b\} \cup \{d_a\} \cup D_i)$$

Thus:

$$g(e) \leq \frac{h(\mathcal{S}_A)}{\arg \max(\{d_b\} \cup \{d_a\} \cup D_i)} \quad (6.2)$$

After χ_1 days, \mathcal{S}_A and \mathcal{S}_T diverge, with \mathcal{S}_T repeating itself. The maximum heat of e in this period $h(\mathcal{S}_T, e)$ is therefore given by:

$$h(\mathcal{S}_T, e) = \max_{d \in D} (h(\mathcal{S}_T, e, d)) = g(e) \cdot \arg \max(\{(d_b + d_a)\} \cup D_i)$$

If $h(\mathcal{S}_T, e) > h(\mathcal{S}_A)$, it follows that $h(\mathcal{S}_T, e) = g(e) \cdot (d_b + d_a)$. Which, along with (6.2) shows that for all $e \in R_T$:

$$h(\mathcal{S}_T, e) \leq h(\mathcal{S}_A) \frac{d_b + d_a}{\arg \max(\{d_b\} \cup \{d_a\} \cup D_i)} \leq 2h(\mathcal{S}_A)$$

Hence, $h(\mathcal{S}_T, R_T) \leq 2h(\mathcal{S}_A)$. □

Let $C := \chi_1(P, R)$ be the number of colours required for edge colouring the graph (P, R) . Observe that for an OPS instance \mathcal{O} , one can always obtain a feasible schedule from a proper edge colouring $c : e \rightarrow [C]$ of the graph (P, R) , where $e \in R$, by taking any round-robin schedule of the C colours. More formally, we can define a **Coloured** Schedule $\mathcal{S}_C(t) := \{e \in R : c(e) \equiv t \pmod{C}\}$. As shown by [GSW24][Proposition 5.1], unweighted OPS is the same problem as edge colouring; thus, the round-robin schedule \mathcal{S}_C of the coloured edges will cover all of the edges.

Finally, we construct the Polynomial Schedule \mathcal{S}_P by interleaving the Truncated Schedule \mathcal{S}_T with the Coloured Schedule \mathcal{S}_C . This means that for any edge $e \in R_T$, the recurrence times obey $2r_{\mathcal{S}_T}(e) \leq r_{\mathcal{S}_P}(e)$. Now consider the complement to R_T , $R_T^c := R \setminus R_T$. As edges $e' \in R_T^c$ do not appear in the first χ_1 days of \mathcal{S}_A , their growth rate is bounded by $g(e') < \frac{h(\mathcal{S}_A)}{\chi_1}$, and they grow to heat $h(\mathcal{S}_C, R_T^c) < h(\mathcal{S}_A)$ over the course of \mathcal{S}_C . This completes the claim: for any edge $e' \in R_T^c$, we have $2r_{\mathcal{S}_C}(e') = r_{\mathcal{S}_P}(e')$. In other words, the recurrence time of any edge in \mathcal{S}_P is at most twice its recurrence time in either \mathcal{S}_T or \mathcal{S}_C .

This finally demonstrates that $h(\mathcal{S}_P) \leq 2h(\mathcal{S}_C) \leq 2h(\mathcal{S}_T, R_T)$ and therefore, from Claim 6.2.14, we have $h(\mathcal{S}_P) \leq 4h(\mathcal{S}_A)$. \square

Corollary 6.2.15.

As BGT is a special case of OPS on star graphs, Theorem 6.2.13 also applies to BGT.

Remark 6.2.16.

Only the first χ_1 days of \mathcal{S}_A appear in \mathcal{S}_P , and only edges $e' \in R_T^c$ need to appear in \mathcal{S}_C . Further works may apply this process to unsustainable intermediary schedules, taking advantage of these facts to reduce overall heat.

6.3 Poly Density

This section introduces two key results: Section 6.3.1 provides improved upper and lower bounds by proving Theorem 6.1.1, while Section 6.3.2 shows that poly density for DPS generalizes density for PWS and proves Theorem 6.1.2, showing that DPS instances with $d \leq \frac{1}{4}$ are schedulable.

6.3.1 Bounding Poly Density for OPS polycules

Recall the notion of poly density introduced in Section 5.5.2, which follows the same intuition as Pinwheel Scheduling: relax the requirement that one complete task must be done each day, instead allowing any fractions of a day to be spent on each task, and use this relaxation to identify classes of schedulable or unschedulable instances. Pinwheel Scheduling uses an implicit schedule of unit length in which each task i with frequency f_i has contribution $y_i \geq \frac{1}{f_i}$. Task i is then considered to be performed after $\frac{1}{y_i}$ days, with the density of the PWS instance given by the sum of these contributions $d = \sum_{i=1}^k \frac{1}{f_i}$. If $d > 1$ then these contributions cannot fit into a single day, and the instance clearly cannot be scheduled.

Section 5.5.2 applied similar reasoning to determine the poly density of Optimisation Polyamorous Scheduling problems, obtaining a lower bound on the optimal heat h^* . First, compose an optimisation problem using the same logic as Pinwheel Scheduling: construct a fractional schedule of unit length by assigning contribution $y_{\mathcal{M}} \in [0, 1]$ to each matching \mathcal{M} . The heat \bar{h} of the best such schedule provides a lower bound on h^* .

$$\min \bar{h} \tag{6.3}$$

$$\text{s.t.} \quad \sum_{\mathcal{M} \in \mathbf{M}} y_{\mathcal{M}} \leq 1 \tag{6.4}$$

$$\frac{g(e)}{\sum_{\mathcal{M} \in \mathbf{M}: e \in \mathcal{M}} y_{\mathcal{M}}} \leq \bar{h} \quad \forall e \in R \tag{6.5}$$

$$y_{\mathcal{M}} \in [0, 1] \tag{6.6}$$

We previously defined the poly density \bar{h}^* of an OPS polycule $\mathcal{O} = (P, R, g)$ as the optimal objective value of the fractional-schedule LP.² From the dual of this LP, we obtained a more explicit term for \bar{h}^* . A dual solution consists of weights z_e for edges $e \in R$ with $\sum_{e \in R} z_e = 1$. Intuitively these account for their growth rates g_e and the inclusion-maximal matchings $\mathcal{M} \in \mathbf{M}$ in which they appear. Any such weighting implies a lower bound $\bar{h}(z)$ on \bar{h}^* and hence on h^* , with the best choice of z yielding the poly density

$$\bar{h}^* = \max_{z \in \mathcal{Z}} \bar{h}(z), \quad \text{where } \bar{h}(z) = \min_{\mathcal{M} \in \mathbf{M}} \frac{1}{\sum_{e \in \mathcal{M}} \frac{z_e}{g(e)}}, \tag{6.7}$$

and \mathcal{Z} is the set of all weightings.

Recall from Section 3.2.2 that the maximum personal growth over some polycule $\mathcal{O} = (P, R, g)$ is given by $G^* = \max_{v \in P} \sum_{e \in R: v \in e} g(e)$. We now show that the poly density is within a constant factor of G^* , proving Theorem 6.1.1 in two parts: The lower bound by Lemma 6.3.1 and the upper bound by Lemma 6.3.3.

Lemma 6.3.1.

For any OPS instance $\mathcal{O} = (P, R, g)$, the poly density \bar{h}^ is bounded by $\bar{h}^* \geq G^*$.*

Proof. Let v^* be the vertex with the maximum local heat G^* , i.e.

$$v^* = \arg \max_{v \in P} \sum_{e \in R: v \in e} g(e).$$

Consider the weighting z^* with $z_e = \frac{g(e)}{G^*}$ when $v^* \in e$ and $z_e = 0$ otherwise. For z^* , it holds that

$$\max_{\mathcal{M} \in \mathbf{M}} \sum_{e \in \mathcal{M}} \frac{z_e}{g(e)} = \frac{1}{G^*}$$

and $\bar{h}(z) = G^*$. Since $\bar{h}^* \geq \bar{h}(z)$ for all weights z , the inequality holds. □

²A change of variables $\ell = 1/\bar{h}$ and taking reciprocals in the second constraint yields this LP.

Lemma 6.3.2.

For any OPS instance $\mathcal{O} = (P, R, g)$ with integer growth rates, $\bar{h}^* \leq \frac{3}{2}G^*$.

Proof. Begin by constructing an unweighted multigraph $\mathcal{O}' = (P, R')$ such that each edge $e \in R$ is replaced by a bundle of $g(e)$ parallel edges in R' , noting that the maximum degree Δ of \mathcal{O}' is exactly G^* . By [Sha49], we can colour edges in \mathcal{O}' using $C \leq \frac{3}{2}\Delta$ colours. Given that each colour induces a matching in \mathcal{O} such that each edge $e \in R$ appears in exactly $g(e)$ colours, construct a fractional solution by assigning a contribution $y_{\mathcal{M}} = \frac{1}{C}$ to each colour class \mathcal{M} . Checking Equations (6.4)–(6.6) shows that this is a valid fractional schedule with heat

$$\bar{h} \geq g(e) \Big/ \sum_{\mathcal{M} \in \mathbf{M}: e \in \mathcal{M}} y_{\mathcal{M}} = C$$

for all edges. Hence, $\bar{h} = C \leq \frac{3}{2}G^*$ for this assignment, and $\bar{h}^* \leq \frac{3}{2}G^*$ globally. \square

Lemma 6.3.3.

The poly density \bar{h}^* of any OPS instance $\mathcal{O} = (P, R, g)$ is bounded by $\bar{h}^* \leq \frac{3}{2}G^* + \epsilon$ for any $\epsilon > 0$.

Proof. When all heat growth rates are rational numbers, we can scale all of them by their common denominator to reduce them to the integral case. If any growth rate is irrational, we can upper-bound it with a rational number that is arbitrarily close. \square

6.3.2 Poly Density of DPS Polycules

In this section, we turn our attention to to the poly density of Decision Polyamorous Scheduling: showing that it generalizes the density of Pinwheel Scheduling with Theorem 6.3.5, introducing the PolyGreedy algorithm, and analyzing it to schedule two classes of DPS Polycules. In Section 5.5.2 we chiefly discussed poly density in the context of OPS, but the poly density of a DPS instance $\mathcal{D} = (P, R, f)$ was defined as the poly density of a paired OPS instance $\mathcal{O} = (P, R, \frac{1}{f})$:

Definition 6.3.4 Poly Density of DPS instances.

The poly density d of a DPS polycule $\mathcal{D} = (P, R, f)$ is given by

$$d = \max_{z \in \mathcal{Z}} \frac{1}{\max_{\mathcal{M} \in \mathbf{M}} \sum_{e \in \mathcal{M}} z_e \cdot f_e}$$

where \mathcal{Z} is the set of all weightings applying $z_e \in [0, 1]$ to each edge $e \in R$ such that $\sum_{e \in R} z_e = 1$ and \mathbf{M} is the set of inclusion maximal matchings \mathcal{M} in the graph (P, R) .

Given the equivalence between DPS polycules on star graphs $\mathcal{D}^* = (P, R, f)$ and Pinwheel Scheduling instances with the same frequency set f , this is a generalisation of the density of Pinwheel Scheduling:

Theorem 6.3.5.

The poly density d of any DPS polycule $\mathcal{D}^* = (P, R, f)$ whose graph (P, R) is a star is given by $d = \sum_{e \in R} \frac{1}{f_e}$

Proof. Begin with the poly density defined by Definition 6.3.4. As \mathcal{D}^* is a star graph, each matching $\mathcal{M} \in \mathbf{M}$ contains a single edge $e \in R$, so $d = \max_{z \in \mathcal{Z}} 1 / (\max_{e \in R} z_e \cdot f_e)$. This will be solved by the value of z which minimizes $\max_{e \in R} z_e \cdot f_e$. As $\sum_{e \in R} z_e = 1$, any assignment \mathcal{Z} which does not set $z_e \cdot f_e = z_{e'} \cdot f_{e'}$ for all $e, e' \in R$ will have a higher maximum than one that does. Try

$$z_e = \frac{1}{f_e \sum_{e' \in R} \frac{1}{f_{e'}}},$$

observing that

$$\sum_{e \in R} z_e = \frac{1}{\sum_{e' \in R} \frac{1}{f_{e'}}} \cdot \sum_{e \in R} \frac{1}{f_e} = 1,$$

and that $d = \sum_{e \in R} \frac{1}{f_e}$. □

The PolyGreedy Algorithm

We will now introduce PolyGreedy, an algorithm for DPS inspired by similar ideas presented in [Hol+89], which motivated this analysis³. Theorem 6.3.6 shows that the PolyGreedy algorithm is capable of scheduling the entire “Power Couples” class of DPS instances; this

³Note that the greed here is exercised by the scheduler – in practical applications, “PolyDecisive” or “PolyKnowingWhereYouWantToEat” may better capture the spirit of this algorithm.

allows us to prove Theorem 6.1.2 by showing that the class of polycules with $d \leq \frac{1}{4}$ is dominated by the “Power Couples” class⁴.

```

POLYGREEDY( $P, R = \{e_1, \dots, e_{|R|}\}, f$ )
  // Assumes  $f(e_i) = 2^{k_i}$  for some  $k_i \in \mathbb{N}$  for all  $e_i \in R$ 
  // W.l.o.g. let  $f(e_1) \leq f(e_2) \leq \dots \leq f(e_{|R|})$ 
1   $f_{\max} = f(e_{|R|})$ 
2  for  $t = 0, \dots, f_{\max}$ 
3     $\mathcal{S}[t] := \emptyset$ 
4  end for
5  for  $i = 1, \dots, |R|$ 
6     $s_i := \min\{t \in [0..f_{\max}) : \mathcal{S}[t] \cup \{e_i\} \text{ is a matching}\}$ 
7    for  $k = 1, \dots, f_{\max}/f_i$ 
8       $t := s_i + (k - 1) \cdot f_i$ 
9       $\mathcal{S}[t] := \mathcal{S}[t] \cup \{e_i\}$ 
10   end for
11 end for
12 return  $\mathcal{S}[0..f_{\max})$ 

```

Algorithm 6.1: PolyGreedy($P, R = \{e_1, \dots, e_{|R|}\}, f$)

Power Couples and Low-Density Polycules

Theorem 6.3.6 Power Couples.

Let $\mathcal{D} = (P, R, f)$ be a DPS instance such that

1. $\forall e \in R, f_e$ is a power of two
2. $\forall v \in P, \sum_{e \in R: v \in e} \frac{1}{f_e} \leq \frac{1}{2}$.

Then, there exists a valid schedule for \mathcal{D} .

Proof. Let f_{\max} be the maximum frequency among all edges and construct a periodic schedule of size f_{\max} using Algorithm 6.1. By Definition 3.2.1, schedules which lack conflicts and respect frequencies are valid; we proceed by showing inductively that schedules produced by Algorithm 6.1 have both properties.

In the initial state, \mathcal{S} is empty and thus contains no conflicts. Assuming that there are no conflicts after $i - 1$ edges have been added, observe that adding edge e_i initially places it

⁴Formalizing another common feature of real-world Polyamorous Scheduling [CA19].

in a conflict-free slot s_i , then in slots $s_i + f_i, s_i + 2f_i, s_i + 3f_i, \dots$. As all frequencies f_e are powers of two and all edges placed thus far obey $f_j \leq f_i$, slots $s_i + f_i, s_i + 2f_i, s_i + 3f_i, \dots$ must also be conflict-free for e_i .

If we again consider an arbitrary edge e_i at any point in \mathcal{S} after its initial inclusion s_i , it is clear that the schedule is frequency respecting. Initially, e_i is placed in the first slot with no edges that conflict with e_i . If $e_i = \{u, v\}$, the number of slots with conflicting edges is at most

$$\begin{aligned} \sum_{\substack{e_j: u \in e_j \\ f_j < f_i}} \frac{f_i}{f_j} + \sum_{\substack{e_j: v \in e_j \\ f_j < f_i}} \frac{f_i}{f_j} &= f_i \left(\sum_{\substack{e_j: u \in e_j \\ f_j < f_i}} \frac{1}{f_j} + \sum_{\substack{e_j: v \in e_j \\ f_j < f_i}} \frac{1}{f_j} \right) \\ &< f_i \left(\frac{1}{2} + \frac{1}{2} \right) \\ &= f_i. \end{aligned}$$

Hence, there must be at least one slot s_i among the first f_i slots with no conflicting edges. The last occurrence of e_i in \mathcal{S} is at $s_i + f_{\max} - f_i$, so the recurrence time r between the first and last instances of e_i in the periodic solution is given by f_i \square

Theorem 6.1.2 (restated). Density Threshold.

Any DPS instance $\mathcal{D} = (P, R, f)$ with poly density $d \leq \frac{1}{4}$ is schedulable.

Proof of Theorem 6.1.2. Begin by constructing an OPS instance $\mathcal{O}' = (P, R, g)$ with $g_e = \frac{1}{f_e}$ for all edges $e \in R$; by Definition 6.3.4, the poly density \bar{h}^* of \mathcal{O}' equals the density d of the corresponding $\mathcal{D} = (P, R, f)$. By Lemma 6.3.1, the maximum personal growth rate G^* of \mathcal{O}' is at most $\frac{1}{4}$, so

$$\frac{1}{4} \geq G^* = \max_{v \in P} \sum_{e \in R: v \in e} g(e) = \max_{v \in P} \sum_{e \in R: v \in e} \frac{1}{f_e},$$

and $\forall v \in V, \sum_{e \in R: v \in e} \frac{1}{f_e} \leq \frac{1}{4}$.

Now construct $\mathcal{D}' = (P, R, f')$ such that $\forall e \in R, f'(e) = 2^{\lfloor \log_2 f_e \rfloor}$, i.e. round all the frequencies down to the nearest power of two such that $\sum_{e: v \in e} \frac{1}{f'_e} \leq \sum_{e: v \in e} \frac{2}{f_e} \leq 2 \cdot \frac{1}{4} = \frac{1}{2}$. Applying Theorem 6.3.6 then gives a feasible schedule for \mathcal{D}' , which must also solve \mathcal{D} . \square

6.4 Inapproximability

In this section, we prove Theorem 6.1.3: showing, via a reduction from 3SAT, that Bipartite OPS does not allow efficient $(1 + \delta)$ -approximation algorithms for $\delta < \frac{1}{12}$ unless $P = NP$.

6.4.1 Overview of Proof

The proof of Theorem 6.1.3 has two steps: a reduction from 3SAT to Bipartite DPS, and a conversion to Bipartite OPS such that a $\frac{13}{12}$ approximation is shown to be NP-hard.

3SAT is the following problem: given a 3-CNF formula $\varphi = c_1 \wedge c_2 \wedge \dots \wedge c_m$ with clauses $C = \{c_1, c_2, \dots, c_m\}$, each consisting of exactly 3 literals over variables $X = \{x_1, x_2, \dots, x_n\}$, decide whether there is an assignment of Boolean values to the variables in X such that all m clauses in C are satisfied.

Lemma 6.4.1 $3SAT \leq_p$ Bipartite DPS.

For any 3-CNF formula φ with m clauses, we can construct in polynomial time a bipartite decision polycule \mathcal{D}_φ which has a valid schedule if and only if all clauses of φ can be simultaneously satisfied.

The bulk of Section 6.4 will introduce our construction of \mathcal{D}_φ , which we will use to prove this Lemma. This construction is designed to be modular and communicable, rather than efficient with constant factors.

The second step in the proof of Theorem 6.1.3 is to convert the decision polycule \mathcal{D}_φ from Lemma 6.4.1 to an optimisation polycule \mathcal{O}_φ using Lemma 3.2.4. It will be immediately obvious from the construction that the largest frequency in \mathcal{D}_φ is $F = 12$. So by Lemma 3.2.4, \mathcal{O}_φ has a schedule with $h^* = 1$ if \mathcal{D}_φ is feasible or has schedules where $h^* \geq \frac{13}{12}$ otherwise.

Theorem 6.1.3 (restated). SAT Hardness of Approximation.

Unless $P = NP$, there is no polynomial-time $(1 + \delta)$ -approximation algorithm for the Bipartite Optimisation Polymorous scheduling problem for any $\delta < \frac{1}{12}$.

Proof of Theorem 6.1.3. Assume that there is a polynomial-time approximation algorithm A , which solves Bipartite OPS with approximation ratio $\alpha < \frac{13}{12}$. If \mathcal{O}_φ has optimal heat $h^* = 1$, A produces a schedule with heat $1 \leq h \leq \alpha < \frac{13}{12}$; if $h^* \geq \frac{13}{12}$, A must produce a schedule of heat $\frac{13}{12} \leq h \leq \alpha \cdot \frac{13}{12}$. By running A , we are therefore able to distinguish between $h^* \leq \frac{13}{12}$ and $h^* \geq \frac{13}{12}$ for \mathcal{O}_φ , and hence between the feasibility or infeasibility of \mathcal{D}_φ . This then discriminates between Yes and No instances of 3SAT, via the polynomial-time reduction from Lemma 6.4.1. 3SAT is NP-hard, hence $P = NP$ follows. \square

Let us denote by α^* the approximability threshold for Bipartite OPS, that is: efficient polynomial-time approximation algorithms with approximation ratio α exist if and only if $\alpha \geq \alpha^*$ (assuming $P \neq NP$). Theorem 6.1.3 shows that $\alpha^* \geq \frac{13}{12}$, while Theorem 6.2.10 shows that $\alpha^* \leq 5.24$, leaving a substantial gap. We conjecture that the constant $\frac{13}{12}$ can be improved by careful analysis of our construction, but we leave this to future work.

Conjecture 6.4.2.

The approximability threshold for Bipartite OPS is $\alpha^ \geq \frac{4}{3}$.*

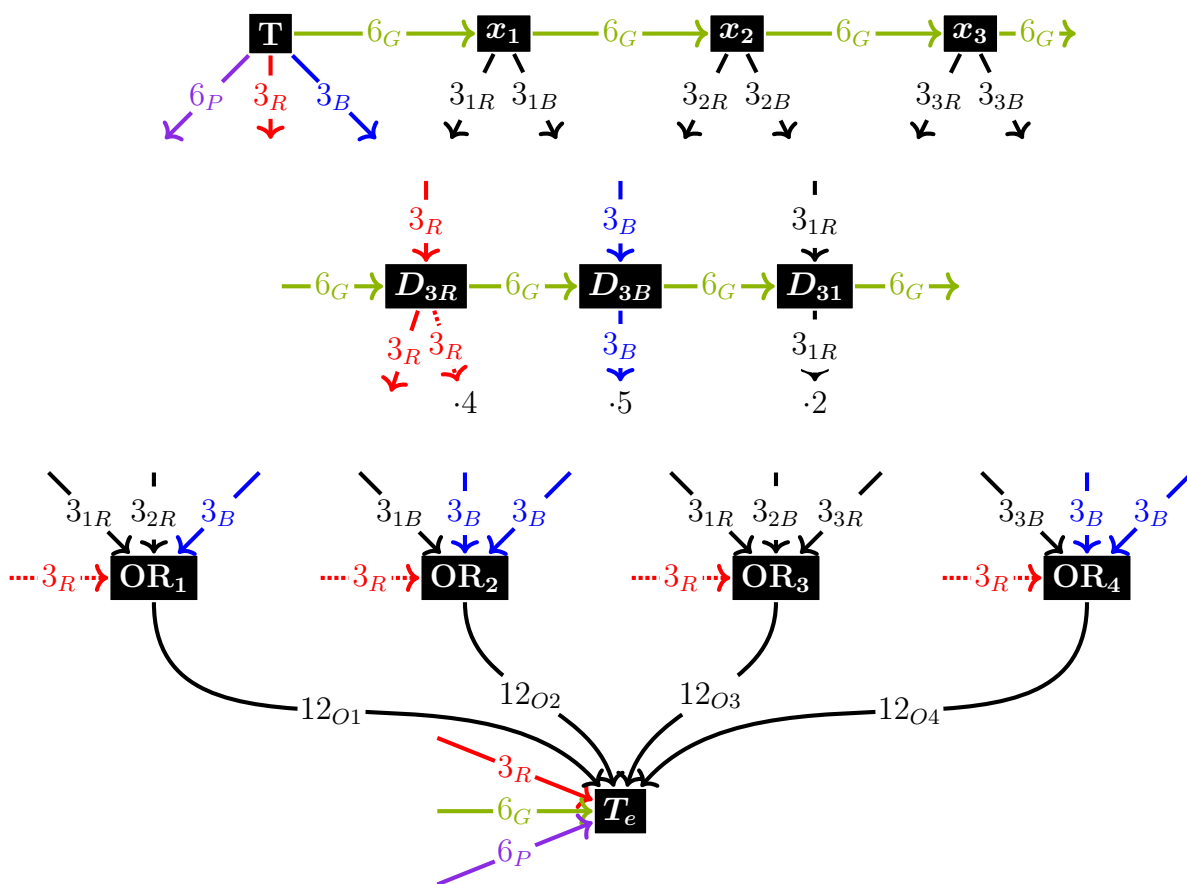


Figure 6.2: A Bipartite DPS polycule which is schedulable iff there is some assignment that satisfies $(x_1 \vee x_2), (\bar{x}_1), (x_1 \vee \bar{x}_2 \vee x_3)$, and (\bar{x}_3) (which is not possible). Connections between layers are omitted for clarity but flow from top to bottom, starting with the variable layer, then a layer duplicating nodes with frequency 3, the OR layer, and finally the tensioning layer. Note that larger problems will also require a D_6 layer to support additional tension gadgets.

6.4.2 Overview of Reduction

We now give the proof of Lemma 6.4.1. For the remainder of this section, we will assume that a 3-CNF formula $\varphi = c_1 \wedge \cdots \wedge c_m$ over variables $X = \{x_1, \dots, x_n\}$ is given. Using this, we will show how to construct a DPS instance \mathcal{D}_φ which admits a schedule iff there is a variable assignment $v : X \rightarrow \{\text{True}, \text{False}\}$ that satisfies all clauses in $C = \{c_1, \dots, c_m\}$. This construction directly represents components of Boolean formulas as Bipartite DPS “gadgets”:

- *variables* (Section 6.4.4),
- clauses (*OR gadgets*) (Section 6.4.7), and
- the *tension gadget* (Section 6.4.8), which checks that all clauses are True.

To make these gadgets work, we require further auxiliary gadgets:

- a “*True Clock*” to break ties between symmetric choices for schedules (Section 6.4.3),
- slot duplication gadgets: D_3 duplicators and D_6 duplicators (Section 6.4.6).

The overall conversion algorithm is given by Definition 6.4.3 and a worked example is shown in Figure 6.2.

Definition 6.4.3 \mathcal{D}_φ polycules.

The decision polycule $\mathcal{D}_\varphi = (P, R, f)$ is constructed in layers as follows:

(a) Variable layer:

The variable layer consists of a variable gadget with outputs 3_{iR} and 3_{iB} for each variable $x_i \in X$, as well as a single special variable: the True Clock.

(b) Duplication layer:

The duplication layer duplicates outputs of the variable layer: D_3 duplicators create one 3_{iR} edge for each x_i in clauses $c_j \in C$, and one 3_{iB} edge for each occurrence of \bar{x}_i in clauses $c_j \in C$. They also create as many 3_R and 3_B edges as are needed, while D_6 duplicators do the same for 6_G and 6_P edges. Both male and female constant edges are created, and unused edges are connected to pendant nodes.

(c) Clause layer:

The clause layer consists of one OR gadget for each clause $c_j \in C$. OR gadgets have three inputs, each corresponding to a literal in c_j : 3_{iR} for x_i , 3_{iB} for \bar{x}_i . For clauses with less than 3 literals, 3_B edges fill the OR gadget’s unused inputs.

(d) Tension layer:

The tension layer attaches tension gadgets to all outputs of the sorting layer; any spare inputs to the tension layer are connected to pendant nodes.

6.4.3 The True Clock & Colour Slots

Figure 6.3 introduces gadgets representing sample variables x_1 and x_2 , as well as a special variable: the True Clock T , which acts as a drumbeat for the polycule as a whole. Variables, including T , have four relationships: $[3, 3, 6, 6]$, so their local schedules must all have the following form: $[3_a, 3_b, 6_a, 3_a, 3_b, 6_b]$. As schedules are cyclic, we can choose to start this schedule with the 3_a edge of T without loss of generality. We then assign names to these edges as dictated by the slots they are in:

Definition 6.4.4 Slots.

We call days $t \in \mathbb{N}_0$ with $t \equiv 0 \pmod{3}$ the *red slots*, days $t \equiv 1 \pmod{3}$ *blue slots*, days $t \equiv 2 \pmod{6}$ *green slots*, and days $t \equiv 5 \pmod{6}$ *purple slots*.

The local schedule of the True Clock is therefore given by $[3_R, 3_B, 6_G, 3_R, 3_B, 6_P]$. As 3_R is scheduled on day 0, it will always be assigned in red slots, with 3_B , 6_G , and 6_P edges also restricted to slots of their respective colours. We will sometimes represent this by underlining elements or gaps in a schedule, e.g., $[\underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}]$, or by referring to edges as being red, blue, green, or purple.

All gadgets introduced below are constructed such that the lengths of their schedules are integer multiples of 6; in the final polycule \mathcal{D}_φ they will be connected (usually through intermediaries) to the True Clock, such that their edges must stick to certain slots. To keep correctness proofs of individual gadgets readable, we call a schedule \mathcal{S} that schedules all coloured edges in slots of the given colour *slot-respecting*.

Drawing Conventions Gadgets are connected by input and output edges, represented by incoming and outgoing arrows respectively – each arrow corresponding to half of a relationship between two people from different gadgets. In addition to their frequencies, input and output edges share restrictions on their permissible schedules, carrying them from one gadget to another as discussed in the proofs associated with each gadget.

In shorthand gadgets, vertical incoming edges are the primary *input* to a gadget, encoding the value of some variable or logical function; horizontal inputs contain edges of fixed colour, which we will refer to as *constants*. Similarly, vertical outgoing edges represent primary

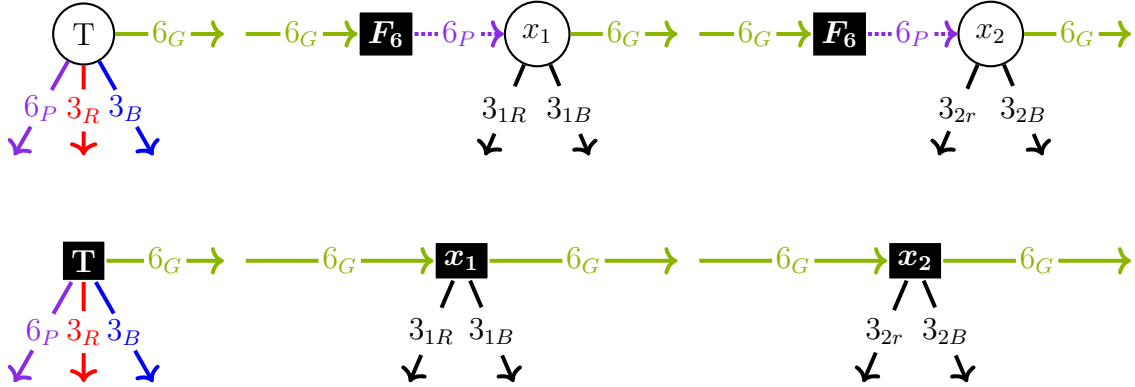


Figure 6.3: Gadgets for the True Clock and for sample variables x_1 and x_2 (top, left to right), with shorthand versions shown below. Gadgets are shown connected as they would be in a sample variable layer, and their colours and schedules are discussed in Section 6.4.3. Further variables can be added to the right.

outputs that encode the result of the gadget, while horizontal outgoing edges represent incidentally created constants which may either be used by other gadgets or connected to pendant vertices.

Some incoming and outgoing edges will have end labels of the form “ i ”, indicating i connections of the given type, each with a different person.

To show that \mathcal{D}_φ is bipartite, we assign a gender to each node: male nodes have solid borders; female nodes have dotted borders. We will also use a directed graph to show the gender of the originating node: edges coming from male nodes have solid edges, while edges coming from female nodes will have dotted edges. Note that this convention is only used to show that the complete graph is bipartite while reasoning about parts of it; when fully assembled, the overall graph is undirected (as we would expect for polyamorous scheduling).

6.4.4 Variables

Figure 6.3 also introduces the gadget for a sample variable, x_1 , which again has four relationships: $[3_{1R}, 3_{1B}, 6_G, 6_P]$. The key property of variable gadgets is summarized in the following Lemma:

Lemma 6.4.5 Variable gadget schedules.

The x_i node in each variable gadget x_i has a valid local schedule which must be of the form $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$ in any slot-respecting global schedule.

Proof. x_i nodes have tasks $[3_{1R}, 3_{1B}, 6_G, 6_P]$ and local density $D = 1$, so 3_{iR} and 3_{iB} must be scheduled exactly once in each 3-day period, forcing every 3 days to be of the form $[3_a, 3_b, _]$, and every 6-day schedule to be of the form $[3_a, 3_b, _, 3_a, 3_b, _]$; this leaves two remaining slots, which must contain 6_G and 6_P .

The 6_G edge of the first variable node, x_1 , is shared with T such that it must be green, so valid schedules for x_1 are of the form $[3_a, 3_b, 6_G, 3_a, 3_b, _]$, which must be completed as $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$ – a valid schedule for x_1 .

The 6_P edge leaving x_1 is then connected to an F_6 flipper gadget, which returns a 6_G edge by Lemma 6.4.6. Further variables are each connected to the 6_G edge returned from the F_6 flipper gadget of the previous variable gadget, so their schedules must be of the same form. It therefore follows inductively that each variable gadget x_i has a valid schedule, and that it must be of the form $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$. \square

In accordance with Lemma 6.4.5, the incident 6_G edge is used to restrict the valid schedules for x_1 , leaving two possibilities: $[3_{1R}, 3_{1B}, 6_G, 3_{1R}, 3_{1B}, 6_P]$ and $[3_{1B}, 3_{1R}, 6_G, 3_{1B}, 3_{1R}, 6_P]$. The former schedule, where 3_{1R} is scheduled in red slots, corresponds to a variable assignment where x_1 is True, whereas the the second schedule corresponds to x_1 being assigned False.

This technique of connecting 3_R , 3_B , 6_G , or 6_P edges to nodes in a gadget in order to limit their valid local schedules and force relationships between edges, slots, and particular meanings will be used extensively in what follows.

Note that 3_{iB} has the opposite value to 3_{iR} , so using 3_{iB} edges in the polycule corresponds to the negated literal \bar{x}_i , just as 3_{iR} edges correspond to the literal x_i .

6.4.5 Flippers

Flippers are introduced by Figure 6.4, and are used to ensure that $\mathcal{D}_\varphi = (P, R, f)$ is bipartite by converting between male and female edges. They allow variable and duplicator gadgets to each consume a constant, rather than linear, amount of 3_B , 6_G , and 6_P edges.

Lemma 6.4.6 F_6 Flipper gadget schedules.

The F_6 node in each F_6 flipper gadget has a valid local schedule which must be of the form $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$ in any slot-respecting global schedule.

Proof. Note that F_6 flippers and variables share the same tasks: $[3, 3, 6, 6]$. By the proof of Lemma 6.4.5, schedules for F_6 flippers must be of the form $[3_a, 3_b, _, 3_a, 3_b, _]$, where

the remaining slots must contain 6_G and 6_P edges. The incoming edge, either a 6_G or 6_P , must be in a corresponding slot, forcing schedules to be of the form $[3_a, 3_b, 6_G, 3_a, 3_b, _]$ or $[3_a, 3_b, _, 3_a, 3_b, 6_P]$ respectively. Both must yield the complete and valid local schedule $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$. \square

Lemma 6.4.7 F_3 Flipper gadget schedules.

The F_3 node in each F_3 flipper gadget has a valid local schedule which must be of the form $[3_R, 3_B, 6_G, 3_R, 3_B, 6_P]$ in any slot-respecting global schedule.

Proof. F_3 flipper nodes have two inputs, $[3_i, 6_i]$ and two outputs, $[3_o, 6_o]$. 3_i and 6_i are indirectly connected to the True Clock such that 3_i must be red or blue, and 6_i must be either green or purple. This forces partial schedules to be of the form $[3_i, _, 6_i, 3_i, _, _]$ or $[3_i, 6_i, _, 3_i, _, _]$. As F_3 nodes have density $D = 1$, the 6_o edge must be placed 3 days after the 6_i edge to avoid violating the condition of the 3_o edge, for schedules of the form $[3_i, 3_o, 6_i, 3_i, 3_o, 6_o]$ and $[3_i, 6_i, 3_o, 3_i, 6_o, 3_o]$ respectively. If 3_i is red, 3_o will be blue, and vice versa; similarly, one 6 edge will be purple and the other green. In either case, the resultant schedule will be valid and of the form $[3_R, 3_B, 6_G, 3_R, 3_B, 6_P]$. \square

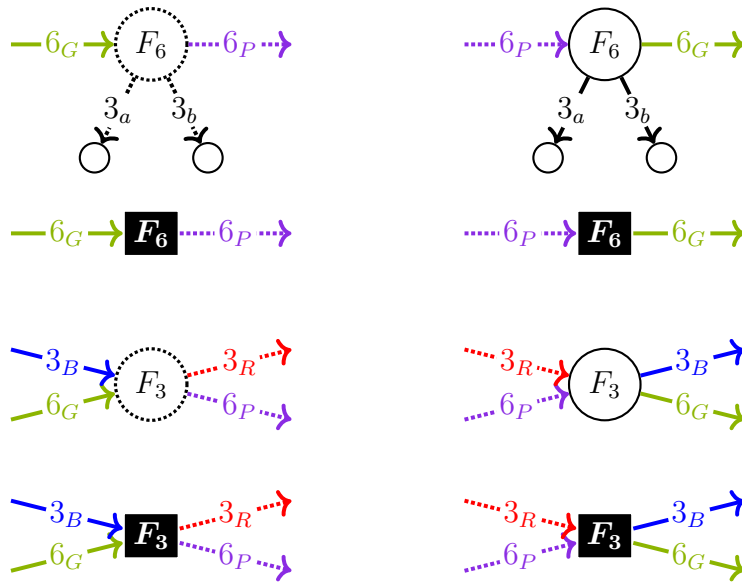


Figure 6.4: Four sample flipper gadgets, each with shorthand versions below them: 6_G male to 6_P female (top left), 6_P female to 6_G male (top right), 3_B male to 3_R female (bottom left), and 3_R female to 3_B male (bottom right). Note that F_3 flippers act incidentally as F_6 flippers, but are used separately in our construction for clarity.

6.4.6 Duplication of Variables and Constants

Variables may appear in multiple clauses, while the constant 3_R , 3_B , 6_G , and 6_P edges are used in multiple gadgets, engendering a need for the duplication of variables and constants.

3-Duplicators

A gadget for duplicating edges with period 3 is shown in Figure 6.5 and proven to accurately reproduce its input by Lemma 6.4.8.

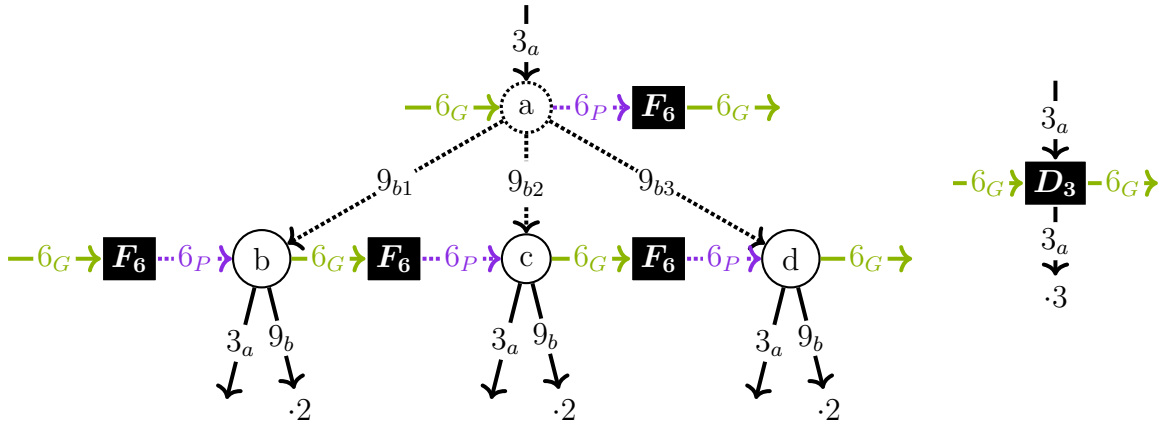


Figure 6.5: A gadget for duplicating input edges with frequency 3, with a shorthand version on the right. Note that the input can be duplicated indefinitely many times by adding further layers which each use the 9_b edges from the previous layer, but the output will alternate between male and female 3_a edges. Odd layers will have female main nodes with F_6 flipper nodes on the right, as shown in the top layer, while even layers will have male main nodes with F_6 flipper nodes on the left, as shown on the bottom layer. 6_P edges can be connected continuously from layer to layer, with the final edge released to another gadget.

Lemma 6.4.8 3-Duplicator gadget schedules.

In any slot-respecting global schedule, the non-flipper nodes in 3-duplicator gadgets D_3 have two forms of valid local schedules, either:

$$[3_a, 9_b, 6_G, 3_a, 9_{b'}, 6_P, 3_a, 9_{b''}, 6_G, 3_a, 9_b, 6_P, 3_a, 9_{b'}, 6_G, 3_a, 9_{b''}, 6_P] \text{ or}$$

$$[9_b, 3_a, 6_G, 9_{b'}, 3_a, 6_P, 9_{b''}, 3_a, 6_G, 9_b, 3_a, 6_P, 9_{b'}, 3_a, 6_G, 9_{b''}, 3_a, 6_P].$$

Proof. Each non-flipper node in D_3 has tasks $[3_a, 6_P, 6_G, 9_{b1}, 9_{b2}, 9_{b3}]$ and has local density $D = 1$, so each task with frequency f must appear exactly once every f days. Further, in any slot-respecting schedule, the 6_G edge must be scheduled in green slots, forcing partial schedules of the form $[_, _, 6_G, _, _, _]$.

Considering node a , note that if incident edge 3_a is scheduled on a combination of red and blue days then either it will appear more than once in some 3-day period or its constraint will be violated. This demonstrates that schedules must be of the form:

$$[3_a, _, 6_G, 3_a, _, _, 3_a, _, 6_G, 3_a, _, _, 3_a, _, 6_G, 3_a, _, _] \text{ or}$$

$$[_, 3_a, 6_G, _, 3_a, _, _, 3_a, 6_G, _, 3_a, _, _, 3_a, 6_G, _, 3_a, _],$$

depending on the colour of the incident 3_a edge.

Assume towards a contradiction that the 6_P edge is not consistently scheduled in purple slots. This produces the schedules:

$$[3_a, 6_P, 6_G, 3_a, _, _, 3_a, 6_P, 6_G, 3_a, _, _, 3_a, 6_P, 6_G, 3_a, _, _] \text{ and}$$

$$[6_P, 3_a, 6_G, _, 3_a, _, 6_P, 3_a, 6_G, _, 3_a, _, 6_P, 3_a, 6_G, _, 3_a, _],$$

and their cyclic permutations. Both schedules have two free slots in the first 9 day period, and four free slots in the second 9 day period – forcing a contradiction when the 9_{b1} , 9_{b2} and 9_{b3} edges are added.

Thus, two partial schedules remain:

$$[3_a, _, 6_G, 3_a, _, 6_P, 3_a, _, 6_G, 3_a, _, 6_P, 3_a, _, 6_G, 3_a, _, 6_P] \text{ and}$$

$$[_, 3_a, 6_G, _, 3_a, 6_P, _, 3_a, 6_G, _, 3_a, 6_P, _, 3_a, 6_G, _, 3_a, 6_P],$$

In either case, 9_{b1} , 9_{b2} and 9_{b3} must occupy the remaining slots, which match the schedules shown in the Lemma for some mapping of 9_{b1} , 9_{b2} and 9_{b3} to 9_b , $9_{b'}$ and $9_{b''}$.

Now consider an arbitrary non-flipper node $n \neq a$, with an incident 9_b node. If the 3_a edge incident to a is red, the schedule for n must be of the form:

$$[_, 9_b, 6_G, _, _, _, _, _, 6_G, _, 9_b, _, _, _, 6_G, _, _, _],$$

$$[_, _, 6_G, _, 9_b, _, _, _, 6_G, _, _, _, 9_b, 6_G, _, _, _], \text{ or}$$

$$[_, _, 6_G, _, _, _, _, 9_b, 6_G, _, _, _, _, 6_G, _, 9_b, _].$$

In any case, if we schedule the outgoing 3_a edge in either blue or purple slots, its constraint will be violated, so it must be scheduled in the remaining red slots, leading to the following partial schedules:

$$\begin{aligned} & [\underline{3}_a, \underline{9}_b, \underline{6}_G, \underline{3}_a, _, _, \underline{3}_a, _, \underline{6}_G, \underline{3}_a, \underline{9}_b, _, \underline{3}_a, _, \underline{6}_G, \underline{3}_a, _, _], \\ & [\underline{3}_a, _, \underline{6}_G, \underline{3}_a, \underline{9}_b, _, \underline{3}_a, _, \underline{6}_G, \underline{3}_a, _, _, \underline{3}_a, \underline{9}_b, \underline{6}_G, \underline{3}_a, _, _], \text{ or} \\ & [\underline{3}_a, _, \underline{6}_G, \underline{3}_a, _, _, \underline{3}_a, \underline{9}_b, \underline{6}_G, \underline{3}_a, _, _, \underline{3}_a, _, \underline{6}_G, \underline{3}_a, \underline{9}_b, _]. \end{aligned}$$

If either the $9_{b'}$ edge or the $9_{b''}$ edge is scheduled in the remaining purple slots, it will conflict with the 6_G edge, so the $9_{b'}$ and $9_{b''}$ edges must be scheduled in the remaining blue slots, and the 6_P edge must be purple. This leads to full schedules of the form:

$$[\underline{3}_a, \underline{9}_b, \underline{6}_G, \underline{3}_a, \underline{9}_{b'}, \underline{6}_P, \underline{3}_a, \underline{9}_{b''}, \underline{6}_G, \underline{3}_a, \underline{9}_b, \underline{6}_P, \underline{3}_a, \underline{9}_{b'}, \underline{6}_G, \underline{3}_a, \underline{9}_{b''}, \underline{6}_P],$$

which again matches the Lemma. If the 3_a edge incident to a is blue instead, the same logic applies, with all 3_a edges also being blue and all 9_b edges being red. \square

6-Duplicators

Figure 6.6 and Lemma 6.4.9 introduce a gadget which duplicates incident 6_G or 6_P edges.

Lemma 6.4.9 6-Duplicator gadget schedules.

In any 6-duplicator gadget D_{6P} with an input 6_P edge, the non-flipper nodes have valid local schedules, which must be of the form:

$$[\underline{3}_R, \underline{3}_B, \underline{12}_{G1}, \underline{3}_R, \underline{3}_B, \underline{6}_P, \underline{3}_R, \underline{3}_B, \underline{12}_{G2}, \underline{3}_R, \underline{3}_B, \underline{6}_P]$$

in any slot-respecting global schedule.

Proof. Each non-flipper node in D_{6P} has tasks $[\underline{3}_R, \underline{3}_B, \underline{6}_P, \underline{12}_{G1}, \underline{12}_{G2}]$, giving them density $D = 1$; it follows that each task with frequency f must appear exactly once every f days.

Consider node a , which has inputs 3_B and 6_P . In any slot-respecting schedule these are scheduled in slots of their respective colours, which forces partial schedules for a to be of the form $[_, \underline{3}_B, _, _, \underline{3}_B, \underline{6}_P]$.

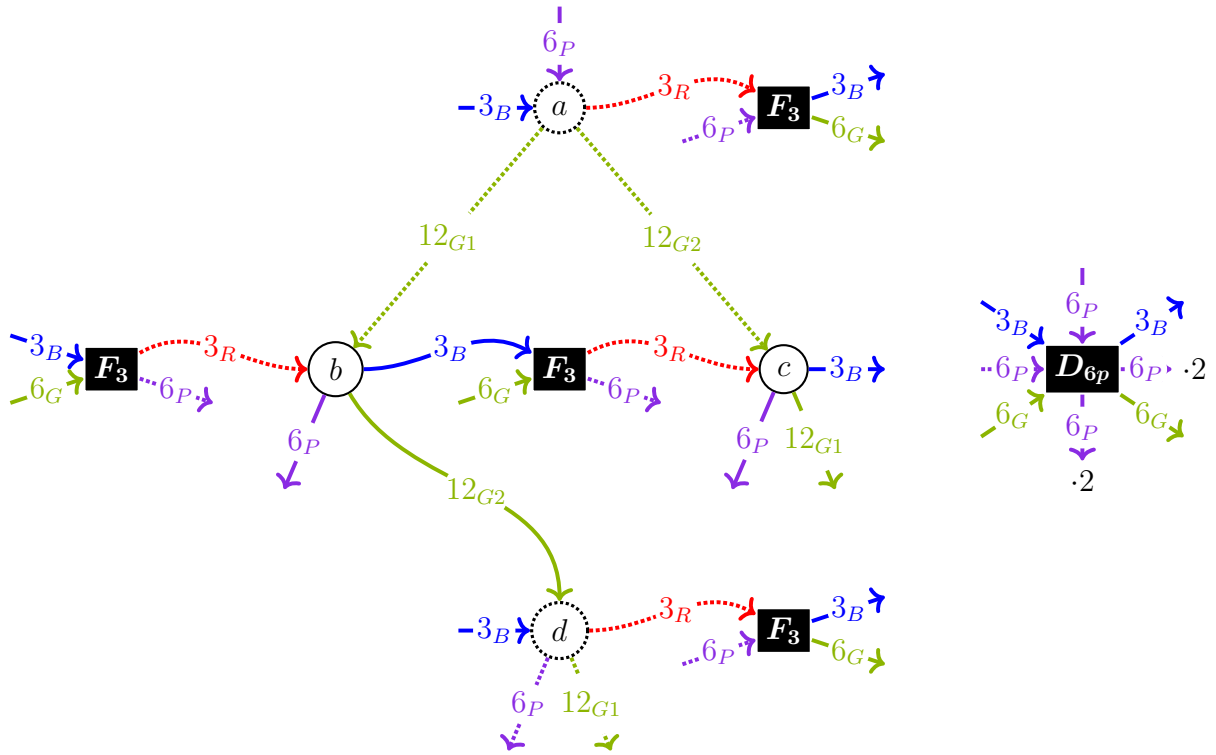


Figure 6.6: A gadget for duplicating 6_P input edges, with a shorthand version on the right. The gadget consists of a female root node (a), male branch nodes (b, c), and female branch node (d). The root node consumes one male 6_P edge, one female 6_P edge, and produces one male 6_G edge; male branch nodes produce one male 6_P edge and one female 6_P edge while consuming one male 6_G edge; female branch nodes produce one male 6_G edge. Constant edges produced by some node and consumed by some other node are connected in some legal way not shown, with production and consumption cancelling each other out. The gadget can be extended by adding a pair of branch nodes, one male and one female, to produce one more male 6_P edge and one more female 6_P edge. As shown, male nodes have F_3 flippers on their left, while female nodes have F_3 flippers on their right. 6_G edges can be duplicated with a very similar D_{6G} gadget simply by replacing the topmost 6_P edge with a 6_G edge.

Exactly two of these slots must be filled by 3_R , and if these are not both red slots, the constraint of 3_R will be violated. Thus, the schedule for a must be of the form:

$$[3_R, 3_B, _, 3_R, 3_B, 6_P, 3_R, 3_B, _, 3_R, 3_B, 6_P].$$

Two spaces remain, which must then contain 12_{G1} and 12_{G2} , as shown in the Lemma.

Now consider an arbitrary female node which is neither a nor a flipper; all such nodes will have inputs 12_{Ga} and 3_R , forcing their partial schedules to be of the form:

$$[3_R, _, 12_{Ga}, 3_R, _, _, 3_R, _, _, 3_R, _, _].$$

As with a , exactly four of these slots must schedule 3_B edges, and these must be the blue spaces for the 3_B constraint not to be violated, forcing schedules of the form:

$$[3_R, 3_B, 12_{Ga}, 3_R, 3_B, _, 3_R, 3_B, _, 3_R, 3_B, _].$$

Similarly, male non-flipper nodes such as b and c have 12_{Ga} and 3_B inputs which force their partial schedules to be of the form:

$$[_, 3_B, 12_{Ga}, _, 3_B, _, _, 3_B, _, _, 3_B, _].$$

We must add 3_R edges to exactly four of these slots, and these must be the red slots for the 3_R constraint not to be violated, again giving schedules of the form:

$$[3_R, 3_B, 12_{Ga}, 3_R, 3_B, _, 3_R, 3_B, _, 3_R, 3_B, _].$$

One of these slots must contain the 12_{Gb} edge, with the other two scheduling the 6_P edge. If the 12_{Gb} edge is not scheduled in the remaining green slot, the constraint on the 6_P edge will be violated, so the schedule must be as shown in the Lemma. \square

Corollary 6.4.10.

Note that by replacing the 6_P input edge with a 6_G input edge, a D_{6G} replicator will be made instead that produces 6_G edges according to the same logic.

6.4.7 Clauses

Clauses in C are disjunctions of at most 3 literals, i.e. the logical OR of at most 3 variables x_i , or their negations $\overline{x_i}$. A gadget which determines the truth value of a clause given the values of its variables is shown in Figure 6.7. Lemma 6.4.11 shows that the 12_O output edges of this gadget can be blue iff the corresponding clause is evaluated to be True; Definition 6.4.3 and Lemma 6.4.12 ensure that all 12_O edges must be blue in any valid global schedule. Thus, for \mathcal{D}_φ to have a valid schedule, φ must have a satisfying assignment.

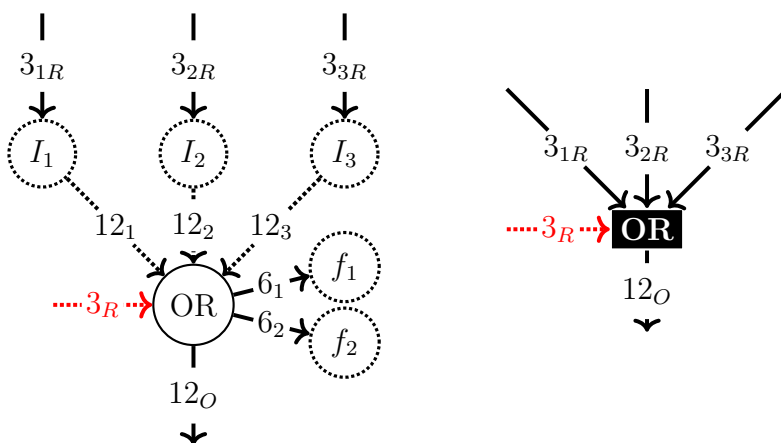


Figure 6.7: A gadget which computes $x_1 \vee x_2 \vee x_3$ (left), along with a shorthand version (right). To compute $x_1 \vee x_2 \vee \overline{x_3}$, replace the incoming 3_{3R} edge with a 3_{3B} edge. To instead compute $x_1 \vee x_2$, replace the incoming 3_{3R} edge with a male 3_B edge. Lemma 6.4.11 shows that the 12_O edge can be blue iff at least one input is assigned True, while Definition 6.4.3 and Lemma 6.4.12 show that 12_O edges must be blue in any valid global schedule.

Lemma 6.4.11 OR gadget schedules.

Consider an OR gadget whose input edges 3_{1R} , 3_{2R} , and 3_{3R} are either red or blue. There exists a slot-respecting schedule in which the 12_O output edge of this OR gadget is blue iff at least one of its input edges is red.

While this Lemma only considers the clause $x_1 \vee x_2 \vee x_3$, all other clauses can be handled similarly: For a literal x_j , we use 3_{jR} as input and for negated literals $\overline{x_j}$, we instead use 3_{jB} . Clauses with fewer literals also share the same gadget and proof: here, we replace each unused 3_{jR} edge with a male 3_B edge, effectively replacing clauses such as $x_1 \vee x_2$ with analogous clauses in the form $x_1 \vee x_2 \vee \perp$.

Proof of Lemma 6.4.11. The node labelled OR has tasks $[3_R, 6_1, 6_2, 12_1, 12_2, 12_3, 12_O]$ and local density $D = 1$, so each task with frequency f must appear exactly once in every f day period. Any slot-respecting schedule must assign the 3_R edge to red slots, so schedules for the OR node must be of the form $[3_R, _, _, 3_R, _, _]$.

Consider an inverter node $I_{i \in \{1,2,3\}}$, which has tasks $[3_{iR}, 12_i]$ where 3_{iR} is either red or blue by assumption. If the input edge of this node, 3_{iR} , is red, then partial schedules for I_i must be of the form $[3_{iR}, _, _, 3_{iR}, _, _]$ and the 12_i edge must be either green, purple, or blue. Similarly, if 3_{iR} is scheduled in blue slots, then partial schedules for I_i must instead be of the form $[_, 3_{iR}, _, _, 3_{iR}, _]$ and the 12_i edge must be either green, purple, or red; however, this 12_i edge is also connected to the OR node, which has no empty red slots, further restricting it to be either green or purple.

Suppose that one input edge, 3_{tR} , is scheduled in red slots, while the others, 3_{iR} and $3_{i'R}$, may be scheduled in either red or blue slots. As 3_{tR} is red, the output of the associated inverter, 12_t , may be blue, green, or purple. Consider the schedule

$$[3_R, 12_t, 6_a, 3_R, 6_b, 12_i, 3_R, 12_O, 6_a, 3_R, 6_b, 12_{i'}],$$

observing that 12_O is scheduled in blue slots and that the schedule is slot-respecting.

Now suppose that all three inputs, 3_{1R} , 3_{2R} , and 3_{3R} , are scheduled in blue slots. By the reasoning above, 12_1 , 12_2 , and 12_3 must now be scheduled in green or purple slots. This will force schedules for the OR node to be of the form:

$$[3_R, _, 12_a, 3_R, _, 12_b, 3_R, _, 12_c, 3_R, _, _] \text{ or}$$

$$[3_R, _, 12_a, 3_R, _, 12_b, 3_R, _, _, 3_R, _, 12_c],$$

for some mapping of 12_1 , 12_2 , and 12_3 onto 12_a , 12_b , and 12_c . Assume towards a contradiction that the 12_O edge is blue, observing an immediate constraint violation when scheduling 6_1 and 6_2 . This demonstrates that the 12_O edge cannot be blue and the schedule for the OR node must be of the form:

$$[3_R, 6_a, 12_a, 3_R, 6_b, 12_b, 3_R, 6_a, 12_c, 3_R, 6_b, 12_d]$$

(likewise, for some mapping of 12_1 , 12_2 , 12_3 , and 12_O onto 12_a , 12_b , 12_c , and 12_d). Thus, if all input edges are blue, then the 12_O edge must be either green or purple; it cannot be blue, confirming the Lemma. \square

6.4.8 Tension

$P_{d\varphi}$ is intended to have a valid schedule iff the underlying φ is satisfied by some assignment of values to its variables. Lemma 6.4.11 shows that the 12_O output edge of an OR gadget can be blue iff the associated clause is satisfied by this assignment. This section introduces Tension gadgets (shown in Figure 6.8), which ensure that in any global schedule for $P_{d\varphi}$ all 12_O edges must be blue, and hence that all clauses must be True.

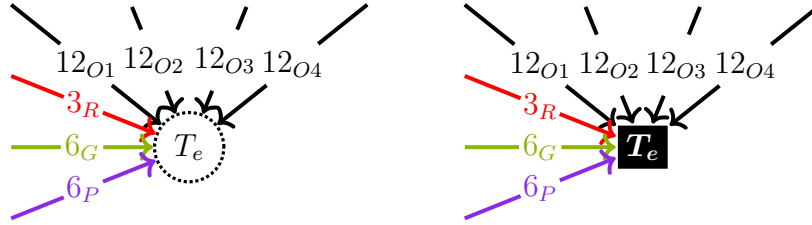


Figure 6.8: A gadget (left) which applies tension to four inputs, ensuring that either 12_{O_1} , 12_{O_2} , 12_{O_3} , and 12_{O_4} are scheduled in blue slots or no schedule can be found for T_e . To apply tension to fewer than 4 inputs, connect any unneeded inputs to their own male pendant node.

Lemma 6.4.12 Tension gadget schedules.

Each Tension gadget T_e has a valid local schedule iff all 12_O input edges are scheduled in blue slots.

Proof. The single node of each tension gadget T_e , as shown in Figure 6.8, has tasks $[3_R, 6_G, 6_P, 12_{O_1}, 12_{O_2}, 12_{O_3}, 12_{O_4}]$ and density $D = 1$, so each task with frequency f must appear exactly once in every f day period. Further, the 3_R , 6_G , and 6_P edges are indirectly connected to the True Clock such that they must be scheduled in slots of their respective colours. Partial schedules for T_e nodes must therefore be of the form $[3_R, _, 6_G, 3_R, _, 6_P]$. All empty slots in schedules of this form are blue, so the remaining $12_{O_1}, 12_{O_2}, 12_{O_3}, 12_{O_4}$ edges must be scheduled in blue slots, restricting valid schedules to the form:

$$[3_R, \underline{12_{O_a}}, 6_G, 3_R, \underline{12_{O_b}}, 6_P, 3_R, \underline{12_{O_c}}, 6_G, 3_R, \underline{12_{O_d}}, 6_P],$$

for any mapping of $12_{O_1}, 12_{O_2}, 12_{O_3}$, and 12_{O_4} onto $12_{O_a}, 12_{O_b}, 12_{O_c}$, and 12_{O_d} . □

6.4.9 Proof of Reduction

With these preparations we can, at long last, prove Lemma 6.4.1, and hence complete the proof of Theorem 6.1.3.

Lemma 6.4.1 (restated). $3\text{SAT} \leq_p \text{Bipartite DPS}$.

For any 3-CNF formula φ with m clauses, we can construct in polynomial time a bipartite decision polycule \mathcal{D}_φ which has a valid schedule if and only if all clauses of φ can be simultaneously satisfied.

Proof of Lemma 6.4.1. Consider a polycule \mathcal{D}_φ built from some 3-CNF formula $\varphi = c_1 \wedge \dots \wedge c_m$ over variables $X = \{x_1, \dots, x_n\}$ using the algorithm defined by Definition 6.4.3.

First, suppose that there is a variable assignment $v : X \rightarrow \{\text{True}, \text{False}\}$ which satisfies all clauses $c_i \in C$. We construct a slot-respecting schedule \mathcal{S} for \mathcal{D}_φ by combining the following:

- (a) At the variable gadget for each $x_i \in X$, schedule 3_{iR} in red slots if $v(x_i) = \text{True}$ and in blue slots otherwise. This fixes a schedule for all edges in the variable layer.
- (b) Lemma 6.4.8 and Lemma 6.4.9 give schedules for duplication gadgets, whose number should be sufficient to cover the needs of all other gadgets.
- (c) As v satisfies all clauses $c_i \in C$, each clause has at least one literal (x_i or \bar{x}_i) which evaluates to True. Hence, each clause has one input 3_{iR} or 3_{iB} edge which will be scheduled in red slots. By Lemma 6.4.11, there therefore exists a valid schedule for each OR gadget in the clause layer such that the 12_O output edge of that gadget is scheduled in blue slots.
- (d) Lemma 6.4.12 shows that this implies a valid schedule for each Tension gadget.

Now, assume that we are given a schedule \mathcal{S} for \mathcal{D}_φ . By applying Lemma 6.4.5 to the True Clock, we can assign coloured slots to \mathcal{S} . It then follows from the construction of \mathcal{D}_φ that \mathcal{S} must be slot-respecting. Set $v(x_i) = \text{True}$ if \mathcal{S} schedules 3_{iR} in red slots and $v(x_i) = \text{False}$ otherwise. Lemma 6.4.11 shows that for each $c_i \in C$, at least one input edge must be red for the output edge to be blue; Lemma 6.4.12 shows that all such output edges must be blue given the existence of \mathcal{S} . Thus, all clauses $c_i \in C$ evaluate to True under v , and the claim follows.

It remains to argue that our reduction can be realised in polynomial time. The size of the polycule \mathcal{D}_φ is clearly polynomial in the size of the formula, leaving the gadgets themselves. Variable, OR , and Tension gadgets have both constant size and constant number. D_6 duplicator gadgets are linear in size with respect to the number of 6_P and 6_G edges

consumed by the system as a whole, bounded by the number of Tension gadgets, hence by the number of clauses. Blue D_3 duplicator gadgets are linear with respect to the number of 3_B edges consumed by OR gadgets, i.e. the number of clauses. 3_R edges are consumed by both Tension gadgets and OR gadgets, but this still leaves the size of their D_3 duplicator gadgets bounded by the number of clauses and hence by the size of the formula. Thus, it is easy to implement our reduction in polynomial time. \square

* * * * *

Part II

Multiway Powersort

Chapter 7

Multiway Powersort

Outline This chapter presents *Multiway Powersort*: a stable Mergesort variant that exploits existing runs and finds nearly-optimal merging orders for *k-way* merges with negligible overhead. We will demonstrate that our 4-way Powersort implementation can achieve substantial speedups, both over standard 2-way Powersort and over other stable sorting methods without compromising the optimally run-adaptive performance of Powersort.

Section 7.1 is a general introduction to the topic, including a brief account of the development of adaptive sorting methods, a summary of results, and a review of relevant literature. Section 7.2 introduces key ideas and terminology: Shannon entropy, memory-transfers, and definitions pertaining to run-adaptive Mergesort. Section 7.3 defines k-way Powersort, shows examples of 2-way and 4-way Powersort, and proves several theoretical results. Section 7.4 describes our implementation and experiments, introduces the key idea of sentinels, and provides strong evidence for 3 hypotheses – compared with 2-way Powersort:

1. 4-way Powersort can yield significant performance improvements;
2. Scanned elements are a good predictor for this speedup;
3. 4-way Powersort halves the merge cost of 2-way Powersort.

Section 7.5 concludes the chapter with a summary.

7.1 Introduction

Sorting a list of items is one of the foundational problems of computer science and is studied in a wide variety of contexts, from sorting punch-cards in the early 1900s [Hol89; Hol17] to sorting databases today [Kuh+25]. Here we will focus on a simple modern context: internal comparison-based methods for sorting relatively simple objects using a single core on a single machine. Prominent sorting algorithms include Quicksort and Mergesort, for which highly engineered implementations exist. While Quicksort is typically faster, Mergesort has the optimal $\Theta(n \log(n))$ worst-case behavior and is stable, i.e., it preserves the relative order of elements that compare equal under the sorting criterion. Stability is a central requirement in many programming libraries, for example, the Java runtime library requires the sorting method for objects to be stable. Indeed, the need for a fast and stable general-purpose sorting method for the CPython reference implementation of the Python programming language motivated Tim Peters to develop a widely adopted variant of Mergesort, known as Timsort [Pet02].

Apart from its stability, Mergesort is also particularly well-suited for *adaptive sorting*, i.e., for saving effort when the input is already partially sorted (see also Section 7.2). Again, Timsort did pioneering work in bringing adaptive sorting to most modern standard libraries (including Python, Java, Android runtimes, the V8 Javascript engine, Rust, Swift, Apache Spark, Octave, and the NCBI C++ Toolkit), giving users linear complexity for sufficiently sorted inputs.

More specifically, Timsort takes advantage of existing *runs*: contiguous regions of the input that are already in sorted order. Timsort detects such runs on the fly and maintains a stack of runs yet to be merged. When the next run is found, a particular merge policy triggers a (potentially empty) sequence of merges before that new run is added to a stack; this merge policy determines which merges are included. Since the runs in the input can vary wildly in length, the chosen merge order can have a huge impact on efficiency. Consider as an indicative example an input with one run of length $n/2$ and $n/4$ runs of length 2. Repeatedly merging the long run with one small run would take $\Theta(n^2)$ overall time, whereas first merging all small runs into a single run (using a balanced tree) and then combining this run with the original long run is much cheaper: $\Theta(n \log n)$ time.

Despite its overwhelming success in practice, increased scrutiny revealed several issues with the original merge policy of Timsort: First, an algorithmic bug allowed the run stack to grow larger than anticipated, *twice* leaving widely deployed libraries vulnerable to memory

access violations when sorting adversarial inputs¹ [Gou+19; Aug+18]. Second, when compared with an optimal merge policy, Timsort incurs an overhead of up to 50% for certain inputs [BK19; Aug+18; Aug+18] with noticeable effects on running times [MW18]. These deficiencies have sparked a flurry of publications suggesting possible alternatives [Aug+18; MW18; BK19; Jug24], some with much stronger efficiency guarantees. After proving efficient in running time studies [Pet+18], Powersort [MW18] has now replaced Timsort’s original merge policy in CPython, PyPy, and AssemblyScript [Pet21].

Multiway algorithms have also seen recent adoption, with a surprising change in the Java runtime library’s unstable method for sorting primitive-typed arrays. There, a state-of-the-art implementation of classic Quicksort – the standard sort for decades – was replaced in 2011 by a novel dual-pivot variant after the latter proved 10% faster in practice [Wil12]. Detailed analysis revealed that the observed speedup is due to memory transfers: dual-pivot Quicksort profits from savings in “*scanned elements*”, the number of touched memory cells not in cache [Kus+13; ADK16; NWM16]. Since improvements in CPU speed have outpaced improvements in memory bandwidth for several decades, it now pays to avoid memory transfers in internal sorting, even at the expense of slightly increasing effort inside the CPU or running more complicated code [ADK16; Wil16]. Multiway partitioning in Quicksort and multiway merging in Mergesort do exactly that; by getting more of the sorting task done in one pass over the data, the overall volume of data transferred between the memory and the CPU is reduced. Abstract cost measures such as scanned elements and the less general “*merge cost*” (the total length of the output from a merge method) have proven appropriate to capture the corresponding effects.

We will use merge cost and scanned elements somewhat interchangeably because they will always be proportional in our algorithms and implementations; this is not, however, generally true. Merging a run of length a with a run of length b will always have a merge cost of $a + b$, but the number of scanned elements will depend on the merge method. a and b will both be scanned while being merged; our methods copy them both to a cache before merging, for a total of $2(a + b)$ scanned elements. If we copied only the shorter run to the cache like Timsort does, the number of scanned elements would instead be $a + b + \min(a, b)$, but this would add complexity and gives diminishing returns with multi-way merging.

While multiway merging is an elementary trick of the trade to reduce the memory-transfer cost in Mergesort, applying it to state-of-the-art adaptive sorting methods is a delicate task:

¹although Java is memory-safe, so attempts to overflow the allocated memory should only result in a failed sort

to not forsake the gains on partially sorted inputs, we have to generalize the policy for deciding which runs to merge. Indeed, in the case of Timsort it seems quite unclear how to utilize multiway merges; Timsort’s local rules are tailored to 2-way merges. Apart from Peeksort and Powersort, the same is true for all other practical methods suggested to replace Timsort, in particular adaptive Shivers Sort [Jug24] and α -MergeSort [BK19]. By contrast, Powersort is based on a principled construction, mimicking nearly-optimal binary search tree algorithms – this allows us to present a multiway generalization.

Powersort has two principal inspirations: Like Timsort, it scans the input from left to right, merging runs somewhat lazily by maintaining a logarithmic stack of merges delayed until their time is right. This substantially reduces the working memory requirements of these methods, and with it the rate of cache misses – improving performance. From the *Huffman-Merge* of Burge [Bur58], Golin & Sedgewick [GS93], Takaoka [Tak98], Barbay & Navarro [BN09], and Chandramouli & Goldstein [CG14] it takes the idea of conforming a merge tree to a well studied data structure (though it replaces Huffman trees with nearly-optimal binary search trees), as well as guarantees in terms of *run-length entropy* (defined in Section 7.2).

7.1.1 Results

In this chapter, we show how to use multiway merging in nearly-optimal Mergesort variants and that significant performance improvements result from that. More precisely, our contributions are as follows:

(1) We design k -way Powersort, a provably optimal generalization of Powersort which simultaneously merges k runs wherever this is possible. To our knowledge, this is the first run-adaptive multiway Mergesort variant presented in the literature. We prove that – just like standard 2-way Powersort – k -way Powersort uses $\leq n \cdot \mathcal{H} + 3n$ comparisons to sort a list of n elements, where \mathcal{H} is the run-length entropy when k is a power of two (Theorem 7.3.3); this is optimal in the worst case. We furthermore show that the asymptotic worst-case merge cost for k -way Powersort is a factor $\log_2(k)$ smaller than for 2-way Powersort. Here the merge cost counts the total number of output elements of all merges conducted by an algorithm and thus approximates the amount of memory transfers in Mergesort variants.

(2) We engineer an efficient implementation of 4-way Powersort in C++ and demonstrate in an extensive running time study that it realizes performance improvements between 15% and 20% over standard 2-way Powersort. This observation is robust against changes of

the size of elements to be sorted (we compare both integer and record data) and different amounts of presortedness of the input. Besides wall-clock running times, we determine scanned elements and cache performance in `valgrind/cachegrind` to explain the observed speedup.

(3) We confirm that the results reported in [MW18] for Java implementations of the algorithms are reproducible in a low-level implementation.

7.1.2 Related Work

Adaptive sorting algorithms are those which exploit some kind of structure in the input to improve their performance. They date back to methods from the 1950s [Bur58] that sort faster when the number of inversions is small. A systematic treatment of adaptive sorting algorithms, including several other measures of presortedness (e.g., the number of inversions, the number of runs, and the number of shuffled up-sequences), the relationship between them, and how to sort *adaptively* w.r.t. these measures is discussed by Estivill-Castro and Wood [EW92].

In addition to introducing Powersort and Peeksort, Munro and Wild give a more up-to-date survey of adaptive sorting and tell the story of Timsort in their 2018 paper [MW18]. Since then, Buss and Knop [BK19] pointed out the suboptimal worst-case of Timsort’s merge policy, and several papers suggested alternatives [Aug+18; Jug24; MW18]. The most noteworthy of these is Adaptive Shivers Sort [Jug24], which shares many favorable properties with Powersort and proves the existence of a merge policy that is optimal up to an $O(n)$ term in the theoretical model of Buss and Knop [BK19]. It is not clear whether Shivers Sort can easily be generalized to multiway merging.

Another key paper studies Timsort’s galloping merge method in detail: Ghasemi et al. [Gha+25] show that most Mergesort variants can sort in $\mathcal{H}^*n + O(n)$ comparisons when using a galloping merge, where \mathcal{H}^* is the “dual-run-length entropy” (which is at most the classical entropy of multiplicities of equal elements). Note that for sorting arrays, galloping only saves on comparisons; one still has to move elements around. The improvements obtainable from galloping merges are thus largely orthogonal to our work.

7.2 Preliminaries

If considering a bottom-up approach for Mergesort, one can take advantage of continuous regions of the input that are already sorted. We call such a region a *run* and define both maximal weakly increasing (i.e., non-decreasing), and strictly decreasing regions as runs – the latter can easily be reversed without causing stability issues. We will always denote the number of runs in the input by r and their respective lengths by L_0, \dots, L_{r-1} , such that $L_0 + \dots + L_{r-1} = n$ holds. From an algorithmic point of view, the unique partitioning of the input into sorted segments (runs) can be found by an initial scan of the input.

7.2.1 Lower Bound

The binary Shannon entropy \mathcal{H} often shows up in lower bounds for the cost of sorting: For $p_1, \dots, p_m \in (0, 1)$ with $p_1 + \dots + p_m = 1$ we define $\mathcal{H}(p_1, \dots, p_m) = \sum p_i \lg(1/p_i)$, where here and throughout $\lg = \log_2$.

Theorem 7.2.1 [BN13, Thm. 2].

Sorting a list of n elements with initial runs of lengths L_0, \dots, L_{r-1} requires $\mathcal{H}(\frac{L_0}{n}, \dots, \frac{L_{r-1}}{n})n - O(n)$ comparisons in the worst case.

Where the run lengths are clear from the context, we will use \mathcal{H} to abbreviate $\mathcal{H}(\frac{L_0}{n}, \dots, \frac{L_{r-1}}{n})$.

7.2.2 Merging and its Memory-transfer Cost

Merging describes the process of combining two or more sorted lists (i.e., runs) into a single sorted list containing the same elements, by means of pairwise comparisons of these elements. All of our merge methods retain the input runs in a buffer area and produce the merged output in another area; in-place methods exist, but their time overhead renders them uncompetitive for our purposes. Important metrics for merging methods include the number of key comparisons and the number of *scanned elements*, i.e., memory accesses (read or write) that are not served from cache [NWM16].

Merging can be achieved by iteratively finding the minimum among the smallest remaining elements of the runs and moving it to the output. When merging k input runs at once, maintaining a *tournament tree* [Knu98] of the run minima (initialized using $k - 1$ comparisons) allows us to find the next output element after $\lceil \lg(k) \rceil$ or $\lfloor \lg(k) \rfloor$ comparisons. This lets us merge k runs with n total elements using at most $\lceil \lg(k) \rceil n + k - 1$ comparisons; unless the

input runs are of very different lengths, this strategy is essentially optimal (for worst-case comparison cost). When the input runs already reside in the buffer area, we can assume that each element is “scanned” exactly twice (fetched once and output once); it can be read and compared many times, but these accesses can be assumed to be cached (for moderate k).

7.2.3 Run-adaptive Mergesort

Mergesort operates by successively merging runs until only a single one remains. To get started, classic top-down Mergesort considers each individual element as a trivial run. An *adaptive* (i.e., *run-adaptive*) Mergesort instead finds contiguous segments of the input that are already in order.

Since both scanned-element count and comparison count are linear in the output size of a merge (for fixed k and sufficiently long runs), a convenient abstract measure to compare different Mergesort variants is the *merge cost* [BK19], defined as the total size of the outputs of all the merges.

Let R_0, \dots, R_{r-1} be the runs in the input, recalling that L_i denotes the length of R_i . Any stable run-adaptive Mergesort can be described as a *merge tree* T : an ordered, rooted tree with leaves R_0, \dots, R_{r-1} (appearing in that order from left to right). Each internal node v of T corresponds to the result of merging its children; we write $M(v)$ for the total length of the result of that merge. The total merge cost of T is then $M = M(T) = \sum_{v \in T} M(v)$. T is a k -way merge tree if all its nodes have degree at most k . Note that it is not generally possible to have all merge-tree nodes combine exactly k runs, therefore a k -way merge tree is allowed to contain nodes of smaller degree (though we do not allow unary nodes).

We further denote by B_i the boundary between the $(i-1)$ st and i th run ($i = 1, \dots, r-1$). In binary merge trees ($k = 2$), there is a one-to-one correspondence between the B_i and the (inner/non-leaf) merge-tree nodes; for multiway merging, this becomes many-to-one: each merge-tree node v of degree d is associated with exactly $d-1$ (not necessarily adjacent) run boundaries $B(v, 1), \dots, B(v, d-1)$; conversely each run boundary B_i is assigned to exactly one merge-tree node $m(B_i)$.

Let d_i be the *depth* of leaf R_i in T , where depth is the number of edges on the path to the root. Every element in run R_i is counted exactly d_i times in $\sum_v M(v)$, so we have $M = \sum_{i=0}^{r-1} d_i \cdot L_i$.

7.3 Multiway Powersort

We now introduce Powersort’s merge policy for general k -way merging. We expressly include the special case $k = 2$ here; the resulting merge policy is equivalent to Powersort from [MW18].

7.3.1 Intuition and the Merge Tree

Let $k \geq 2$ be fixed, and first assume for ease of presentation that $n = k^m$, $m \in \mathbb{N}$. The intuition behind Powersort is to imagine a complete k -ary tree T_V sitting on top of the array, where each element $A[i]$ of the input corresponds to a leaf of T_V . T_V corresponds to the merge tree of a non-adaptive k -way Mergesort (starting with individual elements). Now each run boundary B_j also corresponds to an (inner) node w of this (virtual) k -ary tree T_V , namely the lowest common ancestor of the two leaves adjacent to B_j .

We cannot follow T_V if we want to retain all existing runs, as most will usually straddle node boundaries; instead, we will use the depths d_w of the nodes of T_V as a guide: Any non-leaf node w in T_V splits a range of the array into k segments. At the $k - 1$ boundaries between these segments, we draw a vertical bar of height $m - d_w$ (see Figure 7.1). Then, we assign each boundary B_j its *midpoint interval*: the horizontal interval from the middle of R_{j-1} (exclusive) up to the middle of R_j (inclusive). Midpoint intervals are shown in blue in Figure 7.1. Now, the *power* P_j of B_j is defined as the depth d_w of the node w contributing the tallest vertical bar inside B_j ’s midpoint interval (Definition 7.3.1).

Since the midpoint intervals are disjoint, each vertical bar is assigned at most one B_j , and so each node w in T_V is assigned at most $k - 1$ run boundaries. These $k - 1$ run boundaries define one merge-tree node $v(w)$; the collection of $v(w)$ forms the k -way merge tree T of Powersort. Intuitively, T is a good merge tree since it uses boundaries as close to the perfectly balanced T_V as is possible while respecting existing runs.

Note that going from 2-way powers to 4-way powers as in Figure 7.1 corresponds to a simple transformation: We “squish” pairs of adjacent levels of 2-way powers into one layer of 4-way powers: $P_i^{(4)} = \lfloor (P_i^{(2)} - 1)/2 \rfloor + 1$. We thus obtain the merge tree of 4-way Powersort from the one for 2-way Powersort by letting all nodes at even depth absorb their children (and adopt their grandchildren and great grandchildren).

When n is not a power of k , the virtual tree T_V does not align with element boundaries, but we can treat the array and its runs as *continuous* intervals, working in the same way. An example of this is shown in Figure 7.2. Formally, we map the array to the unit interval,

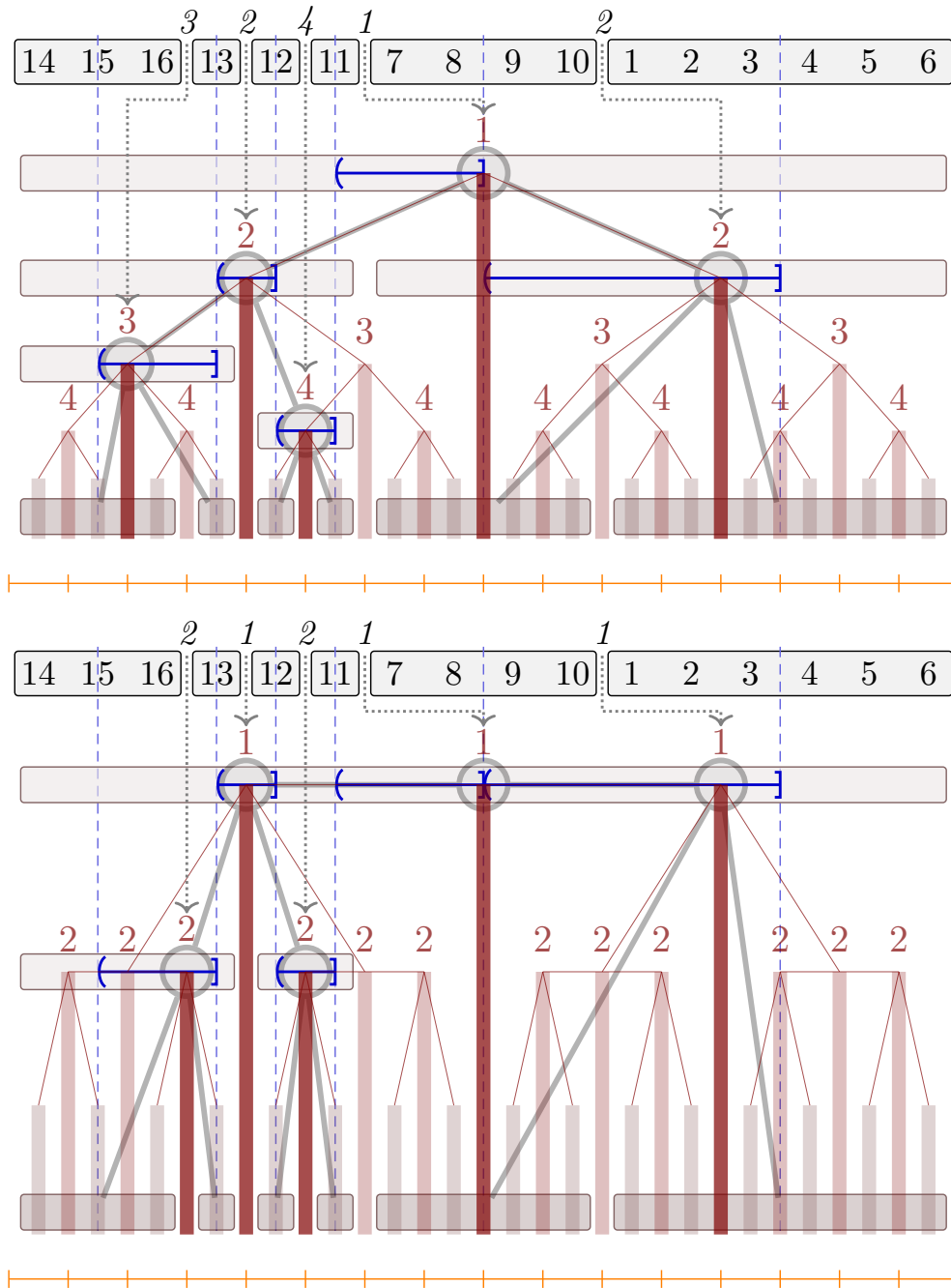


Figure 7.1: Example merge tree for Powersort (top) and 4-way Powersort (bottom) for an input of size $n = 16$: Run boundaries are mapped (grey arrows) to nodes of a (virtual) perfectly balanced binary resp. 4-ary tree T_V (shown in light red), which has the array elements as its leaves. The midpoints of two adjacent runs form the horizontal range in which this node is allowed to move; it then “snaps” to the highest pole in that range.

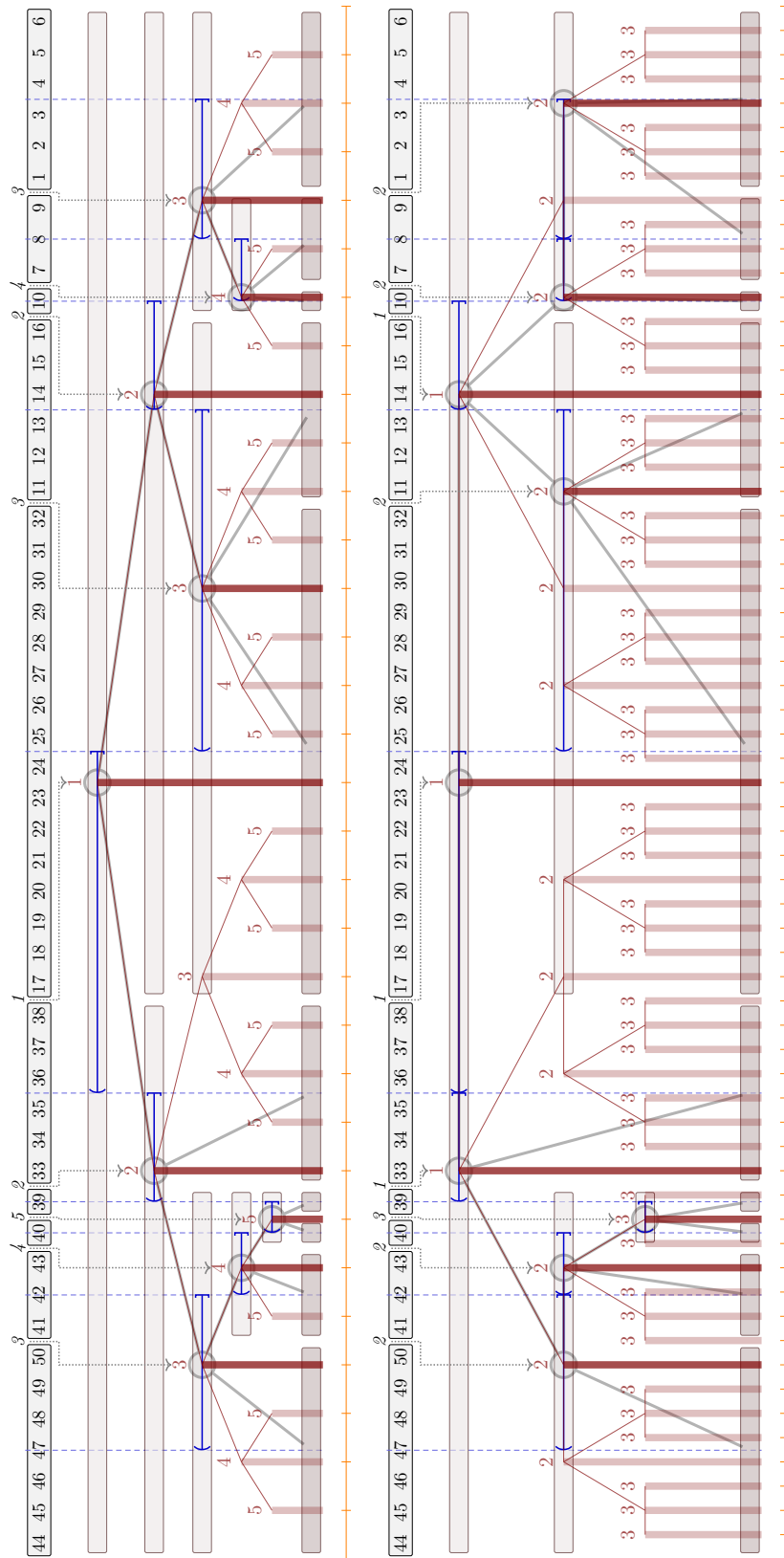


Figure 7.2: Example merge tree for Powersort (top) and 4-way Powersort (bottom) for an input of size $n = 50$: Here, merges are “rounded” to nodes in a (virtual) perfectly balanced binary resp. 4-ary tree T_V (shown in light red); drawing style as in Figure 7.1.

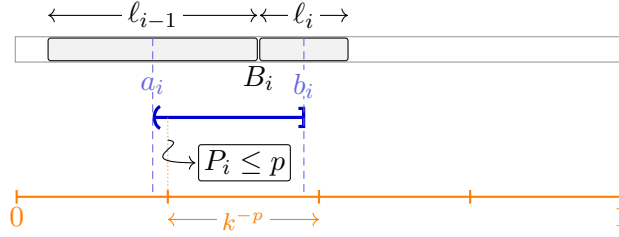


Figure 7.3: Illustration of the power of a run boundary. Since there is an orange tick inside the midpoint interval, we obtain $P_i^{(k)} \leq p$; in the example, the midpoint interval is $(a_i, b_i] = (0.225, 0.475]$, $k = 4$ and $p = 1$, so $P_i^{(4)} = 1$.

partitioned by the runs to define the power of run boundaries:

Definition 7.3.1 Run Boundary Power.

Let $k \geq 2$ be fixed. For run lengths L_0, \dots, L_{r-1} , set $\ell_i = L_i/n$. For $1 \leq i < r$, let B_i be the run boundary between the $(i - 1)$ st and i th run. The $(k$ -way) power of B_i is

$$P_i^{(k)} = \min \left\{ p \in \mathbb{N} : \lfloor a_i \cdot k^p \rfloor < \lfloor b_i \cdot k^p \rfloor \right\},$$

where $a_i = \sum_{j=0}^{i-1} \ell_j - \frac{1}{2}\ell_{i-1}$, $b_i = \sum_{j=0}^{i-1} \ell_j + \frac{1}{2}\ell_i$.

We drop the superscript when k is clear from context. Figure 7.3 illustrates Definition 7.3.1. Intuitively, a_i is the normalized distance between the left hand side of the input and the middle of the run *before* B_i and b_i is the normalized distance between the left hand side of the input and the middle of the run *after* B_i .

Based on (just) P_1, \dots, P_{r-1} , we can recover the merge tree T by recursively finding all occurrences of the smallest occurring power and using these run boundaries as the splitting points for recursion. The pseudocode in Algorithm 7.1 makes this concrete.

This is just a conceptual algorithm; Multiway Powersort will produce the same merge tree as $\text{KWAYTREE}_k(A[0..n])$, but without having to search for minimal powers explicitly or even knowing all runs and powers at any point in time (as described below). KWAYTREE_k makes it obvious that P_j is the “intended” depth of $m(B_j)$ since we combine nodes recursively by increasing power. Nodes can end up *higher* in the merge tree if not all power values occur. Since we count edges for the depth, but powers start at 1, this yields the following useful inequality:

Fact 7.3.2.

$$\text{depth}(m(B_j)) \leq P_j - 1.$$

```

KWAYTREEk(A[b..e])
1  Let  $s_0 = b, s_1, \dots, s_{r-1}, s_r = e$  be the start indices of runs in  $A[b..e]$ 
2  Let  $P_1, \dots, P_{r-1}$  be the boundary powers
3   $\{i_1, \dots, i_m\} = \arg \min\{P_i : i \in [1..r]\}$ 
4   $i_0 = s_0; i_{m+1} = s_r$ 
5  for  $j = 0, \dots, m$ 
6      KWAYTREEk(A[ $i_j..i_{j+1}$ ]) // recurse
7  end for
8  MERGE(A[ $i_0..i_1$ ],  $\dots$ , A[ $i_m..i_{m+1}$ ])

```

Algorithm 7.1: Conceptual algorithm to compute the merge tree of k -way Powersort.

A simple proof by induction shows that in each recursive call of $\text{KWAYTREE}_k(A[0..n])$, we have $m \leq k - 1$, so we are merging at most k runs at once.

7.3.2 k -way Powersort

Powersort is presented in Algorithm 7.2. Like Timsort, it processes the merge tree from left to right. We maintain a stack S of runs and powers, subject to the invariant that powers are weakly increasing in S (from bottom to top). When the next run boundary B_i has a power P_i larger than or equal to $S.top()$, we push (R_{i-1}, P_i) onto S and continue. Otherwise, we merge R_i with *all runs* on S with power *equal* to $S.top()$; this is a key difference from standard Powersort, where we always merge with $S.top()$. We repeatedly perform such merges until we can push (R_{i-1}, P_i) onto S without violating the invariant.

The definition of k -way powers ensures that there are at most $k - 1$ equal powers on S at any time, so all merges executed in the above process combine at most k runs. Once all run boundaries have been treated that way, the remaining stack is merged top to bottom (similar to Timsort).

7.3.3 Analysis

We state a few properties about Multiway Powersort, most importantly a bound on its merge cost. The case where $k = 2$ has been covered in [MW18], but the following self-contained proof is arguably more direct:

Theorem 7.3.3.

The merge cost of k -WAY POWERSORT is $M \leq \frac{1}{\lg(k)} \mathcal{H}(\frac{L_0}{n}, \dots, \frac{L_{r-1}}{n})n + 2n$ and the number of comparisons $C \leq \frac{\lceil \lg k \rceil}{\lg k} \mathcal{H}(\frac{L_0}{n}, \dots, \frac{L_{r-1}}{n})n + (1 + 2 \frac{\lceil \lg k \rceil}{\lg k})n + (k - 1)r$.

```

KWAYPOWERSORTk(A[0..n])
1  S := empty stack // capacity (k - 1)⌈logk(n) + 1⌉
2  b1 := 0
3  e1 = FIRSTRUNOF(A[b1..n])
4  while e1 < n
5      b2 := e1
6      e2 := FIRSTRUNOF(A[b2..n])
7      P := POWERk(n, b1, e1, b2, e2)
8      while S.top().power > P
9          P' := S.top().power
10         L := empty list
11         L.append(S.pop())
12         while S.top().power == P'
13             L.append(S.pop())
14         end while
15         // merge runs in L with A[b1..e1]
16         (b1, e1) := MERGE(L, A[b1..e1])
17     end while
18     S.push(A[b1, e1], P)
19     b1 := b2
20     e1 := e2
21 end while // Now A[b1..e1] is the rightmost run
22 while ¬S.empty()
23     // pop (up to) k - 1 runs, merge with A[b1..e1]
24     (b1, e1) := MERGE(S.pop(k - 1), A[b1..e1])
25 end while

```

```

POWERk(n, b1, e1, b2, e2)
1  n1 := e1 - b1; n2 := e2 - b2; p := 0
2  a := (b1 + n1/2)/n; b := (b2 + n2/2)/n
3  while ⌊a · kp⌋ == ⌊b · kp⌋ do p := p + 1 end while
4  return p

```

Algorithm 7.2: Multiway Powersort pseudocode. The function FIRSTRUNOF finds the leftmost run in an array and returns its endpoint.

Proof. We actually show $M/n \leq \sum_{i=0}^{r-1} \ell_i \lceil \log_k(1/\ell_i) + 1 \rceil$, which implies the claim. We start with the expression from Section 7.2.3 for the merge cost: $M = \sum_{i=0}^{r-1} d_i \cdot L_i$ where d_i is the depth of leaf R_i in T . Since R_i is a leaf, it must be the (direct) child of either $m(B_i)$ or $m(B_{i+1})$ in T and hence $d_i = 1 + \max\{\text{depth}(m(B_i)), \text{depth}(m(B_{i+1}))\}$; by formally setting $\text{depth}(m(B_0)) = \text{depth}(m(B_r)) = 0$, this equality holds for all $i = 0, \dots, r-1$. Now, Fact 7.3.2 yields $d_i \leq \hat{P}_i$, $i = 0, \dots, r-1$, where $\hat{P}_i := \max\{P_i, P_{i+1}\}$ and we similarly set $P_0 = P_r = 0$. So we have $M/n \leq \sum_{i=0}^{r-1} \ell_i \hat{P}_i$.

We will now show that $k^{-p} \leq \ell_i/2$ implies $P_i^{(k)} \leq p$. It is easy to see that with a_i and b_i as in Definition 7.3.1, we have $P_i^{(k)} = \min\{p : \exists c \in \mathbb{N} : c \cdot k^{-p} \in (a_i, b_i]\}$; (cf. Figure 7.3). It suffices to note that $|(a_i, b_i]| \geq \ell_i/2$ and whenever $k^{-p} \leq |(a_i, b_i]|$ is given, $(a_i, b_i]$ clearly contains a value ck^{-p} (an orange “tick” in Figure 7.3). Note that since $|(a_{i+1}, b_{i+1})| \geq \ell_i/2$ similarly holds, the above argument also shows that $k^{-p} \leq \ell_i/2$ implies $P_{i+1}^{(k)} \leq p$.

Now, $k^{-p} \leq \ell_i/2$ iff $p \geq \log_k(1/\ell_i) + 1$, so setting $p = \lceil \log_k(1/\ell_i) + 1 \rceil$ implies that condition and hence $P_i^{(k)} \leq \lceil \log_k(1/\ell_i) \rceil + 1$ as well as $P_{i+1}^{(k)} \leq \lceil \log_k(1/\ell_i) \rceil + 1$; hence $\hat{P}_i \leq \lceil \log_k(1/\ell_i) \rceil + 1$. Inserting this into $M/n \leq \sum_{i=0}^{r-1} \ell_i \hat{P}_i$ gives $M/n \leq \sum_{i=0}^{r-1} \ell_i \lceil \log_k(1/\ell_i) + 1 \rceil$ as claimed.

For the number of comparisons, recall that the tournament-tree merge uses $\leq \lceil \lg(k) \rceil$ comparisons per output element, except for the first output element of each merge, where we initialize the tournament using $k-1$ comparisons. Let μ be the number of merges; the exact number depends on the merge policy, but $\lceil \frac{r-1}{k-1} \rceil \leq \mu \leq r-1$. Including the $n-1$ comparisons to find runs, we overall spend $C \leq M \cdot \lceil \lg k \rceil + \mu \cdot (k-1 - \lceil \lg k \rceil) + n-1$. Inserting the bound for M , we obtain the claimed $C \leq \frac{\lceil \lg k \rceil}{\lg k} \cdot (\mathcal{H}n + 2n) + (k-1)r + n$ comparisons. \square

Remark 7.3.4.

We note that the analysis of C could be slightly sharpened, but we are mostly interested in the case where k is a small power of two; there, the above bound is tight up to lower order terms.

Remark 7.3.5.

For $k = 2$, the bounds simplify to Thm. 5 of [MW18], for which we above give an alternative proof that is entirely self-contained and elementary.

Proposition 7.3.6.

The maximal stack height for k -way Powersort is $(k - 1)\lceil \log_k(n) + 1 \rceil$.

Proof. We use that $P_i \leq \lceil \log_k(1/\ell_i) + 1 \rceil$ (see above); since $\ell_i \geq 1/n$, we obtain $1 \leq P_i \leq \lceil \log_k(n) + 1 \rceil$. So there are $\lceil \log_k(n) + 1 \rceil$ different possible values for P_i . The run stack S is weakly increasing and can contain at most $k - 1$ entries with equal power, so $|S| \leq \lceil \log_k(n) + 1 \rceil(k - 1)$. \square

7.3.4 k-way Peeksor

Peeksor, the other algorithm from [MW18], can also be generalized to k -way merges, as shown in Algorithms 7.3 and 7.4. This is a top-down version of multiway Mergesort, producing a similar merge tree to multiway Powersort.

2-way Peeksor uses a single midpoint and finds the run boundary closest to it; instead, we use $k - 1$ equidistant midpoints and use their closest run boundaries to bound further recursive calls. These midpoints are scanned from left to right, recursing as necessary, with the start of each run stored in a merge stack S until the end of the pass. In 2-way Peeksor, known initial or final runs can cover the whole input, contain the midpoint, or neither; likewise, discovered runs can cover the whole input, be left-aligned, or be right-aligned.

Each case of 2-way Peeksor has an analogous case in the k -way version, but the extra midpoints add several subtleties: If the known initial run $A[l..e]$ or known final run $A[s..r]$ contains a midpoint, then they must be at least $\lfloor \frac{r-l}{k} \rfloor$ long, and we treat them as resolved regions by sending them to the merge stack S (the cases LONGINITIAL and LONGFINAL of Algorithm 7.4). If we discover a run containing multiple midpoints, this is at least as long and should be treated in the same way (lines 5-13 of LEFTALIGNED, again in Algorithm 7.4).

We expect that k -way Peeksor will give similar analytical guarantees to k -way Powersort, but will have worse practical performance due to the increased overheads inherent to top-down algorithms:

Conjecture 7.3.7.

The merge cost of k -WAY PEEKSORT is $M \leq \frac{1}{\lg(k)} \mathcal{H}(\frac{L_0}{n}, \dots, \frac{L_{r-1}}{n})n + \mathcal{O}(n)$ and the number of comparisons $C \leq \frac{\lceil \lg k \rceil}{\lg k} \mathcal{H}(\frac{L_0}{n}, \dots, \frac{L_{r-1}}{n})n + \mathcal{O}(n, k, r)$.

```

KWAYPEEKSORTk(A[l..r], e, s)
1  if e == r or s == l then return
2  for i ∈ [1..k)
3      mi := l + i⌈ $\frac{r-l}{k}$ ⌉      // the k - 1 midpoints
4  end for
5  S := empty stack      // capacity k
6  S.push(l)
7  c := e
8  i := 1
9  if mi ≤ e
10     S, c, i := LONGINITIAL(e, m, S, c, i)
11 end if
12 while i < k
13     if mi ≥ s
14         LONGFINAL(A, r, s, S, c)
15     return
16 end if
    // discover the run A[a..b] containing index mi
17     a := EXTENDRUNLEFT(A, mi, l)
18     b := EXTENDRUNRIGHT(A, mi, r)
19     if a == l and b == r then return
20     if mi - a ≤ b - mi
21         S, c, i := LEFTALIGNED(A, r, m, S, a, b, c, i)
22     else
23         S, c, i := RIGHTALIGNED(A, r, S, a, b, c, i)
24     end if
25 end while
26 KWAYPEEKSORTk(A[S.top..r], c, s)
27 MERGE(A, S, r)

```

Algorithm 7.3: Multiway Peeksort pseudocode. $A[l..r]$ is the domain, $A[l..e]$ is the known first run, and $A[s..r]$ is the known last run; the initial call is $\text{KWAYPEEKSORT}_k(A[1..n], 1, n)$. S contains the first index of each run waiting to be merged, while c is the last character of the known first run for the next recursive call (becoming the e for that call). Methods for handling nontrivial runs are shown in Algorithm 7.4; `LONGINITIAL` and `LONGFINAL` handle long runs (those which contain multiple midpoints), while `LEFTALIGNED` and `RIGHTALIGNED` handle runs whose midpoints are closer to the left or right edge respectively.

LONGINITIAL(e, m, S, c, i)

```
1 S.push( $e + 1$ )
2  $c := e + 1$ 
3 while  $m_i \leq e$ 
4      $i := i + 1$ 
5 end while
6 return  $S, c, i$ 
```

LONGFINAL(A, r, s, S, c)

```
1 KWAYPEEKSORT $k$ ( $A[S.top..s - 1], c, s - 1$ )
2 S.push( $s$ )
3 MERGE( $A, S, r$ )
```

LEFTALIGNED(A, r, m, S, a, b, c, i)

```
1 KWAYPEEKSORT $k$ ( $A[S.top..a - 1], c, a - 1$ )
2 S.push( $a$ )
3  $c := b$ 
4  $i := i + 1$ 
5 if  $m_i \leq b$            // handle a long discovered run
6     if  $b < r$ 
7         S.push( $b + 1$ )
8          $c := b + 1$ 
9     end if
10    while  $m_i \leq b$ 
11         $i := i + 1$ 
12    end while
13 end if
14 return  $S, c, i$ 
```

RIGHTALIGNED(A, r, S, a, b, c, i)

```
1 KWAYPEEKSORT $k$ ( $A[S.top..b], c, a$ )
2  $c := b$ 
3  $i := i + 1$ 
4 if  $b + 1 \leq r$ 
5     S.push( $b + 1$ )
6      $c := b + 1$ 
7 end if
8 return  $S, c, i$ 
```

Algorithm 7.4: pseudocode for functions which handle four cases encountered in Algorithm 7.3, which implements Multiway Peeksor.

7.4 Results

This section describes our empirical results around Multiway Powersort.

7.4.1 Implementations and Compilation

We have implemented 2-way and 4-way Powersort in C++ and engineered both for performance under the GNU Compiler Collection. We also compare them to a few standard algorithms including `std::sort` and `std::stable_sort` from the GNU implementation of the C++ Standard Library. Key parts of the code are shown in Appendix B, and the full implementations, including instructions to reproduce our running-time study, are available on GitHub: <https://github.com/sebawild/powersort>. We comment on a few implementation details that are relevant for the running-time comparisons.

Tournament tree For 4-way merging, we use a tournament tree with 3 internal nodes. We represent it as a winner tree, i.e., each node stores the run contributing the smaller element among its two children; more specifically, each node stores a pointer to the first element of a run that has not been output yet. The root additionally stores whether the minimum there came from the left or right subtree. That way each element moved to the output entails 2 tournaments (node recomputations), and we know which ones these are. Since our tournament tree is small and will entirely reside in registers, directly accessing the two children of a node for playing a match is very inexpensive, so there seems to be little benefit from using a loser tree instead of the simpler winner tree. Whenever an element of a run enters the tournament tree (because it has won against its sibling run), we already advance the pointer for the leaves of the tournament tree; this decouples the advancement of the pointer from writing the output.

Sentinel values We implemented merging methods that assume the availability of a value that is strictly larger than any element in the list to be sorted, usually $+\infty$. Such values are often available: floating-point types directly support them, integer types may require that the maximum value is reserved for this use, while strings and characters may be able to use the null character. We can place such a sentinel value at the end of runs to avoid a pointer comparison in the inner loop of the merging methods, which saves a few instructions and branches per iteration. Below, we report on results for algorithms with and without

sentinels, and we also propose a method for extending the range of keys with which sentinels can be used in Section 8.1.4.

Sentinel-free merging by stages When a type does not (efficiently) support a $+\infty$ value, it can be beneficial to have a merge method that does not use sentinels. For 2-way Powersort, a trivial such merge implementation is only slightly slower overall than using sentinel-based 2-way merge. For 4-way merging using tournament trees, there are many more cases to distinguish (which runs are empty), making a trivial implementation inefficient. We combine two tricks to improve upon that: First, we merge in “stages” where a stage ends whenever one run is exhausted. This allows us to reduce the number of cases to distinguish in the code (at the expense of longer code). Second, if s is the length of the shortest remaining (nonempty) run, we can clearly do at least s iterations without checking boundaries, after which we recompute s . When s reaches 0, we know that the current stage is over. “4-way Powersort (no sentinel)” uses merging by stages.

Buffer for runs All our merge methods have an in-place interface, i. e., they produce the merged result in the array initially occupied by the input runs (and assume those to be adjacent), but they use a linear-size buffer for efficient merging. Most methods copy all runs to the buffer (which tends to give the fastest code for the merging phase), but we also include a 2-way method that only copies the smaller run to the buffer and merges “into the gap” (a standard trick used, e. g., in Timsort). This method does not use sentinels.

Minimal run length All our Mergesort variants use Insertionsort on subproblems of size ≤ 24 ; for Timsort-based methods like Powersort, this means that when a new run is found with fewer elements, it is extended to 24 elements.

7.4.2 Experimental Setup

The experiments were run on a PC running Ubuntu 20.04.4 LTS. The CPU has 4 cores running 2 hyperthreads each and 16 GB of main memory. The model of the CPU is Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz. Each core has a private level-1 cache with 32KB for data and 32KB for instructions and a unified level-2 cache of 256KB. Moreover, there’s a shared level-3 cache of 8MB. All caches are organized into 64B cache lines and are 8-way associative; the level-3 cache is 16-way associative.

The experiments use `g++` from the GNU Compiler Collection 9.4.0, specifically Ubuntu 9.4.0-1ubuntu1 20.04.1, with optimization flags `-Ofast -march=native -mtune=native`. (Results did not change noticeably when using `-O3` instead.)

7.4.3 Hypothesis 1: 4-way Powersort can yield significant performance improvements

To test this hypothesis, we conducted a running-time study varying several dimensions to investigate the speedup obtainable from 4-way merging in Powersort.

As an indicative first scenario, Figure 7.4 shows normalized running times for sorting a mildly presorted list of `ints` (as in [MW18]): across four orders of magnitude of input sizes, 4-way Powersort is consistently around 20% faster. Figure 7.5 shows that the distribution of running times is concentrated around the mean, showing a robust difference. Figure 7.4 also shows that adaptive mergesort methods outperform the Quicksort-based `std::sort`.

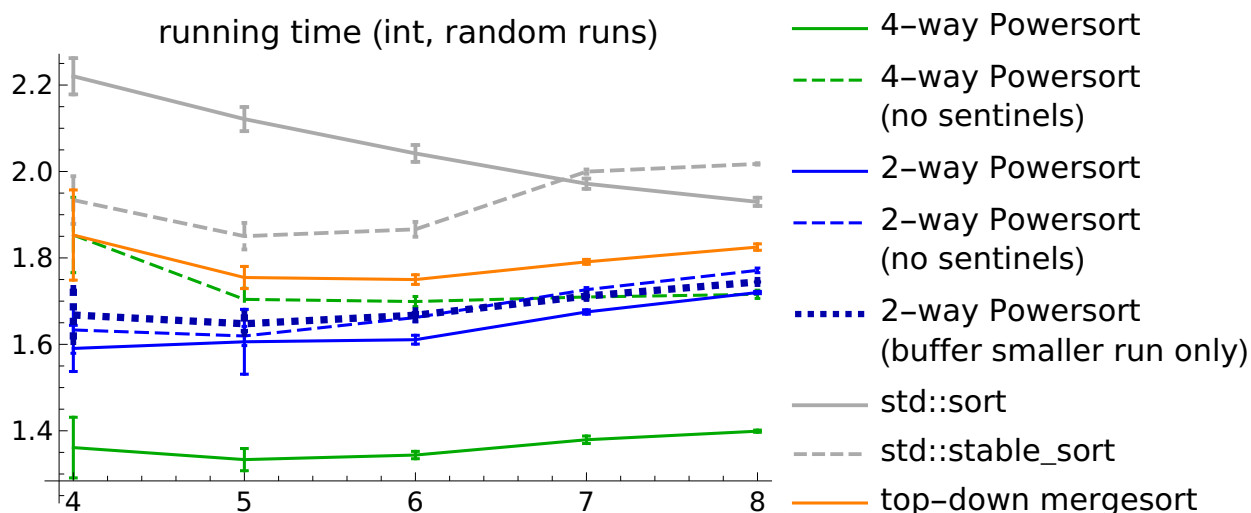


Figure 7.4: Normalized **running times** for sorting `ints` (4 byte signed integers) where the input has **random** initial **runs** with a geometric distribution and expected length \sqrt{n} . The x -axis shows $\log_{10}(n)$, the y -axis shows average running time in ms multiplied by $10^6/n \lg n$; error bars show one standard deviation. We used 1000 repetitions up to 10^6 elements and 100 repetitions for 10^7 and 10^8 elements.

The random-runs inputs have substantial sorted parts; in expectation \sqrt{n} runs with an expected \sqrt{n} elements each, albeit with substantial variation. However, the lower bound $n\mathcal{H}$ approaches $\frac{1}{2}n \lg(n)$, so they are still far from fully sorted; in particular, sorting these still has linearithmic complexity (as clearly visible in Figure 7.4).

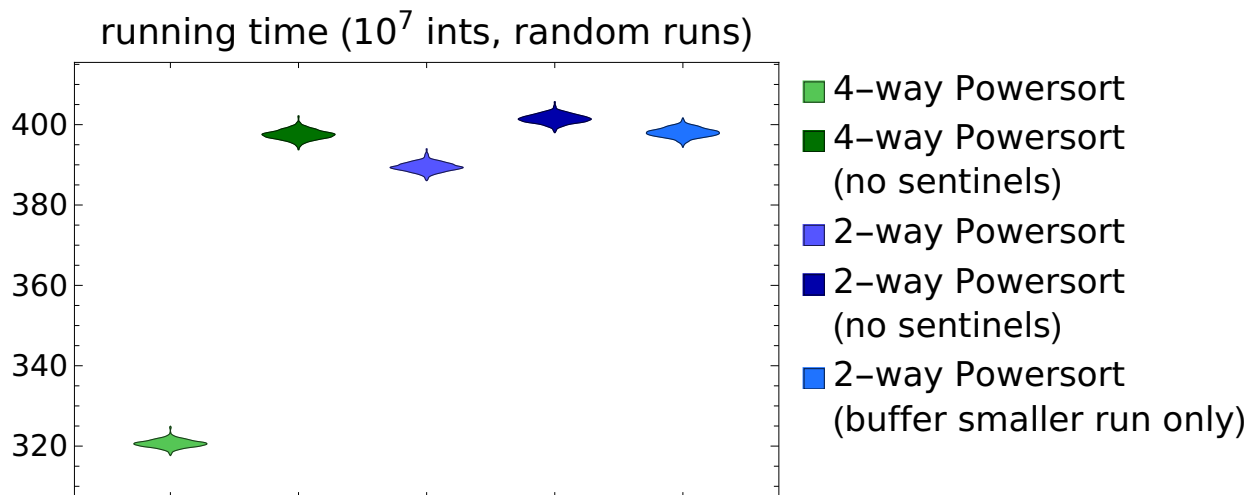


Figure 7.5: Distribution of running times for $n = 10^7$ from Figure 7.4 (sorting `ints` with random runs).

A first variation considers a more realistic data type than integers: Stable sorting is most relevant for sorting objects that have various properties / instance variables and which are (partially) sorted according to one of its properties. We model this by sorting records containing a `long` key and a pointer. Figure 7.6 shows the respective running times.

Since elements are now 4 times larger (16 vs. 4 bytes), it is not surprising that all methods are slower than for `ints`, but they are not uniformly so. 4-way Powersort with the sentinel-free merging by stages was roughly as fast as 2-way Powersort, which in turn had essentially equal performance under all three considered 2-way merge methods. Note that both the 4-way method and the 2-way method that copies only the smaller run to a buffer substantially outperform the other 2-way Powersort variants. 4-way Powersort with sentinel-based merge remains substantially faster still (between 15 and 20% across the studied input-size range).

Finally, we vary the input distribution. In particular, any adaptive sorting method should remain competitive with the best general purpose methods when no significant presortedness exists to be exploited. Figure 7.7 shows running times for random permutations. Somewhat surprisingly, even where effectively no presortedness exists, 4-way Powersort is still 15–20% faster than the fastest 2-way stable sorting method and almost as fast as `std::sort`.

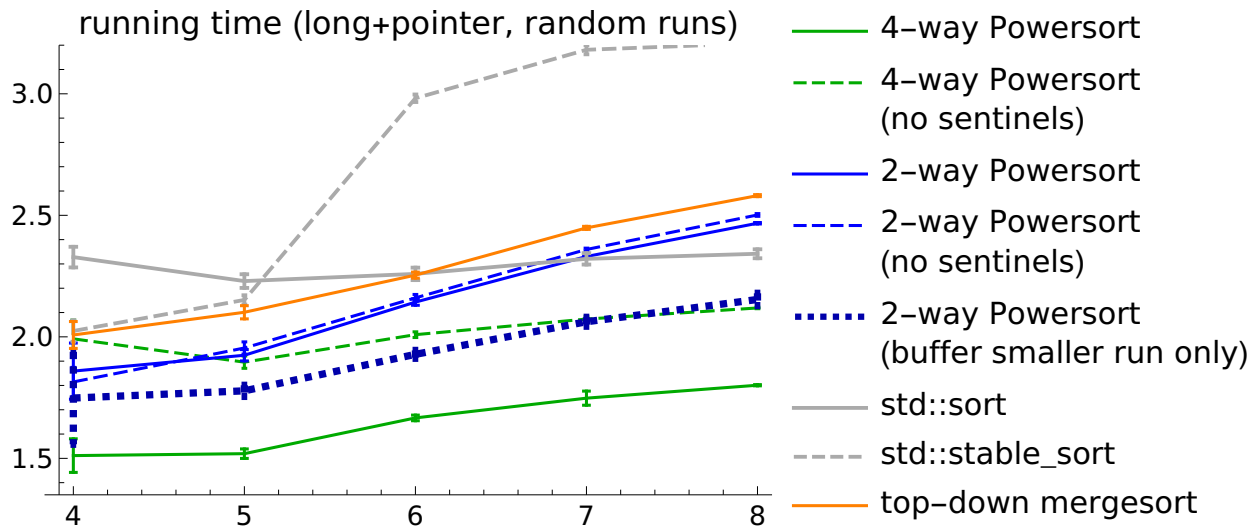


Figure 7.6: Normalized **running times** for sorting **records** of a long key and a pointer (16 bytes in total) where the input has **random** initial runs with a geometric distribution and expected length \sqrt{n} . Axes as in Figure 7.4.

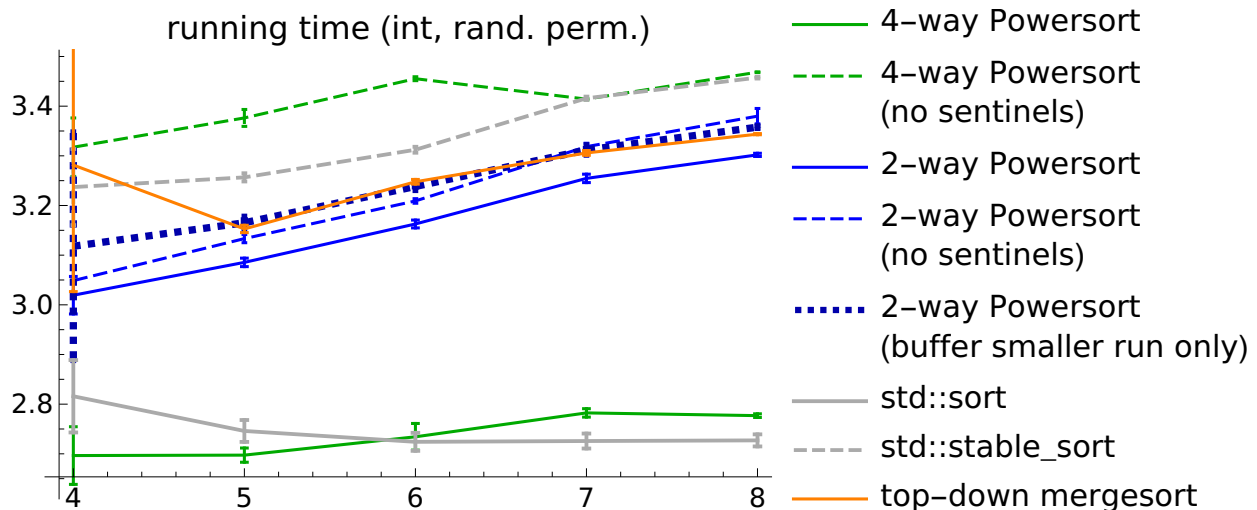


Figure 7.7: Normalized **running times** for sorting ints where the input is a **random permutation** of n . The x -axis shows $\log_{10}(n)$, the y -axis shows average running time in ms multiplied by $10^6/n \lg n$; error bars show one standard deviation.

We have taken care to implement the merge methods (2-way and multiway) as efficiently as possible. The presented methods are chosen from a much larger collection of implementation alternatives we have experimented with. As Figures 7.4–7.7 show, a $+\infty$ value, that can be used as a sentinel value for stopping loops (where otherwise a range check for the iterator would be needed), can yield a substantial speedup in multiway merging.

7.4.4 Hypothesis 2: Scanned elements explain the speedups

After documenting the speedup realized by 4-way Powersort, we now explore likely explanations for these observations. To that end, we ran our code through the cachegrind/callgrind memory hierarchy simulator for valgrind; Table 7.1 shows the results. While there is an 8% saving in the plain number of executed instructions – owing to the extremely lean inner loops in our sentinel-based 4-way method – the much more dramatic improvement lies in the number of cache misses: these are reduced to 54% of the 2-way method that copies both runs to buffer resp. 73% when copying the smaller only. Our data shows that 4-way Powersort simultaneously reduces both the number of instructions and the number of cache misses.

Algorithm	L1 rm	L1 wm	instructions	cycle estimate	merge cost	buffer cost	comparisons
nop	17	6 250 001	400 000 361	1 087 503 111	—	—	—
4P	90 998 034	90 941 221	14 386 631 421	26 834 656 351	678 233 797	678 244 262	1 401 885 151
4P w/o sent	91 104 862	90 991 531	21 788 074 703	34 238 304 273	678 233 797	678 233 797	1 401 804 460
2P	168 472 841	168 392 912	15 593 213 610	36 636 357 390	1 298 329 585	1 298 349 759	1 398 341 256
2P w/o sent	124 242 042	118 893 488	17 678 030 619	32 327 153 489	1 298 329 585	603 947 459	1 398 280 296

Table 7.1: Cachegrind results for sorting one input of $n = 10^8$ ints with random runs of expected length $\sqrt{n} = 10^4$. The table shows level 1 cache read misses (L1 rm), level 1 cache write misses (L1 wm), the total number of executed instructions, and cachegrind’s total-time estimate, defined as $\text{cycles} = \text{instr} + 10 \cdot \text{L1-misses} + 100 \cdot \text{LL-misses}$, where LL = level 3 cache. The algorithms are “nop” = no sorting at all, “4P” = 4-way Powersort, “2P” = 2-way Powersort, “4P w/o sent” = 4-way Powersort without sentinels, “2P w/o sent” = 2-way Powersort without sentinels and copying only the smaller run to the buffer. Here, “nop” is used to get a baseline; the reads come from checking that the input is correctly sorted as part of our running time setup. The reported numbers do not include the cost of generating the random input. The L1 cache lines can hold 16 elements (for our machine); read and write misses are determined by cachegrind. The last three columns were determined directly from the code using counters.

All memory accesses in our sorting methods happen in sequential scans, so the memory access patterns are already as cache-friendly and prefetcher-friendly as can be; instead of cache misses, rather the sheer data volume transferred from memory is the issue. Since the

memory scans in merging are either read-only or write-only, the total number of scanned elements can be computed very accurately: On the used machine, a cache line has 64 bytes, i.e., 16 ints, so we obtain scanned element counts as $16 \cdot (L1rm + L1wm)$ (Figure 7.8). As a sanity check, we can compare numbers as follows: Each merge with merge cost m entails $2m$ reads and $2m$ writes, moreover, we do (in total) one scan over the input for run detection (n reads) and initialize the buffer (n writes); this yields 100.034% of the reported L1 read misses and 100.097% of L1 write misses for 4P (slightly more since a few reads are cached).

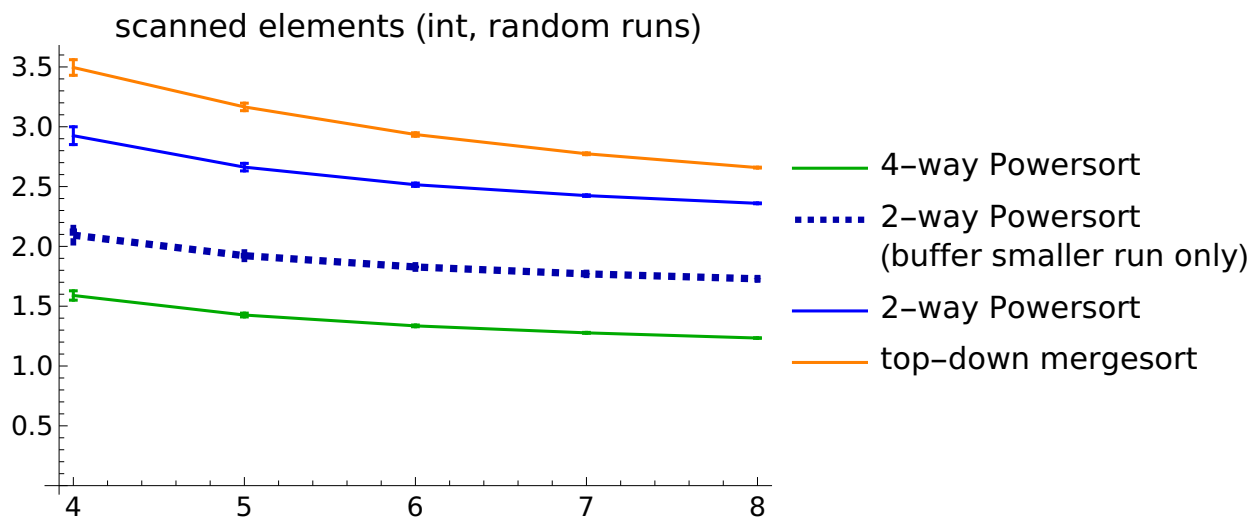


Figure 7.8: Normalized **scanned elements** when the input has random initial runs with a geometric distribution and expected length \sqrt{n} . The x -axis shows $\log_{10}(n)$, the y -axis shows average merge cost divided by $n \lg(n/24)$. Error bars show one standard deviation.

We use cachegrind’s simple linear-combination model as a first-order approximation for explaining running times from scanned elements and instruction counts; see column cycle estimate in Table 7.1. Comparing Figure 7.4 (at $n = 10^8$) to the cycle estimate, we see that despite its simplicity, this model is able to correctly identify 4P as the most efficient variant and it even gives a usable approximation of the relative improvement over the 2-way methods. While it does not rank the remaining algorithms (whose running time is very similar) correctly, we conclude that the large reduction in scanned elements is the most likely reason for the speedup in 4-way Powersort. However, reducing scanned elements at the expense of a substantial increase in executed instructions is not fruitful.

7.4.5 Hypothesis 3: 4-way Powersort halves merge cost

Last, we tie the reduction in scanned elements to the key algorithmic innovation. Figure 7.9 shows the merge cost over different input sizes.

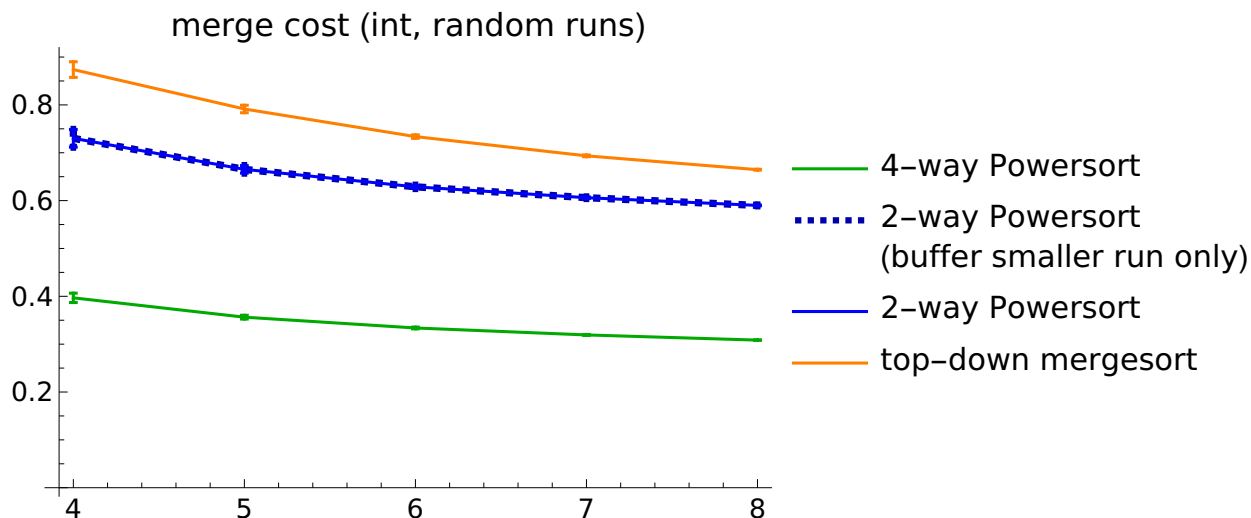


Figure 7.9: Normalized **merge cost** when the input has random initial runs with a geometric distribution and expected length \sqrt{n} . The x -axis shows $\log_{10}(n)$, the y -axis shows average merge cost divided by $n \lg(n/24)$. Error bars show one standard deviation.

We note that for worst-case inputs and as $\mathcal{H} \rightarrow \infty$, Theorem 7.3.3 shows that the merge cost of 4-way Powersort is asymptotically half of that for 2-way Powersort. Whether that is possible to realize for practical values of n is not clear a priori, though. As Figure 7.9 shows, 4-way Powersort reduces the merge cost to around 52% of that of Powersort, very close to halving costs.

7.5 Conclusions

Powersort, introduced in 2018 by Munro & Wild, is a stable (binary) Mergesort variant that exploits existing runs and finds nearly-optimal merging orders with negligible overhead. In this chapter, we presented Multiway Powersort, a generalization of Powersort, that uses k -way merges in order to speed up the algorithm. While the extension of classic 2-way Mergesort to k -way merges is straightforward, doing the same with optimal run-adaptive performance is an algorithmic challenge. In solving that, we also generalize the underlying

algorithm for nearly-optimal *binary* search trees to nearly-optimal *k*-way search trees, which might be of independent interest.

The motivation for our generalization are expected advantages of multiway merges for the memory hierarchy of modern computers. We used a highly engineered implementation of 4-way Powersort to conduct extensive experiments around the following hypotheses – compared with 2-way Powersort:

1. 4-way Powersort can yield significant performance improvements;
2. Scanned elements are a good predictor for this speedup;
3. 4-way Powersort halves the merge cost.

Our experiments fully confirmed the first hypothesis. Considering presorted integer inputs, 4-way Powersort was about 20% faster than the 2-way variant. This speedup was robust in terms of the size of the data items to be sorted (record data instead of integers) as well as the amount of presortedness. However, significant speedups are only realized when the data type allows sentinels to be used. If these are not available (because there is no largest possible value), the running time advantages are made up for by additional range checks.

Regarding the second hypothesis, we find that compared to different versions of 2-way Powersort, 4-way Powersort incurs between 54% and 73% of the cache misses, and scanned-element counts very accurately reflect that. This is a plausible explanation for the observed speedup. A simple linear combination of the number of executed instructions and scanned elements explains most (but not all) of the difference in performance.

Last but not least, we find that the reduced flexibility of doing 4-way merges instead of 2-way merges hardly affects the effectiveness of Powersort’s nearly-optimal merge policy: The merge cost is reduced to 52% for moderate-sized inputs, close to the theoretical optimum of $\frac{1}{2}$ approached in the limit.

In conclusion, 4-way Powersort provides notable advantages over both 2-way Powersort, and other known stable sorting algorithms. Thus, Multiway Powersort is a strong contender for the fastest general, stable sorting algorithm – especially in cases where presorted data can be expected. Both Mergesort variants use $\Theta(n)$ words of extra space for buffer areas; while binary merges can easily work with a buffer for $n/2$ elements, the same is not easily achieved with 4-way merging, where our current implementation uses space n elements.

* * * * *

Conclusion

Chapter 8

Conclusion

This thesis addressed four algorithmic problems:

In Chapter 4, we introduced the state-of-the-art algorithm for solving arbitrary Pinwheel Scheduling instances, as well as demonstrating the existence of finite Pareto surfaces for Pinwheel Scheduling and successfully developing methods for finding them. The contributions of this chapter took us one step closer to solving the longstanding 5/6 Conjecture (Question 1.1.1).

In Chapters 5 and 6, we introduced two novel problems: Decision Polyamorous Scheduling and Optimisation Polyamorous Scheduling. We showed that these NP-hard problems generalize Pinwheel Scheduling and Bamboo Garden Trimming and introduced poly density, which we showed yields sufficient conditions for both solvable and unsolvable polycules. We also provided multiple upper and lower bounds – including demonstrating the first inapproximability results in the field of Periodic Scheduling.

Chapter 7 begins with 2-way Powersort, a sorting algorithm recently adopted by multiple major programming languages, and takes it to another level: We generalize this to k -way Powersort and show that $k = 4$ gives significant practical speedups. As well as providing a sorting method suitable for mass adoption, we expand the study of *why* some methods out-compete others by showing that scanned elements explain the improved performance of 4-way Powersort.

8.1 Future Work

8.1.1 Periodic Scheduling

Section 2.1 introduces seven defining features of Standard Periodic Scheduling problems and surveys several known results; it remains for future works to expand on this taxonomy, study unexamined combinations of variables, and generalize results from individual problems to classes of them where possible.

Bin Scheduling

Bin Scheduling¹ has several avenues of future work opened by recent developments. First, Theorem 4.1.1 clearly extends to Bin Scheduling: There must be finite Pareto surfaces for Bin Scheduling, as each worker in a Bin Scheduling instance is assigned a set of k tasks which, in a valid assignment, form a schedulable Pinwheel Scheduling problem. Given this, what surfaces can be proven to exist for Bin Scheduling? The Pinwheel Schedules provided in Table 4.1 and methods developed in Section 4.1.2 are a starting point, but this work of generalizing relevant proofs and formulating and implementing relevant methods remains to be done.

The recent proof of the 5/6 Density Conjecture (Question 1.1.1) also provides opportunities for the study of Bin Scheduling. It seems likely that the following generalization holds:

Conjecture 8.1.1 Bin Scheduling Density Threshold – Weak.

All Bin Scheduling instances \mathcal{W} with $|a|$ workers and density $d \leq \frac{5}{6}|a|$ are schedulable.

This conjecture is supported by the simplicity of the partition it requires; most Bin Scheduling instances allow some workers to be assigned solvable workloads with $d > \frac{5}{6}$; each of these workers takes us one step closer to being able to spread the remaining tasks such that each worker has $d \leq \frac{5}{6}$. Even if true, however, this conjecture may not be tight; it remains possible that the following holds, or that there is some threshold between the two:

Question 8.1.2 Bin Scheduling Density Threshold – Strong.

Are all Bin Scheduling instances \mathcal{W} with $|a|$ workers and density $d \leq |a| - \frac{5}{6}$ schedulable?

¹inexact periodic scheduling with unit tasks and multiple agents with unit workloads, sometimes called Windows Scheduling

8.1.2 Pinwheel Scheduling

Pinwheel Scheduling has a variety of avenues for future work, but our 2^k and Kernel Conjectures (Conjecture 4.1.3 and the equivalent Conjecture 4.1.4) are particularly interesting:

Conjecture 4.1.3 (restated). 2^k Conjecture.

Let $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ be a loosely feasible Pinwheel Scheduling instance. Then A admits a schedule \mathcal{S} with a holiday at least every 2^k days.

Conjecture 4.1.4 (restated). Kernel Conjecture.

Let $A = (a_1 \leq a_2 \leq \dots \leq a_k)$ be a feasible Pinwheel Scheduling instance. Then there exists another Pinwheel Scheduling instance $B = (b_1 \leq b_2 \leq \dots \leq b_k)$ such that:

- (a) B is also feasible,
- (b) B dominates A , $B \leq A$, and
- (c) $b_k \leq 2^{k-1}$.

Theorem 4.1.1 shows that each finite k has a finite Pareto surface \mathbf{C}_k , and hence that there is *some* maximum frequency f_{\max} such that, for all tasks $f_{i,j} \in \mathcal{P}_j$ in all instances $\mathcal{P}_j \in \mathbf{C}_k$, $f_{\max} \geq f_{i,j}$; these conjectures merely claim that this f_{\max} accords with the surfaces we have generated for $\mathbf{C}_{k \leq 5}$ and $\mathbf{C}(\mathbf{P}_{k \leq 12, d \leq 5/6})$.

Proof of this conjecture would confirm Remark 4.1.6, showing that Pinwheel Scheduling has an FPT-kernel that is $\mathcal{O}(k^2)$. It would also substantially reduce the search space for future works using Pareto surfaces, and could allow even stronger versions for Pinwheel Scheduling instances with prefixes other than $2, 4, 8, \dots$

On the practical side, the Bamboo Garden Trimming problem has recently received attention in an extensive experimental work [DDN19] in the context of approximation algorithms. Our Pareto surfaces for Pinwheel Scheduling immediately imply similar equivalence classes for Bamboo Garden Trimming; the consequences of these results for approximation algorithms in Bamboo Garden Trimming deserve further exploration.

8.1.3 Polyamorous Scheduling

In Section 5.3, we showed that $(\frac{4}{3} - \varepsilon)$ -approximations for Optimisation Polyamorous Scheduling can only exist if $\text{P} = \text{NP}$, while Section 6.2.2 introduces a 5.24-approximation; closing this gap is the most natural open problem for Optimisation Polyamorous Scheduling. As we will

go on to say in Conjecture A.1.2, we expect that further analysis of the SAT reduction given in Appendix A may demonstrate better inapproximability results for OPS in the general case.

Section 6.3.2 shows that any Decision Polyamorous Scheduling instance with poly density $\bar{h}^* \leq \frac{1}{4}$ admits a schedule, while Section 2.3 and Section 6.3.2 show that Decision Polyamorous Scheduling inherits a version of the unschedulable $(2, 3, *)$ instance from Pinwheel Scheduling. This leaves a substantial gap; somewhere between $\bar{h}^* = \frac{1}{4} + \varepsilon$ and $\bar{h}^* = \frac{5}{6}$, there is a density threshold below which instances are schedulable and above which they may not be:

Open Problem 8.1.3 Poly Density Threshold.

What is the highest constant c such that every Decision Polyamorous Scheduling instance $\mathcal{D} = (P, R, f)$ with poly density $\bar{h}^(\mathcal{D}) \leq c$ admits a valid schedule?*

Polyamorous Scheduling Variants

Polyamorous scheduling has several interesting generalizations including Fungible Polyamorous Scheduling, whose decision version we define as:

Definition 8.1.4 Fungible Decision Polyamorous Scheduling (FDPS).

An FDPS instance $\mathcal{D}_f = (P, R, s, f)$ (a “(fungible decision) polycule”) consists of an undirected graph (P, R) where the vertices $P = \{p_1, \dots, p_n\}$ are n classes of fungible persons and the edges R are pairwise relationships between those classes. Classes have integer sizes $s : P \rightarrow \mathbb{N}$ and relationships have integer frequencies $f : R \rightarrow \mathbb{N}$.

The goal is find an infinite schedule $\mathcal{S} : \mathbb{N}_0 \rightarrow 2^P$, such that

- (a)** *(no overflows) for all days $t \in \mathbb{N}_0$, $\mathcal{S}(t)$ is a multiset of elements from P such that each node $p \in P$ appears at most $s(p)$ times, and*
- (b)** *(frequencies) for all $e \in R$ and $t \in \mathbb{N}_0$, we have $e \in \mathcal{S}(t) \cup \mathcal{S}(t+1) \cup \dots \cup \mathcal{S}(t+f(e)-1)$; or to report that no such schedule exists.*

FDPS also has an optimisation version, which again allows each class $p \in P$ to have at most $s(p)$ meetings each day. These problems have clear implications for the scheduling of staff, locations, resources etc. in real-world applications.

Another natural generalisation is Secure Polyamorous Scheduling. Suppose that Adam is dating both Brady and Charlie, who are also dating each other. In a DPS or OPS polycule, on any day, Adam must choose to meet with either Brady or Charlie, who each face the same dilemma; but why can't he meet both?² The Secure Decision Polyamorous scheduling problem allows this:

Definition 8.1.5 Secure Decision Polyamorous Scheduling (SDPS).

An SDPS instance $\mathcal{D}_s = (P, R, f)$ (a “(secure decision) polycule”) consists of an undirected graph (P, R) where the vertices $P = \{p_1, \dots, p_n\}$ are n persons, and the edges R are pairwise relationships, with integer frequencies $f : R \rightarrow \mathbb{N}$ for each relationship.

The goal is find an infinite schedule $\mathcal{S} : \mathbb{N}_0 \rightarrow 2^R$, such that

- (a) (no third-wheels) for all days $t \in \mathbb{N}_0$, $\mathcal{S}(t)$ is a set of disjoint cliques in (P, R) , and
 - (b) (frequencies) for all $e \in R$ and $t \in \mathbb{N}_0$, we have $e \in \mathcal{S}(t) \cup \mathcal{S}(t+1) \cup \dots \cup \mathcal{S}(t+f(e)-1)$;
- or to report that no such schedule exists.

Again, this has a natural optimisation version.

Can poly density be generalized to these problems? Can it identify classes of instances which do or do not permit solutions? And finally, how well do Reduce-Fastest(x) and other known heuristics perform on them?

Restricted Polyamorous Scheduling and Other Motivated Problems

Polyamorous Scheduling also motivates the study of several restricted versions of Pinwheel Scheduling and Bamboo Garden Trimming, including partial scheduling (wherein some portion of the schedule is fixed as part of the problem and the challenge is to find the remainder of the schedule), and fixed holidays (where the fixed part of the schedule consists of periodic gaps).

Another natural research direction is to look at restricted classes of polycules. For example, can we obtain better approximation guarantees on graphs with bounded maximum degree Δ (beyond the simple results from Chapter 5)? Can we exploit bipartite polycules towards a better approximation?

Given the similarities between these problems, it is also natural to ask whether Bamboo Garden Trimming can be approximated arbitrarily well, or whether it too has provable hardness-of-approximation results.

²A key part of polyamory [Thr11]!

8.1.4 Multiway Powersort

Multiway Powersort admits several directions for future study: Can we produce similar speedups when sorting pointers to objects, as always happens in CPython? Can multiway merging be combined with the galloping merge strategy used by Timsort, and could this improve performance, i.e. when comparisons are expensive?

Another interesting question is whether values of k larger than 4 offer some benefit. On one hand, a further reduction of memory transfers is likely beneficial for large inputs; on the other hand, the merging method becomes more complicated and thus potentially causes additional overheads. It is also conceivable that with larger k , the fraction of merge cost that arises from non-full merges, i.e., merging $< k$ runs, would grow. A more aggressive strategy would be to examine k -way Powersort more generally, potentially opening up adaptive- k methods to future study.

Virtual Sentinels

Also of interest are methods for addressing the disparity between 4-way Powersort with and without sentinels (as shown in Section 7.4.3). When a $+\infty$ value is not available but a maximum value is, the following *Virtual Sentinel* method could be employed: When finding runs in an input $A[0..n)$, scan the input from right to left and send any element with a maximum-valued key directly to the output. This frees up the maximum value for use as a sentinel, potentially allowing multiway Powersort with sentinels to sort inputs where the keys are integers or other common data types with a maximum value but no $+\infty$.

* * * * *

Bibliography

- [Hol89] Herman Hollerith. *Art of compiling statistics*. Patent US395782A. 1889.
- [Hol17] Herman Hollerith. *Sorting-machine*. Patent US1237646A. 1917.
- [Sha49] Claude E Shannon. ‘A theorem on coloring the lines of a network’. In: *Journal of Mathematics and Physics* 28.1-4 (1949), pp. 148–152.
- [Bur58] William H Burge. ‘Sorting, trees, and measures of order’. In: *Information and Control* 1.3 (1958), pp. 181–197.
- [Viz65] Vadim G Vizing. ‘The chromatic class of a multigraph’. In: *Cybernetics* 1.3 (1965), pp. 32–41.
- [CAG73] Larry Clemmons, Ken Anderson and Vance Gerry. *Robin Hood (Movie)*. 1973.
- [Gra+79] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra and Alexander Hendrik George Rinnooy Kan. ‘Optimization and approximation in deterministic sequencing and scheduling: a survey’. In: *Annals of discrete mathematics*. Vol. 5. Elsevier, 1979, pp. 287–326.
- [Hol81] Ian Holyer. ‘The NP-completeness of edge-coloring’. In: *SIAM Journal on computing* 10.4 (1981), pp. 718–720.
- [WL83] Wandu Wei and Chung Laung Liu. ‘On a periodic maintenance problem’. In: *Operations Research Letters* 2.2 (1983), pp. 90–93. ISSN: 0167-6377. DOI: [https://doi.org/10.1016/0167-6377\(83\)90044-5](https://doi.org/10.1016/0167-6377(83)90044-5). URL: <https://www.sciencedirect.com/science/article/pii/0167637783900445>.
- [Hol+89] Robert Holte, Al Mok, Louis Rosier, Igor Tulchinsky and Donald Varvel. ‘The pinwheel: a real-time scheduling problem’. In: *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*. Vol. 2. 1989, pp. 693–702. DOI: 10.1109/HICSS.1989.48075.

- [CC92] Mee Yee Chan and Francis Y. L. Chin. ‘General Schedulers for the Pinwheel Problem Based on Double-Integer Reduction’. In: *IEEE Transactions on Computers* 41.6 (1992), pp. 755–768. DOI: 10.1109/12.144627.
- [EW92] Vladimir Estivill-Castro and Derick Wood. ‘A survey of adaptive sorting algorithms’. In: *ACM Computing Surveys* 24.4 (Dec. 1992), pp. 441–476. DOI: 10.1145/146370.146381.
- [HL92] C.C. Han and K.J. Lin. ‘Scheduling distance-constrained real-time tasks’. In: *Proceedings Real-Time Systems Symposium*. IEEE Comput. Soc. Press, 1992. DOI: 10.1109/REAL.1992.242649.
- [Hol+92] Robert Holte, Louis Rosier, Igor Tulchinsky and Donald Varvel. ‘Pinwheel scheduling with two distinct numbers’. In: *Theoretical Computer Science* 100.1 (1992), pp. 105–135. DOI: 10.1016/0304-3975(92)90365-M.
- [MG92] Jayadev Misra and David Gries. ‘A constructive proof of Vizing’s theorem’. In: *Information Processing Letters*. 1992.
- [CC93] Mee Yee Chan and Francis Y. L. Chin. ‘Schedulers for larger classes of pinwheel instances’. In: *Algorithmica* 9.5 (1993), pp. 425–462. ISSN: 0178-4617,1432-0541. DOI: 10.1007/BF01187034.
- [GS93] Mordecai J Golin and Robert Sedgewick. ‘Queue-mergesort’. In: *Information Processing Letters* 48.5 (1993), pp. 253–259.
- [LL97] Shun-Shii Lin and Kwei-Jay Lin. ‘A Pinwheel Scheduler for Three Distinct Numbers with a Tight Schedulability Bound’. In: *Algorithmica* 19.4 (1997), pp. 411–426. DOI: 10.1007/PL00009181. URL: <https://doi.org/10.1007/PL00009181>.
- [HL98] Chih-Wen Hsueh and Kwei-Jay Lin. ‘On-line schedulers for pinwheel tasks using the time-driven approach’. In: *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168)*. 1998, pp. 180–187. DOI: 10.1109/EMWRTS.1998.685083.
- [Knu98] Donald E. Knuth. *The Art Of Computer Programming: Searching and Sorting*. 2nd. Addison Wesley, 1998, p. 780. ISBN: 978-0-20-189685-5.
- [Tak98] Tadao Takaoka. ‘A New Measure of Disorder in Sorting – Entropy’. In: *The Fourth Australasian Theory Symposium (CATS 1998)*. 1998, pp. 77–86.

- [Bar+02] Amotz Bar-Noy, Randeep Bhatia, Joseph Naor and Baruch Schieber. ‘Minimizing service and operation costs of periodic scheduling’. In: *Mathematics of Operations Research* 27.3 (2002), pp. 518–544. DOI: 10.1287/moor.27.3.518.314.
- [FL02] Peter C Fishburn and Jeffrey C Lagarias. ‘Pinwheel scheduling: Achievable densities’. In: *Algorithmica* 34.1 (2002), pp. 14–38. DOI: 10.1007/s00453-002-0938-9.
- [Pet02] Tim Peters. *[Python-Dev] Sorting*. <https://mail.python.org/pipermail/python-dev/2002-July/026837.html>. July 2002. URL: <https://mail.python.org/pipermail/python-dev/2002-July/026837.html> (visited on 13/04/2018).
- [BL03] Amotz Bar-Noy and Richard E Ladner. ‘Windows scheduling problems for broadcast systems’. In: *SIAM Journal on Computing* 32.4 (2003), pp. 1091–1113.
- [FC05] Eugene A. Feinberg and Michael T. Curry. ‘Generalized Pinwheel Problem’. In: *Mathematical Methods of Operations Research* 62.1 (2005), pp. 99–122. DOI: 10.1007/s00186-005-0443-4. URL: <https://doi.org/10.1007/s00186-005-0443-4>.
- [BLT07] Amotz Bar-Noy, Richard E Ladner and Tami Tamir. ‘Windows scheduling as a restricted version of bin packing’. In: *ACM Transactions on Algorithms* 3.3 (2007). DOI: 10.1145/1273340.1273344.
- [BN09] Jérémy Barbay and Gonzalo Navarro. ‘Compressed Representations of Permutations, and Applications’. In: *26th International Symposium on Theoretical Aspects of Computer Science STACS 2009*. IBFI Schloss Dagstuhl. 2009, pp. 111–122.
- [Thr11] John Threepwood. *Why not both? / Why don't we have both?* Ed. by ADPuckey, Yes Man Junior (KeurBro), Trollkeeper, amanda b., madcat, 3kole5, andcallmeshirley and ZachEditors. <https://knowyourmeme.com/memes/why-not-both-why-dont-we-have-both>. Aug. 2011. URL: <https://knowyourmeme.com/memes/why-not-both-why-dont-we-have-both>.
- [Wil12] Sebastian Wild. ‘Java 7’s dual pivot quicksort’. MA thesis. Technische Universität Kaiserslautern, 2012.

- [BN13] J eremy Barbay and Gonzalo Navarro. ‘On compressing permutations and adaptive sorting’. In: *Theoretical Computer Science* 513 (Nov. 2013), pp. 109–123. DOI: 10.1016/j.tcs.2013.10.019.
- [Kus+13] Shrinu Kushagra, Alejandro L opez-Ortiz, Aurick Qiao and J. Ian Munro. ‘Multi-Pivot Quicksort: Theory and Experiments’. In: *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Nov. 2013, pp. 47–60. DOI: 10.1137/1.9781611973198.6.
- [CG14] Badrish Chandramouli and Jonathan Goldstein. ‘Patience is a virtue: Revisiting merge and sort on modern processors’. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 2014, pp. 731–742.
- [JL14] Tobias Jacobs and Salvatore Longo. *A new perspective on the windows scheduling problem*. 2014.
- [ADK16] Martin Aum uller, Martin Dietzfelbinger and Pascal Klaue. ‘How Good Is Multi-Pivot Quicksort?’ In: *ACM Transactions on Algorithms* 13.1 (Dec. 2016), pp. 1–47. DOI: 10.1145/2963102.
- [NWM16] Markus E. Nebel, Sebastian Wild and Conrado Mart inez. ‘Analysis of pivot sampling in dual-pivot quicksort: A holistic analysis of Yaroslavskiy’s partitioning scheme’. In: *Algorithmica* 75 (2016), pp. 632–683.
- [Wil16] Sebastian Wild. ‘Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential’. Dissertation (Ph.D. thesis). University of Kaiserslautern, 2016. ISBN: 978-3-00-054669-3. URL: <https://www.wild-inter.net/publications/wild-2016>.
- [Ga +17] Leszek Ga sieniec, Ralf Klasing, Christos Levcopoulos, Andrzej Lingas, Jie Min and Tomasz Radzik. *Bamboo garden trimming problem (perpetual maintenance of machines with different attendance urgency factors)*. Vol. 10139. Lecture Notes in Computer Science. Springer, 2017. ISBN: 978-3-319-51962-3. DOI: 10.1007/978-3-319-51963-0.
- [Aug+18] Nicolas Auger, Vincent Jug e, Cyril Nicaud and Carine Pivoteau. ‘On the Worst-Case Complexity of TimSort’. In: *26th Annual European Symposium on Algorithms (ESA 2018)*. Ed. by Hannah Bast Yossi Azar and Grzegorz Herman. Leibniz International Proceedings in Informatics (LIPIcs). 2018. DOI: 10.4230/LIPIcs.ESA.2018.4.

- [MW18] J. Ian Munro and Sebastian Wild. ‘Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs’. In: *European Symposium on Algorithms (ESA)*. Ed. by Yossi Azar, Hannah Bast and Grzegorz Herman. Vol. 112. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 63:1–63:16. DOI: 10.4230/LIPIcs.ESA.2018.63. URL: <https://www.wild-inter.net/publications/munro-wild-2018>.
- [Pet+18] Tim Peters, Koos Zevenhoven, Vincent Jugé, Laurent Lyaudet and Carl Friedrich Bolz-Tereick. *Replace list sorting merge_collapse()*? <https://github.com/python/cpython/issues/78742>. 2018. URL: <https://github.com/python/cpython/issues/78742>.
- [BFK19] Michael A. Bender, Martín Farach-Colton and William Kuszmaul. ‘Achieving optimal backlog in multi-processor cup games’. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1148–1157. ISBN: 9781450367059. DOI: 10.1145/3313276.3316342.
- [BK19] Sam Buss and Alexander Knop. ‘Strategies for Stable Merge Sorting’. In: *Symposium on Discrete Algorithms (SODA 2019)*. SIAM, Jan. 2019, pp. 1272–1290. DOI: 10.1137/1.9781611975482.78.
- [CA19] Charlotta Carlström and Catrine Andersson. ‘The queer spaces of BDSM and non-monogamy’. In: *Journal of Positive Sexuality*. 5.1 (2019), pp. 14–19.
- [Cic+19] Serafino Cicerone, Gabriele Di Stefano, Leszek Gąsieniec, Tomasz Jurdzinski, Alfredo Navarra, Tomasz Radzik and Grzegorz Stachowiak. ‘Fair Hitting Sequence Problem: Scheduling Activities with Varied Frequency Requirements’. In: *International Conference on Algorithms and Complexity (CIAC)*. Springer, 2019, pp. 174–186. DOI: 10.1007/978-3-030-17402-6_15.
- [DDN19] Mattia D’Emidio, Gabriele Di Stefano and Alfredo Navarra. ‘Bamboo Garden Trimming Problem: Priority Scheduling’. In: *Algorithms* 12.4 (2019), p. 74.
- [Gou+19] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot and Dominic Steinhöfel. ‘Verifying OpenJDK’s Sort Method for Generic Collections’. In: *Journal of Automated Reasoning* (Aug. 2019). DOI: 10.1007/s10817-017-9426-4.

- [Del20] Federico Della Croce. *An enhanced pinwheel algorithm for the bamboo garden trimming problem*. arXiv preprint 2003.12460. 2020.
- [Din20] Wei Ding. ‘A branch-and-cut approach to examining the maximum density guarantee for pinwheel schedulability of low-dimensional vectors’. In: *Real-Time Systems* 56.3 (2020), pp. 293–314. DOI: 10.1007/s11241-020-09349-w.
- [KS20] Akitoshi Kawamura and Makoto Soejima. ‘Simple strategies versus optimal schedules in multi-agent patrolling’. In: *Theoretical Computer Science* 839 (Nov. 2020), pp. 195–206. DOI: 10.1016/j.tcs.2020.07.037.
- [Kus20] William Kuszmaul. ‘Achieving Optimal Backlog in the Vanilla Multi-Processor Cup Game’. In: *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2020, pp. 1558–1577. DOI: 10.1137/1.9781611975994.96.
- [BK21] Michael A. Bender and William Kuszmaul. ‘Randomized cup game algorithms against strong adversaries’. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 2059–2077.
- [Ee21] Martijn van Ee. ‘A $12/7$ -approximation algorithm for the discrete Bamboo Garden Trimming problem’. In: *Operations Research Letters* 49.5 (Sept. 2021), pp. 645–649. DOI: 10.1016/j.orl.2021.07.001.
- [Pet21] Tim Peters. *Timsort (listsort.txt)*. <https://github.com/python/cpython/blob/main/Objects/listsort.txt#L343>. 2021. URL: <https://github.com/python/cpython/blob/main/Objects/listsort.txt#L343>.
- [Smi21] Benjamin Smith. *Towards the $5/6$ -Density Conjecture in Pinwheel Scheduling*. Nov. 2021. DOI: 10.5281/zenodo.5636327.
- [Afs+22] Peyman Afshani, Mark de Berg, Kevin Buchin, Jie Gao, Maarten Löffler, Amir Nayyeri, Benjamin Raichel, Rik Sarkar, Haotian Wang and Hao-Tsung Yang. ‘On Cyclic Solutions to the Min-Max Latency Multi-Robot Patrolling Problem’. In: *International Symposium on Computational Geometry (SoCG)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Mar. 2022. DOI: 10.4230/LIPICS.SOCG.2022.2.
- [Bil+22] Davide Bilò, Luciano Gualà, Stefano Leucci, Guido Proietti and Giacomo Scornavacca. ‘Cutting bamboo down to size’. In: *Theoretical Computer Science* 909 (2022), pp. 54–67.

- [Bos+22] Thomas Bosman, Martijn van Ee, Yang Jiao, Alberto Marchetti-Spaccamela, R. Ravi and Leen Stougie. ‘Approximation Algorithms for Replenishment Problems with Fixed Turnover Times’. In: *Algorithmica* 84.9 (May 2022), pp. 2597–2621. ISSN: 1432-0541. DOI: 10.1007/s00453-022-00974-4. URL: <http://dx.doi.org/10.1007/s00453-022-00974-4>.
- [GSW22] Leszek Gąsieniec, Benjamin Smith and Sebastian Wild. ‘Towards the 5/6-Density Conjecture of Pinwheel Scheduling’. In: *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Ed. by C. A. Phillips and B. Speckmann. SIAM, Jan. 2022, pp. 91–103.
- [Kus22] John Kuszmaul. ‘Bamboo Trimming Revisited: Simple Algorithms Can Do Well Too’. In: *Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, July 2022. DOI: 10.1145/3490148.3538580.
- [Gel+23] William Cawley Gelling, Markus E. Nebel, Benjamin Smith and Sebastian Wild. ‘Multiway Powersort’. In: *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX) (2023)*, pp. 190–200.
- [HS23] Felix Höhne and Rob van Stee. ‘A 10/7-Approximation for Discrete Bamboo Garden Trimming and Continuous Trimming on Star Graphs’. en. In: *Conference on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. DOI: 10.4230/LIPICS.APPROX/RANDOM.2023.16.
- [Gąs+24] Leszek Gąsieniec, Tomasz Jurdziński, Ralf Klasing, Christos Levcopoulos, Andrzej Lingas, Jie Min and Tomasz Radzik. ‘Perpetual maintenance of machines with different urgency requirements’. In: *Journal of Computer and System Sciences* 139 (Feb. 2024), p. 103476.
- [GSW24] Leszek Gąsieniec, Benjamin Smith and Sebastian Wild. ‘Polyamorous Scheduling’. In: *12th International Conference on Fun with Algorithms (FUN 2024)*. Schloss Dagstuhl—Leibniz Center for Informatics. 2024.
- [Jug24] Vincent Jugé. ‘Adaptive shivers sort: an alternative sorting algorithm’. In: *ACM Transactions on Algorithms* 20.4 (2024), pp. 1–55.

- [Kaw24] Akitoshi Kawamura. ‘Proof of the Density Threshold Conjecture for Pinwheel Scheduling’. In: *STOC’24—Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. ACM, New York, 2024, pp. 1816–1819. ISBN: 979-8-4007-0383-6. DOI: 10.1145/3618260.3649757. URL: <https://doi.org/10.1145/3618260.3649757>.
- [Bik+25] Yuriy Biktairov, Leszek Gaşieniec, Wanchote Po Jiamjitrak, Namrata, Benjamin Smith and Sebastian Wild. ‘Simple approximation algorithms for Polyamorous Scheduling’. In: *2025 Symposium on Simplicity in Algorithms (SOSA)*. SIAM, 2025, pp. 290–314.
- [Gha+25] Elahe Ghasemi, Vincent Jugé, Ghazal Khalighinejad and Helia Yazdanyar. ‘Gallop­ing in fast-growth natural merge sorts’. In: *Algorithmica* 87.2 (2025), pp. 242–291.
- [Kuh+25] Marius Kuhrt, Bernhard Seeger, Sebastian Wild and Goetz Graefe. ‘Adaptive sorting for large keys, strings, and database rows’. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik, Bonn, 2025, pp. 217–239.

Part III
Appendices

Appendix A

Original Inapproximability Proof

In this appendix, we will use a reduction from MAX-3SAT to prove Theorem 5.1.1:

Theorem 5.1.1 (restated). SAT Hardness of approximation.

Unless $P = NP$, there is no polynomial-time $(1 + \delta)$ -approximation algorithm for the Optimisation Polyamorous Scheduling problem for any $\delta < \frac{1}{12}$.

A.1 Overview of Proof

The proof of Theorem 5.1.1 has two steps: The first step is a reduction from the decision version of MAX-3SAT (D-MAX-3SAT) to DPS. D-MAX-3SAT is the following problem: given a 3-CNF formula $\varphi = c_1 \wedge c_2 \wedge \dots \wedge c_m$ with clauses $C = \{c_1, c_2, \dots, c_m\}$ over variables $X = \{x_1, x_2, \dots, x_{n'}\}$ and integer k , decide whether there is an assignment of Boolean values to the variables in X such that at least k clauses in C are satisfied.

Lemma A.1.1 D-MAX-3SAT \leq_p DPS.

For any 3-CNF formula φ with m clauses and integer $k \leq m$, we can construct in polynomial time a decision polycule $\mathcal{P}_{d\varphi k}$ which has a valid schedule if and only if at least k clauses of φ can be simultaneously satisfied.

Lemma A.1.1 is our key technical contribution and its proof will be given over the course of the remainder of this section.

The second step in the proof of Theorem 5.1.1 is to convert the decision polycule $\mathcal{P}_{d\varphi k}$ from Lemma A.1.1 to an optimisation polycule $\mathcal{P}_{o\varphi k}$ using Lemma 3.2.4. It will be immediate from the construction that the largest frequency in $\mathcal{P}_{d\varphi k}$ is $F = 12$. So by Lemma 3.2.4, $\mathcal{P}_{o\varphi k}$ has either $h^* = 1$, namely if $\mathcal{P}_{d\varphi k}$ is feasible, or $h^* \geq \frac{13}{12}$, otherwise.

Proof of Theorem 5.1.1. Assume that there is a polynomial-time approximation algorithm A for OPS with approximation ratio $\alpha < \frac{13}{12}$. If $\mathcal{P}_{o\varphi k}$ has optimal heat $h^* = 1$, A produces a schedule with heat $1 \leq h \leq \alpha < \frac{13}{12}$, whereas if $h^* \geq \frac{13}{12}$, A must produce a schedule of heat $\frac{13}{12} \leq h \leq \alpha \cdot \frac{13}{12}$. So, by running A , we are able to distinguish between $h^* \leq 1$ and $h^* \geq \frac{13}{12}$ for $\mathcal{P}_{o\varphi k}$, hence between feasibility or infeasibility of $\mathcal{P}_{d\varphi k}$ and, therefore, between Yes and No instances of D-MAX-3SAT via the polynomial-time reduction from Lemma A.1.1. As a generalisation of 3SAT, D-MAX-3SAT is NP-hard, hence $P = NP$ follows. \square

Let us denote by α^* the approximability threshold for Optimisation Poly Scheduling, that is: efficient polynomial-time approximation algorithms with approximation ratio α exist if and only if $\alpha \geq \alpha^*$ (assuming $P \neq NP$). Theorem 5.1.1 shows that $\alpha^* \geq \frac{13}{12}$ and Theorem 5.1.4 shows that $\alpha^* = O(\log n)$, leaving a substantial gap. We conjecture that the constant $\frac{13}{12}$ can be improved by careful analysis of our construction, but we leave this to future work.

Conjecture A.1.2.

$$\alpha^* \geq \frac{4}{3}.$$

Remark A.1.3 Towards stronger inapproximability results.

Our reduction includes additional degrees of freedom not currently used towards the proof of Lemma A.1.1. In particular, for the current statement, a reduction for standard 3SAT would have sufficed, removing the sorting network from the construction. However, to find better constant or even superconstant lower bounds for α^ it seems likely that starting with a gapped MAX-3SAT problem can provide stronger gaps for the outcome. For that, we need that our construction allows us to specify freely how many clauses have to be satisfiable for $\mathcal{P}_{d\varphi k}$ to be feasible. While we leave this to future work, we include here the required features in the construction which may facilitate these results.*

A.2 Reduction Overview

We now give the proof of Lemma A.1.1. For the remainder of this section, we assume a 3-CNF formula $\varphi = c_1 \wedge \dots \wedge c_m$ over variables $X = \{x_1, \dots, x_{n'}\}$ and integer k are given. We will describe how to construct the DPS instances $\mathcal{P}_{d\varphi k}$ that admit a schedule iff there is a variable assignment $v : X \rightarrow \{\text{True}, \text{False}\}$ that satisfies at least k clauses in $C = \{c_1, \dots, c_m\}$. The construction is based on building components of Boolean formulas via DPS “gadgets”:

- *variables* (Section A.4),
- *clauses (OR gadgets)* (Section A.6),
- a *sorting network*, comprised of *SWAP gadgets* (Section A.7.5) to group satisfied outputs together, and
- a check for $\geq k$ true clauses, the *tension gadget* (Section A.8).

To make those gadgets work, we require further auxiliary gadgets:

- a “*True Clock*” to break ties between symmetric choices for schedules (Section A.3),
- slot duplication gadgets: D_3 *duplicators* (Section A.5.1), D_6 *duplicators* (Section A.5.2), D_{12} *duplicators* (Section A.7.1), and
- *slot splitting gadgets* S_{B_6} , $S_{B_{12}}$, $S_{G_{12}}$ (Section A.7.4).

The overall conversion algorithm is stated in Definition A.2.1 below; a worked example is shown in Figure A.1.

Definition A.2.1 $\mathcal{P}_{d\varphi k}$ polycules.

The decision polycule $\mathcal{P}_{d\varphi k} = (P, R, f)$ is constructed in layers as follows:

(a) Variable layer:

The variable layer consists of a *True Clock* and a variable gadget with outputs 3_{iR} and 3_{iB} for each variable $x_i \in X$.

(b) Duplication layer:

The duplication layer duplicates outputs of the variable layer: D_3 duplicators create one 3_{iR} edge for each $x_i \in C$, and one 3_{iB} edge for each $\bar{x}_i \in C$. They also create as many 3_R and 3_B edges as are needed, while D_6 duplicators do the same for 6_G and 6_P edges. Unused edges are connected to pendent nodes.

(c) Clause layer:

The clause layer consists of one *OR gadget* for each clause $c_j \in C$. *OR gadgets* have three inputs, each corresponding to a literal in c_j : 3_{iR} for x_i , 3_{iB} for \bar{x}_i . For clauses with less than 3 literals, 3_B edges fill the *OR gadget*'s unused inputs.

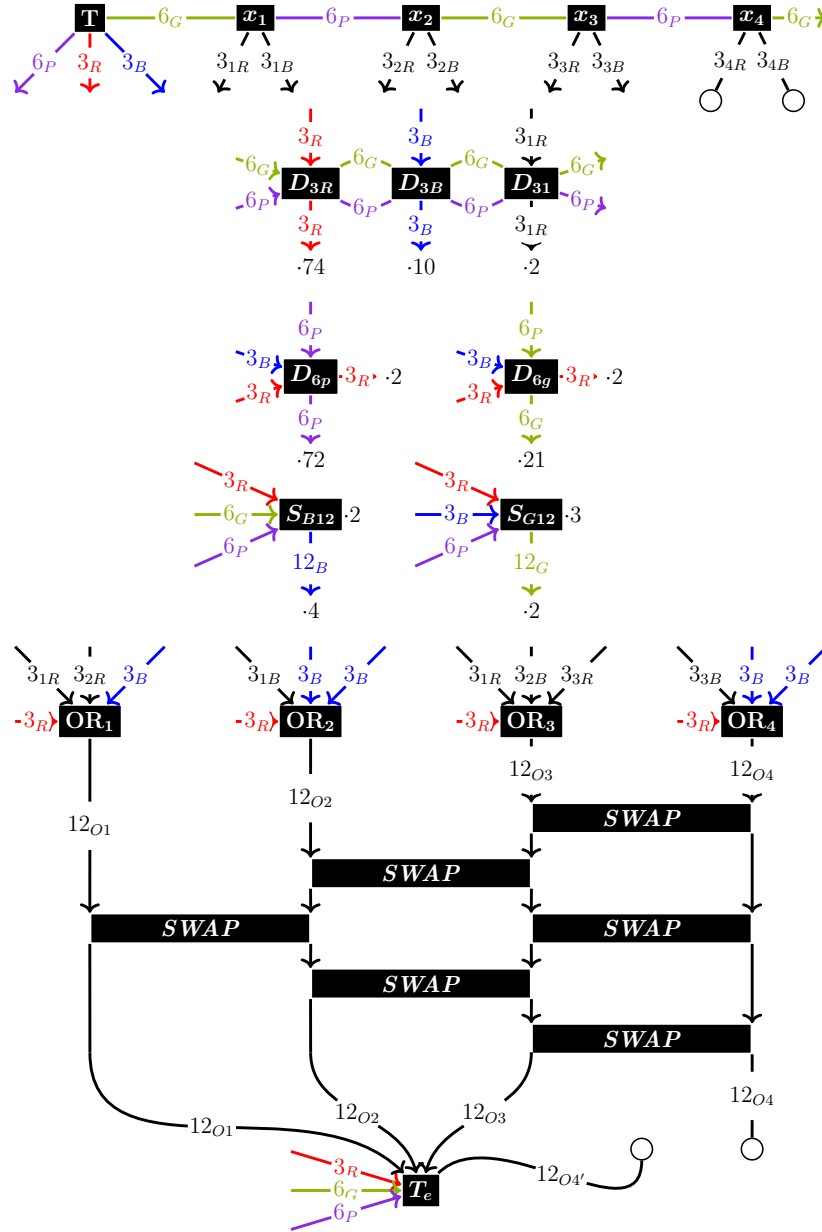


Figure A.1: A DPS polycule which is schedulable iff there is some assignment that simultaneously satisfies at least 3 of $(x_1 \vee x_2)$, $(\overline{x_1})$, $(x_1 \vee \overline{x_2} \vee x_3)$, and $(\overline{x_3})$ (possible in this case). Note that re-unifying the bottom-most 12_{O_4} edge with the $12_{O_4'}$ edge will make a polycule which is schedulable iff all 4 clauses can be satisfied (and hence has no valid schedule here). Similarly, breaking the 12_{O_3} edge in the middle and connecting the two new ends to pendent nodes will create a polycule which is schedulable iff 2 clauses can be satisfied. Connections between layers are omitted for clarity but flow from top to bottom, starting with the variable layer, then three layers concerned with the duplication of variables, the *OR* layer, the sorting network, and the tension layer.

(d) *Sorting network:*

The next layer is a single gadget – a sorting network. The 12_O output edges of the clause layer will each be the input to one of m channels, each of which terminates with a 12_O output edge. This gadget also consumes 12_B and 12_G edges created by S_{B12} and S_{G12} gadgets.

(e) *Tension layer:*

The tension layer attaches tension gadgets to the leftmost k outputs of the sorting layer – any other outputs from this layer and any spare inputs to the tension layer are connected to pendent nodes.

A.3 The True Clock & Colour Slots

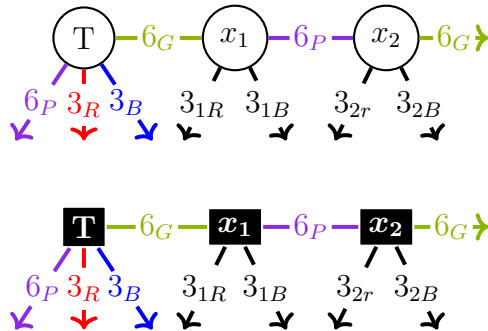


Figure A.2: Gadgets for the True Clock and for sample variables x_1 and x_2 (top, left to right), with shorthand versions shown below. Gadgets are shown connected as they would be in a sample variable layer, and their colours and schedules are discussed in Section A.3. Further variables can be added to the right, and must be added in pairs to conserve the 6_G output edge (though the final variable may be connected to pendent nodes if not otherwise needed).

Figure A.2 introduces gadgets representing sample variables x_1 and x_2 , as well as a special variable: the True Clock T , which acts as a drumbeat for the polycule as a whole. Variables, including T , have four relationships: $[3, 3, 6, 6]$, so their local schedules must all have the following form: $[3_a, 3_b, 6_a, 3_a, 3_b, 6_b]$. As schedules are cyclic, we can choose to start this schedule with the 3_a edge of T without loss of generality. We then assign names to these edges as dictated by the True Clock.

Definition A.3.1 Slots.

We call days $t \in \mathbb{N}_0$ with $t \equiv 0 \pmod{3}$ the *red slots*, days $t \equiv 1 \pmod{3}$ *blue slots*, days $t \equiv 2 \pmod{6}$ *green slots*, and days $t \equiv 5 \pmod{6}$ *purple slots*.

The local schedule of the True Clock is therefore given by $[3_R, 3_B, 6_G, 3_R, 3_B, 6_P]$. As 3_R is scheduled on day 0, it will always be assigned in red slots, with 3_B , 6_G , and 6_P edges also restricted to slots of their respective colours. We will sometimes represent this by underlining elements or gaps in a schedule, e.g., $[\underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}]$, or by referring to edges as being red, blue, green, or purple.

All gadgets introduced below will be constructed such that the lengths of their schedules are integer multiples of 6. In the final polycule $\mathcal{P}_{d\varphi k}$ they will be connected (usually through intermediaries) to the True Clock, such that their edges must stick to certain slots.

To keep correctness proofs of individual gadgets readable, we call a schedule S that schedules all coloured edges in slots of the given colour *slot-respecting*.

Drawing Conventions Gadgets are connected by input and output edges, represented by incoming and outgoing arrows respectively – each one being half of a relationship between two people from different gadgets. In addition to their frequencies, input and output edges share restrictions on their permissible schedules, carrying them from one gadget to another as discussed in the proofs associated with each gadget.

In shorthand gadgets, vertical incoming edges are the primary *input* to a gadget, encoding the value of some variable or logical function; horizontal inputs contain edges of fixed colour, which we will refer to as *constants*. Similarly, vertical outgoing edges represent primary *outputs* that encode the result of the gadget, while horizontal outgoing edges represent incidentally created constants which may either be used by other gadgets or connected to pendent vertices.

Some incoming and outgoing edges will have end labels of the form “ $\cdot i$ ”, indicating i connections of the given type, each between different people.

A.4 Variables

Figure A.2 also introduces the gadget for a sample variable, x_1 , which again has four relationships: $[3_{1R}, 3_{1B}, 6_G, 6_P]$. The key property of variable gadgets is summarized in the following lemma:

Lemma A.4.1 Variable Gadget Schedules.

Any valid global schedule must yield a local schedule for the variable gadget x_i of the form $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$.

Proof. Note that x_i has local density $D = 1$, so 3_{iR} and 3_{iB} must be scheduled exactly once in each 3-day period, forcing every 3 days to be of the form $[3_a, 3_b, _]$, and every 6-day schedule to be of the form $[3_a, 3_b, _, 3_a, 3_b, _]$; this leaves two remaining slots, which must contain 6_G and 6_P .

The 6_G edge of the first variable, x_1 , must be green due to its connection with T , so schedules for x_1 must be of the form $[3_a, 3_b, 6_G, 3_a, 3_b, _]$, which can then only be completed as $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$. This proceeds for x_2 , whose 6_P edge is shared with x_1 such that it must be purple, forcing the partial schedule $[3_a, 3_b, _, 3_a, 3_b, 6_P]$ which likewise must be completed $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$.

Further pairs of variables are each connected to the 6_G edge returned by the previous pair, so their schedules must be of the same form. Thus, by induction, schedules for each variable gadget x_i must be of the form $[3_a, 3_b, 6_G, 3_a, 3_b, 6_P]$. \square

According to Lemma A.4.1, the incident 6_G edge is used to restrict the valid schedules for x_1 , leaving two possibilities: $[3_{1R}, 3_{1B}, 6_G, 3_{1R}, 3_{1B}, 6_P]$ and $[3_{1B}, 3_{1R}, 6_G, 3_{1B}, 3_{1R}, 6_P]$. The former schedule, where 3_{1R} is scheduled in red slots, corresponds to a variable assignment where x_1 is True, whereas the the second schedule corresponds to x_1 being assigned False.

This technique of connecting 3_R , 3_B , 6_G , or 6_P edges to people in a gadget to limit their valid local schedules and force relationships between edges, slots, and particular meanings will be used extensively in what follows.

Note that 3_{iB} has the opposite value to 3_{iR} , so using 3_{iB} edges in the polycule corresponds to the negated literal \bar{x}_i , just as 3_{iR} edges correspond to the literal x_i .

A.5 Duplication of Variables and Constants

Variables may appear in multiple clauses, while the constant 3_R , 3_B , 6_G , and 6_P edges are used in multiple gadgets, engendering a need for the duplication of variables and constants.

A.5.1 3-Duplicators

A gadget for duplicating edges with period 3 is shown in Figure A.3 and proven to accurately reproduce its input by Lemma A.5.1.

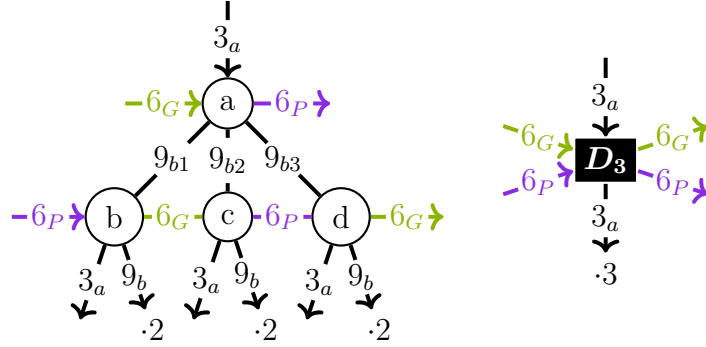


Figure A.3: A gadget for duplicating input edges with frequency 3 (left), with a shorthand version (right). Note that the input can be duplicated indefinitely many times by repeating the second layer using the 9_b edges from the previous layer and a 6_P or 6_G edge from two layers above.

Lemma A.5.1 3-Duplicator Gadget Schedules.

Any slot-respecting schedule must yield a local schedule for each node in any 3-duplicator gadget D_3 of the same form, either:

$$[\underline{3_a}, \underline{9_b}, \underline{6_G}, \underline{3_a}, \underline{9_{b'}}, \underline{6_P}, \underline{3_a}, \underline{9_{b''}}, \underline{6_G}, \underline{3_a}, \underline{9_b}, \underline{6_P}, \underline{3_a}, \underline{9_{b'}}, \underline{6_G}, \underline{3_a}, \underline{9_{b''}}, \underline{6_P}] \text{ or}$$

$$[\underline{9_b}, \underline{3_a}, \underline{6_G}, \underline{9_{b'}}, \underline{3_a}, \underline{6_P}, \underline{9_{b''}}, \underline{3_a}, \underline{6_G}, \underline{9_b}, \underline{3_a}, \underline{6_P}, \underline{9_{b'}}, \underline{3_a}, \underline{6_G}, \underline{9_{b''}}, \underline{3_a}, \underline{6_P}].$$

Proof. Each node in D_3 has tasks $[3_a, 6_P, 6_G, 9_{b1}, 9_{b2}, 9_{b3}]$ and has local density $D = 1$, so each task with frequency f must appear exactly once every f days. Further, in any slot-respecting schedule, tasks 6_P and 6_G are scheduled in purple and green slots respectively, forcing partial schedules of the form: $[_, _, \underline{6_G}, _, _, \underline{6_P}]$.

Considering node a , note that if incident edge 3_a is scheduled on a combination of red and blue days then either it will appear more than once in some 3-day period or its constraint will be violated. This demonstrates that schedules must be of the form:

$$[\underline{3_a}, _, \underline{6_G}, \underline{3_a}, _, \underline{6_P}, \underline{3_a}, _, \underline{6_G}, \underline{3_a}, _, \underline{6_P}, \underline{3_a}, _, \underline{6_G}, \underline{3_a}, _, \underline{6_P}] \text{ or}$$

$$[_, \underline{3_a}, \underline{6_G}, _, \underline{3_a}, \underline{6_P}, _, \underline{3_a}, \underline{6_G}, _, \underline{3_a}, \underline{6_P}, _, \underline{3_a}, \underline{6_G}, _, \underline{3_a}, \underline{6_P}],$$

depending on the colour of the incident 3_a edge. In either case, 9_{b1} , 9_{b2} and 9_{b3} must occupy the remaining slots, which match the schedules shown in the lemma for some mapping of 9_{b1} , 9_{b2} and 9_{b3} to 9_b , $9_{b'}$ and $9_{b''}$.

Now consider an arbitrary node $p \neq a$, with an incident 9_b node. If the 3_a edge incident to a is red, the schedule for n must be of the form:

$$[_, \underline{9_b}, \underline{6_G}, _, _, \underline{6_P}, _, _, \underline{6_G}, _, \underline{9_b}, \underline{6_P}, _, _, \underline{6_G}, _, _, \underline{6_P}],$$

with $9_{b'}$, $9_{b''}$, and 3_a filling the remaining slots. If either $9_{b'}$ or $9_{b''}$ are ever scheduled in a red slot, the constraint of 3_a will be violated, therefore either partial schedule leads to the full schedule:

$$[\underline{3_a}, \underline{9_b}, \underline{6_G}, \underline{3_a}, \underline{9_{b'}}, \underline{6_P}, \underline{3_a}, \underline{9_{b''}}, \underline{6_G}, \underline{3_a}, \underline{9_b}, \underline{6_P}, \underline{3_a}, \underline{9_{b'}}, \underline{6_G}, \underline{3_a}, \underline{9_{b''}}, \underline{6_P}]$$

matching the schedule of a , and of the lemma. If the 3_a edge incident to a is blue, the same logic applies, with all 3_a edges also being blue and the 9_b nodes being red. \square

A.5.2 6-Duplicators

Figure A.4 and Lemma A.5.2 introduce a gadget which duplicates incident 6_G or 6_P edges.

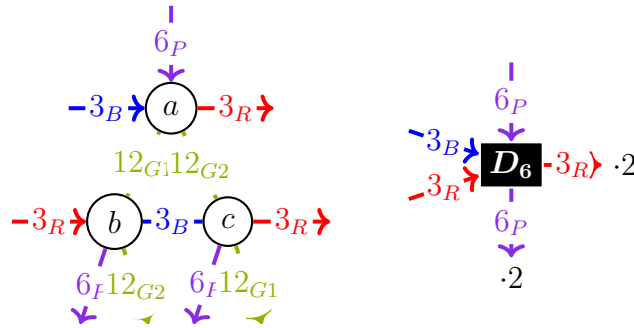


Figure A.4: A gadget for duplicating 6_P input edges (left), with a shorthand version (right). Note that if additional 6_P edges are needed, additional layers can be added, re-using constants from above. Also note that 6_G edges can be duplicated with the same gadget simply by replacing the topmost 6_P edge with a 6_G edge, forcing the 12_1 and 12_2 edges to be purple.

Lemma A.5.2 6-Duplicator Gadget Schedules.

Any slot-respecting schedule must yield a local schedule for each node in any 6-duplicator gadget D_6 of the form:

$$[3_R, 3_B, 12_{G1}, 3_R, 3_B, 6_P, 3_R, 3_B, 12_{G2}, 3_R, 3_B, 6_P].$$

Proof. Each node in D_6 has tasks $[3_R, 3_B, 6_P, 12_{G1}, 12_{G2}]$. It also has density $D = 1$, so each task with frequency f must appear exactly once every f days.

Consider node a , which has inputs 3_B and 6_P . In any slot-respecting schedule these are scheduled in slots of their respective colours, which forces partial schedules for a to be of the form $[_, 3_B, _, _, 3_B, 6_P]$. Exactly two of these slots must be filled by 3_R , and if these are not both red slots, the constraint of 3_R will be violated. Thus, the schedule for a must be of the form:

$$[3_R, 3_B, _, 3_R, 3_B, 6_P, 3_R, 3_B, _, 3_R, 3_B, 6_P].$$

Two spaces remain, which must then contain 12_{G1} and 12_{G2} , as shown in the lemma.

Now consider an arbitrary node other than a . All such nodes will have inputs 12_{Ga} and 3_R , forcing their partial schedules to be of the form:

$$[3_R, _, 12_{Ga}, 3_R, _, _, 3_R, _, _, 3_R, _, _].$$

As with a , exactly four of these slots must schedule 3_B edges, and these must be the blue spaces for the 3_B constraint not to be violated, forcing schedules of the form:

$$[3_R, 3_B, 12_{Ga}, 3_R, 3_B, _, 3_R, 3_B, _, 3_R, 3_B, _].$$

One of these slots must contain the 12_{Gb} edge, with the other two scheduling the 6_P edge. If the 12_{Gb} edge is not scheduled in the remaining green slot, the constraint on the 6_P edge will be violated, so the schedule must be as shown in the lemma. \square

A.6 Clauses

Clauses in C are disjunctions of at most 3 literals, i.e., the logical OR of at most 3 variables, any of which may be negated. A gadget which determines the truth value of a clause given the values of its variables is shown in Figure A.5. Lemma A.6.1 and Remark A.6.2 show that the output of this gadget (12_O) can be scheduled in blue slots iff the corresponding clause is evaluated to be True, though it can always be scheduled in green slots.

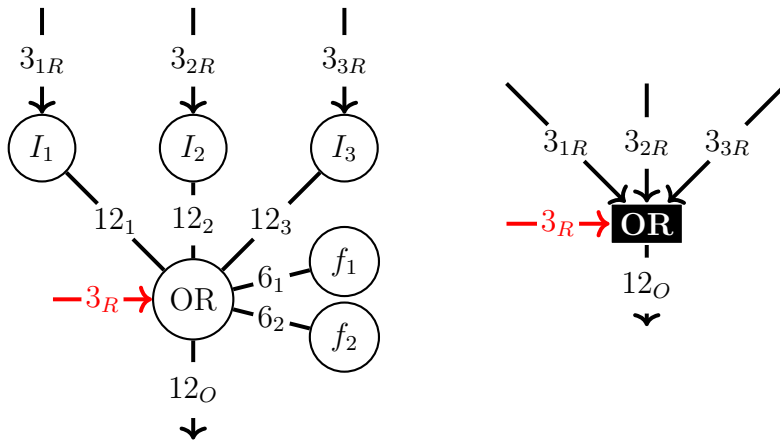


Figure A.5: A gadget which computes $x_1 \vee x_2 \vee x_3$ (left), along with a shorthand version (right). To compute $x_1 \vee x_2 \vee \bar{x}_3$, replace the incoming 3_{3R} edge with a 3_{3B} edge. To instead compute $x_1 \vee x_2$, replace the incoming 3_{3R} edge with a 3_B edge. According to Lemma A.6.1 and Remark A.6.2, 12_O will be scheduled in green slots if all inputs are assigned False and may be scheduled in green or blue slots if any input is assigned True.

Recall the intuitive meaning of slots assigned to the outputs from variable gadgets: a schedule that schedules 3_{iR} in red slots corresponds to a variable assignment where x_i is True. Unfortunately, the output of the OR gadget has to be encoded with a different (and weaker) invariant.

Lemma A.6.1 OR Gadget Schedules.

In any slot-respecting schedule where 3_{1R} , 3_{2R} , and 3_{3R} are scheduled in red or blue slots, the local schedule of the OR gadget has the following form:

- *If all of 3_{1R} , 3_{2R} , and 3_{3R} are scheduled in blue slots, then 12_O will be scheduled in green or purple slots.*
- *If at least one of 3_{1R} , 3_{2R} , and 3_{3R} are scheduled in red slots, then 12_O will be scheduled in green, purple, or blue slots.*

The lemma only covers the clause $x_1 \vee x_2 \vee x_3$, but all other clauses can be handled similarly: For a literal x_j , we use 3_{jR} as input, and for negated literals \bar{x}_j , we use the 3_{jB} edge instead of 3_{jR} .

Proof. The node labelled OR has tasks $[3_R, 6_1, 6_2, 12_1, 12_2, 12_3, 12_O]$ and local density $D = 1$, so each task with frequency f must appear exactly once every f days. Any slot-respecting schedule must schedule the 3_R edge in red slots so schedules for the OR node must be of the form $[3_R, _, _, 3_R, _, _]$.

Consider an inverter node I_i , $i = 1, 2, 3$, which has tasks $[3_{iR}, 12_i]$ where 3_{iR} is red or blue by assumption. Given these constraints, if the input edge 3_{iR} is scheduled in red slots, then partial schedules for I_i must be of the form $[3_{iR}, _, _, 3_{iR}, _, _]$ and the 12_i edge must be scheduled in either green, purple, or blue slots. Similarly, if the input edge 3_{iR} is scheduled in blue slots then partial schedules for I_i must be of the form $[_, 3_{iR}, _, _, 3_{iR}, _]$, restricting the 12_i edge to green, purple, or red slots. However, the 12_i edge is also connected to the OR node which has no empty red slots, further restricting it to green or purple slots.

Suppose that all inputs 3_{1R} , 3_{2R} , 3_{3R} are scheduled in blue slots. By the reasoning above, 12_1 , 12_2 , 12_3 are then scheduled in green or purple slots. This will force schedules for the OR node to be of the form:

$$[3_R, _, \underline{12_a}, 3_R, _, \underline{12_b}, 3_R, _, \underline{12_c}, 3_R, _, _]$$
 or

$$[3_R, _, \underline{12_a}, 3_R, _, \underline{12_b}, 3_R, _, _, 3_R, _, \underline{12_c}]$$

(for some mapping of 12_1 , 12_2 , and 12_3 onto 12_a , 12_b , and 12_c). Assume towards a contradiction that the 12_O edge is blue. This immediately causes a constraint violation when scheduling either 6_1 or 6_2 , which demonstrates that the 12_O edge cannot be blue, and the schedule for the OR node must be of the form:

$$[3_R, \underline{6_a}, \underline{12_a}, 3_R, \underline{6_b}, \underline{12_b}, 3_R, \underline{6_a}, \underline{12_c}, 3_R, \underline{6_b}, \underline{12_d}]$$

(likewise, for some mapping of 12_1 , 12_2 , and 12_3 , and 12_O onto 12_a , 12_b , 12_c , and 12_d). This ensures that the 12_O edge must be scheduled in either green or purple slots if all input edges are blue.

Now suppose that one input edge, 3_{tR} , is scheduled in red slots, while the other two, 3_{iR} and $3_{i'R}$ may be either red or blue. Consider the schedule:

$$[3_R, \underline{12_t}, \underline{6_a}, 3_R, \underline{6_b}, \underline{12_i}, 3_R, \underline{12_O}, \underline{6_a}, 3_R, \underline{6_b}, \underline{12_{i'}}],$$

where 12_O is scheduled in blue slots and no constraints are violated. However, even if 3_{tR} is assigned red, the corresponding edge 12_t may be also be scheduled in green, or purple slots, permitting schedules of the form:

$$[3_R, \underline{6_a}, \underline{12_a}, 3_R, \underline{6_b}, \underline{12_b}, 3_R, \underline{6_a}, \underline{12_c}, 3_R, \underline{6_b}, \underline{12_d}],$$

in which 12_O is scheduled in green or purple slots. This shows that if one or more inputs are red, then 12_O may be scheduled in blue, green, or purple slots. \square

Remark A.6.2 No purple output for OR.

Note that in $\mathcal{P}_{d\varphi k}$, the 12_O edge of an OR gadget will always be connected to the I node of a D_{12} gadget (discussed below, see Figure A.6). Due to this node's 6_P edge, I has no empty purple slots, so the purple slots in Lemma A.6.1 for 12_O are not actually possible once the gadget is part of the overall polycule.

A.7 Sorting Networks

The previous section introduced gadgets whose output edges ($12_{O1}, 12_{O2}, \dots, 12_{Om}$) each capture the truth value of one clause from φ by being blue *or* green if the clause can be satisfied, but restricted to green slots if it cannot. The green ambiguity will be resolved by a gadget that applies *Tension* to the system by forcing a subset of k of the 12_O edges to be scheduled in blue slots in any valid schedule (Section A.8). However, this Tension gadget requires us to pick a *fixed* k -subset; to be able to capture the difference between *any* k clauses being potentially blue and at most $k - 1$ being blue, we build a *sorting network* gadget. This moves edges scheduled in green slots to the right, which in turn moves edges scheduled in blue slots to the left. We can then apply tension to the leftmost k outputs to obtain a polycule which is schedulable iff the corresponding φ has at least k simultaneously satisfiable clauses.

Sorting networks have two components: *wires*, which carry data, and *comparators*¹, which compare two inputs a and b , re-ordering them if necessary. More specifically, the left output always contains $\max\{a, b\}$ and the right output $\min\{a, b\}$. Note that since we are sorting Boolean values, we can compute these as $a \vee b$ and $a \wedge b$, respectively.

Figure A.1 includes the sorting network for 4 inputs where the wires are represented by *channels* of 12_O edges, and comparators by *SWAP* gadgets (introduced below). We use the simple $\Theta(m^2)$ -size insertion/bubble sort network [Knu98, §5.3.4]. While asymptotically better sorting networks are obviously available, this simple method is sufficient for our reduction as we are only aiming for polynomial time overall. We give the general construction here for reference.

Definition A.7.1 DPS Sorting Networks.

Our DPS sorting network takes m input 12_O edges arranged in vertical channels which are connected by layers of SWAP nodes and terminate with output 12_O edges. 12_{O_1} and 12_{O_2} are connected by 1 SWAP node in layer 0, 12_{O_2} and 12_{O_3} are connected by 2 SWAP nodes in layers 1 and -1 , and 12_{O_k} and $12_{O_{(k+1)}}$ are connected by k SWAP nodes in layers $k - 1, k - 3, \dots, 1 - k$.

The goal of the rest of this section is to establish the following lemma for the sorting network:

Lemma A.7.2 Sorting Network Gadget Schedules.

Consider a DPS sorting network with m each of input 12_O edges and output 12_O edges.

- (a)** *(sufficient input) for all ℓ , every slot-respecting schedule in which the leftmost ℓ output edges are scheduled in blue slots must schedule at least ℓ input edges in blue slots.*
- (b)** *(sorted output) for any assignment $C : [m] \rightarrow \{\text{green}, \text{blue}\}$ of colours to input edges such that at least ℓ are blue, there exists a slot-respecting schedule S for the sorting Network that assigns in which the leftmost ℓ output edges are scheduled in blue slots.*

The proof relies on the *SWAP* gadgets, which in turn require a few auxiliary gadgets, so we introduce the latter first before we return to Lemma A.7.2 in Section A.7.6.

A.7.1 12-Edge Duplicator

We begin by introducing our first auxiliary gadget: a frequency-12 edge duplicator, D_{12} , shown in Figure A.6.

¹sometimes called modules or comparator modules

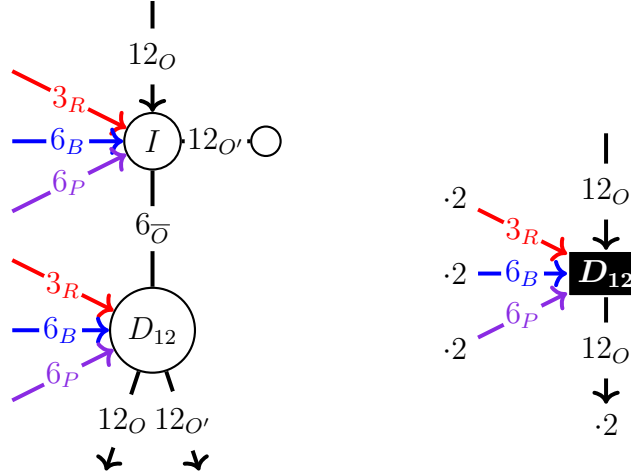


Figure A.6: A gadget for duplicating input edges with frequency 12 (left), and a shorthand version (right). Note that while it can be easily modified to have three 12_O outputs, two are sufficient for our purposes. Also note that the 12_O outputs will not both be scheduled concurrently with the 12_O input, merely in a slot of the same colour (green or blue).

Lemma A.7.3 12-duplicator Gadget Schedules.

Any slot-respecting schedule must yield a local schedule for the D_{12} gadget where all edges labelled 12_O are scheduled in slots of the same colour.

Proof. Nodes I and D_{12} each have tasks $[3_R, 6_B, 6_P, 6_{\bar{O}}, 12_O, 12_{O'}]$ and density $D = 1$, so each task with frequency f must appear exactly once in every f -day period in the schedules for either node. Further, tasks 3_R , 6_B , and 6_P are indirectly connected to the True Clock such that their partial schedules must either be of the form:

$$[3_R, 6_B, _, 3_R, _, 6_P, 3_R, 6_B, _, 3_R, _, 6_P], \quad \text{or}$$

$$[3_R, _, _, 3_R, 6_B, 6_P, 3_R, _, _, 3_R, 6_B, 6_P].$$

Note that all empty slots in both schedules are either blue or green.

Assume towards a contradiction that the $6_{\bar{O}}$ edge is scheduled in a mixture of blue and green slots and note that, as it is also scheduled twice in each 12-day period, its constraint must be violated. This means that either the $6_{\bar{O}}$ edge is consistently green and all 12_O and $12_{O'}$ edges are blue, or the $6_{\bar{O}}$ edge is consistently blue and all 12_O and $12_{O'}$ edges are green. Either way, all 12_O and $12_{O'}$ edges are scheduled in slots of the same colour. \square

A.7.2 OR2 Gadget

Next, we introduce the \vee gadget, shown in Figure A.7. Note that the surrounding construction in the sorting layer provides weaker guarantees than for the OR gadget of the clause layer, requiring a slightly different approach.

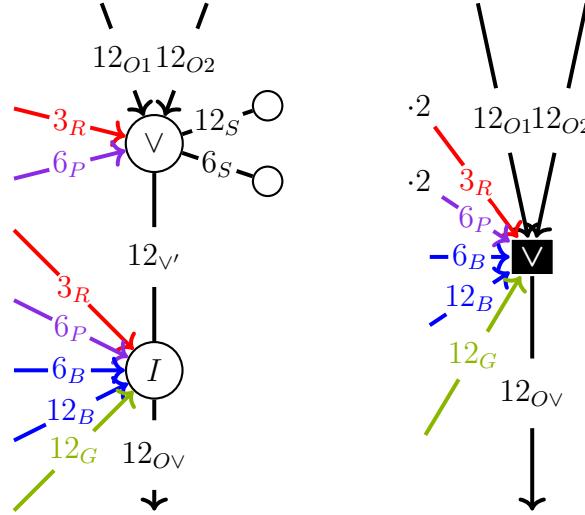


Figure A.7: A gadget for computing $y_1 \vee y_2$ (left), along with a shorthand version (right). 12_{O1} , 12_{O2} , and 12_{OV} can always be scheduled in green slots, but may be scheduled in blue slots only if the corresponding logical term is assigned True. Again, ambiguity introduced by True logical terms being scheduled in green slots will be addressed in Section A.8. Note that while Figure A.5 also calculates a logical OR, its incoming edges both have different frequencies and a different mapping from truth values to edge colours.

Lemma A.7.4 \vee Gadget Schedules.

Any slot-respecting schedule must yield a local schedule for the \vee gadget where the 12_{OV} edge is either blue or green. Further, if the 12_{OV} edge is blue then either 12_{O1} , 12_{O2} , or both are also blue.

Proof. Consider the node labelled “ \vee ” which, by Figure A.7, has tasks $[3_R, 6_P, 6_S, 12_S, 12_{V'}, 12_{O1}, 12_{O2}]$ and density $D = 1$, so each task with frequency f must appear exactly once in every f -day period. Further, 3_R and 6_P are indirectly connected to the True Clock such that partial schedules must be of the form:

$$[3_R, \underline{\quad}, \underline{\quad}, 3_R, \underline{\quad}, 6_P, 3_R, \underline{\quad}, \underline{\quad}, 3_R, \underline{\quad}, 6_P].$$

Suppose that neither 12_{O_1} nor 12_{O_2} is blue – instead, schedule both in the remaining green slots. This forces partial schedules to be of the form:

$$[3_R, _, \underline{12_O}, 3_R, _, 6_P, 3_R, _, \underline{12_{O'}}, 3_R, _, 6_P],$$

that is, schedules where all remaining slots are blue, including the slot assigned to $12_{V'}$.

Now consider the alternative, scheduling some 12_O input edges (12_{O_1} or 12_{O_2} or both) in blue slots. Under this assumption, partial schedules must be of the form:

$$[3_R, \underline{12_O}, _, 3_R, _, 6_P, 3_R, _, _, 3_R, _, 6_P] \quad \text{or}$$

$$[3_R, _, _, 3_R, \underline{12_O}, 6_P, 3_R, _, _, 3_R, _, 6_P].$$

Note that either case allows schedules where $12_{V'}$ is green, and also schedules where $12_{V'}$ is blue. Thus it is always possible for $12_{V'}$ to be scheduled in blue slots, but if 12_{O_1} or 12_{O_2} are scheduled in blue slots then $12_{V'}$ must be scheduled in green slots.

Now consider schedules for node I , which has tasks $[3_R, 6_P, 6_B, 12_B, 12_G, 12_{V'}, 12_{O_V}]$, and has density $D = 1$, with the same implication. Partial schedules for I have two free slots in each 12-day period, with all other slots taken by $3_R, 6_P, 6_B, 12_B$, and 12_G – all of which must be scheduled in slots of their corresponding colour due to their indirect connections to the True Clock. These free slots are blue and green, so either $12_{V'}$ is blue and 12_{O_V} must be green, or $12_{V'}$ is green and 12_{O_V} must be blue.

It, therefore, follows that if both 12_{O_1} and 12_{O_2} are green then $12_{V'}$ will be blue and 12_{O_V} must be green, while if either 12_{O_1} or 12_{O_2} are scheduled in blue slots then $12_{V'}$ and 12_{O_V} may both be scheduled in either green or blue slots. \square

A.7.3 AND2 Gadget

We will now introduce the \wedge gadget, shown in Figure A.8.

Lemma A.7.5 \wedge Gadget Schedules.

Any slot-respecting schedule must yield a local schedule for the \wedge gadget where the 12_{O_\wedge} edge is either blue or green. Further, if the 12_{O_\wedge} edge is blue then both 12_{O_1} and 12_{O_2} must also be blue.

Proof. Consider the node labelled “ \wedge ” in Figure A.8, which has tasks $[3_R, 6_P, 6_\wedge, 12_{O_1}, 12_{O_2}, 12_S, 12_{S'}]$ and density $D = 1$, forcing each task with frequency f to appear exactly once in

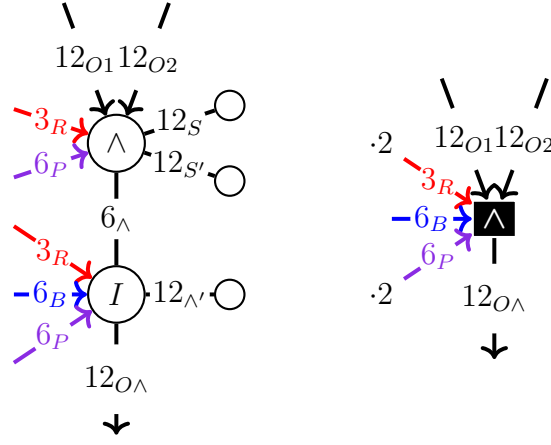


Figure A.8: A gadget for computing $y_1 \wedge y_2$ (left), along with a shorthand version (right). Here, 12_{O_i} will be scheduled in blue or green slots if y_i is True and restricted to green slots if y_i is False. Similarly, 12_{O^\wedge} will be scheduled in blue or green slots if $y_1 \wedge y_2$ is True and restricted to green slots otherwise. As with other such gadgets, ambiguities resulting from edges corresponding to True variables being scheduled in green slots will be addressed in Section A.8.

every f -day period. Further, 3_R and 6_P are indirectly connected to the True Clock such that partial schedules must be of the form $[3_R, _, _, 3_R, _, 6_P, 3_R, _, _, 3_R, _, 6_P]$.

Assume towards a contradiction that the 6_\wedge edge is scheduled in a mixture of blue and green slots. Under this assumption, either 6_\wedge occurs more than once in some 6-day periods, or its constraint must be violated. Thus, it must be scheduled consistently in either blue slots or green slots.

Suppose that some 12_O edge, either 12_{O_1} or 12_{O_2} , is scheduled in a green slot. In this case, partial schedules for \wedge will either be of the form:

$$[3_R, _, \underline{12_O}, 3_R, _, 6_P, 3_R, _, _, 3_R, _, 6_P] \quad \text{or}$$

$$[3_R, _, _, 3_R, _, 6_P, 3_R, _, \underline{12_O}, 3_R, _, 6_P].$$

Either way, 6_\wedge must then be scheduled in a blue slot to avoid constraint violations. If we instead suppose that both 12_{O_1} and 12_{O_2} are scheduled in blue slots, then two valid schedules for \wedge are:

$$[3_R, \underline{6_\wedge}, \underline{12_S}, 3_R, \underline{12_{O_1}}, 6_P, 3_R, \underline{6_\wedge}, \underline{12_{S'}}, 3_R, \underline{12_{O_2}}, 6_P] \quad \text{and}$$

$$[3_R, \underline{12_{O_1}}, \underline{6_\wedge}, 3_R, \underline{12_{O_2}}, 6_P, 3_R, \underline{12_S}, \underline{6_\wedge}, 3_R, \underline{12_{S'}}, 6_P],$$

demonstrating that in this case 6_P may be scheduled in either blue or green slots.

Now consider the node labelled I , which has tasks $[3_R, 6_B, 6_P, 6_\wedge, 12_{O_\wedge}, 12_{O_{\wedge'}}]$, and density $D = 1$ with the same implication as above for the \wedge node. Again, partial schedules of the coloured nodes (3_R , 6_B , and 6_P) are restricted by their indirect connections to the True Clock, and must either be of the form:

$$[3_R, 6_B, _, 3_R, _, 6_P, 3_R, 6_B, _, 3_R, _, 6_P] \quad \text{or}$$

$$[3_R, _, _, 3_R, 6_B, 6_P, 3_R, _, _, 3_R, 6_B, 6_P].$$

In either case, if 6_\wedge is scheduled in blue slots then only green slots remain for 12_{O_\wedge} and $12_{O_{\wedge'}}$. Likewise, if 6_\wedge is scheduled in green slots then 12_{O_\wedge} and $12_{O_{\wedge'}}$ must be scheduled in the remaining blue slots. Thus 12_{O_\wedge} must be either green or blue, and further can only be blue if 6_\wedge is green, which is only possible if both 12_{O_1} and 12_{O_2} are blue, as claimed. \square

A.7.4 Slot Splitting Gadgets

Several of the gadgets introduced in this section have constant input edges which are scheduled in slots with a specific colour, but with a larger frequency than those produced by the gadgets introduced in Section A.5. These can be produced by the simple slot-splitter gadgets shown in Figure A.9.

Lemma A.7.6 Slot Splitting Gadget Schedules.

Any slot-respecting schedule must yield a local schedule for the slot splitting gadgets where 6_B , 12_B , and 12_G edges produced by S_{B6} , S_{B12} , and S_{G12} nodes must be scheduled in blue, blue, and green slots, respectively.

Proof. In addition to their respective output edges, the S_{B6} and S_{B12} nodes each have 3_R , 6_G , and 6_P input edges, which are connected indirectly to the True Clock such that they must be scheduled in slots of their respective colour, with partial schedules of the form: $[3_R, _, 6_G, 3_R, _, 6_P, 3_R, _, 6_G, 3_R, _, 6_P]$. In these schedules, all remaining slots are blue, so the output edges (6_B and $6_{B'}$ for the S_{B6} node, and 12_B , $12_{B'}$, $12_{B''}$, $12_{B'''}$ for the S_{B12} node) must be scheduled in blue slots.

Similarly, the S_{G12} node has 3_R , 3_B , and 6_P input edges, and partial schedules of the form $[3_R, 3_B, _, 3_R, 3_B, 6_P, 3_R, 3_B, _, 3_R, 3_B, 6_P]$, where, as above, all remaining slots are green, so the output edges (12_G and $12_{G'}$) must be scheduled in green slots. \square

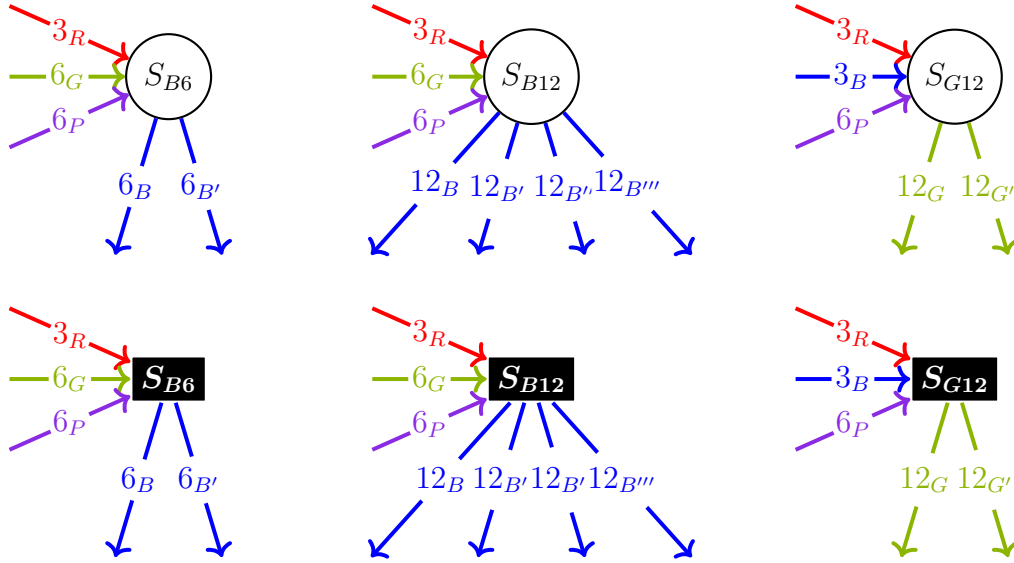


Figure A.9: Gadgets for generating 6_B edges (top left) and 12_B edges (top centre), and 12_G edges (top right), with shorthand versions below. Excess 6_B , 12_B , and 12_G edges should be connected to pendent nodes.

A.7.5 SWAP Gadgets

With these preparations, we are finally able to build the *SWAP* gadgets which act as comparators in the sorting network. The internal structure of *SWAP* gadgets is shown in Figure A.10.

Lemma A.7.7 *SWAP* Gadget Schedules.

Any slot-respecting schedule must yield a local schedule for the SWAP gadget with the following properties:

- Both output edges, $12_{O\vee}$ and $12_{O\wedge}$, are scheduled in either green or blue slots.
- If $12_{O\vee}$ is blue, then either 12_{O1} , 12_{O2} , or both are blue.
- If $12_{O\wedge}$ is blue, then both 12_{O1} and 12_{O2} are blue.

Note that this is again a weaker invariant than a true \vee and \wedge ; *SWAP* correctly sorts its inputs, sending green to the right and blue to the left, but it is allowed to “swallow” a blue value, turning it green. Fortunately, this is good enough for our purposes, as we will go on to demonstrate.

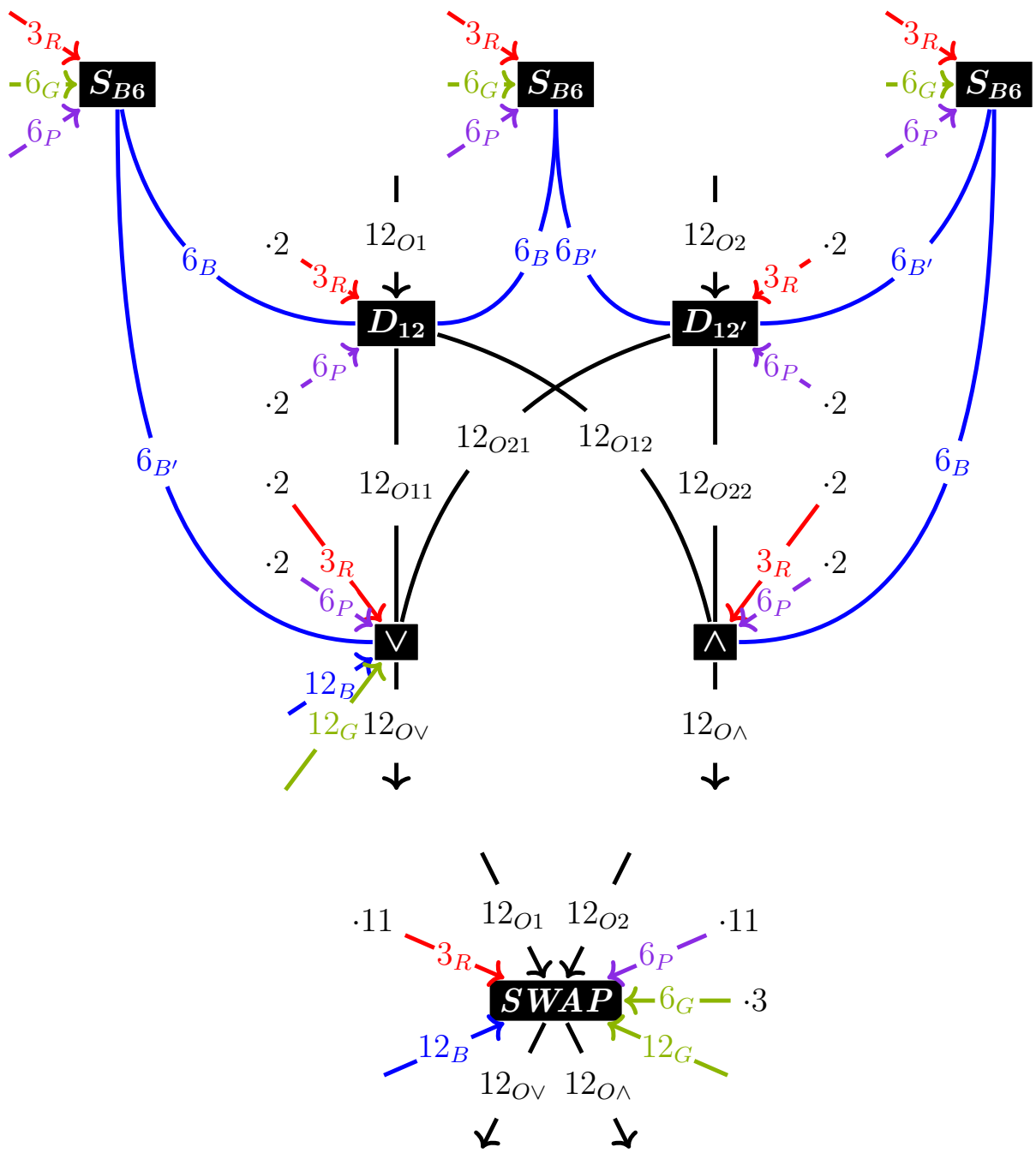


Figure A.10: A gadget for comparing and re-ordering two input edges (top), and a shorthand version (bottom). The 6_B , 12_B , and 12_G edges used by $SWAP$ nodes come from slot splitting gadgets, which are not shown because they are shared between multiple $SWAP$ nodes. If the gadget has as many outputs as it has inputs, 12_{O1} and 12_{O2} must be scheduled in blue or green slots and $12_{O\vee}$ and $12_{O\wedge}$ will match them in all cases save one: if 12_{O1} is green and 12_{O2} is blue, $12_{O\vee}$ will be blue and $12_{O\wedge}$ will be green.

Proof of Lemma A.7.7. Assume a slot-respecting global schedule S is given, i.e., all coloured edges are scheduled in slots of their respective colours. At D_{12} , the $12_{O_{11}}$ and $12_{O_{12}}$ edges must be scheduled in slots of the same colour as 12_{O_1} ; similarly, at $D_{12'}$, edges $12_{O_{21}}$ and $12_{O_{22}}$ must be scheduled in slots of the same colour as 12_{O_2} (both by Lemma A.7.3). By Lemma A.7.4, $12_{O_{\vee}}$ must be green or blue, and can only be blue if at least one of its inputs, $12_{O_{11}}$ and $12_{O_{21}}$, is also blue. Similarly, by Lemma A.7.5, $12_{O_{\wedge}}$ must be green or blue, and further can only be blue if both of its inputs, $12_{O_{12}}$ and $12_{O_{22}}$, are also blue. \square

A.7.6 Sorting Network Schedules

We now finally show that under the assumptions enforced on the schedule by the surrounding gadgets, our sorting network correctly groups green edges on the right, with enough blue edges on the left to suit our purposes.

Lemma A.7.2 (restated). Sorting Network gadget schedules.

Consider a DPS sorting network with m each of input 12_O edges and output 12_O edges.

- (a) *(sufficient input) for all ℓ , every slot-respecting schedule in which the leftmost ℓ output edges are scheduled in blue slots must schedule at least ℓ input edges in blue slots.*
- (b) *(sorted output) for any assignment $C : [m] \rightarrow \{\text{green}, \text{blue}\}$ of colours to input edges such that at least ℓ are blue, there exists a slot-respecting schedule S for the sorting Network that assigns in which the leftmost ℓ output edges are scheduled in blue slots.*

Proof of Lemma A.7.2. (a) Assume a slot-respecting schedule S that schedules the leftmost ℓ output edges in blue slots. Then, starting at the outputs of the sorting network, move backwards through the sorting network, one *SWAP* gadget at a time (sweeping from bottom to top in Figure A.1). We will show by induction that while moving through the sorting network in this way, there are always at least ℓ blue edges between all channels. Most importantly, this holds at the inputs – implying the claim.

The inductive basis at the output level is true by assumption. When moving over each *SWAP* gadget, we replace the two outputs of that *SWAP* with its two inputs. By Lemma A.7.7, there are several different valid colour combinations for the inputs, but we always have at least as many blue values among the inputs as are among the outputs: If both outputs are green or blue, this is obvious. If exactly one output is blue it will either be the $12_{O_{\vee}}$ input (implying that at least one input is blue), or the $12_{O_{\wedge}}$ input (implying that both inputs are blue, but the \vee gadget chose a green output). Hence the number of blue values cannot drop when moving over a *SWAP* gadget.

(b) We build the local slot-respecting schedule inductively by again moving through the sorting network one *SWAP* node at a time, only now we will move forwards (that is, top to bottom in Figure A.1). We will also restrict S to the special case S' , where each *SWAP* node has as many blue output 12_O edges as it has blue input 12_O edges (possible due to Lemma A.7.7). Note that not all schedules satisfying the lemma will necessarily have this property, but it is sufficient to show that one such schedule exists.

In this class of schedules S' , *SWAP* gadgets behave exactly like the comparator modules in a sorting network for binary inputs: if both inputs are blue then both outputs will be blue, if both inputs are green then both outputs will be green, and if exactly one input is blue then the 12_{O_V} output will be blue and the 12_{O_\wedge} output will be green (all by Lemma A.7.7). By the correctness of the insertion/bubble sort network, we hence end up with an output layer with exactly ℓ blue edges on the left, followed by $m - \ell$ green edges on the right. \square

A.8 Tension

Lemma A.7.2 assumes that the leftmost k output edges of a sorting network must be scheduled in blue slots – we ensure this by connecting Tension gadgets (shown in Figure A.11) to these output edges.

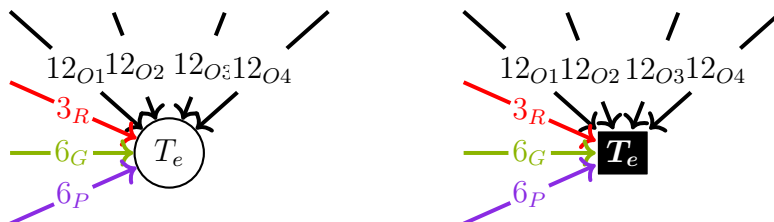


Figure A.11: A gadget (left) which applies tension to four inputs, ensuring that a schedule for T_e can be found iff all 12_O inputs are scheduled in blue slots. To apply tension to less than four inputs, connect any unneeded input to its own pendent node.

Lemma A.8.1 Tension Gadget Schedules.

Any slot-respecting schedule must yield a local schedule for a tension gadget T_e where each 12_O edge must be scheduled in blue slots.

Proof. The single node of each tension gadget T_e , as shown in Figure A.11, has tasks $[3_R, 6_G, 6_P, 12_{O_1}, 12_{O_2}, 12_{O_3}, 12_{O_4}]$ and density $D = 1$, so each task with frequency f must

appear exactly once in every f -day period. Further, the 3_R , 6_G , and 6_P edges are indirectly connected to the True Clock such that they must be scheduled in slots of their respective colours. Partial schedules for T_e nodes must therefore be of the form $[3_R, _, 6_G, 3_R, _, 6_P, 3_R, _, 6_G, 3_R, _, 6_P]$. All empty slots in schedules of this form are blue, so the remaining $12_{O1}, 12_{O2}, 12_{O3}, 12_{O4}$ edges must be scheduled in blue slots. \square

A.9 Correctness Proof of Reduction

With these preparations we can, at long last, prove Lemma A.1.1, and hence complete the proof of Theorem 5.1.1.

Lemma A.1.1 (restated). D-MAX-3SAT \leq_p DPS.

For any 3-CNF formula φ with m clauses and integer $k \leq m$, we can construct in polynomial time a decision polycule $\mathcal{P}_{d\varphi k}$ which has a valid schedule if and only if at least k clauses of φ can be simultaneously satisfied.

Proof of Lemma A.1.1. Consider a polycule $\mathcal{P}_{d\varphi k}$ built from some 3-CNF formula φ using the algorithm defined by Definition A.2.1.

First, suppose that there is a variable assignment $v : X \rightarrow \{\text{True}, \text{False}\}$ such that clauses c_{i_1}, \dots, c_{i_k} evaluate to True under v ($1 \leq i_1 < i_2 < \dots < i_k \leq m$). We construct a slot-respecting schedule S for $\mathcal{P}_{d\varphi k}$ as follows: At the variable gadget for each $x_i \in X$, schedule 3_{iR} in red slots if $v(x_i) = \text{True}$ and in the blue slots otherwise. This fixes a schedule for all edges in the variable and duplication layers (by Lemma A.5.1 and Lemma A.5.2). In the clause layer, we schedule the 12_O output edges of the OR gadgets for clauses c_{i_j} , $j = 1, \dots, k$, in blue slots, and schedule all other 12_O output edges in green slots (noting that, by Lemma A.6.1, this yields a valid schedule for all OR gadgets). In the sorting layer, we now have at least k blue inputs and at most $m - k$ green inputs. By Lemma A.7.2–(b), we can extend this schedule to the sorting layer such that the leftmost k output edges of the sorting layer are scheduled in blue slots. This then also yields a valid schedule for the Tension gadgets (Lemma A.8.1). Overall, this shows that a schedule S for $\mathcal{P}_{d\varphi k}$ exists.

Now assume that we are given a schedule S for $\mathcal{P}_{d\varphi k}$. By applying Lemma A.4.1 to the True Clock, we can assign coloured slots to S . It then follows from the construction of $\mathcal{P}_{d\varphi k}$ that S must be slot-respecting. Set $v(x_i) = \text{True}$ if S schedules 3_{iR} in red slots and $v(x_i) = \text{False}$ otherwise. Next, show that v satisfies at least k clauses in φ : By Lemma A.8.1, the k leftmost output edges of the sorting layer must all be scheduled in blue slots. By

Lemma A.7.2–(a), this means that there are (at least) k output edges of the clause layer that are scheduled in blue slots, say those for clauses c_{i_1}, \dots, c_{i_k} for some $1 \leq i_1 < i_2 < \dots < i_k \leq m$. Considering this along with Lemma A.6.1 implies that for each c_{i_j} , at least one input is scheduled in a red slot. By the definition of v , this means that clauses c_{i_1}, \dots, c_{i_k} all evaluate to True under v , and the claim follows.

It remains to argue that our reduction can be realised in polynomial time. The size of the polycule $\mathcal{P}_{d\varphi k}$ is clearly polynomial in the size of the formula, with the quadratic sorting network contributing the most persons. All other gadgets, including the *SWAP* nodes composing the sorting network, have constant size and thus are easy to implement in polynomial time. \square

* * * * *

Appendix B

Powersort – C++ Code

We provide some key parts of our C++ implementation (slightly redacted for readability).

B.1 4-way merge with sentinels

We first give the sentinel-based 4-way merge method; this is the method that gave the best performance.

```
/**
 * Merge runs [l..g1) and [g1..g2) and [g2..g3) and [g3..r) in-
 * ↪ place into [l..r)
 * using a buffer at B of length at least r-l+4.
 */.
template<typename Iter, typename Iter2>
void merge_4runs(Iter l, Iter g1, Iter g2, Iter g3, Iter r, Iter2
↪ B) {
    auto n = r - l;
    // Copy runs to B and append a sentinel value after each.
    std::copy(l, g1, B);                *(B + (g1 - l)) =
    ↪ plus_inf_sentinel();
    std::copy(g1, g2, B + (g1 - l) + 1); *(B + (g2 - l) + 1) =
    ↪ plus_inf_sentinel();
    std::copy(g2, g3, B + (g2 - l) + 2); *(B + (g3 - l) + 2) =
    ↪ plus_inf_sentinel();
```

```

std::copy(g3, r, B + (g3 - 1) + 3); *(B + (r - 1) + 3) =
    ↪ plus_inf_sentinel();
Iter2 c[4]; // pointers to runs in B.
c[0] = B, c[1] = B + (g1-1)+1, c[2] = B + (g2-1)+2, c[3] = B +
    ↪ (g3-1)+3;
// initialize tournament tree
Iter2 x, y; std::pair<Iter2, bool> z;
//           z           x,y,z store iterator
//   /       \         from winning run;
//  x         y         z also whether min
// / \       / \       min came from left subtree
// 0  1     2  3
if (*c[0] <= *c[1]) x = c[0]++; else x = c[1]++;
if (*c[2] <= *c[3]) y = c[2]++; else y = c[3]++;
if (*x <= *y) z = {x, true}; else z = {y, false};

*l++ = *(z.first); // vacate root into output
for (auto i = 1; i < n; ++i) {
    if (z.second) { // min came from c[0] or c[1], so
        ↪ recompute x.
        if (*c[0] <= *c[1]) x = c[0]++; else x = c[1]++;
    } else { // min came from c[2] or c[3], so recompute y.
        if (*c[2] <= *c[3]) y = c[2]++; else y = c[3]++;
    }
    // always recompute z
    if (*x <= *y) z = {x, true}; else z = {y, false};
    *l++ = *(z.first);
}
}

```

B.2 3-way merge and 2-way merge

The method for merging 3 runs is similar, but doesn't use the right subtree; instead y is simply also set to $c[2]$. For 2-way merging many variations have been explored; when the

type supports a sentinel value, the following method has a very efficient merging loop.

```

void merge_2runs(Iter l, Iter m, Iter r, Iter2 B) {
    auto n1 = m-l, n2 = r-m;
    std::copy(l, m, B);          *(B+(m-1)) = plus_inf_sentinel
        ↪ ();
    std::copy(m, r, B+(m-1+1)); *(B+(r-1)+1) = plus_inf_sentinel
        ↪ ();
    auto c1 = B, c2 = B + (m - l + 1), o = l;
    while (o < r) *o++ = *c1 <= *c2 ? *c1++ : *c2++;
}

```

B.3 4-way Powersort

Our implementation of 4-way Powersort uses a stack of records maintained in a fixed-size array. As in Timsort, we extend the natural runs in the input to a minimal length `minRunLen`; setting this parameter to 1 disables that optimization.

```

struct run { Iter begin; Iter end; };
struct run_n_power { Iter begin; Iter end; int power = 0; };
struct run_begin_n_power { Iter begin; int power; };
run_begin_n_power NULL_RUN_N_POWER{};

/** sorts [begin, end) */
void powersort_4way(Iter begin, Iter end) {
    auto n = end - begin;
    auto maxStackHeight = 3*(ceil_log4(n)+1);
    auto stack = new run_n_power[maxStackHeight];
    run_begin_n_power *top = stack; // topmost valid stack element
    *top = NULL_RUN_N_POWER; // power 0 as sentinel entry
    run_begin_n_power * const endOfStack = stack + maxStackHeight;

    run_n_power runA = {begin, extend_and_reverse_run(begin, end),
        ↪ 0};
    // extend to minimal run length

```

```

if (auto lenA = runA.end - runA.begin < minRunLen) {
    runA.end = std::min(end, runA.begin + minRunLen);
    insertion_sort(runA.begin, runA.end, lenA); // skips first
        ↪ lenA iterations
}
while (runA.end < end) {
    run runB = {runA.end, extend_and_reverse_run(runA.end, end
        ↪ )};
    if (size_t lenB = runB.end - runB.begin < minRunLen) {
        runB.end = std::min(end, runB.begin + minRunLen);
        insertion_sort(runB.begin, runB.end, lenB);
    }
    runA.power = node_power(0, n, runA.begin - begin, runB.
        ↪ begin - begin, runB.end - begin);
    // Invariant: powers on stack are weakly increasing from
        ↪ bottom to top
    while (top->power > runA.power)
        merge_loop(top, runA);
    *(++top) = {runA.begin, runA.power}; // push new run onto
        ↪ stack
    runA = {runB.begin, runB.end, 0};
}
merge_down(stack, top, runA);
delete [] stack;
}

```

To make the code more readable, the body of the main loop has been moved to a function `merge_loop`; `g++` inlined the call as part of its compiler optimizations. `merge_loop` counts the number of (contiguous) entries on top of the stack with equal power and then merges these with `runA`.

```

void merge_loop(run_begin_n_power * &top_of_stack, run_n_power &
    ↪ runA) {
    int nRunsSamePower = 1;
    while ((top_of_stack - nRunsSamePower)->power == top_of_stack->

```

```

    ↪ power)
    ++nRunsSamePower;
if (nRunsSamePower == 1) { // 2way
    Iter g[] = {top_of_stack->begin};
    merge_2runs(g[0], runA.begin, runA.end, _buffer.begin());
    runA.begin = g[0];
} else if (nRunsSamePower == 2) { // 3way
    Iter g[] = {(top_of_stack-1)->begin, top_of_stack->begin};
    merge_3runs(g[0], g[1], runA.begin, runA.end, _buffer.
        ↪ begin());
    runA.begin = g[0];
} else { // 4way
    Iter g[] = {(top_of_stack-2)->begin, (top_of_stack-1)->
        ↪ begin, top_of_stack->begin};
    merge_4runs(g[0], g[1], g[2], runA.begin, runA.end,
        ↪ _buffer.begin());
    runA.begin = g[0];
}
top_of_stack -= nRunsSamePower; // pop runs
}

```

The function `merge_down` successively merges the top 4 elements on the stack; since runs are typically exponentially increasing in size as we work our way through the stack, it is beneficial to first bring the number of runs to $3k + 1$ for a $k \in \mathbb{N}$ using a single 2-way or 3-way merge at the beginning. Then all remaining merges are 4-way merges.

```

void merge_down(run_begin_n_power *begin_of_stack,
    ↪ run_begin_n_power * &top_of_stack, run_n_power &runA) {
    // we have the entire stack of runs, so instead of following
        ↪ exactly the powersort rule, we can
    // be slightly more clever and make sure we have 4way merges
        ↪ all the way through except the first merge
    auto nRuns = top_of_stack - begin_of_stack + 1; // stack size
        ↪ + runA
    // We want 3k+1 runs, so that repeatedly merging 4 and putting

```

↪ *the result back gives 4way merges all the way through.*

```

switch (nRuns % 3) {
  case 0: // merge topmost 3 runs
    merge_3runs<mergingMethod>((top_of_stack-1)->begin ,
      ↪ top_of_stack->begin ,
                                     runA.begin , runA.end ,
                                     ↪ _buffer.begin());
    runA.begin = (top_of_stack-1)->begin;
    top_of_stack -= 2;
    break;
  case 2: // merge topmost 2 runs
    merge_2runs(top_of_stack->begin , runA.begin , runA.end ,
      ↪ _buffer.begin());
    runA.begin = top_of_stack->begin;
    —top_of_stack;
    break;
  default :
    break;
}
assert(((top_of_stack - begin_of_stack) % 3) == 0);
// merge remaining stack 4way each
while (top_of_stack > begin_of_stack) {
  merge_4runs((top_of_stack-2)->begin , (top_of_stack-1)->
    ↪ begin ,
                                     top_of_stack->begin , runA
                                     ↪ .begin , runA.end ,
                                     ↪ _buffer.begin());
  runA.begin = (top_of_stack-2)->begin;
  top_of_stack -= 3;
}
}

```

Note that this merge-down strategy can lead to a slightly different merge tree compared to the “pure Powersort” merge tree realized by `kway_treek`; however, it is easy to show this change can only reduce the resulting merge cost.

B.4 Sentinel-free 2-way merge

If $+\infty$ values are not available, we need to include boundary checks. This changes the code to the following method.

```
void merge_2runs_no_sentinel(Iter l, Iter m, Iter r, Iter2 B)
    ↪ {
    auto n1 = m-l, n2 = r-m;
    std::copy(l,r,B);
    auto c1 = B, e1 = B + n1, c2 = e1, e2 = e1 + n2; auto
        ↪ o = l;
    while (c1 < e1 && c2 < e2) *o++ = *c1 <= *c2 ? *c1++
        ↪ : *c2++;
    while (c1 < e1) *o++ = *c1++;
    while (c2 < e2) *o++ = *c2++;
}
```

For 2-way merging, we also use the following merge method that only copies the smaller run to the buffer.

```
void merge_2runs_copy_smaller(Iter l, Iter m, Iter r, Iter2 B)
    ↪ {
    auto n1 = m-l, n2 = r-m;
    if (n1 <= n2) {
        std::copy(l,m,B);
        auto c1 = B, e1 = B + n1; auto c2 = m, e2 = r, o = l;
        while (c1 < e1 && c2 < e2) *o++ = *c1 <= *c2 ? *c1++ : *
            ↪ c2++;
        while (c1 < e1) *o++ = *c1++;
    } else {
        std::copy(m,r,B);
        auto c1 = m-1, s1 = l, o = r-1; auto c2 = B+n2-1, s2 = B;
        while (c1 >= s1 && c2 >= s2) *o-- = *c1 <= *c2 ? *c2-- :
            ↪ *c1--;
        while (c2 >= s2) *o-- = *c2--;
    }
}
```

B.5 Sentinel-free 4-way merge (merging by stages)

We created a 4-way merging method that does not use sentinels and works in “stages” (as described in the main text). We use C++ templates to exploit similarities of the stages; during compilation, these templates are unfolded and code optimization runs on the deflated code.

```
struct tournament_tree_node {  Iter2 it; bool fromRun0Or1;  };  
enum number_runs {  TWO = 2, THREE = 3, FOUR = 4  };
```

```
template<number_runs nRuns>  
long compute_safe(std::vector<Iter2> &c, std::vector<Iter2> &e,  
    ↪ std::vector<long> &nn) {  
    for (int i = 0; i < nRuns; ++i) nn[i] = e[i] - c[i];  
    long safe = *(std::min_element(nn.begin(), nn.end()));  
    return safe;  
}
```

```
template<number_runs nRuns>  
void initialize_tournament_tree(std::vector<Iter2> &c, std::vector  
    ↪ <Iter2> &e,  
                                std::array<tournament_tree_node<  
                                ↪ Iter2 >, 3> &N) {  
    // tournament tree:  
    //      N[0]  
    //     /  \  
    //  N[1]  N[2]  
    //  / \  
    // 0  1  2  3  
    if (*c[0] <= *c[1]) N[1] = {c[0]++, true}; else N[1] = {c  
        ↪ [1]++, true};  
    if (nRuns == 4)  
        if (*c[2] <= *c[3]) N[2] = {c[2]++, false}; else N[2] = {c  
            ↪ [3]++, false};  
    else
```

```

        N[2] = {c[2]++, false};
    N[0] = *(N[1].it) <= *(N[2].it) ? N[1] : N[2];
}

template<number_runs nRuns>
void update_tournament_tree(std::vector<Iter2> &c, std::vector<
    ↪ Iter2> &e,
                            std::array<tournament_tree_node<Iter2
    ↪ >, 3> &N) {
    if (N[0].fromRun0Or1) {
        if (*c[0] <= *c[1]) N[1] = {c[0]++, true}; else N[1] = {c
    ↪ [1]++, true};
    } else { // otherwise min came from c[2] or c[3], so recompute
    ↪ y.
        if (nRuns == 4)
            if (*c[2] <= *c[3]) N[2] = {c[2]++, false}; else N[2]
    ↪ = {c[3]++, false};
        else
            N[2] = {c[2]++, false};
    }
    // always recompute z
    N[0] = *(N[1].it) <= *(N[2].it) ? N[1] : N[2];
}

```

```

template<number_runs nRuns>
bool rollback_tournament_tree(std::vector<Iter2> &c, std::vector<
    ↪ Iter2> &e,
                              std::array<tournament_tree_node<
    ↪ Iter2 >, 3> &N,
                              std::vector<long> &nn) {
    auto other = N[0].fromRun0Or1 ? N[2] : N[1];
    // roll back into 'its' run
    int rollbacks = 0;
    for (auto i = 0; i < nRuns; ++i)

```

```

    if (c[i] - 1 == other.it) {
        --c[i], ++nn[i], ++rollbacks; break;
    }
int i = std::find(nn.begin(), nn.end(), 0) - nn.begin();
if (i == nRuns) {
    // rolled back into run that got empty; nasty special case
    ↪ .
    // But we made progress in the root, so just continue one
    ↪ more round with same nRuns.
    // need to rebuild the tree for that
    initialize_tournament_tree<nRuns>(c, e, N);
    return false;
} else {
    c.erase(c.begin() + i);
    e.erase(e.begin() + i);
    return true;
}
}

```

```

template<number_runs nRuns>
bool do_merge_runs(Iter &l, Iter const r, std::vector<Iter2> &c,
    ↪ std::vector<Iter2> &e) {
    if (nRuns == TWO) {
        // simple twoway merge
        while (c[0] < e[0] && c[1] < e[1])    *l++ = *c[0] <= *c[1]
            ↪ ? *c[0]++ : *c[1]++;
        while (c[0] < e[0]) *l++ = *c[0]++;
        while (c[1] < e[1]) *l++ = *c[1]++;
        return true;
    } else {
        // use tournament tree
        std::array<tournament_tree_node<Iter2>, 3> N;
        initialize_tournament_tree<nRuns>(c, e, N);
        std::vector<long> nn(nRuns); // run sizes
    }
}

```

```

while (l < r) {
    long safe = compute_safe<nRuns>(c, e, nn);
    if (safe > 0) {
        for (; safe > 0; —safe) {
            *l++ = *(N[0].it); // output root
            update_tournament_tree<Iter2, nRuns>(c, e, N);
        }
    } else {
        // one run is exhausted; need to handle elements
        ↪ in the tree
        *l++ = *(N[0].it); // easy for the root (
        ↪ guaranteed min)
        // rollback other element into its run
        if (rollback_tournament_tree<nRuns>(c, e, N, nn))
            // occasionally, we rollback into an empty run
            ↪ and have to keep going; otherwise,
            ↪ terminate loop.
            break;
    }
}
return false;
}
}

```

```

void detect_and_remove_empty_runs(std::vector<Iter2> & c, std::
↪ vector<Iter2> & e) {
    int nRuns = c.size(); long safe;
    while (true) {
        std::vector<long> nn(nRuns);
        for (int i = 0; i < nRuns; ++i) nn[i] = e[i] - c[i];
        safe = *(std::min_element(nn.begin(), nn.end()));
        if (safe > 0) return;
        int i = std::find(nn.begin(), nn.end(), 0) - nn.begin();
        c.erase(c.begin() + i); e.erase(e.begin() + i); —nRuns;
    }
}

```

```

    }
}

template<typename Iter, typename Iter2>
void merge_4runs_no_sentinels(Iter l0, Iter g1, Iter g2, Iter g3,
    ↪ Iter r, Iter2 B) {
    Iter l = l0; const auto n = r - l;
    std::copy(l, g1, B);
    std::copy(g1, g2, B + (g1 - l));
    std::copy(g2, g3, B + (g2 - l));
    std::copy(g3, r, B + (g3 - l));
    *(B+n) = *(B+n-1); // sentinel value so that accesses to
        ↪ endpoints don't fail
    std::vector<Iter2> c {B, B+(g1-l), B+(g2-l), B+(g3-l)    };
        ↪ // current element
    std::vector<Iter2> e { B+(g1-l), B+(g2-l), B+(g3-l), B+n};
        ↪ // endpoints
    detect_and_remove_empty_runs(c, e);
    while (l < r) {
        switch (c.size()) {
            case 4: if (do_merge_runs<FOUR> (l, r, c, e)) break;
            case 3: if (do_merge_runs<THREE>(l, r, c, e)) break;
            case 2: if (do_merge_runs<TWO> (l, r, c, e)) break;
            case 1: return;
        };
    }
}

```

* * * * *

Glossary

Bamboo Garden Trimming A periodic scheduling problem in which a single agent schedules trimming of k plants whose height h grows daily, with the goal of minimizing the maximum observed height of any plant. Instances are represented as $\mathcal{B} = (g) = (g_1, g_2, \dots, g_k)$, while classes of Bamboo Garden Trimming problems are represented by **B**. 17, 18, 24–28, 31, 64, 74, 75, 140, 142, 144, 193

Decision Polyamorous Scheduling A periodic scheduling problem in which persons are connected by a graph and must schedule pairwise meetings, with constraints on the maximum time between such meetings. Instances are represented as $\mathcal{D} = (P, R, f)$, where P is a set of n persons, R is a set of relationships, and f is a set of frequencies, one per relationship. Classes of Decision Polyamorous Scheduling problems are represented by **D**. 11, 12, 15, 25, 28, 32, 61, 64, 75–77, 91, 140, 143, 193

Optimisation Polyamorous Scheduling A periodic scheduling problem in which persons are connected by a graph and must schedule pairwise meetings. Each relationship has heat h , which grows daily, and the goal is to minimize the maximum observed heat of any relationship. Instances are represented as $\mathcal{O} = (P, R, g)$, where P is a set of n persons, R is a set of relationships, and g is a set of heat growth rates, one per relationship. Classes of Optimisation Polyamorous Scheduling problems are represented by **O**. 11, 12, 15, 25, 28, 32, 33, 61–63, 66–68, 71, 75–77, 80, 89, 140, 142, 155, 193

Pinwheel Scheduling A periodic scheduling problem in which a single agent schedules k tasks of unit length with constraints on the maximum time between repetitions. Instances are represented as $\mathcal{P} = (f) = (f_1, f_2, \dots, f_k)$, while classes of Pinwheel Scheduling problems are represented by **P**. 7–10, 15, 17–31, 36, 37, 39–45, 47–49, 51–58, 60, 61, 63, 64, 70, 73–75, 77, 78, 89, 91, 92, 140–144, 193

schedule Schedules for Pinwheel Scheduling, Bamboo Garden Trimming, Optimisation Polyamorous Scheduling, and Decision Polyamorous Scheduling are all expressed as mappings: on day i , schedule this task or these tasks. Schedules are represented by \mathcal{S} , with elements s_1, s_2, \dots . Classes of schedules are represented by \mathbf{S} . 7, 9–12, 18, 20–41, 44, 45, 47, 51, 52, 54, 55, 57, 59–71, 73, 74, 76, 77, 79–81, 84, 87–89, 91, 93–95, 97–104, 106–110, 142–144