Online Cardinality Joint Replenishment with Delay

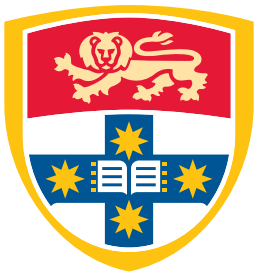
RYDER CHEN
SID: 480344054

Supervisor: Dr. Seeun William Umboh

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Science/Bachelor of Advanced Studies (Honours)

School of Computer Science
The University of Sydney
Australia

12 November 2021



THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Ryder Chen

Signature:  **Date**: November 12, 2021

Abstract

We study the Online Cardinality Joint Replenishment Problem (JRP) with Delays. The JRP is an extensively studied inventory management problem wherein requests for different item types arrive at various points in time. A request is served by ordering its corresponding item type. When a set of requests is served, a cost is paid depending on the item types ordered. Under the deadline model, each request has a deadline and must be served before its deadline. Under the delay model, each request incurs a delay penalty: a non-decreasing, continuous function of the time the request has been left unserved, that must be paid so long as the request is unserved. The objective of any algorithm is to minimise the total service and delay costs.

Cardinality JRP is a variant of JRP where the cost of ordering is a non-decreasing, concave function of the number of item types ordered. We provide the first constant competitive, polynomial time, online algorithm for this problem with delay. Along the way, we develop a new online algorithm for Cardinality JRP with deadlines which acts as a stepping stone to developing the algorithm for the delay case.

Weighted Cardinality JRP is a generalisation of Cardinality JRP where item types have different weights. We give the first constant competitive, online algorithm for Weighted Cardinality JRP with delays by reducing to Cardinality JRP with delays.

Acknowledgements

I would like to thank my supervisor, Dr William Umboh, for his time and guidance over the past year.

CONTENTS

Student Plagiarism: Compliance Statement ii

Abstract iii

Acknowledgements iv

List of Figures vii

Chapter 1 Introduction 1

1.1 Results and Techniques 4

1.2 Further Related Work 6

1.3 Thesis Outline 10

Chapter 2 Overview of Cardinality JRP Algorithms 11

2.1 Problem Definition: Piecewise Cardinality JRP 11

2.2 A reduction to Piecewise Cardinality JRP 12

2.3 Algorithm Challenges 14

2.4 Algorithm Intuition Overview 15

Chapter 3 Piecewise Cardinality JRP with Deadlines Algorithms 18

3.1 Intuition: Rent-or-Buy JRP with Deadlines 18

3.2 Rent-Or-Buy Cardinality JRP with Deadlines Algorithm 22

3.2.1 Algorithm Description 22

3.2.2 Analysis Sketch 24

3.3 Intuition: Piecewise Cardinality JRP with Deadlines 25

3.4 Piecewise Cardinality JRP with Deadlines Algorithm 28

3.4.1 Algorithm Description 28

3.4.2 Analysis Overview 32

3.4.3 Analysis: Solution Structure 33

3.4.4 Analysis: Charging service costs 33

3.4.5 Analysis: Charging to OPT 37

Chapter 4 Piecewise Cardinality JRP with Delay Algorithm 42

4.1 Intuition 42
4.2 Algorithm Description 47
4.3 Analysis Overview 52
4.4 Analysis: Solution Structure 53
4.5 Analysis: Charging service costs 57
4.6 Analysis: Charging to OPT 60
4.7 Time Complexity Considerations 70

Chapter 5 Weighted Cardinality JRP 74

5.1 Algorithm Description 74
5.2 Analysis 75

Chapter 6 Conclusions and Open Problems 83

Bibliography 84

Appendix A Rent-Or-Buy Cardinality JRP with Deadlines Pseudocode 86

Appendix B Rent-Or-Buy Cardinality JRP with Deadlines Analysis 88

B.1 Analysis: Solution Structure 88
B.2 Analysis: Charging tail service costs 89
B.3 Analysis: Charging to OPT 91

List of Figures

1.1 Example instances with possible solutions. The intervals represent requests with the left end point being the arrival time and right end point being the deadline or service time in the case with delay. Red requests are served in Service 1 and Blue requests are served in Service 2. 2

1.2 Overview of the online problems discussed with the best known results. 7

2.1 Example approximation of a concave function $f(\lambda)$ with a piecewise affine function $g(\lambda)$. 12

3.1 Illustration of different triggering requests. The arrows indicate which service each triggering request's corresponding service points to. 19

3.2 Illustration of chains and the different types of services for Rent-or-Buy. Gray boxes linked by arrows represent our service chains. 20

3.3 B/δ intersecting charged requests which cause a buy service to be made. 21

3.4 Illustration of chains and the different types of services. Gray boxes linked by arrows represent our service chains. 26

3.5 Illustration of Lemma 3.2.7's proof. 36

3.6 Illustration of Observation 3.2.10's proof. 38

4.1 Illustration of chains and the different types of services. Gray boxes linked by arrows represent our service chains. 44

4.2 Level l charged investment intervals that cause a level l service to upgrade to level $l+1$. The grey intervals represent requests that are level l at the current time t and the red intervals represent the level l charged investment intervals that begin after time a_q. 45

4.3 Illustration of Lemma 4.1.9 Case 2's proof. 59

4.4 Illustration of Lemma 4.1.17's charging argument. 64

B.1 Illustration of Lemma B.1.6's proof. 90

B.2 Illustration of Observation B.1.8's proof. 92

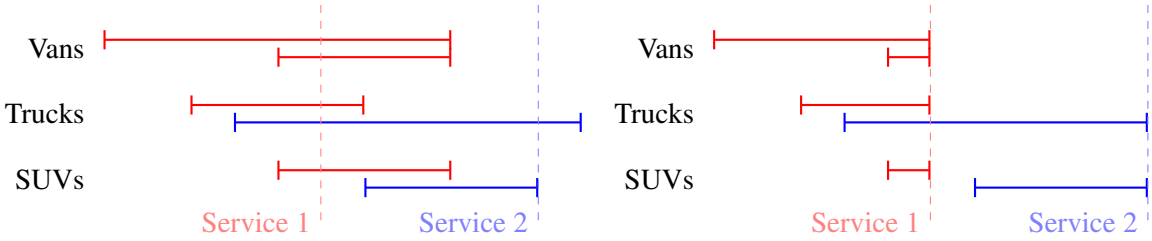
CHAPTER 1

Introduction

In this thesis, we will be considering the Joint Replenishment Problem (JRP): a class of optimisation problems that are fundamental to inventory management theory. The problem involves a sequence of requests for items that arrive at various times. Serving a set of requests or item types incurs a cost that is determined by a given cost function. The goal of the problem is to serve all requests whilst minimising the total cost incurred. There are two variants under which this problem is often studied. In the deadline variant, each request has an associated deadline that is must be served before whilst in the delay variant, each request incurs a delay penalty which must be paid. The delay is a non-decreasing, continuous function of the time the request has been left unserved. Under the delay model, the goal is to minimise the total service and delay costs. We will be considering the JRP under a *make-to-order* mechanism (Dellaert and Melo, 1995), where items must be made to serve some request and cannot be held in inventory. This mechanism models scenarios where items produced are highly customised. We remark that a deadline can be modelled by a delay function that sharply rises from 0 to infinity. Hence, any results developed for the delay model will also apply to the deadline model.

An example application of JRP is in a car manufacturing factory. Suppose there is a factory that produces 3 different cars: vans, trucks and SUVs. To power up the factory and begin producing cars, the factory must pay a start up cost of c_0. To produce vans, the factory must rent highly specialised equipment which costs c_1. After renting the specialised equipment, the extra cost of actually producing vans is negligible. Hence, regardless of how many vans are produced at once, the factory will always pay a fixed cost of c_1. Likewise, the factory requires special equipment for trucks and SUVs, costing c_2 and c_3 respectively. Customers request cars from this factory over time and the factory must decide when to serve each customer's request. Figure 1.1 shows an example sequence of requests arriving and a possible solution. For the instance with deadlines, Service 1 costs $c_0 + c_1 + c_2 + c_3$ whilst Service 2 costs $c_2 + c_3$, giving an overall cost of $c_0 + c_1 + 2c_2 + 2c_3$. We note again that despite Service 1 serving two requests for Vans, it only needed to pay a cost of c_1 for both these requests. For the instance with delay, assume

for simplicity that each request accumulates a delay of 1 by the time it is served, then since Service 1 serves a total of 4 requests, it costs $c_0 + c_1 + c_2 + c_3 + 4$ whilst Service 2 costs $c_0 + c_2 + c_3 + 2$ giving a total cost of $c_0 + c_1 + 2c_2 + 2c_3 + 6$.



(a) JRP with Deadlines (b) JRP with Delay

FIGURE 1.1: Example instances with possible solutions. The intervals represent requests with the left end point being the arrival time and right end point being the deadline or service time in the case with delay. Red requests are served in Service 1 and Blue requests are served in Service 2.

There are two settings under which JRP can be studied. In the classic offline setting, all requests are given beforehand. It has been shown that offline JRP is NP-Hard (Becchetti et al., 2006), hence, algorithms often approximate the optimal solution. The *approximation ratio* of an algorithm is measured as the ratio between its cost and the cost of the optimal solution. In the online setting, requests arrive over time and at any point in time, the algorithm may choose to serve some set of requests. Decisions made in the past are irrevocable and the algorithm has no knowledge of the future. In the online setting, this paper will be using the *clairvoyant* model, where the deadline or delay function of a request is revealed to the algorithm upon its arrival. This is the standard model used by previous works on online JRP and related problems. In an online setting, the performance of any algorithm is measured using its *competitive ratio* which is the ratio between the cost of the algorithm and the cost of the optimal, offline solution that has been given all information beforehand. Oftentimes, online algorithms are only concerned with their competitive ratio and have little regard for their time complexity. This is because the challenge of designing an online algorithm is dealing with the lack of information and only knowing part of the total input and hence the information-theoretic competitive ratio is the main interest.

We will now introduce different classes of JRP that are defined based on the properties of their cost function.

Classical Joint Replenishment Problem. In the classical JRP, each item type i has a corresponding *item ordering cost* K_i and there is a fixed *joint ordering cost* K_0. Whenever a set of items is served,

the cost incurred is K_0 plus K_i for each item type i served. We note that regardless of the number of units of an item type i that gets served, only a fixed K_i is paid. This fact prioritises ordering in bulk and applies to scenarios where creating additional units of a particular item incurs some negligible cost. The best known results in the offline setting with deadlines are a 1.574-approximation algorithm and a 1.207 approximation ratio lower bound (Bienkowski et al., 2013). In the online setting with delay, Buchbinder et al. (2008) have developed a 3-competitive algorithm whilst also proving a lower bound of 2.64 on the competitive ratio of any deterministic algorithm.

Submodular Joint Replenishment Problem. Submodular JRP is a natural extension of classical JRP wherein the cost is a monotonically increasing and submodular function on the item types ordered. Unlike classical JRP where each item type has a fixed, corresponding cost, each possible subset of item types has a corresponding cost. The monotonically increasing property ensures that ordering additional item types always incurs additional costs which is a logical assumption as extra items should not decrease costs. The second property of submodularity captures the property of decreasing marginal costs. That is, each additional item type ordered increases the total cost by a decreasing amount. This embodies the concept of economies of scale that is often observed in the real world and hence encourages ordering in bulk, much like in classical JRP. Mathematically, we define submodularity as follows: given the sets of items to be ordered S and T, our cost function f satisfies the following property: $f(S) + f(T) \geq f(S \cup T) + f(S \cap T)$ or equivalently, define $S \subseteq T$ and let i be any item type then $f(S \cup \{i\}) - f(S) \geq f(T \cup \{i\}) - f(T)$. Unlike classical JRP, additional item types do not increment the cost by a fixed amount, hence, generalising classical JRP. In the offline setting with deadlines, Nagarajan and Shi (2016) present the best known result: a sub-logarithmic $O(\frac{\log T}{\log \log T})$-approximation algorithm where T is the number of time intervals considered. In an online setting with deadlines and delay, Khatkar (2020) has shown that the framework proposed by Azar and Touitou (2020) can be used to produce an $O(\log n)$-competitive algorithm where n is the number of item types.

Cardinality Joint Replenishment Problem. This problem was introduced by Cheung et al. (2016) as a special case of submodular JRP. For this problem, the cost function is a non-decreasing concave function on the number of item types being served, hence the name Cardinality JRP. This problem captures scenarios where different item types may only differ slightly, for example by colour, and hence incur identical costs. Cheung et al. (2016) gave a 5-approximation algorithm for Cardinality JRP with deadlines in an offline setting. Khatkar (2020) studied the problem in an online setting with deadlines and formulated an $O(1)$-competitive algorithm. However, Cardinality JRP has not been studied in an

online setting with delay and hence the best known result is the $O(\log n)$-competitive algorithm from submodular JRP. Seeing as there is a constant competitive algorithm for Cardinality JRP with deadlines, this motivates the search for a constant competitive algorithm for Cardinality JRP with delays which is the primary goal of this thesis.

Weighted Cardinality Joint Replenishment Problem. This problem is a natural generalisation of Cardinality JRP where different item types i have an associated weight w_i. The cost of serving a set of item types λ is a function of the sum of their weights. That is, it is a non-decreasing, concave function of $\sum_{i\in\lambda} w_i$. This problem was introduced by Khatkar (2020) as a generalisation of both Cardinality JRP and Classical JRP and Khatkar (2020) has proposed an $O(1)$-competitive online algorithm for the problem under the deadline model.

1.1 Results and Techniques

The main result of this paper is a nearly optimal $O(1)$-competitive online algorithm for Cardinality JRP with delay. Whilst online algorithms are generally only concerned with their competitive ratios, we will also show that our algorithm can be made to run in polynomial time.

THEOREM 1.1. *There is an $O(1)$-competitive, deterministic, polynomial time online algorithm for Cardinality JRP with Delay.*

Seeing as our algorithm is nearly optimal and runs in polynomial time, it has the potential to be used in industry to handle various inventory management problems. As an example, it could be used to minimise the cost of manufacturing luxury cars of different colours.

Limitations of previous work. Previous work on JRP does not easily generalise to online Cardinality JRP with delay. Hence, this paper first proposes a new $O(1)$-competitive algorithm for online Cardinality JRP with deadlines that can then be generalised to the delay variant. This algorithm is inspired by the work of Khatkar (2020) and Azar and Touitou (2020) and uses the same ideas of approximating the concave cost function using a piecewise affine function then assigning levels to requests to indicate, roughly speaking, which piece of the piecewise function should be used to serve them. This algorithm also utilises the ideas of assigning budgets to services made by the algorithm and upgrading the level of requests when certain conditions are met. The key insight which differentiates our algorithm from the one given by Khatkar (2020) and which enables it to be generalised to the delay case is in the condition

used to upgrade a request's level. Khatkar (2020) bounds the number of requests served at a particular level over an interval of time and updates the level of requests once a certain threshold has been met. However, this generalises poorly to the delay variant since the requests have arbitrary delay functions and hence, over any interval of time, the delay accumulated by requests is arbitrary and unbounded. Therefore, our algorithm explicitly bounds the number of requests that have been served at a particular level at any point in time and utilises this bound to determine when to update levels. By no longer relying on an interval of time, when generalised to the delay case, this algorithm does not encounter issues with an unbounded amount of delay penalty being incurred.

Our techniques to handle delay. To generalise the algorithm to the delay variant, we adapt and extend the idea of investments used by Azar and Touitou (2020). Conceptually, instead of thinking of the delay penalty on requests being paid off continuously as time progresses, online algorithms with delay normally pay off the delay accumulated by requests when the request is served. However, with the notion of investments, services will pay off the delay accumulated by all requests, regardless of whether or not it serves it, and will also pay off and 'invest' in the delay requests may accumulate in the future. This can be thought of as services investing in the delay requests accumulate and might accumulate in the future and incrementally paying it off as opposed to paying it all off in one go at service time. Our key innovation to extend upon the idea of investments used by Azar and Touitou (2020) is to keep track of and utilise how much has been invested into each request. More specifically, we will invest in requests and once a sufficient amount has been invested into a set of requests of the same item type, we will serve the set of requests. To then generalise the level updating condition used in the deadline variant, our algorithm will explicitly bound the amount invested in requests of a particular level and use this bound to determine when we update levels.

Despite using the notion of investments introduced by Azar and Touitou (2020), we remark that their algorithm does not easily generalise to Cardinality JRP with delay. Khatkar (2020) has shown that their algorithm is $\Omega(\sqrt{\log n})$-competitive when directly adapted to Cardinality JRP with deadlines. Since delay generalises deadlines, it follows that their algorithm is also $\Omega(\sqrt{\log n})$-competitive when applied to the delay model.

Our analysis techniques. Our analysis uses similar ideas to Khatkar (2020) and Azar and Touitou (2020) by defining service pointers to define different types of services and then breaking up our algorithm's cost into the costs of different types of services. However, our analysis substantially deviates

from the work done by Khatkar (2020) due to the difficulties of requests being able to accumulate an arbitrary amount of delay over some time interval. Using our extended notion of investments, we introduce a novel charging argument that is unprecedented in literature. Previous works on online JRP and related problems often charge their algorithm's delay costs to the costs of its services which are then charged to the optimal solution. Our analysis differs by instead charging the algorithm's service costs to the delay costs we have invested in and then charging these costs to the optimal solution.

Extensions of our work. We then extend our algorithm for Cardinality JRP with Delay to give an $O(1)$-competitive online algorithm for Weighted Cardinality JRP with Delay:

THEOREM 1.2. *There exists an $O(1)$-competitive, deterministic algorithm for Weighted Cardinality JRP with Delay.*

This will be shown through a reduction to Cardinality JRP with Delay. Khatkar (2020) also used a reduction to formulate an $O(1)$-competitive algorithm for Weighted Cardinality JRP with Deadlines. However, their reduction involves first reducing to an intermediate problem that is then reduced to Cardinality JRP. In contrast, we will reduce our problem directly to Cardinality JRP. In our opinion, this reduction is more straightforward and leads to a much simpler analysis.

1.2 Further Related Work

We now explore previous works on JRP and related problems and the algorithms and results proposed. For the reader tight on time, the results and ideas proposed by Khatkar (2020) for Online Cardinality JRP and by Azar and Touitou (2020) for Online Network Design are most relevant to this thesis. Figure 1.2 gives a hierarchy of the problems discussed as well as their best known online results.

Classical JRP. Classical JRP has been studied extensively both offline and online. In an offline setting with deadlines, Becchetti et al. (2006) have shown the problem is NP-Hard which was furthered by Nonner and Souza (2009) who showed it is APX-hard. Nonner and Souza (2009) also developed a 5/3-approximation algorithm using a Linear Programming approach and randomisation to convert the result into an Integer Program solution. Using the same ideas, Bienkowski et al. (2013) developed a 1.574-approximation algorithm as well as proving the first known approximation ratio lower bound of 1.207. As of now, these results are the best known upper and lower bounds for classical JRP with deadlines in the offline setting. In the online setting with delays, Buchbinder et al. (2008) have used the ideas of

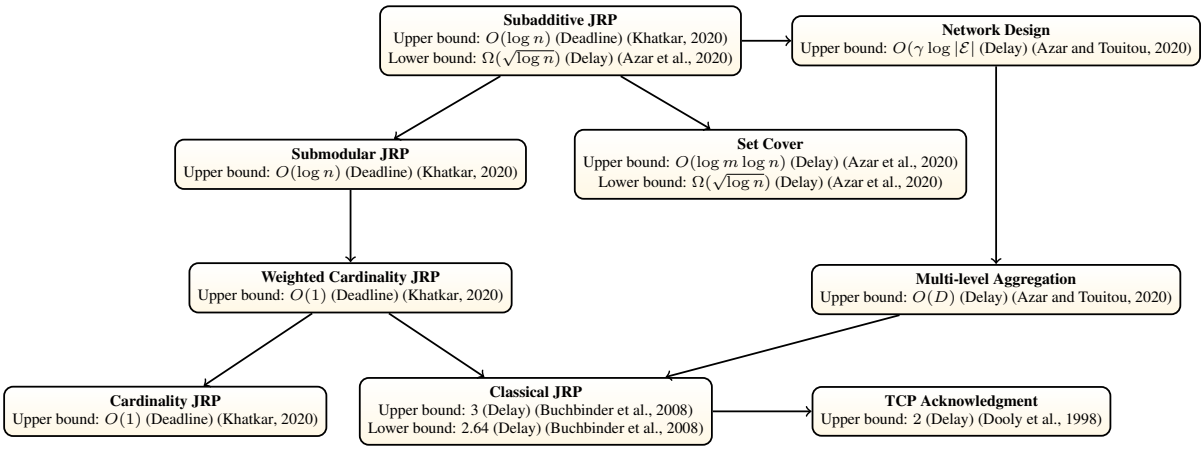


FIGURE 1.2: Overview of the online problems discussed with the best known results.

Linear Programming utilised in the offline case to develop a 3-competitive algorithm. However, their work deviates from the offline case by introducing a simulation phase wherein the delay accumulated in the future is simulated to simultaneously construct a feasible primal and dual solution to the Linear Program. Buchbinder et al. (2008) have also shown that any online algorithm for classical JRP with delays has a competitive ratio lower bounded by 2.64. The results developed for the online variant with delay easily generalises to the online variant with deadlines since the delay model is a generalisation of the deadline model. We also remark that Classical JRP is a generalisation of the well-known TCP Acknowledgment problem. This is because TCP Acknowledgment is simply JRP with only one item type. Dooly et al. (1998) have given a 2-competitive online algorithm for this problem with delay.

Offline Submodular and Cardinality JRP. Cheung (2014) first introduced submodular JRP as an extension of classical JRP that more accurately captured real world scenarios. Cheung (2014) propose an $O(\sqrt{T}\log T)$-approximation algorithm for offline submodular JRP with deadlines (where T is the number of time periods considered) by approximating the submodular function and making calls to an oracle. Cheung et al. (2016) then developed an $O(\log NT)$-approximation algorithm (where N is the number of requests) by using a reduction from submodular JRP to the set cover problem then applying existing greedy approximation algorithms for set cover. Nagarajan and Shi (2016) improve upon this result by using Linear Programming to develop a sub-logarithmic $O(\frac{\log T}{\log\log T})$-approximation algorithm. Cheung et al. (2016) also introduced Cardinality JRP as a special case of submodular JRP and studied the problem in the offline setting and produced a near optimal constant approximation ratio using Linear Programming. To convert the Linear Program solution into a solution for Cardinality JRP, the authors partitioned requests on each item type into time intervals according to the Linear Program solution

and then grouped the various intervals across item types to determine how services will be made. We remark that our algorithm does not utilise a Linear Program approach since the formulation of the Online Cardinality JRP with Delay problem as a Linear Program is too complicated.

Online Cardinality JRP. There has been little work on other variants of JRP in the online setting. The only known results are from Khatkar (2020) who developed a constant competitive online algorithm for Cardinality JRP with deadlines. This algorithm uses the fact that any concave monotone function can be approximated using a piecewise affine function to define a new problem: Piecewise Cardinality JRP. A constant competitive algorithm for Piecewise Cardinality JRP with deadlines was then developed using the notion of levels inspired by Azar and Touitou (2020) to determine which affine piece is used to serve each request. Khatkar (2020) then extend their work to propose an $O(1)$-competitive online algorithm for **Weighted Cardinality JRP** with deadlines. This was done by reducing the problem to a special instance of Weighted Cardinality JRP where all weights are either 0 or 1, then solving the instance using their algorithm for online Cardinality JRP.

Online Submodular and Subadditive JRP. Khatkar (2020) also shows that the algorithm produced by Azar and Touitou (2020) can be easily generalised to online subadditive JRP, a generalisation of submodular JRP where the cost function is subadditive. This shows the existence of an $O(\log n)$-competitive algorithm for online subadditive JRP with deadlines where n is the total number of item types. By extension, this algorithm then also applies to online submodular JRP with deadlines since subadditivity is a generalisation of submodularity. We also remark that Set Cover with Delay is a special case of subadditive JRP with Delay since for every subset S of elements in set cover, we can define $f(S)$ as the minimum cost of covering S. Then f will necessarily be subadditive and hence the problem can be represented as an instance of subadditive JRP. Therefore, the competitive ratio lower bound of $\Omega(\sqrt{\log n})$ on any online algorithm for set cover with delay proposed by Azar et al. (2020) also applies to subadditive JRP.

Multi-level Aggregation. A problem closely related to JRP is the Multi-level Aggregation (MLA) problem. The MLA problem involves a weighted, rooted tree and requests that arrive at nodes in the tree which can be served by sending it to the root. Upon sending a request to the root, it incurs a cost equal to the weights of all edges on the path from the request node to the root. Requests on multiple nodes can be aggregated and served together and in this manner, the service cost is the weight of the subtree spanning all nodes and including the root. MLA is a generalisation of classical JRP as we can

represent the JRP as a tree with a root connected to a dummy node d with weight equal to the joint ordering cost K_0 as well as leaf nodes for each item type i with a weighted edge to d whose weight corresponds to the item ordering cost K_i. Bienkowski et al. (2016) were the first to look at online MLA with a tree of arbitrary depth D and proposed an $O(D^2 2^D)$-competitive algorithm for online MLA with deadlines as well as an $O(D^4 2^D)$-competitive algorithm for online MLA with delay. Both algorithms reduce the problems to the case where the tree is L-decreasing, meaning that along any path from root to leaf, every subsequent edge has its weight decreasing by a factor of at least L. In this case, requests are served by order of urgency, with urgency being defined by deadlines in the deadline variant and maturity in the delay variant where maturity is measured as the time when a node has accumulated delay equal to its service cost. Improving upon this work, Buchbinder et al. (2017) proposed an $O(D)$-competitive algorithm for online MLA with deadlines by using a reduction to 3-decreasing trees and then recursively assigning each node a budget which it can use to serve requests in its subtree.

Network Design. Network design problems are those that involve a graph, a set of elements (for example the edges or the nodes) and a cost associated with each element. *Connectivity requests* arrive which request for a network with certain properties that must then be constructed by selecting elements and paying the cost for the elements selected. The goal is to satisfy all connectivity requests whilst paying minimal costs. This class of problems generalises many graph and tree based problems such as the Steiner tree problem, MLA and JRP. It can also be shown that the cost of ordering two networks together is cheaper than ordering the networks separately and hence the cost function in network design is subadditive, making it a special case of subadditive JRP. Azar and Touitou (2020) have proposed a deterministic framework for online network design problems under both the deadline and delay models. They show that if there exists a γ-approximation algorithm for an offline network design problem with deadlines then there exists an $O(\gamma \log |\mathcal{E}|)$-competitive algorithm in the online variant where $\mathcal{E}$ is the set of elements. Their paper also shows that if there exists a γ-approximation for the prize collecting variant of a network design problem then there exists an $O(\gamma \log |\mathcal{E}|)$-competitive algorithm in the online variant with delays. Their algorithm utilises the notion of levels with each request being assigned a level and a service of level l can only serve requests of level at most l. Moreover, their analysis uses the idea of pointers to relate different services and construct different types of services which can be analysed separately. These ideas heavily motivate the work done by Khatkar (2020). The results of this framework can be applied to produce an $O(\log n)$-competitive algorithm for online Steiner trees and since Steiner trees generalise MLA, this matches the $O(D)$-competitive algorithm produced for online MLA by Buchbinder et al. (2017).

1.3 Thesis Outline

Chapter 2 will give a brief overview of the ideas that will be used to produce our algorithms for Cardinality JRP. This chapter will formally introduce Piecewise Cardinality JRP and the challenges our algorithms must address. Chapter 3 will then provide our algorithms and analysis for Cardinality JRP with Deadlines. These ideas will then be generalised to the delay variant in Chapter 4. A reader short on time may skip Chapter 3 since the algorithm and analysis proposed for the delay model in Chapter 4 generalises and hence applies to the deadline model as well. However, this will result in a loss of intuition for the algorithm and analysis since much of the work and ideas behind the algorithm are adopted from the deadline case. Chapter 5 will then extend our work to produce an algorithm for Weighted Cardinality JRP with Delay. Finally, Chapter 6 will conclude the thesis and discuss areas for further work.

CHAPTER 2

Overview of Cardinality JRP Algorithms

2.1 Problem Definition: Piecewise Cardinality JRP

Formally, Online Piecewise Cardinality JRP is a sub-problem of Cardinality JRP where the cost function is a concave, piecewise function defined by taking the minimum of n affine functions, where n is arbitrary. More precisely, the cost of a service λ is $g(\lambda) = \min_i\{\sigma_i + \delta_i|\lambda| : i \in [1, n]\}$ where the cardinality $|\lambda|$ is the number of item types served in λ. The affine functions must also satisfy $2\sigma_i \leq \sigma_{i+1}$ and $\delta_i \geq 2\delta_{i+1}$ for all $i \in [1, n)$. We will also require that $\sigma_i \geq \delta_i$ for all i otherwise this problem would not be interesting as the σ_i costs would become irrelevant. Requests will then arrive over time and upon arrival, they will have an associated deadline or delay function that is revealed.

For the deadline variant, requests must be served before their deadlines for a solution to be feasible. We can assume without loss of generality that all deadlines are distinct since deadlines can be perturbed by some random, small ϵ to force uniqueness. The goal under this model is to serve all requests before their deadlines whilst minimising service costs.

For the delay variant, the delay penalty function $d_q(t)$ for a request q is a non-decreasing, continuous function of the time the request has been left unserved. We will also assume that the delay penalty for each request tends to infinity as the time tends to infinity which is a natural assumption also made by Azar and Touitou (2020) to ensure that all requests must eventually be served.

When a service is made paying $\sigma_i + \delta_i|\lambda|$ for some i, we will say that a *level i service* has been made and call the σ_i paid the *shared cost* of the service and the $\delta_i|\lambda|$ cost paid the *individual cost* of the service. Typically, a solution to Piecewise Cardinality JRP would specify when services are made and what requests are served by these services. The cost of this service would then be determined by taking the minimum of the piecewise affine functions. However, an equivalent and more useful way of thinking about the solution to this problem is that for each service, we specify not only the requests it serves but

also the level it is serving at. If we think of a solution this way, the cost of our solution is at least as much as the true cost of the solution if we did not specify a service level for each of our services. Hence, any results upper bounding the cost of our solution will also upper bound the true cost of the solution, had it not specified levels for each service. Thinking of our solution in this way is beneficial as we can easily determine the cost of each of our services without needing to take the minimum of multiple affine pieces which may further complicate our analysis.

2.2 A reduction to Piecewise Cardinality JRP

Similarly to the work done by Khatkar (2020), we will produce our constant competitive algorithm for Cardinality JRP with delay by reducing to Piecewise Cardinality JRP with delay. This is because the concave, monotonically increasing cost function $f(\lambda)$ used in Cardinality JRP can be approximated using a piecewise affine function with an arbitrary number of affine pieces as done by Gupta et al. (2017) and Khatkar (2020). More precisely, we can approximate $f(\lambda)$ using $g(\lambda) = \min_i\{\sigma_i + \delta_i|\lambda| : i \in [1, n]\}$ where $|\lambda|$ is the cardinality of the service λ, or in other words, the number of item types served in λ. Figure 2.1 shows an example of such a piecewise approximation. Whilst not apparent in the figure, we will also require that $f(0) = g(0) = 0$ since the cost of serving no items should trivially be 0.

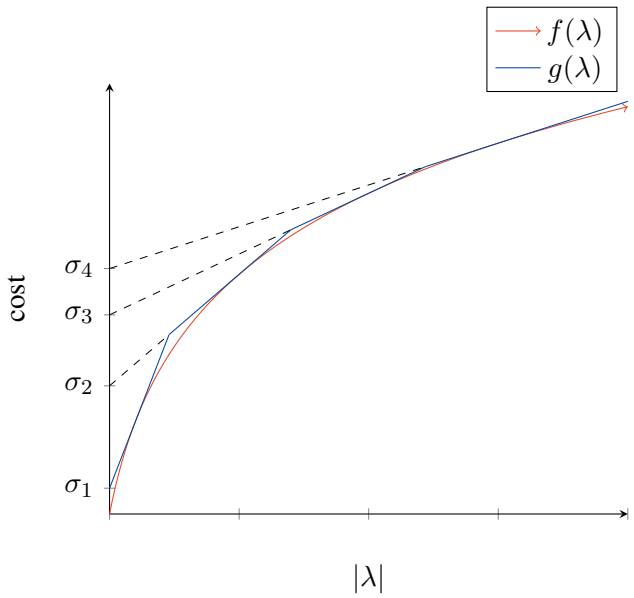


FIGURE 2.1: Example approximation of a concave function $f(\lambda)$ with a piecewise affine function $g(\lambda)$.

Furthermore, we will assume that the Piecewise Cardinality JRP cost function satisfies $2\sigma_i \leq \sigma_{i+1}$ and $\delta_i \geq 2\delta_{i+1}$ for each $i \in [1, n)$. Under such an assumption, Guha et al. (2001) have shown that our piecewise approximation will only lose a constant factor. That is, $f(\lambda) \leq g(\lambda) \leq O(1) \cdot f(\lambda)$. Consequently, when we reduce from Cardinality JRP to Piecewise Cardinality JRP, the cost of our solution to Piecewise Cardinality JRP is only increased by a constant factor when used for Cardinality JRP. Hence, any constant competitive algorithm from Piecewise Cardinality JRP will remain constant competitive after the reduction.

LEMMA 2.1.1. *Suppose we have an $O(1)$-competitive algorithm for Piecewise Cardinality JRP with Delay. Then there is an $O(1)$-competitive, deterministic online algorithm for Cardinality JRP with Delay.*

PROOF. Given an instance of Cardinality JRP with delay with cost function $f(\lambda)$, we will construct an instance of Piecewise Cardinality JRP where the requests and their delay penalties are exactly the same but the cost function is the piecewise affine function $g(\lambda) = \min_i\{\sigma_i + \delta_i|\lambda| : i \in [1, n]\}$ where n is the total number of item types. $g(\lambda)$ is constructed by using the tangents to $f(\lambda)$ at each integral value in $[1, n]$. The affine pieces of g will also satisfy $2\sigma_i \leq \sigma_{i+1}$ and $\delta_i \geq 2\delta_{i+1}$ for each $i \in [1, n)$. This can be achieved by iterating over the affine pieces $\sigma_i + \delta_i|\lambda|$ in ascending order of i and adjusting the σ_i and δ_i values. For this instance of Piecewise Cardinality JRP, our constant competitive algorithm will produce the solution ALG_g. We will then use this solution as our solution to the original, Cardinality JRP instance.

Let OPT_f denote the optimal solution for the original instance of Cardinality JRP. Let OPT_g denote the optimal solution for the instance of Piecewise Cardinality JRP. $f(S)$ and $g(S)$ will refer to the service costs of the solution S under the cost functions f and g respectively. $D(S)$ will refer to the delay costs of the solution S. We note that for a given solution S, $D(S)$ is always the same regardless of which cost function f or g is used since the cost function does not affect the service times of the solution and hence the delay paid for the solution.

We will now show that our algorithm for Piecewise Cardinality JRP with delay produces an $O(1)$-comeptitive solution for Cardinality JRP with delay. That is,

$$f(\mathrm{ALG}_g) + D(\mathrm{ALG}_g) \leq O(1) \cdot (f(\mathrm{OPT}_f) + D(\mathrm{OPT}_f))$$

Guha et al. (2001) have shown that our piecewise affine approximation of f satisfies $f(x) \leq g(x) \leq O(1) \cdot f(x)$. Hence,

$$
\begin{aligned}
& f(\mathrm{ALG}_g) + D(\mathrm{ALG}_g) \\
& \leq g(\mathrm{ALG}_g) + D(\mathrm{ALG}_g) && (f(x) \leq g(x)) \\
& \leq O(1) \cdot (g(\mathrm{OPT}_g) + D(\mathrm{OPT}_g)) && (\mathrm{ALG}_g \text{ is constant competitive by assumption}) \\
& \leq O(1) \cdot (g(\mathrm{OPT}_f) + D(\mathrm{OPT}_f)) && (\text{Optimality of } \mathrm{OPT}_g \text{ under } g(x)) \\
& \leq O(1) \cdot (O(1) \cdot f(\mathrm{OPT}_f) + D(\mathrm{OPT}_f)) && (g(x) \leq O(1) \cdot f(x)) \\
& \leq O(1) \cdot (f(\mathrm{OPT}_f) + D(\mathrm{OPT}_f))
\end{aligned}
$$

□

Since this proof relies on having a constant competitive algorithm for Piecewise Cardinality JRP with delay, the primary goal of this thesis will be to produce such an algorithm. Chapter 4 will show that we do indeed have a constant competitive algorithm for Piecewise Cardinliaty JRP with Delay. Section 4.7 will then show that this reduction and our algorithm for Piecewise Cardinliaty JRP with Delay take polynomial time. From these results and Lemma 2.1.1, our main theorem, Theorem 1.1, easily follows.

2.3 Algorithm Challenges

We will now discuss the challenges that must be addressed in designing an algorithm for Piecewise Cardinality JRP.

Analogous to the challenges faced in Classical JRP, any algorithm must decide when to make a service but making too many services may accumulate too high of a shared cost and is undesirable. Whenever a service is made, since the fixed shared cost of σ_i must always be paid, it is desirable to serve as many requests together as possible. However, serving multiple requests of the same item type together means that fewer individual costs are incurred since each time an item type is served, a cost of δ_i is paid regardless of the number of requests served. Therefore, it may be desirable to not serve a certain request and instead wait for more requests on the same item type to arrive before serving them together. Hence, any algorithm must balance both serving as many requests at once as possible as well as leaving some requests unserved to wait for future requests on the same item type.

The additional challenge in Piecewise Cardinality JRP is deciding which level each service serves at. Since $2\sigma_i \leq \sigma_{i+1}$ and $\delta_i \geq 2\delta_{i+1}$ for all $i \in [1, n)$, a lower level service pays a lower shared cost but higher individual cost for each item type and is hence preferred for services with fewer items whilst a higher level service must pay a larger shared cost but can then serve items at a lower individual cost and should thus be used when serving more item types.

Ultimately, the three core problems that must be addressed to design an algorithm are when should we make a service, what level service should we make and how many and which requests should be served in the service.

2.4 Algorithm Intuition Overview

We will now give a brief and high level overview of the main ideas used to design our algorithms. This will provide an intuition for how our algorithms are developed and interrelated. A more detailed explanation of the intuition behind our algorithms will be provided later.

Since $2\sigma_i \leq \sigma_{i+1}$ and $2\delta_{i+1} \leq \delta_i$, the σ_i and δ_i values in our piecewise affine functions form a geometric series. This geometric property is the key to designing a constant competitive algorithm. If we can bound the amount spent at each level l and at any point in time by σ_l and δ_l, then by summing up the amount spent at each level l and using the geometric sum induced, we can bound the total amount spent at any time by a constant multiple of some σ_{l*} and δ_{l^*}. This total amount spent can then be charged to a constant times what the optimal solution would spend at this time. Hence, our algorithms are designed with this geometric property in mind and aim to bound the amount spent at any level l and at any point in time by σ_l and δ_l in order to make use of the geometric sums induced.

As mentioned earlier, to design an algorithm for Piecewise Cardinality JRP with delay, we will design and generalise an algorithm for Piecewise Cardinality JRP with deadlines. As a warm up, we will first look at Rent-or-Buy Cardinality JRP with deadlines, a simplified version of the problem where the cost function consists of only two affine pieces: $g(\lambda) = \min\{\sigma + \delta|\lambda|, B\}$, where $B \geq 2\sigma$.

Rent-or-Buy Cardinality JRP with Deadlines Intuition. The idea is we will make a service whenever a request reaches its deadline. This is because we must serve all requests before their deadlines and waiting until a deadline gives ample time for extra requests of the same item type to arrive. To decide which level to serve at, we will start with level 1 services (services costing $\sigma + \delta|\lambda|$) by default and

wait until there is a point in time where the amount spent by level 1 services is enough to make a level 2 service (a service costing B). This way, the level 2 service costs will always be at most the level 1 service costs and hence can be charged to the level 1 services. This will also explicitly bound the number of requests that can be served at level 1 at any point in time. Whenever a level 2 service is made, we may as well serve all available requests since doing so will not incur any additional costs as the maximum possible cost of B has already been paid. For level 1 services, there is a need to balance serving as many requests as possible to capitalise on the shared cost σ that is paid and the need to wait longer for more requests of the same item type to arrive. To balance these elements, we will use a *service budget*. The budget will force the service to serve up to σ/δ requests by ascending order of request deadlines. This enables us to both serve as many requests as possible whilst also leaving some requests unserved in hopes that more requests of the same item type arrive. Moreover, the requests left unserved will be those with later deadlines as these requests will then have more time to wait for other requests of the same item type to arrive.

Piecewise Cardinality JRP with Deadlines Intuition. The ideas utilised in Rent-or-Buy will then be directly generalised to develop our algorithm for the piecewise case with an arbitrary number of affine pieces. Our condition for triggering services will be the same. To account for having more levels, we will introduce the concept of *request levels*. Each request will have a level that determines which level services it can be served in. In the rent-or-buy case, we explicitly bounded the number of requests that can be served at level 1 at any time and used this to determine when level 2 services should be made instead of level 1 services. We will generalise this condition and use it to bound the number of requests that can be served by level l services at any time and we will use it to determine when level $l+1$ services should be made instead of level l services. This will then be the means by which our service and request levels upgrade. Ultimately, this will let us determine which level to serve on and hence address one of the challenges of this problem. To determine which requests and how many requests to serve, we will again use the idea of service budgets which will now depend on the service level to balance serving as many requests as possible while also leaving some requests unserved to wait for extra requests on the same item type.

Piecewise Cardinality JRP with Delay Intuition. Finally, our ideas from the deadline variant of Piecewise Cardinality JRP will be adapted to form our algorithm for the delay variant. The main problem is, without deadlines, we no longer have a natural sense of urgency and a time frame within which to serve our requests. Hence, our condition to trigger a service will be when the delay accumulated on

requests is comparable to the cost of the service that will be made. This is an idea commonly adopted for designing online algorithms with delay. To determine the level we serve at, we will generalise our condition used in the deadline variant. In the deadline case, we bounded the number of requests that can be served at level l at any time which consequently bounds the total cost that can be incurred by requests at any time. Since our algorithm will utilise the idea of investments, the analogue to this condition in the delay case is to bound the amount that can be invested into requests at level l from any time. As will be explained later, our algorithm will be designed in a way such that bounding the investments on a request bounds both the delay and service costs of the request and hence our condition lets us bound the total cost that can be incurred by a request from any time. We will use this invariant to determine which level to serve at. Lastly, to decide which requests and how many requests to serve, we again use the idea of a service budget. Our services will continually invest in the delay accumulated by requests until the total amount invested reaches a certain budget. A request will then be served when the amount invested into it is comparable to the cost of serving it. Once again, by setting a budget, we will be attempting to serve as many requests as possible but by limiting requests to only be served once an investment threshold is reached, we are giving requests sufficient time to wait for other requests of the same item type to arrive.

CHAPTER 3

Piecewise Cardinality JRP with Deadlines Algorithms

3.1 Intuition: Rent-or-Buy JRP with Deadlines

To build towards an algorithm for piecewise cardinality JRP and build an intuition towards how our algorithm challenges can be handled, we will first look at Rent-or-Buy Cardinality JRP.

Rent-Or-Buy Cardinality JRP is a specific variant of piecewise cardinality JRP where the cost function only consists of two affine pieces: $g(\lambda) = \min\{\sigma + \delta|\lambda|, B\}$ where $B \geq 2\sigma$. For this function to be interesting, we have that $\delta < \sigma$ which means each additional request served in a service is marginally cheaper and hence our algorithm should prioritise serving in bulk as opposed to making singleton services. Services made using the $\sigma + \delta|\lambda|$ piece will be referred to as rent services and services made by paying B will be called buy services. We will say a request is *active* at time t if it has arrived by time t but has not been served yet. The *time interval* of a request is then the time between its arrival and deadline.

When do we make a service? Our algorithm will make a service when any active request reaches its deadline. The request which reaches its deadline and triggers a service will be called the *triggering request*. Such a condition will ensure that all requests get served since whenever any request reaches its deadline it will trigger a service that will serve it. By triggering services on deadlines, we are waiting as long as possible before serving requests which gives time for requests of the same item type to arrive. We remark that since request deadlines are distinct, only one service should be triggered at any point in time.

Service chains. Our algorithm will introduce a notion of *service chains* which are necessary for our analysis. A chain links together different services to form a linear structure and allows us to classify services into different categories based on their position in the chain. Our analysis will then rely on

bounding the costs of the different service categories individually. When a triggering request triggers a service λ, if its time interval does not intersect with any previous triggering request's time interval then we will begin a new chain and this first service of the chain will be the *primary service*. If the interval does intersect with some other triggering request's time interval then we will consider the intersecting triggering request that most recently triggered a service, λ'. We will say that our new service λ has a *service pointer* to λ' and that λ is the subsequent service in the chain that λ' belongs to. Figure 3.1 shows an example of how triggering requests determine where their service points.

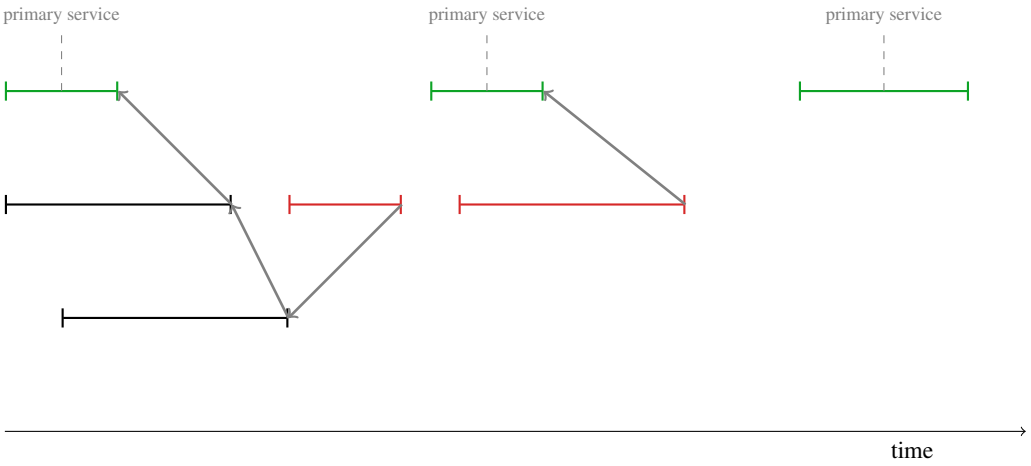


FIGURE 3.1: Illustration of different triggering requests. The arrows indicate which service each triggering request's corresponding service points to.

We will later show that such a definition creates a well-defined linear structure. The last service of a non-singleton chain will be called a *tail service*. That is, if a chain only consists of a single primary service then it will not have a tail service. All non-primary and non-tail services will be called the *normal services*. Figure 3.2 shows an illustration of service chains and the different types of services. We note that the types of services in this figure correspond to the services triggered by the triggering requests in Figure 3.1.

From the definition of chains, it is apparent that the triggering requests of primary services are all disjoint. This is an important fact since for any set of disjoint request time intervals, we are guaranteed that the optimal solution OPT has to make at least one service within each disjoint time interval otherwise it would not be a feasible solution. This idea then enables us to easily charge the cost of the primary services to OPT.

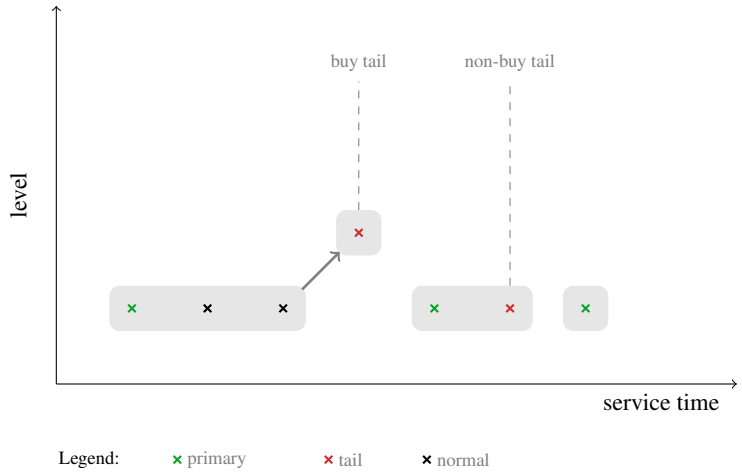


FIGURE 3.2: Illustration of chains and the different types of services for Rent-or-Buy. Gray boxes linked by arrows represent our service chains.

What level service should we make? In the rent-or-buy case there are only two levels, the *rent* service level made by paying $\sigma + \delta|\lambda|$ and the *buy* service level made by paying B. To determine which level to use, we will adopt an idea commonly used in online rent-or-buy problems which is to make a buy service only when a similar amount has been spent on rent services. The cost of the buy services can then be charged to the cost of the rent services. Khatkar (2020) applies this idea directly by having phases within which requests are continually served by rent services until a total of B has been spent by rent services. At this point, a buy service is made which ends the phase and whose cost can be charged to the rent services of the phase. This results in an algorithm that bounds the number of requests served by rent services over a period of time. However, as mentioned earlier, this generalises poorly to the delay model. We observe that this algorithm implicitly bounds the number of requests served by rent services at any point in time. Hence, a natural alternative would be to explicitly bound the number of requests served by rent services at any point in time instead. More precisely, at any point in time t, the number of requests that have been served by rent services and whose time intervals contain t should be bounded. However, such an invariant was still found to be too restrictive.

Ultimately, we refined this idea to produce the following invariant which our algorithm will maintain: at any time t, there are at most $O(1) \cdot B/\delta$ *charged requests* whose time intervals contain t. When a rent service serves multiple requests of the same item type, it only needs to pay the individual cost of δ once for the item type and hence only one of the requests served should be seen as 'contributing' to the service's individual cost. Amongst the requests of the same item type, the choice of which request contributed to the individual cost of the service is arbitrary. Therefore, a charged request will be defined

as a request that is served by a normal, rent service and which contributed to the service's individual cost. To enforce our invariant, when a service is triggered, it will be a buy service if there exists an active request within whose time interval there is a time at which at least B/δ charged requests' time intervals intersect. Figure 3.3 illustrates this condition.

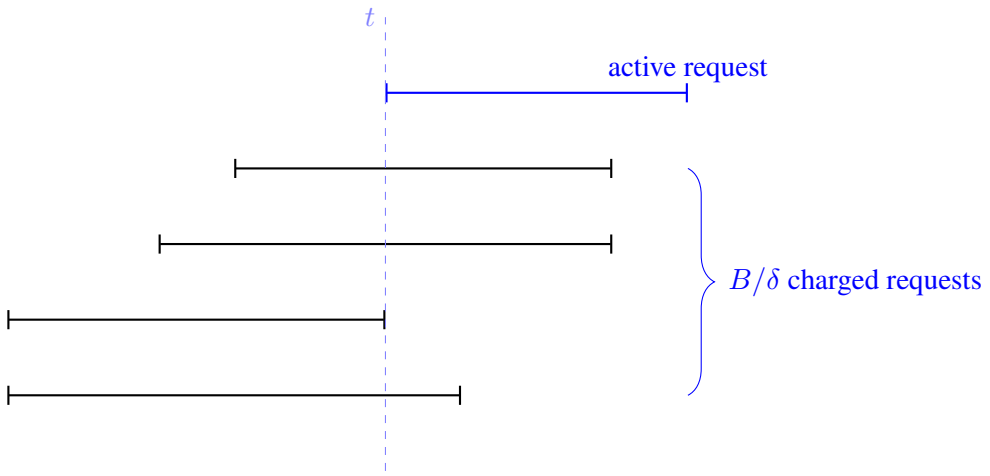


FIGURE 3.3: B/δ intersecting charged requests which cause a buy service to be made.

Later we will show that this condition does in fact preserve the invariant by showing that if we have more than $O(1) \cdot B/\delta$ charged requests intersecting at any point in time, one of these charged requests should have triggered a buy service. As a consequence of this condition, the cost of our buy services can be charged to the cost of the normal, rent services which is analogous the charging argument originally utilised by Khatkar (2020).

How many and which requests do we serve? Whenever a buy service is made, we may as well serve all available requests since doing so will not incur any additional costs as the maximum possible cost has already been paid. For rent services, we have to be more careful with deciding which requests to serve. As discussed earlier, there is a need to balance serving as many requests as possible to capitalise on the shared cost σ that is paid and the need to wait longer for more requests of the same item type to arrive. To balance these elements, we will use the concept of a *service budget*. That is, we will allow our service to serve as many requests as possible until its total individual cost exceeds σ. The choice of σ as our service budget is motivated by the idea that if a service exceeds its budget then we are guaranteed that its total individual cost exceeds its shared cost of σ. Consequently, it is possible to bound the shared cost of the service by its total individual cost. This is useful since we can then charge the cost of this

service to OPT by simply showing that for each request that contributed an individual cost of δ to this service, OPT must also pay a cost similar to δ when serving this request.

The next consideration is which requests should be served. A common strategy used for online algorithms under a deadline model which we will also adopt is to serve active requests in ascending order of deadlines since these deadlines reflect the urgency of the requests. Whenever any request on a particular item type is served, all requests of the same item type may as well be added since they can be added for free. This serving strategy enables us to both serve as many requests as possible whilst also leaving some requests unserved in hopes that more requests of the same item type arrive, thus addressing one of the core algorithm challenges. Moreover, serving by ascending deadline order is logical since the requests left unserved will be those with later deadlines which will then have more time to wait for other requests of the same item type to arrive.

3.2 Rent-Or-Buy Cardinality JRP with Deadlines Algorithm

3.2.1 Algorithm Description

We will design an algorithm that satisfies the following

THEOREM 3.1. *There exists an $O(1)$-competitive, deterministic algorithm for Rent-or-Buy Cardinality JRP with Deadlines.*

Triggering a service. Recall that a request is active at time t if it has arrived but has not been served yet and the time interval of a request is the time between its arrival and deadline. When an active request q reaches its deadline, it will trigger a service λ and be referred to as a triggering request. Let T_λ be the set of triggering requests whose time intervals intersect with the time interval of q. From the requests in T_λ, let q' be the triggering request with the latest deadline. This is necessarily the most recent triggering request in T_λ to have triggered a service. Let the service triggered by q' be λ'. Our newly triggered service λ will have a pointer to λ' which will determine which chain it belongs to. If T_λ is an empty set then λ will be a primary service and begin a new chain.

Determining the service level. When a service λ is triggered at time t, we can either make a rent or a buy service. If our service is primary then we will make a rent service. To determine whether a non-primary service is upgraded to a buy service, we will look at all active requests A_t at this time t. For

each request $q \in A_t$, we look at its time interval $[a_q, d_q]$. If there is a time $t' \in [a_q, d_q]$ at which the time intervals of at least B/δ charged requests intersect, then we will make a buy service. More precisely, we require there to be at least B/δ charged requests q_c with time intervals $[a_{q_c}, d_{q_c}]$ where $a_{q_c} \leq t' \leq d_{q_c}$ for some fixed time $t' \in [a_q, d_q]$. This request q whose time interval contains enough charged requests to trigger the buy will be called the *buy triggering request*.

Serving and charging requests. If the service is a buy service then we will simply serve all active requests. A rent service will involve continually adding active requests in order of ascending deadline to the service λ. When a request for an item type that has not been added to the service yet is added, if the service is non-primary then the request gets *charged*. When a request is added, all other active requests of the same item type are also added. We continue adding requests until either all active requests are served or the total individual cost exceeds σ, that is, $\delta|\lambda| \geq \sigma$. The cost of a service will be the total amount it spent to serve requests, which is B in the case of a buy service or the shared cost σ plus the total individual cost $\delta|\lambda|$ in the case of a rent service.

Uncharging requests. If a service ends up serving all requests then all charged requests are uncharged since it becomes a tail service. Whenever we make a primary service, if there was a previously active chain, meaning there was a chain that had not yet served all active requests, this primary service will end the previous chain. This is because any triggering requests q that may continue an old chain must intersect with the old chain's triggering request which must be before this new chain's triggering request. Therefore, we have that q intersects with both the triggering requests of the old and new chain and since this new chain's service is more recent it will by definition take priority in deciding the chain of q. Therefore, there is no way for an old chain to be continued once a new chain has started and thus whenever a primary service is made, all previous chains must be ended. When a primary service ends a previously active chain, the last service of the chain will by definition become a tail service and all requests charged by that service must be uncharged. This will allow us to maintain the fact that only normal services charge requests.

Since the pseudocode for this algorithm is almost identical to the pseudocode for Piecewise Cardinality JRP with Deadlines that will be provided later in Section 3.4, it will be deferred to Appendix A for readers who are interested.

3.2.2 Analysis Sketch

Since the proof for Piecewise Cardinality JRP with Deadlines will be a generalised and almost identical version of this proof, we will simply give a sketch for how to show this algorithm is constant competitive. For interested readers, the full proof is available in Appendix B.

The different types of services ALG can make are primary services, buy services (which are necessarily tail services), non-buy tail services and normal services. At a high level, we will bound the costs of these different types of services individually then sum up all the different categories to get the desired result.

This involves first observing that every request is served before its deadline and hence our solution is feasible. We can then argue that our chains are linear structures and well defined since every service can only point to and be pointed to by one other service. The last key property of our solution that we observe is that any rent service costs at most 3σ since it pays the shared cost of σ and by construction of our algorithm, it can only pay a total individual cost of at most 2σ.

We then begin our charging argument by showing that the cost of the non-buy tail services is at most the cost of the primary services. This is achieved through noticing that each tail service must end a chain with a corresponding, unique primary service. We will also show that the cost of the buy services is at most the cost of the normal services by using the fact that through preserving our invariant, a buy service costing B is only made when some set of normal services have paid an amount that is at least B. Overall, this means the cost of any tail service (buy or non-buy) can be charged to the normal and primary services.

Lastly, we will charge all our costs to OPT. We can show that the primary service costs can be charged to OPT by utilising the disjointness of these services. By definition, the time intervals of the triggering requests of each primary service must be disjoint and we know the optimal solution must make some service costing at least σ within each of these disjoint time intervals. Consequently, the cost of any primary service can be charged to the cost of the optimal service made within its triggering request's time interval. The crux of our analysis will be charging the normal service costs to OPT. This is done by noticing that normal services always have a total individual cost that exceeds σ and hence the cost of a normal service is dominated by its total individual cost. The total individual cost of all normal services corresponds precisely to the total individual cost incurred by all charged requests. Hence, we will show that for each charged request which incurs an individual cost of δ, OPT will pay a similar amount. This can be achieved by observing two crucial facts: at any point in time, there are at most two charged

requests on the same item type whose time intervals intersect and at any point in time, the number of charged requests that can intersect is at most $O(1) \cdot B/\delta$ due to our algorithm's invariant. This covers the costs of all the different types of service and by summing up all these different types, we will hence charge all costs to OPT.

3.3 Intuition: Piecewise Cardinality JRP with Deadlines

The ideas utilised in Rent-or-Buy will now be generalised to develop our algorithm for the piecewise case with n affine pieces, where n is arbitrary. Our cost function in this case will be $g(\lambda) = \min_i\{\sigma_i + \delta_i|\lambda|\}$ for $i = 1, \ldots, n$ where $2\sigma_i \leq \sigma_{i+1}$ and $\delta_i \geq 2\delta_{i+1}$ for all $i \in [1, n)$. We recall that serving on level l involves serving requests using the affine piece $\sigma_l + \delta_l|\lambda|$.

Many of the definitions, ideas and constructs will be the same as in the rent-or-buy case. To account for having more levels, we will introduce the concept of *request levels* to generalise the invariant we used to determine which level to serve on. When requests arrive, they will have their level set to 1 and this level will be constantly updated as the algorithm progresses. The level of a request will never decrease.

When do we make a service? Our algorithm will use the same idea as the rent-or-buy case to trigger services. That is, we will trigger a service when an active request reaches its deadline as this will ensure we serve all requests whilst also waiting as long as possible before doing so to allow more requests of the same item type to arrive. The only variation is that when a service is triggered, we will set its service level equal to the level of its triggering request. This service level will correspond to which level this service serves requests on. Once again, we remark that since deadlines are distinct, only one service should ever be triggered at a time.

Service chains. We will also use the same notion of service chains as in the rent-or-buy case which will then be used for our analysis. To capture the fact that we have more levels, we will introduce the concept of a *level l upgrade service* which is the first level $l > 1$ service of a chain. Once again, the primary service will be the first service of the chain, the tail service will be the last service of the chain (only if the chain does not end with a primary or upgrade service) and lastly, the normal services will be all other services. Figure 3.4 illustrates this generalised concept of chains and the different types of services. We will then later show that these chains are linear structures and when a level l service is made, all previous chains of level at most l must end.

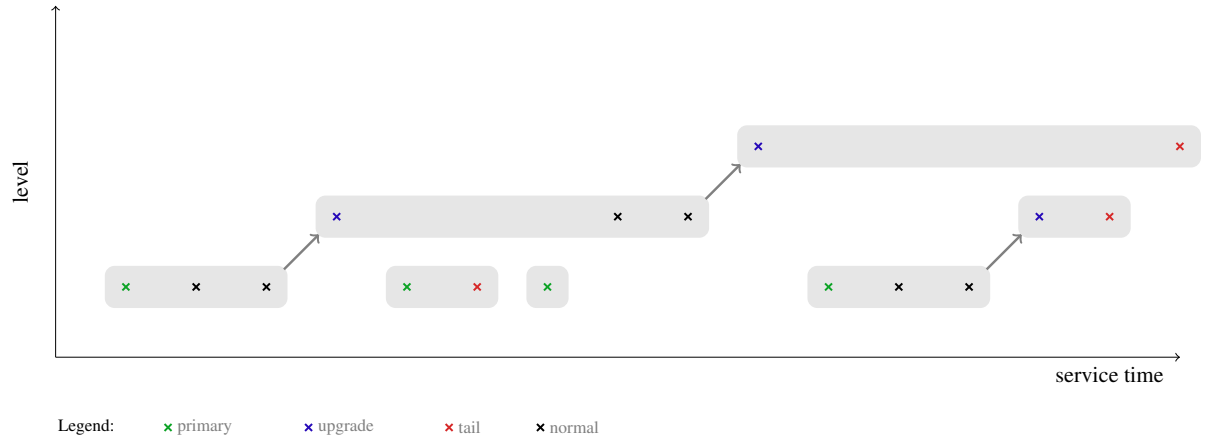


FIGURE 3.4: Illustration of chains and the different types of services. Gray boxes linked by arrows represent our service chains.

One slight variation in our chain definition in the piecewise case is how we determine the subsequent service in a chain. We introduce the notion of *request pointers* to generalise the rent-or-buy idea of looking for the most recently intersecting triggering request's time interval. We remark that the request pointers are distinct from the service pointers. We say a request is *eligible* for a level l service at time t if it is active at time t and has a request level of at most l. A request pointer then corresponds to the last service in which a request was eligible. A request's pointer is initially set to NULL and is updated at the end of each service it is eligible for. When a service is triggered, its service pointer is set to be the pointer of its triggering request. If this pointer is NULL, then we make a primary service and start a new chain. It is easy to see that this generalises the rent-or-buy service pointer definition since the most recent triggering request whose time interval intersects with our new service's triggering request would also be the most recent service to modify the pointer of the new service's triggering request. Hence, like in the rent-or-buy variant, we would also have that primary services have disjoint triggering requests which enables us to easily charge their costs to OPT. This generalised way of defining service pointers and chains is easier to keep track of which is necessary when working with multiple levels. We also remark that since deadlines are unique, so are the triggering requests and hence the service pointers.

What level service should we make? As mentioned earlier, the service level will correspond to the level we serve requests on. Initially, the service level is set to be the level of its triggering request. However, we will need a criterion to change the service level otherwise our algorithm will be permanently stuck on level 1. For this, we will generalise the invariant used in the rent-or-buy case to decide whether

or not the level of a service should be upgraded by 1. In the rent-or-buy case, we defined charged requests as those that were served in a normal rent (level 1) service and contributed to its total individual cost. Analogously, we will define a *level l charged request* to be one that is served in a level l normal service and contributed to this service's total individual cost. For rent-or-buy, the idea was to bound the number of charged requests that intersect at any time to be at most $O(1) \cdot B/\delta$. In the piecewise case where we have more levels, we will simply apply this condition individually across all levels. More precisely, at any level l and at any time t, the number of level l charged requests whose time intervals contain t will be at most $O(1) \cdot \sigma_{l+1}/\delta_l$. This invariant will be achieved by upgrading level l services to level $l+1$ services if at the service time there exists an eligible request within whose time interval there is a time when at least σ_{l+1}/δ_l level l charged requests intersect. This is clearly a generalisation of our condition in the rent-or-buy case and once again, we will later show that such a condition ensures our invariant. Moreover, this condition ensures that the cost of our upgrade services can be charged to the cost of the previous level's normal services, hence generalising the idea of charging the buy services to the rent services but across an arbitrary number of levels. Furthermore, by maintaining this invariant, we can bound the total individual costs incurred by normal services at any point in time which will be crucial to our analysis.

After a service is made, it will set the level of all eligible requests to be equal to the level of the service. This will allow the level of requests to increase and hence trigger higher level services later. This idea means that once a request has been eligible for a level l service it can only be served by a service of level at least l and is inspired by the algorithm proposed by Azar and Touitou (2020) which uses the same idea. This idea is crucial for maintaining our invariant. This is because a level l service gets upgraded to level $l+1$ and upgrades all eligible requests to level $l+1$ when there are too many intersecting requests that have already been served at level l. By upgrading eligible requests to level $l+1$ and preventing their level from decreasing, we prevent them from ever becoming level l charged requests and subsequently breaking the invariant.

How many and which requests to serve? Having decided the level we will serve on, we then need to decide which requests to serve. To do so, we will adopt the ideas of the rent-or-buy case and allocate a service budget and continually serve requests in ascending deadline order until the budget is reached. More precisely, a level l service will continually serve eligible requests in ascending deadline order until all eligible requests are served or the total individual cost exceeds the budget of σ_l. Whenever a request of a particular item type is served, we will serve all other eligible requests of the same item type. This

is a direct generalisation of the rent-or-buy case and once again allows us to balance serving as many requests as possible with leaving less urgent requests for later in hopes that more requests of the same item type arrive. Like in rent-or-buy, the budget is deliberately chosen to be σ_l such that services that exceed their budget will have their shared cost bounded by their individual cost. Since the individual cost dominates the total cost of these services that have exceeded their budget, we can then charge their cost to OPT by showing that when OPT serves any of the requests served by these services, it must pay a similar individual cost.

3.4 Piecewise Cardinality JRP with Deadlines Algorithm

3.4.1 Algorithm Description

We will design an algorithm that satisfies the following

THEOREM 3.2. *There exists an $O(1)$-competitive, deterministic algorithm for Piecewise Cardinality JRP with Deadlines.*

Initialisation. Recall a request is active at time t if it has arrived and has not yet been served and its time interval is the time between its arrival and deadline. Each request will also keep track of its level which is set to 1 when it arrives. Recall that a request is eligible for a level l service at time t if it is active and of level at most l at time t. Each request will also keep track of a pointer to represent the most recent service the request has been eligible for. This pointer is initially set to NULL when requests arrive. We will also keep track of the active chain at each level. This will be necessary to maintain our set of charged requests. Algorithm 3.1 covers this process.

Triggering a service and determining the service level. A service λ is triggered when a request reaches its deadline and such a request will be called the service's triggering request. The pointer of λ will be set to be the pointer of its triggering request and this will determine which chain it belongs to. If its pointer is NULL then it is a primary service and will begin a new chain. The level of λ will initially be set to be the level of its triggering request l. If the service is not a primary service and not a final level service (that is, $l < n$), then we must determine whether or not to upgrade the service level to $l + 1$. We do so by considering all requests eligible for the service E_λ, and if there exists a request $q \in E_\lambda$ within whose time interval, there is a time t' where at least σ_{l+1}/δ_l level l charged requests'

time intervals intersect, then we will upgrade the service. More precisely, we look at the time interval $[a_q, d_q]$ of $q \in E_\lambda$ and check if there exists at least σ_{l+1}/δ_l level l charged requests q_c with time intervals $[a_{q_c}, d_{q_c}]$ where $a_{q_c} \leq t' \leq d_{q_c}$ for some fixed $t' \in [a_q, d_q]$. We will call this request that caused the level to be upgraded the *upgrade request* and will call the intersecting level l charged requests that caused the upgrade the *witness set* of λ. Algorithm 3.2 gives the pseudocode for this process.

Serving and charging requests. To make the service λ of level l, we will continually add eligible requests in order of ascending deadline to λ until either all eligible requests have been served or the total individual cost exceeds σ_l, that is, $\delta_l|\lambda| \geq \sigma_l$. We remark that if a service was upgraded from level $l-1$ to l, then its set of eligible requests will now also include the active requests of level l. Whenever a request for an item type that has not been added to the service yet is added, if the service is non-primary and non-upgrade, then this request that is added will become a level l charged request. Whenever a request is added, all other eligible requests of the same item type, are also added to λ since these can be served for free. Note that, these extra requests added for free do not contribute to the service's total individual cost and hence do not become charged requests. At the end of the service, all unserved requests have their levels set to be equal to the level of the service as well as having their pointers updated to point at this service. We observe that the level of a request can never decrease since a higher level request can never be eligible for a lower level service and thus cannot have its level downgraded. Therefore, once a chain has upgraded to level l it cannot be downgraded back to previous levels. The cost of a service comprises of its shared cost σ_l and the total individual cost $\delta_l|\lambda|$ it pays to serve requests. Algorithm 3.3 outlines this process.

Uncharging requests. Now, we will discuss how the set of charged requests are maintained by the algorithm. If a service ends up serving all active requests and becomes a tail service then all charged requests created during the service are removed. Next, we observe that whenever a level l service λ is made, all previous chains of levels at most l must have ended. This is because any triggering requests that could continue these old chains must have been active during a service for one of these old chains. This would then also make these requests eligible for λ, after which their pointers would be set to λ which makes them unable to continue any old chains. Therefore, any chains that end in this way will have their last service become a tail service and the charged requests created by these services are removed. To utilise this fact, we will constantly keep track of the last service of any active chain at all levels and after every service, we will end any lower level chains and remove the charged requests in their last service.

This allows us to maintain that only normal services can create charged requests. Algorithm 3.4 outlines this process.

Algorithm 3.1: Procedures to handle requests arriving and keeping track of chains

```
/* initialise the last service of the active chain at each
   level to be NULL                                          */
for l ← 1 to n do
    active(l) ← NULL
Function OnArrival(request q)
    /* set level to 1 and pointer to NULL                   */
    level(q) ← 1;
    pointer(q) ← NULL;
```

Algorithm 3.2: Procedure to handle service triggering

```
Function OnDeadline(request q)
    /* triggering request reaches its deadline so we make a
       service                                               */
    Start a new service λ;
    pointer(λ) ← pointer(q);
    /* Flag for whether or not the service is a normal service */
    if pointer(λ) = NULL then
        normal(λ) ← False;
    else
        normal(λ) ← True;
    /* determine the service level                          */
    l ← level(q);
    Let the requests eligible for λ be E_λ;
    if normal(λ) and l ≠ n then
        foreach q' ∈ E_λ do
            if ∃t' ∈ [a_q', d_q'] and at least σ_{l+1}/δ_l level l − 1 charged requests q_c s.t.
              a_{q_c} ≤ t' ≤ d_{q_c} then
                l ← l + 1;
                /* upgrade services cannot be normal services */
                normal(λ) ← False;
                break;
    level(λ) ← l;
    ServeRequests(λ);
    UnchargeRequests(λ);
```

Algorithm 3.3: Procedure to handle serving and charging requests

Function *ServeRequests(service λ)*

$l \leftarrow level(\lambda)$;
Let the requests eligible for λ be E_λ;
/* keep track of which requests are to be served */
$Q_\lambda \leftarrow \emptyset$;
while $Q_\lambda \neq E_\lambda$ **do**
  Let $q_{last} \in E_\lambda \setminus Q_\lambda$ be the request with earliest deadline;
  Add the item type of q_{last} to λ;
  if $normal(\lambda)$ **then**
    Make q_{last} a level l charged request;
  /* add eligible requests of the same item type as q_{last} */
  foreach $q' \in E_\lambda$ *whose item type is the same as* q_{last} **do**
    $Q_\lambda \leftarrow Q_\lambda \cup \{q'\}$;
  /* stop if the total individual cost exceeds σ_l */
  if $\delta_l|\lambda| \geq \sigma_l$ **then**
    break;
Serve Q_λ;
/* updating unserved requests' pointers and levels */
foreach $q \in E_\lambda \setminus Q_\lambda$ **do**
  $level(q) \leftarrow l$;
  $pointer(q) \leftarrow \lambda$;

Algorithm 3.4: Procedure to handle uncharging requests

Function *UnchargeRequests(service λ)*

/* uncharge requests if λ serves everything */
if $E_\lambda \setminus Q_\lambda = \emptyset$ **then**
  Uncharge all requests in Q_λ;
/* end all previous active chains of levels at most l and remove the charged requests of their last service */
for $i \leftarrow 1$ **to** $level(\lambda)$ **do**
  if $active(i)$ *is in a different chain from* λ **then**
    Uncharge all requests in $Q_{active(i)}$;
  $active(i) \leftarrow$ NULL;
/* keep track of the new last service of the active chain at level $level(\lambda)$ */
if $E_\lambda \setminus Q_\lambda \neq \emptyset$ **then**
  $active(level(\lambda)) \leftarrow \lambda$;

3.4.2 Analysis Overview

To show constant competitiveness, we will look at the costs of all the different types of services individually then sum them together for the final result. The different types of services ALG can make are: primary services, upgrade services, tail services and normal services.

We will first examine the properties of our solution in Section 3.4.3. This will involve observing feasibility, arguing that our chains are well-defined and linear structures and showing the cost of any level l service is at most $3\sigma_l$. The proofs in this section are rather straightforward.

Next Section 3.4.4 will begin our charging argument. We will first show that the cost of any tail service is at most 3 times the cost of the primary and upgrade services. This will be done by observing that for any tail service, it must end a chain with a unique, corresponding primary or upgrade service. We will then charge the upgrade service costs to the cost of the normal services. This is possible since level l upgrade services are only made when we have enough intersecting level $l-1$ charged requests whose total individual cost is enough to cover the cost of the upgrade service. We will also prove that any charged request that belongs to the witness set of an upgrade service and is hence used to upgrade the service cannot be uncharged afterwards.

Lastly, Section 3.4.5 will charge the primary service costs and normal service costs to OPT. To charge the primary services to OPT, we utilise the disjointness of the triggering requests and the fact that within each disjoint triggering request's time interval, OPT must make some service costing at least σ_1. To charge the normal service costs to OPT we notice that its cost is dominated by its total individual cost which corresponds exactly to the total individual cost of the charged requests. We will then show that for each charged request, OPT must pay a similar amount to the individual cost it incurs. This is done by first showing that at any time, there can only be at most 2 level l charged requests on the same item type whose time intervals intersect and at most $2\sigma_{l+1}/\delta_l$ level l charged requests that intersect. Using these observations, we can then show that for any service λ^* made by OPT, the individual costs incurred by the charged requests its serves can be charged to the cost of λ^*.

We remark that our analysis is almost identical to the rent-or-buy case apart from the final lemma when we charge the cost of the normal services to OPT.

3.4.3 Analysis: Solution Structure

OBSERVATION 3.2.1. ALG *is a feasible solution.*

PROOF. To show feasibility, we show that all requests are served before their deadlines. Whenever a request reaches its deadline, a service will be triggered that serves this request. Hence, no request will ever be left unserved past its deadline. Thus, every request will be served before its deadline and our solution is feasible. □

OBSERVATION 3.2.2. *Every service can point to and be pointed to by at most one other service.*

PROOF. By definition of service pointers, every service can only point to one other service. Now suppose that a level l service λ' gets pointed to by two distinct services λ_1 and λ_2 where wlog λ_2 is served after λ_1. Consider the triggering requests of λ_1 and λ_2 and let them be q_1, q_2 respectively. When q_1 triggers λ_1, it has a pointer equal to λ' in order to cause λ_1 to point to λ'. By definition of pointers being the last service in which a request was eligible, this means that q_1 was last eligible for λ' which must have set the level of q_1 to be l. Therefore when q_1 triggers λ_1, it will initially be a level l service (prior to any upgrading).

Now at the service time of λ_1, q_2 has arrived, since it was eligible for λ' before λ_1. q_2 must also be unserved otherwise it can not trigger λ_2 after λ_1. Lastly, q_2 has a level of l since by definition of pointers, λ' is the last service to modify q_2's level until it triggers the service λ_2 after λ_1. Consequently, q_2 is eligible for λ_1 after which its pointer will not longer be λ', making it unable to make a service that points to λ', a contradiction. □

LEMMA 3.2.3. *Any level l service made by* ALG *costs at most* $3\sigma_l$

PROOF. The cost of any service is composed of two components: the shared, fixed cost of σ_l that is paid whenever the level l piece of the function is used and the individual cost of δ_l for each item type served in the service. By design of our algorithm, services continually serve requests until the total individual cost exceeds σ_l. Hence, in the worst case scenario, our service could pay a total individual cost of $\sigma_l + \delta_l < 2\sigma_l$. Therefore, the total cost of the service is at most $3\sigma_l$. □

3.4.4 Analysis: Charging service costs

LEMMA 3.2.4. *The cost of the tail services is at most 3 times the cost of the primary and upgrade services.*

PROOF. Consider any tail service of level l. Using Lemma 3.2.3, it costs at most $3\sigma_l$. The fact that it is a level l tail service means that it must end a chain that has reached level l and for this to happen, the chain must have had a level l upgrade service or a primary service in the case that $l = 1$. The level l upgrade/primary service costs at least σ_l since it must at least pay the shared service cost and since upgrade and primary services cannot be tail services we can charge the cost of the tail service to 3 times the cost of its corresponding upgrade/primary service. Lastly, we note that an upgrade or primary service cannot be charged to twice. This is because by Observation 3.2.2, each service can only point and be pointed to by one other service and hence our chains are linear structures. Therefore any upgrade/primary service can only ever upgrade/start one chain and thus every tail service will correspond to a unique upgrade/primary service. Therefore, considering all tail services, when we charge them to their corresponding upgrade/primary service, we will always be charging to a different upgrade/primary service and hence we get that the total cost of all tail services is at most 3 times the cost of the upgrade and primary services. □

LEMMA 3.2.5. *Suppose there is a level l charged request q_c with time interval $[a_c, d_c]$ served at time t. There cannot be any level $l' \geq l$ service between times t and d_c (inclusive) other than the service that served q_c.*

PROOF. By definition of q_c being level l charged, q_c must have been served by some level l normal service λ. By definition of normal services, there must be a subsequent service that directly continues the chain λ belongs to. Let the triggering request for this subsequent service be q' with time interval $[a', d']$. We know that $a' < t$ since q' must have been eligible for λ to have its pointer set such that it can continue the chain. We also have that $d' > d_c$ since q' was eligible for the same service as q_c but was left unserved and since requests eligible for a service are served in ascending deadline order this implies that $d' > d_c$. After this service, we also know that q' will have a level of l and cannot have its level modified until it is served at time d' since otherwise its pointer would have been overwritten and it could not have been the triggering request to directly continue the chain after λ.

Now suppose for a contradiction that there is a another level $l' \geq l$ service between times t and d_c inclusive. We know that q' will be eligible for this service since its level is at most l' and it is active since it has arrived at time $a' < t$ and cannot be served until time $d' > d_c$ as noted earlier. However, after this service, the pointer of q' will have been modified to point at this level l' service which would mean that q' can no longer be the triggering request for the service that is immediately after λ in its chain which is a contradiction. □

LEMMA 3.2.6. *Any charged request that belongs to an upgrade service's witness set and is hence used to upgrade the service cannot be uncharged after the upgrade service.*

PROOF. Consider any level $l-1$ charged request q that belongs to a level l upgrade service's witness set. That is, the charged request is one of the at least σ_l/δ_{l-1} charged requests whose time interval intersects with the upgrade service's upgrade request. We wish to show that this charged request cannot be uncharged after the upgrade service.

If the upgrade service belongs to the same chain as the service that served q then clearly the fact that we are making the upgrade service means the chain is continuing and hence previous services cannot be tail services and thus previously charged requests cannot ever be uncharged. If the upgrade service belongs to a different chain, then q necessarily belongs to a previous chain. If q belongs to a previous chain whose level has already exceeded level $l-1$ then this would imply that there is at least one level l service in its chain. Hence, the level $l-1$ service that q is served in must be non-tail since it is succeeded by some level l service. Thus the only scenario in which q could be potentially uncharged is if it belongs to a previous chain that is level $l-1$ and still active. Since our level l upgrade service was originally triggered as a level $l-1$ service, this means its chain must have had a level $l-1$ upgrade/primary service to reach level $l-1$. As noted in the algorithm description, such a service would end all previous chains that are still active at level $l-1$. In particular, it would end the chain that q belongs to and hence should already have uncharged q prior to the upgrade service. Therefore, any charged request belonging to another chain that could be uncharged should already have been uncharged by the time of our upgrade service. Therefore, we can guarantee that by the time the upgrade service is made, any of the level $l-1$ charged requests in its witness set will not be uncharged after the upgrade service. □

LEMMA 3.2.7. *Every charged request can only belong to the witness set of at most one upgrade service.*

PROOF. Suppose for a contradiction that this was not the case and there is a level l charged request q_c with time interval $[a_c, d_c]$ that belongs to the witness set of two level $l+1$ upgrade services. Let the time q_c is served and charged be t. The two level $l+1$ upgrade service q_c must occur after t since otherwise q_c would not be charged yet. By Lemma 3.2.6, the fact that q_c belongs to the witness set of any upgrade service implies that it will always persist as a charged service until the end of the algorithm. Consequently, we can apply Lemma 3.2.5 which tells us that there cannot be any service of level at least

l between times t and d_c. Therefore, the upgrade services whose witness sets contain q_c can only occur after time d_c.

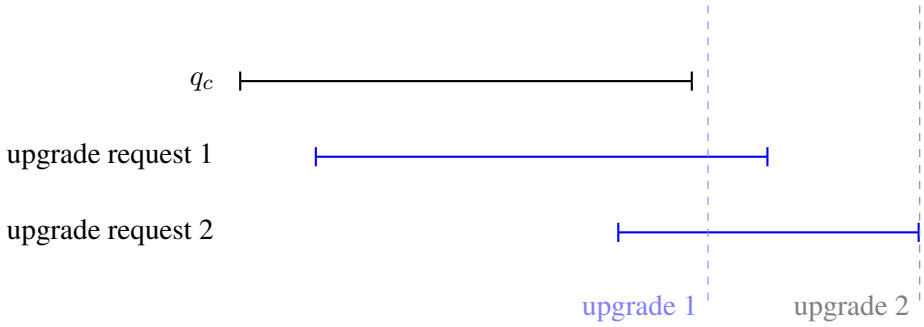


FIGURE 3.5: Illustration of Lemma 3.2.7's proof.

Now consider the upgrade requests of the upgrade services. By definition of being a level $l+1$ upgrade services' upgrade request, these requests must have level at most l prior to their upgrade services. Since by assumption these upgrade requests' intervals must contain a time that intersects with q_c we must have that both arrive before d_c. Now consider the earlier upgrade service to be made which as noted earlier must be at some time after d_c. At this time, both upgrade requests are eligible for this upgrade service since they have arrived, cannot have been served yet and are of level at most l. After this earlier service, the upgrade request for the second upgrade service would have been upgraded to level $l+1$ and by definition, it is unable to be an upgrade request for a level $l+1$ upgrade service. Therefore, there cannot be a second upgrade service that uses q_c, a contradiction. Figure 3.5 shows an illustration of this proof. □

LEMMA 3.2.8. *The cost of the upgrade services is at most 3 times the cost of the normal services.*

PROOF. Consider any level l upgrade service. Using Lemma 3.2.3 it costs at most $3\sigma_l$. By construction of our upgrade condition, the witness set of the upgrade service must contain at least σ_l/δ_{l-1} level $l-1$ intersecting charged requests. By definition of charged requests, a request is level $l-1$ charged iff it contributed a distinct cost of δ_{l-1} to some level $l-1$ normal service. Hence, since there are at least σ_l/δ_{l-1} charged requests in the witness set, the total individual cost they would have contributed to level $l-1$ normal services is at least $\sigma_l/\delta_{l-1} \cdot \delta_{l-1} = \sigma_l$. Therefore, we can charge the cost of the upgrade service to 3 times the individual costs incurred by the charged requests in its witness set.

Thus, we can charge all upgrade service costs to the costs of their corresponding witness set's charged requests. By Lemma 3.2.7, we know that each upgrade service's witness set is disjoint and thus we will

be charging to distinct charged requests. Overall, this gives us that the total cost of the upgrade services is at most 3 times the total individual cost of the charged requests. By definition, the charged requests are those that contribute some distinct individual cost to a normal service. Hence, the total individual cost of the charged requests is at most the total individual cost of the normal services. Moreover, by Lemma 3.2.6, we know that the charged requests in the witness set of an upgrade service will never be uncharged and will remain charged until the end of the algorithm. Therefore, we are guaranteed that these charged requests we charge the upgrade services to do indeed contribute an individual cost to a normal service as opposed to some service that later becomes a tail. Ultimately, we get that the cost of upgrade services is at most 3 times the cost of the normal services. □

3.4.5 Analysis: Charging to OPT

LEMMA 3.2.9. *The cost of the primary services is at most 3 times the cost of* OPT.

PROOF. We first observe that the triggering requests of primary services are disjoint, that is, their time intervals are disjoint. Suppose towards a contradiction that this was not the case then there are two triggering requests q_1 and q_2 with intersecting intervals $[a_1, d_1]$, $[a_2, d_2]$ respectively. Since these intervals are not disjoint, we can assume wlog that $a_1 < a_2 < d_1$. If $d_1 < d_2$ then at time d_1 when q_1 triggers a primary service, q_2 is eligible and will have its pointer set to be the service of q_1. Since the second triggering request no longer has a NULL pointer it, by definition, cannot trigger a primary service. Likewise, if $d_2 < d_1$ then when q_2 triggers its primary service, q_1 would be eligible and have its pointer become non-NULL, making it unable to trigger a primary service.

Now consider the time interval of any primary services' triggering request. The corresponding primary service is by definition level 1 and hence has a cost of at most $3\sigma_1$ using Lemma 3.2.3. OPT must make a service at some time within this interval otherwise it would not be a feasible solution as it would have left a request unserved. If OPT makes a level l service during this time, it would cost at least $\sigma_l \geq \sigma_1$ since OPT must pay at least the shared cost. Therefore, we can charge the cost of the primary service to 3 times the cost of the OPT service that served the service's triggering request. Using the disjointness of the triggering requests, we get that when we charge each primary service to its corresponding OPT service that serves its triggering request, we will always be charging to a distinct service since no OPT service can serve two primary service's triggering requests together due to their disjointness. Therefore, we get that the cost of all the primary services is at most 3 times the cost of OPT as required. □

OBSERVATION 3.2.10. *At any time t^*, there can only be at most 2 level l charged requests on the same item type whose time intervals intersect t^*.*

PROOF. Consider three level l charged requests q_1, q_2, q_3 on the same item type with time intervals $[a_1, d_1]$, $[a_2, d_2]$, $[a_3, d_3]$ respectively. Since these requests are level l charged, this means by definition that they must have been served in different level l normal services since if two requests of the same item type are served in the same normal service, only one request will be charged. Let the time of the service that served q_1 be t_1, q_2 be t_2 and q_3 be t_3 and we can assume wlog that $t_1 < t_2 < t_3$.

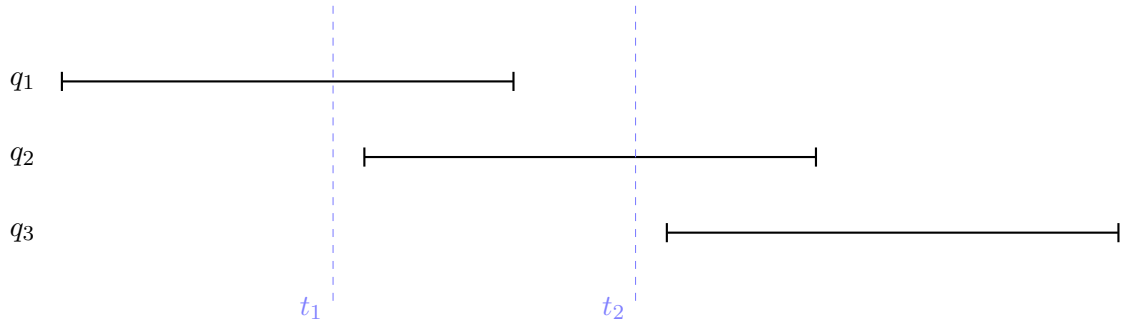


FIGURE 3.6: Illustration of Observation 3.2.10's proof.

We now note that $a_2 > t_1$ otherwise q_2 would have been eligible to be served with q_1 at time t_1 since it would have arrived by t_1 and have level at most l since its final level when served is at most l. Similarly we have that $a_3 > t_2$. Using Lemma 3.2.5, we know that t_2 cannot be between t_1 and d_1 and since $t_2 > t_1$ we must have that $t_2 \geq d_1$. Combining this with the fact that $a_3 > t_2$ we have that $a_3 > d_1$ and therefore the time interval of q_3 is completely disjoint from the interval of q_1. This implies that at any time, there can only be at most two level l charged requests on the same item type whose time intervals intersect with this time as required. □

OBSERVATION 3.2.11. *For a fixed level $l < n$, the maximum number of level l charged requests whose time intervals intersect at some time t^* is at most $2\sigma_{l+1}/\delta_l$.*

PROOF. Any level l normal service made by the algorithm can only serve and hence charge at most $\lceil \sigma_l/\delta_l \rceil \leq \left\lceil \frac{\sigma_{l+1}}{2\delta_l} \right\rceil < \sigma_{l+1}/\delta_l$ requests. Now suppose for a contradiction that at some time t^* we have more than $2\sigma_{l+1}/\delta_l$ level l charged requests and consider the last service to have served and charged any of these charged requests. This service is necessarily a level l normal service by definition of level l charged requests. At the time of this service but before the service has been made, the number of charged requests intersecting at time t^* is at least $2\sigma_{l+1}/\delta_l - \sigma_{l+1}/\delta_l = \sigma_{l+1}/\delta_l$ since this final service could

only have charged at most σ_{l+1}/δ_l requests as noted earlier. At the time of this last service, there must have been an eligible request that intersects with time t^* that could become a level l charged request since by definition of this last service, such a request will be served and charged by this service. But we also know that at time t^* there already exists at least σ_{l+1}/δ_l level l charged requests which means this request should have been an upgrade request and the service should have been a level $l+1$ upgrade service and thus been unable to create any more level l charged requests, a contradiction. Hence, this shows the required result. □

LEMMA 3.2.12. *The cost of a normal service is at most 2 times its total individual cost.*

PROOF. Consider any normal service of level l. Its cost is composed of two elements: the shared cost of σ_l and the total individual cost of $\delta_l|\lambda|$. By design of our algorithm, a normal service must have a total individual cost of at least σ_l. If this were not the case, it would have served all eligible requests and thus been a tail service as there is no unserved request to trigger a subsequent service that continues the chain. Therefore, the shared cost of the normal service is at most the total individual cost of the normal service and we get that the cost of the service is at most 2 times its total individual cost. □

LEMMA 3.2.13. *The cost of the normal services is at most 8 times the cost of* OPT.

PROOF. Using Lemma 3.2.12 the total cost of all normal services is at most 2 times the total individual cost of all normal services. Now we observe that the only requests that contribute an individual cost to a normal service are precisely the charged requests. Any uncharged requests are either not in a normal service or included in a normal service since a charged request of the same item type had been included and hence its inclusion incurs no additional individual cost. Therefore, the total individual cost of all our normal services can be charged to OPT by looking at the charged requests and showing that for each level l charged request, OPT pays at least $O(1) \cdot \delta_l$ uniquely for this request. Therefore, we will consider each service λ^* made by OPT and show that the individual cost incurred by each charged request it serves can be charged to the cost of λ^*. The cost of λ^* will consist of its shared and individual cost paid to serve requests. Since OPT must serve all requests and thus all charged requests, this will allow us to charge all the individual costs incurred by charged requests to OPT.

Consider any service λ^* made by OPT at level l^* and at time t^*. Consider the charged requests of level at least l^* served by λ^*. Consider any fixed level $l \geq l^*$, and any fixed item type i. By Observation 3.2.10, at most 2 level l charged requests of item type i can intersect at any time t^* and hence at most 2 level l charged requests of item type i could have been served by λ^*. Since each level l charged request

has an associated individual cost of δ_l that we need to charge to OPT, these level l charged requests on item type i incur an individual cost of at most $2\delta_l$ that must be charged to λ^*. Considering the item i across all levels $l \geq l^*$, the total individual cost that must be charged to λ^* is at most $\sum_{l=l^*}^{n} 2\delta_l$.

Using the fact that $\delta_i > 2\delta_{i+1}$ for each $i \in [1, n-1]$, we can turn this sum into a geometric sum and bound it by its limiting sum. This gives us

$$\begin{aligned}\sum_{l=l^*}^{l=n} 2\delta_l &< 2\delta_{l^*} \sum_{l=l^*}^{n} (\frac{1}{2})^{l-l^*} \\ &< 2\delta_{l^*} \sum_{i=0}^{\infty} (\frac{1}{2})^{i} \\ &= 4\delta_{l^*}\end{aligned}$$

Now, to serve this particular item type i, λ^* will have to pay a distinct cost of δ_{l^*}. Hence, we can charge the individual costs of all the charged requests on item type i to this 4 times this cost. Applying this argument across all item types and noting that each item type gets charged to a different δ_{l^*} payment in λ^* we get that the cost of all charged requests served by λ^* and of level at least l^* is at most 4 times the individual cost of λ^*.

Now we will consider the charged requests of level less than l^* that are served by λ^*. For a fixed level $l < l^*$, we know by Observation 3.2.11 that the maximum number of level l charged requests that can intersect at a time t^* is at most $2\sigma_{l+1}/\delta_l$ and hence this is also the maximum number of level l charged requests that λ^* can serve. Since each level l charged request has an associated individual cost of δ_l that we need to charge to OPT, the level l charged requests served by λ^* accumulate a total individual cost of at most $2\sigma_{l+1}/\delta_l \cdot \delta_l = 2\sigma_{l+1}$ that must be charged to λ^*. By considering the charged requests served by λ^* at each level $l < l^*$, the total individual cost of all these charged requests is at most $\sum_{l=1}^{l^*-1} 2\sigma_{l+1}$.

We know that $\sigma_{i+1} > 2\sigma_i$ so once again, we can use the sum of a limiting geometric series to bound the total individual cost of these charged requests as follows

$$\begin{aligned}\sum_{l=1}^{l^*-1} 2\sigma_{l+1} &< 2\sigma_{l^*} \sum_{l=1}^{l^*-1} (\frac{1}{2})^{l^*-1-l} \\ &< 2\sigma_{l^*} \sum_{i=0}^{\infty} (\frac{1}{2})^{i} \\ &= 4\sigma_{l^*}\end{aligned}$$

Now we know λ^* is a service at level l^* and hence must pay a shared cost of σ_{l^*} so, therefore, we can charge the individual costs of all the charged requests of level less than l^* to 4 times the shared cost of λ^*. Combining this fact with the previous case, we get that for each OPT service λ^*, we can charge the individual costs of the charged requests it serves to 4 times the shared plus individual cost of λ^* which is precisely 4 times the cost of λ^*.

By applying this argument across all services made by OPT and noting that OPT must serve all requests and in particular all charged requests, we have that all the individual costs incurred by charged requests will be charged to some OPT service and therefore the total individual cost incurred by charged requests is at most $4 \cdot \mathrm{OPT}$. Ultimately, since the total cost of all the normal services is at most 2 times the total individual cost of the normal services and knowing that this cost corresponds to the total individual cost incurred by the charged requests, we conclude that the total cost of all the normal services is at most $8 \cdot \mathrm{OPT}$ as required. □

Finally, we prove our Theorem

THEOREM 3.2. *There exists an $O(1)$-competitive, deterministic algorithm for Piecewise Cardinality JRP with Deadlines.*

PROOF. We have already observed in Observation 3.2.1 that our algorithm is feasible so it remains to show that it is constant competitive, or in other words, $\mathrm{ALG} \leq O(1) \cdot \mathrm{OPT}$. The cost of ALG can be broken down into the costs of the different types of services it makes which are the primary, upgrade, tail and normal services. Let the costs of these services be P, U, T and N respectively then

$$
\begin{aligned}
\mathrm{ALG} &= P + U + T + N \\
&\leq 4P + 4U + N && \text{(Lemma 3.2.4)} \\
&\leq 4P + 13N && \text{(Lemma 3.2.8)} \\
&\leq 12\,\mathrm{OPT} + 13N && \text{(Lemma 3.2.9)} \\
&\leq 116\,\mathrm{OPT} && \text{(Lemma 3.2.13)}
\end{aligned}
$$

Therefore, we have that $\mathrm{ALG} \leq O(1) \cdot \mathrm{OPT}$ as required. □

CHAPTER 4

Piecewise Cardinality JRP with Delay Algorithm

Our cost function in this problem is still $g(\lambda) = \min_i\{\sigma_i + \delta_i|\lambda|\}$ for $i = 1, \ldots, n$ where $2\sigma_i \leq \sigma_{i+1}$ and $\delta_i \geq 2\delta_{i+1}$ for all $i \in [1, n)$. The only difference when compared to the deadline variant is that when a request q arrives at time a_q, it has an associated delay function $d_q(t)$ that corresponds to the amount of delay accumulated by q up till time t. When q is served at time t', the delay penalty $d_q(t')$ must be completely paid off. Moreover, prior to q's arrival, it must have no delay penalty: $d_q(t) = 0$ for all $t \leq a_q$.

4.1 Intuition

The ideas used for the deadline variant will be generalised to the delay model. The main problem with generalising to the delay model is we no longer have an immediate notion of urgency. Previously with deadlines, we would serve requests in ascending deadline order which represents their urgency but with delays, we must find a new notion of urgency. Furthermore, it is no longer obvious when exactly a service should be triggered as there are no deadlines that must be met. Moreover, without deadlines, we must be more careful when balancing serving as many requests as possible with leaving requests unserved to wait for other requests of the same item type. This is because, with delays, every request could potentially wait an arbitrary time for other requests of the same item type. Overall, we must define our own notion of urgency and use that to determine when services are made and which requests to serve. The ideas used previously for deciding which level to serve at must also be tweaked since we no longer have clearly defined time intervals for our requests.

When do we make a service? Akin to the deadline variant, a request is active at time t if it has arrived by time t and is unserved. To trigger services, we will adopt an idea commonly used for online algorithms with delay and used by Azar and Touitou (2020) which is to trigger a service once the delay accumulated is similar to the service cost.

To do so, we will first define the notion of *residual delay*. As mentioned previously, the idea of investments that will be used by our algorithm allows services to incrementally pay off the delay that requests have accumulated and might accumulate in the future. Consequently, we need some notion to capture how much delay on a request has not been paid off yet. Therefore, we define the residual delay on a request q at time t as the amount of delay accumulated by q up to time t that has not been paid for by some service.

Akin to the deadline case, our requests will have levels that are initially set to 1 upon arrival. We will then trigger a level l service when the active requests of level at most l accumulate a total residual delay of σ_l. The intuition here is that our algorithm will make level l services whose total cost is $O(1) \cdot \sigma_l$ and hence waiting until σ_l unpaid delay accumulates means that our delay cost will be comparable to the service cost and hence can easily be charged to the service cost. Moreover, this also provides sufficient time to wait for more requests of the same item type to arrive. Similar to the deadline variant, we will then define the set of *triggering requests* for a level l service as the level l active requests whose residual delay is positive when the service is triggered and hence contributed to triggering the service.

Service chains. We will once again use the notion of service chains. Our service chains and the different types of services will be identical to the deadline variant. That is, we will have primary, normal, level l upgrade and tail services. The primary service will be the first service that begins a chain, the level l upgrade service is the first level l service in a chain, the tail service is the last service of a chain (if the last service is not a primary/upgrade service) and the normal services are all other services. We will argue later that our service chains are linear structures. Figure 4.1 illustrates the service chains and the different types of services and we remark that this diagram is the same as in the deadline case since our definitions are unchanged.

The only deviation of our chains from the deadline case is in how we will define the service pointers which will then determine the chains. We will once again say a request is eligible for a level l service at time t if it is active at time t and is of level at most l. We will also reuse the idea of request pointers which represent the last service a request was eligible for. Once again, request pointers are set to NULL upon arrival. In the deadline case, we had a unique triggering request whose pointer was used to determine the service pointer but in the delay variant, we instead have a set of triggering requests which we must use to determine the service pointer. If all the triggering requests have a NULL pointer then the service pointer will be NULL and hence a primary service will be made and a new chain will begin. Otherwise, if any of the triggering requests have a non-NULL pointer, we will set the service pointer to be any

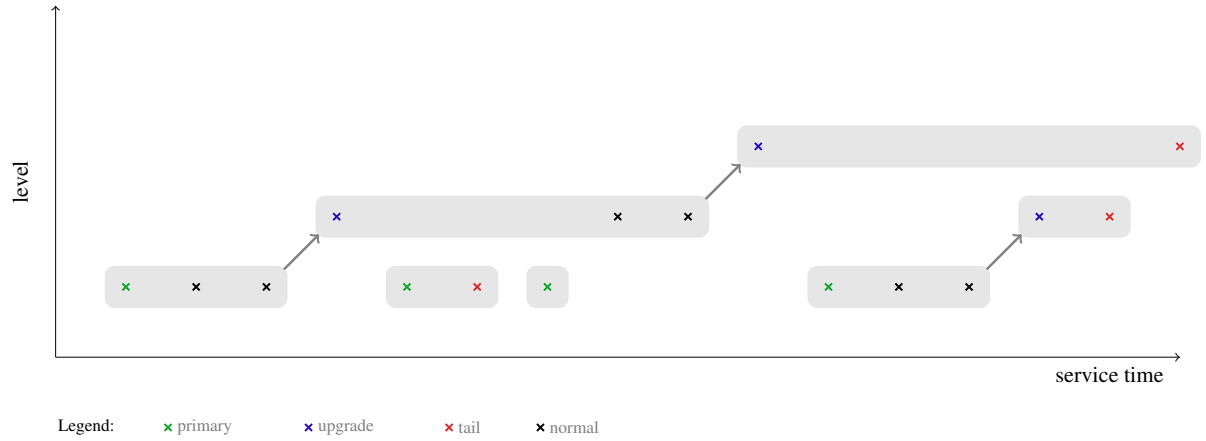


FIGURE 4.1: Illustration of chains and the different types of services. Gray boxes linked by arrows represent our service chains.

of these non-NULL pointers arbitrarily. We will later show that all non-NULL pointers in the set of triggering requests are identical and hence it does not matter which pointer we choose. Therefore, our chain structures will always be unique and not vary depending on this choice.

Analogous to the deadline case, primary services will always have some unique and disjoint set of triggering requests and this disjointness enables us to charge the cost of the primary services to OPT.

What level service should we make? When we trigger a level $l < n$ service, λ, if it is not a primary service then similarly to the deadline case, we will determine whether or not to upgrade the service to level $l + 1$. For this, we will be adapting the invariant used in the deadline case.

The purpose of our invariant in the deadline case was to restrict the number of requests that could have contributed to the individual costs of normal services at any time and hence bound the total individual costs incurred by normal services at any time. This was possible since each request has a natural time interval within which it must be served. In the case of delays, it is unclear whether we can construct an interval within which we are guaranteed that a request must be served. Furthermore, in the delay case, requests not only incur individual costs when they are served but also incur costs over time due to their delay penalty. Therefore, we will need a generalised invariant that can bound both the individual costs and the delay costs at any time. As will be elaborated upon later when we explain the details of our notion of investment, we will only serve requests once the amount invested into them equals the individual cost of serving the request and hence, it suffices to create an invariant that bounds the investment costs as this will consequently bound both the delay and individual costs.

To handle no longer having well defined time intervals for our requests, we will introduce the notion of *level l charged investment intervals* for requests. When a level l normal service invests in the future residual delay accumulated by a request from time t to τ, we will construct the level l charged investment interval $[t, \tau]$ for this request with the cost of this interval corresponding to the amount the normal service has invested. Then our invariant will be that at any time t', the intersecting level l requests (that is, the level l requests that were active at time t') will have their total future level l charged intervals bounded by $O(1) \cdot \sigma_{l+1}$. That is, whenever we consider any time t' and look at all level l requests that were active at this time, then take the sum of all these requests' level l charged investment intervals that begin after time t', this sum will be at most $O(1) \cdot \sigma_{l+1}$.

To achieve this invariant, when we trigger a level l service at time t that is eligible for an upgrade we will look at the earliest arriving eligible request. Let its arrival time be a_q then we will look at all level l requests that have level l charged investment intervals beginning after time a_q. A request is considered to be a level l request if it is level l at the current time t or has been served at level l sometime before t. If the sum of these investment intervals is at least σ_{l+1} then we will upgrade the service to level $l + 1$. Figure 4.2 illustrates this condition.

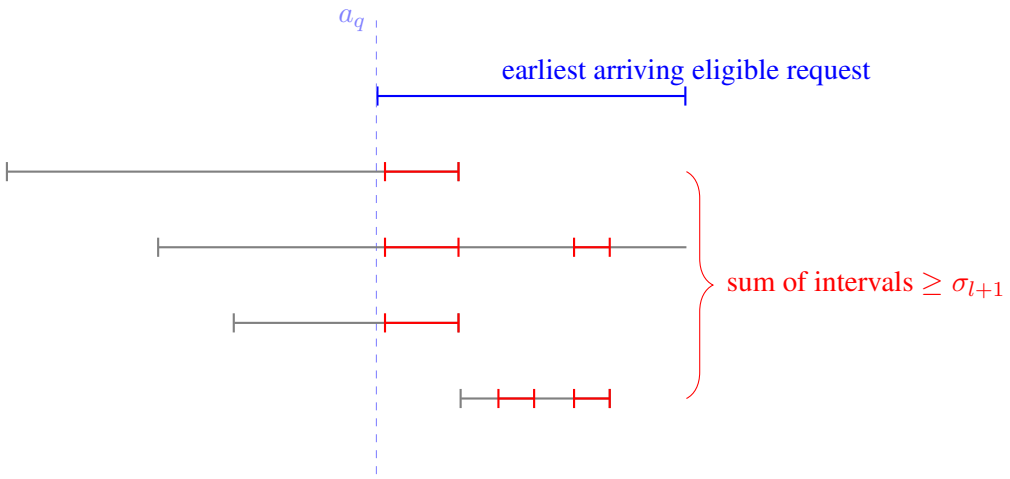


FIGURE 4.2: Level l charged investment intervals that cause a level l service to upgrade to level $l + 1$. The grey intervals represent requests that are level l at the current time t and the red intervals represent the level l charged investment intervals that begin after time a_q.

Whilst this condition for upgrading is more complicated than the deadline case, it will indeed achieve the invariant. This is because whenever a level l service could create a level l charged investment interval that causes the invariant to break, it would have to cause the invariant to break at some time after a_q since the

service cannot have been made before this time and hence cannot make any investment intervals before this time. However, this would imply that prior to creating this new interval to break the invariant, the level l investment intervals beginning after a_q are already at least σ_{l+1} which would have upgraded the service to level $l+1$ and hence prevented it from creating any more level l charged investment intervals. As will be seen later, this invariant will be crucial to our analysis.

Much like the deadline case, at the end of a service, all eligible requests will be upgraded to the level of the service. Once again, this ensures our request levels are non-decreasing and hence they cannot create lower levelled investment intervals which could break the invariant.

How many and which requests to serve? To serve our requests we will be using the idea of investments, and once a sufficient amount has been invested in a request to cover its individual cost we will serve the request. This means our investment costs will always cover our individual costs and hence to limit how much we serve in a service and ensure we are properly balancing serving as many requests as possible whilst also leaving requests to wait for future requests of the same item type, it suffices to restrict how much we spend investing. For this, we will adopt the same idea from the deadline case where we use a service budget and continually invest until the budget is met. Consequently, this will enable us to attempt to serve as much as possible whilst also allowing requests to wait for more requests on the same item type.

We will now describe how exactly our services invest. Each level l service will have an investment phase to invest in requests. This phase will determine which requests to serve by using our *level l investment counters*. For each possible item type i, we will have a level l investment counter for that item type that is initially set to 0 to count how much has been invested in requests of item type i by level l services. Then our investment phase will begin by setting the future time τ to be t then continuously increase τ to simulate and invest in the residual delay that unserved, eligible requests accumulate up to the future time τ. As we invest in requests, we will add the amount invested to each request's item type's corresponding level l investment counter. Whenever a level l investment counter for item type i reaches δ_l, we have accumulated sufficient investment costs to cover the individual cost of service and will hence serve all eligible requests of item type i and reset the investment counter to 0. We will continue simulating and investing in future delays until either all requests are served or the total invested amount reaches our investment budget of σ_l. Like the deadline case, this budget of σ_l is deliberately chosen such that if it is reached, the shared cost of our service can be bound by the investment cost and hence the total cost of the service can be charged to OPT by simply charging the investment costs.

This investment process also implicitly defines a notion of urgency as more urgent requests can be interpreted as those that accumulate delay faster and hence will have investment counters reach the required threshold faster. Therefore, this nicely generalises the notion of serving in ascending order of urgency utilised in the deadline variant.

4.2 Algorithm Description

Our algorithm will satisfy the following

THEOREM 4.1. *There exists an $O(1)$-competitive, deterministic algorithm for Piecewise Cardinality JRP with Delay.*

Initialisation. When any request arrives we set its level to 1 and its pointer to NULL. Our algorithm must also keep track of a level l investment counter for item type i for each level and item type. This is initialised as 0 for all levels l and item types i. Our algorithm will also need to keep track of the currently active chain at each level which will be used later to maintain the set of charged investment intervals. As will be discussed later, only one chain can be active at any level at any time and hence for any active chain at level l we will record the last service of this chain. Pseudocode for this initialisation process is given in Algorithm 4.5.

Triggering a service. Recall a request is active at time t if it has arrived and has not yet been served and the residual delay on a request at time t is any delay it has accumulated up to time t that has not been paid off by any service. A level l service is triggered when active requests of level at most l accumulate σ_l residual delay. Recall that the triggering requests for this service are the active level l requests with positive residual delay whose delay contributed to this service triggering. The triggered service's pointer will be determined by looking at its triggering requests. If all the triggering requests have a NULL pointer then the service pointer will be NULL and we will make a primary service and start a new chain. Otherwise, the service pointer will be set to be any of the non-NULL triggering request pointers and we will continue a previous chain. As will be shown later, all requests in the triggering requests set with a non-NULL pointer must in fact point to the same service so it does not matter which non-NULL triggering request pointer we choose.

Determining the service level. When a level l service λ is triggered at time t if it is non-primary and not of the final level n already then it is eligible for a level upgrade. To determine whether or not we upgrade the service level to $l+1$, we look at the earliest arriving eligible request q. Let this request have arrival time a_q. Then we consider the set W_λ of level l charged investment intervals that begin after time a_q and who belong to requests that are currently level l. This set will be called λ's *witness set*. A request is said to be currently level l if it is active and level l at the time t or has been served before t by a level l service. Then if the cost of the intervals in the witness set is at least σ_{l+1}, that is if $\sum_{c \in W_\lambda} cost(c) \geq \sigma_{l+1}$, we will upgrade λ's level to $l+1$. Note that upgrading the level of a service will not change its triggering requests or pointer but after upgrading, all requests of level $l+1$ will now also be eligible for λ and hence λ's set of eligible requests E_λ may increase. Algorithm 4.6 gives the pseudocode for the process of triggering a service and upgrading its level.

Making the service. Our level l service λ at time t first pays off the residual delay that each eligible request $q \in E_\lambda$ has accumulated up to the service time t. Let $r_q(t_1, t_2)$ denote the residual delay accumulated by request q from times t_1 to t_2 then for each eligible request $q \in E_\lambda$, we will pay off all residual delay since their arrival time, that is, $r_q(a_q, t)$. Then the service will begin the investment phase with an investment budget of σ_l where it invests in the future residual delay that the eligible requests accumulate from t. This begins with setting the future time $\tau \leftarrow t$, the previous time $t' \leftarrow t$, the set of served requests $Q_\lambda \leftarrow \emptyset$ and the service investment counter to be 0. We then continuously increase τ. Each time τ increases, the residual delay incurred from time t' to τ by each request $q \in E_\lambda \setminus Q_\lambda$ is paid off and invested in. That is, we pay off $r_q(t', \tau)$ for each $q \in E_\lambda \setminus Q_\lambda$. This residual delay $r_q(t', \tau)$ invested in is added to the level l investment counter for q's corresponding item type as well as the service investment counter. We will also add this amount to $I_q(\lambda)$ which will keep track of how much λ has invested in the request q and will be used later to construct our charged intervals. If the investment counter for an item type i reaches δ_l then all eligible requests of item type i are added to the set Q_λ to be served and we stop investing in these requests. Finally, we set $t' \leftarrow \tau$ and iteratively continue the process. This process terminates if all eligible requests have been served and added to Q_λ or if the service investment counter equals σ_l which signifies a total of σ_l has been invested in future delay incurred by eligible requests. We note that at the end of this process, all eligible requests will have their delay paid off until time τ and hence can only accumulate residual delay beginning from time τ. To finish the service, we serve the requests in Q_λ, paying a fixed cost of σ_l as well as δ_l for each item type in Q_λ. The total cost of this service consists of the residual delay paid off at the beginning, the

future residual delay invested in during the investment phase, the shared cost of the service and the total individual cost of the service. This process is captured in Algorithm 4.7.

Updating request pointers, levels, investment counters and investment intervals. All requests eligible for λ have their pointer set to λ and levels set to l at the end of the service. If λ serves item type i then the level l investment counter for item i is reset to 0. If the service is not a primary or upgrade service then for each request that was eligible for λ, we construct the level l charged investment interval $[t, \tau]$ for this request with a cost corresponding to how much λ invested in the request during its investment phase, which will be stored in $I_q(\lambda)$. Note, all eligible requests for this service will have the same charged investment interval start and end times but the cost of each interval may differ. Pseudocode for these steps is given at the end of Algorithm 4.7.

Removing old charged investment intervals. Since our charged investment intervals must correspond to investments made by normal services, we have to remove previously created investment intervals if a previous service becomes a tail service. If λ serves all eligible requests then it becomes a tail service so all the charged intervals created by λ are removed. When λ is served, all previous chains of level at most l will end since any request pointing to a service from another chain of level at most l that could continue a previous chain will have been eligible for λ and thus have its pointer overridden after λ's service. Consequently, the chains which end this way will have their last service become a tail service if it was not an upgrade or primary service and any charged intervals constructed by such a service will thus be removed. This is why our algorithm will keep track of the last service of each active chain at each level as this will allow us to easily remove charged investment intervals once these chains are ended. This is outlined in Algorithm 4.8.

Algorithm 4.5: Initialisation procedures

```
/* initialise the last service of the active chain at each
   level                                                    */
for l ← 1 to n do
    active(l) ← NULL
/* initialise the investment counters                       */
for l ← 1 to n do
    foreach item type i do
        counter(l, i) ← 0
Function OnArrival(request q)
    level(q) ← 1;
    pointer(q) ← NULL;
```

Algorithm 4.6: Procedure to handle triggering and upgrading services

```
Function OnTriggering(level l)
    Start a new service λ at current time t;
    /* determine service pointer using triggering requests  */
    Let Q_trigger be all requests of level l with positive residual delay;
    pointer(λ) ← NULL;
    foreach request q in Q_trigger do
        if pointer(q) ≠ NULL then
            pointer(λ) = pointer(q);
            break;
    /* flag to keep track of whether the service is normal  */
    if pointer(λ) = NULL then
        normal(λ) ← False;
    else
        normal(λ) ← True;
    /* upgrade the service level if possible                */
    if pointer(λ) ≠ NULL and l ≠ n then
        if Σ_{c∈W_λ} cost(c) ≥ σ_{l+1} then
            l ← l + 1;
            normal(λ) ← False;
            break;
    level(λ) ← l;
    MakeService(λ, t);
    RemoveOldIntervals(λ);
```

Algorithm 4.7: Procedure to handle serving requests and updating information post-service

Function *MakeService(service λ, service time t)*

```
/* pay off all residual delay on eligible requests          */
Let E_λ be all eligible requests;
foreach q ∈ E_λ do
    Pay off the residual delay r_q(a_q, t);
    /* set up counter to track how much λ invests in each
       request                                                */
    I_q(λ) ← 0;
/* Investment Phase                                         */
l ← level(λ); τ ← t; t' ← t; Q_λ ← ∅; invested ← 0;
while Q_λ ≠ E_λ and invested < σ_l do
    Increment τ;
    foreach q ∈ E_λ \ Q_λ do
        Let the item type of q be i;
        counter(l, i) ← counter(l, i) + r_q(t', τ);
        invested ← invested + r_q(t', τ);
        I_q(λ) ← I_q(λ) + r_q(t', τ);
        if counter(l, i) ≥ δ_l then
            Add all requests in E_λ of item type i to Q_λ;
            counter(l, i) ← 0;
    t' ← τ;
Serve Q_λ;
foreach q ∈ E_λ do
    /* updating request pointers and levels                 */
    level(q) ← l;
    pointer(q) ← λ;
/* construct charged investment intervals if service is
   normal                                                   */
if normal(λ) then
    foreach q ∈ E_λ do
        Construct the level l charged investment interval [t, τ] with cost I_q(λ);
```

Algorithm 4.8: Procedure to handle removing old charged intervals

Function *RemoveOldIntervals(service λ)*

```
l ← level(λ);
/* end all previous chains of level at most l                    */
for l' ← 1 to l do
    if active(l') belongs to a different chain to λ then
        Remove all intervals constructed by active(l');
    active(l') ← NULL;
/* keep track of the new last service of the active chain at
   level l                                                       */
active(l) ← λ;
/* check if λ is a tail service                                  */
if Q_λ = E_λ then
    Remove all investment intervals constructed by λ;
    active(l) ← NULL;
```

4.3 Analysis Overview

Analogous to the analysis used in the deadline case, we will look at the costs of all the different types of services individually: primary services, normal services, upgrade services and tail services. By noting that every service must fall under one of these categories, the total cost of our algorithm will be given by summing up the costs of all these different types of services. To account for delay costs, we will also show that the delay costs of requests are captured in the service costs. To ultimately show that our algorithm is constant competitive, we will once again use a charging argument and charge the costs of these different service types to OPT.

To do so we will first examine the structure and properties of our solution in Section 4.4. This will involve arguing that all delay is paid off when requests are served, proving that non-NULL pointers in a set of triggering requests must all be the same, arguing our chains are linear structures as mentioned earlier and showing that the cost of any service can be bounded. Many of the proofs in this section are rather straightforward but establish important Lemmas that will underpin our analysis. Most importantly, this section will break down the cost of any level l service into 4 components: its *shared cost* σ_l, its *total individual cost* $\delta_l|\lambda|$, its *triggering delay cost* which is the residual delay on all eligible requests that is paid off at the beginning of the service and the *investment cost* which are the delay costs invested in during the service's investment phase. As mentioned in the algorithm description, these 4 components cover all costs of the service. This section will then show that the total individual cost incurred by all

services is at most the investment cost incurred by all services and hence our analysis only needs to consider 3 of the 4 cost components. This leads us to the notion of the *charged cost* of a service which is the shared cost plus the triggering delay cost plus the investment cost. The charged cost represents the costs of a service that are relevant and need to be charged to OPT for our analysis.

Next, Section 4.5 will begin our charging argument. We will show that the charged cost of the tail services can be charged to the charged costs of the primary and upgrade services. The proof for this will be similar to in the deadline case and rely on the fact that our service chains are linear structures and hence every tail service has a corresponding and unique primary or upgrade service in the same chain. This section will also charge the charged cost of the upgrade services to the charged costs of the normal services. This will also be a similar argument to the deadline case as we can charge the charged cost of each upgrade service to the cost of the charged intervals in its witness set which corresponds to the investment costs of the normal services.

Finally, Section 4.6 will finish our charging argument by charging the charged costs of the primary services and normal services to OPT. Since Section 4.5 has shown that the charged costs of the tail and upgrade services can be charged to the primary and normal services, by charging the primary and normal services to OPT, we will effectively be able to charge all costs to OPT. Charging the charged costs of the primary services is straightforward and is simply an adaptation of the disjointness argument used in the deadline case. Charging the charged costs of the normal services to OPT is much more involved and the crux of this analysis. We will show that the charged costs of any normal service can be charged to its investment cost and since the investment costs of all normal services correspond exactly to the costs of all charged investment intervals, we will show that for any charged investment interval, its cost can be charged to OPT. This will be done by considering any service λ^* made by OPT. For any request q, served in λ^*, we can show that the total sum of its charged investment intervals can be charged to the costs λ^* pays for q.

4.4 Analysis: Solution Structure

OBSERVATION 4.1.1. ALG *serves all requests and there is no residual delay remaining at the conclusion of* ALG.

PROOF. We first argue that ALG serves all requests. This is because, by definition, the delay on each request tends to infinity as time tends to infinity so any set of requests will eventually accumulate

enough delay to trigger a service and then have enough invested to trigger the service of the requests. This is because as the delay tends to infinity, ALG will never reach a scenario where it stops making services or stops investing in requests.

Now we show that when ALG concludes, there is no residual delay remaining. Since ALG serves all requests, there can not be an unserved request with residual delay. Hence, it suffices to show that every served request has no residual delay. Consider any level l service made by ALG. We notice that a service can only service eligible requests and cannot serve any requests of level higher than l. However, all requests of level at most l will have their residual delay paid off at the start of the service by construction of the algorithm. Hence, when the algorithm chooses to serve a request, it will not have any residual delay and all delay would have been accounted for and paid off. Hence, when ALG concludes, none of the requests served will have any residual delay and thus ALG has no residual delay. □

Observation 4.1.1 implies that our solution is feasible since all requests are served and when they are served, all delay accumulated on the request has been paid off. Moreover, we know that all costs incurred by ALG are captured within the costs of the services made by ALG since there cannot be any residual delay accumulated that has not been paid off by some service. Therefore, to analyse the cost of ALG, it suffices to bound the costs of ALG's services.

OBSERVATION 4.1.2. *A service's set of triggering requests must be non-empty and all the triggering requests with a non-NULL pointer must have the same pointer.*

PROOF. We first observe that the set of triggering requests must be non-empty. If this was not the case, then by definition there is no level l eligible request with positive residual delay when this service is triggered. Hence, all the requests whose residual delay contributed to triggering this service would have a level less than l and since $\sigma_{l-1} < \sigma_l$, a lower level service should have been triggered instead. Therefore, the level l service should never have been triggered in the first place.

Next we notice that all triggering requests with a non-NULL pointer must point to the same service. Suppose towards a contradiction that this was not the case and that for a level l service λ at time t, there are two triggering requests q_1 and q_2 with pointers λ_1 and λ_2 respectively where the service time t_1 of λ_1 is before the service time t_2 of λ_2, i.e., $t_1 < t_2$. We first notice that $t_1 < t_2 < t$ since in order for q_1 and q_2 to have their pointers set as λ_1 and λ_2 by the time λ is triggered, the service of λ must occur after both λ_1 and λ_2. Since q_1 is a triggering request for λ, this means its level must be l at time t and thus

there must have been some previous service that set its level to l. By definition of pointers, λ_1 was the last service before t in which q_1 was eligible and would have modified its level. Hence, λ_1 must have set its level to be l and thus been a level l service itself. Likewise, λ_2 is also a level l service. Now if we consider the service of λ_2 at time $t_2 > t_1$, q_1 is already level l at this time and must be active since it is eligible for the later service at time $t > t_2$ and therefore it is eligible for λ_2 in which case its pointer should have been modified to become λ_2, a contradiction. □

From Observation 4.1.2, we know that a service pointer will always be uniquely defined and not dependent on which triggering request was chosen to determine the pointer as alluded to in the algorithm description. Next, we will observe that our service chain structures are linear and well defined.

LEMMA 4.1.3. *Every service can only point to at most one other service and be pointed at by at most one other service.*

PROOF. By construction, a service pointer can only be one other service and hence every service can only point to at most one other service. Now we will argue that two services cannot point to the same service λ' of level l. Suppose towards a contradiction that two services λ_1 and λ_2 both point at λ' with λ_1 served at time t_1, before λ_2's service time of t_2. For this to happen, we must have both t_1 and t_2 are after the service time of λ'.

Now consider the triggering requests T_{λ_2} of λ_2 that have non-NULL pointers. We will show that every $q \in T_{\lambda_2}$ is also eligible for λ_1. Since every $q \in T_{\lambda_2}$ has a non-NULL pointer and λ_2 points to λ' then by Observation 4.1.2, we must have that each $q \in T_{\lambda_2}$ in fact points to λ'. Any request $q \in T_{\lambda_2}$ is active at time $t_2 > t_1$ in order to be eligible for λ_2. Moreover, since q has a pointer of λ', it must have been eligible for λ' which is before t_1 which implies it must have arrived before time t_1. Therefore, q has arrived before time t_1 and must be active at $t_2 > t_1$ which implies it will also be active at time t_1.

Secondly, by definition of pointers, λ' is the last service in which every $q \in T_{\lambda_2}$ was eligible. Consequently, λ' is the last service to have modified q's level prior to λ_2 and since λ' is level l it would have made each $q \in T_{\lambda_2}$ level l. Each $q \in T_{\lambda_2}$ will remain level l until the service of λ_2 at time t_2 otherwise its pointer would have been changed. Therefore, we know that each $q \in T_{\lambda_2}$ if of level at most l until time $t_2 > t_1$. By an identical argument, the triggering requests of λ_1 with non-NULL pointers must be level l at the service time of λ_1. Since the triggering requests have a level equal to the service level (prior to the level upgrading), we must have that λ_1 is initially level l (prior to any level upgrading).

Ultimately, we know that every $q \in T_{\lambda_2}$ is active at time t_1, is of level at most l at time t_1 and λ_1 is at least a level l service. This implies that every $q \in T_{\lambda_2}$ should have been eligible for λ_1, after which they should have had their pointers changed to λ_1. Since λ_1 is after λ' then each $q \in T_{\lambda_2}$ can no longer point to λ' and therefore λ_2 could not possibly point to λ'. □

LEMMA 4.1.4. *The individual cost incurred by all services of* ALG *is at most the investment cost incurred by all services of* ALG.

PROOF. A level l service pays an individual cost of δ_l for each item type it serves. By construction, a level l service should only serve an item type i when the level l investment counter for item i reaches δ_l. Therefore, we can charge the individual cost of serving a set of requests on item type i to the cost of a level l investment counter. Since a level l investment counter on item i corresponds exactly to the investments made by level l services in item i, we are thus charging to the investment costs incurred by ALG.

Next, we will show that a particular investment cost made by ALG can not be charged to more than once. By construction, when we add investment costs to a counter, we add it to exactly one counter at one particular level. Hence, multiple counters cannot reflect the same investment cost. Moreover, once a counter reaches δ_l and a service of item i is made, the counter is reset to 0. Hence, the investment costs in item i can only be used in the service of item type i at most once. Any subsequent services of item i on the same level will be charged to new and distinct investment costs and cannot charge to previous investment costs that have already been charged to. We conclude that any of our investment costs cannot be 'double counted' and charged to more than once. Therefore, when we charge all the individual costs to their corresponding investment counters and thus to the investment costs incurred by ALG, we do not charge to any particular investment cost more than once and hence we have the desired result. □

Therefore, to bound the cost of ALG, it suffices to bound the triggering delay, shared and investment cost of each service, or in other words, the charged cost of each service.

OBSERVATION 4.1.5. *The triggering delay costs incurred by a level l service is at most σ_l.*

PROOF. Consider any level l service made by ALG. If the service is not an upgrade service then, the triggering delay cost equals exactly σ_l as this is the condition required to trigger the service. If the service is a level l upgrade service then, the service was triggered because the eligible requests of level

at most $l-1$ had a residual delay of σ_{l-1} but then its level was upgraded and its pool of eligible requests increased to include the level l requests. However, the fact that a level l service was not triggered at this time and instead a level $l-1$ service was triggered implies that the eligible requests of level at most l must have a residual delay less than σ_l. Therefore, in either case, the triggering delay is at most σ_l as required. □

LEMMA 4.1.6. *The charged cost of a level l service is at most $3\sigma_l$.*

PROOF. The charged cost of a level l service as defined earlier is comprised of the triggering delay cost, the shared cost and the investment cost. By Observation 4.1.5, the triggering delay cost is at most σ_l. The shared cost is either σ_l or 0 in the case that no requests are served and hence the shared cost does not need to be paid. Lastly, by construction of the algorithm, the investment phase should stop once the total invested amount reaches σ_l and hence the investment cost of the service is always at most σ_l. Adding these three components up, the total charged cost is at most $3\sigma_l$ as required. □

4.5 Analysis: Charging service costs

LEMMA 4.1.7. *The charged costs of the tail services are at most 3 times the charged costs of the primary and upgrade services.*

PROOF. Consider any tail service of level l. By definition of a tail service, it must end a chain that has reached level l and for this to happen, the chain must have had a level l upgrade service or a primary service, in the case that $l = 1$. The level l upgrade/primary service that corresponds to this tail must have an investment cost of exactly σ_l otherwise the service would have served all eligible requests and the chain should have ended already. Therefore, the level l upgrade/primary service corresponding to this tail has a charged cost of at least σ_l. By Lemma 4.1.6, the tail service's charged cost is at most $3\sigma_l$. Therefore, we can charge the charged cost of each tail service to 3 times the charged cost of its corresponding upgrade/primary service in the same chain. Lastly, we note that each primary or upgrade service can only be charged to by at most one tail service. This is because, by Lemma 4.1.3, our chains are linear structures so each tail service can only end one unique chain and this unique chain will have a level l upgrade or primary service that is unique to the chain. Therefore, the charged cost of each tail service is charged to a unique primary or upgrade service. Summing up across all tail services, we get that their charged costs are at most 3 times the charged costs of the primary and upgrade services. □

LEMMA 4.1.8. *A charged investment interval belonging to the witness set of an upgrade service cannot be removed after the service.*

PROOF. Consider a level l charged interval $[t, \tau]$ created by the level l service λ that belongs to the witness set $W_{\lambda'}$ of a level $l+1$ upgrade service λ' at time t'. Since our algorithm may remove previously created charged intervals if a previous service becomes a tail service, we will argue that after being used to upgrade λ', $[t, \tau]$ can never be removed after time t'.

The first case we consider is if λ and λ' belong to the same chain. In this case, we clearly have that λ is non-tail since λ' continues the chain and hence the interval cannot be removed after time t'. The second case is if λ and λ' belong to different chains. If the chain λ belongs to is of level at least $l+1$ then λ can not be a tail service since it must be succeeded by some level $l+1$ service for the chain to be at least level $l+1$ and hence its charged intervals cannot be removed. The only scenario in which the interval $[t, \tau]$ could be removed is if λ is the last service of an active level l chain. Since λ' was originally a level l service that was upgraded to level $l+1$. For this to happen, the chain λ' is in must have reached level l using a level l upgrade service λ'' at some time before t'. Such a service would have ended all previous chains of level l. This is because any request whose pointer points to a service in a previous chain of level l and is unserved would be eligible for λ'' and have their pointer overwritten, thus making it impossible for the previous chain to be continued. In particular, the chain that λ belongs to would have been ended and hence if λ could become a tail service it should already have become a tail service after the service of λ'' and before time t'. Therefore, if the interval $[t, \tau]$ were to be removed, it should have already been removed before time t' and been unable to be in the witness set of λ' in the first place. Thus any charged interval in the witness set of λ' will never be removed after time t'. □

LEMMA 4.1.9. *A charged investment interval cannot belong to the witness set of more than one upgrade service.*

PROOF. Suppose towards a contradiction that there is a request q_c whose level l charged investment interval $[t, \tau]$ belongs to the witness sets of two upgrade services. Let the two level $l+1$ upgrade services be λ_1 and λ_2 with service times $t_1 < t_2$ respectively.

The first case is at the time t_1, q_c has not been served yet. Since q_c's interval belongs to λ_1's witness set, this would imply by construction that q_c is level l at time t_1. Consequently, q_c would be eligible for λ_1 and hence be upgraded to level $l+1$ after the service. Then at time t_2, since q_c is at least level $l+1$

after time $t_1 < t_2$, none of q_c's intervals can belong to the witness set of λ_2 since the witness set can only consist of charged intervals from level l requests at time t_2.

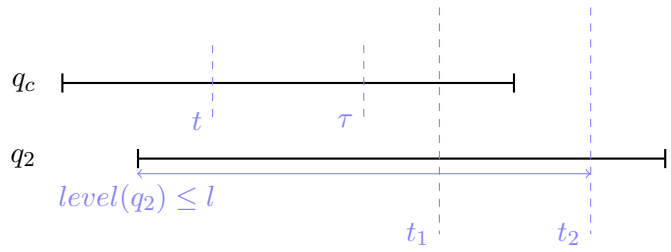


FIGURE 4.3: Illustration of Lemma 4.1.9 Case 2's proof.

The second case is if at time t_1, q_c has already been served. In this case, $t_1 > t$ since the service time of q_c must be at least the start time of its last charged investment interval and hence at least t. Now consider λ_2. In order for $[t, \tau]$ to belong to its witness set, a request q_2 that arrived before t must be eligible for λ_2. The fact that q_2 is eligible for λ_2 at time t_2 implies that it is at most level l and active for all times prior to t_2. Moreover, the fact that q_2 arrives before t implies that it also arrives before $t_1 > t$. Therefore, at time t_1, q_2 has arrived, is active and is of level at most l which makes it eligible for λ_1. After λ_1, q_2 would have its level set to $l + 1$ and hence it would no longer be eligible for λ_2 when λ_2 is initially triggered prior to upgrading. Since this argument applies to all possible q_2 that arrived before t, it follows that $[t, \tau]$ could never have belonged to the witness set of λ_2 as there can not be a request eligible for λ_2 that arrives before the interval begins. Figure 4.3 is a visual aid for the proof of this case.

Therefore, in both cases we have contradictions and hence there cannot be a charged investment interval belonging to the witness set of two upgrade services, as required.

□

LEMMA 4.1.10. *The charged costs of the upgrade services are at most 3 times the charged costs of the normal services.*

PROOF. Consider any level l upgrade service at time t. By construction of the algorithm, it was made since the total cost of the charged investment intervals in its witness set is at least σ_l. By Lemma 4.1.6, this level l upgrade service at time t has a charged cost of at most $3\sigma_l$ and hence we can charge the charged cost of this upgrade service to 3 times the cost of the level $l-1$ charged investment intervals in its witness set. Using Lemma 4.1.9, we know that each charged investment interval can only be in the witness set of at most one upgrade service and hence can only be charged to once by any upgrade

service. Therefore, charging all upgrade services to their witness sets, we get that their total charged cost is at most 3 times the cost of all the charged investment intervals. Lastly, we note that the costs of the charged investment intervals correspond to exactly the investment costs made by normal services. Moreover, Lemma 4.1.8 tells us that the charged intervals we charge to cannot be removed and hence will always exist and correspond to the investment cost of some normal service as opposed to some service that later becomes a tail. Since the investment costs of the normal services are at most the charged costs of the normal services it follows that the cost of all the charged investment intervals is at most the charged cost of all the normal services and hence we get the desired result. □

4.6 Analysis: Charging to OPT

LEMMA 4.1.11. *The charged cost of the primary services is at most 3 times the cost of* OPT.

PROOF. Consider any primary service made by ALG at time t and consider its set of triggering requests. Let the earliest arriving triggering request have arrival time a_q. We will define the time interval $[a_q, t]$ for this primary service. These intervals defined for all primary services of ALG must be disjoint. Suppose for a contradiction that this was not the case then there are time intervals $[a_q, t]$ and $[a'_q, t']$ corresponding to primary services that are not disjoint. We will assume wlog that the primary service at time t is served first and hence $t \leq t'$. Since the intervals are not disjoint then $a'_q \leq t \leq t'$. At time t' when the corresponding primary service is triggered, its earliest arriving triggering request which by definition arrives at time a'_q would have been eligible for the earlier primary service at time $t \leq t'$. After which, it would have its pointer changed and no longer have a non-NULL pointer. Therefore, the service at t' could not be a primary service which is a contradiction.

These disjoint time intervals correspond to distinct primary services and using Lemma 4.1.6 alongside the fact that primary services are level 1, we know the charged cost of each corresponding primary service is at most $3\sigma_1$. If OPT decides to make a service within one of these intervals, it must pay at least σ_1 since OPT must serve at some level l in which case it will always pay the fixed shared cost of $\sigma_l \geq \sigma_1$ for all l. Therefore, we can charge the charged cost of the primary service corresponding to the interval to 3 times the cost of this service made by OPT. We apply this argument to each primary service that has an OPT service within its interval. By disjointness of the intervals, when we charge all primary services to the corresponding OPT services within their intervals, no OPT service can intersect with more than one interval and hence be charged to more than once.

Lastly, we consider any interval within which OPT decides to not make any service. By construction, the triggering requests for the corresponding primary service would have accumulated a delay of σ_1 within this time interval. OPT would have to pay this delay cost since it has not served any of the triggering requests within the interval and could not have served any of the triggering requests prior to this time interval as none of them would have arrived yet. Hence, in this case, we can charge the charged cost of the primary service corresponding to this interval to 3 times the delay cost paid by OPT for the triggering requests of this primary service. Moreover, since the triggering requests for each primary service are unique, each primary service under this case will be charging to delay costs corresponding to unique requests. Therefore when we charge all primary services that fall under this case to the delay costs incurred by OPT, we will never charge to a delay cost on a particular request more than once.

Overall, all charged primary service costs can be charged to distinct service or delay costs incurred by OPT, hence achieving the desired result. □

LEMMA 4.1.12. *If a request has a level l charged investment interval $[t, \tau]$ then there cannot be a level $l' \geq l$ service triggered between times t and τ (inclusive) other than the service that created $[t, \tau]$ at time t.*

PROOF. By definition of $[t, \tau]$ being a charged investment interval, the service that created it at time t must be a normal service λ. By definition of normal services, they must have a subsequent service in their service chain. This is only possible if at time t, λ left some requests unserved which would then have their pointers set to λ and later have become the triggering requests for a service λ_s that would directly succeed λ in its chain. Let these requests form the set T_{λ_s}. Since λ pays off the delay on eligible requests until time τ by construction, the requests in T_{λ_s} cannot accumulate any residual delay until after time τ. Therefore, λ_s must be after time τ.

Now suppose for a contradiction that there are level $l' \geq l$ services between t and τ that are not λ. Let the earliest of these services be λ', occurring at time t' where $t \leq t' \leq \tau$ and of level l'. Consider any request $q \in T_{\lambda_s}$. We will show that q is eligible for λ'. By definition, q must have a pointer of λ until λ_s is triggered in order for λ_s to succeed λ in its chain. By definition of pointers being the last service to modify a request's level, λ is the last service to modify q's level and would have set it to be l. This level cannot be changed until λ_s is served after time τ otherwise, q's pointer would have been overwritten. Therefore, at time $t' \leq \tau$, we know q is of level at most l. Next, q must have arrived before t to have its pointer set to λ and it must remain active until the service time of λ_s after τ. In particular, it must be

active at time $t' \leq \tau$. Therefore, at time t', q is active and of level at most l and hence it will be eligible for the level $l' \geq l$ service λ'. After this service, the pointer of q would have been overwritten. Since this argument applies to all $q \in T_{\lambda_s}$, any request with a pointer of λ that could cause a service after time τ to directly succeed λ in its chain would not be able to do so as their pointers are overwritten. Moreover, the service to succeed λ in its service chain could not occur before time τ as mentioned earlier so ultimately, there is no way for a service to succeed λ. This contradicts the fact that λ is a normal service and should have some service continuing its chain. □

LEMMA 4.1.13. *The charged cost of a normal service is at most 3 times its investment cost.*

PROOF. Consider any level l normal service made by ALG. Its investment cost is exactly σ_l. This is because if it were less, this would imply that the service served all eligible requests in which case it would become a tail service instead of a normal service since there are no unserved requests that can continue the service chain. Using Lemma 4.1.6, we know that the charged cost of this normal service is at most $3\sigma_l$ and since the investment cost is exactly σ_l, this gives us the desired result. □

Using Lemma 4.1.13, the charged cost of any normal service is at most 3 times its investment cost. Hence, to bound the charged costs of all normal services, it suffices to only look at the investment costs incurred by the normal services. By definition, the investment costs incurred by the normal services correspond to exactly the cost of all the charged investment intervals constructed for our requests. Hence, we will show that the cost of any charged investment interval belonging to any request can be charged to some distinct cost incurred by OPT.

LEMMA 4.1.14. *For any request, its charged investment intervals are all disjoint.*

PROOF. Assume towards a contradiction there is a request with two non-disjoint charged investment intervals $[t_1, \tau_1]$ of level l_1 and $[t_2, \tau_2]$ of level l_2 where wlog $t_1 < t_2 < \tau_1$. We first notice that $l_2 \geq l_1$ since the level of a request can never decrease and after the service at time t_1, this request would be a level of at least l_1. Then using Lemma 4.1.12, we know there can be no service with level at least l_1 between times t_1 and τ_1 which contradicts our assumption. Hence, this shows our required result. □

LEMMA 4.1.15. *Consider any two level l charged investment intervals that intersect. They must be created by the same level l normal service.*

PROOF. Suppose this was not the case and that we have two intersecting level l charged investment intervals $[t_1, \tau_1]$, $[t_2, \tau_2]$ created by different services where wlog $t_1 \leq t_2 \leq \tau_1$. This would imply that we have a level l normal service at time t_1 and another normal level l service at time $t_2 \geq t_1$. However, using Lemma 4.1.12, we know that there cannot be another level l service between times t_1 and τ_1 inclusive and hence the service at time t_2 should not have occurred, a contradiction. □

OBSERVATION 4.1.16. OPT *must serve all requests.*

PROOF. Suppose for a contradiction OPT left some request q unserved. By definition of q's delay function, it is a function that tends to infinity as time tends to infinity. Hence, OPT would incur a delay cost of infinity on q which is clearly sub-optimal. □

LEMMA 4.1.17. *The total cost of the charged investment intervals of all requests is at most 6 times the cost of* OPT.

PROOF. Consider a service λ^* made by OPT at level l^* and time t^* and let the set of requests served by λ^* be Q_{λ^*}. We will define the cost of λ^* to be the shared cost, the total individual cost as well as the delay penalty paid on each $q \in Q_{\lambda^*}$. We remark that OPT does not use any notion of investing, so the delay penalty paid for each request is the total delay accumulated from arrival time to service time. For the requests $q \in Q_{\lambda^*}$ we will prove three claims: the total cost of the charged investment intervals ending before t^* can be charged to the delay costs of λ^*, the total cost of the charged investment intervals intersecting with time t^* can be charged to 2 times the shared and individual cost of λ^* and the total cost of the charged investment intervals beginning after time t^* can be charged to 4 times the shared and individual cost of λ^*. Overall, this lets us charge the total cost of all charged investment intervals belonging to requests $q \in Q_{\lambda^*}$ to 6 times the cost of λ^*. Figure 4.4 illustrates this charging argument. By Observation 4.1.16, every request must be in some OPT service. Hence, applying this argument across all services made by OPT, we will be able to charge the cost of all charged investment intervals to 6 times the cost of OPT.

Therefore, proving the three claims will also prove our Lemma. The first claim is rather straightforward while the other two are much more involved and are the crux of our analysis.

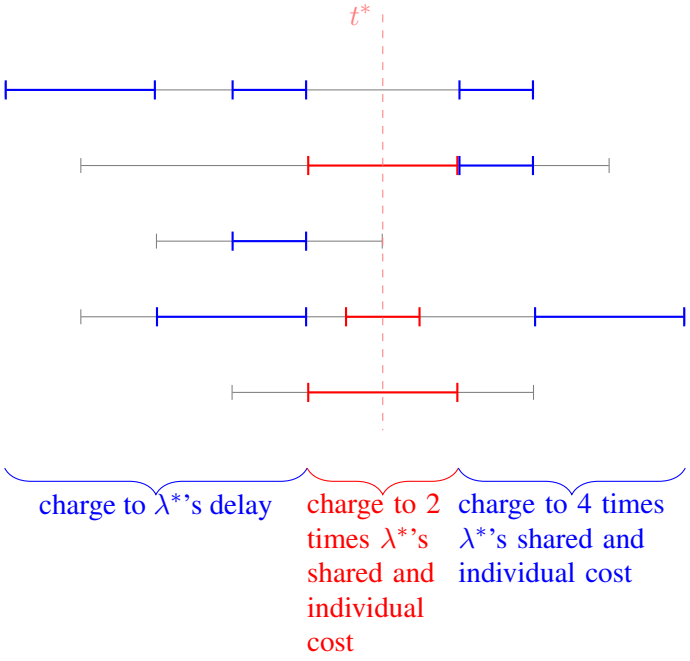


FIGURE 4.4: Illustration of Lemma 4.1.17's charging argument.

CLAIM 4.1.18. *The total cost of the charged investment intervals ending before time t^* and belonging to requests in Q_{λ^*} is at most the delay cost of λ^*.*

Proof: Consider any $q \in Q_{\lambda^*}$ that has a charged investment interval ending before t^*. By construction of our algorithm, the cost of any of q's charged investment interval corresponds exactly to the residual delay accumulated by q within the interval. Moreover, by Lemma 4.1.14, q's charged investment intervals are all disjoint so there is no delay incurred by q that can contribute to two different charged investment intervals. Therefore, the total cost of all of q's charged investment intervals ending before time t^* is at most the delay accumulated by q from arrival up to time t^*. This is exactly the delay cost paid by λ^* for the request q and hence can be charged to the delay cost. For each $q \in Q_{\lambda^*}$, λ^* must pay a unique delay cost and hence when we apply this charging argument for all $q \in Q_{\lambda^*}$, we will always be charging to different and unique delay costs incurred by λ^*. Therefore, no delay cost incurred by λ^* is charged to twice and the total cost of all charged investment intervals ending before time t^* is at most the delay cost of λ^*. ■

CLAIM 4.1.19. *The total cost of the charged investment intervals that intersect with time t^* and belong to requests in Q_{λ^*} is at most 2 times the shared and individual cost of λ^*.*

Proof: For the charged investment intervals $[t, \tau]$ which intersect with time t^*, we have that $t \leq t^* \leq \tau$. To prove this claim, we will look at two cases, the intervals of level $l < l^*$ and the intervals of level $l \geq l^*$. We will show that the intervals under the first case cost at most the shared cost of λ^* and the intervals under the second case cost at most two times the total individual cost of λ^*.

Consider the intervals of a fixed level $l < l^*$. Since all these intervals must intersect at time t^* then we know they must all have been created by the same level l normal service λ using Lemma 4.1.15. The total cost of all these investment intervals must correspond to exactly the investment cost of λ which by construction of the algorithm is at most σ_l. Summing up across all levels $l < l^*$, we get the total charged interval cost in this case is at most $\sum_{l=1}^{l^*-1} \sigma_l$. Since $\sigma_{i+1} \geq 2\sigma_i$ for all $i < n$, we get the geometric sum $\sum_{l=1}^{l^*-1} \sigma_l \leq \sigma_{l^*} \sum_{l=1}^{l^*-1} (\frac{1}{2})^l$ which can be bounded by its limiting sum to get $\sum_{l=1}^{l^*-1} \sigma_l \leq \sigma_{l^*}$. Since λ^* is a level l^* service it must pay a fixed shared cost of σ_{l^*} so we can charge the cost of the investment intervals that fall under this case to the shared cost paid by λ^*.

Now consider the intervals of a fixed level $l \geq l^*$ and for a fixed item type i. By Lemma 4.1.15, all the different item i intervals falling under this case must have been created by the same level l service λ. Hence, their corresponding item i requests must have all been eligible for λ. By construction, λ will only be able to invest at most δ_l into the level l investment counter for item i before serving all these eligible item i requests, and hence the total sum of all these level l, item i investment intervals is at most δ_l. Summing up across all levels $l \geq l^*$, we get that the total charged interval cost incurred by item i is at most $\sum_{l=l^*}^{l=n} \delta_l$. Once more, we use the fact that $\delta_i \geq 2\delta_{i+1}$ for all $i < n$ to get the geometric sum $\sum_{l=l^*}^{l=n} \delta_l \leq \delta_{l^*} \sum_{l=l^*}^{l=n} (\frac{1}{2})^{l-l^*}$ that can be bounded by its limiting sum to get $\sum_{l=l^*}^{l=n} \delta_l \leq 2\delta_{l^*}$. Since λ^* must serve this item i, it must pay a unique cost of δ_{l^*} for this item type i. Hence, we can charge the cost of the investment intervals on item i and of level $l \geq l^*$ to twice the cost paid by λ^* for item i. We can apply this charging argument across all item types and since each item type's charged intervals gets charged to an individual cost unique to that item type, no individual cost gets charged to more than once. Hence, the total cost of all investment intervals that fall under this case can be charged to two times the total individual cost of λ^*.

Combining the two cases, we get that all charged investment intervals that intersect with time t^* can have their costs charged to the shared cost plus 2 times the individual cost of λ^* which is at most 2 times the shared and individual cost of λ^*. ■

CLAIM 4.1.20. *The total cost of the charged investment intervals that begin after time t^* and belong to requests in Q_{λ^*} is at most 4 times the shared and individual cost of λ^*.*

Proof: We will again look at two cases: the intervals of level $l < l^*$ and those of level $l \geq l^*$. We will show that the intervals under the first case have a total cost of at most 4 times the shared cost of λ^* and the intervals under the second case have a total cost of at most 2 times the individual cost of λ^*.

Consider the intervals with level $l < l^*$. Consider any fixed level $l < l^*$ and consider all the investment intervals of this level that begin after time t^*. Let their corresponding requests be the set Q_l. Now let time t be the latest start time from all these level l intervals and let the request that contains this interval be q (if multiple requests contain this interval starting at t, we pick an arbitrary one to be q). Let the intervals that begin before time t (but still after t^*) form the set $I_{<t}$ and the intervals that begin at time t be I_t. By definition, no intervals can begin after time t and these two sets cover all intervals beginning after time t^*.

First, we argue that at time t, all requests in Q_l must have level at most l. By definition, all requests in Q_l have a level l interval beginning after time t^* which is only possible if they are all eligible for some level l service beginning after time t^*. This implies that at time t^*, each request in Q_l is of level at most l otherwise it could not have been eligible for a level l service after time t^*. Now suppose for a contradiction that not all requests in Q_l have level at most l at time t. Since all requests have level at most l at time t^*, it follows that there must have been a level $l' > l$ service at some time t' where $t^* \leq t' \leq t$. We will show that q is eligible for this service. For q to contain a level l interval starting at time t, it must be eligible for a level l service at time t. This implies that q is active and of level at most l until time t. q must have arrived before time $t^* \leq t'$ to have been served by λ^*. Therefore, we conclude that at time $t' \leq t$, q has arrived, is active and is of level at most l which would make it eligible for this service at time t'. After this service, its level would have been upgraded to $l' > l$ which means that q could never have been eligible for the level l service at time t that constructs its interval, a contradiction.

Now we will show that the cost of the intervals in $I_{<t}$ is at most σ_{l+1}. Any request corresponding to one of these intervals is of level at most l at time t as argued earlier. Moreover, any request corresponding to one of these intervals is of level at least l. This is because, by definition of $I_{<t}$, it has a level l interval at some time $t' < t$ which implies it was eligible for a level l service at time $t' < t$ after which it would be of level at least l. Therefore, the requests corresponding to the intervals in $I_{<t}$ are exactly level l at time t. Now, for q to have an interval starting at time t, it must have been eligible for a normal service λ at time t. By construction of the algorithm, the witness set of λ must have a total cost less than σ_{l+1} since λ creates a level l investment interval at time t which implies it did not upgrade and stayed at level l. We also notice that, for q to be served by λ^* at time t^*, it must have arrived at time $a_q \leq t^*$. Then

the witness set of λ is by definition, at least all the level l charged intervals beginning after $a_q \leq t^*$ and belonging to requests which are currently level l at time t. Hence, $I_{<t}$ is a subset of the witness set since all the intervals in $I_{<t}$ begin after time $t^* \geq a_q$ and belong to requests that are currently level l. Therefore, it follows that the cost of the intervals in $I_{<t}$ is at most σ_{l+1}.

Now we argue that the cost of the intervals in I_t is at most σ_l. The level l investment intervals beginning at time t must all be created by the same level l service λ since Lemma 4.1.12 implies that there cannot be another level l service at time t. The total cost of the charged intervals that λ creates is at most the investment cost of λ which is at most σ_l by construction. Therefore the cost of the intervals in I_t is at most σ_l.

We conclude that since all intervals of a fixed level l and beginning after time t^* must belong to either $I_{<t}$ or I_t, the total cost of all such intervals is at most $\sigma_{l+1} + \sigma_l < 2\sigma_{l+1}$ since $\sigma_l < \sigma_{l+1}$. Then applying this argument across all levels $l < l^*$ and summing up, we get the total charged investment interval cost in this case is at most $\sum_{l=1}^{l^*-1} 2\sigma_{l+1} \leq 2\sigma_{l^*} \sum_{l=1}^{l^*-1} (\frac{1}{2})^{l-1} \leq 4\sigma_{l^*}$, once more using the fact that $\sigma_{i+1} \geq 2\sigma_i$ for all $i < n$ to get a geometric sum that can be bounded by its limiting sum. Since λ^* must pay the fixed shared cost of σ_{l^*} to serve its requests, we can charge the cost of all the intervals under this case to 4 times the shared cost of λ^*.

Now we consider the second case: the charged investment intervals that begin after time t^* and are of level $l \geq l^*$. Consider a fixed level $l \geq l^*$ and a fixed item type i and consider all the charged investment intervals that begin after t^*, are of level l and their corresponding request is of item type i.

We will argue that any two of these charged investment intervals must have their costs contributing to the 'same' level l investment counter for item i. Every time an investment counter is reset to 0, we will consider this as a different investment counter starting. Hence intervals contribute to the same investment counter if the counter does not get reset between contributions. Suppose towards a contradiction that we have two level l intervals $[t_1, \tau_1]$ and $[t_2, \tau_2]$ that contribute to different level l investment counters. We can assume wlog that $t_1 \leq t_2$. Let the item i requests that contain these intervals be q_1 and q_2 respectively. This can only happen if the counter was reset between times t_1 and t_2 which by construction of the algorithm only happens if item i was served in a level l service between these times. First we notice that since q_2 is served by λ^* it must have arrived before $t^* < t_1$. Moreover, since it has a level l charged interval at time t_2, this would imply it is eligible for a level l normal service at time t_2 which means it must be active and of level at most l until t_2. Therefore, q_2 will have arrived, be active and be eligible

for any service of level at least l between times t_1 and t_2. Therefore, if item i is served at level l at some time between t_1 and t_2, q_2 would have been eligible for this service and would have been served as well by construction of the algorithm. This makes it impossible for q_2 to have been eligible for a level l service at time t_2, which contradicts the fact that it has a level l interval starting at time t_2.

Now by construction, the investment cost of any level l charged investment interval is always added to some level l investment counter. We have just argued that all the level l intervals on item i in this case contribute to the same counter, so none of these level l intervals on item i in this case can accumulate investment costs in different counters. Therefore, the sum of the costs of all level l charged investment intervals on item i in this case is at most the maximum value of any level l investment counter on item i. By construction of the algorithm, this is always at most δ_l otherwise ALG would have served the requests on item i. Keeping the item i fixed and summing up across all levels $l \geq l^*$, we get that the total charged interval cost incurred by requests on item i in this case is at most $\sum_{l=l^*}^{l=n} \delta_l \leq \delta_{l^*} \sum_{l=l^*}^{l=n} (\frac{1}{2})^{l-l^*} \leq 2\delta_{l^*}$, once more using the fact that $\delta_i \geq 2\delta_{i+1}$ to get a geometric sum and bounding it by its limiting sum.

Therefore, for a particular item type i, the total cost of its charged investment intervals which occur after time t^* and are of level $l \geq l^*$ is at most $2\delta_{l^*}$. Since λ^* must serve this item i, it must pay a unique cost of δ_{l^*} for this item type i so we can charge this cost to 2 times the cost paid by λ^* for item i. Considering this across all item types, we get that the total cost of all investment intervals that occur after t^* and are of level $l \geq l^*$ can be charged to 2 times the total individual cost of λ^*. This is because for each item type i, its charged intervals are being charged to a distinct individual cost payment made by λ^* for item i alone and hence no individual cost will be charged to more than once.

Overall, in this case, we get that the total cost of all charged investment intervals that start after time t^* is at most 4 times the shared cost of λ^* plus 2 times the individual cost of λ^* which is at most 4 times the shared and individual cost of λ^*. ■

Therefore, combining our three claims, we get that the total cost of all the charged intervals belonging to requests in Q_{λ^*} is at most the delay cost of λ^* plus 2 times the shared and individual cost of λ^* plus 4 times the shared and individual cost of λ^*. Overall, this is at most 6 times the cost of λ^*. As mentioned earlier, we consider this argument across all services made by OPT and our Lemma easily follows. □

LEMMA 4.1.21. *The charged cost of the normal services is at most 18 times the cost of* OPT.

PROOF. The charged cost of the normal services is at most 3 times the cost of the investment costs made by normal services as shown in Lemma 4.1.13. However, the investment costs made by normal services correspond exactly to the total cost of all the charged investment intervals. By Lemma 4.1.17, the total cost of all charged investment intervals is at most 6 times the cost of OPT. Therefore, it easily follows that the charged cost of all the normal services is at most 18 times OPT. □

Finally, we can prove our Theorem

THEOREM 4.1. *There exists an $O(1)$-competitive, deterministic algorithm for Piecewise Cardinality JRP with Delay.*

PROOF. By Observation 4.1.1, we know that ALG serves all requests and has no residual delay at the conclusion of ALG. Therefore, we know ALG is a feasible solution. Since the algorithm uses a notion of investment, the delay costs are incorporated into the service costs of ALG. Since ALG finishes with no residual delay, this would mean that all delay costs are completely incorporated within the service costs so to bound the cost of ALG, it suffices to bound the costs of the services made by ALG.

We know the services made by ALG incur a shared cost, individual cost, triggering delay cost and investment cost and Lemma 4.1.4 lets us bound the total individual cost incurred by the total investment cost incurred. Therefore, the cost of ALG is at most 2 times its total shared cost plus total triggering delay cost plus total investment cost, or in other words, 2 times its total charged service cost.

The total charged service costs can be broken up into the charged service costs of all the different types of services that can occur. These are the charged primary service costs P, the charged upgrade service costs U, the charged tail service costs T and the charged normal service costs N. This gives us

$$\begin{aligned}
\mathrm{ALG} &\leq 2(P+U+T+N) \\
&\leq 2(4P+4U+N) && (\text{ Lemma 4.1.7}) \\
&\leq 2(4P+13N) && (\text{ Lemma 4.1.10}) \\
&\leq 24\,\mathrm{OPT} + 26N && (\text{ Lemma 4.1.11}) \\
&\leq 492\,\mathrm{OPT} && (\text{ Lemma 4.1.21})
\end{aligned}$$

Hence, we get that $\mathrm{ALG} \leq O(1) \times \mathrm{OPT}$ as desired. □

4.7 Time Complexity Considerations

As discussed previously, online algorithms are only concerned with the competitive ratio bound. However, we will show that with some extra assumptions and minor modifications to our algorithm, it can be made to run in polynomial time. Here, polynomial time refers to being polynomial in the total number of requests received N.

Restricting the number of services. We first remark that our algorithm can be modified to make at most N services. To achieve this, whenever we make a level l service that does not serve any requests, we will arbitrarily serve any of the eligible requests at the end of the service. This will lead to the service incurring a shared cost of σ_l as well as an individual cost of $\delta_l < \sigma_l$ that can be charged to the shared cost. Overall, the additional individual costs incurred due to this new requirement can be charged to the shared costs and hence the costs of our services are still bound by their charged costs. Thus, our analysis bounding the charged costs and hence the entire cost of the algorithm still applies despite this new restriction. With this modification, it easily follows that we will have at most N services since each service must serve at least one request. Hence, knowing that the number of services we will make is polynomial in N, it remains to show that making these services also takes polynomial time.

Our delay assumption. Recall that each request q has an associated delay function $d_q(t)$ that describes the delay penalty q has accumulated up till time t. We will assume that for any request, its delay function and its inverse can be efficiently computed for any time. That is, for any request q and time t, we can efficiently compute the delay $d_q(t)$ and for any delay amount D, we can efficiently compute the time t such that $d_q(t) = D$. Here, efficient refers to being a polynomial time computation. Furthermore, we will assume that the sum of multiple requests' delay penalties and their inverse can also be computed efficiently. By extension, this will also mean that the residual delay on requests and the sum of residual delays can be computed efficiently and their inverses can also be computed efficiently.

Calculating service times. With this delay assumption, we can now show that it takes a polynomial time to figure out when our services should be triggered. Our algorithm states that a level l service should be triggered when requests of level at most l accumulate σ_l residual delay. The naive implementation of this condition would be to continuously check the residual delay at each level at each time. This may lead to a non-polynomial algorithm if the number of times we check is large enough. A polynomial time implementation would be to pre-compute the expected service time at each level. The number

of levels is at most the number of requests and hence we only need to make a polynomial number of computations. For each level l, we take the sum of the residual delay functions of all requests of level at most l and compute when this sum would equal to σ_l. By our delay assumption, this can be done efficiently. Therefore computing the service times takes polynomial time and these service times only need to be recomputed when a new request arrives or after any service has been made since these are the only times at which the sum of the residual delay functions at each level will change. Since the number of times a request can arrive is N and the number of services that can be made is at most N, we only need to recalculate the service times a polynomial number of times. In conclusion, figuring out the times of our services takes a polynomial time.

Thus, we can figure out when to trigger our services in a polynomial time. It remains to show that each time we trigger a service, it takes polynomial time. Knowing that we make at most N services, if each one takes polynomial time then overall, all aspects of our algorithm will take polynomial time as desired. To bound the time it takes to make a service, we will refer to our pseudocode from before to analyse the time taken for each step of the service.

Initialisation time (Algorithm 4.5). Our initialisation procedure takes polynomial time since we iterate over the levels and number of items, of which there are at most N and initialise counters in constant time. When requests arrive, we initialise their level and pointers which overall takes a polynomial time since the number of requests that can arrive is at most N and the initialisation process is clearly a constant time operation.

Service triggering and upgrading time (Algorithm 4.6). Figuring out the triggering requests set and the pointer takes polynomial time as we simply iterate over all requests, of which there are at most N, and determine whether they are a triggering request and whether they have a pointer which takes constant time. Creating the normal service flag clearly takes constant time. To determine the witness set, we iterate over every charged interval and determine in constant time whether they satisfy the conditions to belong in the witness set. We have at most N requests, and for each request, it can have at most N charged intervals since these intervals must be created by normal services of which we have at most N. Since we are iterating over a polynomial number of charged intervals and performing constant time work on each interval, determining the witness set takes polynomial time overall. Lastly, taking the sum of the intervals in the witness set also takes polynomial time since we have at most a polynomial number of charged intervals in the set. Ultimately, this step of the service takes polynomial time.

Making the service (Algorithm 4.7). Our service begins at time t by paying off the residual delay on all eligible requests which takes polynomial time as we iterate over all eligible requests, of which there are at most N, and compute and pay off their residual delay which by assumption can be done efficiently. Updating the pointers, levels and creating charged intervals after the investment phase takes polynomial time since we iterate over at most N requests and do constant time operations. We will now explain how the investment phase can be implemented in polynomial time.

Instead of continually increasing τ which may take a non-polynomial time if we consider too many times, we will directly compute the time at which we would have invested a total amount of σ_l in our requests. That is, we will take the sum of the residual delay functions of every request in $E_\lambda \setminus Q_\lambda$ and compute the time τ at which this sum equals $\sigma_l - invested$. Recall that $invested$ refers to the total amount invested by our service so far. Then for each request $q \in E_\lambda \setminus Q_\lambda$ on item i, the amount invested into q is $counter(l,i) + r_q(t,\tau) - \delta_l$ if $counter(l,i) + r_q(t,\tau) > \delta_l$, otherwise, it is $r_q(t,\tau)$. That is, the invested amount is the residual delay accumulated by q up till time τ unless this residual delay causes the investment counter $counter(l,i)$ to exceed δ_l in which case we cap the investment amount at the amount required to make the counter reach δ_l exactly. All other steps remain the same. That is, once we know the amount to be invested in q, we add this amount to the q's item type's counter, the service investment counter as well as the counter for how much has been invested in q. We then check if the counter has exceeded δ_l in which case we serve all eligible requests of item i which will require iterating over at most N eligible requests. Overall, for any request $q \in E_\lambda \setminus Q_\lambda$, its investment process takes polynomial time since we compute the amount to be invested which can be done efficiently by our assumption then add this amount to counters in constant time before finally serving requests if required which takes polynomial time. Since we do this for all requests in $E_\lambda \setminus Q_\lambda$, of which we have at most N, the entire process of investing once we have computed τ takes polynomial time.

The last thing we must consider is how many times τ must be calculated and hence how many times we must repeat this investment process. Since τ is directly calculated as the time at which the invested amount should be σ_l, we only need to recalculate τ if the amount invested in a request is less than calculated, or in other words, it was capped by taking the $\min$ since it caused its investment counter to exceed δ_l. Each time this happens, we know that an item type will be served and hence the maximum number of times τ needs to be computed is equal to the number of items we can serve which is at most N. Overall, we calculate τ at most N times and each time we calculate it, we perform a polynomial time investment process so the entire investment phase takes polynomial time.

This modified process of performing the investments does not affect our analysis as our investment intervals can still be correctly constructed with costs corresponding to the amounts invested within them. Moreover, τ reflects the time at which we have paid off all residual delays until and will still accurately reflect this under our new procedure. The only case in which this might not be the case is when our service serves all eligible requests and hence τ overshoots when the actual time we invested up to should be. However, in this case, we would have a tail service, in which case τ is not used in the analysis anyways.

Removing old intervals (Algorithm 4.8). This step involves considering the last service of the active chain at each level, of which there are at most N. For each service, they can have at most N charged intervals since they can only create at most one interval per request. Hence, iterating over the last service at each level and removing their charged intervals if required takes polynomial time since we will be iterating over at most N^2 intervals and removing an interval takes constant time. Finally, checking if our new service is a tail and removing its intervals if so takes polynomial time since we iterate over at most N intervals.

Reduction to Piecewise Cardinality JRP. Since our online algorithm for Piecewise Cardinality JRP with Delay can be made to run in polynomial time, it remains to show that our reduction from Cardinality JRP to Piecewise Cardinality JRP is a polynomial reduction. If we assume the cost function and its gradient for Cardinality JRP can be efficiently computed at any integral value, then our algorithm for Cardinality JRP with Delay also runs in polynomial time. This is because the reduction from Cardinality JRP only requires approximating the cost function with a piecewise affine function and this can be done in polynomial time as shown by Guha et al. (2001). This involves first efficiently evaluating the original cost function and its gradient for each integral value in $[1, n]$ and using these values to construct n tangents which will form our initial affine approximation. This takes $n \leq N$ polynomial time computations. We can then iterate over the resulting $n \leq N$ affine pieces $\sigma_i + \delta_i|\lambda|$ in ascending order of i and adjust them in constant time by increasing σ_i or decreasing δ_i such that they satisfy $2\sigma_i \leq \sigma_{i+1}$ and $\delta_i \geq 2\delta_{i+1}$ for all $i \in [1, n)$. Overall, this takes polynomial time as required.

CHAPTER 5

Weighted Cardinality JRP

Weighted Cardinality JRP is an extension of Cardinality JRP where each item type i has an associated weight w_i and the cost of serving a set of items is now a function of their total weight as opposed to a function of the number of items. That is, the cost of serving λ is $f(\sum_{i \in \lambda} w_i)$ where f is a non-decreasing concave function. We also remark that the weights are positive integers. This is because as mentioned previously, $f(0) = 0$ since the cost of serving no items should trivially be 0 and hence any items with a weight of 0 can be served for free. We will show that a reduction to Cardinality JRP gives a constant competitive solution to Weighted Cardinality JRP. We will be looking at Weighted Cardinality JRP under the delay model since results under delay also apply to the deadline variant.

5.1 Algorithm Description

We will design an algorithm that satisfies

THEOREM 1.2. *There exists an $O(1)$-competitive, deterministic algorithm for Weighted Cardinality JRP with Delay.*

Given an instance of Weighted Cardinality JRP (WJRP), we construct an instance of Cardinality JRP (CJRP) as follows. If WJRP has cost function f then CJRP will also have cost function f. For each possible item type i with weight w_i in WJRP, CJRP will have the *surrogate item types* $i_1, i_2, \ldots, i_{w_i}$. In this way, every item type in WJRP will have w_i surrogate item types in CJRP. Then, any time a request for item type i arrives in the WJRP instance, we will release a request for each of the w_i surrogate item types in the CJRP instance. More precisely, consider any request q in the WJRP instance on item i arriving at time a_q, with weight w_i and delay function $d_q(t)$. In our CJRP instance, we will release the requests $q_1, q_2, \ldots, q_{w_i}$ on the item types $i_1, i_2, \ldots i_{w_i}$ respectively. These requests will all arrive at time

a_q and have delay function $d_q(t)/w_i$. We will call the requests $\{q_1, \ldots, q_{w_i}\}$ the *surrogate requests* of the original request q.

We will then use our algorithm for Cardinality JRP (which is in fact our algorithm for Piecewise Cardinality JRP) to produce the solution ALG_g. To turn this into a solution ALG_w for WJRP, we make some simple modifications. Any time ALG_g makes a service, we will also make a service in ALG_w. If the ALG_g service serves the item type i_j then the corresponding ALG_w service will serve the item type i.

5.2 Analysis

We will refer to the optimal solution for the WJRP instance as OPT_w and the optimal solution for the CJRP instance constructed as OPT_g.

Analysis Overview. We will first show that any service in ALG_g that serves a request q_j must in fact serve all the surrogate requests $\{q_1, \ldots, q_{w_i}\}$ of q. This will be the crux of our analysis, from which our other Lemmas will easily follow. As a consequence, we can then argue that the costs of the solutions ALG_g and ALG_w are the same. We will then show that the cost of OPT_g is at most the cost of OPT_w. This is done by transforming OPT_w into a solution for CJRP with the same cost then utilising the optimality of OPT_g. Finally, combining these facts with the fact that ALG_g is constant competitive for Cardinality JRP, we will get our desired result.

LEMMA 5.1.1. *If an* ALG_g *service serves a request* q_j *for some* j *then it must serve* $\{q_1, q_2, \ldots, q_{w_i}\}$. *That is,* ALG_g *will always serve all requests in a set of surrogate requests together.*

PROOF. We first remark that by construction of our algorithm for Cardinality JRP, ALG_g is actually the solution to the Piecewise Cardinality JRP instance constructed from the CJRP instance. Hence, we can analyse our solution ALG_g by looking at what happens in the Piecewise Cardinality JRP instance. We recall that the Piecewise Cardinality JRP instance will be exactly the same as CJRP except it will have a different cost function.

Intuition: At a high level, the intuition for this Lemma is quite simple. All requests in a set of surrogate requests Q arrive at the same time and have the same delay functions by construction. Hence, whenever one request in Q is eligible for a service, all the requests in Q will be eligible for the same service. This

is because the requests in Q arrived and were set to level 1 at the same time and all have the same delay functions, and hence they will have accumulated the same amount of delay at all times. Consequently, the delay accumulated and levels of all requests in Q will be synchronised, making the requests in Q eligible for the same services. Moreover, since our investment phase in the Piecewise Cardinality JRP algorithm works by calculating the delay requests accumulate up to some point in the future and investing in this amount, for each request in Q, the amount invested into each request's corresponding item type will always be the same since they have the same delay functions. Overall, we have that all requests in Q will always be eligible for the same services and that their corresponding investment counters are always equal and therefore it follows that when one investment counter reaches the necessary threshold to serve the item type, all the requests in Q should also have their investment counters reach this threshold. Therefore, when a request gets served, the entire set of surrogate requests should be served. Formalising this proof is much more involved.

Consider an item type i and all requests in WJRP on this item type. Let the surrogate requests of these requests on item i be the sets $Q_1, Q_2, \dots Q_M$. Let $\lambda_1, \lambda_2, \dots, \lambda_m$ be all the services which serve some surrogate request in $Q_1 \cup Q_2 \cup \cdots \cup Q_M$. We assume wlog that these services are listed in chronological order and hence λ_k has service time t_k before λ_{k+1}'s service time of t_{k+1} for all $k \in [1, m)$. Then we will prove that for each $k \in [1, m]$, λ_k always serves all requests in a set of surrogate requests Q_j together and after each service λ_k, the level l investment counters on items $i_1, \dots, i_{w_i}$ are the same for each level l.

To prove this, we first prove some useful claims. For notation, if there is a service λ at time t, then t^- will refer to the time t but before the service λ has been made.

CLAIM 5.1.2. *Consider a surrogate requests set Q_j at time t_k. Suppose all of the requests in Q_j are active at time t_k^-.*

Then for any service λ with service time $t \leq t_k$, if any request in Q_j was eligible for λ, all requests in Q_j must have been eligible for λ.

Proof: Fix the set of surrogate requests Q_j. Let $\mu_1, \mu_2, \dots, \mu_{m'}$ be all the services with service time at most t_k in which any request from Q_j was eligible. We assume wlog that these services are listed in chronological order. Then we will prove that all requests in Q_j are in fact eligible for each $\mu_{k'}$ where $k' \in [1, m']$. We will do so by induction on k'.

For our base case, consider μ_1 served at time s_1. All the requests in Q_j must be level 1 at time s_1^- as by definition, they could not have been eligible for any other service yet. All requests in Q_j are also unserved by assumption. Therefore, since all requests in Q_j are unserved and level 1 when μ_1 is made, they should all have been eligible for this service. This is because μ_1 is a service of level at least 1 and hence every unserved request of level 1 is eligible.

Assume for an inductive hypothesis that all requests in Q_j are eligible for $\mu_1, \ldots, \mu_{k'-1}$ for some $k' \in [2, m']$. Now consider the service $\mu_{k'}$ made at time $s_{k'}$. At the time $s_{k'}^-$, all requests in Q_j are the same level since by our inductive hypothesis, they were all eligible for the level l service $\mu_{k'-1}$ which would have made all requests in Q_j level l. Moreover, by definition of $\mu_{k'}$ being the next service in which some request from Q_j is eligible, there cannot have been another service between $\mu_{k'-1}$ and $\mu_{k'}$ that could have altered the levels of any request in Q_j. Lastly, by assumption, none of the requests in Q_j have been served yet. Therefore, if one request from Q_j is eligible for $\mu_{k'}$, then all requests in Q_j should be eligible since they are all active and of the same level l. This proves the inductive case and hence we conclude by induction that, for all services with service times at most t_k, if a request from Q_j is eligible then all requests from Q_j are eligible. ■

COROLLARY 5.1.3. *Suppose that at time t, the level l investment counters for items $i_1, \ldots, i_{w_i}$ are all the same for each level l. Suppose further that at time t any set of surrogate requests Q_j is either completely served or completely unserved. Let t_k be the time of the next service after time t to serve any surrogates of item i, that is, $t_k = \min\{t_{k'} : k' \in [1, m], t_{k'} > t\}$.*

Then the level l investment counters for items $i_1, \ldots, i_{w_i}$ are all the same at time t_k^- and for each level l.

Proof: Consider any set of surrogates Q_j. We first remark that either Q_j has been completely served by time t or all requests are active up till time t_k^-. If any of the requests in Q_j were served prior to or at time t then by assumption, all of the requests should have been served. If none of the requests in Q_j have been served by time t then none of the requests in Q_j can be served between times t and t_k since by definition, t_k is the time of the next service after t that can serve any of the surrogates. Consequently, all requests in Q_j will be active up till time t_k^-. Therefore, for any set of surrogates Q_j, it is either completely served by time $t < t_k^-$ or completely unserved up till time t_k^-.

Now we assume for a contradiction that the investment counters are not synchronised at time t_k^-. Since by assumption the counters were synchronised at time t, this would imply that at some time between t and t_k, there was a service λ in which the amount invested into some item i_p differs from the amount

invested into some item $i_{p'}$. We know that a request for item i_p or $i_{p'}$ must have been eligible for λ otherwise nothing would have been invested into both items i_p and $i_{p'}$ in the first place. We can assume wlog that a request q_{i_p} for item i_p is eligible for λ. Let Q_j be the set of surrogate requests that q_{i_p} belongs to. Since q_{i_p} is eligible for a service between times t and t_k, this implies that q_{i_p} has not been served by time t and hence by our earlier remark, the entire set Q_j must be unserved up till time t_k^-. Consequently, Claim 5.1.2 applies which tells us that all requests in Q_j must be eligible for λ. Since all requests in Q_j have the same delay function by construction, the amount of delay each request accumulates at each time is identical. Consequently, during the investment process of λ, the amount of delay each request in Q_j accumulates that needs to be invested in is identical. In particular, the amount of delay accumulated and hence invested in by requests q_{i_p} and $q_{i'_p} \in Q_j$ of item $i_{p'}$ is identical. Since this argument applies to all requests on items i_p and $i_{p'}$ eligible for λ, it follows that the amount λ invests into the item types i_p and $i_{p'}$ could not possibly differ, a contradiction. ■

CLAIM 5.1.4. *Consider a service λ at time t that serves one of the item types $i_1, \ldots, i_{w_i}$. Suppose that at time t^-, the level l investment counters on items $i_1, \ldots, i_{w_i}$ are the same at each level l. Suppose further that if a request in one of the surrogate request sets Q_j is eligible for λ then the entire set Q_j is eligible for λ.*

Then, λ must in fact serve all of the item types $i_1, \ldots, i_{w_i}$ and after the service of λ, the level l investment counters on items $i_1, \ldots, i_{w_i}$ are the same at each level l.

Proof: Let λ be a level l service, then it is only able to modify level l investment counters by construction of our algorithm. Since the level $l' \neq l$ investment counters on items $i_1, \ldots, i_{w_i}$ are the same at every level prior to the service of λ, then after the service of λ, they will also remain the same since they are not modified. If λ serves all of the item types $i_1, \ldots, i_{w_i}$, then by construction of the algorithm, it would set the level l investment counters for the items $i_1, \ldots, i_{w_i}$ to 0. Therefore, at the conclusion of λ, all investment counters on items $i_1, \ldots, i_{w_i}$ are the same at each level. Hence, it remains to show that λ does indeed serve all of the item types $i_1, \ldots, i_{w_i}$.

At the beginning of the service, all the level l investment counters on items $i_1, \ldots, i_{w_i}$ are the same by assumption. Hence, we only need to consider the rate at which these investment counters increase during the service of λ. We will argue that the rates of increase of all these investment counters are the same. Suppose towards a contradiction that this was not true and there are two distinct item types i_p and $i_{p'}$ whose investment counters are increasing at different rates. Consider any request q_1 on item i_p that

is eligible for λ then by assumption, the request q_2 on item $i_{p'}$ belonging to the same set of surrogates as q_1 must also be eligible. Both q_1 and q_2 have the same delay functions by construction and hence over any period of time, the amount of delay accumulated and hence invested in by these two requests will be identical. This argument applies to all requests on item i_p or $i_{p'}$ eligible for λ and hence it follows that the amount invested and added onto the investment counters for i_p and $i_{p'}$ must be identical throughout λ's investment phase, a contradiction. Therefore, we know that the rates of increase of all the investment counters are the same. Hence, this implies that at any time, the amount invested by λ into the item types $i_1, \ldots, i_{w_i}$ is the same.

Ultimately, since λ serves one of the item types $i_1, \ldots, i_{w_i}$, this would imply by construction of our Piecewiswe Cardinality JRP algorithm that the amount invested into one of the item types reached the necessary threshold of δ_l required to serve the item type. Now the investment counters for the items $i_1, \ldots, i_{w_i}$ are the same at the start of λ, and as argued earlier, the amount invested by λ into these counters is the same at any point in time. Hence, the fact that one of these counters reached the required threshold must imply that all the counters reached their thresholds since all level l investment counters have the same threshold of δ_l. Therefore, it follows that all of the item types $i_1, \ldots, i_{w_i}$ must be served as required. ■

Finally, we can prove our initial statement: for each $k \in [1, m]$, λ_k always serves all requests in a set of surrogate requests Q_j together and after each service λ_k, the level l investment counters on items $i_1, \ldots, i_{w_i}$ are the same for each level l. We will prove this statement by induction on k.

Base case: Consider λ_1 served at time t_1. At time 0, all the investment counters are 0 since they were just initialised. Therefore, we can apply Corollary 5.1.3 to get that the level l investment counters on items $i_1, \ldots, i_{w_i}$ are all the same for all levels l at time t_1^-. Furthermore, consider a request in some surrogate requests set Q_j that is eligible for λ_1. The entire set Q_j must be active at time t_1^- since λ_1 is by definition, the first service to have served any of the surrogates. Therefore, we can apply Claim 5.1.2, which tells us that the entire set Q_j is eligible for λ_1. This argument applies across all surrogate sets Q_j. Lastly, by definition, λ_1 serves one of the item types $i_1, \ldots, i_{w_i}$. Therefore, we have the necessary conditions to apply Claim 5.1.4 which tells us that λ_1 serves all of the item types $i_1, \ldots, i_{w_i}$ and after the service of λ_1, all investment counters on $i_1, \ldots, i_{w_i}$ across all levels are the same. Therefore, if some request in some surrogate set Q_j gets served by λ_1, the entire set Q_j must be served since we have argued that the entire set Q_j is eligible and λ_1 serves all items $i_1, \ldots, i_{w_i}$. Thus we have proven the base case.

Inductive case: Assume that each service up till λ_{k-1} for $k \in [2, m]$ has served all requests in a set of surrogate requests together and after each service, the level l investment counters on items $i_1, \ldots, i_{w_i}$ are the same for each level l. Now we will consider the k-th service λ_k and show that this invariant still holds.

At time t_k^-, all the investment counters on on item types $i_1, \ldots, i_{w_i}$ are synchronised at each level. This is because our inductive hypothesis tells us that they were synchronised at time t_{k-1}. Moreover, our inductive hypothesis states that any service up till time t_{k-1} that served a surrogate request in fact served sets of surrogate requests together so at time t_{k-1}, each set of surrogate requests must either be completely served or completely unserved. Hence, Corollary 5.1.3 applies which gives us the desired result. Now consider any request in a surrogate requests set Q_j that is eligible for λ_k. If any of the requests in Q_j are served at or prior to time t_{k-1} then our inductive hypothesis implies that the entire set must have been served which means none of the requests could be eligible for λ_k. Therefore the entire set Q_j must be unserved at time t_{k-1}. Furthermore, none of the requests in Q_j can be served between times t_{k-1} and t_k since by definition of our service times, there are no services that serve any of the surrogate requests between these times. Therefore, at time t_k^-, the requests in Q_j are all active and since one of the requests is eligible for λ_k, Claim 5.1.2 tells us that all requests in Q_j are in fact eligible for λ_k. This argument applies to all requests across all surrogate request sets. Lastly, we know by definition that λ_k serves one of the item types $i_1, \ldots, i_{w_i}$. Hence, we have all the conditions required to apply Claim 5.1.4. This tells us that λ_k serves all of the item types $i_1, \ldots, i_{w_i}$ and after the service, all investment counters on $i_1, \ldots, i_{w_i}$ across all levels are the same. Thus if there is some request in some surrogate set Q_j that gets served by λ_k, then the entire set Q_j must have been served since the entire set is eligible and λ_k serves all of the items $i_1, \ldots, i_{w_i}$. This hence proves our inductive case.

We conclude by induction that, λ_k, for each $k \in [1, m]$ always serves all requests in a set of surrogate requests Q_j together and after each service, the level l investment counters on items $i_1, \ldots, i_{w_i}$ are the same for each level l. Therefore, any service which serves a surrogate request of item i, $q_{j,p} \in Q_j$, must in fact serve the entire set of surrogate requests Q_j. As this applies to all items i and their surrogates, this proves that any ALG_g service must serve sets of surrogate requests together as required. □

LEMMA 5.1.5. *The cost of* ALG_w *as a solution for the original WJRP instance is exactly the same as the cost of* ALG_g *as a solution to the CJRP instance constructed.*

PROOF. First we consider the delay costs of ALG_w. For any request q on item type i served at time t, ALG_w will pay the delay on q from its arrival time till time t. That is, it will pay $d_q(t)$. Now if we consider q's surrogate requests in the Cardinality JRP instance, by construction of ALG_w and by Lemma 5.1.1 the surrogate requests $q_1, \ldots, q_{w_i}$ must have all been served at time t as well. Therefore, each surrogate request will have to pay a delay of $d_q(t)/w_i$ and in total, the w_i surrogate requests pay a total delay of $d_q(t)$, exactly what ALG_w pays for q. Therefore, for any request q served by ALG_w, its w_i surrogate requests served by ALG_g will incur the same total delay cost. Hence, the delay costs of ALG_w and ALG_g are identical.

Now we consider the service cost. We first remark that by construction, both problems use the same cost function f. Now consider any service λ made by ALG_w. It pays a cost of $f(\sum_{i\in\lambda} w_i)$. By construction of ALG_w, there must be a corresponding service λ' in ALG_g that serves some i_j for each $i \in \lambda$. However, if λ' serves some i_j, since Lemma 5.1.1 implies that surrogate requests must be served together, we must have that ALG_g in fact serves $i_1, i_2, \ldots, i_{w_i}$ as well. Since this holds for each $i \in \lambda$, we know that the λ' must serve w_i item types for each item type i served by λ. In total, λ' must serve $\sum_{i\in\lambda} w_i$ item types. Overall, this means λ' costs $f(\sum_{i\in\lambda} w_i)$, which is exactly the same as its corresponding service λ in ALG_w. This argument applies to all services so we have that all services in ALG_w and their corresponding service in ALG_g cost the same.

The fact that all service costs are the same and all delay costs are the same means that both ALG_w and ALG_g must cost the same as desired. □

LEMMA 5.1.6. *The cost of* OPT_g *is at most the cost of* OPT_w.

PROOF. We will show that OPT_w can be transformed into a feasible solution for the CJRP instance without the cost changing. Since any feasible solution must cost at least as much as the optimal solution then our result will follow.

Given OPT_w, we construct the solution to the Cardinality JRP instance OPT'_w as follows. Whenever OPT_w makes a service, so too will OPT'_w and whenever OPT_w serves the request q on item type i, the corresponding service in OPT'_w will serve the requests $q_1, q_2, \ldots, q_{w_i}$ on item types $i_1, i_2, \ldots i_{w_i}$ respectively. Clearly, OPT'_w is a feasible solution since in the delay model any solution, regardless of whether or not it serves all requests, is a feasible solution. Hence, it remains to show that the cost of OPT'_w is the same as the cost of OPT_w. The proof of this will be similar to the proof of Lemma 5.1.5.

We first show that all delay costs are the same. When OPT_w serves the request q at time t it pays the delay penalty of $d_q(t)$. q's surrogate requests in the Cardinality JRP instance are $q_1, \ldots, q_{w_i}$, each of which get served at time t in OPT'_w by construction. Each surrogate request accumulates a delay of $d_q(t)/w_i$ by time t and hence in total, all of q's surrogate requests accumulate a total delay of $d_q(t)$ by the time they are served. This is exactly the same delay cost as q and since this applies to all requests, we have that the delay costs between the two solutions are the same.

Next we show that the service costs are the same. When OPT_w makes a service λ, it pays a cost of $f(\sum_{i\in\lambda} w_i)$. The corresponding service λ' made in OPT'_w will by construction serve the item types $i_1, i_2, \ldots, i_{w_i}$ for each item type i served by λ. In total, λ' will have to serve $\sum_{i\in\lambda} w_i$ distinct item types and thus will have to pay $f(\sum_{i\in\lambda} w_i)$ which is exactly the same as the cost of λ. Since this applies to all services, and we know that there is a one-to-one correspondence between services in OPT_w and services in OPT'_w, we get that the total service costs are the same.

In conclusion, OPT_w and OPT'_w have the same delay and service costs and thus cost the same. Therefore, $\mathrm{OPT}_w = \mathrm{OPT}'_w \geq \mathrm{OPT}_g$ using the fact that OPT_g is optimal for Cardinality JRP and hence costs at most the cost of any feasible solution. □

Finally, we will combine our Lemmas to prove our Theorem

THEOREM 1.2. *There exists an $O(1)$-competitive, deterministic algorithm for Weighted Cardinality JRP with Delay.*

PROOF. First, we remark that ALG_w is indeed a feasible solution to the WJRP instance. This is because, in the delay model, any solution is feasible since there is no requirement that we need to serve all requests. Now we prove that it is constant competitive by utilising our earlier results.

$$\begin{aligned}
\mathrm{ALG}_w &= \mathrm{ALG}_g && \text{(Lemma 5.1.5)}\\
&\leq O(1) \cdot \mathrm{OPT}_g && \text{(Theorem 1.1)}\\
&\leq O(1) \cdot \mathrm{OPT}_w && \text{(Lemma 5.1.6)}
\end{aligned}$$

□

CHAPTER 6

Conclusions and Open Problems

In this thesis, we have studied the Cardinality Joint Replenishment Problem. We have provided a new approach to Online Cardinality JRP with Deadlines that deviates from the algorithm proposed by Khatkar (2020). This nearly optimal $O(1)$-competitive algorithm was then generalised to produce an $O(1)$-competitive, polynomial time algorithm for Online Cardinality JRP with Delays. This is the first known result for Online Cardinality JRP with Delays. Whilst this result is nearly optimal, further work could tighten the competitive ratio.

Our algorithm for Cardinality JRP with Delays was used to produce an $O(1)$-competitive online algorithm for Weighted Cardinality JRP with Delays, once more producing a nearly optimal algorithm. However, we remark that the reduction from Weighted Cardinality JRP to Cardinality JRP was pseudo-polynomial and hence it remains an open question whether a polynomial time, constant competitive algorithm can be developed.

For other variants of JRP, such as Submodular and Subadditive JRP, there has been little work done in the online setting and hence future work could investigate these problems and potentially extend our ideas and techniques to these problems. The notion of investments introduced by Azar and Touitou (2020) and extended by us is also quite novel and could be applied to different online problems with delay.

Bibliography

Yossi Azar, Ashish Chiplunkar, Shay Kutten, and Noam Touitou. 2020. Set cover with delay - clairvoyance is not required. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 8:1–8:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Yossi Azar and Noam Touitou. 2020. Beyond tree embeddings - a deterministic framework for network design with deadlines or delay. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1368–1379. IEEE.

Luca Becchetti, Peter Korteweg, Alberto Marchetti-Spaccamela, Martin Skutella, Leen Stougie, and Andrea Vitaletti. 2006. Latency constrained aggregation in sensor networks. In Yossi Azar and Thomas Erlebach, editors, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 88–99. Springer.

Marcin Bienkowski, Martin Böhm, Jaroslaw Byrka, Marek Chrobak, Christoph Dürr, Lukáš Folwarczný, Lukasz Jez, Jiri Sgall, Nguyen Kim Thang, and Pavel Veselý. 2016. Online algorithms for multi-level aggregation. In Piotr Sankowski and Christos D. Zaroliagis, editors, *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Marcin Bienkowski, Jaroslaw Byrka, Marek Chrobak, Neil B. Dobbs, Tomasz Nowicki, Maxim Sviridenko, Grzegorz Swirszcz, and Neal E. Young. 2013. Approximation algorithms for the joint replenishment problem with deadlines. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, volume 7965 of *Lecture Notes in Computer Science*, pages 135–147. Springer.

Niv Buchbinder, Moran Feldman, Joseph (Seffi) Naor, and Ohad Talmon. 2017. O(depth)-competitive algorithm for online multi-level aggregation. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1235–1244. SIAM.

Niv Buchbinder, Tracy Kimbrel, Retsef Levi, Konstantin Makarychev, and Maxim Sviridenko. 2008. Online make-to-order joint replenishment model: primal dual competitive algorithms. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 952–961. SIAM.

Maurice Cheung, Adam N. Elmachtoub, Retsef Levi, and David B. Shmoys. 2016. The submodular joint replenishment problem. *Math. Program.*, 158(1-2):207–233.

Sin-Shuen Cheung. 2014. The submodular facility location problem and the submodular joint replenishment problem. In Evripidis Bampis and Ola Svensson, editors, *Approximation and Online Algorithms - 12th International Workshop, WAOA 2014, Wrocław, Poland, September 11-12, 2014, Revised Selected Papers*, volume 8952 of *Lecture Notes in Computer Science*, pages 71–82. Springer.

Nico P. Dellaert and M. Teresa Melo. 1995. Heuristic procedures for a stochastic lot-sizing problem in make-to-order manufacturing. *Ann. Oper. Res.*, 59(1):227–258.

Daniel R. Dooly, Sally A. Goldman, and Stephen D. Scott. 1998. TCP dynamic acknowledgment delay: Theory and practice (extended abstract). In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 389–398. ACM.

Sudipto Guha, Adam Meyerson, and Kamesh Munagala. 2001. A constant factor approximation for the single sink edge installation problems. In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 383–388. ACM.

Anupam Gupta, R. Ravi, Kunal Talwar, and Seeun William Umboh. 2017. LAST but not least: Online spanners for buy-at-bulk. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 589–599. SIAM.

Jahanvi Khatkar. 2020. Online joint replenishment with deadline on subadditive costs. The University of Sydney, Honours Thesis.

Viswanath Nagarajan and Cong Shi. 2016. Approximation algorithms for inventory problems with submodular or routing costs. *Math. Program.*, 160(1-2):225–244.

Tim Nonner and Alexander Souza. 2009. A 5/3-approximation algorithm for joint replenishment with deadlines. In Ding-Zhu Du, Xiaodong Hu, and Panos M. Pardalos, editors, *Combinatorial Optimization and Applications, Third International Conference, COCOA 2009, Huangshan, China, June 10-12, 2009. Proceedings*, volume 5573 of *Lecture Notes in Computer Science*, pages 24–35. Springer.

APPENDIX A

Rent-Or-Buy Cardinality JRP with Deadlines Pseudocode

Algorithm 1.9: Procedures to handle requests arriving and tracking the currently active chain

```
/* variable to track of the most recent service in the
   currently active chain                                    */
active ← NULL;
Function OnArrival(request q)
    /* set level to 1 and pointer to NULL                    */
    level(q) ← 1;
    pointer(q) ← NULL;
```

Algorithm 1.10: Procedure to handle service triggering

```
Function OnDeadline(request q)
    /* triggering request reaches its deadline so we make a
       service                                                */
    Start a new service λ at current time t;
    level(λ) ← 1;
    /* flag to keep track of whether the service is normal   */
    normal(λ) ← False;
    if T_λ ≠ ∅ then
        /* non-primary service is normal and eligible for an
           upgrade                                            */
        normal(λ) ← True;
        foreach q' ∈ A_t do
            if ∃t' ∈ [a_q', d_q'] and at least B/δ level charged requests q_c s.t. a_{q_c} ≤ t' ≤ d_{q_c} then
                level(λ) ← 2;
                /* buy services cannot be normal services     */
                normal(λ) ← False;
                break;
    ServeRequests(λ);
    UnchargeRequests(λ);
```

Algorithm 1.11: Procedure to handle serving and charging requests

Function *ServeRequests(service λ)*

```
/* keep track of which requests are to be served          */
Q_λ ← ∅;
if level(λ) = 2 then
    /* buy service serves everything                      */
    Q_λ ← A_t;
else
    while Q_λ ≠ A_t do
        Let q_last ∈ A_t \ Q_λ be the request with earliest deadline;
        Add the item type of q_last to λ;
        if normal(λ) then
            Make q_last a charged request;
        /* add eligible requests of the same item type as q_last */
        foreach q' ∈ A_t whose item type is the same as q_last do
            Q_λ ← Q_λ ∪ {q'};
        /* stop if the total individual cost exceeds σ        */
        if δ|λ| ≥ σ then
            break;
    Serve Q_λ;
```

Algorithm 1.12: Procedure to handle uncharging requests

Function *UnchargeRequests(service λ)*

```
/* uncharge requests if λ serves everything               */
if A_t \ Q_λ = ∅ then
    Uncharge all requests in Q_λ;
/* end the previously active chain and remove the charged
   requests of their last service if our service was primary
   */
if T_λ ≠ ∅ then
    Uncharge all requests in Q_active;
    active ← NULL;
/* keep track of the new last service of the currently active
   chain                                                  */
if A_t \ Q_λ ≠ ∅ then
    active ← λ;
```

APPENDIX B

Rent-Or-Buy Cardinality JRP with Deadlines Analysis

B.1 Analysis: Solution Structure

OBSERVATION B.1.1. ALG *is a feasible solution.*

PROOF. To show feasibility, we must show that all requests are served before their deadlines. Whenever a request reaches its deadline a service will be triggered that will serve this request that has reached its deadline. Therefore, no request will ever be left unserved past their deadline. Thus, every request must be serviced before its deadline and our solution is feasible. □

OBSERVATION B.1.2. *Every service can only point and be pointed to by at most one other service.*

PROOF. By construction, every service can only point to at most one other service. Now suppose for a contradiction that a service λ served at time t gets pointed to by services λ_1 and λ_2 which have triggering requests q_1 and q_2 respectively. Let these triggering requests have time intervals are $[a_1, d_1]$ and $[a_2, d_2]$ respectively. We assume wlog that λ_1 occurs before λ_2 and hence $d_1 < d_2$ since the services are triggered on the deadlines of their triggering requests. Since λ_2 points to λ, we must have $a_2 < t$ since q_2's time interval must intersect with λ's triggering request's time interval in order to point at λ. In order for λ_1 to point to λ, the service must obviously be triggered after λ and hence $d_1 > t$. Therefore, we know that $a_2 < t < d_1$, which implies that q_2's time interval intersects with q_1's time interval. Then when q_2 triggers λ_2 after λ_1 has already been triggered, since q_2's time interval intersects with q_1's, λ_2 should point at λ_1 instead of λ since λ_1 is more recent, a contradiction. □

LEMMA B.1.3. *Any rent service made by* ALG *costs at most* 3σ.

PROOF. The cost of a rent service is composed of two components: the shared, fixed cost of σ that is paid whenever any rent service is made and the individual cost of δ for each item type served in the

service. By design of our algorithm, rent services continually serve requests until the total individual cost has exceeded σ. Hence, in the worst case scenario, our service could pay a total individual cost of $\sigma + \delta < 2\sigma$. Therefore, the total cost of the service is at most 3σ. □

B.2 Analysis: Charging tail service costs

LEMMA B.1.4. *The non-buy tail services cost at most 3 times the cost of the primary services.*

PROOF. By Observation B.1.2, our chains are well-defined linear structures with no branching so each tail service must have a corresponding and distinct primary service which was responsible for beginning the same chain. Hence, considering any non-buy tail service, we know its cost is at most 3σ using Lemma B.1.3 and we know its corresponding, unique primary service costs at least σ since any primary service is a rent service and any rent service must pay at least the shared σ cost. Therefore, we know that for any non-buy tail service, its cost is at most 3 times its corresponding primary service's cost. Therefore, summing across all the non-buy tail services and noting that no primary service gets charged twice since each tail corresponds to a unique primary service we get the desired result. □

LEMMA B.1.5. *Suppose there is a normal service at time t that serves a request q with deadline d. Then there cannot be any other service between times t and d (inclusive).*

PROOF. The fact that q was served in a normal service λ implies by definition that there must be a subsequent service in the same chain. Hence, this is only possible if λ leaves some request unserved which would then be the triggering request $q_{trigger}$ for the subsequent service in λ's chain.

$q_{trigger}$ must have arrived by the time λ was triggered since by definition, $q_{trigger}$ must intersect with λ's triggering request to continue the chain. Moreover, we must have that the deadline of $q_{trigger}$ is after time d since by design of the algorithm, requests are served in ascending deadline order so the fact that $q_{trigger}$ is active during λ's service but left unserved whilst q whose deadline is d was served implies that its deadline must be later.

Therefore, $q_{trigger}$ cannot trigger the service that continues λ's chain until some time after d. If there were some service between times t and d, this new service would be more recent than λ and $q_{trigger}$ should also intersect the triggering request of this service since $q_{trigger}$ has arrived before time t. Therefore, by construction of chains, $q_{trigger}$ can no longer be the request which triggers the subsequent

service in λ's chain since it would instead trigger a service that points to this new service between times t and d. This is a contradiction, hence, proving the desired result. □

LEMMA B.1.6. *The cost of the buy services is at most the cost of the normal services.*

PROOF. Consider any buy service. It gets triggered because at the time of the service, there was some active request within whose time interval there exists at least B/δ intersecting charged requests that triggered the buy. We observe that a request is charged iff it contributes a cost of δ to some normal service since requests are only charged when it gets included in a normal service and is an item type that the service has not yet included and hence pays an individual cost of δ for the item. Moreover, each charged request can only contribute a cost of δ to one distinct normal service since no request can be served twice by two different services. Therefore, the fact that there are at least B/δ charged requests, each of which contributes a unique cost of δ to some normal service means that in total, these charged requests that have triggered the buy incur a total individual cost of at least B across all the normal services. Since our buy service costs B, we can charge its cost to the individual costs of the charged requests that triggered it.

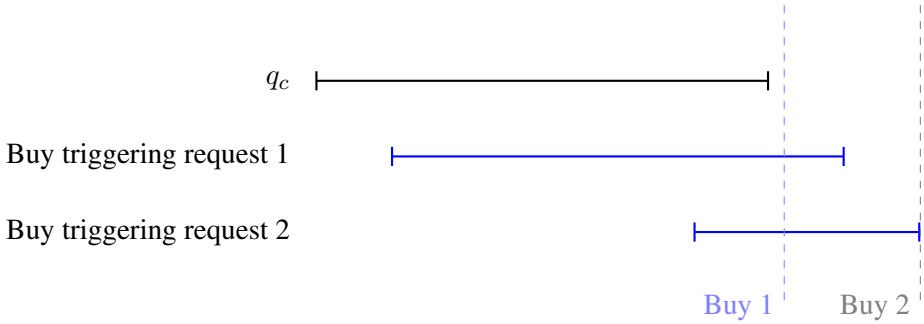


FIGURE B.1: Illustration of Lemma B.1.6's proof.

To conclude the proof we will show that a charged request cannot be used to trigger two different buy services and hence cannot be charged to more than once. Suppose for a contradiction that this was not true then there is a charged request q_c with time interval $[a_c, d_c]$ to which two buy services charge. For a buy service to charge q_c, the buy service must occur after q_c has already been served and charged by a normal service at time t. However, using Lemma B.1.5, we know that there cannot be a service between times t and d_c. Therefore, the two buy services that charge q_c must occur after d_c. By design of the algorithm, the buy triggering requests of these buy services must have arrival times before d_c in order to intersect with q_c's time interval and charge to it. Now consider when the first buy service is triggered. The buy triggering request for the second buy service must be active at this time since it arrives before

d_c which as argued earlier is before the service time of this first buy service and it cannot have been served yet otherwise the second buy service would not be a buy service. However, the first buy service will serve all active requests by construction, hence also serving the second buy service's buy triggering request. Therefore, the second buy service could never have been triggered, which is a contradiction. Figure B.1 illustrates this argument.

Therefore, the fact that each charged request can only be charged by at most one buy service allows us to charge each buy service to some unique set of charged requests. Therefore, the costs of our buy services is at most the individual cost incurred by all the charged requests and this is, by definition, at most the total individual cost of all the normal services. Hence, we can charge our buy service costs to the normal service costs as required. □

B.3 Analysis: Charging to OPT

LEMMA B.1.7. *The cost of the primary services is at most 3 times the cost of* OPT.

PROOF. Consider the triggering request of a primary service. It triggers a primary service that costs at most 3σ using Lemma B.1.3. At some time within the time interval of the triggering request, OPT must make a service that serves the request otherwise it would not produce a feasible solution. Any service made by OPT must cost at least σ since OPT either pays $B > \sigma$ or it pays $\sigma + \delta|\lambda| > \sigma$. Therefore, we can charge the cost of the primary service that this request triggers to the corresponding OPT service that serves this triggering request. That is, we get the cost of the primary service is at most 3 times the cost of the OPT service that served the triggering request.

Now, consider the set of triggering requests that trigger primary services. The time intervals of the requests in this set must be pairwise disjoint otherwise the request should not have triggered a primary service by definition. Hence, we know that any OPT service cannot serve more than one primary service's triggering request since their time intervals are all completely disjoint. Thus, any OPT service can only be charged to by at most once one primary service. Therefore, we can charge each primary service to a distinct OPT service which gives us the desired result. □

OBSERVATION B.1.8. *Any service by* OPT *can only include at most 2 charged requests on the same item type.*

PROOF. Consider three charged requests q_1, q_2, q_3 on the same item type with time intervals $[a_1, d_1]$, $[a_2, d_2]$, $[a_3, d_3]$ respectively. Note that since these requests on the same item type are charged, this means our algorithm must have served them in separate services since additional requests of the same item type in a service are not charged. We will assume that q_1, q_2, q_3 are served at times t_1, t_2, t_3 respectively where wlog $t_1 < t_2 < t_3$.

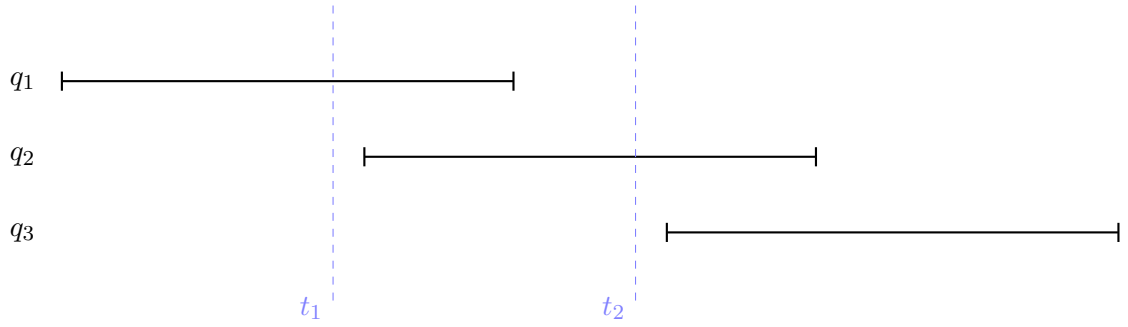


FIGURE B.2: Illustration of Observation B.1.8's proof.

We first note that $a_2 > t_1$ since otherwise q_2 would have been active during the service time of q_1 and thus been included in the same service and served for free and been left uncharged. Likewise, $a_3 > t_2$ for the same reason. Next, using Lemma B.1.5 we know that there cannot be any service between times t_1 and d_1 and since $t_2 > t_1$ it follows that $t_2 > d_1$. Combining this with the earlier fact that $a_3 > t_2$, we can conclude that $a_3 > t_2 > d_1$ which implies that q_3 is completely disjoint from q_1. Therefore, at any time, there can only be at most two charged requests of the same item type whose time intervals intersect with this time. Therefore, this means that any OPT service can serve at most two charged requests of the same item type. □

OBSERVATION B.1.9. *The maximum number of charged requests whose time intervals intersect at some time t^* is at most $2B/\delta$.*

PROOF. Any normal service made by the algorithm can only charge at most $\lceil \sigma/\delta \rceil < \lceil \frac{B}{2\delta} \rceil < B/\delta$ requests. Now suppose for a contradiction that at some time t^* we have more than $2B/\delta$ charged requests and consider the last service to have served and created any of these charged requests. Prior to the service, the number of charged requests intersecting at time t^* is at least $2B/\delta - B/\delta = B/\delta$ since this final service could only have charged at most B/δ requests. Therefore, at the time of this last service, there must have been an active request that intersects with time t^* since by definition of this last service such a request will be served and charged by this service but we also know that at time t^* there

already exists at least B/δ charged requests which means this service should have become a buy service and thus been unable to charge any additional requests, a contradiction. □

LEMMA B.1.10. *The cost of the normal services is at most 4 times the cost of* OPT.

PROOF. For any normal service, it must be a rent service by design and rent services are comprised of two costs: the shared cost of σ that must always be paid and the individual cost $\delta|\lambda|$. Observe that the total individual cost is at least σ. If this was not the case, then by design of the algorithm, the service will have served all active requests and thus been a tail service as there cannot be any requests whose time intervals intersect with this service's triggering request to continue the chain. Therefore, for any normal service, we can charge the shared cost of σ to the total individual cost of the service. Therefore the cost of a normal service is at most 2 times its total individual cost.

Now we consider the total individual cost across all normal services. We first observe that the only requests that contribute to the individual cost of any normal service are precisely the charged requests. Requests served by a normal service and incurring an individual cost will become charged. Any uncharged requests that are included in a normal service must have been included since another request of the same item type was included and charged and it is this other request that contributes to the individual cost. Hence, the total individual cost across all normal services is exactly the total individual cost incurred by all the charged requests. Therefore, we will show that for any charged request, since it incurs an individual cost of δ, OPT must also pay at least $O(1) \cdot \delta$ for this request alone.

Consider any service λ^* made by OPT. There are two cases, the first being λ^* is a rent service, that is, it is made using the cost function piece $g(\lambda^*) = \sigma + \delta|\lambda^*|$. Consider any item type served by λ^*. λ^* will have to pay a distinct cost of δ for this item type. Using Observation B.1.8, we know that λ^* can only serve at most 2 charged requests of the same item type at any time. These charged requests of the same item type accumulate a total individual cost of at most 2δ that we need to charge to λ^*. Hence, we can charge this total individual cost to 2 times the individual cost δ that λ^* pays for this item type. Considering this across all item types, and noting that for each item type, λ^* must make a distinct δ payment, we can charge the total individual cost of all charged requests served by λ^* to 2 times the individual cost of λ^*.

The second case to consider is if λ^* is a buy service, meaning, it has used the cost function piece $g(\lambda^*) = B$. Now using Observation B.1.9, we know that the maximum number of charged requests that can exist at any time t^* is $2B/\delta$ and thus any λ^* buy service can only serve at most $2B/\delta$ charged

requests. These $2B/\delta$ charged requests incur a total individual cost of $2B$ which we need to charge to λ^*. Since λ^* is a buy service, it must cost exactly B. Therefore, we can charge this total individual cost of the charged requests to 2 times the cost of λ^*. Since no optimal service can serve any request twice, each charged request served by an OPT buy service can only be charged to one unique buy service. Overall, we have that the individual cost of charged requests that are served in a buy OPT service are at most 2 times the cost of the OPT buy services.

Therefore, we have shown that for any service λ^* made by OPT, the total individual cost of all the charged requests it serves can be charged to 2 times the cost of λ^*. That is, any OPT service's cost will cover the individual costs incurred by the charged requests it serves. Considering this argument across all services made by OPT and noticing that OPT must serve all requests and hence all charged requests, we get that all charged requests and the individual costs they incur will be charged to some OPT service. Therefore, the total individual cost incurred by all charged requests is at most 2 times OPT. Then to conclude we have that the cost of normal services is at most two times the individual cost of the charged requests and hence consequently at most 4 times the cost of OPT as required. □

Putting everything together, we can now prove our Theorem

THEOREM 3.1. *There exists an $O(1)$-competitive, deterministic algorithm for Rent-or-Buy Cardinality JRP with Deadlines.*

PROOF. Having observed that our algorithm is feasible in Observation B.1.1, we simply need to prove it is constant competitive. The cost of ALG can be broken down into the costs of the different types of services it has made. These are the primary services, buy services, non-buy tail services and normal services. Let the costs of these services be P, B, T and N respectively, then

$$\begin{aligned}
\mathrm{ALG} &= P + B + T + N \\
&\leq 4P + B + N && \text{(Lemma B.1.4)} \\
&\leq 4P + 2N && \text{(Lemma B.1.6)} \\
&\leq 12\,\mathrm{OPT} + 2N && \text{(Lemma B.1.7)} \\
&\leq 20\,\mathrm{OPT} && \text{(Lemma B.1.10)}
\end{aligned}$$

Therefore, we have that $\mathrm{ALG} \leq O(1) \cdot \mathrm{OPT}$ as desired. $\square$