Online Weighted Cardinality Joint Replenishment Problem with Delay

Ryder Chen
The University of Sydney, Australia

Jahanvi Khatkar
The University of Sydney, Australia

Seeun William Umboh
The University of Sydney, Australia

Abstract

We study a generalization of the classic Online Joint Replenishment Problem (JRP) with Delays that we call the Online Weighted Cardinality 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. The cost of serving a set of requests depends on the item types ordered. Furthermore, each request incurs a delay penalty while it is left unserved. The objective is to minimise the total service and delay costs. In the Weighted Cardinality JRP, each item type has a positive weight and the cost of ordering is a non-decreasing, concave function of the total weight of the item types ordered. This problem was first considered in the offline setting by Cheung et al. (2015) but nothing is known in the online setting. Our main result is a deterministic, constant competitive algorithm for this problem.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Online Algorithms, Delay, Joint Replenishment Problem

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.40

Category Track A: Algorithms, Complexity and Games

1 Introduction

The Joint Replenishment Problem (JRP) is a class of optimisation problems that are fundamental to inventory management theory. The problem involves a sequence of requests on items that arrive at various times. Serving a set of requests incurs a cost that is determined by a given cost function that depends on the set of items the requests are on. 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 more general 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 was 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 [18], where items must be made to serve some request and cannot be held in inventory. In this paper, we consider the online setting. Here, requests arrive over time together with their deadline or delay functions and at any point in time, the algorithm may choose to serve some set of requests.

In the Classic 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 problem captures the well-known TCP Acknowledgement Problem. Buchbinder et al. [15] gave a 3-competitive algorithm for

49th International Colloquium on Automata, Languages, and Programming (ICALP 2022).
Editors: Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff;
Article No. 40; pp. 40:1–40:18


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Online Classic JRP with Delays and a lower bound of 2.64. Then, Bienkowski et al. [10] improved the lower bound to 2.754 and showed that the optimal competitive ratio for Online Classic JRP with Deadlines is 2.

With the competitive ratio of Online Classic JRP with Delays mostly settled, there has been a lot of interest in generalisations of the problem where different costs of serving requests are considered. One of these generalizations is Multi-Level Aggregation [9, 14] where we are given a rooted tree T with node costs and requests arrive at nodes of the tree. The cost of serving a set of requests is the cost of the subtree of T induced by the root and the nodes corresponding to the requests. Note that when T is depth 2, this captures JRP. The current best upper bounds for this problem depends on either the depth of T [14, 7], or are logarithmic in the number of vertices of T [8]. No super-constant lower bounds are known.

This problem was later generalized to Online Network Design with Delays [8] where we are given a graph G with edge/node costs and receive connectivity requests over time. The cost of serving a set of connectivity requests is the cost of the subgraph of G that satisfies the requests. This captures a wide class of problems such as Multi-Level Aggregation and Set Cover with Delay. Azar and Touitou [8] provided a framework to reduce a network design problem with delay or deadline to the classic offline variant without delay or deadline while incurring a logarithmic loss in the competitive ratio. Using this framework, they gave polylogarithmic-competitive algorithms for many network design problems with delays or deadlines. Recently, Touitou [30] recently showed that a sub-logarithmic competitive ratio is not possible for Online Network Design with Delays in its full generality. Thus, as powerful as this framework is, it cannot be used as is to improve the competitive ratios for problems such as Multi-Level Aggregation. For these problems, we will need to take advantage of the special structure of these problems to improve their competitive ratios.

In this paper, we introduce a natural generalisation of Online Classic JRP with Delays called the Online Weighted Cardinality JRP with Delays and show that the Azar-Touitou framework can be refined to give a constant competitive algorithm for the problem. In the Online Weighted Cardinality JRP with Delays, each item type i has an associated weight w_i and we are also given a non-negative, non-decreasing concave function f. The cost of serving a set of requests on a set λ of item types is $f(\sum_{i\in\lambda} w_i)$. The concave cost function captures a natural type of economies-of-scale in the real-world production of goods. It has also been considered in other optimization problems such as buy-at-bulk network design (see Section 1.2 for details). The special case of unit weights (called Cardinality JRP) was first studied by Cheung et al. [17] in the offline setting, and they gave a 5-approximation algorithm. We remark that this problem captures the Classic JRP by setting f to be the affine function $f(x) = K_0 + x$ and the weights $w_i = K_i$.

The main technical result of this paper is a constant-competitive algorithm for the case of unit weights.

▶ **Theorem 1.** *There is an $O(1)$-competitive, deterministic polynomial time algorithm for Online Cardinality JRP with Delay.*

For the weighted variant, we design a pseudo-polynomial time reduction from Weighted Cardinality JRP to Cardinality JRP to get the following result.

▶ **Theorem 2.** *There exists an $O(1)$-competitive, deterministic algorithm for Online Weighted Cardinality JRP with Delay.*

1.1 Our Techniques

We now outline the key ideas behind our algorithm for Online Cardinality JRP with Delay. First, as is common when dealing with concave functions, we focus on the special case where the function $f(x)$ is the minimum of n affine functions $g_i(x) = \sigma_i + \delta_i x$, and the σ_i's are geometrically increasing with i while the δ_i's are geometrically decreasing with i. Moreover, when we make a service, we specify which of these affine functions we use to pay for the service. When we make a service λ using g_i, we call it a *level* i service. The cost of the service is $g_l(\lambda) = \sigma_i + \delta_i|\lambda|$; we call σ_i the *shared cost* of the service and δ_i its *individual cost*. We call this the *Piecewise Cardinality JRP*. In the remainder of this section, we discuss our approach for Piecewise Cardinality JRP.

In Classic JRP, the main challenge is to balance the competing demands of aggregating requests into few orders to minimize the number of services and hence the total joint ordering costs incurred, and of aggregating requests on the same item types to minimize the total item ordering costs incurred. 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. On the other hand, 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.

Our algorithm is inspired by the Azar-Touitou framework [8] and augmented with ideas from Gupta et al. [26] to decide which level we should make services at. In fact, we also show that applying the Azar-Touitou framework directly leads to a logarithmic competitive ratio at best (see full version). We first discuss how to handle the simpler deadline setting.

In the deadline setting, requests have levels which are initialised to 1 on arrival. When an unserved request of level j reaches a deadline, it triggers a level j service. We then upgrade its level to $j + 1$ if there are sufficient level j services made recently whose total individual cost can pay for the shared cost of the upgraded services. This makes sense because if there are too many level j services in a relatively small time period then the optimal solution could have aggregated the requests served by these requests into fewer higher-level services and incurred lower cost overall. While this seems to lead to a competitive ratio that depends on the number of levels, by making a careful choice of which level j services to charge to, we are able to achieve a constant competitive ratio. Once we have decided on the level of the service, say at level j', we set the budget of the service to its shared cost $\sigma_{j'}$ and serve unserved requests of level at most j' in ascending order of their deadline until the individual cost of the service reaches or exceeds the budget.

To generalise the algorithm to the delay setting, we adapt and extend the idea of investments used by [8]. 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 [8] 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 by recent services and use this bound to determine when we update levels.

Our analysis uses similar ideas to [8] 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. Using our extended notion of investments, we introduce a novel charging argument. 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.

1.2 Related work

Multi-Level Aggregation was first studied by Bienkowski et al. [9] who gave an algorithm whose competitive ratio is exponential in the depth D of the tree T. This was later improved to $O(D)$ for the deadline setting by Buchbinder et al. [14] and then to $O(D^2)$ for the general delay model by Azar and Touitou [7]. The framework of Azar-Touitou [8] yields a competitive ratio that is logarithmic in the number of vertices of T.

Online Network Design with Delays was first studied by Azar and Touitou [8]. They proved polylogarithmic competitive ratios for many network design problems with delay and gave a $\Omega(\sqrt{log|V|})$ lower bound for the case of online node-weighted Steiner tree with delay and online directed Steiner tree with delay. This was recently improved to $\Omega(\log|V|/\log\log|V|)$ by Touitou [30].

A related problem is Set Cover with Delays where we are given a universe of elements and a collection of sets with costs, requests arrive on elements. The cost of serving a set of requests is the min-cost set cover for the corresponding set of elements. This problem was first studied by Carrasco et al. [16]. They gave a $O(\log N)$-competitive algorithm (where N is the number of requests) and proved a matching lower bound. Later, Azar et al. [3] gave an algorithm that is polylogarithmic in the number of sets and elements.

Two other online problems with delay that have received a lot of attention are matching [19, 1, 20, 12, 4, 13, 11, 6] and k-server [5, 13, 7, 24]. In Matching with Delays, requests arrive on points of a metric space and accumulate delay until they are matched. The objective is to minimize the length of the matching and the total delay cost. In the k-Server with Delays problem, we have k servers in a metric space and requests arrive on points of the metric space. A request is served by moving a server to its location. The goal is to minimize the total distance traveled by the servers and the delay incurred by the requests.

Concave cost functions have been widely-studied in the network design literature, both in the offline and online settings. The problem that is most closely relevant to our paper is Offline Single-Sink Buy-at-Bulk Network Design. We are given an undirected graph $G = (V, E)$ with edge lengths d_e, a concave cost function f, a sink t and a set of sources s_i. The cost of routing x_e units of flow on edge e is $f(x_e) \cdot d_e$. The total cost is the sum of the routing cost over all edges. The goal is to route one unit of flow between from each source s_i to the sink t with minimum total cost. The problem is known to be NP-hard and admits constant-factor approximation algorithms [28, 23, 21, 29, 25, 22, 27]. In the online setting, the sources arrive one-by-one. For the online problem, there is a tight deterministic $O(\log k)$-competitive algorithm [2, 31, 26].

2 Preliminaries

As mentioned in the Introduction, we will be mainly dealing with Piecewise Cardinality JRP with Delay. The Piecewise Cardinality JRP is a special case 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. Requests will then arrive over time and upon arrival, they have an associated deadline or delay function that is revealed.

When a service with cost $\sigma_i + \delta_i|\lambda|$ for some i is made, 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. Equivalently, we can also require the solution to specify for each service not only the requests it serves but also the level it serves it in. This is a more useful formulation that we will be using in the sequel.

For the delay variant, which is the variant under which we are studying this problem, 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 [8] to ensure that all requests must eventually be served.

Using standard techniques for dealing with concave functions we can reduce Cardinality JRP to Piecewise Cardinality JRP losing only a constant factor in the approximation ratio. We defer the details to the full version.

3 Piecewise Cardinality JRP with Delay

We now prove the following theorem. Omitted proofs can be found in the full version.

▶ **Theorem 3.** *There is a deterministic $O(1)$-competitive algorithm for Piecewise Cardinality JRP with Delay.*

3.1 Algorithm Intuition

We first introduce some terminology. We will say a request is *active* if it has arrived and is unserved. We will also assign each request a *request level* which is initially set to 1 upon arrival and is updated as the algorithm progresses. We say that an request is *eligible* for a level l service at time t if it is active and has level at most l at time t. Our algorithm allows services to incrementally pay off the delay that requests have accumulated in the past and may accumulate in the future; we call the latter an *investment cost*. In particular, we say that a service λ at time t *invests* an amount x into request q when λ pays off x amount of the delay cost that may be incurred by q after t. The *residual delay* of a request q at time t is defined as the amount of delay accumulated by the request up to time t that has not been paid for by some service.

When do we make a service? A level l service λ is *triggered* when the set of requests E eligible for a level l service accumulate a total residual delay of σ_l; we say that E are the eligible requests of λ. 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.

What level service should we make? When a level l service λ is triggered at time t, we first check if we can upgrade it to $l+1$. Roughly speaking, we upgrade to level $l+1$ if the amount invested by recent level l services so far is at least σ_{l+1}. This invariant allows us to bound our investment costs. We then proceed to the next step of deciding which requests to serve.

Which requests to serve? Next, the algorithm uses an investment process to decide which requests to serve. For each item type we add to our service, we incur an additional cost of δ_l. The investment process continually invests into the future delay of eligible requests. Once the total investment into requests of an item type i, from this service as well as prior services, reaches δ_l, we add i to the service, reset the counter to 0 and stop investing in requests of type i. The entire investment process stops once the total amount invested by the service reaches σ_l or the service contains the item types of all eligible requests, and we make the service. Intuitively, this investment process allows us to serve requests by order of urgency. Urgent requests can be interpreted as those that accumulate delay faster and hence will have investment counters that reach the required service threshold faster.

3.2 Algorithm Description

We now formally describe the algorithm. Our algorithm maintains the following information.

Service and request pointers. Each request and service will be assigned a pointer that points to some service. This will be specified in more detail later on. We classify services into the following types.

▶ **Definition 4** (Service types). *A service λ is:*

- *a* primary service *if it does not point to any service;*
- *an* upgrade service *if it is triggered initially at level l and the algorithm decided to upgrade it to level $l+1$;*
- *a* tail service *if it is neither a primary nor an upgrade service and no service points to λ;*
- *a* normal service *if it is not one of the above types, i.e. it is not a primary nor an upgrade service and there is a service pointing to it.*

Note that a tail service can become a normal service later on.

Investment counters. The algorithm maintains an investment counter $counter(l, i)$ for level l and each item type i. This counter keeps track of the amount invested into requests of type i since the last level l service containing type i or the beginning of time if there is no such service.

Investment intervals. To decide whether to upgrade service, the algorithm needs to keep track of the investments made by recent normal services. At the end of each service λ that is neither primary nor upgrade, for each eligible request q, the algorithm creates a *investment interval* $[t, \tau]$ on q with cost equal to the amount that λ invested in q. Here, t is the service time of λ. The interval also has a level, which is the level of the service. The details of the investment process and the definition of τ will be specified later.

Our algorithm consists of the following components:

Initialisation. Before any requests arrive, the investment counters $counter(l, i)$ are initialised to 0. When a request q arrives, we set its level to 1, its pointer to NULL.

Serving requests. A level l service λ is triggered when active requests of level at most l accumulate a total of σ_l residual delay. The algorithm then proceeds through the following four steps:

Step 1: Setting service pointer. The *triggering requests* for λ are the active level l requests with positive residual delay. The service pointer of λ 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. Otherwise, the service pointer will be set to be any of the non-NULL triggering request pointers. As will be shown later (Observation 8), 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.

Step 2: Determining the service level. When a level l service λ is triggered, if it is non-primary and not of the highest level n already then it is eligible for a level upgrade. To determine whether to upgrade the service, we use the following notion of witness sets.

▶ **Definition 5** (Witness sets). *Let λ be a level l service and a_q be the earliest arrival time among its eligible requests. The* witness set *of λ is the set W_λ of level l investment intervals that begin after a_q and were created by previous level l normal services. The cost of the witness set W_λ is the total cost of the investment intervals in it and is denoted by $c(W_\lambda)$.*

If $c(W_\lambda) \geq \sigma_{l+1}$, we upgrade λ's level to $l+1$; otherwise, it stays at level l. 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 1 gives the pseudocode for Steps 1 and 2.

Step 3: 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 pay off all residual delay since their arrival time, that is, $r_q(a_q, t)$.

Then the service begins 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 initializing the following variables: 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 also add this amount to a variable $I_q(\lambda)$ which keeps track of how much λ has invested in the request q and will be used later to construct our investment 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. This process is captured in Algorithm 2.

Note that it is possible for Q_λ to be empty.[1] In this case, we say that λ is an *empty service*. The delay costs paid off by the algorithm remains paid but it does not pay the shared or individual cost.

Step 4: Updating request pointers, levels, investment counters and investment intervals. At the end of the service, all eligible requests for λ left unserved have their pointers set to λ and levels set to l. 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 investment interval $[t, \tau]$ for this request with cost $I_q(\lambda)$ which is how much λ invested in the request. Note that all eligible requests for this service will have the same 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 2.

Algorithm 1 Procedure to handle triggering and upgrading services.

Function *OnTrigger(level l)*
- Start a new service λ at current time t;
- `/* determine service pointer using triggering requests */`
- Let E_λ be all eligible requests;
- Let $Q_{trigger} \subseteq E_\lambda$ be those with positive residual delay and level l;
- $pointer(\lambda) \leftarrow pointer(q)$ for an arbitrary request $q \in Q_{trigger}$;
- `/* upgrade the service level if possible */`
- **if** $pointer(\lambda) \neq NULL$ *and* $l \neq n$ **then**
  - Let a_q be the earliest arrival time among requests in E_λ;
  - Let W_λ be the set of investment intervals created by level l normal services and begin after a_q;
  - **if** $\sum_{c \in W_\lambda} cost(c) \geq \sigma_{l+1}$ **then**
    - $l \leftarrow l + 1$;
    - Update E_λ to be the eligible requests for level $l + 1$;
- $level(\lambda) \leftarrow l$;
- **MakeService**(λ, t);

3.3 Analysis

We will bound the costs of 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, this will enable us to bound the total cost of our algorithm. In the following, we abuse notation and use ALG to denote both the algorithm's solution and its cost, and OPT to denote both the optimal solution and its cost.

We first examine the structure and properties of our solution in Section 3.3.1. In particular, we will argue that the directed graph induced by the services and service pointers consist of node-disjoint directed paths and that all non-NULL pointers in a set of triggering requests must be the same.

Next, we look at the structure of our costs in Section 3.3.2. Most importantly, we will introduce the notion of *charged costs* which represent the costs of our services that need to be charged to OPT and show that the charged cost of a level l service is at most $3\sigma_l$.

[1] This can happen, for instance, if there are many eligible requests on many different item types so the service investment counter reaches σ_l before any of the $counter(l, i)$ reaches δ_l.

Algorithm 2 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 */`
- **foreach** $q \in E_\lambda$ **do**
  - Pay off the residual delay $r_q(a_q, t)$;
  - `/* set up counter to track how much` λ `invests in each request */`
  - $I_q(\lambda) \leftarrow 0$;
- `/* Investment Phase */`
- $l \leftarrow level(\lambda)$; $\tau \leftarrow t$; $t' \leftarrow t$; $Q_\lambda \leftarrow \emptyset$; $invested \leftarrow 0$;
- Let $E_{\lambda,i}$ be the set of eligible requests of item type i;
- **while** $Q_\lambda \neq E_\lambda$ *and* $invested < \sigma_l$ **do**
  - Continuously increment τ until either $invested + \sum_{q \in E_\lambda \setminus Q_\lambda} r_q(t', \tau) = \sigma_l$, or for some type i not in Q_λ, $counter(l,i) + \sum_{q \in E_{\lambda,i} \setminus Q_\lambda} r_q(t', \tau) = \delta_l$;
  - `/* Update investment counters and` t' `*/`
  - $invested \leftarrow invested + \sum_{q \in E_\lambda \setminus Q_\lambda} r_q(t', \tau)$;
  - **foreach** *item type* i **do**
    - $counter(l,i) \leftarrow counter(l,i) + \sum_{q \in E_{\lambda,i} \setminus Q_\lambda} r_q(t', \tau)$;
  - **foreach** $q \in E_\lambda \setminus Q_\lambda$ **do**
    - $I_q(\lambda) \leftarrow I_q(\lambda) + r_q(t', \tau)$;
    - Pay off the residual delay $r_q(t', \tau)$;
  - $t' \leftarrow \tau$;
  - **if** *for some type* i *not in* Q_λ, $counter(l,i) = \delta_l$ **then**
    - $Q_\lambda \leftarrow Q_\lambda \cup E_{\lambda,i}$;
    - $counter(l,i) \leftarrow 0$;
- Serve Q_λ;
- `/* update pointers and levels of unserved eligible requests */`
- **foreach** $q \in E_\lambda \setminus Q_\lambda$ **do**
  - $level(q) \leftarrow l$ and $pointer(q) \leftarrow \lambda$;
- `/* construct investment intervals */`
- **if** λ *is not a primary or upgrade service* **then**
  - **foreach** $q \in E_\lambda$ **do**
    - Construct a level l investment interval $[t, \tau]$ on q with cost $I_q(\lambda)$;

Section 3.3.3 will then begin our charging argument by showing that the charged cost of all services can be bounded by the charged costs of the primary and normal services. This is done by bounding the charged cost of the tail services by the charged cost of the primary and upgrade services and then charging the charged cost of the upgrade services to the charged cost of the normal services.

Finally, we finish our charging argument in Section 3.3.4 by charging the charged costs of the primary and normal services to OPT. This is the crux of our analysis. The charged cost of primary services is charged to OPT using a disjointness argument. The charged cost of the normal services is charged to the cost of our investment intervals. These investment intervals' costs are then charged to OPT by showing that for any service λ^* made by OPT and any request q served in λ^*, the total cost of q's investment intervals can be charged to the costs of λ^*.

To simplify our analysis, we assume that at most one service can be triggered at any given time. This is without loss of generality since the delay functions can be perturbed by an infinitesimal amount to ensure this holds.

3.3.1 Analysis: Solution Structure

We begin our analysis by proving properties of ALG's solution structure.

▶ **Observation 6.** ALG *serves all requests eventually. Moreover, when a request q is served at time t, the algorithm would have paid off its delay up till at least time t by the end of the service.*

Proof. We first argue that ALG serves all requests. This is because the delay on each request tends to infinity with time so each request 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. The second part of the observation follows from the fact it was eligible for the service it was served in, and the service would have ensured that the delay of q up till time t has been paid off entirely. ◀

The following lemma implies that service pointers belonging to services of the same level are, in a sense, non-overlapping.

▶ **Lemma 7.** *Suppose a service λ' points to another service λ and their service times are $t' > t$, respectively. Let l be the level of λ. Then, there cannot be a service of level at least l made between t and t'.*

Proof. Suppose, towards a contradiction, that there is a level l service λ_0 made between t and t'. By definition of pointers, there exists a triggering request q for λ' that pointed to λ at the start of the service λ'; moreover, q is active and has level l between t and t'. Thus, q is eligible for λ_0. Since λ_0 did not serve q, it must have overwritten q's pointer and so q would not have pointed to λ at the start of λ', a contradiction. ◀

▶ **Observation 8.** *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. Consider any level l service λ. 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 λ is triggered. Hence, all the requests whose residual delay contributed to triggering λ would have a level less than l and since $\sigma_{l-1} < \sigma_l$, a lower level service should have been triggered instead. Therefore, λ should never have been triggered in the first place.

Next we show that all triggering requests with a non-NULL pointer must point to the same service. Let q_1 be the triggering request whose pointer λ_1 was used as the service pointer for λ and let q_2 be another triggering request with pointer λ_2. Since q_1 and q_2 are level l requests, they must point to level l services. Moreover, the pointer of q_2 must have been set after λ_1 but before λ since otherwise it would have been eligible for λ_1 and been changed. Then, Lemma 7 implies that λ_2 must in fact be λ_1. ◀

From Observation 8, 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 show that the directed graph induced by the services and service pointers consist of node-disjoint directed paths, which we call *service chains*.

▶ **Lemma 9.** *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 there are two services λ_1 and λ_2 that point at λ'. By construction of service pointers, λ_1 and λ_2 must have been level l when they were triggered. By Lemma 7, we get that λ_1 and λ_2 must be the same service. Thus, each service can be pointed to by at most one other service. ◀

The following lemma implies that at any time, only the last service of a level can become normal in the future, all previous services of the level that are tail must remain tail. This ensures that our witness sets will not exclude any investment intervals that belong to tail services that later become normal services which is required later to prove Lemma 24.

▶ **Lemma 10.** *Let λ_1 and λ_2 be two level l services with service times $t_1 < t_2$, respectively. If λ_1 is tail at time t_2, then it will stay a tail service at all future times $t > t_2$.*

Proof. Suppose, towards a contradiction, that λ_1 became normal later on. Then, at some time $t_3 > t_2$, a level l service λ_3 was triggered and λ_3 pointed to λ_1. However, this contradicts Lemma 7 so λ_1 must stay a tail. ◀

3.3.2 Analysis: Cost Structure

We now analyse the properties of different costs incurred by our algorithm.

▶ **Definition 11** (Triggering, investment and charged costs). *For each level l service λ, we define the following three costs: (1) its* triggering delay cost *which is the residual delay on all eligible requests that is paid off at the beginning of the service; (2) its* investment cost *which are the delay costs invested in during the λ's investment phase; (3) its* charged cost *which is its shared cost plus its triggering delay cost plus its investment cost.*

Note that the algorithm may make an "empty service" in which no requests are served, in which case its charged cost is its triggering delay cost and its investment cost. We will first show that in order to bound the cost of ALG, it suffices to look at the charged costs.

▶ **Lemma 12.** *The total cost of* ALG *is at most twice the total charged cost of its services.*

Proof. The total cost of ALG is the total shared cost of its services plus the total individual cost of its services plus its delay cost. It suffices to bound the latter two costs in terms of total investment and triggering delay costs. A level l service only serves an item type i when the level l investment counter for i reaches δ_l and the counter is reset to 0 when it reaches δ_l. So, the total individual cost is at most the total investment cost. Observation 6 implies that the delay cost of ALG is at most the triggering delay cost plus investment cost. Putting the above bounds together gives the lemma. ◀

Next, we will bound the charged cost of any service which will later allow us to charge these costs to OPT.

▶ **Observation 13.** *The triggering delay costs incurred by a level l service is at most σ_l.*

Proof. Let λ be a level l service made by ALG. If λ is not an upgrade service, then its triggering delay cost is exactly σ_l, by construction. On the other hand, if λ is an upgrade service, then it was triggered because the active 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 implies that the active 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 14.** *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 13, 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 cost of the service is at most σ_l. Adding these three components up, the total charged cost is at most $3\sigma_l$ as required. ◀

The following lemma bounds the investment cost of normal services which will be used later to charge the costs of the normal services to OPT.

▶ **Lemma 15.** *Every normal service of level l has investment cost σ_l.*

Proof. The investment cost of a level l normal service λ is exactly σ_l. This is because by construction, the investment cost of a level l service is at most σ_l and can only be less if λ served all eligible requests. However, that would mean that λ is a tail service since there are no unserved requests pointing to it. ◀

3.3.3 Analysis: Bounding charged costs

We now begin our charging argument by showing that the charged costs of ALG can be bounded by the charged costs of the primary and normal services.

▶ **Lemma 16.** *The total charged costs of the tail services are at most 3 times the total charged costs of the primary and upgrade services.*

Proof. Consider any tail service λ of level l. By definition, it ends a chain that has reached level l and so the chain has an earlier service λ' that is a level l upgrade service or a primary service, in the case that $l = 1$. The service λ' has an investment cost of exactly σ_l as otherwise the service would have served all eligible requests and the chain would have ended already. Therefore, λ' has a charged cost of at least σ_l. By Lemma 14, λ's charged cost is at most 3 times that of λ'. Lastly, Lemma 9 implies that each primary or upgrade service is charged by at most one tail 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. ◀

We now bound the charged cost of the upgrade services by the charged cost of the normal services. In order to do so, we first show that the witness sets of our upgrade services must be pairwise disjoint.

▶ **Lemma 17.** *An 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 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 $t_1 < t_2$.

By construction, the interval $[t, \tau]$ was created by a service at time t. Since it belongs to the witness set of λ_1, it must have been created before t_1, and thus, $t < t_1$. Now consider λ_2. In order for $[t, \tau]$ to belong to its witness set, λ_2 must have an eligible request q_2 that arrived before t. 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 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). This contradicts the fact that q_2 is eligible for λ_2. See Figure 1 for an illustration of this proof. ◀

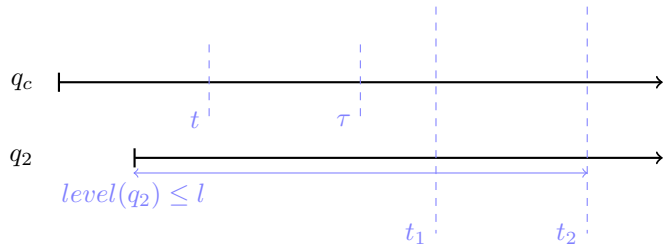

Figure 1 Illustration of the proof of Lemma 17.

▶ **Lemma 18.** *The total charged costs of the upgrade services is at most 3 times the total charged costs of the normal services.*

Proof. Let λ be a level l upgrade service at time t. By construction, the cost of its witness set W_λ is at least σ_l. By Lemma 14, λ has a charged cost of at most $3\sigma_l \leq 3cost(W_\lambda)$. Using Lemma 17, we get that the total charged costs of upgrade services is at most 3 times the total investment costs made by normal services. Since the investment costs of the normal services are at most the charged costs of the normal services, the lemma follows. ◀

3.3.4 Analysis: Charging to OPT

Having bounded the cost of ALG by the charged costs of the primary and normal services, it remains to charge these costs to OPT.

▶ **Lemma 19.** *The charged cost of the primary services is at most* $3\,\mathrm{OPT}$.

Proof. Let Λ_1 be the set of primary services made by ALG. For each primary service $\lambda \in \Lambda_1$, let I_λ denote the interval $[a_\lambda, t_\lambda]$ where a_λ is the earliest arrival time among λ's triggering requests and t_λ is the service time of λ. By definition of primary services, the set of intervals $\mathcal{I} = \{I_\lambda\}_{\lambda \in \Lambda_1}$ are pairwise disjoint.

Consider an interval $I_\lambda \in \mathcal{I}$. If OPT made a service λ^* during I_λ, the shared cost of λ^* is at least σ_1. On the other hand, if OPT did not make a service during I_λ, then it must have incurred a total delay of at least σ_1 on the triggering requests of λ since they arrive no earlier than the start of I_λ. In both cases, OPT pays a cost of at least σ_1 during the interval I_λ.

Since the intervals $\mathcal{I}$ are disjoint, we have that $\mathrm{OPT} \geq \sigma_1|\mathcal{I}|$. Thus, by Lemma 14, the charged cost of primary services is at most $3\sigma_1|\mathcal{I}| \leq 3\,\mathrm{OPT}$. ◀

▶ **Lemma 20.** *The charged cost of a normal service is at most 3 times its investment cost.*

Proof. This follows from Lemmas 15 and 14. ◀

Using Lemma 20, 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 which, by definition is equal to the cost of the investment intervals made by normal services. Hence, we will show that the cost of every investment interval made by a normal service can be charged to some distinct cost incurred by OPT.

We will refer to investment intervals created by normal services as *normal investment intervals* and will now analyse the properties of these intervals.

▶ **Lemma 21.** *Let $[t, \tau]$ be a level l normal investment interval created by the normal service λ at time t. Then there cannot be a level $l' \geq l$ service triggered between times t and τ (inclusive) other than λ.*

Proof. Since λ is a normal service, there exists a later service λ_s at time t_s pointing to λ. Thus, there is a request q that is eligible for λ but is left unserved and later became a triggering request for λ_s. By definition of pointers and triggering requests, at time t_s, q points to λ and has positive residual delay. Since λ paid off the residual delay on eligible requests (which includes triggering requests) until time τ, λ_s occurred at time $t_s > \tau$.

Now we show that after t and before t_s, q is level l and there is no service for which q is eligible. Since q points to λ at time t_s, there cannot be a service between t (the service time of λ) and t_s for which q is eligible; otherwise, that service would have changed q's pointer and q would no longer be pointing to λ at time t_s. Thus, the level of q between times t and $\tau < t_s$ is l, the level of λ, and so there is no service of level at least l between t and τ. ◀

▶ **Lemma 22.** *For any request, its normal investment intervals are all disjoint.*

Proof. Assume towards a contradiction there is a request q with two non-disjoint normal 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 so after the service at time t_1, q is of level at least l_1. By Lemma 21, there is no service of level at least l_1 between times t_1 and τ_1 which contradicts our assumption. ◀

▶ **Lemma 23.** *Consider any two level l normal 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 investment intervals $[t_1, \tau_1]$, $[t_2, \tau_2]$ created by distinct normal 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, Lemma 21 implies 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. ◀

We now charge the cost of our normal investment intervals to OPT which will be the crux of our analysis. For this lemma, we will refer to "normal investment intervals" simply as "investment intervals" unless otherwise specified. The full proof of the lemma can be found in the Appendix.

▶ **Lemma 24.** *The total cost of normal investment intervals is at most* $6\,\mathrm{OPT}$.

Proof sketch. 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^*}.

For any request $q \in Q_{\lambda^*}$ we will first consider its investment intervals ending before t^*. These investment intervals ending before time t^* correspond to delay costs incurred prior to time t^* that had been invested in by a normal service and is hence at most the total delay accrued by the request up till time t^*. Since OPT pays the delay for the request q up till time t^*, the cost of the investment intervals ending before time t^* can be charged to the delay costs paid by OPT.

Next we consider the investment intervals belonging to requests in Q_{λ^*} that contain time t^*. There are two cases to consider: the intervals of level $l < l^*$ and the intervals of level $l \geq l^*$. For the intervals of a fixed level $l < l^*$ that intersect with time t^* we know by Lemma 23 that they must have all been created by the same normal service. Since a normal service can only invest a maximum amount of σ_l, the total cost of these intervals of level l must be at most σ_l. Summing up across all levels $l < l^*$ and noting that our σ values form a geometric series, we conclude that the cost of all the intervals with level $l < l^*$ is at most σ_{l^*} which is the shared cost that OPT must pay in serving λ^*. For the intervals of a fixed level $l \geq l^*$ we will consider a fixed item type i. Once again, by Lemma 23, these intervals must have been created by the same normal service and since a normal service can only invest at most δ_l in a particular item type before serving it, the total cost of the intervals at level l and for item i requests is at most δ_l. Summing up across all levels $l \geq l^*$ and noting that our δ values form a geometric series, we conclude that the cost of all item i intervals of level $l \geq l^*$ is at most $2\delta^*$ which is 2 times the individual cost paid by λ^* to serve the item type i. This argument applies to all item types served by λ^*.

Lastly, we consider the investment intervals belonging to requests in Q_{λ^*} that begin after time t^*. Once again we consider two cases: the intervals of level $l < l^*$ and the intervals of level $l \geq l^*$. For the intervals of level $l < l^*$, we consider a fixed level l and look at the latest starting interval at time t. There must have been a normal service at time t that created this interval and this implies that the witness set of this normal service cost less than σ_{l+1} since it was a normal service as opposed to an upgrade service. Since all requests in Q_{λ^*} must arrive before t^*, the witness set of the service at time t is a superset of the intervals of level l beginning after time t^*. Moreover, the intervals created by the service at time t must cost at most σ_l since the service can only invest at most σ_l. Overall, the intervals of level l that begin after time t^* cost at most $\sigma_{l+1} + \sigma_l < 2\sigma_{l+1}$. Summing up over all levels $l < l^*$ and using the geometric property of σ, we get a total investment interval cost of $4\sigma_l^*$ which is 4 times the shared cost paid by λ^*. For the intervals of level $l \geq l^*$ we once again fix a level $l \geq l^*$ and item type i. The cost of each of these item i, level l intervals must be added to the same investment counter without the counter being reset to 0. This is because all requests in Q_{λ^*} arrive before t^*, so if at any time after t^*, the level l investment counter for item i is reset to 0, all requests on item i and of level at most l must have been served and thus there is no way for a level l intervals to contribute to an investment counter after it has been reset. Since the investment counter by design has a maximum value of δ_l, this implies that the total cost of the level l, item i investment intervals is at most δ_l. Fixing the item i, summing up across all levels $l \geq l^*$ and utilising the geometric property of δ, we get that the investment intervals of item i and of level $l \geq l^*$ cost at most $2\delta_{l^*}$ which is 2 times what λ^* pays for the item i. We can once again apply this argument for all item types i served by λ^*.

Hence, it follows that for any service λ^*, the cost of the investment intervals in Q_{λ^*} can be charged to the cost of λ^* and by applying this argument across all services made by OPT, the lemma follows. ◀

▶ **Lemma 25.** *The charged cost of the normal services is at most* 18 OPT.

Proof. This follows from Lemmas 20 and 24. ◀

We conclude by combining the above ingredients to show that the algorithm is constant-competitive.Let P, U, T, N denote the charged service costs of primary services, upgrade services, tail services and normal services, respectively. Using Lemma 12 and the fact that the total charged service cost is $P + U + T + N$, we get

$$\begin{aligned}
\text{ALG} &\leq 2(P + U + T + N) \\
&\leq 2(4P + 4U + N) && \text{(Lemma 16)} \\
&\leq 2(4P + 13N) && \text{(Lemma 18)} \\
&\leq 24\,\text{OPT} + 26N && \text{(Lemma 19)} \\
&\leq 492\,\text{OPT} && \text{(Lemma 25)}
\end{aligned}$$

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

References

1. Itai Ashlagi, Yossi Azar, Moses Charikar, Ashish Chiplunkar, Ofir Geri, Haim Kaplan, Rahul Makhijani, Yuyi Wang, and Roger Wattenhofer. Min-cost bipartite perfect matching with delays. In Klaus Jansen, José D. P. Rolim, David Williamson, and Santosh S. Vempala, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, volume 81 of *LIPIcs*, pages 1:1–1:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.APPROX-RANDOM.2017.1.
2. Baruch Awerbuch, Yossi Azar, and Yair Bartal. On-line generalized steiner problem. *Theor. Comput. Sci.*, 324(2-3):313–324, 2004. doi:10.1016/j.tcs.2004.05.021.
3. Yossi Azar, Ashish Chiplunkar, Shay Kutten, and Noam Touitou. 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, 2020. doi:10.4230/LIPIcs.ESA.2020.8.
4. Yossi Azar and Amit Jacob Fanani. Deterministic min-cost matching with delays. *Theory Comput. Syst.*, 64(4):572–592, 2020. doi:10.1007/s00224-019-09963-7.
5. Yossi Azar, Arun Ganesh, Rong Ge, and Debmalya Panigrahi. Online service with delay. *ACM Trans. Algorithms*, 17(3):23:1–23:31, 2021. doi:10.1145/3459925.
6. Yossi Azar, Runtian Ren, and Danny Vainstein. The min-cost matching with concave delays problem. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 301–320. SIAM, 2021. doi:10.1137/1.9781611976465.20.
7. Yossi Azar and Noam Touitou. General framework for metric optimization problems with delay or with deadlines. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 60–71. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00013.
8. Yossi Azar and Noam Touitou. 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, 2020. doi:10.1109/FOCS46700.2020.00129.
9. Marcin Bienkowski, Martin Böhm, Jaroslaw Byrka, Marek Chrobak, Christoph Dürr, Lukáš Folwarczný, Lukasz Jez, Jiri Sgall, Nguyen Kim Thang, and Pavel Veselý. 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, 2016. doi:10.4230/LIPIcs.ESA.2016.12.

10 Marcin Bienkowski, Jaroslaw Byrka, Marek Chrobak, Lukasz Jez, Dorian Nogneng, and Jirí Sgall. Better approximation bounds for the joint replenishment problem. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 42–54. SIAM, 2014. doi:10.1137/1.9781611973402.4.

11 Marcin Bienkowski, Artur Kraska, Hsiang-Hsuan Liu, and Pawel Schmidt. A primal-dual online deterministic algorithm for matching with delays. In Leah Epstein and Thomas Erlebach, editors, *Approximation and Online Algorithms - 16th International Workshop, WAOA 2018, Helsinki, Finland, August 23-24, 2018, Revised Selected Papers*, volume 11312 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2018. doi:10.1007/978-3-030-04693-4_4.

12 Marcin Bienkowski, Artur Kraska, and Pawel Schmidt. A match in time saves nine: Deterministic online matching with delays. In Roberto Solis-Oba and Rudolf Fleischer, editors, *Approximation and Online Algorithms - 15th International Workshop, WAOA 2017, Vienna, Austria, September 7-8, 2017, Revised Selected Papers*, volume 10787 of *Lecture Notes in Computer Science*, pages 132–146. Springer, 2017. doi:10.1007/978-3-319-89441-6_11.

13 Marcin Bienkowski, Artur Kraska, and Pawel Schmidt. Online service with delay on a line. In Zvi Lotker and Boaz Patt-Shamir, editors, *Structural Information and Communication Complexity - 25th International Colloquium, SIROCCO 2018, Ma'ale HaHamisha, Israel, June 18-21, 2018, Revised Selected Papers*, volume 11085 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2018. doi:10.1007/978-3-030-01325-7_22.

14 Niv Buchbinder, Moran Feldman, Joseph (Seffi) Naor, and Ohad Talmon. 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, 2017. doi:10.1137/1.9781611974782.80.

15 Niv Buchbinder, Tracy Kimbrel, Retsef Levi, Konstantin Makarychev, and Maxim Sviridenko. Online Make-to-Order Joint Replenishment Model: Primal-Dual Competitive Algorithms. *Oper. Res.*, 61(4):1014–1029, 2013. doi:10.1287/opre.2013.1188.

16 Rodrigo A. Carrasco, Kirk Pruhs, Cliff Stein, and José Verschae. The online set aggregation problem. In Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro, editors, *LATIN 2018: Theoretical Informatics - 13th Latin American Symposium, Buenos Aires, Argentina, April 16-19, 2018, Proceedings*, volume 10807 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2018. doi:10.1007/978-3-319-77404-6_19.

17 Sin-Shuen Cheung. 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, 2014. doi:10.1007/978-3-319-18263-6_7.

18 Nico P. Dellaert and M. Teresa Melo. Heuristic procedures for a stochastic lot-sizing problem in make-to-order manufacturing. *Ann. Oper. Res.*, 59(1):227–258, 1995. doi:10.1007/BF02031749.

19 Yuval Emek, Shay Kutten, and Roger Wattenhofer. Online matching: haste makes waste! In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 333–344. ACM, 2016. doi:10.1145/2897518.2897557.

20 Yuval Emek, Yaacov Shapiro, and Yuyi Wang. Minimum cost perfect matching with delays for two sources. *Theor. Comput. Sci.*, 754:122–129, 2019. doi:10.1016/j.tcs.2018.07.004.

21 Naveen Garg, Rohit Khandekar, Goran Konjevod, R. Ravi, F. Sibel Salman, and Amitabh Sinha II. On the integrality gap of a natural formulation of the single-sink buy-at-bulk network design problem. In *Integer Programming and Combinatorial Optimization, 8th International IPCO Conference, Utrecht, The Netherlands, June 13-15, 2001, Proceedings*, pages 170–184, 2001. doi:10.1007/3-540-45535-3_14.

22 Fabrizio Grandoni and Giuseppe F. Italiano. Improved approximation for single-sink buy-at-bulk. In *Algorithms and Computation, 17th International Symposium, ISAAC 2006, Kolkata, India, December 18-20, 2006, Proceedings*, pages 111–120, 2006. doi:10.1007/11940128_13.

23 Sudipto Guha, Adam Meyerson, and Kamesh Munagala. A constant factor approximation for the single sink edge installation problem. *SIAM J. Comput.*, 38(6):2426–2442, 2009. doi:10.1137/050643635.

24 Anupam Gupta, Amit Kumar, and Debmalya Panigrahi. Caching with time windows. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 1125–1138. ACM, 2020. doi:10.1145/3357713.3384277.

25 Anupam Gupta, Amit Kumar, and Tim Roughgarden. Simpler and better approximation algorithms for network design. In *35th STOC*, pages 365–372, 2003.

26 Anupam Gupta, R. Ravi, Kunal Talwar, and Seeun William Umboh. 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, 2017. doi:10.1137/1.9781611974782.38.

27 Raja Jothi and Balaji Raghavachari. Improved approximation algorithms for the single-sink buy-at-bulk network design problems. *J. Discrete Algorithms*, 7(2):249–255, 2009. doi:10.1016/j.jda.2008.12.003.

28 F. Sibel Salman, Joseph Cheriyan, R. Ravi, and S. Subramanian. Buy-at-bulk network design: Approximating the single-sink edge installation problem. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana.*, pages 619–628, 1997. URL: http://dl.acm.org/citation.cfm?id=314161.314397.

29 Kunal Talwar. The single-sink buy-at-bulk LP has constant integrality gap. In *Integer Programming and Combinatorial Optimization, 9th International IPCO Conference, Cambridge, MA, USA, May 27-29, 2002, Proceedings*, pages 475–486, 2002. URL: http://link.springer.de/link/service/series/0558/bibs/2337/23370475.htm, doi:10.1007/3-540-47867-1_33.

30 Noam Touitou. Nearly-tight lower bounds for set cover and network design with deadlines/delay. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan*, volume 212 of *LIPIcs*, pages 53:1–53:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ISAAC.2021.53.

31 Seeun Umboh. Online network design algorithms via hierarchical decompositions. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1373–1387, 2015. doi:10.1137/1.9781611973730.91.