Beyond Tree Embeddings – a Deterministic Framework for Network Design with Deadlines or Delay

Yossi Azar
azar@tau.ac.il
Tel Aviv University

Noam Touitou
noamtouitou@mail.tau.ac.il
Tel Aviv University

***Abstract*—We consider network design problems with deadline or delay. All previous results for these models are based on randomized embedding of the graph into a tree (HST) and then solving the problem on this tree. We show that this is not necessary. In particular, we design a deterministic framework for these problems which is not based on embedding. This enables us to provide deterministic poly-log(n)-competitive algorithms for Steiner tree, generalized Steiner tree, node weighted Steiner tree, (non-uniform) facility location and directed Steiner tree with deadlines or with delay (where n is the number of nodes).**

Our deterministic algorithms also give improved guarantees over some previous randomized results. In addition, we show a lower bound of poly-log(n) for some of these problems, which implies that our framework is optimal up to the power of the poly-log. Our algorithms and techniques differ significantly from those in all previous considerations of these problems.

I. Introduction

In online minimization problems with deadlines, requests are released over a timeline. Each request has an associated deadline, by which it must be served by any feasible solution. The goal of an algorithm is to give a solution which minimizes the total cost incurred in serving the given requests.

Another model, which generalizes the deadline model, is that of online problems with delay. In those problems, requests again arrive over a timeline. While requests no longer have a deadline, each pending request (i.e. a request which has been released but not yet served) incurs growing delay cost. The total cost of the algorithm is the cost of serving requests plus the total delay incurred over those requests; the delay cost thus motivates the algorithm to serve requests earlier.

In this paper, we consider classic network design problems in the deadline/delay setting. In the classic (offline) setting of network design, one is given a graph of n nodes and a set of connectivity requests (e.g. pairs of nodes to connect). The input contains a collection of elements (e.g. edges) with associated cost. A request is satisfied by any subset of elements which serves the connectivity request (e.g. a set of edges which connects the requested pair of nodes). A feasible solution for the offline problem is a set of elements which simultaneously satisfies all connectivity requests.

Such an offline network design problem induces an online problem with deadlines/delay as follows. The input graph is again given in advance. The requests, however, arrive over a timeline (with either a deadline or a delay function). At any point in time, the algorithm may choose to *transmit* an offline solution (i.e. a set of elements); each pending request that is served by the transmitted solution in the offline setting is served by this transmission in the online setting. In keeping with previous work on these problems, this paper considers the *clairvoyant* model, in which the deadline of a request – or its future accumulation of delay – is revealed to the algorithm upon the release of the request.

We next discuss such induced network design problems with deadlines/delay that have been previously considered. The usual solution for such problems is to randomly embed the general input into a tree, incurring a distortion to the metric space, then solving the problem on the resulting tree. In this paper, we present frameworks which *bypass* this usual mode of work, enabling improved guarantees, generality and simplicity.

Steiner tree with deadlines/delay: In this problem, requests are released on nodes of a graph with costs to the edges. Serving these requests comprises transmitting a subgraph which connects the request and a designated root node of the graph. This problem was studied in the case in which the graph is a tree – in this case it is called the **multilevel aggregation problem** (first presented in [8]). With D the depth of the input tree, the best known results for multilevel aggregation are $O(D)$ competitiveness for the deadline model by Buchbinder *et al.* [14], and $O(D^2)$ competitiveness for the delay model in [6]. Thus, a simple algorithm for general Steiner tree with deadlines/delay based on metric tree embedding for this problem is to embed a general graph into a tree, and then using the best multilevel aggregation algorithms; in both the deadline and delay

Supported in part by the Israel Science Foundation (grant No. 1506/16 and 2304/20). A full version of this paper can be found at https://arxiv.org/abs/2004.07946.

2575-8454/20/$31.00 ©2020 IEEE
DOI 10.1109/FOCS46700.2020.00129

case, this can be seen to yield $O(\log^2 n)$-competitive randomized algorithms.

Facility location with deadlines/delay: In this problem, presented in [6], the input graph has weights to the edges and facility costs to the nodes. Requests arrive on the nodes of the graph, to be served by transmissions. A transmission consists of a set of facilities U, and a collection of pending requests Q. The transmission serves the requests of Q, and has a cost which is the sum of facility costs of the nodes in U, plus the sum of distances from each request of Q to the closest facility in U. The best known algorithms for both the deadline and delay variants of this problem, also based on tree embedding, are randomized and $O(\log^2 n)$ competitive – but apply only to the uniform problem, where the nodes' facility costs are identical.

This paper introduces a general deterministic framework for solving such network design problems on general graphs, with deadlines or with delay, which does not rely on tree embeddings. This framework obtains improved results to both previous problems, as well as new results for Steiner forest, nonuniform facility location, multicut, Steiner network, node-weighted Steiner forest and directed Steiner tree.

A. Our Results

We now state specifically our results for network design problems with deadlines/delay. Let $\mathcal{E}$ be the collection of elements in an offline network design problem. In this paper, we show the following results.

1) If there exists a deterministic (randomized) γ-approximation for the offline network design problem which runs in polynomial time, then there exists an $O(\gamma \log |\mathcal{E}|)$-competitive deterministic (randomized) algorithm for the induced problem with deadlines, which also runs in polynomial time.
2) If there exists a deterministic (randomized) γ-approximation for the *prize-collecting* variant of the offline network design problem, then there exists an $O(\gamma \log |\mathcal{E}|)$-competitive deterministic (randomized) algorithm for the induced problem with delay, which also runs in polynomial time.

Each of those results is obtained through designing a framework which encapsulates the given approximation algorithm.

We consider several network design problems on a graph of n nodes, which are described in Subsection I-C. Plugging into our frameworks previously-known offline approximations (for either the original or prize-collecting variants) yields the results summarized in Table I. Except for the algorithm for directed Steiner tree (which is randomized and runs in quasi-polynomial time due to the encapsulated approximation), all algorithms are deterministic and run in polynomial time.

Table I
FRAMEWORK APPLICATIONS

	With deadlines	With delay
Multilevel aggregation	$O(\log n)$	$O(\log n)$
Edge-weighted Steiner forest	$O(\log n)$	$O(\log n)$
Multicut	$O(\log^2 n)$	$O(\log^2 n)$
Edge-weighted Steiner network	$O(\log n)$	$O(\log n)$
Node-weighted Steiner forest	$O(\log^2 n)$	$O(\log^2 n)$
Facility location (non-uniform)	$O(\log n)$	$O(\log n)$
Directed Steiner tree	$O\left(\frac{\log^3 n}{\log \log n}\right)$	? [1]

Our frameworks improve on previous results in the following way:

1) For Steiner tree with deadlines/delay, we give $O(\log n)$-competitive deterministic algorithms, while the best previously-known algorithms are randomized and $O(\log^2 n)$-competitive [8], [6].
2) For facility location with deadlines/delay, the best previously-known algorithms are randomized, $O(\log^2 n)$-competitive [6], and apply only for the uniform case (where facilities have the same opening cost). We give $O(\log n)$-competitive, deterministic algorithms which apply also for the non-uniform case.

For node-weighted Steiner forest and directed Steiner tree, our results are relatively close to the optimal solution – in the full version of this paper, we show an $\Omega(\sqrt{\log n})$ lower bound on competitiveness through applying the lower bound of [3] for set cover with delay. As an information-theoretic lower bound, it applies for algorithms with unbounded computational power.

While the common regime in problems with deadlines/delay is that the number of requests k is unbounded and the number of nodes n is finite, we also address the opposite regime in which k is small – the latter being more popular in classic network design problems. We achieve the best of both worlds – namely, we show a modification to the deadline/delay frameworks which replaces n by $\min\{n, k\}$ in the competitiveness guarantees. This modification applies to all problems considered in this paper except for facility location, but conjecture that a similar algorithm would apply there as well. The results for this regime appear in the full version of this paper.

[1] We could find no approximation result for prize-collecting directed Steiner tree. We conjecture that such an approximation algorithm exists which loses only a constant factor apart from the best approximation for the original offline problem, in which case we obtain an identical guarantee to the deadline case.

B. Our Techniques

The **deadline framework** performs services (i.e. transmissions) of various costs; the logarithmic class of the cost of a service is called its level. Pending requests also have levels, which are maintained by the algorithm. Whenever a pending request of level j reaches its deadline, a service of level $j+1$ starts. This service is only meant to serve requests of lower or equal level (we call such requests eligible for the service). After a service concludes, the level of remaining eligible requests is raised to that of the service. Intuitively, this means that once a pending request has seen a service of cost 2^j, it refuses to be served by any cheaper service. This makes use of the aggregation property – higher-cost services tend to be more cost-effective per request.

When a service is triggered, it has to choose which of the eligible requests to serve, subject to its budget constraint. The service prioritizes requests of earlier deadline, adding them until the budget is exceeded. The cost of serving those requests is estimated using the encapsulated approximation algorithm.

The main idea of levels exists in the **delay framework** as well. However, handling general delay functions requires more intricate procedures – namely, for triggering a service and for choosing which requests to serve. The delay framework maintains an *investment counter* for each pending request, which allows a service to pay for the delay of a request (i.e. the delay cost is charged to the budget of the service). A service is started when a large amount of delay for which no service has paid has accumulated on the requests of a particular level j – the started service is of level $j+1$.

When choosing which of the eligible requests to serve, the algorithm considers the first point in time in which an eligible request would accumulate delay which is not paid for by its investment counter. Using its budget of 2^j, it then attempts to push back this point in time farthest into the future – it does so either by raising the investment counters, or by serving requests. The way to balance these two methods is problem-specific – the framework thus formulates a prize-collecting instance, where the penalties represent future delay, and calls the encapsulated prize-collecting approximation algorithm to solve it.

C. Considered Problems

In this paper, we consider the induced deadline/delay problems of several network design problems. We now introduce those problems.

Steiner tree and Steiner forest. In the Steiner forest problem, each request is a pair of terminals (i.e. nodes in the input graph), and the elements are the edges. A request is satisfied by a set of edges if the two terminals of the request are connected by those edges. The Steiner tree problem is an instance of Steiner forest in which the input also designates a specific node as the root, such that every request contains the root as one of its two terminals. A special case of the Steiner tree problem is the multilevel aggregation problem, in which the graph is a tree.

We also consider a stronger variant of the Steiner forest problem, in which each request is a *subset* of nodes to be connected. While this problem is identical to the original Steiner forest in the offline setting (as the subset can be broken down to pairs), their induced deadline/delay problems are substantially different.

Multicut. In the offline multicut problem, each request is again a pair of terminals, and the elements are again the edges. A request is satisfied by a set of edges which, if removed from the original graph, would disconnect the pair of terminals.

As in Steiner forest, it makes sense to define the stronger variant in which each request is a subset of nodes which must be disconnected from each other – while both variants are equivalent in the offline setting, their induced deadline/delay problems are distinct.

Node-weighted Steiner forest. In this problem, the elements are the nodes, rather than edges. Each request is again a pair of terminals, and is satisfied by a solution which contains (in addition to the terminals themselves) nodes that connect the pair of terminals.

Edge-weighted Steiner network. This problem is identical to the Steiner forest problem, except that each request q comes with a demand $f(q) \in \mathbb{N}$. A request is satisfied by a set of edges that contains $f(q)$ edge-disjoint paths between the terminals.

Directed Steiner tree. This problem is identical to the Steiner tree problem, except that the graph is now directed. Each pair request, where one of its terminals is the root, is satisfied by a set of edges that contain a directed path from the root to the other terminal.

Facility location. In the facility location problem, the requests are on the nodes of the graph. The elements are the nodes of the graph, upon which facilities can be opened. The cost of the solution is the total cost of opened facilities (opening cost) plus the distances from each request to the closest facility (connection cost). Note that the connection cost prevents facility location from being strictly compliant to the analysis of the framework we present. However, we nonetheless show that the framework itself applies to facility location as well.

D. Related Work

The classic online consideration of network design problems has been studied in numerous papers (e.g. [22], [20], [7], [24], [21], [1]). In this genre of problems, the connectivity requests arrive one after the other in a sequence (rather than over time), and must be served immediately by buying some elements which serve the request. These bought elements remain bought until the end of the sequence, and can thus be used to serve future requests. This is in contrast to the deadline/delay model considered in this paper, where the elements are *transmitted* rather than bought, and thus future use of these elements requires transmitting them again (at additional cost).

There is no connection between the classic online variant of a problem and the deadline/delay variant – that is, neither problem is reducible to the other. There could be a stark difference in competitiveness between the two models, which depends on the network design problem. For example, in the multilevel aggregation problem, the classic online problem is Steiner tree on a tree, which is trivially 1-competitive (while the best known algorithms for multilevel aggregation with deadlines/delay have polylogarithmic ratio).

The multilevel aggregation problem was first considered by Bienkowski *et al.* [8], who gave an algorithm with competitiveness which is exponential in the depth D of the input tree, for the delay model. This result was then improved, first to $O(D)$ for the deadline model by Buchbinder *et al.* [14], and then to $O(D^2)$ for the general delay model in [6]. These results yield $O(\log^2 n)$-competitive randomized algorithms for Steiner tree with deadlines/delay on general graphs, through metric embeddings; for more general Steiner problems (e.g. Steiner forest, node-weighted Steiner tree) no previously-known algorithm exists.

The multilevel aggregation also generalizes some past lines of work – the TCP acknowledgement problem [17], [23], [15] is multilevel aggregation with $D = 1$, and the joint replenishment problem [16], [13], [9] is multilevel aggregation with $D = 2$.

Another problem studied in the context of delay is that of matching with delay [2], [19], [18], [4], [10], [11]. In this problem, requests arrive on points of a metric space, and gather delay until served. The algorithm may choose to serve two pending requests, at a cost which is the distance between those two requests in the metric space. This problem seems hard without making assumptions on the delay function, and thus is usually considered when the delay functions are identical and linear.

The k-server problem in the deadline/delay context has also been studied [5], [12], [6]. In this problem, k servers exist in a metric space, and requests again arrive on points of the space, gathering delay. To serve a request, the algorithm must move a server to that request, paying the distance between the server and the request.

II. Model and Deadline Framework

We are given a set $\mathcal{E}$ of elements, with costs $c : \mathcal{E} \to \mathbb{R}^+$. Requests are released over time, and we denote the release time of a request q by r_q. Each request has a deadline d_q, by which it must be served. At any point in time, the algorithm may transmit a subset of elements $E \subseteq \mathcal{E}$, at a cost $\sum_{e \in E} c(e)$.

Each request q is satisfied by a collection of subsets $X_q \subseteq 2^{\mathcal{E}}$ which is *upwards-closed* – that is, if $E_1 \subseteq E_2 \subseteq \mathcal{E}$ and we have that $E_1 \in X_q$ then $E_2 \in X_q$. If the algorithm transmits the set of elements E, then all pending requests q such that $E \in X_q$ are served by that transmission.

To give a concrete example of this abstract structure, consider the Steiner forest problem. In this problem, the elements $\mathcal{E}$ are the edges of a graph. For a request q for the terminals (u_1, u_2), the collection X_q is the collection of edge sets E' such that (u_1, u_2) are in the same connected component in the spanning subgraph with edges E'.

One can also look at the corresponding offline problem – given a set of requests Q, find a subset of elements E' of the minimal total cost such that $E' \in X_q$ for every $q \in Q$.

Now, consider a class of problems of this form – such as Steiner tree for example – and denote this class by ND. The main result of this section is the following.

Theorem II.1. *If there exists a γ deterministic (randomized) approximation algorithm for* ND *which runs in polynomial time, then there exists an $O(\gamma \log |\mathcal{E}|)$-competitive deterministic (randomized) algorithm for* ND *with deadlines, which also runs in polynomial time.*

Remark II.2. If the approximation algorithm runs in *quasi-polynomial* time, then the online algorithm also runs in quasi-polynomial time.

Remark II.3. In this paper, we consider randomized approximation algorithms which have deterministic approximation guarantees and expected running time guarantees. Converting a randomized algorithm of *expected* approximation guarantee and *deterministic* running time to the format we consider can be achieved with repeated running of the algorithm until the resulting approximation is at most a factor of 2 from the expected guarantee – Markov's inquality ensures that the expected running time of this new algorithm is small.

The only requirement for this conversion is that the algorithm is able to know whether its approximation meets the expected guarantee – this requirement is met, for example, in all approximation algorithms based on LP solving + rounding (and in particular, all randomized algorithms in this paper).

For a set of requests Q, we denote the solution for the offline problem returned by the γ approximation by $\text{ND}(Q)$. We also denote the optimal solution by $\text{ND}^*(Q)$.

A. The Framework

We now present a framework for encapsulating an approximation algorithm for ND to obtain a competitive algorithm for ND with deadlines, thus proving Theorem II.1.

Calls to approximation algorithm: The framework makes calls to the approximation algorithm for ND – we denote such a call on a set of requests Q by $\text{ND}(Q)$ (the universe of elements $\mathcal{E}$, and the elements' costs, are identical to those of the online problem). Similarly, we denote the optimal solution for this set of requests by $\text{ND}^*(Q)$.

The framework also makes calls to ND where the costs of the elements are modified – namely, that the cost of some subset of elements $E_0 \subseteq \mathcal{E}$ is set to 0. We use $\text{ND}_{E_0 \leftarrow 0}$ to denote such calls.

When calling the approximation algorithm, we store the resulting solution (i.e. subset of elements) in a variable. If a solution is stored in a variable S, we use $c(S)$ to refer to the cost of that solution. Note that this cost is not necessarily the sum of costs of elements in that solution – it is possible that the solution is for an instance in which the costs of some set of elements E_0 are set to 0.

Algorithm's description: The framework is given in Algorithm 1. For each pending request q, the algorithm maintains a level ℓ_q. Upon the arrival of a new request q, the function UPONREQUEST is called. This function assigns the initial value of the level of q, which is initially supposed to be the logarithmic class of the cost of the least expensive (offline) solution for q – the algorithm approximates this by making a call to the approximation algorithm on $\{q\}$, then dividing by the approximation ratio γ. Over time, the level of a request may increase.

Whenever a deadline of a pending request is reached, the function UPONDEADLINE is called, and the algorithm starts a service. Services also have levels – the level of a service λ, denoted by ℓ_λ, is always $\ell_q + 1$, where q is the request which triggered the service. Intuitively, the service λ is "responsible" for all pending requests of level at most ℓ_λ – these requests are called the *eligible* requests for λ. Overall, the service spends $O(\gamma \cdot 2^{\ell_\lambda})$ cost solely on serving these eligible requests.

The service constructs a transmission, which occurs at the end of the service. First, the service adds to the transmission all "cheap" elements – those that cost at most $\frac{2^{\ell_\lambda}}{|\mathcal{E}|}$. Then, the service decides which of the eligible requests to serve, using the following procedure. It considers the requests by order of increasing deadline, adding them to the set of requests to serve. This process stops when either the cost of serving those requests, as estimated by the approximation algorithm, exceeds the budget ($O(\gamma \cdot 2^{\ell_\lambda})$), or the requests are all served.

Since the amount by which the budget was exceeded in the ultimate iteration is unknown, the service transmits the solution found in the *penultimate* iteration, in addition to a "singleton" solution to the last request to be served.

The final step in the service is to "upgrade" the level of all eligible requests which are still pending after the transmission of the service. The level of those requests is assigned the level of the service.

B. Analysis

To prove Theorem II.1, we require the following definitions.

Definitions and Algorithm's Properties*:* Before delving into the proof of Theorem II.1, we first define some terms used throughout the analysis, and prove some properties of the algorithm.

For a service λ, we call the value set to ℓ_λ the *level* of λ; observe that this value does not change once defined. Similarly, for a request q, we call ℓ_q the level of q. Note that unlike services, the level of a request may change over time (more specifically, the level can be increased).

Definition II.4 (Service Pointer)**.** Let q be a request. We define ptr_q to be the last service λ such that λ sets $\ell_q \leftarrow \ell_\lambda$ in Line 19. If there is no such service, we write $\text{ptr}_q = \text{NULL}$. Similarly, we define $\text{ptr}_q(t)$ to be the last service λ before time t such that λ sets $\ell_q \leftarrow \ell_\lambda$ in Line 19 (with $\text{ptr}_q(t) = \text{NULL}$ if there is no such service).

Definition II.5 (Eligible Requests)**.** Consider a service λ and a request q which is pending upon the start of λ, and has $\ell_q \leq \ell_\lambda$ at that time. We say that q was *eligible* for λ.

Definition II.6 (Types of Services)**.** For a service λ, we say that:

1) λ is *charged* if there exists some future service λ', which is triggered by a pending request q reaching its deadline such that $\text{ptr}_q(t_{\lambda'}) = \lambda$. We say that λ' *charged* λ.

Algorithm 1: Network Design with Deadlines Framework

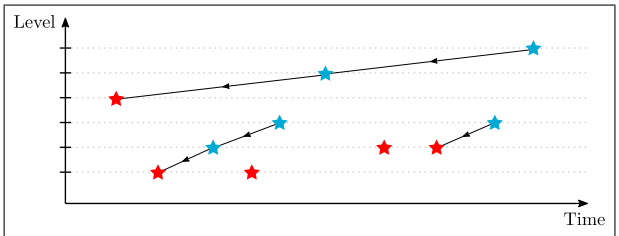
1 **Event Function** UPONREQUEST(q)
2 Set $S_q \leftarrow \text{ND}(\{q\})$
3 Set $I_q \leftarrow \frac{c(S_q)}{\gamma}$.
4 Set $\ell_q \leftarrow \lfloor \log(I_q) \rfloor$ *// the level of the request*

5 **Event Function** UPONDEADLINE(q) *// upon the deadline of a pending request q*
6 Start a new service λ, which we now describe.
7 Set $\ell_\lambda \leftarrow \ell_q + 1$.
8 Set $Q_\lambda \leftarrow \emptyset$.
// buy all cheap elements
9 Set $E_0 \leftarrow \left\{ e \in \mathcal{E} \middle| c(e) \leq \frac{2^{\ell_\lambda}}{|\mathcal{E}|} \right\}$.
// add eligible requests by order of deadline, until budget is exceeded
10 Set $S \leftarrow \emptyset$.
11 **while** *there exists a pending $q' \notin Q_\lambda$ such that $\ell_{q'} \leq \ell_\lambda$* **do**
12 Let $q_{\text{last}} \notin Q_\lambda$ be the pending request with the earliest deadline such that $\ell_{q'} \leq \ell_\lambda$.
13 Set $Q_\lambda \leftarrow Q_\lambda \cup \{q_{\text{last}}\}$
14 Set $S' \leftarrow \text{ND}_{E_0 \leftarrow 0}(Q_\lambda)$.
15 **if** $c(S') \geq \gamma \cdot 2^{\ell_\lambda}$ **then break**;
16 Set $S \leftarrow S'$.
17 Transmit the solution $E_0 \cup S \cup S_{q_{\text{last}}}$. *// serve Q_λ*
// upgrade still-pending requests to service's level
18 **foreach** *pending request q' such that $\ell_{q'} \leq \ell_\lambda$* **do**
19 Set $\ell_{q'} \leftarrow \ell_\lambda$

2) λ is *imperfect* if the **break** command of Line 15 was reached in λ. Otherwise, we say that λ is *perfect*.
3) λ is *primary* if, when triggered by the expired deadline of the pending request q, this request q has $\text{ptr}_q(t_\lambda) = \text{NULL}$. Otherwise, λ is *secondary*.

A visualization of a possible set of services can be seen in Figure 1.

Fix any input set of requests Q. We denote by Λ the final set of services by the algorithm. For every service $\lambda \in \Lambda$, we denote by Q_λ the set of requests served by λ (this is identical to the final value of the variable Q_λ in the algorithm). We define $c(\lambda)$ to be the cost of



This figure shows a possible set of services in a run of the algorithm. Each service is denoted by a star, where the location of the star indicates the time and level of the service. Primary services are denoted by red stars, and secondary services are denoted by blue stars. Each secondary service charges a previous service, of level one below its own; this charging is denoted by a directed edge from the secondary service to the charged service. Since every service can charge – or be charged – at most once, the edges form disjoint paths. A property maintained by the algorithm is that a service "dominates" the quadrant of lesser-or-equal level and time – once such a service occurs, no future secondary service would charge a service in this quadrant.

Figure 1: Visualization of Services

the service λ. For any subset $\Lambda' \subseteq \Lambda$, we also write $c(\Lambda') = \sum_{\lambda \in \Lambda'} c(\lambda)$. Note that $\text{ALG} = c(\Lambda)$.

We denote the set of primary services made by the algorithm by Λ_1, and the set of secondary services by Λ_2, such that $\Lambda = \Lambda_1 \cup \Lambda_2$. We denote the set of charged services by Λ°.

Proposition II.7. *Each service $\lambda \in \Lambda^\circ$ is charged by at most one service.*

Proof: Assume for contradiction that λ is charged by both λ_1 and λ_2, at times t_1 and t_2 respectively, and assume without loss of generality that $t_1 < t_2$. λ_2 charged λ due to the pending request q_2, such that $\ell_{q_2} = \ell_\lambda$ and $\text{ptr}_{q_2}(t_{\lambda_2}) = \lambda$. Note that q_2 was pending before both λ and λ_2, and was thus pending before λ_1. But after λ_1, all pending requests are of level at least $\ell_{\lambda_1} = \ell_\lambda + 1$, in contradiction to having $\ell_{q_2} = \ell_\lambda$ immediately before λ_2. ■

The following lemma we prove shows that for a set of requests which exist in the same time, the collection of charged services which serve them has at most one service from each level.

Definition II.8. We say that a set of requests $Q' = \{q_1, \cdots, q_k\}$ is *intersecting* if there exists time t such that $t \in [r_{q_i}, d_{q_i}]$ for every $i \in \{1, \cdots, k\}$. We call t an *intersection time* of Q'.

Lemma II.9. *Let Q' be an intersecting set of requests. Let $\Lambda_{Q'} \subseteq \Lambda^\circ$ be the set of charged services in which a*

request from Q' is served. Then for every $j \in \mathbb{Z}$, there exists at most one service $\lambda \in \Lambda_{Q'}$ such that $\ell_\lambda = j$.

Proof: Assume for contradiction that there exists $j \in \mathbb{Z}$ for which there exist two distinct services $\lambda_1, \lambda_2 \in \Lambda_{Q'}$ such that $\ell_{\lambda_1} = \ell_{\lambda_2} = j$. Assume without loss of generality that $t_{\lambda_1} < t_{\lambda_2}$. In addition, let $q_1 \in Q'$ be a request served by λ_1, and define $q_2 \in Q'$ to be a request served by λ_2. Let t be an intersection time of Q'.

Since λ_1 is charged, there exists a request q' which was pending at its deadline, triggering a service λ', such that $\mathrm{ptr}_{q'}(t_{\lambda'}) = \lambda_1$. From the definition of $\mathrm{ptr}_{q'}$, we have that $\ell_{q'} = \ell_\lambda$ at time $t_{\lambda'}$. Thus, the service λ' must be of level exactly $j+1$. Also note that q' was eligible for λ_1. Consider the following two cases:

1) $t_{\lambda'} > t_{\lambda_2}$. Since q' was pending at t_{λ_1} and at $t_{\lambda'}$, and since $t_{\lambda_1} < t_{\lambda_2} < t_{\lambda'}$, we have that q' was pending at t_{λ_2}. Observe that $\ell_{q'} = \ell_{\lambda_1}$ at t_{λ_2}, since λ_1 occurred before λ_2. But this means that q' was eligible for λ_2, but was not served (since it was pending at $t_{\lambda'}$). Thus, λ_2 set $\ell_{q'} \leftarrow \ell_{\lambda_2}$ in Line 19, in contradiction to having $\mathrm{ptr}_{q'}(t_{\lambda'}) = \lambda_1$.
2) $t_{\lambda'} < t_{\lambda_2}$. Consider that since $\mathrm{ptr}_{q'}(t_{\lambda'}) = \lambda_1$, we know that q' was eligible for λ_1. The service λ_1 added eligible requests by order of increasing deadline, and thus we know that the deadline of q' is after the deadline of q_1. We know that Q' is an intersecting set of requests, and thus $r_{q_2} \leq d_{q_1}$. Therefore, we have that $r_{q_2} < d_{q'} = t_{\lambda'} < t_{\lambda_2}$, and thus q_2 was pending at $t_{\lambda'}$. We know that q_2 was eligible for λ_2, and thus $\ell_{q_2} \leq j$ at that time. But this contradicts the fact that after λ', every pending request has level at least $\ell_{\lambda'} = j+1$.

■

We now move on to proving Theorem II.1. The proof consists of upper-bounding the cost of the algorithm and lower-bounding the cost of the optimal solution.

Upper-bounding ALG*:* We prove the following lemma, which provides an upper bound on the cost of the algorithm.

Lemma II.10. *It holds that*

$$\mathrm{ALG} \leq O(\gamma) \cdot \left(\sum_{\lambda \in \Lambda_1} 2^{\ell_\lambda} + \sum_{\lambda \in \Lambda^\circ} 2^{\ell_\lambda} \right)$$

Proposition II.11. *The total cost of a service λ is at most $O(\gamma) \cdot 2^{\ell_\lambda}$.*

Proof: The cost of the service λ is the cost of the transmission in Line 17. The cost of this transmission is at most the sum of the three following costs: $c(E_0)$, $c(S)$, and $c(S_{q_{\text{last}}})$. The total cost of E_0, by definition of E_0, is at most 2^{ℓ_λ}.

The cost $c(S)$ is at most $\gamma \cdot 2^{\ell_\lambda}$. To see this, observe that the loop of Line 11 either ends in the first iteration (in which case $S = \emptyset$ and the cost is zero), or continues for two or more iterations. In the second case, consider the iteration before last – since we did not break out of the loop, we have that $c(S) \leq \gamma \cdot 2^{\ell_\lambda}$.

As for the cost $c(S_{q_{\text{last}}})$, consider the initial level of q_{last}. Levels only increase over time, and we know that upon the service λ we had that $\ell_{q_{\text{last}}} \leq \ell_\lambda$. Thus, the initial level of q_{last} was at most ℓ_λ. According to the way in which the initial level is set, we thus have that $c(S_{q_{\text{last}}}) \leq 2\gamma \cdot 2^{\ell_\lambda}$.

Summing over the three costs completes the proof. ■

Proposition II.12. *Only imperfect services can be charged.*

Proof: Observe that a perfect service serves all eligible requests. Thus, Line 19 is not called in such a service, which implies that the service is not charged. ■

Proof of Lemma II.10: Observe that $\mathrm{ALG} = c(\Lambda_1) + c(\Lambda_2)$. First, observe that through Proposition II.11 we have that $c(\Lambda_1) \leq O(\gamma) \cdot \sum_{\lambda \in \Lambda_1} 2^{\ell_\lambda}$.

It remains to show that $c(\Lambda_2) \leq O(\gamma) \cdot \sum_{\lambda \in \Lambda^\circ} 2^{\ell_\lambda}$. Observe that every secondary service λ of level j charges a previous service $\lambda' \in \Lambda^\circ$ of level $(j-1)$. From Proposition II.11, we have that $c(\lambda) \leq O(\gamma) \cdot 2^j$, and thus $c(\lambda) \leq O(\gamma) \cdot 2^{\ell_{\lambda'}}$. Summing over all secondary services completes the proof, where Proposition II.7 guarantees that no charged service is counted twice. ■

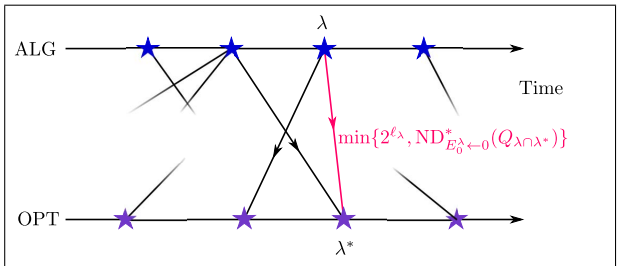
Lower-bounding OPT*:* Fix the optimal solution for the given input, which consists of the services Λ^* made in various points in time. Denote by OPT the cost of this optimal solution. To complete the proof of Theorem II.1, we require the following two lemmas which lower-bound the cost of the optimal solution.

Lemma II.13. $\sum_{\lambda \in \Lambda_1} 2^{\ell_\lambda} \leq O(1) \cdot \mathrm{OPT}$

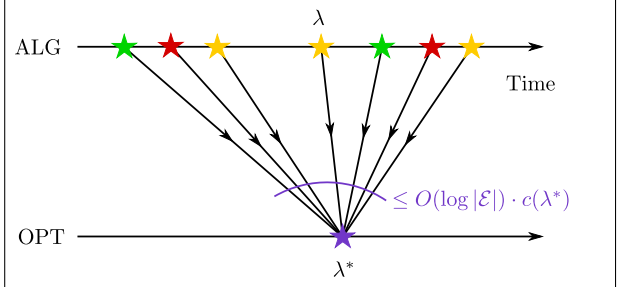
Lemma II.14. $\sum_{\lambda \in \Lambda^\circ} 2^{\ell_\lambda} \leq O(\log |\mathcal{E}|) \cdot \mathrm{OPT}$

It remains to prove Lemma II.14, i.e. charging 2^{ℓ_λ} for each service $\lambda \in \Lambda^\circ$ to the optimal solution times $O(\log |\mathcal{E}|)$. To do this, we split this charge of 2^{ℓ_λ} between the services of the optimal solution. Proposition II.15 shows that this charge is valid.

For a service $\lambda^* \in \Lambda^*$ made by the optimal solution, denote the set of requests served in λ^* by Q_{λ^*}. Recall that for a service $\lambda \in \Lambda$ made by the algorithm, Q_λ is the set of requests served by λ. For every $\lambda \in \Lambda$ and $\lambda^* \in \Lambda^*$, we define for ease of notation $Q_{\lambda \cap \lambda^*} \triangleq Q_\lambda \cap Q^*_\lambda$.



(a) Charging Scheme



(b) Charges to Optimal Service

Subfigure 2a shows the services of Λ° and the services of the optimal algorithm, as well as the charging of costs to the optimal solution. The amount $\min\{2^{\ell_\lambda}, \mathrm{ND}^*_{E_0^\lambda \leftarrow 0}(q_{\lambda \cap \lambda^*})\}$ is charged by the service $\lambda \in \Lambda^\circ$ to the optimal service λ^*. In the proof of Lemma II.14, we show that these charges are sufficient, i.e. each service $\lambda \in \Lambda^\circ$ charges at least 2^{ℓ_λ}.

Subfigure 2b shows the validity of the charging, given in Proposition II.15. This proposition shows that the total amount charged to an optimal service λ^* exceedes its cost by a factor of at most $O(\log |\mathcal{E}|)$. This is shown by partitioning the services which charge cost to λ^* into three types. The first type (green) is low-level services, which are shown to charge a total of at most $O(1) \cdot c(\lambda^*)$. The second type (yellow) is medium-level services. Each of these charges at most $c(\lambda^*)$, but there are at most $O(\log |\mathcal{E}|)$ such yellow services. The last type (red), high-level services, are shown to charge 0 to λ^*.

Figure 2: Visualization of Services

For a set of requests Q', we denote the cost of the optimal offline solution for ND on Q' by $\mathrm{ND}^*(Q')$. We also use $\mathrm{ND}^*_{E_0 \leftarrow 0}(Q')$ to refer to the cost of the optimal offline solution for Q' where the costs of the elements $E_0 \subseteq \mathcal{E}$ is set to 0. For a service $\lambda \in \Lambda$, we denote by E_0^λ the value set to E_0 in Line 9 during the service λ.

Proposition II.15. *There exists a constant β such that for every optimal service $\lambda^* \in \Lambda^*$, we have that*

$$\sum_{\lambda \in \Lambda^\circ} \min\{2^{\ell_\lambda}, \mathrm{ND}^*_{E_0^\lambda \leftarrow 0}(Q_{\lambda \cap \lambda^*})\} \leq \beta \log |\mathcal{E}| \cdot c(\lambda^*) \tag{1}$$

The proofs of Lemmas II.13 and II.14, as well as Proposition II.15, are given in the full version of this paper. Figure 2 shows the overview of the charging scheme used to prove Lemma II.14.

Proof of Theorem II.1: The competitiveness of the algorithm results immediately from Lemmas II.10, II.13 and II.14.

As for the running time, it is clear that the main cost of the algorithm is calling the approximation algorithm ND, and that this is done $O(|Q|)$ times (every iteration of the loop in Line 11 adds a request to the ongoing service). In addition, constructing E_0 for the service takes $O(|\mathcal{E}| \cdot |Q|)$ time overall (since there are at most $|Q|$ services). ■

III. Applications of the Deadline Framework

Applying the framework of Section II for network design problems with delay for known approximation algorithms for network design problems, we obtain algorithms with the competitiveness bounds given in Table II. The algorithms in this table are all deterministic and run in polynomial time, with the exception of the algorithm for directed Steiner tree (which is randomized and runs in quasi-polynomial time). The discussion and proof for the individual problems is in the full version of this paper.

Table II
Deadline Framework Applications

Problem with deadlines	Competitiveness guarantee
Multilevel aggregation	$O(\log n)$
Edge-weighted Steiner forest	$O(\log n)$
Multicut	$O(\log^2 n)$
Edge-weighted Steiner network	$O(\log n)$
Node-weighted Steiner forest	$O(\log^2 n)$
Facility location (non-uniform)	$O(\log n)$
Directed Steiner tree	$O\left(\frac{\log^3 n}{\log \log n}\right)$

In online algorithms, one is often interested in the information-theoretic bounds on competitiveness, without limitations on running time. The framework of Section II supports such constructions – plugging in the algorithm which solves the offline problem optimally yields the following theorem.

Theorem III.1. *There exists an $O(\log |\mathcal{E}|)$-competitive algorithm for* ND *with deadlines (with no guarantees on running time). In particular, there exists an $O(\log n)$-competitive algorithm for all problems in this paper, where n is the number of nodes in the input graph.*

IV. Delay Framework

We now consider the ND problem with delay. This problem is identical to the problem with deadlines, except that instead of a deadline, each request q is

associated with a continuous, monotone-nondecreasing delay function $d_q(t)$, which is defined for every t, and tends to infinity as t tends to infinity (ensuring that every request must be served eventually).

The framework we present for problems with delay requires an approximation algorithm for the prize-collecting variant of the offline problem. In the prize-collecting ND problem, denoted PCND, the input is again a set of requests Q, and an additional penalty function $\pi : Q \to \mathbb{R}^+$. A solution is a subset of elements E which serves some subset $Q' \subseteq Q$ of the requests. The cost of the solution is $\sum_{e \in E} c(e) + \sum_{q \in Q \setminus Q'} \pi(q)$ – that is, the total cost of the elements bought plus the penalties for unserved requests.

Theorem IV.1. *If there exists a γ deterministic (randomized) approximation algorithm for* PCND *which runs in polynomial time, then there exists a $O(\gamma \log |\mathcal{E}|)$-competitive deterministic (randomized) algorithm for* ND *with delay, which runs in polynomial time.*

Note that Remarks II.2 and II.3 apply here as well.

A. The Framework

We now describe the framework for ND with delay.

Calls to the prize-collecting approximation algorithm: The framework makes calls to the approximation algorithm PCND for the prize-collecting problem. Such a call is denoted by $\text{PCND}(Q, \pi)$, where Q is the set of requests and $\pi : Q \to \mathbb{R}^+$ is the penalty function. Some calls are made with the subscript $E_0 \leftarrow 0$, for some subset of elements E_0. This notation means calling PCND on the modified input in which the cost of the elements E_0 is set to 0. The framework also makes calls to ND, an approximation algorithm for the original (not prize-collecting) variant of ND. This approximation algorithm is obtained through calling PCND with penalties of ∞ for each request.

Investment counter: The algorithm maintains for each request q an *investment counter* h_q. Raising this counter corresponds to paying for delay (both past and future) incurred by the request q. When referring to the value of the counter at a point in time t, we write $h_q(t)$.

Definition IV.2 (Residual delay)**.** We define the *residual delay* of a pending request q at time t to be $\rho_q(t) = \max(0, d_q(t) - h_q(t))$. Intuitively, this is the amount of delay incurred by q which no service has covered until time t. For a set of requests Q pending at time t, we also define $\rho_Q(t) = \sum_{q \in Q} \rho_q(t)$.

Definition IV.3 (Penalty function $\pi_{t \to t'}$)**.** At a time t, and for every future time $t' > t$, we define the penalty function $\pi_{t \to t'}$ on pending requests at time t in the following way. For a request q pending at time t, we have that $\pi_{t \to t'}(q) = \max(0, d_q(t') - h_q(t))$. Intuitively, the penalty for a request, as evaluated at time t, is the future residual delay of the request if the algorithm does not raise its investment counter until time t'.

As in the deadline framework, the delay framework assigns a level ℓ_q to each pending request q.

Definition IV.4 (Critical level)**.** At any point during the algorithm, we say that a level j becomes *critical* if the total residual delay of requests of level at most j reaches 2^j.

Algorithm's description: The framework is given in Algorithm 2. The algorithm consists of waiting until any level j becomes critical, and then calling UPONCRITICAL(j). Whenever a new request q is released, the function UPONREQUEST(q) is called.

The algorithm maintains the level of each pending request q, denoted ℓ_q. This level is initially the logarithmic class of the cost of the cheapest solution (i.e. set of elements) serving q (in fact, the algorithm estimates this by calling the approximation algorithm ND and dividing by its approximation ratio). Over time, the level of a request may increase.

When a level j becomes critical, this triggers a service λ of level $\ell_\lambda = j + 1$. Intuitively, the service λ is responsible for all pending requests of level at most ℓ_λ – these are called the eligible requests for λ. The service first starts by raising the investment counters of eligible requests until they all have zero residual delay.

After doing so, the service observes the first point in the future in which such an eligible request has positive residual delay. The goal of the service is to push this point in time (called the forwarding time) as far into the future as possible, while spending at most $O(\gamma \cdot 2^{\ell_\lambda})$ cost.

There are two methods of accomplishing this: the first is to raise the investment counters of the requests, and the second is serving the requests. The best course of action is to combine both methods in a smart manner – deciding which eligible requests are to be served, and raising the investment counter for the remainder of the eligible requests.

To achieve this, the service finds a solution to a prize-collecting instance which captures the problem of pushing back the forwarding time to some future time t'. In this instance, the requests are the eligible requests for λ, and the penalty for a request q is the amount by which its investment counter h_q must be raised so that q's future residual delay would be 0 at time t'. The forwarding time, as well as the corresponding prize-collecting solution, are returned by the call to the function FORWARDTIME.

If the solution returned by FORWARDTIME does not serve any requests (i.e. it only raises investment counters), the service modifies it to serve some arbitrary eligible request. While this does not affect the approximation ratio of the algorithm, it bounds the number of services by the number of requests, which bounds the running time of the algorithm.

Now, the algorithm increases the investment counter of eligible requests which are not served by the solution (paying for their future delay until the forwarding time). The algorithm also upgrades the level of those requests, in a similar way to the deadline algorithm.

Finally, the service transmits its solution, serving the remainder of the eligible requests.

The analysis of Algorithm 2, and the proof of Theorem IV.1, appear in the full version of the paper.

Algorithm 2: Network Design with Delay Framework

Event Function UPONREQUEST(q)
  Set $S_q \leftarrow \text{ND}(\{q\})$
  Set $I_q \leftarrow \frac{c(S_q)}{\gamma}$.
  Set $\ell_q \leftarrow \lfloor \log(I_q) \rfloor$ *// the level of the request*

Event Function UPONCRITICAL(j) *// Upon a level j becoming critical at time t*
  Start a new service λ, which we now describe.
  Set $\ell_\lambda \leftarrow j+1$.
  foreach *request q such that $\ell_q \leq \ell_\lambda$* **do**
    // Clean residual delay of eligible requests
    Set $h_q \leftarrow h_q + \rho_q(t)$
  Set $E_0 = \left\{ e \in \mathcal{E} \middle| c(e) \leq \frac{2^{\ell_\lambda}}{|\mathcal{E}|} \right\}$. *// Buy all cheap elements*
  // Forward time
  Let Q_λ be all pending requests of level at most ℓ_λ.
  Set $(\tau, S) \leftarrow \text{FORWARDTIME}(E_0, Q_\lambda, \ell_\lambda)$.
  Let $Q'_\lambda \subseteq Q_\lambda$ be the subset of requests served in S.
  // make sure that the service serves at least one pending request
  if $Q'_\lambda = \emptyset$ **then** for an arbitrary $q \in Q_\lambda$, set $Q'_\lambda \leftarrow \{q\}$ and $S \leftarrow S_q$.
  // pay for future delay of requests unserved by the transmission, and upgrade requests
  foreach $q \in Q_\lambda \setminus Q'_\lambda$ **do**
    Raise h_q by $\pi_{t \to \tau}(q)$.
    Set $\ell_q \leftarrow \ell_\lambda$.
  Transmit the solution $E_0 \cup S$, serving the requests Q'_λ.[2]

[2] For the sake of the algorithm and its analysis, no requests outside Q'_λ are considered served by this transmission.

V. APPLICATIONS OF THE DELAY FRAMEWORK

As for the deadline case, we now apply the framework of section IV to various network design problems with delay. For the encapsulated prize-collecting algorithms, we use either previously-known prize-collecting algorithms or previously-known approximations for the original offline problem (in which case we show a construction which reduces the prize-collecting problem to the original problem). The competitiveness of the algorithms we obtain are summarized in Table III. All algorithms are deterministic and run in polynomial time. The full discussion of these results appears in the full version of this paper.

As for directed Steiner tree, we are unaware of any competitive prize-collecting algorithm. Such an algorithm, if obtained, can be plugged into our framework and yield an algorithm for directed Steiner tree with delay. We pose designing such an algorithm as an open question.

As in the deadline case, when allowing unlimited running time, one can use the framework of Section IV to obtain the following information-theoretic upper bound on competitiveness.

Theorem V.1. *There exists an $O(\log |\mathcal{E}|)$-competitive algorithm for* ND *with delay (with no guarantees on running time). In particular, there exists an $O(\log n)$-competitive algorithm for all problems considered in this paper, where n is the number of nodes in the input graph.*

VI. CONCLUSIONS AND OPEN PROBLEMS

This paper presented frameworks for network design problems with deadlines or delay, which encapsulate approximation algorithms for the offline network design problem, with competitiveness which is a logarithmic factor away from the approximation ratio of the underlying approximation algorithm. The running time of these frameworks has a polynomial overhead over the running time of the encapsulated approximation algorithm.

In particular, in the formal online model with unbounded computation, this provides $O(\log n)$ upper bounds (with n the number of vertices in the graph), when the offline problem is solved exactly. For some network design problems, as seen in the full version of this paper, this is relatively tight – that is, an

Procedure 3: Time Forwarding Procedure

/ This function, called at time t, returns a future time t'' and a solution $S \subseteq \mathcal{E}$ to transmit which is a "good" solution to minimize the future delay of Q_λ until time t''. This guarantee is made formal in the analysis. */*

Function FORWARDTIME(E_0, Q_λ, j)
  Set $t' \leftarrow t$, $Q'_\lambda \leftarrow \emptyset$ and $S \leftarrow \emptyset$.
  while $Q_\lambda \backslash Q'_\lambda \neq \emptyset$ **do**
    Let $t'' > t$ be the time in which $\sum_{q \in Q_\lambda \backslash Q'_\lambda} (\pi_{t \to t''}(q) - \pi_{t \to t'}(q))$ reaches $\gamma \cdot 2^j$.
    Set $S' \leftarrow \text{PCND}_{E_0 \leftarrow 0}(Q_\lambda, \pi_{t \to t''})$.
    if $c(S') \geq \gamma \cdot 2^j$ **then break**
    Set $Q'_\lambda \subseteq Q_\lambda$ to be the set of requests served in S'.
    Set $t' \leftarrow t''$ and $S \leftarrow S'$.
  return (t'', S)

Table III
DELAY FRAMEWORK APPLICATIONS

Problem with delay	Competitiveness guarantee
Multilevel aggregation	$O(\log n)$
Edge-weighted Steiner forest	$O(\log n)$
Multicut	$O(\log^2 n)$
Edge-weighted Steiner network	$O(\log n)$
Node-weighted Steiner forest	$O(\log^2 n)$
Facility location (non-uniform)	$O(\log n)$

information-theoretic lower bound of $\Omega(\sqrt{\log n})$ exists. Whether there exists an improved framework which can bridge this gap remains open.

For the remaining network design problems, the gap is still large, as no non-constant lower bound is known. This raises the possibility of designing a framework which works for a restricted class of network design problems (which excludes node-weighted Steiner tree and directed Steiner tree), but yields constant competitiveness results for this restricted class. Either designing such a framework, or showing lower bounds, is an open problem.

An additional open problem is to design a good approximation for prize-collecting directed Steiner tree. Applying Theorem IV.1 to such a result would yield a competitive algorithm for directed Steiner tree with delay.

REFERENCES

[1] Noga Alon, Baruch Awerbuch, Yossi Azar, Niv Buchbinder, and Joseph Naor. A general approach to online network optimization problems. *ACM Trans. Algorithms*, 2(4):640–660, 2006.

[2] Itai Ashlagi, Yossi Azar, Moses Charikar, Ashish Chiplunkar, Ofir Geri, Haim Kaplan, Rahul M. Makhijani, Yuyi Wang, and Roger Wattenhofer. Min-cost bipartite perfect matching with delays. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, pages 1:1–1:20, 2017.

[3] Yossi Azar, Ashish Chiplunkar, Shay Kutten, and Noam Touitou. Set cover and vertex cover with delay. *CoRR*, abs/1807.08543, 2018.

[4] Yossi Azar and Amit Jacob Fanani. Deterministic min-cost matching with delays. In *Approximation and Online Algorithms - 16th International Workshop, WAOA 2018, Helsinki, Finland, August 23-24, 2018, Revised Selected Papers*, pages 21–35, 2018.

[5] Yossi Azar, Arun Ganesh, Rong Ge, and Debmalya Panigrahi. Online service with delay. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 551–563, 2017.

[6] Yossi Azar and Noam Touitou. General framework for metric optimization problems with delay or with deadlines. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 60–71, 2019.

[7] Piotr Berman and Chris Coulston. On-line algorithms for steiner tree problems (extended abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 344–353, New York, NY, USA, 1997. ACM.

[8] 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 *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, pages 12:1–12:17, 2016.

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

[10] Marcin Bienkowski, Artur Kraska, Hsiang-Hsuan Liu, and Pawel Schmidt. A primal-dual online deterministic algorithm for matching with delays. In *Approximation and Online Algorithms - 16th International Workshop, WAOA 2018, Helsinki, Finland, August 23-24, 2018, Revised Selected Papers*, pages 51–68, 2018.

[11] Marcin Bienkowski, Artur Kraska, and Pawel Schmidt. A match in time saves nine: Deterministic online matching with delays. In *Approximation and Online Algorithms - 15th International Workshop, WAOA 2017, Vienna, Austria, September 7-8, 2017, Revised Selected Papers*, pages 132–146, 2017.

[12] Marcin Bienkowski, Artur Kraska, and Pawel Schmidt. Online service with delay on a line. In *Structural Information and Communication Complexity - 25th International Colloquium, SIROCCO 2018, Ma'ale HaHamisha, Israel, June 18-21, 2018, Revised Selected Papers*, pages 237–248, 2018.

[13] Carlos Fisch Brito, Elias Koutsoupias, and Shailesh Vaya. Competitive analysis of organization networks or multicast acknowledgment: How much to wait? *Algorithmica*, 64(4):584–605, 2012.

[14] Niv Buchbinder, Moran Feldman, Joseph (Seffi) Naor, and Ohad Talmon. *O*(depth)-competitive algorithm for online multi-level aggregation. In *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, 2017.

[15] Niv Buchbinder, Kamal Jain, and Joseph Naor. Online primal-dual algorithms for maximizing ad-auctions revenue. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, pages 253–264, 2007.

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

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

[18] Yuval Emek, Shay Kutten, and Roger Wattenhofer. Online matching: haste makes waste! In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 333–344, 2016.

[19] Yuval Emek, Yaacov Shapiro, and Yuyi Wang. Minimum cost perfect matching with delays for two sources. In *Algorithms and Complexity - 10th International Conference, CIAC 2017, Athens, Greece, May 24-26, 2017, Proceedings*, pages 209–221, 2017.

[20] Dimitris Fotakis. On the competitive ratio for online facility location. *Algorithmica*, 50(1):1–57, 2008.

[21] Anupam Gupta, Ravishankar Krishnaswamy, and R. Ravi. Online and stochastic survivable network design. *SIAM J. Comput.*, 41(6):1649–1672, 2012.

[22] Makoto Imase and Bernard M. Waxman. Dynamic steiner tree problem. *SIAM J. Discrete Math.*, 4(3):369–384, 1991.

[23] Anna R. Karlin, Claire Kenyon, and Dana Randall. Dynamic TCP acknowledgment and other stories about e/(e-1). *Algorithmica*, 36(3):209–224, 2003.

[24] J. Naor, D. Panigrahi, and M. Singh. Online node-weighted steiner tree and related problems. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 210–219, 2011.