List Update with Delays or Time Windows

Yossi Azar
School of Computer Science, Tel Aviv University, Israel

Shahar Lewkowicz
School of Computer Science, Tel Aviv University, Israel

Danny Vainstein
School of Computer Science, Tel Aviv University, Israel
Google Research, Tel Aviv, Israel

Abstract

We address the problem of **List Update**, which is considered one of the fundamental problems in online algorithms and competitive analysis. In this context, we are presented with a list of elements and receive requests for these elements over time. Our objective is to fulfill these requests, incurring a cost proportional to their position in the list. Additionally, we can swap any two consecutive elements at a cost of 1. The renowned "Move to Front" algorithm, introduced by Sleator and Tarjan, immediately moves any requested element to the front of the list. They demonstrated that this algorithm achieves a competitive ratio of 2. While this bound is impressive, the actual cost of the algorithm's solution can be excessively high. For example, if we request the last half of the list, the resulting solution cost becomes quadratic in the list's length.

To address this issue, we consider a more generalized problem called **List Update with Time Windows**. In this variant, each request arrives with a specific deadline by which it must be served, rather than being served immediately. Moreover, we allow the algorithm to process multiple requests simultaneously, accessing the corresponding elements in a single pass. The cost incurred in this case is determined by the position of the furthest element accessed, leading to a significant reduction in the total solution cost. We introduce this problem to explore lower solution costs, but it necessitates the development of new algorithms. For instance, Move-to-Front fails when handling the simple scenario of requesting the last half of the list with overlapping time windows. In our work, we present a natural $O(1)$ competitive algorithm for this problem. While the algorithm itself is intuitive, its analysis is intricate, requiring the use of a novel potential function.

Additionally, we delve into a more general problem called **List Update with Delays**, where the fixed deadlines are replaced with arbitrary delay functions. In this case, the cost includes not only the access and swapping costs, but also penalties for the delays incurred until the requests are served. This problem encompasses a special case known as the prize collecting version, where a request may go unserved up to a given deadline, resulting in a specified penalty. For this more comprehensive problem, we establish an $O(1)$ competitive algorithm. However, the algorithm for the delay version is more complex, and its analysis involves significantly more intricate considerations.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Online, List Update, Delay, Time Window, Deadline

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.15

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2304.06565 [12]

Funding *Yossi Azar*: Supported in part by the Israel Science Foundation (grant No. 2304/20).

1 Introduction

One of the fundamental problems in online algorithms is the **List Update** problem. In this problem we are given an ordered list of elements and requests for these elements that arrive over time. Upon the arrival of a request, the algorithm must serve it immediately by accessing the required element. The cost of accessing an element is equal to its position in

© Yossi Azar, Shahar Lewkowicz, and Danny Vainstein;
licensed under Creative Commons License CC-BY 4.0

the list. Finally, any two consecutive elements in the list may be swapped at a cost of 1. The goal in this problem is to devise an algorithm so as to minimize the total cost of accesses and swaps. Note that it is an online algorithm and hence does not have any knowledge of future requests and must decide what elements to swap only based on requests that have already arrived.

Although the list update problem is a fundamental and simple problem, its solutions may be costly. Consider the following example. Assume that we are given requests to each of the elements in the farther half of the list. Serving these requests sequentially results in quadratic cost (quadratic in the length of the list). However, in many scenarios, while the requests arrive simultaneously, they do not have to be served immediately. Instead, they arrive with some deadline such that they must be served some time in the (maybe near) future. If this is the case, and the requests' deadlines are further in the future than their arrival, they may be jointly served; thereby incurring a linear (rather than quadratic) cost in the former example. This example motivates the following definition of List Update with Time Windows problem which may improve the algorithms' costs significantly.

The **List Update with Time Windows** problem is an extension of the classical List Update problem. Requests are once again defined as requests that arrive over time for elements in the list. However, in this problem they arrive with future deadlines. Requests must be served during their time window which is defined as the time between the corresponding request's arrival and deadline. This grants some flexibility, allowing an algorithm to serve multiple requests jointly at a point in time which lies in the intersection of their time windows. The cost of serving a set of requests is defined as the current position of the farthest of those elements (i.e. serving a request for the i-th item in the list causes all the other active requests for the first i elements in the list to be served as well in this access operation). In addition, as in the classical problem, swaps between any two consecutive elements may be performed at a cost of 1. Note that both accessing elements (or, serving requests) and swapping consecutive elements is done instantaneously (i.e., time does not advance during these actions). The goal is then to devise an online algorithm so as to minimize the total cost of serving requests and swapping elements. Also note that this problem encapsulates the original List Update problem. In particular, the List Update problem can be viewed as List Update with Time Windows where each time window consist of a unique single point.

We also consider a generalization of the time-window version – **List Update with Delays**. In this problem each request is associated with an arbitrary delay function, such that an algorithm accumulates delay cost while the request remains pending (i.e., unserved). The goal is to minimize the cost of serving the requests plus the total delay. This provides an incentive for the algorithm to serve the requests early.

Another interesting and related variant is the prize collecting variant, which has been heavily researched in other fields as well. The price collecting problem is a special case of List Update with Delay and a generalization of List Update with Time Windows. In the context of List Update, the prize collecting problem is defined such that a request must be either served until some deadline or incur some penalty. Note that List Update with Delays encapsulates this variant by defining a delay function that incurs 0 cost and thereafter (at the deadline) immediately jumps to the penalty cost. The prize collecting problem encapsulates List Update with Time Windows when the penalty is arbitrarily large.

While the flexibility introduced in the list update with time windows or delays problems allow for lower cost solutions, it also introduces complexity in the considered algorithms. In particular, the added lenience will force us to compare different algorithms (our online algorithm compared to an optimal algorithm, for instance) at different time points in the input sequence. Since the problem definition allows for serving requests at different time points, this results in different sets of unserved requests when comparing the algorithms – this divergence will prove to be the crux of the problem and will result in significant added complexity compared to the classical List Update problem.

Originally, the List Update problem was defined to allow for free swaps to the accessed element: i.e., immediately after serving an element e, the algorithm may move e towards the head of the list - free of charge. All other swaps between consecutive elements still incur a cost of 1. In our work, it will be convenient for us to consider the version of the problem where these free swaps are not allowed and all swaps between two consecutive elements incur a cost of 1. We would like to stress that while these two settings may seem different, this is not the case. One may easily observe that the difference in costs between a given solution in the two models is at most a multiplicative factor of 2. This can be seen to be true since the cost of the free swaps may be attributed to the cost of accessing the corresponding element (that was swapped) which is always at least as large. Thus, our results extend easily to the model with free swaps to the accessed element (by losing a factor of 2 in the competitive ratio). In particular, an algorithm which is constant competitive for one of the models is also constant-competitive for the other.

Using the standard definitions an **offline algorithm** sees the entire sequence of requests in advance and thus may leverage this knowledge for better solutions. Conversely, an **online algorithm** only sees a request (i.e., its corresponding element and entire time window or a delay function) upon its arrival and thus must make decisions based only on requests that have already arrived[1]. To analyze the performance of our algorithms we use the classical notion of **competitive ratio**. An online algorithm is said to be c-competitive (for $c \geq 1$) if for every input, the cost of the online algorithm is at most c times the cost of the optimal offline algorithm[2].

1.1 Our Results

In this paper, we show the following results:

- For the List Update with Time Windows problem we provide a 24-competitive algorithm.
- For the List Update with Delays we provide a 336-competitive algorithm.

For the **time windows** version the algorithm is natural. Upon a deadline of a request for an element, the algorithm serves all requests up to twice the element's position and then moves that element to the beginning of the list. Note that the algorithm does not use the fact that the deadline is known when the request arrives. I.e. our result holds even if the deadline is unknown until it is reached (as in non-clairvoyant models). Also note that while the algorithm is deceptively straightforward - its resulting analysis is tremendously more involved.

In the **delay** version the algorithm is more sophisticated. (See the full version [12] for counter examples to some simpler algorithms). The algorithm maintains two types of counters: request counters and element counters. For every request, its request counter increases over time at a rate proportional to the delay cost the request incurred. The request

[1] In principle the time when a request arrives (i.e., is revealed to the online algorithm) need not be the same as the time when its time window or delay begins (i.e., when the algorithm may serve the request). Note however that the change makes no difference with respect to the offline algorithms but only allows for greater flexibility of the online algorithms. Therefore, any competitiveness results for our problem will transcend to instances with this change.

[2] We note that the lists of both the online algorithm and the optimum offline algorithm are identical at the beginning.

counter will be deleted at some point in time after the request has been served (it may not happen immediately after the request is served, but rather further in the future). Unlike the request counters, an element counter's scope is the entire time horizon. The element counter increases over time at a rate that is proportional to the sum of delay costs of unserved requests to that element. Once the requests are served, the element counter ceases to increase. There are two types of events that cause the algorithm to take action: prefix-request-counter events and element-counter events. A prefix-request-counter event takes place when the sum of the request counters of the first ℓ elements reaches a value of ℓ. This event causes the algorithm to access the first 2ℓ elements in the list and delete the request counters for requests to the first ℓ elements. The request counters of the elements in positions $\ell + 1$ up to 2ℓ remain undeleted but cease to increase (Note that this will also result in the first 2ℓ element counters to also cease to increase). An element-counter event takes place when an element counter's value reaches the element's position. Let ℓ be that position. This event causes the algorithm to access the first 2ℓ elements in the list. Thereafter, the algorithm deletes all request counters of requests to that element. Finally, the element's counter is zeroed and the algorithm moves the element to the front.

It is interesting to note that List Update with Delay in the clairvoyant case can be reduced to the special case of prize collecting List Update (which is a generalization of List Update with Time Windows) by creating multiple requests with appropriate penalties. However, neither our algorithm for Delay nor our proof are getting simplified for this case, therefore we present our algorithm and proof for the general case (i.e. for List Update with Delay). Moreover, the reduction from List Update with Delay to prize collecting holds only for the clairvoyant case while our algorithm works on the non-clairvoyant model as well. The full version of this paper contains all the omitted proofs, figures and additional counter examples, and appears in [12].

1.2 Previous Work

We begin by reviewing previous work relating to the classical List Update problem. Sleator and Tarjan [27] began this line of work by introducing the deterministic online algorithm Move to Front (i.e. MTF). Upon a request for an element e, this algorithm accesses e and then moves e to the beginning of the list. They proved that MTF is 2-competitive in a model where free swaps to the accessed element are allowed. The proof uses a potential function defined as the number of inversions between MTF's list and OPT's list. An inversion between two lists is two elements such that their order in the first list is opposite to their order in the second list. A simple lower bound of 2 for the competitive ratio of deterministic online algorithms is achieved when the adversary always requests the last element in the online algorithm's list and OPT orders the elements in its list according to the number of times they were requested in the sequence. Since the model with no free swaps differs in the cost by at most a factor of 2 this immediately yields that MTF is 4-competitive for the model with no free swaps. The simple 2 lower bound also holds for this model. Previous work regarding randomized upper bounds for the competitive ratio have been done by many others [25, 26, 2, 1, 5]. Currently, the best known competitiveness was given by Albers, Von Stengel, and Werchner [3], who presented a random online algorithm and proved it is 1.6 competitive. Previous work regarding lower bounds for this problem have also been made [28, 26, 5]. The highest of which was achieved by Ambühl, Gartner and Von Stengel [6], who proved a lower bound of 1.50084 on the competitive ratio for the classical problem. With regards to the offline classical problem: Ambühl proved this problem is NP-hard [4].

Problems with time windows have been considered for various online problems. Gupta, Kumar and Panigrahi [24] considered the problem of paging (caching) with time windows. Bienkowski et al. [16] considered the problem of online multilevel aggregation. Here, the problem is defined via a weighted rooted tree. Requests arrive on the tree's leaves with corresponding time windows. The requests must be served during their time window. Finally, the cost of serving a set of requests is defined as the weight of the subtree spanning the nodes that contain the requests. Bienkowski et al. [16] showed a $O(D^4 2^D)$ competitive algorithm where D denotes the depth of the tree. Buchbinder et al. [21] improved this to $O(D)$ competitiveness. Later, Azar and Touitou [14, 15] provided a framework for designing and analyzing algorithms for these types of metric optimization problems.

In addition, set cover with deadline [9] was also considered as well as online service in a metric space [20, 11]. To all these problems poly-logarithmic competitive algorithms were designed. It is interesting to note that in contrast to all these problems we show that for our list update problem constant competitive algorithms are achievable. We note that problems with deadline can be also extended to problems with delay where there is a monotone penalty function for each request that is increasing over time until the request is served (and is added to the original cost). Many of the results mentioned above can be extended to arbitrary penalty function. The main exception is matching with delays that can be efficiently solved (i.e. with poly-logarithmic competitive ratio) only for linear functions [22, 8, 7] as well as for concave functions [13]. For other problems that tackle deadlines and delays see: [17, 23, 10, 18, 19].

1.3 Our Techniques

While introducing delays or time windows introduces the option of serving multiple requests simultaneously thereby drastically improving the solution costs, this lenience requires the algorithms and their analyses to be much more intricate.

The "freedom" given to the algorithm compared with the classical List Update problem requires more decisions to be made: for example, in the time windows version assume there are currently two active requests: a request for an element e_1 which just reached its deadline and a request for a further element in the list, e_2 but its deadline has not been reached yet. Should the algorithm access only e_1, pay its position in the list and leave the request for e_2 to be served later or access both e_1 and e_2 together and pay the position of e_2 in the list? If no more requests arrive until the deadline of the second active request, the latter option is better. However, requests that might arrive before the deadline of the second active request might cause the former option to be better after all. In the delay version the decision is more complicated since it may be the case that there are various requests for elements, each request accumulated a small or medium delay but their total is large. Hence, we need to decide at what stage and to what extend serving these requests. Moreover it is more tricky to decide which element to move to the front of the list and at which point in time.

As for the analysis, we need to handle the fact that the online algorithm and the optimal algorithm serve requests at different times. Further, since both algorithms may serve different sets of requests at different times, we may encounter situations wherein a given request at a given time would have been served by the online algorithm and not the optimal algorithm (and vice versa). This, combined with the fact that the algorithms' lists may be ordered differently at any given time, will prove to be the crux of our problem and its analysis.

To overcome these problems, we introduce new potential functions (one for the time windows case and one for the delays case). We note that the original List Update problem was also solved using a potential function [27], however, due to the aforementioned issues, the original function failed to capture the resulting intricacies and we had to introduce novel (and more involved) functions. Ultimately, this resulted in constant competitiveness for both settings.

List Update with Time Windows. Here, the potential function consists of three terms. The first accounts for the difference (i.e., number of inversions) between the online and optimal algorithms' lists at any given time (similar to that of Sleator and Tarjan [27]). The second term accounts for the difference in the set of served requests between the two algorithms. Specifically, whenever the optimal algorithm serves a request not yet served by the online algorithm, we add value to this term which will be subtracted once the online algorithm serves the request. The third term accounts for the movement costs made by the online algorithm incurred by requests that were already served by the optimal algorithm.

At any given time point, our proof considers separately elements that are positioned (significantly) further in the list in the online algorithm compared to the optimal algorithm, as opposed to all other elements (which we will refer to as "the closer" elements). To understand the flavor of our proofs, e.g., the incurred costs of "the further" elements is charged to the first term of the potential function. In contrast, the change in the first term is not be enough to cover the incurred costs of "the closer" elements (the term may even increase). Fortunately, the second term is indeed enough to cover both the incurred costs and the (possible) increase in the first term. Specifically, the added value is of the same order of magnitude as the access cost incurred by the optimal algorithm for serving the corresponding requests. This follows from (a) only requests for elements in ALG which are located at a position which is of the same order of magnitude as the location in OPT get "gifts" in the second term. (b) The fact that the number of trigger elements and their positions in ALGs list is bounded because upon a deadline of a trigger, ALG serves all the elements located up to twice the position of the trigger in its list. The definition of the term "trigger" appears in the beginning of Section 3.

Note however that the analysis above holds only as long as the optimal algorithm does not move an element further in the list between the time it serves it and the time the online algorithm serves it. In such a case, the third term will offset the costs.

List Update with Delays. Here, the potential function consists of five terms. The first term is similar to that of the time windows setting with the caveat that defining the distance between the online and optimal algorithms' lists should depend on the values of the element counters as well. Consider the following example. Assume that the ordering of i, j is reversed when comparing it between the online and optimal algorithms and assume it is ordered (i, j) in the online algorithm. As we defined our algorithm, once the element counter of j is filled, it is moved to the front and therefore the ordering will be reversed. Therefore, intuitively, if j element counter is almost filled we consider the distance between this pair smaller than the case where its element counter is completely empty. Therefore, we would like the contribution to the potential function to be smaller in the former case.

Note that the contribution of the inversion (i, j) depends on the element counter of j but not on the element counter of i (i.e. the contribution is asymmetric). Even if the element counter of j is very close to its position in the online algorithm's list, we still need a big contribution of the pair (i, j) in order to pay for the next element counter event on j. However, if the element counter of j is far from its position in the online algorithm's list, we will need even more contribution of the pair (i, j) to the potential function in order to also cover future delay penalty which the algorithm may suffer on the element j that will not cause an element counter event on j to occur in the short term.

The second part of the potential function consists of the delay cost that both the online and optimal algorithms incurred for requests which were active in both algorithms. This term is used to cover the next element counter events for the elements required in these requests. The third part of the potential function offsets the requests which have been served by the optimal algorithm but not by the online algorithm. This part is very similar to the gifts in the second term of the potential function in time windows and the ideas behind it are similar. Again, the gifts are only given to requests which are located by the online algorithm at a position which is of the same order of magnitude as the location in the optimal algorithm. The gift is of the same order of magnitude as the total delay the online algorithm pays for the request (including the delay it will pay in the future). This is used in order to offset the next element counter event in the online algorithm on the element. However, this gift also decreases as the online algorithm suffers more delay for the request because we want this term in the potential function to also cover the future delay penalty the online algorithm will pay for the request.

The fourth and fifth terms in the potential function are very similar to the third term in the potential function of time windows but each one of them has its own purposes: The fourth term should cover the next element counter event on the element while the fifth term should cover the scenario in which the optimal algorithm served a request and then moved the element further in its list but the online algorithm will suffer more delay penalty for this request in the future. The fifth term should cover this delay cost that the online algorithm pays and thus it is proportional to the fraction between the future delay the online algorithm pays for the request and the position of the element in the online algorithm's list.

2 The Model for Time Windows and Delays

Given an input σ and algorithm ALG we denote by $ALG(\sigma)$ the cost of its solution. Recall that in the **time windows** setting $ALG(\sigma)$ is defined as the sum of (1) the algorithm's access cost: the algorithm may serve multiple requests at a single time point and then the access cost is defined as the position of the farthest element in this set of requests. $ALG(\sigma)$ also accounts for (2) the total number of element swaps performed by ALG. In total, $ALG(\sigma)$ is equal to the sum of access costs and swaps. In the **delay** setting $ALG(\sigma)$ accounts (1) and (2) as above in addition to (3) the sum of the delay incurred by all requests. The delay is defined via a delay function that is associated with each request. The delay functions may be different per request and are each a monotone non-decreasing non-negative function. In total, $ALG(\sigma)$ is equal to the sum of access costs, swaps and delay costs. As is traditional when analysing online algorithms, we denote by $OPT(\sigma)$ the cost of the optimal solution to input σ. Furthermore, we say that ALG is c-competitive (for $c \geq 1$) if for every input σ, $ALG(\sigma) \leq c \cdot OPT(\sigma)$. Throughout our work, when clear from context, we use $ALG(\sigma)$ to denote both the cost of the solution and the solution itself. Our algorithms work also in the non-clairvoyant case: In the time windows version we only know the deadline of a request upon its deadline (and not upon its arrival). In the delay version we know the various delay functions of the requests only up to the current time. Next we introduce several notations that will aid us in our proofs.

▶ **Definition 1.** *Let $\mathbb{E}$ be the set of the elements.*

- *Let n denote the number of elements in our list ($|\mathbb{E}| = n$) and m the number of requests.*
- *Let r_k denote the kth request and e_k the requested element by r_k.*
- *Let $y_k \in [n]$ denote the position of e_k in OPTs list at the time OPT served r_k. Let $x_k \in [n]$ denote the position of e_k in ALGs list at the time OPT (and not ALG) served r_k[3].*

[3] In the delay version, x_k and y_k are defined only in case OPT indeed served the request r_k at some time.

Throughout our work, given an element in the list, we oftentimes consider its neighboring elements in the list. We therefore introduce the following conventions to avoid confusion. Given an element in the list we refer to its **previous** element as its neighbor which is closer to the head of the list and its **next** element as its neighbor which is further from the head of the list.

3 The Algorithm for Time Windows

Prior to defining our algorithm, we need the following definitions.

▶ **Definition 2.** *We define the* ***triggering element****, when a deadline of a request is reached, as the farthest element in the list such that there exists an active request for it which just reached its deadline. We define the* ***triggering request*** *as one of the active requests for the triggering element that just reached its deadline - arbitrary.*

When clear from context we will use the term "trigger" instead of "triggering request" or "triggering element". Next, we define the algorithm.

Algorithm 1 Algorithm for Time Windows (i.e. Deadlines).

Upon deadline of a request **do**:
    $i \leftarrow$ triggering element's position
    Serve the set of requests in the first $2i - 1$ elements in the list
    Move-to-front the triggering element

We prove the following theorem for the above algorithm in Appendix A.

▶ **Theorem 3.** *For each sequence of requests σ, we have that*

$$ALG(\sigma) \leq 24 \cdot OPT(\sigma).$$

4 The Algorithm for Delays

Our algorithm maintains two types of counters in order to process the input: requests counters and element counters. We begin by defining the **request counters**. The algorithm maintains a separate request counter for every incoming request. For a given request r_k we denote its corresponding counter as RC_k. The counter is initialized to 0 the moment the request arrives and increases at the same rate that the request incurs delay. Once the request is served, the counter ceases to increase. Finally, our algorithm deletes the request counters - it will do so at some point in the future after the request is served (but not necessarily immediately when the request is served).

Next we define the **element counters**. Unlike the request counters, element counters exist throughout the entire input (i.e., they are initialized at the start of the input and do not get deleted). We define an element counter EC_e for every element $e \in \mathbb{E}$. These counters are initialized to 0 and increase at a rate equal to the total delay incurred by requests to the specific element.

We define two types of events that cause the algorithm to act: prefix-request-counter events and element-counter events. A **prefix request counters event on ℓ** for $\ell \in [n]$ occurs when the sum of all the request counters of requests for the first ℓ elements in the list reaches the value of ℓ. When this type of event takes place, the algorithm performs the following two actions. First, it serves the requests of the first 2ℓ elements. Second, it deletes the request counters that belong to the first ℓ elements. Note that these are the request elements that contributed to this event and are therefore deleted. Also note that the request counters of the elements $\ell + 1$ to 2ℓ and the element counters of the first 2ℓ elements cease to increase since their requests have been served.

An **element counter event on e** for $e \in \mathbb{E}$ occurs when EC_e reaches the value of ℓ, where $\ell \in [n]$ is the position of the element e in the list, currently. When this type of event takes place, the algorithm performs the following three actions. First, it serves the requests on the first 2ℓ elements. Second, it deletes all request counters of requests to the element e. Third, it sets EC_e to 0 and perform move-to-front to e.

Note that the increase in an element counter equals to the sum of the increase of all the request counters to this element. In particular, the value of the element counter is at least the sum of the non-deleted request counters for the element (It may be larger since request counters may be deleted in request counters events while the element counter maintains its value). Hence when we zero an element counter, we also delete the request counters of requests for this element in order to maintain this invariant.

Next, we present the algorithm.

Algorithm 2 Algorithm for Delay.

Initialization:
    For each $e \in \mathbb{E}$ **do**:
        $EC_e \leftarrow 0$
Upon arrival of a new request r_k **do**:
    $RC_k \leftarrow 0$
Upon prefix-request-counters event on $\ell \in [n]$ **do**:
    Serve the set of requests in the first 2ℓ elements in the list
    Delete the request counters for the first ℓ elements in the list
Upon element-counter event on e (let ℓ denote e's current position) **do**:
    Serve the set of requests in the first 2ℓ elements in the list
    Delete all the request counters of requests for the element e
    $EC_e \leftarrow 0$
    Move-to-front the element e

We prove the following theorem for the above algorithm in the full version.

▶ **Theorem 4.** *For each sequence of requests σ, we have that*

$$ALG(\sigma) \leq 336 \cdot OPT(\sigma).$$

5 Potential Functions for Time Windows and Delay

Our proofs use potential functions. In particular we prove for each possible event that

$$\Delta ALG + \Delta\Phi \leq c \cdot \Delta OPT$$

where Φ is the potential and c is the competitive ratio. In this section we describe the potential functions. The detailed proofs that use these potential functions appear in the full version.

5.1 Time Windows

As mentioned earlier, our potential function used for the time windows setting consists of three terms. We will define them separately. We begin with the first term that aims to capture the difference between ALG and OPT's lists at any given moment.

▶ **Definition 5.** *Let $\phi(t)$ denote the number of **inversions** between ALG's and OPT's lists at time t. Specifically,*

$$\phi(t) = |\{(i,j) \in \mathbb{E}^2 |\ \textit{At time } t,\ i \textit{ is before } j \textit{ in } ALG\textit{'s list and after } j \textit{ in } OPT\textit{'s list}\}|.$$

The second term accounts for the difference in the set of served requests between the two algorithms. Specifically, whenever the optimal algorithm serves a request not yet served by the online algorithm, we add value to this term which will be subtracted once the online algorithm serves the request. Before defining this term, we need the following definition.

▶ **Definition 6.** *For each time t, let $\lambda(t) \subseteq [m]$ be the set of all the request indices k such that the request r_k arrived and was served by OPT but was not served by ALG at time t.*

Recall that for request r_k we denote by y_k the position of e_k in OPT's list at the time that OPT served r_k. Furthermore, we denote by x_k the position of e_k in ALG's list at the time OPT (and not ALG) served r_k. We are now ready to define the second term in our potential function.

▶ **Definition 7.** *For $k \in \lambda(t)$ we define $\psi(x_k, y_k) \geq 0$ as*

$$\psi(x,y) = \begin{cases} 7x & \text{if } 1 \leq x \leq y \\ 8y - x & \text{if } y \leq x \leq 8y \\ 0 & \text{if } 8y \leq x \end{cases}$$

Next, we define the third term of our potential function.

▶ **Definition 8.** *We define $\mu_k(t)$ as the number of swaps OPT performed between e_k and its next element in the list from the time OPT served the request r_k until time t.*

Finally, we combine the terms and define our potential function.

▶ **Definition 9.** *We define our potential function for Time Windows as*

$$\Phi(t) = 4 \cdot \phi(t) + \sum_{k \in \lambda(t)} \psi(x_k, y_k) + 4 \cdot \sum_{k \in \lambda(t)} \mu_k(t).$$

5.2 Delay

In the delays setting, we define a different potential function that is comprised of five terms. We will define the terms separately first and thereafter use them to compose our potential function. We begin with the first term.

As mentioned in Our Techniques, the first term also aim to capture the distance between ALG's and OPT's lists. In the time windows setting, we defined this term as the number of element inversions. In the delays case this does not suffice; we have to take into the account the elements' counters as well. To gain some intuition as to why this addition is needed, consider the following example. Assume that elements i, j are ordered (i, j) in ALG and reversed in OPT. Recall that ALG is defined such that when j' element counter is filled, then we move it to the front (thereby changing the ALG's ordering to (j,i)). Therefore, if it is the case that j's element counter is nearly filled, intuitively we may say that i,j's ordering in ALG and OPT are closer to each other than if j's element counter would have been empty. Therefore, we would like the contribution to the potential function to be smaller in the former case.

Note that the contribution of the inversion (i,j) depends on the element counter of j but not on the element counter of i (i.e. the contribution is asymmetric). Even if the element counter of j is very close to its position in the online algorithm's list, we still need a big contribution of the pair (i,j) in order to pay for the next element counter event on j. However, if the element counter of j is far from its position in the online algorithm's list, we need even more contribution of the pair (i,j) to the potential function in order to also cover future delay penalty which the algorithm may suffer on the element j that does not cause an element counter event on j to occur in the short term. Before formally defining this term, we define the following.

▶ **Definition 10.** *For a time t and an element $e \in \mathbb{E}$ we define:*
- *EC_e^t to be the value of the element counter EC_e at time t.*
- *$x_e^t \in [n]$ ($y_e^t \in [n]$ resp) to be the position of e in ALGs (OPTs resp) list at time t.*
- *$I_e^t = \{i \in \mathbb{E} | i$ is before e in ALGs list and after e in OPTs list at time $t\}$.*

▶ **Definition 11.** *For element $e \in \mathbb{E}$ we define $\rho_e(t) = |I_e^t| \cdot (28 - 8 \cdot \frac{EC_e^t}{x_e^t})$*

Observe that each $i \in I_e^t$ contributes $20 + 8 \cdot (1 - \frac{EC_e^t}{x_e^t})$ to $\rho_e(t)$. The additive term of 20 is used in order to cover the next element counter event for e while the second term is used to cover the delay penalty ALG will pay in the future for requests for e. Note that the term $1 - \frac{EC_e^t}{x_e^t}$ is the fraction of EC_e which is not "filled" yet. If this term is very low, ALG is very close to have an element counter event on e, which causes the order of i and e in ALGs list and OPTs list to be the same, thus it makes sense that the contribution of i to $\rho_e(t)$ is lower compared with the case where $1 - \frac{EC_e^t}{x_e^t}$ would be higher.

Next, we consider the second term. First, we denote the total incurred delay by a request as $d_k(t)$. Formally, this is defined as follows.

▶ **Definition 12.** *For a given request r_k and time t let $d_k(t)$ denote the total delay incurred by the request by ALG up to time t. (Note that it is defined as 0 before the request arrived and remains unchanged after the request is served). Let $d_k = \sup_t d_k(t)$. Note that this is a supremum and not maximum for the case that r_k is never served. Note that $d_k \leq n$ because ALG always serves r_k before $d_k > n$.*

Our second term is a sum of incurred delay costs of specific elements.

▶ **Definition 13.** *For each $k \in [m]$, the request r_k is considered:*
- ***active*** *in ALG (resp. OPT) from the time it arrives until it is served by ALG (resp. OPT).*
- ***frozen*** *from the time it is served by ALG until EC_{e_k} is zeroed in an e_k element counter event.*

▶ **Definition 14.** *For time t we define $\lambda(t) \subseteq [m]$ as the set of requests (request indices) which are either active* ***or*** *frozen in ALG at time t. We define $\lambda_1(t) \subseteq \lambda(t)$ as the set of requests that are also active in OPT at time t and $\lambda_2(t) \subseteq \lambda(t)$ as the set of requests that are also not active in OPT at time t.*

Finally, we define our second term.

▶ **Definition 15.** *We define the second term of the Delays potential function as* $\sum_{k\in\lambda_1(t)} d_k(t)$.

The third term is defined as follows (we use x_k and y_k as previously defined).

▶ **Definition 16.** *We define the third term as* $\sum_{k\in\lambda_2(t)}(42d_k - 6d_k(t)) \cdot 1[x_k \le 4y_k]$.

Note that $42d_k - 6d_k(t) = 36d_k + 6 \cdot (d_k - d_k(t))$. Therefore each request index $k \in \lambda_2(t)$ contributes two terms to Φ: $36d_k$ is used to cover the next element counter on e_k while the second term is 6 times the delay *ALG* will pay for r_k in the future, which will be used to cover this exact delay penalty that *ALG* will pay in the future for r_k.

The fourth term is defined to cover the next element counter event on a given element as follows.

▶ **Definition 17.** *Let $\mu_e(t)$, for $e \in \mathbb{E}$, be the number of swaps OPT performed between e and its next element in its list ever since the last element counter event before time t on e by ALG (or the beginning of the time horizon if there was not such an event).*

Finally, we define the fifth term. The fifth term should cover the scenario in which the optimal algorithm served a request and then moved the element further in its list but the online algorithm will suffer more delay penalty for this request in the future. It will also cover the delay cost that the online algorithm will pay and thus it is proportional to the fraction between the future delay the online algorithm will pay for the request and the position of the element in the online algorithm's list.

▶ **Definition 18.** *Let $\mu_k(t)$, for $k \in \lambda_2(t)$, be the number of swaps OPT performed between e_k and its next element in its list ever since OPT served the request r_k (by accessing e_k).*

▶ **Definition 19.** *We define the fifth term of the Delays potential function as*

$$8 \cdot \sum_{k\in\lambda_2(t)} \frac{d_k - d_k(t)}{x^t_{e_k}} \cdot \mu_k(t).$$

We are now ready to define our potential function.

▶ **Definition 20.** *We define our potential function for the delays setting as*

$$\Phi(t) = \sum_{e\in\mathbb{E}} \rho_e(t) + 36 \cdot \sum_{k\in\lambda_1(t)} d_k(t) + \sum_{k\in\lambda_2(t)} (42d_k - 6d_k(t)) \cdot 1[x_k \le 4y_k] + \\ + 48 \cdot \sum_{e\in\mathbb{E}} \mu_e(t) + 8 \cdot \sum_{k\in\lambda_2(t)} \frac{d_k - d_k(t)}{x^t_{e_k}} \cdot \mu_k(t)$$

6 Conclusion and Open Problems

In this paper, we presented the List Update with Time Windows and Delay, which generalize the classical List Update problem.

- We presented a 24-competitive ratio algorithm for the List Update with Time Windows problem.
- We presented a 336-competitive ratio algorithm for the List Update with Delays problem.
- Open problems: The main issue left unsolved is the gap between the upper and lower bounds. Currently, the best lower bound for both problems considered is 2. Note that this is the same lower bound given to the original List Update problem. An interesting followup would be to improve upon this result and show a better lower bound. On the other hand, one may improve the upper bound - our algorithms are non-clairvoyant in the

sense that our proofs and algorithms hold even when the deadlines/delays are unknown. It would be interesting to understand whether clairvoyance may improve the upper bound. Another interesting direction would be to consider randomization as a way of improving our bounds.

References

1. Susanne Albers. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27(3):682–693, 1998.
2. Susanne Albers and Michael Mitzenmacher. Revisiting the counter algorithms for list update. *Information processing letters*, 64(3):155–160, 1997.
3. Susanne Albers, Bernhard Von Stengel, and Ralph Werchner. A combined bit and timestamp algorithm for the list update problem. *Information Processing Letters*, 56(3):135–139, 1995.
4. Christoph Ambühl. Offline list update is np-hard. In *European Symposium on Algorithms*, pages 42–51. Springer, 2000.
5. Christoph Ambühl, Bernd Gärtner, and Bernhard von Stengel. Optimal projective algorithms for the list update problem. In *International Colloquium on Automata, Languages, and Programming*, pages 305–316. Springer, 2000.
6. Christoph Ambühl, Bernd Gärtner, and Bernhard Von Stengel. A new lower bound for the list update problem in the partial cost model. *Theoretical Computer Science*, 268(1):3–16, 2001.
7. 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 *APPROX/RANDOM*, pages 1:1–1:20, 2017.
8. Yossi Azar, Ashish Chiplunkar, and Haim Kaplan. Polylogarithmic bounds on the competitiveness of min-cost perfect matching with delays. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1051–1061, 2017.
9. 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`.
10. Yossi Azar, Yuval Emek, Rob van Stee, and Danny Vainstein. The price of clustering in bin-packing with applications to bin-packingwith delays. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 1–10, 2019.
11. Yossi Azar, Arun Ganesh, Rong Ge, and Debmalya Panigrahi. Online service with delay. In *STOC*, pages 551–563, 2017.
12. Yossi Azar, Shahar Lewkowicz, and Danny Vainstein. List update with delays or time windows, 2023. `arXiv:2304.06565`.
13. 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`.
14. 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`.
15. Yossi Azar and Noam Touitou. Beyond tree embeddings – A deterministic framework for network design with deadlines or delay. In Sandy Irani, editor, *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`.

16 Marcin Bienkowski, Martin Böhm, Jaroslaw Byrka, Marek Chrobak, Christoph Dürr, Lukáš Folwarczný, Lukasz Jez, Jiri Sgall, Kim Thang Nguyen, 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.

17 Marcin Bienkowski, Martin Böhm, Jaroslaw Byrka, and Jan Marcinkowski. Online facility location with linear delay. In Amit Chakrabarti and Chaitanya Swamy, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2022, September 19-21, 2022, University of Illinois, Urbana-Champaign, USA (Virtual Conference)*, volume 245 of *LIPIcs*, pages 45:1–45:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.APPROX/RANDOM.2022.45.

18 Marcin Bienkowski, Jaroslaw Byrka, Marek Chrobak, Neil B. Dobbs, Tomasz Nowicki, Maxim Sviridenko, Grzegorz Swirszcz, and Neal E. Young. Approximation algorithms for the joint replenishment problem with deadlines. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming – 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, volume 7965 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2013. doi:10.1007/978-3-642-39206-1_12.

19 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.

20 Marcin Bienkowski, Artur Kraska, and Pawel Schmidt. Online service with delay on a line. In *SIROCCO*, 2018.

21 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.

22 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.

23 Leah Epstein. On bin packing with clustering and bin packing with delays. *CoRR*, abs/1908.06727, 2019. arXiv:1908.06727.

24 Anupam Gupta, Amit Kumar, and Debmalya Panigrahi. Caching with time windows and delays. *SIAM J. Comput.*, 51(4):975–1017, 2022. doi:10.1137/20m1346286.

25 Sandy Irani. Two results on the list update problem. *Information Processing Letters*, 38(6):301–306, 1991.

26 Nick Reingold, Jeffery Westbrook, and Daniel D Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994.

27 Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

28 Boris Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47(1):5–9, 1993.

A The Analysis for the Algorithm for Time Windows

In this section we will prove Theorem 3. Throughout we will denote Algorithm 1 as ALG.

▶ **Definition 21.** *For each $k \in [m]$ we use a_k and q_k to denote the arrival time and deadline of the request r_k.*

As a first step towards proving Theorem 3 we prove in Lemma 22 that it is enough to consider inputs that only contain triggering requests. The proof appears in the full version [12].

▶ **Lemma 22.** *Let σ be a sequence of requests and let $\sigma^{'}$ be σ after omitting all the non-triggering requests (with respect to ALG). Then*

$$\frac{ALG(\sigma)}{OPT(\sigma)} \leq \frac{ALG(\sigma^{'})}{OPT(\sigma')}.$$

▶ **Corollary 23.** *We may assume w.l.o.g. that the input σ only contains triggering requests (with respect to ALG).*

The following lemma is simple but will be very useful later. Recall that k refers to the index of the k'th request in the input σ and that e_k denotes its requested element.

▶ **Lemma 24.** *For every $k \in [m]$, the position of e_k in ALG's list remains unchanged throughout the time interval $[a_k, q_k)$. Hence x_k denotes the location of e_k in ALGs list during the time interval $[a_k, q_k)$.*

Proof. During the time interval between a_k and q_k, ALG did not access the element e_k and did not access any element located after e_k in its list, because otherwise it would be a contradiction to our assumption that r_k is a trigger request. Therefore, during the time interval mentioned above, ALG only accessed (served) elements which were before e_k in its list and performed move-to-fronts on them. But these move-to-fronts did not change the position of e_k in ALG's list. ◀

▶ **Definition 25.** *Let z_k denote the position of the farthest element OPT accesses at the time it served r_k.*

Note that z_k defines the cost OPT pays for serving the set of requests that contain r_k.

Recall that ALG serves all requests separately (since all requests are triggering requests - Corollary 23). OPT, on the other hand, may serve multiple requests simultaneously. Note that at the time OPT serves the request r_k, it pays access cost of z_k (and it is guaranteed that $z_k \geq y_k$). The strict inequality $z_k > y_k$ occurs in case OPT serves a request for an element located further than e_k in its list and by accessing this far element, OPT also accesses e_k, thus serving r_k.

▶ **Lemma 26.** *The cost of ALG is bounded by*

$$ALG(\sigma) \leq 3 \cdot \sum_{k=1}^{m} x_k$$

Proof. For each $k \in [m]$, e_k is located at position x_k at the time when ALG serves r_k. ALG pays an access cost of at most $2x_k - 1$ when it serves the request r_k (Observe that ALG may pay an access cost of less than $2x_k - 1$ in case $n < 2x_k - 1$). ALG also pays a cost of $x_k - 1$ for performing move-to-front on e_k. Therefore, ALG suffers a total cost of at most $(2x_k - 1) + (x_k - 1) \leq 3x_k$ for serving this request. If we sum for all the requests, we get that $ALG(\sigma) \leq 3 \cdot \sum_{k=1}^{m} x_k$. ◀

▶ **Lemma 27.** *Let t be a time when the active request indices in ALG are $R = \{k_1, k_2, ..., k_d\}$ where $1 \leq x_{k_1} < x_{k_2} < ... < x_{k_d} \leq n$. We have:*

1. *For each $\ell \in [d-1]$, we have $q_{k_\ell} \leq q_{k_{\ell+1}}$, i.e., ALG serves the request r_{k_ℓ} before it serves $r_{k_{\ell+1}}$.*

2. *For each* $\ell \in [d-1]$ *we have that* $2x_{k_\ell} \le x_{k_{\ell+1}}$.
3. $d \le \log n + 1$*, i.e., at any time, there are at most* $\log n + 1$ *active requests in ALG.*

Proof. If ALG serves $r_{k_{\ell+1}}$ before it serves r_{k_ℓ}, it serves also r_{k_ℓ} by passing through e_{k_ℓ} when it accesses $e_{k_{\ell+1}}$, contradicting our assumption that r_{k_ℓ} is a trigger request (Observation 23).

If we have $x_{k_{\ell+1}} \le 2x_{k_\ell} - 1$, then when ALG serves r_{k_ℓ}, it will also access $e_{k_{\ell+1}}$, thus serving $r_{k_{\ell+1}}$, contradicting our assumption that $r_{k_{\ell+1}}$ is a trigger request (Observation 23).

Using what we have already proved, a simple induction can be used in order to prove that for each $\ell \in [|R_t^{OPT}|]$, we have that $2^{\ell-1}x_{k_1^t} \le x_{k_\ell^t}$. Therefore, we have that $2^{|R_t^{OPT}|-1}x_{k_1^t} \le x_{k^t_{|R_t^{OPT}|}}$. We also have that $1 \le x_{k_1^t}$ and $x_{k^t_{|R_t^{OPT}|}} \le n$. These three inequalities yield to $|R_t^{OPT}| \le \log n + 1$. ◀

Next we consider OPT's solution.

▶ **Definition 28.** *Let* T_{OPT} *be the set of times when OPT served requests. We then define:*

- *For each time* $t \in T_{OPT}$*, let* $R_t^{OPT} = \{k_1^t, k_2^t, ..., k^t_{|R_t^{OPT}|}\}$ *be the non-empty set of request indices that OPT served at time t where* $1 \le x_{k_1^t} < x_{k_2^t} < ... < x_{k^t_{|R_t^{OPT}|}} \le n$.
- *Let* $J(t) = \arg\max_{k \in R_t^{OPT}} \{y_k\}$.

By definition for each $t \in T_{OPT}$, we have

$$1 \le y_{J(t)} = z_{k_1^t} = z_{k_2^t} = ... = z_{k^t_{|R_t^{OPT}|-1}} = z_{k^t_{|R_t^{OPT}|}} \le n$$

Observe that at time $t \in T_{OPT}$, OPT serves the requests R_t^{OPT} together by accessing the $y_{J(t)}$'s element in its list. Therefore, OPT pays an access cost of $y_{J(t)}$ at time t.

▶ **Observation 29.** *For any* $t \in T_{OPT}$*, the total cost OPT pays for accessing elements at time t is* $y_{J(t)}$.

▶ **Lemma 30.** *Let* $t \in T_{OPT}$*. We have:*

1. *For each* $\ell \in [|R_t^{OPT}| - 1]$ *we have that* $2x_{k_\ell^t} \le x_{k_{\ell+1}^t}$.
2. $|R_t^{OPT}| \le \log n + 1$*, i.e, OPT serves at most* $\log n + 1$ *triggers at the same time.*

Proof. Since OPT served the requests R_t^{OPT} at time t, all these requests arrived at time t or before it. Therefore, from Observation 32 we get that all the requests R_t^{OPT} were active in ALG at time t. Therefore, we get that this lemma holds due to Lemma 27. Note that there may be additional requests which were active in ALG at time t but OPT did not serve at time t (meaning they were not in R_t^{OPT}), but this does not contradict the conclusion. ◀

The following lemma allows us to consider from now on only algorithms such that if they serve requests at time t, then at least one of these requests has a deadline at t. In particular, we can assume that OPT has this property. Observe that ALG also has this property.

▶ **Lemma 31.** *For every algorithm A, there exists an algorithm B such that for each sequence of request* σ *we are guaranteed that:*

1. *B only serves requests upon some deadline.*
2. $B(\sigma) \le A(\sigma)$.

For convenience, we assume that when both ALG and OPT are serving σ, in case both OPT and ALG perform access or swapping operations at the same time - we first let OPT perform its operations and only then ALG will perform its operations.

On the other hand, for elements which are not served at the same time by OPT and ALG, by combining the fact that ALG serves requests at their deadline (see Corollary 23) with the fact that OPT must serve requests before the deadline, we get that again OPT serves the request before ALG. Combining the two cases yields Observation 32.

▶ **Observation 32.** *For each $k \in [m]$, OPT serves the request r_k before ALG serves r_k.*

▶ **Definition 33.** *We define the set of **events** P which contains the following 3 types of events:*
1. *ALG serves the request r_k at time q_k.*
2. *OPT serves the requests R_t^{OPT} at time t.*
3. *OPT swaps two elements.*

Recall that the potential function Φ is defined in Section 5.1 as follows:

$$\Phi(t) = 4 \cdot \phi(t) + \sum_{k \in \lambda(t)} \psi(x_k, y_k) + 4 \cdot \sum_{k \in \lambda(t)} \mu_k(t)$$

where the terms ϕ, λ, ψ and μ_k are also defined in that section.

▶ **Definition 34.** *For each event $p \in P$, we define:*
- *ALG^p (OPT^p) to be the cost ALG (OPT) pays during p.*
- *For any parameter z, Δz^p to be the value of z after p minus the value of z before p.*

Clearly, we have $ALG(\sigma) = \sum_{p \in P} ALG^p$ and $OPT(\sigma) = \sum_{p \in P} OPT^p$. Observe that Φ starts with 0 (since at the beginning, the lists of ALG and OPT are identical) and is always non-negative. Therefore, if we prove that for each event $p \in P$, we have

$$ALG^p + \Delta\Phi^p \leq 24 \cdot OPT^p$$

then, by summing it up for over all the events, we will be able to prove Theorem 3. Note that we do not care about the actual value $\Phi(t)$ by itself, for any time t. We will only measure the change of Φ as a result of each type of event in order to prove that the inequality mentioned above indeed holds. The three types of events that we will discuss are:
1. The event where ALG serves the request r_k at time q_k (event type 1) is analyzed in Lemma 35.
2. The event where OPT serves the requests R_t^{OPT} at time t (event type 2) is analyzed in Lemma 39.
3. The event where OPT swaps two elements (event type 3) is analyzed in Lemma 40.

We begin by analyzing the event where ALG served a request.

▶ **Lemma 35.** *Let $p \in P$ be the event where ALG served the request r_k (where $k \in [m]$) at time q_k. We have*

$$ALG^p + \Delta\Phi^p \leq 0 \ (= OPT^p)$$

In order to prove Lemma 35, we separate the movement of ALG versus the movement of OPT. The final proof is the superposition of the two movements. Firstly we assume that OPT did not increase the position of e_k in its list ever since it served the request r_k until ALG served it, then we remove this assumption.

▶ **Lemma 36.** *Let $p \in P$ be the event where ALG served the request r_k (where $k \in [m]$) at time q_k. Assume that ever since OPT served r_k until ALG served r_k, OPT did not increase the position of e_k in its list. We have that*

$$3x_k + 4 \cdot \Delta\phi^p - \psi(x_k, y_k) \leq 0$$

Proof. The assumption means that at time q_k, the position of e_k in OPT's list is at most y_k (it may be even lower, due to movements which may be performed by OPT to e_k towards the beginning of its list, after OPT served r_k). Recall that after ALG serves r_k, the position of e_k in ALG's list changes from x_k to 1, as a result of the move-to-front ALG performs on e_k. In order to prove the required inequality, we consider the following cases, depending on the value of y_k:

- The case $1 \leq x_k \leq y_k$. We have $\psi(x_k, y_k) = 7x_k$.
  Therefore, observe that it is sufficient to prove that $\Delta\phi^p \leq x_k$.
  This is indeed the case, because moving e_k from position x_k to position 1 in ALG's list required ALG to perform $x_k - 1$ swaps, each one of those caused ϕ to either increase by 1 or decrease by 1. Therefore, all these $x_k - 1$ swaps cause ϕ to increase by at most $x_k - 1$.
- The case $y_k \leq x_k \leq n$.
  On one hand, there are at least $x_k - y_k$ elements which were before e_k in ALG's list and after e_k in OPT's list before the move-to-front ALG performed on e_k, but they will be after e_k in ALG's list after this move-to-front. This causes ϕ to decrease by at least $x_k - y_k$. On the other hand, there are at most $y_k - 1$ elements which were before e_k in both OPT's list and ALG's list before the move-to-front ALG performed on e_k, but they will be after e_k in ALG's list after this move-to-front. This causes ϕ to increase by at most $y_k - 1$. Therefore, we have that

  $$\Delta\phi^p \leq -(x_k - y_k) + (y_k - 1) = 2y_k - x_k - 1$$

  Hence,

  $$3x_k + 4 \cdot \Delta\phi^p \leq 3x_k + 4 \cdot (2y_k - x_k - 1) \leq 8y_k - x_k$$

  Now we distinguish between these two following cases, depending on the value of y_k:
  - The case $y_k \leq x_k \leq 8y_k$. We have that $\psi(x_k, y_k) = 8y_k - x_k$. Hence

    $$3x_k + 4 \cdot \Delta\phi^p - \psi(x_k, y_k) \leq 8y_k - x_k - \psi(x_k, y_k) = 8y_k - x_k - (8y_k - x_k) = 0$$

  - The case $8y_k \leq x_k \leq n$. We have that $\psi(x_k, y_k) = 0$. Hence

    $$3x_k + 4 \cdot \Delta\phi^p - \psi(x_k, y_k) \leq 8y_k - x_k - \psi(x_k, y_k) = 8y_k - x_k \leq 8y_k - 8y_k = 0$$

◀

Now we are ready to complete the proof of Lemma 35.

Proof of Lemma 35. Since OPT has already served this request, we have $OPT^p = 0$. As explained in the proof of Lemma 26, we have $ALG^p \leq 3x_k$. Therefore, we are left with the task to prove that

$$3x_k + \Delta\Phi^p \leq 0$$

Observe that $\psi(x_k, y_k)$ and $\mu_k(t)$ are dropped (and thus are subtracted) from Φ as a result of ALG serving r_k. Therefore, we are left with the task to prove that

$$3x_k + 4 \cdot \Delta\phi^p - \psi(x_k, y_k) - 4 \cdot \mu_k(t) \leq 0$$

We first assume that ever since OPT served r_k until ALG served r_k, OPT did not increase the position of e_k in its list (later we remove this assumption). This assumption means that $\mu_k(t) = 0$. Therefore, due to Lemma 36, we have that the above inequality holds. We are left with the task to prove that the above inequality continues to hold even without this assumption.

Assume that ever since OPT served r_k until ALG served r_k, OPT performed a swap between e_k and another element where e_k's position has been increased as a result of this swap. We shall prove that the above inequality continues to hold nonetheless.

On one hand, this swap causes either an increase of 1 or a decrease of 1 to $\Delta\phi^p$. Therefore, the left term of the inequality will be increased by at most 4. On the other hand, the left term of the inequality will certainty be decreased by 4 as a result of this swap, because μ_k will certainty be increased by 1. To conclude, a decrease of at least $4-4=0$ will be applied to the left term of the inequality, thus the inequality will continue to hold after this swap as well.

By using the argument above for each swap of the type mentioned above, we get that the above inequality continues to hold even without the assumption that OPT did not increase the position of e_k in its list since it served r_k until ALG served r_k, thus the lemma has been proven. ◀

Now that we analyzed the event when ALG serves a request, the next target is to analyze the event where OPT serves multiple request together. The following observation contains useful properties of ψ that will be used later on. The reader may prove them algebraically.

▶ **Observation 37.** *For each $x, x', y, y' \in [1, \infty)$ such that $x \le x'$ and $y \le y'$, the function ψ satisfies the following claims:*

1. $0 \le \psi(x, y) \le 7x$.
2. *If $y \le x \le x'$ then $\psi(x', y) \le \psi(x, y)$.*
3. $\psi(x, y) \le \psi(x, y')$.

The target now is to analyze the event when OPT serves the requests R_t^{OPT} together at time t. Recall that when OPT serves a request r_k, the value $\psi(x_k, y_k)$ is added to Φ. The following lemma will be needed in order to analyze this event.

▶ **Lemma 38.** *Let $a > 0$ and let $f : [0, \infty) \to [0, \infty)$ be the function defined as follows:*

$$f(x) = \begin{cases} 7x & \text{if } 0 \le x \le a \\ 8a - x & \text{if } a \le x \le 8a \\ 0 & \text{if } 8a \le x \end{cases}$$

Consider the optimization problem Q of choosing a (possibly infinite) subset $U \subseteq (0, 8a)$ that will maximize $\sum_{x \in U} f(x)$ with the requirement $\forall x, y \in U : x < y \implies 2x \le y$. Then the optimal value of Q is $24a$.

Now we can use Lemma 38 in order to analyze the event when OPT serves multiple requests together. The proofs of Lemmas 38 and 39 appear in the full version [12].

▶ **Lemma 39.** *Let $p \in P$ be the event where OPT served the requests R_t^{OPT} at time t. We have*

$$ALG^p + \Delta\Phi^p \le 24 \cdot OPT^p$$

We analyzed the event where ALG serves a request and the event when OPT serves multiple requests together. The only event which is left to be analyzed is the event when OPT performs a swap. We analyze it in the lemma below. The proof is in the full version [12].

▶ **Lemma 40.** *Let $p \in P$ be the event where OPT performed a swap at time t. We have*

$$ALG^p + \Delta\Phi^p \le 8 \cdot OPT^p$$

We are now ready to prove Theorem 3.

Proof of Theorem 3. Due to Lemma 35, Lemma 39 and Lemma 40, we have for each event $p \in P$ that

$$ALG^p + \Delta\Phi^p \leq 24 \cdot OPT^p$$

The theorem follows by summing it up for over all events and use the fact that Φ starts with 0 and is always non-negative. ◀