Online Maintenance of k-Medians and k-Covers on a Line *

Rudolf Fleischer† Mordecai J. Golin‡ Yan Zhang

July 30, 2004

Abstract

The standard dynamic programming solution to finding *k-medians on a line* with n nodes requires $O(kn^2)$ time. Dynamic programming speed-up techniques, e.g., use of the quadrangle inequality or properties of totally monotone matrices, can reduce this to $O(kn)$ time. However, these speed-up techniques are inherently static and cannot be used in an online setting, i.e., if we want to increase the size of the problem by one new point. Then, in the worst case, we could do no better than recalculating the solution to the entire problem from scratch in $O(kn)$ time. The major result of this paper is to show that we can *maintain the dynamic programming speedup* in an online setting where points are added from left to right on a line. Computing the new k-medians after adding a new point takes only $O(k)$ amortized time and $O(k \log n)$ worst case time (simultaneously). Using similar techniques, we can also solve the *online k-coverage with uniform coverage on a line* problem with the same time bounds.

1 Introduction

In the *k-median* problem we are given a graph $G = (V, E)$ with nonnegative edge costs. We want to choose k nodes (the *medians*) from V so as to minimize the sum of the distances between each node and its closest median. As motivation, the nodes can be thought of as *customers*, the medians as *service centers*, and the distance between a customer and a service center as the cost of servicing the customer from that center. In this view, the k-median problem is about choosing a set of k service centers that minimizes the total cost of servicing all customers.

The k-median problem is often extended so that each customer (node) has a weight, corresponding to the *amount* of service requested. The distance between a customer and its closest service center (median) then becomes the cost of providing *one unit* of service, i.e., the cost of servicing a customer will then be the weight of the customer node times its distance from the closest service center. Another extension of the problem is to assign a *start-up cost* to each node representing the cost of building a service center at that node. The total cost we wish to minimize is then the sum of the start-up costs of the chosen medians plus the cost of servicing each of the customer requests. This is known as the *facility location problem*.

*Authors' Address: Department of Computer Science, Hong Kong UST, Clear Water Bay, Kowloon, Hong Kong. *email:* {rudolf, golin, cszy}@cs.ust.hk

†Work partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. HKUST6010/01E)

‡Work partially supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project Nos. HKUST6162/00E, HKUST6082/01E and HKUST6206/02E)

Lin and Vitter [7] proved that, in general, even finding an $o(\ln n)$-approximate solution to the k-median problem is NP-hard. They were able to show, though, that it is possible in polynomial time to achieve a cost within $O(1+\epsilon)$ of optimal if one is allowed to use $(1+1/\epsilon)(\ln n+1)k$ medians. The problem remains hard if restricted to metric spaces. Guha and Khuller [5] proved that this problem is still MAX-SNP hard. Charikar, Guha, Tardos and Shmoys [4] showed that constant-factor approximations can be computed for any metric space. In the specific case of points in Euclidean space, Arora, Raghavan, and Rao [2] developed a PTAS.

There are some special graph topologies for which fast polynomial time algorithms exist, though. In particular, this is true for trees [8, 10] and lines [6]. In this paper we will concentrate on the line case, in which all of the nodes lie on the real line and the distance between any two nodes is the Euclidean distance. See Fig. 1 for the exact definition of the k-median on a line problem (k`ML`) and Fig. 2 for an illustration.

The k-Median on a Line Problem (k`ML`)

Let $k \geq 0$. Let $x_1 < x_2 < \cdots < x_n$ be points on the real line. With each point x_j there are associated a *weight* $w_j \geq 0$ and a *start-up cost* $c_j \geq 0$. A *k-placement* is a subset $S \subseteq V_m = \{x_1, \ldots, x_m\}$ of size $|S|$ at most k. We define the *distance* of point x_j to S by

$$d_j(S) = \min_{y \in S} |x_j - y| \,.$$

The *cost* of S is (i) the cost of creating the service centers in S plus (ii) the cost of servicing all of the requests from S:

$$cost(S) = \sum_{x_i \in S} c_i + \sum_{j=1}^{n} w_j d_j(S) \,.$$

The *k-median on a line problem (k`ML`)* is to find a k-placement S minimizing $cost(S)$. In *online* k`ML`, the points are given to us in the order $x_1, x_2, \ldots$, and we have to compute optimal solutions for the known points at any time.

Figure 1: The k-median on a line problem.

There is a straightforward $O(kn^2)$ dynamic programming (DP) algorithm for solving k`ML`. It fills in $\Theta(kn)$ entries in a dynamic programming table[1] where calculating each entry requires minimizing over $O(n)$ values, so the entire algorithm needs $O(kn^2)$ time. Hassin and Tamir [6] showed that this DP formulation possesses a quadrangle or concavity property. Thus, the time to calculate the table entries can be reduced by an order of magnitude to $O(kn)$ using known DP speed-up techniques, such as those found in [9]. This speed up can be viewed as providing a way to calculate each DP table entry in $O(1)$ time.

In this paper we study online k`ML`, where new points are always added to the right of old points. As will soon be seen, adding such points retains all of old entries in the dynamic programming table and only adds $O(k)$ new entries to the table. Since static k`ML` can be solved in $O(kn)$ time, or $O(1)$ (amortized) time per entry, we would hope to be able to calculate the $O(k)$ new entries in $O(k)$ total time, maintaining the dynamic programming speed up.

[1] We do not give the details here because the DP formulation is very similar to the one shown in Lemma 1.

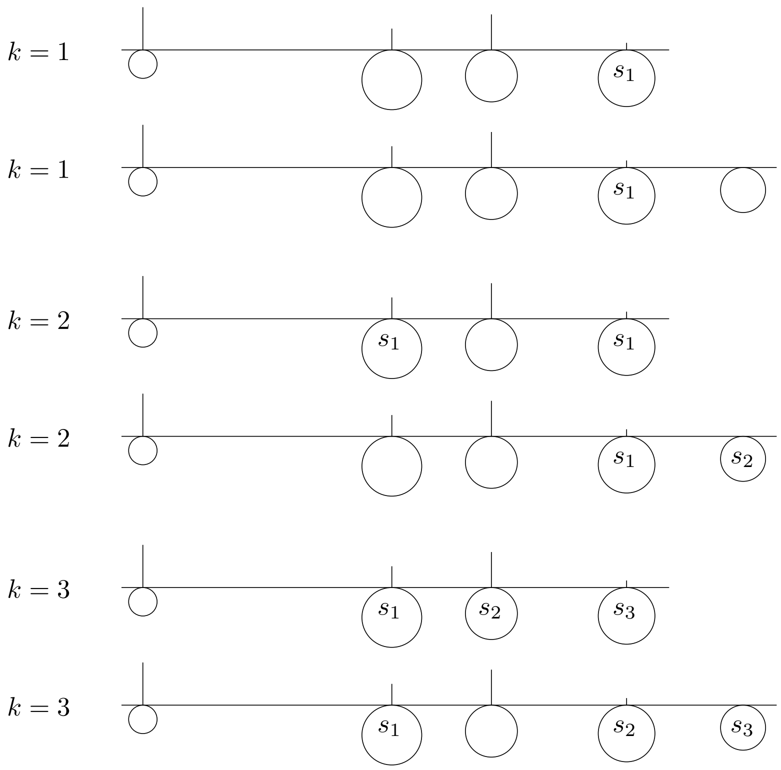

Figure 2: k-median on a line example. The data are taken from the example in Section 2.6. Each node is drawn at its x-coordinate with a vertical bar and a circle. The length of the vertical bar illustrates the start-up cost, and the area of the circle corresponds to the weight. The optimal locations of resources are indicated by s_i $(1 \leq i \leq k)$. The six figures show the optimal locations when the number of nodes is 4 or 5 and the number of resources k ranges from 1 to 3, respectively.

The difficulty here is that Hassin and Tamir's approach cannot be made online because most DP speed-up techniques such as those in [9] are inherently static. The best that can be done using their approach is to totally recompute the dynamic programming matrix entries from scratch at each step using $O(kn)$ time per step[2].

Later, Auletta, Parente and Persiano [3] studied k`ML` in the special case of *unit lengths*, i.e., $x_{i+1} = x_i + 1$ for all i, and *no start up costs*, i.e., $c_i = 0$ for all i. Being unaware of Hassin and Tamir's results they developed a new technique for solving the problem which enabled them to add a new point in amortized $O(k)$ time, leading to an $O(kn)$ time algorithm for the static problem.

The major contribution of this paper is to bootstrap off of Auletta, Parente and Persiano's result to solve online k`ML` when (i) the points can have arbitrary distances between them and (ii) start up costs are allowed. In Section 2 we prove the following theorem.

Theorem 1 *We can solve the online k-median on a line problem in $O(k)$ amortized and $O(k \log n)$ worst case time per update. These time bounds hold simultaneously.* □

A variant of k`ML` is the *k-coverage problem (k`CL`)* where the cost of servicing customer x_j is zero if it is closer than r_j to a service center, or w_j otherwise. See Fig. 3 for the exact definition of k`CL` and Fig. 4 for an illustration.

The k-Coverage on a Line Problem (k`CL`)

In addition to the requirements of k`ML`, each node x_j is also given a *coverage radius* r_j. It is *covered* by a k-placement S if $d_j(S) \leq r_j$. In that case, the service cost for x_j is zero. Otherwise, the service cost is w_j. The cost of S is then

$$cost(S) = \sum_{x_i \in S} c_i + \sum_{j=1}^{n} w_j I_j(S) ,$$

where

$$I_j(S) = \begin{cases} 0 & \text{if } d_j(S) \leq r_j \\ 1 & \text{if } d_j(S) > r_j . \end{cases}$$

The *k-coverage on a line problem (k`CL`)* is to find a k-placement S minimizing $cost(S)$. *Online* k`CL` is defined similarly to online k`ML`.

Figure 3: The k-coverage on a line problem.

Hassin and Tamir [6] showed how to solve static k`CL` in $O(n^2)$ time (independent of k), again using the quadrangle inequality/concavity property. In Section 3 we restrict ourselves to the special case of uniform coverage, i.e., there is some $r > 0$ such that $r_j = r$ for all j. In this situation we can use a similar (albeit much simpler) approach as in Section 2 to maintain optimal partial solutions S as points are added to the right of the line. In Section 3 we will prove the following theorem.

Theorem 2 *We can solve the online k-coverage on a line problem with uniform coverage in $O(k)$ amortized and $O(k \log n)$ worst case time per update. These time bounds hold simultaneously.* □

[2] Although not stated in [6] it is also possible to reformulate their DP formulation in terms of finding row-minima in k $n \times n$ *totally monotone matrices* and then use the SMAWK algorithm [1] — which finds the row-minima of an $n \times n$ totally monotone matrix in $O(n)$ time — to find an $O(kn)$ solution. This was done explicitly in [11]. Unfortunately, the SMAWK algorithm is also inherently static, so this approach can also not be extended to solve the online problem.

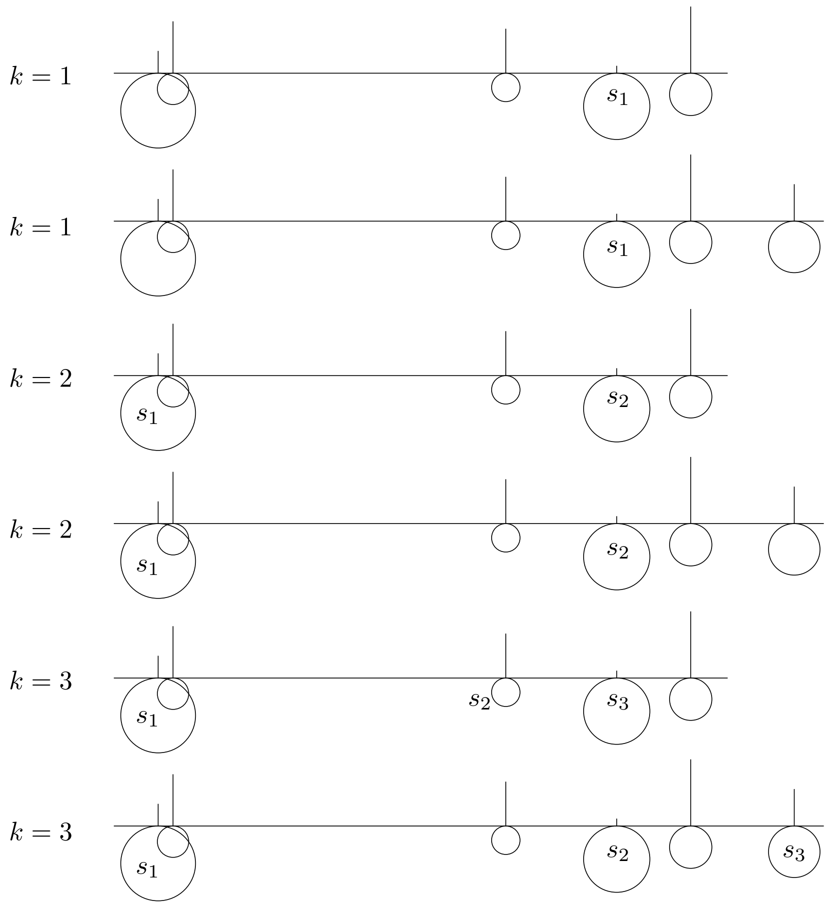

Figure 4: k-coverage on a line example. The data are taken from the example in Section 3.1. Each node is drawn at its x-coordinate with a vertical bar and a circle. The length of the vertical bar illustrates the start-up cost, and the area of the circle corresponds to the weight. The optimal locations of resources are indicated by s_i $(1 \leq i \leq k)$. The six figures show the optimal locations when the number of nodes is 5 or 6 and the number of resources k ranges from 1 to 3, respectively.

2 The k-Median Problem

2.1 Notations and Preliminary Facts

In the online k-median problem, we start with an empty line and, at each step, append a new node to the right of all of the previous nodes. So, at step m we will have m points

$$x_1 < x_2 < \cdots < x_m$$

and when adding the $(m+1)$st point we have $x_m < x_{m+1}$. Each node x_j will have a weight w_j, and a start-up cost c_j associated with it. At step m, the task is to pick a set S of at most k nodes from $x_1, x_2, \ldots, x_m$ that minimizes

$$cost(S) = \sum_{x_i \in S} c_i + \sum_{j=1}^{m} w_j d_j(S)\,. \tag{1}$$

Our algorithm actually keeps track of $2k$ median placements for every step. The first k placements will be optimal placements for *exactly* i resources, for $1 \le i \le k$. More specifically, let

$$OPT_i(m) = \min_{S \subseteq V_m,\, |S|=i} \left(\sum_{x_i \in S} c_i + \sum_{j=1}^{m} w_j d_j(S) \right)\,.$$

We will see later how to efficiently compute all the $OPT_i(m)$ values during step m. Once the $OPT_i(m)$ values are calculated, they will be kept for the rest of the algorithm.

The remaining k placements are called *pseudo-optimal* placements. These are optimal placements under the constraint that x_m must be one of the chosen resources. That is, for $i = 1, \ldots, k$

$$POPT_i(m) = \min_{\substack{S \subseteq V_m,\, |S|=i \\ x_m \in S}} \left(\sum_{x_i \in S} c_i + \sum_{j=1}^{m} w_j d_j(S) \right)\,.$$

In particular, note that if $i = 1$, then $S = \{x_m\}$ and

$$POPT_1(m) = c_m + \sum_{j=1}^{m-1} w_j (x_m - x_j). \tag{2}$$

As with $OPT_i(m)$, all of these values are computed in step m and, once computed, will be kept for the rest of the algorithm. Optimal and pseudo-optimal placements are related by the following straightforward equations.

Lemma 1

$$OPT_i(m) = \min_{1 \le j \le m} \left(POPT_i(j) + \sum_{l=j+1}^{m} w_l \cdot d(j,l) \right) \tag{3}$$

$$POPT_i(m) = \min_{1 \le j \le m-1} \left(OPT_{i-1}(j) + \sum_{l=j+1}^{m-1} w_l \cdot d(l,m) \right) + c_m\,, \tag{4}$$

where $d(j,l) = x_l - x_j$ is the distance between x_j and x_l. □

Proof: In (3), index j corresponds to the choice of location of the rightmost median. Given that the rightmost median is at location j, $POPT_i(j)$ is the best way of servicing all of the nodes $x_1, \ldots, x_j$ and $\sum_{l=j+1}^{m} w_l \cdot d(j,l)$ is the cost of servicing nodes $x_{j+1}, \ldots, x_m$ (using node j).

In (4), we assume that there is a median at node m. Index j is the rightmost node that is *not* serviced by node m so $OPT_{i-1}(j)$ is the best way of servicing nodes $x_1, \ldots, x_j$ using the remaining $i-1$ medians while $\sum_{l=j+1}^{m-1} w_l \cdot d(l,m)$ is the cost of servicing nodes $x_{j+1}, \ldots, x_m$ (using node m). $\square$

Denote by $MIN_i(m)$ the index j at which the "min" operation in Eq. (3) achieves its minimum value and by $PMIN_i(m)$ the index j at which the "min" operation in Eq. (4) achieves its minimum value. When computing the $OPT_i(m)$ and $POPT_i(m)$ values the algorithm will also compute and keep the $MIN_i(m)$ and $PMIN_i(m)$ indices.

The optimum cost we want to find is $OPT = \min_{1 \le i \le k}(OPT_i(n))$.[3] It is not difficult to see that, knowing all values of $OPT_i(m)$, $MIN_i(m)$, $POPT_i(m)$ and $PMIN_i(m)$ for $1 \le i \le k$, $1 \le m \le n$, we can unroll the equations in Lemma 1 in $O(k)$ time to find the optimal set S of at most k medians that yields OPT. So, maintaining these $4nk$ variables suffices to solve the problem.

A straightforward calculation of the minimizations in Lemma 1 permits calculating the value of $POPT_i(m)$ from those of $OPT_{i-1}(j)$ in $O(m)$ time and the value of $OPT_i(m)$ from those of $POPT_i(j)$ in $O(m)$ time. This permits a dynamic programming algorithm that calculates *all* of the $OPT_i(m)$ and $POPT_i(m)$ values in $O\left(k \sum_{m=1}^{n} m\right) = O(kn^2)$ time, solving the problem. Section 2.6 provides a worked example of $OPT_i(j)$ and $POPT_i(j)$ values and how they provide a solution.

As discussed in the previous section, this is very slow. The rest of this section is devoted to improving this by an order of magnitude; developing an algorithm that, at step m for each i, will calculate the value of $POPT_i(m)$ from those of $OPT_{i-1}(m)$ and the value of $OPT_i(m)$ from those of $POPT_i(m)$ in $O(1)$ amortized time and $O(\log n)$ worst case time.

2.2 The Functions $V_i(j,m,x)$ and $V_i'(j,m,x)$

As mentioned, our algorithm is actually an extension of the algorithm in [3]. In that paper, the authors defined two sets of functions which played important roles. We start by rewriting those functions using a slightly different notation which makes it easier to generalize their use. For all $1 \le i \le k$ and $1 \le j \le m$ define

$$V_i(j,m,x) = POPT_i(j) + \sum_{l=j+1}^{m} w_l \cdot d(j,l) + x \cdot d(j,m)\,. \tag{5}$$

For all $1 \le i \le k$ and $1 \le j \le m-1$ define

$$V_i'(j,m,x) = OPT_{i-1}(j) + \sum_{l=j+1}^{m-1} w_l \cdot d(l,m) + x \cdot \sum_{l=j+1}^{m-1} w_l\,. \tag{6}$$

Then Lemma 1 can be written as

$$OPT_i(m) = \min_{1 \le j \le m} V_i(j,m,0) \tag{7}$$

$$POPT_i(m) = \min_{1 \le j \le m-1} V_i'(j,m,0) + c_m\,. \tag{8}$$

[3]Note that the optimum might *not* be $OPT_k(n)$. That is, startup costs might be so expensive that it can sometimes be cheaper not to use all k allowed vertices. Section 2.6 provides a concrete example of this.

The major first point of departure between this section and [3] is the following lemma, which basically says that $V_i(j,m,x)$ and $V_i'(j,m,x)$ can be computed in constant time when needed. This will permit us to design an algorithm that works efficiently online.

Lemma 2 *Suppose we are given,*

$$W(m)=\sum_{l=1}^{m} w_l \quad \textit{and} \quad M(m)=\sum_{l=1}^{m} w_l\cdot d(1,l)\ .$$

Then, given the values of $POPT_i(j)$, the function $V_i(j,m,x)$ can be evaluated at any x in constant time. Similarly, given the values of $OPT_{i-1}(j)$, the function $V_i'(j,m,x)$ can be evaluated at any x in constant time.

Proof: We first examine $V_i(j,m,x)$. We already know $POPT_i(j)$ so we only need to compute the terms

$$\sum_{l=j+1}^{m} w_l\cdot d(j,l)+x\cdot d(j,m).$$

It is easy to verify that

$$\sum_{l=j+1}^{m} w_l\cdot d(j,l)=[M(m)-M(j)]-[W(m)-W(j)]\cdot d(1,j)$$

which can be computed in constant time. For $V_i'(j,m,x)$, we also only need to compute

$$\sum_{l=j+1}^{m-1} w_l\cdot d(l,m)+x\cdot\sum_{l=j+1}^{m-1} w_l.$$

But

$$\sum_{l=j+1}^{m-1} w_l\cdot d(l,m)=[W(m-1)-W(j)]\cdot d(1,m)-[M(m-1)-M(j)]$$

and

$$\sum_{l=j+1}^{m-1} w_l=W(m-1)-W(j)$$

which can both be computed in constant time. □

In the next two subsections we will see how to use this lemma to efficiently calculate $POPT_i(j)$ and $OPT_i(j)$.

2.3 Computing $OPT_i(m)$

We start by explaining how to maintain the values of $OPT_i(m)$. Our algorithm uses k similar data structures to keep track of the k sets of $OPT_i(m)$ values, for $1\le i\le k$. Since these k structures are essentially the same we will fix i and consider how the i^{th} data structure permits the computation of the values of $OPT_i(m)$ as m increases.

2.3.1 The Data Structures

Recall Eq. (5). Consider the m functions $V_i(j, m, x)$ for $1 \le j \le m$. They are all linear functions in x so the lower envelope of these functions is a piecewise linear function to which each $V_i(j, m, x)$ contributes at most one segment.

We are only interested in $OPT_i(m) = \min_{1\le j\le m} V_i(j, m, 0)$ (Eq. (7)) which is equivalent to evaluating this lower envelope at $x = 0$. In order to update the data structure efficiently, though, we will see that we will need to store the *entire* lower envelope for $x \ge 0$. We store the envelope by storing the *changes* in the envelope.

More specifically, our data structure for computing the values of $OPT_i(m)$ consists of two arrays

$$\Delta_i(m) = (\delta_0, \delta_1, \ldots, \delta_s) \tag{9}$$

and

$$Z_i(m) = (z_1, \ldots, z_s), \tag{10}$$

such that

$$\text{if } \delta_{h-1} < x + W(m) < \delta_h, \text{ then } V_i(z_h, m, x) = \min_{j\le m} V_i(j, m, x)\,. \tag{11}$$

The reasons for the shift term $W(m) = \sum_{l=1}^{m} w_l$ will become clear later. Since we only keep the lower envelope for $x \ge 0$, we have $\delta_0 \le W(m) < \delta_1$.

An important observation is that the slope of $V_i(j, m, x)$ is $d(j, m)$ which decreases as j increases, so we have $z_1 < \cdots < z_s$ and $z_s = m$ at step m. In particular, note that $V(m, m, x)$, which is the rightmost part of the lower envelope, has slope $0 = d(m, m)$ and is a horizontal line.

Given this data structure, computing the value of $OPT_i(m)$ becomes trivial. We simply have $MIN_i(m) = z_1$ and $OPT_i(m) = V_i(z_1, m, 0)$.

2.3.2 Updating the Data Structures

After all of the setup this subsection is the heart of the algorithm and explains why the algorithm is efficient. Assume that the data structure given by Eq. (9), (10) and (11) is storing the lower envelope after step m and, in step $m + 1$, point x_{m+1} is added. We now need to recompute the lower envelope of $V_i(j, m + 1, x)$, for $1 \le j \le m + 1$ and $x \ge 0$. Note that in step m we have m functions

$$\{V_i(j, m, x) \,:\, 1 \le j \le m\}$$

but we now have $m + 1$ functions

$$\{V_i(j, m + 1, x) \,:\, 1 \le j \le m + 1\}\,.$$

If we only consider the lower envelope of the first m functions $V_i(j, m+1, x)$ for $1 \le j \le m$, then the following lemma guarantees that the two arrays $\Delta_i(m)$ and $Z_i(m)$ do not change.

Lemma 3 *Assume $V_i(z_h, m, x)$ minimizes $V_i(j, m, x)$ for $1 \le j \le m$ when $\delta_{h-1} < x + W(m) < \delta_h$. Then $V_i(z_h, m + 1, x)$ minimizes $V_i(j, m + 1, x)$ for $1 \le j \le m$ when $\delta_{h-1} < x + W(m + 1) < \delta_h$.*

Proof: It is easy to verify that for $1 \le j \le m$

$$V_i(j, m + 1, x) = V_i(j, m, x + w_{m+1}) + (x + w_{m+1}) \cdot d(m, m + 1)\,.$$

Since $\delta_{h-1} < x + W(m + 1) < \delta_h$ iff $\delta_{h-1} < (x + w_{m+1}) + W(m) < \delta_h$, the above formula is minimized when $j = z_h$. □

This lemma is the reason for defining Eq. (9), (10) and (11) as we did with the shift term instead of simply keeping the breakpoints of the lower envelope in $\Delta_i(m)$. Note that the lemma *does not* say that the lower envelope of the functions remains the same (this could not be true since all of the functions have been changed). What the lemma does say is that the *structure* of the breakpoints of the lower envelope is the same after the given shift.

Now, we consider $V_i(m+1, m+1, x)$. As discussed in the previous subsection, $V_i(m+1, m+1, x)$ is the rightmost segment of the lower envelope *and* is a horizontal line. So, we only need to find the intersection point between the lower envelope of $V_i(j, m+1, x)$ for $1 \leq j \leq m$ and the horizontal line $y = V_i(m+1, m+1, x)$. Assume they intersect at the segment $V_i(z_{max}, m+1, x)$. Then, $Z_i(m+1)$ becomes $(z_1, \ldots, z_{max}, m+1)$, and $\Delta_i(m+1)$ changes correspondingly.

We can find this point of intersection either by using a binary search or a sequential search. The binary search would require $O(\log m)$ worst case comparisons between $y = V_i(m+1, m+1, x)$ and the lower envelope. The sequential search would scan the array $Z_i(m)$ from right to left, i.e. from z_s to z_1, discarding segments from the lower envelope until we find the intersection point of $y = V_i(m+1, m+1, x)$ with points on the lower envelope. The sequential search might take $\Theta(m)$ time in the worst case but only uses $O(1)$ in the amortized case since lines thrown off the lower envelope will never be considered again in a later step.

In both methods a comparison operation requires being able to compare the constant $V_i(m+1, m+1, x)$ to $V_i(j, m+1, x)$ for some j and some arbitrary value m. Recall from Lemma 2 that we can evaluate $V_i(j, m+1, x)$ at any particular x in constant time. Thus, the total time required to update the lower envelope is $O(\log m)$ worst case and $O(1)$ amortized.

To combine the two bounds we perform the sequential and binary search alternately, i.e., we use sequential search in odd numbered comparisons and binary search in even numbered comparisons. The combined search finishes when the intersection value is first found. Thus, the running time is proportional to the one that finishes first and we achieve both the $O(1)$ amortized time and the $O(\log m)$ worst case time.

Since we only keep the lower envelope for $x \geq 0$, we also need to remove from $Z_i(m+1)$ and $\Delta_i(m+1)$ the segments corresponding to negative x values. Set $z_{min} = \max\{z_h : \delta_{h-1} < W(m+1) < \delta_h\}$. Then $Z_i(m+1)$ should be $(z_{min}, \ldots, z_{max}, m+1)$, and $\Delta_i(m+1)$ should change correspondingly.

To find z_{min}, we also use the technique of combining sequential search and binary search. In the sequential search, we scan from left to right, i.e., from z_1 to z_s. The combined search also requires $O(1)$ amortized time and $O(\log m)$ worst case time.

2.4 Computing $POPT_i(m)$

In the previous section we showed how to update the values of $OPT_i(m)$ by maintaining a data structure that stores the lower envelope of $V_i(j, m, x)$ and evaluating the lower envelope at $x = 0$, i.e., $OPT_i(m) = \min_{1 \leq j \leq m} V_i(j, m, 0)$. In this section we will show how, in a very similar fashion, we can update the values of $POPT_i(m)$ by maintaining a data structure that stores the lower envelope of $V'_i(j, m, x)$. We can then use Eq. (8) to find

$$POPT_i(m) = c_m + \min_{1 \leq j \leq m-1} V'_i(j, m, 0),$$

i.e., evaluating the lower envelope at $x = 0$ and adding c_m.

As before we will be able to maintain the lower envelope of $V'_i(j, m, x)$, $1 \leq j \leq m-1$, in $O(1)$ amortized time and $O(\log m)$ worst case time. The data structure is almost the same as the one for maintaining $V_i(j,m,x)$ in the previous section so we only quickly sketch the ideas.

As before the algorithm uses k similar data structures to keep track of the k lower envelopes; for our analysis we fix i and consider the data structures for maintaining the lower envelope of $V_i'(j,m,x)$ (and thus $POPT_i(m)$) as m increases.

2.4.1 The Data Structures

By their definitions in Eq. (6) the $m-1$ functions $V_i'(j,m,x)$, for $1 \le j \le m-1$, are all linear functions, so their lower envelope is a piecewise linear function to which each $V_i(j,m,x)$ contributes at most one segment.

As before, in order to compute the values of $POPT_i(m)$, we only need to know the value of the lower envelope at $x = 0$ but, in order to update the structure efficiently, we will need to store the entire lower envelope.

Our data structures for computing the values of $POPT_i(m)$ consist of two arrays

$$\Delta_i'(m) = (\delta_0', \delta_1', \ldots, \delta_s') \tag{12}$$

and

$$Z_i'(m) = (z_1', \ldots, z_s'), \tag{13}$$

such that

$$\text{if } \delta_{h-1}' < x + d(1,m) < \delta_h', \text{ then } V_i'(z_h', m, x) = \min_{j \le m-1} V_i'(j,m,x)\,. \tag{14}$$

Since we only keep the lower envelope for $x \ge 0$, we have $\delta_0' \le d(1,m) < \delta_1'$. Since the slopes $\left(\sum_{l=j+1}^{m-1} w_l\right)$ of $V_i'(j,m,x)$ decrease when j increases, we have $z_1' < \cdots < z_s'$ and $z_s' = m-1$ at step m. In particular, note that $V'(m-1,m,x)$, the rightmost part of the lower envelope, has slope 0 and is a horizontal line.

Given such data structures, computing the value of $POPT_i(m)$ becomes trivial. We simply have $PMIN_i(m) = z_1'$ and $POPT_i(m) = c_m + V_i'(z_1', m, 0)$.

2.4.2 Updating the Data Structures

Given the lower envelope of $V_i'(j,m,x)$, for $1 \le j \le m-1$ at step m we need to be able to recompute the lower envelope of $V_i'(j,m+1,x)$, for $1 \le j \le m$ after x_{m+1} is added.

As before, we will first deal with the functions $V_i'(j,m+1,x)$ for $1 \le j \le m-1$, and then later add the function $V_i'(m,m+1,x)$.

If we only consider the functions $V_i'(j,m+1,x)$ for $1 \le j \le m-1$, we have an analogue of Lemma 3 for this case that guarantees that the two arrays $\Delta_i'(m)$ and $Z_i'(m)$ do not change.

Lemma 4 *Assume $V_i'(z_h', m, x)$ minimizes $V_i'(j,m,x)$ for $1 \le j \le m-1$ when $\delta_{h-1}' < x + d(1,m) < \delta_h'$. Then $V_i'(z_h', m+1, x)$ minimizes $V_i'(j,m+1,x)$ for $1 \le j \le m-1$ when $\delta_{h-1}' < x + d(1,m+1) < \delta_h'$.*

Proof: It is easy to verify that for $1 \le j \le m-1$

$$V_i'(j,m+1,x) = V_i'(j,m,x + d(m,m+1)) + (x + d(m,m+1)) \cdot w_{m+1}\,.$$

Since $\delta_{h-1}' < x + d(1,m+1) < \delta_h'$ iff $\delta_{h-1}' < (x + d(m,m+1)) + d(1,m) < \delta_h'$, the above formula is minimized when $j = z_h'$. □

Since $V_i'(m, m+1, x)$ must be the rightmost segment of the lower envelope, we only need to find the intersection point between the lower envelope of $V_i'(j, m+1, x)$ for $1 \leq j \leq m-1$ and the line $y = V_i'(m, m+1, x)$. Assume they intersect at the segment of $V_i'(z'_{max}, m+1, x)$. Then, $Z_i'(m+1)$ becomes $(z'_1, \ldots, z'_{max}, m)$, and $\Delta_i'(m+1)$ changes correspondingly.

We again use both a binary and a sequential search to find z'_{max}, alternating between the steps of the two. The binary search requires $O(\log m)$ time in the worst case. The sequential search scans the array $Z_i'(m)$ from right to left and requires $O(1)$ time amortized time. The total search therefore requires $O(1)$ amortized and $O(\log m)$ worst case time per step (simultaneously).

Since we only keep the lower envelope for $x \geq 0$, we also need to remove from $Z_i'(m+1)$ and $\Delta_i'(m+1)$ the segments corresponding to negative x values. Set $z'_{min} = \max\{z'_h : \delta'_{h-1} < d(1, m+1) < \delta'_h\}$; then $Z_i(m+1)$ should be $(z'_{min}, \ldots, z'_{max}, m)$, and $\Delta_i'(m+1)$ should change correspondingly. Also, z'_{min} can be found by a combined binary/sequential search in both $O(1)$ amortized and $O(\log m)$ worst case time per step (simultaneously).

2.5 The Algorithm

Given the data structures developed in the previous section the algorithm is very straightforward. After nodes $x_1 < x_2 < \cdots < x_m$ have been processed in step m the algorithm maintains

- $W(j) = \sum_{l=1}^{j} w_l$ and $M(j) = \sum_{l=1}^{j} w_l \cdot d(1, l)$, for $1 \leq j \leq m$.
- For $1 \leq i \leq k$, the data structures described in Sections 2.3.1 and 2.4.1 for storing the lower envelopes $\min_{j \leq m} V_i(j, m, x)$ and $\min_{j \leq m-1} V_i'(j, m, x)$.
- For $1 \leq i \leq k$ and $1 \leq j \leq m$, all of the values $OPT_i(j)$, $POPT_i(j)$ and corresponding indices $MIN_i(j)$, $PMIN_i(j)$.

After adding x_{m+1} with associated values w_{m+1} and c_{m+1} the algorithm updates its data structures by

- Calculating $W(m+1) = W(m) + w_{m+1}$ and $M(m+1) = M(m) + w_{m+1} d(1, m+1)$ in $O(1)$ time.
- Updating the $2k$ lower envelopes as described in Sections 2.3.2 and 2.4.2 in $O(\log m)$ worst case and $O(1)$ amortized time (simultaneously) per envelope.
- For $1 \leq i \leq k$, calculating $OPT_i(m+1) = \min_{j \leq m+1} V_i(j, m+1, 0)$ and $POPT_i(m+1) = c_m + \min_{j \leq m} V_i'(j, m+1, 0)$ in $O(1)$ time each.

Thus, in each step, the algorithm uses, as claimed, only a total of $O(k \log n)$ worst case and $O(k)$ amortized time (simultaneously).

The algorithm above only fills in the dynamic programming table. But, given the values $OPT_i(j)$, $POPT_i(j)$ and the corresponding indices $MIN_i(j)$, $PMIN_i(j)$ one can construct the optimal set of medians in $O(k)$ time so this fully solves the problem and finishes the proof of Theorem 1.

2.6 A k-Median Example

We show an example for illustration. $n = 9$ is the total number of nodes, and $k = 3$ is the maximum number of resources. The x-coordinates of the 9 nodes are 0, 5, 7, 10, 12, 13, 55, 72, 90. The start-up costs c_j of the nodes are 5400, 2100, 3100, 100, 0, 9900, 8100, 7700, 13000. And the weights w_j are 14, 62, 47, 51, 35, 8, 26, 53, 14.

Tables 1 to 4 show the values of OPT, MIN, $POPT$ and $PMIN$, respectively. From these tables, we can see that the optimal placement of 3 resources to cover all 9 points is to place two resources at x_4 and x_5 (and do not use the third resource).

Figure 5 shows the functions $V_2(j, 8, x)$ and $V_2(j, 9, x)$. If $m = 8$, the two arrays for the lower envelope are $Z_2(8) = (5, 8)$ and $\Delta_2(8) = (296, 361.5, +\infty)$. If $m = 9$, the two arrays for the lower envelope are $Z_2(9) = (5, 8, 9)$ and $\Delta_2(9) = (310, 361.5, 669.4, +\infty)$. As we can see, the intersection point of line 5 and line 8 in the upper part of Figure 5 shifts to the left by w_9 when we add x_9 in the next step (lower half of the figure), i.e., from 65.6 to 51.5. Actually, all intersection points will shift the same amount when a new node is added. That is why the partitioning value 361.5 does not change in the arrays $\Delta_2(8)$ and $\Delta_2(9)$ ($361.5 = 65.5 + W(8) = 51.5 + W(9)$).

3 The k-Coverage Problem

In this section we describe how to solve online k`CL` with uniform coverage, i.e., to maintain a k-placement S minimizing

$$cost(S) = \sum_{x_i \in S} c_i + \sum_{j=1}^{n} w_j I_j(S) ,$$

where

$$I_j(S) = \begin{cases} 0 & \text{if } d_j(S) \leq r \\ 1 & \text{if } d_j(S) > r \end{cases}$$

as m grows, where r is some fixed constant. As we will see, this problem has a simpler DP solution than the k-median problem, albeit one with a similar flavor.

In what follows we will say that x_j is *covered* by a point in S if $d_j(S) \leq r$. For a point x_j, let cov_j denote the index of the smallest of the points $x_1, \ldots, x_j$ covered by x_j, and unc_j the index of largest of the points $x_1, \ldots, x_j$ not covered by x_j:

$$cov_j = \min\{i \,:\, i \leq j \text{ and } r + x_i \geq x_j\} \quad \text{and}$$

$$unc_j = \max\{i \,:\, i < j \text{ and } r + x_i < x_j\} .$$

Note that x_{unc_j} is the point to the left of x_{cov_j}, i.e., $unc_j = cov_j - 1$ if this point exists. The points that can cover x_j are exactly the points in $[x_{cov_j}, x_j]$. As before, let $OPT_i(m)$ denote the minimum cost of an i-cover for the first m points $x_1, \ldots, x_m$, for $i = 1, \ldots, k$. If all start-up costs c_i are zero, we can iteratively, for $m = 1, \ldots, n$, compute the value of $OPT_i(m)$ as

$$OPT_i(m) \quad = \quad \min\{OPT_i(m-1) + w_m, OPT_{i-1}(unc_{cov_m})\} , \tag{15}$$

for $i = 1, \ldots, k$. The first term in the minimum accounts for the possibility that the new point x_m is not covered in the optimal solution $OPT_i(m)$. The second term is the minimum cost if x_m is covered, because in this case we can assume that the service center covering x_m is located at cov_m (it cannot be cheaper to place it further to the right) and thus covers all of the points between $x_{unc_{cov_m}}$ (not included) and x_m. Computing cov_m and unc_m, the point to its left takes time $O(\log n)$ in the worst case, but only constant amortized time over all iterations. (Once cov_j and unc_j are known for all $j \leq m$, unc_{cov_m} itself can be calculated in constant time.) Thus, the time per iteration to compute all the values $OPT_i(m)$, $1 \leq i \leq k$, is $O(k)$ amortized and $O(k + \log n)$ worst case.

		m								
		1	2	3	4	5	6	7	8	9
i	1	5400	2170	2264	691	761	785	1955	5241	6337
	2	-	7500	5270	2364	691	699	1817	4997	6089
	3	-	-	10600	5370	2364	2372	3490	6670	7762

Table 1: The values of $OPT_i(m)$.

		m								
		1	2	3	4	5	6	7	8	9
i	1	1	2	2	4	4	4	4	4	5
	2	-	2	3	4	5	5	5	5	5
	3	-	-	3	4	5	5	5	5	5

Table 2: The values of $MIN_i(m)$.

		m								
		1	2	3	4	5	6	7	8	9
i	1	5400	2170	3322	691	939	11048	18362	22093	32721
	2	-	7500	5270	2364	691	10626	8885	8927	15649
	3	-	-	10600	5370	2364	10591	8799	8841	15563

Table 3: The values of $POPT_i(m)$.

		m								
		1	2	3	4	5	6	7	8	9
i	1	-	-	-	-	-	-	-	-	-
	2	-	1	2	3	4	4	6	6	6
	3	-	-	2	3	4	5	6	6	6

Table 4: The values of $PMIN_i(m)$.

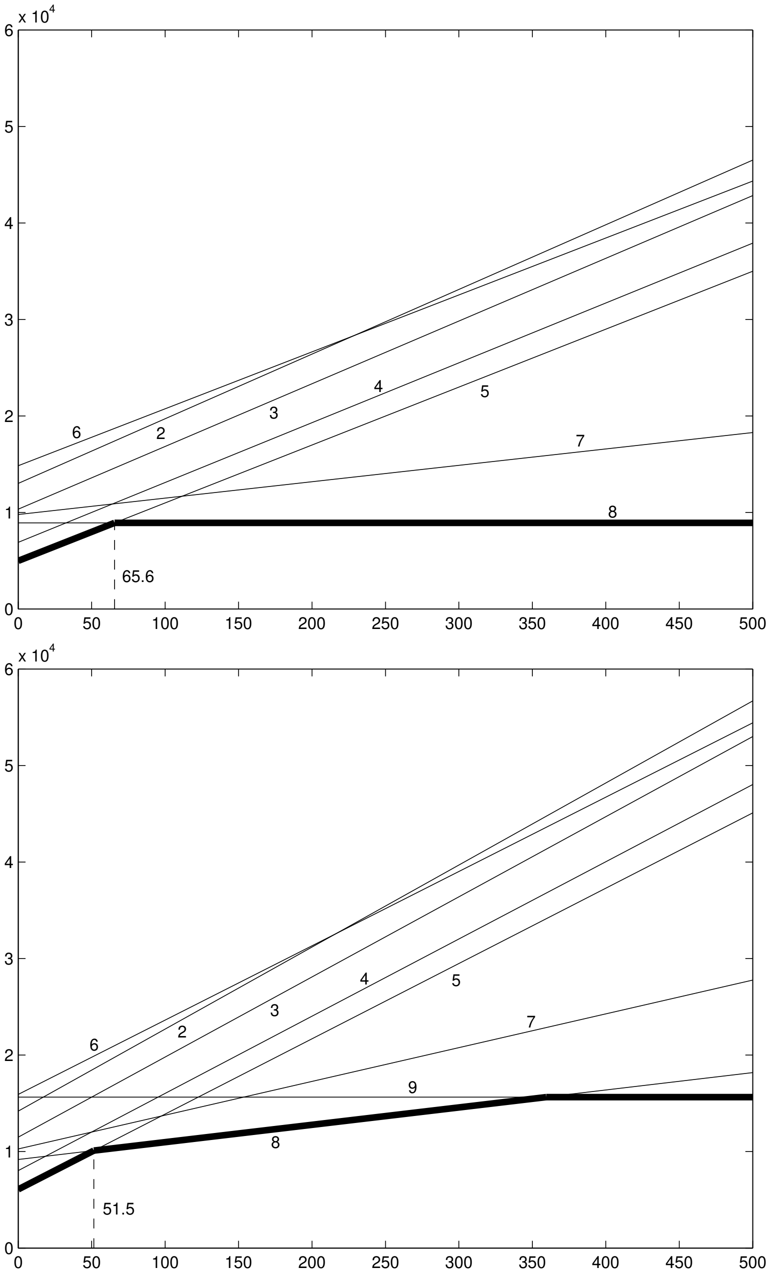

Figure 5: Functions $V_2(j, 8, x)$, for $j = 2, \ldots, 8$, and $V_2(j, 9, x)$, for $j = 2, \ldots, 9$. The lines are labelled by j. The thick lines are the lower envelopes.

If the costs c_i are not all zero, then Eq. (15) becomes the two step recurrence:

$$OPT_i(m) = \min\left\{ w_m + OPT_i(m-1)\,,\ \min_{cov_m \le j \le m} POPT_i(j) \right\} \tag{16}$$
$$POPT_i(m) = c_m + \min_{unc_m \le j \le m-1} OPT_{i-1}(j). \tag{17}$$

$POPT_i(m)$ is the minimum cost of covering $x_1, \ldots, x_m$ if x_m is one of the resources. The first term in the minimum of Eq. (16) corresponds to the possibility that x_m is not covered; the second term to the possibility that x_m is covered. It ranges over all possible covers.

In order to solve the problem in an online fashion we will need to be able to calculate the values of $OPT_i(m)$ and $POPT_i(m)$ efficiently at step m when processing x_m. We have already seen that it is possible to maintain cov_j and unc_j for $j \le m$ in $O(1)$ amortized and $O(\log m)$ worst case time per step. The only hard part that remains would be calculating $\min_{cov_m \le j \le m} POPT_i(j)$ and $\min_{unc_m \le j \le m-1} OPT_{i-1}(j)$ efficiently for each i as m increases. We will only show how to calculate the values of $\min_{cov_m \le j \le m} POPT_i(j)$. Calculating the values of $\min_{unc_m \le j \le m-1} OPT_{i-1}(j)$ can be done similarly. Let $y_j = POPT_i(j)$ and $l_m = cov_m$. Then the problem can be restated as follows: *maintain* $\min\{y_j : l_m \le j \le m\}$ *as* m *increases under the constraint that* $l_m \le l_{m+1}$.

We can do this by keeping the sequential list of the *right-to-left minimum (RTLM) sequence* of $Y_m = \{y_j : l_m \le j \le m\}$. A point y_j is a right-to-left minimum of sequence $y_1, y_2, \ldots, y_t$ if $y_s \ge y_j$ for all $s \ge j$. For example, the RTLM of sequence $(6, 8, 2, 5, 14, 12, 10, 15)$ are $(2, 5, 10, 15)$. Note that an RTLM sequence is monotonically increasing. Given the RTLM of a sequence, the full sequence's minimum value can be calculated in constant time; it is simply the first entry in the RTLM sequence.

Along with the RTLM sequence we will also need to keep the indices corresponding to the original location of the RTLM entries in the original sequence. For example, if $(6, 8, 2, 5, 14, 12, 10, 15)$ is our original sequence with $y_1 = 6$ and $y_8 = 15$ then we will keep the corresponding indices $(3, 4, 7, 8)$ along with the RTLM sequence $(2, 5, 10, 15)$. Note that the indices sequence is also monotonically increasing. The operations that we need to perform to maintain our data structure are to update the RTLM sequence when (a) adding a new item y_{m+1} to the right of Y_m and (b) deleting $y_{l_m}, \ldots, y_{l_{m+1}-1}$ from the left of Y_m.

When a new item y_{m+1} is added to the *right* of a sequence its RTLM sequence is updated by (i) discarding all of the current RTLM values not smaller than y_{m+1} and then (ii) appending y_{m+1} to the right of the RTLM sequence.

Since the RTLM sequence is monotonically increasing this can be done either by sequentially scanning the RTLM sequence from right to left, discarding all items not smaller than y_{m+1} until an item smaller than y_{m+1} is found, or by using a binary search to find the first item in the RTLM sequence smaller than y_{m+1} and then chopping off everything in the RTLM sequence after it. Once an item is discarded from the RTLM sequence it never returns, so sequentially discarding uses $O(1)$ amortized time per update (but can be arbitrarily bad in the worst case). The binary search method requires $O(\log m)$ worst case time. We can therefore alternate steps between the two methods (as described in the k-median algorithm of the previous section) to get $O(1)$ amortized time and $O(\log m)$ worse case time simultaneously.

Deleting items $y_{l_m}, \ldots, y_{l_{m+1}-1}$ from the left of Y_m is even easier. All that needs to be done is to find the first index in the RTLM sequence which is not smaller than $y_{l_{m+1}}$ and chop off everything to the left of this index. Again, this can be done in $O(1)$ amortized time per update using a sequential scan from the left or an

$O(\log m)$ worst case time binary search. Combining the two gives $O(1)$ amortized time and $O(\log m)$ worse case time simultaneously.

We have just seen that we can update the RTLM sequence of Y_m to the RTLM sequence of Y_{m+1} in $O(1)$ amortized time and $O(\log m)$ worse case time simultaneously. Once we have done this we can calculate the $OPT_i(m+1)$ values in $O(1)$ time. We need to do this for each i, $1 \le i \le m$ so the entire update operation uses $O(k)$ amortized time and $O(k \log m)$ worse case time simultaneously.

As we mentioned before, the values of $POPT_i(m)$ can be calculated similarly in $O(k)$ amortized time and $O(k \log m)$ worse case time per update. This finishes the proof of Theorem 2.

3.1 A k-Coverage Example

We show an example to illustrate the k-coverage algorithm. $n = 9$ is the total number of nodes, and $k = 3$ is the maximum number of resources. The x-coordinates of the 9 nodes are 2, 4, 49, 64, 74, 87, 90, 94, 99. The set-up costs c_j of the nodes are 29, 68, 59, 7, 88, 49, 89, 76, 66. The weights w_j are 97, 17, 14, 76, 31, 46, 34, 1, 33. And the radius r is 20.

Tables 5 to 8 show the values of OPT, MIN, $POPT$ and $PMIN$, respectively. From these tables, we can see that the optimal placement when $m = 9$ is to place 3 resources at x_1, x_4 and x_6.

Table 9 shows the changes of the RTLM sequence for $POPT_3(j)$ as m increases. For example, the RTLM sequence is $(85, 112)$ when $m = 8$, and it changes to $(85, 102)$ when x_9 was added.

4 Conclusion and Open Problems

In this paper we discussed how to solve the online k-median on a line problem in $O(k)$ amortized time and $O(k \log n)$ worst case time per point addition. This algorithm maintains in the online model the dynamic programming speed-up for the problem that was first demonstrated for the static version of the problem in [6]. The technique used is a generalization of one introduced in [3]. We also showed how a simpler form of our approach can solve the online k-coverage on a line problem with uniform coverage radius in the same time bounds. It is not clear how to extend our ideas to the non-uniform coverage radius case.

A major open question is how to solve the *dynamic* k-median and k-coverage on a line problem. That is, points will now be allowed to be inserted (or deleted!) anywhere on the line and not just on the right hand side. In this case would it be possible to maintain the k-medians or k-covers any quicker than recalculating them from scratch each time?

We would also like to propose a simpler extension of the problem, the *two-sided online k-median (and k-coverage) problem.* In this extension, nodes can be added both to the left and right of the existing nodes, not just to the right. While initially this might sound like an easy extension there are reasons for believing that it will be much more complex than the one-sided online problem studied in this paper. Essentially, the problem studied in this paper was to fill in the $O(kn)$ sized dynamic programming table given by Lemma 1. Adding new points to the right of the line *added $O(k)$ new entries to the table but did not change any of the old entries.* This dynamic program is known in advance to possess special properties, i.e., the quadrangle inequality/concavity, that permits solving it quickly, e.g., [6]. What we did in this paper was find a way to maintain this dynamic programming speed-up while calculating the $O(k)$ new values.

		m								
		1	2	3	4	5	6	7	8	9
i	1	29	29	43	119	121	167	201	202	235
	2	-	97	88	36	36	82	116	117	150
	3	-	-	156	95	95	85	85	85	85

Table 5: The values of $OPT_i(m)$.

		m								
		1	2	3	4	5	6	7	8	9
i	1	1	1	1	1	4	4	4	4	4
	2	-	1	3	4	4	4	4	4	4
	3	-	-	3	4	4	6	6	6	6

Table 6: The values of $MIN_i(m)$.

		m								
		1	2	3	4	5	6	7	8	9
i	1	29	68	173	121	216	253	293	280	301
	2	-	97	88	36	131	168	208	195	187
	3	-	-	156	95	124	85	125	112	102

Table 7: The values of $POPT_i(m)$.

		m								
		1	2	3	4	5	6	7	8	9
i	1	-	-	-	-	-	-	-	-	-
	2	-	1	2	2	3	4	4	4	5
	3	-	-	2	3	4	4	4	4	5

Table 8: The values of $PMIN_i(m)$.

j		1	2	3	4	5	6	7	8	9
$POPT_3(j)$		-	-	156	95	124	85	125	112	102
m	cov_m									
3	3			156						
4	3				95					
5	4				95	124				
6	5						85			
7	5						85	125		
8	5						85		112	
9	6						85			102

Table 9: The changes of the RTLM sequences for $POPT_3(j)$ as nodes are added. The shaded regions are the intervals $[cov_m, m]$, where the RTLM sequences is considered. The values of $POPT_3(j)$ are shown if it is a right-to-left minimum in the RTLM sequence.

Being able to add points to both sides of the line could totally change *all* of the $\Theta(kn)$ entries in the table. A dynamic programming approach would therefore require updating all $\Theta(kn)$ entries, requiring $\Theta(kn)$ time. Since we can solve the static problem in $O(kn)$ time it therefore appears that we could not use a Dynamic Programming approach for efficiently updating the two-sided online k-median problem and would therefore have to find a totally different technique.

Acknowledgments

We would like to thank Gerhard Trippen for his help in proofreading and latexing the figures.

References

[1] A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber. "Geometric applications of a matrix-searching algorithm," *Algorithmica,* **2**(2), 1987, pp. 195-208.

[2] Sanjeev Arora, Prabhakar Raghavan, and Satish Rao. "Approximation schemes for Euclidean k-medians and related problems," *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC'98)*, 1998, pp. 106-113.

[3] V. Auletta, D. Parente, and G. Persiano. "Placing resources on a growing line," *Journal of Algorithms*, **26**, 1998, pp. 87-100.

[4] M. Charikar, S. Guha, E. Tardos, and D.B. Shmoys. "A constant-factor approximation algorithm for the k-median problem," *Journal Computer System Sciences*, **65**, 2002, pp. 129-149.

[5] S. Guha and S. Khuller. "Greedy strikes back: improved facility location algorithms," *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA'98)*, 1998, pp. 649-657.

[6] R. Hassin and A. Tamir. "Improved complexity bounds for location problems on the real line." *Operations Research Letters* **10**, 1991, pp. 395-402.

[7] J-H. Lin and J.S. Vitter. "ϵ-approximations with minimum packing constraint violation," *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing (STOC'92)*, 1992, pp. 771-782.

[8] A. Tamir. "An $O(pn^2)$ algorithm for the p-median and related problems on tree graphs," *Operations Research Letters*, **19** 1996, pp. 59-64.

[9] R. Wilber. "The concave least-weight subsequence problem revisited," *Journal of Algorithms,* **9**, 1988, pp. 418-425.

[10] A. Vigneron, L. Gao, M. Golin, G. Italiano and B. Li. "An algorithm for finding a k-median in a directed tree," *Information Processing Letters,* **74**, 2000, pp. 81-88.

[11] G. Woeginger. "Monge strikes again: optimal placement of web proxies in the internet." *Operations Research Letters,* **27**, 2000, pp. 93-96.