1 The $k$-server Problem

The $k$-server problem is one of the most classic problems in the area of online algorithms — it was introduced in 1988 and is still being studied thirty years later!

In the $k$-server problem, we assume that $k$ servers are located at some arbitrary starting points in a metric space.\(^1\) This part of the input (the metric space and $k$) are known in advance (before the online input sequence starts). The online input is a sequence of requests, where each request is located at some point of the metric space (in the case of the line, each request is some $x_i \in \mathbb{R}$). Every time a request arrives, the algorithm must move a server to the location of the request, and the cost for this move is the distance between the location of the server (before the move) and the location of the request. (We will see an example soon when discussing the greedy algorithm.)

In the previous lecture, we discussed online problems where the goal was to maximize the objective function, and defined the competitive ratio of an algorithm for this type of problems. Here the goal is to minimize the cost incurred by the algorithm (the total distance traveled by the servers), so we need to define the competitive ratio for minimization problem.

**Definition 1.1.** An algorithm $A$ (for an online problem with a minimization objective) is $\alpha$-competitive if for every input $I$, $A(I) \leq \alpha \cdot OPT(I) + \beta$, where $\beta$ is a constant independent of the online input.

The constant $\beta$ can depend on the metric space, the initial location of the servers, and $k$, but may not depend on the online sequence of inputs.

1.1 The Competitive Ratio of Greedy Is Unbounded

One of the most natural algorithms to consider for the $k$-server problem is the greedy algorithm: whenever a request arrives, move the server that is closest to the request. This algorithm, however, turns out to have an unbounded competitive ratio, even when $k = 2$.

The example that shows that for $k = 2$ appears in Figure 1. It uses requests at three points: $a$, $b$, and $c$, where $a$ and $b$ are close to each other, and $c$ is far apart from both. Assume the two servers are originally located at any of these three points ($a$, $b$, $c$). The first request arrives at $c$, so the algorithm must move a server to that location. The subsequent requests alternate between $a$ and $b$, and since $a$ and $b$ are closer to each other than, greedy moves one server between $a$ and $b$.

\(^1\)If you are not familiar with metric spaces, they are sets of points with a distance function that satisfies the triangle inequality. For the purpose of this lecture, you can assume that the metric space is just the real line $\mathbb{R}$, and where the distance between any two points $x, y \in \mathbb{R}$ is $|x - y|$.\]
(while keeping the server located at $c$ in place). Then, the cost incurred by the greedy algorithm is linear in the number of requests (since each time a request arrives at $a$ or $b$, greedy pays the distance between them), which can be arbitrarily high.

On the other hand, when the number of requests is high, the optimal algorithm will move one server to $a$ and the other to $b$ (after serving the first request at $c$), and pay a constant. Hence, we get that the competitive ratio of greedy is unbounded (for a large enough number of requests).

In this lecture, we will see a $k$-competitive algorithm for $k$-servers on the real line, and a lower bound of $k$ on the competitive ratio of any deterministic algorithm for $k$-servers on any metric space with at least $k+1$ points.

2 The Double Coverage Algorithm

In this section, we consider the special case of $k$-server where the metric space is the real line $\mathbb{R}$ (with the usual distance $|x - y|$ between any $x, y \in \mathbb{R}$). The double coverage algorithm (sometimes called double cover) is presented as Algorithm 1.

**Algorithm 1** The double coverage algorithm for $k$-server on the line

When a request arrives at $x$:

1. If all the servers are currently to the left (or the right) of $x$, move the closest server to $x$.

2. Otherwise, find the closest server to $x$ from each side, and move them both toward $x$ until one of them reaches $x$. (Formally, if the server are located at $x_1 \leq x_2 \leq \cdots \leq x_k$, find $i$ such that $x_i \leq x \leq x_{i+1}$, and move the servers at $x_i, x_{i+1}$ the distance $\min\{x_{i+1} - x, x - x_i\}$ toward $x$.)

Note that every time the algorithm moves two servers, moving one of them is unnecessary and may appear to be a “wasted” cost. It is also possible to implement a lazy version of the algorithm where only one server moves each time: For each server, we also keep track of a virtual location, which is where that server would have been if it had followed the double coverage as stated above (moving two servers each time). When a new request arrives, we use the virtual locations to decide which two servers might need to move, but then only move one of them (the one that serves the request).

We now show that the double coverage is $k$-competitive. The analysis will use a potential function, which intuitively represents the difference in configuration between our algorithm and the
optimal offline algorithm. The idea is that while we cannot prove that for each request, the cost incurred by the algorithm is at most \( k \) times the cost incurred by OPT, we may able to charge the extra cost to the potential function. By keeping track of the changes in the potential function, we will be able to compare the total costs of the algorithm and OPT on the requests up until now.

This general technique is very powerful. It is used in the analysis of many algorithms, and in particular, it can be used for *amortized analysis*. Many times, especially in data structures, we cannot get a bound on the cost of each operation in the worst case, so we aim to bound the cost of each operation on average (over a sequence of operations). A concrete example is the implementation of dynamic arrays (such as `std::vector` in C++ or lists in Python). These arrays have variable size, and we can always append a new element to the array (therefore increasing its size). The first implementation one might come up is the following: When the array contains \( n \) elements, the memory allocated is exactly the memory needed to store \( n \) elements. Then, every time a new element is added to the array, we will need to reallocate the memory. This can take \( O(n) \) time per insertion, since we need to copy the array to the newly allocated space. Another possible implementation may sometimes allocate more space than the array currently needs: Suppose we have \( n \) elements in the array, and the allocated space is all used up. When an element is inserted, instead of allocating space for \( n + 1 \) elements, allocate space for \( 2n \) elements. This reallocation will take \( O(n) \) time, but each of the subsequent \( n - 1 \) insertions will take \( O(1) \) time. Hence, the amortized run time of insertion is \( O(1) \) (the run time of insertion on average over the \( n \) operations).

This was a brief high-level discussion to introduce the idea of amortized analysis — there are many more examples.

We are now ready to analyze the competitive ratio of the double coverage algorithm.

**Theorem 2.1.** The double coverage (DC) algorithm is \( k \)-competitive.

**Proof.** Consider any input sequence \( \sigma \) and fix some optimal algorithm OPT for the input \( \sigma \). At time \( t \) (that is, after the first \( t \) requests have been served), denote the ordered locations of the servers when running DC by \( x^t = (x^t_1, \ldots, x^t_k) \) (such that \( x^t_1 \leq \cdots \leq x^t_k \)). Similarly, denote the ordered locations of the servers when running OPT by \( y^t = (y^t_1, \ldots, y^t_k) \) (such that \( y^t_1 \leq \cdots \leq y^t_k \)). Given two locations vectors, \( x \) for DC and \( x \) for OPT, we define the potential \( \phi(x, y) \) as

\[
\phi(x, y) = \phi_1(x, y) + \phi_2(x, y),
\]

where

\[
\phi_1(x, y) = k \sum_{i=1}^{k} |x_i - y_i|
\]

\[
\phi_2(x, y) = \sum_{i<j} |x_i - y_j|.
\]

For simplicity, assume that in the beginning all the servers are located at the same point (otherwise, we pay an extra additive constant), so \( \phi(x^0, y^0) = 0 \). Our goal is to prove that after seeing the entire input sequence \( \sigma \),

\[
DC(\sigma) + \left( \phi(x_{\text{final}}, y_{\text{final}}) - \phi(x^0, y^0) \right) \leq k \cdot OPT(\sigma). \tag{2.1}
\]

Since \( \phi \) is nonnegative and \( \phi(x^0, y^0) = 0 \), this will prove that DC is \( k \)-competitive. We prove this inequality by considering by how much each side changes after serving a request. In particular, let
\( \Delta DC(\sigma) \) and \( \Delta OPT(\sigma) \) denote the cost incurred by DC and OPT (respectively) for serving the request at time \( t \), and let \( \Delta \phi = \phi(x^t, y^t) - \phi(x^{t-1}, y^{t-1}) \). The inequality we are going to prove is

\[
\Delta DC(\sigma) + \Delta \phi \leq k \cdot \Delta OPT(\sigma).
\] (2.2)

Summing this inequality over all \( t \) implies (2.1), and finishes the analysis. We prove (2.2) by analyzing the handling of request \( t \) in two stages: (i) OPT moves, and (ii) DC moves.

Consider first the stage where OPT moves. The change in each summand is:

- **\( \Delta OPT(\sigma) \):** The way to move from \( y^{t-1} \) to \( y^t \) that minimizes the distance traveled by the servers is to move the server located at \( y_i^{t-1} \) to \( y_i^t \) (left as an exercise). Then \( \Delta OPT(\sigma) = \sum_{i=1}^k |y_i^t - y_i^{t-1}| \).

- **\( \phi_1 \):** We need to bound how much the distance between the positions of servers in DC and OPT changed (formally, we need to bound \( k \sum_{i=1}^k |x_i^{t-1} - y_i^t| - k \sum_{i=1}^k |x_i^{t-1} - y_i^{t-1}| \). By the triangle inequality, \( |x_i^{t-1} - y_i^t| \leq |x_i^{t-1} - y_i^{t-1}| + |y_i^{t-1} - y_i^t| \). So, the difference in \( \phi_1 \) is at most \( k \sum_{i=1}^k |y_i^t - y_i^{t-1}| = k \Delta OPT(\sigma) \).

- **\( \phi_2 \):** Note that when OPT moves, \( \phi_2 \) remains the same as it depends only on the positions of the servers of DC.

- Since only OPT moves, no cost is incurred by DC.

Combining all these, we get that after OPT moves, (2.2) is maintained. (Formally, we proved \( \phi(x^{t-1}, y^t) - \phi(x^{t-1}, y^{t-1}) \leq k \cdot \Delta OPT(\sigma) \).

Now, consider the stage where DC moves. Since OPT does not move, we only need to consider the change in potential and show that it is at most \( \Delta DC(\sigma) \). There are two cases:

1. **DC moves just one server.** Let \( d \) denote the cost incurred by DC. Assume the request is to the left of all the servers (the other case is analogous). Then the first server (at \( x_1^{t-1} \)) moves to \( x_1^t = x_1^{t-1} - d \). Note that at time \( t \), OPT has already moved one server to the location of the request, \( x_1^t \). So there is some \( i \) such that \( y_i^t = x_1^t \). Moreover, the location of the leftmost server of OPT satisfies \( y_1^t \leq y_i^t = x_1^t \), so DC moved the leftmost server closer to \( y_1^t \), with the distance by \( d \). As a result, \( \phi_1 \) changed by \(-kd\). On the other hand, the leftmost server of DC moved away from the \( k - 1 \) other servers (a distance of \( d \)), so \( \phi_2 \) changed by \((k - 1)d\). Overall, the change in \( \phi \) was \(-kd + (k - 1)d = -d\), and we get \( \Delta DC(\sigma) + \phi(x^t, y^t) - \phi(x^{t-1}, y^t) = 0 \).

2. **DC moves two servers, at indices \( i \) and \( i + 1 \).** Assume that each of the two servers moves a distance of \( d \). Then \( \Delta DC(\sigma) = 2d \). Note that the change in \( \phi_2 \) is \(-2d\); The servers \( i \) and \( i + 1 \) became closer by \( 2d \). The changes in all the other distances sum up to 0. This is due to the fact that only distances involving server \( i \) or \( i + 1 \) could change, and for any other server, one of \( i, i + 1 \) got closer (by \( d \)) and the other moved farther away (by \( d \)).

To show that the change in potential is at most \( \Delta DC(\sigma) \), it is left to show that the change in \( \phi_1 \) is nonpositive. We need to consider the change in distance between the \( i \)-th servers of

---

2Similarly, \( \Delta DC(\sigma) \) is the cost of moving from \( x^{t-1} \) to \( x^t \) and \( \Delta OPT(\sigma) \) is the cost of moving from \( y^{t-1} \) to \( y^t \).

3More formally, this means that we first consider moving from \( y^{t-1} \) to \( y^t \) (showing \( \phi(x^{t-1}, y^t) - \phi(x^{t-1}, y^{t-1}) \leq k \cdot \Delta OPT(\sigma) \)), and then moving from \( x^{t-1} \) to \( x^t \) (showing \( \Delta DC(\sigma) + \phi(x^t, y^t) - \phi(x^{t-1}, y^t) \leq 0 \)). Summing up these two inequalities results in (2.2).
DC and OPT, and the \((i + 1)\)-th servers of DC and OPT. Assume that server \(i\) is the server that reached the request at time \(t\), that is, \(x^t_i\) is the location of the request (the other case, where server \(i + 1\) served the request is analogous). There are two cases. If the \(i\)-th server of OPT is at \(x^t_i\) (or to the right), \(y^t_i \geq x^t_i\), server \(i\) of DC got closer to \(y^t_i\) by \(d\), and the change in the distance of the servers indexed \(i + 1\) could decrease by at most \(d\), so overall \(\phi_1\) could only decrease. The second case is that \(y^t_i < x^t_i\). Since server \(i\) of DC served the request at time \(t\), and some server of OPT also served the request at time \(t\), we know there must be some server of OPT at \(x^t_i\). Since we are considering servers in order, and we know that server \(i\) of OPT is to left of the request, server \(i + 1\) of OPT must be either at the request or to the left of it (that is, \(y^t_{i+1} \leq x^t_i\)). Therefore, we know that server \(i + 1\) of DC moved distance \(d\) closer to server \(i + 1\) of OPT, and since the distance between the servers at index \(i\) could increase by at most \(d\), the overall change in \(\phi_1\) is nonnegative.

In both cases, we have shown that \((2.2)\) is maintained (or formally, \(\Delta DC(\sigma) + \phi(x^t, y^t) - \phi(x^{t-1}, y^t) \leq 0\)). This completes the proof of the theorem.

### 3 Lower Bound for Deterministic Algorithms

We conclude the lecture with a lower bound on the competitive ratio of any deterministic algorithm for \(k\)-servers. The lower bound applies for \(k\)-servers on any metric space with \(k + 1\) points or more.

**Theorem 3.1.** In any metric space with at least \(k + 1\) points, the competitive ratio of any deterministic algorithm for \(k\)-server is at least \(k\).

**Proof.** Fix a deterministic algorithm \(A\). Pick any \(k + 1\) points in the metric space. The input sequence \(\sigma\) will always place a request on one of these \(k + 1\) points, and specifically, on the point where \(A\) does not have a server. For convenience, denote the \(k + 1\) points by \(x_1, \ldots, x_{k+1}\), and assume that in the beginning, each of the \(k\) servers is located at each of \(x_1, \ldots, x_k\). Then the first request is going to be at \(x_{k+1}\), and each subsequent request will be at \(x_i\) where \(A\) does not have a server.

Instead of comparing to one specific optimal algorithm, we describe \(k\) algorithms \(A_1, \ldots, A_k\), and relate the cost of \(A\) to the sum of costs incurred by these \(k\) algorithms. Then we will show the cost of one of \(A_1, \ldots, A_k\) is at most \(\frac{1}{k}\) the cost of \(A\) (up to a constant), which will in turn establish the lower bound.

Note that since we have \(k + 1\) points and \(k\) servers, each algorithm has servers at all of the points except one. Informally, we say that the algorithm has a “vacancy” at that point. The algorithms \(A_1, \ldots, A_k\) together with \(A\) will maintain the invariant that at any time, each of these \(k + 1\) algorithms will not place a server on a different point, that is, each algorithm will have a “vacancy” at a different point. To maintain that invariant, the algorithms \(A_1, \ldots, A_k\) are defined as follows:

1. Before the first request arrives, algorithm \(A_i\) moves a server from \(x_i\) to \(x_{k+1}\). That guarantees that the invariant is satisfied before the requests start arriving.

2. When a request arrives at some \(x_i\) (which is where \(A\) has a “vacancy”), assume that \(A\) moves a server from \(x_j\) to \(x_i\). Note that all of \(A_1, \ldots, A_k\) have a server at \(x_i\), so they can serve the request. After it serves the request, the server that has a “vacancy” at \(x_j\) moves a server from \(x_i\) to \(x_j\). The invariant is then maintained.
We now compare the costs of $A$ and $A_1, \ldots, A_k$. When the request at time $t$ arrives, $A$ must move a server. Denote the cost incurred by $A$ for serving the request at time $t$ by $d_t$. Note that only one of the benchmark algorithms $A_1, \ldots, A_k$ moves a server at time $t$. In particular, if $A$ moves a server from $x_j$ to $x_i$ and pays $d_t$, the benchmark algorithm moves one server from $x_i$ to $x_j$, and therefore also pays $d_t$. Overall, $A$ pays $d_t$ for the request at time $t$, and algorithms $A_1, \ldots, A_k$ together pay $d_t$ for the same request.

Summing over all the requests, the total cost of $A$ is

$$A(\sigma) = \sum_t d_t.$$ 

Assume that the total cost incurred by $A_1, \ldots, A_k$ for the first step (moving one server to $x_{k+1}$) is some constant $B$. So the total cost of $A_1, \ldots, A_k$ is

$$\sum_{i=1}^k A_i(\sigma) = B + \sum_t d_t = B + A(\sigma).$$ 

Hence, there must be some $i$ such that

$$A_i \leq \frac{1}{k} A(\sigma) + \frac{B}{k}.$$ 

The cost of $A$ is at least $k$ times the cost of $A_i$ (up to the constant $B$), so the competitive ratio of $A$ is no better than $k$. \hfill \square