1 Introduction

Assume you finished this course and hopefully learned some skills with respect to online algorithm. You want to purse your career in industry, and you are having an interview. The interviewer sees your resume and asks you what online algorithms are. Since it is likely that he/she does not know much about these algorithms, you should give an example which resonates for them. For that purpose, you should talk about the paging problem, which is very practical and every computer-scientist has learned about it in their operating system class. Then you can continue talking about k-server problem, which is an extension of paging and very important for theoreticians. These problems lie in the heart of online algorithms, and these is still many open problems with respect to them, some of which we review in these notes.

2 Caching problem

Paging is a classic online problem which has received a lot of attention since the beginning of online algorithms. This problem roots in practical applications that were raised with the introduction of operating systems, where we need to move ‘pages’ of memory from the slow memory (e.g., hard disk) to a faster memory (e.g., RAM). In this class, we focus on the simplest version of the problem, where pages have uniform length 1 and the slow memory has space for $k$ pages. We often refer to this easy version as ‘caching’ problem; here the fast memory is the CPU cache and the slow memory is RAM. The problem can be defined as follows:

**Definition 1.** In the caching problem, we are given a cache of size $k (\geq 2)$ and a slow memory of unbounded size. The cache is initially empty. The input is an online requests $\langle x_1, x_2, \ldots, x_n \rangle$ to pages. For each request $x_i$, page $x_i$ should be present in the cache. In case it is already in the cache, we say a **hit** occurs. In case the requested page is not in the cache, a **fault** occurs, and the algorithm should bring the page to the cache at a **cost** of 1. In doing so, if the cache is full, it is required to **evict** one of the pages in the cache to make space for the requested page. The goal of an online algorithm is to evict pages in a way that the total cost, i.e., total number of page-faults, is minimized.
2.1 Caching Algorithms

In order to manage the cache, it suffices to define an eviction policy. This is because if there is a hit, we can avoid/postpone updating the cache and ‘do-nothing’. Similarly, if there is a fault and the cache is not full, we can avoid eviction and just bring the requested page to the cache.

Let’s look at some caching algorithm. One simple eviction strategy is to evict the Least-Recently-Used page; the resulting algorithm is called **LRU** and is widely used in practice. Another algorithm is to evict the ‘oldest’ page in the cache, i.e., the page that was first brought to the cache. The resulting algorithm works based on First-In-First-Out rule and is called **FIFO**. One other interesting algorithm ‘flashes’ the cache in case an eviction is necessary, i.e., it evicts all pages in the cache. As we will see later, the resulting algorithm, called Flash-when-Full (**FWF**), can be implemented in a more rational way where it does evict all pages at once, but instead evicts only one and labels the rest as deleted.

Caching is an inherently online problem, i.e., its offline variant has no practical significance. But, in order to analyse and compare online algorithms under competitive analysis, it is useful to think about offline algorithms. Consider an eviction strategy that evicts the page whose next request is furthest-in-future among all pages in the cache. The resulting algorithm, called **FIF**, is offline in the sense it inquires at which point in the future each page in the cache is requested again.

**Theorem 1.** **FIF** is the optimal offline algorithm for caching.

We skip the details of the proof for the above theorem and just provide a big picture. The idea behind the proof is recurrent in many similar proofs where we want to show an algorithm **X** is the optimal algorithm for a given problem. We start with an arbitrary algorithm **Y** and convert it to **X** (here **FIF**) in a sequence of ‘steps’ which involve intermediate algorithms \( Y = Y_0 \rightarrow Y_1 \rightarrow \ldots \rightarrow Y_m = X \). Each algorithm differs from the previous one in only one step. For the first step, we consider the first time that **X** and **Y** behave differently. We change actions of **Y** to match that of **X** in the first step, and in doing so, we ensure that the cost of **Y** is not increased; the result is algorithm **Y_1** which behaves similarly to **Y = Y_0** except for one step. We repeat this subsequently to convert **Y_i** to **Y_{i+1}** until **Y_m** becomes equivalent to **X**. In the case of Theorem 1, the conversion from **Y_i** to **Y_{i+1}** requires a rather involved case-analysis which we skip here. Theorem 1 implies that we know the nature of \( \text{Opt} \) (i.e., **FIF**) for caching. This is in contrast to problems such as clustering and list update, where the optimal offline algorithm was unknown (since offline problem was NP-hard).

**Example:** Consider an input sequence \( \sigma = \langle a, b, c, a, b, d, a, c, a, b, \ldots \rangle \). On the request to **d**, the above algorithms act like this:

<table>
<thead>
<tr>
<th></th>
<th>LRU:</th>
<th>FIFO:</th>
<th>FWF:</th>
<th>FIF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td>a b c a b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a b c d</td>
<td>a b c d</td>
<td>a b c d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a b c a b</td>
<td>a b c d</td>
<td>a b c a b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a b c d</td>
<td>a b c a b</td>
<td></td>
</tr>
</tbody>
</table>

2.2 Lower bound for competitive ratio

As before, we use competitive analysis to analyze caching algorithms. First, we show that no deterministic online algorithm can have a competitive ratio better than \( k \):

**Theorem 2.** For a cache of size \( k \), no deterministic caching algorithm can have a competitive ratio better than \( k \).
Proof. Consider an arbitrary online algorithm $A$. We create an adversarial sequence $\sigma$ of length $n$ on $k+1$ pages so that $A$ has a fault on every single request. It is possible to create such sequence since there are a total of $k+1$ pages and at least one of them is outside of the cache at each given time. So, the cost of $A$ for $\sigma$ will be $n$.

Next, we consider the cost of $Opt$. Recall that by Theorem 1, $Opt$ is equivalent to FIF. We claim that if FIF has a fault at one request, it will have hits in the following $k-1$ requests. Assume FIF evicts page $x$ on a fault for a request to $z$; so all $k+1$ pages except $x$ are in the cache after serving the request. In other words, there will be no fault until the next request to $x$. But, since $x$ was the furthest in the future when evicted, we know all $k-1$ pages (all pages in the cache except potentially $z$) have been request at least once before the next request to $x$. It means that before the next fault on the request to $x$, there will be at least $k-1$ hits. In other words, in FIF, for each fault, there are at least $k-1$ hits. It means if there are a total of $c$ faults by FIF for serving $\sigma$, there will be $d \geq c(k-1)$ hits; let $d = \alpha c$ for some $\alpha \geq k-1$. We know the total number of requests is $c + d = n$, i.e., $c + \alpha c = n$, i.e. $c = \frac{n}{1+\alpha}$. Since $\alpha \geq k-1$, we conclude the cost of FIF (and hence $Opt$) is at most $\frac{n}{k}$ for $\sigma$.

To summarize, for any algorithm we can create a sequence for which the cost of $\sigma$ is $n$ and the cost of $Opt$ (FIF) is at most $\frac{n}{k}$. We conclude that the competitive ratio of any algorithm is at least $k$.

The above theorem implies that no deterministic algorithm can be better than $k$-competitive. In other words, no algorithm is 'competitive' in the sense that the competitive ratio depends of the instance of the problem and is not always a constant. However, a competitive ratio of $k$ is much better than a ratio that depends on $n$. This is because $k$ is a parameter of the problem and somehow fixed as opposed to $n$ which is the length of the input and set by adversary (and can be arbitrary large).

2.3 Competitiveness of LRU & marking algorithms

What is the competitive ratio of LRU? Given a fixed input, let’s ‘partition’ it into phases so that each phase includes exactly $k$ pages and the page requested right after the end of a phase is not among the $k$ pages that form that phase. It is possible to achieve such partitioning by adding pages, one by one, to an initially empty phase and in case the phase includes $k$ pages and the next page $x$ is not among those $k$ pages that form the phase, create a new phase and add the $x$ to it. Here is an example of phase partitioning of a sequence when $k = 4$:

$$
\sigma = a \ b \ c \ b \ a \ d \ c \ \underline{e} \ f \ a \ c \ d \ e \ f \ a \ b \ f \ b \ d \ a \ f \ b \ f \ d \ a \ a \ f \ a \ f \ b \ f \ c \ b \ a \ e \ \ldots
$$

We use this partitioning to prove upper bound for competitive ratio of many caching algorithms.

**Theorem 3.** Competitive ratio of LRU is at most $k$.

**Proof.** We compare the cost of LRU and $Opt$ on each phase. There are exactly $k$ different pages requested in each phase. LRU might incur a cost of 1 to bring each of these pages to the cache on the first request to the page. After the first request, LRU never evicts that page until the end of the phase because that page will be among the $k$ most-recently used pages in the cache. So, the cost of LRU is at most $k$ for each phase. So, if there are $m$ phases in an input sequence, the cost of LRU will be at most $mk$.  

3
On the other hand, OPT incurs a cost of $k$ for the first phase (none of them are in the cache at the beginning). For subsequent phases, pages requested in each phase plus the starting page in the next phase form $k + 1$ distinct pages. Any algorithm, in particular OPT, incurs a cost of at least one for the requests to these $k + 1$ pages (because not all of these $k + 1$ pages fit in the cache at the same time). So, for the requests in each phase $i > 1$ and the first phase of phase $i + 1$ make OPT pay a cost of at least 1. So, there are $m - 2$ phases (excluding the first and the last phase) which form sequences of $k + 1$ distinct pages that make OPT pay a cost of at least 1. To summarize, the cost of OPT will be at least $k + m - 2 \geq m$ (note that $k \geq 2$).

So, for any input, we showed that the cost of LRU is at most $mk$ and the cost of OPT is at least $m$. The competitive ratio of LRU is therefore at most $k$.

A careful look at the above proof shows that any algorithm that shares the following property with LRU also has a competitive ratio of at most $k$. The property is that the algorithm should occur at most $k$ faults for any formed by $k$ distinct requests. This leads us to definition of a broad family of caching algorithms named marking algorithms.

A marking algorithm maintains a bit (‘mark’) for each page in the cache. Upon a fault, if eviction is required, an unmarked page is evicted (different marking algorithms might chose different unmarked pages to evict). If all pages in the cache are marked, all of them are unmarked first and any of them is evicted. After a page is brought to the cache, it will be marked. Here is an example of a marking algorithm:

```
  a b c a b  →  a✓ b✓ c✓ [ ] d  
  d✓ b  c  →  d✓ b [ ] c✓  
  d✓ a✓ c✓  
```

**Theorem 4.** All marking algorithm have competitive ratio of at most $k$.

**Proof.** The proof is similar to that of Theorem 3. Here, we just show that the cost of any marking algorithm is at most $k$ for each phase. Other aspects of the proof are the same as before.

We inductively maintain an ingredient that at the beginning of a phase $p$ all pages in the cache are marked, and the first page of $p$ is not in the cache. For the base of induction, at the end of the first phase $k$ pages are brought to the cache; all are marked, and the next phase is not in the cache. Now, for the phase $p$, the first request results in unmarking all pages in the cache. The first request to any page $x$ in $p$ might results in eviction of some other page. Regardless, after the first request, $x$ becomes marked and stays in the cache until the end of the phase. Subsequent evictions in the phase do not result in eviction of $x$ because it is marked, and before its eviction as a marked page, all pages should become marked. This only happens at the end of the phase, where all $k$ pages are requested (marked) and the page in the next phase is not in the cache. Note that at the end, the ingredient maintained by induction is preserved. We conclude that any marking algorithm incurs a cost of at most 1 for each page in each phase (at the very first request to the page). As a consequence, the cost of the marking algorithm for each phase is at most $k$.

As a result of the above theorem, in order to prove an algorithm is $k$-competitive, it suffices to show that it is a marking algorithm. In fact, algorithms like LRU and FWF are marking and hence have competitive ratio $k$. FIFO is not marking but yet has a competitive ratio of $k$. In what follows, we see an example of a proof for showing that an algorithms is marking:

**Theorem 5.** LRU is a marking algorithm.

**Proof.** Assume LRU does not behave like a marking algorithm for an input $\sigma$. Consider a phase-partitioning of $\sigma$ and consider the first time that LRU evicts a marked page $x$ at some phase for a
request to \( y \). Since \( x \) is marked at the time \( y \) is requested, both \( x \) and \( y \) belong to the same phase. Since LRU evicts \( x \), page \( x \) should be the least recently used page when \( y \) is requested, i.e., there should be \( k - 1 \) pages requested after \( x \) and before \( y \). Adding \( x \) and \( y \), there will be \( k + 1 \) pages in the phase which contradicts the definition of a phase being formed by \( k \) pages.

2.4 A brief look at randomized paging

The results in the previous section clarify the situation for deterministic paging problem: no deterministic algorithm can achieve a competitive ratio better than \( k \), and there are deterministic algorithms that indeed achieve this ratio (e.g., all marking algorithms like LRU and FWF). In this section, we briefly review the existing results for randomized paging algorithms.

First, let’s consider a simple randomized algorithm that, upon an eviction, randomly evicts any page from the cache. This algorithm is known to have a competitive ratio of \( k \), which is no better than what deterministic algorithms such as LRU achieve. Can we do better? The answer is Yes!

We learned earlier that marking algorithms all have a competitive ratio of \( k \), i.e., they all achieve the best competitive ratio that a deterministic algorithm can achieve. When we have a situation like this, where a family of algorithms all perform well, it is natural to devise a randomized algorithm that randomly selects between these strategies. MARK is a randomized algorithm that works based on the above intuition. As the name suggests, MARK is a marking algorithm. When an eviction is required, MARK evicts an unmarked page like any other marking algorithm. However, in taking its decision, it randomly selects the unmarked page to be evicted. This improves the competitive ratio, as summarized by the following theorem:

**Theorem 6.** MARK has a competitive ratio of \( 2H_k \) where \( H_k \) is the \( k \)'th harmonic number, i.e.,

\[
H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}.
\]

We skip the proof of the above theorem. Note that for large values of \( k \), we have \( \ln(k) \leq H_k \leq \ln(k) + 1 \). That implies that, unlike deterministic algorithm where the competitive ratio was linear to \( k \), for randomized algorithms the ratio can be reduced to be logarithmic to \( k \). In fact, it is known that no randomized algorithm can be better than \( H_k \), which implies that MARK is an optimal randomized algorithm for caching (ignoring constant factors).

2.5 A note on Belady’s anomaly

For online problems that involve a sort of ‘resource’, such as paging which involves a cache, we often face with algorithms that suffer from a sort of anomaly. Such anomaly for an algorithm implies that if we increase the amount of resources, unlike what is expected, the algorithm performs worse for some sequences. This phenomenon was observed by Belady in the context of First Fit algorithm for paging. Consider sequence \( \langle a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \rangle \). For a cache of size \( k = 3 \), First Fit incurs 9 faults while for a cache of size \( k = 4 \), the number of faults increases to 10. In other words, for this sequence, the cost of First Fit increases as the amount of resources (cache size) increase. This anomalous behaviour of First Fit does not happen for LRU.
2.6 Paging and advice

In this section, we briefly review the paging problem under advice setting. Recall that in this setting, an offline oracle provides some information about the input sequence in form of an advice. This information can be any thing about the input, and the online algorithm knows its meaning. Clearly, more advice results in better competitive ratios. In this section, we mostly consider online algorithms that are aimed to achieve an optimal solution.

What is the simplest advice for paging? Consider the simple scheme in which advice precisely tells what Opt evicts for each fault. So, for each request that involves a fault in Opt’s scheme, we use $\Theta(\log k)$ bits of advice to indicate the page that is evicted by Opt. Assuming there are $m \leq n$ faults by Opt, the total size of advice will be $m\Theta(\log k) = O(n \log k)$. Provided with this advice, the algorithm can emulate Opt by evicting the same pages, and its cache will be similar to that of Opt at each step. Clearly, this results in an optimal algorithm. However, the size of advice is too high in this naive solution, and in fact it can be improved by a logarithmic factor.

**Theorem 7.** There is an algorithm that receives $n$ bits of advice and achieves an optimal solution for paging.

**Proof.** Given any algorithm, in particular Opt (which is FIF), we can define the status of pages in the cache to be mortal or resident. A page $x$ is mortal if it is brought to the cache at some request $i$ and is evicted before the next request for $x$. In contrast, a page $x$ is resident if it is brought to the cache and stays in the cache until the next request to it. For example, consider FIF for the sequence $\sigma = (a b c b a d c f a c d c f a b a e)$. After the first three requests pages $a, b, c$ are in the cache and are resident. This is because they stay in the cache of FIF until the next request to each of them (e.g., the next request to $a$ is at the fifth request and there is no eviction until then by FIF). Now, the next request (the fourth one in the sequence) is to $b$, and after this request the status of $b$ changes from resident to mortal. This is because FIF evicts $b$ sometime before the next access to $b$. The above definition implies that if Opt has a hit on a request to $x$, then $x$ has been resident before the access. Similarly, if there is a fault on $x$, then either it is the first access to $x$ or $x$ has been evicted since its previous access which implies that $x$ has been a mortal page in the cache.

Consider an offline algorithm ResMor that is aware of the status of pages in the cache of Opt (i.e., it knows whether each page is mortal or resident). When an eviction is necessary, ResMor evicts any mortal page. This ensures that resident pages of Opt are always present in ResMor’s cache. (When there is a fault, Opt and ResMor might evict different mortal pages, which does not affect the resident pages). We claim that ResMor has the same cost of Opt. To see that, consider otherwise, i.e., assume Opt has smaller cost. This implies that there is a request to page $x$ that is a hit by Opt and a miss by ResMor. Note that Opt has a hit when there is request to a resident page. So, $x$ is resident page in Opt and is not present in ResMor’s cache. This contradicts our previous observation that resident pages of Opt are always present in ResMor’s cache (in other words, we ResMor has evicted a resident page of Opt at some point, which contradicts its definition).

In summary, ResMor is an optimal algorithm which only needs to know the state of each page (being resident/mortal) in the cache of Opt. To device an optimal online algorithm with advice, assume that each request is accompanied with one bit of advice that indicates whether the requested page will be resident or mortal in Opt’s cache after the request. This way, we know the state of each page in the optimal cache, and hence behave like ResMor, which is optimal. We can conclude the above discussion as follows:

- What does the advice encode? For each request to a page $x$, it encodes whether $x$ will be
resident or mortal in Opt’s cache.

- What does the advice encode? Each request is accompanied with one bit of advice to separate resident and mortal pages. The total length of advice will be $n$ bits.

- How does algorithm behave, provided with the above advice? It behaves similarly to ResMor algorithm, i.e., it evicts a mortal page whenever eviction is necessary.

- Why the algorithm is optimal? It maintains the same resident pages as Opt; so in case there is a hit by Opt there will be a hit by the algorithm as well.

Theorem 7 advice of linear size is sufficient to achieve an optimal paging algorithm. Can we do better? Indeed, we see later in the course that we cannot achieve an optimal algorithm with sub-linear advice. So, advice of size $n$ is asymptotically the best we can hope in terms of the size of advice for achieving an optimal solution. However, we can slightly improve the coefficient by giving up on competitive ratio. It is known that with roughly $\log\left(\frac{1}{r+1}\right) \cdot n$ bits of advice, one can achieve a competitive ratio of $r$. For example, with roughly $0.27n$ bits, we can achieve a competitive ratio of 2 and with $0.24n$ bits, we can achieve a competitive ratio of 3.

3 k-server problem

$k$-server problem is one of the most important online problems for theoreticians. The problem is defined on a metric which is home to $k$ servers. Recall that a metric is defined with a set of items (e.g., vertices of a connected graph or points in plane) where each pair of items have a distance (e.g., Euclidean distance in plane or shortest path in graph) so that the distance function follows the triangle inequality, i.e., for any three items $x, y, z$, we have $d(x, y) \leq d(x, z) + d(z, y)$. You can easily verify that Euclidean plan defines a metric as well as any graph with non-negative weights on its edges. In this section, we often assume our metric is an unweighted graph, i.e., distance of adjacent vertices is 1 and other distances are shortest-path distances. We define the $k$-server problem as follows:

**Definition 2.** The input to the $k$-server problem is formed by a connected graph of size $m$, and $k$ servers initially located on some vertices of the graph. A sequence of requests appear in an online manner. Each request points to a vertex in the graph. To serve a request, an online algorithm should move servers (if required) so that at least one of the servers is present at the requested node. In doing so, the goal is to minimize the total distance moved by all servers for serving all requests in the sequence.

So, our metric can be any graph. What happens if it is a complete graph? The problem becomes paging! Here, each page is associated with a vertex in the graph. Those pages that are in the cache are the vertices on which a server is located. A hit means there is a request to a vertex which has a server in it (the cost will be 0). A fault means that one server should move from a vertex (associated with an evicted page) to the requested vertex (requested page). This requires moving one unit and has a cost of 1. As a conclusion, $k$-server is a generalization of the paging problem. In fact, the initial idea behind studying this problem was to find a problem with the ‘right level of difficulty’, compared to paging which seemed to be too narrow (and easy in the sense we know the best possible deterministic and randomized paging algorithms) and the metrical-task-systems, which was too broad for further studies.
3.1 Greedy Algorithm

Let’s start with a very easy greedy algorithm which, for serving a request at vertex \( x \), moves the closest server to \( x \) (if a server is at \( x \) it does not move any server). This greedy algorithm indeed has a pretty bad competitive ratio:

**Theorem 8.** For any graph of diameter \( d \geq 3 \), the competitive ratio of greedy is at least \( \frac{n}{2d} \).

**Proof.** Consider four vertices \( A, B, C \) so that \( A \) and \( B \) are neighbors and we have \( d(A, C) > 1 \) and \( d(B, C) > 1 \), i.e., \( A \) and \( B \) are connected and \( C \) is a little further. As long as the graph has diameter \( d \geq 3 \), we can find the above three vertices (why?). Assume all servers are located at node \( C \) and the input sequence repeatedly makes requests to \( A \) and \( B \); the greedy algorithm moves one server from \( C \) to \( A \) for the first request and moves that server for all subsequent requests (since that server has distance 1 to the requested vertex while other servers have distance at least 2). The cost of the greedy algorithm is more than \( n \) for \( n \) requests (the requested node is never at the location of one of greedy servers).

On the other hand, an optimal algorithm moves 2 servers to cover both \( A \) and \( B \). For that it incurs cost of at most \( 2d \) (at most \( d \) for each server) and does not incur any cost after that (all subsequent requests are to vertices with a server in them). So, the ratio between the cost of the greedy algorithm and that of \( \text{Opt} \) is at least \( \frac{n}{2d} \) for the above sequence, and so is the competitive ratio of the greedy algorithm.

Note that the lower bound \( n/(2d) \) for competitive ratio of the greedy algorithm increases by the length \( n \) of the input sequence and hence greedy is a pretty bad algorithm.

3.2 General lower bound

We learned earlier that deterministic paging algorithm have a competitive ratio of at least \( k \), i.e., when the underlying metric is a complete graph, we cannot hope for a competitive ratio better than \( k \). Can we extend this to any metric space? The answer is indeed yes:

**Theorem 9.** For any graph \( G \), the competitive ratio of any deterministic \( k \)-server algorithm \( \text{Alg} \) is at least \( k \).

**Proof.** Consider a subset of \( k + 1 \) vertices of \( G \) so that all \( k \) servers are initially located at distinct vertices in this subset. An adversary forms the input sequence by repeatedly asking for vertices at which \( \text{Alg} \) does not have a server. Note that, since there are \( k + 1 \) vertices, there is always a vertex at which \( \text{Alg} \) has no server. Let the resulting sequence be \( \sigma = \langle r_1, r_2, \ldots, r_n \rangle \).

In order to analyze performance of \( \text{Alg} \) for \( \sigma \), we form \( k \) offline algorithms that maintain distinct configurations from each other and from \( \text{Alg} \). Here, by configuration, we mean the location of servers in the graph. Assuming that no two servers are located at the same node, there are \( k + 1 \) configurations, each defined by a vertex at which no server is located.

Consider the following sequence of steps: in step 1, \( \text{Alg} \) serves \( r_1 \) while all offline algorithms serve \( r_1 \) and \( r_2 \). After that at each given step \( i \), \( \text{Alg} \) serves request \( r_i \) while all offline algorithms serve request \( r_{i+1} \) at a total cost of \( d \). We show that at each given step, the cost of \( \text{Alg} \) is the same as all offline algorithms. To see that, consider step 1 and assume \( r_1 = x \), i.e., the first request is to vertex \( x \). Online algorithm has no server at \( x \) and moves a server from \( y \) to \( x \) at a cost of \( d = \text{dist}(x, y) \). Offline algorithms all have hits and incur no cost at this step; meanwhile, since the next request is \( r_2 = y \), exactly one offline algorithm which has no server at \( y \) moves a server from \( x \) to \( y \) to serve the second request to \( r_2 \). Note that this offline algorithm and \( \text{Alg} \) have swapped their configurations.
Figure 1: Lower-bound argument for $k = 5$. Assume the adversarial input is $\sigma = (ADF\ldots)$. Initially, all algorithms have different configuration and Alg has no server at $r_1 = A$ (a). At the first step Alg serves the request to $A$ be moving the server at $r_2 = D$. Meanwhile all offline algorithms have a hit for $r_1 = A$; for second request one offline algorithm (off3) moves a server from $A$ to $D$ (opposite direction of Alg) to serve request at $r_2 = D$. (b) At step 2, Alg moves a server from $r_3 = F$ to $r_2 = D$. Meanwhile an offline algorithm (off2) moves a server from $D$ to $F$ to serve $r_3 = F$. by moving servers in opposite directions. This procedure repeats in the subsequent steps: at each step $i$, Alg moves a server from node $y'$ to $x'$ for request $r_i = x'$ while exactly one offline algorithm moves a server from $y'$ to $x'$ for request $r_{i+1} = y'$. Other offline servers have a server located at $y'$ and do not make any move (see Figure 1). Let $C$ be the total cost of Alg for $\sigma$. Since at each step only one offline algorithm makes a move at the same distance as Alg, the cost of Alg and the total cost of all offline algorithms is equal for the adversarial input sequence $\sigma$. That implies that at least one of the offline algorithm has cost less than or equal to the average of $C/k$. To summarize, the cost of Alg is $C$ and the cost of an offline algorithm is at most $C$ and so is the cost of Opt. Hence, the competitive ratio of Alg is at least $C/k$.

Theorem 9 implies that no deterministic algorithm can achieve a competitive ratio smaller than $k$. But, is it possible to achieve a competitive ratio of $k$? This relates to the famous $k$-server conjecture that states that for any graph, there is an algorithm that has a competitive ratio of $k$. After more than thirty years, this conjecture is still open. However, we know that it is true at least for some graphs. First, we know it is true for complete graphs (since there are $k$-competitive paging algorithms). In the following sections, we see that it is also true for paths and trees.

### 3.3 Double-coverage-algorithm for paths

Double-coverage-algorithm (DCA) is an important algorithm which answers the $k$-server conjecture in the affirmative from graphs such as paths, trees, and also situations when $k = 2$. Here, we study the simplest version of the algorithm, which is defined on paths.

The double coverage algorithm for paths works as follows. Assume there is a request to $x$. If a server is located at $x$, the algorithm does not move any servers. Assume there is no server at $x$. 
There are two fundamental cases: In the first case, there are servers at both sides of $x$. In this case, DCA moves the closest server on left and closest server on right at the same ‘speed’ towards $x$ until one meets $x$. This implies that, if the closest server is at distance $d$, the algorithm incurs a cost of $2d$ to move tow servers at a distance of $d$ towards $x$. In the second case, there is no server on the left (or on the right) of $x$; in this case, the algorithm only moves the closest server to serve the request.

A review of potential function method

In order to analyze the double-coverage-algorithm, we use the potential function method. Recall that, in this method, we define a ‘potential’ as a function of the state of the algorithm with respect to that of Opt. In a sense, the potential captures how bad the current state is. If an algorithm incurs a cost of $c$ for a request and at the same time increases the potential, the increase also counts in the amortized cost of the algorithms (i.e., the amortized cost is more than the actual cost $c$). Similarly, if an algorithm manages to decrease potential, the amortized cost is also reduced. To be more precise, the amortized cost of serving a request is defined as actual cost plus the difference in potential. In order to prove that an algorithm Alg has a competitive ratio of at most $c$, we bound its amortized cost. In summary, we take the following steps to prove an algorithm is $c$-competitive:

1. Define the potential as a function of the states of Alg and Opt at time $t$ (before serving the $t$’th request).
2. Define the amortized cost at time $t$ as the summation of the actual cost and the difference in potential before and after serving the $t$’th request, i.e., $\text{amortized cost}(t) = \text{actual cost}(t) + \Phi(t+1) - \Phi(t)$.
3. Assuming the potential is defined properly, we should be able to show $\text{amortized cost}(t) \leq c \cdot \text{Opt}(t)$.

In order to take the above steps, after defining potential, we should figure how actions of Alg and Opt change the potential. For that we need to:
1) find what the cost of Opt is; we denote this cost with $\text{cost}(\text{Opt})$. 2) find how actions of Opt changes potential; for that we compute $\Delta_{\text{Opt}}\Phi$ which is the increase in potential as a result of actions of Opt 3) find what the actual cost of Alg is. 4) find how actions of Alg change the potential; for that we compute $\Delta_{\text{Alg}}\Phi$ which is the increase in potential as a result of actions of Alg. Provided with the above value, we have $\Phi(t+1) - \Phi(t) = \Delta_{\text{Opt}}\Phi + \Delta_{\text{Alg}}\Phi$ and hence it suffices to show at each request $t$:

$$amortized\text{cost}(t) = actual\text{cost}(t) + \Delta_{\text{Opt}}\Phi + \Delta_{\text{Alg}}\Phi \leq c \cdot \text{Opt}(t)$$

Upper bound for competitive ratio of DCA

We follow the above steps to prove DCA has a competitive ratio of at most $k$ for path metrics.

**Theorem 10.** Double-coverage-algorithm (DCA) has a competitive ratio of at most $k$ for paths.

**Proof.** First, we define a potential function as the summation of two components. The first component is defined based on the difference between the state (configuration) of DCA and Opt. As we saw earlier for the list update, it is desirable to be in a state that is close to that of Opt. Let $p(s_i)$ be the distance between the location of server $s_i$ in DCA configuration and the location of $s_i$ in Opt’s configuration. We define the first component, denoted by $P$, as $P = k \times (p(s_1) + p(s_2) + \ldots + p(s_k))$. So, the first component is the total distance that servers should move to get to Opt’s configuration
scaled by $k$. The multiplicative factor $k$ gives more weight to the first component compared to the second component. It will become clear later. The second component of the potential measures how far servers of DCA are from each other. This component does not depend on Opt’s configuration and is solely defined as a function of DCA’s configuration. For any pair $s_i, s_j$ of servers, let $d(s_i, s_j)$ be the distance between the locations of $s_i$ and $s_j$ in DCA’s configuration. The second component, denoted by $Q$, is defined as $Q = \sum_{i \neq j} q(s_i, s_j)$. Note that the closer the servers are to each other, the lower is the second component of the potential. Intuitively, we prefer servers to be close to each other. To see the reason, consider a sequence in which we repeatedly ask for two neighboring nodes in the path. Initially, one server $s_1$ serves these requests. However, unlike greedy algorithm, in DCA, another server $s_2$ also moves towards the request and at some point both servers are located in the neighboring vertices. How long does it take for $s_2$ to get close to $s_1$? It depends on the distance between the two servers. So, in a sense, it is better to have these two neighboring vertices closer to each other. Note that the second component is the summation of $\binom{k-2}{2}$ distances while the first component only involves $k$ distances. As a result, multiplying the first component by $k$ makes these two components comparable, which is required for our later calculations to work. To summarize, potential is defined as $\Phi = P + Q$ where $P$ is the summation of distances that servers should move to get to Opt’s configuration (scaled by $k$) and $Q$ is the total pairwise distances of servers in DCA’s configuration.

Now that we have defined the potential, we need to calculate all components involved in Inequality 1 to prove that the amortized cost is bounded. First, let’s consider the actions of Opt. Assume there is a request at node $x$ and Opt moves a server for a distance of $j \geq 0$ to serve it. Note that if Opt already has a server at $x$, we have $j = 0$. So, we have cost(Opt) = $j$. Next, we investigate how the potential is increased as the result of Opt’s move. The first component measures the distance between servers of DCA and Opt. When Opt moves a server a distance of $j$, the distance of that server with its location at DCA’s configuration increases by at most $j$. So, the first component of potential ($P$) increase by $k \cdot j$. The second component of potential ($Q$) does not depend on Opt’s configuration and hence it does not change when Opt moves a server. In summary, we have $\Delta_{\text{Opt}} \Phi \leq k \cdot j$.

Recall that DCA has two fundamental cases based on whether there are two or one servers on the two sides of a requested node. For each of these cases, we calculate how actions of DCA change the potential.

First, assume DCA moves only one server $w$ to server the request at node $x$. This happens when there is no server on left or right of DCA. W.I.o.g. assume there was no vertex on the left of $x$ and $w$ is the leftmost server that is moved $d$ distance to the left to serve $x$. Clearly, the actual cost of DCA is $d$. Now, the distance between $w$’s positions in DCA’s and Opt’s configurations is decreased by $d$. This is because $w$ is the leftmost server and, in Opt’s configuration, it is located at $x$ or on the left of $x$ (because Opt has served the request and has a server on $x$). Since the distance between locations of $w$ in configurations of DCA and Opt is decreased by $d$, we conclude that the first component $P$ of potential is increased by $-k \cdot d$. For the second component $Q$, note that the distance of $w$ and any other vertex is increased by $d$ since $w$ is the leftmost vertex which has moved $d$ more steps towards left. Since there are $k - 1$ other servers, the total pairwise distance $w$ and other servers is increased by $(k - 1)d$. Note that other servers are not moved and their pairwise distances do not change. To summarize, the actions of DCA increase the first and second components of potential by respectively $-k \cdot d$ and $(k - 1)d$, i.e., we have $\Delta_{\text{DCA}} \Phi = -k \cdot d + (k - 1)d = -d$. Recall that $\text{amortized \ cost}(t) = \text{actual \ cost}(t) + \Delta_{\text{Opt}} \Phi + \Delta_{\text{DCA}} \Phi$. Plugging $\text{actual \ cost}(t) = d$, $\Delta_{\text{Opt}} \Phi \leq k \cdot j$, and $\Delta_{\text{DCA}} \Phi = -d$, we get $\text{amortized \ cost}(t) \leq d + (k \cdot j) + (-d) \leq k \cdot j$. Since $\text{cost(Opt)} = j$, we have $\text{amortized \ cost}(t) \leq k \cdot \text{cost(Opt)}$. 

11
Next, assume there is a server on both sides of a requested node \( x \). In this case, DCA moves two servers \( L \) and \( R \) a distance of \( d \) to serve the request. First, note that the actual cost of DCA is \( 2d \). As before, one of the two servers moves closer to its position in \( \text{Opt} \)'s configuration while the other one might move further. So, the first component \( P \) of potential increases by at most 0. For the second component \( Q \), note that for any server \( s \notin \{R,S\} \), the distance of \( s \) with one of the moved servers increases by \( d \) and its distance with the other server decreases. So, total sum of all pairs of servers, except for pair \( (R,S) \), does not change. For the pair \( (R,S) \), since they move towards each other, their distance decreases by \( 2d \). In total, \( P \) increases by at most 0 and \( Q \) increases by at most \(-2d\), i.e., we have \( \Delta_{DCA} \Phi \leq 0 - 2d = -2d \). As before, we have \( \text{amortized}_\text{cost}(t) = \text{actual}_\text{cost}(t) + \Delta_{\text{Opt}} \Phi + \Delta_{DCA} \Phi \). Plugging \( \text{actual}_\text{cost}(t) = 2d, \Delta_{\text{Opt}} \Phi \leq k \cdot j \), and \( \Delta_{DCA} \Phi = -2d \), we get \( \text{amortized}_\text{cost}(t) \leq 2d + (k \cdot j) + (-2d) \leq k \cdot j \). Similarly to the previous case, since \( \text{cost}(\text{Opt}) = j \), we have \( \text{amortized}_\text{cost}(t) \leq k \cdot \text{cost}(\text{Opt}) \).

A careful look at the above proof shows that when DCA moves one server (case 1), the first component \( P \) of potential decreases while the second component increases. This is somehow reversed when DCA moves both servers, i.e., the first component \( P \) might remain the same while the it is the second component that decreases. This might serve an example where two or more components of the potential help us bound the amortized cost in different scenarios.

### 3.4 Double-coverage-algorithm for other metrics

Theorem 3.3 shows that DCA is the optimal algorithm for paths. This is because it has a competitive ratio of at most \( k \) and by Theorem 1 no deterministic algorithm can have a better competitive ratio.

Can we extend DCA to other metrics? The answer is Yes. Consider the following extension of DCA for trees. Upon a request to node \( x \), move all ‘neighbouring’ servers of \( x \) towards \( x \). We say a server \( s \) is neighbour to \( x \) if there is no other vertex on the unique path between \( s \) and \( x \). On a request to \( x \), all neighbours of \( x \) move 1 unit towards \( x \). After that, some servers might not be neighbours to \( x \) any more. In the second step, those servers which remained neighbours to \( x \) move one unit towards it, and this continues until at least one server arrives to \( x \).

**Theorem 11.** Double-coverage-algorithm (DCA) has a competitive ratio of at most \( k \) for trees.

**Proof.** The proof is similar to the case for paths. The potential is the same as before. If the cost of \( \text{Opt} \) is \( j \), we have \( \Delta_{\text{Opt}} \Phi \leq k \cdot j \). As before, we consider two cases based on whether DCA moves 1 or at least 2 servers. The analysis is the same for case 1 when 1 server is moved, i.e., the first component \( P \) of potential is decreased by \( k \cdot d \) (recall that \( d \) denotes the distance moved by server) while the second component increases by \( (k - 1)d \). The total increase in potential is \(-d \) and the amortized cost is \( \frac{d}{\text{actual}_\text{cost}} + \frac{k \cdot j}{\Delta_{\text{Opt}} \Phi} + \frac{-d}{\Delta_{DCA} \Phi} = kj \), which is no more than \( k \) times the cost \( j \) of \( \text{Opt} \).

In the second case, we calculate the difference in potential for each step. Assume at a given step \( \alpha \leq k \) servers move one unit towards \( x \). The actual cost for the step is \( \alpha \). Among these servers, at least one is getting closer to its position in \( \text{Opt} \)'s configuration while \( k - 1 \) server might get further. So, the first component \( P \) of potential increases by at most \(-k + k \cdot (\alpha - 1) = k(\alpha - 2) \). For the second component \( Q \), note that \( k - \alpha \) servers are not moved; for these servers \( \alpha - 1 \) servers are getting closer and one server is getting further (the server located on their path to \( x \)). So, \( Q \) decreases by \((k - \alpha) \cdot (\alpha - 2) \) for these servers. For each pair of servers that are moved, the distance is decreased by 2; there are \( \binom{\alpha}{2} \) such pairs, which implies that \( Q \) is decreased by
\[ \frac{\alpha(\alpha - 1)}{2} \cdot 2 = \alpha(\alpha - 1). \] In total, \( Q \) is decreased by \( (k - \alpha)(\alpha - 2) + \alpha(\alpha - 1) = k\alpha + \alpha - 2k \).

Recall that \( P \) was increased by at most \( k\alpha - 2k \). So, the total increase in potential is at most \(-\alpha\), which is equal to the actual cost of the algorithm for the step. Summing up for all steps, we have \( \text{actual cost} = \alpha_1 + \alpha_2 + \ldots + \alpha_m \) and \( \Delta_{DCA} \Phi \leq -\alpha_1 - \alpha_2 + \ldots - \alpha_m \). The amortized cost for serving the request will be \( \text{actual cost} + \Delta_{DCA} \Phi + \Delta_{Opt} \Phi = \Delta_{Opt} \Phi = kj \), which is no more than \( k \) times the cost \( j \) of Opt.

Lazy algorithms

Note that the double-coverage-algorithm might move more than one server to serve a single request. In a sense, the algorithm is moving a server just to make sure it is in a good state compared to Opt. An algorithm is called lazy if it moves at most one server to serve each request. Clearly, DCA is not a lazy algorithm. However, one can think of ‘delaying’ the movement of all servers except one that serves the request. Such delay does not increase the cost of the algorithm. This is formalized in the following theorem:

**Theorem 12.** Any non-lazy algorithm \( \text{Alg} \) can be converted to a lazy algorithm \( A' \) without increasing its cost.

**Proof.** In \( A' \), for each server, we maintain a real position and a virtual position. Virtual positions are maintained in a similar way that \( \text{Alg} \) moves servers. When \( \text{Alg} \) moves \( p \) servers for a request to a node \( x \), we only move one server to serve \( x \)(i.e., we update the real position of one server that arrives to \( x \)), while the virtual positions of other servers are updated according the way \( \text{Alg} \) moves them. In a sense, we ‘delay’ movements of other servers. This way, each server in \( A' \) serves the same sequence of requests that it servers in \( \text{Alg} \). In doing so, it might just avoid repetitive moves when it moves on \( \text{Alg} \) without serving a request. In short, the distances moved by each server in \( A' \) is no more than that of \( \text{Alg} \).

The case of \( k = 2 \) servers

We can use a variant of DCA to achieve a \( k \)-competitive algorithm when \( k = 2 \). We have to be careful on how we define such algorithm. Assume the two servers are located at nodes \( A \) and \( B \) of a graph and the request is at node \( C \). We consider the triangle \( A,B,C \) formed by the shortest paths between the three vertices. We can embed this triangle into a star with a center \( O \) that preserves the distances between vertices (see Figure 2).

When devising the DCA algorithm for \( k = 2 \), we ‘assume’ servers move on the stars according to double coverage algorithm, i.e., they move towards \( O \) until one blocks the other and after that only the closer server moves. In reality, the star is not a part of the tree and we cannot move vertices on the star; however, we only use star to maintain virtual positions of servers, i.e., we use

![Figure 2](image_url) **Figure 2:** The shortest paths on three vertices can be embedded into a star.
lazy algorithms that mimics the movements on the star by only moving one server for each request, while updating virtual positions on the star (see Theorem 12). In summary, upon a request at node $C$, we augment the metric by adding the star formed by shortest paths between two servers and the requested node and move servers on that star. In reality, a lazy algorithm only moves one server on its shortest path to the request in the actual graph. The distance that this server moves is the same as the distance it moves on the red star.

As an example, consider the graph of Figure 3 where there is a request at node $C$ and the two servers are located on $A$ and $B$. For serving the request, we move servers on the red, virtual star that embeds the shortest paths of $A, B, C$. Note that the shortest distances between these vertices is the same in the graph and in the star. Now, in order to serve request at $x$, we apply the DCA algorithm for trees (the star here), i.e., move both servers one unit towards $x$ on the red star; after that, server 1 blocks server 2 and in the next steps we only move server 2 towards $x$ until it arrives $x$. At the end, server 1 will be at node $C$ and server 2 will be on node $u$ of the red star. In the non-lazy version, we do not move server 2. So, we do not worry about $u$ not being a part of the actual graph. Assume the next request $y$ is to vertex $D$ of the graph; as before, we form a star that embeds the shortest paths between two servers and the request. To form this star, we note that the shortest distance between $(u, D)$ and $(C, D)$ are respectively 2 and 3. DCA moves server 2 on the darker red star to serve $y$ while server one moves 1 unit closer on the dark star, i.e., it moves to node $v$. In the lazy version, server 2 is not moved and server one moves from node $B$ to $D$.

**Theorem 13.** Double-coverage-algorithm (DCA) has a competitive ratio of at most $k$ when $k = 2$.

**Proof.** We provide a sketch of the proof. You complete it in your assignment. We use potential function method to prove the cost of the non-lazy DCA algorithm that moves vertex between their virtual positions (on red stars) is at most $k$ times the cost of Opt. By Theorem 12, this will imply that the cost of the lazy algorithm is at most $k$ times that of Opt.

As before, the potential has two parts, and it is defined similarly to the case of paths and trees. For the first part, we define $p(s_i)$ as the ‘distance’ of server $s_i$ in DCA’s configuration and Opt’s configuration. The first part $P$ of potential is defined as the summation of all $p(s_i)$’s for all servers. Here, by ‘distance’ we only consider the distances defined on the red stars. This is the same for the second component, which should be defined similarly to the case of paths and trees having the above note about distances in mind. Having this definition of potential, other aspects of the proof are straightforward. We should consider two steps in serving any request: in the first step both servers approach node $O$ (center of the star), and in the second step only one of the servers moves.
to serve the request. Following the same steps as for the trees, the proof follows.

Note that the above proof does not hold for \( k \geq 3 \). This is because, unlike the case of \( k = 2 \), servers and the requested node form 4 vertices whose pairwise shortest distances cannot be embedded into a tree.

Recall that paging is equivalent to \( k \)-server when the underlying metric is uniform, i.e., the pairwise distances between any two vertices is 1. Clearly, a complete graph is a uniform metric. Now, instead of a complete graph, consider a star graph in which all nodes except the center is associated with a page. Clearly, distances between all pairs of vertices (excluding the center) is the same. Hence, any sequence of requests to non-center vertices can be associated with an instance of the paging problem. Clearly, we can apply DCA for the star (which is a tree). In fact, DCA is equivalent to Flash-When-Full (FWF) paging algorithm. The non-center vertices which host a server represent pages which are in the cache. If there are \( \beta \) servers in the center, then the cache has \( \beta \) empty spots. Now, when there is a request to a vertex without a server, a fault has happened and a server should move to the requested vertex. If there is a server on the center, DCA sends that server to the node; in the context of paging, there is an empty space to the cache and there is no need for eviction. If there is no server in the center, DCA moves all \( k \)-serves towards the requested node, i.e., all of them move to the center. In the context of paging, the cache has been full and moving servers to the center is equivalent to flushing the cache, i.e., evicting all pages. After the flush, DCA moves only one server to the requested node, i.e., the requested page is brought to the cache.

Balancing algorithms

Another family of online algorithms for \( k \)-server are based on balancing the total distance moved by each server. In the simplest version Balance is an algorithm that, upon a request to a vertex \( x \), moves the server which after (potentially) serving the request, has moved less than other servers. It is not hard to see the competitive ratio of Balance can be much more than \( k \) in general. For that, consider the graph of Figure ?? and an input sequence \( \sigma = (D C B A)^n \). For each repetition of \( D, C, B, A \), Balance moves one server from \( B \) to \( D \) and back to \( A \), and the other serves makes similar moves from \( A \) to \( C \) and back to \( A \). The cost of Balance is \( 4nd \) (\( d \) for each request) while an optimal algorithm incurs a cost of \( d + 4n \) \((d \) to move one server to the other ‘end’ of the graph and one for each subsequent request). The competitive ratio of Balance is therefore at least \( \frac{4nd}{4n+d} \), which is roughly \( d \) for large values of \( n \). We conclude that Balance is not \( k \)-competitive for the general case. However, it is a good algorithm for the special case that there are only \( k + 1 \) vertices in the graph. This is an important case since the lower bound of Theorem 4 was based on this case. The following theorem indicates that Balance is the optimal algorithm for this case. We skip the proof in this class.

**Theorem 14.** Balance is \( k \)-competitive for metrics with \( k + 1 \) nodes

![Figure 4: The metric associated with a worst-case sequence for Balance.](image)
Work function algorithm

In this section, we study the work function algorithm, which is the best existing deterministic algorithm for general metrics and has a competitive ratio of at most $2k - 1$. The algorithm is known to be $k$-competitive (hence optimal) for lines, stars, and graphs with $m \leq k + 2$ vertices. In addition, the work-function algorithm is conjectured to be $k$-competitive for any metric spaces. If this conjecture is true, the work-function algorithm answers the $k$-server conjecture in the affirmative.

Let’s consider a ‘work-function’ which has two arguments: a configuration $X$ which indicates the location of $k$ servers in the graph and a time $t$ which indicates an index in the input sequence. The work-function is denoted with $w_t(X)$ and indicates the cost of an optimal algorithm $\text{Opt}$ for serving the first $t$ requests of the input sequence and ending up at configuration $X$. As an example, consider $k = 3$, let $X = (a, b, d)$ and $t = 10$; now, $w_{10}(X) = 12$ indicates that if you restrict $\text{Opt}$ to have its servers at nodes $a, b$, and $c$ after serving 10 requests, then the cost of $\text{Opt}$ will be 12. For two configurations $X$ and $Y$, let the distance between $X$ and $Y$ be the total distance that servers should move in order to get to configuration $Y$ when starting from configuration $X$.

Consider an input sequence $\sigma = (x_1, x_2, \ldots, x_t, \ldots, x_n)$. In order to get to configuration $X$ after serving $t$ request, $\text{Opt}$ should serve request $x_t$ before getting to configuration $X$. One way to think of that is, $\text{Opt}$ serves request $x_1, x_2, \ldots, x_t$ and end-up at a configuration $Z$ which has a server at $x_t$; until this point $\text{Opt}$ incurs a cost of $w_{t-1}(Z)$. After this, $\text{Opt}$ incurs a cost of 0 for serving $x_t$ (since $Z$ has a server at $x_t$) and then moves from configuration $Z$ to $X$ by moving servers a distance of $d(Z, X)$. In this formulation, $Z$ can be any configuration which has a server at $x_t$. Hence, we can write the following recursive formula for the work-function algorithm:

$$w_t(X) = \min_Z \{w_{t-1}(Z) + d(X, Z)\} \text{ so that } x_t \in Z$$  

(2)

Note that for $t = 0$ we have $w_0(X) = d(X, C_0)$ where $C_0$ is the initial configuration of servers in the graph; this is because if we restrict $\text{Opt}$ to be at configuration $X$ before serving any request, it just needs to move servers from $C_0$ to $X$ at a cost of $d(X, C_0)$. We can use a dynamic programming approach to calculate and maintain the values of work-function at time $t$ in an online manner. It means that we can calculate the cost of $\text{Opt}$ if we know the input ends right now. However, in order to devise an optimal algorithm, we should build the whole dynamic-programming table, find the minimum value of work-function in the last column (i.e., the configuration for which $\text{Opt}$ has minimum cost after serving all requests), and back-track to realize how $\text{Opt}$ moves servers to end-up at that configuration with minimum cost. Clearly, the backtracking step cannot be done in an online manner. Regardless, the work-function algorithm maintains the dynamic-programming table in an online manner.

At this point, we are ready to describe how the work-function algorithm works. As mentioned, it maintains the dynamic-programming table (in fact only the last column) in an online manner. Initially, it sets the values of $w_0(X)$ for all configurations; recall that $w_0(X) = d(X, C_0)$ where $C_0$ is the initial configuration. In addition, the algorithm pre-computes all pairwise distances between configurations and keeps these distances in a distance table. Now assume request $x_t$ appears. The algorithms takes the following steps:

- First, for any configuration $X$, the algorithm calculates $w_t(X)$ using Equation 2 and the values of the work function calculated in the previous step (i.e., $w_{t-1}(Y)$’s). For that, the algorithm refers to the distance table.

- Assume the algorithm is currently at configuration $C_{t-1}$. For any configuration $X$, the algorithm finds the value of $w_t(X) + d(C_{t-1} + X)$, i.e., for any configuration $X$, we find the summation of
the distance of $X$ to the current configuration and the value of work-function for $X$. Intuitively, the first component is a greedy component which implies how much we should move servers to get to configuration $X$ and the second component maintains how much costly it is for $Opt$ to be at configuration $X$ if the sequence ends after this request.

- Among all configurations, the algorithm chooses the configuration $X$ with minimum value of $w_t(X) + d(C_{t-1} + X)$ and moves servers to this configuration (i.e., $C_t = X$). Such configuration will have a server at $x_t$. Hence, by moving servers from $C_{t-1}$ to $C_t$, the algorithm serves request $x_t$.

Let's see an example of the work function algorithm for graph of Figure 5a. There are $k = 2$ servers initially located at nodes $C_0 = (A, D)$. The algorithm pre-computes pairwise distance between all configurations as in Figure 5b. Also, it maintains the values of work-function using dynamic programming as in Figure 5c. Initially, only the first column is set where $w_{t_0}(X)$ is simply $d(X, C_0)$. Assume the first request $x_1$ is a request to node $B$. In the first step, the algorithm calculates the work-function at time $t = 1$; i.e., the cost of $Opt$ for serving the first request to $B$ and ending up at a given configuration. For example, $w_1(A, B)$ is 2 because in order to have its servers at $(A, B)$ after serving $B$, $Opt$ should move the server at $D$ to $B$ at a cost of 2. The values of work-function can be calculated using Equation 2. Provided with the values of work-function at time 1, the algorithm finds the configuration which minimizes $W_1(X) + d(X, C_0)$. These values for all configurations (in the order they appear in the tables) are respectively $2 + 2, 3 + 2, 2 + 0, 3 + 1, 3 + 3, 1 + 1, 2 + 2, 2 + 1, 3 + 2, 2 + 2, 2 + 0, 3 + 1, 3 + 3, 1 + 1, 2 + 2, 2 + 1, 3 + 2, 2 + 2$, and the minimum value is 2, which is given by configuration $(B, D)$. It implies that, for serving the first request to $B$, the algorithm moves servers from configuration $(A, D)$ to configuration $(A, B)$, i.e., it moves the server located at $A$ to serve the request.

The above procedure repeats for the step that follow. Assume at time $t - 1$ the values of work-function are calculated as in Figure 5c, the request at time $t$ is to vertex $C$, and the current configuration is $(A, B)$. As before, the algorithm uses Equation 2 to calculate work-function values for time $t$ (next column in Figure 5c). After that, it finds the value of $w_{t-1}(X) + d(X, (A, B))$ for each configuration $X$. The configuration that minimizes this value is the one that the algorithm chooses to move servers to. As an example, consider configuration $(A, C)$; the value of work-function at time $t$ is defined through configurations that include the current request $C$. For each of these configurations, we sum the work-function at time $t - 1$ and its distance to the configuration $(A, C)$; the minimum value will be the work-function of $(A, C)$ at time $t$. For the four configurations $(A, C), (B, C), (C, D)$, and $(C, E)$ that include $C$, these values will be respectively $6 + 0, 9 + 1, 8 + 3, 11 + 2$, and the work-function $w_t(A, C)$ will be 6. Similarly, the work-function for $(B, C)$ is $w_t(B, C) = \min\{6 + 1, 9 + 0, 8 + 2, 11 + 1\} = 7$. The algorithm finds $w_{t-1}(X)$ for all configurations. In the second step, for each configuration $X$, the algorithm finds the summation of work-function at time $t$ and the distance of $X$ and the current configuration $(A, B)$. For example, for $X = (A, C)$, this sum will be $w_t(A, C) + d((A, C), (A, B)) = 6 + 1 = 7$ while for $X = (B, C)$, the sum is $w_t(B, C) + d((B, C), (A, B)) = 7 + 1 = 8$. Among these two configurations, the algorithm prefers $(A, C)$.

3.5 A review of randomized algorithms for $k$-server

Similar to many other problems, randomization can help in improving competitive ratio of $k$-server algorithms. It is conjectured that, for any metric space, there is a randomized algorithm which achieves a competitive ratio of $O(\log k)$. This conjecture is known as randomized $k$-server conjecture. The best existing algorithm has a competitive ratio of $\Theta(\log^2 k \log^3 m)$ for metrics of size $m$, which
Figure 5: An example of the work function algorithm.

is far from the value suggested by the conjecture. Note that this ratio is better than $k - 2$ of the work-function algorithm only when $k$ is sufficiently large with respect to $m$.

In this section, we review a technique used for introducing randomized algorithms in the context of cycles. For a cycle of length $C$, consider an algorithm CIRC which selects a point $P$, uniformly at random, from the cycle. CIRC interprets $P$ as a ‘road-block’ and applies the Double-Coverage-Algorithm for the resulting segment $L$. Observe that for any pair $(A,B)$ of points, road-block point $P$ appears on the shortest path between $A$ and $B$ with probability $d(A,B)/C$ (see Figure 6).

Let $OPT_{\text{Line}}$ be the optimal offline algorithm when restricted to line segment $L$. Since CIRC applies DCA for on $L$, we have $\text{cost}(\text{CIRC}) \leq k \cdot \text{cost}(\text{OPT}_{\text{Line}})$ (recall that DCA is $k$-competitive for lines). We claim that $\text{cost}(\text{OPT}_{\text{Line}}) \leq 2 \cdot \text{cost}(\text{Opt})$; if this is true, we can conclude the cost of CIRC is with a ratio $k$ of Opt, i.e., CIRC has a competitive ratio of at most $k$. To prove the claim, assume Opt makes moves of lengths $d_1, d_2, \ldots, d_n$. We modify Opt to get another algorithm which is restricted to the line: such algorithm applies the same moves as Opt; with additional penalty of at most $C$ if a server passes through $P$ in Opt’s scheme; the penalty means you go all the way through other side. Recall that the chance of passing $P$ on a move of length $d_i$ is $d_i/C$. The cost of the update algorithm can be written as $d_1 + \frac{d_1}{C} + d_2 + \frac{d_2}{C} + \ldots + d_n + \frac{d_n}{C} = 2\text{cost}(\text{Opt})$. Note that the updated algorithm is restricted to line, and hence its cost is at least equal to $\text{OPT}_{\text{line}}$, i.e., $\text{cost}(\text{OPT}_{\text{Line}}) \leq 2\text{cost}(\text{Opt})$. This proves the claim and hence CIRC has a competitive ratio of $2k$. Note that CIRC uses randomization and still has a competitive ratio worse than $2k - 1$ of work-function algorithm. However, the above technique for ‘distortion’ of a metric to a tree has been used in a variety of other algorithms.

4 k-server & advice

The advice complexity of online algorithms was first studied in the context of k-server problem. For a sequence of length $n$, it is easy to see advice of size $n \log k$ bits is sufficient to achieve an optimal solution. To see that, recall there is a lazy optimal algorithm (Theorem 12). In order to mimic such algorithm, an online algorithm requires to known which server should be moved for serving each request. This requires advice of size $O(\log k)$ for each request, which sums to $O(n \log k)$ bits of advice for inputs of length $n$.

Next, consider the special case where the underlying metric is a path. In this case, it suffices to know whether the lazy optimal algorithm serves each request by the left or by the right server.
Figure 6: CIRC selects a point $P$ randomly on the cycle and applies DCA on the resulting segment.

(note that if there is a server located at the node, the lazy algorithm does nothing and no advice is required). Knowing whether the left or right serve is used by $\text{Opt}$ is sufficient because one can easily check there are optimal algorithms in which no server passes the location of another server to serve a request. We conclude that $O(n)$ bits are sufficient to achieve an optimal solution. For each request, the advice indicates, in one bit, whether $\text{Opt}$ moves the left or right server. This sums to $n$ bits of advice. An online algorithm uses this advice to move the same servers as $\text{Opt}$ and hence mimics $\text{Opt}$. We conclude the following theorem:

**Theorem 15.** In order to achieve an optimal $k$-server algorithm for paths, an advice of size $n$ suffices for any sequence of length $n$.

### 4.1 Binary guessing problem & lower bounds

Theorem 15 implies that, in order to achieve an optimal algorithm, linear advice is sufficient. Can we further decrease the advice size to a sublinear function of $n$ and yet achieve an optimal algorithm? In this section we see that the answer is No. For that, we introduce Binary guessing problem.

Assume a sequence of length $w$ of bits appears in a deterministic online manner. Before the content of each bit is revealed at each step, a deterministic online algorithm needs to ‘guess’ whether it is ‘0’ or ‘1’. The algorithm’s objective is to correctly guess a maximum number of bits. Clearly, when the online algorithm has no information about future, an adversary can generate the sequence so that the algorithm’s guess is incorrect for each bit. This is because the adversary knows what the deterministic online algorithm will guess and sets the content of the bit to be the opposite. So, in a purely online setting, all guesses of the algorithm will be wrong in the worst case. Now, assume the algorithm receives one bit of advice which indicates whether ‘0’s or ‘1’s are more frequent in the input sequence. The algorithm can use this bit of advice to correctly guess at least half of bits: it just needs to always guess the more frequent bit indicated by the advice. Clearly, the more frequent bin is requested more than $w/2$ times and hence at least half of guesses are correct. It turns out that any other type of advice with size more than 1 bit cannot help to further improve this result unless the size of advice becomes linear to $n$. This is formalized in the following binary-guessing lemma:

**Lemma 1.** On an input of length $w$, any deterministic algorithm that guesses correctly on more than $\alpha w$ bits, for $1/2 < \alpha < 1$, requires at least $(1 + (1 - \alpha) \log(1 - \alpha) + \alpha \log \alpha) \cdot w$ bits of advice.

One easy way to interpret the binary-guessing lemma is this: “to guess more than $\alpha w$ bits for $\alpha > 1/2$, linear advice of size $\Omega(w)$ is required."
4.2 From k-server to binary guessing

In order to achieve a lower bound for the size of advice required to achieve an optimal algorithm for the k-server problem on paths, we reduce the k-server problem to binary guessing. For that, consider a simple setting of the problem where the metric is a line graph formed by \( m = 3 \) nodes and assume \( k = 2 \) servers are initially located at nodes 1 and 3. We show that, in order to achieve an optimal solution even in this simple setting, advice of linear size is required. To see that, consider input sequences formed by rounds of two types. A round of type 0 is defined by requests (2,1,2,1,3) and a round of type 1 is defined by (2,3,2,1,3). Note that the two rounds are identical except for the second request. Now, consider input sequences formed by \( w \) rounds; there will be \( 2^w \) such sequences. Each of these sequences can be described by a binary string \( B \) of length \( w \) in which the \( i \)'th bit indicates the type of the \( i \)'th round in the input sequence. We show that any k-server algorithm for this instance of k-server problem can be used for guessing the binary string \( B \) associated with the input sequences.

Note that each round, regardless of its type, ends with consecutive requests to nodes 1 and 3. This implies that that any ‘reasonable’ algorithm has servers at nodes 1 and 3 before start of any phase. It is formalized in the following lemma:

**Lemma 2.** Any algorithm \( \text{ALG} \) can be modified to another algorithm \( \text{ALG}' \) in which the servers are positioned at vertices 1 and 3 before starting to serve each phase.

**Proof.** Consider the first round \( R_t \) such that \( \text{ALG} \) does not have the servers positioned at 1 and 3, i.e., it has the servers positioned at 2 and 3. Note that since the last request of the previous round has been to vertex 3, there is a server at 3. This implies that the last two requests of the previous round \( R_{t-1} \) were served by the same server, and \( \text{ALG} \) incurs a cost of at least 2 for serving these requests. Consider an algorithm \( \text{ALG}' \), which moves the same servers as \( \text{ALG} \) but positions them at vertices 1 and 3. This requires a cost of at most 1 which happens when one of the servers is positioned at 2. Hence, \( \text{ALG}' \) incurs a cost of at most 1 for the last two requests in \( R_{t-1} \) and, compared to \( \text{ALG} \), saves a cost of at least 1 in round \( R_{t-1} \). At the beginning of round \( R_t \), servers of \( \text{ALG} \) are positioned at 2 and 3 and servers of \( \text{ALG}' \) are at 1 and 3. In future rounds, \( \text{ALG}' \) moves the server positioned at 1 in the same way that \( \text{ALG} \) moves the server position at 2. The total cost would be the same for both algorithms except that the cost for the first request served by the server positioned at 1 in \( \text{ALG}' \) might be at most 1 unit more when served by the server positioned at vertex 2 in \( \text{ALG} \).

To summarize, when compared to \( \text{ALG} \), \( \text{ALG}' \) saves a cost of at least 1 for requests in \( R_{t-1} \) and incurs an extra cost of at most 1 for requests in rounds after \( R_{t-1} \). Hence, the cost of \( \text{ALG}' \) is no more than that of \( \text{ALG} \).

According to the above lemma, any reasonable algorithm keeps servers at positions 1 and 3 before each round (otherwise, it can be modified to become reasonable without increasing its cost).

The first request of all rounds is to vertex 2. Assume the second request is to vertex 3, i.e., the round has type 1. An algorithm can move the left vertex \( s_l \) positioned at vertex 1 to serve the first request (to vertex 2) and use the right server \( s_r \) positioned at 3 to serve the second request the same vertex. For serving other requests of the round, the algorithm can move the servers to their initial positions and incur a total cost of 2 for the round (see Figure 7a). Note that this is the minimum cost that an algorithm can incur for a round. Next, assume that the algorithm moves the right vertex \( s_r \) to serve the first request (to vertex 2). The algorithm has to serve the second request (to vertex 3) also with \( s_r \). The third request (to vertex 2) can be served by any of the servers. Regardless, the cost of the algorithm will not be less than 4 for the round (see Figure 7b).
With a symmetric argument, in case the second request is to vertex 1 (i.e., the round has type 0), if an algorithm moves the right server to serve the first request, it incurs a total cost of 2, and if it moves the left server for the first request, it incurs a cost of at least 4 for the round.

To summarize, an algorithm should ‘guess’ the type of a round at the time of serving the first request of the round. In case it makes a right guess, it incurs a total cost of 2, and if it makes a wrong guess, it incurs a cost of at least 4 for the round.

From binary guessing lemma (Lemma 1), we know that advice of linear size is required to guess more than half of phases, i.e., a k-server algorithm requires advice of size $\Omega(w)$ to correctly guess the types of more than half of all phases. If the algorithm guesses half of rounds correctly and half incorrectly, its cost will be at least $\frac{w}{2} \times 2$ for correct guesses and $\frac{w}{2} \times 4$ for incorrect guesses; this sums to a total cost of $3w$ for all rounds. Note that $\text{OPT}$ serves all phases correctly and incurs a cost of $2w$. This implies that, if the algorithm guesses half of rounds correctly, it achieves a competitive ratio of $3/2$. By guessing lemma, however, guessing more than half of rounds requires advice of size $\Omega(w)$. We can conclude that, in order to achieve a competitive ratio better than $3/2$, advice of size $\Omega(w)$ is required. Note that $w = n/5$ and hence $\Omega(w) = \Omega(n)$. We conclude the following theorem:

**Theorem 16.** In order to achieve a competitive ratio better than $3/2$ in paths, any $k$-server algorithm requires advice of size $\Omega(n)$ for inputs of $n$ requests.

The above theorem implies that, not only for achieving optimal solution, but also for achieving any competitive ratio smaller than $3/2$, advice of size $\Omega(n)$ is required in paths. This implies that the simple scheme which receives $n$ bits of advice to achieve an optimal solution is asymptotically the best advice for the $k$-server problem.