1 Online algorithms and competitive analysis

An online algorithm is one that can process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the start. In contrast, an offline algorithm is given the whole problem data from the beginning.[1]

Online algorithms are distinguished from offline algorithms by the following features:

- They do not receive all of its input at once, but instead work on sequence of requests $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$, and if $\sigma(t), \sigma(t') = t' > t$.
- They work in a local fashion, only with the current moment and the past.
- They are not always optional; often because it is impossible to make global assumption about the data, and the decisions that are made may turn out to not be optimal in the feature.

In order to analyze online algorithms we use Competitive analysis. The main idea is to compare the online algorithm to its offline version, and show that it performs worse only be some constant factor $k$. From now on we denote the cost of online algorithm with $C_A$, and the cost of its offline(optimal) version with $C_{OPT}$.

Finally, we say that algorithm is $k$-competitive if:

$$C_A \leq kC_{OPT} + a,$$

where $a$ is a constant

2 List update problem

An example of online algorithm is the List update problem: Given a set of items in a list where the cost of accessing an item is proportional to its distance from the head of the list, and a request sequence of accesses, the problem is to come up with a strategy of reordering the list so that the total cost of accesses is minimized[2].

We consider unordered linked list, and implement the following operations:

- Access - to access an element we have to access all elements before it. The cost is $i$, where $i$ is the position of the element from the head of the list.
- Insert - to insert an element we have check if it is not already in the list. The cost is $n + 1$, where $n$ is the size of the list.
- Delete - to delete an element we have to access it and update the pointers. The cost is $i + 1$. 

1
If we want to exchange the position of elements in the list we have to possibilities:

- Free exchanges - if two elements are subsequent you just update have to update the pointers in constant time.
- Paid exchanges - to exchange two elements from position $i$ to position $k$, where $k > i$, you have to go through all $k - i$ elements

### 2.1 Move-To-Front algorithm

In order to speed up access operation algorithms are allowed to rearrange the list. The closer one element is to the beginning of the list, the faster we can access it. Let’s consider Move-To-Front (MTF) algorithm: where on access the element is moved at the beginning of the list.

We will prove the following theorem:

**Theorem**  MTF is 2-competitive.

**Proof**  We have to show that $C_{MTF} \leq 2 \cdot C_{OPT} + a$, where $a$ is a constant.

First we will prove the following auxiliary inequality. Let $t$ be particular access request, then:

$$C_{MTF}(t) + \Delta \Phi(t) \leq 2 \cdot C_{OPT}(t) - 1 \quad (1)$$

We sum over infinite sequence of $m$ requests:

$$\sum_{t=1}^{m} C_{MTF}(t) + \sum_{t=1}^{m} \Delta \Phi(t) \leq 2 \sum_{t=1}^{m} C_{OPT}(t) - m$$

$$\sum_{t=1}^{m} C_{MTF}(t) + \Phi(2) - \Phi(1) + \Phi(3) - \Phi(2) + \ldots + \Phi(m) - \Phi(m-1) \leq 2 \sum_{t=1}^{m} C_{OPT}(t) - m$$

$$C_{MTF} + \Phi(1) + \Phi(m) \leq 2 \cdot C_{OPT} - m$$

$$C_{MTF} \leq 2 \cdot C_{OPT} - m - \Phi(1) - \Phi(m) \leq 2 \cdot C_{OPT} + a \quad (2)$$

Note that in inequality (2), the term $-m - \Phi(1) - \Phi(m)$ always becomes smaller, and that is why we can bound it be some constant $a$.

We have shown that in order to prove (2) it is sufficient to show (1). We use this conclusion later in the proof.
Consider the following two lists from Figure 1 and Figure 2.

![Figure 1: MTF list](image1.png)

![Figure 2: MTF list](image2.png)

Both lists have the same $k$ elements in front. MTF list has $l$ elements before $x$, and $j$ elements after $x$. OPT list has $j$ elements before $x$, and $l$ elements after $x$.

The potential of MTF list is determined by the number of elements before the element we want to access compared to the number of elements before the same element in the OPT list. To formalize this intuition we introduce *inversions*.

We say that two elements $a$, $b$ are inversions if:

- $a$ is before $b$ in the MTF list
- $a$ is after $b$ in the OPT list

Thus the potential function $\Phi = \text{number of inversions}$.

Until now we did not make any assumption about the OPT algorithm. In fact we do not know what is the optimal algorithm. So, we have to investigate all possible offline algorithms for OPT, and show that $C_{MTF} \leq 2C_{OPT} + a$.

### 2.1.1 OPT is static

In the static version of OPT no elements are moved after access operation. So the access costs $C_{MTF}(t)$ and $C_{OPT}(t)$ are the number of elements before the element we want to access plus some constant operation after that. With some help from figure 1 and 2 we derive the following formulas:

$$C_{MTF}(t) = k + l + 1$$

$$C_{OPT}(t) = k + j + 1$$

3
To analyze the change in potential let’s consider what happens to MTF list and OPT list after one access on element $x$. For MTF list the element $x$ is put in front of the list as shown in figure 5, and OPT list stays unchanged as in figure 2. We are interested in the number of inversion: let $t - 1$ be the first access request, for $t - 1$ the potential is $\Phi(t - 1) = j + l$; in the second access $t$, the number of inversion changed to $\Phi(t) = k + j$. Thus we can conclude that:

$$\Delta \Phi(t) = k - l$$

Now, using inequality (1) we prove that MTF is 2-competitive for OPT static:

$$C_{MTF}(t) + \Delta \Phi(t) = k + l + 1 + k - l = 2k + 1 \leq 2(k + j + 1) - 1 = 2.C_{OPT} - 1$$

2.1.2 OPT is dynamic

There are two possibilities: the accessed element is moved $b$ elements backward or forward. We have to analyze both cases.

**Backward move**

```
K.x   J.x   L.x or x.L
```

```
k     j     X
```

```
b moves
```

Figure 4: OPT move backward list after one access on $x$

The access cost $C_{OPT}(t)$ is the number of elements before the element plus $b$ moves forward to locate the exchange element. The change in the potential between $t - 1$ and $t$ is the same as with the static OPT plus $b$ because $b$ more elements from $l$ are before $x$. We summarize as follows:

$$C_{MTF}(t) = k + l + 1$$

$$C_{OPT}(t) = k + j + 1 + b$$
\[ \Phi(t - 1) = j + l \]
\[ \Phi(t) = k + j + b \]
\[ \Delta\Phi(t) = k + b - l \]

Now, using inequality (1) we prove that MTF is 2-competitive for backward moving dynamic OPT:

\[ C_{MTF}(t) + \Delta\Phi(t) = k + l + 1 + k + b - t = 2k + 1 + b \leq 2(k + j + 1 + b) - 1 = 2C_{OPT} - 1 \]

**Forward move**

![OPT move forward list after one access on x](image)

The access cost \( C_{OPT}(t) \) is the same as with the static version because when moving to element \( x \), we store pointers and indexes of accessed elements, and after locating \( x \) we can move \( b \) elements back in constant time. The change in the potential between \( t - 1 \) and \( t \) is the same as with the static OPT minus \( b \) because \( b \) less elements from \( l \) are before \( x \). We summarize as follows:

\[ C_{MTF}(t) = k + l + 1 \]
\[ C_{OPT}(t) = k + j + 1 \]
\[ \Phi(t - 1) = j + l \]
\[ \Phi(t) = k + j - b \]
\[ \Delta\Phi(t) = k - b - l \]

Now, using inequality (1) we prove that MTF is 2-competitive for forward moving dynamic OPT:

\[ C_{MTF}(t) + \Delta\Phi(t) = k + l + 1 + k - b - t = 2k + 1 - b \leq 2(k + j + 1 + b) - 1 = 2C_{OPT} - 1 \]
3 Cache Algorithm / Page Replacement Algorithm

When CPU executes sequence of operations $q_1 q_2 \ldots q_m$ it reads data from the memory. Cache memory access is very fast, but with limited capacity because the bigger memory is the slower you can access it. When operation $q_i$ is run, the processor looks for required frame in the cache. If it is not found, page fault event is generated (see Figure 6). Missing part have to be brought from the main memory, which is of high capacity, but slow access.

In order to maximize efficiency we need to minimize number of page faults. Many cache algorithms has been developed for this purpose. If processor needs new page, but cache is full, they decide which page should be discarded. Examples:

- FIFO (First-In-First-Out),
- LRU (Least Recently Used),
- LFU (Least-Frequently Used).

**FIFO algorithm (First-In-First-Out)** In this strategy, we remove the oldest page from the cache. The assumption is that old page finished its job, so it will not be used in a future. Bad side of this approach is that page will be deleted even if it is used frequently, only because of its age.

**LRU algorithm (Least Recently Used)** We choose the least recently used page to be discarded. If some page was required in the close past, probably we
will need it again soon.

**LFU algorithm (Least-Frequently Used)** We choose the page which is used the least number of times. On page fault we throw out the page with lowest counter value. Disadvantage of this strategy is that old pages are privileged because counter of the new one starts with 0.

**Example 3.1** We will illustrate the performance of the above algorithms on cache with size \( k = 3 \) and input set \( Q = 23123412 \).

<table>
<thead>
<tr>
<th>( Q )</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIFO</strong></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td>√</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td><strong>LRU</strong></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td>x</td>
<td>√</td>
<td>x</td>
</tr>
<tr>
<td><strong>LFU</strong></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td>x</td>
<td>x</td>
<td>√</td>
</tr>
</tbody>
</table>

x - page fault; √ - page found in cache.

**OPTIMAL PAGING ALGORITHM:** Discard the page for which the next request occurs the latest in a future.

<table>
<thead>
<tr>
<th>( Q )</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OPT</strong></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td>√</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

**Theorem 3.1** \((\forall Q = q_1q_2...q_m) \) \( LFU(Q) \leq cOPT(Q) + a \)

Let \( k \) be the size of the cache, and \( Q = 11...122...2kk...k k+1,k+2,k+1,k+2,...,k+1,k+2 \) input than we have:

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( m )</th>
<th>( m )</th>
<th>( m )</th>
<th>( m )</th>
<th>( 2m )</th>
<th>( k+1,k+2,k+1,k+2,...,k+1,k+2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LFU</strong></td>
<td>x</td>
<td>√</td>
<td>...</td>
<td>√</td>
<td>x</td>
<td>...</td>
</tr>
<tr>
<td><strong>OPT</strong></td>
<td>x</td>
<td>√</td>
<td>...</td>
<td>√</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

\( LFU(Q) = k + 2m \) page faults
\( OPT(Q) = k + 2 \) page faults

**Theorem 3.2** \((\forall Q = q_1q_2...q_m) \) \( LRU(Q) \leq kOPT(Q) \) where \( k \) is the size of the cache.

Let’s prove: \( OPT(Q) \geq lowerbound \) and \( LRU(Q) \leq upperbound \)

\( \forall Q = q_1q_2...q_m, OPT(Q) \geq \) of distinct pages in \( Q \) with respect to current cache.
LRU(Q), k=3:  

1st dist. \hspace{1cm} a \hspace{1cm} \text{x x} \hspace{1cm} \text{x x}

2nd dist. \hspace{1cm} \text{x x} \hspace{1cm} \sqrt{\sqrt{\sqrt{x x}}}

3rd dist. \hspace{1cm} \text{x x} \hspace{1cm} \text{xx} \hspace{1cm} \text{xxฐาน} \hspace{1cm} \text{x x}

This approach will not work because \( d \neq a, b, c \) and \( g \neq d, c, f \) \( \Rightarrow g \neq a, b, c \) (operation \( \neq \) is not transitive).

OPT(Q), k=3:  

\( \text{x x} \sqrt{\sqrt{\sqrt{x x}}} \sqrt{\sqrt{x x}} \)

d e f

We have to prove that for every of \( l \) distinct pages we need to have at least one page fault.

Let’s prove it by induction on number of phases:

Phase 0: the first \( k \) distinct pages (\( k \) is the size of stack)

Phase \( i+1 \): starts from one distinct page in the phase \( i \), contains \( k \) distinct pages. \( Q = \{11...1 22...2 kk...k \} \)

\( LRU(Q) \leq k \cdot \text{number of phases} \)

\( OPT(Q) \geq \text{number of phases} \)

**Example 3.2** Size of cache \( k=3 \) and input set \( Q = 342 \)

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11123</td>
<td>4432</td>
</tr>
<tr>
<td>142</td>
<td></td>
</tr>
</tbody>
</table>

**Theorem 3.3** No deterministic on-line algorithm can achieve a better competitive rate than \( k \) where \( k \) is the size of cache, and \( Q \) is input set.

It is enough to be proven that \( (\exists Q) \ ALG \geq k \cdot OPT(Q) + a \). Such (tight) example will be \( Q \) that force algorithm ALG to discard \( i \) in \( k-th \) (i.e. \( i-th \) step), discard \( j \) in \( i-th \) step and continue to do it in all further steps. We can see it in table below:

<table>
<thead>
<tr>
<th>Q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>k</th>
<th>k+1</th>
<th>i</th>
<th>j</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALG</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>( \text{remove} )</td>
<td>( \text{remove} )</td>
<td>( \text{remove} )</td>
<td>( \text{x} )</td>
</tr>
<tr>
<td>OPT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>( \text{x} )</td>
<td>( \text{x} )</td>
<td>( \text{x} )</td>
<td>( \text{x} )</td>
<td>( \text{...} )</td>
</tr>
</tbody>
</table>

However, optimal algorithm can see the future and it will remove page which will be used latest.

**References**
