Randomized Algorithms for Online Matching with Two Sources

Bachelor Thesis

Thai Duong Nguyen

nguyetha@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Yuyi Wang
Prof. Dr. Roger Wattenhofer

September 9, 2017

Acknowledgements

I thank my supervisor Yuyi Wang for his help during the work on this thesis. He provided good advice on what aspects to analyze and assisted in the design of the algorithms.

Abstract

In 2016, Emek et al. [1] introduced the online *min-cost perfect matching with delays (MPMD)* problem. In this problem, requests arrive in a continuous time fashion and should be matched with each other into pairs. These requests will arrive at one out of *k sources*, which are given by a description of a metric space. An algorithm is allowed to either match two requests with each other or delay the matching. Whenever a matching is done, the algorithm pays a cost, which is equal to the distance between the two sources of the requests plus the time both requests have waited from its arrival until the matching. The *MPMD for two sources (2-MPMD)* problem is a restricted version of the MPMD problem, in which there are always exactly two sources (k = 2) with a unit distance between them. The quality of a 2-MPMD algorithm is measured by its *competitive ratio*, which essentially expresses how well the algorithm fares in the worst case in comparison to an optimal solution. In this thesis we analyze whether randomized online algorithms are able to achieve better results than deterministic ones. We present a randomized version of the deterministic algorithm in [2] and the reasoning and design decisions that we made in order to arrive at that algorithm. Furthermore, we prove that every randomized 2-MPMD algorithm has a competitive ratio of at least 1.8. Finally, we present an algorithm which calculates the cost and matching of an optimal offline solution specifically for the two source version with a runtime complexity of $O(n)$, where n is the amount of requests of the input.

Contents

Chapter 1

Introduction

1.1 Motivation

In current days, there exists a large multitude of online games and the popularity keeps rising. In many of those games, for example, Chess or FIFA, it is required that the gaming platform matches users with each other in order to be able to play the game. While matching users, the gaming platform has to consider two important criteria. One is to match suitable players with each other (in terms of skill level or network distance or other aspects), the other is to minimize the time of a player waiting during the matching process. Due to the online environment, these two criteria are often in conflict. What if the current players in the pool are a poor match? Should the platform wait? It is not guaranteed that a better matching player will arrive in the future, so how long should it wait?

The *min-cost perfect matching with delays (MPMD)* problem is a formalization of this challenge done by Emek et al. in 2016 [1]: Requests arrive in an online fashion at sources of a finite metric space, which is known in advance. The online algorithm matches these requests with each other by partitioning the request set into pairs. It is also allowed to delay the matching, in the hope of a more suitable request appearing in the future. Each match incurs a cost which consists of two parts: the space cost, which is equal to the distance between the sources of the metric space, and the waiting cost, which is equal to the waiting time since the arrival of both requests involved. The quality of an algorithm for this kind of problem is measured by its *competitive ratio* (refer to Section 1.2 for an exact definition).

A special case of the MPMD problem is the *two source min cost perfect matching with delays* problem, where the metric space always consists of only two sources with a unit distance between them. This problem is referred to as 2-MPMD and it abstracts away the space costs of the MPMD problem, leaving only the wait-or-match (cf. rent-or-buy) question for a request. Emek et al. [2] showed that any deterministic 2-MPMD algorithm has a competitive ratio of at least 3 and in addition to that provide a concrete algorithm which achieves a competitive ratio of exactly 3. In this thesis, we analyze whether a random-

ized algorithmic approach is able to achieve a better competitive ratio than 3 by designing a randomized algorithm and examining a lower bound for such algorithms. Additionally, a quick look is taken at how an offline algorithm could operate in order to produce an optimal offline solution specifically for the two source case.

1.2 Model

In an instance of the 2-MPMD problem, there are two *sources* denoted by a and b and a set R of requests. We also refer to R as a *sequence of requests* or *request sequence*. A request is characterized by its source $x \in \{a, b\}$ and its arrival time $t(r) \in R_{\geq 0}$. The algorithm receives the requests in an online continuous time fashion, which means that each request $r \in R$ is visible by the algorithm from the time $t(r)$ on. We assume throughout this thesis for every request sequence R that the amount of requests (denoted by $|R|$) it contains is even.

Any algorithm, on input of R, outputs a partition of R into unordered request pairs. The algorithm is allowed to delay the matching of requests in R. To be more precise, given two requests $r_1, r_2 \in R$ let $m(r_1, r_2, t)$ be the match operation which assigns r_1 to r_2 from time t on. Matching m incurs a cost which consists of two parts: A *space cost* and a *time cost*. If r_1 and r_2 have different sources ($x(r_1) \neq x(r_2)$) then we call m an *external match*, otherwise an *internal match*. The space cost of m is 1 if it is an external match and 0 if it is an internal match. The time cost of m is equal to the sum of the waiting time of r_1 and r_2, i.e. it is equal to $t - t(r_1) + t - t(r_2)$. $cost(m)$ denotes the total cost of match operation m and is equal to the sum of the space and time costs of the two involved requests. For an algorithm ALG, $M_{ALG}(R)$ denotes the set containing all match operations done by ALG on the sequence R. The total incurred cost of ALG on a sequence R is denoted by $cost_{ALG}(R)$ and is the sum of costs of all matching operations done by ALG while processing R. It is defined as $cost_{ALG}(R) = \sum_{m \in M_{ALG}(R)} cost(m)$.

In the context of some algorithm, an already arrived request is either *open* or *matched*. It is open if it was not yet involved in any match operation, otherwise matched. Furthermore, we assume that no two requests with the same source arrive at the same time.

In order to assess the quality of an algorithm, we use a measurement called *competitive ratio*. An algorithm ALG is said to be $\alpha-$*competitive* if there exists a universal constant β such that $cost_{ALG}(R) \leq \alpha \cdot cost_{OPT}(R) + \beta$ for every request sequence, where OPT is an optimal offline algorithm. $\frac{cost_{ALG}(R)}{cost_{OPT}(R)}$ is called the *cost ratio of ALG on sequence R*. If ALG is randomized then the *expected cost* of ALG is used: $E[cost_{ALG}(R)]$. One of the notably big differences between ALG and OPT is that ALG has no a priori knowledge of R.

1.3 Related Work

The work of Edmonds [3, 4] paved the way for matching as a classic problem in graph theory and combinatorial optimization. Karp, Vazirani, and Vazirani [5] ignited the interest in online matching and allowed many different versions of this problem to form. In most versions, it is assumed that requests belong to one side of a bipartite graph and the other half is given in advance.

Emek et al. [1] introduce a different version of the previously studied online matching problems in the aspect that the underlying graph (or metric space) is given in advance and the computational challenge lies within the unknown locations and arrival times of the requests (which are also unbounded in number). This is known as the MPMD problem and they provide a randomized algorithm with a competitive ratio of $O(\log^2 n + \log \Delta)$, where n is the number of points in the metric space and Δ is the aspect ratio, which is calculated as the quotient of the biggest distance and smallest distance of any two sources in the metric space. It is shown by Wang and Wattenhofer [6] that the algorithm of [1], if modified correctly, can also treat the bipartite version of the MPMD problem and achieve the same competitiveness. A different online MPMD algorithm with an improved logarithmic competitive ratio is presented by Azar et al. [7] and they also prove that no (randomized) online MPMD algorithm can have a competitive ratio of lower than $\Omega(\sqrt{\log n})$ in the all-pairs version and $\Omega(\log^{1/3} n)$ in the bipartite version.

In [2] Emek et al. studied a more restricted version of the MPMD problem: the 2-MPMD problem. They establish an upper bound of 3 on its competitive ratio by providing a deterministic variant of the online algorithm of [1] restricted to the 2-MPMD case. Additionally, a lower bound of 3 is proved by showing that any deterministic online 2-MPMD algorithm must have a competitive ratio of at least 3, effectively establishing a tight bound on the competitive ratio of deterministic online 2-MPMD algorithms. It is also shown that the competitive ratio of a special family of randomized algorithms that include the algorithm of [1] - the *memoryless* online 2-MPMD algorithms - is greater than 3. The 2-MPMD problem captures the essence of the *ski rental problem*, since the main question for a request in the 2-MPMD problem is to either wait (rent) or match (buy). As such, one could also consider work on the ski rental problem [8, 9, 10, 11, 12], in which randomness proved to be helpful, as related work to the 2-MPMD problem.

Chapter 2

Designing a Randomized 2-MPMD Algorithm

This chapter presents the design decisions and reasoning that were made in order to arrive at a randomized 2-MPMD algorithm, that is suspected to have a competitive ratio of less than 3. Considering a worst case algorithm is the same as having an *adversary* that always finds the worst possible sequence for that algorithm. The idea why randomization can potentially achieve better results stems from the fact that an adversary cannot abuse a specific property of the algorithm to achieve bad results, because the algorithm has a chance to process the requests in a different way (where that bad property might not hold).

2.1 Background

The designed algorithm is based on appropriate modifications of the deterministic algorithm DM2 in [2]:

Algorithm 0 Algorithm DM2 at time step t.

if there exist two open requests $r_1 \neq r_2$ with $x(r_1) = x(r_2)$ **then**
  $match(r_1, r_2)$
else if there exist two open requests $r_1 \neq r_2$ with $x(r_1) \neq x(r_2)$ **then**
  $T \leftarrow T + dt$
  if $T = 1$ **then**
    $match(r_1, r_2)$
    $T \leftarrow 0$
  end if
end if

DM2 is designed for a continuous time environment but is more easily understood when described as if it were in a discrete time environment, in which it takes discrete time steps dt. dt is infinitesimally small so that we can assume

that every request arrives in a separate time step. Whenever a request r_1 arrives and there exists another open request r_2 in the same source, then DM2 matches the two requests immediately. Apart from that, DM2 also keeps a timer T which increases only whenever an open request *in each source* is present at the current time. In a discrete sense: Given two consecutive time steps t_1, t_2, at the start of t_2 if there were open requests in each source during time step t_1 then add $dt = t_2 - t_1$ to T. Whenever T reaches 1, DM2 matches the currently two open requests, which reside in different sources, externally and resets T back to 0.

We use the notion of *smart* algorithms from [2]. An algorithm A (online or offline) is said to be *smart* if it satisfies the following property: If request r arrives at a source where there already exists an open request $r' \neq r$, i.e. $x(r') = x(r)$ and $t(r') < t(r)$, then A matches r and r' immediately, that is, at time $t(r)$. Notice that any smart algorithm will never have more than two open requests for a positive duration of time. Algorithm $DM2$, for example, is clearly smart by definition.

Lemma 2.1. *There exists an online method that transforms any algorithm A into a smart algorithm $\tilde{A}$ without increasing the total cost incurred by the algorithm.*

For the proof refer to [2, p. 5].

We subsequently assume that every algorithm we consider is smart (including the optimal offline algorithm OPT). The cost incurred by a smart algorithm A is comprised of three *cost components*:
(C1) the space cost incurred by A for matching externally;
(C2) the time cost incurred by A while there exists as single open request; and
(C3) the time cost incurred by A while there exist two open requests (one at each source)

Lemma 2.2. *The parity of the number of open requests is the same for any 2-MPMD algorithm and OPT at any time t.*

Proof. Let A be an arbitrary 2-MPMD algorithm. Let OPT and A run on the same sequence S. At any point in time t we have for A and OPT that the amount of seen requests up to t (i.e. all requests with arrival time less or equal to t) is equal to the sum of the matched requests and the open requests (these sets might differ for A and OPT, but the total amount is the same). The amount of matched requests must be even because matching is done in pairs. Therefore, the set of open requests of OPT and A must have the same parity. □

With Lemma 2.2 we are able to ignore cost component (C2) in the analyses regarding the cost ratio of some smart algorithm A. This is equivalent to only considering request sequences, where requests always arrive in both sources at

the same time. For the remainder of this thesis, we assume this property for every sequence considered and introduce a more compact notation:

We represent every pair of request r', r^* with $x(r') \neq x(r^*)$ and $t(r') = t(r^*)$ as a single request r. Since it is not necessary to specify the source of a request anymore it suffices to represent a sequence S in the following way:
$S = \{a, b, c, d\}$ is a sequence consisting of four requests r_1, r_2, r_3, r_4 with

$$t(r_1) = a \qquad t(r_2) = b \qquad t(r_3) = c \qquad t(r_4) = d$$

Matching r_1 at time t then means to do match operation $m(r'_1, r^*_1, t)$ and matching r_1, r_2 internally at time t means $m(r'_1, r'_2, t)$ and $m(r^*_1, r^*_2, t)$. Similarly, we consider the amount of requests in S (denoted by $|S|$) to be 4, even though it actually represents 8 requests (4 in each source). We refer to $|t(r_i) - t(r_j)|$ as the *gap between* r_i, r_j. Using this notation, DM2 can be rewritten into:

Algorithm 0 Algorithm DM2 at time step t.

```
if there exist two open requests r1 ≠ r2 then
    match(r1, r2)
else if there exists one open requests r1 then
    T ← T + dt
    if T = 1 then
        match(r1)
        T ← 0
    end if
end if
```

Emek, Shapiro and Wang also defined a special class of randomized algorithms in [2]: the *memoryless algorithms*. They also proved that any memoryless algorithm has a competitive ratio of greater than 3. Therefore, it is not necessary to consider any algorithms that fall into the class of the memoryless algorithms during the design of our algorithm.

2.2 Initial Idea

The starting point is a randomized version of the algorithm DM2.

In contrast to DM2, RDM2 flips a coin whenever T reaches 0.5 and depending on the outcome will either match and reset T or do nothing. If T reaches 1, RDM2 will still match r_1, r_2 deterministically. One of the reasons why deterministic algorithms are lower bounded by a competitive ratio of 3 is that they have to wait for a significant amount of time before being able to safely match a request pair externally. E.g. if DM2 would decide to match request pairs externally at $T < 1$ then the adversary could simply design a sequence where a request pair

Algorithm 1 Algorithm RDM2 at time step t.

if there exist two open requests $r_1 \neq r_2$ **then**
    $match(r_1, r_2)$
else if there exists one open requests r_1 **then**
    $T \leftarrow T + dt$
    if $T = 0.5$ **then** with probability $p = 0.5$:
        $match(r_1)$
        $T \leftarrow 0$
    else if $T = 1$ **then**
        $match(r_1)$
        $T \leftarrow 0$
    end if
end if

arrives immediately after the external match. Therefore, waiting until $T = 1$ is necessary for DM2. RDM2 introduces the possibility of matching a request pair externally sooner than DM2 in a random fashion, such that the adversary is not able to design a bad sequence in the same way as for DM2.

For the remainder of this thesis, let ϵ be a sufficiently small, positive number. We prove that the competitive ratio of RDM2 is greater than 3 by providing a counter example:

$$S = \{0, 0.5 + \epsilon, 2, 2.5 + \epsilon\} \tag{2.1}$$

See Figure 2.1 for an illustration of sequence S.

ϵ is necessary in order to have a defined behaviour for RDM2. If the second request would be exactly at 0.5, it would be unclear what RDM2 would actually do (already flipped the coin or not?). In this case ϵ allows us to model the situation in which a request appears immediately after RDM2 has flipped a coin at $T = 0.5$. For the cost calculations, we treat ϵ as if it were 0.

The cost ratio of RDM2 on S is as follows:

$$\frac{E[cost_{RDM2}(S)]}{cost_{OPT}(S)} = \frac{6.325}{2} = 3.1875 \tag{2.2}$$

This shows that the competitive ratio of RDM2 is greater than 3.

Note that in S the gap between r_2, r_3 can be arbitrarily large as long as it is greater than 1 (in our case it is 1.5). The idea is that RDM2 should not have the possibility of matching r_2, r_3 internally.

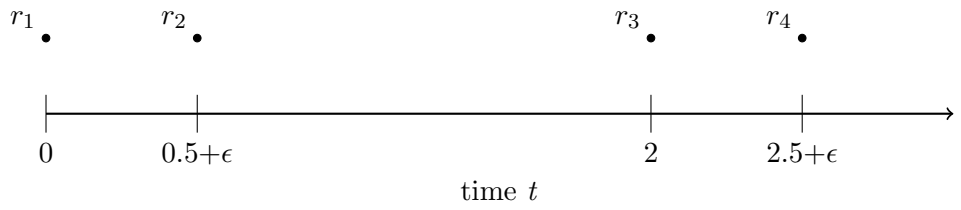

Figure 2.1: Illustration of sequence S. Note that every request r_i should represent two requests arriving at the same time in each source.

2.3 Modifying the Probability

The first aspect that we consider to modify in RDM2 is the probability p at line 5. Analyzing $E[cost_{RDM2}(S)]$, using a general p shows:

$$\begin{aligned} S_1 &:= p[2] + (1-p)[3] \\ S_2 &:= p[2 + S_1] + (1-p) \\ E[cost_{RDM2}(S)] &= p[2 + S_1 + S_2] + (1-p)[3 + S_1] = 6 + 4p^2 - p^3 - p \end{aligned}$$

In order to get $\frac{E[cost_{RDM2}(S)]}{cost_{OPT}(S)} < 3$, a value p must be found such that $4p^2 - p^3 - p < 0$ holds and we find that setting $p = 0.25$ yields a cost ratio of ~ 2.99 which is less than 3 for sequence S.

Algorithm 2 Algorithm RDM2 v2 at time step t with $p = 0.25$.

```
1:  if there exist two open requests r_1 ≠ r_2 then
2:      match(r_1, r_2)
3:  else if there exists one open requests r_1 then
4:      T ← T + dt
5:      if T = 0.5 then with probability p = 0.25:
6:          match(r_1)
7:          T ← 0
8:      else if T = 1 then
9:          match(r_1)
10:         T ← 0
11:     end if
12: end if
```

Although Algorithm 2 does have a cost ratio of less than 3 on sequence S, it is still very close to 3. There is a good chance that there exists a different sequence which actually has a cost ratio of greater than 3 for Algorithm 2, which proves to be true. We present counter example S'.

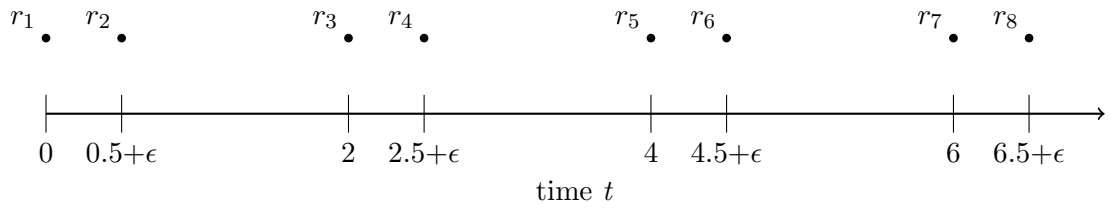

Figure 2.2: Illustration of sequence S'. Recall that every request r_i should represent two requests arriving at the same time in each source.

$$S' = \{0, 0.5 + \epsilon, 2, 2.5 + \epsilon, 4, 4.5 + \epsilon, 6, 6.5 + \epsilon\}$$

See Figure 2.2 for an illustration of sequence S'.

Similar to sequence S, the gaps between (r_2, r_3), (r_4, r_5) and (r_6, r_7) can be arbitrarily large as long as they are greater than 1. The cost ratio of $RDM2v2$ on S' is

$$\frac{E[cost_{RDM2v2}(S')]}{cost_{OPT}(S')} = \frac{12.1006}{4} = 3.0251$$

2.4 Avoiding Large Gaps

Looking at the counter examples S and S' in more detail, we notice a pattern: They use large gaps to connect duplicates of the same subsequence S_{sub} together, where S_{sub} consists of two requests separated by a gap of $0.5 + \epsilon$. Doing this impacts our suggested algorithms in a bad way. Usually, one could expect that if any two sequences A and B were merged together into a sequence AB by placing a large gap between them (e.g. between the last request of A and the first request of B), then the cost of an algorithm running on AB would be the sum of the costs of A and B. While this is true for OPT it is not for Algorithm 2: When processing S_{sub} it will either match both requests internally with each other or both externally. When matching both externally, it already made a poor decision, whereas matching both internally will leave the timer T at a value larger than 0.5, making the next instance of S_{sub} match poorly. This results in the cost of two S_{sub} connected together being greater than just two times the cost of S_{sub}, while OPT only has a doubled cost.

To counteract this property we improve our algorithm by letting it *reset timer T to 0* whenever it observes a *large gap*, which would be any gap of size 1 or greater. For the following algorithm let r_{last} and $r_{secondLast}$ with $t(r_{last}) >$

$t(r_{secondLast})$ be the two most recently seen requests (i.e. with the greatest arrival times at that time step).

Algorithm 3 Algorithm RDM2 v3 with resetting T at large gaps at time step t.

```
1: if t(r_last) − t(r_secondLast) > 1 then
2:     T ← 0
3: end if
4: if there exist two open requests r_1 ≠ r_2 then
5:     match(r_1, r_2)
6: else if there exists one open requests r_1 then
7:     T ← T + dt
8:     if T = 0.5 then with probability p:
9:         match(r_1)
10:        T ← 0
11:    else if T = 1 then
12:        match(r_1)
13:        T ← 0
14:    end if
15: end if
```

We notice a special property about Algorithm 3:

Lemma 2.3. *Let A, B be arbitrary request sequences. Let AB be the sequence obtained by connecting A and B with a large gap between the last request of A and the first request of B. We have that*

$$E[cost_{RDM2v3}(AB)] = E[cost_{RDM2v3}(A)] + E[cost_{RDM2v3}(B)]$$

Proof. The timer T is always set to 0 during the large gap between subsequence A and subsequence B. Due to T being the only state information that the algorithm keeps, A cannot influence the behaviour of the algorithm while processing B and vice versa. This means that after the algorithm has processed subsequence A in AB it will start processing subsequence B as if it would be a separate input, which implies that $E[cost_{RDM2v3}(AB)] = E[cost_{RDM2v3}(A)] + E[cost_{RDM2v3}(B)]$. □

With Lemma 2.3, analyzing the competitive ratio of Algorithm 3 is equivalent to ignoring lines 1-3 and only allowing request sequences with no gap larger than 1 as input.

In Algorithm 1, the improvement gained by changing p to 0.25 stems from the fact that it is better to wait and match internally in counter example S (i.e. to less likely execute lines 5-7). With Algorithm 3 we addressed a different aspect of the counter examples and therefore the case with $p = 0.5$ must be analyzed again.

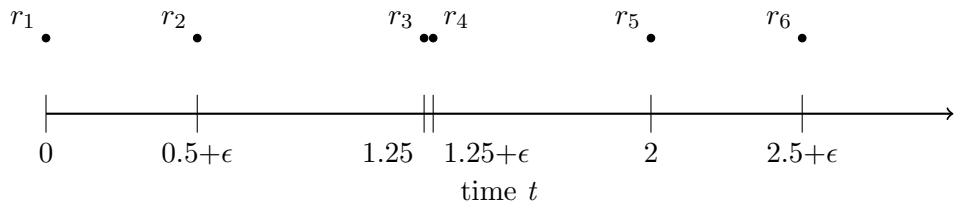

Figure 2.3: Illustration of sequence $S^*_{0.5}$. Recall that every request r_i should represent two requests arriving at the same time in each source.

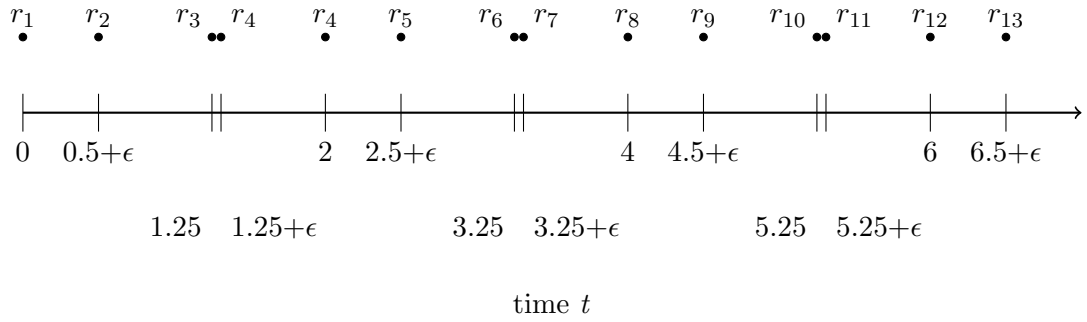

Figure 2.4: Illustration of sequence $S^*_{0.25}$. Recall that every request r_i should represent two requests arriving at the same time in each source.

Unfortunately, there is still a counter example possible in a similar pattern as before. We present two counter examples $S^*_{0.5}$ and $S^*_{0.25}$ for the cases of $p = 0.5$ resp. $p = 0.25$.

$$S^*_{0.5} = \{0, 0.5+\epsilon, 1.25, 1.25+\epsilon, 2, 2.5+\epsilon\}$$
$$S^*_{0.25} = \{0, 0.5+\epsilon, 1.25, 1.25+\epsilon, 2, 2.5+\epsilon, 3.25, 3.25+\epsilon, 4, 4.5+\epsilon, 5.25, 5.25+\epsilon, 6, 6.5+\epsilon\}$$

See Figure 2.3 resp. Figure 2.4 for illustrations.

Let $RDM2v3_{0.5}$ denote Algorithm 3 with $p = 0.5$ and analogously $RDM2v3_{0.25}$ with $p = 0.25$. Then the cost ratios are

$$\frac{E[cost_{RDM2v3_{0.5}}(S^*_{0.5})]}{cost_{OPT}(S)} = \frac{6.325}{2} = 3.1875$$
$$\frac{E[cost_{RDM2v3_{0.25}}(S^*_{0.5})]}{cost_{OPT}(S)} = \frac{12.1006}{4} = 3.0251$$

2.4.1 An Additional Timer

Considering the cost ratios of RDM2 v3 on its counter examples we notice that they are exactly the same costs as with RDM2 v2 and RDM2 on S' and S respectively. This shows us that there is an alternative way to put sequences together apart from using large gaps: In S^* a gap of 1.5 was simulated by *two gaps of size 0.75 with two extremely close requests inserted between them.* The two inserted requests achieve the same costs for RDM2 v3 as if they would not be there, while separating the large gap into two smaller gaps.

Consider the sequence $S^*_{0.5}$. Clearly, it would be the same request sequence as S if r_3 and r_4 were removed. To avoid confusion, we rename the requests of S into s_1, s_2, s_3, s_4. We claim that the expected cost of RDM2 v3 is the same as the expected cost of RDM2 running on S. We show that r_3, r_4 have corresponding cases in S, where the costs are the same. There are two ways how requests r_3 and r_4 can be matched by $RDM2v3_{0.5}$:

1) r_3 and r_4 get matched internally with each other: $m_1(r_3, r_4)$ or
2) r_2 and r_3 match internally and r_4 matches externally: $m_2(r_2, r_3), m_3(r_4)$

Case 1) can occur in two different situations depending on what happened before r_3, r_4 arrived: r_1 and r_2 could have either been matched internally with each other or both externally. If we ignore $m_1(r_3, r_4)$ of RDM2 v3, we can find equivalent situations when considering RDM2 running on S. The only difference which $m_1(r_3, r_4)$ can make, is a cost difference of ϵ and can thus be neglected.

Case 2) occurs if previously r_1 matched externally and r_2 does not match externally by itself. This is equivalent to the case in S where s_1 gets matched externally (at time 0.5) and s_2 matches externally at time 1.5. It is not difficult to see that $cost(m_2) + cost(m_4)$ is equivalent to the cost of the external match of s_2 at time 1.5.

Using this insight, we improve our algorithm such that it also recognizes *simulated large gaps.* We achieve this by introducing a second timer G that will reset timer T to 0 whenever it reaches 1. G increases whenever there is no open request present. In other words, G increases whenever the first timer T is not increasing. G is reset to 0 whenever the algorithm does an external match or G reaches 1.

Note that Lemma 2.3 still holds for Algorithm 4 and it suffices to only consider request sequences which only contain gaps that are less than 1, when reasoning about its competitive ratio.

Algorithm 4 Algorithm RDM2 v4 time step t.

if there exists no open request **then**
    $G \leftarrow G + dt$
    if $G = 1$ **then**
        $T \leftarrow 0$
        $G \leftarrow 0$
    end if
else if there exist two open requests $r_1 \neq r_2$ **then**
    $match(r_1, r_2)$
else if there exists one open requests r_1 **then**
    $T \leftarrow T + dt$
    if $T = 0.5$ **then** with probability p:
        $match(r_1)$
        $T \leftarrow 0$
        $G \leftarrow 0$
    else if $T = 1$ **then**
        $match(r_1)$
        $T \leftarrow 0$
        $G \leftarrow 0$
    end if
end if

Chapter 3

Lower Bound for 2-MPMD

This chapter presents a lower bound established for randomized algorithms and focuses on proving the following theorem:

Theorem 3.1. *Any randomized 2-MPMD algorithm has a competitive ratio of at least 1.8.*

We prove this theorem by using Yao's min-max principle [13, 14, 15], which states that the expected cost of a randomized algorithm on the worst case input cannot be better than the worst case random probability distribution of a deterministic algorithm which performs best for that distribution. In other words, it suffices to provide a distribution over inputs, such that every deterministic algorithm will have an expected cost ratio of at least 1.8 on that input distribution. The cost of an algorithm on a distribution of inputs, is the expected cost over those inputs.

We give an input distribution D which consists of two cases D_1, D_2.
D_1 consists of only a single request arriving at time 0 (recall that one request represents two requests, one in each source, at the same time). D_2 consists of two requests, where one request arrives at time 0 and a second request arrives somewhere within the time interval $[\epsilon, \frac{2}{3}]$ with a uniform distribution. See Figure 3.1 for an illustration of D_2. Both cases D_1, D_2 have an equal chance of appearing.

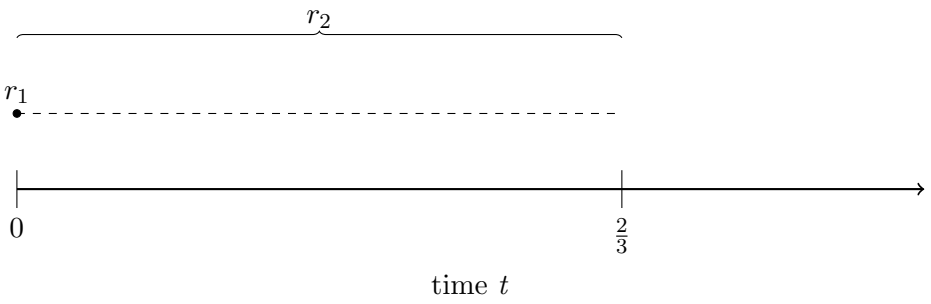

Figure 3.1: Illustration of sequence S_2. Request r_2 appears within the shown time window with a uniform probability.

We proceed in showing that every deterministic algorithm has an expected cost ratio of at least 1.8 on D.
An optimal offline algorithm would be able to distinguish D_1 from D_2 and will always match externally immediately after the request arrived in case of D_1 and otherwise wait until the second request arrives in case of D_2. The expected cost of an optimal offline algorithm OPT would then be:

$$E[cost_{OPT}(D)] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (\frac{2}{3} \cdot \frac{1}{2}) = \frac{5}{6}$$

The only two (reasonable) options a deterministic algorithm has for this input distribution is:

1) Match everything immediately as they arrive externally or
2) Wait some amount of time u before matching the first request externally, if within that waited time a second request arrives, match the first and second request internally with each other, otherwise match the second request externally immediately after arrival.
Let ALG_1 and ALG_2 denote the deterministic algorithms operating as described in 1) and 2) respectively.

The cost ratio for ALG_1 on D is

$$E[cost_{ALG_1}(D)] = \frac{1}{2} + \frac{1}{2} \cdot 2 = \frac{3}{2}$$
$$\frac{E[cost_{ALG_1}(D)]}{E[cost_{OPT}(D)]} = \frac{3}{2} \cdot \frac{6}{5} = \frac{9}{5} = 1.8$$

In order to determine the cost ratio of ALG_2 on D we minimize its cost term over the waiting parameter $u \in [0, \frac{2}{3}]$.

$$E[cost_{ALG_2}(D)] = \frac{1}{2}(1 + 2u) + \frac{1}{2}(\frac{3}{2}u(u) + (1 - \frac{3}{2}u)(2 + 2u))$$
$$\min_{\forall u \in [0, \frac{2}{3}]} E[cost_{ALG_2}(D)] = \frac{3}{2} \tag{3.1}$$

Where Equation 3.1 minimizes to $\frac{3}{2}$ at $u = \frac{2}{3}$. The cost ratio of ALG_2 is thus:

$$\frac{E[cost_{ALG_2}(D)]}{E[cost_{OPT}(D)]} = \frac{3}{2} \cdot \frac{6}{5} = \frac{9}{5} = 1.8$$

With which we have shown that every (reasonable) deterministic algorithm has a cost ratio of at least 1.8 on input distribution D. This in turn implies that every randomized algorithm has a competitive ratio at least 1.8.

Chapter 4

Optimal Offline 2-MPMD Algorithm

One way to obtain an optimal offline 2-MPMD solution would be to apply the general optimal offline algorithm for MPMD: Model the request sequence as a *min weight perfect matching* problem and then apply known algorithms on it. But since the general algorithm is designed for the general case, it might be more efficient to have an algorithm specifically for the 2-MPMD problem. In this chapter, we provide an optimal offline algorithm for the 2-MPMD case, which has a complexity of $O(n)$ where n is the number of requests in the input sequence.

The algorithm receives the input sequence S in form of $D(S)$ where $D(S)$ is an array containing the *gaps between the requests in S*. $D(S)[0]$ contains the first gap of S, $D(S)[1]$ the next gap and so on. E.g. the request sequence consisting of 4 requests arriving at times $\{0, 0.2, 0.7, 1\}$ would then be the array $[0.2, 0.5, 0.3]$, where 0.2 is the gap between the first and second request, 0.5 is the gap between the second and third request and so on.

4.1 Dynamic Program

We present a dynamic program DP that will determine the optimal offline cost and matching for an input sequence S. DP fills a 1-dimensional table T of length $n-1$, which is defined such that $T[0]$ contains the optimal offline cost for the first request of S, $T[1]$ contains the optimal offline cost for the first two requests of S and so on. We initialize the first two entries of T in the following way:

$$T[0] := 1 \qquad T[1] := \min(2, 2 \cdot D(S)[0])$$

Additionally, if $2 \cdot D(S)[0] < 2$, then a pointer is also stored in entry $T[1]$ which points to $T[0]$. DP then fills the rest of T in the following manner:

$$T[i] = \min(2 \cdot D(S)[i-1] + T[i-2], 1 + T[i-1]) \tag{4.1}$$

In addition to calculating the entry $T[i]$, DP also stores *a pointer* for each entry beyond the first, which points to either entry $T[i-1]$ or entry $T[i-2]$, depending on which part was actually the minimum in the calculation of $T[i]$. If $2 \cdot D(S)[i-1] + T[i-2] < 1 + T[i-1]$ then store a pointer to $T[i-2]$ else to $T[i-1]$.

The cost of an optimal offline solution to sequence S can be looked up in entry $T[n-1]$ and we can obtain the matched pairs of S by backtracking the pointers in the entry, starting with entry $T[n-1]$: If the pointer of entry $T[i]$ points only one entry back, the $(i+1)$-th request of S was externally matched, otherwise it was internally matched with the i-th request of S and we next consider the pointer of $T[i-2]$.

DP is correct because for any request in its input sequence, DP considers every possible action for it (match it externally, internally with previous request or internally with the following request) and picks the best out of those. It is clear that DP requires only a constant amount of time for each entry of T and therefore has a complexity of $O(n)$.

Chapter 5

Conclusion

5.1 Summary

Based on the deterministic algorithm DM2 from [2], we have designed a randomized 2-MPMD algorithm but also found counterexamples to it. Analyzing the nature of those counterexamples we found that the algorithm has a bad performance when processing specific sequences that contain *large gaps*. Using this insight we improved the algorithm accordingly and arrived at the current algorithm to beat: Algorithm 4. Using Yao's principle we concluded that any randomized algorithm has a competitive ratio of at least 1.8. We have also constructed a dynamic program which can be used as an optimal offline algorithm specifically for the 2-MPMD problem with a runtime complexity of $O(n)$.

5.2 Future Work

Most of our results are possibly only intermediate results and can be a good point to base future work on. One could analyze Algorithm 4 more in detail and establish an upper bound of less than 3 with it or find a counter example of a different nature, which does not rely on large gaps. Future works could also try to refine the lower bound we provided. Furthermore, the calculation of the expected cost for Algorithm 4 is suspected to be NP-hard but no proof was provided for that yet.

Bibliography

[1] Emek, Y., Kutten, S., Wattenhofer, R.: Online matching: Haste makes waste! In: 48th Annual Symposium on Theory of Computing (STOC). (June 2016)

[2] Emek, Y., Shapiro, Y., Wang, Y.: Minimum cost perfect matching with delays for two sources. In: Algorithms and Complexity: 10th International Conference (CIAC). (2017)

[3] Edmonds, J.: Paths, trees, and flowers. In: Canadian Journal of Mathematics. (1965)

[4] Edmonds, J.: Maximum matching and a polyhedron with 0, 1-vertices. In: Journal of Research of the National Bureau of Standards. (1965)

[5] Karp, R.M., Vazirani, U.V., Vazirani, V.V.: An optimal algorithm for online bipartite matching. In: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA. (2008)

[6] Wang, Y., Wattenhofer, R.: Perfect bipartite matching with delays. (2017)

[7] Azar, Y., Chiplunkar, A., Kaplan, H.: Polylogarithmic bounds on the competitiveness of min-cost perfect matching with delays. (2017)

[8] Seiden, S.S.: A guessing game and randomizd online algorithms. In: Annual ACM Symposium on Theory of Computing. (2000)

[9] Karlin, A.R., Manasse, M.S., McGeoch, L.A., Owicki, S.: Competitive randomized algorithms for non-uniform problems. In: In Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms. (1990)

[10] Karlin, A.R., Manasse, M., Rudolph, L., Sleator, D.: Competitive snoopy caching. In: Algorithmica, 3(1): 79 - 119. (1988)

[11] Dooly, D.R., Goldman, S.A., Scott, S.D.: Tcp dynamic acknolwedgement delay: theory and practice. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC), Dallas, TX, pp. 389-398. (1998)

[12] Karlin, A.R., Kenyon, C., Randall, D.: Dynamic tcp acknowledgement and other stories about e/(e-1). In: 33th Annual ACM Symposium on Theory of Computing (STOC). (2001)

[13] Borodin, A., El-Yaniv, R.: On randomization in on-line computation. In: Inf. Comput., 150(2):244-267. (1999)

[14] Stougie, L., Vestjens, A.P.A.: Randomized algorithms for on-line scheduling problems: how low can't you go? In: Oper. Res. Lett., 30(2):89-96. (2002)

[15] Yao, A.C.C.: Probabilistic computations: Toward a unified measure of complexity (extended abstract). In: 18th Annual Symposium on Foundations of Computer Science, pages 222-227. (1977)