Universität Trier

Competitive Analysis of Scheduling Problems and List Accessing Problems

DISSERTATION

zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt dem Fachbereich IV - Mathematik der Universität Trier.

Author:
Yida Zhu, MSc

Berichterstatter:
Prof. Dr. Sven de Vries
Dr. habil. Jan Pablo Burgard
Prof. Dr. Leonhard Frerick

October 23, 2019

Contents

Chapter 1

Introduction

An optimization problem consists of a set of *instances* together with the associated objective function value (*cost* or *profit*) which depends on the instance. For every instance, an algorithm must produce a sequence of decisions that will have an impact on the final objective function value. In the case of a *deterministic* algorithm, all of its decisions are made only based on the knowledge of the instance. Hence the objective function value depends only on the instance. By contrast, the objective function value of the result of a *randomized* algorithm[1] is further influenced by the random choices made by the algorithm.

An algorithm ALG is called *offline*, if the instance is completely known before ALG produces any decision. Otherwise, it is said to be *online*, i.e. a part of information about the instance is revealed step by step and ALG has to make decisions every time after the knowledge is updated.

Competitive analysis, suggested by Sleator and Tarjan (1985), is a method invented for analyzing online algorithms, in which the performance of an online algorithm is compared to the performance of an optimal offline algorithm. More precisely, let $C_{\text{ALG}}(\text{Ins})$ denote the returned cost after solving the instance Ins with algorithm ALG. An optimal algorithm OPT is characterized by $C_{\text{OPT}}(\text{Ins}) \leq C_{\text{ALG}}(\text{Ins})$ for all instances Ins and all algorithms for this problem.

Definition 1.1. *A deterministic online algorithm is c-**competitive**, if there exists a positive constant $c \geq 1$ such that for every instance* Ins*:*

$$C_{\text{ALG}}(\text{Ins}) \leq c \cdot C_{\text{OPT}}(\text{Ins}).$$

As one can imagine, one of the challenges in the aspect of competitive analysis is to bound the cost of the unknown optimal algorithm. In this thesis, two online optimization problems, scheduling and list accessing, are considered where the competitive analysis is commonly applied as a measure of performance.

In Chapter 2, we introduce the measure "competitive ratio" by considering scheduling problems. As a decision making process, scheduling problems play an important role

[1]see Section 3.5 for instance.

in many real world environments. Roughly speaking, an instance of one scheduling problem consists of a set of machines, a set of jobs and an objective function. The decision maker determines when to process which job on which machine in order to optimize the objective function. To name a few examples, machines may be processing unit in a computing environment, crew members of an airline company, or real machines in a workshop. The corresponding jobs may be executions of computer programs, flights that should be served by the crew members, and operations in a production process.

Two classes of scheduling problems are considered together with the known algorithms and their performance. During this chapter we will see that one key for solving scheduling problem in these two classes well is to find a good order of the jobs and finish the jobs in that order. At the end of this chapter, we provide the results of computer experiments, which reveal the strength of different algorithms descripted in this chapter. We end this chapter by pointing out that the list accessing problem shares this key property with scheduling problems.

The list accessing problem was originally considered by Sleator and Tarjan (1985) when they proposed the concept of competitive analysis. An instance of the list accessing problem consists of a list of distinct elements, a sequence of requests to elements in the list and the objective of minimizing the cost of serving the request sequence. Finding the requested element incurs a cost depending on where it is in the list, rearranging the list incurs a cost depending on the list order before and after the rearrangement. The decision maker determines how to rearrange the list after serving each request to achieve the minimal cost. List accessing algorithms are typically applied in the context of data compression, as will be explained in Chapter 3 and 4.

In Chapter 3, we introduce two classical proof techniques, the *potential function method* and the *list factoring technique*, to deepen the understanding of competitive analysis. Both techniques can be applied to prove that the competitive ratio of BIT, the central randomized online algorithm for list accessing problems, is bounded by 1.75. By analyzing the proof of the potential function method closely, this chapter ends with a conjecture of how to find a better bound for BIT. Such bounds are typically approached only by large instances.

To establish an overview, Chapter 4 focuses on average case analysis for small instances, where the requests are i.i.d. variables. A new closed formula for the expected cost of BIT, one of the best known algorithms for list accessing problem, are derived.

Finally, the last chapter describes computer experiments to support the results in Chapter 3 and 4. Since the problem itself is NP-hard, only small instance are considered for competitive analysis (as the optimal solution has to be computed explicitly). For larger instances, we compare the performance of different algorithms on empirical instance directly with each other.

Chapter 2

Competitive Analysis of Scheduling Problems

In general, scheduling problems can be understood as the problem of allocating resources over time to perform a set of tasks. Tasks have a set of different characteristics and they compete for common resources individually. In addition, different criteria may be taken into account to measure the quality of the performance of a set of tasks.

It is easy to imagine that scheduling problems emerge frequently in real-world situations.

Example 2.1. *Consider an exam in which every wrong answer is penalized with minus points. The resource to be allocated is the time. Tasks are the questions to be answered during the exam and the measurement is the score.*

Example 2.2 (Pinedo,2012)**.** *One of the functions of a multi-tasking computer operating system is to schedule the time that the CPU devotes to the different programs that have to be executed. The resource is the CPU time. Tasks are the programs waiting for being executed. Roughly speaking, the aim is to finish the calculations of as many important programs as early as possible.*

During this chapter, we use these two examples to illustrate the competitive analysis of scheduling problems. This chapter is organized as follows: Section 2.1 introduces two possibilities to model Example 2.1 and 2.2 formally. Section 2.2 presents the known results about these two models, including complexity, known algorithms and their performance in an online environment. In the last section, we provide computer codes to simulate the scheduling process with different characteristics to test the algorithms and support the results proposed in this chapter.

2.1 Notation

2.1.1 Graham Notation

Definition 2.3. *A **job** j is a task presented to the decision maker which has to be processed on a **machine** m for a certain period of time. The set of jobs is denoted by J and the set of machines is denoted by M. A **schedule** is an assignment of jobs to machines over the time horizon.*

Due to the widespread applicability, jobs and machines usually come up with additional attributes or constraints and there exist different objectives which may be considered. Usually, every machine is able to process at most one job and every job may be processed by at most one machine simultaneously. A schedule is said to be *feasible*, if these requirements and all constraints arising from the concrete problem are fulfilled. For the purpose of understanding competitive analysis, Definition 2.4 introduces only a limited set of necessary attributes, constraints and objective functions for this chapter[1].

Definition 2.4. *Depending on the application, a scheduling problem is equipped with*

- *a **single machine**, if only one machine is available,*
- ***identical machines**, if more than one machines are available with the same efficiency.*

Each job j is equipped with the following attributes:

- *The **release time** r_j is the point in time, from which the job j is available to be processed,*
- *the **processing time** p_j is the amount of time needed to process the job j,*
- *the **due date** d_j is the point in time, from which the job j is no more available to be processed, and*
- *the **weight** w_j is a priority factor, denoting the importance of job j relative to other jobs.*

*Once a machine m starts to process a job j at the point in time t, that job begins to accumulate time units until p_j is reached or the process is interrupted by the decision maker. If the latter case is allowed, the scheduling problem is said to be **preemptive**. Otherwise, the problem is **non-preemptive** and machine m is serving j until j is **finished** (the point in time $t + p_j$).*

- *The **completion time** c_j is the point in time, when j is finished.*

[1]We refer to Pinedo (2012) and Blazewicz et al. (2014) for a comprehensive list of attributes of scheduling problems.

- *The **unit penalty** u_j indicates if the job j is finished late, i.e.*

$$u_j = \begin{cases} 1, & \text{if } c_j > d_j, \\ 0, & \text{otherwise.} \end{cases}$$

Remark 2.5. *Notice that the attributes r_j, p_j, d_j, w_j are determined completely by the job, while c_j and u_j also depend on the applied schedule. Capital letters are used in the literature to indicate this dependency. In later chapters, random variables are used in some analysis, which are usually denoted with capital letters too. To avoid ambiguity, completion time and unit penalty are denoted with lower cases in this chapter.*

In Example 2.1, only one machine[2], namely the student himself, is available. Every question j is considered as a job with the following attributes:

- The release time $r_j = 0$ is the point in time when the exam starts for all j in this case,
- the processing time p_j is the time the student needs to solve the question j,
- the due date d_j is the point in time when the exam ends for all j, and
- the weight w_j is the score for answering the question j correctly.

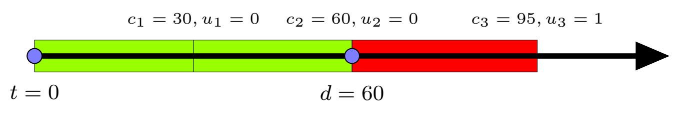
The aim is to achieve the highest score, or equivalently, to minimize the weighted sum of unit penalty $\sum\limits_{j \in J} \omega_j u_j$, i.e. the sum of points of those questions which are not finished in time.

Question j	Time p_j (min.)	Points w_j
1	30	3
2	30	3
3	35	4

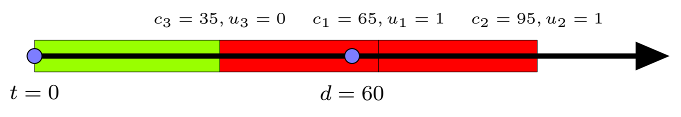
Table 2.1: The attributes for Example 2.1

Example 2.6. *Consider Example 2.1: The exam takes 60 minutes and consists of three questions. Given the attributes in Table 2.1, schedules can be visualized via a box chart, see Figure 2.1, where the color indicates if the question is finished in time. This chart is also known as Gantt chart.*

[2] A few assumptions are necessary to ensure the formal correctness of this example: We assume that the student is conservative (i.e. he answers only those questions at which he knows the correct answer to avoid the penalty for wrong answers) and honest (i.e. he does not try to cheat by copying the answer from other students). Furthermore, we assume that the time needed to solve a question is known beforehand.



(a) The schedule of finishing the first two questions but failed to answer the third question in time.



(b) The schedule of finishing the third question but failed to answer other two questions in time.

Figure 2.1: Possible schedules for doing exam.

Example 2.7. *In Example 2.2, the set M of machines is the set of available CPUs. Every program j is considered as a job with the following attributes:*

- *The release time r_j is the point in time when the user executes that program j,*
- *the processing time p_j is the time needed to finish the calculation required by j,*
- *the due date is not present in this example, and*
- *the weight w_j is the score of importance for the user of finishing j.*

The aim is to finish as many important programs as early as possible, this can be implemented as to minimize the sum of weighted completion time $\sum\limits_{j \in J} \omega_j c_j$.

Graham et al. (1979) introduce a three field notation $\alpha \,|\, \beta \,|\, \gamma$ to denote all relevant information of a scheduling problem where

- the field α includes the attributes of machines,
- the field β includes the relevant attributes of jobs, and
- the field γ includes the optimality criteria.

For instance: Example 2.1 can be denoted as $\mathbf{1} \,|\, d_j = d \,|\, \sum_{j \in J} \omega_j u_j$ [3], where $\mathbf{1}$ indicates that this is a single machine problem and $d_j = d$ indicates that the due dates are constant (not depended on j). If the specification r_j is not provided in the β field, then all jobs are released at the beginning. Example 2.2 can be denoted as $Pm \,|\, r_j, \text{prmp} \,|\, \sum_{j \in J} \omega_j c_j$, where Pm indicates that there are m parallel machine available and r_j indicates that the release time may be different and prmp is an abbreviation of preemptive. Certainly, maximizing both of these objective functions is not meaningful (as they are unbounded from above), the term "minimizing" is omitted here in the γ field. This notation, known as "Graham notation", is commonly used in the literature.

[3]Formally, preemption (switching to another question before finishing the current one) is allowed during the exam. But this is not a successful strategy from the experience, and is thus excluded by a conservative student.

2.1.2 Notation in Mathematical Programming

A large class of scheduling problems is known to be NP-hard, including Example 2.1 and 2.2. Mathematical programming is a commonly used tool to solve or approximate such NP-hard scheduling problems. For instance, the scheduling problem arising from Example 2.1 can be formulated as follows.

Lemma 2.8. *Consider an instance of the scheduling problem arising from Example 2.1. A schedule can be presented as a binary vector $(x_{mjt})_{m\in M, j\in J, t\in[d]}$ where the value of x_{mjt} is equal to 1 if and only if machine m begins to process job j at the point in time t. In the case of Example 2.1, the quantifier $\forall m \in M$ is omitted as there is only one machine available. A schedule $(x_{jt})_{j\in J, t\in[d]}$ is feasible if and only if the following constraints are satisfied:*

$$
\begin{aligned}
\textstyle\sum_{j\in J} x_{jt} &\le 1 && \forall t \in [d] && \text{(2.1a)}\\
x_{jt} &= 0 && \forall t > d - p_j && \text{(2.1b)}\\
\textstyle\sum_{t\in[d]} x_{jt} &\le 1 && \forall j \in J && \text{(2.1c)}\\
p_j(x_{jt} - 1) + \textstyle\sum_{j'\in J\setminus\{j\}} \sum_{k=t+1}^{t+p_j-1} x_{j'k} &\le 0 && \forall j \in J,\, t \in [d] && \text{(2.1d)}\\
x_{jt} &\in \{0,1\} && \forall j \in J,\, t \in [d] && \text{(2.1e)}
\end{aligned}
$$

The constraints (2.1a) *ensure that, simultaneously, only one question is assigned to the student* [4]*. In order to finish question j in time, the student has to start to answer j no later than $d - p_j$, constraints* (2.1b) *prohibit starting to answer a question too late.*

The no-preemption-rule is expressed by constraints (2.1c) *and* (2.1d)*: By* (2.1c)*, every question can be started at most once. If the student begins to answer j at the point in time t, then he is not able to start another question j' for the following p_j time units, i.e. $x_{jt} = 1$ implies $x_{j'k} = 0$ for all other questions j' and the entire period of time $k \in \{t+1, \cdots, t+p_j-1\}$. Thus, we have*

$$x_{jt} + x_{j'k} \le 1 \quad \forall j \in J, t \in [d] \forall j' \in J - j, k \in \{t+1, \cdots, t+p_j-1\} \tag{2.2}$$

(2.1d) *can be obtained by applying* (2.1a) *on the sum*[5] *of* (2.2) *over all $j' \in J - j$ and $k \in \{t+1, \cdots, t+p_j-1\}$.*

It remains to present the unit penalty u_j using x_{mjt}. Notice that, by (2.1b)*, a question j will be finished in time if and only if it is started early enough during the exam, i.e.*

$$u_j = \begin{cases} 0, & \sum_{k=1}^{d-p_j} x_{jk} = 1, \\ 1, & \textit{otherwise,} \end{cases}$$

[4] In general, if there is more than one machine available, it is necessary to add $\sum_{m\in M} x_{mjt} \le 1$ for all $j \in J$, $t \in [d]$ to ensure that the same job is assigned to at most one machine.

[5] Replacing a set of constraints with their sum leads to a weaker formulation in general. However, together with binary constraint, (2.1d) and (2.2) are actually equivalent.

or equivalently, $u_j = 1 - \sum_{k=1}^{d-p_j} x_{jk}$. *Therefore, the scheduling problem*

$$\mathbf{1} \,|\, d_j = d \,|\, \sum_{j\in J} \omega_j u_j$$

can be formulated as

$$\begin{aligned} \textit{minimize} \quad & \sum_{j\in J} \omega_j (1 - \sum_{k=1}^{d-p_j} x_{mjk}) \\ \textit{subject to} \quad & (2.1a), (2.1b), (2.1c), (2.1d), (2.1e) \end{aligned}$$

2.2 Known Complexity Result and Related Algorithms

In complexity theory, a distinction is made between *optimization problems* and *decision problems*. The question raised in a decision problem requires either a "Yes" or a "No" answer. For example, $\mathbf{1} \,|\, d_j = d \,|\, \sum_{j\in J} \omega_j u_j$ is an optimization problem to minimize the total weighted unit penalty. For any given z, one may also ask if it is possible to find a schedule such that the objective value is no more than z. We call the latter problem the *decision version* associated with the optimization problem $\mathbf{1} \,|\, d_j = d \,|\, \sum_{j\in J} \omega_j u_j$.

A fundamental concept in complexity theory is the concept of *problem reduction*. To reduce one problem to another, one provides a constructive transformation[6] mapping any instance of the first problem into an equivalent instance of the second, such that any algorithm that solves the second problem can be converted to an algorithm that solves the first problem. For example, it is easy to see that the decision version of an optimization problem can be reduced to the optimization problem itself: To answer the question if there exists a schedule with

$$\sum_{j\in J} \omega_j u_j \le z,$$

one may consider the optimization problem $\mathbf{1} \,|\, d_j = d \,|\, \sum_{j\in J} \omega_j u_j$ and calculate the optimal value z^* using some algorithm. Clearly, the answer to the decision version is "Yes" if $z^* < z$ and "No" if otherwise. Thus, if there exists an efficient algorithm to solve one scheduling problem (as an optimization problem) optimally, then this algorithm efficiently solves its decision version. Informally, the scheduling problem is at least as hard as its decision version[7].

Very often, one scheduling problem can be reduced to another scheduling problem

[6] Technically, the existence of a polynomial-time-transformation between these two problems is required in order to define the reducibility entirely. We refer to Karp (1972) for a formal correct definition.

[7] Actually, the converse is also true: The decision version of scheduling problem is at least as hard as the optimization problem itself, see Garey and Johnson (2002)

that allows further restrictions, e.g. the problem $\mathbf{1}\,|\,d_j = d\,|\sum_{j\in J}\omega_j u_j$ is a special case of (can be reduced to) $\mathbf{1}\,|\,d_j\,|\sum_{j\in J}\omega_j u_j$. We denote this reducibility by

$$\mathbf{1}\,|\,d_j = d\,|\sum_{j\in J}\omega_j u_j \quad\propto\quad \mathbf{1}\,|\,d_j\,|\sum_{j\in J}\omega_j u_j.$$

More general, since $d_j = d$ is a special case of d_j, we have the reducibility

$$\alpha\,|\,\beta, d_j = d\,|\,\gamma \quad\propto\quad \alpha\,|\,\beta, d_j\,|\,\gamma$$

for any environment α, β, γ. This fact is denoted simply by $d_j = d \,\propto\, d_j$.

In this section, we present various chains of reducible scheduling problems to establish an overview of the models arising from Example 2.1 and 2.2. In the manner of complexity, either efficient algorithms or a reduction from NP-complete problems are provided.

The following NP-complete decision problems are important from the scheduling point of view.

Definition 2.9 (Knapsack problem). *Given a set $\{1,\ldots,n\}$ of n items, each with a weight w_i and a value v_i, along with a maximum weight capacity W and a given target value v, it is to answer if the target value v is achievable without violating the weight capacity w, i.e. if there exists a subset $S \subseteq [n]$ such that*

$$\sum_{i\in S} w_i x_i \le W \text{ and } \sum_{i\in S} v_i x_i \ge v$$

Definition 2.10 (Partition problem). *Given n positive integers $a_1,\ldots,a_n$ and define*

$$b = \frac{1}{2}\sum_{i=1}^{n} a_i.$$

It is to answer if there exists a subset $S \subseteq [n]$ such that

$$\sum_{i\in S} a_i = b.$$

2.2.1 Known Result of Total Completion Time Related Scheduling Problem

In this subsection, we discuss the complexity of total completion time related scheduling problems. We distinguish the cases where

- either a single machine ($\mathbf{1}$) or identical machines (Pm) are available,
- if preemption (prmp or non-prmp) and different release time ($p_j = 0$ or p_j) are present, and

- whether an unweighted ($\sum_{j\in J} c_j$) or a weighted ($\sum_{j\in J} \omega_j c_j$) sum of completion times is the objective.

It is easy to see that the following reductions hold:

$$\begin{aligned} \mathbf{1} &\propto Pm, && \text{(denoted by } \dashrightarrow \text{ in Figure 2.2)} \\ r_j = 0 &\propto r_j, && \text{(denoted by } \longrightarrow \text{ in Figure 2.2)} \\ \sum_{j\in J} c_j &\propto \sum_{j\in J} \omega_j c_j. && \text{(denoted by } \boldsymbol{\longrightarrow} \text{ in Figure 2.2)} \end{aligned}$$

In total of 16 different scheduling problems can be obtained by combining these environments, we enumerate these 16 problems in Table 2.2.

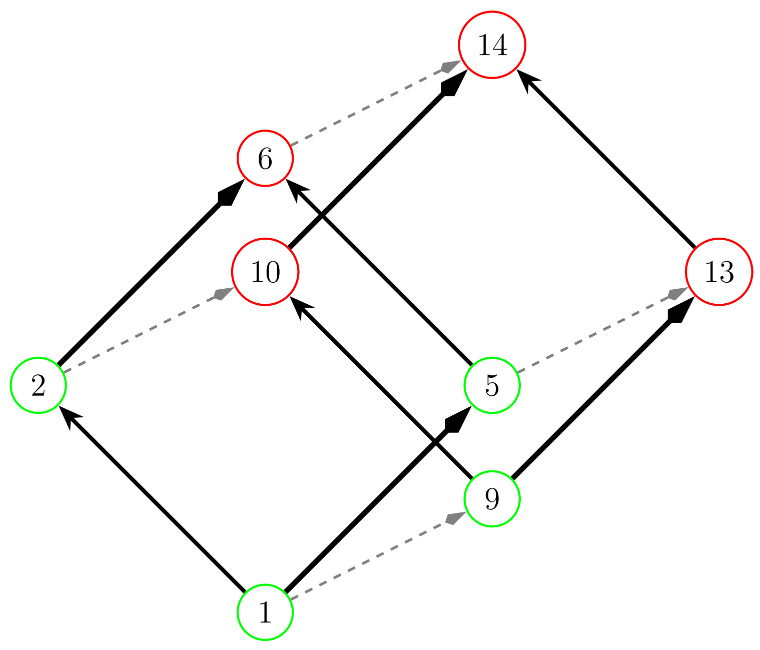
1	$\mathbf{1}\|\,\text{prmp}\,\|\sum_{j\in J} c_j$	5	$\mathbf{1}\|\,\text{prmp}\,\|\sum_{j\in J} \omega_j c_j$
2	$\mathbf{1}\|r_j,\text{prmp}\,\|\sum_{j\in J} c_j$	6	$\mathbf{1}\|r_j,\text{prmp}\,\|\sum_{j\in J} \omega_j c_j$
3	$\mathbf{1}\|\,\|\sum_{j\in J} c_j$	7	$\mathbf{1}\|\,\|\sum_{j\in J} \omega_j c_j$
4	$\mathbf{1}\|r_j\|\sum_{j\in J} c_j$	8	$\mathbf{1}\|r_j\|\sum_{j\in J} \omega_j c_j$
9	$Pm\|\,\text{prmp}\,\|\sum_{j\in J} c_j$	13	$Pm\|\,\text{prmp}\,\|\sum_{j\in J} \omega_j c_j$
10	$Pm\|r_j,\text{prmp}\,\|\sum_{j\in J} c_j$	14	$Pm\|r_j,\text{prmp}\,\|\sum_{j\in J} \omega_j c_j$
11	$Pm\|\,\|\sum_{j\in J} c_j$	15	$Pm\|\,\|\sum_{j\in J} \omega_j c_j$
12	$Pm\|r_j\|\sum_{j\in J} c_j$	16	$Pm\|r_j\|\sum_{j\in J} \omega_j c_j$

Table 2.2: Enumeration of total completion time related scheduling problems.

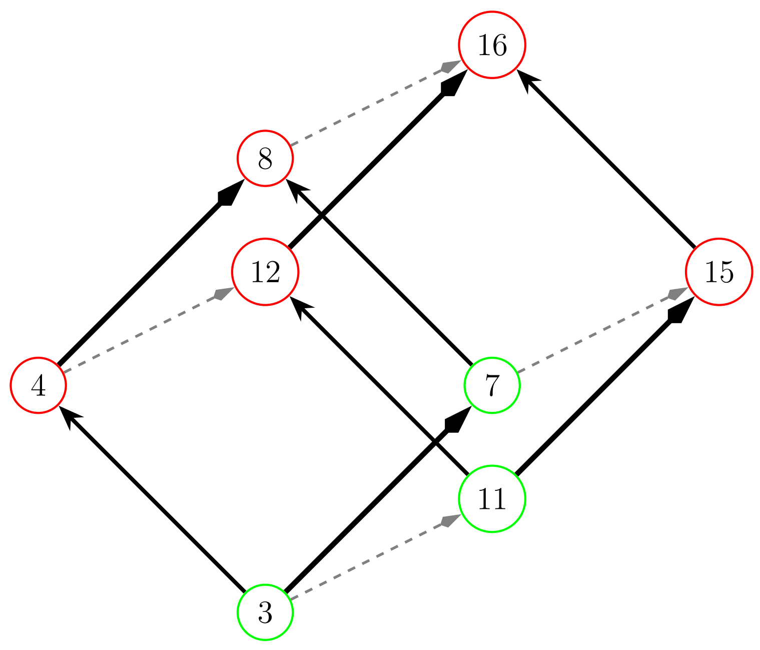
The complexity of these 16 problems can be summarized as directed graphs, see Figure 2.2. Vertices are numbered and present the scheduling problem according to Table 2.2, the color of vertices indicates the complexity of that corresponding problem, red for NP-hard and green for a problem in P. Arcs always direct from one problem to its reduction. For instance, the dashed edge from vertex 7 to 15 indicates the reduction

$$\mathbf{1}\,|\,|\sum_{j\in J} \omega_j c_j \quad \propto \quad Pm\,|\,|\sum_{j\in J} \omega_j c_j$$

from the P-problem $\mathbf{1}\,|\,|\sum_{j\in J} \omega_j c_j$ to the NP-problem $Pm\,|\,|\sum_{j\in J} \omega_j c_j$. Every directed path is a chain of reductions.



(a) Complexity hierarchy of preemptive scheduling problems listed in Table 2.2.



(b) Complexity hierarchy of non-preemptive scheduling problems listed in Table 2.2.

Figure 2.2: Three simple graphs

In order to verify the complexities shown in Figure 2.2, it suffices to show that problems $2, 5, 9, 7, 11$ are in P and problems $4, 6, 10, 13, 15$ are NP-hard.

Reduction from NP complete problems

The decision version of problems $4, 6, 10, 13$, and 15 can be reduced from partition problem. We summarize the basic idea of these reductions here.

Lemma 2.11 (Lenstra et al.,1977)**.** *Consider the scheduling problem* $\boldsymbol{1}\,|\,|\sum_{j\in J}\omega_j c_j$*. If, in a given instance, we have* $\omega_j = p_j$ *for every* $j \in J$*, then the objective value* $\sum_{j\in J}\omega_j c_j$ *is a constant, independent of the chosen schedule.*

Proof. In general, a schedule in this environment can be seen as an order of J. Whenever the machine is available, it takes the first job according to this order which has not been processed at that point in time.

As the release times of all jobs are equal to 0, it does not make sense to have idle time, that is, a period of time when the machine does not process any job. For a given schedule A, we enumerate the jobs according to the corresponding order and derive the following equations:

$$c_j = \sum_{i=1}^{j} p_i \quad \forall j \in [n]$$

and hence the resulting objective value is equal to

$$\sum_{j\in J}\omega_j c_j = \sum_{j\in[n]}\omega_j\left(\sum_{i=1}^{j} p_i\right).$$

Assuming that $\omega_j = p_j$ for every $j \in J$, it is thus to show that

$$\sum_{j\in[n]}\omega_{\pi(j)}\left(\sum_{i=1}^{j} p_{\pi(i)}\right) = \sum_{j\in[n]}\omega_j\left(\sum_{i=1}^{j} p_i\right)$$

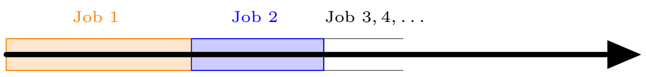
holds for all permutation $\pi \in S_n$. Since the symmetric group S_n is generated by transpositions $(\sigma_i)_{i\in[n-1]}$ where $\sigma_i = (i, i+1)$, it suffices to show the equation for all σ_i. We apply the standard *pairwise exchange* technique to prove this equation for σ_1, the other equations can be obtained by using exactly the same proof.

Consider another schedule B with the order $(2, 1, 3, 4, \ldots, n)$. Figure 2.3 illustrates these two schedules in Gantt chart. Beginning from the third job, all completion times in both schedules are identical. Thus, it only remains to show that $\omega_1 p_1 + \omega_2(p_1 + p_2) = \omega_2 p_2 + \omega_1(p_1 + p_2)$ which is equivalent to

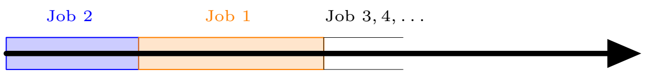
$$\omega_2 p_1 = \omega_1 p_2$$

which is satisfied trivially by the assumption $\omega_j = p_j$. □

To illustrate the common idea, we apply Lemma 2.11 to construct the reduction from partition problem to $P2\,|\,|\sum_{j\in J}\omega_j c_j$. This is, as the number of available machines is more restricted, certainly a special case of Problem 15: $Pm|\,|\sum_{j\in J}\omega_j c_j$.



(a) Schedule A.



(b) Schedule B.

Figure 2.3: Schedule A and B of Lemma 2.11

Lemma 2.12 (Lenstra et al.,1977)**.**

$$\textit{Partition} \propto P2 \,|\,|\, \sum_{j\in J} \omega_j c_j.$$

Proof. Given positive integers $a_1, \ldots, a_n$, define $b = \frac{1}{2}\sum_{i=1}^{n} a_i$.

Consider the Problem $P2\,|$ prmp $|\sum_{j\in J}\omega_j c_j$, for each a_i we construct a job whose processing time p_i and weight ω_i are both equal to a_i. Define $z = \sum_{i,j\in[n],i\le j} a_i a_j - b^2$. We claim that this partition problem has a “Yes”-answer if and only if there exists a schedule where $\sum_{j\in J}\omega_j c_j = z$.

For each subset $S \subseteq [n]$ we can construct a schedule $A(S)$ where the jobs in S are assigned to the first machine and jobs in $[n]-S$ are assigned to the second machine. We compare the objective value of $A(S)$ with the trivial schedule $A([n])$ where all jobs are assigned to the first machine (see Figure 2.4).

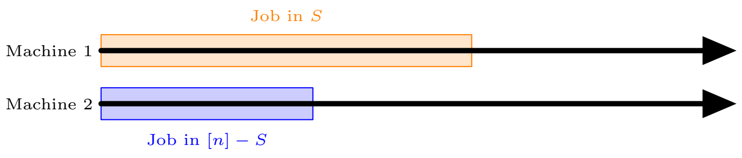
By Lemma 2.11, the order in which a machine processes these jobs is irrelevant and the objective value of $A([n])$ is equal to $\sum_{i,j\in[n],i\le j} p_i p_j$. The completion times of jobs in S are exactly the same in both schedules, whereas jobs in $[n]-S$ are finished $\sum_{j\in S} p_j$ earlier according to $A(S)$ than according to $A([n])$. Hence the objective value of $A(S)$ is equal to

$$\sum_{i,j\in[n],i\le j} p_i p_j - \left(\sum_{j\in S} p_j\right)\cdot\left(\sum_{j\in[n]-S}\omega_j\right) = \sum_{i,j\in[n],i\le j} a_i a_j - \left(\sum_{j\in S} a_j\right)\cdot\left(\sum_{j\in[n]-S} a_j\right).$$

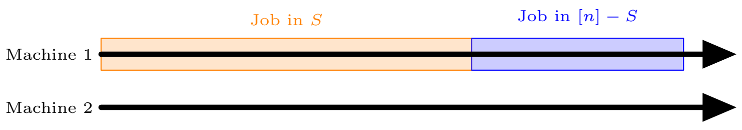
As $\left(\sum_{j\in S} a_j\right) + \left(\sum_{j\in[n]-S} a_j\right) = \sum_{j\in[n]} a_j = 2b$, the product $\left(\sum_{j\in S} a_j\right)\cdot\left(\sum_{j\in[n]-S} a_j\right)$ is less than or equal to b^2, Therefore, the objective value of $A(S)$ is greater than or equal to z for all S. The equation holds if and only if $\sum_{j\in S} a_j = b$, which is, if and only if the partition problem has a “Yes”-answer. □

The construction idea presented in the proof of Lemma 2.12 can be extended to reduce partition problem to Problem 4, 6 and 10. We refer to the corresponding references in the first half of Table 2.3 for a proof in full detail.

Given an arbitrary instance (M, J) of Problem 15: $Pm\,|\,|\sum_{j\in J}\omega_j c_j$, i.e. a set M of machines and a set J of jobs together with their processing times and weights, every schedule of this instance can also be interpreted as a schedule of the same instance (M, J)



(a) Schedule $A(S)$.



(b) Schedule $A([n])$.

Figure 2.4: Schedule $A(S)$ and $A([n])$ of Lemma 2.12

Id	Problem reduced from partition	Reduction
4	$\mathbf{1}\,|\,r_j\,|\sum_{j\in J}c_j$	Kan (2012)
6	$\mathbf{1}\,|\,r_j,\text{prmp}\,|\sum_{j\in J}\omega_jc_j$	Labetoulle et al. (1984)
10	$Pm\,|\,r_j,\text{prmp}\,|\sum_{j\in J}c_j$	Du et al. (1990)
13	$Pm\,|\,\text{prmp}\,|\sum_{j\in J}\omega_jc_j$	Problem 15
15	$Pm\,|\,|\sum_{j\in J}\omega_jc_j$	Lenstra et al. (1977)
Id	Problem solvable in polynomial time	Algorithm
2	$\mathbf{1}|r_j,\text{prmp}\,|\sum_{j\in J}c_j$	SRPT, Schrage (1968)
5	$\mathbf{1}|\,\text{prmp}\,|\sum_{j\in J}\omega_jc_j$	Problem 7
7	$\mathbf{1}|\,|\sum_{j\in J}\omega_jc_j$	WSPT, Smith (1956)
9	$Pm|\,\text{prmp}\,|\sum_{j\in J}c_j$	Problem 11
11	$Pm|\,|\sum_{j\in J}c_j$	SPT, Conway et al. (2003)

Table 2.3: The complexity results of problems from Table 2.2.

of Problem 13: $Pm\,|\,\text{prmp}\,|\sum_{j\in J}\omega_jc_j$. Thus, the optimal objective value v_{15} of (M,J) in Problem 15 is obviously an upper bound of the optimal objective value v_{13} of (M,J) in Problem 13. Interestingly, McNaughton (1959) shows that v_{15} cannot be reduced by allowing preemption, meaning that $v_{15}=v_{13}$. By applying Lemma 2.12, Problem 13 is also NP-hard.

Furthermore, this result of McNaughton also applies to Problem 7 and Problem 11, since they can be considered as special case of Problem 15. Thus, if Problem 7 and Problem 11 can be solved efficiently, then Problem 5 and Problem 9 can also be solved efficiently.

Offline Algorithms SPT and its Variants

It remains to show that Problem 2, 7, and 11 can be solved efficiently. The second half of Table 2.3 summerizes the optimal offline algorithms for these three problems.

One of the most often used general strategies for solving scheduling problems is *list* *scheduling* algorithm, whereby a priority list of the available jobs is maintained, and at each step the first available machine is assigned to process the first available job according to that priority list. In particular, all algorithms used in the second half of Table 2.3 are list algorithms.

Definition 2.13. *Let* ***Shortest Processing Time first*** *(short:* SPT*) denote the algorithm which always assigns the job j with the minimal p_j among all available jobs to a machine m as soon as m is idle.*

In the case that weights of jobs are present, let ***Weighted Shortest Processing Time first*** *(short:* WSPT*) denote the variant of* SPT *where jobs are assigned according to $\frac{p_j}{\omega_j}$ instead of p_j.*

In the case that preemption and release time are present, let ***Shortest Remaining Processing Time first*** *(short:* SRPT*) denote the variant of* SPT *where jobs are assigned according to the remaining processing time.*

SPT *and* WSPT *update the priority list whenever some machine m becomes available. Compared to this,* SRPT *also updates its priority list at every point in time whenever a job is released in addition. By such occasions, the job, which has the longest remaining processing time among all jobs already assigned to a machine, will be replaced by the first available job (is able to be processed but not assigned to any machine) according to the priority list.*

Example 2.14. *Consider an instance of Problem 14: $Pm|r_j, \mathrm{prmp}\,|\sum_{j\in J}\omega_j c_j$ with 2 machines and 4 jobs, see Table 2.4 for the attributes of these jobs. The schedule provided by* SPT*,* WSPT*, and* SRPT *are illustrated in Figure 2.5.*

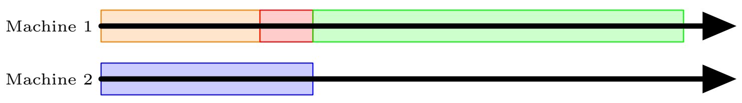
At $t = 0$, two machines are available. SPT *and* SRPT *share the same priority list $(1, 2, 3)$, whereas* WSPT *has a list $(3, 1, 2)$. Job 4 is excluded from these priority lists initially as it is still not released.* SPT*,* WSPT*, and* SRPT *assign the first two jobs according to their own priority list to available machine 1 and 2. Most assignments in this example are in this fashion.*

The only preemption happens at $t = 1$, when Job 4 is released. SRPT *updates the priority list to $(4, 1, 2, 3)$. Since job 1 and 2 are assigned with the remaining processing time 2 and 3 respectively, job 3, which has the longest remaining processing time 6, is interrupted and replaced by job 4.*

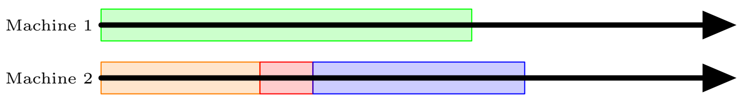
Job	Release time	Processing time	Weight
1	0	3	3
2	0	4	4
3	0	7	8
4	1	1	2

Table 2.4: Attributes of jobs in Example 2.14.

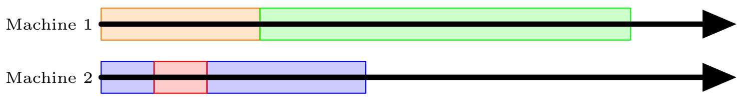
Remark 2.15. *Due to the common spirit of* SPT*,* WSPT*, and* SRPT*, Lemmas 2.16, 2.17 and 2.18 apply the same proof technique "pairwise exchange" to characterize optimal*



(a) SPT Schedule.



(b) WSPT Schedule.



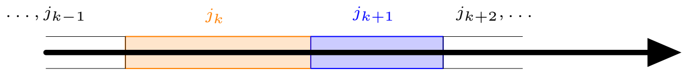
(c) SRPT Schedule.

Figure 2.5: Schedule produced by SPT, WSPT, and SRPT.

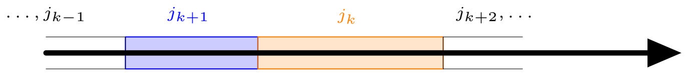
schedules in the corresponding problems, e.g. an optimal schedule of Problem 7 must follow WSPT*-rule. Since all of* SPT, WSPT, SRPT *are deterministic, the schedules produced by them are unique (apart from a permutation of jobs with the equal processing time, processing time by weight or the remaining processing time at any point in time t), a characterization of optimal schedule is thus equivalent to a proof of optimality of the corresponding algorithm.*

Lemma 2.16 (Smith,1956)**.** *Problem 7: $\mathbf{1}\,|\,|\sum_{j\in J}\omega_j c_j$ can be solved optimally by using* WSPT *rule.*

Proof. Given an arbitrary instance of $\mathbf{1}\,|\,|\sum_{j\in J}\omega_j c_j$ with n jobs and consider an arbitrary schedule A violating WSPT. We enumerate the jobs by the order of how A assigns them to machines, and let z be the number of ordered pairs $(j_i, j_{i'})$ of jobs violating the WSPT-rule, i.e. $\frac{p_i}{\omega_i} > \frac{p_{i'}}{\omega_{i'}}$ but $i < i'$. In particular, there must exist a job j_k, such that (j_k, j_{k+1}) violates the WSPT-rule, see Figure 2.6a.



(a) A Schedule A violating WSPT-rule.



(b) Improved Schedule A'.

Figure 2.6: Schedules in Lemma 2.16

We show that A can be improved by exchanging the jobs j_k and j_{k+1} on its schedule. Let A' denote the schedule after this exchange, see Figure 2.6b. Obviously, the completion time of all jobs other than j_k and j_{k+1} are identical in both schedules. We thus restrict our focus to the contributions to objective value from jobs j_k and j_{k+1}.

By deleting the first $k-1$ common initial jobs, it suffices to show that

$$\omega_k p_k + \omega_{k+1}(p_k + p_{k+1}) > \omega_{k+1}p_{k+1} + \omega_k(p_k + p_{k+1})$$

which is equivalent to $\frac{p_k}{\omega_k} > \frac{p_{k+1}}{\omega_{k+1}}$ after resorting the terms. This inequality holds trivially by the assumption that A violates WSPT-rule.

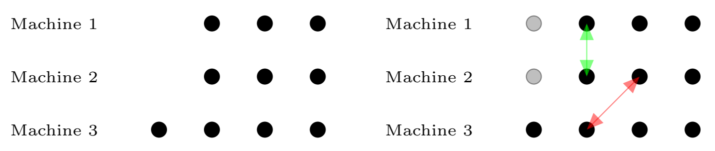
Therefore, any schedule with $z > 0$ is not optimal. In contraposition, an optimal schedule must follow WSPT-rule. □

Lemma 2.17 (Conway et al.,2003)**.** *Problem* 11*:* $Pm \,||\, \sum_{j\in J} c_j$ *can be solved optimally by using* SPT *rule.*

Proof. Consider an instance of $Pm \,||\, \sum_{j\in J} c_j$ with n jobs and an optimal schedule A. The job set J can be partitioned according to A as follows:

$$J = \dot{\bigcup}_{m\in M} \{j \in J \,|\, A \text{ assigns } j \text{ to machine } m\}.$$

Let J_m denote the set of jobs assigned to machine m. For the sake of a clear indexing, we add a number of dummy jobs with processing time 0 to each machine such that every machine receives an equal number of jobs, say $q = |J_m| = \lceil \frac{n}{m} \rceil$ for all $m \in M$. All dummy jobs are processed at the point in time $t = 0$, resulting in a completion time of 0 (see Figure 2.7 for an illustration). Hence the objective value is not influenced and thus the problem as well as the assumed optimality of A remains unchanged.



(a) Black points present jobs assigned by A.

(b) Grey points present added dummy jobs.

Figure 2.7: Adding dummy jobs to given schedule.

Let A' denote the schedule after adding dummy jobs and $j_{m,k}$ denote the k-th job assigned to machine m by A'. It is easy to see that the $c_{m,k} = \sum_{l=1}^{k} p_{m,l}$, i.e. the completion time of the job $j_{m,k}$ is equal to the sum of processing times of the first k jobs assigned to m. Thus, the contribution $z_m = \sum_{j\in J_m} c_j$ to objective value from machine m is equal to

$$\sum_{l=1}^{q} (q + 1 - l)p_{m,l}. \tag{2.4}$$

Observe that increasing (decreasing) the processing time $p_{m,k}$ by p results in an increase (decrease) of z_m by $(q-k)p$. Hence we conclude:

- Exchanging the k-th job on one machine m with the k-th job on another machine m' (the green double arrow in Figure 2.7b) does not influence the objective value at all.

To see this let $p = |p_{m,k} - p_{m,k'}|$. Since the contribution of one machine is increased by $(q-k)p$ and the contribution of the other machine is decreased by the same amount, the sum of them remains unchanged.

- The optimality of A' implies that $p_{m,k}$ is no smaller than $p_{m',k-1}$ for all $m' \in M$.

  Since otherwise, say $p_{m',k-1} - p_{m,k} = p > 0$, one can exchange $j_{m,k}$ with $j_{m',k-1}$ (the red double arrow in Figure 2.7b) so that $z_m + z_{m'}$ is decreased by p, contradicting the optimality of A'. In particular, the number of jobs assigned to one machine is at most one more than the number assigned to another machine. After adding dummy jobs, this can be seen easily by exchanging one real job with one dummy job (the red double arrow in Figure 2.8) and applying exactly the same argument as above.

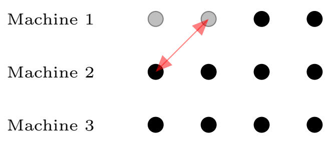


Figure 2.8: Unbalanced assignment is not optimal.

Furthermore, from Lemma 2.16 we know that the jobs in J_m must be assigned in a SPT-fashion. Therefore, every optimal schedule, after adding dummy jobs if necessary, must admit to the following property:

$$p_{m,k} \leq p_{m',k+1} \quad \forall m, m' \in M, k = 1, \ldots, q-1$$

where m' may be the same as m. In other words, an optimal schedule must follow SPT-rule apart from a permutation of jobs in $R(k)$ where

$$R(k) = \{j_{m,k} \,|\, m \in M\}. \qquad \square$$

Lemma 2.18 (Pinedo,2012)**.** *Problem 2: $\mathit{1} \,|\, r_j, \text{prmp} \,|\, \sum_{j \in J} c_j$ can be solved optimally by using* SRPT *rule.*

Proof. Similar to the proof of Lemma 2.16, we prove that every optimal schedule A must follow SRPT rule by showing if SRPT is violated, then the schedule can be improved.

Without loss of generality we may assume that, according to A, the machine does not take any idle time as long as there is still a job available. Since otherwise we may assign one available job to the machine to trivially reduce the objective value. Let t be the earliest point in time when SRPT is violated. Let $p_{j,t}$ denote the remaining processing time of j at the point in time t. Then there exist two available jobs j and j' such that $p_{j,t} > p_{j',t}$ and A assigns j to the machine. Let c_j and $c_{j'}$ denote the completion times of j and j' according to A respectively.

We adjust A to an alternative schedule A' in the following way:

- A' imitates every preemption and assignment from A of jobs other and j, j',

- A' imitates every preemption and assignment from A of jobs j, j' up to t, and
- after t, A' assigns only j' to the machine whenever either j or j' is assigned to the machine by A until j' is finished at the point in time $c'_{j'}$,
- after j' is finished, A' assigns only j to the machine whenever either j or j' is assigned to the machine by A until j is finished at the point in time c'_j.

Since the adjusted time intervals are all used by A to process either j or j', it is easy to see that the total length of these intervals are equal to $p_j + p_{j'}$ in both schedules and thus $\max\{c_j, c_{j'}\} = \max\{c'_j, c'_{j'}\}$. Since A assigns the job j with longer remaining processing time to the machine for a period of time, $\min\{c'_j, c'_{j'}\} = c'_{j'}$ is certainly smaller than $\min\{c_j, c_{j'}\}$. Therefore, the sum of completion times according to A' is less then the sum of completion times according to A. □

2.2.2 Known Result of Due Date Related Scheduling Problems

Being a rather complicated objective function, most research on $\sum_{j\in J} \omega_j u_j$-related scheduling problems focuses on the single machine environment. In this subsection, we discuss the complexity of some variants of $\mathbf{1} \,|\, d_j = d \,|\, \sum_{j\in J} \omega_j u_j$. We distinguish the following cases:

- if individual due dates ($d_j = d$ or d_j) are present, and
- if the weight of the jobs ($\sum_{j\in J} u_j$ or $\sum_{j\in J} \omega_j u_j$) is considered.

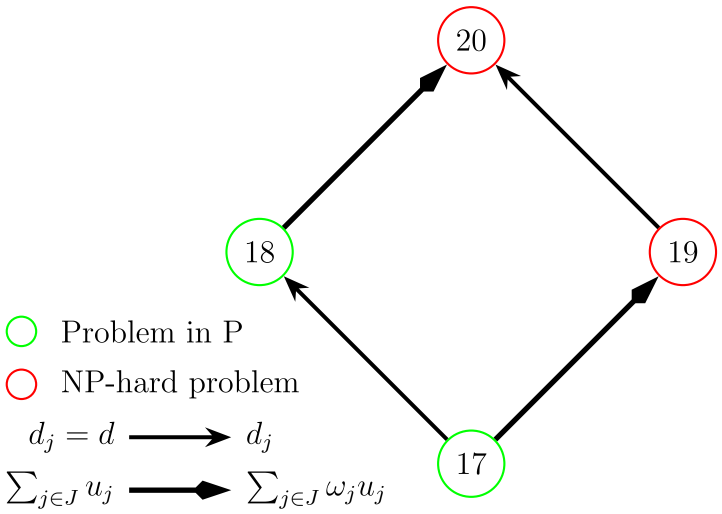


Figure 2.9: Complexity hierarchy of problems listed in Table 2.5.

It is easy to see that the following reductions hold:

$$d_j = d \propto d_j, \qquad \text{(denoted by } \longrightarrow \text{ in Figure 2.9)}$$

$$\sum_{j\in J} u_j \propto \sum_{j\in J} \omega_j u_j. \qquad \text{(denoted by } \relbar\joinrel\blacklozenge \text{ in Figure 2.9)}$$

A total of 4 different scheduling problems can be obtained by combining these environments, we enumerate these 4 problems in Table 2.5.

17	$\mathbf{1}\vert d_j = d\vert \sum_{j\in J} u_j$	19	$\mathbf{1}\vert d_j = d\vert \sum_{j\in J} \omega_j u_j$
18	$\mathbf{1}\vert d_j\vert \sum_{j\in J} u_j$	20	$\mathbf{1}\vert d_j\vert \sum_{j\in J} \omega_j u_j$

Table 2.5: Enumeration of due date related scheduling problems.

Similar as in the previous section, we summarize the complexity of these four problems in Figure 2.9. In order to verify the complexities shown in this graph, it suffices to show that Problem 18 is in P while Problem 19 is NP-hard.

Reduction from NP complete problem

Lemma 2.19 (Pinedo,2012)**.**

$$\textit{Knapsack problem} \propto \mathbf{1} \mid d_j = d \mid \sum_{j\in J} \omega_j u_j$$

Proof. The reduction identifies the parameters of knapsack problem and $\mathbf{1} \mid d_j = d \mid \sum_{j\in J} \omega_j u_j$ to each other, see Table 2.6. □

$\mathbf{1} \mid d_j = d \mid \sum_{j\in J} \omega_j u_j$	Knapsack
Common due date d	Weight capacity W
Question j	item i
Time p_j	Weight w_i
Points ω_j	Value v_i

Table 2.6: Identification of parameters in Example 2.1 and Knapsack.

Optimal Offline Algorithms for $\mathbf{1}|d_j|\sum_{j\in J} u_j$

The optimal algorithm for $\mathbf{1}|d_j|\sum_{j\in J} u_j$ is a variation of a list algorithm EDD.

Definition 2.20. *Let **Earliest Due Date first** (short:* EDD*) denote the algorithm which always assigns the job j with the minimal d_j among all available jobs to a machine m as soon as m is idle.*

```
Input: A set $\{j_1, j_2, \dots, j_n\}$ of jobs sorted by EDD.
Output: A set $J_n \subseteq \{j_1, j_2, \dots, j_n\}$.
1  $J = \emptyset$, $J^c = \{j_1, j_2, \dots, j_n\}$ and $J^d = \emptyset$;
2  for $k = 1 : n$ do
3      Add $j_k$ to $J$;
4      Delete $j_k$ from $J^c$;
5      if $\sum_{j \in J} p_j > d_k$ then
6          Add $l$ to $J^d$ where $p_l = \max_{j \in J}\{p_j\}$;
7          Delete $l$ from $J$;
8      end
9      Set $J_k = J$;
10 end
11 Return $J_n$.
```

Algorithm 1: Algorithm for Problem 18.

In words the algorithm can be described as follows: After each iteration, set J presents a set of jobs which can be finished in time by executing them in an EDD-fashion, set J^c presents the jobs which are not considered yet and set J^d contains the jobs each of which will fail to meet its due date in the final schedule. The jobs are added to J in an EDD-manner. If including job j implies that it cannot be finished in time (Line 5), then the job with the longest processing time in J will be discarded.

The output J_n itself is a subset of the given job set. The corresponding schedule assigns jobs in J_n to the machine in an EDD-manner.

Lemma 2.21 (Moore,1968)**.** *Problem* 18*:* $\mathit{1}|d_j|\sum_{j \in J} u_j$ *can be solved optimally by using Algorithm 1.*

For the sake of clarity, the proof is divided into two lemmas following Pinedo (2012). We introduce the following terminologies during this proof: Let $N = \{j_1, \dots, j_n\}$ denote the job set. A subset $S \subseteq N$ is *feasible* if the schedule, which assigns $j \in S$ in an EDD-manner to the machine, is able to finish all $j \in S$ in time. A feasible set is *optimal* in N, if its cardinality is maximal among all feasible subset of N.

Lemma 2.22. *For all* $k = 1, \dots, n$ *holds that* J_k *from Algorithm 1 is feasible.*

Proof. Observe that, as long as the if-check in Line 5 is false uninterruptedly (before the first true occurs), jobs in J can be finished in time in an EDD-manner. Consider the case where the if-check is true for the first time, i.e. $\sum_{j \in J_{k-1}+j_k} p_j > d_k$ for the k-th iteration. Let j_l denote the job being deleted from J at Line 7. Essentially, it is enough to show that

$$\sum_{j \in J'} p_j \leq d_k$$

where $J' = J_{k-1} + j_k - j_l$. This inequality is a conclusion from the definition of p_l, minimality of k and the fact that jobs are sorted by EDD:

$$\begin{aligned}\sum_{j\in J'} p_j &= p_k - p_l + \sum_{j\in J_{k-1}} p_j \\ &\le \sum_{j\in J_{k-1}} p_j && [p_l \text{ is maximal in } J] \\ &\le d_{k-1} && [k \text{ is chosen minimal}] \\ &\le d_k && [\text{Jobs are sorted EDD}] \qquad \square\end{aligned}$$

Lemma 2.23. *After the k-th iteration, there exists an optimal subset $S \subseteq N$ avoiding all jobs being deleted by Algorithm 1 sofar.*

Proof. We apply induction to prove this lemma. The claim is trivial for $k = 1$. Assume it is true for $k-1$, i.e. there exists an optimal $J' \subseteq J_{k-1} \cup \{j_k, j_{k+1}, \dots, j_n\}$, We may further assume that a job $q \in J_{k-1}$ is deleted during the k-th iteration which is contained in J', for otherwise the set J' itself fulfills all requirements to complete the induction.

The fact "q is deleted" implies that the set $J_{k-1} + j_k$ is not feasible, thus any feasible set is not able to comprise $J_{k-1} + j_k$. In particular, we find $r \in J_{k-1} + j_k$ with $r \notin J'$. Consider the set $J'' = J' + r - q$. Certainly, J'' does not include any deleted jobs deleted in the first k-iterations. As its cardinality is equal to another optimal set J', it only remains to show that J'' is feasible.

Notice that $J' \cap \{j_k, j_{k+1}, \dots, j_n\} = J'' \cap \{j_k, j_{k+1}, \dots, j_n\}$, thus it suffices to show $J'' \cap \{j_1, \dots, j_{k-1}\}$ is feasible and to finish them consumes less time in total than to finish jobs in $J' \cap \{j_1, \dots, j_{k-1}\}$.

The feasibility is a direct consequence of $J'' \cap \{j_1, \dots, j_{k-1}\} \subseteq J_{k-1}$. For the latter condition note that job q is deleted in the k-th iteration, its processing time p_q must be greater than or equal to all jobs in $J_{k-1} + j_k$. In particular, we have $p_r \le p_q$. Hence we have

$$\begin{aligned}\sum_{j\in J''\cap\{j_1,\dots,j_{k-1}\}} p_j &= \sum_{j\in (J'+r-q)\cap\{j_1,\dots,j_{k-1}\}} p_j \\ &= p_r - p_q + \sum_{j\in J'\cap\{j_1,\dots,j_{k-1}\}} p_j \\ &\le \sum_{j\in J'\cap\{j_1,\dots,j_{k-1}\}} p_j. \qquad \square\end{aligned}$$

In other words, Algorithm 1 produces a feasible subset of the first k jobs after the k-th iteration, which always can be extended into an optimal set. Therefore, the optimality of J_n is a trivial consequence from this two properties.

2.3 Competitive Result in the Online Environment

Notice that all algorithms we introduced in the previous sections assume that the information (e.g. number of jobs, their attributes r_j, p_j, d_j, ω_j and so on) about the problem are completely known in advance. In an online environment, the information about job j are (partially) revealed to the decision maker only when j is released.

Definition 2.24 (Pinedo,2012). *In an* ***online scheduling problem****, the following restrictions apply additionally:*

- *The decision maker becomes aware of the existence of a job only when the job is released and presented to him,*
- *jobs released at the same point in time t are presented one after each other, the number of jobs released at t is known to the decision maker only after the last one of them has been presented,*
- *the processing time of a job becomes known only when the job has been finished.*

In the case that individual weight ω_j or release time r_j are involved in the problem,

- *the weight of a job becomes revealed when the job is presented to the decision maker, whereas*
- *the number of not yet released jobs is unknown to him at any point in time.*

Example 2.25. *Recall* SPT*-schedule presented in Figure 2.5a with the attributes in Table 2.4. The knowledge revealed to the decision maker along the time can be visualized as shown in Figure 2.10.*

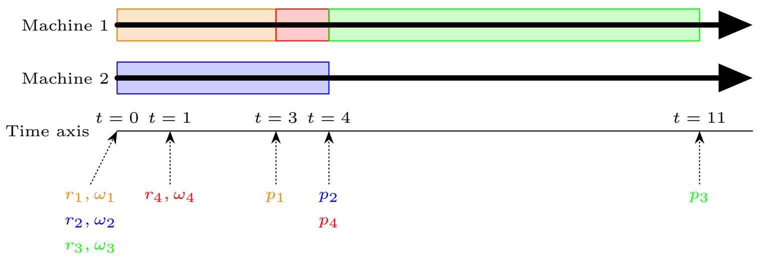


Figure 2.10: Revealed knowledge about jobs over time.

Due to this lack of information, the cost of an offline optimal algorithm OPT is not achievable by an online algorithm ALG in most of the cases. Instead of asking if an algorithm is optimal, one compares the cost of ALG with the cost of OPT (as defined in Definition 1.1).

In competitive analysis, an algorithm ALG performs not well if it is not competitive at all, i.e. for any c there exists an instance Ins_c such that $C_{\text{ALG}}(\text{Ins}_c)$ is greater than $c \cdot C_{\text{OPT}}$.

Consider Example 2.1 and the related Problem 19: $\mathbf{1}|d_j = d|\sum_{j\in J}\omega_j u_j$. Recall from Lemma 2.19 that knapsack problem can be reduced to Problem 19. In the online version of Problem 19, the questions are presented to the student one after each other and the student must decide if he is going to work on the currently presented question before another question is revealed to him. This online environment can be extended to the optimization version of knapsack problem too. In Marchetti-Spaccamela and Vercellis (1995), the authors have shown that no deterministic online algorithm is competitive for the online knapsack problem. Their proof can be used directly to show the same result for Problem 19.

Lemma 2.26 (Marchetti-Spaccamela and Vercellis,1995). *Given an arbitrary deterministic online algorithm* ALG *of Problem* 19 *and any positive integer n, there exists an instance* Ins *such that* $C_{\text{ALG}}(\text{Ins}) \le n \cdot C_{\text{OPT}}(\text{Ins})$

Proof. Consider the following two instances of Problem 19:

- Ins_1 consists of only one job j with processing time $p_j = 1$, weight $\omega_j = 1$ and the common due date is $d = 1$,
- Ins_2 consists of two jobs j, j' with equal processing time $p_j = p_{j'} = 1$, different weights $\omega_j = 1, \omega_{j'} = n$ and the same common due date $d = 1$.

Obviously, OPT will finish j if Ins_1 is presented and j' if Ins_2 is presented. Recall the objective function is $\sum_{j\in J}\omega_j u_j$, whence we have $C_{\text{OPT}}(\text{Ins}_1) = 0$ and $C_{\text{OPT}}(\text{Ins}_2) = 1$.

Due to the restriction of online environment, ALG is not able to differentiate Ins_1 and Ins_2 at the point in time when j is presented. Since ALG is deterministic, its decision whether to process or refuse j while solving Ins_1 must be the same as its decision while solving Ins_2.

In the case that ALG decides to process j, it is easy to see that ALG is not able to process j' in Ins_2, whence $C_{\text{ALG}}(\text{Ins}_2) = n$ and we deduce that ALG is no better than n-competitive. If ALG refuses to process j, then a unit penalty $\omega_j u_j$ is incurred in Ins_1. Notice that OPT is able to achieve an objective value of 0 in this case. Thus, the cost $C_{\text{ALG}}(\text{Ins}_1)$ cannot be bounded by $c \cdot C_{\text{OPT}}(\text{Ins}_1)$ for any c.

To summarize, no matter which decision is made by ALG after j is presented, there exists an instance to assure that ALG is no better than n-competitive. □

While negative results can be proved by providing critical instances, the positive results require certain knowledge about the optimal offline algorithms. Consider Problem 9: $Pm|\,\text{prmp}\,|\sum_{j\in J} c_j$. The proof of Lemma 2.17 can be adjusted to show that Problem 9 can be solved optimally using SRPT-rule. Since, in the online version of Problem 9, the processing time is not known in advance, this list scheduling algorithm is not applicable. However, it is possible to imitate the idea of SRPT by applying preemptions frequently.

Definition 2.27. *Let **Round Robin** (short:* RR*) denote the algorithm which cycles through the list of presented jobs, giving each job a fixed unit of processing time in turn.*

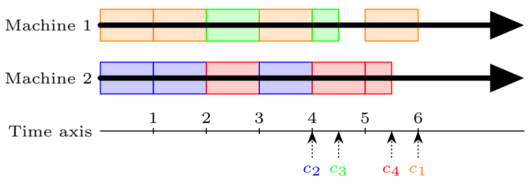
Example 2.28. *To clarify* RR *precisely, we involve the release time environment for this example. Consider the problem* $P2|\,\mathrm{prmp}, r_j|\sum_{j\in J} c_j$ *with four jobs, see Table 2.7 for their attributes.* RR *maintains a list of presented jobs (Table 2.8) and distributes the processing time among these jobs as shown in Figure 2.11a.*

Job	Release time	Processing time
1	0	4
2	0	3
3	1.5	1.5
4	2	2.5

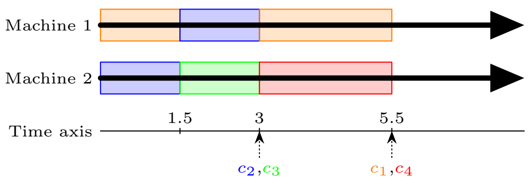
Table 2.7: Jobs in Example 2.28.

Point in time	List of presented jobs	
	before assignment	after assignment
0	(1,2)	()
1	(1,2)	()
1.5	(3)	(3)
2	(3,4,1,2)	(1,2)
3	(1,2,3,4)	(3,4)
4	(3,4,1)	(1)
5	(1,4)	()

Table 2.8: List of presented jobs over time.



(a) RR Schedule.



(b) SRPT Schedule.

Figure 2.11: The schedules in Example 2.28.

Remark 2.29. *In words the rule of how* RR *maintains the list of presented jobs can be summarized as follows:*

Once a job is released, it is appended immediately to the list. At the begin of each time unit, RR *stops all machines and appends the interrupted jobs to the end of the list. Thereafter, the first* 2 *jobs on the list are assigned to machines.*

Notice that for a set of jobs with the same release time, RR *ensures that at all times any two uncompleted jobs have received an equal amount of processing time or one job has* *received just one time unit more than the other. Roughly speaking,* RR *tries to detect the job with shortest (remaining) processing time by equally distributing the processing time among all presented jobs, while* SRPT *focuses on finishing jobs with shortest remaining processing time as early as possible. If the time unit is small in comparison to the processing times of the jobs, it is easy to see that* RR *finishes the jobs in the same order as* SRPT *apart from a permutation of jobs with identical processing times.*

Lemma 2.30 (Pinedo,2012)**.** *If the time unit is small in comparison to the processing times of the jobs, i.e.* $\min_{j\in J} p_j \gg 1$, RR *is* 2*-competitive for Problem* $Pm|\,\mathrm{prmp}\,|\sum_{j\in J} c_j$.

Proof. We show that RR is no better than 2-competitive by providing a critical instance. To hit RR as hard as possible, consider a set J of nm jobs, each of which has an identical integer processing time p. For the sake of clarity, let $c_{j,\mathrm{ALG}}$ denote the completion time of job j according to the algorithm ALG.

It is easy to see that SPT finishes m jobs at each point in time of $p, 2p, \ldots, np$, thus the objective value is equal to

$$\sum_{j\in J} c_{j,\mathrm{SPT}} = \sum_{k=1}^{n} mkp = \frac{n(n+1)}{2}mp.$$

In comparison to this, RR partitions every job into p parts and assigns the k-th part of a job to a machine only if the $(k-1)$-th part of all jobs are finished. We deduce that before the $(p-1)$-th parts of all jobs are finished, RR is not able to completely finish any job, meaning that $c_{j,\mathrm{RR}} > \frac{nm(p-1)}{m} = n(p-1)$, thus the objective value of RR can be bounded by

$$\sum_{j\in J} c_{j,\mathrm{RR}} > n(p-1)\cdot nm = n^2m(p-1).$$

Hence we conclude that RR is no better than 2-competitive.

To show that RR is no worse than 2-competitive consider any given instance with job set $J = \{j_1, \ldots, j_n\}$. Assume that the processing time of these jobs $j \in J$ satisfies $p_1 \geq p_2 \geq \cdots \geq p_n$ and partition the job set J according to the index of j as follows:

$$J = \bigcup_{l=1}^{\lceil \frac{n}{m} \rceil} R(l) = \bigcup_{l=1}^{\lceil \frac{n}{m} \rceil} \{p_k \,|\, (l-1)m < k \leq lm\}.$$

In words, $R(1)$ contains $j_1, \ldots, j_m$ (the first m longest jobs), $R(2)$ contains jobs $j_{m+1}, \ldots, j_{2m}$ (the second m longest jobs), and so on. Hence, recall the proof of Lemma 2.17, $R(1)$ is the set of the last job assigned to each machine by SPT, $R(2)$ is the set of the second last job assigned to each machine by SPT, and so on, see Figure 2.12 for an illustration.

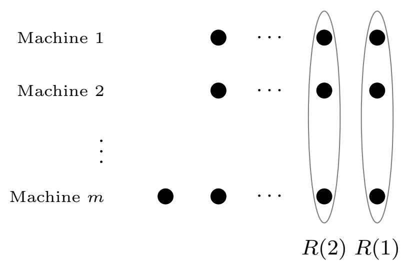


Figure 2.12: Illustration of $R(l)$.

Thus, one can rewrite Equation (2.4) to obtain

$$\sum_{j\in J} c_{j,\mathrm{SPT}} = \sum_{l=1}^{\lceil \frac{n}{m} \rceil} \sum_{j\in R(l)} l p_j. \tag{2.5}$$

Consider now the objective value $\sum_{k=1}^{n} c_{k,\mathrm{RR}}$ of RR. Since the time unit of RR is small, the jobs in J are completed in the reverse order of their processing time. In particular, at the point in time $c_{m+1,\mathrm{RR}}$, all jobs $j_{m+1}, j_{m+2}, \ldots, j_n$ are already finished. It only remains to process j_k with a remaining processing time of $p_k - p_{m+1}$ for $k = 1, \ldots, m$. Certainly, as we have m machines for m jobs, further preemptions are not necessary. It is easy to see that

$$c_{k,\mathrm{RR}} = c_{m+1,\mathrm{RR}} + p_k - p_{m+1} \tag{2.6}$$

for $k = 1, \ldots, m$.

Similarly, a recursive formula of $c_{k,\mathrm{RR}}$ can be deduced:

$$c_{k,\mathrm{RR}} - c_{k+1,\mathrm{RR}} = \frac{k}{m}(p_k - p_{k+1}) \tag{2.7}$$

for $m < k < n$ and

$$c_{n,\mathrm{RR}} = \frac{n}{m} p_n \tag{2.8}$$

for $k = n$. To see Equation (2.7), observe the point in time $c_{k+1,\mathrm{RR}}$ when j_{m+1} is finished. Since j_k has a remaining processing time of $p_k - p_{k+1}$, we know that, until its completion time $c_{k,\mathrm{RR}}$, every job receives a period of processing time equal to this amount. As there are still k uncompleted jobs, we have to distribute a total processing time of $k(p_k - p_{k+1})$ to m machines. Thus, (2.7) holds. Equation (2.8) can be established by the same argument. Eliminating the recurrence yields for $k > m$

$$c_{k,\mathrm{RR}} \overset{(2.7),(2.8)}{=} \frac{k}{m} p_k + \frac{1}{m} \sum_{l=k+1}^{n} p_l. \tag{2.9}$$

Hence we have

$$\begin{aligned} c_{k,\mathrm{RR}} &\overset{(2.6)}{=} c_{m+1,\mathrm{RR}} + p_k - p_{m+1} \\ &\overset{(2.9)}{=} \frac{m+1}{m} p_{m+1} + \left(\frac{1}{m} \sum_{l=m+2}^{n} p_l \right) + p_k - p_{m+1} \\ &= p_k + \frac{1}{m} \sum_{l=m+1}^{n} p_l \end{aligned} \tag{2.10}$$

for $k \leq m$. By Equations (2.9) and (2.10), the objective value of RR is equal to

$$\sum_{j \in J} c_{j,\mathrm{RR}} = \sum_{k=1}^{m} p_k + \sum_{k=m+1}^{n} \frac{2k-1}{m} p_k. \tag{2.11}$$

Thus, the case $n \leq m$ is not interesting as the objective value of RR is equal to the objective value of SPT. To find the worst case, we assume $n > m$.

Consider the job $j_k \in R(l)$ for some $l > 1$. By the definition of $R(l)$ we know that $k \leq lm$, leading to the inequality

$$\frac{2k-1}{m} p_k < 2lp_k.$$

Hence the summation $\sum_{k=m+1}^{n} \frac{2k-1}{m} p_k$ can be bounded by

$$\sum_{k=m+1}^{n} \frac{2k-1}{m} p_k < \sum_{l=2}^{\lceil \frac{n}{m} \rceil} \sum_{j \in R(l)} 2lp_j. \tag{2.12}$$

Therefore, we conclude

$$\begin{aligned} \frac{\sum_{j \in J} c_{j,\mathrm{RR}}}{\sum_{j \in J} c_{j,\mathrm{SPT}}} &\overset{(2.5),(2.11)}{=} \frac{\sum_{k=1}^{m} p_k + \sum_{k=m+1}^{n} \frac{2k-1}{m} p_k}{\sum_{l=1}^{\lceil \frac{n}{m} \rceil} \sum_{j \in R(l)} lp_j} \\ &\overset{(2.12)}{<} \frac{\sum_{j \in R(1)} p_j + \sum_{l=2}^{\lceil \frac{n}{m} \rceil} \sum_{j \in R(l)} 2lp_j}{\sum_{l=1}^{\lceil \frac{n}{m} \rceil} \sum_{j \in R(l)} lp_j} \\ &< 2. \end{aligned}$$

□

Fortunately, the offline optimal algorithm for Problem 9 is known. In most cases, the cost of the optimal algorithm can only be bounded by another known algorithm or by a relaxation of the original problem. This issue will be discussed and treated in the next chapter.

2.4 Computer Experiments

In this section, we want to study and compare, empirically, the performance of the algorithms descripted in this chapter. Besides RR, EDD and the variations of SPT, we also include the two commonly used heuristics FIFO and RO, which serve only for comparison.

Definition 2.31. *Let **First In First Out** (short:* FIFO*) denote the algorithm which always assigns the job j with the minimal r_j among all available jobs to a machine m as soon as m is idle.*

*Let **Random Order** (short:* RO*) denote the algorithm which chooses uniformly one job j from the set of all currently available jobs and assigns j to a machine m as soon as m is idle.*

Computer codes in Julia (version 1.0.2) are used to simulate the scheduling process and to visualize this comparison. The framework is summarized in Algorithm 2. More precisely, J presents a set of jobs j with release time $j.r$, processing time $j.p$, remaining processing time $j.rp$, due date $j.d$, weight $j.\omega$, and completion time $j.c$ (which is equal to 0 initially). The number of available machines is denoted by m and the limit T is a trivial bound of time unit needed for solving this instance.

```
Input: $J$, $m$, $T$, and the applied algorithm ALG.
Output: The completion time $j.c$ for $j \in J$.
1  $J_1 = J$; $J_2 = J_3 = J_4 = \emptyset$; Temp $= \emptyset$; $L_{\text{RR}} =$ empty list;
2  $t = 0$;
3  for $j \in J$ do
4      $j.c = 0$;
5  end
6  while $t < T$ and $|J_4| < |J|$ do
7      Temp $= \{j \in J_1 \mid j.r == t\}$;
8      $J_1 = J_1 \setminus$ Temp; $J_2 = J_2 \cup$ Temp;
9      Temp $= \{j \in J_2 \mid j.d < t\}$;
10     $J_2 = J_2 \setminus$ Temp;
11     ALG reorganizes the job to machine assignment if necessary;
12     $t = t + 1$;
13     for $j \in J_3$ do
14         $j.rp = j.rp - 1$;
15     end
16     Temp $= \{j \in J_3 \mid j.rp == 0\}$;
17     $J_3 = J_3 \setminus$ Temp; $J_4 = J_4 \cup$ Temp;
18     for $j \in$ Temp do
19         $j.c = t$;
20     end
21 end
22 return $j.c$ for $j \in J$.
```

Algorithm 2: Simulation of scheduling

In our case,

$$T = \begin{cases} \max_{j\in J}\{j.d\}, & \text{if due date is present,} \\ \max_{j\in J}\{j.r\} + \sum_{j\in J} j.p, & \text{otherwise.} \end{cases}$$

The simulation returns the completion time c_j of every job in J according to ALG. For the sake of simplicity, we assume that all attributes of jobs are given in integer value. At any point in time t, J is partitioned into four disjoint subsets:

- the set $J_1(t)$ of unreleased jobs,
- the set $J_2(t)$ of available jobs,
- the set $J_3(t)$ of jobs which have been processed by some machine, and
- the set $J_4(t)$ of finished jobs.

In unambiguous cases, the dependence on t is omitted.

In words, Temp in Line 7 consists of jobs each of which is released at the beginning of this time unit, the code at Line 8 moves them from the set of unreleased jobs to the set of available jobs. Temp in Line 7 consists of jobs whose due date are not met, the code at Line 10 removes them from the set of available jobs. Thereafter, ALG considers all available information and reorganizes the job to machine assignment. During this time unit, the remaining processing time *j.rp* of every assigned job j is decreased by 1. At the end of a time unit, jobs are marked as "finished" if the remaining processing time has reached 0 during this time unit. These jobs are removed from the machine (Line 17), and their completion times are recorded.

Once the completion time of all jobs are computed, it is straightforward to evaluate the objective value. The corresponding codes are omitted.

2.4.1 The Implementation of Algorithms for Scheduling Problems

The algorithms SPT, WSPT, SRPT, EDD, RR, and FIFO, RO are considered in this section. At any point in time, each of them maintains a priority list which is a permutation of J_2. The main difference between these algorithms is when and how to update their priority lists, see Table 2.9 for a summary.

In the offline environment, ALG has access to the complete input and all variables generated during the process (J_1, J_2, J_3, J_4, Temp and t). If online environment is present, J and J_1 are not revealed to ALG. Furthermore, recall from Definition 2.24 that *j.p* and *j.rp* are not known for $j \in J_2 \dot\cup J_3$. Since the first three algorithms SPT, WSPT, and SRPT require *j.p* or *j.rp* to determine their priority list, they are only applicable in an offline environment. The other algorithms can be implemented in offline or online environment.

ALG	When to update	Rule
SPT	$\lvert J_2\rvert > 0 \wedge \lvert J_3\rvert < m$	$j.p$
WSPT	$\lvert J_2\rvert > 0 \wedge \lvert J_3\rvert < m$	$j.p/j.\omega$
SRPT	$(\lvert J_2\rvert > 0 \wedge \lvert J_3\rvert < m)$ or ($t = j.r$ for some $j \in J$)	$j.rp$
RR	At the begin of every time unit	Example 2.28
EDD	$\lvert J_2\rvert > 0 \wedge \lvert J_3\rvert < m$	$j.d$
FIFO	$\lvert J_2\rvert > 0 \wedge \lvert J_3\rvert < m$	$j.r$
RO	$\lvert J_2\rvert > 0 \wedge \lvert J_3\rvert < m$	random

Table 2.9: Timing and rule of updating priority list.

Offline algorithms SPT, WSPT, **and** SRPT

```
Input: The set $J_2$ of available jobs, the set $J_3$ of jobs which are in progressing,
       and the number of machines $m$.
Output: $J_2$ and $J_3$ after SPT rearranges the job to machine assigment.
1  while $|J_2| > 0$ and $|J_3| < m$ do
2      $j = \text{argmin}_{j \in J_2} j.p$;
3      $J_2 = J_2 - j$; $J_3 = J_3 + j$;
4  end
5  return $J_2, J_3$.
```

Algorithm 3: SPT.

SPT moves one job with the shortest processing time from J_2 to J_3 as long as there are still available jobs ($|J_2| > 0$) and available machines ($|J_3| < m$), see Algorithm 3. The code of WSPT can be obtained by replacing the $j.p$ in Line 2 with $\frac{j.p}{j.\omega}$.

```
Input: The set $J_2$ of available jobs, the set $J_3$ of jobs which are in progressing,
       and the number of machines $m$.
Output: $J_2$ and $J_3$ after SRPT rearranges the job to machine assigment.
1  if $t == j.r$ for some $j \in J_2$ then
2      $J_2 = J_2 \cup J_3$; $J_3 = \emptyset$;
3      while $|J_2| > 0$ and $|J_3| < m$ do
4          $j = \text{argmin}_{j \in J_2} j.rp$;
5          $J_2 = J_2 - j$; $J_3 = J_3 + j$;
6      end
7  else
8      while $|J_2| > 0$ and $|J_3| < m$ do
9          $j = \text{argmin}_{j \in J_2} j.rp$;
10         $J_2 = J_2 - j$; $J_3 = J_3 + j$;
11     end
12 end
13 return $J_2, J_3$.
```

Algorithm 4: SRPT.

The advantage of preemption allows SRPT to change the job to machine assignment more often than SPT and WSPT. Theoretically, SRPT can check the remaining processing time $j.rp$ for $j \in J_2 \dot{\cup} J_3$ at any point in time and update its priority list accordingly. However, it is easy to see that a check between two consecutive release times does not change the priority list. Thus, it suffices to update the priority list of SRPT when some job is released ($t = j.r$) or there are available jobs and idle machines.

Online algorithms RR, EDD, FIFO**, and** RO

The code of EDD, FIFO, and RO can be obtained from the code of SPT by replacing $\text{argmin}_{j\in J_2} j.p$ in Line 2 in Algorithm 3 with, respectively, $\text{argmin}_{j\in J_2} j.d$, $\text{argmin}_{j\in J_2} j.r$ and a random job from J_2. RR is slightly more complicated. In order to share the processing time fairly, RR should give higher priority to those jobs which are either newly released or which have not been processed since the begin of the last time unit. For this purpose, RR maintains a list L_{RR} which is empty initially and gets updated in each iteration, see Algorithm 5 for the rule of updating L_{RR}.

```
Input: The priortiy list $L_{\text{RR}}$, the current time $t$, the number of machines $m$,
the set $J_2$ of available jobs, and the set $J_3$ of jobs which are in progressing.
Output: Updated $L_{\text{RR}}, J_2, J_3$ after RR rearranges the job to machine
        assigment.
1  for $j \in J_2$ do
2      if $t == j.r$ then
3          append!($L_{\text{RR}}, j$);
4      end
5  end
6  for $j \in J_3$ do
7      append!($L_{\text{RR}}, j$);
8  end
9  $J_2 = J_2 \cup J_3$; $J_3 = \emptyset$;
10 while $|J_2| > 0$ and $|J_3| < m$ do
11     $j$ = popfirst!($L_{\text{RR}}$);
12     $J_2 = J_2 - j$; $J_3 = J_3 + j$;
13 end
14 return $L_{\text{RR}}, J_2, J_3$.
```

Algorithm 5: RR.

L_{RR} can be partitioned into three consecutive sub-sequences: The first one consists of the jobs which are already released before the last time unit and were not assigned to any machine at the beginning of the last time unit (Input L_{RR}). The second one consists of the jobs, which are released at the beginning of this time unit (for loop at Line 2). The last one consists of the jobs, each of which is assigned to a machine during the last time unit and interrupted at the beginning of this time unit (for loop at Line 6).

After updating L_{RR}, RR assigns the first m jobs in L_{RR} to machines (while loop at Line 10). Certainly, jobs, which were processed recently, are not preferred in comparison to other jobs in L_{RR}.

2.4.2 Online Experiments

In this subsection, we compare the performance of RR with SRPT. Notice that RR is designed to detect short jobs and finish them early. In other words, RR should perform well if the processing times of jobs vary considerably. To support this claim, we apply χ^2-distributions and uniform distribution to generate four sets of instances.

More precisely, 200 instances are generated for each set, where each instance consists of 50 jobs. Their release times are uniformly distributed over the first 50 time units (including 0). A constant p is predefined as the expected processing time.

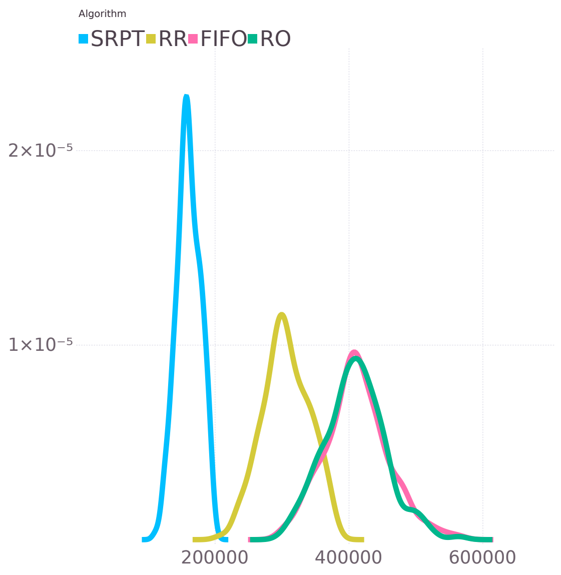
In the first three sets, the χ^2-distributions with degree $i = 1, 2, 3$ are applied, respectively, to generate p_j: for each j and $i = 1, 2, 3$ let x_j be a realization of χ_i^2, then p_j is defined to be $x_j \cdot p/i$ (such that the expected processing times are independent on the degree of χ^2 distributions). In the last set, the processing times of jobs are chosen uniformly from $\{1, \ldots, 2p - 1\}$.

Certainly, χ_1^2 is going to generate more jobs with short processing times than other distributions. Thus, we expect that RR performs well on the set where the processing times of jobs in that set was generated by χ_1^2.

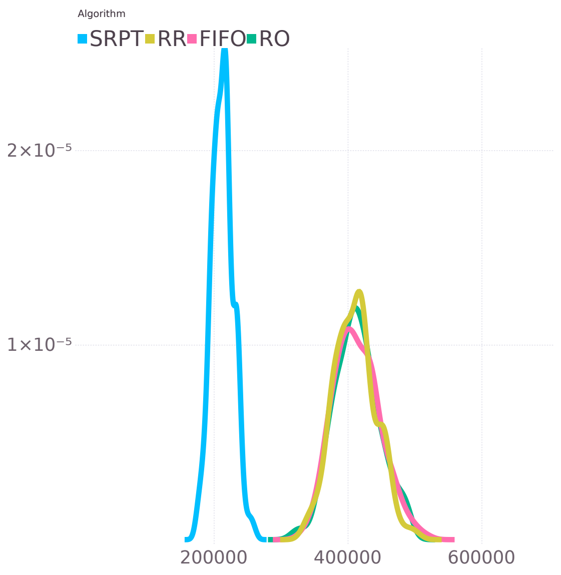
First consider Problem 2: $\mathbf{1}|r_j, \mathrm{prmp}\,|\sum_{j\in J} c_j$. By Lemma 2.18, SRPT is optimal for the offline variant. The experiments reveal that the quality of schedules produced by RR depends considerably on the applied distribution. Figures 2.13 and 2.14 present the density graph of cost and competitive ratio of RR in comparison to other algorithms. In all experiments, the competitive ratio of RR is pretty stable and near 2, this observation supports Lemma 2.30. RR benefits from these short jobs and out-performs FIFO and RO when χ_1^2 is applied.

Similar results are obtained in Pm environment, e.g. Figures 2.15 and 2.16 present the performance of RR in $P4$ environment where all other attributes of jobs are generated in the same way as before. Notice that SRPT is no longer optimal, thus Figure 2.16 serves only as a comparison between RR and SRPT, instead of a competitive ratio as defined in Definition 1.1.

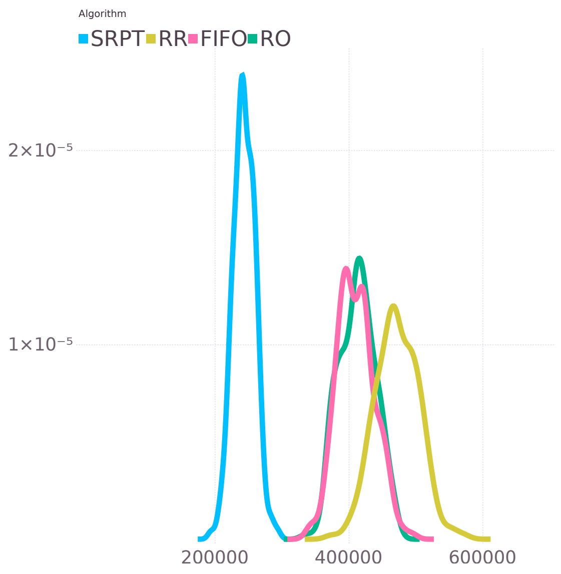
We observe that the quality of schedules produced by RR decreases from the first to the last set. Therefore, we may conclude that RR performs much better on instances where the processing times of jobs vary considerably.



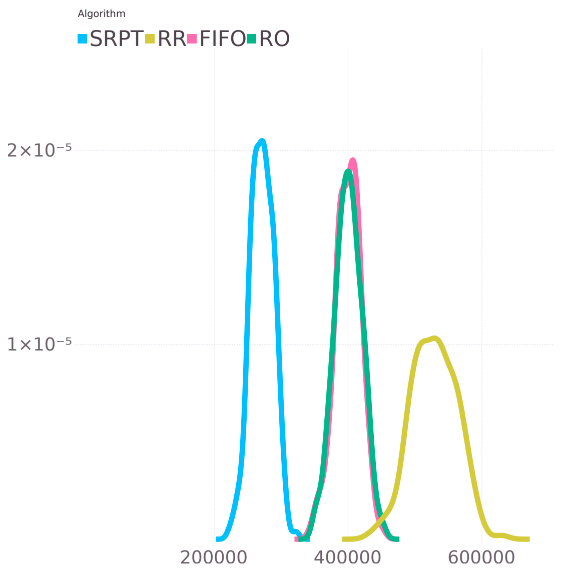
(a) p_j distributed according to χ^2_1.



(b) p_j distributed according to χ^2_2.

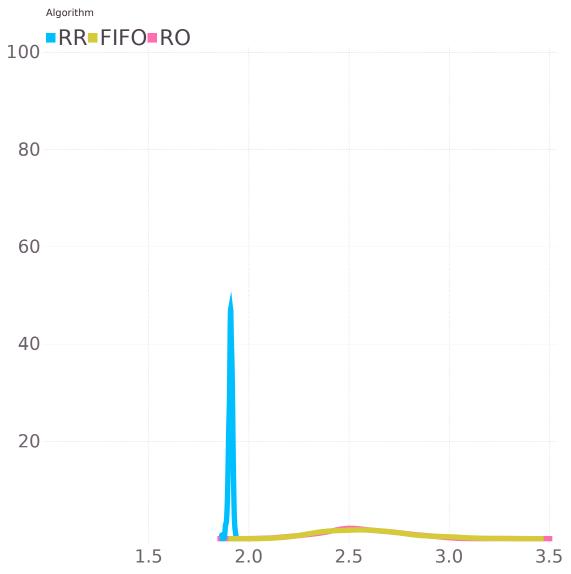


(c) p_j distributed according to χ^2_3.

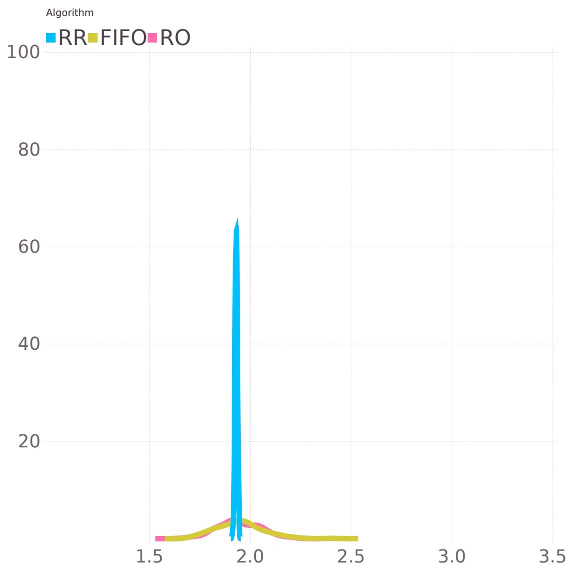


(d) p_j distributed uniformly.

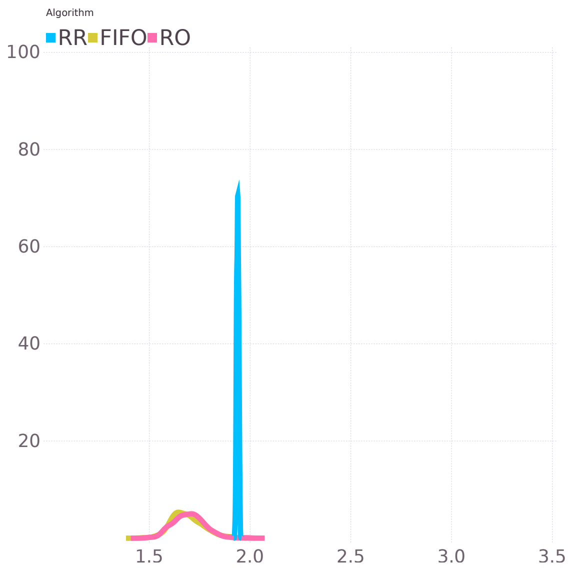
Figure 2.13: Performance of several algorithms in single machine online environment.



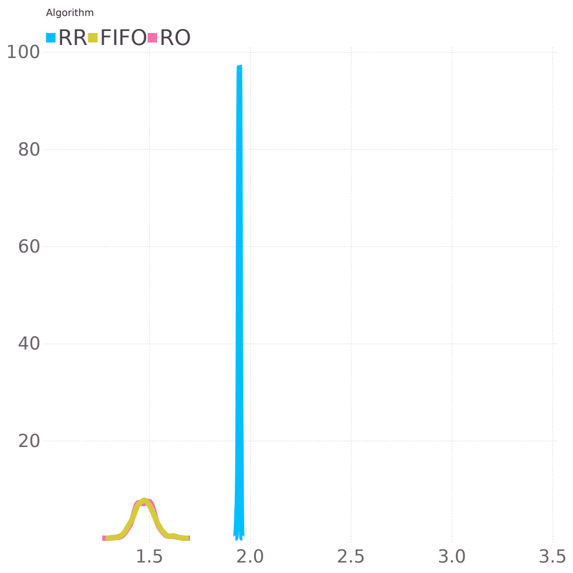
(a) p_j distributed according to χ_1^2.



(b) p_j distributed according to χ_2^2.

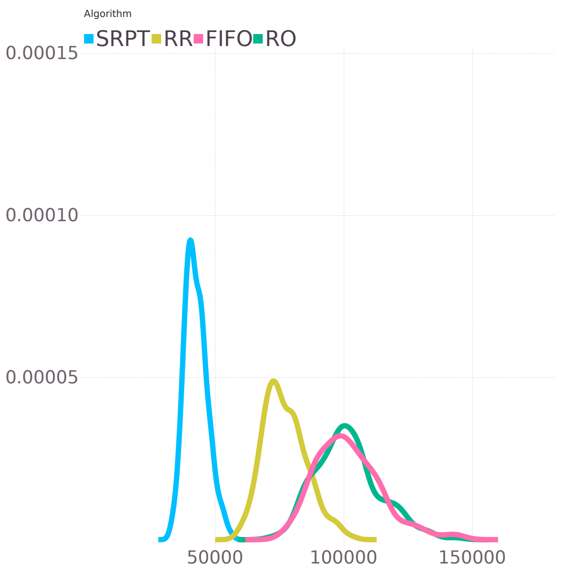


(c) p_j distributed according to χ_3^2.

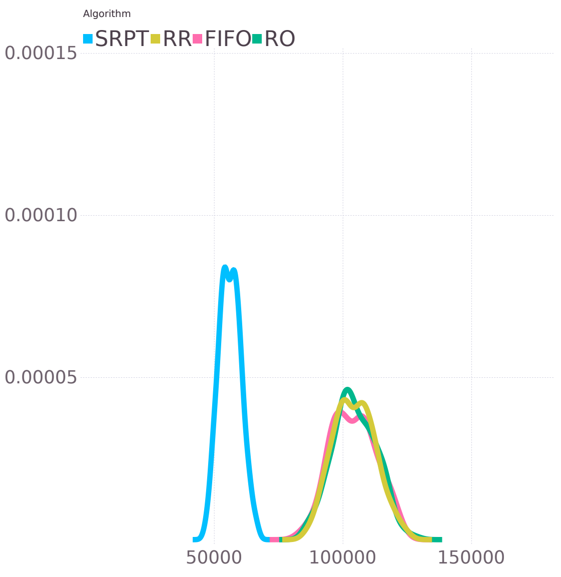


(d) p_j distributed uniformly.

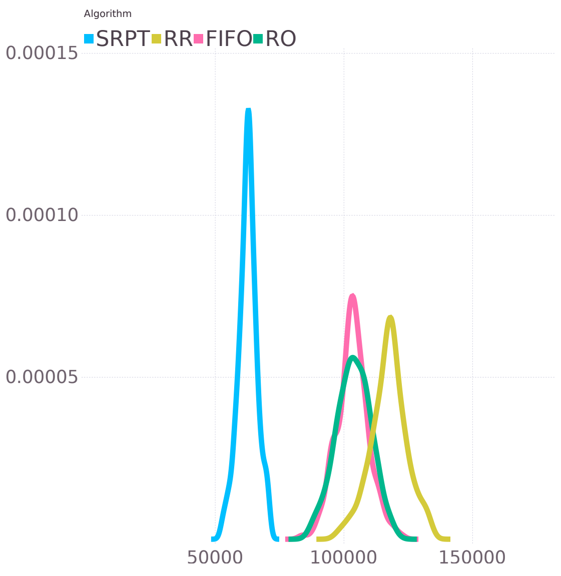
Figure 2.14: Competitive ratio of several algorithms in single machine online environment.



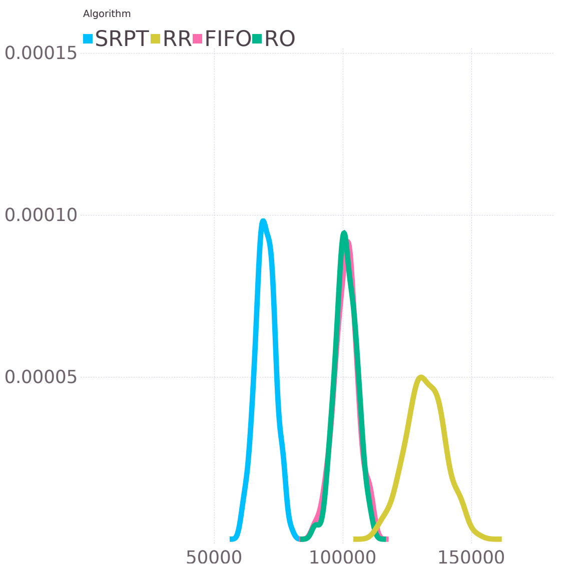
(a) p_j distributed according to χ_1^2.



(b) p_j distributed according to χ_2^2.

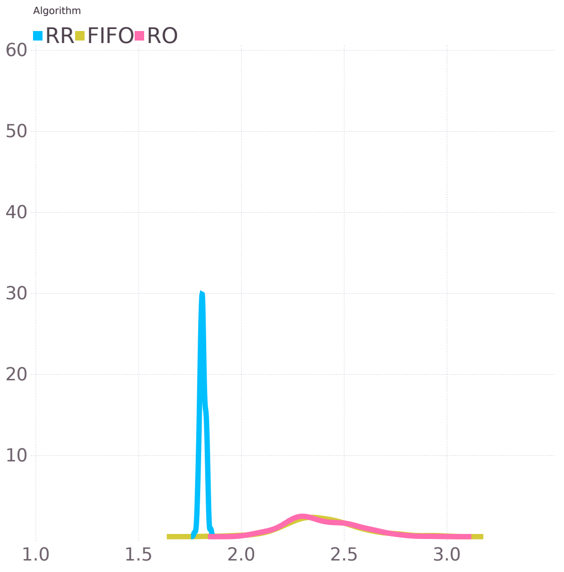


(c) p_j distributed according to χ_3^2.

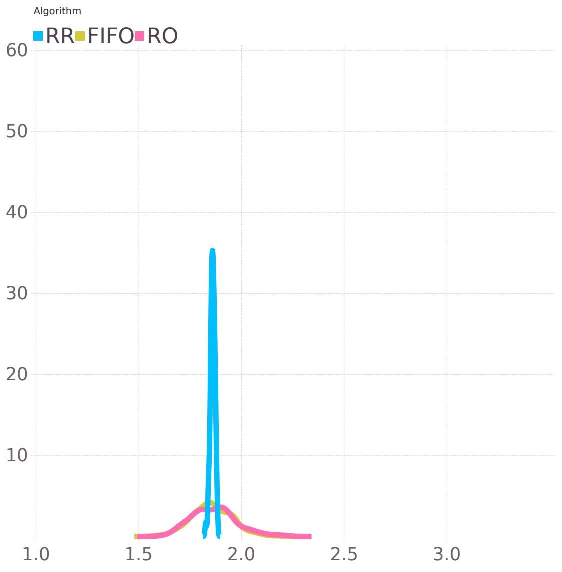


(d) p_j distributed uniformly.

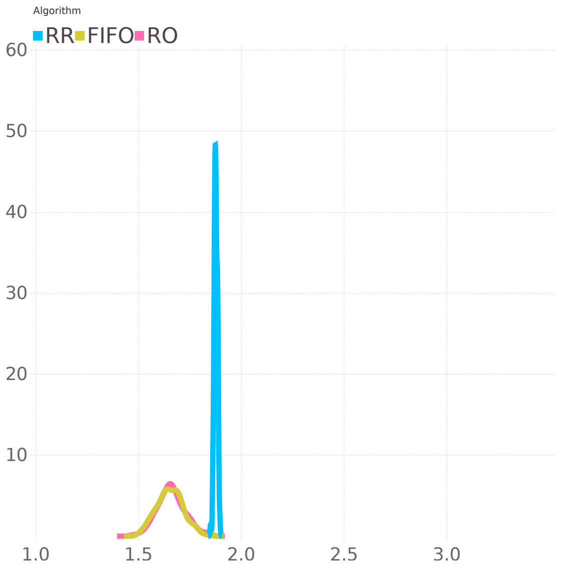
Figure 2.15: Performance of several algorithms in $P4$ online environment.



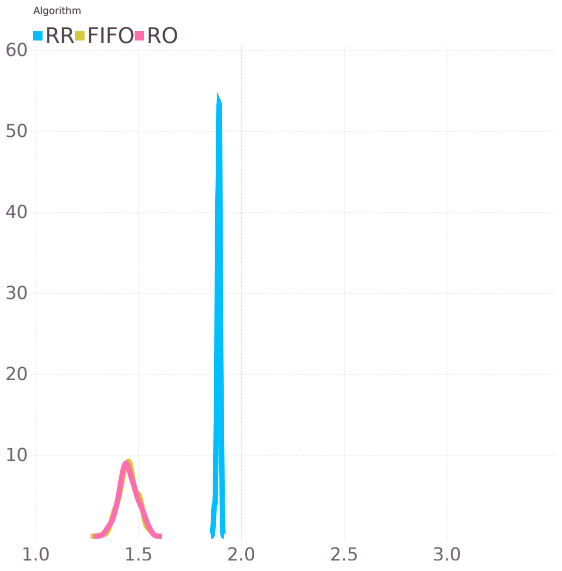
(a) p_j distributed according to χ_1^2.



(b) p_j distributed according to χ_2^2.



(c) p_j distributed according to χ_3^2.



(d) p_j distributed uniformly.

Figure 2.16: "Competitive ratio" of several algorithms in $P4$ online environment.

2.4.3 Offline Experiments

In this subsection, Problem 20: $\mathbf{1}|d_j|\sum_{j\in J}\omega_j u_j$ is considered. We compare EDD with WSPT to analyze their own advantage.

The instances were randomly generated as follows[8]:

For each instance, a set J of 50 jobs are generated. For each job $j \in J$, the processing time p_j and the weight ω_j was generated from the uniform distribution over $\{1,\ldots,100\}$ and $\{1,\ldots,10\}$ respectively.

Instance classes of varying hardness were generated by using different uniform distributions for generating the due dates. For a given relative range of due dates RDD $\in \{0.1, 0.2, 0.5\}$ and a given average tardiness factor TF $\in \{0.1, 0.2, 0.5\}$ a due date d_j for each job j was randomly generated from the uniform distribution over $\{\lfloor p(1 - \mathrm{TF} - \mathrm{RDD}/2)\rfloor, \lceil p(1 - \mathrm{TF} + \mathrm{RDD}/2)\rceil\}$, where $p = \sum_{j\in J} p_j$. It is easy to see that RDD controls the variance of d_j. To see the influence of TF, let us assume that RDD $= 0$ (hence, all jobs share a common due date $d = p(1 - \mathrm{TF})$) and consider the following cases.

- In the case TF $= 0$, i.e. d is equal to the sum of the processing time of all jobs.

  Thus, any schedule, which does not allow idle time of machines, is able to finish all jobs in time. In particular, all of WSPT, EDD, FIFO, and RO achieve an objective value 0.

- In the other extreme case TF $= 1$, the common due date is 0.

  Since the processing time of j is positive, no job can be finished in time.

Therefore, TF can be interpreted as a control parameter, which influences the number of delayed jobs. 50 instances were generated for each of the pairs of values of RDD and TF, yielding 450 instances. The algorithms WSPT and EDD are applied to solve these instances. See Figure 2.17 for their performances.

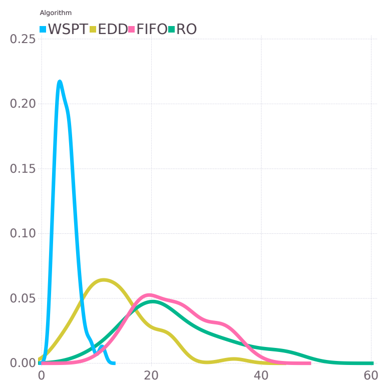
First, consider Figures 2.17g and 2.17h, where EDD is able to achieve the best possible objective value 0. Certainly, the schedule produced by EDD is able to finish all jobs in time. In such cases, the weights of jobs do not influence the optimal objective value, meaning that Problem 20: $\mathbf{1}|d_j|\sum_{j\in J}\omega_j u_j$ is similar to Problem 18: $\mathbf{1}|d_j|\sum_{j\in J} u_j$. By Lemma 2.21, EDD produces the optimal schedule.

By fixing TF in Figure 2.17, one can see that the performance of EDD approaches or surpasses the performance of WSPT while RDD grows. The reason is that EDD benefits from the variability of d_j, which is incurred by large RDD. Informally, the less jobs OPT has to delay, the better EDD performs in comparison to other algorithms.

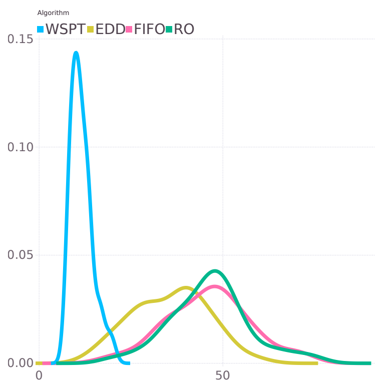
In all other cases, WSPT outperforms EDD. The advantage of WSPT in comparison to other algorithms grows in step with the value of TF. A large value of TF means that a certain amount of jobs will be delayed. In other words, the decision maker may find

[8] The design of experiments comes from: `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/wtinfo.html`. Their experiments focus on the objective function "total weighted tardiness". However, we have found that the same design is also good at visualizing the difference of performances between WSPT and EDD for Problem 20.

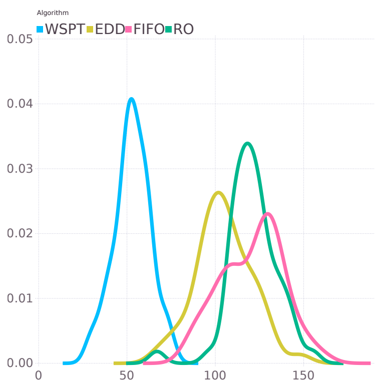
WSPT useful if the system is often overloaded, i.e. there are enough jobs waiting for processing at any point in time.



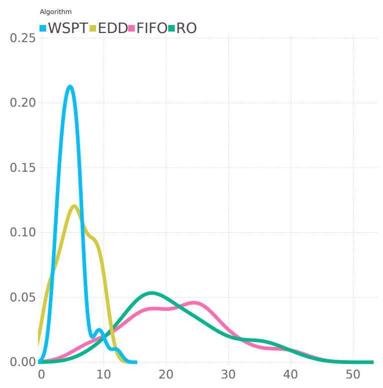
(a) RDD = 0.1, TF = 0.1.



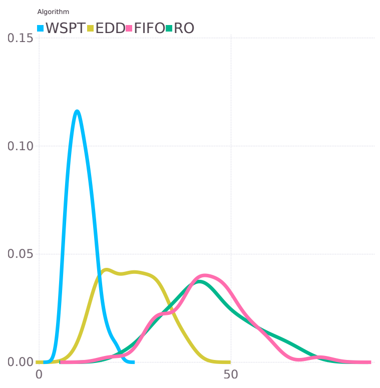
(b) RDD = 0.1, TF = 0.2.



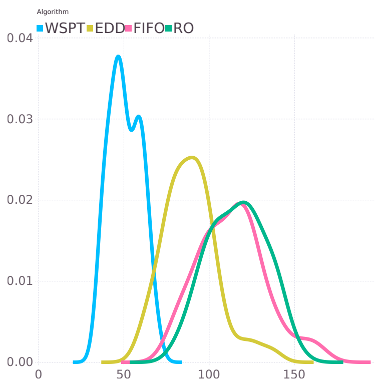
(c) RDD = 0.1, TF = 0.5.



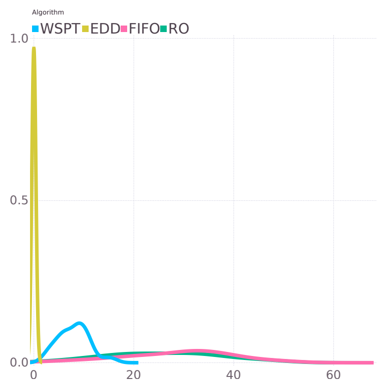
(d) RDD = 0.2, TF = 0.1.



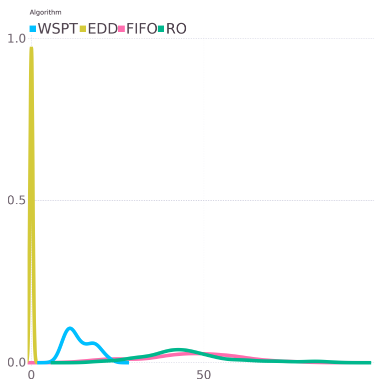
(e) RDD = 0.2, TF = 0.2.



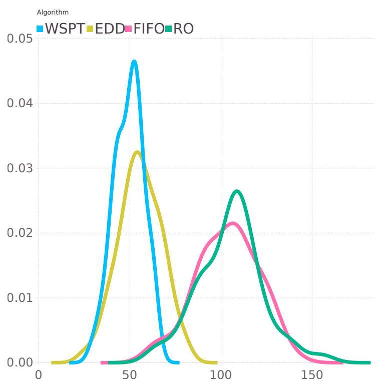
(f) RDD = 0.2, TF = 0.5.



(g) RDD = 0.5, TF = 0.1.



(h) RDD = 0.5, TF = 0.2.



(i) RDD = 0.5, TF = 0.5.

Figure 2.17: Performances of WSPT, EDD, FIFO, and RO in offline environment.

2.4.4 From Scheduling Problems to List Accessing Problem

Since the offline environment is applied, it is possible to create the priority list of WSPT at $t = 0$. The contribution from a job to the objective value is related to the position of that job in the priority list. More precisely,

- jobs with a high priority have a smaller chance to make a contribution to the objective function than the jobs with a low priority, if total unit penalty related objective function is applied,
- the jobs with a high priority make less contribution to the objective function than the jobs with a low priority, if total completion time related objective function is applied.

Informally, this can be thought of as we have a linked list of the jobs. The contribution from a job to the total cost in the scheduling problem corresponds to the cost of locating that job in the list. Scheduling algorithms considered in this chapter can be interpreted as algorithms which maintains a list, such that the total search cost is minimized. This similarity leads to the list accessing problem which was originally considered by Sleator and Tarjan (1985) when they proposed the concept of competitive analysis. Actually, some scheduling problems can be formulated as list accessing problems. After introducing the necessary terminologies, this similarity is presented in Remark 3.22.

Chapter 3

Competitive Analysis of List Accessing Problems

The minimization problem *list accessing problem*, which may also be called *list update problem*, is considered in this chapter. As introduced in Chapter 1, this problem may find its application in the context of data compression.

Example 3.1 (Bentley et al.,1986). *Consider the following simple "telegraph" message:*

THE CAR ON THE LEFT HIT THE CAR I LEFT.

Sender and receiver maintain identical word lists. The list is initially empty. To transmit the word W, the sender looks it up in the list. If it is present in the first position, the sender transmits 1*, which the receiver decodes by writing the element in that position of the list. If W is not in the list of N words, the sender reacts as though it were in the $N+1$-th position and sends the integer $N+1$ followed by the word W (which the receiver expects because $N+1$ is greater than the size of the current list). The message above will be compressed into the following code:*

$$1\mathit{THE}\ 2\mathit{CAR}\ 3\mathit{ON}\ 1\ 4\mathit{LEFT}\ 5\mathit{HIT}\ 1\ 2\ 6I\ 4.$$

In comparison to other words, the word "THE" is used most frequently in this message. Certainly, assigning "THE" to the first position of the word list reduces the total time needed to translate the code back into plain text. Indeed, sorting the words by their frequency is one of the algorithms for the list accessing problem, which will be defined in Section 3.3.

We organize this chapter as follow: Section 3.1 introduces some common terminologies to define list accessing problem formally. We focus on online algorithms of this problem and their competitiveness. Section 3.2 provides overview of the known results about the optimal algorithm OPT, which is required for the calculating of the competitive ratio. In the remaining sections of this chapter, we are going to study the known algorithms of list accessing problem and analyze their competitive ratios.

3.1 Notation

Definition 3.2. *A* ***list*** *is a finite ordered sequence $L = (x_i)_{i=1,\dots,n}$ of distinct list elements, the* ***support*** $\underline{L} = \{x_1, \dots, x_n\}$ *is the set of elements of L. A list element x is at the* ***position*** *i of L if and only if the initial segment $(x_1, \dots, x)$ has length i. A list element x* ***precedes*** *another list element x' in L, if and only if the initial segment $(x_1, \dots, x')$ of L includes x.*

A ***request*** *y for some list element in L is an element from $\underline{L}$. A* ***request sequence*** *I of L is an ordered sequence of requests to some list element in L. An* ***instance*** *(L, I) of the list accessing problem consists of an initial list L together with a request sequence $I = (y_j)_{j=1,\dots,m}$ of L.*

Occasionally, the same element can be referred in different ways which may be involved, as shown in Example 3.3. For the sake of clarity and simplicity, we assume the following notation during this chapter: Unless otherwise specified, list elements in L and requests in I are indexed by i and j respectively. Summations in this chapter are typically indexed by k. In unambiguous cases, the dependence on L is omitted.

Example 3.3. *Let $L = (1, 3, 2)$ and $I = (2, 3, 2, 3)$. The first request y_1 in I is asking for 2, which is the list element x_3 in position 3 of L. To refer to this element 2, we use either y_1 if we are talking about an element of the request sequence, or x_3 if the element in the list is considered.*

Notice that 2 is requested twice ($y_1 = y_3 = 2$) within the same request sequence I, whereas $x_i = x_{i'}$ is prohibited for different i, i' by definition of the list.

There are four possible actions to update the data structure which are considered in literature: *transposition*, *accessing*, insertion and deletion. We focus on the *static list accessing problem*, where only the transpositions and accessing are considered.

Definition 3.4. *Let $L = (x_i)_{i=1,\dots,n}$ be a list.*

For some list element $x' \in \{x_2, \dots, x_n\}$, the action ***transposition*** *$T(x, x')$ exchanges the positions of the element x' with its direct predecessor x in L. The list after transposition is denoted by $T(x, x')(L)$ or simply $T(L)$ in unambiguous cases.*

Clearly $T(x, x')$ is applicable to L if and only if x, x' are list elements in L and x is the direct predecessor of x' in L. A chain $T_s \circ \cdots \circ T_1$ is well defined in L if and only if the term T_{k+1} is well defined in the list $T_k(T_{k-1}(\dots T_1(L) \dots))$ for all $k \in \{1, \dots, s-1\}$.

For some request $y \in \{x_1, \dots, x_n\}$, the action ***accessing*** *y in L compares list elements with y in the order of L. The accessing is completed as soon as the position i is found with $x_i = y$.*

Remark 3.5. *Since L contains only different elements, the position i of any $y \in \underline{L}$ is unique for every request.*

The order of list elements in the list remains unchanged during an access, while transpositions always change this order. Since the well definedness of $T(x, x')$ depends on the positions of x and x' in the current list, a chain $\mathcal{T} = T_s \circ \cdots \circ T_1$ of transpositions is not commutative in general, as shown in the Example 3.6.

Example 3.6. *Let $L = (1, 3, 2)$. The chain $\mathcal{T} = T(1, 2) \circ T(1, 3)$ is well defined: The first transposition $T(1, 3)$ changes L to $L' = T(1, 3)(L) = (3, 1, 2)$. At this point in time, the second transposition $T(1, 2)$ is well defined in the current list L', thus the chain $\mathcal{T}$ is well defined and reorganizes L to $L'' = (3, 2, 1)$.*

Consider the chain $\mathcal{T}' = T(1, 3) \circ T(1, 2)$, the transposition $T(1, 2)$ is not well defined in L, as the list elements $x_1 = 1$ and $x_3 = 2$ are not consecutive in L.

Unless otherwise specified, the term $T(x, x')$ is used for applicable transpositions.

To solve an instance (L, I), one has to access all requests y in the order of I. The cost incurred during this process depends on the applied cost model. In most of the literature the following standard cost model is applied.

Definition 3.7. *In the* ***standard cost model****, accessing an element x at the position i incurs a cost of i.*

It is possible to reorganize the list at any point in time using transpositions. Immediately after accessing x', any sequence of transpositions moving x' forward (say $T(\cdot, x')$) is free. Therefore, x' can be moved without any cost to any position closer to the front of the list. Every other well defined transposition is designated as paid transposition and incurs a cost of 1.

The aim of the list accessing problem is to minimize the sum of costs incurred by accessings and paid transpositions while serving a request sequence I.

3.2 Background on the Optimal Offline Algorithm

Since the competitiveness compares online algorithms to OPT, a study of OPT is necessary. This section provides a brief overview.

In general, as shown by Ambühl (2017) (which is a representation of a conference talk of the same author in 2000), it is NP-hard to find OPT. A trivial upper bound is given by enumerating all possible pairs of permutations of the list and calculating the cost of each pair. This concept leads to a complexity of $\mathcal{O}(m \cdot (n!)^2)$ where n and m, respectively, is the length of the list and request sequence. However, considering certain subsets of the set of all optimal algorithms can reduce this complexity. The next few lemmas characterize an optimal algorithm in such a subset.

As shown by Reingold and Westbrook (1996), the paid transpositions can be necessary in order to realize the optimal cost.

Example 3.8 (Reingold and Westbrook,1996). *Let $L = (1, 3, 2)$ and $I = (2, 3, 2, 3)$. One can reorganize L to $L'' = (3, 2, 1)$ by applying the chain $T(1, 2) \circ T(1, 3)$ of paid transpositions, which incurs a cost of* 2.

If no additional transposition is applied on L'', accessing all requests in I incurs a cost of 6 *in total, meaning that* 8 *is an upper bound of the optimal cost, whereas the following case by case analysis shows that any algorithm without paid transpositions induces a cost of at least* 9.

We first remark an easy observation: For any two different requests y and y', the cost of accessing (y, y') in L is at least the sum of their positions in L.

Denote the access cost of y_j by a_j. As paid transpositions are prohibited, we have $a_1 = 3$ immediately.

Consider the case that no free transposition is applied between accessing y_1 and y_2, i.e. the current list after accessing y_1 is given by $(1, 3, 2)$. Then the cost $a_2 + a_3$ of accessing (y_2, y_3) is at least 5. As $a_j \geq 1$, we conclude that $\sum_{j=1}^{4} a_j \geq 9$.

Otherwise, in the case that free transpositions are applied before accessing y_2, we must have that 3 is in position 3 of the current list, whence $a_2 = 3$. Without paid transposition, the cost $a_3 + a_4$ of accessing (y_3, y_4) is at least 3, no matter at which positions these requests are. Thus, the same inequality $\sum_{j=1}^{4} a_j \geq 9$ holds.

Therefore, the cost of serving I is at least 9 without paid transposition, meaning that the paid transpositions are necessary in order to attain the optimal cost.

Moreover, it is shown that free exchanges are not necessary for an optimal algorithm.

Proposition 3.9 (Reingold and Westbrook,1996)**.** *Given any algorithms* ALG *for the list accessing problem, there exists an algorithm* ALG$'$ *which makes no free exchanges and produces costs equal to the costs of* ALG*.*

In the same work, the authors present an exact algorithm with complexity of $\mathcal{O}(2^n \cdot (n-1)! \cdot m)$. A result of Pietzrak (2001) is reported in Divakaran (2014) with a complexity of $\mathcal{O}(n^3 \cdot n! \cdot m)$. Presently, the best known algorithm, consuming $\mathcal{O}(n^2 \cdot (n-1)! \cdot m)$ to find an OPT, is reported in the preprint by Divakaran (2014).

The key of reducing the complexity is to consider exact algorithms that perform only specific actions. Let $L = (x_i)_{i=1,\ldots,n}$ be a list and assume that x_i is just requested. A chain of transpositions made just prior to the access is called a *subset transfer*, if it moves a subset of $\{x_1, \ldots, x_{i-1}\}$, possibly containing gaps, to just behind x_i in such a way that the relative position of list elements in this subset remains unchanged. After accessing x_i, an *element transfer* is a sequence of (paid as well as free) transpositions involving x_i.

Example 3.10. *Let $L = (1, 2, 3, 4)$ and assume that the list element 4 was just requested. Before accessing 4, a chain of transpositions reorganizing L into $L' = (2, 4, 1, 3)$ is a subset transfer regarding the subset $\{1, 3\}$. After accessing 4, the chain $\mathcal{T} = T(4, 3) \circ T(4, 1)$ of transpositions reorganizing L' into $L'' = (2, 1, 3, 4)$ is an element transfer.*

More precisely, a subset transfer first splits the support of an initial segment $L_{\text{init}} = (x_1, \ldots, x_{i-1})$ into two disjoint subsets $\underline{L_1}$ and $\underline{L_2}$. Then two lists L_1 and L_2 are constructed in a order preserving manner using the elements in $\underline{L_1}$ and $\underline{L_2}$ respectively. In the end, the original list L is reorganized into $(L_1, x_i, L_2, x_{i+1}, \ldots, x_n)$. An element transfer can move the requested element x_i to any position in the list. While the minimal chain for an element transfer can be uniquely determined trivially, a minimal chain of transpositions to transfer a subset behind x_i can be found using Lemma 3.15 or Lemma 3.17. The proof of both lemmas uses the notion of *inversion*, which plays a central rule in this chapter. After we define it in a formal way, we introduce some lemmas to understand the relation between inversion and transpositions.

Definition 3.11. *Let L, L' be two lists with $\underline{L} = \underline{L'}$. An ordered pair (x, x') of two different list elements $x, x' \in \{x_1, \dots, x_n\}$ is an **inversion** regarding (L, L') if and only if x precedes x' in L and the converse is true in L'.*

Notice that the notation of inversion is ordered. An ordered pair (x, x') is an inversion regarding (L, L') if and only if its reverse pair (x', x) is an inversion regarding (L', L).

Example 3.12. *Let $L = (1, 3, 2)$ and $L' = (3, 2, 1)$. Then $(1, 2)$ and $(1, 3)$ are the only two inversions regarding (L, L'). Notice that the ordered pairs $(2, 1)$ and $(3, 1)$ are not inversions regarding (L, L') but regarding (L', L).*

In other words, inversions are those pairs of list elements whose relative position has changed from L to L'. In the case that $L' = \mathcal{T}(L)$, a change of the relative position is only possible if the transposition involving these two corresponding list elements is contained in $\mathcal{T}$.

Lemma 3.13. *Let L be a list and x, x' be two different list elements of L where x precedes x'. Let $\mathcal{T} = T_s \circ T_{s-1} \circ \dots T_1$ be a well defined chain of transpositions of L and denote $L' = \mathcal{T}(L)$. Then (x, x') is an inversion regarding (L, L') if and only if the transposition $T(x, x')$ is included in $\mathcal{T}$ exactly one time more than $T(x', x)$.*

Proof. For $k \in \{1, \dots, s\}$ let L_k denote the list $T_k \circ \cdots \circ T_1(L)$ and $L_0 = L$. Then there exists a k such that x' precedes x for the first time, i.e. the smallest k where x precedes x' in L_{k-1} but x' precedes x in L_k. Since every transposition moves the list elements by at most one position, the transposition T_k is certainly $T(x, x')$.

Notice that $T(x', x)$ is not a valid choice for $T_1, \dots, T_{k-1}$ as x precedes x' in all of $L_0, \dots, L_{k-1}$. If there is any $l > k$ with $T_l = T(x', x)$, then x precedes x' in L_l. One can apply the argument above again to force an index $l < l' \leq s$ with $T_{l'} = T(x, x')$.

Therefore, the transposition $T(x, x')$ occurs always one more time in $\mathcal{T}$ than $T(x', x)$ does. $\square$

In particular, the following corollary is the special case of Lemma 3.13, where $\mathcal{T}$ consists only of one transposition $T(x, x')$.

Corollary 3.14. *Let L and L' be two lists sharing the same support. Then every transposition $T = T(x, x')$ in L either creates or eliminates exactly one inversion.*

More precisely, if (x, x') is an inversion regarding (L, L'), then (x, x') is no longer an inversion regarding $(T(L), L')$ and if (x, x') is not an inversion regarding (L, L'), then (x', x) becomes an inversion regarding $(T(L), L')$.

Proof. The last two claims follow directly from the definition of transposition and inversion. The less trivial part is to show the first claim: Only exactly one inversion involving x and x' can be either created or eliminated.

Notice, by the definition of transposition, that x is the direct predecessor of x' in L. Thus, it suffices to show that

1. inversions regarding (L, L') other than (x, x') remain inversions after transposition,

2. no other inversion than (x', x) can be created.

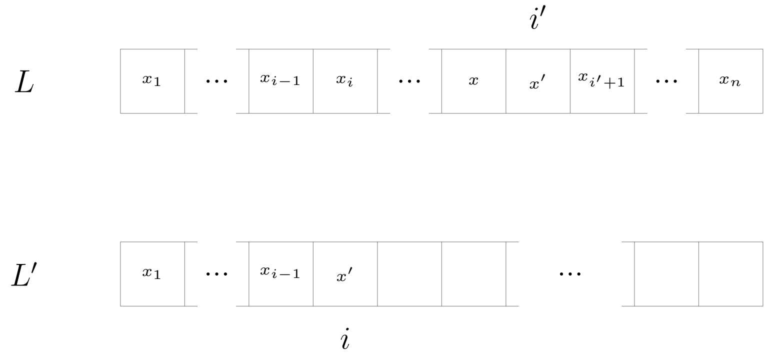
Let $\mathcal{T}_1$ be a chain of transpositions converting L to L'. Then, it is easy to see that $\mathcal{T}_2 = T(x', x) \circ \mathcal{T}$ is a well defined chain of transpositions converting $T(L)$ to L'.

Consider a pair $(\overline{x}, \overline{x}')$ of list elements different to (x, x'), i.e. $\{\overline{x}, \overline{x}'\} \neq \{x, x'\}$. Hence we have that $\mathcal{T}_1$ includes exactly as many $T(\overline{x}, \overline{x}')$ and $T(\overline{x}', \overline{x})$ as $\mathcal{T}_2$ does. By Lemma 3.13, this means that either $(\overline{x}, \overline{x}')$ is an inversion regarding both of (L, L') and $(T(L), L')$ or it is not regarding both of (L, L') and $(T(L), L')$. This completes the proof. □

If there are inversions between (L, L'), there exists a transposition to eliminate exactly one of them.

Lemma 3.15. *Let L and L' be two lists sharing the same support. Then there exists an inversion regarding (L, L') if and only if there exist two consecutive list elements x, x' in L such that (x, x') is an inversion regarding (L, L').*

Proof. The necessity is trivial. Since there exists an inversion regarding (L, L'), there must be some position where L and L' differ from each other. For the sufficiency let i be the first position where $x_i \neq x' = L'_i$. For the sake of clarity, we number the list elements according to their position in L.



Notice that the position of x' in L, say i', must be after i as it is assumed that $x_i \neq x'$ and the common initial segment $(x_1, \ldots, x_{i-1})$ certainly does not include x'. Let x denote the direct predecessor of x' in L. It is then easy to see that the pair (x, x') is an inversion regarding (L, L'). Indeed, as lists do not contain duplications, the direct predecessor x is not included in the initial segment $(x_1, \ldots, x_{i-1}, x')$ of L'. Hence x' must precede x in L'. Therefore, we have that x precedes x' in L and the converse is true in L'. □

Remark 3.16. *Actually, the same argument applies also to each list element in the sublist $(x_i, \ldots, x_{i'-2})$ in L, i.e. the pair $(\overline{x}, x')$ is an inversion regarding (L, L') for every $\overline{x} \in \{x_i, \ldots, x_{i'-2}\}$.*

Lemma 3.17 (Reingold and Westbrook,1996)**.** *Let L and L' be two lists sharing the same support. The number of inversions between L and L' equals the minimum number of transpositions needed to convert L to L'.*

Proof. We use an inductive argument on the length of the lists. For lists with length 1 the result is obvious, as there are no inversions possible. We assume that x is in position 1 in L and in position $i \neq 1$ in L'. Otherwise, one can delete the maximal common initial segment from both list without changing the set of inversions.

Let us number the list elements according to their position in L', i.e. there exists a permutation τ such that $L' = (x_1, x_2, \ldots, x_n)$, $L = (x_{\tau(1)}, x_{\tau(2)}, \ldots, x_{\tau(n)})$ and $x = x_{\tau(1)} = x_i$.

Consider the chain $\mathcal{T} = T(x_1, x) \circ T(x_2, x) \circ \cdots \circ T(x_{i-1}, x)$ which moves $x = x_i$ to the front of L'.

By Corollary 3.14 and Remark 3.16, applying $\mathcal{T}$ eliminates $i-1$ inversions, namely the pair (x, x_k) for every $k \in \{1, \ldots, i\}$. Therefore, the number of inversions decreases by $i-1$. We end up with two lists L and $\mathcal{T}(L')$, where the first element of both lists is x. Deleting the common initial segment and applying the inductions hypothese completes the proof. □

Lemma 3.17 can be used to construct a minimal chain of transpositions to achieve given subset transfers.

```
  Input: $L = (x_1, \ldots, x_n)$, $L' = (x'_1, \ldots, x'_n)$ with $\underline{L} = \underline{L'}$
  Output: Minimal chain $\mathcal{T}$ such that $\mathcal{T}(L) = L'$
1 $\mathcal{T} = \emptyset$;
2 while $L \neq L'$ do
3     Find the smallest $i$ with $x_i \neq x'_i$;
4     Find the position $i' > i$ with $x_{i'} = x'_i$;
5     $\mathcal{T} = T(x_i, x_{i'}) \circ \cdots \circ T(x_{i'-1}, x_{i'}) \circ \mathcal{T}$;
6     $L = \mathcal{T}(L)$.
7 end
```

Algorithm 6: Find a minimal chain of transpositions.

For given L and I, one trivial method to find the optimal solution is enumeration. Construct a weighted complete m-partite graph $G_{opt} = (V, E)$ as illustrated in Figure 3.1. The vertex set V is a disjoint union of $m+1$ independent vertex sets $(V_k)_{k=0,\ldots,m}$ where V_0 contains only one vertex v_0 presenting the initial list L, and any other independent set V_k consists of $n!$ vertices representing $n!$ permutations of L. For $v \in V$, let L_v denote the list represented by v. The weight of an edge uv between V_k and $V_{k'}$ is equal to the minimal cost to reorganize L_u to L_v, providing that y_k was just accessed. These weights can be calculated using Algorithm 6. Then, an optimal solution to serve I can be found by finding a shortest path from V_0 to V_m. The main contribution of Reingold and Westbrook (1996) is to show that there exists a shortest path avoiding a certain subset of edges.

Theorem 3.18 (Reingold and Westbrook,1996). *There exists an optimal offline algorithm that performs only subset transfers.*

In other words, one may drop all edges uv from Figure 3.1, where L_v is not obtainable

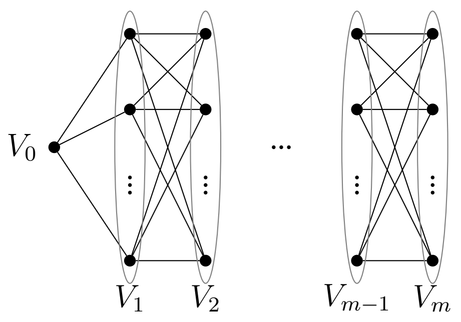


Figure 3.1: Enumeration to find the optimal solution.

from L_u via subset transfer. Thus, the complexity of finding an optimal offline algorithm is reduced.

Divakaran claimed to reduce this cardinality further. Based on an optimal solution obtained by using only subset transfers, Divakaran (2014) attempts to rearrange the transpositions in such a way that the list is reorganized initially before the first accessing and only element transfers are applied subsequently.

The corresponding complexity to construct a graph, assuming that only subset transfers and element transfers are applied, is given by Lemma 3.19.

Lemma 3.19 (Reingold and Westbrook,1996, Divakaran,2014)**.** *The following statements hold for every $k = 1, \dots, m-1$:*

1. *There are $(2^n - 1) \cdot (n-1)!$ pairs of vertices $u \in V_k$ and $v \in V_{k+1}$, such that L_v is obtainable from L_u via subset transfer.*
2. *There are $n \cdot n!$ pairs of vertices $u \in V_k$ and $w \in V_{k+1}$, such that L_w is obtainable from L_u via element transfer.*

To summarize, this line of research focuses on considering a subgraph G'_{opt} induced by a subset $E' \subseteq E$ where the length of shortest V_0, V_m-path in G'_{opt} remains the same as in G_{opt}.

In general, the exponential dependence on n restricts its use in practice, even though the calculation of the weights of E' (as well as in E) can be done highly parallel.

For the restricted case when the list is short, we can state the following lemma.

Lemma 3.20 (Folklore)**.** *The following algorithm achieves the optimal cost on every request sequence with a list of length two: Whenever the next two requests ask for the last element in the current list, move it to the front via free transposition after the first access. Otherwise do nothing.*

One may also restrict the possible actions. A deterministic algorithm is *static* if it reorganizes L before the first access and never applies any free or paid transposition thereafter. Certainly, there are only $n!$ static algorithms for a list with length n. The best algorithm among these $n!$ algorithms can be characterized as follows.

Lemma 3.21 (Folklore). *For any instance (L, I), the following static offline algorithm achieves the optimal cost within the set of static algorithms: Reorganize the list in non-increasing order in terms of request frequencies in I.*

Remark 3.22. *Static list accessing algorithms are similar to list scheduling algorithms if one ignores the transposition costs. Let $L = (1, 2, 3)$ be a list and $I = (3, 2, 3, 2, 2, 3, 1, 1)$ be a request sequence of L. Consider the static list accessing algorithm which resort L into $L' = (2, 3, 1)$ at the beginning.*

Observe that the contribution to the total accessing cost of a list element x is equal to its position in L' times the number of requests to x in I. In other words, whenever x is requested, every list element before x in the list contributes one unit of cost to the total accessing cost. Now consider Problem 3*: $\mathbf{1}||\sum_{j\in J} c_j$ (see Table 2.2). For each list element x we generate a job j_x with a processing time p_x equal to the number of requests to x in I. The static list accessing algorithm above corresponds to the list scheduling algorithm with the priority list (j_1, j_3, j_2) which is the reverse order of L'. Then, the total completion time is equal to*

$$p_1 + (p_3 + p_1) + (p_2 + p_3 + p_1) = p_2 + p_3 \cdot 2 + p_1 \cdot 3.$$

The right hand side can be interpreted as the sum of costs of accessing list element 2 p_2 *times, accessing list element* 3 p_3 *times, and accessing list element* 1 p_1 *times.*

It is easy to see that the optimal static offline algorithm in Lemma 3.21 corresponds to SPT *which is also optimal for Problem* 3*, see Lemma 2.17.*

The understanding of the behaviors of some optimal algorithms turns out to be useful for designing good deterministic online algorithms, as will be discussed in section 3.3.

3.3 Deterministic Algorithms for the List Accessing Problem

In this section, four deterministic online algorithms TRANS, MTF, FC, and TS for the list accessing problem are introduced and analyzed. Most algorithms solving list accessing problems are either variants of one of these four algorithms or a hybrid version of two of them.

Definition 3.23. *Let **Transpose** (short:* TRANS*) denote the algorithm which always applies exactly one free transposition, i.e. after accessing x, the position of x is exchanged with its direct predecessor in the current list.*

*Let **Move To the Front** (short:* MTF*) denote the algorithm which always applies all free transpositions, i.e. after accessing x, x is moved to the first position of the current list.*

*The algorithm **Frequency Count** (short:* FC*) reorganizes the list after each accessing such that the list elements are ordered in non-increasing order of their frequencies. More precisely,* FC *maintains a counter for each list element recording the number of access*

Request	TRANS		MTF		FC		TS	
	List	Cost	List	Cost	List	Cost	List	Cost
Init.	(1 2 3)	0	(1 2 3)	0	(1 2 3)	0	(1 2 3)	0
3	(1 3 2)	3	(3 1 2)	3	(3 1 2)	3	(1 2 3)	3
2	(1 2 3)	6	(2 3 1)	6	(3 2 1)	6	(1 2 3)	5
3	(1 3 2)	9	(3 2 1)	8	(3 2 1)	7	(3 1 2)	8
2	(1 2 3)	12	(2 3 1)	10	(3 2 1)	9	(2 3 1)	11
2	(2 1 3)	14	(2 3 1)	11	(2 3 1)	11	(2 3 1)	12
3	(2 3 1)	17	(3 2 1)	13	(2 3 1)	13	(2 3 1)	14
1	(2 1 3)	20	(1 3 2)	16	(2 3 1)	16	(2 3 1)	17
1	(1 2 3)	22	(1 3 2)	17	(2 3 1)	19	(1 2 3)	20

Table 3.1: The behavior of algorithms on instance $L = (1, 2, 3)$ and $I = (3, 2, 3, 2, 2, 3, 1, 1)$

of this element. All counters are initialized with 0 *and increased by* 1 *after each access. Immediately after updating a frequency counter, the accessed element is moved to the position corresponding to its counter.*

The algorithm ***Timestamp*** *(short:* TS*) moves the accessed list element x directly in front of the first element in the current list that was accessed at most once since the last request to x. If such an element does not exist or if x is requested for the first time, nothing is done.*

Remark 3.24. *Let $L = (1, 2, 3)$ be a list and $I = (3, 2, 3, 2, 2, 3, 1, 1)$ be a request sequence of L. The lists maintained by* TRANS, MTF, FC, *and* TS *are summarized in Table 3.1.*

Although it is proven in Proposition 3.9, that the optimal algorithms may avoid any free transposition, these four well known online deterministic algorithms rely heavily on free transpositions. Indeed, since FC *checks the frequency after every access, the only counter which was changed since last check is the counter of the requested list element. Thus,* FC *moves actually only the requested element to some earlier position in the list. As the counter increases, this list element may either be moved forward or stay at its position. The other three algorithms trivially apply only free transpositions.*

Recall the two actions introduced in Section 3.2, i.e. element transfer and subset transfer. The actions of these algorithms are actually special cases of element and subset transfer.

3.4 Bounds for Competitiveness of Deterministic Online Algorithms

In terms of the competitiveness, no deterministic algorithm can be better than 2-competitive. In Irani (1991), the next theorem is attributed to Karp and Raghavan.

Theorem 3.25 (Karp and Raghavan)**.** *For the static list accessing problem with a list of n elements, deterministic online algorithms have a competitive ratio of at least $2 - \frac{2}{n+1}$.*

Proof of Borodin and El-Yaniv (2005). Let (L, I) be an instance of the list accessing problem with n list elements and m requests. Consider any deterministic online algorithm ALG. Define I such that immediately before ALG accesses y_j, the list element $x_i = y_j$ is always in the last position of the current list. This is possible as ALG is deterministic. Certainly, ALG incurs a cost of mn.

The lower bound of the competitiveness of ALG is found by using an averaging technique over the set of static algorithms. For any permutation $\sigma \in S_n$ let ALG_σ be the static algorithm reorganizing L to $\sigma(L)$. By Lemma 3.17, the cost of this reorganization is at most $\frac{n(n-1)}{2}$. Consider the set $\{\mathrm{ALG}_\sigma\}_{\sigma \in S_n}$ of static algorithms. We first calculate the sum of the cost $C_{\mathrm{ALG}_\sigma}(I)$ for all $\sigma \in S_n$. Notice that any list element x, and hence also any request y, is in position k in exactly $(n-1)!$ lists of $\{\sigma(L)\}_{\sigma \in S_n}$ for all $k = 1, \dots, n$. Thus, the accessing of any element costs $(n-1)! \cdot (1 + 2 + \cdots + n) = (n-1)! \cdot \frac{n(n+1)}{2}$ in total. Since this cost is not dependent on when or which element is requested, we conclude that these $n!$ static algorithms incur together access costs of $m(n-1)! \cdot \frac{n(n+1)}{2}$ and paid transposition costs of at most $n! \cdot \frac{n(n-1)}{2}$. By taking the average, there must exist one σ^* such that

$$C_{\mathrm{ALG}_{\sigma^*}}(L, I) \leq \frac{m(n+1)}{2} + \frac{n(n-1)}{2}.$$

Therefore, the competitive ratio of ALG_σ approaches $\frac{2mn}{m(n+1)} = 2 - \frac{2}{n+1}$ for a large m relative to n, which converges to 2 for large n. □

The algorithms TRANS and FC are not competitive for any finite c, i.e. in the worst case, the ratio $\frac{C_{\mathrm{TRANS}}(I)}{C_{\mathrm{OPT}}(I)}$ and $\frac{C_{\mathrm{FC}}(I)}{C_{\mathrm{OPT}}(I)}$ can be arbitrary large, as we show in the next example.

Example 3.26. *Let $L = (x_1, \dots, x_n)$ be a list. We determine lower bounds for the competitive ratio by comparing the performance of* TRANS *and* FC *on individual request sequence I with* MTF.

By the optimality of OPT*, the cost $C_{\mathrm{OPT}}(I)$ of applying* OPT *is certainly less or equal to the cost of applying* MTF*. Thus, we know that*

$$\frac{C_{\mathrm{ALG}}(I)}{C_{\mathrm{MTF}}(I)} \leq \frac{C_{\mathrm{ALG}}(I)}{C_{\mathrm{OPT}}(I)}$$

is a lower bound of competitive ratio of any algorithm ALG*.*

Observe that MTF *benefits from immediately repeated requests to the same list element. Let I_k be the request sequence that repeatedly requests item x_k for n times for $k = 1, \dots, n$.*

For TRANS *consider the request sequence I_n.* TRANS *exchanges the position of x_n with its direct predecessor after each access, thus reduces the access cost by 1 for every request, leading to costs of $\frac{n(n-1)}{2}$. In comparison to* TRANS*, consider the algorithm* MTF*. The first request incurs costs of n. Before the second access, the list element x_n is moved to the first position and thus every subsequent access costs only 1. Hence we* *have* $C_{\text{MTF}}(I) = 2n - 1$*. We obtain a lower bound*

$$\frac{C_{\text{TRANS}}(I)}{C_{\text{OPT}}(I)} \geq \frac{C_{\text{TRANS}}(I)}{C_{\text{MTF}}(I)} = \frac{n \cdot (n+1)}{4n-2} = \mathcal{O}(n).$$

For FC *consider the request sequence* $I = (I_1, I_2, \ldots, I_n)$*. On the one hand, note that the counter of* x_i *is always less than or equal to all counters of* $x_1, \ldots, x_{i-1}$ *during the entire process. Hence,* FC *does not reorganize the list at all. Thus, we have* $C_{\text{FC}}(I) = n + 2n + \cdots + n^2 = n \cdot \frac{n(n+1)}{2}$*. On the other hand, the cost of serving* I_k *using* MTF *is equal to* $k+n-1$ *for every* $k = 1, \ldots, n$*. Therefore,* MTF *incurs costs of* $\sum_{k=1}^{n}(k+n-1) = n(n-1) + \frac{n(n+1)}{2}$*. We obtain a lower bound of* $\frac{C_{\text{FC}}(I)}{C_{\text{OPT}}(I)}$*:*

$$\frac{C_{\text{FC}}(I)}{C_{\text{OPT}}(I)} \geq \frac{C_{\text{FC}}(I)}{C_{\text{MTF}}(I)} = \frac{n(n+1)}{3n-1} = \mathcal{O}(n).$$

In both cases, the lower bound grows indefinitely.

In contrast to the worst case performance of TRANS and FC, the algorithms MTF and TS are both 2-competitive which is because of Theorem 3.25 best possible.

Theorem 3.27 (Sleator and Tarjan,1985)**.** *Algorithm* MTF *is* 2*-competitive.*

Beginning with the list $L = (x_i)_{i=1,\ldots,n}$, we imagine that MTF and OPT are running side by side for solving the same instance (L, I). To avoid ambiguity, let L^{MTF} and L^{OPT} denote the lists maintained by algorithm MTF and OPT respectively. The process of serving one request y_j is split into several steps as follows.

1. request y_j arrives,
2. MTF accesses y_j in the list L^{MTF},
3. OPT accesses y_j in the list L^{OPT},
4. MTF performs free transpositions (if any),
5. OPT performs free transpositions and paid transpositions (if any).

Among them, the steps $2, 3$, and 5 incur costs. In the original proof, the cost of MTF is bounded from above using a potential function Φ, which is defined via the notion of inversions. Recall that a pair of list elements (x, x') is an inversion regarding $(L^{\text{MTF}}, L^{\text{OPT}})$ if and only if x precedes x' in L^{MTF} and the converse is true in L^{OPT}.

Remark 3.28. *For every inversion* (x, x') *regarding* $(L^{\text{MTF}}, L^{\text{OPT}})$*, the algorithm* MTF *has to pay* 1 *more unit in order to reorganize the* L^{MTF} *to* L^{OPT}*. This is a measure of the distance between* L^{MTF} *and* L^{OPT} *in terms of the number of transpositions needed to convert one to the other.*

All inversions in the proof of Theorem 3.27 are with respect to $(L^{\mathrm{MTF}}, L^{\mathrm{OPT}})$. For the sake of simplicity, this dependence on concrete lists is omitted.

Proof of Theorem 3.27 of Borodin and El-Yaniv (2005). Since MTF does not use paid transpositions, the cost of MTF is incurred only by accessing the requests y, denoted by

$$C_{\mathrm{MTF}}(I) = C^A_{\mathrm{MTF}}(I).$$

By Proposition 3.9, we may assume that OPT does not use free transpositions, the cost of OPT is the sum of the costs of accessing y and the costs of paid transpositions performed before accessing y, denoted by

$$C_{\mathrm{OPT}}(I) = C^A_{\mathrm{OPT}}(I) + C^P_{\mathrm{OPT}}(I).$$

For the 2-competitiveness, it is sufficient to show that:

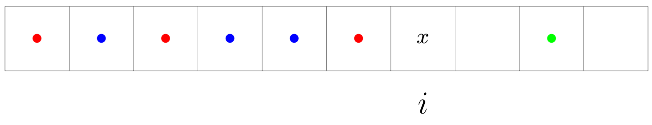
$$C^A_{\mathrm{MTF}}(I) \leq 2 \cdot C^A_{\mathrm{OPT}}(I) - 1 + C^P_{\mathrm{OPT}}(I).$$

Let I be a given request sequence of L. Assume that the requested list element $x(= y)$ is in the i-th position in L^{MTF} and at i'-th position in L^{OPT} at the moment of step 1. In other words, $C^A_{\mathrm{MTF}}(y) = i$ and $C^A_{\mathrm{OPT}}(y) = i'$.

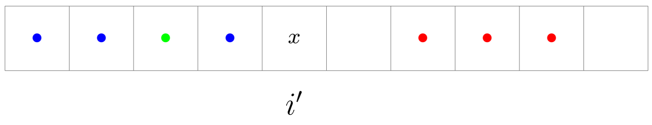
Every list element $\overline{x}$ preceding x in L^{OPT} or L^{MTF} must be exactly one of the next three types:

- $\overline{x}$ precedes x in both of L^{MTF} and L^{OPT}, let z count the number of such $\overline{x}$.
- $(\overline{x}, x)$ is an inversion, let g count the number of such $\overline{x}$.
- $(x, \overline{x})$ is an inversion, let h count the number of such $\overline{x}$.

Figure 3.2 illustrates one possible partition.



(a) L^{MTF} after step 1.



(b) L^{OPT} after step 1.

Figure 3.2: Lists in the proof of Theorem 3.27.

Claim: The following relations hold:

$$\begin{aligned} z + g &= i - 1 \\ z + h &= i' - 1 \\ i &\leq i' + g \\ i' &\leq i + h \end{aligned}$$

Proof: The first two equations are trivial. One of the inequalities is also trivial depending on the sign of $i - i'$. Assume that $i \geq i'$. Since h is non-negative, the inequality $i' \leq i + h$ obviously holds. The remaining inequality can easily be derived from the equations.

$$\begin{aligned} i - i' =& i - 1 - (i' - 1) \\ =& z + g - z - h \\ =& g - h \leq g, \end{aligned}$$

which is equivalent to $i \leq i' + g$. In the case $i' \geq i$, the proof works analogously by switching the role of g and h. ■

Consider the amortized costs $a(y) = C^A_{\text{MTF}}(y) + \Phi_5(y) - \Phi_1(y)$ where $\Phi_k(y)$ is the number of inversions after step k during serving request y. Notice that we have the equation

$$C^A_{\text{MTF}}(I) = \sum_{y \in I} C^A_{\text{MTF}}(y) = \sum_{y \in I} a(y) + \Phi_1(y_1) - \Phi_5(y_m)$$

where $\Phi_1(y_1) = 0$ since MTF and OPT begins with the same list and $\Phi_5(y_m) \geq 0$. Thus, in order to prove

$$C^A_{\text{MTF}}(I) \leq 2 \cdot C^A_{\text{OPT}}(I) - 1 + C^P_{\text{OPT}}(I),$$

it is sufficient to show that

$$a(y) \leq 2 \cdot C^A_{\text{OPT}}(y) - 1 + C^P_{\text{OPT}}(y)$$

holds for all y.

Before Step 5, the only actions changing the list order are the free transpositions of MTF. Since x is moved to the front in L^{MTF}, no list element can precede it anymore. Hence, all inversions counted by g are eliminated, whereas all list elements counted by z become inversions together with x before the transpositions of OPT (Step 5). Thus, we conclude that $\Phi_4 - \Phi_1 \ = z - g$. During Step 5, every transposition of OPT can create at most one inversion by Corollary 3.14, meaning that $\Phi_5 - \Phi_4 \ = C^P_{\text{OPT}}(y)$. Therefore, the amortized costs $a(y)$ are bounded from above by

$$\begin{aligned} a(y) =& i + z - g + C^P_{\mathrm{OPT}}(y) \\ =& i + (i - g - 1) - g + C^P_{\mathrm{OPT}}(y) \\ =& 2(i - g) - 1 + C^P_{\mathrm{OPT}}(y) \\ \le& 2i' - 1 + C^P_{\mathrm{OPT}}(y) \\ =& 2 \cdot C^A_{\mathrm{OPT}}(y) - 1 + C^P_{\mathrm{OPT}}(y) \end{aligned}$$ □

Theorem 3.29 (Albers,1998)**.** *The algorithm* TS *is* 2*-competitive.*

The standard proof of Theorem 3.29 makes use of the *projective property.*

Definition 3.30. *Let (L, I) be an instance of the list accessing problem. For any pair of list elements x, x' let $L_{xx'}$ and $I_{xx'}$, respectively, denote the projection of L and I onto $\{x, x'\}$. More precisely, one constructs $L_{xx'}$ by deleting elements other than x, x' from L and preserving the relative position of x and x'. The request sequence $I_{xx'}$ can be obtained in the same way. Similar to the normal context, duplication is allowed in $I_{xx'}$.*

An algorithm ALG *satisfies the projective property if the relative position of any two list elements x, x', when* ALG *is applied on (L, I), is the same as when* ALG *is applied to $(L_{xx'}, I_{xx'})$.*

The analysis of projective algorithms applies the so called partial cost model.

Definition 3.31. *In the* ***partial cost model****, accessing an element x at the position i incurs a cost of $i - 1$. The transposition cost is defined in the same way as in the standard cost model.*

The advantage of the partial cost model is that the costs incurred by projective algorithms are characterized easily and the upper bounds in the partial cost model applies naturally in the standard cost model.

Remark 3.32. *Let $C^P_{\mathrm{ALG}}(I)$ and $C^S_{\mathrm{ALG}}(I)$ denote the cost incurred by algorithm* ALG *in partial and standard cost model, respectively. Then we have*

$$C^P_{\mathrm{ALG}}(I) = C^S_{\mathrm{ALG}}(I) - m,$$

as every access in the partial model costs 1 *less than in the standard model.*

Notice that the cost model may influence the behavior of optimal strategy in general. Let OPT^P *and* OPT^S*, respectively, denote the optimal strategy in the partial and standard cost model. Although it is not clear if* OPT^P *coincides with* OPT^S*, we have at least*

$$C^P_{\mathrm{OPT}^P}(I) \le C^P_{\mathrm{OPT}^S}(I)$$

by considering OPT^S *as an arbitrary algorithm, which is certainly not able to beat the optimal algorithm* OPT^P *in the partial cost model.*

Assume that ALG *is c-competitive in the partial cost model. Then we have*

$$\begin{aligned}C^S_{\text{ALG}}(I) =&C^P_{\text{ALG}}(I) + m\\ \leq&c \cdot C^P_{\text{OPT}^P}(I) + m\\ \leq&c \cdot C^P_{\text{OPT}^S}(I) + m\\ <&c \cdot (C^P_{\text{OPT}^S}(I) + m)\\ =&c \cdot C^S_{\text{OPT}^S}(I)\end{aligned}$$

Thus, the upper bounds for competitiveness in the partial cost model apply also in the standard cost model. In a similar way, on can show that every lower bound in the standard cost model applies also in the partial cost model, but the converse statement requires individual analysis.

Usually, the lower bound for competitiveness of ALG *is provided together with a particular instance. Due to the lack of knowledge about* OPT*,* ALG *is compared to another well understood algorithm, typically to* MTF*, and the bound is established by*

$$\frac{C^S_{\text{ALG}}(I)}{C^S_{\text{OPT}}(I)} \geq \frac{C^S_{\text{ALG}}(I)}{C^S_{\text{MTF}}(I)},$$

cf. 3.26.

Assume that we are able to show a lower bound d of the competitiveness of ALG *in the partial cost model, i.e. we have*

$$\frac{C^P_{\text{ALG}}(I)}{C^P_{\text{OPT}}(I)} \geq \frac{C^P_{\text{ALG}}(I)}{C^P_{\text{MTF}}(I)} \geq d$$

for some request sequence I. In the standard cost model, we want to bound

$$\frac{C^S_{\text{ALG}}(I)}{C^S_{\text{MTF}}(I)} = \frac{C^P_{\text{ALG}}(I) + m}{C^S_{\text{MTF}}(I) + m}.$$

As long as the cost $C^P_{\text{ALG}}(I)$ is of the order of $\mathcal{O}(m^2)$, the constant difference m disappears by increasing the length of the request sequence (and possibly together with the length of list), see the proof of Lemma 3.44 for instance.

Lemma 3.33 (Borodin and El-Yaniv,2005)**.** *Let* ALG *be an online projective list accessing algorithm which does not use paid transpositions. Then we have for every instance (L, I)*

$$C_{\text{ALG}}(I) = \sum_{x,x' \in \underline{L}, x \neq x'} C_{\text{ALG}}(L_{xx'}, I_{xx'}).$$

Proof. We first show that if I contains only one request, i.e. $I = (\overline{x})$, then

$$C_{\text{ALG}}(\overline{x}) = \sum_{x,x' \in \underline{L}, x \neq x'} C_{\text{ALG}}(L_{xx'}, I_{xx'}).$$

Notice that the request sequence $I_{xx'} = (\overline{x})_{xx'}$ is empty if $\overline{x} \notin \{x, x'\}$, thus

$$\sum_{x,x' \in \underline{L}, x \neq x'} C_{\mathrm{ALG}}(L_{xx'}, I_{xx'}) = \sum_{x,x' \in \underline{L}, x \neq x', \overline{x} \in \{x,x'\}} C_{\mathrm{ALG}}(L_{xx'}, I_{xx'}) = \sum_{x \in \underline{L}, x \neq \overline{x}} C_{\mathrm{ALG}}(L_{x\overline{x}}, \overline{x}).$$

Consider the set $Z = \{x \in \underline{L} \mid x \text{ precedes } \overline{x} \text{ in } L\}$.

On the one hand, as ALG does not use paid transpositions, the cost $C_{\mathrm{ALG}}(\overline{x})$ is equal to the cost of accessing $\overline{x}$. By the definition of the partial cost model, this cost is equal to the number of list elements preceding $\overline{x}$ in L, i.e. $|Z|$.

On the other hand, the relative position of x and $\overline{x}$ in L is the same as initially in $L_{x\overline{x}}$, thus $x \in \underline{L}$ precedes $\overline{x}$ in $L_{x\overline{x}}$ if and only if $x \in Z$, meaning that

$$C_{\mathrm{ALG}}(L_{x\overline{x}}, \overline{x}) = \mathbb{1}_Z(x) = \begin{cases} 1, \text{ if } x \in Z, \\ 0, \text{ otherwise.} \end{cases}$$

Therefore, we have

$$\sum_{x,x' \in \underline{L}, x \neq x'} C_{\mathrm{ALG}}(L_{xx'}, I_{xx'}) = \sum_{x \in \underline{L}, x \neq \overline{x}} C_{\mathrm{ALG}}(L_{x\overline{x}}, \overline{x}) = |Z| = C_{\mathrm{ALG}}(\overline{x}).$$

Notice that $C_{\mathrm{ALG}}(L_{x\overline{x}}, \overline{x}) = \mathbb{1}_Z(x)$ holds if and only if the relative position of x and $\overline{x}$ in L is the same as in $L_{x\overline{x}}$. Since ALG is projective, this statement holds even after accessing $\overline{x}$. Therefore, the proof for I having arbitrary length can be obtained by applying the argument for the case $I = (\overline{x})$ inductively. □

In order to calculate the cost incurred by a projective algorithm ALG on instance (L, I), it is sufficient to consider the cost incurred by ALG on sub-instances $(L_{xx'}, I_{xx'})$. The cost incurred by OPT can be analyzed in a similar way.

Define the projected algorithm OPT^P as follows: OPT^P operates on $\binom{n}{2}$ instances $(L_{xx'}, I_{xx'})$ simultaneously for all $x, x' \in \underline{L}, x \neq x'$. Whenever OPT applies a transposition $T(x, x')$ or $T(x', x)$ in L, OPT^P applies the same, free as well as paid, transposition in $L_{xx'}$. Whenever OPT accesses an element x in L, OPT^P accesses this x in all lists $L_{xx'}$ where $x \in \underline{L}, x \neq x'$. Let $C_{\mathrm{OPT}^P}(L_{xx'}, I_{xx'})$ denote the cost (for accessing and paid transposition) incurred by OPT^P in the sub-list $L_{xx'}$.

Obviously, the relative position of x and x' in the list L maintained by OPT always coincides with the relative position in the list $L_{xx'}$ maintained by OPT^P. Consequently, we have:

- Whenever OPT pays one unit for a paid transposition, OPT^P pays one unit in exactly one of these $\binom{n}{2}$ sub-lists,
- whenever OPT pays k units for accessing an element in the $(k+1)$-th position in L, OPT^P pays 1 unit in those k sub-lists $L_{xx'}$ where x' precedes x in L.

Therefore, we have

$$C_{\text{OPT}}(I) = \sum_{x,x' \in \underline{L}, x \neq x'} C_{\text{OPT}^P}(L_{xx'}, I_{xx'}).$$

Notice that OPT is not projective in general: The projected version OPT^P does not have to be optimal on the sub-instance $(L_{xx'}, I_{xx'})$. To see the difference, we recall Example 3.8 and Lemma 3.20.

By Example 3.8 we know that OPT (and thus also the projected OPT^P) is forced to use paid transpositions on some instance (L, I), whereas Lemma 3.20 states that the optimal algorithm on instance $(L_{xx'}, I_{xx'})$ may avoid paid transpositions completely.

Nevertheless, the following inequality holds

$$C_{\text{OPT}}(L_{xx'}, I_{xx'}) \leq C_{\text{OPT}^P}(L_{xx'}, I_{xx'}).$$

This inequality can be used to bound the competitive ratio of projective algorithms.

Corollary 3.34 (Borodin and El-Yaniv,2005)**.** *Let* ALG *be an online projective list accessing algorithm which does not use paid transpositions. If the inequality*

$$C_{\text{ALG}}(L_{xx'}, I_{xx'}) \leq c \cdot C_{\text{OPT}}(L_{xx'}, I_{xx'})$$

holds for every pair (x, x') of different list elements, then ALG *is c-competitive.*

Proof.

$$\begin{aligned}
C_{\text{ALG}}(I) &= \sum_{x,x' \in \underline{L}, x \neq x'} C_{\text{ALG}}(L_{xx'}, I_{xx'}) \\
&\leq \sum_{x,x' \in \underline{L}, x \neq x'} c \cdot C_{\text{OPT}}(L_{xx'}, I_{xx'}) \\
&\leq \sum_{x,x' \in \underline{L}, x \neq x'} c \cdot C_{\text{OPT}^P}(L_{xx'}, I_{xx'}) \\
&= c \cdot C_{\text{OPT}}(I)
\end{aligned}$$

□

Roughly speaking, the projective property of ALG allows one to break the list and request sequence into sub-lists and sub-sequences consisting of only two different elements and compares the behavior as well as the costs of ALG to the costs of OPT in the sub instance. Most algorithms in this chapter are projective. In particular:

Lemma 3.35. TS *is projective.*

Proof. The proof is straightforward. TS changes the relative position of x, x' in $L_{xx'}$ if and only if the latter element in $L_{xx'}$, say x, is requested and x' is requested at most once since the last request to x in $I_{xx'}$. This is the case if and only if x is requested and x' is requested at most once since the last request to x in I. Since x' might not be the first element which precedes x in L and was requested at most once since the last request to x in I, it is possible that TS moves x much more forward than to the position of x'. Nevertheless, x precedes x' in both lists.

Therefore, TS changes the relative position of x, x' in $L_{xx'}$, if and only if TS changes the relative position of x, x' in L. This observation completes the proof. □

Remark 3.36. *We give an outline of the proof of Theorem 3.29 here and refer to the proof in Borodin and El-Yaniv (2005) for details. Lemma 3.35 shows that* TS *is projective. Thus, we apply Corollary 3.34 and compare only the terms* $C_{\text{TS}}(L_{xx'}, I_{xx'})$ *with* $C_{\text{OPT}}(L_{xx'}, I_{xx'})$ *for every distinct pair* x, x' *of list elements of* L. *One can further partition* $I_{xx'}$ *into sub-sequences, called phases, each of which has one of the forms in Table 3.2 with the last sub-sequence as an exception, which might be a proper prefix of one of these six forms.*

Phase type I	TS	OPT	Phase type II
$(x)^i x' x'$	2	1	$(x')^i x x$
$(x)^i (x'x)^k x' x'$	$2k$	$k+1$	$(x')^i (xx')^k x x$
$(x)^i (x'x)^k x$	$2k-1$	k	$(x')^i (xx')^k x'$

Table 3.2: The phase cost of TS and OPT in the partial cost model.

Recalling Lemma 3.20, the optimal algorithm is fully understood when the list has only two elements, This allows an easy calculation of the costs of TS *and* OPT *for each of these six forms. The competitive ratio for each phase converges to* 2 *for large* k. *This observation completes the proof of Theorem 3.29.*

Among the algorithms proposed in this section, TRANS is the only one without the projective property. The algorithm TRANS was favored in average cost analysis rather than in worst case analysis (like competitiveness in the deterministic case). In average cost analysis, it is assumed that a request sequence is generated randomly using a (typically i.i.d.) stochastic distribution over list elements, and the expected cost of an algorithm is measured. It was expected that the asymptotic behavior of TRANS should be better than MTF, yet MTF converges much faster than TRANS. For details see Hester and Hirschberg (1985) and Rivest (1976). There is one intuitive argument for TRANS: once the list is in a balanced order, say a decreasing order in terms of request frequencies (as in Lemma 3.21), TRANS will not disturb this order dramatically by only applying one transposition each time. However, if MTF is applied, every occasionally requested low-probability element will be moved to the front and causes costs much larger than TRANS as the access costs of subsequent requests will be increased.

The same decreasing order is also attained by FC asymptotically. FC can be seen as an adaptive imitation of the offline algorithm S-OPT in Lemma 3.21. Indeed, its total cost converges to the total cost of S-OPT for i.i.d. distributions. However, FC draws much less attention compared to other projective algorithms.

Until TS was presented, MTF was the only known deterministic algorithm with a competitive ratio of 2. By combining these two algorithms, El-Yaniv (1996) and Schulz

(1998) introduce different families of infinitely many 2-competitive deterministic online algorithms. Since 2-competitiveness is already best possible in the deterministic case with respect to competitive analysis, further research focuses on other measures than competitiveness or on their randomized versions. It is possible to decrease the competitive ratio of an online algorithm to less than 2 by applying randomized techniques, but the measure "competitiveness" is no longer strict in the sense of worst case analysis, which we introduce and analyze in Section 3.5.

3.5 Randomized Algorithms for the List Accessing Problem

The first randomized online algorithm beating the 2-competitiveness is SPLIT, introduced by Irani (1991). SPLIT uses the idea of MTF with a competitiveness of 1.9375. This ratio was beaten three yeas later by BIT (due to Reingold et al. (1994)), having ratio 1.75, which is again a modification of MTF. Thereafter, hybrid algorithms have been developed to further decrease the ratio. To date, the best randomized online algorithm (in terms of competitiveness) is COMB from Albers et al. (1995), which successfully combines BIT and TS to achieve a competitive ratio of 1.6.

In order to measure the behavior of a randomized algorithm, one has to extend the notion of c-competitiveness.

In the deterministic case, the inequality in Definition 1.1 must hold for *every* instance (L, I). This can be interpreted as if there exists a malicious *adversary* with full knowledge of the deterministic algorithm. The adversary always produces such a crucial instance (L, I) such that $\frac{C_{\text{ALG}}(I)}{C_{\text{OPT}}(I)}$ is maximized.

In the randomized case, uncertainty produced by algorithms makes it difficult to identify which request sequence is the crucial one. Standard literature provides two different kinds of adversaries. An *oblivious adversary* must construct the instance in advance based only on the description of the online algorithm but before any moves are made. An *adaptive adversary* can issue requests based on the online algorithms answers to previous ones. The difference between these two kind of adversaries is illustrated in Example 3.38 using the algorithm RMTF.

Definition 3.37. *Let Randomized MTF (short:* RMTF*) denote the algorithm choosing equally one of the following two actions after each access:*

1. *Move the accessed list element to the first position.*

2. *Do nothing.*

Example 3.38. *Let $L = (1, 2, 3)$ be a list and assume that* RMTF *is applied to solve the list accessing problem on L. In order to maximize the cost of* RMTF*, an adversary will always try to request the last element in the current list. This is not possible in general for oblivious adversaries, as they do not observe the random choice made by* RMTF*. There are two request sequences with length 2 which might achieve the maximum cost of*

6: The request sequence $(3,3)$ *incurs a cost of* 6 *if* RMTF *decides to do nothing, or a cost of* 4 *if* 3 *is moved to the front. Thus, the crucial request sequence (from an oblivious adversary) with length* 2 *will be* $(3,2)$ *with an expected cost of* $\frac{1}{2} \cdot (6+5) = 5.5$.

Compared to this, an adaptive adversary requests the element 3 *at the first round and waits for the coin flipping result of* RMTF*. The request sequence provided at the end is either* $(3,2)$ *or* $(3,3)$ *depending on whether* RMTF *reorganizes the list. In both case, the total cost* RMTF *has to pay is always* 6.

The information about the random choice made by randomized algorithm increases the power of adversary significantly. By using an averaging technique similar as in the proof of Theorem 3.25, Reingold et al. (1994) prove that there always exists a crucial request sequence I, such that the cost incurred by any online algorithm against an adaptive adversary is at least 2 times the cost of an offline optimal algorithm serving I.

Thus, we restrict our view to the analysis of oblivious adversary.

Definition 3.39. *A randomized online algorithm* RALG*, distributed over a set* $\{\text{ALG}_k\}$ *of deterministic online algorithms, is c-competitive against an oblivious adversary for some* $c \geq 1$*, if*

$$\mathbb{E}_K[C_{ALG_k}(I)] \leq c \cdot C_{OPT}(I)$$

holds for all instances (L, I)*. Here, the expression* $\mathbb{E}_K[C_{ALG_k}(I)]$ *denotes the expectation with respect to the probability distribution* K *over* $\{\text{ALG}_k\}$ *which defines* RALG.

It is important to observe that the competitive ratio against an oblivious adversary is not a worst case measure. By taking the expectation, it is closer to the average competitive ratio of a set of deterministic algorithms, as mentioned in Kamali and López-Ortiz (2013).

To understand the concept of $\{\text{ALG}_k\}$, we compare RMTF with algorithm BIT, which is another well known variant of MTF proposed by Reingold et al. (1994).

Definition 3.40. *Let* $L = (x_i)_{i=1,\dots,n}$ *be a list and* I *be a request sequence of* L*. Before serving* I*, the algorithm* BIT *assigns an initial* ***bit value*** $b_x \in \{0,1\}$ *for each* $x \in \{x_1, \dots, x_n\}$ *independently and uniformly. The* ***bit setting*** $b(L) = (b_{x_i})_{i=1,\dots,n}$ *is the sequence of bit values of elements in* L*.*

After accessing x*, the corresponding bit value* b_x *will be complemented* $(b_x \leftarrow 1 - b_x)$*. Then* BIT *moves* x *to the front if and only if* $b_x = 1$*.*

Instead of flipping a coin every time when a list element is accessed, the algorithm BIT alternates between moving the accessed list element to the first position and "Do nothing".

Although BIT and RMTF seem to be similar, the sets $\{\text{BIT}_k\}$ and $\{\text{RMTF}_k\}$ differ significantly from each other. More importantly, these two algorithms differ from each other in terms of competitive ratio[1].

[1]See Section 3.6 for details.

Example 3.41. *Let $L = (1, 3, 2)$ and $I = (2, 3, 2, 3)$.*

The set $\{\mathrm{BIT}_k\}$ can be represented by $\{0, 1\}^n$. In this example, every element in $\{0, 1\}^3$ can be identified with a bit setting made by BIT*. Assume that* BIT *assigned $(0, 1, 0) \in \{0, 1\}^3$ as bit setting to L. Then the elements $x_1 = 1$ and $x_3 = 2$ will be moved to the front after they are requested odd times, whereas $x_2 = 3$ will be moved to the front after it is requested even times.*

The set $\{\mathrm{RMTF}_k\}$ can be represented by $\{0, 1\}^m$. In this example, every element in $\{0, 1\}^4$ can be identified with a set of four random choices made by RMTF*. Assume that* RMTF *has decided to move the first accessed and the last accessed element to the front, no matter which list element is requested. Then this decision can be encoded by $(1, 0, 0, 1) \in \{0, 1\}^4$.*

Corollary 3.42. *For given L and I with $|L| = n$ and $|I| = m$, we have $|\{\mathrm{BIT}_k\}| = 2^n$ and $|\{\mathrm{RMTF}_k\}| = 2^m$. Both randomized algorithms use a discrete uniform distribution to draw from the corresponding set of deterministic algorithms.*

Proof. The cardinalities of $\{\mathrm{BIT}_k\}$ and $\{\mathrm{RMTF}_k\}$ follow immediately from the identification in Example 3.41. Since BIT and RMTF produce their decisions uniformly, it is easy to see that a discrete uniform distribution is applied on $\{\mathrm{BIT}_k\}$ and $\{\mathrm{RMTF}_k\}$, respectively. □

RMTF and BIT essentially only use the idea of MTF. They can be seen as a combination of MTF and "Do nothing". In the same work in which BIT is proposed, Reingold et al. (1994) generalize the idea of when to "Do nothing", resulting in a family of randomized algorithms, called COUNTER. Roughly speaking, a COUNTER(s, S)-algorithm maintains an integer value from $\{0, 1, \ldots, s - 1\}$ for every list element. The set S is a subset of $\{0, 1, \ldots, s - 1\}$. Similar to BIT, the counter of x is decreased by 1 after every access to x. The move to the front action will be triggered whenever the corresponding counter has reached a number in the set S. It is easy to see that BIT is actually a COUNTER$(2, \{0\})$-algorithm.

Remark 3.43. *The advantage of "Do nothing" can be explained as follows:* MTF *performs poorly if every list element is requested only once in a reversed order as the initial list, say $I = (x_n, x_{n-1}, \ldots, x_1)$. To serve I, "Do nothing" is indeed the optimal strategy. Conversely, the crucial request sequence against "Do nothing" is $I_n = (x_n, \ldots, x_n)$ where* MTF *behaves optimal.*

The improvement of the competitive ratio from MTF *to* BIT *results actually from the somehow "complementary" performance of these two behaviors.*

Beyond MTF *with "Do nothing", this observation motivates hybrid algorithms with* ALG_1 *and* ALG_2*, such that* ALG_2 *performs well on the request sequences incurring a high ratio $\frac{C_{\mathrm{ALG}_1}(I)}{C_{\mathrm{OPT}}(I)}$ and vice versa.*

3.6 Bounds for Competitiveness of Randomized Online Algorithms

Proposition 3.44 (Borodin and El-Yaniv,2005)**.** *The algorithm RMTF is not better than 2-competitive in expectation.*

The original proof in Borodin and El-Yaniv (2005) involves the request sequence

$$I = (I^l_n, I^l_{n-1}, \dots, I^l_1),$$

where I^l_i is the sub-sequence consisting of l requests to the same list element x_i. We call such a sub-sequence a *block*.

To prove the lower bound of competitiveness of RMTF, Borodin and El-Yaniv (2005) compare the performance of RMTF to the performance of MTF, similar as in Example 3.26. The cost $C_{\text{MTF}}(I)$ can be calculated explicitly as follows.

Corollary 3.45 (Borodin and El-Yaniv,2005)**.** *The cost of applying* MTF *to serve I is equal to $C_{\text{MTF}}(I) = n(n + l - 1)$.*

Proof. By the behavior of MTF, the requested element x_k in I^l_k is actually in the last position of L when x_k is requested for the first time in I^l_k. Since the cost $C_{\text{MTF}}(I_k)$ is constant $n + l - 1$ for every $k = 1, \dots, n$, we have $C_{\text{MTF}}(I) = n(n + l - 1)$. □

For RMTF, Borodin and El-Yaniv (2005) write on page 27 (notation adjusted):

> For large l, with high probability, algorithm RMTF will move x_i to the front while RMTF services the segment $(x_i)^l$. On average, x_i is moved to the front at the second request. Hence, the expected cost for RMTF to serve I is at least $2n^2 + 2nl - 2n$.

We have observed that this estimation is not really precise for small instances.

Example 3.46. *First consider the cases $n = l = 2$, i.e. we have the instance $(L = (x_1, x_2), I = (x_2, x_2, x_1, x_1))$. We use the identification in Example 3.41 to denote the random choices of* RMTF*. It is possible that some decisions of* RMTF *do not influence the overall cost. In order to simplify the case analysis, we use $*$ instead of* $0, 1$ *to indicate such decisions.*

E.g. the tuple $(1, 0, 1, *)$ *denotes the set* $\{(1, 0, 1, 0), (1, 0, 1, 1)\}$ *of two decisions, each of which moves the first and third accessed element to the front, but leaves the second accessed element where it is. It is easy to see that before the fourth accessing, the list is* (x_1, x_2) *and thus the overall cost will be* $2 + 1 + 2 + 1 = 6$*, no matter whether* RMTF *moves the last requested element to the front or not.*

We partition $\{0, 1\}^4$ *into the following subsets:*

$$\{0, 1\}^4 = (0, 0, *, *)\dot\cup(0, 1, 0, *)\dot\cup(0, 1, 1, *)\dot\cup(1, *, 0, *)\dot\cup(1, *, 1, *).$$

The probability and the corresponding incurred cost of these subsets are given in Table 3.3a. Further consider the case $n = 2, l = 3$, where a similar subdivision is given by

$$\begin{aligned}\{0,1\}^6 =&(0,0,0,*,*,*)\\ &\dot\cup(0,0,1,0,0,*)\dot\cup(0,0,1,0,1,*)\dot\cup(0,0,1,1,*,*)\\ &\dot\cup(0,1,*,0,0,*)\dot\cup(0,1,*,0,1,*)\dot\cup(0,1,*,1,*,*)\\ &\dot\cup(1,*,*,0,0,*)\dot\cup(1,*,*,0,1,*)\dot\cup(1,*,*,1,*,*)\end{aligned}$$

together with the corresponding probability and cost in Table 3.3b.

Subset	$(0,0,*,*)$	$(0,1,0,*)$	$(0,1,1,*)$	$(1,*,0,*)$	$(1,*,1,*)$
Probability	1/4	1/8	1/8	1/4	1/4
Cost	6	8	7	7	6

(a) $L = (x_1, x_2)$, $I = (x_2, x_2, x_1, x_1)$.

Subset	$(0,0,1,0,0,*)$	$(0,0,1,0,1,*)$	$(0,0,1,1,*,*)$	$(0,0,0,*,*,*)$
Probability	1/32	1/32	1/16	1/8
Cost	12	11	10	9
Subset	$(0,1,*,0,0,*)$	$(0,1,*,0,1,*)$	$(0,1,*,1,*,*)$	
Probability	1/16	1/16	1/8	
Cost	11	10	9	
Subset	$(1,*,*,0,0,*)$	$(1,*,*,0,1,*)$	$(1,*,*,1,*,*)$	
Probability	1/8	1/8	1/4	
Cost	10	9	8	

(b) $L = (x_1, x_2)$, $I = (x_2, x_2, x_2, x_1, x_1, x_1)$.

Table 3.3: The cost of RMTF for small example instances.

We omit their calculations here, since the numbers in Table 3.3 can be verified easily. It is then easy to see that the expected cost of RMTF *for the cases $n = l = 2$ and $n = 2, l = 3$ is* 6.625 *and* 9.28125 *respectively, whereas the lower bound in Borodin and El-Yaniv (2005) offers* 12 *and* 16*, respectively.*

Notice that Example 3.46 does not refute the lower bound in Borodin and El-Yaniv (2005), as large l is assumed for its validity. Still, this motivates a closer analysis to find the minimal l and n such that the estimation holds. We provide an explicit calculation of the expected cost of RMTF, which, strangely, leads to an upper bound $2n^2 + ln - 2n$ being less than the back of an envelope lower bound from Borodin and El-Yaniv (2005). For the sake of clarity, we divide the calculation into several lemmas. Until end of this calculation, the expectations are calculated over $\{\text{RMTF}_k\}$, i.e. all possible decisions made by RMTF.

Lemma 3.47. *The expected cost $\mathbb{E}[C_{\text{RMTF}}(I_n^l)]$ of serving the first block in I is equal to $2n + l - 2 - \frac{2n-2}{2^l}$.*

Proof. If x_n is moved to the front while serving I_n^l, let j be the smallest index at which RMTF decides to move x_n to the front after accessing the j-th request in I_n^l. Obviously, the first j accesses cost n and other accesses, independent on the random decisions of RMTF after the j-th decision, only cost 1. It is easy to see that the probability of this event is equal to $\left(\frac{1}{2}\right)^j$ for all $j = 1, \ldots, n$. In the case x_n is not moved to the front until the end of serving I_n^l, which has a probability of $\left(\frac{1}{2}\right)^l$, the cost incurred by RMTF is equal to nl.

Therefore, we have the following equation:

$$\begin{aligned}
\mathbb{E}[C_{\text{RMTF}}(I_n^l)] =& nl \cdot \left(\frac{1}{2}\right)^l + \sum_{j=1}^{l} (nj + 1 \cdot (l - j)) \cdot \left(\frac{1}{2}\right)^j \\
=& \frac{1}{2^l} nl + l \cdot \sum_{j=1}^{l} \frac{1}{2^j} + (n-1) \cdot \sum_{j=1}^{l} \frac{j}{2^j} \\
\overset{2}{=}& \frac{1}{2^l} nl + l \left(1 - \frac{1}{2^l}\right) + (n-1) \left(2 - \frac{l+2}{2^l}\right) \\
=& \left(1 - \frac{1}{2^l}\right) \cdot (l + 2n - 2) - \frac{l}{2^l}(n-1) + \frac{1}{2^l} nl \\
=& \left(1 - \frac{1}{2^l}\right)(2n - 2) + l - \frac{l}{2^l} - \frac{nl}{2^l} + \frac{l}{2^l} + \frac{nl}{2^l} \\
=& 2n + l - 2 - \frac{2n-2}{2^l}.
\end{aligned}$$

□

Let L_k^{RMTF} denote the list maintained by RMTF after serving the k-th block I_{n-k+1}^l. Certainly, the list L_k^{RMTF} may differ from L depending on the decision made by RMTF. Recall that the notion $C_{\text{ALG}}(L, I)$ is defined to be the cost of serving the instance (L, I) using the algorithm ALG. In the case that the initial list is not $L = (x_1, \ldots, x_n)$, it is provided explicitly.

Remark 3.48. *Consider the list L_1^{RMTF}. With a probability of $1 - \left(\frac{1}{2}\right)^l$, the list element x_n is moved to the front while serving I_n^l. In such a case, the list order before serving the next block I_{n-1}^l is*

$$L_1^{\text{RMTF}} = (x_n, x_1, x_2, \cdots, x_{n-1}).$$

Notice that the element x_{n-1}, requested by the next block I_{n-1}^l, is again the last element in the list immediately before serving I_{n-1}^l. Hence, the cost of serving the next block remains unchanged.

$$\mathbb{E}[C_{\text{RMTF}}(L_1^{\text{RMTF}}, I_{n-1}^l) \,|\, x_n \textit{ is in position } 1 \textit{ after serving } I_n^l] = \mathbb{E}[C_{\text{RMTF}}(L, I_n^l)].$$

[2] Consider the function $f(x) = \frac{1}{2} \sum_{j=1}^{l} x^j$. Notice that $\sum_{j=1}^{l} \frac{j}{2^j}$ is actually equal to $\frac{d}{dx} f(\frac{1}{2})$. Thus, this term can be calculated by adjusting and differentiating the formula of geometric series for f, see for instance Heuser (2013).

If x_n is still in the last position after serving I_n^l, then, since it is not requested thereafter, it remains in the last position even after serving I, i.e. one can simply delete x_n from the list without changing the cost. More precisely,

$$\begin{aligned}&\mathbb{E}[C_{\text{RMTF}}(L_1^{\text{RMTF}}, I_{n-1}^l)) \mid x_n \textit{ is in position } n \textit{ after serving } I_n^l]\\ =&\mathbb{E}[C_{\text{RMTF}}((x_1,\dots,x_n), I_{n-1}^l)]\\ =&\mathbb{E}[C_{\text{RMTF}}((x_1,\dots,x_{n-1}), I_{n-1}^l)].\end{aligned}$$

Lemma 3.47 can be applied to the instance $(L = (x_1,\dots,x_{n-1}), I_{n-1}^l)$ to observe

$$\mathbb{E}[C_{\text{RMTF}}((x_1,\dots,x_{n-1}), I_{n-1}^l)] = 2(n-1) + l - 2 - \frac{2(n-1)-2}{2^l}.$$

We can see that if x_n was not moved to the front, RMTF *pays*

$$2n + l - 2 - \frac{2n-2}{2^l} - \left(2(n-1) + l - 2 - \frac{2(n-1)-2}{2^l}\right) = 2\left(1 - \frac{1}{2^l}\right)$$

less in average.

In general, we call a block I_k^l *regular* if x_k is moved to the front while serving I_k^l. Otherwise we call it *singular*. Obviously, every block has a probability of $\frac{1}{2^l}$ of being singular. The analysis in Remark 3.48 reveals that the number of singular and regular blocks influences the list order as well as the expected cost of the blocks thereafter. To analyze this influence we assume some realization of the decisions of RMTF and consider L_k^{RMTF}.

Lemma 3.49. *Consider the set $\{x_{n-k},\dots,x_n\}$ of list elements which are already accessed before serving I_{n-k-1}^l. Let $S_{reg}, S_{sing} \subseteq \{x_{n-k},\dots,x_n\}$ be the subset of list elements whose corresponding block is regular and singular, respectively. Then we have*

$$L_k^{\text{RMTF}} = (\widehat{S_{reg}}, x_1, x_2, \dots, x_{n-k-1}, \widehat{S_{sing}}),$$

where $\widehat{S_{reg}}$ and $\widehat{S_{sing}}$ are the (index-increased) ordered sets of S_{reg} and S_{sing}, respectively.

Proof. By the definition of S_{reg}, every element $x_r \in S_{\text{reg}}$ has been moved to the first position after serving I_r^l. At this point in time, x_r precedes all list elements other than itself. If some list element x precedes x_r in L_k^{RMTF}, then x_t must be moved to the front after serving I_r^l. Thus, we know $x_t \in S_{\text{reg}}$ and the block I_t^l is served later than I_r^l, or equivalently $t < r$, meaning that the elements in S_{reg} are ordered in an index-increased order in L_k^{RMTF}.

Furthermore, every element in S_{reg} precedes all elements in $\underline{L}\backslash S_{\text{reg}}$, since elements in $\underline{L}\backslash S_{\text{reg}}$ have never been moved forward while serving the initial segment $(I_n, I_{n-1}, \dots, I_{n-k})$.

It only remains to show that the relative position of elements $x, x' \in \underline{L}\backslash S_{\text{reg}}$ in L_k^{RMTF} is the same as in L. Let $\mathcal{T}$ be the chain of transpositions applied by RMTF while serving the first k blocks, i.e. $L_k^{\text{RMTF}} = \mathcal{T}(L)$. Recall from Lemma 3.13: In order to change the relative position of (x, x'), the chain $\mathcal{T}$ has to contain $T(x, x')$ or $T(x', x)$ at least once. By the definition, all transpositions applied by RMTF only move the accessed elements forwards. In other words, every transposition in $\mathcal{T}$ involves at least one element in S_{reg}. As $x, x' \in \underline{L} \backslash S_{\text{reg}}$, the transpositions $T(x, x')$ and $T(x', x)$ are not included in $\mathcal{T}$. Therefore, the relative position of (x, x') in $L_k^{\text{RMTF}} = \mathcal{T}(L)$ remains the same as in L. □

With the list order provided in Lemma 3.49, we can calculate the expected cost of serving one block explicitly.

Lemma 3.50. *The expected cost of serving the* $(k+1)$*-th block is given by*

$$\mathbb{E}[C_{\text{RMTF}}(L_k^{\text{RMTF}}, I_{n-k-1}^l)] = \mathbb{E}[C_{\text{RMTF}}(L, I_n^l)] - \frac{k}{2^l} \cdot 2\left(1 - \frac{1}{2^l}\right).$$

Proof. We first claim that there are $\frac{k}{2^l}$ singular blocks within the first k blocks in expectation. Indeed, since every block has a probability $\frac{1}{2^l}$ of being singular independent on each other, the number of singular blocks is binomially distributed with a success probability of $\frac{1}{2^l}$. From Lemma 3.49 we know that

$$L_k^{\text{RMTF}} = (\widehat{S_{\text{reg}}}, x_1, x_2, \ldots, x_{n-k-1}, \widehat{S_{\text{sing}}}),$$

where the expected cardinality of S_{sing} is equal to $\frac{k}{2^l}$. By Remark 3.48 we have

$$\begin{aligned} C_{\text{RMTF}}(L_k^{\text{RMTF}}, I_{n-k-1}^l) =& \mathbb{E}[C_{\text{RMTF}}((\widehat{S_{\text{reg}}}, x_1, x_2, \ldots, x_{n-k-1}), I_{n-k-1}^l)] \\ =& \mathbb{E}[C_{\text{RMTF}}(L, I_n^l)] - \frac{k}{2^l} \cdot 2\left(1 - \frac{1}{2^l}\right). \end{aligned}$$

□

Corollary 3.51.

$$\mathbb{E}[C_{\text{RMTF}}(L, I)] = 2n^2 + nl - 2n - \frac{n(n-1)}{2^l}\left(3 - \frac{1}{2^l}\right).$$

Proof. This corollary is a direct consequence from Lemma 3.50. Let L_0^{RMTF} denote the initial list L. Then we have

$$\begin{aligned}
\mathbb{E}[C_{\mathrm{RMTF}}(L, I)] =& \sum_{k=0}^{n-1} \mathbb{E}[C_{\mathrm{RMTF}}(L_k^{\mathrm{RMTF}}, I_{n-k}^l)] \\
=& n\mathbb{E}[C_{\mathrm{RMTF}}(L, I_n^l)] - \sum_{k=0}^{n-1} \frac{k}{2^l} \cdot 2 \left(1 - \frac{1}{2^l}\right) \\
=& n\mathbb{E}[C_{\mathrm{RMTF}}(L, I_n^l)] - \frac{n(n-1)}{2^l} \left(1 - \frac{1}{2^l}\right) \\
=& n(2n + l - 2 - \frac{2n-2}{2^l}) - \frac{n(n-1)}{2^l} \left(1 - \frac{1}{2^l}\right) \\
=& 2n^2 + nl - 2n - \frac{n(n-1)}{2^l} \left(3 - \frac{1}{2^l}\right). \qquad \square
\end{aligned}$$

Thus, the expected cost of RMTF is less than $2n^2 + nl - 2n$ for all n and l. Nevertheless, by combining the result of Corollary 3.51 and 3.45, we conclude that the quotient $\frac{\mathbb{E}[C_{\mathrm{RMTF}}(I)]}{C_{\mathrm{MTF}}(I)}$ converges to 2 for $n \gg l$. This result coincides with the proposition in Borodin and El-Yaniv (2005).

Recall from Theorem 3.27 that the randomization technique by RMTF does not improve the competitiveness at all. In contrast, the competitive ratio of BIT is somewhere between 1.625 and 1.75. With Lemma 3.53, Reingold et al. (1994) prove the upper bound. The lower bound is provided in Example 3.55.

Theorem 3.52 (Reingold et al.,1994)**.** *The algorithm* BIT *is at least 1.75-competitive against an oblivious adversary.*

Lemma 3.53 (Reingold et al.,1994)**.** *For any list element x and any $k \in \mathbb{N}$, the bit value b_x that occurs directly after event k is equally likely to be* 0 *or* 1*, is independent of the position of x in L^{OPT}, and is independent of b_x of every other list element x'.*

The same proof technique is applied to Theorem 3.52 as in the proof of Theorem 3.27. Recall that L^{BIT} and L^{OPT}, respectively, denote the list maintained by BIT and OPT. The standard proof splits the process of serving a request into five steps.

1. request y_j arrives,
2. BIT accesses y_j in the list L^{BIT},
3. OPT accesses y_j in the list L^{OPT},
4. BIT performs free transpositions (if any),
5. OPT performs free transpositions and paid transpositions (if any).

Step 2 increases the cost of BIT, steps 3 and 5 increase the cost of OPT, and steps 4 and 5 reorganize the list which may change the value of the potential function.

The intuition of the potential function in the randomized case is still to capture the potential cost of reorganizing L^{BIT} to L^{OPT}, see Remark 3.28. Since BIT is randomized, the potential function has to be modified. For any given inversion (x, x') regarding $(L^{\text{BIT}}, L^{\text{OPT}})$, the weight $w(x, x')$ is defined as the number of accesses to x' before x' passes x in L^{BIT}. In other words, $w(x, x') = b_{x'} + 1$. Correspondingly, the potential function is defined by

$$\Phi(L^{\text{BIT}}, L^{\text{OPT}}) = \sum_{(x,x') \text{ is inversion}} w(x, x').$$

Remark 3.54. *In the proof of Theorem 3.52, steps* 2 *to* 5 *of every request will be regrouped into different events such that each action (access or transposition) belongs to exactly one event. Let $c_{\text{BIT},k}$ and $c_{\text{OPT},k}$, respectively, denote the change of cost of* BIT *and* OPT *during the event k. It is then easy to see $C_{\text{BIT}}(I) = \sum_{k=1}^{K} c_{\text{BIT},k}$ and $C_{\text{OPT}}(I) = \sum_{k=1}^{K} c_{\text{OPT},k}$ where the number K of events depends on the number of paid transpositions made by* OPT.

For an event k, let Φ_k denote the value of the potential function after event k. Since both algorithms BIT *and* OPT *start with the same list, there are no inversions before the first event and hence we let $\Phi_0 = 0$. The amortized cost of an event k is defined by*

$$a_k = c_{\text{BIT},k} + \Phi_k - \Phi_{k-1}$$

leading to

$$C_{\text{BIT}}(I) = \sum_{k=1}^{K} c_{\text{BIT},k} = \left(\sum_{k=1}^{K} a_k\right) - \Phi_K + \Phi_0 \leq \sum_{k=1}^{K} a_k,$$

as Φ_K is non-negative by definition.

Therefore, in order to prove the $\frac{7}{4}$-competitiveness of BIT, *it is sufficient to show $a_k \leq \frac{7}{4} \cdot c_{\text{OPT},k}$ for every event k.*

Recall from Proposition 3.9, that OPT may avoid free transpositions. Based on the proof of Theorem 3.52 in Borodin and El-Yaniv (2005), we next present a simplified version where one of the cases for b_x becomes trivial.

Proof of Theorem 3.52. Let I be a given request sequence of L. Assume that the j-th requested list element $x = y_j$ is in the i-th position in L^{BIT} and at i'-th position in L^{OPT} at the time of step 1. For the sake of clarity, we number the list element according to their position in L^{OPT}.

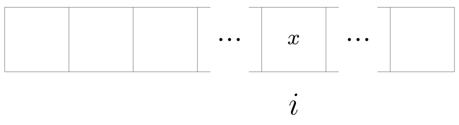
Notice that before step 3, the situation is exactly the same as in the proof of Theorem 3.27. In particular, the same claim holds here.

<u>Claim:</u> Let g and h, respectively, be the number of inversions $(\cdot, x)$ and $(x, \cdot)$, and let z be the number of list elements preceding x in both of L^{BIT} and L^{OPT}. Then the following

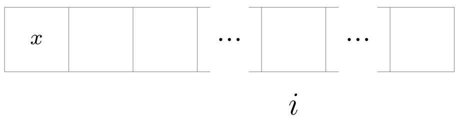
relations hold:

$$z + g = i - 1 \tag{3.1}$$
$$z + h = i' - 1 \tag{3.2}$$
$$i \leq i' + g \tag{3.3}$$
$$i' \leq i + h \tag{3.4}$$

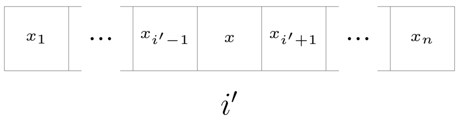
Consider steps 2, 3, and 4 together as one event. Let L_4^{BIT} denote the list maintained by BIT after this event, see Figure 3.3. Notice that list L_4^{BIT} depends on the bit value b_x before BIT accesses x.



(a) L^{BIT} after step 1, L_4^{BIT} if $b_x = 1$.



(b) L_4^{BIT} if $b_x = 0$.



(c) L^{OPT} after step 1 and after step 4.

Figure 3.3: Lists in the proof of Theorem 3.52.

The amortized cost of this event is defined as

$$a_k = c_{\text{BIT},k} + \Delta\Phi,$$

where $\Delta\Phi = \Phi(L_4^{\text{BIT}}, L^{\text{OPT}}) - \Phi(L_1^{\text{BIT}}, L^{\text{OPT}})$.

To analyze $\Delta\Phi$, we express the change as $\Delta\Phi = A + B + C$ where A, B, and C are random variables: A is the contribution of new inversions created, B is the (negative) contribution of eliminated inversions, and C is the contribution from inversions which are already inversions regarding $(L_1^{\text{BIT}}, L^{\text{OPT}})$ and remain inversions regarding $(L_4^{\text{BIT}}, L^{\text{OPT}})$, but change their weight from 2 to 1.

Claim: All inversions counted by A, B, and C are either of the form $(\cdot, x)$ or $(x, \cdot)$.

Proof: Consider a pair of list elements x', x'', where $x \notin \{x', x''\}$. Notice that the list L_4^{BIT} is obtained by applying a chain of free transpositions $(\cdot, x)$ on L_1^{BIT}. Clearly, the transpositions $T(x', x'')$ and $T(x'', x')$ are not included. By Lemma 3.13, the relative position of list elements x' and x'' remains unchanged after applying these free transpositions. Hence, (x', x'') is an inversion regarding $(L_1^{\text{BIT}}, L^{\text{OPT}})$ if and only if it is an inversion regarding $(L_4^{\text{BIT}}, L^{\text{OPT}})$. Thus, the pair (x', x'') is not counted by A and B. It can neither be counted by C, since the weight of (x', x'') can only be changed if the bit value $b_{x''}$ is changed, whereas x is the only element whose bit value is changed during this event. ■

Claim:

$$B + C = -g \tag{3.5}$$

Proof: First consider the inversions $(\cdot, x)$ counted by g.

In the case $b_x = 1$, the list element x remains in position i, meaning that $L_4^{\mathrm{BIT}} = L^{\mathrm{BIT}}$. After BIT accesses x, the bit value b_x is decreased from 1 to 0. Hence, inversions $(\cdot, x)$ reduce their weight from 2 to 1 and consequently $C = -g$. Since the list order is exactly the same after this event as before, we conclude that $B = 0$.

In the case $b_x = 0$, the list element x is moved to the front of L_4^{BIT}. All inversions $(\cdot, x)$ counted by g are eliminated and no inversions $(x, \cdot)$ can be eliminated, as no list element precedes x in L_4^{BIT}. Clearly, we have $C = 0$ and $B = -g$. ■

By the equations (3.3) and (3.5), the expectation of amortized cost over all bit settings can be written as

$$\mathbb{E}[a_k] = \mathbb{E}[c_{\mathrm{BIT},k} + \Delta\Phi] = \mathbb{E}[i + A - g] \leq \mathbb{E}[i' + g + A - g] = i' + \mathbb{E}[A]. \tag{3.6}$$

Claim:

$$\mathbb{E}[A] \leq \frac{3}{4} \cdot (i' - 1) \tag{3.7}$$

Proof: The list L_4^{BIT} depends on b_x, which is a Bernoulli random variable with $p = \frac{1}{2}$ by Lemma 3.53. We give an upper bound for $\mathbb{E}[A]$ in both cases and show that the mean of these two upper bounds is less than or equal to $\frac{3}{4} \cdot (i' - 1)$.

In the case $b_x = 1$, we have $L_4^{\mathrm{BIT}} = L_1^{\mathrm{BIT}}$. The lists maintained by both algorithms are exactly the same. Thus, $A = 0$. In the case $b_x = 0$, x is moved to the first position in L_4^{BIT} and thus precedes all other list elements, see Figure 3.3. Every created inversion must consist of a pair $(x, \overline{x})$ where $\overline{x}$ precedes x in L^{OPT}. Clearly, such $\overline{x}$ belongs to $\{x_1, ..., x_{i'-1}\}$. By Lemma 3.53, the weight $w(x, \overline{x})$ is discrete uniformly distributed over $\{1, 2\}$. Hence, we conclude

$$\mathbb{E}[A | b_x = 0] \leq \sum_{k=1}^{i'-1} \mathbb{E}[w(x, x_k)] = \sum_{k=1}^{i'-1} \left(\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 \right) = \frac{3}{2}(i' - 1). \tag{3.8}$$

Since b_x is uniformly distributed on $\{0, 1\}$, Equation (3.8) completes the proof:

$$\mathbb{E}[A] \leq \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot \frac{3}{2} \cdot (i' - 1) \leq \frac{3}{4}(i' - 1). \tag{3.9}$$

■

Recall the assumption that the request $y_j = x$ is in the i'-th position in L^{OPT} and therefore, the access cost $c_{\mathrm{OPT},k}$ of OPT is equal to i'. Then we have

$$\mathbb{E}[a_k] = i' + \mathbb{E}[A] \leq \frac{7}{4} \cdot i' - \frac{3}{4} < \frac{7}{4} c_{\mathrm{OPT},k}. \tag{3.10}$$

It remains to analyze the paid transpositions of OPT. Consider a single paid transposition $T(x', x'')$ in step 5 as an event. The change $c_{\mathrm{BIT},k}$ of the BIT cost is certainly 0, whereas $c_{\mathrm{OPT},k}$ is equal to 1. It is easy to see that (x', x'') is the only possible inversion which may be created during this event. Again by Lemma 3.53, the weight $w(x', x'')$ is uniformly distributed oner $\{1, 2\}$, hence:

$$\mathbb{E}[a_k] = \mathbb{E}[c_{\mathrm{BIT},k} + \Delta\Phi] \leq 0 + \mathbb{E}[A] = 0 + \left(\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2\right) = \frac{3}{2} < \frac{7}{4} \cdot c_{\mathrm{OPT},k}. \quad (3.11)$$

To sum up, we conclude that the amortized cost a_k is less than or equal the $\frac{7}{4} \cdot c_{\mathrm{OPT},k}$ for all events k. The proof is completed by Remark 3.54. □

Example 3.55 (Albers and Mitzenmacher,1997)**.** *Define the request sequences R_1 and R_2 for $L = (x_1, \dots, x_n)$ as follows:*

$$R_1 = (x_1, x_2, \dots, x_n, I_1^l, I_2^l, \dots, I_n^l), R_2 = (n, n-1, \dots, 1, I_n^l, I_{n-1}^l, \dots, I_1^l),$$

where I_k^l is a sequence of l requests to the same list element x_k. Albers and Mitzenmacher (1997) prove that BIT *achieves only a competitive ratio larger than $\frac{13}{8}$ on a crucial request sequence which request R_1 and R_2 alternatively.*

For a detailed proof we refer to the original work of Albers and Mitzenmacher (1997).

If one analyzes the proof of Theorem 3.52 further, we end up with Conjecture 3.56.

Conjecture 3.56. *Using the notation in the proof of Theorem 3.52, an example request sequence I of better lower bounds for* BIT *may be found by considering those sequences, which have the following properties in as many requests as possible:*

1. *If x is requested, then $i' \leq i$.*

   This means that accessing x costs OPT *less than* BIT*.*

2. *The entire initial segment of L^{OPT} up to x is contained in the initial segment of L^{BIT} up to x.*

   This reduces the gap of Inequality (3.8)*.*

3. OPT *does not apply too many paid transpositions.*

   As shown in the proof, events consisting of one paid transposition has a competitive ratio of only 1.5*.*

Besides of "Do nothing", it is possible to combine the idea of MTF with other algorithms. For combinations of algorithms from COUNTER, a tight lower bound $\frac{12}{7}$ of competitive ratio is provided in Albers and Mitzenmacher (1997). For combinations of projective algorithms, a tight lower bound $\frac{8}{5}$ of competitive ratio is achieved by COMB.

Before any request of request sequence I is revealed, the randomized algorithm COMB chooses one of the following algorithms

- BIT with a probability of 80%, and
- TS with a probability of 20%

to serve I.

Theorem 3.57 (Albers et al.,1995). *The algorithm* COMB *is at least 1.6-competitive against an oblivious adversary in the partial cost model.*

The list factoring technique, used in Remark 3.36 to prove Theorem 3.29, can be applied to prove Theorem 3.57.

Recall that the proof of Theorem 3.29 relies on the projective property of TS, which is only defined for deterministic algorithms. We first extend this definition to the randomized case.

Definition 3.58. *A randomized online algorithm* RALG*, distributed over a set* $\{\text{ALG}_k\}$ *of deterministic online algorithms, is projective if and only if* ALG_k *is projective for all* k.

Obviously, Corollary 3.34 applies also for randomized projective algorithms.

Corollary 3.59. *Let* RALG *be an online projective list accessing algorithm which does not use paid transpositions. If the inequality* $C_{\text{RALG}}(L_{xx'}, I_{xx'}) \leq c \cdot C_{\text{OPT}}(L_{xx'}, I_{xx'})$ *holds for every pair* (x, x') *of different list elements, then* RALG *is c-competitive.*

Remark 3.60. *Observe that* BIT *is a randomized algorithm distributed over the set* {MTF, *"Do nothing"*}. *It is easy to see that* MTF *and "Do nothing" are both projective. Hence,* BIT *is projective. Being a randomized algorithm distributed over* $\{\text{BIT}, \text{TS}\}$, COMB *inherits the projective property from* BIT *and* TS. *This observation can be used to prove Theorem 3.57, which is pretty similar to Remark 3.36. We give an outline here.*

Recall from Remark 3.36 that $I_{xx'}$ *can be partitioned into phases. Similar as for Table 3.2, we calculate the cost incurred by* BIT *for serving each phase. It is easy to see that*

Phase type I	TS	BIT	OPT	Phase type II
$(x)^i x' x'$	2	$\frac{3}{2}$	1	$(x')^i xx$
$(x)^i (x'x)^k x' x'$	$2k$	$\frac{3}{2}k + 1$	$k + 1$	$(x')^i (xx')^k xx$
$(x)^i (x'x)^k x$	$2k - 1$	$\frac{3}{2}k + \frac{1}{4}$	k	$(x')^i (xx')^k x'$

Table 3.4: The phase costs of TS and OPT in the partial cost model.

the weighted sum of TS *and* BIT *is bounded by* $\frac{8}{5}$ *times of the cost of* OPT *for large* k, *as claimed in Theorem 3.57.*

To see the correctness of the values in Table 3.4, we refer to the proof in Borodin and El-Yaniv (2005) in section 2.4.

Furthermore, randomized projective algorithms can not be better than $\frac{8}{5}$-competitive:

Theorem 3.61 (Ambühl et al.,2010). *Randomized projective online algorithms for list accessing problem are no better than* $\frac{8}{5}$*-competitive in the partial cost model.*

Chapter 4

Closer Randomized Analysis of BIT

This chapter focuses on the stochastic analysis of BIT. Request sequences considered in this chapter consist of requests which are i.i.d. over the set of list elements. In Section 4.1, uniform distribution is used to generate requests. It turns out that the cost of BIT can be simulated by throwing a fair die several times and counting the sum of the resulting points, independent of the initial bit setting. A further analysis of the distribution of bit values is performed in this section, resulting in an improved version of Conjecture 3.56. In Section 4.2, a formula for the expected cost of BIT is developed for request sequences generated by discrete distribution. The last section of this chapter provides a brief view of a more general case, which occurs naturally in the context of data compression.

4.1 Uniform Distribution

4.1.1 The Expected Cost and Variance of $C_{\mathrm{BIT}}(b, I)$

Lemma 4.1. *Let L be a list of length n and $\mathcal{I}_m$ be the set of all possible access sequences with length m. Let $C_{\mathrm{BIT}}(b, I)$ be the cost of applying* BIT *with bit setting $b = b(L)$ to serve $I \in \mathcal{I}_m$. Then the equation*

$$\sum_{I \in \mathcal{I}_m} C_{\mathrm{BIT}}(b, I) = \frac{1}{2} n(n+1) \cdot m n^{m-1}$$

holds for all possible bit settings $b = b(L)$.

Proof. We prove the claim by induction over m.

For the base case $m = 1$, observe that $\sum_{I \in \mathcal{I}_1} C_{\mathrm{BIT}}(b, I) = \sum_{l \in L} C_{\mathrm{BIT}}(b, l)$ is the sum of costs of requesting every element from L separately. It is easy to see that

$$\sum_{x \in \underline{L}} C_{\mathrm{BIT}}(b, (x)) = \frac{1}{2} n(n+1).$$

For the inductive step assume that the claim holds for some integer m. The set $\mathcal{I}_{m+1}$ can be partitioned into disjoint subsets:

$$\mathcal{I}_{m+1} = \dot{\bigcup}_{I\in\mathcal{I}_m} \{J = (I,l) \,|\, l \in L\},$$

i.e. the sequences in $\mathcal{I}_{m+1}$ are categorized by its first m requests. In order to avoid possible ambiguities, the sequences in $\mathcal{I}_{m+1}$ are denoted by J.

For any $J \in \{(I,l) \,|\, I \in \mathcal{I}_m,\, l \in L\}$, the cost $C_{\mathrm{BIT}}(b,J)$ is simply equal to $C_{\mathrm{BIT}}(b,I)$ plus the cost of finding l in L', where L' is the list after serving I on L. Hence the summation $\sum_{J\in\{(I,l)\,|\,l\in L\}} C_{\mathrm{BIT}}(b,J)$ is just equal to $|\{(I,l)\,|\,l \in L\}| \cdot C_{\mathrm{BIT}}(b,I)$ plus the sum of costs of finding every element in L' separately, similar as in the base case. This observation leads to the equation

$$\sum_{J\in\{(I,l)\,|\,l\in L\}} C_{\mathrm{BIT}}(b,J) = \sum_{l\in L} C_{\mathrm{BIT}}(b,(I,l)) = nC_{\mathrm{BIT}}(b,I) + \frac{1}{2}n(n+1).$$

Therefore, we can rewrite $\sum_{J\in\mathcal{I}_{m+1}} C_{\mathrm{BIT}}(b,J)$ as

$$\begin{aligned}
\sum_{J\in\mathcal{I}_{m+1}} C_{\mathrm{BIT}}(b,J) &= \sum_{I\in\mathcal{I}_m} \sum_{J\in\{(I,l)\,|\,l\in L\}} C_{\mathrm{BIT}}(b,J) \\
&= \sum_{I\in\mathcal{I}_m} \big(n \cdot C_{\mathrm{BIT}}(b,I) + \frac{1}{2}n(n+1)\big) \\
&= n \cdot \big(\sum_{I\in\mathcal{I}_m} C_{\mathrm{BIT}}(b,I)\big) + |\mathcal{I}_m| \cdot \frac{1}{2}n(n+1) \\
&= n \cdot \frac{1}{2}n(n+1) \cdot mn^{m-1} + n^m \cdot \frac{1}{2}n(n+1) \quad \text{[by induction hypothesis]} \\
&= \frac{1}{2}n(n+1) \cdot (m+1)n^m \qquad \square
\end{aligned}$$

For a uniformly distributed random variable $I \in \mathcal{I}_m$, Lemma 4.1 offers the expectation of $C_{\mathrm{BIT}}(b,I)$ and states that this expectation is independent of the initial binary setting $b(L)$. As one can see in the proof of Lemma 4.1, this independence relies on the symmetry of the uniform distribution of $I \in \mathcal{I}_m$.

Remark 4.2. *Let $L = (x_1,\dots,x_n)$ be a list, and let $\mathcal{I}_m$ denote all length m sequences of requests of L. Consider the cost of* BIT *for serving a uniform random request sequence $I \in \mathcal{I}_m$. Since I is arbitrarily chosen from $\mathcal{I}_m$, every request y_j is distributed uniformly over $\{x_1,\dots,x_n\}$. Thus the position of y_j in the list before accessing y_j, say $L^{(j-1)} = (x_1^{j-1}, x_2^{j-1},\dots,x_n^{j-1})$, is also uniformly distributed along the list, i.e. $P(y_j = x_k^{j-1}) = \frac{1}{n}$ holds for all $k = 1,\dots,n$. Recall the cost of accessing the k-th element in the list is k. Hence, the cost of accessing y_j is uniformly distributed on $\{1,\dots,n\}$.*

Note that the key argument "y_j is distributed uniformly on $\{x_1,\dots,x_n\}$" depends neither on the bit setting nor on the order of list elements in the current list $L^{(j-1)}$.

Therefore, one can conclude that the cost of accessing every request in I is uniformly distributed over $\{1,\dots,n\}$.

In other words, the cost of serving y_j can be seen as the number of throwing a fair die with n faces. Similarly, the cost of serving I is the sum of n dice.

Lemma 4.1 is then an immediate consequence from this argument: The expectation of serving one request is equal to $\frac{n+1}{2}$, hence serving a sequence of m requests costs $\frac{m(n+1)}{2}$. Therefore, the sum $\sum_{I\in\mathcal{I}_m} C_{\mathrm{BIT}}(b,I)$ over all possible request sequences is given by

$$\begin{aligned}\sum_{I\in\mathcal{I}_m} C_{\mathrm{BIT}}(b,I) &= |\mathcal{I}_m|\cdot\frac{m(n+1)}{2}\\ &= n^m\cdot\frac{m(n+1)}{2}\\ &= \frac{1}{2}n(n+1)\cdot mn^{m-1}.\end{aligned}$$

Corollary 4.3. *Let $L=(x_1,\dots,x_n)$ be a list and let Y be a random request sequence chosen uniformly from $\mathcal{I}_m$. Then the cost $C=C_{\mathrm{BIT}}(Y)$ of serving Y using* BIT *is a random variable with support $R_C=\{m,m+1,\dots,nm\}$. The probability distribution of C is given by:*

$$P(C=z)=\frac{1}{n^m}\cdot\sum_{k=0}^{d_{max}}(-1)^k\cdot\binom{m}{k}\cdot\binom{z-nk-1}{m-1}$$

where z is an integer in R_C and d_{max} is defined as $\lfloor\frac{z-m}{n}\rfloor$.

The expectation of C is equal to $\frac{m(n+1)}{2}$.

Proof. Since BIT does not use paid transpositions, the costs of BIT are incurred by accessing. Certainly, the overall cost of accessing m requests is bounded by m and nm, where only the first or the last element in the current list is requested respectively.

By Remark 4.2, it suffices to determine the distribution of the sum of rolling dice. This can be done using a combinatorial approach. Assume that we have z identical objects (corresponds to the sum of rolling dice) together with m distinct bins (corresponds to the number of dice), each of which is identified to a number from 1 to m. The aim is to arrange these objects into bins in such a way that every bin contains at least 1 object and at most n objects (the capacity of bins corresponds to the number of faces of a die). It is to determine the number of such arrangements. This number can be calculated using a modified version of the well known stars and bars method.

Claim: There are $\binom{z-1}{m-1}$ possibilities to arrange z identical objects into m distinct bins such that every bin contains at least 1 object.

Proof: Consider the case that empty bins are not allowed. View the z stars as fixed objects in a line defining $m-1$ gaps between stars, in each of which there may or may not be one bar (a bin partition).

* * * * * * *

A configuration is obtained by choosing $m-1$ of these gaps to actually contain a bar.

$$* \quad * \;\Big|\; * \quad * \;\Big|\; * \;\Big|\; * \quad *$$

Therefore, there are $\binom{z-1}{m-1}$ possible configurations. ■

Notice that some of these $\binom{z-1}{m-1}$ arrangements may violate the capacity n of bins.

For any $i \in \{1, \dots, m\}$ let $A_{\{i\}}$ denote the set of those arrangement having bin i with more than n objects. Hence the set $\bigcup_{i\in\{1,\dots,m\}} A_{\{i\}}$ consists of those arrangements violating the cardinality constraint. Thus, the desired number is equal to

$$\binom{z-1}{m-1} - \left|\bigcup_{i\in\{1,\dots,m\}} A_{\{i\}}\right| = \binom{z-1}{m-1} - \sum_{k=1}^{m}(-1)^{k-1} \sum_{S\subseteq\{1,\dots,m\},|S|=k} \left|\bigcap_{i\in S} A_{\{i\}}\right|$$

by the principle of inclusion and exclusion.

For $S \subseteq \{1, \dots, m\}$ let A_S denote the set of those arrangement having every bin in $S \subseteq \{1, \dots, m\}$ with more than n objects. Obviously, we have $\bigcap_{i\in S} A_{\{i\}} = A_S$. The cardinality of A_S can be determined by the stars and bars method.

Claim: There are $\binom{z-|S|n-1}{m-1}$ ways to pack z identical objects into m distinct bins such that every bin contains at least 1 object and every bin in S contains at least $n+1$ objects.

Proof: The proof is straightforward. After packing bins in S with n objects, it remains to pack $z-|S|n$ objects into all m bins, the number of different possibilities is $\binom{z-|S|n-1}{m-1}$ by the stars and bars method. ■

Thus, the desired number of arrangements is equal to

$$\begin{aligned}\binom{z-1}{m-1} - \left|\bigcup_{i\in\{1,\dots,m\}} A_{\{i\}}\right| &= \binom{z-1}{m-1} - \sum_{k=1}^{m}(-1)^{k-1} \sum_{S\subseteq\{1,\dots,m\},|S|=k} |A_S| \\ &= \binom{z-1}{m-1} - \sum_{k=1}^{m}(-1)^{k-1}\binom{m}{k}\binom{z-kn-1}{m-1}\end{aligned}$$

Certainly, it is not possible to have more than $d_{\max} = \lfloor \frac{z-m}{n} \rfloor$ bins violating the capacity constraint. Therefore, we conclude

$$P(C=z) = \frac{1}{n^m} \cdot \sum_{k=0}^{d_{\max}} (-1)^k \cdot \binom{m}{k} \cdot \binom{z-n\cdot k-1}{m-1}. \qquad \square$$

Remark 4.4. *The analysis in this section relies only on these two important properties:*

- *Requests are i.i.d., and*
- *The costs of serving I is equal to the sum of the cost of accessing requests in I.*

Thus the argument in Remark 4.2 actually applies to any deterministic online algorithms ALG *without paid transpositions.*

4.1.2 The Distribution of Bit Values

Recall Conjecture 3.56, BIT does not perform well if the entire initial segment of L^{OPT} up to the requested element is contained in the initial segment of L^{BIT} up to that element. In order to make the conjecture more precise, it is intuitive to take a closer look at the initial segment of BIT. The proof of this subsection is provided at its end.

For list-length $n = 3$ and length of request sequence $m = 10$, we have generated all 3^{10} request sequences with 10 requests and tested BIT with all 2^3 bit settings. The vector

$$v = (0.69, 0.49, 0.32)$$

denotes the probability of the bit value being equal to 1 in the corresponding position after serving an arbitrary request sequence I with BIT.

Given a list L and request sequence I of L with $|L| = n$ and $|I| = m$. Let x_i^j denote the list element in position i in L_j^{BIT}, which is the list maintained by BIT after the j-th access, and let b_i^j be the bit value of x_i^j, where $i \in \{1\dots, n\}$ and $j \in \{0, \dots, m\}$. Notice that $P(b_i^0 = 0) = P(b_i^0 = 1) = \frac{1}{2}$ holds by definition of BIT for all $i \in \{1\dots, n\}$.

Remark 4.5. *By definition of* BIT *and the assumption that all possible request sequences with length* 10 *are tested, we have the following facts:*

- *the equation $P(b_i^j = 1) + P(b_i^j = 0) = 1$ holds for all i and j,*
- *the incoming request y_{j+1} is uniformly distributed over $\{x : x \in L\}$, independent of the value of b_i^j for all i and j,*
- *the bit values b_{i-1}^j and b_i^j are independent for all i and j.*

After careful case by case analysis, one can deduce that:

$$\begin{aligned} P(b_1^{j+1} = 1) = & P(b_1^j = 1) \cdot P(y_{j+1} \neq x_1^j) \\ & + P(b_1^j = 0) \cdot \big(P(y_{j+1} = x_1^j) + \sum_{s>1}^{n} P(y_{j+1} = x_s^j) \cdot P(b_s^j = 0)\big) \end{aligned} \tag{4.1}$$

and

$$\begin{aligned} P(b_i^{j+1} = 1) = & P(b_{i-1}^j = 0 \wedge b_i^j = 0) \cdot 0 \\ & + P(b_{i-1}^j = 1 \wedge b_i^j = 1) \cdot P(y_{j+1} \neq x_i^j) \\ & + P(b_{i-1}^j = 1 \wedge b_i^j = 0) \cdot \big(P(y_{j+1} = x_i^j) + \sum_{s>i} P(y_{j+1} = x_s^j) \cdot P(b_s^j = 0)\big) \\ & + P(b_{i-1}^j = 0 \wedge b_i^j = 1) \cdot \big(\sum_{s<i} P(y_{j+1} = x_s^j) + \sum_{s>i} P(y_{j+1} = x_s^j) \cdot P(b_s^j = 1)\big) \end{aligned} \tag{4.2}$$

hold for $i = 2, \dots, n$ and $j = 0, \dots, m - 1$.

Equation (4.1) *states that in the step $j+1$, the bit value b_1^{j+1} of the list element in the first position, is going to be* 1 *if and only if one of the following three cases occurs in the last step j:*

- *b_1^j was* 1*, and any list element but not the first is requested,*
- *b_1^j was* 0*, and the first list element is requested,*
- *b_1^j was* 0*, and some list element is requested which will be moved to the front.*

Equation (4.2) *states that in the step $j+1$, the bit value b_i^{j+1} of the list element x_i^j, is going to be* 1 *if and only if one of the following cases occurs in the last step j:*

- *b_{i-1}^j and b_i^j were* 1*, and some list element $x \neq x_i^j$ is requested,*
- *b_{i-1}^j was* 1 *and b_i^j was* 0*, and*
    - *either x_i^j is requested,*
    - *or some list element is requested which will be moved to the front, so that x_{i-1}^j becomes x_i^{j+1},*
- *b_{i-1}^j was* 0 *and b_i^j was* 1 *and some list element is requested so that x_i^j remains x_i^{j+1}, i.e.*
    - *either some list element is requested which is preceding x_i^j,*
    - *or some list element behind x_i^j is requested which will not change its position in the list L_{j+1}^{BIT}.*

Equations (4.1) and (4.2) allow us to derive a recursive formula describing the distribution of bit values 0 and 1 in the list during the progress of list accesses. In the sake of simplicity, let p_i^j and q_i^j denote the probability $P(b_i^j = 1)$ and $P(b_i^j = 0)$ respectively, and let $p_0^j = 1$ for all j for technical reasons.

Lemma 4.6. *The recursive formula*[1]

$$n \cdot (p_i^{j+1} - p_{i-1}^j) = -p_{i-1}^j \cdot p_i^j + (p_{i-1}^j - p_i^j) \cdot (1 - i - \sum_{s>i}^{n} p_s^j) \tag{4.3}$$

holds for all $i = 1, \dots, n$ and $j = 0, \dots, m-1$.

Now we can analyze the probability p_i^j for specified i and j. Recall that the bit value and requests are uniformly distributed over $\{0,1\}$ and $\{x : x \in L\}$ in our experiment, thus the expectation of the sum of bit values must remain constant $\frac{n}{2}$, as shown in the next lemma.

[1]This formula is also confirmed by our computer experiment, a closed formula is developed using another approach, see Section 5.1.2.

Lemma 4.7. *The equation*

$$\sum_{i=1}^{n} p_i^j = n/2$$

holds for all $j = 0, \dots, m$.

In other words, we have $\mathbb{E}_{I,K}[\sum_{i=1}^{n} b_i^j] = \mathbb{E}_{I,K}[\sum_{i=1}^{n} b_{x_i}] = \frac{n}{2}$, i.e. the expectation of $\sum_{i=1}^{n} b_i^j$ over all BIT$_k$ and all request sequences I with fixed list length n is exactly $\frac{n}{2}$ at any point in time. Due to the symmetry, we can deduce that $\mathbb{E}_{I,K}[b_{x_i}] = \frac{1}{2}$. Notice that this is not equivalent to $\mathbb{E}_{I,K}[b_i^j] = \frac{1}{2}$, as $\mathbb{E}_{I,K}[b_{x_i}]$ fixes the list element instead of the position. To see this difference, we claim that the first list element is much more likely to be 1 than to be 0.

Corollary 4.8. *The inequality $p_1^j > \frac{1}{2}$ holds for all $j = 1, \dots, m$.*

Certainly, Corollary 4.8 constradicts the statement $\forall I \, \forall j = 1, \dots, m : \mathbb{E}_K[b_1^j] = \frac{1}{2}$. Actually, the computer experiment reveals that probability of b_i^j being equal to 1 is related to the position i. Lemma 4.9 supports this observation.

Lemma 4.9. *For all $j = 0, \dots, m$, the sequence $(p_i^j)_{i=1,\dots,n}$ is monotonically non-increasing.*

Remark 4.10. *Combining the result in Lemma 4.9 and Corollary 4.8, we conclude that the closer the position i is to the front, the larger is the probability of b_i^j being equal to* 1.

In Lemma 4.11, the partial sum of bit values of the list is studied. Lemma 4.7 is a special case where $i = n$.

Lemma 4.11. *For all $j = 0, \dots, m$ and $i = 1, \dots, n$ we have $\sum_{s=1}^{i} p_s^j \geq \frac{i}{2}$. This holds with equality only if $i = n$ or $j = 0$.*

In other word: Beginning from x_1 and up to x_i, the expected number of list elements x such that $b_x = 1$ is always more than $\frac{i}{2}$, unless the requested element x_i is the last one in the current list.

In further experiments with small request sequences, similar behavior like p_1^j is observed for all p_i^j with $i < \frac{2}{5}n$, while the counter part $p_i^j < \frac{1}{2}$ is true for $i > \frac{2}{5}n$. The open question arises if p_i^j is monotonic in j. We end this section by strengthening Conjecture 3.56 with the knowledge of the distribution of bit values 0 and 1.

Remark 4.12. *Recall from Conjecture 3.56 that the following properties indicate a request sequence I with a small competitive ratio $\frac{C_{\text{BIT}}(I)}{C_{\text{OPT}}(I)}$:*

- *If x is requested, its position in L^{OPT} should be before its position in L^{BIT},*
- *the entire initial segment of L^{OPT} up to x should be contained in the initial segment of L^{BIT} up to x, and*
- OPT *should apply as few paid transpositions as possible.*

Based on the knowledge of the distribution of bit values, we know that

- *A paid transposition changing the bit value* $(1,0)$ *to* $(0,1)$ *has a higher probability to generate an inversion. Thus, such transpositions should be favored, if* OPT *has to apply some paid transpositions.*

Proof of Results in Section 4.1.2

Proof of Lemma 4.6. We first prove that the equation

$$n \cdot (p_i^{j+1} - p_i^j) = -p_{i-1}^j \cdot p_i^j + (p_{i-1}^j - p_i^j) \cdot (n - i + 1 - \sum_{s>i}^{n} p_s^j) \tag{4.4}$$

holds by distinguishing the case $i = 1$ and $i > 1$.

For $i = 1$, recall Equation 4.1:

$$\begin{aligned} P(b_1^{j+1} = 1) =& P(b_1^j = 1) \cdot P(y_{j+1} \neq x_1^j) \\ &+ P(b_1^j = 0) \cdot \big(P(y_{j+1} = x_1^j) + \sum_{s>1}^{n} P(y_{j+1} = x_s^j) \cdot P(b_s^j = 0)\big) \\ =& \frac{n-1}{n} \cdot P(b_1^j = 1) + \frac{1}{n} \cdot P(b_1^j = 0) + \frac{1}{n} \cdot P(b_1^j = 0) \cdot \sum_{s>1}^{n} P(b_s^j = 0). \end{aligned}$$

By applying the notations of p_i^j, q_i^j and reordering terms, we have

$$\begin{aligned} n \cdot (p_1^{j+1} - p_1^j) =& - p_1^j + q_1^j + q_1^j \cdot \sum_{s>1}^{n} q_s^j \\ =& - p_1^j + q_1^j \cdot \big(1 + \sum_{s>1}^{n} (1 - p_s^j)\big) \\ =& - p_1^j + (1 - p_1^j) \cdot (n - \sum_{s>1}^{n} p_s^j) \\ =& - p_0^j \cdot p_1^j + (p_0^j - p_1^j) \cdot (n - \sum_{s>1}^{n} p_s^j). \end{aligned}$$

For $i > 1,$ recall Equation 4.2:

$$\begin{aligned}
P(b_i^{j+1} = 1) =& P(b_{i-1}^j = 0 \wedge b_i^j = 0) \cdot 0 \\
&+ P(b_{i-1}^j = 1 \wedge b_i^j = 1) \cdot P(y_{j+1} \neq x_i^j) \\
&+ P(b_{i-1}^j = 1 \wedge b_i^j = 0) \cdot \big(P(y_{j+1} = x_i^j) + \sum_{s>i} P(y_{j+1} = x_s^j) \cdot P(b_s^j = 0)\big) \\
&+ P(b_{i-1}^j = 0 \wedge b_i^j = 1) \cdot \big(\sum_{s<i} P(y_{j+1} = x_s^j) + \sum_{s>i} P(y_{j+1} = x_s^j) \cdot P(b_s^j = 1)\big) \\
=& \frac{n-1}{n} \cdot P(b_{i-1}^j = 1) \cdot P(b_i^j = 1) \\
&+ P(b_{i-1}^j = 1) \cdot P(b_i^j = 0) \cdot \big(\frac{1}{n} + \frac{1}{n} \cdot \sum_{s>i} P(b_s^j = 0)\big) \\
&+ P(b_{i-1}^j = 0) \cdot P(b_i^j = 1) \cdot \big(\frac{i-1}{n} + \frac{1}{n} \cdot \sum_{s>i} P(b_s^j = 1)\big).
\end{aligned}$$

By applying the notations of p_i^j, q_i^j and reordering terms, we have

$$\begin{aligned}
p_i^{j+1} =& \frac{n-1}{n} \cdot p_{i-1}^j \cdot p_i^j + \frac{1}{n} \cdot p_{i-1}^j \cdot q_i^j + \frac{1}{n} \cdot p_{i-1}^j \cdot q_i^j \cdot \sum_{s>i} q_s^j \\
&+ q_{i-1}^j \cdot p_i^j \cdot (\frac{i-1+n-i}{n} - \frac{1}{n} \cdot \sum_{s>i} q_s^j) \\
=& \frac{n-1}{n} \cdot (p_{i-1}^j \cdot p_i^j + q_{i-1}^j \cdot p_i^j) + \frac{1}{n} \cdot p_{i-1}^j \cdot q_i^j + \frac{1}{n} \cdot (p_{i-1}^j \cdot q_i^j - q_{i-1}^j \cdot p_i^j) \cdot \sum_{s>i}^{n} q_s^j \\
=& \frac{n-1}{n} \cdot p_i^j + \frac{1}{n} \cdot p_{i-1}^j \cdot q_i^j + \frac{1}{n} \cdot (p_{i-1}^j - p_i^j) \cdot \sum_{s>i}^{n} q_s^j.
\end{aligned}$$

Reordering terms yields

$$n \cdot (p_i^{j+1} - p_i^j) = -p_{i-1}^j \cdot p_i^j + (p_{i-1}^j - p_i^j) \cdot (n - i + 1 - \sum_{s>i}^{n} p_s^j).$$

Hence Equation 4.4 holds. Equation 4.3 can be obtained by moving all summands with factor n to the left hand side. □

Proof of Lemma 4.7. We prove this lemma by induction over j.

Case $j = 0$. By definition of BIT, we have $p_i^0 = \frac{1}{2}$ for all i, thus the equation holds trivially.

Case $j > 0$. Assume that the claim holds for some j. Then it is sufficient to show that:

$$\sum_{i=1}^{n} p_i^{j+1} - \sum_{i=1}^{n} p_i^j = 0.$$

Let $u(i,j) = (1-i) \cdot (p_{i-1}^j - p_i^j)$ and $v(i,j) = (p_{i-1}^j - p_i^j) \cdot \sum_{s=i+1}^{n} p_s^j$. Then Equation 4.4 can be reformulated as follows:

$$n \cdot (p_i^{j+1} - p_i^j) = n \cdot (p_{i-1}^j - p_i^j) - p_{i-1}^j \cdot p_i^j + u(i,j) + v(i,j).$$

Hence we conclude

$$n(\sum_{i=1}^{n} p_i^{j+1} - \sum_{i=1}^{n} p_i^j) = n \cdot \sum_{i=1}^{n}(p_{i-1}^j - p_i^j) - \sum_{i=1}^{n} p_{i-1}^j \cdot p_i^j + \sum_{i=1}^{n} u(i,j) - \sum_{i=1}^{n-1} v(i,j).$$

By applying the induction hypothesis, it is easy to see that

$$\begin{aligned}\sum_{i=1}^{n} u(i,j) &= 0 \cdot p_0^j + \sum_{i=1}^{n-1} p_i^j - (n-1) \cdot p_n^j \\ &= -\frac{n}{2} + n \cdot p_n^j. \qquad (4.5)\end{aligned}$$

Recall that p_0^j is defined to be equal to 1. Thus the telescoping sum $n \cdot \sum_{i=1}^{n}(p_{i-1}^j - p_i^j) = n \cdot (p_0^j - p_n^j)$ is equal to $n - n \cdot p_n^j$. Together with Equation 4.5, we deduce that it is sufficient to show:

$$\sum_{i=1}^{n} p_{i-1}^j \cdot p_i^j + \sum_{i=1}^{n-1} v(i,j) = \frac{n}{2}. \qquad (4.6)$$

By careful index shifting, the term $\sum_{i=1}^{n-1} v(i,j)$ can be further simplified:

$$\begin{aligned}
\sum_{i=1}^{n-1} v(i,j) &= \sum_{i=1}^{n-1}(p_{i-1}^j \cdot \sum_{s=i+1}^{n} p_s^j) - \sum_{i=1}^{n-1}(p_i^j \cdot \sum_{s=i+1}^{n} p_s^j) \\
&= p_0^j \cdot \sum_{s=2}^{n} p_s^j + \sum_{i=2}^{n-1}(p_{i-1}^j \cdot \sum_{s=i+1}^{n} p_s^j) - \sum_{i=1}^{n-1}(p_i^j \cdot p_{i+1}^j + p_i^j \cdot \sum_{s=i+2}^{n} p_s^j) \\
&= \sum_{s=2}^{n} p_s^j - \sum_{i=1}^{n-1} p_i^j \cdot p_{i+1}^j + \sum_{i=2}^{n-1}(p_{i-1}^j \cdot \sum_{s=i+1}^{n} p_s^j) - \sum_{i=1}^{n-2}(p_i^j \cdot \sum_{s=i+2}^{n} p_s^j) \\
&= \sum_{s=2}^{n} p_s^j - \sum_{i=1}^{n-1} p_i^j \cdot p_{i+1}^j + \sum_{k=1}^{n-2}(p_k^j \cdot \sum_{s=k+2}^{n} p_s^j) - \sum_{i=1}^{n-2}(p_i^j \cdot \sum_{s=i+2}^{n} p_s^j) \\
&= -p_1^j + \frac{n}{2} - \sum_{i=1}^{n-1} p_i^j \cdot p_{i+1}^j \\
&= -p_0^j \cdot p_1^j + \frac{n}{2} - \sum_{i=1}^{n-1} p_i^j \cdot p_{i+1}^j \\
&= \frac{n}{2} - \sum_{i=0}^{n-1} p_i^j \cdot p_{i+1}^j \\
&= \frac{n}{2} - \sum_{i=1}^{n} p_{i-1}^j \cdot p_i^j
\end{aligned}$$

This equation confirms Equation 4.6. Therefore, the induction step is established. □

Proof of Corollary 4.8. The recursive formula 4.3 in case $i = 1$ is

$$n \cdot (p_1^{j+1} - p_0^j) = -p_0^j \cdot p_1^j + (p_0^j - p_1^j) \cdot (1 - 1 - \sum_{s>1}^{n} p_s^j).$$

Recall that p_0^j are defined to be equal to 1 for all j. Together with Lemma 4.7, we deduce

$$\begin{aligned}
n \cdot (p_1^{j+1} - \frac{1}{2}) &= \frac{n}{2} - p_1^j - \sum_{s>1}^{n} p_s^j + p_1^j \cdot \sum_{s>1}^{n} p_s^j \\
&= \frac{n}{2} - \sum_{s=1}^{n} p_s^j + p_1^j \cdot \sum_{s>1}^{n} p_s^j \\
&= p_1^j \cdot \sum_{s>1}^{n} p_s^j > 0.
\end{aligned}$$ □

Proof of Lemma 4.9. We proof this lemma by induction over j.

Case $j = 0$. By definition of BIT, we have $p_i^0 = \frac{1}{2}$ for all i, thus the claim is trivial.

Case $j > 0$. To avoid the trivial case, we may assume that the list length $n > 1$.

Remark that the notation p_i^j, q_i^j are defined as $P(b_i^j = 0), P(b_i^j = 0)$ receptively, whence the equation $p_i^j + q_i^j = 1$ holds for all i, j. Assume that $(p_i^j)_{i=1,\dots,n}$ is monotonic non-increasing for some j. Hence, the sequence $(q_i^j)_{i=1,\dots,n}$ is monotonic non-decreasing.

Recall Equation 4.3:

$$n \cdot (p_i^{j+1} - p_{i-1}^j) = -p_{i-1}^j \cdot p_i^j + (p_{i-1}^j - p_i^j) \cdot (1 - i - \sum_{s>i}^{n} p_s^j),$$

from which the following equation can be derived:

$$\begin{aligned}
n \cdot (p_i^{j+1} - p_{i+1}^{j+1}) &= n \cdot (p_{i-1}^j - p_i^j) + (1-i) \cdot (p_{i-1}^j - p_i^j) + i \cdot (p_i^j - p_{i+1}^j) \\
&\quad - p_{i-1}^j \cdot p_i^j + p_i^j \cdot p_{i+1}^j - (p_{i-1}^j - p_i^j) \cdot \sum_{s=i+1}^{n} p_s^j + (p_i^j - p_{i+1}^j) \cdot \sum_{s=i+2}^{n} p_s^j \\
&= (n+1) \cdot (p_{i-1}^j - p_i^j) - i \cdot (p_{i-1}^j - 2 \cdot p_i^j + p_{i+1}^j) - p_{i-1}^j \cdot p_i^j \\
&\quad - p_{i-1}^j \cdot \sum_{s=i+1}^{n} p_s^j + 2 \cdot p_i^j \cdot \sum_{s=i+1}^{n} p_s^j - p_{i+1}^j \cdot \sum_{s=i+2}^{n} p_s^j \\
&= (n+1) \cdot (p_{i-1}^j - p_i^j) - i \cdot (p_{i-1}^j - 2 \cdot p_i^j + p_{i+1}^j) - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 \\
&\quad - p_{i-1}^j \cdot \sum_{s=i+1}^{n} p_s^j + 2 \cdot p_i^j \cdot \sum_{s=i+1}^{n} p_s^j - p_{i+1}^j \cdot \sum_{s=i+1}^{n} p_s^j \\
&= (n+1) \cdot (p_{i-1}^j - p_i^j) - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 \\
&\quad - (i + \sum_{s=i+1}^{n} p_s^j) \cdot (p_{i-1}^j - 2 \cdot p_i^j + p_{i+1}^j).
\end{aligned}$$

In the case $p_{i-1}^j - 2 \cdot p_i^j + p_{i+1}^j \geq 0$, estimate $i + \sum_{s=i+1}^n p_s^j$ from above by replacing every p_i^s with 1:

$$\begin{aligned}
n \cdot (p_i^{j+1} - p_{i+1}^{j+1}) &\geq (n+1) \cdot (p_{i-1}^j - p_i^j) - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 - n \cdot (p_{i-1}^j - 2 \cdot p_i^j + p_{i+1}^j) \\
&= n \cdot (p_i^j - p_{i+1}^j) + p_{i-1}^j - p_i^j - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 \\
&= (n-2) \cdot (p_i^j - p_{i+1}^j) + p_{i-1}^j + p_i^j - 2 \cdot p_{i+1}^j - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 \\
&\geq p_{i-1}^j + p_i^j - 2 \cdot p_{i+1}^j - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 \\
&= -(1 - p_{i-1}^j) \cdot (1 - p_i^j) + (1 - p_{i+1}^j)^2 \\
&= (q_{1+1}^j)^2 - q_{i-1}^j \cdot q_i^j.
\end{aligned}$$

As $(q_i^j)_{i=1,\dots,n}$ is monotonic non-decreasing in i, we conclude that $p_i^{j+1} - p_{i+1}^{j+1} \geq 0$.

In the case $p_{i-1}^j - 2 \cdot p_i^j + p_{i+1}^j < 0$, estimate $i + \sum_{s=i+1}^n p_s^j$ from below by replacing i

with $\sum_{s=1}^{i} p_s^j$:

$$\begin{aligned}
n \cdot (p_i^{j+1} - p_{i+1}^{j+1}) &\geq (n+1) \cdot (p_{i-1}^j - p_i^j) - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 - (p_{i-1}^j - 2 \cdot p_i^j + p_{i+1}^j) \sum_{s=1}^{n} p_s^j \\
&\geq n \cdot (p_{i-1}^j - p_i^j) - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 - \frac{n}{2} \cdot (p_{i-1}^j - 2 \cdot p_i^j + p_{i+1}^j) \\
&= \frac{n}{2} \cdot (2 \cdot p_{i-1}^j - 2 \cdot p_i^j - p_{i-1}^j + 2 \cdot p_i^j - p_{i+1}^j) - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 \\
&\overset{4.7}{=} (p_{i-1}^j - p_{i+1}^j) \cdot \sum_{s=1}^{n} p_s^j - p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 \\
&= -p_{i-1}^j \cdot p_i^j + (p_{i+1}^j)^2 + (p_{i-1}^j - p_{i+1}^j) \cdot (p_i^j + p_{i+1}^j) \\
&+ (p_{i-1}^j - p_{i+1}^j) \cdot \sum_{s<i, s>i+1}^{n} p_s^j \\
&= (p_{i-1}^j - p_i^j) \cdot p_{i+1}^j + (p_{i-1}^j - p_{i+1}^j) \cdot \sum_{s<i, s>i+1}^{n} p_s^j
\end{aligned}$$

By the induction hypothesis, both factors $p_{i-1}^j - p_i^j$ and $p_{i-1}^j - p_{i+1}^j$ are positive, hence we conclude that $p_i^{j+1} - p_{i+1}^{j+1} \geq 0$.

Therefore, the sequence $(p_i^{j+1})_{i=1,\dots,n}$ is monotonic non-increasing in either cases, the induction step is established. $\square$

Proof of Lemma 4.11. The equation for the case $j = 0$ is trivial, as $p_i^0 = \frac{1}{2}$ holds for all $i = 1, \dots, n$. Let $j > 0$ be given. By Lemma 4.9, there exists an index $k \in \{1, \dots, n\}$ (depending on j) such that $p_s^j \geq \frac{1}{2}$ and $p_t^j < \frac{1}{2}$ hold for all $s \leq k$ and $t > k$ respectively. By Corollary 4.8, it is easy to see that

$$\forall i \leq k : \sum_{s=1}^{i} p_s^j > \frac{i}{2}.$$

As the case $i = n$ is already established by Lemma 4.7, it remains to show that the inequality holds for all i with $k < i < n$.

For this case we claim first that the sequence

$$A = (a_i)_{i=k,\dots,n} := (\sum_{s=1}^{i} p_s^j - \frac{i}{2})_{i=k,\dots,n}$$

of partial sums is monotonic decreasing. Indeed, the subtraction of any sequence member $\sum_{s=1}^{i} p_s^j - \frac{i}{2}$ with its successor is equal to $\frac{1}{2} - p_{i+1}^j$, which is positive since $i + 1 > k$ and hence $p_{i+1}^j < \frac{1}{2}$. A simple inductive argument completes the proof of the claim.

Hence the monotonic decreasing sequence A begins with positive member $a_k = \sum_{s=1}^{k} p_s^j - \frac{i}{2}$ and ends with $a_n = \sum_{s=1}^{n} p_s^j - \frac{n}{2} = 0$. Therefore, we must have $a_i > 0$ for all $k < i < n$, which is equivalent to the desired inequality. $\square$

4.2 General Independent and Identical Distributions

In this section we analyze the behavior of BIT on request sequences generated by a general discrete distribution D_P.

Definition 4.13. *A discrete distribution is a probability distribution whose sample space is the set of n distinct elements $\{x_1, \dots, x_n\}$. A vector $P = (p_1, \dots, p_n)$ is the parameter of one discrete distribution if $\sum_{i=1}^{n} p_i = 1$ and $p_i \in (0,1)$ holds for all $i = 1, \dots, n$. The probability function is given by*

$$f(X = x_i) = p_i$$

i.e. each p_i represents the probability of seeing x_i as realization of D_P.

4.2.1 The Expected Preceding Indicator

The expected cost of BIT involves two independent random decisions: the requests in request sequence and the bit setting. We have to expand our notation in order to describe this expectation precisely.

Definition 4.14. *Let $L = (x_1, \dots, x_n)$ be a list and $P = (p_1, \dots, p_n)$ be the parameter of a discrete distribution D_P.*

We keep the usage of I for a deterministic request sequence. The letter b denotes a deterministic bit setting of L, since the cost of BIT *depends also on bit settings, we use $C_{\text{BIT}}(L, b, I)$ to denote the cost of serving (deterministic) $I \in \mathcal{I}_m$ using* BIT *together with a (deterministic) bit setting $b \in \{0,1\}^n$. In the case $I = \emptyset$, we define $C_{\text{BIT}}(L, b, \emptyset) = 0$.*

For any integer $z \le m$ the request sequence $I_{[z]}$ is the consecutive initial part $(y_1, \dots, y_z)$ of I. Then, the cost of accessing the j-th request is given by

$$\Delta_j C_{\text{BIT}}(L, b, I) = C_{\text{BIT}}(L, b, I_{[j]}) - C_{\text{BIT}}(L, b, I_{[j-1]}).$$

For two list elements x, x' of L, the ***preceding indicator*** *of x, x' is defined by*

$$\delta_{xx'}(L, b, I) = \begin{cases} 1, & \text{if } x \text{ strictly precedes } x' \text{ after serving } I, \\ 0, & \text{otherwise.} \end{cases}$$

Upper letters $Y = (Y_j)_{j=1,\dots,m}$ and $B = (B_i)_{i=1,\dots,n}$ are used for random request sequence and random bit setting respectively. B_i follows uniform distribution over $\{0,1\}^n$ and Y_j follows discrete distribution D_P over $\underline{L}$. The expected cost of (serving a request *sequence length m generated by P using)* BIT *is defined as*

$$\mathbb{E}_{Y_j \sim D_P, B_i \sim Uni}[C_{\mathrm{BIT}}(L,B,Y)] = \sum_{I \in \mathcal{I}_m, b \in \{0,1\}^n} Pr(Y = I \text{ and } B = b) \cdot C_{\mathrm{BIT}}(L,b,I),$$

the expected cost $\mathbb{E}_{Y_j \sim D_P, B_i \sim Uni}[\Delta_j C_{\mathrm{BIT}}(L,B,Y)]$ of the j-th request and the expected preceding indicator $\mathbb{E}_{Y_j \sim D_P, B_i \sim Uni}[\delta_{xx'}(L,B,Y)]$ are defined in the same way.

In order to keep the notations clear, the specification L and BIT of the cost and the preceding indicator as well as the distribution of Y_j, B_i indexed by the expectation operator are omitted if it is clear from the context. If expectation of Y and B occurs in the same line of calculations, we use, respectively, $\mathbb{E}_Y$ and $\mathbb{E}_B$ and omit the corresponding distributions.

Remark 4.15. *By the definition of* BIT*, the random request sequence Y and random bit setting B are chosen independently. Hence we have*

$$Pr(Y = I \text{ and } B = b) = Pr(Y = I) \cdot Pr(B = b)$$

and may represent $\mathbb{E}[C(B,Y)]$ as follows.

$$\begin{aligned}
\mathbb{E}_{B,Y}[C(B,Y)] &= \sum_{I \in \mathcal{I}_m, b \in \{0,1\}^n} Pr(Y = I \text{ and } B = b) \cdot C_{\mathrm{BIT}}(L,b,I) \\
&= \sum_{I \in \mathcal{I}_m} Pr(Y = I) \cdot \left(\sum_{b \in \{0,1\}^n} Pr(B = b) \cdot C_{\mathrm{BIT}}(L,b,I) \right) \\
&= \sum_{I \in \mathcal{I}_m} Pr(Y = I) \cdot \mathbb{E}_B[C_{\mathrm{BIT}}(L,B,I)]
\end{aligned}$$

For the expected cost of the j-th request we obtain

$$\begin{aligned}
\mathbb{E}_{B,Y}[\Delta_j C(B,Y)] &= \sum_{I \in \mathcal{I}_m, b \in \{0,1\}^n} Pr(Y = I \text{ and } B = b) \cdot \Delta_j C(b,I) \\
&= \sum_{I \in \mathcal{I}_m} Pr(Y = I) \cdot \mathbb{E}_B[\Delta_j C(B,I)].
\end{aligned}$$

Observe that $\Delta_j C(b,I)$ actually does not depend on the realization $(y_{j+1}, \ldots, y_m)$. By *applying the i.i.d. assumption of Y, we may further derive*

$$\begin{aligned}
\mathbb{E}_{B,Y}[\Delta_j C(B,Y)] &= \sum_{I'\in\mathcal{I}_j, I''\in\mathcal{I}_{m-j}} Pr(Y=(I',I''))\cdot\mathbb{E}_B[\Delta_j C(B,I')] \\
&= \sum_{I'\in\mathcal{I}_j} Pr(Y_{[j]}=I')\cdot\left(\sum_{I''\in\mathcal{I}_{m-j}} Pr(Y_{[m]\setminus[j]}=I'')\right)\mathbb{E}_B[\Delta_j C(B,I')] \\
&= \sum_{I'\in\mathcal{I}_j} Pr(Y_{[j]}=I')\cdot 1\cdot\mathbb{E}_B[\Delta_j C(B,I')] \\
&= \sum_{I'\in\mathcal{I}_j} Pr(Y_{[j]}=I')\cdot\mathbb{E}_B[\Delta_j C(B,I')].
\end{aligned}$$

For the expected preceding indicator we have similar calculation.

$$\begin{aligned}
\mathbb{E}[\delta_{xx'}(B,Y)] &= \sum_{I\in\mathcal{I}_m, b\in\{0,1\}^n} Pr(Y=I \text{ and } B=b)\cdot\delta_{xx'}(b,I) \\
&= \sum_{I\in\mathcal{I}_m} Pr(Y=I)\cdot\mathbb{E}_B[\delta_{xx'}(B,I)]
\end{aligned}$$

Notice that $\mathbb{E}[\delta_{xx'}(B,Y)]$ presents the probability of x preceding x' after serving a random request sequence Y.

Remark 4.16. *The objective of this section is to calculate the expected cost $\mathbb{E}[C(B,Y)]$. Since the analysis relies on the projective property of* BIT*, the partial cost model is applied here. The main idea of this section can be outlined as follows.*

- *Cost depends on the relative position of pairs of list elements.*

  Since BIT *does not apply paid transpositions, the cost $\Delta_j C(b,I)$ is only incurred by accessing. The accessing cost of y_j is equal to the number of list element $x' \neq y_j$, which preceds x immediately before accessing it. Formally, we have*

  $$\Delta_j C(b,I) = \sum_{x'\in\underline{L}} \delta_{x'y_j}(b, I_{[j-1]})$$

  and hence

  $$C(b,I) = \sum_{j=1}^{m}\Delta_j C(b,I) = \sum_{j=1}^{m}\sum_{x'\in\underline{L}} \delta_{x'y_j}(b, I_{[j-1]}). \tag{4.7}$$

  A similar result holds in expectation, as will be shown in Lemma 4.17.

- *The relative position of x and x' depends on the last three requests in $I_{xx'}$.*

Recall from Definition 3.30 that the relative position of any two list elements x, x' can be determined easily: Let $(L_{xx'}, I_{xx'})$ denote the instance obtained from (L, I) by deleting every $\overline{x} \notin \{x, x'\}$ from both of L and I. Then, the relative position of x, x' after serving I with BIT*, is the same as the relative position of x, x' after serving $I_{xx'}$ with* BIT.

Given the last three requests of $I_{xx'}$, Lemma 4.18 provides the probability of x preceding x' after serving I.

- *The last three requests in $I_{xx'}$ depend on the parameter P of the discrete distribution.*

  Remark 4.20 describes how the last three requests of $I_{xx'}$ are distributed in $\{x, x'\}^3$ depending on P.

Lemma 4.17. *We have the following relation between the expected cost, the expected cost of each accessing and the expected preceding indicator:*

$$\mathbb{E}[C(B,Y)] = \sum_{j=1}^{m} \mathbb{E}[\Delta_j C(B,Y)] = \sum_{j=1}^{m} \sum_{x \in \underline{L}} Pr(Y_j = x) \cdot \sum_{x' \in \underline{L}} \mathbb{E}[\delta_{x'x}(B, Y_{[j-1]})]. \tag{4.8}$$

Proof. These two equations are inherited from Equation (4.7).
For the first equation we claim that

$$\sum_{I \in \mathcal{I}_m} Pr(Y = I) \cdot \Delta_j C(b, I) = \sum_{I' \in \mathcal{I}_j} Pr(Y_{[j]} = I') \cdot \Delta_j C(b, I) \tag{4.9}$$

holds for all $b \in \{0,1\}^n$ and all $j = 1, \dots, m$. Indeed, since $\Delta_j C(b, I)$ does not depend on the realization of $(Y_{j+1}, \dots, Y_m)$, one can apply the i.i.d. assumption of Y to deduce

$$\begin{aligned}
&\sum_{I \in \mathcal{I}_m} Pr(Y = I) \cdot \Delta_j C(b, I) \\
&= \sum_{I' \in \mathcal{I}_j, I'' \in \mathcal{I}_{m-j}} Pr\left(Y = (I', I'')\right) \cdot \Delta_j C(b, I) \\
&= \sum_{I' \in \mathcal{I}_j} Pr(Y_{[j]} = I') \cdot \left(\sum_{I'' \in \mathcal{I}_{m-j}} Pr\left((Y_{j+1}, \dots, Y_m) = I''\right) \right) \cdot \Delta_j C(b, I) \\
&= \sum_{I' \in \mathcal{I}_j} Pr(Y_{[j]} = I') \cdot \Delta_j C(b, I).
\end{aligned}$$

Hence we apply Equation (4.7) and (4.9) to complete the proof of the first equation in

Equation (4.8).

$$\begin{aligned}
\mathbb{E}[C(B,Y)] =& \frac{1}{2^n} \sum_{b\in\{0,1\}^n} \sum_{I\in\mathcal{I}_m} Pr(Y=I)\cdot C(b,I) \\
=& \frac{1}{2^n} \sum_{b\in\{0,1\}^n} \sum_{I\in\mathcal{I}_m} Pr(Y=I)\cdot \sum_{j=1}^{m} \Delta_j C(b,I) && \text{Equation (4.7)} \\
=& \frac{1}{2^n} \sum_{b\in\{0,1\}^n} \sum_{j=1}^{m} \sum_{I\in\mathcal{I}_m} Pr(Y=I)\cdot \Delta_j C(b,I) \\
=& \frac{1}{2^n} \sum_{j=1}^{m} \sum_{b\in\{0,1\}^n} \sum_{I'\in\mathcal{I}_j} Pr(Y_{[j]}=I')\cdot \Delta_j C(b,I) && \text{Equation (4.9)} \\
=& \sum_{j=1}^{m} \mathbb{E}[\Delta_j C(B,Y)]
\end{aligned}$$

For the second equation it suffices to show

$$\mathbb{E}[\Delta_j C(B,Y)] = \sum_{x\in\underline{L}} \sum_{x'\in\underline{L}} Pr(Y_j = x)\cdot \mathbb{E}[\delta_{x'x}(B, Y_{[j-1]})] \tag{4.10}$$

for all $j = 1,\dots,m$. We claim that the following equation of conditional expectations

$$\mathbb{E}[\Delta_j C(B,Y) \,|\, B=b] = \sum_{x\in\underline{L}} \sum_{x'\in\underline{L}} Pr(Y_j = x)\cdot \mathbb{E}[\delta_{x'x}(B, Y_{[j-1]}) \,|\, B=b] \tag{4.11}$$

holds for all $b \in \{0,1\}^n$ and all $j = 1,\dots,m$. To see this, we apply the i.i.d. assumption of Y to derive

$$\begin{aligned}
\mathbb{E}[\Delta_j C(B,Y) \,|\, B=b] &= \sum_{I\in\mathcal{I}_j} Pr(Y_{[j]}=I)\cdot \Delta_j C(b,I) \\
&= \sum_{(I',x)\in\mathcal{I}_j} Pr(Y_{[j]}=(I',x))\cdot \left(\sum_{x'\in\underline{L}} \delta_{x'x}(b,I')\right) \\
&= \sum_{I'\in\mathcal{I}_{j-1}} \sum_{x\in\underline{L}} Pr(Y_{[j-1]}=I')\cdot Pr(Y_j=x)\cdot \sum_{x'\in\underline{L}} \delta_{x'x}(b,I') \\
&= \sum_{x\in\underline{L}} Pr(Y_j=x)\cdot \sum_{I'\in\mathcal{I}_{j-1}} Pr(Y_{[j-1]}=I') \sum_{x'\in\underline{L}} \delta_{x'x}(b,I') \\
&= \sum_{x\in\underline{L}} \sum_{x'\in\underline{L}} Pr(Y_j=x)\cdot \sum_{I'\in\mathcal{I}_{j-1}} Pr(Y_{[j-1]}=I')\cdot \delta_{x'x}(b,I') \\
&= \sum_{x\in\underline{L}} \sum_{x'\in\underline{L}} Pr(Y_j=x)\cdot \mathbb{E}[\delta_{x'x}(B,Y_{[j-1]}) \,|\, B=b].
\end{aligned}$$

Therefore, the second equation in Equation (4.7) is a direct consequence from Equation (4.10) which can be obtained by taking the mean over $b\in\{0,1\}^n$ on both side of Equation (4.10). The proof is complete. $\square$

Thus, an analytical presentation of expected preceding indicator

$$\mathbb{E}[\delta_{xx'}(B,Y)] = \sum_{I\in\mathcal{I}_m} Pr(Y=I)\cdot \mathbb{E}_B[\delta_{xx'}(B,I)]$$

for all m is sufficient for the purpose of calculating the expected cost of BIT.

4.2.2 $\mathbb{E}_B[\delta_{xx'}(B,I)]$-invariant Partition

Determining $\mathbb{E}_B[\delta_{xx'}(B,I)]$ for every I is rather complicated. Similar to the partition technique in the proof of Lemma 4.1, we investigate a partition

$$\mathcal{I}_k = \dot{\bigcup}_{a\in A} \mathcal{P}_a$$

such that for the most $\mathcal{P}_a$ we have

$$\forall I,I'\in\mathcal{P}_a : \mathbb{E}_B[\delta_{xx'}(B,I)] = \mathbb{E}_B[\delta_{xx'}(B,I')].$$

Such a partition reduces the complexity of calculating the expected preceding indicator since

$$\mathbb{E}[\delta_{xx'}(B,Y)] = \sum_{a\in A} Pr(Y\in\mathcal{P}_a)\cdot \mathbb{E}_B[\delta_{xx'}(B,I^{(a)})]$$

where $I^{(a)} \in \mathcal{P}_a$ is an arbitrary representative of the part $\mathcal{P}_a$.

We first extend a well known result, which is commonly used in the proof of Theorem 3.57, to reveal that the expected preceding indicator depends actually only on a small part of requests rather than the entire sequence.

Lemma 4.18. *Given an instance (L, I) and a pair x, x' of distinct list elements, such that $I_{xx'}$ contains at least three requests. Then, after serving I with* BIT*, the relative position of x and x' depends only on the last three requests of $I_{xx'}$.*

More precisely, let $I_{xx'}^{tail}$ be the last three requests of $I_{xx'}$, and let $I_{xx'}^{head}$ denote the (possibly empty) complement of $I_{xx'}^{tail}$ in $I_{xx'}$, i.e. $I_{xx'} = (I_{xx'}^{head}, I_{xx'}^{tail})$. After serving I, the relative position of x and x' depends only on $I_{xx'}^{tail}$ as follows:

$$P(x \text{ precedes } x' \text{ after serving } I) = \begin{cases} 1, \text{ if } I_{xx'}^{tail} = (x, x, x), \\ \frac{1}{2}, \text{ if } I_{xx'}^{tail} = (x, x, x'), \\ \frac{3}{4}, \text{ if } I_{xx'}^{tail} = (x, x', x), \\ 1, \text{ if } I_{xx'}^{tail} = (x', x, x). \end{cases}$$

The probabilities for the other four cases $I_{xx'}^{tail} \in \{(x', x', x'), (x', x', x), (x', x, x'), (x, x', x')\}$ can be obtained easily by changing the role of x and x'.

Remark 4.19. *By the projective property of* BIT*, it suffices to prove Lemma 4.18 for the instance $(L_{xx'}, I_{xx'})$ where $I_{xx'} = (I_{xx'}^{head}, I_{xx'}^{tail})$. Let $L'_{xx'}$ denote the current list after serving $I_{xx'}^{head}$. It is important to distinguish the influence of $I_{xx'}^{head}$ to the order of $L'_{xx'}$ from the its influence to the bit setting of $L'_{xx'}$.*

The list order of $L'_{xx'}$ may be influenced or even determined directly by $I_{xx'}^{head}$. E.g. if the last two requests of $I_{xx'}^{head}$ are (x, x), then the list order of $L'_{xx'}$ is certainly (x, x'), independent of the initial list $L_{xx'}$ and its bit setting. Thus the probability of x preceding x' after serving $I_{xx'}^{head}$ depends on the distribution followed by $I_{xx'}^{head}$.

However, by the definition of BIT*, both b_x and $b_{x'}$ distribute uniformly over $\{0, 1\}$, independent of $I_{xx'}^{head}$.*

Proof of Lemma 4.18. The cases $I_{xx'}^{\text{tail}} \in \{(x, x, x), (x', x, x)\}$ are trivial as after the last two consecutive requests, the requested element x is in the first position for sure.

The equation for the case $I_{xx'}^{\text{tail}} = (x, x', x)$ can be proven by a straightforward case analysis, see Table 4.1.

List order after accessing $I_{xx'}^{\text{tail}} = (x, x', x)$		
$(b_x, b_{x'})$	$L'_{xx'} = (x, x')$	$L'_{xx'} = (x', x)$
$(0, 0)$	(x', x)	(x', x)
$(0, 1)$	(x, x')	(x, x')
$(1, 0)$	(x, x')	(x, x')
$(1, 1)$	(x, x')	(x, x')

Table 4.1: List order after accessing (x, x', x).

By Remark 4.19, the bit setting b distributes uniformly over $\{0,1\}^2$. Whether or not x precedes x' in $L'_{xx'}$, we always have that x precedes x' with a probability of $\frac{3}{4}$ after serving $I^{\text{tail}}_{xx'}$. Thus, it is not necessary to consider the influence of $I^{\text{head}}_{xx'}$ on the list order of $L'_{xx'}$.

The remaining case $I^{\text{tail}}_{xx'} = (x, x, x')$ can be verified similarly. Since x is in the first position for sure before the last request to x', the probability of x' preceding x is equal to the probability of x' being moved to the front after the last access, which is equal to $\frac{1}{2}$. Obviously, the probability of x being in the front is $1 - \frac{1}{2} = \frac{1}{2}$. $\square$

Recall the definition of the expected preceding indicator for two arbitrary distinct list elements x, x':

$$\mathbb{E}[\delta_{xx'}(B,Y)] = \frac{1}{2^n} \sum_{I \in \mathcal{I}_m} \sum_{b \in \{0,1\}^n} Pr(Y = I) \cdot \delta_{xx'}(b, I)$$

where

$$\delta_{xx'}(b, I) = \begin{cases} 1, \text{ if } x \text{ precedes } x' \text{ after serving } I, \\ 0, \text{ otherwise.} \end{cases}$$

Given such x, x', Lemma 4.18 suggests to regroup $I \in \mathcal{I}_m$ according to the last three requests in $I_{xx'}$. More precisely, for $J \in \{x, x'\}^3$ define

$$\mathcal{I}_m(J) = \{I \in \mathcal{I}_m \,|\, I^{\text{tail}}_{xx'} = J\}, \quad R_{xx'}(\mathcal{I}_m) = \{I \in \mathcal{I}_m \,|\, 3 > |I_{xx'}|\}.$$

Then $\mathcal{I}_m$ can be partitioned as follows:

$$\mathcal{I}_m = \left(\dot{\bigcup}_{J \in \{x,x'\}^3} \mathcal{I}_m(J) \right) \dot{\cup}\, R_{xx'}(\mathcal{I}_m)$$

and it is proven in Lemma 4.18 that $\mathbb{E}[\delta_{xx'}(B, I)] = \frac{1}{2^n} \sum_{b \in \{0,1\}^n} \delta_{xx'}(b, I)$ is constant for every I in the same $\mathcal{I}_m(J)$. Thus we may pick any representative $I^{(J)}$ from $\mathcal{I}_m(J)$ and obtain

$$\begin{aligned}\sum_{I \in \mathcal{I}_m(J)} \sum_{b \in \{0,1\}^n} Pr\,(Y = I) \cdot \delta_{xx'}(b, I) &= \sum_{I \in \mathcal{I}_m(J)} Pr\,(Y = I) \cdot \mathbb{E}[\delta_{xx'}(B, I^{(J)})] \\ &= Pr\,(Y \in \mathcal{I}_m(J)) \cdot \mathbb{E}[\delta_{xx'}(B, I^{(J)})].\end{aligned}$$

Therefore, we derive

$$\begin{aligned}\mathbb{E}[\delta_{xx'}(B,Y)] =& \frac{1}{2^n}\sum_{I\in\mathcal{I}_m}\sum_{b\in\{0,1\}^n} Pr(Y=I)\cdot\delta_{xx'}(b,I)\\ =& \sum_{J\in\{x,x'\}^3} Pr\left(Y\in\mathcal{I}_m(J)\right)\cdot\mathbb{E}[\delta_{xx'}(B,I^{(J)})]\\ &+\sum_{I\in R_{xx'}(\mathcal{I}_m)}\sum_{b\in\{0,1\}^n} Pr(Y=I)\cdot\delta_{xx'}(b,I). \qquad (4.12)\end{aligned}$$

On the right hand side, the terms $Pr(Y\in\mathcal{I}_m(J))$ for every $J\in\{x,x'\}^3$ and $Pr(Y=I)\cdot\delta_{xx'}(b,I)$ for $I\in R_{xx'}(\mathcal{I}_m)$ are not revealed yet. The terms $Pr(Y\in\mathcal{I}_m(J))$ can be determined in a combinatorial way.

Remark 4.20. *Let $Y_{xx'}$ denote the random sub-sequence obtained from Y by deleting all requests different from x and x'. The notion $Y_{xx'}^{tail}$ is defined similarly to $I_{xx'}^{tail}$. By the i.i.d. assumption of Y, the number of requests to x,x' follows a binomial distribution, i.e.*

$$Pr(|Y_{xx'}|=k)=\binom{m}{k}(p_x+p_{x'})^k\cdot(1-p_x-p_{x'})^{m-k}.$$

Thus $Pr(Y\in\mathcal{I}_m(J))$ can be rewritten as:

$$Pr(Y\in\mathcal{I}_m(J))=\sum_{k=3}^{m}Pr(|Y_{xx'}|=k)\cdot Pr(Y_{xx'}^{tail}=J\,|\,k=|Y_{xx'}|).$$

To understand the term $Pr(Y_{xx'}^{tail}=J\,|\,k=|Y_{xx'}|)$, consider the situation where we have an unfair coin with x and x' on both sides. $Y_{xx'}$ is the random variable of the result of flipping this coin k times. $Y_{xx'}^{tail}=J$ is the event such that the last three result are exactly J. Again by the i.i.d. assumption, each flipping has a probability $\frac{p_x}{p_x+p_{x'}}$ of being x, thus we have

$$Pr(Y_{xx'}^{tail}=J\,|\,k=|Y_{xx'}|)=\frac{p_x^{\alpha_x(J)}\cdot p_{x'}^{\alpha_{x'}(J)}}{(p_x+p_{x'})^3}$$

where $\alpha_x(J)$ and $\alpha_{x'}(J)$ denote, respectively, the count of x and x' in J. Therefore, we conclude

$$\begin{aligned}Pr\left(Y\in\mathcal{I}_m(J)\right)=&\sum_{k=3}^{m}\binom{m}{k}(p_x+p_{x'})^k\cdot(1-p_x-p_{x'})^{m-k}\cdot\frac{p_x^{\alpha_x(J)}\cdot p_{x'}^{\alpha_{x'}(J)}}{(p_x+p_{x'})^3}\\ =&\sum_{k=3}^{m}\binom{m}{k}(p_x+p_{x'})^{k-3}\cdot(1-p_x-p_{x'})^{m-k}\cdot p_x^{\alpha_x(J)}\cdot p_{x'}^{\alpha_{x'}(J)}.\end{aligned}$$

Thus the first part of the right hand side of Equation (4.12) is done.

Lemma 4.21.

$$\sum_{J\in\{x,x'\}^3} Pr(Y \in \mathcal{I}_m(J)) \cdot \mathbb{E}[\delta_{xx'}(B,Y) \,|\, Y \in \mathcal{I}_m(J)]$$
$$= \sum_{k=3}^{m} \binom{m}{k} (p_x + p_{x'})^{k-3} \cdot (1 - p_x - p_{x'})^{m-k} \cdot \left(p_x^3 + \frac{9}{4} p_x^2 p_{x'} + \frac{3}{4} p_x p_{x'}^2 \right).$$

Proof. We summarize the result of Lemma 4.18 and Remark 4.20 in Table 4.2. The desired equation is an immediate consequence of this table. □

J	$p_x^{\alpha_x(J)} \cdot p_{x'}^{\alpha_{x'}(J)}$	$\mathbb{E}[\delta_{xx'}(B,Y) \,\|\, Y \in \mathcal{I}_m(J)]$
(x,x,x)	p_x^3	1
(x,x,x')	$p_x^2 \cdot p_{x'}$	1/2
(x,x',x)	$p_x^2 \cdot p_{x'}$	3/4
(x',x,x)	$p_x^2 \cdot p_{x'}$	1
(x,x',x')	$p_x \cdot p_{x'}^2$	0
(x',x,x')	$p_x \cdot p_{x'}^2$	1/4
(x',x',x)	$p_x \cdot p_{x'}^2$	1/2
(x',x',x')	$p_{x'}^3$	0

Table 4.2: The expected value of preceding indicator and corresponding probability for $\mathcal{I}_m(J)$.

4.2.3 The Remainder $R_{xx'}(\mathcal{I}_m)$ and the Expected Cost of BIT

It remains to determine the cost of $\sum_{I\in R_{xx'}(\mathcal{I}_m)} \sum_{b\in\{0,1\}^n} Pr(Y = I)\cdot\delta_{xx'}(b,I)$. Recall from the definition that $R_{xx'}(\mathcal{I}_m)$ consists of those request sequences I in $\mathcal{I}_m$ whose corresponding sub-sequence $I_{xx'}$ contains at most two requests. It turns out that in $R_{xx'}(\mathcal{I}_m)$, the value of $\mathbb{E}[\delta_{xx'}(B,I)]$ depends actually on the initial order of x, x' in L. We are forced to subdivide $R_{xx'}(\mathcal{I}_m)$ further according to the length of $I_{xx'}$

$$R_{xx'}^{(k)}(\mathcal{I}_m) = \{I \in \mathcal{I}_m \,|\, k = |I_{xx'}|\}$$

for $k = 0, 1, 2$. The corresponding $\mathbb{E}[\delta_{xx'}(B,I) \,|\, I \in R_{xx'}^{(k)}(\mathcal{I}_m)]$ has to be calculated more carefully.

Lemma 4.22. *Enumerate the list element according to their initial position in L and* *consider the pair x_s and x_t of list elements for some $s \neq t$. Then we have*

$$\sum_{I \in R_{x_s x_t}(\mathcal{I}_m)} \sum_{b \in \{0,1\}^n} Pr(Y = I) \cdot \delta_{x_s x_t}(b, I) = \binom{m}{0}(1 - p_s - p_t)^m + \binom{m}{1}(1 - p_s - p_t)^{m-1} \cdot \left(p_s + \frac{1}{2}p_t\right) + \binom{m}{2}(1 - p_s - p_t)^{m-2} \cdot \left(p_s^2 + \frac{5}{4}p_s p_t\right)$$

if $s < t$ and

$$\sum_{I \in R_{x_s x_t}(\mathcal{I}_m)} \sum_{b \in \{0,1\}^n} Pr(Y = I) \cdot \delta_{x_s x_t}(b, I) = \binom{m}{1}(1 - p_s - p_t)^{m-1} \cdot \frac{1}{2}p_s + \binom{m}{2}(1 - p_s - p_t)^{m-2} \cdot \left(p_s^2 + \frac{3}{4}p_s p_t\right)$$

otherwise.

Proof. Since the case $s > t$ can be obtained from the case $s < t$ simply by changing the role of x_s and x_t, we assume without loss of generality that $s < t$.

Consider the case $s < t$, we first apply the formula in Remark 4.20 and conclude

$$Pr\left(Y \in R^{(k)}_{x_s x_t}(\mathcal{I}_m)\right) = \binom{m}{k}(p_s + p_t)^k \cdot (1 - p_s - p_t)^{m-k}.$$

In the case $k = 0$, both of x_s and x_t are not requested overall, hence x_s precedes x_t after serving I and thus $\mathbb{E}[\delta_{x_s x_t}(B, Y) \,|\, Y \in R^{(0)}_{x_s x_t}] = 1$. Therefore,

$$\sum_{I \in R^{(0)}_{x_s x_t}(\mathcal{I}_m)} \sum_{b \in \{0,1\}^n} Pr(Y = I) \cdot \delta_{x_s x_t}(b, I) = \binom{m}{0} \cdot (1 - p_s - p_t)^m. \tag{4.13}$$

In the case $k = 1$, the list element x_t precedes x_s after serving I if and only if it has received a bit value 0 when BIT assigned bit values at the beginning and has been requested in I. This event happens with a probability of $\frac{1}{2} \cdot \frac{p_t}{p_s + p_t}$ and hence $\mathbb{E}[\delta_{x_s x_t}(B, Y) \,|\, Y \in R^{(1)}_{x_s x_t}] = 1 - \frac{1}{2} \cdot \frac{p_t}{p_s + p_t}$. Therefore,

$$\begin{aligned} &\sum_{I \in R^{(1)}_{x_s x_t}(\mathcal{I}_m)} \sum_{b \in \{0,1\}^n} Pr(Y = I) \cdot \delta_{x_s x_t}(b, I) \\ =&\binom{m}{1} \cdot (p_s + p_t)(1 - p_s - p_t)^{m-1} \cdot \left(1 - \frac{1}{2} \cdot \frac{p_t}{p_s + p_t}\right) \\ =&\binom{m}{1} \cdot (1 - p_s - p_t)^{m-1} \cdot \left(p_s + \frac{1}{2}p_t\right). \end{aligned} \tag{4.14}$$

For the case $k = 2$, a table similar to Table 4.2 is given here to keep the overview.

$I_{x_s x_t}$	$Pr(Y_{x_s x_t} = I_{x_s x_t} \,\vert\, Y \in R^{(2)}_{x_s x_t})$	$\mathbb{E}[\delta_{xx'}(B,Y) \,\vert\, Y_{x_s x_t} = I_{x_s x_t}]$
(x_s, x_s)	$p_s^2/(p_s+p_t)^2$	1
(x_s, x_t)	$p_s \cdot p_t/(p_s+p_t)^2$	1/2
(x_t, x_s)	$p_s \cdot p_t/(p_s+p_t)^2$	3/4
(x_t, x_t)	$p_t^2/(p_s+p_t)^2$	0

Table 4.3: The expected value of preceding indicator and corresponding probability for $R^{(2)}_{x_s x_t}$.

We still have to verify the numbers in Table 4.3. Indeed, results in column $Pr(Y_{x_s x_t} = I_{x_s x_t} \,|\, Y \in R^{(2)}_{x_s x_t})$ are direct consequences of Remark 4.20. In the third column, the cases (x_s, x_s) and (x_t, x_t) are trivial. The other two cases can be argued in the same way as in the proof of Lemma 4.18. Therefore,

$$\begin{aligned}
&\sum_{I \in R^{(2)}_{x_s x_t}(\mathcal{I}_m)} \sum_{b \in \{0,1\}^n} Pr(Y = I) \cdot \delta_{x_s x_t}(b, I) \\
=&\binom{m}{2} \cdot (p_s+p_t)^2 (1-p_s-p_t)^{m-2} \cdot \left(\frac{p_s^2}{(p_s+p_t)^2} \cdot 1 + \frac{p_s p_t}{(p_s+p_t)^2} \cdot \frac{5}{4} + \frac{p_t^2}{(p_s+p_t)^2} \cdot 0 \right) \\
=&\binom{m}{2} \cdot (1-p_s-p_t)^{m-1} \cdot \left(p_s^2 + \frac{5}{4} p_s p_t \right). \qquad (4.15)
\end{aligned}$$

The desired equation for the case $s < t$ can be observed by summing the Equations (4.13), (4.14) and (4.15).

Consider the case $s > t$, i.e. x_t precedes x_s initially. From the calculation above we know that

$$\begin{aligned}
&\sum_{I \in R_{x_s x_t}(\mathcal{I}_m)} \sum_{b \in \{0,1\}^n} Pr(Y = I) \cdot \delta_{x_t x_s}(b, I) \\
=&\binom{m}{0} (1-p_s-p_t)^m \cdot 1 \\
&+ \binom{m}{1} \left((1-p_s-p_t)^{m-1} \cdot (p_s+p_t) \right) \frac{p_t + \frac{1}{2} p_s}{p_s + p_t} \\
&+ \binom{m}{2} \left((1-p_s-p_t)^{m-2} \cdot (p_s+p_t)^2 \right) \frac{p_t^2 + \frac{5}{4} p_s p_t}{(p_s+p_t)^2}
\end{aligned}$$

Since exactly one of $\delta_{x_s x_t}(b, I)$ and $\delta_{x_t x_s}(b, I)$ is equal to 1. for given b and I, we may substitute $\delta_{x_s x_t}(b,I)$ with $1-\delta_{x_t x_s}(b,I)$ to conclude:

$$\begin{aligned}
&\sum_{I\in R_{x_s x_t}(\mathcal{I}_m)}\sum_{b\in\{0,1\}^n} Pr(Y=I)\cdot\delta_{x_s x_t}(b,I)\\
=&\sum_{I\in R_{x_s x_t}(\mathcal{I}_m)}\sum_{b\in\{0,1\}^n} Pr(Y=I)\cdot(1-\delta_{x_t x_s}(b,I))\\
=&\binom{m}{0}(1-p_s-p_t)^m\cdot(1-1)\\
&+\binom{m}{1}\left((1-p_s-p_t)^{m-1}\cdot(p_s+p_t)\right)\left(1-\frac{p_t+\frac{1}{2}p_s}{p_s+p_t}\right)\\
&+\binom{m}{2}\left((1-p_s-p_t)^{m-2}\cdot(p_s+p_t)^2\right)\left(1-\frac{p_t^2+\frac{5}{4}p_sp_t}{(p_s+p_t)^2}\right)\\
=&\binom{m}{1}(1-p_s-p_t)^{m-1}\cdot\frac{1}{2}p_s\\
&+\binom{m}{2}(1-p_s-p_t)^{m-2}\cdot\left(p_s^2+\frac{3}{4}p_sp_t\right)
\end{aligned}$$

□

Corollary 4.23. *Given positive integer $m'<m$, two indices $s\neq t\in[n]$ and a parameter P of a discrete distribution, we define the constant*

$$\begin{aligned}
\alpha(m',s,t)=&\binom{m'}{0}(1-p_s-p_t)^{m'}\\
&+\binom{m'}{1}(1-p_s-p_t)^{m'-1}\cdot\left(p_s+\frac{1}{2}p_t\right)\\
&+\binom{m'}{2}(1-p_s-p_t)^{m'-2}\cdot\left(p_s^2+\frac{5}{4}p_sp_t\right)\\
&+\sum_{k=3}^{m'}\binom{m'}{k}(p_s+p_t)^{k-3}\cdot(1-p_s-p_t)^{m'-k}\cdot\left(p_s^3+\frac{9}{4}p_s^2p_t+\frac{3}{4}p_sp_t^2\right)
\end{aligned}$$

if $s<t$ and

$$\begin{aligned}
\alpha(m',s,t)=&\binom{m'}{1}(1-p_s-p_t)^{m'-1}\cdot\frac{1}{2}p_s\\
&+\binom{m'}{2}(1-p_s-p_t)^{m'-2}\cdot\left(p_s^2+\frac{3}{4}p_sp_t\right)\\
&+\sum_{k=3}^{m'}\binom{m'}{k}(p_s+p_t)^{k-3}\cdot(1-p_s-p_t)^{m'-k}\cdot\left(p_s^3+\frac{9}{4}p_s^2p_t+\frac{3}{4}p_sp_t^2\right)
\end{aligned}$$

if $s > t$. Then the following claim is true.

For a random request sequence Y' with m' requests, the expected preceding indicator $\mathbb{E}[\delta_{x_s x_t}(B, Y)]$ of a pair x_s, x_t of distinct list element is equal to $\alpha(m', s, t)$.

Proof. This claim follows immediately from Equation (4.12), Lemma 4.21 and Lemma 4.22. □

Theorem 4.24. *Let L be a list of length n and let $P = (p_i)_{i=1,\dots,n}$ be a parameter of a discrete distribution over the support $[n]$. Given a sequence Y of m random requests generated by D_P, the expected cost of serving Y using* BIT *is given by*

$$\mathbb{E}[C(B, Y)] = \sum_{j=1}^{m} \sum_{s \in [n]} p_s \cdot \sum_{t \in [n]} \alpha(j-1, t, s).$$

Proof. This theorem is a direct consequence from Equation (4.8) and Corollary 4.23. □

4.3 Locality Reference

The next generalization is to drop the i.i.d. restriction, leading to the *locality of reference* property.

Informally, a request sequence exhibits locality of reference if, at any time, the newly requested item is likely to be requested again in the near future. In other words, a consecutive sub-sequence requesting only one list element, called *run*, is more likely to be present in such request sequences (e.g. the blocks considered in Lemma 3.47 are runs).

In the context of data compression, Burrows-Wheeler Transformation (short: BWT) is applied to transform a plain text into a string of characters. The string after permutation contains more/longer runs, meaning that it is easier to compress and transmit the transformed string than the original plain text. At the end of this section, we provide an example to illustrate the basic idea of BWT and refer to Adjeroh et al. (2008) for details.

The existing results with the aspect of locality of reference utilize two different techniques to analyze the influence of the locality of reference: factoring technique (see Remark 3.36) and bijective analysis.

Instead of comparing online algorithms with OPT, bijective analysis is a method comparing two online algorithms ALG_1 and ALG_2 directly. Roughly speaking, the bijective analysis aims to pair the request sequences for ALG_1 and ALG_2 using a bijection in such a way that the cost of ALG_1 on input I is no more than the cost of ALG_2 on the image of I, for all request sequences $I \in \mathcal{I}_m$ of the same length m. In this case, intuitively, ALG_1 is no worse than ALG_2.

In Angelopoulos et al. (2008), the locality of reference is parameterized by the number of distinct requests within a consecutive sub-sequence of I with a fixed length. In the deterministic case, by applying the bijective analysis, authors point out that MTF outperforms all other deterministic online algorithms with increasing of the locality of reference.

In Albers and Lauer (2008), the locality of reference is parameterized by runs. Depending on how many requests the run itself and the run before contains, runs are further categorized into different types. In the randomized case, Albers and Lauer (2008) provided upper bound of $C_{\mathrm{BIT}}(B, I)$ which is parameterized by the number of different runs. We refer to Albers and Lauer (2008) for details.

To the best of my knowledge, an expected cost $\mathbb{E}_P[C_{\mathrm{BIT}}(B, Y)]$ is unknown in this aspect. Bijective analysis in Angelopoulos et al. (2008) applies only if the probability distribution is preserved under the bijection f. More precisely, only if for every I we have $P(Y = I) = P(Y = f(I))$. The analysis in Albers and Lauer (2008) has assumed a fixed request sequence I. Although, a further study on the relation between the applied probability distribution and the expected number/length of runs may lead to a bound of $\mathbb{E}_P[C_{\mathrm{BIT}}(B, Y)]$, where Y is a random request sequence with the locality of reference property.

Example 4.25 (Adjeroh et al.,2008). *A permutation $\sigma \in S_n$ is a circular shift if*

$$\sigma(k) = \begin{cases} n, & \text{if } k = 1, \\ k - 1, & \text{if } 1 < k \leq n. \end{cases}$$

Given a string s with n characters, all images of s under circular shifts, i.e. all strings in $\{\sigma^k(s) \mid k = 1, \ldots, n\}$, are called cyclic rotations of s.

Consider the word "BANANA\$", where \$ indicates the end of a word. BWT *can be divided into three steps as follow.*

1. *Generate the set of cyclic rotations of "BANANA\$" and save these rotations into a matrix (Figure 4.1a).*

2. *Sort the rows by the left-to-right lexicographic order (Figure 4.1b). Let M^{BWT} denote the resulting matrix.*

3. *The Burrows-Wheeler Transform of "BANANA\$" is defined as the last column of M^{BWT} (Figure 4.1c).*

$$\begin{array}{ccccccc} B & A & N & A & N & A & \$ \\ A & N & A & N & A & \$ & B \\ N & A & N & A & \$ & B & A \\ A & N & A & \$ & B & A & N \\ N & A & \$ & B & A & N & A \\ A & \$ & B & A & N & A & N \\ \$ & B & A & N & A & N & A \end{array}$$

(a) Unsorted cyclic rotations.

$$\begin{array}{ccccccc} \$ & B & A & N & A & N & A \\ A & \$ & B & A & N & A & N \\ A & N & A & \$ & B & A & N \\ A & N & A & N & A & \$ & B \\ B & A & N & A & N & A & \$ \\ N & A & \$ & B & A & N & A \\ N & A & N & A & \$ & B & A \end{array}$$

(b) Sorted cyclic rotations.

$$\begin{array}{c} A \\ N \\ N \\ B \\ \$ \\ A \\ A \end{array}$$

(c) Burrows-Wheeler transform of "BANANA"

Figure 4.1: Burrows-Wheeler transformation.

Given the string "ANNB\$AA" transformed by BWT *(the last column of* M^{BWT}*), one can sort this string in lexicographic order to recover the first column of* M^{BWT} *immediately (Figure 4.2a). Observe that:*

- *In the same row* i *of* M^{BWT}*, the character* $M^{\mathrm{BWT}}_{i,1}$ *follows the character* $M^{\mathrm{BWT}}_{i,n}$ *in the original string "BANANA\$" in a cyclic manner. The same is true for any two consecutive column in* M^{BWT}*, i.e.:*

$$M^{\mathrm{BWT}}_{i,k} = \sigma(M^{\mathrm{BWT}}_{i,k+1}) \quad \forall i \in \{1, \ldots, n\},\, k \in \{1, \ldots, n-1\},$$

  where σ *is the circular shift in* S_n*.*

- *For any character* c*, the* i*-th occurrence of* c *in the last column* $M^{\mathrm{BWT}}_{\cdot,n}$ *corresponds to the* i*-th occurrence of* c *in the first column* $M^{\mathrm{BWT}}_{\cdot,1}$*. E.g., both of the first A in the last column (*$M^{\mathrm{BWT}}_{1,7}$*) and the first A in the first column (*$M^{\mathrm{BWT}}_{2,1}$*) correspond to the last A in the original string.*

Provided that the receiver has known the first and the last column of M^{BWT}*, these two observations can be used to reconstruct original message efficiently, see Figure 4.2.*

$$\begin{array}{ccc}
\$ & & A_1 \\
A_1 & & N_1 \\
A_2 & & N_2 \\
A_3 & & B \\
B & & \$ \\
N_1 & & A_2 \\
N_2 & & A_3
\end{array}$$

(a) Enumerate duplicated characters.

$$\begin{array}{ccc}
\$ & & A_1 \\
A_1 & & N_1 \\
A_2 & & N_2 \\
A_3 & \xrightarrow{\sigma} & B \\
B & & \$ \\
N_1 & & A_2 \\
N_2 & & A_3
\end{array}$$

(b) Reconstruct the circular shift σ

$$\$ \xrightarrow{\sigma} A_1 \xrightarrow{\sigma} N_1 \xrightarrow{\sigma} A_2 \xrightarrow{\sigma} N_2 \xrightarrow{\sigma} A_3 \xrightarrow{\sigma} B \xrightarrow{\sigma} \$$$

(c) The original message can be obtained in the reverse order by repeatedly applying σ on the last charater \$.

Figure 4.2: Reverse Burrows-Wheeler transformation.

Recall the telegraph message

THE CAR ON THE LEFT HIT THE CAR I LEFT \$

considered in Example 3.1. BWT *transforms this message into the following string of words*

LEFT \$ HIT ON THE THE CAR I THE LEFT CAR

The sender and receiver may agree with any list accessing algorithm to maintain a word list. This string of words will be translated further into code according to the applied algorithm and transmitted from the sender to receiver thereafter. E.g., it can be translated into

$$1\mathit{LEFT}\ 2\$\ 3\mathit{HIT}\ 4\mathit{ON}\ 5\mathit{THE}\ 1\ 6\mathit{CAR}\ 7I\ 3\ 7\ 4,$$

if MTF *is applied to maintain the word list. Table 4.4 illustrates how the word list is updated step by step. Note that a new word is appended to the list when it is requested for the first time. Certainly, it will be moved to the front by* MTF *immediately after it is introduced. Thus, new words always appear in the first position of the lists in Table 4.4. Once the code is transmitted, the receiver can reconstruct the string or words using*

Request y	Word list after accessing y	Transmitted word
1*LEFT*	(*LEFT*)	*LEFT*
2*$*	(*$*, *LEFT*)	*$*
3*HIT*	(*HIT*, *$*, *LEFT*)	*HIT*
4*ON*	(*ON*, *HIT*, *$*, *LEFT*)	*ON*
5*THE*	(*THE*, *ON*, *HIT*, *$*, *LEFT*)	*THE*
1	(*THE*, *ON*, *HIT*, *$*, *LEFT*)	*THE*
6*CAR*	(*CAR*, *THE*, *ON*, *HIT*, *$*, *LEFT*)	*CAR*
7*I*	(*I*, *CAR*, *THE*, *ON*, *HIT*, *$*, *LEFT*)	*I*
3	(*THE*, *I*, *CAR*, *ON*, *HIT*, *$*, *LEFT*)	*THE*
7	(*LEFT*, *THE*, *I*, *CAR*, *ON*, *HIT*, *$*)	*LEFT*
4	(*CAR*, *LEFT*, *THE*, *I*, *ON*, *HIT*, *$*)	*CAR*

Table 4.4: The word list maintained by MTF.

Table 4.4. The original message can be obtained by applying the reverse BWT *to the string or words.*

Chapter 5

Computer Experiments and Average Case Analysis of List Accessing Problems

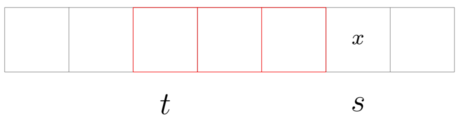
In this chapter, we want to study and compare the performance of the algorithms proposed in Chapter 3. For small instances, an IP is developed to find the optimal solution, allowing an explicit calculation of competitive ratio of given algorithms. For larger instances, we compare the performance of different algorithms on empirical request sequences directly with each other.

The computer language Julia (version 1.0.2) is used for these experiments.

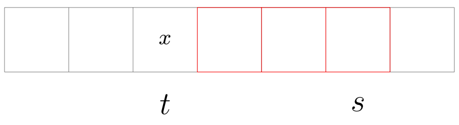
5.1 The Implementation of Algorithms for List Accessing Problem

Each list L consists of two arrays A and B. The first one $A = (x)_{x \in \underline{L}}$ denotes the current arrangement of list elements, the second one $B = (a_x)_{x \in \underline{L}}$ records an additional attribute for each list element. The implementation of B depends on the applied algorithm, e.g. it is used to store the frequency count (Int) in FC, the bit value (Boolean) in BIT etc..

Two basic functions “find” and “move” are implemented for access and transposition. Given a list element $x \in \underline{L}$, find(L, x) returns the position of x in A, which is equal to the cost of accessing x. Given two positions s and t with $s > t$, move(L, s, t) rearranges the list L such that the list element x originally in position s (in A) and its attribute (in B) are moved to position t, while the relative order of other elements in L remains unchanged, see Figure 5.1 for an illustration of the effect of move(L, s, t). This function is called whenever the algorithm decides to apply a chain of free transpositions. Enumerate the list element according to their position in L immediately before calling move, the effect of move(L, s, t) to L is equivalent to the chain $T(x_t, x_s) \circ \cdots \circ T(x_{s-1}, x_s)$.



(a) Before move(L, s, t).



(b) After move(L, s, t).

Figure 5.1: Illustration of move(L, s, t).

5.1.1 Deterministic Algorithms TRANS, MTF, FC, and TS

Since these four algorithms do not apply paid transpositions, the cost function is simply the sum of outputs of find$(\cdot, \cdot)$. To simulate them, it suffices to imitate how they update the list after accessing a request.

TRANS and MTF do not require the aid of B.

```
Input: List $L$ and request $x$ in position $s$ of $L$.
Output: Updated list $L'$
if $s > 1$ then
    $L' = \text{move}(L, s, s-1)$;
end
```

Algorithm 7: Updating rule of TRANS

```
Input: List $L$ and request $x$ in position $s$ of $L$.
Output: Updated list $L'$
$L' = \text{move}(L, s, 1)$;
```

Algorithm 8: Updating rule of MTF

FC maintains B to denote the number of requests to the corresponding element.

```
Input: List $L$ and request $x$ in position $s$ of $L$.
Output: Updated list $L'$
$B[s] = B[s] + 1$;
$p = 0$;
for $t = 1 : s-1$ do
    if $B[t] > B[s]$ then
        $p = t$;
    end
end
if $p = 0$ then
    $L' = \text{move}(L, s, 1)$;
else
    $L' = \text{move}(L, s, p+1)$;
end
```

Algorithm 9: Updating rule of FC

Immediately after accessing the s-th element, FC updates the counter $B[s]$ and finds the largest position t such that $B[t] > B[s]$ (i.e. the element in position t is requested more often than the element in position s). If such t exists, FC applies move$(L, s, t+1)$; Otherwise, x is the most accessed element in the list[1], and thus FC applies move$(L, s, 1)$.

TS is slightly more complicated than the other three approaches. Instead of an array of integers, TS needs an array of pairs of integers to make its decision. More precisely, TS initializes a pair of integers $[a_x^{(1)}, a_x^{(2)}] = [0, 0]$ for each list element x and maintains it so that the indices of the last two requests to x are recorded, e.g. after accessing $I = (x_3, x_2, x_2, x_3, x_2, x_3, x_1, x_1)$, the list element x_2 has the attribute $[3, 5]$, indicating that the last two requests to x_2 are the third and the fifth request in I.

```
**Input:** List $L$ and request $y_j = x$ in position $s$ of $L$.
**Output:** Updated list $L'$
$a_x^{(1)} = a_x^{(2)}$;
$a_x^{(2)} = j$;
**if** $a_x^{(1)} == 0$ **then**
    $L' = L$;
**else**
    **for** $t = 1 : s - 1$ **do**
        **if** $a_{x_t}^{(2)} < a_x^{(1)}$ *or* $a_{x'}^{(1)} < a_x^{(1)} < a_{x'}^{(2)} < a_x^{(2)}$ **then**
            $L' = $ move$(L, s, t+1)$;
            break;
        **else**
            $L' = L$;
        **end**
    **end**
**end**
```

Algorithm 10: Updating rule of TS

Immediately after accessing x, TS updates $[a_x^{(1)}, a_x^{(2)}]$ and tries to find the smallest $t < s$ such that

- x_t has not been requested since the last request to x ($a_{x_t}^{(2)} < a_x^{(1)}$) or
- x_t has been requested only once since the last request to x ($a_{x_t}^{(1)} < a_x^{(1)} < a_{x_t}^{(2)} < a_x^{(2)}$).

where x_t denotes the list element in position t of L. If such t exists, TS moves x to the position immediately after t.

[1]This happens typically when FC processes the first request in I.

5.1.2 Randomized Algorithms RMTF, BIT, COMB

The codes of COMB and RMTF refer simply to the code of MTF, TS, and BIT together with a suitable random number generator. Compared to this, the code of BIT is different.

Using Markov Chain to Simulate BIT

A state of BIT can be presented as a combination of the order of the list together with the corresponding bit setting ($L = [L[1], L[2]]$). Since the order of the list is a permutation of $\underline{L}$, the state space $N(L)$ can be presented by $S_n \times \{0,1\}^n$. An element $s = (\sigma, (b_i)_{i\in[n]}) \in S_n \times \{0,1\}^n$ corresponds to the list $L(s) = \sigma(L)$ together with a bit setting $b(s) = (b_i)_{i\in\{1,\dots,n\}}$. Each request is interpreted as an action that changes the states. The notion $s \to s'$ is used if there exists a sequence $I = (x_j)_{j=1,\dots,m}$ of requests such that: the state $(L(s), b(s))$ will be changed by BIT to $(L(s'), b(s'))$ after serving I. In particular, if the sequence consists only of one element x, this can be specified by $s \xrightarrow{x} s'$. State spaces can be visualized using graphs, e.g. Figure 5.2 illustrates the

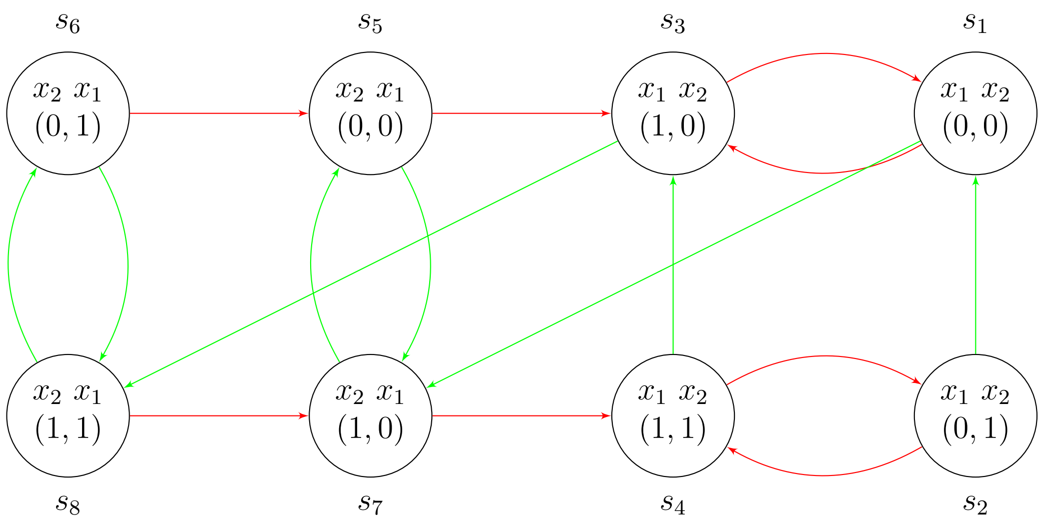


Figure 5.2: State space $N(L = (x_1, x_2))$

graph $G(L)$ of state space $N(L)$ where $L = (x_1, x_2)$. The red and green arcs indicate the next state if x_1 and x_2 are accessed respectively. The code of BIT maintains an array $A_s = [A_s[1], \dots, A_s[n]]$ for each state s where $A_s[i]$ indicates the next state if the list element i is accessed from state s. Then, the matrix $(A_s)_{s\in N(L)}$ is used to calculate the cost and resulting list of BIT to process a given request sequence.

As a side effect, this simulation idea can be used to derive a direct formula of p_i^j in Section 4.1.2. Recall from Section 4.2 that $P = (p_x)_{x\in\underline{L}}$ is a parameter of a discrete distribution. Then, the transition matrix is defined as follows:

$$M(P)_{ij} = \begin{cases} p_x & \exists x : s_i \xrightarrow{x} s_j \\ 0 & \text{otherwise} \end{cases}$$

Notice that the uniform distribution discussed in Section 4.1 is a special case where $p_x = \frac{1}{n}$ for all $x \in \underline{L}$. As every state s corresponds to exactly one column/row in the transition matrix, we omit the index of s and use e_s to denote the unit vector corresponding to that state. The probability of moving from state s to t after m actions (m requests) is given by

$$e_s^\mathsf{T} \cdot M(P)^m \cdot e_t.$$

Since the algorithm BIT chooses uniformly one state s from the set

$$S_{\mathrm{BIT}} := \{\mathrm{id}\} \times \{0,1\}^n,$$

the row vector

$$v_{\mathrm{BIT}} = \left(\frac{1}{|S_{\mathrm{BIT}}|} \cdot \sum_{s \in S_{\mathrm{BIT}}} e_s \right)^\mathsf{T}$$

presents the distribution of the initial states.

Example 5.1. *Let $L = (x_1, x_2)$. Every state has two actions: The red and green edges indicate a request to x_1 and x_2 respectively, and share a Bernoulli distribution with $p = 0.5$. This process is illustrated in Figure 5.2. We first list the state space $S_2 \times \{0,1\}^2$:*

$$\begin{aligned} s_1 &= ((12),(0,0)), & s_5 &= ((21),(0,0)),\\ s_2 &= ((12),(0,1)), & s_6 &= ((21),(0,1)),\\ s_3 &= ((12),(1,0)), & s_7 &= ((21),(1,0)),\\ s_4 &= ((12),(1,1)), & s_8 &= ((21),(1,1)). \end{aligned}$$

Consider the uniform distribution, i.e. $P = (\frac{1}{2}, \frac{1}{2})$, we present the matrix $M(\frac{1}{2}, \frac{1}{2})$ explicitly:

$$\begin{array}{c} \\ s_1\\ s_2\\ s_3\\ s_4\\ s_5\\ s_6\\ s_7\\ s_8 \end{array} \!\!\begin{array}{c} \begin{array}{cccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 \end{array}\\ \begin{pmatrix} 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0\\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0\\ \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2}\\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0\\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2}\\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix} \end{array} = M(\tfrac{1}{2}, \tfrac{1}{2}).$$

E.g. the state s_3 consist of a list $L(s_3) = (x_1, x_2)$ where x_1 and x_2 get the bit values 1 and 0 respectively. If we request x_2, the bit value of x_2 is changed to 1, and will be moved to the front, which is the state $s_8 = ((21),(1,1))$. This event is denoted by $s_3 \overset{x_2}{\to} s_8$, it has the probability $e_3^\mathsf{T} \cdot M(\frac{1}{2}, \frac{1}{2}) \cdot e_8$ to be realized if the current state is s_3.

The set of initial states is given by

$$S_{\mathrm{BIT}} := \{\mathrm{id}\} \times \{0,1\}^2 = \{s_1, s_2, s_3, s_4\},$$

from which BIT *chooses one uniformly random. It is easy to see that*

$$v_{\text{BIT}} = (0.25, 0.25, 0.25, 0.25, 0, 0, 0, 0).$$

Furthermore, the state space can be written as a disjoint union

$$S_2 \times \{0,1\}^2 = \{s_2, s_3, s_6, s_7\} \dot\cup \{s_1, s_4, s_5, s_8\}$$

according to the parity of the sum of the bit values from a state. Certainly, every request either adds or subtracts exactly one from this sum, hence every arc crosses this partition. Thus

- *the graph in Figure 5.2 is bipartite,*
- *there exists a permutation of states such that the matrix $M(\frac{1}{2}, \frac{1}{2})$ has a block structure.*
$$M(P) = \left(\begin{array}{c|c} \mathbf{0} & M' \\ \hline M'' & \mathbf{0} \end{array}\right)$$

These two observations hold for any P.

Recall from Section 4.1.2 that $p_i^j = P(b_i^j = 1) = \mathbb{E}[b_i^j]$ presents the expectation of the bit value in position i in the list after the j-th request provided that bit setting and requests are chosen uniformly. In Section 4.1.2, p_i^j can only be determined via a recursive formula. Lemma 5.2 represents p_i^j in the Markov chain context.

Lemma 5.2. *Let $b(s)_i$ denote the i-th bit value in the bit setting $b(s)$. Then we have*

$$p_i^j = v_{\text{BIT}} \cdot M(P)^j \cdot \sum_{s:b(s)_i=1} e_s.$$

Proof.

$$\begin{aligned}
\mathbb{E}[b_i^j] &= \sum_{t \in S_{\mathrm{BIT}}} \left(P(t \text{ is chosen as initial state}) \cdot \sum_{s \in N(L)} P(t \to s \text{ after } j\text{-th requests}) \cdot b(s)_i \right) \\
&= \sum_{t \in S_{\mathrm{BIT}}} \left(\frac{1}{2^n} \sum_{s : b(s)_i = 1} P(t \to s \text{ after } j\text{-th requests}) \right) \\
&= \sum_{t \in S_{\mathrm{BIT}}} \left(\frac{1}{2^n} \sum_{s : b(s)_i = 1} e_t^{\mathsf{T}} \cdot M(P)^j \cdot e_s \right) \\
&= \sum_{t \in S_{\mathrm{BIT}}} \left(\frac{1}{2^n} e_t^{\mathsf{T}} \cdot M(P)^j \cdot \left(\sum_{s : b(s)_i = 1} e_s \right) \right) \\
&= \left(\sum_{t \in S_{\mathrm{BIT}}} \frac{1}{2^n} e_t^{\mathsf{T}} \right) \cdot M(P)^j \cdot \left(\sum_{s : b(s)_i = 1} e_s \right) \\
&= v_{\mathrm{BIT}} \cdot M(P)^j \cdot \sum_{s : b(s)_i = 1} e_s .
\end{aligned}$$

□

5.1.3 Constructing Optimal Solutions for Small Instances by Integer Programming

One may also generalize the idea of simulating BIT to design an IP for finding OPT for small instances. Recall the enumeration method to find the optimal solution, illustrated in Figure 3.1. In this section, an IP corresponding to the idea of enumeration is developed to permit a calculation of optimal offline algorithms for small instances. Some results of these IPs for a list of three elements are summarized in Table 5.1.

length of I	worst I	OPT cost	BIT cost (worst/mean/best)	performance
3	(3 2 1)	6	9/7.5/6	1.25
4	(3 2 1 1)	7	12/9/7	1.29
5	(3 2 2 1 1)	9	15/11.5/9	1.28
6	(3 2 1 2 1 1)	10	15/12.75/11	1.27

Table 5.1: BIT vs OPT

- The cost of BIT for given I is depending on the initial sequence of bit values of the list elements. There are 8 different initial sequences, each of which incurs a possibly different cost after accessing all elements in I. The column "BIT cost" reports the worst/mean/best of these 8 costs.
- The performance is defined as the mean of the cost of BIT cost divided by the cost of OPT, using the terminology from competitive analysis.

E.g. the $I = (32211)$ is one request sequence with the largest performance 1.25 among all request sequences with length 5. The optimal cost for accessing I is equal to 9, while the BIT needs 11.5 in average. The best and worst initial sequence leads to a cost of 9 and 15 respectively.

In what follows we consider the case $n = 3, m = 6$ to explain how this IP works. The aim is to find one optimal solution for every $I \in \mathcal{I}_6$ of $L = (x_1, x_2, x_3)$. We first enumerate all possible permutations of L:

$$\begin{aligned} L^{(1)} &= (x_1, x_2, x_3), \\ L^{(2)} &= (x_1, x_3, x_2), \\ L^{(3)} &= (x_2, x_1, x_3), \\ L^{(4)} &= (x_2, x_3, x_1), \\ L^{(5)} &= (x_3, x_1, x_2), \\ L^{(6)} &= (x_3, x_2, x_1). \end{aligned}$$

The matrix C denotes the cost of finding a list element in each of these six lists, see Figure 5.3. More precisely, the entry c_{st} denotes the cost of accessing the element x_t in

$$\begin{array}{c} \\ L^{(1)} \\ L^{(2)} \\ L^{(3)} \\ L^{(4)} \\ L^{(5)} \\ L^{(6)} \end{array} \begin{array}{c} \begin{array}{ccc} x_1 & x_2 & x_3 \end{array} \\ \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \\ 3 & 2 & 1 \end{pmatrix} \end{array} =: C = (c_{ij}).$$

Figure 5.3: cost matrix C of access.

list $L^{(s)}$. It is then easy to see that the cost of accessing $y_j = x_t$ in list $L^{(i)}$ is equal to

$$e_i^\top \cdot C \cdot \bar{e}_t$$

where e_i and $\bar{e}_t$ are the i-th and t-th unit vector with dimension 6 and 3 respectively.

$$\begin{array}{c|cccccc} & (123) & (132) & (213) & (231) & (312) & (321) \\ \hline (123) & 0 & 1 & 1 & 2 & 2 & 3 \\ (132) & 1 & 0 & 2 & 3 & 1 & 2 \\ (213) & 1 & 2 & 0 & 1 & 3 & 2 \\ (231) & 2 & 3 & 1 & 0 & 2 & 1 \\ (312) & 2 & 1 & 3 & 2 & 0 & 1 \\ (321) & 3 & 2 & 2 & 1 & 1 & 0 \end{array} = M^{(0)}$$

(a) Reordering cost before the first access.

$$\begin{array}{c|cccccc} & (123) & (132) & (213) & (231) & (312) & (321) \\ \hline (123) & 0 & 1 & 1 & 2 & 2 & 3 \\ (132) & 1 & 0 & 2 & 3 & 1 & 2 \\ (213) & 0 & 1 & 0 & 1 & 2 & 2 \\ (231) & 0 & 1 & 0 & 0 & 1 & 1 \\ (312) & 1 & 0 & 2 & 2 & 0 & 1 \\ (321) & 1 & 0 & 1 & 1 & 0 & 0 \end{array} = M^{(1)}$$

(b) Reordering cost after accessing x_1.

$$\begin{array}{c|cccccc} & (123) & (132) & (213) & (231) & (312) & (321) \\ \hline (123) & 0 & 1 & 0 & 1 & 2 & 2 \\ (132) & 0 & 0 & 0 & 1 & 1 & 1 \\ (213) & 1 & 2 & 0 & 1 & 3 & 2 \\ (231) & 2 & 3 & 1 & 0 & 2 & 1 \\ (312) & 1 & 1 & 1 & 0 & 0 & 0 \\ (321) & 2 & 2 & 1 & 0 & 1 & 0 \end{array} = M^{(2)}$$

(c) Reordering cost after accessing x_2.

$$\begin{array}{c|cccccc} & (123) & (132) & (213) & (231) & (312) & (321) \\ \hline (123) & 0 & 0 & 1 & 1 & 0 & 1 \\ (132) & 1 & 0 & 2 & 2 & 0 & 1 \\ (213) & 1 & 1 & 0 & 0 & 1 & 0 \\ (231) & 2 & 2 & 1 & 0 & 1 & 0 \\ (312) & 2 & 1 & 3 & 2 & 0 & 1 \\ (321) & 3 & 2 & 2 & 1 & 1 & 0 \end{array} = M^{(3)}$$

(d) Reordering cost after accessing x_3.

Figure 5.4: Reordering cost matrices.

The optimal costs of reordering one list to another are recorded in matrices $M^{(i)}$, see Figure 5.4. More precisely, the cost of reordering list $L^{(i)}$ to $L^{(i')}$ (using only paid transpositions) is given by $m_{i,i'}$ in $M^{(0)}$, which can be determined by applying Lemma 3.17 and Algorithm 6. If x_i is just accessed, then the corresponding cost can be found in the matrix $M^{(i)}$ for $i \in \{1, 2, 3\}$.

E.g. suppose request $y = x_2$ is just accessed from the list $(x_3, x_2, x_1) = L^{(6)}$ and the optimal algorithm reorders $L^{(6)}$ to $(x_1, x_2, x_3) = L^{(1)}$, then the cost of paid transpositions arising from this sorting is given by $e_6^\mathsf{T} \cdot M^{(2)} \cdot e_1$.

Let $X = (X_{ij})_{i,j\in[6]} \in \{0,1\}^{6,6}$ where the j-th column vector $X_{(j)}$ of X is a unit vector in $\mathbb{R}^6$ indicating the list chosen by OPT before accessing the j-th request. Then, the list accessing problem with $L = (x_1, x_2, x_3)$ and $I \in \mathcal{I}_6$ can be formulated as:

$$\begin{aligned} \text{minimize} \quad & e_1^\mathsf{T} \cdot M^{(0)} \cdot X_{(1)} + \sum_{s=1}^{5} X_{(s)}^\mathsf{T} \cdot M^{(I_s)} \cdot X_{(s+1)} + \sum_{t=1}^{6} X_{(t)} \cdot C \cdot \overline{e}_{I_t} \\ \text{subject to} \quad & \sum_{i=1}^{6} X_{ij} = 1 \quad \forall j = 1, \ldots, 6 \\ & X_{ij} \in \{0,1\} \quad \forall i, j = 1, \ldots, 6 \end{aligned}$$

OPT may decide to reorder the list before accessing any requests (e.g. S-OPT in Lemma 3.21 applies such reordering). At this point in time, no free transposition is available, hence the cost of reordering the initial list (represented by e_1) to the favor of OPT (represented by $X_{(1)}$) is given by $e_1^\mathsf{T} \cdot M^{(0)} \cdot X_{(1)}$.

Notice that the reordering cost thereafter depends on the latest accessed list element, as some of the transpositions become free. Given a set of lists chosen by OPT (represented by $X_{(1)}, \ldots, X_{(6)}$), the second term $\sum_{s=1}^{5} X_{(s)}^\mathsf{T} \cdot M^{(I_s)} \cdot X_{(s+1)}$ comprises the cost of reordering after every access, where I_s is the index of the list element corresponding to the request y_s, e.g. if $y_1 = x_3$, then $I_1 = 3$. The last term $X_{(t)} \cdot C \cdot \overline{e}_{I_t}$ provides the cost of accessing. The index I_t are defined similar as I_s in the second term.

This is a quadratic problem in $X_{ij} \in \{0,1\}$ for $i, j \in [6]$. To keep the objective function linear, one can use the standard linearization by replacing each product $X_{s,j} \cdot X_{t,j+1}$ with one auxiliary variable z_{stj} and adding the conditions

$$\begin{aligned} z_{stj} &\in \{0,1\} \\ z_{stj} &\le X_{s,j} \\ z_{stj} &\le X_{t,j+1} \\ z_{stj} &\ge X_{s,j} + X_{t,j+1} - 1 \end{aligned}$$

into the model.

Remark 5.3. *This IP is best suited for small instances. For general $n = |L|$ and $m = |I|$, the size of C and M are $n! \cdot n$ and $(n!)^2 \cdot n$. The model itself needs $(n!) \cdot m$ variables and $(n!)^2 \cdot (m-1)$ auxiliary variables with constraints in the same order.*

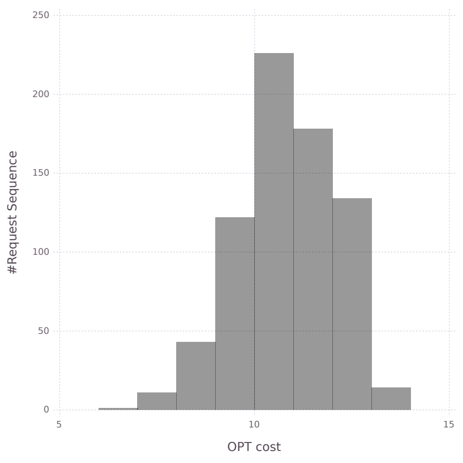
It is possible to apply Proposition 3.9 and Theorem 3.18 to further reduce the size.

In the next section, we use the implementations introduced so far to examine the performance of algorithms proposed in this thesis. In Subsection 5.2, these algorithms are compared with OPT. We have to restrict the size of the instance, since it is harder to calculate the cost of OPT as n and m grow. Subsection 5.3 compares these algorithms directly with each other. By omitting the calculation of OPT, we are able to test these algorithms on larger instances.

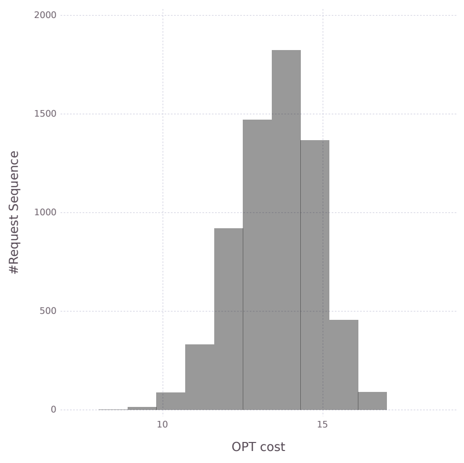
5.2 Expected Cost and Competitive Ratio on Small Instances

5.2.1 Optimal Algorithm

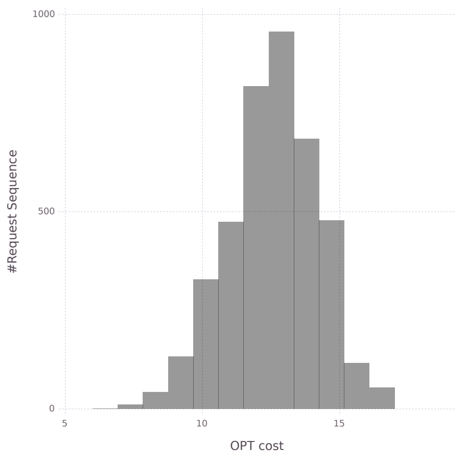
Using the IP developed in Section 5.1.3, the value of $C_{OPT}(I)$ can be computed precisely for small n and m. In Figure 5.5, the optimal cost $C_{OPT}(I)$ is plotted for the case $n = 3, 4$ with $m = 6, 8$. More precisely, we consider a list L with n elements and generate all request sequences $I \in \mathcal{I}_m$. Instances (L, I) are grouped by the optimal cost $C_{OPT}(I)$ (represented by the x axis) and the y axis represents the number of I belonging to each group.



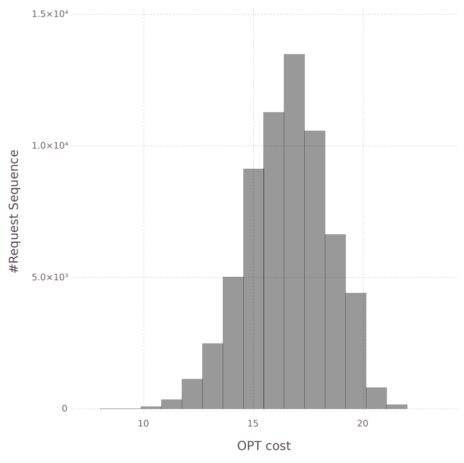
(a) $n = 3, m = 6$



(b) $n = 3, m = 8$



(c) $n = 4, m = 6$

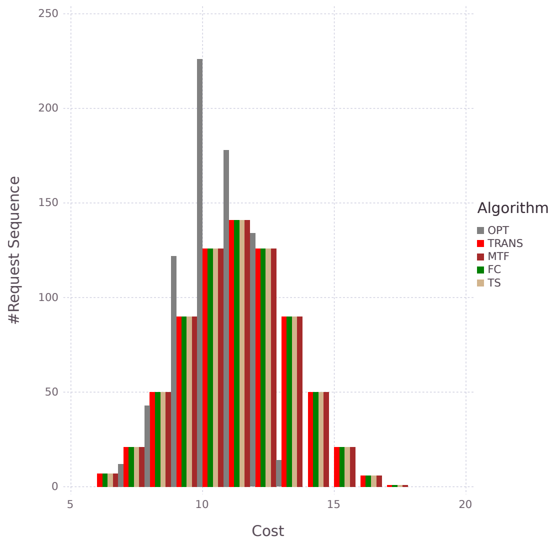


(d) $n = 4, m = 8$

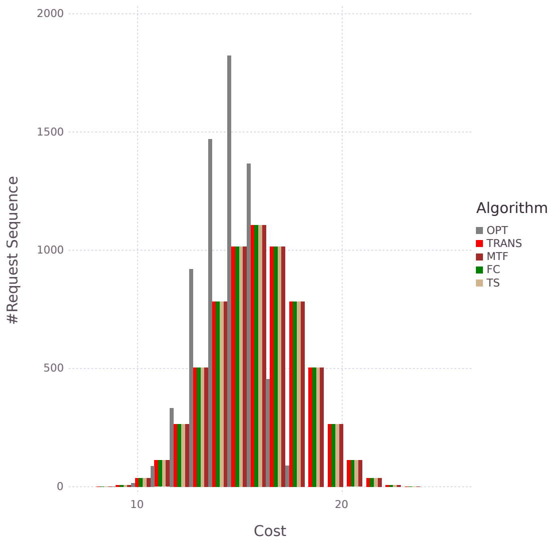
Figure 5.5: OPT cost for small instances.

5.2.2 Deterministic Algorithms

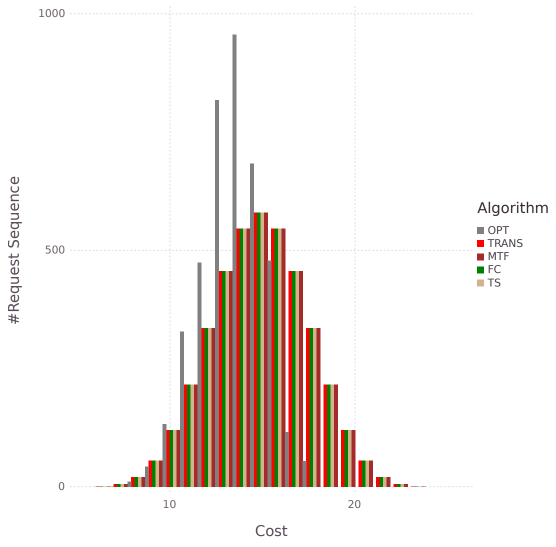
The cost of OPT, TRANS, MTF, FC and TS for serving $I \in \mathcal{I}_m$ are presented in Figure 5.6 for $n = 3, 4$ with $m = 6, 8$. Recall from the Remark 4.4, that the mean and variance of these four algorithms are exactly the same, as one can observe in the figures.



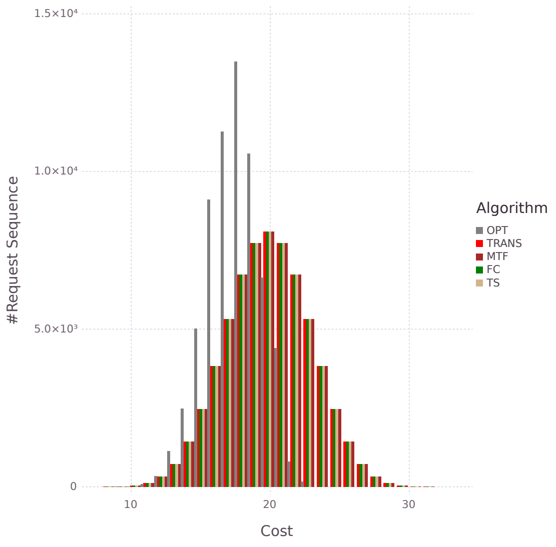
(a) $n = 3, m = 6$



(b) $n = 3, m = 8$



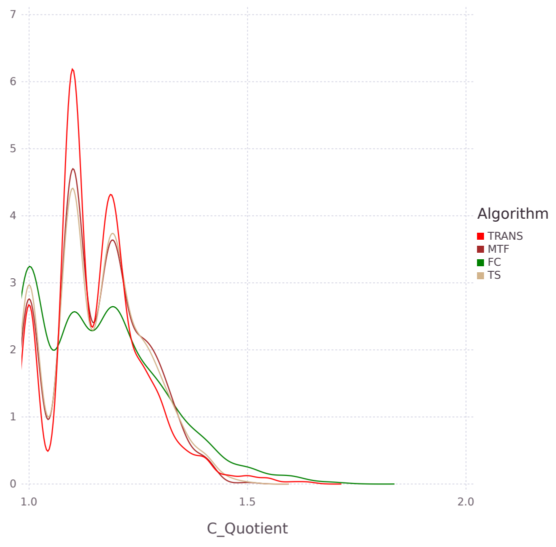
(c) $n = 4, m = 6$



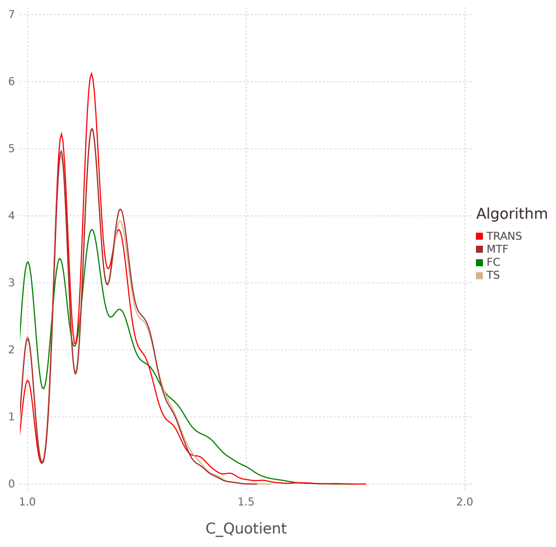
(d) $n = 4, m = 8$

Figure 5.6: Cost of deterministic algorithms for small instances.

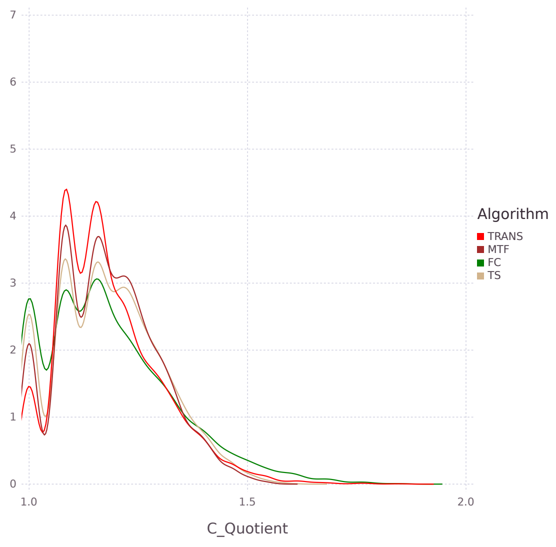
We have further considered the competitive ratio $\frac{C_{\text{ALG}}(I)}{C_{\text{OPT}}(I)}$ which is used to define the competitiveness. Figure 5.7 presents the densities of this quotient over $\mathcal{I}_m$.



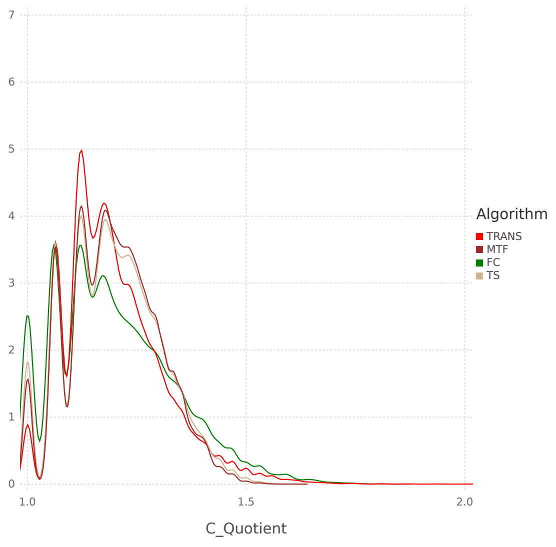
(a) $n = 3, m = 6$



(b) $n = 3, m = 8$



(c) $n = 4, m = 6$



(d) $n = 4, m = 8$

Figure 5.7: Competitive ratio of deterministic algorithms for small instances.

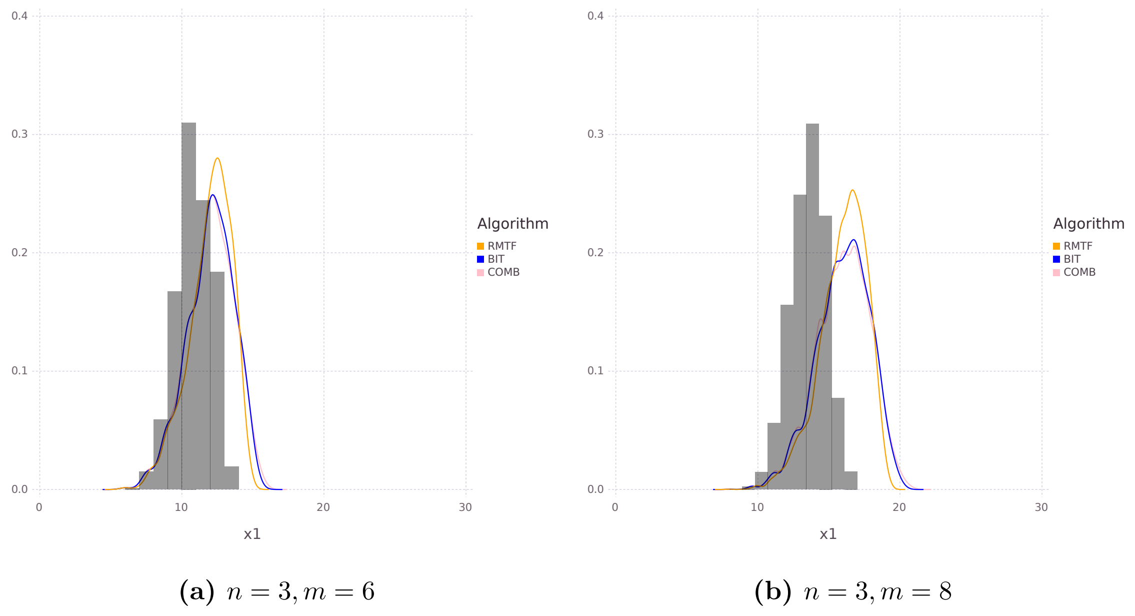
5.2.3 Randomized Algorithms

The density of the expected cost of RMTF, BIT, and COMB over their corresponding random choices[2] in comparison to the cost of OPT are presented in Figure 5.8. Recall from Example 3.41 that the set of random choices of RMTF and MTF can be identified

[2]The request sequence is already determined, the expectation is only over the random choices of the randomized algorithms.

with $\{0,1\}^m$ and $\{0,1\}^n$ respectively. More precisely,

- the expected cost of RMTF is equal to $\frac{1}{2^m}\left(\sum_{L[2]\in\{0,1\}^m} C_{\text{RMTF}}(L[2],I)\right)$,
- the expected cost of BIT is equal to $\mathbb{E}_B[C_{\text{BIT}}(B,I)]$ (see Section 4.2), and
- the expected cost of COMB is equal to $\frac{4}{5}\mathbb{E}_B[C_{\text{BIT}}(B,I)] + \frac{1}{5}C_{\text{TS}}(I)$.



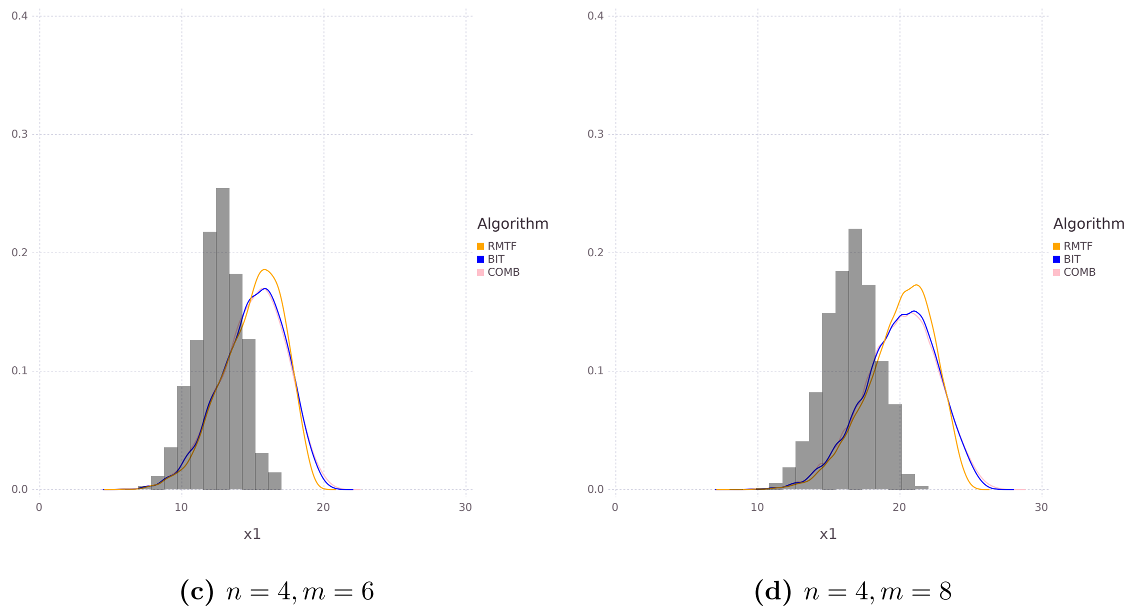
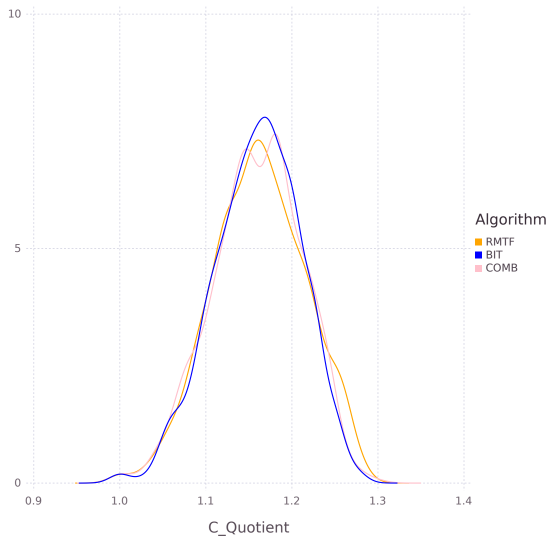
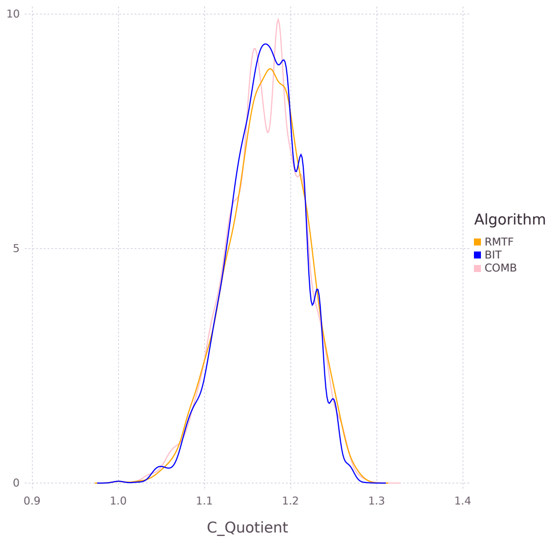


Figure 5.8: Cost of randomized algorithms for small instances.

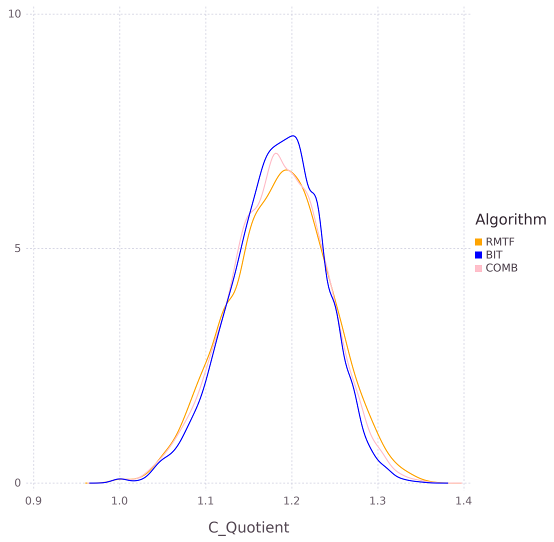
We have further considered the competitive ratio $\frac{C_{\mathrm{ALG}}(I)}{C_{\mathrm{OPT}}(I)}$. Figure 5.9 presents the densities of this quotient over $\mathcal{I}_m$.



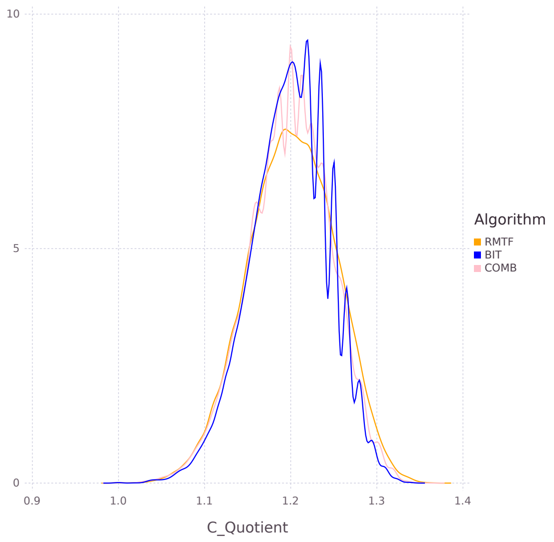
(a) $n = 3, m = 6$



(b) $n = 3, m = 8$



(c) $n = 4, m = 6$



(d) $n = 4, m = 8$

Figure 5.9: Competitive quotient of randomized algorithms for small instances.

5.3 Performance on i.i.d. Generated Request Sequences

Proposed algorithms are compared directly to each other in this subsection. Notice that in the previous subsection, all possible decisions of randomized algorithms are considered (for the purpose of calculating the expected cost), whereas these decisions are generated randomly in this subsection.

To assess the influence of list length and number of requests to the cost of algorithms, the experiments are subdivided into groups according to the parameters of tested instances, see Table 5.2. Each of these groups is equipped with either a uniform distribution U or a discrete distribution D_P where certain list elements (20%) are requested at a high frequency (80%) and the other list elements share 20% chance of being requested.

Group	# List elements	# Requests	# Tests
1	$n = 25$	$m \in \{100, 200, \ldots, 3000\}$	10
2	$n \in \{1, 2, \ldots, 50\}$	$m = 2000$	10
3	$n = 26$	$m \in \{100, 200, \ldots, 3000\}$	10

Table 5.2: Parameters of instances in different groups.

Each experiment is indicated by the ID of the groups it uses together with the distributions which is applied to generate the request sequences, e.g. in Experiment 1-U (considered in Figure 5.10a) consists instances defined as follows:

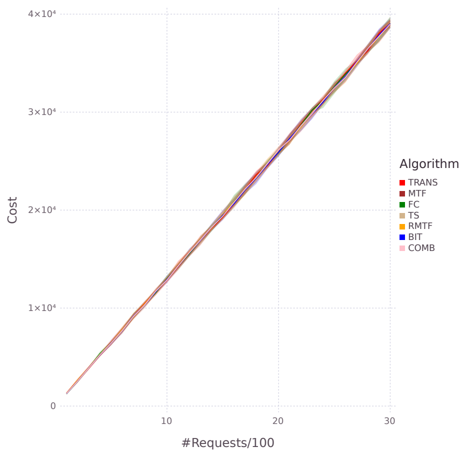
- The length n of the list L is fixed by 25.
- For each $m \in \{100, 200, \ldots, 3000\}$, 10 instances is generated, each of which contains m requests.
- The requests are are uniformly distributed over the support of L.

The first observation is that the cost of solving an instance increases linearly with the length n of the list and the length m of the request sequence, respectively. In the case that uniform distribution is applied, all algorithms perform almost equally well. However, if some list elements are favored by the discrete distribution D_P, TRANS and FC out-performs other algorithms. Their advantages compared to other algorithms increase in step with the size of the instance. On the other hand, MTF and RMTF incur the largest costs for solving large instances.

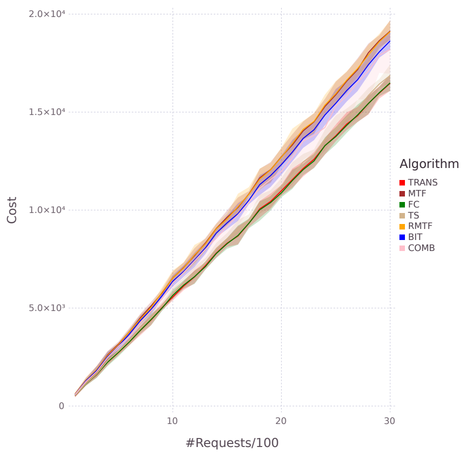
Discrete Distribution of the letter frequency in English

In the context of data compression, the frequencies of letters in English texts is a discrete distribution of particular interest. Peter Norvig provides[3] this distribution based on the Google book data.

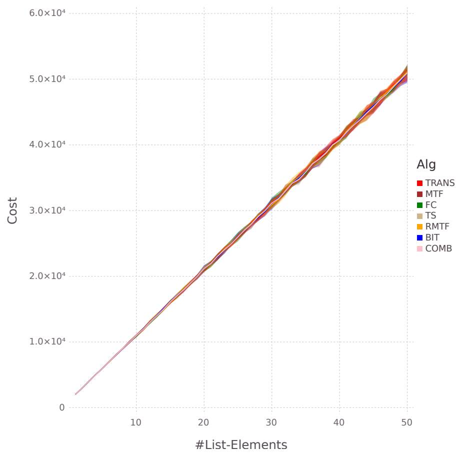
[3]See `https://norvig.com/mayzner.html`



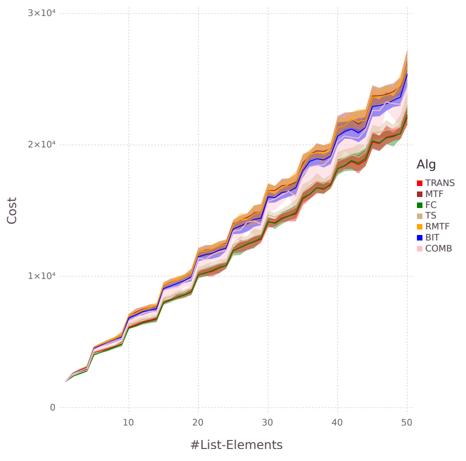
(a) Experiment 1-U



(b) Experiment 1-P



(c) Experiment 2-U



(d) Experiment 2-P

Figure 5.10: Experiment.

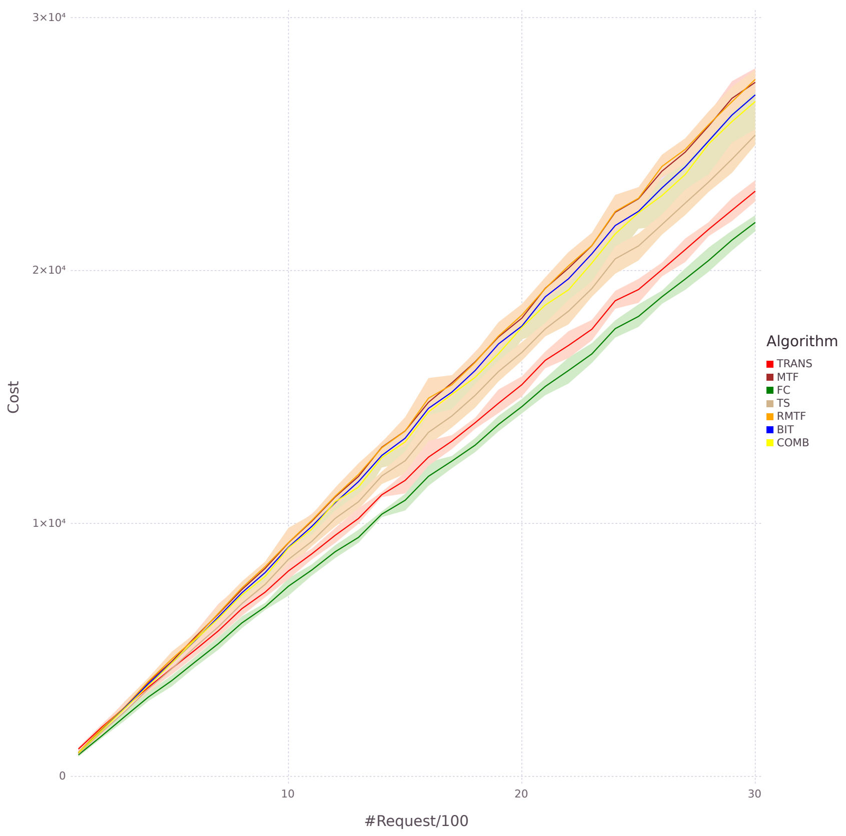


Figure 5.11: Experiment 3-P_1.

We have generated further instances using this distribution, denoted by P_1, provided by Peter Norvig and the statistics of Wikipedia[4]. More precisely, we assume that an article contains 640 words with an average length of 4.79 letters (about 3000 letters).

The Experiment 3-P_1 considers instances (L, I) where the length of L (alphabet) is equal to 26, each request sequence I (random generated "article") contains 3000 requests (letters), which are distributed over $\underline{L}$ following D_{P_1}.

A total of 10 request sequences are generated. Figure 5.11 presents the result of Experiment 3-P_1, which is similar to the result of Experiment 1-P. More precisely, one can use Julia-packages[5] to fit a mixed model, which reveals that the cost

- FC will increase the total cost by 722.5 for every 100 letters in expectation.
- MTF and RMTF will increase the total cost by 917.0 for every 100 letters in expectation.

[4]see `https://de.wikipedia.org/wiki/Wikipedia:Statistik`
[5]E.g. MixedModels.jl from `https://github.com/dmbates/MixedModels.jl`

Bibliography

Adjeroh, D., Bell, T., and Mukherjee, A. (2008). *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching.* Springer Science & Business Media.

Albers, S. (1998). Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27(3): 682–693.

Albers, S. and Lauer, S. (2008). On list update with locality of reference. In *International Colloquium on Automata, Languages, and Programming*, 96–107. Springer.

Albers, S. and Mitzenmacher, M. (1997). Revisiting the counter algorithms for list update. *Information processing letters*, 64(3): 155–160.

Albers, S., Von Stengel, B., and Werchner, R. (1995). A combined bit and timestamp algorithm for the list update problem. *Information Processing Letters*, 56(3): 135–139.

Ambühl, C. (2017). Offline list update is np-hard. *SIGACT News Online Algorithms Column 31*, 48: 68–82.

Ambühl, C., Gärtner, B., and von Stengel, B. (2010). Optimal projective algorithms for the list update problem. *CoRR, abs/1002.2440*, 16.

Angelopoulos, S., Dorrigiv, R., and López-Ortiz, A. (2008). List update with locality of reference. In *Latin American Symposium on Theoretical Informatics*, 399–410. Springer.

Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. (1986). A locally adaptive data compression scheme. *Communications of the ACM*, 29(4): 320–330.

Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., and Weglarz, J. (2014). *Handbook on scheduling.* Springer.

Borodin, A. and El-Yaniv, R. (2005). *Online computation and competitive analysis.* Cambridge University Press.

Conway, R. W., Maxwell, W. L., and Miller, L. W. (2003). *Theory of scheduling*. Courier Corporation.

Divakaran, S. (2014). An optimal offline algorithm for list update. *arXiv preprint arXiv:1404.7638*.

Du, J., Leung, J. Y.-T., and Young, G. H. (1990). Minimizing mean flow time with release time constraint. *Theoretical Computer Science*, 75(3): 347–355.

El-Yaniv, R. (1996). *There are infinitely many competitive-optimal online list accessing algorithms*. Hebrew University of Jerusalem.

Garey, M. R. and Johnson, D. S. (2002). *Computers and intractability*, volume 29. wh freeman New York.

Graham, R. L., Lawler, E. L., Lenstra, J. K., and Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of discrete mathematics*, volume 5, 287–326. Elsevier.

Hester, J. H. and Hirschberg, D. S. (1985). Self-organizing linear search. *ACM Computing Surveys (CSUR)*, 17(3): 295–311.

Heuser, H. (2013). *Lehrbuch der Analysis*. Springer-Verlag.

Irani, S. (1991). Two results on the list update problem. *Information Processing Letters*, 38(6): 301–306.

Kamali, S. and López-Ortiz, A. (2013). A survey of algorithms and models for list update. In *Space-Efficient Data Structures, Streams, and Algorithms*, 251–266. Springer.

Kan, A. R. (2012). *Machine scheduling problems: classification, complexity and computations*. Springer Science & Business Media.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, 85–103. Springer.

Labetoulle, J., Lawler, E. L., Lenstra, J. K., and Kan, A. R. (1984). Preemptive scheduling of uniform machines subject to release dates. In *Progress in combinatorial optimization*, 245–261. Elsevier.

Lenstra, J. K., Kan, A. R., and Brucker, P. (1977). Complexity of machine scheduling problems. In *Annals of discrete mathematics*, volume 1, 343–362. Elsevier.

Marchetti-Spaccamela, A. and Vercellis, C. (1995). Stochastic on-line knapsack problems. *Mathematical Programming*, 68(1-3): 73–104.

McNaughton, R. (1959). Scheduling with deadlines and loss functions. *Management Science*, 6(1): 1–12.

Moore, J. M. (1968). An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management science*, 15(1): 102–109.

Pinedo, M. (2012). *Scheduling*. Springer.

Reingold, N. and Westbrook, J. (1996). Off-line algorithms for the list update problem. *Information Processing Letters*, 60(2): 75–80.

Reingold, N., Westbrook, J., and Sleator, D. D. (1994). Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1): 15–32.

Rivest, R. (1976). On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2): 63–67.

Schrage, L. (1968). Letter to the editor–a proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3): 687–690.

Schulz, F. (1998). Two new families of list update algorithms. In *International Symposium on Algorithms and Computation*, 100–109. Springer.

Sleator, D. D. and Tarjan, R. E. (1985). Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2): 202–208.

Smith, W. E. (1956). Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2): 59–66.