Alternative Settings and Models of the Certain News Network Problem

by

Sameer Naib

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Computer Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
February 2020

© Copyright 2020 by Sameer Naib
Alternative Settings and Models of the Certain News Network Problem

Abstract

The classic $k$-server problem has been studied extensively in the context of competitive analysis for online problems. In this problem, a set of $k$ servers move on a metric space to serve requests that appear in an online fashion on the nodes of the space. The goal is to move servers in a way to minimize the total distance moved by all of them.

The Certain News Network (CNN) problem is a variant of the $k$-server problem where there is only one server ‘serving’ requests on a grid metric space. To serve a request, the server should be vertically or horizontally aligned (not necessarily co-located) with the request. The orthogonal CNN problem is a restricted setting of the CNN problem in which every new request is aligned with the previous request. The best existing lower and upper bounds for the competitive ratios in the orthogonal setting are respectively 3 and 6.464 [1]. In this thesis, we tighten the gap between upper and lower bounds for the orthogonal CNN problem by providing better algorithms that achieves a competitive ratio of at most 5. Our algorithm, named Walk-and-Jump (WJ) algorithm, is based on the concept of work function that is studied before for both $k$-server [2] and CNN problems [3]. We show that work function algorithm, the way studied before, as well as many other “natural” algorithms, do not provide an
improved competitive ratio for the orthogonal CNN problem. We do, however, use the concept of work function in designing the Walk-and-Jump algorithm. Moreover, in the worst-case sequences that we studied, the WJ algorithm has a cost that is no more than three times the cost of an optimal offline algorithm. Given this observation, we conjecture that the analysis can be improved to show the WJ algorithm is indeed optimal and has a competitive ratio of 3.

Besides this main result, we also consider the advice model for the orthogonal CNN problem. Under this model, the online algorithm receives certain “advice” about the input sequence that is generated by a benevolent offline oracle. We show that advice of size $\theta(n)$ is sufficient and necessary to achieve an optimal solution for the orthogonal CNN problem.
Contents

Abstract .................................................. ii
Table of Contents ......................................... v
List of Figures ............................................. vi
List of Tables .............................................. vii
Acknowledgments ........................................... viii
Dedication .................................................. ix

1 Introduction ............................................. 1
   1.1 Introduction to Online Algorithms .................. 1
   1.2 The k-server Problem ................................ 3
   1.3 The CNN Problem ................................... 5
   1.4 Literature Review .................................. 7

2 Preliminary Results .................................... 10
   2.1 Lower Bound ....................................... 11
   2.2 Clock Algorithm & Potential Function Method .... 12
   2.3 Basic Algorithms .................................. 16

3 The Walk and Jump Algorithm ....................... 20
   3.1 Potential Function ................................ 23
   3.2 Basic Observations ................................ 28
      3.2.1 Properties of Optim ................................ 28
      3.2.2 Properties of the Potential Function ......... 30
   3.3 The Main Proof via Case Analysis .................. 31
      3.3.1 Neither OPT nor WJ move their servers ....... 32
      3.3.2 OPT’s server does not move while WJ’s server move .... 34
      3.3.3 WJ’s server does not move while OPT’s server move .... 39
      3.3.4 Both OPT and WJ move to serve the request .. 42
      3.3.5 Summarizing the Results ...................... 45

4 The CNN Problem with Advice ..................... 47
# List of Figures

1.1 An example of the $k$-server problem ........................................... 4
1.2 An example of the CNN problem .................................................. 6

2.1 An example of the Orthogonal CNN problem ............................... 11
2.2 Lower bound for the Orthogonal CNN problem ............................ 12
2.3 The worst-case event in the analysis of CLOCK ............................. 14
2.4 The worst-case input for CLOCK on a $d$ by $d$ grid ..................... 15
2.5 An example of unbounded Greedy Algorithm ............................... 17
2.6 The worst-case sequence for the Work-Function (WF) algorithm .... 19

3.1 An illustration of dangerous points .............................................. 22
3.2 An illustration of predominance ................................................. 24
3.3 Different cases for the main component of the potential ............... 26
3.4 Properties of Opt ................................................................. 29
3.5 The maximum increase in Predominance is 2. ............................. 31
3.6 Amortized analysis when WJ and Opt do not move ....................... 34
3.7 Amortized analysis when WJ moves but Opt does not .................. 37
3.8 Amortized analysis when Opt moves and WJ does not .................. 43
3.9 Amortized analysis when both Opt and WJ move ......................... 46

4.1 An illustration of reduction of binary guessing to the CNN problem .. 51

A.1 A summary of Scenario 1 .......................................................... 57
A.2 A summary of Scenario 2 .......................................................... 59
A.3 A summary of Scenario 3 .......................................................... 62
A.4 A summary of Scenario 4 .......................................................... 65
List of Tables

A.1 A summary of Scenario 1. ........................................... 58
A.2 A summary of Scenario 2. ........................................... 61
A.3 A summary of Scenario 3. ........................................... 64
A.4 A summary of Scenario 4. ........................................... 68
Acknowledgments

I would like to begin by thanking my advisor Dr. Shahin Kamali for his consistent support and guidance during my research and study in the University of Manitoba. Your shared knowledge and inspirational words provides me with a priceless model for my career. Besides my advisor, I would like to thank the members of my thesis committee: Dr. Stephane Durocher and Dr. Avery Miller, for their insightful comments on the thesis proposal and encouragement during the course of this thesis. My sincere thanks to all professors and support staff for helping me grow as a student.

Finally, I express my profound gratitude to my parents, brother and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of research and writing this thesis. This accomplishment would not have been possible without them. A heartiest thanks to my grandmothers for always showering their blessings on me. Thank you.
This thesis is dedicated to my loving father, whose hard work, dedication
and encouragement has always inspired me to push a little harder and
believe in myself.
Chapter 1

Introduction

1.1 Introduction to Online Algorithms

Online algorithms, with their vast scope of real world applications, have been always interesting to researchers in both theoretical and applied facets of computer science. Traditionally, when an algorithm is designed to solve a problem, the input to the problem is fully known in advance and hence the algorithm is “offline”. In many practical applications, however, the algorithms are not provided with the full input at once; instead, the input is revealed in a piece by piece and “online” manner, and the algorithm needs to take irrevocable decisions without any prior knowing the future. As an example, algorithms designed for the paging problem, which asks for page replacement in a fast memory of an operating system, are inherently online as the request for pages appear in an online manner and, at the time of a request, the future requests are not known. As another example, self adjusting data structures like Splay trees also reflect the online property as requests to nodes of the tree are
made in an online manner and, at the time of adjusting the tree, the future requests are not known. Various other applications areas like data compression [4], contracts bidding [5] and data centers for resource allocations on cloud [6] fall under the class of online algorithms.

Online algorithms have been part of computational science for approximately fifty years. In the early days, the cost of online algorithms\(^1\) were computed on stochastic input and were presented as a function of the input size, and this function was used as a criteria to compare online algorithms [7, 8]. In terms of worst-case analysis, the first steps were taken by Graham for the problem of makespan scheduling [9, 10] and Johnson for bin packing [11], where the cost of an online algorithms was compared with that of an optimal offline algorithm Opt over the same input. The maximum ratio between the cost of the online algorithm and Opt over the same input was considered as a measure of performance and was called the “worst-case performance ratio”. This ratio was later called “competitive ratio” in the seminal paper of Sleator and Tarjan in 1984 [12] which concerned paging and list update problems. Since then, competitive analysis has been established as the main measure for comparing online algorithms.

When comparing the performance of an online algorithm with that of Opt, it is always assumed that Opt has unbounded computational power and is aware of the future requests. Since the comparison is made in the worst case scenarios, we assume the input is generated with an adversary who knows (the code of) the algorithm and tries to fail the online algorithm in order to maximize its competitive ratio.

\(^1\)Throughout the thesis, we only focus on minimization problems; the goal of an algorithm is to minimize an underlying “cost”.
Randomization can help to improve the competitive ratio in many online problems. An online algorithm can use random bits to confuse the adversary in designing a worst case sequence. Here, we assume an “oblivious” adversary that knows the code of the algorithm but is not aware of the random bits used by the algorithm. The concept of advice [13] is another tool provided to the online algorithms to improve their competitive ratio. In this setting, certain information in the form of advice bits are provided to the online algorithm. Advice bits can encode any information about the input sequence or the optimal solution, and are generated by a benevolent offline oracle with unbounded computational power. Clearly, the larger is the size of the advice, the better is the competitive ratio. One relevant question in the context of advice is the size of the advice that is necessary and sufficient for an online algorithm to achieve an optimal solution.

1.2 The k-server Problem

The k-server problem is a famous online problem introduced by Manasse et al. [14]. This problem has been the generalization of multiple online problems that have eventually contributed to the development of the online computation field. The input to the problem is a metric space (often an unweighted graph) of large (but constant) size $m$ and an online sequence of requests to the nodes of the metric space. There are $k$ servers ($k < m$) which are located on the metric and move between vertices. Upon a request to a node, at least one of the servers should be located on that node; if before the request, no server is on the node, at least one server should move to that
Chapter 1: Introduction

Figure 1.1: (a) Request is at node C. (b) Server $s_1$ moves to C with a cost 1.

node. Figure 1.1 shows a basic example of the $k$-server problem. The objective of the problem is to serve all requests with minimum cost, where the cost is the total distance travelled by all servers. The $k$-server conjecture states that there exists a deterministic $k$-competitive algorithm for every metric space. This conjecture was also put forward in the same paper by Manasse et al. [14] and has been open for around three decades. The $k$-server problem has been extensively studied on specific metric spaces like paths, trees, complete graphs alongside with the general graphs [15, 16, 2].

The generalize k-server problem, introduced by Koutsoupias and Taylor in 2000 [17], is an extension of the k-server problem in which each server has a (potentially different) metric and only moves on that metric. The generalized CNN problem is quite different from the regular k-server problem, and the techniques used to analyze $k$-server problem do not extend to this generalized variant. The generalized 2-server problem was one of the first problems studied in this area [17, 18]. Consider the generalized 2-server problem in a setting where the metric spaces associated with the

\[2\] Throughout this thesis, we use the colors blue, pink, and green to show the location and trajectory of the request, the online algorithm, and the optimal algorithm, respectively.
two servers are two different paths (on the same set of vertices). This special case of the 2-server problem is equivalent to having a grid metric with a single server on it that serves requests of the form \((i, i)\) by becoming vertically or horizontally aligned with them. Here, a request at node \((i, i)\) on the grid means a request to node \(i\) of the generalized 2-server problem, and the single server becoming “aligned” with the request implies that one of the two servers serve the request. For example, when the request is at \((i, i)\) and the single server is at \((i, j)\) (i.e., it is vertically aligned with the request), that means one of the two servers of the generalized 2-server problem is at \(i\) (and serves the request) while the other is at node \(j\). The CNN problem is a generalization of this setting in which the constraint that the requests on the grid have form \((i, i)\) is relaxed and they can be in the form \((i, j)\) (see Section 1.3 for details).

In addition to the CNN problem, the generalized 2-server problem has been generalized to various other problems which have gained interest among the research community \[19, 1\]. It is believed that studying the generalized \(k\)-server problems will be helpful in introducing new techniques as well as robust online algorithms \[3\]. Thus, there has been a keen interest and drive among the research community to come up with better algorithms for these problems.

### 1.3 The CNN Problem

The CNN problem was formally introduced by Koutsoupias and Taylor when the importance of the generalized \(k\)-server problem was realized \[17\]. The input to the problem is an online sequence of requests to a grid graph on which a single server is located. Upon a request to a node at position \((i, j)\) of the grid, the server should
serve the request by moving in a way that it becomes horizontally or vertically aligned with the request, that is, the server should move to a position $(i, j')$ or $(i', j)$ for some values of $j'$ and $i'$. The goal of an algorithm is to minimize the total distance moved by the grid.

The problem’s name was inspired by the popular news channel CNN. Assume the crew of CNN are given a task of streaming events in the various locations of Manhattan, and assume their powerful cameras have the capability of capturing any event at a given street by being positioned at any intersection of the street and then zooming in from that position. Events represents the online request points, and reporting an event is equivalent to serving the request by being aligned with it. Figure 1.2 shows a basic example of the CNN problem.

The orthogonal CNN problem, introduced by Iwama and Yonezawa \cite{19}, is a special setting of the CNN problem in which every new request should be either $x$-aligned or $y$-aligned with the previous request. This special setting of the CNN problem is the main focus of this thesis and will be introduced in further details in Chapter 2. There are other variants of the CNN problem that have been studied in the past decade.
Augustine and Gravin [1] studied the continuous CNN problem in which the server has to serve the incoming requests that continuously move on the plane, just like cameras covering a sports match. The movements of the request is not necessarily \(x\)- or \(y\)-aligned and it could move in any direction; but the server still has to serve the request by being horizontally or vertically aligned with it. Augustine and Gravin provided upper and lower bounds for the competitive ratio in the continuous setting and showed that their results can be extended to the orthogonal CNN problem. Iwama and Yonezawa [20] introduced the Axis bound CNN problem in which the server can only move in the \(x\) and \(y\) axis. The box-bound CNN problem [21] is another variant of the CNN problem where the requests arrive in a bounded box, and the server can only move on the boundaries of this box. These different versions of the CNN problem were mostly introduced to better understand the generalized \(k\)-server problem. Nevertheless, they have ended up being interesting on their own.

1.4 Literature Review

Online Algorithms have played a major role in theoretical computational science in the past few decades. The advancements in the field of online algorithms are proved helpful in a variety of applications. For example, the online bin packing algorithms are used in areas like memory scheduling, air cargo, managing cloud services on the web and even recording musical pieces on the disks [22, 23, 24]. The classic \(k\)-server problem has been a base for generalizing various online problems, in particular the paging problem, which is crucial in memory management component of operating systems [14, 25], is a special case of the \(k\)-server problem on uniform metrics. The
self-adjusting data structures are often more efficient than static data structures and also have applications in other areas such as data compression \[26, 27\]. This thesis is focused on an interesting variant of the \(k\)-server problem. As such, we start with an overview of the results for the \(k\)-server problem.

Manasse et al. \[14\] introduced the \(k\)-server problem and gave a lower bound of \(k\) for competitive ratio of any deterministic online algorithm; this lower bound holds for any metric space that has at least \(k + 1\) points. For metric spaces having \(k + 1\) points, they proved an upper bound of \(k\) for the competitive ratio. Given these results, they conjectured that there exists a deterministic \(k\)-competitive algorithm for every metric space. This conjectured, named deterministic \(k\)-server conjecture\[^3\] has been the driving force for the research for various online problems and remains open to date. The conjecture has been proven correct for \(k = 2\) and certain metric spaces like trees \[14, 15, 16\]. The Work-function algorithm by Koutsoupias and Papadimitriou \[2\] is the best existing deterministic algorithm for the \(k\)-server problem and is proved to have a competitive ratio of at most \(2k - 1\). In fact it is conjectured to be \(k\)-competitive \[28\]. The work function algorithm can be defined for a variety of problems, and will be explained later in this thesis in the context of the orthogonal CNN problem. The \(k\)-server problem remains open for simple settings such as when there are only 3 servers in the metric or when the metric is a cycle. The best algorithms for both cases is \(2k - 1\) competitive, namely the work-function algorithm. Any algorithm that provides better upper bounds on the competitive ratio for these cases is likely to help answering the \(k\)-server conjecture.

The CNN problem was introduced as a variant of the generalized \(k\)-server problem

\[^3\]There are similar conjectures for the randomized setting.
Chapter 1: Introduction

[29] The CNN problem had no known competitive algorithm for a while. Koutsoupias and Taylor [17] proved that any deterministic online algorithm has a competitive ratio of at least 10.12. Sitters et al. [30] were the first to provide an algorithm with constant competitive ratio for the CNN problem with a high upper bound of $10^5$. Subsequently, they improved the bound to be at most 879 [18]. Notice that there is a huge gap between the upper and lower bounds for the CNN problem.

Iwama and Yonezawa [19] introduced the orthogonal CNN problem and gave an algorithm that achieves a competitive ratio of at most 9. They used a mixture of horizontal and vertical (L-shaped) moves, called knight moves, to serve each request. Augustine and Gravin [1] introduced the continuous version of the CNN problem and provided upper and lower bounds for the competitive ratio in the continuous setting and showed that their results can be extended to the orthogonal CNN problem. In particular, with a simple argument on a unit square, they proved a lower bound of 3 for the orthogonal CNN problem. For their algorithm, they used bishop and rook moves to serve each request. Their algorithm has a competitive ratio of at most 6.464, ultimately reducing the upper bound or competitive from 9 to 6.464. In this thesis, we tighten the gap between the lower bound of 3 and the upper bound of 6.464 by introducing a new algorithm that has a competitive ratio of at most 5. Our algorithm uses tools and components from the work-function algorithm but is quite different from the algorithm itself. As we will see, a variety of algorithms, including the work-function algorithm, are far from optimal for orthogonal CNN problem.
Chapter 2

Preliminary Results

Recall that in the CNN problem, the input is an online sequence of requests to nodes in the grid, and a single server has to vertically or horizontally align with the position of each request to serve it. The orthogonal CNN problem is a restricted variant where the request itself has to be \( x \)-aligned or \( y \)-aligned with its previous position at each given time-step (see Figure 2.1). The restriction introduced in the orthogonal CNN problem helps in better understanding of the nature of the Optimal offline algorithm, which is crucial for introducing better competitive algorithms. Meanwhile the problem remains non-trivial even as the adversary can choose various strategies to harm the online version.

In this chapter, we establish the above intuitions by showing some preliminary results about the orthogonal CNN problem. We start by considering the simple case of unit square, where the metric is a two-by-two grid. This special case is particularly important because no deterministic algorithm has a competitive ratio better than 3

\[ \text{Throughout this thesis, we use the terms “nodes” and “points” interchangeably.} \]
for this seemingly easy metric. We note that this lower bound on a unit square is
the best known general lower bound for the orthogonal CNN problem. We review
this lower bound in Section 2.1. We also provide an algorithm, named CLOCK,
that achieves a competitive ratio of 3 for the unit square metric. While analyzing
CLOCK, we introduce the potential function method which will be used later in our
more complex analysis in Chapter 3. We also review some basic algorithms with bad
competitive ratios. These include greedy algorithms, variants of the knight algorithm
(introduced in [19]), and the work-function algorithm. The results presented in this
chapter are mostly aimed to prepare the reader for Chapter 3, where the main result
is presented.

2.1 Lower Bound

In this section, we review the simple lower bound of 3 for competitive ratio of any
deterministic algorithm for the orthogonal CNN problem. This results is from [1].
Chapter 2: Preliminary Results

Theorem 2.1.1. [1] The competitive ratio of any deterministic algorithm $\text{Alg}$ for the orthogonal CNN problem is at least 3.

Proof. Consider the unit square of Figure 2.2 in which the servers $\text{Alg}$ and $\text{Opt}$ are both located at vertex $A$ before serving any request. The request sequence moves from vertex $B$ to vertex $C$ as shown in Figure 2.2. Without loss of generality, assume $\text{Alg}$ moves to vertex $B$. The sequence continues with repeated requests to $(C, D, A, D, C)$. It is best for the algorithm to move to the position $D$ in order to server these repeated requests at no additional cost. As such, the cost of the algorithm is at least 3 (for the initial move from $A$ to $B$ and the consequent move from $B$ to $D$). On the other hand, $\text{Opt}$ incurs a cost of 1 by simply moving from $A$ to $D$. Hence a lower bound of 3 for the competitive ratio. 

2.2 Clock Algorithm & Potential Function Method

In what follows, we introduce an algorithm named $\text{Clock}$ which has a competitive ratio of 3 when the metric space is formed by a unit square. Hence, $\text{Clock}$ is the
optimal algorithm for the orthogonal CNN problem for the unit square metric. The algorithm is indeed very simple: when the request moves on the vertices of a square, move the server in the clockwise direction and stops as soon as the request is served.

We provide an upper bound for the competitive ratio of CLOCK using the potential function method, which is a widely used method to analyze online algorithms, and is also the basis for our analysis in Chapter 3. In the potential function method, at each given time, the configuration of the offline and online algorithms are mapped to a positive integer using a potential function. The values of the potential function (simply called the potential) should be bounded by a constant, that is, the potential function maps configurations to positive integers independent of the input length. The amortized cost of an algorithm at time \( t \) is the summation of its actual cost and the difference in potential function before and after serving the request at time \( t \).

In order to bound the competitive ratio of an algorithm, it is sufficient to bound its amortized cost at each given time. In other words, if we can show that the amortized cost of an algorithm ALG at each given time-step is within a factor \( c \) of the cost of OPT at time \( t \), then ALG has a competitive ratio of at most \( c \). We refer the reader to the book by Borodin and El-Yaniv [31] for more information on the potential function method.

**Theorem 2.2.1.** CLOCK has a competitive ratio of 3 when the request moves on a unit square.

**Proof.** Define the directed distance between two vertices as the clockwise distance between them in the unit square. Define the potential to be the directed distance between the location the server of CLOCK and that of OPT. There are a few cases
Chapter 2: Preliminary Results

Figure 2.3: The worst-case event in the analysis of CLOCK (a) Before the move. (b) After the move.

to consider. First, for events that neither of CLOCK and OPT move their servers, the potential and amortized cost of CLOCK is 0, which is bounded above by 3 times the cost 0 of OPT. For an event where OPT does not move and CLOCK moves its server, CLOCK’s server gets closer to OPT’s server, and the directed distance between OPT and CLOCK decreases by 1. In this case the cost of the algorithm is 1 and the difference in potential is -1, giving an amortized cost of 0, which is again bounded by the cost of OPT. Next, if OPT and CLOCK both move from the same position and in the same direction, their directed distance does not change and the difference in potential will be 0; the amortized cost of the algorithm is 1 which is no more than 3 times the cost 1 of OPT. The worst case happens when both servers are at the same position, the request moves to diagonal vertex as shown in Figure 2.3 and the two servers move in different directions making increasing their directed from 0 to 2, i.e., the potential will be added by 2. The amortized cost of CLOCK will be 3 which is bounded by 3 times the cost of 1 of OPT.

One can easily extend CLOCK to a general grid: if the algorithm’s server and the request are aligned, do nothing; otherwise, form a rectangle that has the server
Figure 2.4: The worst-case input for \textsc{Clock} on a $d$ by $d$ grid. The request moves on a rectangle of size 1 by $d$ and the server of \textsc{Clock} “chases” it by repeatedly taking a distance of $d$.

and request as its two opposite corners, and move the server on the clockwise direction on that rectangle. Unfortunately however, this algorithm does not have a good competitive ratio for general metrics:

\textbf{Theorem 2.2.2.} \textsc{Clock} has a competitive ratio of at least $d$ when the grid is of size $d \times d$.

\textit{Proof.} Consider a rectangle $ABCD$ of size 1 by $d$ as a subgraph of the input metric (see Figure 2.4). Assume the server is at $A$ and the request is at $D$. The request sequence continues with requests $(CBAD)^n$, where $n$ is a large number. The Clock Algorithm serves the request in the clockwise direction irrespective of the distance it has to move for each request. As such, the algorithm incurs a total cost of $2n(1 + d)$. An optimal offline algorithm just moves on the shorter side for each phase of sequence and pays a cost of $2n$. Therefore, the algorithm has a cost that is a factor $1 + d$ of $\text{Opt}$.

\hfill $\square$
2.3 Basic Algorithms

In this section, we review two algorithms for the orthogonal CNN problem. These include the Greedy algorithm, which is the most basic algorithm for the problem, and the work-function algorithm (Wf) which is widely used in the context of the k-server problem. We show that none of these algorithms have a good competitive ratio. Our worst-case sequences not only help in understanding these basic algorithms, but also help in devising an algorithm that is equipped to perform well for these sequences.

The Greedy Algorithm: Greedy algorithm is the first and foremost algorithm that comes to mind while analyzing the online problems. The Greedy algorithm for the Orthogonal CNN problem works as follows. For every new request that is not aligned with the server, the algorithm has two options to serve the request, namely, to move the server horizontally to serve the request via a vertical alignment or to move the server vertically to serve the request via a horizontal alignment. The Greedy always chooses the option that involves the smaller movement of the server.

Theorem 2.3.1. The Greedy algorithm is not competitive, that is, its competitive ratio grows with the input length.

Proof. Consider the sequence $\langle X, X', X \rangle^n$ on the metric in Figure 2.5 that is, the request repeatedly moves between $X$ and $X'$. Assuming $d > 1$, the Greedy algorithm prefers to serve each request by taking the shorter path of length 1 instead of a longer path of length $d$. This would give a total cost of $2n$ for Greedy. The optimal algorithm, on the other hand, takes a one-time longer path at a cost of $d$, hence a competitive ratio of at least $\frac{2n}{d} = n$ (In worst case, $d$ is 2).
Chapter 2: Preliminary Results

Figure 2.5: When the request repeatedly moves between $X$ and $X'$, the Greedy algorithm prefers to repeatedly between $A$ and $A'$ while Opt moves once from $A$ to $X$.

The Work-Function Algorithm: The work function algorithm ($W_f$) was first introduced in the context of the $k$-server problem. This algorithm, however, can be applied to a variety of other problems, including the CNN problem. In this section, we review the work-function algorithm for the orthogonal CNN problem. The algorithm that we present in the next chapter uses some of the concepts introduced in this section.

Define a configuration as the position of the single server at a given time. Assume at time $t$ (i.e., on the $t$’th index of the input sequence), the request moves from its current position to a neighboring vertex on the grid. If the new location of the request is not aligned with the server’s position, $W_f$ algorithm has two options for serving the new request either horizontally or vertically. Each option defines a new configuration. The work-function at time $t$ of a given point $P$ in the grid, denoted by $wf(t, P)$, is the cost of an optimal algorithm for serving all request up and including the $t$’th
request such that the server is located at position $P$ after serving these requests. We note that, in order to compute the work-functions at time $t$, the requests that are made after $t$ are not needed. As such, an online algorithm can compute the work-function values at each given time (in fact, it can be done efficiently using dynamic programming). Now, we are ready to describe the work-function algorithm.

Assume that the request moves from its current position to a new position at time $t$. If the server of $W_f$ is still aligned with the request, it does not move. Otherwise, let $P_h$ and $P_v$ be the two candidate points to move the server to serve the new request (via a horizontal or vertical alignment). Also, let $d_h$ and $d_v$ be the distances of the current position of the server to $P_h$ and $P_v$, respectively. The $W_f$ algorithm computes the values of $c_h = w_f(t, P_h) + d_h$ and $c_v = w_f(t, P_v) + d_v$. If $c_h \leq c_v$, the algorithm moves the server to $P_h$; otherwise, it moves the server to $P_v$.

Given that $W_f$ algorithm is the best existing deterministic algorithm for the $k$-server problem, one might ask whether it provides close-to-optimal solutions for the orthogonal CNN problem. The following theorem answers this question in the negative.

**Theorem 2.3.2.** The work-function algorithm $(W_f)$ has a competitive ratio of at least $d + 1$ when the grid is of size $d \times d$.

**Proof.** Consider Figure 2.6(a-e) in which the server and request are located at the top of the grid and at a distance 2 from each other (part (a) of the figure). When the request moves down, the server has to options to server it via a horizontal or vertical alignment. Here we have $c_h = 1+1$ and $c_v = 2+2$. Consequently the server selects the horizontal alignment and moves down to serve the request (part (b) of the figure).
Chapter 2: Preliminary Results

Figure 2.6: The worst-case sequence for the Work-Function (WF) algorithm. The request and the WF’s server move from top of the grid to the bottom while maintaining a distance of 2. Meanwhile OPT stay on top and serves all requests by a single move of cost 2 at the beginning.

In the consequent moves, the request repeatedly moves downwards until it gets to the bottom of the grid. Each move increases the work-function of the two candidate points $P_h$ and $P_v$ by 1 unit. Meanwhile, the distance of these two points from the server remain the same, namely 1 and 2 for $P_h$ and $P_v$, respectively. Consequently, the algorithm repeatedly chooses $P_h$ over $P_v$ and moves downwards to serve the first $d$ requests (part (c-d) of the figure). Meanwhile, OPT moves two units to the right to serve all requests. So far, WF’s server has moved $d$ units and is at distance $d + 2$ of OPT’s server. The sequence continues by repeatedly moving the request on the two axes that intersect at OPT’s position. Assuming the request keeps alternating between these two axes, WF either incurs a further cost of at least $d + 1$ to get to the position of OPT or incurs a cost proportional to the input length. In the best case for the algorithm, its cost would be at least $2d + 2$. Given that OPT has a cost of 2, the competitive ratio will be at least $2d + 2/2 = d + 1$. 

□
Chapter 3

The Walk and Jump Algorithm

In this chapter, we introduce the Walk and Jump (Wj) algorithm for the Orthogonal CNN problem. It has a competitive ratio of at most 5 and we prove it using the potential function method. We have not been able to come up with sequences for which the cost of Wj is more than three times the optimal cost. As such, we conjecture that a tighter analysis might give an (optimal) upper bound of 3 for the competitive ratio of this algorithm. Some of the “hard” instances of the problem that have inspired us in designing the Wj algorithm are included in the appendix. These instances not only provide an insight on how the algorithm works but also give intuitions on how the potential function develops throughout the course of serving these sequences.

The Wj algorithm uses the concept of work function. As discussed in Section 2.3, the work function of any point $(x, y)$ at time $t$ is the minimum cost of serving the first $t$ requests and ending up at position $(x, y)$. The algorithm maintains all these values at any given time and for any given location on the grid.
The following terms are defined for any given time $t$ and later used for defining the WJ algorithm and its analysis.

- **Dominance**: A point $N_i$ is said to “dominate” node $N_j$ if we have $w_f(N_j) = w_f(N_i) + d_{ij}$, where $w_f(N_i)$ and $w_f(N_j)$ respectively denote the work function values for $N_i$ and $N_j$ and $d_{ij}$ is the distance between the two points.

- **Independent points**: The set of points that are not dominated by any other point.

- **Server axis**: in order to serve the request, the server stays aligned with the request’s position in the horizontal or vertical direction. The axis between the server and the request is called the server axis. If the server and requests are co-located, either of the two axes that pass through their position can be considered as a server axis.

- **Dangerous point**: Let $P$ be any independent point on the server’s axis. For any independent point $P'$ that is not on the server’s axis, the rectangle that has $P$ and $P'$ as its two opposite corners finds a corner on the request’s position and one other corner on a point that we call a “dangerous point” associated with $P$. In the special case when the request itself is an independent point, we assume all independent points on the server axis are dangerous points (see Figure 3.1).

- **Idle move**: When the request moves in the axis of the server, the server does not make any move. This is called an idle move.\footnote{With respect to terminology, “idle” refers to the algorithm (which does not move), and “move” refers to the request (which changes its location).}
Figure 3.1: An illustration of dangerous points. Here, $IP_1$ and $IP_2$ are independent points on the server’s axis. $A$ and $B$ are dangerous points associated with $IP_1$, $IP_4$ and $IP_2$, $IP_4$, respectively. Moreover, since the request is located on an independent point, $IP_1$ and $IP_2$ are dangerous points associated with $IP_3$.

- **Walk and Jump moves:** When the request moves one unit in a direction orthogonal to an algorithm $Alg$’s axis, there are two potential points to which the server of $Alg$ can move to serve the request. One point is one unit away from the original position of the server; if the server moves to that point, the move is called a “walk move” (we say $Alg$ walks to serve the request); this is when the server moves parallel to the request. The other move is when the server moves to the original position of the request; this is called a “jump move” (we say $Alg$ jumps to serve the request) and involve moving a potentially much-larger distance. Note that $Alg$ can refer to either $WJ$ or $OPT$.

Provided with the above terminology, we are ready to define the $WJ$ algorithm. At each given time $t$, the algorithm computes (updates) the work-function values for all points$^3$. If the request moves on the server axis, the algorithm serves it without moving. When the request moves on a direction orthogonal to the server axis, the

\[2\text{In principle, the request might move from a node } u \text{ to a node } v \text{ at distance } d > 1. \text{ Such a move, however, can be broken down into } d \text{ unit-distance moves without any change in the analysis.} \]

\[3\text{Although we are not concerned with the time complexity, we note that work function values can be found in polynomial time via dynamic programming [31].} \]
algorithm has to make a decision between a walk move or a jump move to serve the request. This decision is made according to the following simple rule. If after potentially serving the request with a walk move, there is at least one independent point in the server axis that dominates all the dangerous points associated with it, the algorithm selects the walk move. Otherwise, it selects the jump move.

3.1 Potential Function

We use the potential function method to prove that the competitive ratio of the WJ algorithm is at most 5. We refer to Section 2.2 for a review of the potential function method. Our potential definition is the sum of two components. We refer to the first component as the “main component” and the second component as the “predominance” of the Optimal server.

We first explain the predominance of OPT, which is slightly simpler. Let’s refer to the axis formed between OPT and the request as the ”OPT-axis”. In case OPT and request are at the same position, then OPT-axis is the axis between OPT and server. In a special case when all OPT, server, and request are collocated, any of the two axes that passes through them can be OPT-axis. Figure 3.2 shows different cases for the OPT-axis. The predominance of OPT is defined as follows:

$$Predominance = \max_{\forall P \in OPT-axis} \{wf(OPT) - wf(P) + d(OPT, P)\}$$

Note that for any point P on the OPT-axis that is dominated by OPT, the value of $wf(Opt) - wf(P) + d(OPT, P)$ is 0. In particular, if all points on the OPT-axis are dominated by OPT, the predominance is 0. On the other hand, if there is an
independent point IP with relatively small work function at a “high distance” from Opt, then the predominance is a large value. To see the intuition behind the large value of predominance in this case, consider a situation in which the request moves to Opt’s location and starts walking indefinitely on the two axes that pass through Opt. Clearly, Opt incurs no additional cost, while a competitive algorithm, in particular WJ, eventually moves the server to the Opt’s position. The presence of an IP on the Opt-axis (which causes a large predominance), however, delays the eventual move of WJ to the Opt’s position. The small value of the work function for the IP implies that it dominates its dangerous points for a longer time and the eventual move of WJ to Opt’s position is delayed. Consequently, the algorithm pays extra cost until it gets to Opt’s position. This undesirable situation projects to higher potential in this relatively bad configuration for the algorithm.

The second component of the potential depends on whether WJ and Opt’s servers are on the same axis or not. Intuitively speaking, these cases are treated differently because an adversary continues the sequence in a different way depending on whether Opt and WJ servers are on the same axis or not.
(i) When \( W_j \) and \( \text{OPT} \) servers are on the same axis, the main component of the potential is:

\[
\Phi_{\text{main}} = \max\{\phi_{ia}, \phi_{ib}\}
\]

\[
\phi_{ia} = d(s, \text{OPT}) \quad \text{(distance between } W_j \text{'s server and } \text{OPT}'\text{'s)}
\]

\[
\phi_{ib} = \max_{\forall IP \in \text{orthogonal-axis}} \{wf(\text{OPT}) - wf(IP) + d(\text{server, IP})\}
\]

The potential in case (ia) captures a situation where the request moves to the \( \text{OPT} \)'s position and indefinitely moves orthogonal to the server's axis. After making some initial walks, the server of \( W_j \) eventually has to jump to the \( \text{OPT} \)'s location (and pays a cost of \( \phi_{ia} \)). Note that, the server might walk for a certain distance before such jump; the cost paid the algorithm for these walks is captured by the predominance factor of the potential.

In case (ib), the “orthogonal axis” is the axis that is orthogonal to the \( \text{OPT} \)-axis at the location of the request. Intuitively, the potential in case (ib) captures a situation where the adversary moves the request along the orthogonal axis to the extent that \( W_j \) eventually moves its server to that axis; the request continues back to the position of \( \text{OPT} \) (and keeps moving on its axes indefinitely). The algorithm “makes a mistake” by jumping to the orthogonal axis while \( \text{OPT} \) keeps walking. The potential in case (ib) captures the extra distance that \( W_j \) moves because of the mistake in this case (this would become more clear in worst case examples in appendix.).

(ii) Consider the case when \( W_j \)'s server and \( \text{OPT} \) are not on the same axis (they
Chapter 3: The Walk and Jump Algorithm

(a) server and OPT on the same axis (b) server and OPT on the different axis (c) Special Case

Figure 3.3: Different cases for the main component of the potential.

are not co-linear on the grid). In this case, any independent point IP on the server’s axis, along with OPT’s server and the request form three corners of a rectangle; let X(IP) be the fourth corner. The potential in this case is defined as follows:

$$\phi_{ii} = \max_{\forall IP \in \text{server’s axis}} \{ (w_f(OPT) + d(OPT, X(IP))) - (w_f(IP) + d(IP, X(IP))) \}$$

$$+ d(\text{server}, OPT)$$

By definition of the WJ algorithm, at least one IP exists on the server axis that dominates all its dangerous points; X(IP) is one of those dangerous points. Intuitively, the main component of the potential in this case captures the situation where the presence of the IP delays the server from moving towards OPT’s position in the opposite axis. The first component (in the max form) of $\phi_{(ii)}$ captures the extra walks the WJ server takes before jumping to the OPT’s axis and eventually moving to the OPT’s position with an additional cost of $d(s, Opt)$. As an example, in Figure 3.3(b), the $\phi_{ii}$ is $(OPT + d_2) - (IP + d_3) + d_1 + d_2 + d_3$.

We note that in the special case where request, WJ’s server, and OPT’s server are
collocated, there are two ways to define the potential. From the two axes that pass through the common point, one should be considered as the OPT-axis (the points on which define the predominance) and the other should be considered as the orthogonal axis that defines $\phi_{\text{main}}$. In what follows, we show the potential remains the same in both cases (see Figure 3.3(c)):

- Assuming the axis that contains the $IP_2$ is considered as OPT-axis and the axis containing $IP_1$ is regarded as orthogonal-axis, the main component of the potential would be $\phi_{ib}$ as $\phi_{ia}$ is 0. In this case we have:

\[
\text{predominance} = w_f(OPT) + d_2 - w_f(IP_2) \\
\phi_{ib} = w_f(OPT) - w_f(IP_1) + d_1 \\
\phi = 2w_f(OPT) - w_f(IP_1) - w_f(IP_2) + d_1 + d_2
\]

- Consider the case where the axis containing $IP_1$ is regarded as the Opt-axis and the axis containing $IP_2$ is considered as the orthogonal-axis. In this case we have:

\[
\text{predominance} = w_f(Opt) + d_1 - w_f(IP_1) \\
\phi_{ib} = w_f(Opt) - w(IP_2) + d_2 \\
\phi = 2w_f(Opt) - w(IP_1) - w_f(IP_2) + d_1 + d_2
\]

Thus, the potential is the same in both cases. This equality is important when we move from one state of the potential to another, as will be clarified later.

Throughout our analysis, we say $W_j$ is at “state i” if the servers of $W_j$ and OPT are on the same axis and at state ii otherwise. Moreover, when the algorithm is at
state $i$, we say it is at state $ia$, if $\phi_{ia} > \phi_{ib}$; otherwise, the algorithm is at state $ib$. In other words, the state of the algorithm is in accordance to the term that defines the value of the potential.

If the algorithm is at state $ib$, the main component of the potential is defined through a *special independent point* $P^*$ which maximizes the term $w_f(\text{Opt}) - w_f(\text{IP}) + d(s, IP)$. Similarly, in state $ii$, the special independent point is a point that maximizes $(w_f(\text{Opt}) + d(\text{Opt}, X)) - (w_f(IP) + d(IP, X))$. Note that there is no special independent point when $WJ$ is at state $ia$. Moreover, at any given time, the predominance component of the potential is defined through a *predominance point* $PM^*$ that maximizes the value of $w_f(\text{Opt}) + d(\text{Opt}, P) - w_f(P)$. In case there are multiple (independent) points that maximize the terms in potential, any of them can be considered as the special independent point or predominance points.

### 3.2 Basic Observations

In this section, we establish some basic properties for the optimal algorithm as well as the potential function. These properties are later used when we bound the amortized cost of $WJ$ in the potential-function framework.

#### 3.2.1 Properties of Opt

The following two lemmas establish natural, to-be-expected properties of the optimal offline algorithm which are later referred to in our analysis.

**Lemma 3.2.1.** There is an optimal algorithm $\text{Opt}$ that is lazy, i.e., $\text{Opt}$ moves its
Chapter 3: The Walk and Jump Algorithm

Figure 3.4: (a) A non-lazy $Opt_{nl}$ walks with the request. (b) A lazy $OPT$ jumps when the request moves orthogonal.

Proof. Consider an optimal algorithm $Opt_{nl}$ that makes a non-lazy move, that is, it moves the server from point $x$ to $x'$ while the request has moved on the axis formed by $x$ and $x'$. One can delay the move without increasing the cost of $Opt_{nl}$. Only when the request eventually moves on the orthogonal axis formed by $x$ and $x'$, the algorithm moves the server to that axis. Clearly, the cost of the algorithm is not increased and at least one of its non-lazy moves are removed. Repeating this for all non-lazy moves completes the proof. Figure 3.4 provides an illustration.

Lemma 3.2.2. There is an optimal algorithm whose server is always located on independent points.

Proof. Independent points are the set of points in the grid with the minimum work function values as the work function of all other points depend on them. A lazy optimal offline algorithm that knows the entire sequence calculates the work function values of the grid for all the request sequence until the end. Dynamic programming can be used to calculate these values. This is followed by a backtracking method in which $OPT$ marks the best independent points at each instance. Thus, all the
requests are served using the marked points. Consequently, Optimal offline algorithm pays the minimum cost.

\[ \square \]

### 3.2.2 Properties of the Potential Function

In what follows, we establish two lemmas that concern the properties of the potential function.

**Lemma 3.2.3.** Assuming Opt, Wj, and the request are all located on the same axis, their ordering does not have an impact on the value of the potential.

**Proof.** The value of predominance does not depend on the location of request and the server. As such, their relative order with Opt’s position does not impact predominance. For the main component of potential, we note that \( \phi_{ia} \) is the distance between the servers of Opt and Wj, which is independent of the relative positions; \( \phi_{ii} \) is defined when server and Opt are not in the same axis and is not relevant in the case of this lemma. Finally, we have \( \phi_{ib} = wf(Opt) - wf(IP) + d(server, IP) \), in which the work function values are independent of the servers and the request positions. The distance between the server and the independent point is also independent of the relative positions.

\[ \square \]

**Lemma 3.2.4.** The predominance increases by at most 2 when Opt walks to serve the request.

**Proof.** For this proof, it helps to see Figure 3.5. Assume that at time \( t \), the request

---

4In the figures that we use to illustrate our analysis, we simply use “Opt” to indicate the value of the work function of the point that Opt is located at time \( t \) (before serving the request). Similarly, \( R \) and \( s \) respectively indicates the work-function of the points that the request and server are located at time \( t \). Finally, \( IP_x \) indicates the work-function of the independent point \( IP_x \) at time \( t \).
Figure 3.5: The maximum increase in Predominance is 2.

moves orthogonal to OPT’s axis and OPT walks to serve the request. There are two possibilities for $PM^*$: it is either an independent point on the OPT-axis or it is the position of the request which is dominated by an independent point on the axis orthogonal to the OPT-axis. If $PM^*$ is an independent point on the OPT’s axis, then there is no change in predominance; this is because the value of the predominance at time $t + 1$ becomes $(wf(OPT) + 1) + d(OPT, IP_2) - (wf(IP_2) + 1)$, which is equal to its value of $wf(OPT) + d(OPT, IP_2) - wf(IP_2)$ at time $t$. If $P$ is the position of the request, dominated by some point on the server’s axis, then the predominance increases by at most 2. In Figure 3.5, the request position is dominated by $IP_3$. The predominance at time $t$ is $wf(OPT) + d_3 - wf(R)$. At time $t+1$, The predominance will be $wf(OPT) + 1 + d_3 - (R - 1)$ (R is dominated by $IP_3$ and the request moves towards $IP_3$).

3.3 The Main Proof via Case Analysis

In this section, we prove that the Walk and Jump algorithm has a competitive ratio of at most 5. We use the potential function defined in the previous section.
Chapter 3: The Walk and Jump Algorithm

We consider the following basic cases with respect to how WJ and OPT move their servers to serve a given request. Each case has a few sub-cases which will be discussed in detail in subsequent sections.

1. Neither OPT nor WJ move their servers.

2. OPT’s server does not move while WJ’s server move.

3. WJ’s server does not move while OPT’s server move.

4. Both OPT and WJ move their servers.

We consider the above cases separately and in different sections. For each case, we prove that the amortized cost of WJ is no more than 5 times the cost of OPT, which ultimately shows the competitive ratio of the algorithm is no more than 5.

In what follows, particularly in the pictures, ‘s’ stands for the server of the WJ algorithm. ‘OPT’ denotes the Optimal server. The current request is shown with letter ‘R’. Independent points and dangerous points are denoted by ‘IP’ and ‘DP’ respectively.

3.3.1 Neither Opt nor WJ move their servers

In this case, servers of WJ and OPT’s are located on the same axis and the new request moves on their shared axis. Hence, they both serve the request without moving their servers. The cost of OPT is 0, and we should show the amortized cost is no more than 0. We note that there is no change in the predominance as the OPT-axis remains the same. The algorithm is either in state ia or in state ib, and we should consider the following state transitions:
1. ia to ia: The initial potential is the distance between the Wj’s server and OPT and the final potential will be same as well. The potential remains the same regardless of whether request moves away or closer to the Wj. Refer to Figure 3.6(a).

2. ib to ib: There is no change in the potential when the request moves away from Wj. But, when the request moves closer to the Wj, the potential decreases by 2. The amortized cost will not be more than 0. Details can be found in Figure 3.6(b, c).

3. ib to ia: The potential decreases when the request moves closer to Wj and the state can change from ib to ia. The decrease in potential shows that the amortized cost is less than zero. Figure 3.6(d) shows this case.

4. ia to ib: we show this transition cannot happen because the value of $\phi_{ia}$ remains the same while the value of $\phi_{ib}$ does not increase, hence, if the value of $\max\{\phi_{ia}, \phi_{ib}\}$ is $\phi_{ia}$ before serving the request, it remains as such after serving the request. The value of $\phi_{ia}$ is simply the distance between Wj and OPT and neither algorithm moves (hence no change in $\phi_{ia}$). The fact that $\phi_{ib}$ does not increase is direct from the discussion in parts 2 and 3.

From the above discussion, We can conclude the following lemma.

**Lemma 3.3.1.** In an event that both Wj and OPT serve the request without moving their servers, the amortized cost of Wj is no more than 0, which is no more than 5 times the cost $\theta$ of OPT.
3.3.2 Opt’s server does not move while Wj’s server move.

In this case, the request moves in the Opt-axis and orthogonal to the Wj’s server axis. The amortized cost should be no more than 0, provided that Opt does not move its server. The following are the state changes that can happen in this case.

1. ii to ii: Wj and Opt are not on the same axis and Wj walks to serve the request. The predominance remains the same. We claim that the change in potential is -1, regardless of whether the request gets closer to or further from Opt. This implies that the amortized cost is zero. We observe that $P^\ast$ remains the same because it maximizes the value of $(wf(Opt) + d(Opt), X(IP)) -$
(\(wf(IP) + d(IP, X(IP))\)) over all IP’s, and all components of this sum stay the same except for \(wf(IP)\) which is increased by 1 for all IP’s and \(d(IP, X(IP))\) which is either increased or decreased by 1 for all IP’s. Given the fact that \(P^*\) stays the same, it is easy to verify that the difference in potential is -1. Details can be found in Figure 3.7(a, b).

2. ii to ib: \(W_j\) jumps to the \(OPT\)-axis to serve the request. The predominance remains the same. We show that \(P^*\) also remains the same. Before the jump, \(P^*\) was maximizing \((wf(OPT) + d(OPT, X(IP))) - (wf(IP) + d(IP, X(IP)))\) over all IP’s on the \(W_j\)'s server axis; this is equal to \((wf(OPT) + d(R, IP)) - (wf(IP) + d(R, OPT))\). After the state changes, \(P^*\) maximizes \((wf(OPT) - wf(IP) + d(s, IP))\) over all IP’s on the axis orthogonal to \(OPT\)-axis; this term is equal to \((wf(OPT) - wf(IP) + d(s, R) + d(R, IP))\). We note that the two axes over which the max is taken are the same (see Figure 3.7(c, d)). Meanwhile, all components of the two sums remain the same except for \(d(R, IP) - wf(IP)\) which changes in the same way for all IP’s. That is, the two points that maximize the two sums are the same, i.e., \(P^*\) remains the same after \(W_j\) jumps. Given this, it is rather easy to verify that amortized cost is no more than zero. Details can be found in Figure 3.7(c, d).

3. ii to ia: In what follows, we prove that this case is not possible. In the proof, we use the definitions of the algorithm and the potential function (see Figure 3.7(e)). Consider a base case in which \(W_j\) and \(OPT\) are not on the same axis. By definition of \(W_j\), there is an independent point on the server axis that dominates all dangerous points associated with it. Assume \(IP'\) is such point at
a distance $d'$ from the request. The following statements hold according to the algorithm's definition.

\[
wf(OPT) + d' \geq wf(IP') + d_3 \quad (1)
\]
\[
d' - wf(IP') \geq d_3 - wf(OPT) \quad (2)
\]

Given that the state transition is from ii to ia, WJ jumps to the OPT-axis to serve the request. Let $IP_1$ be the point that maximizes the main component of the potential at time $t$, i.e., $IP_1$ is $P^*$. We have:

\[
w(\text{OPT}) + d_2 - wf(IP_1) - d_3 \geq wf(\text{OPT}) + d' - wf(IP') - d_3 \quad (3)
\]
\[
d_2 - wf(IP_1) \geq d' - wf(IP') \quad (4)
\]

From statements 2 and 4, the following can be deduced:

\[
d_2 - wf(IP_1) \geq d_3 - wf(OPT) \quad (5)
\]
\[
d_3 \leq wf(OPT) + d_2 - wf(IP_1) \quad (6)
\]

The state transition is from ii to ia only if $\phi_{ia} \geq \phi_{ib}$ after serving the request. This holds under the following conditions:

\[
d_3 > wf(OPT) - wf(IP_1) - 1 + d_2 + 1
\]
\[
d_3 > wf(OPT) - wf(IP_1) + d_2 \quad (7)
\]

The statements 6 and 7 contradict each other and hence the state transition from ii to ia cannot happen.
Chapter 3: The Walk and Jump Algorithm

Figure 3.7: Amortized cost analysis for an event when OPT’s server does not move, while Wj’s server moves to serve the request.
4. ib to ib: In this case, \(W_j\) and \(\text{OPT}\) are on the same axis and the request is at \(\text{OPT}\)'s position. The new request moves orthogonal to \(W_j\)'s axis and \(W_j\) jumps to the \(\text{OPT}\)'s position to serve the request.

Let \(A\) be \(P^*\) and \(B\) be \(PM^*\) just before the jump. We claim that \(A\) becomes \(PM^*\) and \(B\) becomes \(P^*\) after the jump. Before the jump, \(P^*(A)\) was maximizing \(wf(\text{OPT}) - wf(IP) + d(s, IP)\) over all IP’s on the \(W_j\)'s axis. Now, after the jump, the new \(PM^*\) will be point that maximizes \(wf(\text{OPT}) + d(\text{OPT}, IP) - wf(IP)\) on the same axis \((d(\text{OPT}, IP) = d(R, IP))\). All the terms in these two formulas are the same except \(d(s, R)\) which changes in the same way for all IP’s. This means that \(PM^*\) after the jump is indeed \(A\). Similarly, before the jump \(PM^*(B)\) was maximizing \((wf(\text{OPT}) -wf(IP) + d(\text{OPT}, IP)\) over all IP’s on the \(\text{OPT}\)'s axis. After the jump, the new \(P^*\) will be point that maximizes \(wf(\text{OPT}) - wf(IP) + d(s, IP)\) on the same axis; this value is indeed \(wf(\text{OPT}) - wf(IP) - 1 + d(\text{OPT}, IP) + 1\). All terms in these two formulas are the same. This means that \(P^*\) after the jump is indeed \(B\). Given the fact that \(A\) and \(B\) changed the roles, it is straightforward to verify that amortized cost is indeed no more than 0. Details can be found in Figure 3.7(f).

5. ib to ii: In this case, \(W_j\) and \(\text{OPT}\) are on the same axis and the request is at \(\text{OPT}\). The new request moves orthogonal to the \(W_j\)'s axis and \(W_j\) walks to serve the request. A similar analysis to case 4 shows that \(P^*\) and \(PM^*\) interchange their roles. Given this fact, it is straightforward to compute the change in potential as -1. The amortized cost is consequently 0 as \(W_j\) pays 1. Details can be found in Figure 3.7(g).
6. ia to ii and ia to ib: These state transitions are possible and have similar initial settings as cases 4 and 5. The initial state is (ia) because $\phi_{ia} \geq \phi_{ib}$. cases 4 and 5 discussed above guarantee that the amortized cost is no more than 0 when the state changes from ib to ii and ib to ib. These observations are enough to prove that the amortized cost will be no more than 0 when state transition is from ia to ii or from ia to ib.

From the above discussion, We can conclude the following lemma.

**Lemma 3.3.2.** In an event that Wj moves to serve the request while OPT does not move, the amortized cost of Wj is no more than 0, which is no more than 5 times the cost 0 of OPT.

### 3.3.3 WJ’s server does not move while Opt’s server move.

In this case, the request moves orthogonal to the OPT-axis. The amortized cost should be less than 5 times OPT, given that WJ pays no cost. The followings are different state transition that we should consider:

1. ii to ii: The WJ’s server and OPT are not in the same axis and the request moves orthogonal to the OPT’s axis. OPT walks in this case to serve the request. Consider a worst-case scenario for the algorithm in which the predominance is increased by the maximum possible value. By Lemma 3.2.4, in this case $PM^*$ stays the same; indeed, $PM^*$ is the position of R which is dominated by an independent point X (on the WJ’s axis) such that X gets closer to the request after the request moves ($X$ is $IP_3$ and $IP_2$ in Figures 3.8(a, b) respectively).
Moreover, a similar analysis to case 1 of section 3.3.2 proves that $P^*$ also stays the same after Opt's move. Given that $P M^*$ and $P^*$ remain the same and the predominance is increased by at most 2, it is straightforward to show that the difference in potential is at most 5 when the request moves away from $P^*$ and 1 when the request moves closer to $P^*$. Given that $W_j$ did not move, the amortized cost is at most 5, while the cost of Opt is at least 1. Details can be seen in Figure 3.8(a, b).

2. ii to ib: In this case, the request moves orthogonal to the Opt’s axis and Opt jumps to the requested point. This case has the similar initial setting as case 1 with a difference that the requested point is dominated by Opt at time $t$. The request point at time $t+1$ becomes an independent point and Opt jumps on this point (Lemma 3.4). A similar analysis to case 4 of section 3.3.2 shows that the $P^*$ and $P M^*$ interchange their roles after the move. Given these observations, we can verify that, when the request moves away from the $W_j$’s server, the amortized cost is 3 times the cost of Opt and when the request gets closer to $W_j$ the amortized cost is $3 \text{Opt} - 2$. Details can be found in Figure 3.8(c, d).

3. ii to ia: This case is not possible as discussed in subsection 3.3.2.

4. ib to ii: In this case, $W_j$ and Opt are on the same axis and the request and $W_j$ are at the same position. The new request moves orthogonal to Opt and Opt walks to serve the request; as such, the Opt-axis remains the same. A similar analysis of case 1 shows that the predominance increases by at most 2 while $P M^*$ remains the same. Moreover, a similar analysis of case 2 of Section 3.3.2 proves
that $P^*$ remains the same after the request moves. Given that $P^*$ and $PM^*$ remain the same and predominance increases by at most 2, it is straightforward to see the potential increases respectively by at most 5 and 3 when the request moves away and closer to $P^*$. In summary, the amortized cost is no more than 5 while the cost of OPT is 1. Refer to Figure 3.8(e, f) for details.

5. ib to ib: In this case, the request moves orthogonal to the OPT’s axis and OPT jumps to serve the request. This case has the similar initial setting as Case 4 with a difference that the requested point is dominated by OPT at time $t$. The request point at time $t + 1$ becomes an independent point and OPT jumps to this point to serve the request. A similar analysis to Case 4 of Section 3.3.2 shows that the $P^*$ and $PM^*$ interchange their roles after the move. Given these observations, we can verify that the amortized cost is 3 times the cost of OPT. Details can be found in Figure 3.8(g).

6. ia to ii and ia to ib: The analyses in these cases reduce to the similar cases when the initial state is ib. Note that the analysis in Cases 4 and 5 hold when the state transition is from ib to ii and ib to ib. Meanwhile, since the initial state is ia, we have $\phi_{ia} \geq \phi_{ib}$. Hence, the same analyses can be applied to prove that the amortized cost will be within 5 times the cost of OPT when the state transition is from ia to ii or from ia to ib.

From the above discussion, we can conclude the following lemma.

**Lemma 3.3.3.** In an event that $W_j$ does not move to serve the request while OPT moves its server, the amortized cost of $W_j$ is no more than 5 times the
cost of Opt.

3.3.4 Both Opt and WJ move to serve the request.

In this case, WJ and Opt are initially at the same axis and the request moves orthogonal to their shared axis. As discussed in Case 6 of Section 3.3.2 when the initial state is ia, the analysis reduces to a similar case when the initial state is ib. So, these cases are omitted from the analysis. The followings are the remaining possible cases for the state transition:

1. ib to ib: Server and Opt both walk and the state remains ib. Consider a worst-case scenario in which the predominance is increased by the maximum possible value. By Lemma 3.2.4, in this case $PM^*$ stays the same; indeed, $PM^*$ is the position of $R$ which is dominated by an independent point $X$ (on the WJ’s axis) such that $X$ gets closer to the request after the request moves ($X$ is $IP_2$ and $IP_1$ in Figures 3.9(a, b), respectively). Moreover, a similar analysis to Case 1 of Section 3.3.2 proves that $P^*$ stays the same after the request moves. Given these observations, it is straightforward to prove that the difference in potential is at most 4 when the request moves away from $P^*$ and at most 2 when the request gets closer to $P^*$ and given that WJ’s server pays 1, amortized cost is at most 5, while the cost of Opt is 1. Details can be seen in Figure 3.9(a, b).

It is clear from the above analysis that when the initial state is ib and both WJ and Opt walk, the potential never decreases. So, the state of WJ cannot change from ib to ia (since the potential is the max of the two). So, in what follows, we consider a state transition from ia to ia.
Chapter 3: The Walk and Jump Algorithm

(a) Request moves away from WJ and OPT walks (R is dominated by IP*)

(b) Request moves towards WJ and OPT walks (R is dominated by IP*)

(c) Request moves away from WJ and OPT jumps (R is dominated by OPT)

(d) Request moves towards WJ and OPT jumps (R is dominated by OPT)

(e) Request moves orthogonal to OPT-axis away from P* and OPT walks (R is at server and is dominated by IP*)

(f) Request moves orthogonal to OPT-axis towards P* and OPT walks (R is at server and is dominated by IP*)

(g) Request moves orthogonal to OPT and it jumps (R is at WJ and is dominated by OPT)

Figure 3.8: Amortized cost analysis for an event when WJ’s server does not move, while OPT’s server moves to serve the request.
2. ia to ia: In this case, the request moves orthogonal to the OPT-axis and towards an independent point IP that dominates PM*. Both OPT and WJ walk to serve the request. We assume the worst case for the predominance as in Case 1, where the predominance increases by 2 and PM* remains the same point (refer to Figure 3.9(c)). The value of $\phi_{ia}$ remains the same as both WJ and OPT walk on the same axis and their distance stays the same. The amortized cost is at most 3 and OPT incurs a cost of 1.

3. ib to ii: In this case, the request moves in the orthogonal axis of the OPT-axis. WJ walks and OPT jumps to serve the request. A similar analysis to Case 4 of Section 3.3.2 shows that $P^*$ and PM* interchange their roles after the move. Given these observations, we can verify that the amortized cost is at most 3 times the cost of OPT. Details can be seen in Figure 3.9(d).

4. ib to ii: In this case, OPT walk and WJ jumps to serve the request. By Lemma 3.2.4 in the worst case, PM* is the location of $R$ and the increase in predominance is 2. A similar case analysis of Case 1 of Section 3.3.2 shows that $P^*$ stays the same after the move. It is then easy to verify that amortized cost is at most 5 while OPT incurs a cost of at least 1. The details can be seen in Figure 3.9(e).

5. ib to ii: In this case, both OPT and WJ jump to serve the request. A similar analysis to Case 4 of Section 3.3.2 indicates that the $P^*$ and PM* stay the same after serving the request. Given these observations, it is not hard to prove that the amortized cost is at most 3 times the cost of OPT. Details can be found in
Figure 3.9(f).

6. ib to ia: This case is similar to Case 5, where both OPT and W\textsubscript{j} jump to serve the request, except that the state transition is to ia instead of ii. A similar analysis to Case 4 of Section 3.3.2 indicates that the $P M^*$ and $P^*$ interchanges their roles after the request moves. The state transition to ia means that the main component of the potential is 0; hence, there is no need to check $P^*$ at time $t + 1$. Given these observations, it is not hard to prove that the amortized cost is less than 2 times the cost OPT in this case. Details can be found in Figure 3.9(g).

From the above discussion, We can conclude the following lemma.

**Lemma 3.3.4.** In an event that both W\textsubscript{j} and OPT move to serve the request, the amortized cost of W\textsubscript{j} is no more than 5 times the cost of OPT.

### 3.3.5 Summarizing the Results

Le\textsubscript{mmas} 3.3.1, 3.3.2, 3.3.3 and 3.3.4 show that the amortized cost of W\textsubscript{j} is bounded by 5 times the cost of OPT in all the possible state transitions. Subsequently, the competitive ratio of W\textsubscript{j} is at most 5. We conclude the following theorem on the basis of our case analysis:

**Theorem 3.3.1.** The competitive ratio of W\textsubscript{j} is at most 5 for the Orthogonal CNN problem.
Figure 3.9: Amortized cost analysis for an event when Wj’s and Opt’s server moves to serve the request.
Chapter 4

The CNN Problem with Advice

Competitive analysis compares the performance of online algorithms to $\text{Opt}$ on the worst-case input sequences. Moreover, $\text{Opt}$ knows the future request sequence which makes this comparison unfair and competitive results are sometimes considered too pessimistic. One way to provide a more fair comparison is to augment online algorithms with extra power against $\text{Opt}$. For example, in the paging problem, one can assume the size of online algorithms is twice the size of the cache of $\text{Opt}$ (see [31] for results with respect to resource augmentation). Another way to augment online algorithms is to provide them with some information about the input sequence via some bits of advice [32]. The implicit assumption is that the advice bits are provided by a benevolent offline oracle which knows the input and has unbounded computation power. The online algorithm knows the meaning of the advice bits and can use them to improve its competitive ratio. In most cases, a better competitive ratio is achieved if we keep on increasing the size of advice. In an extreme case, the offline oracle can encode the whole input as the advice and the problem becomes
offline. It is, however, more interesting to see how a small number of advice bits helps in improving online algorithms. In particular, an interesting question is “how many advice bits are required and sufficient to achieve an optimal algorithm” for a given problem. In this chapter, we answer this question for the CNN problem.

The advice model for online problems was first proposed in [33, 13]. Since then, many models of advice have been proposed. We refer to the survey of Boyar et al. [34] for different advice models. The model that we focus on is called advice-on-tape model and was introduced by Bockenhauer et al. [32]. Under this model, which is widely accepted as the standard advice model, the advice is stored on an advice tape of an infinite length by an oracle who has access to the input. The advice is available to the algorithm since the beginning. The advice complexity is the number of bits that the algorithm accesses from the tape and, as mentioned, the objective is to provide algorithms that use a small number of advice bits while also guaranteeing a good competitive ratio.

In the past decade, the notion of advice has been applied to various online problems such as the $k$-server, bin packing and list update problems [33, 35, 36, 37]. The CNN problem has not been studied under the advice model. In this chapter, we are interested in the most basic question about the advice complexity of the orthogonal CNN problem, namely, the size of advice that is required and sufficient to optimally serve a request. We start with the following upper bound:

**Theorem 4.0.1.** An advice of $m$ bits is sufficient to achieve an optimal solution in case of the CNN problem, where $m \leq n$; $n$ being the length of the input sequence.

**Proof.** The proof is based on a ‘lazy optimal’ algorithm which serves every request
by moving the server only horizontally or vertically. It is not hard to convert every optimal algorithm to a lazy optimal algorithm without any increase in the cost. Similarly to the proof of Lemma 3.2.1, we can delay any non-lazy move to achieve a lazy algorithm without increasing the cost. Let $\text{Opt}$ be a lazy optimal algorithm that only takes at most a single horizontal or vertical move for serving each request. Clearly, such a move can be encoded using 1 bit of advice. Let $m$ be the total number of times $\text{Opt}$ moves to serve the request where $m \leq n$ ($\text{Opt}$ will not move when request moves in its own axis). Hence, an online algorithm can follow the same moves of $\text{Opt}$ using $m$ bits of advice.

We note that the above upper bound holds for the general CNN problem, and consequently to its special cases such as orthogonal CNN problem. We complement this upper bound with a lower bound for the size of advice required to achieve an optimal solution. In what follows, we prove a lower bound that holds for the orthogonal CNN problem and consequently also holds of the general case as well.

To prove the lower bound, we use a reduction from the binary guessing problem to the CNN problem. In the binary guessing problem, a bitstring appears in an online and sequential manner. Before the content of each bit is revealed, an online algorithm should “guess” the content, and the goal is to correctly guess a maximum number of bits. The problem is not interesting in the pure online setting (the adversary can ensure that all guesses made by the online algorithm are wrong). But, provided with only one bit of advice, it is possible to guess half of the bits correctly (advice indicates whether ‘0’ or ‘1’ is more frequent and the algorithm guesses the more frequent bit for all bits). Interestingly, more than one bit of advice is not useful unless a linear
number of bits is provided. This is stated using the following lemma by Bockenhauer et.al.

**Lemma 4.0.1.** [38, 35]. For an input of length \( m \), any deterministic algorithm that wants to make \( \alpha m \) guesses correctly for \( \frac{1}{2} < \alpha < 1 \), needs to read \((1 + (1 - \alpha) \log(1 - \alpha) + \alpha \log \alpha) m \) bits of advice.

We use a reduction from the binary guessing problem to the orthogonal CNN problem and give the following lower bound.

**Theorem 4.0.2.** In order to achieve a competitive ratio of \( \beta \) for any value of \( \beta \) in the range \( 1 < \beta < \frac{5}{4} \), advice of size \((1 + (2\beta - 2) \log(2\beta - 2) + (3 - 2\beta) \log(3 - 2\beta))n/16 \) bits is required for any sequence of size \( n \).

**Proof.** The reduction enables us to guess the value of each bit in the bitstring based on the movement of a CNN algorithm on a \( 2 \times 2 \) grid. For a request that is not aligned with the position of the server, the algorithm has to move the server horizontally or vertically; such a move is associated with a binary guess. A sequence can be constructed in a way that, in the case of a wrong guess, the algorithm has to pay some extra cost to serve the requests which follow the initial request. In a sense, the input is formed by a sequence of *phases* and at the beginning of each phase, the algorithm has to make a guess about the type of the phase. A wrong guess imposes additional cost to the algorithm.

Figure [4.1] shows an instance of the CNN problem on a \( 2 \times 2 \) grid with the server initially at position 6. A request sequence of size \( n \) is provided; the sequence is formed by \( m \) phases, each having type \( A \) or \( B \). Looking at the phase types, we can state that
any algorithm $\text{ALG}$ can be modified to another algorithm $\text{ALG}'$, without increase in its cost, such that the position of server at the start of every phase is at position 6. This is because any algorithm that does not place the server at position 6 at the end of a phase has to move a distance of at least 8 to serve the last 8 requests of the phases. Meanwhile, an algorithm that places the server at position 6 has to pay at most 4 to reach at 6 to serve all these requests. In what follows, we assume the server is at location 6 at the start of every phase.

The first request is to node 1 at the start of both phases. A lazy algorithm serves requests by moving from 6 to either 8 or 3. An optimal algorithm that knows the type of the phase moves the server to 3 and 8 for phases of type $A$ and $B$ respectively and has a cost of 4 in both phases. Consequently, the cost of an optimal algorithm is at least $4m$.

An online algorithm has to make a guess of the phase type before moving the server for serving 1. Moving the server to 3 (respectively 8) means guessing the type of the phase as $A$ (respectively $B$). If the algorithm makes a correct guess, it incurs a cost of at least 4, while an incorrect guess means a cost of at least 6 for the algorithm. Note that this statement holds regardless of the type of the phase.
In summary, serving the requests in the orthogonal CNN problem translates to making guesses for the type of the phases.

According to Lemma 4.0.1, to guess the type of $\alpha m$ phases correctly, advice of size at least $(1 + (1 - \alpha) \log(1 - \alpha) + \alpha \log \alpha)m$ is required. The cost of the algorithm in the case of $\alpha m$ correct guesses is $4\alpha m + (1 - \alpha)6m = 6m - 2\alpha m$. Recall that OPT pays $4\alpha m$. So, the competitive ratio of the algorithm would be $\frac{6m - 2\alpha m}{4\alpha m}$ which is $\frac{3 - \alpha}{2}$. Any competitive ratio better than this value implies guessing more than $\alpha m$ phases correctly and hence requires an advice of size $(1 + (1 - \alpha) \log(1 - \alpha) + \alpha \log \alpha)m = (1 + (1 - \alpha) \log(1 - \alpha) + \alpha \log \alpha)n/16$ bits of advice. Replacing $\frac{3 - \alpha}{2}$ with $\beta$ completes the proof.

From Theorems 4.0.2 and 4.0.1 we conclude the following.

**Corollary 4.0.2.1.** Advice of size $\Theta(n)$ is necessary and sufficient to achieve an optimal algorithm for the CNN problem. The same statement holds for the orthogonal CNN problem.
Chapter 5

Conclusions

In this thesis, we studied new algorithms and models for the orthogonal CNN problem. Our main contribution was to present a novel algorithm, named Walk and Jump (WJ), which uses the concept of work function to serve requests in a careful manner to avoid worst-case scenarios. We used the potential function method to prove the competitive ratio of WJ is at most 5. Our potential has a few components which are defined based on the way the algorithm moves the server in different worst-case scenarios. Because of the various components in the potential, we had to apply a rather extensive case analysis to prove the upper bound of 5 for the competitive ratio of WJ. Besides this major contribution, we also considered basic settings such as unit square metric, for which we showed that the lower and upper bounds for the competitive ratio of online algorithms match at 3. We also considered the advice settings for the orthogonal CNN problem, where we showed advice of size $\Theta(n)$ is sufficient and necessary to achieve an optimal online algorithm.

As a future work, we consider improving the analysis of the WJ algorithm to make
it simpler and hopefully establish a competitive ratio better than 5 for the algorithm. Given the fact that the WJ algorithm does not have a cost more than 3 times \( \text{OPT} \) in the worst case input sequences that we considered, we conjecture that the algorithm has a competitive ratio of 3, which makes it the optimal algorithm for the orthogonal CNN problem. Another interesting question is to investigate how the WJ algorithm can be adapted to the CNN problem, and whether this adapted algorithm would have a competitive ratio better competitive ratio than 879 (which is the best existing upper bound for the CNN problem). Finally, the exact trade-off between the size of advice and the attainable competitive ratio is another topic for future research. In particular, it is interesting to investigate what is the best competitive ratio that an online algorithm can achieve with an advice of constant size.
Appendix A

Worst-case Examples

In Chapter 3 we introduced the WJ algorithm and proved that it has a competitive ratio of at most 5. While designing the algorithm and devising its potential, we have been inspired by a few “worst-case” sequence for which many alternatives, often simpler, algorithms failed. In a sense, the WJ algorithm is designed to avoid the worst-case scenarios given in these sequences. We note that the algorithm has a competitive ratio of at most 3 for all these cases, which makes us believe the upper bound for the competitive ratio of the algorithm can be improved to 3 (that would make the algorithm the optimal deterministic algorithm for the orthogonal CNN problem). In this chapter, we review how the WJ algorithm handles these adversarial sequences and how the potential function and amortized cost develop in each case. This is aimed to highlight the intuitions behind the WJ algorithm and the potential that we used in our analysis.

The chapter has four sections, each considering one adversarial scenario. In each case, a table is included that summarizes how the request moves, how the potential
changes, and how the amortized cost of the WJ algorithm compares to that of OPT for serving each request.

**Scenario 1: Opt Jumps at the Beginning**

At the beginning of Scenario 1, the request is horizontally aligned with the servers of OPT and WJ (which are co-located). Assume the request is on the right of the two servers. The sequence starts by moving the request one unit down. Knowing the future requests, OPT makes a jump of length \( d \) to serve the request; this would be the only move that OPT makes for the entire sequence. The server \( s \) of the WJ algorithm, however, has to make a few walks and jumps before reaching the OPT’s position. As we will see, the total distance moved by \( s \) is \( 3d \). Figures A.1(a-f) show the flow of the request sequence and the corresponding moves made by the OPT and \( s \). These cases are discussed in detail below. For each case, it helps to refer to the relevant subfigure in Figure A.1.

(a) Figure a shows the initial configuration as discussed above, i.e., \( s \) and OPT are initially at the same position serving the request at a distance \( d \) on the right. This is the base case, so the work function values of the server and OPT are both 0. The sequence continues by moving the request downwards.

(b) OPT makes a jump of \( d \) to serve the request. On the other hand, WJ checks if there is an independent point on the server axis that would dominate all its dangerous points after a potential walk. The point at which the server is located is one (the only) such independent point that dominates its only dangerous point DP. As such, WJ walks to serve the request.
Appendix A: Worst-case Examples

Figure A.1: A summary of Scenario 1. 

(c) The adversary makes the WJ algorithm to walk as far as possible from Opt. This is done by repeatedly moving the request downwards until the dangerous point DP is dominated by the location of both Opt and WJ’s server. Before this “transition point”, the DP is always dominated by the location of WJ and hence the algorithm keeps walking to serve the request.

(d) One further move down by the requests results in the dangerous point DP not being dominated by the independent point on the server’s axis (the location of the server) any more. The independent point on the server axis has work function value $d + 1$, which cannot dominates the DP. So, WJ’s server jumps to the nearest point on the other axis.

(e) The sequence continues by moving the request towards Opt. Since the WJ server is vertically aligned with the request, it does not move. The request
moves upwards until it reaches OPT’s position.

(f) After reaching OPT, the request moves one unit to the right. There is no independent point on the server axis after serving the request, so, server jumps to the OPT’s position.

The total cost of the algorithm is $3d$ and the cost of OPT is $d$. Table A.1 summarizes how the potential changes in the course of serving this sequence. In all cases, the amortized cost of the algorithm would be no more than 3 times the cost of OPT.

<table>
<thead>
<tr>
<th>StateChange</th>
<th>$\phi_t$(state)</th>
<th>$\phi_{t+1}$(state)</th>
<th>Cost(ALg)</th>
<th>$\Delta \phi$</th>
<th>ac</th>
<th>Cost(Opt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a to b)</td>
<td>0 ($\phi_{ia}$)</td>
<td>3d-1($\phi_{ii}$)</td>
<td>1</td>
<td>3d-1</td>
<td>3d</td>
<td>d</td>
</tr>
<tr>
<td>(b to c)</td>
<td>3d-1($\phi_{ii}$)</td>
<td>2d ($\phi_{ii}$)</td>
<td>d-1</td>
<td>1-d</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(c to d)</td>
<td>2d($\phi_{ii}$)</td>
<td>d ($\phi_{ia}$)</td>
<td>d</td>
<td>-d</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(d to e)</td>
<td>d($\phi_{ia}$)</td>
<td>d ($\phi_{ia}$)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(e to f)</td>
<td>d($\phi_{ia}$)</td>
<td>0 ($\phi_{ia}$)</td>
<td>d</td>
<td>-d</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table A.1: A summary of Scenario 1. The first column indicates how the states change in accordance to subfigures of Figure A.1. Other columns provide a summary of the change in potential and the amortized cost in comparison to the cost of OPT.

**Scenario 2: Opt Walks Throughout**

Scenario 2 has the same initial configuration as Scenario 1, namely, the request is located at a distance $d$ on the right of servers OPT and WJ (which are co-located). The sequence starts with $d+1$ requests each moving the request one unit downwards. As we will see, OPT keeps walking to serve the request at a cost of $d+1$. On the other hand, WJ jumps towards the request after initially walking with it for a distance of $d$. This jump makes it further from OPT and in fact is a “mistake”. The request continues towards OPT, which imposes a few more moves for WJ before it reaches...
Appendix A: Worst-case Examples

Figure A.2: A summary of Scenario 2. Wj and Opt both walk for a distance of $d$ to server the first $d$ request. The next requests results in Wj jumping a distance of $d$ (a mistake) while Opt keeps walking. The request continues by moving towards Opt.

Figures A.2(a-h) show the flow of the request sequence and the corresponding moves made by the Opt and s. These cases are discussed in detail below. For each case, it helps to refer to the relevant subfigure in Figure A.2.

(a) The initial configuration is similar to Scenario 1 with both Opt and Wj’s server $s$ at the same position and the request at a distance $d$ from them.

(b) Opt walks with the server axis. Meanwhile, Wj checks if, after a potential walk, any independent point of the server axis dominates the dangerous point. There
is one independent point (the position of the server) which has work function 2 and dominates its only dangerous point DP. So, Wj servers the request by walking.

(c) The request keeps moving downwards for another $d$ requests. Opt serves it by walking. Wj also walks to serve request as long as the independent point on the server axis (the position of the server) dominates DP. It is the case for all requests except the very last one.

(d) On the last move of the request downwards, the independent point on the server axis, which has a work function value $d + 1$, no longer dominates the DP $(d + 1 + d + 1 = 2d + 2)$; $DP = d + d = 2d$). So, Wj’s server jumps to the nearest point on the other axis. Recall that Opt walks to serve this request. As such, this request involves moving Wj away from Opt, a mistake by the algorithm.

(e) The request moves one unit to the right. Opt is aligned with the request and does not move. Wj is not aligned, however, and should make another move. After a potential walk, the independent point of the server’s axis dominates the dangerous point $(d + 1 + d + 1 = 2d + 2)$. So, Wj’s server walks (serves the request by moving right), which makes it further away from Opt.

(f) The request moves another unit to the right. Again, Opt does not move. This time, the independent point of the server’s axis does not dominates the dangerous point after a potential walk $(d + 2 + d + 2 = 2d + 4)$; $DP = d + 1 + d + 1 = 2d + 2)$. So, Wj serves the request by a jump of length 1. At this point, Opt and Wj are aligned with the request on the same line.
(g) The request keeps moving left until it gets to Opt’s position. WJ’s server does not move as it is aligned with the request.

(h) The request moves one unit down. Again Opt does not move. For WJ, after a potential walk, there will be no independent point on the server axis. Consequently, WJ serves the request by jumping to the Opt’s position.

The total cost of the algorithm is $3d + 3$ and the cost of Opt is $d + 1$. Table [A.2] summarizes how the potential changes in the course of serving this sequence. In all cases, the amortized cost of the algorithm would be no more than 3 times the cost of Opt.

<table>
<thead>
<tr>
<th>StateChange</th>
<th>$\phi_t$(state)</th>
<th>$\phi_{t+1}$ (state)</th>
<th>Cost(ALg)</th>
<th>$\Delta \phi$</th>
<th>ac</th>
<th>Cost (Opt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a to b)</td>
<td>0 ($\phi_{ia}$)</td>
<td>2 ($\phi_{ib}$)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(b to c)</td>
<td>2 ($\phi_{ib}$)</td>
<td>2d ($\phi_{ib}$)</td>
<td>d-1</td>
<td>2d-2</td>
<td>3(d-1)</td>
<td>d-1</td>
</tr>
<tr>
<td>(c to d)</td>
<td>2d($\phi_{ib}$)</td>
<td>d+3 ($\phi_{ii}$)</td>
<td>d</td>
<td>3-d</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(d to e)</td>
<td>d+3 ($\phi_{ii}$)</td>
<td>d+2 ($\phi_{ii}$)</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(e to f)</td>
<td>d+2 ($\phi_{ii}$)</td>
<td>d+1 ($\phi_{ia}$)</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(f to g)</td>
<td>d+1 ($\phi_{ia}$)</td>
<td>d+1 ($\phi_{ia}$)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(g to h)</td>
<td>d+1 ($\phi_{ia}$)</td>
<td>0 ($\phi_{ia}$)</td>
<td>d-1</td>
<td>-1-d</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table A.2: A summary of Scenario 2. The first column indicates how the states change in accordance to subfigures of Figure [A.2]. Other columns provide a summary of the change in potential and the amortized cost in comparison to the cost of Opt.

### Scenario 3: The Request Vibrates

The third scenario also starts with the request being at distance $d$ of Opt and WJ servers (which are initially co located). This time, however, the request does not go all the way down; instead, it “vibrates” around its initial position by repeatedly
Appendix A: Worst-case Examples

Figure A.3: A summary of Scenario 3. OPT moves a distance of \(d\) with a single move at the beginning, while WJ makes 2\(d\) walks before a jump of length \(d\); this gives WJ a total cost of 3\(d\).

moving down and up. Similar to Scenario 1, OPT makes a jump of length \(d\) to serve all requests at no subsequent cost. On the other hand, WJ makes 2\(d\) walks before jumping (at a cost of \(d\)) to the request position. The cost of WJ would be 3\(d\) in comparison to the cost \(d\) of OPT.

Figures A.3(a-e) show the flow of the request sequence and the corresponding moves made by the OPT and s. These cases are discussed in detail below. For each case, it helps to refer to the relevant subfigure in Figure A.3.

(a) The initial configuration is similar to Scenarios 1 and 2, where both OPT and server s on the same position and request at a distance \(d\).

(b) The request moves one unit down. OPT makes a jump of length \(d\) to serve the new request. On the other hand, WJ checks if the independent point of the server axis is dominating its dangerous point (in the case of a potential walk).
The independent point is the position of the server itself with work function 2 and it dominates its dangerous point DP. so, Wj servers the request by walking its server.

(c) The request moves back to its original position. The independent point on the server axis is itself the dangerous point, so Wj walks to serve the request. At this point, Wj’s server and request are at their original positions (but the work-functions and consequently the potential have changed). Repeating the above, the adversary makes Wj to keep serving request by walking in parallel to the vibrating request. This continues for the first $2d - 2$ requests.

(d) After $2d - 2$ requests, the dangerous point is dominated by two independent points. This is a “transition state”. A subsequent move of the request results in Wj jumping towards the request.

(e) The request moves upwards. The dangerous point is no longer dominated by the independent point on the server axis ($2d + 1 + 1 = 2d + 2$; $DP = d + d = 2d$). So, Wj jumps towards the request. At this point, Wj and OPT servers are collocated.

The total cost of the algorithm is $3d$ and the cost of OPT is $d$. Table A.3 summarizes how the potential changes in the course of serving this sequence. In all cases, the amortized cost of the algorithm would be no more than 3 times the cost of OPT.
Appendix A: Worst-case Examples

<table>
<thead>
<tr>
<th>State Change</th>
<th>$\phi_t$ (state)</th>
<th>$\phi_{t+1}$ (state)</th>
<th>Cost (ALg)</th>
<th>$\Delta \phi$</th>
<th>ac</th>
<th>Cost (OPT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a to b)</td>
<td>0 ($\phi_{ia}$)</td>
<td>3d-1($\phi_{ii}$)</td>
<td>1</td>
<td>3d-1</td>
<td>3d</td>
<td>d</td>
</tr>
<tr>
<td>(b to c)</td>
<td>3d-1($\phi_{ii}$)</td>
<td>3d-2 ($\phi_{ia}$)</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(c to d)</td>
<td>3d-2 ($\phi_{ia}$)</td>
<td>d ($\phi_{ia}$)</td>
<td>2d-2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(d to e)</td>
<td>d($\phi_{ia}$)</td>
<td>0 ($\phi_{ia}$)</td>
<td>-d</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table A.3: A summary of Scenario 3. The first column indicates how the states change in accordance to subfigures of Figure A.3. Other columns provide a summary of the change in potential and the amortized cost in comparison to the cost of OPT.

Scenario 4: Multiple Independent Points

Scenario 4 illustrates a situation where there are multiple independent points on the server’s axis. Unlike the previous three scenarios, where the cost of $W_j$ was equal to 3 times $OPT$, in Scenario 4, the cost of $W_j$ is strictly less than 3 times $OPT$. Figures A.4(a-h) show the flow of the request sequence and the corresponding moves made by the $OPT$ and $s$. These cases are discussed in detail below. For each case, it helps to refer to the relevant subfigure in Figure A.4.

(a) As in the other scenarios, at the beginning, server $W_j$ and $OPT$ are initially at the same position while the request is at a distance $d$ on their right. The work function of the point where $W_j$ and $OPT$ are located is 0.

(b) The request moves downwards. $OPT$ makes a jump of $d$ to serve the request. $W_j$ checks if the independent point of the server axis (after a potential walk) is dominating its dangerous point. The independent point is the position of the server itself with work function 2 and it dominates its dangerous point DP. So, $W_j$ walks.

(c) Request goes one further unit down. Similarly to step (b), $W_j$ serves the request
Figure A.4: A summary of Scenario 4, where there are multiple independent points associated on the server axis. $\text{Opt}$ moves a distance of $d$ at the beginning and then an extra cost of 2 for two walks at steps (d) and (e). On the other hand, $W_j$ walks a total distance of $3d + 4$, which is less than 3 times the cost of $\text{Opt}$.

by walking its server.

(d) The request moves one unit right on the server’s axis. $W_j$ does not move but $\text{Opt}$ goes one unit right for serving the request.

(e) The new request one unit left, back to its previous position. $\text{Opt}$ again walks and also gets back to its previous position. As before, $W_j$ does not move.
(f) The request moves down. $\text{OPT}$ does not need to move but $W_j$ should move its server. The independent point on the server’s axis is the position of the server itself. There are two dangerous points to check for this independent point. The first one is associated with the independent point at which $\text{OPT}$ is located, and the other is associated with another independent point that is created as a result of the horizontal moves of the request. Since both dangerous points are dominated by the independent point (the position of the server), $W_j$ walks to serve the request.

The sequence continues by moving the request further down. $W_j$ walks to serve the requests as long as both dangerous points are dominated by the independent points that are associated with.

This happens until server $s$ is at a distance $d + 1$ from the dangerous point of $\text{OPT}$ ($DP_{\text{server}} = d + 1 + d + 1 = 2d + 2$; $DP_{\text{Opt}} = d + 2 + d = 2d + 2$). The request moves further down.

(g) When the request gets to distance $d+1$ from the dangerous point associated with the location of $\text{OPT}$, the independent point does not dominate all its dangerous point any more; as such $W_j$ jumps towards the request to serve it.

(h) The request starts moving up towards $\text{OPT}$. Up until the request reaches the location of $\text{OPT}$, the server of $W_j$ does not move.

(i) The new request moves right on the $\text{OPT}$’s axis. The algorithm checks if there is an independent point on the server axis dominating the dangerous point (after a potential walk). The independent point that was created because of
the horizontal move at steps (d-e) has work function \( d + 3 \) and dominates its
dangerous point. So, WJ walks to sever the request.

(j) The request moves one unit further right. The independent point that prevented
WJ from jump in the previous step no longer dominates its dangerous point.
Consequently, WJ jumps to the Opt’s axis to serve the request.

(h) The request moves left, back to Opt’s position. Neither Opt nor WJ do not
need to move.

(i) The request moves upwards. There is no point dominating the dangerous point
after a potential walk. So WJ jumps to the location of Opt to serve the request.

The total cost of the algorithm is \( (d+1) + d + (1 + d + 1 + 1) = 3d + 4 \) and the cost
of Opt is \( d + 2 \). The competitive ratio is less than 3. Table A.4 summarizes how the
potential changes in the course of serving this sequence. In all cases, the amortized
cost of the algorithm would be less than 3 times the cost of Opt.
### Table A.4: A summary of Scenario 4

The first column indicates how the states change in accordance to subfigures of Figure A.4. Other columns provide a summary of the change in potential and the amortized cost in comparison to the cost of Opt.

<table>
<thead>
<tr>
<th>State Change</th>
<th>$\phi_t$ (state)</th>
<th>$\phi_{t+1}$ (state)</th>
<th>Cost (ALg)</th>
<th>$\Delta \phi$</th>
<th>ac</th>
<th>Cost (Opt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a to b)</td>
<td>0 ($\phi_{ia}$)</td>
<td>3d-1 ($\phi_{ii}$)</td>
<td>1</td>
<td>3d-1</td>
<td>3d</td>
<td>d</td>
</tr>
<tr>
<td>(b to c)</td>
<td>3d-1 ($\phi_{ii}$)</td>
<td>3d-2 ($\phi_{ii}$)</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(c to d)</td>
<td>3d-2 ($\phi_{ii}$)</td>
<td>3d+1 ($\phi_{ii}$)</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(d to e)</td>
<td>3d+1 ($\phi_{ii}$)</td>
<td>3d+2 ($\phi_{ii}$)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(e to f)</td>
<td>3d+2 ($\phi_{ii}$)</td>
<td>2d+3 ($\phi_{ii}$)</td>
<td>d-1</td>
<td>1-d</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(f to g)</td>
<td>2d+3 ($\phi_{ii}$)</td>
<td>d+3 ($\phi_{ia}$)</td>
<td>d</td>
<td>-d</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(g to h)</td>
<td>d+3 ($\phi_{ia}$)</td>
<td>d+3 ($\phi_{ia}$)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(h to i)</td>
<td>d+3 ($\phi_{ia}$)</td>
<td>d+2 ($\phi_{ii}$)</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(i to j)</td>
<td>d+2 ($\phi_{ii}$)</td>
<td>1 ($\phi_{ia}$)</td>
<td>d+1</td>
<td>-1-d</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(j to k)</td>
<td>1 ($\phi_{ia}$)</td>
<td>1 ($\phi_{ia}$)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(k to l)</td>
<td>1 ($\phi_{ia}$)</td>
<td>0 ($\phi_{ia}$)</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Bibliography


[38] H.-J. Böckenhauer, J. Hromkovič, D. Komm, S. Krug, J. Smula, and A. Sprock,