UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ONLINE LEARNING OF COMBINATORIAL OBJECTS

A thesis submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Holakou Rahmanian

September 2018

The Dissertation of Holakou Rahmanian
is approved:

Professor Manfred K. Warmuth, Chair

Professor S.V.N. Vishwanathan

Professor David P. Helmbold

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Contents

List of Figures

List of Tables

Abstract

Online Learning of Combinatorial Objects

by

Holakou Rahmanian

This thesis develops algorithms for learning combinatorial objects. A combinatorial object is a structured concept composed of components. Examples are permutations, Huffman trees, binary search trees and paths in a directed graph. Learning combinatorial objects is a challenging problem: First, the number of combinatorial objects is typically exponential in terms of number of components. Second, the convex hull of these objects is a polytope whose characterization in the original space may have exponentially many facets or a description of the polytope in terms of facets/inequalities may not be even known. Finally, the loss of each object could be a complicated function of its component and may not be simply additive as a function of the components. In this thesis, we explore a wide variety of combinatorial objects and address the challenges above. For each combinatorial object, we go beyond the original space of the problem and introduce auxiliary spaces and representations. The representation of the objects in these auxiliary spaces admits additive losses and polytopes with polynomially many facets. This allows us to extend well-known algorithms like Expanded Hedge and Component Hedge to these combinatorial objects for the first time.

To BIHE students,

who are deprived of higher education in their home country – Iran,

nevertheless, always thirsty for knowledge,

and hungry for learning.

Acknowledgments

I wish to express my sincere gratitude to those who have contributed to this Ph.D dissertation and supported me throughout this challenging journey.

First and foremost, I would like to profoundly thank my advisors, Professors Manfred Warmuth and S.V.N. Vishwanathan (Vishy). Manfred has contributed in a lot of ways to this dissertation. He is a source of endless number of ideas, unlimited enthusiasm, and at the same time, admirable humility. Manfred's sincere passion for science, great enthusiasm in brainstorming, and active presence in writing and presentation taught me lots of lessons and helped significantly to this thesis. He graciously hosted me as a guest to his beautiful home at different times and showed a great hospitality at his parties and dinners.

This thesis would have been impossible without Vishy's guidance and mentorship. He showed a lot of faith in my abilities and gave me the space and self-confidence to walk in the path of science and discoveries. His trust, support and encouragement helped me substantially in this journey especially at difficult times when the research hit what seemed to be a "dead end". His limitless hunger for research and never-ending energy for exploration is a great source of inspiration.

I would also like to deeply thank the reader of my thesis and my co-author, Professor David Helmbold. His careful and meticulous eye on important details, together with his nice and welcoming personality and his infinite patience always assisted this dissertation throughout the years.

In addition to UCSC, I feel extremely privileged to have the chance to collaborate with great researchers at NYU and Google Research NYC. I would like to thank my co-authors Professor Mehryar Mohri, Corinna Cortes, and Vitaly Kuznetsov. This thesis has benefited extremely from their precious technical ideas and insights. I would also like to thank Professor Michael Collins for his helpful comments and valuable discussions.

The fellow machine learning students at Santa Cruz have been not only good friends but also supportive colleagues. I would like to thank Michal Derezinski, Parameswaran Raman, Sriram Srinivasan, Ehsan Amid, Andrei Ignat, Kuan-Sung Huang, Dhanya Sridhar, Sabina Tomkins, Jay Pujara, Jiazhong Nie, and Tommy Schmitz. The helpful feedback and comments that they shared with me helped me with this thesis.

Last but definitely not least, I would like to express my sincere gratitude to my family for their unconditional love and support throughout my entire life without whom none of my accomplishments would be possible. My parents, Changiz Rahmanian and Goli Toghyani, have always encouraged me for pursuing higher education. My sisters, Ima and Anahita, and my brother, Houtan, are the best supporters anyone could ask for.

Chapter 1

Introduction

This thesis shows some significant novel results in *online learning* which is a subfield of machine learning (ML). Online learning has the following characteristics. The learning is *online*: Data points are presented in a sequential fashion. The learning is *supervised*: The algorithm receives feedback on the quality of its prediction. The data is *adversarial*: There is no assumption regarding the underlying distribution which generates the data points [Cesa-Bianchi and Lugosi, 2006, Littlestone and Warmuth, 1994].

In this thesis, we consider online learning of *combinatorial objects*. Informally, a combinatorial object is a structured concept composed of *components*. Examples are k-sets, permutations, paths in graph. The main challenge in these problems is to deal with large number of combinatorial objects typically exponential in terms of the number of components.

Two well-known algorithms proposed for these problems in previous papers

are *Expanded Hedge (EH)* [Takimoto and Warmuth, 2003] and *Component Hedge (CH)* [Koolen et al., 2010]. These algorithms, however, cannot be efficiently applied to all combinatorial objects because they require certain conditions which are described later in this chapter. In this thesis, we explore several classes of such combinatorial objects. For each class, we introduce auxiliary spaces and representations which allow us to extend well-known online learning algorithms like EH and CH to a significantly wider class of combinatorial objects than was possible before.

We start in this chapter with an introduction to online learning as well as an overview of the problems discussed in the thesis. This chapter itself is organized as follows: Section 1.1 explains basic concepts of online learning, such as the interactive learning protocol, the adversarial data setting and worst case regret bounds. The online learning setting and its challenges are further discussed in Section 1.2 for learning combinatorial objects. In Section 1.3, we discuss the main algorithms for online learning of combinatorial objects currently in the literature. Finally, in Section 1.4, we give an overview of the main chapters of the thesis (Chapters 2, 3 and 4).

1.1 The Basics of Online Learning

Online learning is a rich and vibrant area, see Vovk [1990], Littlestone and Warmuth [1994], Cesa-Bianchi et al. [1996, 1997] for some early papers, and Cesa-Bianchi and Lugosi [2006] for a textbook treatment. The online learning setting is a game between the *learner* and the *adversary* on a set of *experts* over a series of trials.

This game is illustrated in Prediction Game 1 and summarized in the following: In each trial, the learner makes a prediction with an expert, observes the losses of all the experts, and finally, incurs the loss of its prediction. The algorithm can then update its internal representation based on this feedback and the process moves on to the next trial.

Prediction Game 1 Prediction game for Experts $\mathcal{E} = \{E_1, \ldots, E_N\}$.

1: Given a set of N experts $\mathcal{E} = \{E_1, \ldots, E_N\}$

2: For each trial $t = 1, \ldots, T$

3: The **learner** randomly predicts with the i_tth expert $i_t \in [N]$.

4: The **adversary** reveals the loss of each expert as a loss vector $\boldsymbol{\ell}_t \in [0, 1]^N$.

5: The **learner** incurs a (expected) loss $\mathbb{E}[\ell_{t,i_t}]$.

Unlike batch learning settings, there is no assumed distribution from which losses are randomly drawn. Instead the losses are drawn adversarially. In general, an adversary can force arbitrarily large loss on the algorithm. So instead of measuring the algorithm's performance by the total loss incurred, the algorithm is measured by its *regret*, the amount of loss the algorithm incurs above that of the single best expert in the set of experts. Therefore the regret of the algorithm can be viewed as the cost of not knowing the best expert ahead of time:

$$\text{Regret} = \sum_{t=1}^{T} \mathbb{E}[\ell_{t,i_t}] - \underbrace{\min_{i \in [N]} \sum_{t=1}^{T} \ell_{t,i}}_{L^*}$$

One way to create algorithms for these problems is to use one of the well-known so-called "experts algorithms" like Randomized Weighted Majority [Littlestone

and Warmuth, 1994] or Hedge [Freund and Schapire, 1997]. In these algorithms, the learner maintains a weight vector $\boldsymbol{w}_t \in \mathbb{R}^N_{\geq 0}$ on the simplex throughout the trials, i.e. $\sum_{i=1}^N w_{t,i} = 1$ for all $t \in \{1 \dots T\}$. This weight vector is initialized to the uniform distribution $\boldsymbol{w}_1 = [\frac{1}{N} \dots \frac{1}{N}]$. The expected loss of the learner at the tth trial is $\mathbb{E}[\ell_{t,i_t}] = \boldsymbol{w}_t \cdot \boldsymbol{\ell}_t$. After observing the losses in each trial t, the learner updates its weight vector multiplicatively by the exponentiated factors: $\widehat{w}_{t+1,i} = w_{t,i} \exp(-\eta \ell_{t,i})$ where $\eta > 0$ is the learning rate. Finally, the learner normalizes the $\widehat{\boldsymbol{w}}_{t+1}$ weights resulting in the weight vector $\boldsymbol{w}_{t+1}$ for the next trial. With proper tuning of the learning rate η, the regret of these algorithms is logarithmic in the number of experts.

Theorem 1 (Littlestone and Warmuth [1994], Freund and Schapire [1997])**.** *With proper tuning of the learning rate η, the Hedge and Randomized Weighted Majority algorithms achieve the regret bound*

$$\sum_{t=1}^{T} \boldsymbol{w}_t \cdot \boldsymbol{\ell}_t - \min_{i \in [N]} \sum_{t=1}^{T} \ell_{t,i} \leq \sqrt{2\, L^*\, \log N} + \log N,$$

where L^* is the cumulative loss of the best expert in hindsight.

1.2 Learning Combinatorial Objects

In Prediction Game 1, we consider the scenario where the experts are *combinatorial objects*. A combinatorial object is represented as a n-tuple of non-negative integers where each integer is associated with a *component* of the object. The *combinatorial class* is the finite set of all combinatorial objects denoted by[1] $\mathcal{H} \subset \mathbb{N}^n$. Table 1.1

[1] $\mathbb{N}$ denotes the set of non-negative integers.

Combinatorial Object	Component	Example
Permutation	position assignment	$[4, 2, 3, 1, 5]$ with $n = 5$
k-Set	presence of the element	$[0, 1, 1, 0, 0]$ with $n=5$ and $k=2$
Binary Search Tree	depth of the node	$[2, 3, 4, 1, 2]$ with $n = 5$ (see Figure 1.1(a))
Huffman Tree	depth of the leaf	$[2, 3, 3, 2, 2]$ with $n = 5$ (see Figure 1.1(b))
Paths	presence of the edge	$[0, 1, 0, 0, 1, 0, 1]$ with $n = 7$ (see Figure 1.1(c))

Table 1.1: Examples of combinatorial objects with their components.

and Figure 1.1 provide a few examples of combinatorial objects and their components. Figure 1.1(a) illustrates a binary search tree containing the keys $k_1 < \ldots < k_5$ located at depths $2, 3, 4, 1$ and 2, respectively. In Figure 1.1(b) a Huffman tree is shown which encodes the symbols $s_1, \ldots, s_5$ which are located at leaves in depths[2] $2, 3, 3, 2$ and 2, respectively. Figure 1.1(c) shows a path from the source s to the sink t (illustrated in blue) in a directed graph with edges as components. Since the edges e_2, e_5 and e_7 are present in the path, then their associated bits are 1s and the bit-vector representation of the path is $[0, 1, 0, 0, 1, 0, 1]$.

When learning combinatorial objects online, the adversary reveals a piece of information about every component in each trial. Each component contributes to the loss of the combinatorial object and this loss can be easily computed. In several cases, the adversary reveals a *loss vector* which contains the loss of each component and the loss of the objects is linear in terms of the components. For example, in learning k-sets, the adversary reveals the loss of each element. Then the loss of the k-set is the sum over

[2] Unlike binary search trees, the depths in Huffman trees start from zero at the root. This is because each edge represent a bit in the encoding of the symbols and the depth of each symbol is its code length in the encoding.

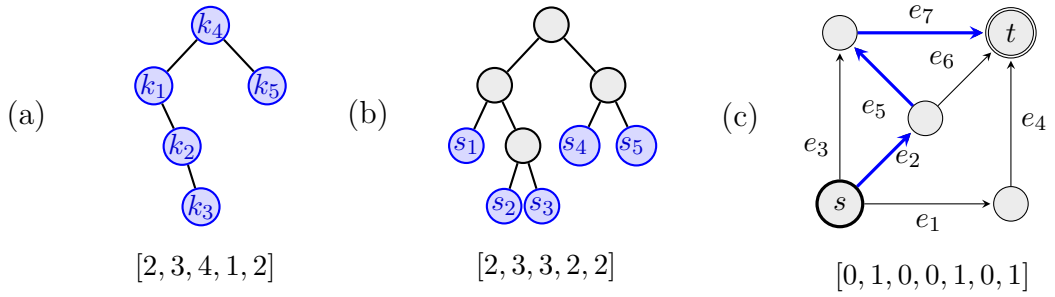


Figure 1.1: Examples of (a) a binary search tree, (b) a Huffman tree, and (c) a path in a directed graph.

the losses of the k elements in the subset [Warmuth and Kuzmin, 2008]. As another example, in online shortest path learning, the adversary reveals the cost of each edge and the loss of each path is typically the sum of the costs of the edges along that path [Takimoto and Warmuth, 2003].

1.2.1 Challenges in Learning Combinatorial Objects

Learning combinatorial objects is typically a challenging problem. The combinatorial nature of this objects and the structure of their losses may bring certain challenges:

- *Exponentially Many Experts.* The number of experts N is typically huge in terms of number of components n. Examples are $N = n!$ for permutations, $N = \binom{n}{k}$ for k-sets, and $N = C_n$ for binary search trees where $C_n = \frac{1}{n+1}\binom{2n}{n}$ is the nth Catalan number. Thus a naive attempt to implement algorithms like Hedge (i.e. maintaining one weight per expert) results in an inefficient algorithm. It is simply

impractical to keep track of a distribution over the combinatorial objects via a weight vector of exponential size.

- *Ill-behaved Polytopes.* Even maintaining a mean vector of a distribution over all combinatorial objects could be challenging. Mean vectors live in the convex hull $\mathcal{F}$ of the combinatorial objects. The description of the polytope $\mathcal{F}$ in its natural space may have exponentially many facets or a characterization of the facets in terms of inequalities may not be even known.

- *Non-Additive Losses.* The loss or the gradient thereof appears in the exponent of the multiplicative update factors for each combinatorial object. Additivity of the loss in terms of the components turns the update factors into products over the components which makes the updates mathematically convenient. The loss of the combinatorial object, however, may not be additive in terms of the components. Several modern machine learning applications like machine translation, automatic speech recognition, optical character recognition and computer vision can be represented as a learning a minimum loss path in a directed graph where the loss of each path is not additive in the edges along the path.

In this thesis, we explore a wide variety of classes of combinatorial objects and address the challenges above. In each of these classes of combinatorial object, we provide an efficient solution. The common theme of our solutions is to go beyond the original space of the problem and use an auxiliary space. In that auxiliary space, the loss will be additive in terms of the components of the auxiliary space and the convex

hull of the objects is a polytope with a polynomial number of facets. This will allow us to deal with exponentially many combinatorial experts.

Main Idea of Thesis

The main idea of this thesis is to use auxiliary spaces and representations to deal with the challenges in online learning of combinatorial objects.

1.3 Background

We give a brief overview of the main algorithms for online learning of combinatorial objects with additive losses currently in the literature.

1.3.1 Expanded Hedge (EH)

When Hedge is applied to combinatorial objects, we call it *Expanded Hedge (EH)* because it is applied to a combinatorially "expanded domain". In this setting, there is one expert per object. In some combinatorial objects, an efficient implementation of EH can be achieved by exploiting the structure of the experts and losses.

Consider the additive path learning problem in a directed graph with one designated source node and one designated sink node. The experts are the set of paths from the source to the sink. The loss of each path is the sum of the losses of the edges along that path. Takimoto and Warmuth [2003] introduced an efficient implementation of EH by exploiting the additivity of the loss over the edges of a path. Viewing each path as an expert, the weight w_π of a path π is proportional to $\prod_{e\in\pi} \exp(-\eta L_e)$, where

L_e is the cumulative loss of edge e. The algorithm maintains one weight w_e per edge $e \in E$. These weights are in *stochastic form*, that is, the total weight of all edges leaving any non-sink node sums up to 1. The weight of each path is in *product form* $w_\pi = \prod_{e \in \pi} w_e$ and sampling a path is easy. At the end of the current trial, each edge e receives additional loss ℓ_e, and path weights are updated. The multiplicative updates with exponentiated loss for the paths decomposes over the edges due to additivity of the loss over the edges. Thus the updated path weights will be

$$w_\pi^{\text{new}} = \frac{1}{Z} w_\pi \exp(-\eta \sum_{e \in \pi} \ell_e) = \frac{1}{Z} \prod_{e \in \pi} w_e \exp(-\eta \ell_e),$$

where Z is a normalization. Now a certain efficient procedure called *weight pushing* [Mohri, 2009b] is applied. It finds new edge weights w_e^{new} which are again in stochastic product form, i.e. the total outflow out of each node is one and the updated weights are $w_\pi^{\text{new}} = \prod_{e \in \pi} w_e^{\text{new}}$, facilitating sampling.

1.3.2 Follow The Perturbed Leader (FPL)

The *Follow the Perturbed Leader (FPL)* [Kalai and Vempala, 2005] is another algorithm for learning combinatorial objects with additive losses. FPL adds random perturbations to the cumulative loss of each component in each trial. Then it predicts with the combinatorial object that has the minimum perturbed loss.

1.3.3 Component Hedge (CH)

The *Component Hedge (CH)* algorithm [Koolen et al., 2010] is the main generic approach for learning combinatorial objects with additive losses. Each object is repre-

sented as a bit vector over the set of components where the 1-bits indicate the components appearing in the object. The algorithm maintains a mean vector $\boldsymbol{f}$ representing a mixture over all objects. The weight space of CH is thus the convex hull of the weight vectors representing the objects. This convex hull is a polytope $\mathcal{F}$ of dimension n with the objects as corners.

In each trial, the weight of each component (i.e. coordinate) f_i of $\boldsymbol{f}$ is updated multiplicatively by its associated exponentiated loss: $f_i \leftarrow f_i\, e^{-\eta\, \ell_i}$. Then the weight vector $\boldsymbol{f}$ is projected back to the polytope $\mathcal{F}$ via relative entropy projection. $\mathcal{F}$ is often characterized with a set of equality constraints (i.e. it is an intersection of affine subspaces). Iterative Bregman projection [Bregman, 1967] is often used; it enforces each constraint in turn. Although this can violate previously satisfied constraints, repeatedly cycling through them is guaranteed to converge to the proper projection if all the facets of the polytope are equality constraints. For the efficiency of CH it is required that the polytope $\mathcal{F}$ has a small number of facets (polynomial in n).

The CH algorithm predicts with a random corner of the polytope whose expectation equals the maintained mean vector in the polytope. The prediction step is usually done by first decomposing the weight vector into a small convex combination of combinatorial objects, and then randomly sampling from the convex combination. The decomposition step is typically done using a greedy approach: At each iteration, one combinatorial object is chosen in a greedy fashion such that removing that object from the weight vector zeros out one component of the remaining weight vector.

1.4 Overview of Chapters

We now sketch the contents of the main chapters of this dissertation. Chapter 2 and Chapter 3 have been published at ALT 2018 and NIPS 2017 conferences [Rahmanian et al., 2018, Rahmanian and Warmuth, 2017], respectively. Chapter 4 is based on the arXiv preprint [Cortes et al., 2018].

1.4.1 Overview of Chapter 2: Online Learning with Extended Formulations

The standard techniques for online learning of combinatorial objects perform multiplicative updates followed by projections into the convex hull of all the objects. However, this methodology can be expensive if the convex hull contains many facets. For example, the convex hull of n-symbol Huffman trees is known to have exponentially many facets [Maurras et al., 2010]. We get around this difficulty by exploiting extended formulations [Kaibel, 2011], which encode the polytope of combinatorial objects in a higher dimensional "extended" space with only polynomially many facets. We develop a general framework for converting extended formulations into efficient online algorithms with good relative loss bounds. We present applications of our framework to online learning of Huffman trees and permutations. The regret bounds of the resulting algorithms are within a factor of $\mathcal{O}(\sqrt{\log(n)})$ of the state-of-the-art specialized algorithms for permutations, and depending on the domain of the loss vectors, improve on or match the state-of-the-art for Huffman trees. Our method is general and can be

applied to other combinatorial objects. Furthermore, we believe this technique provides a promising approach for the bandit setting as well as problems with more complex losses.

1.4.2 Overview of Chapter 3: Online Dynamic Programming

We consider a special method of constructing extended formulations for online learning from a dynamic programming algorithm. We propose a general method for learning combinatorial objects whose offline optimization problem can be solved efficiently via a dynamic programming algorithm with arbitrary min-sum recurrence relations. Examples include Binary Search Trees, Matrix-Chain Multiplication, k-sets, Knapsack, Rod Cutting, and Weighted Interval Scheduling.

Using the underlying graph of subproblems (called *multi-DAG*) induced by the dynamic programming algorithm for these problems, we define a representation of the combinatorial objects by encoding them as a specific type of subgraph called *multipaths*. These multipaths encode each object as a series of successive decisions (i.e. the components) over which the loss is linear, even though the loss may not be linear in the original representation (e.g. for Matrix-Chain Multiplication). Then we prove that minimizing a specific dynamic programming problem from this class over trials reduces to online learning of multipaths in the induced multi-DAG. We show that the multi-DAGs and multipaths are natural generalization of DAGs and paths. Also the associated polytope has a polynomial number of facets in this representation. These properties allow us to generalize the existing EH [Takimoto and Warmuth, 2003]

and CH [Koolen et al., 2010] algorithms from online shortest path problem to learning multipaths. Additionally, we also introduce a new and faster prediction technique for CH for multipaths which directly samples from an appropriate distribution, bypassing the need to create a decomposition with small support.

1.4.3 Overview of Chapter 4: Online Non-Additive Path Learning

We study the problem of online path learning with non-additive gains, which is a central problem appearing in several applications, including ensemble structured prediction. We present new online algorithms for path learning with non-additive count-based gains for the three settings of full information, semi-bandit and full bandit. These algorithms admit very favorable regret guarantees and their guarantees can be viewed as the non-additive counterparts to the best known guarantees in the additive case. A key component of our algorithms is the definition and computation of an intermediate context-dependent automaton that enables us to use existing algorithms designed for additive gains. We further apply our methods to the important application of ensemble structured prediction. Finally, beyond count-based gains, we give an efficient implementation of the EXP3 algorithm for the full bandit setting with an arbitrary (non-additive) gain.

Chapter 2

Online Learning with Extended Formulations

This chapter introduces a general methodology for developing efficient and effective algorithms for learning combinatorial structures. Examples include learning the best permutation of a set of elements for scheduling or assignment problems, or learning the best Huffman tree for compressing sequences of symbols. Online learning algorithms are being successfully applied to an increasing variety of problems, so it is important to have good tools and techniques for creating good algorithms that match the particular problem at hand.

The online learning setting proceeds in a series of trials where the algorithm makes a prediction or takes an action associated with an object in the appropriate combinatorial space and then receives the loss of its choice in such a way that the loss of any of the possible combinatorial objects can be easily computed (see Prediction

Prediction Game 2 Prediction game for the combinatorial class $\mathcal{H} \subset \mathbb{N}^n$.

1: For each trial $t = 1, \ldots, T$

2: The **learner** randomly predicts with an object $\widehat{\boldsymbol{h}}_{t-1}$ in class $\mathcal{H}$.

3: The **adversary** reveals a loss vector $\boldsymbol{\ell}_t \in [0,1]^n$.

4: The **learner** incurs a (expected) linear loss $\mathbb{E}[\widehat{\boldsymbol{h}}_{t-1} \cdot \boldsymbol{\ell}_t]$.

Game 2). The algorithm can then update its internal representation based on this feedback and the process moves on to the next trial. The algorithm's performance is measured by its *regret*, the amount of loss the algorithm incurs above that of the single best predictor in some comparator class. Usually the comparator class is the class of objects in the combinatorial space being learned. To make the setting concrete, consider the case of learning Huffman trees for compression[1]. In each trial, the algorithm would (perhaps randomly) predict a Huffman tree, and then obtain a sequence of symbols to be encoded. The loss of the algorithm on that trial is the average number of bits per symbol to encode the sequence using the predicted Huffman tree. This loss is essentially the inner product of the frequency vector of the symbols and the code lengths of the symbols. More generally, the loss could be defined as the inner product of any loss vector from the unit cube and the code lengths of the symbols. The total loss of the algorithm is the expected average bits per symbol summed over trials. The regret of the algorithm is the difference between its total loss and the sum over trials of the average bits per symbol for the single best Huffman tree chosen in hindsight. Therefore the

[1]Huffman trees [Cormen et al., 2009] are binary trees which construct prefix codes (called Huffman codes) for data compression. The plaintext symbols are located at the leaves of the tree and the path from the root to each leaf defines the prefix code for the symbols at the leaves.

regret of the algorithm can be viewed as the cost of not knowing the best combinatorial object ahead of time. With proper tuning, the regret is typically logarithmic in the number of combinatorial objects.

One way to create algorithms for these combinatorial problems is to use one of the well-known "experts algorithms" like the Randomized Weighted Majority [Littlestone and Warmuth, 1994] or Hedge algorithm [Freund and Schapire, 1997] where the set of combinatorial objects is the set of "experts". However, unless some additional structure is used, this requires explicitly keeping track of one weight for each of the exponentially many combinatorial objects, and this results in an inefficient algorithm. Furthermore, in this case an additional loss range factor appears in the regret bounds. There has been much work on creating efficient algorithms that implicitly encode the weights over the set of exponentially many combinatorial objects using concise representations. For example, many distributions over the 2^n subsets of n elements can be encoded by the probability of including each of the n elements. In addition to subsets, such work includes permutations [Helmbold and Warmuth, 2009, Yasutake et al., 2011, Ailon, 2014], paths [Takimoto and Warmuth, 2003, Kuzmin and Warmuth, 2005], and k-sets [Warmuth and Kuzmin, 2008].

There are also some general tools for learning combinatorial objects such as the Component Hedge algorithm of Koolen et al. [2010]. This algorithm maintains one weight per component (instead of one weight per combinatorial object) and it does multiplicative updates on these weights. However, the vector of component weights is typically constrained to lie in a convex polytope. Therefore Bregman projections are

used after the update to return the weight vector to the desired polytope. A limitation of Component Hedge is its projection step which is generally only computationally efficient when there are a small (polynomial) number of constraints on the weights.

Suehiro et al. [2012] introduced another projection-based algorithm which specializes the Component Hedge algorithm for structures that can be formulated by submodular functions[2]. It guarantees the same regret bounds as Component Hedge but offers efficient projection and prediction when the convex hull of combinatorial objects is a specialized polytope characterized by a submodular function [Fujishige, 2005].

There are also projection-free algorithms for online learning such as the *Follow the Perturbed Leader (FPL)* algorithm [Kalai and Vempala, 2005] and its generalizations [Dudík et al., 2017]. These algorithms are based on adding random perturbations to the cumulative loss of each component, and then predicting with the combinatorial object with minimum perturbed loss. FPL is very efficient and usually offers the same regret bound guarantees as the Expanded Hedge algorithm when applied to combinatorial objects.

In this chapter we focus on the combinatorial objects whose convex hull in the original space has exponentially many facets. Examples are permutation and Huffman trees. A powerful technique – namely *extended formulations* – has been developed to represent these polytopes in auxiliary spaces using far fewer (polynomial instead of exponential) constraints [Kaibel and Pashkovich, 2013, Kaibel, 2011, Conforti et al., 2010]. Using this technique, we extend Component Hedge to combinatorial objects

[2]For instance, permutations belong to such classes of structures (see Suehiro et al. [2012]); but Huffman trees do not as the sum of the code lengths of the symbols is not fixed.

with ill-behaved polytopes.

Contributions: The main contributions of this chapter are:

1. The application of extended formulation techniques for online learning.

   In particular, the fusion of Component Hedge with extended formulations results in a new methodology for designing efficient online algorithms for complex classes of combinatorial objects. Our methodology uses a redundant representation for the combinatorial objects where one part of the representation allows for a natural loss measure while another enables the simple specification of the class using only polynomially many constraints. We are unaware of previous online learning work exploiting this kind of redundancy. To better match the extended formulations to the machine learning applications, we augment the extended formulation with slack variables.

2. A new and faster prediction technique.

   Component Hedge applications usually predict by first re-expressing the algorithm's weight vector as a convex combination of combinatorial objects with small support, and then randomly sample from the convex combination. The redundant representation often allows for a more direct and efficient way to generate the algorithm's random prediction, bypassing the need to create convex combinations. This is always the case for extended formulations based on "reflection relations" (as in permutations and Huffman Trees).

3. A new and elegant initialization method.

   Component Hedge style loss bounds depend on the distance from the initial hypothesis to the best predictor in the class, and a roughly uniform initialization is usually a good choice. The initialization of the redundant representation is more delicate. Rather than directly picking a feasible initialization, we introduce the idea of first creating an infeasible encoding with good distance properties, and then projecting it into the feasible polytope. This style of implicit initialization improves the regret bounds for some existing work (e.g. the algorithm of Yasutake et al. [2011] has now the state-of-the-art regret bound for permutations) and will be used to good effect in another domain (e.g. for repeatedly solving a variant of the same dynamic programming problem in successive trials[3] [Rahmanian and Warmuth, 2017]).

Chapter Outline: Section 2.1 contains a few additional remarks about the Component Hedge algorithm (see Section 1.3 for a detailed explanation) and an overview of extended formulations. Section 2.2 explains our methodology. In Section 2.3, we explore the concrete application of our method to Huffman trees and permutations. In these applications, the extended formulations are constructed by reflection relations. Section 2.4 describes our fast prediction technique. Next, in Section 2.5 we discuss in detail how to do projection. In Section 2.6, we contrast our bounds with those of FPL [Kalai and Vempala, 2005], Hedge [Freund and Schapire, 1997] and OnlineRank [Ailon,

[3] This is described in Chapter 3.

2014]. We show the regret bounds of our algorithm are within a factor of $O(\log(n))$ of those of OnlineRank (the state-of-the-art specialized algorithm for permutations), and depending on the loss regimes, improve on or match the state-of-the-art for Huffman trees. Finally, we conclude with suggesting some directions for future work.

2.1 Background

The implicit representations for structured concepts (sometimes called 'indirect representations') have been used for a variety of problems [Helmbold et al., 2002, Helmbold and Schapire, 1997, Maass and Warmuth, 1998, Takimoto and Warmuth, 2002, 2003, Yasutake et al., 2011, Koolen et al., 2010]. Recall from the Prediction Game 2, that $t \in \{1..T\}$ is the trial index, $\mathcal{H}$ is the class of combinatorial objects, $\widehat{\boldsymbol{h}}_{t-1} \in \mathcal{H}$ is the algorithm's selected object at the beginning of trial t, and $\boldsymbol{\ell}_t$ is the loss vector revealed by the adversary.

Component Hedge: Koolen et al. [2010] developed a generic algorithm called *Component Hedge* which results in efficient and effective online algorithms over combinatorial objects in $\mathbb{R}^n_+$ with linear loss. Component Hedge maintains a mean vector $\boldsymbol{v}$ in the polytope $\mathcal{V}$ which is the convex hull of all objects in the combinatorial class $\mathcal{H}$. See Section 1.3 for a more detailed overview of the Component Hedge algorithm.

Component Hedge relies heavily on an efficient characterization of the polytope $\mathcal{V}$ (i.e. a description of the facets in terms of inequalities) both for projection and decomposition steps. If directly characterizing the polytope $\mathcal{V}$ is either unknown or

requires exponentially many facets, the direct application of Component Hedge is not efficient. In those cases, we show how extended formulations can help with efficiently describing the polytope $\mathcal{V}$.

Extended Formulations: Many classes of combinatorial objects have polytopes whose discription requires exponentially many facets in the original space (e.g. see Maurras et al. [2010]). This has triggered the search for more concise descriptions in alternative spaces. In recent years, the combinatorial optimization community has given significant attention to the technique of *extended formulation* where difficult polytopes are represented as a linear projection of a higher-dimensional polyhedron [Magnanti and Wolsey, 1995, Conforti et al., 2010, Kaibel, 2011, Kaibel and Pashkovich, 2013]. There are many complex combinatorial objects whose associated polyhedra can be described as the linear projection of a much simpler, but higher dimensional, polyhedra (see Figure 2.1).

Concretely, assume a polytope $\mathcal{V} \subset \mathbb{R}^n_+$ is given and described with exponentially many constraints in matrix-vector multiplication form as $\mathcal{V} = \{\boldsymbol{v} \in \mathbb{R}^n_+ \mid M_1\boldsymbol{v} \leq \boldsymbol{d}\}$ in the original space $\mathbb{R}^n_+$. We assume that using some additional variables $\boldsymbol{x} \in \mathbb{R}^m_+$, $\mathcal{V}$ can be written efficiently as

$$\mathcal{V} = \{\boldsymbol{v} \in \mathbb{R}^n_+ \mid \exists \boldsymbol{x} \in \mathbb{R}^m_+ : M_2\boldsymbol{v} + M_3\boldsymbol{x} \leq \boldsymbol{f}\}^4 \tag{2.1}$$

with $r = \text{poly}(n)$ inequality constraints of $M_2\boldsymbol{v} + M_3\boldsymbol{x} \leq \boldsymbol{f}$. Vector $\boldsymbol{x} \in \mathbb{R}^m_+$ is an

[4]Note that for each $\boldsymbol{v} \in \mathcal{V}$ there exists a $\boldsymbol{x} \in \mathcal{X}$, but it is not necessarily unique.

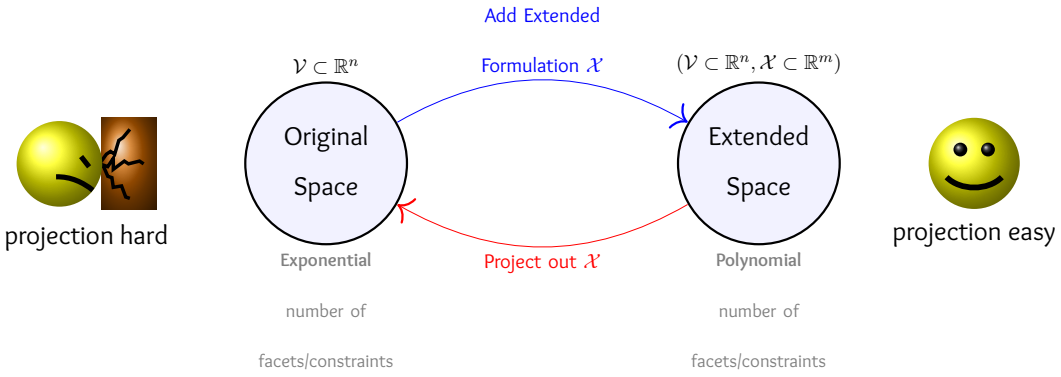


Figure 2.1: Extended formulation.

extended formulation[5] which belongs to the set

$$\mathcal{X} = \{\boldsymbol{x} \in \mathbb{R}^m_+ \mid \exists \boldsymbol{v} \in \mathbb{R}^n_+ : M_2\boldsymbol{v} + M_3\boldsymbol{x} \leq \boldsymbol{f}\}. \tag{2.2}$$

Extended formulations incur the cost of additional m variables for the benefit of a simpler (although, higher dimensional) polytope.

2.2 The Method

Here we describe our general methodology for using extended formulations to develop new learning algorithms. Our focus is on combinatorial objects whose convex hull in the original space has exponentially many facets. To implement efficient algorithms, we need to work with polytopes with small number of facets. Each facet is characterized by an equality constraint and the polytope is defined as the intersection of the constraints. To obtain a well-behaved polytope, we work with another

[5]Throughout this chapter, we assume w.l.o.g. that $\boldsymbol{x}$ is in positive quadrant of $\mathbb{R}^n$, since an arbitrary point in $\mathbb{R}^n$ can be written as $\boldsymbol{x} = \boldsymbol{x}_+ - \boldsymbol{x}_-$ where $\boldsymbol{x}_+, \boldsymbol{x}_- \in \mathbb{R}^n_+$.

space/representation where we add some extended variables to the original set of variables.

Consider a class $\mathcal{H}$ of combinatorial objects and its convex hull $\mathcal{V}$. We assume there is no efficient description of $\mathcal{V}$ in $\mathbb{R}^n_+$, but it can be efficiently characterized via an extended formulation $\boldsymbol{x} \in \mathcal{X}$ as in Equations (2.1) and (2.2). As described in Section 2.1, in order to apply Component Hedge (especially the projection), we need to have equality constraints instead of inequality ones. Thus, we introduce slack variables $\boldsymbol{s} \in \mathbb{R}^r_+$, where r is the number of constraints. Equation (2.1) now becomes

$$\mathcal{V} = \{\boldsymbol{v} \in \mathbb{R}^n_+ \mid \exists \boldsymbol{x} \in \mathbb{R}^m_+, \boldsymbol{s} \in \mathbb{R}^r_+ : M_2\boldsymbol{v} + M_3\boldsymbol{x} + \boldsymbol{s} = \boldsymbol{f}\}$$

Now, in order to keep track of a mean vector $\boldsymbol{v} \in \mathcal{V}$, we use the following novel representation:

$$\mathcal{W} = \{\underbrace{(\boldsymbol{v}, \boldsymbol{x}, \boldsymbol{s})}_{\boldsymbol{w}} \in \mathbb{R}^{n+m+r}_+ \mid M_2\boldsymbol{v} + M_3\boldsymbol{x} + \boldsymbol{s} = \boldsymbol{f}\}$$

where $\mathcal{W}$ is characterized by r affine constraints. We refer to $\mathcal{W}$ as *the augmented formulation*. Observe that, despite potential redundancy in representation, all three constituents are useful in this new encoding: $\boldsymbol{v}$ is needed to encode the right loss, $\boldsymbol{x}$ is used for efficient description of the polytope, and $\boldsymbol{s}$ is incorporated to have equality constraints.

2.2.1 XF-Hedge Algorithm

Having developed the well-equipped space/representation $\mathcal{W}$, we can now apply Component Hedge. Since $\boldsymbol{v}$ is the only constituent over which the loss vector $\boldsymbol{\ell}_t$

is defined, we work with $\boldsymbol{L}_t = (\boldsymbol{\ell}_t, \mathbf{0}, \mathbf{0}) \in [0,1]^{n+m+r}$ in the augmented formulation space $\mathcal{W}$. We introduce a new type of Hedge algorithm combined with the extended formulation – *XF-Hedge* (See Algorithm 3). Similar to Component Hedge, XF-Hedge consists of three main steps: *Prediction*, *Update*, and *Projection*.

Algorithm 3 XF-Hedge

1: $\boldsymbol{w}_0 = (\boldsymbol{v}_0, \boldsymbol{x}_0, \boldsymbol{s}_0) \in \mathcal{W}$ – a proper prior distribution discussed in 2.2.2

2: For $t = 1, \ldots, T$

3: $\quad \widehat{\boldsymbol{h}}_{t-1} \leftarrow$ **Prediction**$(\boldsymbol{w}_{t-1})$ where $\widehat{\boldsymbol{h}}_{t-1} \in \mathcal{H}$ is a random object s.t. $\mathbb{E}\left[\widehat{\boldsymbol{h}}_{t-1}\right] = \boldsymbol{v}_{t-1}$

4: $\quad$ Incur a loss $\widehat{\boldsymbol{h}}_{t-1} \cdot \boldsymbol{\ell}^t$

5: $\quad$ **Update**:

6: $\quad\quad \tilde{v}^{t-1,i} \leftarrow v_{t-1,i}\, e^{-\eta\, \ell_i^t}$ for all $i \in [n]$

7: $\quad \boldsymbol{w}_t \leftarrow$ **Projection**$\underbrace{(\widetilde{\boldsymbol{v}}_{t-1}, \boldsymbol{x}_{t-1}, \boldsymbol{s}_{t-1})}_{\widetilde{\boldsymbol{w}}_{t-1}}$ where $\boldsymbol{w}_t = \underset{\boldsymbol{w} \in \mathcal{W}}{\arg\min}\, \Delta\left(\boldsymbol{w} || \widetilde{\boldsymbol{w}}_{t-1}\right)$

Prediction: Randomly select an object $\widehat{\boldsymbol{h}}_{t-1}$ from the combinatorial class $\mathcal{H}$ in such a way that $\mathbb{E}\left[\widehat{\boldsymbol{h}}_{t-1}\right] = \boldsymbol{v}_{t-1}$ where $\boldsymbol{w}_{t-1} = (\boldsymbol{v}_{t-1}, \boldsymbol{x}_{t-1}, \boldsymbol{s}_{t-1})$. The details of this step depend on the combinatorial class $\mathcal{H}$ and the extended formulation used for $\mathcal{W}$. In Component Hedge and its applications [Helmbold and Warmuth, 2009, Koolen et al., 2010, Yasutake et al., 2011, Warmuth and Kuzmin, 2008], this step is usually done by decomposing[6] $\boldsymbol{v}_{t-1}$ into a convex combination of objects in $\mathcal{H}$. In Section 2.4, using other components of $\boldsymbol{w}_{t-1}$ (i.e. $\boldsymbol{v}_{t-1}$ and $\boldsymbol{x}_{t-1}$), we present a faster $\mathcal{O}(m+n)$ prediction

[6]Note that according to Caratheodory's theorem, such decomposition exists in $\mathcal{W}$ using at most $n+m+r+1$ objects (i.e. corners of the polytope $\mathcal{W}$).

method for combinatorial classes $\mathcal{H}$ where the additional variables and hyperplanes of the extended formulation are constructed by reflection relations.

Update: Having defined $\boldsymbol{L}_t = (\boldsymbol{\ell}_t, \mathbf{0}, \mathbf{0})$, the updated $\widetilde{\boldsymbol{w}}_{t-1} = (\widetilde{\boldsymbol{v}}_{t-1}, \widetilde{\boldsymbol{x}}_{t-1}, \widetilde{\boldsymbol{s}}_{t-1})$ is obtained using a trade-off between the linear loss and the unnormalized relative entropy [Koolen et al., 2010]:

$$\widetilde{\boldsymbol{w}}_{t-1} = \operatorname*{arg\,min}_{\boldsymbol{w}\in\mathbb{R}^r} \Delta(\boldsymbol{w}||\boldsymbol{w}_{t-1}) + \eta\, \boldsymbol{w}\cdot\boldsymbol{L}_t, \qquad \text{where} \qquad \Delta(\boldsymbol{a}||\boldsymbol{b}) = \sum_i a_i \log\frac{a_i}{b_i} + b_i - a_i$$

Using Lagrange multipliers, it is fairly straight-forward to see that only the $\boldsymbol{v}$ components of $\boldsymbol{w}_{t-1}$ are updated:

$$\forall i \in \{1..n\}: \ \tilde{v}_{t-1,i} = v_{t-1,i}\, e^{-\eta\, \ell^t_i}; \qquad \widetilde{\boldsymbol{x}}_{t-1} = \boldsymbol{x}_{t-1}; \qquad \widetilde{\boldsymbol{s}}_{t-1} = \boldsymbol{s}_{t-1}.$$

Thus this step takes $\mathcal{O}(n)$ time.

Projection: We use an unnormalized relative entropy Bregman projection to project $\widetilde{\boldsymbol{w}}_{t-1} = (\widetilde{\boldsymbol{v}}_{t-1}, \widetilde{\boldsymbol{x}}_{t-1}, \widetilde{\boldsymbol{s}}_{t-1})$ back into $\mathcal{W}$ obtaining the new $\boldsymbol{w}_t = (\boldsymbol{v}_t, \boldsymbol{x}_t, \boldsymbol{s}_t)$ for the next trial.

$$\boldsymbol{w}_t = \operatorname*{arg\,min}_{\boldsymbol{w}\in\mathcal{W}} \Delta(\boldsymbol{w}||\widetilde{\boldsymbol{w}}_{t-1}) \tag{2.3}$$

Let $\Psi_0, \ldots, \Psi_{r-1}$ be the r facets where the r constraints specifying the polytope $\mathcal{W}$ (i.e. the ones of $M_2\boldsymbol{v} + M_3\boldsymbol{x} + \boldsymbol{s} = \boldsymbol{f}$) are satisfied. Then $\mathcal{W}$ is the intersection of the Ψ_k's. Since the non-negativity constraints are already enforced by the definition of $\Delta(\cdot||\cdot)$, it is possible to solve (2.3) using iterative Bregman projections[7] [Bregman,

[7] In Helmbold and Warmuth [2009] Sinkhorn balancing is used for projection which is also a special case of iterative Bregman projection.

1967]. Starting from $\boldsymbol{p}_0 = \widetilde{\boldsymbol{w}}_{t-1} \notin \mathcal{W}$, we iteratively compute:

$$\boldsymbol{p}_k = \underset{\boldsymbol{p} \in \Psi_{(k \bmod r)}}{\arg\min} \; \Delta(\boldsymbol{p} || \boldsymbol{p}_{k-1}) \tag{2.4}$$

repeatedly cycling through the constraints. Finding the solution $\boldsymbol{p}_k$ to the convex optimization (2.4) for general constraints and Bregman divergences can be found in Section 3 of Dhillon and Tropp [2007]. For the sake of completeness, we discuss the details of this step in Section 2.5 for combinatorial classes $\mathcal{H}$ whose extended formulation is constructed by reflection relations. It is known that $\boldsymbol{p}_k$ converges in Euclidean norm to the unique solution of (2.3) [Bregman, 1967, Bauschke and Borwein, 1997]. These kinds of cyclic Bregman projections are believed to have fast linear convergence [Dhillon and Tropp, 2007], and empirically are very efficient [Koolen et al., 2010]. Moreover, the projection step typically dominates the overall running time of the algorithm[8].

2.2.2 Regret Bounds

Similar to Component Hedge, the general regret bound depends on the initial weight vector $\boldsymbol{w}_0 \in \mathcal{W}$. For each object $\boldsymbol{h} \in \mathcal{H}$, let $\boldsymbol{w}(\boldsymbol{h})$ be the representation of $\boldsymbol{h}$ in the augmented formulation $\mathcal{W}$. The regret bound of XF-Hedge depends on $\Delta(\boldsymbol{w}(\boldsymbol{h}) || \boldsymbol{w}_0)$ where $\boldsymbol{w}_0 \in \mathcal{W}$ is the initial weight vector and $\boldsymbol{h}$ is the combinatorial object against which the algorithm is compared (the best $\boldsymbol{h}$ for the adversarially chosen sequence of losses).

The following lemma provides bounds in complete knowledge case where the

[8] If the model is being trained on data where predictions are not required, then the expensive projection step can be deferred until the predictions are needed.

loss of the best combinatorial object is known in order to tune the learning rate η. However only slightly worse bounds can be achieved with doubling tricks to handle the unknown loss (see, e.g. Cesa-Bianchi et al. [1997], Cesa-Bianchi and Lugosi [2006]).

Lemma 2. *Let* $L^* := \min_{\boldsymbol{h}\in\mathcal{H}} \sum_{t=1}^{T} \boldsymbol{h}\cdot\boldsymbol{\ell}_t$. *By proper tuning of the learning rate* η*:*

$$\mathbb{E}\left[\sum_{t=1}^{T} \widehat{\boldsymbol{h}}_{t-1}\cdot\boldsymbol{\ell}_t\right] - \min_{\boldsymbol{h}\in\mathcal{H}} \sum_{t=1}^{T} \boldsymbol{h}\cdot\boldsymbol{\ell}_t \leq \sqrt{2L^*\,\Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_0)} + \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_0)$$

Proof. The proof uses standard techniques from the online learning literature (e.g. [Koolen et al., 2010]). Assuming $\boldsymbol{w} = (\boldsymbol{v}, \boldsymbol{x}, \boldsymbol{s})$ and $\boldsymbol{L} = (\boldsymbol{\ell}, \boldsymbol{0}, \boldsymbol{0})$:

$$\begin{aligned}
(1-e^{-\eta})\boldsymbol{v}_{t-1}\cdot\boldsymbol{\ell}_t = (1-e^{-\eta})\boldsymbol{w}_{t-1}\cdot\boldsymbol{L}_t &\leq \sum_i w_{t-1,i}(1-e^{-\eta\, L_{t,i}}) \\
&= \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_{t-1}) - \Delta(\boldsymbol{w}(\boldsymbol{h})||\widetilde{\boldsymbol{w}}_{t-1}) + \eta\,\boldsymbol{w}(\boldsymbol{h})\cdot\boldsymbol{L}_t \\
&= \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_{t-1}) - \Delta(\boldsymbol{w}(\boldsymbol{h})||\widetilde{\boldsymbol{w}}_{t-1}) + \eta\,\boldsymbol{h}\cdot\boldsymbol{\ell}_t \\
&\leq \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_{t-1}) - \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_t) + \eta\,\boldsymbol{h}\cdot\boldsymbol{\ell}_t
\end{aligned}$$

The first inequality is obtained using $1-e^{-\eta x} \geq (1-e^{-\eta})x$ for $x \in [0,1]$ as done in Littlestone and Warmuth [1994]. The second inequality is a result of the Generalized Pythagorean Theorem [Herbster and Warmuth, 2001], since $\boldsymbol{w}_t$ is a Bregman projection of $\widehat{\boldsymbol{w}}_{t-1}$ into the convex set $\mathcal{W}$ which contains $\boldsymbol{w}(\boldsymbol{h})$. By summing over $t = 1\ldots T$ and using the non-negativity of divergences, we obtain:

$$\begin{aligned}
&(1-e^{-\eta})\sum_{t=1}^{T} \boldsymbol{v}_{t-1}\cdot\boldsymbol{\ell}_t \leq \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_0) - \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_T) + \eta\sum_{t=1}^{T} \boldsymbol{h}\cdot\boldsymbol{\ell}_t \\
&\longrightarrow \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{h}_{t-1}\cdot\boldsymbol{\ell}_t\right] \leq \frac{\eta\sum_{t=1}^{T} \boldsymbol{h}\cdot\boldsymbol{\ell}_t + \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_0)}{1-e^{-\eta}}
\end{aligned}$$

Let $L^* = \min_{\boldsymbol{h}\in\mathcal{H}} \sum_{t=1}^{T} \boldsymbol{h}\cdot\boldsymbol{\ell}_t$. We can set the learning rate $\eta = \sqrt{\frac{2\,\Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_0)}{L^*}}$ as instructed in Koolen et al. [2010] and obtain the following regret bound:

$$\mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{h}_{t-1}\cdot\boldsymbol{\ell}_t\right] - \min_{\boldsymbol{h}\in\mathcal{H}} \sum_{t=1}^{T} \boldsymbol{h}\cdot\boldsymbol{\ell}_t \leq \sqrt{2L^*\,\Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_0)} + \Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_0)$$

■

In order to get good bounds, the initial weight $\boldsymbol{w}_0$ must be "close" to all corners $\boldsymbol{h}$ of the polytope, and thus in the"middle" of $\mathcal{W}$. In previous works [Koolen et al., 2010, Yasutake et al., 2011, Helmbold and Warmuth, 2009], the initial weight is explicitly chosen and it is often set to be the uniform mean of the objects. This explicit initialization approach may be difficult to perform when the polytope has a complex structure.

Here, instead of explicitly selecting $\boldsymbol{w}_0 \in \mathcal{W}$, we implicitly design the initial point. First, we find an intermediate "middle" point $\widetilde{\boldsymbol{w}} \in \mathbb{R}^{n+m+r}$ with good distance properties, and then project $\widetilde{\boldsymbol{w}}$ into $\mathcal{W}$ to obtain the initial $\boldsymbol{w}_0$ for the first trial.

A good choice for $\widetilde{\boldsymbol{w}}$ is $U\,\mathbf{1}$ where $\mathbf{1} \in \mathbb{R}^{n+m+r}$ is the vector of all ones, and $U \in \mathbb{R}_+$ is an upper-bound on the infinity norms of the corners of polytope $\mathcal{W}$. This leads to the nice bound $\Delta(\boldsymbol{w}(\boldsymbol{h})||\widetilde{\boldsymbol{w}}) \leq (n+m+r)U$ for all objects $\boldsymbol{h} \in \mathcal{H}$. The Generalized Pythagorean Theorem [Herbster and Warmuth, 2001] ensures that the same bound holds for $\boldsymbol{w}_0$.

Lemma 3. *Assume that there exists $U \in \mathbb{R}_+$ such that $\|\boldsymbol{w}(\boldsymbol{h})\|_\infty \leq U$ for all $\boldsymbol{h} \in \mathcal{H}$. Then the initialization method finds a $\boldsymbol{w}_0 \in \mathcal{W}$ such that for all $\boldsymbol{h} \in \mathcal{H}$, $\Delta(\boldsymbol{w}(\boldsymbol{h})||\boldsymbol{w}_0) \leq (n+m+r)\,U$.*

Proof. Let $\widetilde{\boldsymbol{w}} = U\,\mathbf{1}$ in which $\mathbf{1} \in \mathbb{R}^{m+n+r}$ is a vector with all ones in its components. Now let $\boldsymbol{w}_0$ be the Bregman projection of $\widetilde{\boldsymbol{w}}$ onto $\mathcal{W}$, that is:

$$\boldsymbol{w}_0 = \arg\min_{\boldsymbol{w} \in \mathcal{W}} \Delta(\boldsymbol{w} || \widetilde{\boldsymbol{w}})$$

Now for all $\boldsymbol{h} \in \mathcal{H}$, we have:

$$\begin{aligned}
\Delta(\boldsymbol{w}(\boldsymbol{h}) || \boldsymbol{w}_0) &\leq \Delta(\boldsymbol{w}(\boldsymbol{h}) || \widetilde{\boldsymbol{w}}) && \text{(Generalized Pythagorean Thm.)} \\
&= \sum_{i \in \{1..n+m+r\}} \left((\boldsymbol{w}(\boldsymbol{h}))_i \log \frac{(\boldsymbol{w}(\boldsymbol{h}))_i}{U} + U - (\boldsymbol{w}(\boldsymbol{h}))_i \right) \\
&\leq \sum_{i \in \{1..n+m+r\}} U && \text{since } (\boldsymbol{w}(\boldsymbol{h}))_i \leq U \\
&= (n+m+r)\,U.
\end{aligned}$$

■

Combining Lemmas 2, and 3 gives the following guarantee.

Theorem 4. *If each $\boldsymbol{\ell}_t \in [0,1]^n$ and $\|\boldsymbol{w}(\boldsymbol{h})\|_\infty \leq U$ for all $\boldsymbol{h} \in \mathcal{H}$, then the regret of XF-Hedge is bounded as follows:*

$$\mathbb{E}\left[\sum_{t=1}^{T} \widehat{\boldsymbol{h}}_{t-1} \cdot \boldsymbol{\ell}^t\right] - \min_{\boldsymbol{h} \in \mathcal{H}} \sum_{t=1}^{T} \boldsymbol{h} \cdot \boldsymbol{\ell}^t \leq \sqrt{2L^*\,(n+m+r)\,U} + (n+m+r)\,U.$$

2.3 XF-Hedge Examples Using Reflection Relations

One technique for constructing extended formulations is to introduce additional variables, constraints and spaces via *reflection relations* [Kaibel and Pashkovich, 2013]. This technique can be used to efficiently describe the polytopes of permutations

and Huffman trees. Here we describe how reflection relations can be used with the XF-Hedge framework to create concrete learning algorithms for permutations and Huffman trees.

As in Yasutake et al. [2011] and Ailon [2014], we consider losses that are linear in the first order representation of the objects [Diaconis, 1988]. For permutations of n items, the first order representation is vectors $\boldsymbol{v} \in \mathbb{R}^n$ where each of the elements of $\{1, 2, \ldots, n\}$ appears exactly once[9] and for Huffman trees on n symbols, the first order representation is vectors $\boldsymbol{v} \in \mathbb{R}^n$ where each v_i is an integer indicating the depth of the leaf corresponding to symbol i in the coding tree. At each trial the loss is $\boldsymbol{v} \cdot \boldsymbol{\ell}$ where the adversary's $\boldsymbol{\ell}$ is a loss vector in the unit cube $[0, 1]^n$. This type of loss is sufficiently rich to capture well-known natural losses like *average code length* for Huffman trees (when ℓ is the symbol frequencies) and *sum of completion times* for permutations[10] (when ℓ is the task completion times).

Constructing Extended Formulations from Reflection Relations. Kaibel and Pashkovich [2013] show how to construct polynomial size extended formulations using a canonical corner of the polytope and a fixed sequence of hyperplanes. These have the property that any corner of the desired polytope can be generated by reflecting the canonical corner through a subsequence of the hyperplanes. These reflections are *one-sided* in the sense that they map the half-space containing the canonical corner to

[9] In contrast, Helmbold and Warmuth [2009] work with the second order representation (i.e. the Birkhoff polytope). Consequently the losses they consider are more general (see Yasutake et al. [2011] for comparison).

[10] To easily encode the sum of completion times, the predicted permutation represents the *reverse* order in which the tasks are to be executed.

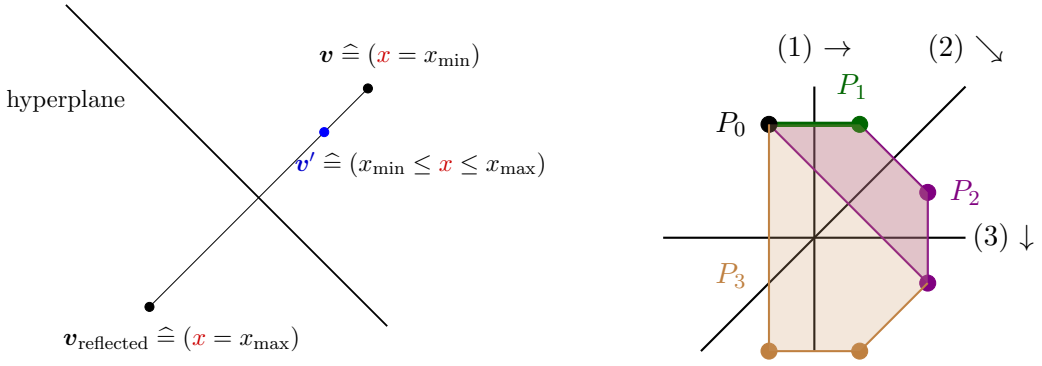


Figure 2.2: (Left) A partial reflection of $\boldsymbol{v}$ to $\boldsymbol{v}'$ corresponds to (denoted by $\mathrel{\widehat{=}}$) a variable x indicating how far $\boldsymbol{v}'$ moves towards $\boldsymbol{v}$'s image $\boldsymbol{v}_{\text{reflected}}$. (Right) The 6 corners of the polytope are generated by subsequences of one-sided reflections through lines (1), (2), and (3), starting from the canonical point P_0. Using partial reflections, we can generate the entire polytope.

the other half-space. For example, the corners of Figure 2.2 (Left) can be generated in this way. Of course the hard part is to find a good sequence of hyperplanes with this property.

A key idea for generating the entire polytope is to allow "partial reflections" where the point to be reflected can not just be kept (skipping the reflection) or replaced by its reflected image, but mapped to any point on the line segment joining the point and its reflected image as illustrated in Figure 2.2 (Right). Since any point in the convex hull of the polytope can be constructed by at least one sequence of partial one-sided reflections, every point in the polytope has an alternative representation in terms of how much each reflection was used to generate it from the canonical corner (see Figure 2.2). Each of these parameterized partial reflections is a *reflection relation*,

For each reflection relation, there will be one additional variable indicating the extent to which the reflection occurs, and two additional inequalities for the extreme cases of complete reflection and no reflection. Therefore, if the polytope can be expressed with polynomially many reflection relations, then it has an extended formulation of polynomial size with polynomially many constraints.

Instead of starting with a single corner, one could also consider passing an entire polytope as an input through the sequence of (partial) reflections to generate a new polytope. Using this fact, Theorem 1 in Kaibel and Pashkovich [2013] provides an inductive construction of higher dimensional polytopes via sequences of reflection relations. Concretely, let P^n_{obj} be the polytope of a given combinatorial object of size n. The typical approach is to properly embed $P^n_{\text{obj}} \subset \mathbb{R}^n$ into $\widehat{P}^n_{\text{obj}} \subset \mathbb{R}^{n+1}$, and then feed it through an appropriate sequence of reflection relations as an input polytope in order to obtain an extended formulation for $P^{n+1}_{\text{obj}} \subset \mathbb{R}^{n+1}$. Theorem 1 in Kaibel and Pashkovich [2013] provides sufficient conditions for the correctness of this procedure. Again, if polynomially many reflection relations are used to go from n to $n+1$, then we can construct an extended formulation of polynomial size for P^n_{obj} with polynomially many constraints. In this chapter, however, we work with batch construction of the extended formulation as opposed to the inductive construction.

Extended Formulations for Objects Closed under Re-Ordering. Assume we want to construct an extended formulation for a class of combinatorial objects which is closed under any re-ordering (both Huffman trees and permutations both have this

property). Then reflection relations corresponding to swapping pairs of elements are useful. Swapping elements i and j can be implemented with a hyperplane going through the origin and having normal vector $\boldsymbol{e}_i - \boldsymbol{e}_j$ (here $\boldsymbol{e}_i$ is the ith unit vector). The identity permutation is the natural canonical corner, so the one-sided reflections are only used for $\boldsymbol{v}$ where $v_i \leq v_j$.

Implementing the reflection relation for the i, j swap uses an additional variable along with two additional inequalities. Concretely, assume $\boldsymbol{v} \in \mathbb{R}^n$ is going into this reflection relation and $\boldsymbol{v}' \in \mathbb{R}^n$ is the output, so $\boldsymbol{v}'$ is in the convex combination of $\boldsymbol{v}$ and its reflection. It is natural to encode this as $\boldsymbol{v}' = \gamma \boldsymbol{v} + (1-\gamma)\boldsymbol{v}_{\text{reflected}}$. However, we found it more convenient to parameterize $\boldsymbol{v}'$ by its absolute distance x from $\boldsymbol{v}$, rather than the relative distance $\gamma \in [0, 1]$. Using this parameterization, we have $\boldsymbol{v}' = \boldsymbol{v} + x\,(\boldsymbol{e}_i - \boldsymbol{e}_j)$ constrained by $(\boldsymbol{e}_i - \boldsymbol{e}_j) \cdot \boldsymbol{v} \leq (\boldsymbol{e}_i - \boldsymbol{e}_j) \cdot \boldsymbol{v}' \leq -(\boldsymbol{e}_i - \boldsymbol{e}_j) \cdot \boldsymbol{v}$. Therefore the possible relationships between $\boldsymbol{v}'$ and $\boldsymbol{v}$ can be encoded with the additional variable x and the following constraints:[11]

$$\boldsymbol{v}' = \boldsymbol{m}\, x + \boldsymbol{v} \quad \text{where } \boldsymbol{m} = \boldsymbol{e}_i - \boldsymbol{e}_j, \qquad 0 \leq x \leq v_j - v_i. \tag{2.5}$$

Notice that x indicates the amount of change in the ith and jth elements which can go from zero (remaining unchanged) to the maximum swap capacity $v_j - v_i$.

Suppose the desired polytope is described using m reflection relations and with canonical point $\boldsymbol{c}$. Then starting from $\boldsymbol{c}$ and successively applying the equation in (2.5), we obtain the connection between the extended formulation space $\mathcal{X}$ and original space

[11] In general $\boldsymbol{v}$ (and thus v_j and v_i) may be functions of the variables for previous reflection relations.

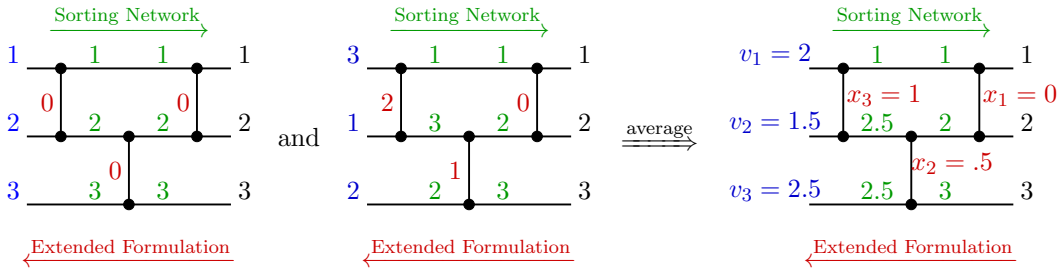


Figure 2.3: An extended formulation for permutation on $n = 3$ items. The canonical permutation is $[1, 2, 3]$. Elements of $\boldsymbol{v}$ are in blue, $\boldsymbol{x}$ in red, and the intermediate values are in green.

$\mathcal{V}$:

$$\boldsymbol{v} = M\,\boldsymbol{x} + \boldsymbol{c}, \quad \boldsymbol{v}, \boldsymbol{c} \in \mathcal{V} \subset \mathbb{R}^n,\ \boldsymbol{x} \in \mathcal{X} \subset \mathbb{R}^m,\ M \in \{-1, 0, 1\}^{n \times m}.$$

Kaibel and Pashkovich [2013] showed that the m reflection relations corresponding to the m comparators in an arbitrary n-input sorting network[12] generates the permutation polytope (see Figure 2.3). A similar extended formulation for Huffman trees can be built using an arbitrary sorting network along with $O(n \log n)$ additional comparators and simple linear maps (which do not require extra variables) and the canonical corner $\boldsymbol{c} = [1, 2, \ldots, n-2, n-1, n-1]^T$ (see Section 2.24 in Pashkovich [2012] for more details). Note that the reflection relations are applied in reverse order than their use in the sorting network (see Figure 2.3).

Learning Permutations and Huffman Trees. As described in the previous subsections, the polytope $\mathcal{V}$ of both permutations and Huffman trees can be efficiently

[12] A *sorting network* is a sorting algorithm where the comparisons are fixed in advance [Cormen et al., 2009].

described using m inequality and n equality constraints[13]:

$$\mathcal{V} = \{\boldsymbol{v} \in \mathbb{R}_+^n | \exists \boldsymbol{x} \in \mathbb{R}_+^m : A\boldsymbol{x} \leq \boldsymbol{b} \text{ and } \boldsymbol{v} = M\boldsymbol{x} + \boldsymbol{c}\}$$

Adding the slack variables $\boldsymbol{s} \in \mathbb{R}_+^m$, we obtain the augmented formulation $\mathcal{W}$:

$$\mathcal{W} = \{\boldsymbol{w} = (\boldsymbol{v}, \boldsymbol{x}, \boldsymbol{s}) \in \mathbb{R}_+^{n+2m} | A\boldsymbol{x} + \boldsymbol{s} = \boldsymbol{b} \text{ and } \boldsymbol{v} = M\boldsymbol{x} + \boldsymbol{c}\} \tag{2.6}$$

Lemma below shows that A and $\boldsymbol{b}$ can be computed in terms of M and $\boldsymbol{c}$.

Lemma 5. *In augmented formulation $\mathcal{W}$ in Equation 2.6:*

$$A = \mathit{Tri}(M^T M) + I, \quad \boldsymbol{b} = -M^T \boldsymbol{c}$$

in which $\mathit{Tri}(\cdot)$ is a function over square matrices which zeros out the upper triangular part of the input including the diagonal.

Proof. Let $\boldsymbol{v}^k$ be the vector in $\mathcal{V}$ after going through the kth reflection relation. Also denote the kth column of M by M_k. Observe that $\boldsymbol{v}^0 = \boldsymbol{c}$ and $\boldsymbol{v}^k = \boldsymbol{c} + \sum_{i=1}^{k} M_i x_i$. Let $M_k = \boldsymbol{e}_r - \boldsymbol{e}_s$. Then, using (2.5), the inequality associated with the kth row of $A\boldsymbol{x} \leq \boldsymbol{b}$ will be obtained as below:

$$x_k \leq v_s^{k-1} - v_r^{k-1} = -M_k^T \boldsymbol{v}^{k-1} = -M_k^T \left(\boldsymbol{c} + \sum_{i=1}^{k-1} M_i x_i \right)$$
$$\longrightarrow x_k + \sum_{i=1}^{k-1} M_k^T M_i x_i \leq -M_k^T \boldsymbol{c} = b_k$$

[13]Positivity constraints are excluded as they are already enforced due to definition of $\Delta(\cdot||\cdot)$, and Huffman trees require additional $\mathcal{O}(n \log n)$ inequality constraints beyond those corresponding to the sorting network.

Thus:

$$\forall\, i,j \in [m]: \quad A_{ij} = \begin{cases} M_i^T M_j & i > j \\ 1 & i = j \\ 0 & i < j \end{cases}, \qquad \forall\, k \in [m]: \quad b_k = -M_k^T \boldsymbol{c}$$

which concludes the proof. ■

Note that all the wire values (i.e. v_i's), as well as x_i's and s_i's are upperbounded by $U = n$. Using the AKS sorting networks with $m = \mathcal{O}(n \log n)$ comparators [Ajtai et al., 1983], we can obtain the regret bounds below from Theorem 4:

Corollary 6. *XF-Hedge has the following regret bound when learning either permutations or Huffman trees:*

$$\mathbb{E}\left[\sum_{t=1}^{T} \widehat{\boldsymbol{h}}_{t-1} \cdot \boldsymbol{\ell}^t\right] - \min_{\boldsymbol{h}\in\mathcal{H}} \sum_{t=1}^{T} \boldsymbol{h} \cdot \boldsymbol{\ell}^t = \mathcal{O}\left(n\,(\log n)^{\frac{1}{2}}\sqrt{L^*} + n^2\,\log n\right)$$

2.4 Fast Prediction with Reflection Relations

From its current weight vector $\boldsymbol{w} = (\boldsymbol{v}, \boldsymbol{x}, \boldsymbol{s}) \in \mathcal{W}$, XF-Hedge randomly selects an object $\widehat{\boldsymbol{h}}$ from the combinatorial class $\mathcal{H}$ in such a way that $\mathbb{E}\left[\widehat{\boldsymbol{h}}\right] = \boldsymbol{v}$. In Component Hedge and similar algorithms [Helmbold and Warmuth, 2009, Koolen et al., 2010, Yasutake et al., 2011, Warmuth and Kuzmin, 2008], this is done by decomposing $\boldsymbol{v}$ into a convex combination of objects in $\mathcal{H}$ followed by sampling. In this section, we give a new more direct prediction method for combinatorial classes $\mathcal{H}$ whose extended formulation is constructed by reflection relations. Our method is faster due to avoiding the decomposition.

The values $\boldsymbol{x}$ and $\boldsymbol{x}+\boldsymbol{s}$ can be interpreted as amount swapped and the maximum swap allowed for the comparators in the sorting networks, respectively. Therefore, it is natural to define $x_i/(x_i+s_i)$ as *swap probability* associated with the ith comparator for $i \in \{1..m\}$. Algorithm 4 incorporates the notion of swap probabilities to construct an efficient sampling procedure from a distribution $\mathcal{D}$ which has the right expectation. It starts with the canonical object (e.g. identity permutation) and feeds it through the reflection relations. Each reflection i is taken with probability $x_i/(x_i+s_i)$. The theorem below guarantees the correctness and efficiency of this algorithm.

Theorem 7. *Given* $(\boldsymbol{v},\boldsymbol{x},\boldsymbol{s}) \in \mathcal{W} \subseteq \mathbb{R}^{n+2m}$,

(i) Algorithm 4 samples $\boldsymbol{h}$ *from a distribution* $\mathcal{D}$ *such that* $\mathbb{E}_{\mathcal{D}}[\boldsymbol{h}] = \boldsymbol{v}$.

(ii) The time complexity of Algorithm 4 is $\mathcal{O}(n+m)$.

Proof. First we prove part (i). Let $\boldsymbol{x} = [x_1, x_2, \ldots, x_m]^T$. Using induction, we prove that by the end of the ith loop of Algorithm 4, the obtained distribution $\mathcal{D}^{(i)}$ has the right expectation for $\boldsymbol{x}^{(i)} = [x_1, \ldots, x_i, 0, \ldots, 0]$. Concretely, $\sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(i)}}[\boldsymbol{h}] \cdot \boldsymbol{h} = M\,\boldsymbol{x}^{(i)} + \boldsymbol{c}$. The desired result is obtained by setting $i = m$ as $\boldsymbol{v} = M\,\boldsymbol{x} + \boldsymbol{c}$. The base case $i = 0$ (i.e. before the first loop of the algorithm) is indeed true, since $\mathcal{D}^{(0)}$ is initialized to follow $P_{\mathcal{D}^{(0)}}[\boldsymbol{c}] = 1$, and $\boldsymbol{x}^{(0)} = \boldsymbol{0}$, thus we have $\boldsymbol{v}^{(0)} = M\boldsymbol{x}^{(0)} + \boldsymbol{c} = \boldsymbol{c}$. Now assume that by the end of the $(k-1)$st iteration we have the right distribution $\mathcal{D}^{(k-1)}$, namely $\boldsymbol{v}^{(k-1)} = \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot \boldsymbol{h}$. Also assume that the kth comparator is applied on ith and jth element[14]. Thus the kth column of M will be $M_k = \boldsymbol{e}_i - \boldsymbol{e}_j$.

[14]Note that $j > i$ as in sorting networks the swap value is propagated to lower wires

Now, according to (2.5) the swap capacity at kth comparator is:

$$\begin{aligned} x_k + s_k &= v_j^{k-1} - v_i^{k-1} \\ &= \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot (h_j - h_i) \\ &= -\sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot M_k^T\, \boldsymbol{h} \\ &= -M_k^T \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot \boldsymbol{h} \end{aligned} \tag{2.7}$$

Now observe:

$$\begin{aligned} \boldsymbol{v}^{(k)} &= M\boldsymbol{x}^{(k)} + \boldsymbol{c} \\ &= x_k M_k + M\boldsymbol{x}^{(k-1)} + \boldsymbol{c} \\ &= x_k M_k + \boldsymbol{v}^{(k-1)} \\ &= x_k M_k + \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot \boldsymbol{h} \\ &= \frac{x_k}{x_k + s_k} M_k\,(x_k + s_k) + \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot \boldsymbol{h} \\ &= -\frac{x_k}{x_k + s_k}\, M_k M_k^T \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot \boldsymbol{h} + \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot \boldsymbol{h} \quad \text{According to (2.7)} \\ &= \left(I - \frac{x_k}{x_k + s_k} M_k M_k^T\right) \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot \boldsymbol{h} \\ &= \left(\frac{s_k}{x_k + s_k} I + \frac{x_k}{x_k + s_k} T_{ij}\right) \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}] \cdot \boldsymbol{h} \\ &= \sum_{\boldsymbol{h}\in\mathcal{H}} \underbrace{\frac{s_k}{x_k + s_k} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}]}_{P_{\mathcal{D}^{(k)}}[\boldsymbol{h}]} \cdot \boldsymbol{h} + \underbrace{\frac{x_k}{x_k + s_k} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}]}_{P_{\mathcal{D}^{(k)}}[T_{ij}\,\boldsymbol{h}]} \cdot T_{ij}\,\boldsymbol{h} \\ &= \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k)}}[\boldsymbol{h}] \cdot \boldsymbol{h} \end{aligned}$$

in which T_{ij} is the row-switching matrix obtained by swapping the ith and jth rows of the identity matrix. For Huffman trees, the linear maps introduced in Pashkovich [2012] are used to set the depths of the leaves. It is straightforward to see that these linear maps maintain the equality $\boldsymbol{v}^{(k)} = \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k)}}[\boldsymbol{h}] \cdot \boldsymbol{h}$ when applied to $\boldsymbol{v}^{(k)}$ and all $\boldsymbol{h}$'s in $\mathcal{H}$. This concludes the inductive proof.

We now prove part (ii). The final distribution $\mathcal{D}$ over objects $\boldsymbol{h} \in \mathcal{H}$ is decomposed into individual actions of swap/pass through the network of comparators independently. Thus one can draw an instance according to the distribution by simply doing independent Bernoulli trials associated with the comparators. It is also easy to see that the time complexity of the algorithm is $\mathcal{O}(n+m)$ since one just needs to do m Bernoulli trials.

■

Using the AKS sorting networks [Ajtai et al., 1983], the number of comparators is $m \in \mathcal{O}(n \log n)$, and thus Algorithm 4 predicts in $\mathcal{O}(n \log n)$ time. This improves the previously known $\mathcal{O}(n^2)$ prediction procedure for mean-based algorithms[15] for permutations [Yasutake et al., 2011, Suehiro et al., 2012].

2.5 Projection with Reflection Relations

In this section, we explore the details of the projection step when working with reflection relations. This is step is done via iterative Bregman projections [Bregman,

[15]It also matches the time complexity of prediction step for non-mean-based permutation-specialized OnlineRank [Ailon, 2014] and also the general FPL [Kalai and Vempala, 2005] algorithm both for permutations and Huffman Trees.

Algorithm 4 Fast-Prediction

Input: $(\boldsymbol{x}, \boldsymbol{s}) \in \mathbb{R}_+^{2m}$

Output: A prediction $\widehat{\boldsymbol{h}} \in \mathcal{H} \subseteq \mathbb{R}^n$

$\widehat{\boldsymbol{h}} \leftarrow \boldsymbol{c}$ – *the canonical corner in* $\mathbb{R}^n$

for $k = 1$ to m **do**

    $(i_k, j_k) \leftarrow$ wire indices associated with the k-th comparator

    if $x_i = 0$ **then**

        continue

    else

        Switch the i_kth and j_kth components of $\widehat{\boldsymbol{h}}$ w.p. $x_k/(x_k + s_k)$.

    end if

end forreturn $\widehat{\boldsymbol{h}}$

1967]. It cycles through the constraints and projects onto their associated facets until convergence. We first discuss the projection onto each constraint in XF-Hedge. Next, we bound the cost of working with approximate projections in XF-Hedge. Finally, we provide a time complexity analysis.

2.5.1 Projection onto Each Constraint in XF-Hedge

Each constraint of the polytope in the augmented formulation is of the form $\boldsymbol{a}^T \boldsymbol{w} = a_0$. Formally, the projection $\boldsymbol{w}^*$ of a give point $\boldsymbol{w}$ to this constraint is solution to the following:

$$\underset{\boldsymbol{a}^T \boldsymbol{w}^* = a_0}{\arg\min} \sum_i w_i^* \log\left(\frac{w_i^*}{w_i}\right) + w_i - w_i^*$$

Finding the solution to the projection above for general constraints and Bregman divergence can be found in Section 3 of Dhillon and Tropp [2007]. Nevertheless, for the sake of completeness, we also provide the solution for the particular case of Huffman trees and permutations. We compute the solution of the projection above to each facet of $\mathcal{W}$ which is described in Section 2.3. Using the method of Lagrange multipliers, we have:

$$L(\boldsymbol{w}^*, \mu) = \sum_i w_i^* \log\left(\frac{w_i^*}{w_i}\right) + w_i - w_i^* - \mu\left(\sum_{j=1}^{2m+n} a_i\, w_i^* - a_0\right)$$

$$\frac{\partial L}{\partial w_i^*} = \log\left(\frac{w_i^*}{w_i}\right) - \mu a_i = 0, \quad \forall i \in [n+2m]$$

$$\frac{\partial L}{\partial \mu} = \sum_{j=1}^{2m+n} a_i\, w_i^* - a_0 = 0$$

Replacing $\rho = e^{-\mu}$, we have $w_i^* = w_i\, \rho^{a_i}$. By enforcing $\frac{\partial L}{\partial \mu} = 0$, one needs to

find $\rho > 0$ such that:

$$\sum_{i=1}^{n+2m} a_i \, w_i \, \rho^{a_i} - a_0 = 0 \tag{2.8}$$

Observe that due to the structure of matrices M and A (see Lemma 5), $a_i \in \mathbb{Z}$ and $a_i \geq -1$ for all $i \in [n + 2m]$, and furthermore $a_0 \geq 0$. Multiplying by ρ, we can re-write equation (2.8) as the polynomial below:

$$f(\rho) = \phi_k \rho^k + \ldots + \phi_2 \rho^2 - \phi_1 \rho - \phi_0 = 0$$

in which all ϕ_i's are non-negative real numbers and $k \leq n$. Note that $f(0) < 0$ and $f(\rho) \to +\infty$ as $\rho \to +\infty$. Thus $f(\rho)$ has at least one positive root. However, it can not have more than one positive roots and we can prove it by contradiction. Assume that there exist $0 < r_1 < r_2$ such that $f(r_1) = f(r_2) = 0$. Since f is convex on positive real line, using Jensen's inequality, we can obtain the contradiction below:

$$0 = f(r_1) = f\left(\frac{r_2 - r_1}{r_2} \times 0 + \frac{r_1}{r_2} \times r_2\right) < \frac{r_2 - r_1}{r_2} f(0) + \frac{r_1}{r_2} f(r_2) = \frac{r_2 - r_1}{r_2} f(0) < 0$$

Therefore f has exactly one positive root which can be found by Newton's method starting from a sufficiently large initial point. Note that if the constraint belongs to $\boldsymbol{v} = M\boldsymbol{x} + \boldsymbol{c}$, then all the a_i's are in $\{-1, 0, 1\}$ and polynomial $f(\rho)$ will be quadratic, so there is a closed form for the positive root.

2.5.2 Additional Loss with Approximate Projection in XF-Hedge

Each iteration of Bregman Projection is described in Section 2.5.1. Since it is basically finding a positive root of a polynomial (which $n/(n + m)$ of the time is

quadratic), each iteration is arguably efficient. Now suppose, using iterative Bregman projections, we reached at $\widehat{\boldsymbol{w}} = (\widehat{\boldsymbol{v}}, \widehat{\boldsymbol{x}}, \widehat{\boldsymbol{s}})$ which is ϵ-close to the exact projection $\boldsymbol{w} = (\boldsymbol{v}, \boldsymbol{x}, \boldsymbol{s})$, that is $\|\boldsymbol{w} - \widehat{\boldsymbol{w}}\|_2 < \epsilon$. In this analysis, we work with a two-level approximation: 1) approximating mean vector $\boldsymbol{v}$ by the mean vector $\widetilde{\boldsymbol{v}} := M\widehat{\boldsymbol{x}} + \boldsymbol{c}$ and 2) approximating the mean vector $\widetilde{\boldsymbol{v}}$ by the mean vector $\boldsymbol{v}(\widehat{\boldsymbol{p}})$ (where $\widehat{\boldsymbol{p}} = \widehat{\boldsymbol{x}}/(\widehat{\boldsymbol{x}} + \widehat{\boldsymbol{s}})$ with coordinate-wise division) obtained from Algorithm 4 with $\widehat{\boldsymbol{x}}$ and $\widehat{\boldsymbol{s}}$ as input. First, observe that:

$$\begin{aligned} \|\boldsymbol{v} - \widetilde{\boldsymbol{v}}\|_2 &= \|M\,(\boldsymbol{x} - \widehat{\boldsymbol{x}})\|_2 \\ &\leq \|M\|_F\,\|\boldsymbol{x} - \widehat{\boldsymbol{x}}\|_2 \\ &\leq (\sqrt{2\,n})\,\epsilon \end{aligned} \tag{2.9}$$

Now suppose we run Algorithm 4 with $\widehat{\boldsymbol{x}}$ and $\widehat{\boldsymbol{s}}$ as input. Let M_k be the k-th column of M, and let $T_{\alpha\beta}$ be the row-switching matrix that is obtained from switching α-th and β-th row in identity matrix. Additionally, let $\boldsymbol{v}^{(k)}(\widehat{\boldsymbol{p}})$ be the mean vector associated with the distribution $\mathcal{D}^{(k)}$ obtained by the end of k-th loop of the Algorithm 4 i.e. $\boldsymbol{v}^{(k)}(\widehat{\boldsymbol{p}}) := \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k)}}[\boldsymbol{h}] \cdot \boldsymbol{h}$ (so $\boldsymbol{v}^{(m)}(\widehat{\boldsymbol{p}}) = \boldsymbol{v}(\widehat{\boldsymbol{p}})$). Also for all $k \in \{1..m\}$ define $\widetilde{\boldsymbol{v}}^{(k)} := \boldsymbol{c} + \sum_{i=1}^{k} M_i\,\widehat{x}_i$ (thus $\widetilde{\boldsymbol{v}}^{(m)} = \widetilde{\boldsymbol{v}}$). Furthermore, let $\boldsymbol{\delta}^{(k)} := \boldsymbol{v}^{(k)}(\widehat{\boldsymbol{p}}) - \widetilde{\boldsymbol{v}}^{(k)}$. Now we can write:

$$
\begin{aligned}
\boldsymbol{v}^{(k)}(\widehat{\boldsymbol{p}}) &= \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k)}}[\boldsymbol{h}]\cdot\boldsymbol{h} \\
&= \sum_{\boldsymbol{h}\in\mathcal{H}} \frac{\widehat{s}_k}{\widehat{x}_k+\widehat{s}_k} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}]\cdot\boldsymbol{h} + \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}]\cdot T_{\alpha\beta}\,\boldsymbol{h} \\
&= \Big(\frac{\widehat{s}_k}{\widehat{x}_k+\widehat{s}_k}\, I + \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, T_{\alpha\beta}\Big) \sum_{\boldsymbol{h}\in\mathcal{H}} P_{\mathcal{D}^{(k-1)}}[\boldsymbol{h}]\cdot\boldsymbol{h} \\
&= (I - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T)\, \boldsymbol{v}^{(k-1)}(\widehat{\boldsymbol{p}}) \qquad \text{since } I - T_{\alpha\beta} = M_k\, M_k^T \\
&= (I - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T)\, \widetilde{\boldsymbol{v}}^{(k-1)} + (I - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T)\, \boldsymbol{\delta}^{(k-1)} \\
&= (I - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T)\, \left(\boldsymbol{c} + \sum_{i=1}^{k-1} M_i\, \widehat{x}_i\right) + (I - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T)\, \boldsymbol{\delta}^{(k-1)} \\
&= \left(\boldsymbol{c} + \sum_{i=1}^{k-1} M_i\, \widehat{x}_i\right) - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T\, \left(\boldsymbol{c} + \sum_{i=1}^{k-1} M_i\, \widehat{x}_i\right) \\
&\quad + (I - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T)\, \boldsymbol{\delta}^{(k-1)} \\
&= \widetilde{\boldsymbol{v}}^{(k)} \underbrace{-M_k\widehat{x}_k - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T\, \left(\boldsymbol{c} + \sum_{i=1}^{k-1} M_i\, \widehat{x}_i\right) + (I - \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}\, M_k\, M_k^T)\, \boldsymbol{\delta}^{(k-1)}}_{\boldsymbol{\delta}^{(k)}}
\end{aligned}
$$

Now define $\widehat{p}_k := \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k}$ and let $\mathrm{err}_k := -M_k^T\, \left(\boldsymbol{c} + \sum_{i=1}^{k-1} M_i\, \widehat{x}_i\right) - (\widehat{x}_k + \widehat{s}_k)$, which is – according to Lemma 5 – the error in the k-th row of $A\boldsymbol{x} + \boldsymbol{s} = \boldsymbol{b}$ using $\widehat{\boldsymbol{x}}$ and $\widehat{\boldsymbol{s}}$ i.e. amount by which $(A\widehat{\boldsymbol{x}} + \widehat{\boldsymbol{s}})_k$ falls short of b_k, violating the k-th constraint of $A\boldsymbol{x} + \boldsymbol{s} = b$. Thus we obtain:

$$
\begin{aligned}
\boldsymbol{\delta}^{(k)} &= -M_k\widehat{x}_k + \widehat{p}_k\, M_k\, (\widehat{x}_k + \widehat{s}_k + \mathrm{err}_k) + (I - \widehat{p}_k\, M_k\, M_k^T)\, \boldsymbol{\delta}^{(k-1)} \\
&= \widehat{p}_k\, M_k\, \mathrm{err}_k + (I - \widehat{p}_k\, M_k\, M_k^T)\, \boldsymbol{\delta}^{(k-1)} \qquad\qquad (\widehat{p}_k = \frac{\widehat{x}_k}{\widehat{x}_k+\widehat{s}_k})
\end{aligned}
$$

Observe that $\boldsymbol{\delta}^{(0)} = \boldsymbol{c} - \boldsymbol{c} = \mathbf{0}$. Thus, by unrolling the recurrence relation above, we have:

$$\boldsymbol{v}(\widehat{\boldsymbol{p}}) - \widetilde{\boldsymbol{v}} = \boldsymbol{\delta}^{(m)} = \sum_{k=1}^{m} \widehat{p}_k \, M_k \, \mathrm{err}_k \prod_{i=k+1}^{m} (I - \widehat{p}_i \, M_i \, M_i^T)$$

Note that since $I - \widehat{p}_i \, M_i \, M_i^T$ is a $n \times n$ doubly-stochastic matrix, $\prod_{i=1}^{k-1}(I - \widehat{p}_i \, M_i \, M_i^T)$ is also a $n \times n$ doubly-stochastic matrix, and consequently, its Frobenius norm is at most $\sqrt{n}$. Thus we have:

$$\begin{aligned}
\|\boldsymbol{v}(\widehat{\boldsymbol{p}}) - \widetilde{\boldsymbol{v}}\|_2 = \|\boldsymbol{\delta}^{(m)}\|_2 &\leq \sum_{k=1}^{m} |\widehat{p}_k| \, \|M_k\|_2 \, |\mathrm{err}_k| \, \sqrt{n} \leq \sqrt{2\,n} \sum_{k=1}^{m} |\mathrm{err}_k| \\
&= \sqrt{2\,n} \, \|\mathbf{err}\|_1 && \mathbf{err} := (\mathrm{err}_1, \ldots, \mathrm{err}_m) \\
&\leq \sqrt{2\,n\,m} \, \|\mathbf{err}\|_2 && (2.10)
\end{aligned}$$

Observe that we can bound the 2-norm of the vector $\mathbf{err}$ as follows:

$$\begin{aligned}
\|\mathbf{err}\|_2 &= \| - A\widehat{\boldsymbol{x}} - \widehat{\boldsymbol{s}} + \boldsymbol{b}\|_2 \\
&= \|A(\boldsymbol{x} - \widehat{\boldsymbol{x}}) + (\boldsymbol{s} - \widehat{\boldsymbol{s}})\|_2 && (\boldsymbol{b} = A\boldsymbol{x} + \boldsymbol{s}) \\
&\leq \|A\|_F \, \|\boldsymbol{x} - \widehat{\boldsymbol{x}}\|_2 + \|\boldsymbol{s} - \widehat{\boldsymbol{s}}\|_2 \\
&\leq \|M^T M + I\|_F \, \epsilon + \epsilon \\
&\leq \left(\|M^T\|_2 \, \|M\|_2 + \|I\|_2\right) \epsilon + \epsilon \\
&= (2n + \sqrt{n} + 1)\, \epsilon && (2.11)
\end{aligned}$$

Therefore, if we perform Algorithm 4 with inputs $\widehat{\boldsymbol{x}}$ and $\widehat{\boldsymbol{s}}$, combining the inequalities (2.9), (2.10), and (2.11), the generated mean vector $\boldsymbol{v}(\widehat{\boldsymbol{p}})$ can be shown to

be close to the mean vector $\boldsymbol{v}$ associated with the exact projection:

$$\begin{aligned}
\|\boldsymbol{v} - \boldsymbol{v}(\widehat{\boldsymbol{p}})\|_2 &\leq \|\boldsymbol{v} - \widetilde{\boldsymbol{v}}\|_2 + \|\widetilde{\boldsymbol{v}} - \boldsymbol{v}(\widehat{\boldsymbol{p}})\|_2 \\
&\leq (\sqrt{2\,n})\,\epsilon + \sqrt{2\,n\,m}\,(2n + \sqrt{n} + 1)\,\epsilon \\
&= \sqrt{2\,n}\,(1 + \sqrt{m}\,(2n + \sqrt{n} + 1))\,\epsilon
\end{aligned}$$

Now we can compute the total expected loss using approximate projection:

$$\begin{aligned}
\left|\sum_{t=1}^{T} \boldsymbol{v}^{t-1}(\widehat{\boldsymbol{p}}) \cdot \boldsymbol{\ell}^t\right| &= \left|\sum_{t=1}^{T} \left(\boldsymbol{v}^{t-1} + (\boldsymbol{v}^{t-1} - \boldsymbol{v}^{t-1}(\widehat{\boldsymbol{p}}))\right) \cdot \boldsymbol{\ell}^t\right| \\
&= \left|\sum_{t=1}^{T} \boldsymbol{v}^{t-1} \cdot \boldsymbol{\ell}^t + \sum_{t=1}^{T} (\boldsymbol{v}^{t-1} - \boldsymbol{v}^{t-1}(\widehat{\boldsymbol{p}})) \cdot \boldsymbol{\ell}^t\right| \\
&\leq \left|\sum_{t=1}^{T} \boldsymbol{v}^{t-1} \cdot \boldsymbol{\ell}^t\right| + \left|\sum_{t=1}^{T} (\boldsymbol{v}^{t-1} - \boldsymbol{v}^{t-1}(\widehat{\boldsymbol{p}})) \cdot \boldsymbol{\ell}^t\right| \\
&\leq \left|\sum_{t=1}^{T} \boldsymbol{v}^{t-1} \cdot \boldsymbol{\ell}^t\right| + \sum_{t=1}^{T} \|\boldsymbol{v}^{t-1} - \boldsymbol{v}^{t-1}(\widehat{\boldsymbol{p}})\|_2 \, \|\boldsymbol{\ell}^t\|_2 \\
&\leq \left|\sum_{t=1}^{T} \boldsymbol{v}^{t-1} \cdot \boldsymbol{\ell}^t\right| + T\,\left(\sqrt{2\,n}\,(1 + \sqrt{m}\,(2n + \sqrt{n} + 1))\,\epsilon\right)\,\sqrt{n}
\end{aligned}$$

For Huffman trees, the linear maps introduced in Pashkovich [2012] have this property that $\|F(\boldsymbol{a}) - F(\boldsymbol{a}')\|_2 \leq \|\boldsymbol{a} - \boldsymbol{a}'\|_2$ for all vectors $\boldsymbol{a}$ and $\boldsymbol{a}'$ where $F(\cdot)$ is the linear map. Using this property, it is straightforward to observe that this analysis can be extended for Huffman trees in which these linear maps are also used along with the reflection relations. Setting $\epsilon = \frac{1}{6n^2\sqrt{mT}}$, we add at most one unit to the expected cumulative loss with exact projections.

2.5.3 Running Time

The projection step dominates the running time of the XF-Hedge algorithm. Projecting onto any of the $r = m + n$ facets reduces to finding the sole non-negative (real) zero of a univariate polynomial of degree at most n so Newton's method takes $\mathcal{O}(n \log\log(1/\epsilon_1))$ time to get an ϵ_1-close solution. With $m + n$ constraints, each cycle through the constraints takes $\mathcal{O}(m\, n \log\log(1/\epsilon))$ time. Letting C_ϵ be the number of cycles to have an ϵ-accurate projection, the whole projection step takes $\mathcal{O}(C_\epsilon\, mn \log\log(1/\epsilon_1))$ time. These kinds of cyclic Bregman projections are believed to have fast linear convergence [Dhillon and Tropp, 2007], and empirically are very efficient [Koolen et al., 2010]. Note that exact convergence is not essential. For example, if the projection step estimates each $\boldsymbol{w}_t$ within $\epsilon = 1/6n^2\sqrt{m}T$ then the additional loss over the entire sequence of T trials is less than 1 unit as shown Section 2.5.2. Therefore, with the linear convergence assumption, the projection step takes $\mathcal{O}(mn \log(nmT) \log\log(1/\epsilon_1))$ time.

2.6 Conclusion and Future Work

Table 2.1 contains a comparison of the regret bounds for the new XF-Hedge algorithm, OnlineRank [Ailon, 2014], Follow the Perturbed Leader (FPL) [Kalai and Vempala, 2005], and the Hedge algorithm [Freund and Schapire, 1997] which inefficiently maintains an explicit weight for each of the exponentially many permutations or Huffman trees. For permutations, the regret bound of general XF-Hedge methodology is within a factor $\sqrt{\log n}$ of the state-of-the-art algorithm OnlineRank [Ailon, 2014].

Algorithm	Efficient?	Permutation	Huffman Tree	
			$\ell^t \in$ Unit Cube	$\ell^t \in$ Unit Simplex
XF-Hedge	**Yes**	$\mathcal{O}(n(\log n)^{\frac{1}{2}}\sqrt{L^*})$	$\mathcal{O}(n(\log n)^{\frac{1}{2}}\sqrt{L^*})$ ***Best***	$\mathcal{O}(n(\log n)^{\frac{1}{2}}\sqrt{L^*})$ ***Best***
OnlineRank	**Yes**	$\mathcal{O}(n\sqrt{L^*})$ ***Best***	–	–
FPL	**Yes**	$\mathcal{O}(n^{\frac{3}{2}}(\log n)^{\frac{1}{2}}\sqrt{L^*})$	$\mathcal{O}(n^{\frac{3}{2}}(\log n)^{\frac{1}{2}}\sqrt{L^*})$	$\mathcal{O}(n(\log n)^{\frac{1}{2}}\sqrt{L^*})$ ***Best***
Hedge Algorithm	**No**	$\mathcal{O}(n^{\frac{3}{2}}(\log n)^{\frac{1}{2}}\sqrt{L^*})$	$\mathcal{O}(n^{\frac{3}{2}}(\log n)^{\frac{1}{2}}\sqrt{L^*})$	$\mathcal{O}(n(\log n)^{\frac{1}{2}}\sqrt{L^*})$ ***Best***

Table 2.1: Comparing the regret bounds of XF-Hedge with other existing algorithms in different problems and different loss regimes.

When compared with the generic explicit Hedge algorithm (which is not computationally efficient) and FPL, XF-Hedge has a better loss bound by a factor of $\sqrt{n}$.

When comparing on Huffman trees, we consider two loss regimes: one where the loss vectors are from the general unit cube, and consequently, the per-trial losses are in $\mathcal{O}(n^2)$ (like permutations), and another where the loss vectors represent frequencies and lie on the unit simplex so the per-trial losses are in $\mathcal{O}(n)$. In the first case, as with permutations, XF-Hedge has the best asymptotic bounds. In the second case, the lower loss range benefits Hedge and FPL, and the regret bounds of all three algorithms match.

In conclusion, we have presented a general methodology for creating online learning algorithms from extended formulations. Our main contribution is the XF-Hedge algorithm that enables the efficient use of Component Hedge techniques on complex classes of combinatorial objects. Because XF-Hedge is in the Bregman projection family of algorithms, many of the tools from the expert setting are likely to carry over.

This includes lower bounding weights for shifting comparators [Herbster and Warmuth, 1998], long-term memory [Bousquet and Warmuth, 2002], and adapting the updates to the bandit setting [Audibert et al., 2011]. Several important areas remain for potentially fruitful future work:

More Applications. There is a rich literature on extended formulation for different combinatorial objects [Conforti et al., 2010, Kaibel, 2011, Pashkovich, 2012, Afshari Rad and Kakhki, 2017, Fiorini et al., 2013]. Which combinatorial classes have both natural online losses and suitable extended formulations so XF-Hedge is appropriate? For instance, building on the underlying ideas of XF-Hedge, Chapter 3 develops a family of learning algorithms focusing on extended formulations constructed by dynamic programming.

More Complex Losses. The redundant representation we introduced can be used to express different losses. Although our current applications do not assign loss to the extended formulation variables $\boldsymbol{x}$ and their associated slack variables $\boldsymbol{s}$, these additional variables enable the expression of different kinds of losses. For what natural losses could be these additional variables useful?

Chapter 3

Online Dynamic Programming

In this chapter we focus on online learning of a large family of combinatorial objects. We consider the combinatorial objects whose offline optimizations can be formulated as a dynamic programming problem with min-sum recurrence relations. A typical example of such combinatorial problem is to learn paths in a given fixed directed acyclic graph (DAG) with designated source and sink nodes. The loss of each path from the source to the sink is *additive*, that is, it is the sum of the losses of the edges along that path. For the explanation purposes, we start with path learning problem. This will be generalized later.

We view the problem of finding the path with minimum loss in a given DAG $G = (V, E)$ as a rudimentary dynamic programming problem. For every node $v \in V$ in the given DAG, let $\mathrm{OPT}(v)$ be the loss of the best path from v to the sink. The following min-sum recurrence holds for OPT:

$$\mathrm{OPT}(v) = \min_{u:(v,u)\in E} \{\mathrm{OPT}(u) + \ell_{(v,u)}\}$$

where ℓ_e is the loss of the edge $e \in E$.

Online learning of paths in G proceeds in a series of trial. In each trial, the learner predicts with a path in G. Then, the adversary reveals the losses of all the edges in E. Finally, the learner incurs the loss of its predicted path. The goal is to minimize *regret* which is the total loss of the learner minus the total loss of the single best path in hindsight.

A natural solution is to use one of the well-known so-called "expert algorithms" like *Randomized Weighted Majority* [Littlestone and Warmuth, 1994] or *Hedge* [Freund and Schapire, 1997] with one weight per path (i.e. "expert"). Maintaining one weight per path, however, is impractical since it requires keeping track of a weight vector of exponential size. Exploiting the additivity of the loss, Takimoto and Warmuth [2003] introduced the *Expanded Hedge (EH)* algorithm which is an efficient implementation of the Hedge algorithm in the path learning problem. EH assigns weights to the edges and maintains a distribution over paths where the probability of each path is proportionate to the product of the weights of the edges along that path. Another efficient algorithm for learning paths is the *Component Hedge (CH)* algorithm of Koolen et al. [2010] which is a generic algorithm for learning combinatorial objects with additive losses. Instead of a distribution, CH maintains a mean vector over the paths and assigns *flows* to the edges. This mean vector lives in the *unit-flow polytope* which is the convex hull of all paths in the graph. Comparing to EH, CH guarantees better regret bounds as it does not have additional loss range factors in its bounds.

Going beyond paths, we generalize the online learning problem to any combina-

torial object which can be "recognized" by a min-sum dynamic programming problem. Similar to the paths, the online learning in a given dynamic programming problem proceeds in a series of trials. In each trial, the learner makes a prediction with a solution in the appropriate combinatorial space. Then, it receives the loss of its choice in such a way that the loss of any of the possible solution can be easily computed. Despite batch learning settings, there is no assumed distribution from which the losses are randomly drawn. Instead the losses are drawn in adversarial fashion. The goal is to minimize *regret* which is the total loss of the learner minus the total loss of the single best solution in hindsight. Therefore the regret of the learner can be interpreted as the cost of not knowing the best solution ahead of time. With proper tuning, the regret is typically logarithmic in the number of solutions.

To make the setting more concrete, consider the case of learning the best Binary Search Tree (BST) for a given set of n keys [Cormen et al., 2009]. In each trial, the learner plays with a BST. Then the adversary reveals a set of probabilities for the n keys and the learner incurs a linear loss of *average search cost*, which is simply the dot product between the vector of probabilities and the vector of depth values of the keys in the tree. The regret of the learner is the difference between its total loss and the sum over trials of the average search cost for the single best BST chosen in hindsight.

The number of solutions is typically exponential in n where n is the number of components in the structure of the solutions. In a BST, for instance, the components are the depth values of the n keys in the tree, and the number of possible BSTs is the nth Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$ [Cormen et al., 2009]. Thus, a naive implementation

of Randomized Weighted Majority or Hedge (i.e maintaining one weight per solution) is impractical since it requires keeping track of a weight vector of exponential size.

Moreover, the results of CH (and its current extensions [Suehiro et al., 2012, Rajkumar and Agarwal, 2014, Gupta et al., 2016]) cannot be directly applied to problems like BST. CH maintains a mean vector of the BSTs which lives in the convex hull of all BSTs with the representation above. This polytope does not have a characterization with polynomially many facets [1].

We close this gap by exploiting the dynamic programming algorithm which solves the BST offline optimization problem. This gives us a polytope with a polynomial number of facets while the loss is linear in the natural components of the BST problem. This well-behaved polytope allows us to implement CH efficiently. Moreover, the dynamic programming algorithm provides a representation with which EH can be efficiently implemented.

Contributions. We propose a general method for learning combinatorial objects whose offline optimization problem can be solved efficiently via a dynamic programming algorithm with an arbitrary min-sum recurrence relation. Examples include BST, Matrix-Chain Multiplication, Knapsack, k-sets, Rod Cutting, and Weighted Interval Scheduling. Using the underlying graph of subproblems (called *multi-DAG*) induced by the dynamic programming algorithm for these problems, we define a representation of

[1] There is an alternate polytope for BSTs with a polynomial number of facets (called the *associahedron* [Loday, 2005]) but the average search cost is not linear in terms of the components used for this polytope. CH and its extensions, however, rely heavily on the additivity of the loss over the components. Thus they cannot be applied to the associahedron.

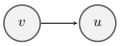
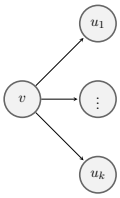
Graph with the set of vertices V	$\Longrightarrow$	**Multigraph** with the set of vertices V
Edge (v, u) $u, v \in V$	$\Longrightarrow$	**Multiedge** (v, U) $v \in V,\ U \subset V$
		
Path	$\Longrightarrow$	**Multipath**
DAG	$\Longrightarrow$	**Multi-DAG**

Table 3.1: From graphs to multi-graphs

the combinatorial objects by encoding them as a specific type of subgraphs called *multipaths*. These multipaths encode each object as a series of successive decisions (i.e. the components) over which the loss is linear, even though the loss may not be linear in the original representation (e.g. Matrix-Chain Multiplication). We show that optimizing a specific dynamic programming problem from this class over trials reduces to online learning of multipaths.

We generalize the definition of edge as an ordered pair (v, u) of vertices to *multiedge* which is an ordered pair (v, U) where the first element v is a vertex and the second element U is a *subset of the vertices*. Following from this generalization, we extend the definitions of paths, graphs and directed acyclic graphs (DAG) accordingly (see Table 3.1; Section 3.2 contains the formal definitions). These extensions allow us to generalize the existing EH [Takimoto and Warmuth, 2003] and CH [Koolen et al., 2010] algorithms from online shortest path problem to learning multipaths. Moreover, we also introduce a new and faster prediction technique for CH for multipaths which

directly samples from an appropriate distribution, bypassing the need to create convex combinations.

Chapter Outline. We start with online learning of paths in a directed graph in Section 3.1 and give an overview on existing algorithms: Expanded Hedge and Component Hedge. Section 3.2 introduces multipaths and generalizes the path learning algorithms to multipaths. In Section 3.3, we define a class of combinatorial objects recognized by dynamic programming algorithms. Then we prove that minimizing a specific dynamic programming problem from this class over trials reduces to online learning of multipaths. In Section 3.4, we apply our methods to several dynamic programming problems. Finally, Section 3.5 concludes with comparison to other algorithms and future work.

3.1 Background

One of the core combinatorial online learning problems is learning a minimum loss path in a directed acyclic graph (DAG). The online shortest path problem has been explored both in the full information setting [Takimoto and Warmuth, 2003, Koolen et al., 2010, Cortes et al., 2015a] and various bandit settings [György et al., 2007, Audibert et al., 2013, Awerbuch and Kleinberg, 2008, Dani et al., 2008, Cortes et al., 2018]. In the full information setting, the problem is as follows. A DAG $\mathcal{G} = (V, E)$ is given along with a designated source node $s \in V$ and sink node $t \in V$. In each trial, the algorithm predicts with a path from s to t. Then for each edge $e \in E$, the adversary reveals a loss $\ell_e \in [0, 1]$. The loss of the algorithm is given by the sum of the losses of

the edges (components) along the predicted path. The goal is to minimize the regret which is the difference between the total loss of the algorithm and that of the single best path chosen in hindsight. In the remainder of this section, we give a few remarks on EH and provide an overview of CH as the the two main algorithms for online path learning in full information setting.

3.1.1 Expanded Hedge on Paths

Takimoto and Warmuth [2003] introduced an efficient implementation of EH by exploiting the additivity of the loss over the edges of a path. See Section 1.3 for an overview of EH in path learning. EH provides the regret guarantees below.

Theorem 8 (Takimoto and Warmuth [2003]). *Given a DAG $\mathcal{G} = (V, E)$ with designated source node $s \in V$ and sink node $t \in V$, assume $\mathcal{N}$ is the number of paths in $\mathcal{G}$ from s to t, L^* is the total loss of best path, and D is an upper-bound on the number of edges of the paths in $\mathcal{G}$ from s to t. Then with proper tuning of the learning rate η over the trials, EH guarantees the following regret bound:*

$$\mathcal{R}_{EH} \leq \sqrt{2\, L^*\, D\, \log \mathcal{N}} + D\, \log \mathcal{N}. \tag{3.1}$$

3.1.2 Component Hedge on Paths

The generic Component Hedge algorithm of Koolen et al. [2010] can be applied to the online shortest path problem. The components are the edges E in the DAG. Each path is encoded as a bit vector $\boldsymbol{\pi}$ of $|E|$ components where the 1-bits indicate the presence of the edges in the path $\boldsymbol{\pi}$. The convex hull of all paths is called the

unit-flow polytope and CH maintains a mixture vector $\boldsymbol{f} = [f_e]_{e \in E}$ in this polytope. The constraints of the polytope enforce an outflow of 1 from the source node s, and flow conservation at every other node but the sink node t. In each trial, each edge e receives a loss of ℓ_e and the weight of that edge f_e is updated multiplicatively by the factor $\exp(-\eta \ell_e)$. Then the weight vector is projected back to the unit-flow polytope via a relative entropy projection. This projection is achieved by iteratively projecting onto the flow constraint of a particular vertex and then repeatedly cycling through the vertices [Bregman, 1967]. Finally, to sample with the same expectation as the mixture vector in the polytope, this vector is decomposed into paths using a greedy approach which removes one path at a time and zeros out at least one edge in the remaining mixture vector in each iteration. CH provides the regret guarantees below.

Theorem 9 (Koolen et al. [2010])**.** *Given a DAG $\mathcal{G} = (V, E)$ with designated source node $s \in V$ and sink node $t \in V$, let D be an upper-bound on the number of edges of the paths in $\mathcal{G}$ from s to t. Also denote the total loss of the best path by L^*. Then with proper tuning of the learning rate η over the trials, CH guarantees the following regret bound:*

$$\mathcal{R}_{CH} \leq \sqrt{4\, L^*\, D\, \log |V|} + 2\, D \log |V|. \tag{3.2}$$

Remark. In a moment, we will compare the regret bounds of EH (3.1) and CH (3.2). We will observe that compared to EH, CH guarantees better regret bounds as it does not have additional loss range factors in its bounds. In fact, the regret bounds of CH are typically optimal. Koolen et al. [2010] prove lower bounds which matches the guar-

antees of CH for various combinatorial objects such as k-sets and permutations. The lower bounds are shown by embedding the combinatorial online learning into the original expert problem. To form the experts, a set of cominatorial objects is chosen which partitions all of the components. Moreover, all objects in the set must have the same number of present components (i.e. same number of ones in the bit-vector representation). Given this proof technique, a lower bound on the regret for arbitrary graphs is difficult to obtain since choosing a set of paths with the aforementioned characteristics is non-trivial. Perhaps the regret of CH is tight within constant factors for all graphs, but this question is still open.

3.1.3 Component Hedge vs Expanded Hedge

To have a concrete comparison between CH and EH on paths, consider the following path learning setting. Let $\mathcal{G} = (V, E)$ be a complete DAG with $V = \{v_1, \ldots, v_n\}$ where for all $1 \leq i < j \leq n$, v_i is connected to v_j. Also let $s = v_1$ and $t = v_n$ be the designated source and sink nodes, respectively. Note that the number of edges in any path in $\mathcal{G}$ from s to t is at most $D = n - 1$. Also the total number of paths in $\mathcal{G}$ is $\mathcal{N} = 2^{n-2}$. The corollary below shows the superiority of the performance of CH over EH in terms of regret bound which is a direct result of Theorems 8 and 9. EH offers worse regret guarantee as its bound has an additional loss range factor.

Corollary 10. *Given a complete DAG $\mathcal{G}$ with n nodes, let L^* be the total loss of the best path. Then with proper tuning of the learning rate over the trials for both EH and*

CH, we obtain the following regret guarantees:

$$\mathcal{R}_{EH} = \mathcal{O}(n\sqrt{L^*}), \qquad \mathcal{R}_{CH} = \mathcal{O}(n^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\sqrt{L^*}).$$

Remark. For EH, projections are simply a renormalization of the path weights which is done efficiently via weight pushing [Mohri, 2009b, Takimoto and Warmuth, 2003]. On the other hand, for CH, iterative Bregman projections [Bregman, 1967] are needed for projecting back into the unit-flow [Koolen et al., 2010]. These methods are known to converge to the exact projection [Bregman, 1967, Bauschke and Borwein, 1997]; however, there will always be a gap to the full convergence. These remaining gaps to the exact projections have to be accounted for as additional loss in the regret bounds (e.g. see Section 2.5.2). Additionally, the relatively expensive projection operation in CH makes it less computationally efficient compared to EH.

3.2 Learning Multipaths

We begin with defining directed multigraphs, multiedges[2] and multi-DAGs.

Definition 1 (Directed Multigraph)**.** *A* ***directed multigraph*** *is an ordered pair* $\mathcal{H} = (V, M)$ *comprising of a set* V *of vertices or nodes together with a set* M *of* ***multiedges****. Each multiedge* $m \in M$ *is an ordered pair* $m = (v, U)$ *where* $v \in V$ *and* $U \subseteq V$*. Furthermore, we denote the set of "outgoing" and "incoming" multiedges for vertex* v

[2] Our definitions of multigraphs and multiedges are closely related to *hyper-graphs* and *hyper-arcs* in the literature (see e.g. Martin et al. [1990]).

by $M_v^{(out)}$ *and* $M_v^{(in)}$*, respectively, which are defined as*

$$M_v^{(out)} := \{m \in M \mid m = (v, U) \text{ for some } U \subseteq V\},$$

$$M_v^{(in)} := \{m \in M \mid m = (u, U) \text{ for some } u \in V \text{ and } U \subseteq V \text{ s.t. } v \in U\}.$$

Definition 2 (Base Directed Graph)**.** *The* ***base directed graph*** *of a given directed multigraph* $\mathcal{H} = (V, M)$ *is a directed graph* $\mathcal{B}(\mathcal{H}) = (V, E)$ *where*

$$E = \{(v, u) \mid \exists (v, U) \in M \text{ s.t. } u \in U\}.$$

Definition 3 (Multi-DAG)**.** *A directed multigraph* $\mathcal{H} = (V, M)$ *is a* ***multi-DAG*** *if it has a single "source" node* $s \in V$ *with no incoming multiedges and its base directed graph* $\mathcal{B}(\mathcal{H})$ *is acyclic. Additionally, we refer to the set of nodes in* V *with no outgoing multiedges as the set of "sink" nodes which is denoted by* $\mathcal{T} \subset V$*.*

Intuitively speaking, a multi-DAG is simply a directed multigraph with no "cycles". "Acyclicity" in directed multigraphs can be extended from the definition of acyclicity in the underlying directed graph.

Each multipath in a multi-DAG $\mathcal{H} = (V, M)$ can be generated by starting with a single multiedge at the source, and then choosing inflow many (i.e. as many as the number of incoming edges of the multipath in $\mathcal{B}(\mathcal{H})$) successor multiedges at the internal nodes until we reach the sink nodes in $\mathcal{T}$. An example of a multipath is given in Figure 3.1. Recall that paths were described as bit vectors $\boldsymbol{\pi}$ of size $|E|$ where the 1-bits were the edges in the path. In multipaths, however, each multiedge $m \in M$ is associated with a non-negative count π_m that can be greater than 1.

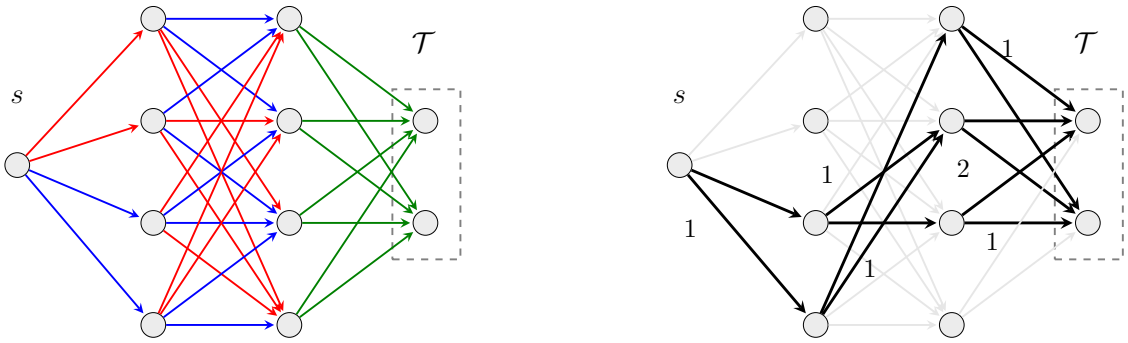


Figure 3.1: On the left we give an example of a multi-DAG. The source s and the nodes in the first layer each have two multiedges depicted in red and blue. The nodes in the next layer each have one multiedge depicted in green. An example of multipath in the multi-DAG is given on the right. The multipath is represented as an $|M|$-dimensional count vector $\boldsymbol{\pi}$. The grayed multiedges are the ones with count $\pi_e = 0$. All non-zero counts π_m are shown next to their associated multiedges m.

Definition 4 (Multipath)**.** *Given a multi-DAG $\mathcal{H} = (V, M)$, let*[3] $\boldsymbol{\pi} \in \mathbb{N}^{|M|}$ *in which π_m is associated with the multiedge $m \in M$. For every vertex $v \in V$, define the inflow $\pi_{in}(v) := \sum_{m \in M_v^{(in)}} \pi_m$ and the outflow $\pi_{out}(v) := \sum_{m \in M_v^{(out)}} \pi_m$. We call $\boldsymbol{\pi}$ a* ***multipath*** *if it has the properties below:*

1. *The outflow $\pi_{out}(s)$ of the source s is 1.*

2. *For each vertex $v \in V - \mathcal{T} - \{s\}$, the outflow is equal to the inflow, i.e. $\pi_{out}(v) = \pi_{in}(v)$.*

[3] $\mathbb{N}$ is the set of non-negative integers.

Multipath Learning Problem. Having established all definitions for multipaths, we shall now define the problem of *online learning of multipaths* on a given multi-DAG $\mathcal{H} = (V, M)$ as follows. In each trial, the algorithm randomly predicts with a multipath $\boldsymbol{\pi}$. Then for each multiedge $m \in M$, the adversary reveals a loss $\ell_m \in [0, 1]$ incurred during that trial. The linear loss of the algorithm during this trial is given by $\boldsymbol{\pi} \cdot \boldsymbol{\ell}$. Observe that the online shortest path problem is a special case when $|\mathcal{T}| = 1$ and $|U| = 1$ for all multiedges $(v, U) \in M$.

In the remainder of this section, we generalize the algorithms in Section 3.1 to the online learning problem of multipaths. Moreover, we also introduce a faster prediction technique for CH.

3.2.1 Expanded Hedge on Multipaths

We implement EH efficiently for learning multipaths by considering each multipath as an expert. Recall that each multipath can be generated by starting with a single multiedge at the source and choosing inflow many successor multiedges at the internal nodes. Multipaths are composed of multiedges as components and with each multiedge $m \in M$, we associate a weight w_m. We maintain a distribution W over multipaths defined in terms of the weights $\boldsymbol{w} \in \mathbb{R}_{\geq 0}^{|M|}$ on the multiedges. The distribution W will be in *stochastic product form* which is defied as below.

Definition 5 (Stochastic Product Form)**.** *The distribution W over the multipaths is in* ***stochastic product form*** *in terms of the weights $\boldsymbol{w}$ if it has the following properties:*

1. *The weights are in* ***product form****, i.e.* $W(\boldsymbol{\pi}) = \prod_{m \in M} (w_m)^{\pi_m}$.

2. *The weights are* ***stochastic****, i.e.* $\sum_{m \in M_v^{(out)}} w_m = 1$ *for all* $v \in V - \mathcal{T}$.

3. *The total path weight is one*[4] *, i.e.* $\sum_{\boldsymbol{\pi}} W(\boldsymbol{\pi}) = 1$.

Using these properties, sampling a multipath from W can be easily done as follows. We start with sampling a single multiedge at the source and continue sampling inflow many successor multiedges at the internal nodes until the multipath reaches the sink nodes in $\mathcal{T}$. Observe that π_m indicates the number of times the multiedge m is sampled through this process. EH updates the weights of the multipaths as follows:

$$\begin{aligned} W^{\text{new}}(\boldsymbol{\pi}) &= \frac{1}{Z} W(\boldsymbol{\pi}) \, \exp(-\eta \, \boldsymbol{\pi} \cdot \boldsymbol{\ell}) \\ &= \frac{1}{Z} \left(\prod_{m \in M} (w_m)^{\pi_m} \right) \exp\left[-\eta \sum_{m \in M} \pi_m \, \ell_m \right] \\ &= \frac{1}{Z} \prod_{m \in M} \Big(\underbrace{w_m \, \exp\big[-\eta \, \ell_m \big]}_{:= \widehat{w}_m} \Big)^{\pi_m}. \end{aligned}$$

Thus the weights w_m of each multiedge $m \in M$ are updated multiplicatively to $\widehat{w}_m$ by multiplying the w_m with the exponentiated loss factors $\exp[-\eta \, \ell_m]$ and then renormalizing with Z.

Generalized Weight Pushing. We generalize the *weight pushing algorithm* of Mohri [2009b] to multipaths to reestablish the three canonical properties of Definition 5. Observe that for every multiedge $m \in M$, $\widehat{w}_m = w_m \exp(-\eta \ell_m)$. The new weights are $W^{\text{new}}(\boldsymbol{\pi}) = \frac{1}{Z} \widehat{W}(\boldsymbol{\pi})$ where $\widehat{W}(\boldsymbol{\pi}) := \prod_{m \in M} (\widehat{w}_m)^{\pi_m}$. The generalized weight pushing algorithm takes a set of arbitrary weights on the multiedges $\widehat{w}_m$ and changed them into Stochastic Product Form.

[4] The third property is implied by the first two properties. Nevertheless, it is mentioned for the sake of clarity.

Note that the new weights $W^{\text{new}}(\boldsymbol{\pi}) = \frac{1}{Z}\widehat{W}(\boldsymbol{\pi})$ sum to 1 (i.e. Property (3) holds) since Z normalizes the weights. Our goal is to find new multiedge weights w_m^{new} so that the other two properties hold as well, i.e. $W^{\text{new}}(\boldsymbol{\pi}) = \prod_{m\in M}(w_m^{\text{new}})^{\pi_m}$ for all multipaths $\boldsymbol{\pi}$ and $\sum_{m\in M_v^{(\text{out})}} w_m^{\text{new}} = 1$ for all nonsinks v. For this purpose, we introduce a normalization Z_v for each vertex $v \in V$:

$$Z_v := \sum_{\boldsymbol{\pi}\in\mathcal{P}_v} \widehat{W}(\boldsymbol{\pi}). \tag{3.3}$$

where $\mathcal{P}_v$ is the set of all multipaths sourced from v and sinking at $\mathcal{T}$. Intuitively, Z_v is the normalization constant for the subgraph sourced at $v \in V$ and sinking in $\mathcal{T}$. Thus for a sink node $v \in \mathcal{T}$, $Z_v = 1$. Moreover $Z = Z_s$ is the normalization factor for the multi-DAG $\mathcal{H}$ where $s \in V$ is the source node. The *generalized weight pushing* finds all Z_v's recursively starting from the sinks and then it computes the new weights w_m^{new} for the multiedges to be used in the next trial:

1. For sinks $v \in \mathcal{T}$, $Z_v = 1$.

2. Recursing backwards in the multi-DAG, let $Z_v = \sum_{m\in M_v^{(\text{out})}} \widehat{w}_m \prod_{u\in U:m=(v,U)} Z_u$ for all non-sinks v.

3. For each multiedge $m = (v, U)$, $w_m^{\text{new}} := \widehat{w}_m \left(\prod_{u\in U} Z_u\right)/Z_v$.

Figure 3.2 illustrates an example of the weight pushing algorithm. For simplicity, we demonstrate this algorithm on a regular DAG, that is, a multi-DAG where $|U| = 1$ for all multiedges $(v, U) \in M$. The DAG on the left shows the unnormalized

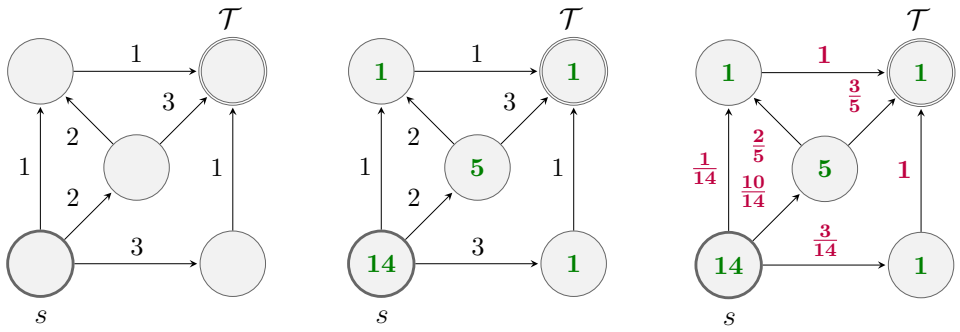


Figure 3.2: Example of weight pushing for regular DAGs i.e. when $|U| = 1$ for all multiedges $(v, U) \in M$. (Left) the unnormalized weights $\widehat{w}_m$ for all multiedges/edges m in the DAG. (Middle) the normalizations Z_v for all vertices $v \in V$ using the Steps 1 and 2 of the weight pushing algorithm. (Right) the new weights w_m^{new} which are in Stochastic Product Form using the Step 3 of the weight pushing algorithm.

weights $\widehat{w}_m$ for all multiedges/edges m in the DAG. In the DAG in the middle, we compute all the normalizations Z_v for all vertices $v \in V$ using Steps 1 and 2 of the weight pushing algorithm. Finally, in the DAG on the right, we find the new weights w_m^{new} which are in Stochastic Product Form using Step 3 of the weight pushing algorithm. Lemma below proves the correctness and time complexity of this generalized weight pushing algorithm.

Lemma 11. *The weights w_m^{new} generated by the generalized weight pushing are in Stochastic Product Form (see Definition 5) and for all multipaths $\boldsymbol{\pi}$, $\prod_{m \in M} (w_m^{new})^{\pi_m} = \frac{1}{Z_s} \prod_{m \in M} (\widehat{w}_m)^{\pi_m}$. Moreover, the weights w_m^{new} can be computed in $\mathcal{O}(c\,|M|)$ time where c is an upper-bound on the branching factor of each multiedge (i.e. for all $m = (v, U) \in M$, $|U| < c$).*

Proof. First, we show that the recursive relation in Step 2 and the initialization in Step 1 hold for Z_v defined in Equation (3.3). For a sink node $v \in \mathcal{T}$, the normalization constant Z_v is vacuously 1. Thus Step 1 is justified. To prove the recursive relation in Step 2, consider any non-sink $v \in V - \mathcal{T}$. We "peel off" the first multiedge leaving v and then recurse:

$$Z_v = \sum_{\boldsymbol{\pi} \in \mathcal{P}_v} \widehat{W}(\boldsymbol{\pi}) = \sum_{m \in M_v^{(\text{out})}} \sum_{\substack{\boldsymbol{\pi} \in \mathcal{P}_v \\ \text{starts with } m}} \widehat{W}(\boldsymbol{\pi}).$$

Recall that $\widehat{W}(\boldsymbol{\pi}) = \prod_{m' \in M} (\widehat{w}_{m'})^{\pi_{m'}}$. Thus, we can factor out the weight $\widehat{w}_m$ associated with multiedge $m \in M_v^{(\text{out})}$. Assume the multiedge m comprised of edges from the node v to the nodes $u_1, \ldots, u_k$. Notice, excluding m from the multipath, we are left with k number of multipaths from the u_i's:

$$\begin{aligned} Z_v &= \sum_{m \in M_v^{(\text{out})}} \sum_{\substack{\boldsymbol{\pi} \in \mathcal{P}_v \\ \text{starts with } m}} \widehat{W}(\boldsymbol{\pi}) \\ &= \sum_{m \in M_v^{(\text{out})}} \widehat{w}_m \sum_{\boldsymbol{\pi}_1 \in \mathcal{P}_{u_1}} \sum_{\boldsymbol{\pi}_2 \in \mathcal{P}_{u_2}} \cdots \sum_{\boldsymbol{\pi}_k \in \mathcal{P}_{u_k}} \prod_{i=1}^{k} \widehat{W}(\boldsymbol{\pi}_i). \end{aligned}$$

After factoring $\widehat{w}_m$ out, the sum $\sum_{\boldsymbol{\pi}_1 \in \mathcal{P}_{u_1}} \sum_{\boldsymbol{\pi}_2 \in \mathcal{P}_{u_2}} \cdots \sum_{\boldsymbol{\pi}_k \in \mathcal{P}_{u_k}}$ iterates over all combinations of all multipaths sourced from all u_i's associated with the multiedge m. Recall that $\mathcal{P}_{u_i}$ is the set of all multipaths sourced from u_i and sinking at $\mathcal{T}$. Since each $\boldsymbol{\pi}_i$ iterates over all multipaths in $\mathcal{P}_{u_i}$, we can turn the sum of products into product of

sums as below:

$$
\begin{aligned}
Z_v &= \sum_{m \in M_v^{(\text{out})}} \widehat{w}_m \sum_{\boldsymbol{\pi}_1 \in \mathcal{P}_{u_1}} \sum_{\boldsymbol{\pi}_2 \in \mathcal{P}_{u_2}} \cdots \sum_{\boldsymbol{\pi}_k \in \mathcal{P}_{u_k}} \prod_{i=1}^{k} \widehat{W}(\boldsymbol{\pi}_i) \\
&= \sum_{m \in M_v^{(\text{out})}} \widehat{w}_m \sum_{\boldsymbol{\pi}_1 \in \mathcal{P}_{u_1}} \widehat{W}(\boldsymbol{\pi}_1) \underbrace{\sum_{\boldsymbol{\pi}_2 \in \mathcal{P}_{u_2}} \cdots \sum_{\boldsymbol{\pi}_k \in \mathcal{P}_{u_k}} \prod_{i=2}^{k} \widehat{W}(\boldsymbol{\pi}_i))}_{\text{does not depend on } \boldsymbol{\pi}_1 \in \mathcal{P}_{u_1}} \\
&= \sum_{m \in M_v^{(\text{out})}} \widehat{w}_m \left(\sum_{\boldsymbol{\pi}_2 \in \mathcal{P}_{u_2}} \cdots \sum_{\boldsymbol{\pi}_k \in \mathcal{P}_{u_k}} \prod_{i=2}^{k} \widehat{W}(\boldsymbol{\pi}_i) \right) \left(\sum_{\boldsymbol{\pi}_1 \in \mathcal{P}_{u_1}} \widehat{W}(\boldsymbol{\pi}_1) \right) \\
&= \quad \cdots \qquad \text{(Repeating for each sum } \sum_{\boldsymbol{\pi}_j \in \mathcal{P}_{u_j}} \text{)} \\
&= \sum_{m \in M_v^{(\text{out})}} \widehat{w}_m \prod_{i=1}^{k} \left(\underbrace{\sum_{\boldsymbol{\pi} \in \mathcal{P}_{u_i}} \widehat{W}(\boldsymbol{\pi})}_{Z_{u_i}} \right) \\
&= \sum_{m \in M_v^{(\text{out})}} \widehat{w}_m \prod_{i=1}^{k} Z_{u_i}. \qquad (3.4)
\end{aligned}
$$

Equation (3.4) justifies Step 2. Now we prove that the new weight assignment in Step 3 will result in Stochastic Product Form with correct distribution. For all $v \in V - \mathcal{T}$ and for all $m \in M_v^{(\text{out})}$, set $w_m^{\text{new}} := \widehat{w}_m \frac{\prod_{u:(v,u)\in m} Z_u}{Z_v}$ (Step 3). Property (2) of Definition 5 (stochasticity) is true since for all $v \in V - \mathcal{T}$:

$$
\begin{aligned}
\sum_{m \in M_v^{(\text{out})}} w_m^{\text{new}} &= \sum_{m \in M_v^{(\text{out})}} \widehat{w}_m \frac{\prod_{u:(v,u)\in m} Z_u}{Z_v} \\
&= \frac{1}{Z_v} \underbrace{\sum_{m \in M_v^{(\text{out})}} \widehat{w}_m \prod_{u:(v,u)\in m} Z_u}_{Z_v} = 1. \qquad \text{(Equation (3.4))}
\end{aligned}
$$

We now prove that Property (1) of Definition 5 (product form) is also true

since for all $\boldsymbol{\pi} \in \mathcal{P}_s$:

$$\begin{aligned}
\prod_{m \in M} (w_m^{\text{new}})^{\pi_m} &= \prod_{v \in V-\mathcal{T}} \prod_{m \in M_v^{(\text{out})}} (w_m^{\text{new}})^{\pi_m} \\
&= \prod_{v \in V-\mathcal{T}} \prod_{m \in M_v^{(\text{out})}} \left(\widehat{w}_m \frac{\prod_{u:(v,u) \in m} Z_u}{Z_v} \right)^{\pi_m} \\
&= \left[\prod_{v \in V-\mathcal{T}} \prod_{m \in M_v^{(\text{out})}} (\widehat{w}_m)^{\pi_m} \right] \left[\prod_{v \in V-\mathcal{T}} \prod_{m \in M_v^{(\text{out})}} \left(\frac{\prod_{u:(v,u) \in m} Z_u}{Z_v} \right)^{\pi_m} \right].
\end{aligned}$$

Notice that $\prod_{v \in V-\mathcal{T}} \prod_{m \in M_v^{(\text{out})}} \left(\frac{\prod_{u:(v,u) \in m} Z_u}{Z_v} \right)^{\pi_m}$ telescopes along the multiedges in the multipath $\boldsymbol{\pi}$. After telescoping, since $Z_v = 1$ for all $v \in \mathcal{T}$, the only remaining term will be $\frac{1}{Z_s}$ where s is the souce node. Therefore we obtain:

$$\begin{aligned}
\prod_{m \in M} (w_m^{\text{new}})^{\pi_m} &= \left[\prod_{v \in V-\mathcal{T}} \prod_{m \in M_v^{(\text{out})}} (\widehat{w}_m)^{\pi_m} \right] \left[\prod_{v \in V-\mathcal{T}} \prod_{m \in M_v} \left(\frac{\prod_{u:(v,u) \in m} Z_u}{Z_v} \right)^{\pi_m} \right] \\
&= \left[\prod_{m \in M} (\widehat{w}_m)^{\pi_m} \right] \left[\frac{1}{Z_s} \right] \\
&= \frac{1}{Z_s} \prod_{m \in M} (\widehat{w}_m)^{\pi_m} = W^{\text{new}}(\boldsymbol{\pi}).
\end{aligned}$$

Regarding the time complexity, we first focus on the the recurrence relation $Z_v = \sum_{m \in M_v} \widehat{w}_m \prod_{u:(v,u) \in m} Z_u$. Note that for each $v \in V$, Z_v can be computed in $\mathcal{O}(c\,|M_v^{(\text{out})}|)$. Thus the computation of all Z_v's takes $\mathcal{O}(c\,|M|)$ time. Now observe that w_m^{new} for each multiedge $m = (v, U) \in M$ can be found in $\mathcal{O}(c)$ time using $w_m^{\text{new}} = \widehat{w}_m \frac{\prod_{u \in U} Z_u}{Z_v}$. Hence the computation of w_m^{new} for all multiedges $m \in M$ takes $\mathcal{O}(c\,|M|)$ time. Therefore the generalized weight pushing algorithm runs in $\mathcal{O}(c\,|M|)$ time.

■

Regret Bound. In order to apply the regret bound of EH we have to initialize the distribution W on multipaths to the uniform distribution. This is achieved by setting all $\widehat{w}_m$ to 1 followed by an application of generalized weight pushing. Note that Theorem 8 is a special case of the theorem below when $|U| = 1$ for all multiedge $(v, U) \in M$ and $|\mathcal{T}| = 1$.

Theorem 12. *Given a multi-DAG $\mathcal{H} = (V, M)$ with designated source node $s \in V$ and sink nodes $\mathcal{T} \subset V$, assume $\mathcal{N}$ is the number of multipaths in $\mathcal{H}$ from s to $\mathcal{T}$, L^* is the total loss of best multipath, and D is an upper-bound on the 1-norm of the count vectors of the multipaths (i.e. $\|\boldsymbol{\pi}\|_1 \leq D$ for all multipaths $\boldsymbol{\pi}$). Then with proper tuning of the learning rate η over the trials, EH guarantees the following regret bound:*

$$\mathcal{R}_{EH} \leq \sqrt{2\, L^*\, D\, \log \mathcal{N}} + D\, \log \mathcal{N}.$$

3.2.2 Component Hedge on Multipaths

We implement CH efficiently for learning of multipaths in a multi-DAG $\mathcal{H} = (V, M)$. Here the multipaths are the objects which are represented as $|M|$-dimensional count vectors $\boldsymbol{\pi}$ (Definition 4). The algorithm maintains an $|M|$-dimensional mixture vector $\boldsymbol{f}$ in the convex hull of count vectors. This hull is the following polytope over weight vectors obtained by relaxing the integer constraints on the count vectors:

Definition 6 (Unit-Flow Polytope)**.** *Given a multi-DAG $\mathcal{H} = (V, M)$, let $\boldsymbol{f} \in \mathbb{R}_{\geq 0}^{|M|}$ in which f_m is associated with $m \in M$. Define the inflow $f_{in}(v) := \sum_{m \in M_v^{(in)}} f_m$ and the outflow $f_{out}(v) := \sum_{m \in M_v^{(out)}} f_m$. $\boldsymbol{f}$ belongs to the **unit-flow polytope** of $\mathcal{H}$ if it has*

the following properties:

1. *The outflow $f_{out}(s)$ of the source s is 1.*

2. *For each vertex $v \in V - \mathcal{T} - \{s\}$, the outflow is equal to the inflow, i.e. $f_{out}(v) = f_{in}(v)$.*

In each trial, the weight of each multiedge f_m is updated multiplicatively to $\widehat{f}_m = f_m \exp(-\eta \ell_m)$ and then the weight vector $\widehat{\boldsymbol{f}}$ is projected back to the unit-flow polytope via a relative entropy projection:

$$\boldsymbol{f}^{\text{new}} := \operatorname*{arg\,min}_{\boldsymbol{f} \in \text{unit-flow polytope}} \Delta(\boldsymbol{f} || \widehat{\boldsymbol{f}}), \quad \text{where} \quad \Delta(\boldsymbol{a} || \boldsymbol{b}) = \sum_i a_i \log \frac{a_i}{b_i} + b_i - a_i.$$

This projection is achieved by repeatedly cycling over the vertices and project onto the local flow constraints at the current vertex. This method is called iterative Bregman projections [Bregman, 1967]. The following lemma shows that projection to each local flow constraint is simply equivalent to scaling the in- and out-flows to the appropriate values.

Lemma 13. *The relative entropy projection to the local flow constraint at vertex $v \in V$ is done as follows:*

1. *If $v = s$, normalize the $f_{out}(v)$ to 1.*

2. *If $v \in V - \mathcal{T} - \{s\}$, scale the incoming and outgoing multiedges of v such that*

$$f_{out}(v) := f_{in}(v) := \sqrt{f_{out}(v) \cdot f_{in}(v)}.$$

Proof. Formally, the projection $\boldsymbol{f}$ of a given point $\widehat{\boldsymbol{f}} \in \mathbb{R}_{\geq 0}^{|M|}$ to constraint C is the solution to the following:

$$\arg\min_{\boldsymbol{f} \in C} \sum_{m \in M} f_m \log\left(\frac{f_m}{\widehat{f}_m}\right) + \widehat{f}_m - f_m.$$

C can be one of the two types of constraints mentioned in Definition 6. We use the method of Lagrange multipliers in both cases. Observe that if $|U| = 1$ for all multiedge $m = (v, U) \in M$, then the updates in Koolen et al. [2010] are recovered.

Constraint Type 1. The outflow from the source s must be 1. Assume $f_{m_1}, \ldots, f_{m_d}$ are the weights associated with the outgoing multiedges $m_1, \ldots, m_d$ from the source s. Then:

$$L(\boldsymbol{f}, \lambda) := \sum_{m \in M} f_m \log\left(\frac{f_m}{\widehat{f}_m}\right) + \widehat{f}_m - f_m - \lambda\left(\sum_{j=1}^{d} f_{m_j} - 1\right)$$

$$\frac{\partial L}{\partial f_m} = \log\frac{f_m}{\widehat{f}_m} = 0 \longrightarrow f_m = \widehat{f}_m \qquad \forall m \in M - \{m_1, \ldots, m_d\}$$

$$\frac{\partial L}{\partial f_{m_j}} = \log\frac{f_{m_j}}{\widehat{f}_{m_j}} - \lambda = 0 \longrightarrow f_{m_j} = \widehat{f}_{m_j} \exp(\lambda) \tag{3.5}$$

$$\frac{\partial L}{\partial \lambda} = \sum_{j=1}^{d} f_{m_j} - 1 = 0. \tag{3.6}$$

Combining equations (3.5) and (3.6) results in normalizing $f_{m_1}, \ldots, f_{m_d}$, that is:

$$\forall j \in \{1..d\} \quad f_{m_j} = \frac{\widehat{f}_{m_j}}{\sum_{j'=1}^{d} \widehat{f}_{m_{j'}}}.$$

Constraint Type 2. Given any internal node $v \in V - \mathcal{T} - \{s\}$, the outflow from v must be equal to the inflow of v. Assume $f_1^{(\text{in})}, \ldots, f_a^{(\text{in})}$ and $f_1^{(\text{out})}, \ldots, f_b^{(\text{out})}$ are

the weights associated with the incoming and outgoing multiedges from/to the node v, respectively. Then:

$$L(\boldsymbol{w},\lambda) := \sum_{m\in M} f\log\left(\frac{f_m}{\widehat{f}_m}\right) + \widehat{f}_m - f_m - \lambda\left(\sum_{b'=1}^{b} f_{b'}^{(\text{out})} - \sum_{a'=1}^{a} f_{a'}^{(\text{in})}\right)$$

$$\frac{\partial L}{\partial f_m} = \log\frac{f_m}{\widehat{f}_m} = 0 \longrightarrow f_m = \widehat{f}_m \quad \forall m \text{ non-adjacent to } v$$

$$\frac{\partial L}{\partial f_{b'}^{(\text{out})}} = \log\frac{f_{b'}^{(\text{out})}}{\widehat{f}_{b'}^{(\text{out})}} - \lambda = 0 \longrightarrow f_{b'}^{(\text{out})} = \widehat{f}_{b'}^{(\text{out})}\exp(\lambda) \qquad \forall b' \in \{1..b\} \tag{3.7}$$

$$\frac{\partial L}{\partial f_{a'}^{(\text{in})}} = \log\frac{f_{a'}^{(\text{in})}}{\widehat{f}_{a'}^{(\text{in})}} + \lambda = 0 \longrightarrow f_{a'}^{(\text{in})} = \widehat{f}_{a'}^{(\text{in})}\exp(-\lambda) \qquad \forall a' \in \{1..a\} \tag{3.8}$$

$$\frac{\partial L}{\partial \lambda} = \sum_{b'=1}^{b} f_{b'}^{(\text{out})} - \sum_{a'=1}^{a} f_{a'}^{(\text{in})} = 0. \tag{3.9}$$

Letting $\beta = \exp(\lambda)$, for all $a' \in \{1..a\}$ and all $b' \in \{1..b\}$, we can obtain the following by combining equations (3.7), (3.8) and (3.9):

$$\beta\left(\sum_{b'=1}^{b}\widehat{f}_{b'}^{(\text{out})}\right) = \frac{1}{\beta}\left(\sum_{a'=1}^{a}\widehat{f}_{a'}^{(\text{in})}\right) \longrightarrow \beta = \sqrt{\frac{\sum_{a'=1}^{a}\widehat{f}_{a'}^{(\text{in})}}{\sum_{b'=1}^{b}\widehat{f}_{b'}^{(\text{out})}}}$$

$$\forall\, b' \in \{1..b\},\ f_{b'}^{(\text{out})} = \widehat{f}_{b'}^{(\text{out})}\sqrt{\frac{\sum_{a''=1}^{a}\widehat{f}_{a''}^{(\text{in})}}{\sum_{b''=1}^{b}\widehat{f}_{b''}^{(\text{out})}}},$$

$$\forall\, a' \in \{1..a\},\ f_{a'}^{(\text{in})} = \widehat{f}_{a'}^{(\text{in})}\sqrt{\frac{\sum_{b''=1}^{b}\widehat{w}_{b''}^{(\text{out})}}{\sum_{a''=1}^{a}\widehat{w}_{a''}^{(\text{in})}}}.$$

This indicates that to enforce the flow conservation property at each internal node, the weights must be multiplicatively scaled up/down so that the new outflow and inflow is the geometric average of the old outflow and inflow. ■

Prediction. In this step, the algorithm needs to randomly predict with a multipath $\boldsymbol{\pi}$ from a distribution $\mathcal{D}$ such that $\mathbb{E}_{\mathcal{D}}[\boldsymbol{\pi}] = \boldsymbol{f}$. In Component Hedge and similar algorithms

[Helmbold and Warmuth, 2009, Koolen et al., 2010, Yasutake et al., 2011, Warmuth and Kuzmin, 2008], $\mathcal{D}$ is constructed by decomposing $\boldsymbol{f}$ into a convex combination of small number of objects. In this section, we give a new and more direct prediction method for multipaths. We construct a distribution $\mathcal{D}$ with the right expectation in Stochastic Product Form (Definition 5) by defining a new set of weights $\boldsymbol{w}$ using the flow values $\boldsymbol{f}$. For each multiedge $m = (v, U) \in M$, we set the weight $w_m = f_m / f_{\text{in}}(v)$. The induced distribution will be in Stochastic Product Form with the right expectation $\mathbb{E}_{\mathcal{D}}[\boldsymbol{\pi}] = \boldsymbol{f}$. This gives us a faster prediction method as the decomposition is avoided. Lemma 14 shows the correctness and time complexity of our method.

Lemma 14. *For each multiedge $m = (v, U) \in M$, define the weight $w_m = f_m / f_{in}(v)$. Let the distribution $\mathcal{D}$ over the multipaths be $\mathcal{D}(\boldsymbol{\pi}) := \prod_{m \in M} (w_m)^{\pi_m}$. Then:*

1. *$\mathcal{D}$ is in Stochastic Product Form.*
2. *$\mathbb{E}_{\mathcal{D}}[\boldsymbol{\pi}] = \boldsymbol{f}$.*
3. *Constructing $\mathcal{D}$ from the flow values $\boldsymbol{f}$ can be done in $\mathcal{O}(c|M|)$ time where c is an upper-bound on the branching factor of each multiedge (i.e. for all $m = (v, U) \in M$, $|U| < c$).*

Proof. $\mathcal{D}(\boldsymbol{\pi})$ is in product form by construction. The weights are also stochastic since for each non-sink vertex v:

$$\sum_{m \in M_v^{(\text{out})}} w_m = \sum_{m \in M_v^{(\text{out})}} \frac{f_m}{f_{\text{in}}(v)} = \frac{1}{f_{\text{in}}(v)} \sum_{m \in M_v^{(\text{out})}} f_m = \frac{1}{f_{\text{in}}(v)} f_{\text{out}}(v) = 1$$

Thus the $\mathcal{D}$ is in Stochastic Product Form (Definition 5). Now we show that $\mathcal{D}$ will result in the desired expectation. Let $\widehat{\boldsymbol{f}} := \mathbb{E}_{\mathcal{D}}[\boldsymbol{\pi}]$ be the flow induce by $\mathcal{D}$. Denote $\widehat{f}_{\text{in}}(v) := \sum_{m \in M_v^{(\text{in})}} \widehat{f}_m$. Let $v_1, \ldots, v_n$ be a topological order of the vertices in the underlying DAG. We use strong induction on n to show that $\widehat{f}_{\text{in}}(v) = f_{\text{in}}(v)$ for all $v \in V$. For $v_1 = s$ this is true since $\widehat{f}_{\text{in}}(s) = f_{\text{in}}(s) = 1$. For $i > 1$:

$$\begin{aligned}
\widehat{f}_{\text{in}}(v_i) &= \sum_{m=(v,U) \in M_{v_i}^{(\text{in})}} w_m \, \widehat{f}_{\text{in}}(v) \\
&= \sum_{m=(v,U) \in M_{v_i}^{(\text{in})}} w_m \, f_{\text{in}}(v) && \text{(Inductive hypothesis)} \\
&= \sum_{m=(v,U) \in M_{v_i}^{(\text{in})}} \frac{f_m}{f_{\text{in}}(v)} \, f_{\text{in}}(v) && \text{(Definition of } w_m\text{)} \\
&= \sum_{m=(v,U) \in M_{v_i}^{(\text{in})}} f_m = f_{\text{in}}(v_i)
\end{aligned}$$

and that completes the induction. Now for each multiedge $m = (v, U) \in M$ we have:

$$\widehat{f}_m = \widehat{f}_{\text{in}}(v) w_m = \widehat{f}_{\text{in}}(v) \frac{f_m}{f_{\text{in}}(v)} = f_m$$

Thus $\boldsymbol{f} = \widehat{\boldsymbol{f}}$.

To construct $\mathcal{D}$, we must find all the weights w_m. To do so, we will have two passes over the set of multiedges M. In the first pass, we compute all incoming flows $f_{\text{in}}(v)$ for all $v \in V$ in $O(c|M|)$ time. Then in the second pass we find all the weights $w_m = \frac{f_m}{f_{\text{in}}(v)}$ in $O(|M|)$ time. Having constructed $\mathcal{D}$, we can efficiently sample a multipath with the right expectation. ■

Regret Bound. The regret bound for CH depends on a good choice of the initial weight vector $\boldsymbol{f}^{\text{init}}$ in the unit-flow polytope. We use an initialization technique similar to the one discussed in Section 2.2.2. Instead of explicitly selecting $\boldsymbol{f}^{\text{init}}$ in the unit-flow polytope, the initial weight is obtained by projecting a point $\widehat{\boldsymbol{f}}^{\text{init}}$ outside of the polytope to its inside. This yields the following regret bounds.

Theorem 15. *Given a multi-DAG $\mathcal{H} = (V, M)$, let D be an upper-bound on the 1-norm of the count vectors of the multipaths (i.e. $\|\boldsymbol{\pi}\|_1 \leq D$ for all multipaths $\boldsymbol{\pi}$). Also denote the total loss of the best multipath by L^*. Then with proper tuning of the learning rate η over the trials, CH guarantees:*

$$\mathcal{R}_{CH} \leq \sqrt{2\, L^*\, D\, (\log |M| + \log D)} + D\, \log |M| + D \log D.$$

Moreover, when the multipaths are bit vectors, then:

$$\mathcal{R}_{CH} \leq \sqrt{2\, L^*\, D\, \log |M|} + D \log |M|.$$

Proof. According to Koolen et al. [2010], with proper tuning of the learning rate η, the regret bound of CH is:

$$\mathcal{R}_{\text{CH}} \leq \sqrt{2\, L^*\, \Delta(\boldsymbol{\pi} || \boldsymbol{f}^{\text{init}})} + \Delta(\boldsymbol{\pi} || \boldsymbol{f}^{\text{init}}), \tag{3.10}$$

where $\boldsymbol{\pi} \in \mathbb{N}^{|M|}$ is the best multipath and L^* its loss. Define $\widehat{\boldsymbol{f}}^{\text{init}} := \frac{1}{|M|}\, \mathbf{1}$ where $\mathbf{1} \in \mathbb{R}^{|M|}$ is a vector of all ones. Now let the initial point $\boldsymbol{f}^{\text{init}}$ be the relative entropy projection of $\widehat{\boldsymbol{f}}^{\text{init}}$ onto the unit-flow prolytope[5]

$$\boldsymbol{f}^{\text{init}} = \arg\min_{\boldsymbol{f} \in P} \Delta(\boldsymbol{f} || \widehat{\boldsymbol{f}}^{\text{init}}).$$

[5]This computation can be done as a pre-processing step.

Now we have:

$$\begin{aligned}
\Delta(\boldsymbol{\pi}||\boldsymbol{f}^{\text{init}}) &\leq \Delta(\boldsymbol{\pi}||\widehat{\boldsymbol{f}}^{\text{init}}) && \text{(Generalized Pythagorean Thm.)} \\
&= \sum_{m\in M} \pi_m \log \frac{\pi_m}{\widehat{f}_m^{\text{init}}} + \widehat{f}_m^{\text{init}} - \pi_m \\
&= \sum_{m\in M} \pi_m \log \frac{1}{\widehat{f}_m^{\text{init}}} + \pi_m \log \pi_m + \widehat{f}_m^{\text{init}} - \pi_m \\
&\leq \sum_{m\in M} \pi_m (\log |M|) + \sum_{m\in M} \pi_m \log \pi_m + \sum_{m\in M} \frac{1}{|M|} - \sum_{m\in M} \pi_m && (3.11) \\
&\leq D(\log |M|) + D \log D + |M| \frac{1}{|M|} - \sum_{m\in M} \pi_m \\
&\leq D \log |M| + D \log D.
\end{aligned}$$

Thus, by Inequality (3.10) the regret bound will be:

$$\mathcal{R}_{\text{CH}} \leq \sqrt{2\, L^*\, D\, (\log |M| + \log D)} + D \log |M| + D \log D.$$

Note that if $\boldsymbol{\pi}$ is a bit vector, then $\sum_{m\in M} \pi_m \log \pi_m = 0$, and consequently, the expression (3.11) can be bounded as follows:

$$\begin{aligned}
\Delta(\boldsymbol{\pi}||\boldsymbol{f}^{\text{init}}) &\leq \sum_{m\in M} \pi_m (\log |M|) + \sum_{m\in M} \pi_m \log \pi_m + \sum_{m\in M} \frac{1}{|M|} - \sum_{m\in M} \pi_m \\
&\leq D(\log |M|) + |M| \frac{1}{|M|} - \sum_{m\in M} \pi_m \\
&\leq D \log |M|.
\end{aligned}$$

Again, using Inequality (3.10), the regret bound will be:

$$\mathcal{R}_{\text{CH}} \leq \sqrt{2\, L^*\, D \log |M|} + D \log |M|.$$

■

Notice that by setting $|U| = 1$ for all multiedge $(v, U) \in M$ and $|\mathcal{T}| = 1$, the algorithm for path learning in Koolen et al. [2010] is recovered. Also observe that Theorem 9 is a corollary of Theorem 15 since every path is represented as a bit vector and $|M| = |E| \leq |V|^2$.

3.2.3 Stochastic Product Form vs Mean Form

We discussed the efficient implementation of the two algorithms of EH and CH for learning multipaths. The EH algorithm maintains a *weight* vector $\boldsymbol{w} \in \mathbb{R}^{|M|}$ in the Stochastic Product Form. These weights define a distribution over all multipaths. On the other hand, the CH algorithm keeps track of a *flow* vector $\boldsymbol{f} \in \mathbb{R}^{|M|}$ in the *Mean Form*. These flows define a mean vector over all multipaths and belong to the unit-flow polytope.

For any distribution over the multipaths, there is a unique expectation/mean of the counts of the multiedges according to the given distribution. This expectation is represented by a flow vector. If the distribution is in Stochastic Product Form with the weight vector $\boldsymbol{w}$, the flow vector can be computed efficiently using a dynamic programming algorithm. Initializing with the source s, we set the in-coming flow $f_{\text{in}}(s) = 1$. Then, using the recursive equation $f_m = w_m f_{\text{in}}(s)$ for all $m \in M_s^{(\text{out})}$, we find the flows of the out-going multiedges from the source s by partitioning the in-flow according to its out-going weights. Having computed the flows of all the out-going multiedges, we can find the in-flows of some of the vertices which are connected to the source. By applying the aforementioned recursion over the vertices of $\mathcal{H}$ in the topological order of the

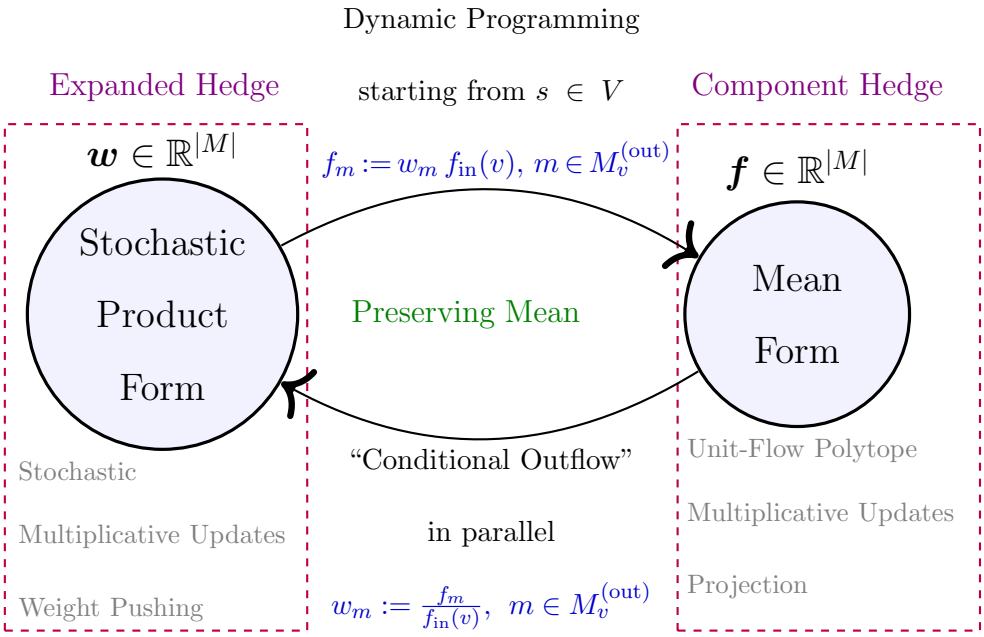


Figure 3.3: Mapping between Stochastic Product Form in EH and Mean Form in CH.

underlying base directed graph $\mathcal{B}(\mathcal{H})$, we can find the flows of all the multiedges. This procedure can be done in $\mathcal{O}(c|M|)$ time where c is an upper-bound on the branching factor of each multiedge (i.e. for all $m = (v, U) \in M$, $|U| < c$).

Conversely, by applying the Lemma 14 on a given flow vector $\boldsymbol{f}$, we can find the weights $\boldsymbol{w}$ defining the distribution $\mathcal{D}$ in the Stochastic Product Form such that it has the right expectation $\mathbb{E}_{\mathcal{D}}[\boldsymbol{\pi}] = \boldsymbol{f}$. In general if we assume no structure on the distributions over the multipaths, there could be several different distributions with the expectation $\boldsymbol{f}$. However, if we limit the distributions to the Stochastic Product Form, then the resulting distribution $\mathcal{D}$ is unique. This is because the in-flows should be distributed according to the local weights in the Stochastic Product Form.

This establishes a 1-1 and onto mapping between the Stochastic Product Form

of EH and the Mean Form of CH (see Figure 3.3). Both directions of the mapping have the additional crucial property of preserving the mean.

3.3 Online Dynamic Programming with Multipaths

We consider the problem of online learning of combinatorial objects whose offline optimization problem can be solved efficiently via a dynamic programming algorithm with arbitrary min-sum recurrence relation. This is equivalent to repeatedly solving a variant of the same min-sum dynamic programming problem in successive trials.

We will use our definition of multi-DAG (Definition 3) to describe a representation of the dynamic programming problem. The vertex set V is a set of subproblems to be solved. The source node $s \in V$ is the "complete subproblem" (i.e. the original problem). The sink nodes $\mathcal{T} \subset V$ are the base subproblems. A multiedge from a node $v \in V$ to a set of nodes $U \subset V$ means that a solution to the subproblem v may use solutions to the (smaller) subproblems in U. Denote the set of all multiedges by M. A step of the dynamic programming recursion is thus represented by a multiedge. Denote the constructed directed multigraph by $\mathcal{H} = (V, M)$. A subproblem is never solved more than once in a dynamic programming. Therefore base directed graph $\mathcal{B}(\mathcal{H})$ is acyclic and $\mathcal{H}$ is a multi-DAG.

There is a loss associated with any sink node in $\mathcal{T}$. Also with the recursions at the internal node v a local loss will be added to the loss of the subproblems that

depends on v and the chosen multiedge $m \in M_v^{(\text{out})}$ leaving v. We can handle arbitrary "min-sum" recurrences:

$$\text{OPT}(v) = \begin{cases} L_{\mathcal{T}}(v) & v \in \mathcal{T} \\ \min_{m \in M_v^{(\text{out})}} \{\sum_{u:(v,u)\in m} \text{OPT}(u) + L_M(m)\} & v \in V - \mathcal{T}. \end{cases}$$

The problem of repeatedly solving an arbitrary min-sum dynamic programming problem over trials now becomes online learning of multipaths in $\mathcal{H}$. Note that due to the correctness of the dynamic programming, every possible solution to the dynamic programming can be encoded as a multipath in $\mathcal{H}$ and vice versa.

The loss of a given multipath is the sum of $L_M(m)$ over all multiedges m in the multipath plus the sum of $L_{\mathcal{T}}(v)$ for all sink nodes v at the bottom of the multipath. To capture the same loss, we can alternatively define losses over the multiedges M. Concretely, for each multiedge $m = (v, U)$ define $\ell_m := L_M(m) + \sum_{u \in U} \mathbb{1}_{\{u \in \mathcal{T}\}} L_{\mathcal{T}}(u)$ where $\mathbb{1}_{\{\cdot\}}$ is the indicator function[6].

3.4 Applications

In this section, we apply our algorithms to various instances of online dynamic programming. In each instance, we define the problem, explore the dynamic programming representation and obtain the regret bounds.

[6] The alternative losses over the multiedges may not be in $[0, 1]$. However, it is straight-forward to see if $\ell_m \in [0, b]$ for some b, the regret bounds for CH and EH will scale up accordingly.

3.4.1 Binary Search Trees

Recall again the online version of optimal binary search tree (BST) problem [Cormen et al., 2009]: We are given a set of n distinct keys $K_1 < K_2 < \ldots < K_n$. In each trial, the algorithm predicts with a BST. Then the adversary reveals a probability vector $\boldsymbol{p} \in [0,1]^n$ with $\sum_{i=1}^n p_i = 1$. For each i, p_i indicates the *search probability* for the key K_i. The loss is defined as the *average search cost* in the predicted BST which is the average depth[7] of all the nodes in the BST:

$$\text{loss} = \sum_{i=1}^{n} \text{depth}(K_i) \cdot p_i.$$

Convex Hull of BSTs. Implementing CH requires a representation where not only the BST polytope has a polynomial number of facets, but also the loss must be linear over the components. Since the average search cost is linear in the $\text{depth}(K_i)$ variables, it would be natural to choose these n variables as the components for representing a BST. Unfortunately the convex hull of all BSTs when represented this way is not known to be a polytope with a polynomial number of facets. There is an alternate characterization of the convex hull of BSTs with n internal nodes called the *associahedron* [Loday, 2005]. This polytope has polynomial in n many facets but the average search cost is not linear in the n components associated with this polytope[8]. Thus CH cannot be applied to associahedron.

[7] Here the root starts at depth 1.

[8] Concretely, the ith component is $a_i\, b_i$ where a_i and b_i are the number of nodes in the left and right subtrees of the ith internal node K_i, respectively.

The Dynamic Programming Representation. The optimal BST problem can be solved via dynamic programming [Cormen et al., 2009]. Each subproblem is denoted by a pair (i, j), for $1 \leq i \leq n+1$ and $i-1 \leq j \leq n$, indicating the optimal BST problem with the keys $K_i, \ldots, K_j$. The base subproblems are $(i, i-1)$, for $1 \leq i \leq n+1$ and the complete subproblem is $(1, n)$. The BST dynamic programming problem uses the following min-sum recurrence:

$$\mathrm{OPT}(i,j) = \begin{cases} 0 & j = i-1 \\ \min_{i \leq r \leq j} \{\mathrm{OPT}(i, r-1) + \mathrm{OPT}(r+1, j) + \sum_{k=i}^{j} p_k\} & i \leq j. \end{cases}$$

This recurrence always recurses on 2 subproblems. Therefore for every multiedge (v, U) we have $|U| = 2$. The associated multi-DAG has the subproblems/vertices $V = \{(i,j) | 1 \leq i \leq n+1, i-1 \leq j \leq n\}$, source $s = (1, n)$ and sinks $\mathcal{T} = \{(i, i-1) | 1 \leq i \leq n+1\}$. Also at node (i, j), the set $M_{(i,j)}^{(\mathrm{out})}$ consists of $(j - i + 1)$ many multiedges. The rth multiedge leaving (i, j) comprised of 2 edges going from the node (i, j) to the nodes $(i, r-1)$ and $(r+1, j)$. Figure 3.4 illustrates the underlying multi-DAG and the multipath associated with a given BST.

Since the above recurrence relation correctly solves the offline optimization problem, every multipath in the DAG represents a BST, and every possible BST can be represented by a multipath of the 2-DAG. We have $|M| = O(n^3)$ multiedges which are the components of our new representation. The loss of each multiedge leaving (i, j) is $\sum_{k=i}^{j} p_k$ and is upper bounded by 1. Most crucially, the original average search cost is linear in the losses of the multiedges and the unit-flow polytope has $O(n^3)$ facets.

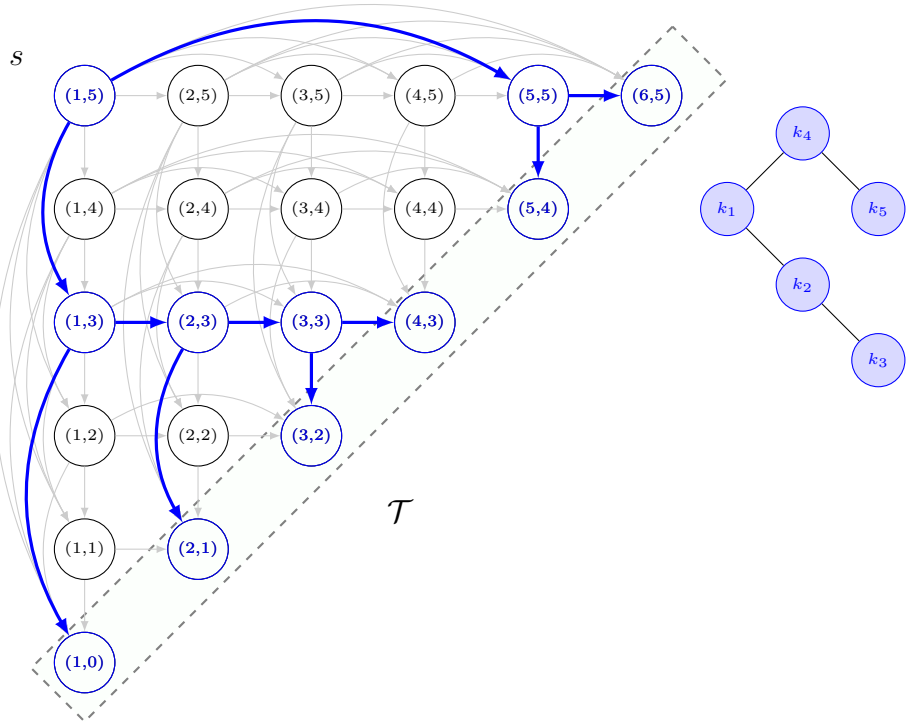


Figure 3.4: (Left) An example of a multipath in blue in the underlying multi-DAG. The nodes in $\mathcal{T}$ represent the subproblems associated with the "gaps" e.g. $(3, 2)$ represents the binary search tree for all values between the keys 2 and 3. (Right) its associated BSTs of $n = 5$ keys. Note that each node, and consequently multiedge, is visited at most once in these multipaths.

Regret Bound. As mentioned earlier, the number of binary trees with n nodes is the nth Catalan number. Therefore $\mathcal{N} = \frac{(2n)!}{n!(n+1)!} \in (2^n, 4^n)$. Also note that each multipath representing a BST consists of exactly $D = n$ multiedges. Thus using Theorem 12, EH achieves a regret bound of $\mathcal{O}(n\sqrt{L^*})$. Moreover, since $|M| = O(n^3)$, using Theorem 15, CH achieves a regret bound of $\mathcal{O}(n^{\frac{1}{2}}(\log n)^{\frac{1}{2}}\sqrt{L^*})$.

3.4.2 Matrix-Chain Multiplication

Given a sequence $A_1, A_2, \ldots, A_n$ of n matrices, our goal is to compute the product $A_1 \times A_2 \times \ldots \times A_n$ in the most efficient way. Using the standard algorithm for multiplying pairs of matrices as a subroutine, this product can be found by a specifying the order which the matrices are multiplied together. This order is determined by a *full parenthesization*: A product of matrices is fully parenthesized if it is either a single matrix or the multiplication of two fully parenthesized matrix products surrounded by parentheses. For instance, there are five full parenthesizations of the product $A_1A_2A_3A_4$:

$$(A_1(A_2(A_3A_4)))$$

$$(A_1((A_2A_3)A_4))$$

$$((A_1A_2)(A_3A_4))$$

$$(((A_1A_2)A_3)A_4)$$

$$((A_1(A_2A_3))A_4).$$

We consider the online version of *matrix-chain multiplication* problem [Cormen et al., 2009]. In each trial, the algorithm predicts with a full parenthesization of the

product $A_1 \times A_2 \times \ldots \times A_n$ without knowing the dimensions of these matrices. Then the adversary reveals the dimensions of each A_i at the end of the trial denoted by $d_{i-1} \times d_i$ for all $i \in \{1..n\}$. The loss of the algorithm is defined as the number of scalar multiplications in the matrix-chain product in that trial. The goal is to predict with a sequence of full parenthesizations minimizing regret which is the difference between the total loss of the algorithm and the total loss of the single best full parenthesization chosen in hindsight. Observe that the number of scalar multiplications in the matrix-chain product cannot be expressed as a linear loss over the dimensions of the matrices d_i's.

The Dynamic Programming Representation. Finding the best full parenthesization can be solved via dynamic programming [Cormen et al., 2009]. Each subproblem is denoted by a pair (i, j) for $1 \leq i \leq j \leq n$, indicating the problem of finding a full parenthesization of the partial matrix product $A_i \ldots A_j$. The base subproblems are (i, i) for $1 \leq i \leq n$ and the complete subproblem is $(1, n)$. The dynamic programming for matrix chain multiplication uses the following min-sum recurrence:

$$\mathrm{OPT}(i,j) = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{\mathrm{OPT}(i,k) + \mathrm{OPT}(k+1,j) + d_{i-1}\, d_k\, d_j\} & i < j. \end{cases}$$

This recurrence always recurses on 2 subproblems, thus for all multiedges $m = (v, U) \in M$ we have $|U| = 2$. The associated multi-DAG has the subproblems/vertices $V = \{(i, j) \mid 1 \leq i \leq j \leq n\}$, source $s = (1, n)$ and sinks $\mathcal{T} = \{(i, i) \mid 1 \leq i \leq n\}$. Also at node (i, j), the set $M^{(\mathrm{out})}_{(i,j)}$ consists of $(j - i)$ many multiedges. The kth multiedge leaving

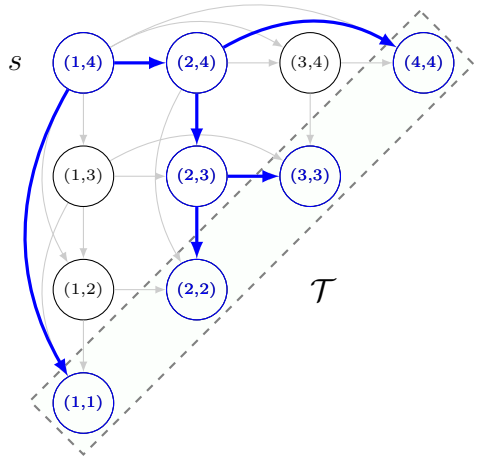


Figure 3.5: Given a chain of $n = 4$ matrices, the multipath associated with the full parenthesization $(A_1((A_2A_3)A_4))$ is depicted in blue.

(i, j) is comprised of 2 edges going from the node (i, j) to the nodes (i, k) and $(k+1, j)$. The loss of the kth multiedge is $d_{i-1}\, d_k\, d_j$. Figure 3.5 illustrates the multi-DAG and multipaths associated with matrix chain multiplications.

Since the above recurrence relation correctly solves the offline optimization problem, every multipath in the multi-DAG represents a full parenthesization, and every possible full parenthesization can be represented by a multipath of the multi-DAG. We have $|M| = O(n^3)$ multiedges which are the components of our new representation. Assuming that all dimensions d_i are bounded as $d_i < d_{\max}$ for some $d_{\max}$, the loss associated with each multiedge is upper-bounded by $(d_{\max})^3$. Most crucially, the original number of scalar multiplications in the matrix-chain product is linear in the losses of the multiedges and the unit-flow polytope has $O(n^3)$ facets.

Regret Bounds. It is well-known that the number of full parenthesizations of a sequence of n matrices is the nth Catalan number [Cormen et al., 2009]. Therefore $\mathcal{N} = \frac{(2n)!}{n!(n+1)!} \in (2^n, 4^n)$. Also note that each multipath representing a full parenthesization consists of exactly $D = n - 1$ multiedges. Thus, incorporating $(d_{\max})^3$ as the loss range for each component and using Theorem 12, EH achieves a regret bound of $\mathcal{O}(n\,(d_{\max})^{\frac{3}{2}}\sqrt{L^*})$. Moreover, since $|M| = O(n^3)$, using Theorem 15 and considering $(d_{\max})^3$ as the loss range for each component , CH achieves a regret bound of $\mathcal{O}(n^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\,(d_{\max})^{\frac{3}{2}}\sqrt{L^*})$.

3.4.3 Knapsack

Consider the online version of the *knapsack* problem [Kleinberg and Tardos, 2006]: We are given a set of n items along with the *capacity* of the knapsack $C \in \mathbb{N}$. For each item $i \in \{1..n\}$, a *heaviness* $h_i \in \mathbb{N}$ is associated. In each trial, the algorithm predicts with a *packing* which is a subset of items whose total heaviness is at most the capacity of the knapsack. After the prediction of the algorithm, the adversary reveals the profit of each item $p_i \in [0, 1]$. The gain is defined as the sum of the profits of the items picked in the packing predicted by the algorithm in that trial. The goal is to predict with a sequence of packings minimizing regret which is the difference between the total gain of the algorithm and the total gain of the single best packing chosen in hindsight.

Note that this online learning problem only deals with exponentially many objects when there are exponentially many feasible packings. If the number of packings

is polynomial, then it is practical to simply run the Hedge algorithm with one weight per packing. Here we consider a setting of the problem where maintaining one weight per packing is impractical. We assume C and h_i's are in such way that the number of feasible packings is exponential in n.

The Dynamic Programming Representation. Finding the optimal packing can be solved via dynamic programming [Kleinberg and Tardos, 2006]. Each subproblem is denoted by a pair (i, c) for $0 \leq i \leq n$ and $0 \leq c \leq C$, indicating the knapsack problem given items $1, \ldots, i$ and capacity c. The base subproblems are $(0, c)$ for $0 \leq c \leq C$ and the complete subproblem is (n, C). The dynamic programming for the knapsack problem uses the following max-sum recurrence:

$$\mathrm{OPT}(i,c) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1,c) & c < h_i \\ \max\{\mathrm{OPT}(i-1,c),\ p_i + \mathrm{OPT}(i-1,c-h_i)\} & \text{else.} \end{cases}$$

This recurrence always recurses on 1 subproblem. Thus the multipaths are regular paths and the problem is essentially the online longest-path problem with several sink nodes. The associated DAG has the subproblems/vertices $V = \{(i,c) \mid 0 \leq i \leq n, \quad 0 \leq c \leq C\}$, source $s = (n, C)$ and sinks $\mathcal{T} = \{(0,c) \mid 0 \leq c \leq C\}$. Also at node (i, c), the set $M_{(i,c)}^{(\mathrm{out})}$ consists of two edges going from the node (i, c) to the nodes $(i-1, c)$ and $(i-1, c-h_i)$. Figure 3.6 illustrates an example of the DAG and a sample path associated with a packing.

Since the above recurrence relation correctly solves the offline optimization

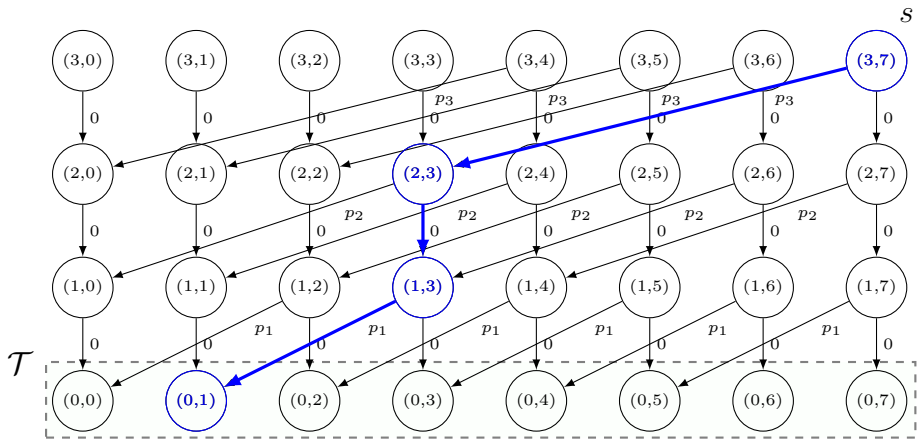


Figure 3.6: An example with $C = 7$ and $(h_1, h_2, h_3) = (2, 3, 4)$. The packing of picking the first and third item is highlighted.

problem, every path in the DAG represents a packing, and every possible packing can be represented by a path of the DAG. We have $|M| = |E| = O(n\,C)$ edges which are the components of our new representation. The gains of the edges going from the node (i, c) to the nodes $(i-1, c)$ and $(i-1, c-h_i)$ are 0 and p_i, respectively. Note that the gain associated with each edge is upper-bounded by 1. Most crucially, the sum of the profits of the picked items in the packing is linear in the gains of the edges and the unit-flow polytope has $O(n\,C)$ facets.

Regret Bounds. We turn the problem into an equivalent shortest-path problem by defining a loss for each edge $e \in E$ as $\ell_e = 1 - g_e$ in which g_e is the gain of e. Call this new DAG $\bar{\mathcal{G}}$. Let $L_{\bar{\mathcal{G}}}(\pi)$ be the loss of path π in $\bar{\mathcal{G}}$ and $G_{\mathcal{G}}(\pi)$ be the gain of path π in $\mathcal{G}$. Since all paths contain exactly $D = n$ edges, the loss and gain are related as follows: $L_{\bar{\mathcal{G}}}(\boldsymbol{\pi}) = n - G_{\mathcal{G}}(\boldsymbol{\pi})$. According to our initial assumption $\log \mathcal{N} = \mathcal{O}(n)$. Thus using

Theorem 12 we obtain:

$$G^* - \mathbb{E}[G_{\text{EH}}] = (nT - L^*) - (nT - \mathbb{E}[L_{\text{EH}}])$$
$$= \mathbb{E}[L_{\text{EH}}] - L^* = \mathcal{O}(n\sqrt{L^*}).$$

Notice that the number of multiedges/edges is $|M| = |E| = O(n\,C)$ and each path consists of $D = n$ edges. Therefore using Theorem 15 we obtain:

$$G^* - \mathbb{E}[G_{\text{CH}}] = (nT - L^*) - (nT - \mathbb{E}[L_{\text{CH}}])$$
$$= \mathbb{E}[L_{\text{CH}}] - L^* = \mathcal{O}(n^{\frac{1}{2}}\,(\log nC)^{\frac{1}{2}}\,\sqrt{L^*}).$$

3.4.4 k-Sets

Consider the online learning of the *k-sets* [Warmuth and Kuzmin, 2008]: We want to learn subsets of size k of the set $\{1..n\}$. In each trial, the algorithm predicts with a k-set. Then, the adversary reveals the loss of each element ℓ_i for $i \in \{1..n\}$. The loss is defined as the sum of the losses of the elements in the k-set predicted by the algorithm in that trial. The goal is to predict with a sequence of k-sets minimizing regret which is the difference between the total loss of the algorithm and the total loss of the single best k-set chosen in hindsight.

The Dynamic Programming Representations. Finding the optimal k-set can be solved via dynamic programming. Each subproblem is denoted by a pair (i, j) for $0 \leq j \leq k$ and $j \leq i \leq j + n - k$, indicating the j-set problem over the set $\{1, \ldots, i\}$. The base subproblem is $(0, 0)$ and the complete subproblem is (n, k). The dynamic programming for the k-set problem uses the following min-sum recurrence:

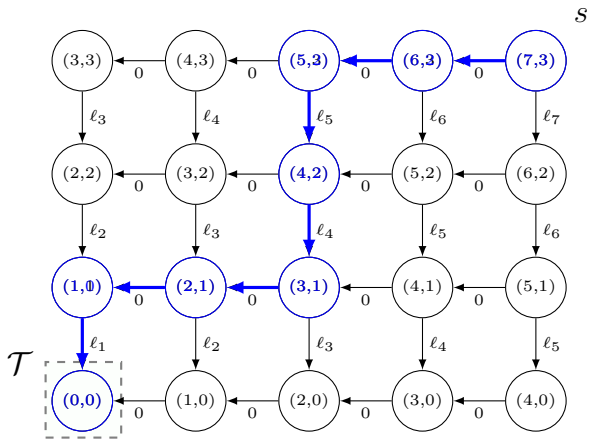


Figure 3.7: An example of k-set with $n = 7$ and $k = 3$. The 3-set of $(1, 0, 0, 1, 1, 0, 0)$ is highlighted.

$$\mathrm{OPT}(i,j) = \begin{cases} 0 & i = j = 0 \\ \mathrm{OPT}(i-1,0) & j = 0 \\ \mathrm{OPT}(i-1,i-1)+\ell_i & j = i \\ \min\{\mathrm{OPT}(i-1,j),\mathrm{OPT}(i-1,j-1)+\ell_i\} & \text{otherwise.} \end{cases}$$

This recurrence always recurses on 1 subproblem. Thus the multipaths are regular paths and the problem is essentially the online shortest-path problem from a source to a sink. The associated DAG has the subproblems/vertices $V = \{(i,j) \mid 0 \le j \le k, \quad j \le i \le j+n-k\}$, source $s = (n,k)$ and sink $\mathcal{T} = \{(0,0)\}$. Also at node (i,j), the set $M^{(\mathrm{out})}_{(i,j)}$ consists of two edges going from the node (i,j) to the nodes $(i-1,j)$ and $(i-1,j-1)$. Figure 3.7 illustrates an example of the DAG and a sample path associated with a k-set.

Since the above recurrence relation correctly solves the offline k-set problem, every path in the DAG represents a k-set, and every possible k-set can be represented by a path of the DAG. We have $|M| = |E| = 2k(n-k)+n$ edges which are the components of our new representation. The losses of the edges going from the node (i, j) to the nodes $(i-1, j)$ and $(i-1, j-1)$ are 0 and ℓ_i, respectively. Note that the loss associated with each edge is upper-bounded by 1. Most crucially, the sum of the losses of the predicted k-set is linear in the losses of the edges and the unit-flow polytope has $O(k(n-k))$ facets.

Regret Bounds. The number of k-sets is $\mathcal{N} = \binom{n}{k}$. Also note that each path representing a k-set consists of exactly $D = n$ edges and its loss is bounded by k. Thus, using Theorem 12, EH achieves a regret bound of $\mathcal{O}(k\,(\log n)^{\frac{1}{2}}\sqrt{L^*})$. Moreover, since $|E| = O(k(n-k))$, using Theorem 15, CH achieves a regret bound of $\mathcal{O}(n^{\frac{1}{2}}\,(\log k(n-k))^{\frac{1}{2}}\sqrt{L^*})$.

Remark. The convex hull of the k-sets in its original space, known as *capped probability simplex*, is well-behaved. This polytope has $n+1$ facets and the exact relative entropy projection to this polytope can be found efficiently [Warmuth and Kuzmin, 2008]. Thus applying CH in the original space will result in more efficient algorithm with better bounds of $\mathcal{O}(k^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\sqrt{L^*})$. Nevertheless, an efficient implementation of the EH algorithm can be obtained via our online dynamic programming framework. Interestingly, in the special case of the k-set, the regret bounds of EH is also $\mathcal{O}(k^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\sqrt{L^*})$ [Kivinen, 2010].

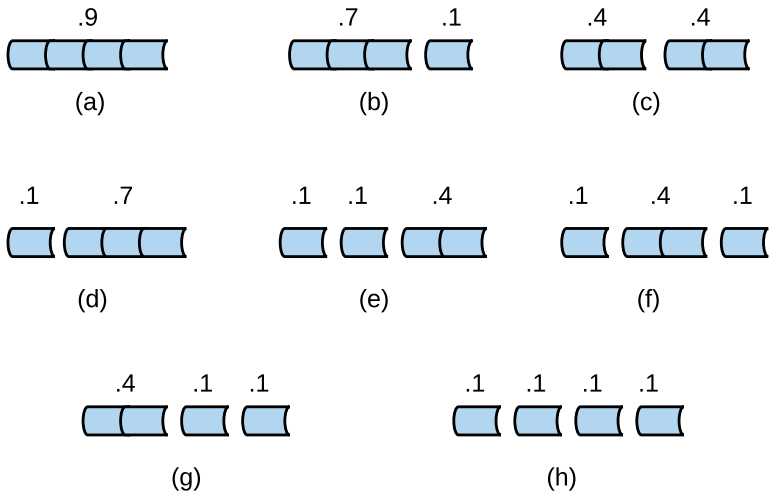


Figure 3.8: All cuttings of a rod of length $n = 4$ and their profits given $(p_1, p_2, p_3, p_4) = (.1, .4, .7, .9)$.

3.4.5 Rod Cutting

Consider the online version of *rod cutting* problem [Cormen et al., 2009]: A rod of length $n \in \mathbb{N}$ is given. In each trial, the algorithm predicts with a *cutting*, that is, it cuts up the rod into smaller pieces of integer length. Then the adversary reveals a *profit* $p_i \in [0, 1]$ for each piece of length $i \in \{1..n\}$ that can be possibly generated out of a cutting. The gain of the algorithm is defined as the sum of the profits of all the pieces generated by the predicted cutting in that trial. The goal is to predict with a sequence of cuttings minimizing regret which is the difference between the total gain of the algorithm and the total gain of the single best cutting chosen in hindsight. See Figure 3.8 as an example.

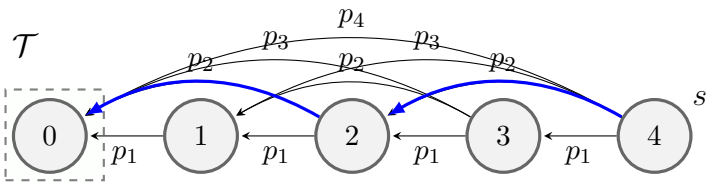


Figure 3.9: An example of rod cutting problem with $n = 4$. The cutting with two smaller pieces of size 2 is highlighted.

The Dynamic Programming Representation. Finding the optimal cutting can be solved via dynamic programming [Cormen et al., 2009]. Each subproblem is simply denoted by i for $0 \leq i \leq n$, indicating the rod cutting problem given a rod of length i. The base subproblem is $i = 0$, and the complete subproblem is $i = n$. The dynamic programming for the rod cutting problem uses the following max-sum recurrence:

$$\mathrm{OPT}(i) = \begin{cases} 0 & i = 0 \\ \max_{0 \leq j \leq i}\{OPT(j) + p_{i-j}\} & i > 0. \end{cases}$$

This recurrence always recurses on 1 subproblem. Thus the multipaths are regular paths and the problem is essentially the online longest-path problem from the source to the sink. The associated DAG has the subproblems/vertices $V = \{0, 1, \ldots, n\}$, source $s = n$ and sink $\mathcal{T} = \{0\}$. Also at node i, the set $M_i^{(\mathrm{out})}$ consists of i edges going from the node i to the nodes $0, 1, \ldots, i-1$. Figure 3.9 illustrates the DAG and paths associated with the cuttings.

Since the above recurrence relation correctly solves the offline optimization problem, every path in the DAG represents a cutting, and every possible cutting can be represented by a path of the DAG. We have $|M| = |E| = O(n^2)$ multiedges/edges which

are the components of our new representation. The gains of the edges going from the node i to the node j (where $j < i$) is p_{i-j}. Note that the gain associated with each edge is upper-bounded by 1. Most crucially, the sum of the profits of all the pieces generated by the cutting is linear in the gains of the edges and the unit-flow polytope has $O(n)$ facets.

Regret Bounds. Similar to the knapsack problem, we turn this problem into a shortest-path problem: We first modify the graph so that all paths have equal length of n (which is the length of the longest path) and the gain of each path remains fixed. We apply a method introduced in György et al. [2007], which adds $\mathcal{O}(n^2)$ vertices and edges (with gain zero) to make all paths have the same length of $D = n$. Then we define a loss for each edge e as $\ell_e = 1 - g_e$ in which g_e is the gain of e. Call this new DAG $\bar{\mathcal{G}}$. Similar to the knapsack problem, we have $L_{\bar{\mathcal{G}}}(\boldsymbol{\pi}) = n - G_{\mathcal{G}}(\boldsymbol{\pi})$ for all paths $\boldsymbol{\pi}$. Note that in both $\mathcal{G}$ and $\bar{\mathcal{G}}$, there are $\mathcal{N} = 2^{n-1}$ paths. Thus using Theorem 12 we obtain[9]

$$\begin{aligned} G^* - \mathbb{E}[G_{\text{EH}}] &= (nT - L^*) - (nT - \mathbb{E}[L_{\text{EH}}]) \\ &= \mathbb{E}[L_{\text{EH}}] - L^* = \mathcal{O}(n\sqrt{L^*}). \end{aligned}$$

Notice that the number of multiedges/edges in $\bar{\mathcal{G}}$ is $|M| = |E| = O(n^2)$ and

[9] We are over-counting the number of cuttings. The number of possible cutting is called *partition function* which is approximately $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$ [Cormen et al., 2009]. Thus if we run the Hedge algorithm inefficiently with one weight per cutting, we will get a better regret bound by a factor of $\sqrt[4]{n}$.

each path consists of $D = n$ edges. Therefore using Theorem 15 we obtain:

$$\begin{aligned} G^* - \mathbb{E}[G_{\text{CH}}] &= (nT - L^*) - (nT - \mathbb{E}[L_{\text{CH}}]) \\ &= \mathbb{E}[L_{\text{CH}}] - L^* = \mathcal{O}(n^{\frac{1}{2}} (\log n)^{\frac{1}{2}} \sqrt{L^*}). \end{aligned}$$

3.4.6 Weighted Interval Scheduling

Consider the online version of *weighted interval scheduling* problem [Kleinberg and Tardos, 2006]: We are given a set of n intervals $I_1, \ldots, I_n$ on the real line. In each trial, the algorithm predicts with a *scheduling* which is a subset of non-overlapping intervals. Then, for each interval I_j, the adversary reveals $p_j \in [0, 1]$ which is the *profit* of including I_j in the scheduling. The gain of the algorithm is defined as the total profit over chosen intervals in the scheduling in that trial. The goal is to predict with a sequence of schedulings minimizing regret which is the difference between the total gain of the algorithm and the total gain of the single best scheduling chosen in hindsight. See Figure 3.10 as an example. Note that this problem is only interesting when there are exponential in n many combinatorial objects (schedulings).

The Dynamic Programming Representation. Finding the optimal scheduling can be solved via dynamic programming [Kleinberg and Tardos, 2006]. Each subproblem is simply denoted by i for $0 \leq i \leq n$, indicating the weighted scheduling problem for the intervals $I_1, \ldots, I_i$. The base subproblem is $i = 0$, and the complete subproblem is $i = n$. The dynamic programming for the weighted interval scheduling problem uses

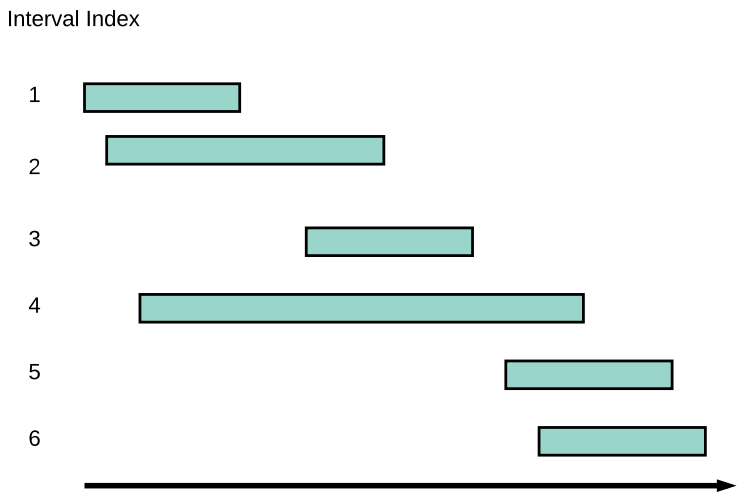


Figure 3.10: An example of weighted interval scheduling with $n = 6$

the following max-sum recurrence:

$$\mathrm{OPT}(i) = \begin{cases} 0 & i = 0 \\ \max\{\mathrm{OPT}(i-1), \mathrm{OPT}(\mathrm{pred}(i)) + p_i\} & i > 0. \end{cases}$$

where

$$\mathrm{pred}(i) := \begin{cases} 0 & i = 1 \\ \max_{\{j<i,\, I_i \cap I_j = \emptyset\}} j & i > 1. \end{cases}$$

This recurrence always recurses on 1 subproblem. Thus the multipaths are regular paths and the problem is essentially the online longest-path problem from the source to the sink. The associated DAG has the subproblems/vertices $V = \{0, 1, \ldots, n\}$, source $s = n$ and sink $\mathcal{T} = \{0\}$. Also at node i, the set $M_i^{(\mathrm{out})}$ consists of 2 edges going from the node i to the nodes $i - 1$ and $\mathrm{pred}(i)$. Figure 3.11 illustrates the DAG and paths associated with the scheduling for the example given in Figure 3.10.

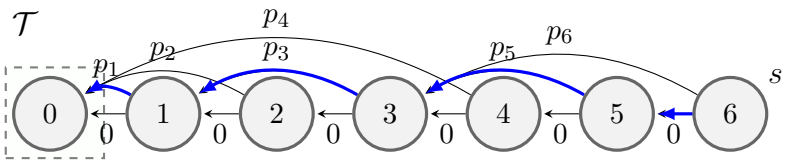


Figure 3.11: The underlying DAG associated with the example illustrated in Figure 3.10. The scheduling with I_1, I_3, and I_5 is highlighted.

Since the above recurrence relation correctly solves the offline optimization problem, every path in the DAG represents a scheduling, and every possible scheduling can be represented by a path of the DAG. We have $|M| = |E| = O(n)$ multiedges/edges which are the components of our new representation. The gains of the edges going from the node i to the nodes $i-1$ and $\text{pred}(i)$ are 0 and p_i, respectively. Note that the gain associated with each edge is upper-bounded by 1. Most crucially, the total profit over chosen intervals in the scheduling is linear in the gains of the edges and the unit-flow polytope has $O(n)$ facets.

Regret Bounds. Similar to rod cutting, this is also the online longest-path problem with one sink node. Like the rod cutting problem, we modify the graph by adding $\mathcal{O}(n^2)$ vertices and edges (with gain zero) to make all paths have the same length of $D = n$ and change the gains into losses. Call this new DAG $\bar{\mathcal{G}}$. Again we have $L_{\bar{\mathcal{G}}}(\boldsymbol{\pi}) = n - G_{\mathcal{G}}(\boldsymbol{\pi})$ for all paths $\boldsymbol{\pi}$. According to our initial assumption $\log \mathcal{N} = \mathcal{O}(n)$. Thus using Theorem

12 we obtain:

$$G^* - \mathbb{E}[G_{\text{EH}}] = (nT - L^*) - (nT - \mathbb{E}[L_{\text{EH}}])$$
$$= \mathbb{E}[L_{\text{EH}}] - L^* = \mathcal{O}(n\sqrt{L^*}).$$

Notice that the number of multiedges/edges in $\bar{\mathcal{G}}$ is $|M| = |E| = O(n^2)$ and each path consists of $D = n$ edges. Therefore using Theorem 15 we obtain:

$$G^* - \mathbb{E}[G_{\text{CH}}] = (nT - L^*) - (nT - \mathbb{E}[L_{\text{CH}}])$$
$$= \mathbb{E}[L_{\text{CH}}] - L^* = \mathcal{O}(n^{\frac{1}{2}} (\log n)^{\frac{1}{2}} \sqrt{L^*}).$$

3.5 Conclusions and Future Work

We developed a general framework for online learning of combinatorial objects whose offline optimization problems can be efficiently solved via "min-sum" dynamic programming algorithms. Table 3.2 gives the performance of EH and CH in our dynamic programming framework and compares it with the Follow the Perturbed Leader (FPL) algorithm. FPL additively perturbs the losses and then uses dynamic programming to find the solution of minimum loss (see Section 1.3). FPL is always worse than EH and CH. CH is better than both FPL and EH in all cases except k-set. In the case of k-sets, CH can be better implemented in the original space by using the capped probability

[10]The loss of a fully parenthesized matrix-chain multiplication is the number of scalar multiplications in the execution of all matrix products. This number cannot be expressed as a linear loss over the dimensions of the matrices. We are thus unaware of a way to apply FPL to this problem using the dimensions of the matrices as the components.

Problem	FPL	EH	CH
Optimal Binary Search Trees	$\mathcal{O}(n\,(\log n)^{\frac{1}{2}}\,\sqrt{L^*})$	$\mathcal{O}(n\,\sqrt{L^*})$	$\mathcal{O}(n^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\,\sqrt{L^*})$ ***Best***
Matrix-Chain Multiplications [10]	—	$\mathcal{O}(n\,(d_{\max})^{\frac{3}{2}}\,\sqrt{L^*})$	$\mathcal{O}(n^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\,(d_{\max})^{\frac{3}{2}}\,\sqrt{L^*})$ ***Best***
Knapsack	$\mathcal{O}(n\,(\log n)^{\frac{1}{2}}\,\sqrt{L^*})$	$\mathcal{O}(n\,\sqrt{L^*})$	$\mathcal{O}(n^{\frac{1}{2}}\,(\log nC)^{\frac{1}{2}}\,\sqrt{L^*})$ ***Best***
k-sets	$\mathcal{O}(k^{\frac{1}{2}}\,n^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\sqrt{L^*})$	$\mathcal{O}(k^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\sqrt{L^*})$ ***Best***	$\mathcal{O}(n^{\frac{1}{2}}\,(\log k(n-k))^{\frac{1}{2}}\sqrt{L^*})$
Rod Cutting	$\mathcal{O}(n\,(\log n)^{\frac{1}{2}}\,\sqrt{L^*})$	$\mathcal{O}(n\,\sqrt{L^*})$	$\mathcal{O}(n^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\,\sqrt{L^*})$ ***Best***
Weighted Interval Scheduling	$\mathcal{O}(n\,(\log n)^{\frac{1}{2}}\,\sqrt{L^*})$	$\mathcal{O}(n\,\sqrt{L^*})$	$\mathcal{O}(n^{\frac{1}{2}}\,(\log n)^{\frac{1}{2}}\,\sqrt{L^*})$ ***Best***

Table 3.2: Performance of various algorithms over different problems in the full information setting. C is the capacity in the Knapsack problem, and $d_{\max}$ is the upper-bound on the dimension in matrix-chain multiplication problem.

simplex as the polytope [Warmuth and Kuzmin, 2008, Koolen et al., 2010] rather than the dynamic programming representation and the unit-flow polytope.

We conclude with a few remarks:

- For EH, projections are simply a renormalization of the weight vector. In contrast, iterative Bregman projections are often needed for projecting back into the polytope used by CH [Koolen et al., 2010, Helmbold and Warmuth, 2009]. These methods are known to converge to the exact projection [Bregman, 1967, Bauschke and Borwein, 1997] and are reported to be very efficient empirically [Koolen et al., 2010]. For the special cases of Euclidean projections [Deutsch, 1995] and Sinkhorn Balancing [Knight, 2008], linear convergence has been proven. However we are un-

aware of a linear convergence proof for general Bregman divergences.

- We hope that many of the techniques from the expert setting literature can be adapted to learning combinatorial objects that are composed of components. This includes lower bounding weights for shifting comparators [Herbster and Warmuth, 1998] and sleeping experts [Bousquet and Warmuth, 2002, Adamskiy et al., 2012].

- In this chapter, we studied the online learning problem in *full information* setting, where the learner receives the loss of its choice in such a way that the loss of any of the possible solution can be easily computed. In the *bandit* setting, however, the learner only observes the loss it incurs. In the multipath learning problem, this means that the learner only observes the loss of its predicted multipath and the losses on the multiedges are not revealed. The algorithms in bandit settings usually apply EH or CH over the *surrogate* loss vector which is an unbiased estimation of the true unrevealed loss vector [Cesa-Bianchi and Lugosi, 2012, György et al., 2007, Audibert et al., 2013, 2011]. Extending our methods to the bandit settings by efficiently computing the surrogate loss vector is a potentially fruitful future direction of this research.

- *Online Markov Decision Processes (MDPs)* [Even-Dar et al., 2009, Dick et al., 2014] is an online learning model that focuses on the sequential revelation of an object using a sequential state based model. This is very much related to learning paths and the sequential decisions made in our dynamic programming framework. Connecting our work with the large body of research on MDPs is a promising

direction of future research.

- There are several important dynamic programming instances that are not included in the class considered in this paper: The Viterbi algorithm for finding the most probable path in a graph, and variants of Cocke-Younger-Kasami (CYK) algorithm for parsing probabilistic context-free grammars. The solutions for these problems are min-sum type optimization problem after taking a log of the probabilities. However taking logs creates unbounded losses. Extending our methods to these dynamic programming problems would be very worthwhile.

Chapter 4

Online Non-Additive Path Learning

One of the core combinatorial online learning problems is learning a minimum loss path in a directed graph. Examples are machine translation, automatic speech recognition, optical character recognition and computer vision. To represent the structure in these problems, the object may be decomposed in possibly overlapping substructures corresponding to words, phonemes, characters, or image patches. The substructures can be represented as a directed graph where each edge represents a different substructure to be predicted.

The number of paths (which serve as *experts*), is typically exponential in the size of the graph. Extensive work has been done to develop efficient algorithms when the loss is "additive", i.e. losses are assigned to the edges and the loss of a path is the sum over the losses of the edges along that path. A variety of algorithms have been developed in the full information and various bandit settings exploiting the additivity of the loss [Takimoto and Warmuth, 2003, Kalai and Vempala, 2005, Koolen et al., 2010,

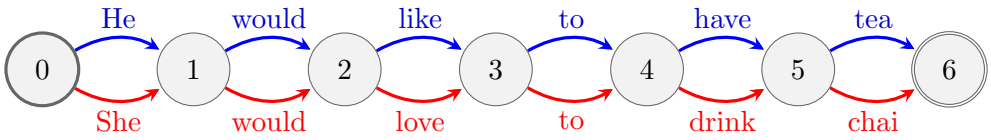


Figure 4.1: Combining two different translators (blue and red). There are 64 interleaved translations represented as paths. BLEU score measures the overlap in n-grams between sequences. In this illustration, a 4-gram is a sequence of 4 words e.g. "like-to-drink-tea".

György et al., 2007, Cesa-Bianchi and Lugosi, 2012].

However, in modern machine learning applications like machine translation, speech recognition and computational biology, the loss of each path is often not additive in the edges along the path. For instance, in machine translation, the BLEU score similarity determines the loss, which is essentially defined as the inner product of the count vectors of the n-gram occurrences in two sequences, where often $n = 4$ (see Figure 4.1). In some computational biology tasks, the losses are in terms of measures such as the gappy n-gram similarity which can also be represented as the inner product of the (discounted) count vectors of the n-gram occurrences, where these occurrences are allowed to have gaps. In other applications, like speech recognition and optical character recognition, the loss is based on edit-distance. As the performance of the algorithms in these applications are measured via these non-additive loss functions, it is natural to seek learning algorithms optimizing these losses directly. This motivates our study of online path learning for non-additive losses.

One of the applications is *ensemble structured prediction*. This application becomes important particularly in the bandit setting. Assume one wishes to combine

the outputs of different translators as in Figure 4.1. Instead of comparing oneself to the outputs of the best translator, the comparator is the best "interleaved translation" where each word in the translation can come from a different translator. Computing the loss (such as BLEU score) of each path can be costly, which requires the learner to resort to learning from partial feedback.

Online path learning with non-additive losses has been previously studied in Cortes et al. [2015b]. This work focuses on the full information case, providing efficient implementations of Expanded Hedge [Takimoto and Warmuth, 2003] and Follow-the-Perturbed-Leader [Kalai and Vempala, 2005] algorithms under some technical assumptions on the outputs of the experts.

In this chapter, we develop algorithms for online path learning in the full information as well as various bandit settings. In the full information setting, we design an efficient algorithm that enjoys regret guarantees that are more favorable than those of Cortes et al. [2015b] and at the same time does not require any additional assumptions. To the best of our knowledge, our algorithms in the bandit setting are the first efficient methods for learning with non-additive losses in this scenario.

The key technical tools used in this work are weighted automata and transducers [Mohri, 2009a]. We transform the original path graph $\mathcal{A}$ (e.g. Figure 4.1) to an intermediate graph $\mathcal{A}'$. The paths in $\mathcal{A}$ are mapped to the paths in $\mathcal{A}'$, but now the losses in $\mathcal{A}'$ are additive along the paths. Remarkably, the size of $\mathcal{A}'$ does not depend on the size of the alphabet (word vocabulary in translation tasks) from which the output labels of edges are drawn. The construction of $\mathcal{A}'$ is highly non-trivial and is our

primary contribution. The alternate graph $\mathcal{A}'$ in which the losses are additive allows us to extend many well-known algorithms in the literature to the path learning problem.

This chapter is structured as follows. We introduce the path learning setup in Section 4.1. We give an overview of weighted finite automata in Section 4.2. In Section 4.3, we explore the large family of non-additive count-based gains and introduce the alternate graph $\mathcal{A}'$ using automata and transducers tools. We introduce our algorithms for three settings of full information, semi-bandit and full bandit in Section 4.4. Then we extend our results to *gappy* count-based gains in Section 4.5. The application of our method to the ensemble structured prediction is explored in Section 4.6. Going beyond count-based gains, in Section 4.7, we give an efficient implementation of the EXP3 algorithm for the full bandit setting with *arbitrary* (non-additive) gains.

4.1 Basic Notation and Setup

We describe our path learning setup in terms of finite automata. Let $\mathcal{A}$ denote a fixed acyclic finite automaton. We call $\mathcal{A}$ the *expert automaton*. $\mathcal{A}$ admits a single *initial state* and one or several *final states* which are indicated by bold and double circles, respectively, see Figure 4.2(a). Each transition of $\mathcal{A}$ is labeled with a unique *name*. Denote the set of all transition names by E. An automaton is *deterministic* if no two outgoing transitions from a given state admit the same name, thus our automaton $\mathcal{A}$ is deterministic by construction. An *accepting path* is a sequence of transitions from the initial state to a final state. The expert automaton $\mathcal{A}$ can be viewed as an indicator

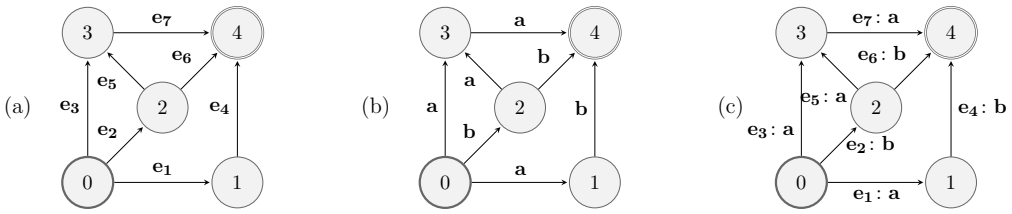


Figure 4.2: (a) The expert automaton denoted by $\mathcal{A}$ labeled with transition names. (b) The output of expert automaton at round t denoted by $\text{out}_t(\mathcal{A})$ labeled with the outputs $\text{out}_t(e)$ for each transition e. (c) The name and output of each transition together separated by a ':'.

function over strings in E^* such that $\mathcal{A}(\pi)=1$ iff π is an accepting path. Each accepting path serves as an expert and we equivalently refer to it as a *path expert*. The set of all path experts is denoted by $\mathcal{P}$.

In each round $t = 1, \ldots, T$, each transition $e \in E$ outputs a *symbol* from a finite non-empty alphabet Σ and is denoted by $\text{out}_t(e) \in \Sigma$. The *prediction* of each path expert $\pi \in E^*$ at round t is the sequence of output symbols along its transitions on that round and is denoted by $\text{out}_t(\pi) \in \Sigma^*$. Also let $\text{out}_t(\mathcal{A})$ be the automaton with the same topology as $\mathcal{A}$ where each transition e is labeled with $\text{out}_t(e)$, see Figure 4.2 (b).

At each round t, a *target sequence* $y_t \in \Sigma^*$ is presented to the learner. The *gain/loss* of each path expert π is $\mathcal{U}(\text{out}_t(\pi), y_t)$ where $\mathcal{U}\colon \Sigma^* \times \Sigma^* \longrightarrow \mathbb{R}_{\geq 0}$. Our focus is $\mathcal{U}$ functions that are not necessarily additive along the transitions in $\mathcal{A}$. For example, $\mathcal{U}$ can be either a *distance function* (e.g. edit-distance) or a *similarity function* (e.g. n-gram gain with $n \geq 2$).

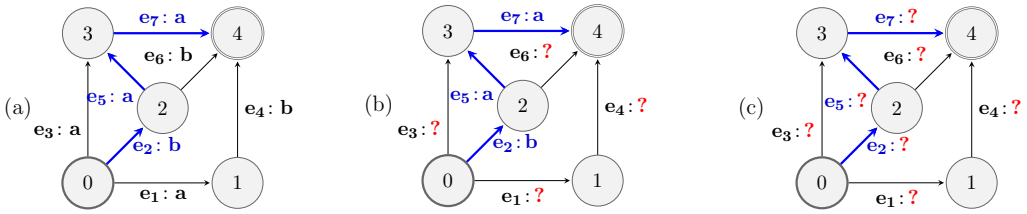


Figure 4.3: Information revealed in different settings: (a) full information (b) semi-bandit (c) full bandit. The name of each transition e and its output symbol (if revealed) are shown next to it separated by a ':'. The blue path indicates the path expert predicted by the learner at round t.

We consider standard online learning scenarios of prediction with path experts. In each round $t \in [T]$, the *learner* picks a path expert π_t and predicts with its prediction $\text{out}_t(\pi_t)$. The learner receives a gain of $\mathcal{U}(\text{out}_t(\pi_t), y_t)$. Depending on the setting, the *adversary* may reveal some information about y_t and the output symbols of the transitions (see Figure 4.3). In the *full information* setting, y_t and $\text{out}_t(e)$ for every transition e in $\mathcal{A}$ are shown to the learner. In the *semi-bandit* setting, the adversary reveals y_t and $\text{out}_t(e)$ for every transition e along π_t. In *full bandit* setting, $\mathcal{U}(\text{out}_t(\pi_t), y_t)$ is the only information that is revealed to the learner. The goal of the learner is to minimize the *regret* which is defined as the cumulative gain of the best path expert chosen in hindsight minus the cumulative expected gain of the learner.

4.2 Overview of Weighted Finite Automata

In this section, we give an overview of deterministic *Weighted Finite Automata (WFA)*. We also formally describe the properties and operations of WFAs. WFA and its machinery will be used in the later sections of this chapter.

A *deterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite set of input symbols called the alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state , and $F \subseteq Q$ is a set of final states. A *deterministic WFA* $\mathcal{W}$ is a deterministic finite automaton whose transitions and final states carry weights. Let $w(e)$ denote the weight of a transition e and $w_f(q)$ the weight at a final state $q \in F$. Recall that an *accepting path* is a path ending in a final state. The weight $\mathcal{W}(\pi)$ of an accepting path π is defined as the product of its constituent transition weights and the weight at the final state: $\mathcal{W}(\pi) := (\prod_{e \in \pi} w(e)) \cdot w_f(\text{dest}(\pi))$, where $\text{dest}(\pi)$ denotes the final state that π ends with.

Observe that in a deterministic WFA, final states can have out-going transitions. So accepting paths can pass through several final states. Thus once a path reaches a final states, it can either stay and end in that final state, or move on and continue with one of the out-going transitions.

Sampling accepting paths from a deterministic WFA $\mathcal{W}$ is straightforward when it is *stochastic*, that is when the weights of all outgoing transitions and the final state weight (if the state is final) sum to one at every state. One can start from the initial state and randomly draw a transition according to the probability distribution

defined by the outgoing transition weights and proceed similarly from the next state, until a final state is reached.

In the rest of this section, we describe two important operations of WFAs. First we go over a more general version of the *weight-pushing* algorithm [Mohri, 1997, 2009a, Takimoto and Warmuth, 2003]. Then we describe an important binary operation called *intersection* [Mohri, 2009a].

4.2.1 Weight Pushing

In Chapter 3, we gave an overview of the weight pushing algorithm for paths in a DAG and extended the algorithm for multipaths in a multi-DAG. The sink nodes (i.e. final states) in the DAG and multi-DAG did not carry any weight. In this section, we describe a more general version of the weight pushing algorithm for WFAs whose final states admit weights and may have out-going transitions.

If an acyclic WFA is not stochastic, we can efficiently compute an equivalent stochastic WFA $\mathcal{W}'$ from any $\mathcal{W}$ using the *weight-pushing* algorithm [Mohri, 1997, 2009a, Takimoto and Warmuth, 2003]: $\mathcal{W}'$ admits the same states and transitions as $\mathcal{W}$ and preserves the ratios of the weights of the accepting paths from the initial state to a final state; but the weights along the paths are redistributed so that $\mathcal{W}'$ is stochastic.

Given an acyclic WFA $\mathcal{W}$, the weight pushing algorithm [Mohri, 2009a] computes an equivalent stochastic WFA. The weight pushing algorithm is defined as follows. For any state q in $\mathcal{W}$, let $d[q]$ denote the sum of the weights of all accepting paths from

q to final states:

$$d[q] = \sum_{\pi \in \mathcal{P}(q)} \left(\prod_{e \in \pi} w(e) \right) \cdot w_f(\text{dest}(\pi)),$$

where $\mathcal{P}(q)$ denotes the set of paths from q to final states in $\mathcal{W}$. Because we have assumed that $\mathcal{W}$ is acyclic, these $d[q]$s can be computed simultaneously for all qs using a dynamic programming algorithm starting from the final states and going in the reverse direction of the topological order of $\mathcal{W}$ [Mohri, 2009a]. The weight pushing algorithm performs the following steps. For any transition $(q, q') \in E$ such that $d[q] \neq 0$, its weight is updated as below:

$$w(q, q') \leftarrow d[q]^{-1} \, w(q, q') \, d[q'].$$

For any final state q, update its weight as follows:

$$w_f(q) \leftarrow w_f(q) \, d[q]^{-1}.$$

Mohri [2009a] showed that the resulting WFA is guaranteed to preserve the ratios of the weights of the accepting paths and to be stochastic . For an acyclic input WFA such as those we are considering in this chapter, the computational complexity of weight-pushing is linear in $|\mathcal{W}|$ which is defined as the sum of the number of states and transitions of $\mathcal{W}$.

4.2.2 Intersection of WFAs

The intersection of two WFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ is a WFA denoted[1] by $\mathcal{A}_1 \circ \mathcal{A}_2$ that accepts the set of sequences accepted by both $\mathcal{A}_1$ and $\mathcal{A}_2$ and the weights of paths are

[1] The circle notation '$\circ$' may look unconventional as the notation '$\cap$' is common for "intersection". However, since this operation will be generalized to the *composition* operation in *finite state transducers* later in this chapter, we use the same notation for both.

defined for all π by[2]

$$(\mathcal{A}_1 \circ \mathcal{A}_2)(\pi) = \mathcal{A}_1(\pi) \cdot \mathcal{A}_2(\pi).$$

There exists a standard efficient algorithm for computing the intersection WFA [Mohri, 2009a]. States $Q \subseteq Q_1 \times Q_2$ of $\mathcal{A}_1 \circ \mathcal{A}_2$ are identified with pairs of states Q_1 of $\mathcal{A}_1$ and Q_2 of $\mathcal{A}_2$, as are the set of initial and final states. Transitions are obtained by matching pairs of transitions from each WFA and multiplying their weights:

$$\left(q_1 \xrightarrow{a|w_1} q_1',\ q_2 \xrightarrow{a|w_2} q_2'\right) \quad \Rightarrow \quad (q_1, q_2) \xrightarrow{a|w_1 \cdot w_2} (q_1', q_2').$$

The worst-case space and time complexity of the intersection of two deterministic WFA is linear in the size of the automaton the algorithm returns. In the worst case, this can be as large as the product of the sizes of the WFA that are intersected (i.e. $O(|\mathcal{A}_1||\mathcal{A}_2|)$. This corresponds to the case where every transition of $\mathcal{A}_1$ can be paired up with every transition of $\mathcal{A}_2$. In practice far fewer transitions can be matched.

Notice that when both $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic, then $\mathcal{A}_1 \circ \mathcal{A}_2$ is also deterministic since there is a unique initial state (pair of initial states of each WFA) and since there is at most one transition leaving $q_1 \in Q_1$ or $q_2 \in Q_2$ labeled with a given symbol $a \in \Sigma$.

[2]The terminology of intersection is motivated by the case where the weights are either 0 or 1, in which case the set of paths with non-zero weights in $\mathcal{A}_1 \circ \mathcal{A}_2$ is the intersection of the sets of paths with with weight 1 in $\mathcal{A}_1$ and $\mathcal{A}_2$.

4.3 Count-Based Gains

In many applications such as machine translation, natural language processing, speech recognition and computational biology, path gains are not additive: they cannot be expressed as a sum over some transition gains. Many of the most commonly used gains in these areas belong to the broad family of *count-based gains*. They are defined in terms of the number of occurrences of a fixed set of patterns $\mathcal{L} = \{\theta_1, \theta_2, \ldots, \theta_p\}$ in the sequence output by a path expert. These patterns may be n-grams, that is sequences of n consecutive symbols, as in a common gain approximation of the BLEU score in machine translation, a set of relevant subsequences of variable-length in computational biology, or patterns described by complex regular expressions in pronunciation modeling and other natural language processing tasks.

For any sequence $y \in \Sigma^*$, let $\Theta(y) \in \mathbb{R}^p$ denote the vector whose kth component is the number of occurrences or matches of pattern θ_k in y, $k \in [p]$. This can be extended to the case of weighted occurrences where more emphasis is assigned to some patterns θ_k whose occurrences are then multiplied by a factor $\alpha_k > 1$, and less emphasis to others. The count-based gain function $\mathcal{U}$ at round t for a path expert π in $\mathcal{A}$ given the target sequence y_t is then defined as follows:

$$\mathcal{U}(\text{out}_t(\pi), y_t) := \Theta(\text{out}_t(\pi)) \cdot \Theta(y_t) \geq 0. \tag{4.1}$$

Such gains are not additive even in the case of n-grams. Consider, for example, the special case of 4-gram-based gains for the graph of expert outputs shown in Figure 4.1. There are $|\Sigma|^4$ unique 4-grams, thus, for any path π, $\Theta(\text{out}_t(\pi))$ is in $\mathbb{R}^{|\Sigma|^4}$. In contrast,

the number of transitions is only 12. Thus, it is not hard to see that in general it is not possible to assign weights to transitions so that gain of each path is the sum of these weights along that path. Thus, standard online path learning algorithms for additive losses or gains do not apply here. The challenge of learning with non-additive gains is even more apparent in the case of *gappy* count-based gains which allow for gaps of varying length in the patterns of interest, with each gap discounted exponentially as a function of its length. We defer the study of gappy-count based gains to Section 4.5.

How can we design efficient algorithms for online path learning with such non-additive gains? Can we design algorithms with favorable regret guarantees with such non-additive gains for all three settings of full information, semi-bandit, and full bandit? The key idea behind our solution is to design a new automaton $\mathcal{A}'$ whose paths correspond the paths in $\mathcal{A}$, and crucially, $\mathcal{A}'$ admits additive gains. We will construct $\mathcal{A}'$ by defining a set of context-dependent rewrite rules, which can be compiled into a finite-state transducer $T_{\mathcal{A}}$. The *context-dependent automaton* $\mathcal{A}'$ can then be obtained by composition of the transducer $T_{\mathcal{A}}$ with $\mathcal{A}$. In addition to playing a key role in the design of our algorithms (Section 4.4), $\mathcal{A}'$ provides a compact representation of the gains since its size is substantially less that the dimension p (number of patterns).

4.3.1 Context-Dependent Rewrite Rules

We will use *context-dependent rewrite rules* to map $\mathcal{A}$ to the new representation $\mathcal{A}'$. These are rules that admit the following general form:

$$\phi \to \psi / \lambda \_\_\_ \rho,$$

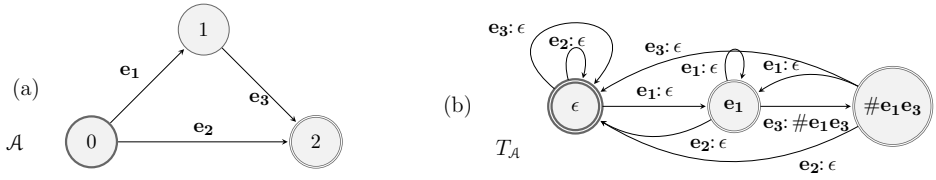


Figure 4.4: (a) An expert automaton $\mathcal{A}$; (b) associated context-dependent transducer $T_{\mathcal{A}}$ for bigrams where every state is a final state. ϵ denotes the empty string. Inputs and outputs are written next to the transitions separated by a ':'.

where ϕ, ψ, λ, and ρ are regular expressions over the alphabet of the rules. These rules must be interpreted as follows: the input ϕ is to be replaced by the output ψ whenever it is preceded by λ and followed by ρ. Thus, λ and ρ represent the left and right contexts of application of the rules. The rule fires whenever the input is matched in the given context (i.e. left and right contexts).

Several types of rules can be considered depending on their being obligatory or optional, and on their direction of application, from left to right, right to left or simultaneous application [Kaplan and Kay, 1994]. We will be only considering rules with simultaneous applications where given a set of rules, the matching and rewriting steps for all rules are done at the same time. Additionally, we assume that the outputs of the rules can be written in any order.

Such context-dependent rules can be efficiently compiled into a *finite-state transducer* (FST), under the technical condition that they do not rewrite their non-contextual part [Mohri and Sproat, 1996, Kaplan and Kay, 1994]. An FST is a finite automaton whose transitions are augmented with an output label, in addition to the

familiar input label.[3]

An FST $\mathcal{T}$ over an input alphabet Σ and output alphabet Σ' defines an indicator function over the pairs of strings in $\Sigma^* \times \Sigma'^*$. Given $x \in \Sigma^*$ and $y \in \Sigma'^*$, $\mathcal{T}(x, y) = 1$ if there exists a path from an initial state to a final state with input label x and output label y, and $\mathcal{T}(x, y) = 0$ otherwise. Figure 4.4 (b) shows an example.

To define our rules, we first introduce the alphabet E' which is the set of transition names for the target automaton $\mathcal{A}'$. These should capture all possible contexts of length r where r is the length of some pattern θ_k:

$$E' = \{\#e_1 \cdots e_r \mid e_1 \cdots e_r \text{ is a path of length } r \text{ in } \mathcal{A}, r \in \{|\theta_1|, \ldots, |\theta_p|\}\}.$$

where the '#' symbol "glues" $e_1, \cdots, e_r \in E$ together and forms one single symbol in E'. We will have one context-dependent rule of the following form for each element $\#e_1 \cdots e_r \in E'$:

$$e_1 \cdots e_r \to \#e_1 \cdots e_r / \epsilon \_\_\_ \epsilon. \tag{4.2}$$

Thus, in our case, the left- and right-contexts are the empty strings, meaning that the rules can apply (simultaneously) at every position. In the special case where the patterns θ_k are the set of n-grams, then r is fixed and equal to n. Figure 4.4 shows the result of the rule compilation in that case for $n = 2$. This transducer inserts $\#e_1e_2$ whenever e_1 and e_2 are found consecutively and otherwise outputs the empty string. We will denote the resulting FST by $T_{\mathcal{A}}$.

[3] Additionally, the rules can be augmented with weights, which can help us cover the case of weighted count-based gains, in which case the result of the compilation is a weighted transducer [Mohri and Sproat, 1996]. Our algorithms and theory can be extended to that case.

4.3.2 Context-Dependent Automaton $\mathcal{A}'$

To construct the context-dependent automaton $\mathcal{A}'$, we will use the *composition* operation. Let $T_{\mathcal{A}}$ denote the FST obtained as just described. The composition of $\mathcal{A}$ and $T_{\mathcal{A}}$ is an FST denoted[4] by $\mathcal{A} \circ T_{\mathcal{A}}$ and defined as the following product of two 0/1 outcomes for all inputs:

$$\forall x \in E^*,\ \forall y \in E'^*: \quad (\mathcal{A} \circ T_{\mathcal{A}})(x, y) := \mathcal{A}(x) \cdot T_{\mathcal{A}}(x, y).$$

There is an efficient algorithm for the composition of FSTs and automata [Pereira and Riley, 1997, Mohri et al., 1996, Mohri, 2009a], whose worst-case complexity is in $O(|\mathcal{A}|\,|T_{\mathcal{A}}|)$. The automaton $\mathcal{A}'$ is obtained from the FST $(\mathcal{A} \circ T_{\mathcal{A}})$ by *projection*, that is by simply omitting the input label of each transition and keeping only the output label. Thus, if we denote by Π the projection operator, $\mathcal{A}'$ is defined as $\mathcal{A}' = \Pi(\mathcal{A} \circ T_{\mathcal{A}})$.

Observe that $\mathcal{A}'$ admits a fixed topology (states and transitions) over all rounds $t \in [T]$. It can be constructed in a pre-processing stage using the FST operations of composition and projection. Additional FST operations such as ϵ-removal and minimization can help further optimize the automaton obtained after projection [Mohri, 2009a].

Notice that the transducer $T_{\mathcal{A}}$ of Figure 4.4(b) is *deterministic*, that is no two transitions leaving a state admit the same input label. In general, when the patterns θ_k are n-grams, $T_{\mathcal{A}}$ can be constructed to be deterministic without increasing its size: the number of transitions coincides with the number of elements in E'. More generally,

[4] We purposefully use the same symbol '$\circ$' for both intersection and composition operations as the latter is a generalization of the former [Mohri, 2009a].

the result of the rule compilation can be determinized using transducer determinization [Mohri, 2009a]. We can therefore assume in the following that $T_{\mathcal{A}}$ is deterministic. In that case, $T_{\mathcal{A}}$ assigns a unique output to a given path expert in $\mathcal{A}$. Proposition 16 ensures that for every accepting path π in $\mathcal{A}$, there is a unique corresponding accepting path in $\mathcal{A}'$. Figure 4.5 shows the automata $\mathcal{A}$ and $\mathcal{A}'$ in a simple case and how a path π in $\mathcal{A}$ is mapped to another path π' in $\mathcal{A}'$.

Proposition 16. *Let $\mathcal{A}$ be an expert automaton and let $T_{\mathcal{A}}$ be a deterministic transducer representing the rewrite rules (4.2). Then for each accepting path π in $\mathcal{A}$ there exists a unique corresponding accepting path π' in $\mathcal{A}' = \Pi(\mathcal{A} \circ T_{\mathcal{A}})$.*

Proof. To establish the correspondence, we introduce $T_{\mathcal{A}}$ as a mapping from the accepting paths in $\mathcal{A}$ to the accepting paths in $\mathcal{A}'$. Since $T_{\mathcal{A}}$ is deterministic, for each accepting path π in $\mathcal{A}$ (i.e. $\mathcal{A}(\pi) = 1$), $T_{\mathcal{A}}$ assigns a unique output π', that is $T_{\mathcal{A}}(\pi, \pi') = 1$. We show that π' is an accepting path in $\mathcal{A}'$. Observe that

$$(\mathcal{A} \circ T_{\mathcal{A}})(\pi, \pi') = \mathcal{A}(\pi) \cdot T_{\mathcal{A}}(\pi, \pi') = 1 \times 1 = 1,$$

which implies that $\mathcal{A}'(\pi') = \Pi(\mathcal{A} \circ T_{\mathcal{A}})(\pi') = 1$. Thus for each accepting path π in $\mathcal{A}$ there is a unique accepting path π' in $\mathcal{A}'$. ■

The size of the context-dependent automaton $\mathcal{A}'$ depends on the expert automaton $\mathcal{A}$ and the set of the lengths of the patterns. Notice that, crucially, its size is independent of the size of the alphabet Σ. Section 4.6 analyzes more specifically the size of $\mathcal{A}'$ in the important application of ensemble structure prediction with n-gram gains. In this application, the size of $\mathcal{A}'$ is exponential in terms of n. However, since n

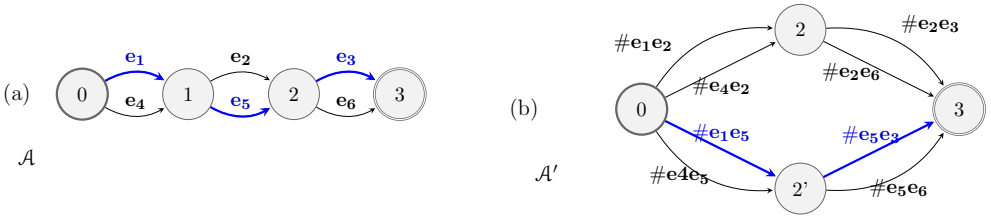


Figure 4.5: (a) An example of the expert automaton $\mathcal{A}$. (b) the associated context-dependent automaton $\mathcal{A}'$ with bigrams as patterns. The path $\pi = e_1e_5e_3$ in $\mathcal{A}$ and its corresponding path $\pi' = \#e_1e_5\#e_5e_3$ in $\mathcal{A}'$ are marked in blue.

usually takes small values in many real world applications (i.e. $n \leq 5$), the size of $\mathcal{A}'$ is essentially polynomial in terms of the size of $\mathcal{A}$.

Recall that the standard online learning scenario consists of T rounds. In each round $t \in [T]$, the learner picks a path expert π_t in $\mathcal{A}$, predicts with its prediction $\text{out}_t(\pi_t)$, and receives a gain of $\mathcal{U}(\text{out}_t(\pi_t), y_t)$, where y_t is the target sequence. We define an equivalent online learning scenario on $\mathcal{A}'$.

At any round $t \in [T]$ and for any $\#e_1 \cdots e_r \in E'$, let $\text{out}_t(\#e_1 \cdots e_r)$ denote the sequence $\text{out}_t(e_1) \cdots \text{out}_t(e_r)$, that is the sequence obtained by concatenating the outputs of $e_1, \ldots, e_r$. Let $\text{out}_t(\mathcal{A}')$ be the automaton with the same topology as $\mathcal{A}'$ where each label $e' \in E'$ is replaced by $\text{out}_t(e')$. The gain of each transition in $\mathcal{A}'$ can be computed based on the target sequence y_t. Once y_t is known, the representation $\Theta(y_t)$ can be found, and consequently, the additive contribution of each transition of $\mathcal{A}'$. The following theorem proves the additivity of the gains in $\mathcal{A}'$.

Theorem 17. *At any round $t \in [T]$, define the gain $g_{e',t}$ of the transition $e' \in E'$ in $\mathcal{A}'$ by $g_{e',t} := [\Theta(y_t)]_k$ if $out_t(e') = \theta_k$ for some $k \in [p]$ and $g_{e',t} := 0$ if no such k exists.*

Then, the gain of each path π in $\mathcal{A}$ at trial t can be expressed as an additive gain of the corresponding unique path π' in $\mathcal{A}'$:

$$\forall t \in [T],\ \forall \pi \in \mathcal{P}: \quad \mathcal{U}(out_t(\pi), y_t) = \sum_{e' \in \pi'} g_{e',t}\ .$$

Proof. By definition, the ith component of $\Theta(\text{out}_t(\pi))$ is the number of occurrences of θ_i in $\text{out}_t(\pi)$. Also, by construction of the context-dependent automaton based on rewrite rules, π' contains all path segments of length $|\theta_i|$ of π in $\mathcal{A}$ as transition labels in $\mathcal{A}'$. Thus every occurrence of θ_i in $\text{out}_t(\pi)$ will appear as a transition label in $\text{out}_t(\pi')$. Therefore the number of occurrences of θ_i in $\text{out}_t(\pi)$ is

$$[\Theta(\text{out}_t(\pi))]_i = \sum_{e' \in \pi'} \mathbf{1}\{\text{out}_t(e') = \theta_i\}, \tag{4.3}$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. Thus, we have that

$$\begin{aligned}
\mathcal{U}(\text{out}_t(\pi), y_t) &= \Theta(y_t) \cdot \Theta(\text{out}_t(\pi)) && \text{(definition of } \mathcal{U}) \\
&= \sum_i [\Theta(y_t)]_i \, [\Theta(\text{out}_t(\pi))]_i \\
&= \sum_i [\Theta(y_t)]_i \sum_{e' \in \pi'} \mathbf{1}\{\text{out}_t(e') = \theta_i\} && \text{(Equation (4.3))} \\
&= \sum_{e' \in \pi'} \underbrace{\sum_i [\Theta(y_t)]_i \, \mathbf{1}\{\text{out}_t(e') = \theta_i\}}_{= g_{e',t}},
\end{aligned}$$

which concludes the proof. ■

As an example, consider the automaton $\mathcal{A}$ and its associated context-dependent automaton $\mathcal{A}'$ shown in Figure 4.5, with bigram gains and $\Sigma = \{a, b\}$. Here, the patterns are $(\theta_1, \theta_2, \theta_3, \theta_4) = (aa, ab, ba, bb)$. Let the target sequence at trial t be $y_t = aba$. Thus $\Theta(y_t) = [0, 1, 1, 0]^T$. The automata $\text{out}_t(\mathcal{A})$ and $\text{out}_t(\mathcal{A}')$ are given in Figure 4.6.

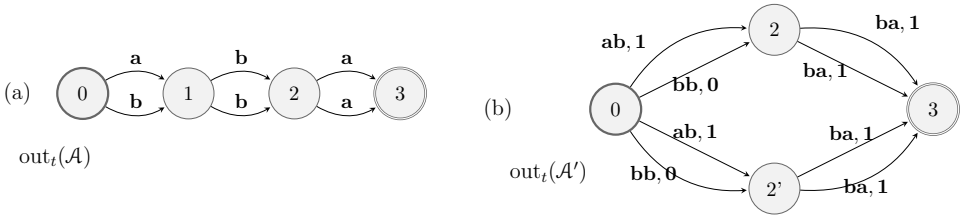


Figure 4.6: (a) the automaton $\text{out}_t(\mathcal{A})$ of $\mathcal{A}$ in Figure 4.5(a), with bigram gains and $\Sigma = \{a, b\}$. (b) the automaton $\text{out}_t(\mathcal{A}')$ given $y_t = aba$. The additive gain contributed by each transition $e' \in E'$ in $\mathcal{A}'$ is written on it separated by a comma from $\text{out}_t(e')$.

4.4 Algorithms

In this section, we present algorithms and associated regret guarantees for online path learning with non-additive count-based gains in the full information, semi-bandit and full bandit settings. The key component of our algorithms is the context-dependent automaton $\mathcal{A}'$.

In what follows, we denote the length of the longest path in $\mathcal{A}'$ by K, an upper-bound on the gain of each transition in $\mathcal{A}'$ by B, the number of path experts by N, and the number of transitions and states in $\mathcal{A}'$ by M and Q, respectively. We note that K is at most the length of the longest path in $\mathcal{A}$ since each transition in $\mathcal{A}'$ admits a unique label.

Remark. The number of accepting paths in $\mathcal{A}'$ could be equal or less than the number of accepting paths in $\mathcal{A}$. In some degenerate cases, several paths $\pi_1, \ldots, \pi_k$ in $\mathcal{A}$ may correspond to one single path π' in $\mathcal{A}'$. This implies that $\pi_1, \ldots, \pi_k$ in $\mathcal{A}$ will always consistently have the same gain in every round and that is the additive gain of π' in

$\mathcal{A}'$. Thus, if π' is predicted by the algorithm in $\mathcal{A}'$, either of the paths $\pi_1, \ldots, \pi_k$ can be equivalently used for prediction in the original expert automaton $\mathcal{A}$.

4.4.1 Full Information: Context-dependent Component Hedge Algorithm

In the full information setting, Koolen et al. [2010] gave an algorithm for online path learning with non-negative additive losses, the Component Hedge (CH) algorithm, that admits the tightest regret guarantees guarantees in terms of the relevant combinatorial parameters of the problem: B, N, M, Q, K. For count-based losses, Cortes et al. [2015b] provided an efficient Rational Randomized Weighted Majority (RRWM) algorithm. This algorithm requires the use of determinization which is only shown to have polynomial computational complexity under some additional technical assumptions on the outputs of the path experts. In this section, we present an extension of CH, the *Context-dependent Component Hedge* (CDCH), for the online path learning problem with non-additive count-based gains. CDCH has better regret guarantees than RRWM (i.t.o. problem parameters) and can be efficiently implemented without any additional assumptions.

Our CDCH algorithm requires a modification of $\mathcal{A}'$ such that all paths admit an equal number K of transitions (same as the longest path). This modification can be done by adding at most $(K-2)(Q-2)+1$ states and zero-gain transitions [György et al., 2007]. Abusing the notation, we will denote this new automaton by $\mathcal{A}'$ in this subsection. At each iteration t, CDCH maintains a weight vector $\boldsymbol{w}_t$ in the unit-flow

polytope P over $\mathcal{A}'$, which is a set of vectors $\boldsymbol{w} \in \mathbb{R}^M$ such that

$$\sum_q w(s,q) = 1 \quad \text{and for all} \quad q \in Q', \sum_{q':(q',q)\in E'} w(q',q) = \sum_{q':(q,q')\in E'} w(q,q'),$$

where s denotes the initial state of $\mathcal{A}'$. For each $t \in \{1,\ldots,T\}$, we observe the gain of each transition $g_{t,e'}$, and define the loss of that transition as $\ell_{e'} = B - g_{t,e'}$. After observing the loss of each transition e' in $\mathcal{A}'$, CDCH updates each component of $\boldsymbol{w}$ as $\widehat{w}(e') \leftarrow w_t(e') \exp(-\eta\, \ell_{t,e'})$ (where η is a specified learning rate), and sets $\boldsymbol{w}_{t+1}$ to the relative entropy projection of the updated $\widehat{\boldsymbol{w}}$ back to the unit-flow polytope by solving the following convex optimization problem:

$$\boldsymbol{w}_{t+1} = \operatorname*{argmin}_{\boldsymbol{w}\in P} \sum_{e'\in E'} \left(w(e') \ln \frac{w(e')}{\widehat{w}(e')} + \widehat{w}(e') - w(e') \right).$$

CDCH predicts by decomposing $\boldsymbol{w}_t$ into a convex combination of at most $|E'|$ paths in $\mathcal{A}'$ and then sampling a single path according to this mixture as described below. Recall that a path $\mathcal{A}'$ uniquely identifies a path in $\mathcal{A}$ which can be recovered in time K. Therefore, the inference step of CDCH algorithm takes at most time polynomial in $|E'|$ steps. To determine a decomposition, we find a path from the initial state to a final state with non-zero weights on all transitions, remove the largest weight on that path from each transition on that path and use it as a mixture weight for that path. The algorithm proceeds in this way until the outflow from initial state is zero. The following theorem gives a regret guarantee for CDCH algorithm.

Theorem 18 (Koolen et al. [2010])**.** *With proper tuning, the regret of CDCH is bounded*

as below:

$$\forall\, \pi^* \in \mathcal{P}: \quad \sum_{t=1}^{n} \mathcal{U}(out_t(\pi^*), y_t) - \mathcal{U}(out_t(\pi_t), y_t) \leq \sqrt{2\,T\,B^2K^2\,\log(K\,M)} + B\,K\,\log(KM).$$

The regret bounds of Theorem 18 are in terms of the count-based gain $\mathcal{U}(\cdot,\cdot)$. Cortes et al. [2015b] gave regret guarantees for RRWM algorithm with count-based losses defined by $-\log \mathcal{U}(\cdot,\cdot)$. In Section 4.4.4, we show that the regret associated with $-\log \mathcal{U}$ is upper-bounded by the regret bound associated with $\mathcal{U}$. Observe that, even with this approximation, the regret guarantees that we provide for CDCH are tighter by a factor of K. In addition, our algorithm does not require additional assumptions for an efficient implementation compared to RRWM algorithm of Cortes et al. [2015b].

4.4.2 Semi-Bandit: Context-dependent Semi-Bandit Algorithm

György et al. [2007] presented an efficient algorithm for online path learning with additive losses. In this section, we present a *Context-dependent Semi-Bandit* (CDSB) algorithm extending that work to solving the problem of online path learning with count-based gains in a semi-bandit setting. To the best of our knowledge, this is the first efficient algorithm for this problem.

As the algorithm of György et al. [2007], CDSB makes use of a set C of *covering paths* with the property that, for each $e' \in E'$, there is an accepting path π in C such that e' belongs to π. At each round t, CDSB keeps track of a distribution p_t over all N path experts by maintaining a weight $w_t(e')$ on each transition e' in $\mathcal{A}'$ such that for each $q \in Q'$, $\sum_{q':\,(q,q')\in E'} w_t(q,q') = 1$, $w(q,q') \geq 0$ for all $q, q' \in Q'$ and $p_t(\pi) = \prod_{e'\in\pi} w_t(e')$,

for all accepting paths π in $\mathcal{A}'$. Therefore, the WFA $\mathcal{A}'$ is stochastic and we can sample a path π in at most K steps by selecting a random transition at each state according to the distribution defined by $\boldsymbol{w}_t$ at that state.

Once a path π'_t in $\mathcal{A}'$ is sampled, we observe the gain of the corresponding path π_t in $\mathcal{A}$, which define the gain along each transition e' of π'_t that are denoted by $g_{t,e'}$. CDSB sets $\widehat{w}(e') = w_t(e') \exp(\eta \widetilde{g}_{t,e'})$, where $\widetilde{g}_{t,e'} = (g_{t,e'} + \beta)/q_{t,e'}$ if $e' \in \pi'_t$ and $\widetilde{g}_{t,e'} = \beta/q_{t,e'}$ otherwise. Here, $\eta, \beta > 0$ and $\gamma \in [0, 1]$ are parameters of the algorithm and $q_{t,e'}$ is the flow through e' in $\mathcal{A}'$, which can be computed using a standard shortest-distance algorithm over the probability semiring [Mohri, 2009a]. Finally, we use the weight-pushing algorithm [Mohri, 1997] to turn $\mathcal{A}'$ into an equivalent stochastic WFA with weights $\boldsymbol{w}_{t+1}$. The computational complexity of each of the steps above is polynomial in the size of $\mathcal{A}'$. The following theorem provides a regret guarantee for CDSB algorithm. This result provides the first favorable regret guarantee for online path learning with (gappy) count-based gains in semi-bandit setting.

Theorem 19 (György et al. [2007])**.** *Let C denote the set of "covering paths" in $\mathcal{A}'$. For any $\delta \in (0, 1)$, with proper tuning, the regret of the CDSB algorithm can be bounded, with probability $1 - \delta$, as:*

$$\forall\, \pi^* \in \mathcal{P}: \quad \sum_{t=1}^{n} \mathcal{U}(out_t(\pi^*), y_t) - \mathcal{U}(out_t(\pi_t), y_t) \leq 2\, B \sqrt{TK} \left(\sqrt{4K|C| \ln N} + \sqrt{M \ln \frac{M}{\delta}} \right).$$

4.4.3 Full Bandit: Context-dependent ComBand Algorithm

Here we present an algorithm for online path learning with count-based gains in the full bandit setting. Cesa-Bianchi and Lugosi [2012] gave an algorithm for online path

learning with additive gains, ComBand. Our generalization, called *Context-dependent ComBand* (CDCB), is the first efficient algorithm with good regret guarantees for learning with count-based gains in this setting.

As with CDSB, CDCB maintains a distribution p_t over all N path experts by using a stochastic WFA $\mathcal{A}'$ with weights $\boldsymbol{w}_t$ on the transitions. To make a prediction, we sample a path in $\mathcal{A}'$ according to a mixture distribution $q_t = (1-\gamma)p_t + \gamma\mu$, where μ is a uniform distribution over path in $\mathcal{A}'$. Note that this sampling can be efficiently implemented as follows. Define a WFA $\mathcal{A}''$ with the same topology and transition names as $\mathcal{A}'$ and weight of one on each transition of $\mathcal{A}''$ and apply the weight-pushing algorithm to obtain an equivalent stochastic automaton. $\mathcal{A}''$ defines a uniform distribution of the set of path experts $\mathcal{P}$. These steps can be carried out offline before running CDCB. Next, we select $\mathcal{A}'$ with probability $1-\gamma$ or $\mathcal{A}''$ with probability γ and sample a random path π from the randomly chosen stochastic weighted automaton.

After observing the scalar gain g_π of the chosen path, CDCB computes a surrogate gain of each transition e' in $\mathcal{A}'$ via $\widetilde{g}_{t,e'} = g_\pi P \boldsymbol{v}_\pi$, where P is the pseudo-inverse of $\mathbb{E}[\boldsymbol{v}_\pi \boldsymbol{v}_\pi^T]$ and $\boldsymbol{v}_\pi \in \{0,1\}^M$ is a bit representation of the path π. As for CDSB, we set $\widehat{w}(e') = w_t(e')\exp(-\eta\widetilde{g}_{t,e'})$ and update $\mathcal{A}'$ via weighted-pushing to compute $\boldsymbol{w}_{t+1}$. We obtain the following regret guarantees for CDCB:

Theorem 20 (Cesa-Bianchi and Lugosi [2012])**.** *Let λ_{min} denote the smallest non-zero eigenvalue of $\mathbb{E}[\boldsymbol{v}_\pi \boldsymbol{v}_\pi^T]$ where $\boldsymbol{v}_\pi \in \{0,1\}^M$ is the bit representation of the path π which is distributed according to the uniform distribution μ. With proper tuning, the regret of*

CDCB can be bounded as follows:

$$\forall\, \pi^* \in \mathcal{P}: \quad \sum_{t=1}^{n} \mathcal{U}(out_t(\pi^*), y_t) - \mathcal{U}(out_t(\pi_t), y_t) \leq 2\, B \sqrt{\left(\frac{2K}{M\lambda_{min}} + 1\right) TM \ln N}.$$

4.4.4 Gains $\mathcal{U}$ vs Losses $-\log(\mathcal{U})$

In some context, the count-based losses are defined by $-\log \mathcal{U}(\cdot,\cdot)$. For instance, Cortes et al. [2015b] gave regret guarantees for RRWM algorithm with count-based losses defined by $-\log \mathcal{U}(\cdot,\cdot)$. Here we show that the regret associated with $-\log \mathcal{U}$ is upper-bounded by the regret bound associated with $\mathcal{U}$.

Let $\mathcal{U}$ be a non-negative gain function. Also let $\pi^* \in \mathcal{P}$ be the best comparator over the T rounds. The regret associated with $\mathcal{U}$ and $-\log \mathcal{U}$, which are respectively denoted by R_G and R_L, are defined as below:

$$R_G := \sum_{t=1}^{T} \mathcal{U}(\text{out}_t(\pi^*), y_t) - \mathcal{U}(\text{out}_t(\pi_t), y_t),$$
$$R_L := \sum_{t=1}^{T} -\log(\mathcal{U}(\text{out}_t(\pi_t), y_t)) - (-\log(\mathcal{U}(\text{out}_t(\pi^*), y_t))).$$

Observe that if $\mathcal{U}(\text{out}_t(\pi_t), y_t) = 0$ for any t, then R_L is unbounded. Otherwise, let us assume that there exists a positive constant $\alpha > 0$ such that $\mathcal{U}(\text{out}_t(\pi_t), y_t) \geq \alpha$ for all $t \in [1, T]$. Note for count-based gains, $\alpha \geq 1$, since all components of the representation $\Theta(\cdot)$ are non-negative integers. Therefore, next proposition shows that for count-based gains we have $R_L \leq R_G$.

Proposition 21. *Let $\mathcal{U}$ be a non-negative gain function. Assume there exists a positive constant $\alpha > 0$ such that $\mathcal{U}(out_t(\pi_t), y_t) \geq \alpha$ for all $t \in [1, T]$. Then $R_L \leq \frac{1}{\alpha} R_G$.*

Proof. The following chain of inequalities hold:

$$
\begin{aligned}
R_L &= \sum_{t=1}^{T} -\log(\mathcal{U}(\text{out}_t(\pi_t), y_t)) - (-\log(\mathcal{U}(\text{out}_t(\pi^*), y_t))) \\
&= \sum_{t=1}^{T} \log(\frac{\mathcal{U}(\text{out}_t(\pi^*), y_t)}{\mathcal{U}(\text{out}_t(\pi_t), y_t)}) \\
&= \sum_{t=1}^{T} \log(1 + \frac{\mathcal{U}(\text{out}_t(\pi^*), y_t) - \mathcal{U}(\text{out}_t(\pi_t), y_t)}{\mathcal{U}(\text{out}_t(\pi_t), y_t)}) \\
&\leq \sum_{t=1}^{T} \frac{\mathcal{U}(\text{out}_t(\pi^*), y_t) - \mathcal{U}(\text{out}_t(\pi_t), y_t)}{\mathcal{U}(\text{out}_t(\pi_t), y_t)} && (\text{since } \log(1+x) \leq x) \\
&\leq \frac{1}{\alpha} \sum_{t=1}^{T} \mathcal{U}(\text{out}_t(\pi^*), y_t) - \mathcal{U}(\text{out}_t(\pi_t), y_t) && (\text{since } \mathcal{U}(\text{out}_t(\pi_t), y_t) \geq \alpha) \\
&\leq \frac{1}{\alpha} R_G,
\end{aligned}
$$

and the proof is complete. ■

4.5 Extension to Gappy Count-Based Gains

We extend the results in Section 4.3 to a larger family of non-additive gains called *gappy count-based gains*: the gain of each path depends on the *discounted* counts of *gappy* occurrences of the members of a finite language $\mathcal{L} = \{\theta_1, \ldots, \theta_p\} \subset \Sigma^*$ of output symbols Σ along that path. In a gappy occurrence, there can be "gaps" between symbols of the pattern. The count of a gappy occurrence will be discounted multiplicatively by γ^k where $\gamma \in [0, 1]$ is a fixed *discount rate* and k is the total length of gaps. For example, the gappy occurrences of the pattern $\theta = aab$ in a sequence $y = babbaabaa$ with discount rate γ are

- $b\,a\,b\,b\,\underline{a\,a\,b}\,a\,a$, length of gap = 0, discount factor = 1

- $b\,\underline{a}\,b\,b\,\underline{a}\,a\,\underline{b}\,a\,a$, length of gap = 3, discount factor = γ^3
- $b\,\underline{a}\,b\,b\,a\,\underline{a}\,\underline{b}\,a\,a$, length of gap = 3, discount factor = γ^3

which makes the total discounted count of gappy occurrences of θ in y to be $1 + 2 \cdot \gamma^3$. Each sequence of symbols $y \in \Sigma^*$ can be represented as a discounted count vector $\Theta(y) \in \mathbb{R}^p$ of gappy occurrences of the patterns whose ith component is "the discounted number of gappy occurrences of θ_i in y". The gain function $\mathcal{U}$ is defined in the same way[5] as in Equation (4.1). A typical instance of such gains is gappy n-gram gains where $\mathcal{L}$ is the set of all n-grams consisting of $|\Sigma|^n$ many patterns (e.g. bigrams $\mathcal{L} = \{aa, ab, ba, bb\}$ for $\Sigma = \{a, b\}$ and $n = 2$).

The key to extending our results in Section 4.3 to gappy n-grams is an appropriate definition of the alphabet E', the rewrite rules, and a new context-dependent automaton $\mathcal{A}'$. Once $\mathcal{A}'$ is constructed, the algorithms and regret guarantees presented in Section 4.4 can be extended to gappy count-based gains. As far as we know, this provides the first efficient online algorithms with good regret guarantees for gappy count-based gains in full information, semi-bandit and full bandit settings.

Context-Dependent Rewrite Rules. We extend the definition of E' so that it also encodes the total length k of the gaps: For every θ_i of length $|\theta_i|$ and non-negative integers k, we introduce elements $(\#e_1 e_2 \dots e_{|\theta_i|})_k$ where $e_1, e_2, \dots, e_{|\theta_i|}$ represent all possible elements in E:

$$E' = \Big\{ (\#e_1 \cdots e_r)_k \mid e_1 \cdots e_r \in E,\ r \in \{|\theta_1|, \dots, |\theta_p|\},\ k \in \mathbb{Z},\ k \geq 0 \Big\}.$$

[5] The regular count-based gain can be recovered by setting $\gamma = 0$.

Note that the discount factor in gappy occurrences does not depend on position of the gaps. Exploiting this fact, for each pattern of length n and total gap length k, we save $\binom{k+n-2}{k}$ times less output symbols by encoding the *number* of gaps as opposed to encoding the *positions* of the gaps.

Next, we extend the rewrite rules in order to incorporate the gappy occurrences. Given $e' = (\#e_{i_1}e_{i_2}\dots e_{i_n})_k$, for all path segments $e_{j_1}e_{j_2}\dots e_{j_{n+k}}$ of length $n+k$ in $\mathcal{A}$ where $\{i_s\}_{s=1}^{n}$ is a subsequence of $\{j_r\}_{r=1}^{n+k}$ with the same initial and final elements (i.e. $i_1 = j_1$ and $i_n = j_{n+k}$):

$$e_{j_1}e_{j_2}\dots e_{j_{n+k}} \longrightarrow (\#e_{i_1}e_{i_2}\dots e_{i_n})_k/\epsilon\_\_\epsilon.$$

Similar to the non-gappy case in Section 4.3, the simultaneous application of all these rewrite rules can be efficiently compiled into a FST $T_{\mathcal{A}}$. The context-dependent transducer $T_{\mathcal{A}}$ maps any sequence of transition names in E, indicating a path segment in $\mathcal{A}$, into a sequence of corresponding gappy occurrences. The example below shows how $T_{\mathcal{A}}$ outputs the gappy trigrams given a path segment of length 5 as input:

$$\begin{aligned} e_1, e_2, e_3, e_4, e_5 \xrightarrow{T_{\mathcal{A}}} & (\#e_1e_2e_3)_0, (\#e_2e_3e_4)_0, (\#e_3e_4e_5)_0, \\ & (\#e_1e_2e_4)_1, (\#e_1e_3e_4)_1, (\#e_2e_3e_5)_1, (\#e_2e_4e_5)_1, \\ & (\#e_1e_2e_5)_2, (\#e_1e_4e_5)_2, (\#e_1e_3e_5)_2. \end{aligned}$$

The Context-Dependent Automaton. Similar to Section 4.3.2, we construct the context-dependent automaton as $\mathcal{A}' := \Pi(\mathcal{A} \circ T_{\mathcal{A}})$ which has a fixed topology through trials. Note that the rewrite rules are constructed in such way that different paths in $\mathcal{A}$

and rewritten differently. Therefore $T_{\mathcal{A}}$ assigns a unique output to a given path expert in $\mathcal{A}$. Proposition 16 ensures that for every accepting path π in $\mathcal{A}$, there is a unique corresponding accepting path in $\mathcal{A}'$.

At any round $t \in [T]$ and for any $e' = (\#e_{i_1} e_{i_2} \dots e_{i_n})_k$, define

$$\mathrm{out}_t(e') := \mathrm{out}_t(e_{i_1}) \dots \mathrm{out}_t(e_{i_n}).$$

Let $\mathrm{out}_t(\mathcal{A}')$ be the automaton with the same topology as $\mathcal{A}'$ where each label $e' \in E'$ is replaced by $\mathrm{out}_t(e')$. Given y_t, the representation $\Theta(y_t)$ can be found, and consequently, the additive contribution of each transition of $\mathcal{A}'$. Theorem 22 proves the additivity of the gains in $\mathcal{A}'$.

Theorem 22. *Given the trial t and discount rate $\gamma \in [0,1]$, for each transition $e' \in E'$ in $\mathcal{A}'$ define the gain $g_{e',t} := \gamma^k [\Theta(y_t)]_i$ if $out_t(e') = (\theta_i)_k$ for some i and k and $g_{e',t} := 0$ if no such i and k exist. Then the gain of each path π in $\mathcal{A}$ at trial t can be expressed as an additive gain of π' in $\mathcal{A}'$:*

$$\forall t \in [1,T],\ \forall \pi \in \mathcal{P}: \quad \mathcal{U}(out_t(\pi), y_t) = \sum_{e' \in \pi'} g_{e',t}\ .$$

Proof. By definition, the ith component of $\Theta(\mathrm{out}_t(\pi))$ is the discounted count of gappy occurrences of θ_i in $\mathrm{out}_t(\pi)$. Also, by construction of the context-dependent automaton based on rewrite rules, π' contains all gappy path segments of length $|\theta_i|$ of π in $\mathcal{A}$ as transition labels in $\mathcal{A}'$. Thus every gappy occurrence of θ_i with k gaps in $\mathrm{out}_t(\pi)$ will appear as a transition label $(\theta_i)_k$ in $\mathrm{out}_t(\pi')$. Therefore the discounted counts of gappy

occurrences of θ_i in $\text{out}_t(\pi)$ is

$$[\Theta(\text{out}_t(\pi))]_i = \sum_{e' \in \pi'} \sum_k \gamma^k \, \mathbf{1}\{\text{out}_t(e') = (\theta_i)_k\}. \tag{4.4}$$

Therefore, the following holds:

$$\begin{aligned}
\mathcal{U}(\text{out}_t(\pi), y_t) &= \Theta(y_t) \cdot \Theta(\text{out}_t(\pi)) && \text{(definition of } \mathcal{U}) \\
&= \sum_i [\Theta(y_t)]_i \, [\Theta(\text{out}_t(\pi))]_i \\
&= \sum_i [\Theta(y_t)]_i \sum_{e' \in \pi'} \sum_k \gamma^k \, \mathbf{1}\{\text{out}_t(e') = (\theta_i)_k\} && \text{(Equation (4.4))} \\
&= \sum_{e' \in \pi'} \underbrace{\sum_i \sum_k \gamma^k \, [\Theta(y_t)]_i \, \mathbf{1}\{\text{out}_t(e') = (\theta_i)_k\}}_{= g_{e',t}},
\end{aligned}$$

and the proof is complete. ■

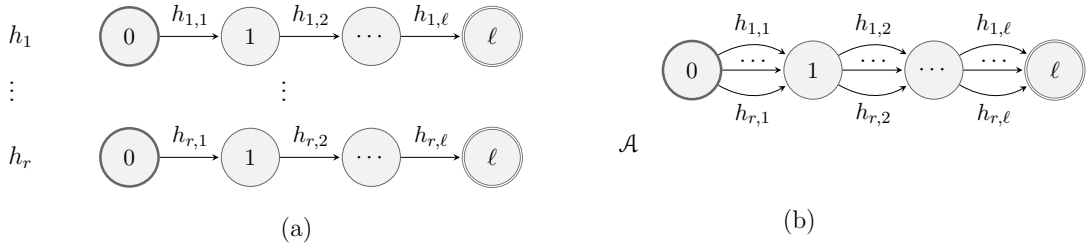
We now can extend the algorithms and regret guarantees presented in Section 4.3 to gappy count-based gains. As far as we know, this provides the first efficient online algorithms with good regret guarantees for gappy count-based gains in full information, semi-bandit and full bandit settings.

4.6 Applications to Ensemble Structured Prediction

The algorithms discussed in Section 4.4 can be used for the online learning of ensembles of structured prediction experts, and as a result, significantly improve the performance of algorithms in a number of areas including machine translation, speech recognition, other language processing areas, optical character recognition, and computer vision. In structured prediction problems, the output associated with a model

h is a structure y that can be decomposed and represented by ℓ substructures $y_1, \ldots, y_\ell$. For instance, h may be a machine translation system and y_i a particular word.

The problem of ensemble structured prediction can be described as follows. The learner has access to a set of r experts $h_1, \ldots, h_r$ to make an ensemble prediction. Therefore, at each round $t \in [1, T]$, the learner can use the outputs of the r experts $\text{out}_t(h_1), \ldots, \text{out}_t(h_r)$. As illustrated in Figure 4.7(a), each expert h_j consists of ℓ substructures $h_j = (h_{j,1}, \ldots, h_{j,\ell})$.



(a) (b)

Figure 4.7: (a) the structured experts $h_1, \ldots, h_r$. (b) the expert automaton $\mathcal{A}$ allowing all combinations.

Represented by paths in an automaton, the substructures of these experts can be *combined together*. Allowing all combinations, Figure 4.7(b) illustrates the expert automaton $\mathcal{A}$ induced by r structured experts with ℓ substructures. The objective of the learner is to find the best path expert which is the combination of substructures with the best expected gain. This is motivated by the fact that one particular expert may be better at predicting one substructure while some other expert may be more accurate at predicting another substructure. Therefore, it is desirable to combine the substructure predictions of all experts to obtain the more accurate prediction.

Consider the online path learning problem with expert automaton $\mathcal{A}$ in Figure 4.7(b) with non-additive n-gram gains described in Section 4.3 for typical small values of n (e.g. $n = 4$). We construct the context-dependent automaton $\mathcal{A}'$ via a set of rewrite rules. The rewrite rules are as follows:

$$h_{j_1,i+1},\ h_{j_2,i+2},\ \ldots, h_{j_n,i+n}\ \rightarrow \#h_{j_1,i+1}h_{j_2,i+2}\ldots h_{j_n,i+n}\,/\,\epsilon\_\_\epsilon,$$

for all $j_1, \ldots, j_n \in [1, r]$, $i \in [0, \ell - n]$. The number of rewrite rules is $(\ell - n + 1)\, r^n$. We compile these rewrite rules into the context-dependent transducer $T_{\mathcal{A}}$, and then construct the context-dependent automaton $\mathcal{A}' = \Pi(\mathcal{A} \circ T_{\mathcal{A}})$.

The context-dependent automaton $\mathcal{A}'$ is illustrated in Figure 4.8. The transitions in $\mathcal{A}'$ are labeled with n-grams of transition names $h_{i,j}$ in $\mathcal{A}$. The context-dependent automaton $\mathcal{A}'$ has $\ell - n + 1$ layers of states each of which acts as a "memory" indicating the last observed $(n-1)$-gram of transition names $h_{i,j}$. With each intermediate state (i.e. a state which is neither the initial state nor a final state), a $(n-1)$-gram is associated. Each layer contains r^{n-1} many states encoding all combinations of $(n-1)$-grams ending at that state. Each intermediate state has r incoming transitions which are the n-grams ending with $(n-1)$-gram associated with the state. Similarly each state has r outgoing transitions which are the n-grams starting with $(n-1)$-gram associated with the state.

The number of states and transitions in $\mathcal{A}'$ are $Q = 1 + r^n(\ell - n)$ and $M = r^n(\ell - n + 1)$, respectively. Note that the size of $\mathcal{A}'$ does not depend on the size of the output alphabet Σ. Also notice that all paths in $\mathcal{A}'$ have equal length of $K = \ell - n + 1$.

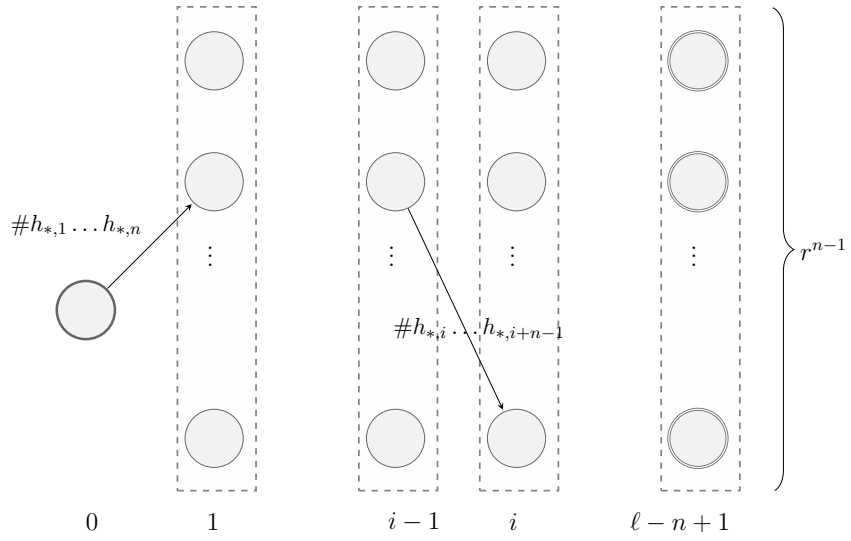


Figure 4.8: The context-dependent automaton $\mathcal{A}'$ for the expert automaton $\mathcal{A}$ depicted in Figure 4.7(b).

Furthermore the number of paths in $\mathcal{A}'$ and $\mathcal{A}$ are the same and equal to $N = r^{\ell}$.

We now apply the algorithms introduced in Section 4.4.

4.6.1 Full Information: Context-dependent Component Hedge Algorithm

We apply the CDCH algorithm to this application in full information setting. The context-dependent automaton $\mathcal{A}'$ introduced in this section is highly structured. We can exploit this structure and obtain slightly better bounds comparing to the general bounds of Theorem 18 for CDCH.

Theorem 23. *Let B denote an upper-bound for the gains of all the transitions in $\mathcal{A}'$, and T be the time horizon. The regret of CDCH algorithm on ensemble structured prediction*

with r predictors consisting of ℓ substructures with n-gram gains can be bounded as

$$\mathit{Regret}_{CDCH} \leq \sqrt{2\,T\,B^2\,(\ell - n + 1)^2\,n\,\log r} + B\,(\ell - n + 1)\,n\,\log r.$$

Proof. First, note that all paths in $\mathcal{A}'$ have equal length of $K = \ell - n + 1$. Therefore there is no need of modifying $\mathcal{A}'$ to make all paths of the same length. At each trial $t \in [T]$, we define the loss of each transition as $\ell_{t,e'} := B - g_{t,e'}$. Extending the results of Koolen et al. [2010], the general regret bound of CDCH is

$$\text{Regret}_{CDCH} \leq \sqrt{2\,T\,K\,B^2\,\Delta(\boldsymbol{v}_{\pi^*}||\boldsymbol{w}_1)} + B\,\Delta(\boldsymbol{v}_{\pi^*}||\boldsymbol{w}_1), \tag{4.5}$$

where $\boldsymbol{v}_{\pi^*} \in \{0,1\}^M$ is a bit vector representation of the best comparator π^*, $\boldsymbol{w}_1 \in [0,1]^M$ is the initial weight vector in the unit-flow polytope, and

$$\Delta(\boldsymbol{w}||\widehat{\boldsymbol{w}}) := \sum_{e' \in E'} \left(w_{e'} \ln \frac{w_{e'}}{\widehat{w}_{e'}} + \widehat{w}_{e'} - w_{e'} \right).$$

Since the initial state has r^n outgoing transitions, and all the intermediate states have r incoming and outgoing transitions, the initial vector $\boldsymbol{w}_1 = \frac{1}{r^n}\mathbf{1}$ falls into the unit-flow polytope, where $\mathbf{1}$ is a vector of all ones. Also $\boldsymbol{v}_{\pi^*}$ has exactly $K = \ell - n + 1$ many ones. Therefore:

$$\Delta(\boldsymbol{v}_{\pi^*}||\boldsymbol{w}_1) = (\ell - n + 1)\,n\,\log r \tag{4.6}$$

Combining the Equations (4.5) and (4.6) gives us the desired regret bound. ■

4.6.2 Semi-Bandit: Context-dependent Semi-Bandit Algorithm

In order to apply the algorithm CDSB in semi-bandit setting in this application, we need to introduce a set of "covering paths" C in $\mathcal{A}'$. We introduce C by

partitioning all the transitions in $\mathcal{A}'$ into r^n paths of length $\ell - n + 1$ iteratively as follows. In each iteration, we choose an arbitrary path π from the initial state to a final state. We add π to the set C and remove all its transitions from $\mathcal{A}'$. Notice that the number of incoming and outgoing transitions for each intermediate state are always equal throughout the iterations. Also note that in each iteration, the number of outgoing edges from the initial state decreases by one. Therefore after r^n iterations, C contains a set of r^n paths that partition the set of transitions in $\mathcal{A}'$.

Furthermore, observe that the number of paths in $\mathcal{A}'$ and $\mathcal{A}$ are the same and equal to $N = r^\ell$. The Corollary below is a direct result of Theorem 19 with $|C| = r^n$.

Corollary 24. *For any $\delta \in (0,1)$, with proper tuning, the regret of the CDSB algorithm can be bounded, with probability $1-\delta$, as:*

$$Regret_{CDSB} \leq 2\,B\,(\ell - n + 1)\,\sqrt{T}\left(\sqrt{4\,r^n\,\ell\,\ln r} + \sqrt{r^n \ln \frac{r^n(\ell - n + 1)}{\delta}}\right).$$

4.6.3 Full Bandit: Context-dependent ComBand Algorithm

We apply the CDCB algorithm to this application in full bandit setting. The Corollary below, which is a direct result of Theorem 20, give regret guarantee for CDCB algorithm.

Corollary 25. *Let λ_{min} denote the smallest non-zero eigenvalue of $\mathbb{E}[\boldsymbol{v}_\pi \boldsymbol{v}_\pi^T]$ where $\boldsymbol{v}_\pi \in \{0,1\}^M$ is the bit representation of the path π which is distributed according to the uniform distribution μ. With proper tuning, the regret of CDCB can be bounded as*

follows:

$$\mathit{Regret}_{CDCB} \leq 2\,B\,\sqrt{T\left(\frac{2(\ell-n+1)}{r^n(\ell-n+1)\lambda_{min}}+1\right)r^n\,(\ell-n+1)\,\ell\,\ln r}.$$

4.7 Path Learning for Full Bandit and Arbitrary Gain

In this section, we go beyond count-based gains and present a general algorithm for path learning in the full bandit setting, when the gain function admits no known structure. The algorithm, EXP3-AG, is an efficient execution of EXP3 for path learning with arbitrary gains using weighted automata and graph operations.

We start with a brief description of the EXP3 algorithm of Auer et al. [2002], which is an online learning algorithm designed for the full bandit setting over a set of N experts. The algorithm maintains a distribution $\boldsymbol{w}_t$ over the set of experts, with $\boldsymbol{w}_1$ initialized to the uniform distribution. At each round $t \in [T]$, the algorithm samples an expert I_t according to $\boldsymbol{w}_t$ and receives (only) the gain g_{t,I_t} associated to that expert. It then updates the weights multiplicatively via the rule $w_{t+1,i} \propto w_{t,i}\,\exp(\eta\,\tilde{g}_{t,i})$ for all $i \in [N]$, where $\tilde{g}_{t,i} = \frac{g_{t,i}}{w_{t,i}}\mathbf{1}\{I_t = i\}$ is an unbiased surrogate gain associated with expert i. The weights $w_{t+1,i}$ are then normalized to sum to one.[6]

In our learning scenario, each expert is a path in $\mathcal{A}$. Since the number of paths is exponential in the size of $\mathcal{A}$, maintaining a weight per path is computationally intractable. We cannot exploit the properties of the gain function since it does not admit any known structure. However, we can make use of the graph representation of

[6] The original EXP3 algorithm of Auer et al. [2002] mixes the weight vector with the uniform distribution in each trial. Later Stoltz [2005] showed that the mixing step is not necessary.

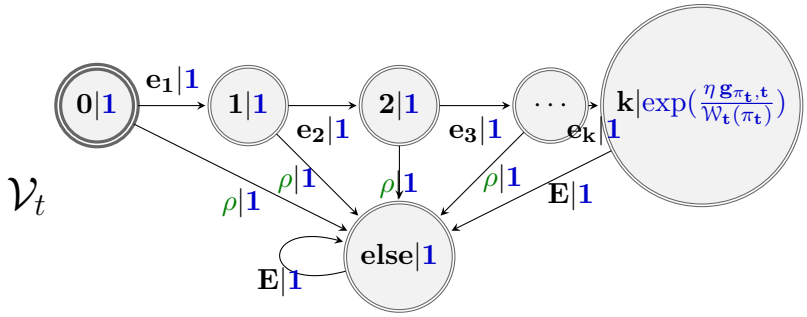


Figure 4.9: The update WFA $\mathcal{V}_t$. The weight of each state and transition is written next to its name separated by "|": $\xrightarrow{e|\text{weight}}$

the experts. We will show that the weights of the experts at round t can be compactly represented by a deterministic stochastic WFA $\mathcal{W}_t$ (see Section 4.2 for the definition and properties). We will further show that sampling a path from $\mathcal{W}_t$ and updating $\mathcal{W}_t$ can be done efficiently.

The WFA $\mathcal{W}_t$ can be efficiently updated using the standard WFA operation of *intersection* (see Section 4.2.2) with a WFA $\mathcal{V}_t$ representing the multiplicative weights that we will refer to as the *update WFA* at time t. $\mathcal{V}_t$ is a deterministic WFA that assigns weight $\exp(\eta\,\widetilde{g}_{t,\pi})$ to path π. Thus, since $\widetilde{g}_{t,\pi} = 0$ for all paths but the path π_t sampled at time t, $\mathcal{V}_t$ assigns weight 1 to all paths $\pi \neq \pi_t$ and weight $\exp\left(\frac{\eta g_{t,\pi_t}}{\mathcal{W}_t(\pi_t)}\right)$ to π_t. $\mathcal{V}_t$ can be constructed deterministically as illustrated in Figure 4.9, using ρ-*transitions* (marked with ρ in green). A ρ-transition admits the semantics of the *rest*: it matches any symbol that is not labeling an existing outgoing transition at that state. For example, the ρ-transition at state 1 matches any symbol other than e_2. ρ-transitions lead to a more compact representation not requiring the knowledge of the full alphabet.

This further helps speed up subsequent intersection operations [Allauzen et al., 2007]).

By definition, the intersection of $\mathcal{W}_t$ and $\mathcal{V}_t$ is a WFA denoted by $(\mathcal{W}_t \circ \mathcal{V}_t)$ that assigns to each path expert π the product of the weights assigned by $\mathcal{W}_t$ and $\mathcal{V}_t$:[7]

$$\forall \pi \in \mathcal{P}\colon \quad (\mathcal{W}_t \circ \mathcal{V}_t)(\pi) = \mathcal{W}_t(\pi) \cdot \mathcal{V}_t(\pi).$$

Since both $\mathcal{W}_t$ and $\mathcal{V}_t$ are deterministic, their intersection $(\mathcal{W}_t \circ \mathcal{V}_t)$ is also deterministic. The WFA $\mathcal{W}_{t+1}$ we obtain after an update may not be stochastic, but we can efficiently compute an equivalent stochastic WFA $\mathcal{W}'$ from any $\mathcal{W}$ using the *weight-pushing* algorithm [Mohri, 1997, 2009a, Takimoto and Warmuth, 2003] described in Section 4.2.1. The following lemma shows that the weight assigned by EXP3-AG to each path expert coincides with those defined by EXP3.

Lemma 26. *At each round $t \in [T]$ in EXP3-AG, the following properties hold for $\mathcal{W}_t$ and $\mathcal{V}_t$:*

$$\mathcal{W}_{t+1}(\pi) \propto \exp(\eta \sum_{s=1}^{t} \widetilde{g}_{s,\pi}), \quad \mathcal{V}_t(\pi) = \exp(\eta\, \widetilde{g}_{t,\pi}),$$

where $\widetilde{g}_{s,\pi} = (g_{s,\pi}/\mathcal{W}_s(\pi)) \cdot \mathbf{1}\{\pi = \pi_s\}$.

Proof. Consider $\mathcal{V}_t$ in Figure 4.9 and let $\pi_t = e_1 e_2 \dots e_k$ be the path chosen by the learner. Every state in $\mathcal{V}_t$ is a final state. Therefore $\mathcal{V}_t$ accepts any sequence of transitions names. Moreover, since the weights of all transitions are 1, the weight of any accepting path is simply the weight of its final state. The construction of $\mathcal{V}_t$ ensures that the weight of every sequence of transition names is 1, except for $\pi_t = e_1 e_2 \dots e_k$.

[7] The terminology of intersection is motivated by the case where the weights are either 0 or 1, in which case the set of paths with non-zero weights in $\mathcal{W}_t \circ \mathcal{V}_t$ is the intersection of the sets of paths with with weight 1 in $\mathcal{W}_t$ and $\mathcal{V}_t$.

Therefore the property of $\mathcal{V}_t$ is achieved:

$$\mathcal{V}_t(\pi) = \begin{cases} \exp\left(\frac{\eta\, g_{t,\pi}}{\mathcal{W}_t(\pi)}\right) & \pi = \pi_t \\ 1 & \text{otherwise} \end{cases}$$

To prove the result for $\mathcal{W}_{t+1}$ we use induction on t. Consider the base case of $t = 0$. $\mathcal{W}_1$ is initialized to the automaton $\mathcal{A}$ with all weights being one. Therefore the weights of all paths are equal to 1 before weight pushing (i.e. $\mathcal{W}_1(\pi) \propto 1$). The inductive step is as follows:

$$\begin{aligned} \mathcal{W}_{t+1}(\pi) &\propto \mathcal{W}_t(\pi) \cdot \mathcal{V}_t(\pi) && \text{(definition of composition)} \\ &= \exp(\eta \sum_{s=1}^{t-1} \widetilde{g}_{s,\pi}) \cdot \exp(\eta\, \widetilde{g}_{t,\pi}) && \text{(induction hypothesis)} \\ &= \exp(\eta \sum_{s=1}^{t} \widetilde{g}_{s,\pi}). \end{aligned}$$

■

Algorithm 5 Algorithm EXP3-AG

1: $\mathcal{W}_1 \longleftarrow \mathcal{A}$

2: For $t = 1, \ldots, T$

3: $\quad \mathcal{W}_t \longleftarrow \textsc{WeightPush}(\mathcal{W}_t)$

4: $\quad \pi_t \longleftarrow \textsc{Sample}(\mathcal{W}_t)$

5: $\quad g_{\pi_t,t} \longleftarrow \textsc{ReceiveGain}(\pi_t)$

6: $\quad \mathcal{V}_t \longleftarrow \textsc{UpdateWFA}(\pi_t, \mathcal{W}_t(\pi_t), g_{t,\pi_t})$

7: $\quad \mathcal{W}_{t+1} \longleftarrow \mathcal{W}_t \circ \mathcal{V}_t$

Algorithm 5 gives the pseudocode of EXP3-AG. The time complexity of EXP3-AG depends mostly on intersection operation in line 7. The worst-case space and time

complexity of the intersection of two deterministic WFA is linear in the size of the automaton the algorithm returns. However, due to specific structure of $\mathcal{V}_t$, the size of $\mathcal{W}_t \circ \mathcal{V}_t$ can be shown to be at most $O(|\mathcal{W}_t| + |\mathcal{V}_t|)$ where $|\mathcal{W}_t|$ is the sum of the number of states and transitions in $\mathcal{W}_t$. This is significantly better than the worst case size of the intersection in general (i.e. $O(|\mathcal{W}_t||\mathcal{V}_t|)$. Recall that $\mathcal{W}_{t+1}$ is deterministic. Thus unlike the algorithms of Cortes et al. [2015b], no further determinization is required. Lemma 27 guarantees the efficiency of EXP3-AG algorithm.

Lemma 27. *The time complexity of* EXP3-AG *at round t is in $O(|\mathcal{W}_t| + |\mathcal{V}_t|)$. Moreover, in the worst case, the growth of $|\mathcal{W}_t|$ over time is at most linear in K where K is the length of the longest path in $\mathcal{A}$.*

Proof. Figure 5 gives the pseudocode of EXP3-AG. The time complexity of the weight-pushing step is in $O(|\mathcal{W}_t|)$, where $|\mathcal{W}_t|$ is the sum of the number of states and transitions in $\mathcal{W}_t$. Lines 4 and 6 in Algorithm 5 take $O(|\mathcal{V}_t|)$ time. Finally, regarding line 7, the worst-case space and time complexity of the intersection of two deterministic WFA is linear in the size of the automaton the algorithm returns. However, the size of the intersection automaton $\mathcal{W}_t \circ \mathcal{V}_t$ is significantly smaller than the general worst case (i.e. $O(|\mathcal{W}_t||\mathcal{V}_t|)$) due to the state "else" with all in-coming ρ-transitions (see Figure 4.9). Since $\mathcal{W}_t$ is deterministic, in the construction of $\mathcal{W}_t \circ \mathcal{V}_t$, each state of $\mathcal{V}_t$ except from the "else" state is paired up only with one state of $\mathcal{W}_t$. For example, if the state is the one reached by $e_1e_2e_3$, then it is paired up with the single state of $\mathcal{W}_t$ reached when reading $e_1e_2e_3$ from the initial state. Thus $|\mathcal{W}_t \circ \mathcal{V}_t| \leq |\mathcal{W}_t| + |\mathcal{V}_t|$,

and therefore, the intersection operation in line 7 takes $O(|\mathcal{W}_t| + |\mathcal{V}_t|)$ time which also dominates the time complexity of EXP3-AG algorithm.

Additionally, observe that the size $|\mathcal{V}_t|$ is in $O(K)$ where K is the length of the longest path in $\mathcal{A}$. Since $|\mathcal{W}_{t+1}| = |\mathcal{W}_t \circ \mathcal{V}_t| \leq |\mathcal{W}_t| + |\mathcal{V}_t|$, in the worst case, the growth of $|\mathcal{W}_t|$ over time is at most linear in K. ■

The following upper bound holds for the regret of EXP3-AG, as a direct consequence of existing guarantees for EXP3 [Auer et al., 2002].

Theorem 28 (Auer et al. [2002])**.** *Let $U > 0$ be an upper bound on all path gains: $g_{t,\pi} \leq U$ for all $t \in [T]$ and all path π. Then, the regret of EXP3-AG with N path experts is upper bounded by $U\sqrt{2\,T\,N\log N}$.*

The $\sqrt{N}$ dependency of the bound suggests that the guarantee will not be informative for large values of N. However, the following known lower bound shows that in the absence of any assumption about the structure of the gains, the dependency cannot be improved in general [Auer et al., 2002].

Theorem 29 (Auer et al. [2002])**.** *Let $U > 0$ be an upper bound on all path gains: $g_{t,\pi} \leq U$ for all $t \in [T]$ and all path π. Then, For any number of path experts $N \geq 2$ there exists a distribution over the assignment of gains to path experts such that the regret of any algorithm is at least $\frac{1}{20}U\min\{\sqrt{T\,N}, T\}$.*

4.8 Conclusion and Open Problems

We considered a large family of non-additive count-based gains and their gappy extension for the path learning problem. From the original graph, we constructed an intermediate automaton in such a way that every path in the original graph corresponds to a path in the intermediate automaton. The equivalent gains in this intermediate automation are additive. Using this construction, we were able to apply the well-known algorithms in the literature for the full information, semi and full bandit settings to non-additive path learning problem. Then we applied our methods to the important application of structured ensemble prediction. Finally, going beyond count-based gains, we developed an efficient implementation of the EXP3 algorithm for the path learning problem in full bandit setting with any (non-additive) gains.

We conclude with two open problems. We assumed here that the expert automaton $\mathcal{A}$ is acyclic and the language of patterns $\mathcal{L}$ is finite. Solving the non-additive path learning problem with cyclic expert automaton together with (infinite) regular language of patterns remains as an open problem.

In this work, regardless of the data and the setting, the context-dependent automaton $\mathcal{A}'$ is constructed in advance as a pre-processing step. Is it possible to construct $\mathcal{A}'$ gradually as the learner goes through trials? Can we build $\mathcal{A}'$ incrementally in different settings and keep it as small as possible as the algorithm is exploring the set of paths and learning about the revealed data.

Chapter 5

Conclusions and Future Work

We developed efficient algorithms for learning combinatorial objects. We explored a wide variety of combinatorial objects consisting of components, such as Huffman trees, permutations, binary search trees, k-sets, paths and multipaths. The main challenge in this learning task is the large number of objects which is exponential in terms of the number of components.

We introduced extended formulation techniques to develop learning algorithms for a new class of combinatorial objects. Standard approaches for learning combinatorial objects (e.g. Component Hedge of Koolen et al. [2010]) typically works with the convex hull of the objects. These approaches, however, cannot be applied to objects whose convex hull is a polytope with too many facets. We fixed this problem by proposing the *XF-Hedge* algorithm. XF-Hedge uses auxiliary representations of such objects that are constructed by extended formulation techniques. The convex hull of the objects in these auxiliary representations has an efficient characterization with polynomially

many facets. This allowed us to extend algorithms like Component Hedge to the class of objects with extended formulations of polynomial size.

As a special – but yet general – family of constructing extended formulations, we then focused on arbitrary dynamic programming with min-sum recurrence relations. The graph of subproblems served as the auxiliary representation where each object was described by a subgraph called multipath. The structure of the multipath representation and the additivity of the loss over its components allowed us to implement Component Hedge and Expanded Hedge.

Finally, we studied the path learning problem in a automaton with non-additive gains. We presented new online algorithms for path learning with non-additive count-based gains for the three settings of full information, semi-bandit and full bandit. The key component of our algorithms is the definition and computation of an auxiliary automaton called *context-dependent automaton* which admits additive gains. This enabled us to use existing algorithms designed for additive gains.

In the remaining of this chapter, we discuss the main areas of future work to investigate:

More Applications with Extended Formulations. In this thesis, we introduced the extended formulation techniques for learning combinatorial objects. In particular, we provided efficient algorithms when the extended formulation is constructed by (1) *sorting networks* (Chapter 2) and (2) *dynamic programming* (Chapter 3). We propose to investigate other techniques of constructing extended formulation to develop

efficient and effective learning algorithms such as (3) *disjunctive programming* [Kaibel, 2011] and (4) *branched polyhedral systems* (BPS) [Kaibel and Loos, 2010].

Extension to Bandit Settings. The main focus of this thesis was the *full information* setting where the adversary reveals the entire loss vector in each trial. In contrast in *full-* and *semi-bandit* settings, the adversary only reveals partial information about the loss. Significant work has already been done in learning combinatorial objects in full- and semi-bandit settings [Audibert et al., 2011, György et al., 2007, Audibert et al., 2013, Kveton et al., 2015, Cesa-Bianchi and Lugosi, 2012]. The bandit algorithms for learning combinatorial objects usually proceed as follows. First an unbiased estimation of the complete loss vector is computed (called *surrogate loss*) based on the partial information about the loss revealed by the adversary. Then Component Hedge or Expanded Hedge is applied with the surrogate loss.

This thesis only did limited work in the bandit settings for combinatorial objects. Can we extend our work in full information setting to semi- and full bandit settings?

Computing the surrogate loss in bandit typically requires finding the probability of occurrence of the components and/or co-occurrence of the pairs of components. Efficient algorithms are developed for this computation for different applications. For example, in additive multipath learning, dynamic programming algorithms are used to compute the probability of occurrence of each multiedge in the chosen multipath (see Section 3.2). Can we extend these algorithms to other combinatorial objects?

Bibliography

D. Adamskiy, M. K. Warmuth, and W. M. Koolen. Putting Bayes to sleep. In *Advances in Neural Information Processing Systems*, pages 135–143, 2012.

M. Afshari Rad and H. T. Kakhki. Two extended formulations for cardinality maximum flow network interdiction problem. *Networks*, 2017.

N. Ailon. Improved bounds for online learning over the Permutahedron and other ranking polytopes. In *AISTATS*, pages 29–37, 2014.

M. Ajtai, J. Komlós, and E. Szemerédi. Sorting inc logn parallel steps. *Combinatorica*, 3(1):1–19, 1983.

C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: a general and efficient weighted finite-state transducer library. In *Proceedings of CIAA*, pages 11–23. Springer, 2007.

J.-Y. Audibert, S. Bubeck, and G. Lugosi. Minimax policies for combinatorial prediction games. In *COLT*, volume 19, pages 107–132, 2011.

J.-Y. Audibert, S. Bubeck, and G. Lugosi. Regret in online combinatorial optimization. *Mathematics of Operations Research*, 39(1):31–45, 2013.

P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

B. Awerbuch and R. Kleinberg. Online linear optimization and adaptive routing. *Journal of Computer and System Sciences*, 74(1):97–114, 2008.

H. H. Bauschke and J. M. Borwein. Legendre functions and the method of random Bregman projections. *Journal of Convex Analysis*, 4(1):27–67, 1997.

O. Bousquet and M. K. Warmuth. Tracking a small set of experts by mixing past posteriors. *Journal of Machine Learning Research*, 3(Nov):363–396, 2002.

L. M. Bregman. The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming. *USSR computational mathematics and mathematical physics*, 7(3):200–217, 1967.

N. Cesa-Bianchi and G. Lugosi. *Prediction, learning, and games*. Cambridge university press, 2006.

N. Cesa-Bianchi and G. Lugosi. Combinatorial bandits. *Journal of Computer and System Sciences*, 78(5):1404–1422, 2012.

N. Cesa-Bianchi, P. M. Long, and M. K. Warmuth. Worst-case quadratic loss bounds for prediction using linear functions and gradient descent. *IEEE Transactions on Neural Networks*, 7(3):604–619, 1996.

N. Cesa-Bianchi, Y. Freund, D. Haussler, D. P. Helmbold, R. E. Schapire, and M. K. Warmuth. How to use expert advice. *Journal of the ACM (JACM)*, 44(3):427–485, 1997.

M. Conforti, G. Cornuéjols, and G. Zambelli. Extended formulations in combinatorial optimization. *4OR: A Quarterly Journal of Operations Research*, 8(1):1–48, 2010.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press Cambridge, 2009.

C. Cortes, V. Kuznetsov, M. Mohri, and M. Warmuth. On-line learning algorithms for path experts with non-additive losses. In *Conference on Learning Theory*, pages 424–447, 2015a.

C. Cortes, V. Kuznetsov, M. Mohri, and M. K. Warmuth. On-line learning algorithms for path experts with non-additive losses. In *Proceedings of The 28th Conference on Learning Theory, COLT 2015, Paris, France, July 3-6, 2015*, pages 424–447, 2015b.

C. Cortes, V. Kuznetsov, M. Mohri, H. Rahmanian, and M. K. Warmuth. On-line non-additive path learning under full and partial information. *arXiv preprint arXiv:1804.06518*, 2018.

V. Dani, S. M. Kakade, and T. P. Hayes. The price of bandit information for online optimization. In *Advances in Neural Information Processing Systems*, pages 345–352, 2008.

F. Deutsch. Dykstras cyclic projections algorithm: the rate of convergence. In *Approximation Theory, Wavelets and Applications*, pages 87–94. Springer, 1995.

I. S. Dhillon and J. A. Tropp. Matrix nearness problems with Bregman divergences. *SIAM Journal on Matrix Analysis and Applications*, 29(4):1120–1146, 2007.

P. Diaconis. Group representations in probability and statistics. *Lecture Notes-Monograph Series*, 11:i–192, 1988.

T. Dick, A. Gyorgy, and C. Szepesvari. Online learning in Markov decision processes with changing cost sequences. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 512–520, 2014.

M. Dudík, N. Haghtalab, H. Luo, R. E. Schapire, V. Syrgkanis, and J. W. Vaughan. Oracle-efficient learning and auction design. In *Proceedings of 58th Annual Symposium on Foundations of Computer Science (FOCS)*, 2017.

E. Even-Dar, S. M. Kakade, and Y. Mansour. Online Markov decision processes. *Mathematics of Operations Research*, 34(3):726–736, 2009.

S. Fiorini, V. Kaibel, K. Pashkovich, and D. O. Theis. Combinatorial bounds on nonnegative rank and extended formulations. *Discrete mathematics*, 313(1):67–83, 2013.

Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

S. Fujishige. *Submodular functions and optimization*, volume 58. Elsevier, 2005.

S. Gupta, M. Goemans, and P. Jaillet. Solving combinatorial games using products, projections and lexicographically optimal bases. *Preprint arXiv:1603.00522*, 2016.

A. György, T. Linder, G. Lugosi, and G. Ottucsák. The on-line shortest path problem under partial monitoring. *Journal of Machine Learning Research*, 8(Oct):2369–2403, 2007.

D. P. Helmbold and R. E. Schapire. Predicting nearly as well as the best pruning of a decision tree. *Machine Learning*, 27(1):51–68, 1997.

D. P. Helmbold and M. K. Warmuth. Learning permutations with exponential weights. *The Journal of Machine Learning Research*, 10:1705–1736, 2009.

D. P. Helmbold, S. Panizza, and M. K. Warmuth. Direct and indirect algorithms for on-line learning of disjunctions. *Theoretical Computer Science*, 284(1):109–142, 2002.

M. Herbster and M. K. Warmuth. Tracking the best expert. *Machine Learning*, 32(2): 151–178, 1998.

M. Herbster and M. K. Warmuth. Tracking the best linear predictor. *The Journal of Machine Learning Research*, 1:281–309, 2001.

V. Kaibel. Extended formulations in combinatorial optimization. *Preprint arXiv:1104.1023*, 2011.

V. Kaibel and A. Loos. Branched polyhedral systems. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 177–190. Springer, 2010.

V. Kaibel and K. Pashkovich. Constructing extended formulations from reflection relations. In *Facets of Combinatorial Optimization*, pages 77–100. Springer, 2013.

A. Kalai and S. Vempala. Efficient algorithms for online decision problems. *Journal of Computer and System Sciences*, 71(3):291–307, 2005.

R. M. Kaplan and M. Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.

J. Kivinen. Unit rule in k-sets. Unpublished manuscript, 2010.

J. Kleinberg and E. Tardos. *Algorithm design*. Addison Wesley, 2006.

P. A. Knight. The Sinkhorn–Knopp algorithm: convergence and applications. *SIAM Journal on Matrix Analysis and Applications*, 30(1):261–275, 2008.

W. M. Koolen, M. K. Warmuth, and J. Kivinen. Hedging structured concepts. In *Conference on Learning Theory*, pages 239–254. Omnipress, 2010.

D. Kuzmin and M. K. Warmuth. Optimum follow the leader algorithm. In *Learning Theory*, pages 684–686. Springer, 2005.

B. Kveton, Z. Wen, A. Ashkan, and C. Szepesvari. Tight regret bounds for stochastic combinatorial semi-bandits. In *Artificial Intelligence and Statistics*, pages 535–543, 2015.

N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994.

J.-L. Loday. The multiple facets of the associahedron. *Proc. 2005 Academy Coll. Series*, 2005.

W. Maass and M. K. Warmuth. Efficient learning with virtual threshold gates. *Information and Computation*, 141(1):66–83, 1998.

T. L. Magnanti and L. A. Wolsey. Optimal trees. *Handbooks in operations research and management science*, 7:503–615, 1995.

R. K. Martin, R. L. Rardin, and B. A. Campbell. Polyhedral characterization of discrete dynamic programming. *Operations Research*, 38(1):127–138, 1990.

J.-F. Maurras, T. H. Nguyen, and V. H. Nguyen. On the convex hull of huffman trees. *Electronic Notes in Discrete Mathematics*, 36:1009–1016, 2010.

M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

M. Mohri. Weighted automata algorithms. In *Handbook of Weighted Automata*, pages 213–254. Springer, 2009a.

M. Mohri. Weighted automata algorithms. In *Handbook of weighted automata*, pages 213–254. Springer, 2009b.

M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 231–238. Association for Computational Linguistics, 1996.

M. Mohri, F. Pereira, and M. Riley. Weighted automata in text and speech processing. In *Proceedings of ECAI-96 Workshop on Extended finite state models of language*, 1996.

K. Pashkovich. *Extended formulations for combinatorial polytopes.* PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2012.

F. Pereira and M. Riley. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing*, pages 431–453. MIT Press, 1997.

H. Rahmanian and M. K. Warmuth. Online dynamic programming. In *Advances in Neural Information Processing Systems*, pages 2824–2834, 2017.

H. Rahmanian, D. P. Helmbold, and S. Vishwanathan. Online learning of combinatorial objects via extended formulation. In *Algorithmic Learning Theory*, pages 702–724, 2018.

A. Rajkumar and S. Agarwal. Online decision-making in general combinatorial spaces. In *Advances in Neural Information Processing Systems*, pages 3482–3490, 2014.

G. Stoltz. *Information incomplete et regret interne en prédiction de suites individuelles.* PhD thesis, Ph. D. thesis, Univ. Paris Sud, 2005.

D. Suehiro, K. Hatano, S. Kijima, E. Takimoto, and K. Nagano. Online prediction under submodular constraints. In *International Conference on Algorithmic Learning Theory*, pages 260–274. Springer, 2012.

E. Takimoto and M. K. Warmuth. Predicting nearly as well as the best pruning of a planar decision graph. *Theoretical Computer Science*, 288(2):217–235, 2002.

E. Takimoto and M. K. Warmuth. Path kernels and multiplicative updates. *The Journal of Machine Learning Research*, 4:773–818, 2003.

V. G. Vovk. Aggregating strategies. *Proc. of Computational Learning Theory, 1990*, 1990.

M. K. Warmuth and D. Kuzmin. Randomized online PCA algorithms with regret bounds that are logarithmic in the dimension. *Journal of Machine Learning Research*, 9(10): 2287–2320, 2008.

S. Yasutake, K. Hatano, S. Kijima, E. Takimoto, and M. Takeda. Online linear optimization over permutations. In *Algorithms and Computation*, pages 534–543. Springer, 2011.