Path Kernels and Multiplicative Updates

Eiji Takimoto[1⋆] and Manfred K. Warmuth[2⋆⋆]

[1] Graduate School of Information Sciences,Tohoku University
Sendai, 980-8579, Japan
[2] Computer Science Department, University of California
Santa Cruz, CA 95064, U.S.A.

Abstract. We consider a natural convolution kernel defined by a directed graph. Each edge contributes an input. The inputs along a path form a product and the products for all paths are summed. We also have a set of probabilities on the edges so that the outflow from each node is one. We then discuss multiplicative updates on these graphs where the prediction is essentially a kernel computation and the update contributes a factor to each edge. Now the total outflow out of each node is not one any more. However some clever algorithms re-normalize the weights on the paths so that the total outflow out of each node is one again. Finally we discuss the use of regular expressions for speeding up the kernel and re-normalization computation. In particular we rewrite the multiplicative algorithms that predict as well as the best pruning of a series parallel graph in terms of efficient kernel computations.

1 Introduction

Assume we have a directed graph with a source and a sink node. A weight v_e is associated with each edge e of the graph so that the total outflow from each non-sink node is one. The weighting on the edges extends to a weighting on the source-to-sink paths: $w_P = \prod_{e \in P} v_e$. Each such path contributes a feature w_P to a feature vector $\boldsymbol{w} = \Phi(\boldsymbol{v})$ (see Figure 1) and in this paper we focus on the kernel associated with the path features of directed graphs. The predictions of the algorithms are often determined by kernel computations, i.e. there is a second weighting x_e on the edges and the prediction of the algorithm is based on:

$$K(\boldsymbol{v}, \boldsymbol{x}) = \Phi(\boldsymbol{v}) \cdot \Phi(\boldsymbol{x}) = \sum_P \prod_{e \in P} v_e x_e.$$

Similar related kernels that are built from regular expressions or pair-HMMs were introduced in [Hau99, Wat99] for the purpose of characterizing the similarity between strings. Any path kernel can easily be used for additive algorithms such as the Perceptron algorithm and Support Vector Machines. Such algorithms compute a linear model in a feature space, i.e. the weight vector $\boldsymbol{w}_t$ at trial t is

⋆ Part of this work was done while the author visited University of California, Santa Cruz.
⋆⋆ Supported by NSF grant CCR 9821087

J. Kivinen and R. H. Sloan (Eds.): COLT 2002, LNAI 2375, pp. 74–89, 2002.
© Springer-Verlag Berlin Heidelberg 2002

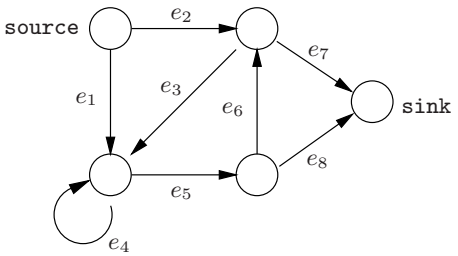

Fig. 1. An example of a digraph. If the weight of e_i is v_i, then its feature vector is $\Phi(\boldsymbol{v}) = \{v_2v_7, v_1v_5v_8, v_2v_3v_5v_8, v_1v_4v_5v_8, v_1v_5v_6v_7, v_2v_3v_4v_5v_8, v_2v_3v_5v_6v_7, v_1v_4v_4v_5v_8, v_1v_5v_6v_3v_5v_8, \ldots\}$

a linear combination of the expanded past inputs $\Phi(\boldsymbol{x}_q)$ $(1 \leq q \leq t-1)$. The linear predictions $\boldsymbol{w}_t \cdot \boldsymbol{x}_t$ can be computed via the dot products $\Phi(\boldsymbol{x}_q) \cdot \Phi(\boldsymbol{x}_t)$.

In this paper we are particularly concerned with multiplicative update algorithms where each path is an expert or a feature. Typically there are exponentially (sometimes infinitely) many features and maintaining one weight per feature is prohibitive. However in this paper we manipulate the exponentially many path weights via efficient kernel computations. A key requirement for our methods is that the loss of a path P decomposes into a sum of the losses over its edges. Except for a normalization, the updates multiply each path weight w_P by an exponential factor whose exponent decomposes into a sum over the edges of P. Thus the path weights are multiplied by a product containing a factor per edge of the path. Viewed differently, the updates multiply each edge by a factor.

We want three properties for the weightings on the paths.

1. The weighting should be in *product form*, i.e. $w_P = \prod_{e \in P} v_e$.
2. The outflow from each node u should be one, i.e. $\sum_{u':(u,u') \in E(G)} v_{(u,u')} = 1$, where $E(G)$ denotes the set of edges of G.
3. The total path weight is one, i.e. $\sum_P w_P = 1$.

The three properties make it trivial to generate random paths: Start at the source and iteratively pick an outgoing edge from the current node according to the prescribed edge probabilities until the sink is reached.

The additional factors on the edges that are introduced by the multiplicative update mess up these properties. However there is an algorithm that rearranges the weights on the edges so that the weights on the path remain unchanged but again have the three properties. This algorithm is called the *Weight Pushing algorithm*. It was developed by Mehryar Mohri in the context of speech recognition [Moh98].

One of the goals of this paper is to show that multiplicative updates nicely mesh with path kernels. The algorithms can easily be described by "direct" algorithms that maintain exponentially many path-weights. The algorithms are then

simulated by "indirect" algorithms that maintain only polynomially many edge-weights. There is a lot of precedents for this type of research [HS97, MW98, HPW02, TW99]. In this paper we hope to give a unifying view and make the connection to path kernels. We will also re-express many of the previously developed indirect algorithms using our methods.

In the next section we review the Weight Pushing algorithm which reestablishes the properties of the edge weights after each edge received a factor. We also give the properties of the update that we need so that the updated weights contribute a factor to each edge. We then apply our methods to a dynamic routing problem (Section 3) and to an on-line shortest path problem (Section 4). We prove bounds for the algorithm that decay with the length of the longest path in the graph. However we also give an example graph where the length of the longest path does not enter into the bound. The set of paths associated with this graph can be concisely presented as a regular expression and was implicitly used in the BEG algorithm for learning disjunctions [HPW02]. In Section 5 we introduce path sets defined as regular expressions and give an efficient implementation for the Weight Pushing algorithm in this case. Finally we rewrite the algorithms for predicting as well as the best pruning of Series Parallel graph using the methods just introduced (Section 5.2).

2 Preliminaries

Let G be a directed graph with a source and a sink node. Assume that the edge weights v_e and the path weights w_P fulfill the three properties given in the introduction.

Now each edge receives a factor b_e and the new weights of the paths become

$$\tilde{w}_P = \frac{w_P \prod_{e \in P} b_e}{\sum_P w_P \prod_{e \in P} b_e} = \frac{\prod_{e \in P} v_e b_e}{\sum_P \prod_{e \in P} v_e b_e}. \tag{1}$$

The normalization needs to be non-zero, i.e. we need the following property:

4. The edge factors b_e are non-negative and for some path P, $\prod_{e \in P} v_e b_e > 0$.

Our goal is to find new edge weights $\tilde{v}_e$ such that the first two properties are maintained. Note that Property 3 is maintained because the path weights in (1) are normalized.

We begin by introducing a kernel. For any node u, let $\mathcal{P}(u)$ denote the set of paths from the node u to the sink. Assume that input vectors $\boldsymbol{v}$ and $\boldsymbol{x}$ to edges are given. For any node u, let

$$K_u(\boldsymbol{v}, \boldsymbol{x}) = \sum_{P \in \mathcal{P}(u)} \prod_{e \in P} v_e \prod_{e \in P} x_e = \sum_{P \in \mathcal{P}(u)} \prod_{e \in P} v_e x_e.$$

Clearly $K_{\texttt{source}}(\boldsymbol{v}, \boldsymbol{x})$ gives the dot product $\Phi(\boldsymbol{v}) \cdot \Phi(\boldsymbol{x})$ given in the introduction that is associated with the whole graph G.

The computation of K depends on the complexity of the graph G. If G is acyclic, then the functions $K_u(\boldsymbol{v}, \boldsymbol{x})$ can be recursively calculated as

$$K_{\mathtt{sink}}(\boldsymbol{v}, \boldsymbol{x}) = 1$$

and in the bottom up order

$$K_u(\boldsymbol{v}, \boldsymbol{x}) = \sum_{u':(u,u')\in E(G)} v_{(u,u')} x_{(u,u')} K_{u'}(\boldsymbol{v}, \boldsymbol{x}).$$

Clearly this takes time linear in the number of edges.

In the case when G is an arbitrary digraph then K_u can be computed using a matrix inversion or via dynamic programming using essentially the Floyd-Warshal all-pairs shortest path algorithm [Moh98]. The cost of these algorithms is essentially cubic in the number of vertices of G. Speedups are possible for sparse graphs.

A straight-forward way to normalize the edge weights would be to divide the weights of the edges leaving a vertex by the total weight of all edges leaving the vertex. However this usually does not give the correct path weights (1) because the product of the normalization factors along different paths is not the same. Instead, the Weight Pushing algorithm [Moh98] assigns the following new weight to the edge $e = (u, u')$:

$$\tilde{v}_e = \frac{v_e b_e K_{u'}(\boldsymbol{v}, \boldsymbol{b})}{K_u(\boldsymbol{v}, \boldsymbol{b})}. \tag{2}$$

It is easy to see that the new weights $\tilde{v}_e$ are normalized, i.e.

$$\sum_{u':(u,u')\in E(G)} \tilde{v}_{(u,u')} = 1.$$

Property 1 is proven as follows: Let $P = \{(u_0, u_1), (u_2, u_3), \ldots, (u_{k-1}, u_k)\}$ be any path from the source u_0 to the sink u_k. Then

$$\begin{aligned}
\prod_{e\in P} \tilde{v}_e &= \prod_{i=1}^{k} \frac{v_{(u_{i-1},u_i)} b_{(u_{i-1},u_i)} K_{u_i}(\boldsymbol{v}, \boldsymbol{b})}{K_{u_{i-1}}(\boldsymbol{v}, \boldsymbol{b})} \\
&= \frac{\prod_{e\in P} v_e b_e K_{\mathtt{sink}}(\boldsymbol{v}, \boldsymbol{b})}{K_{\mathtt{source}}(\boldsymbol{v}, \boldsymbol{b})} \\
&= \frac{w_P \prod_{e\in P} b_e}{K_{\mathtt{source}}(\boldsymbol{v}, \boldsymbol{b})} \\
&= \frac{w_P \prod_{e\in P} b_e}{\sum_P w_P \prod_{e\in P} b_e}.
\end{aligned}$$

The last expression equals $\tilde{w}_P$ (See (1)).

Typically the multiplicative weights updates have the form

$$\tilde{w}_P = \frac{w_P \exp(-\eta L(P))}{\sum_P w_P \exp(-\eta L(P))} \quad \text{or} \quad \tilde{w}_P = \frac{w_P \exp(-\eta \frac{\partial L(P)}{\partial w_P})}{\sum_P w_P \exp(-\eta \frac{\partial L(P)}{\partial w_P})},$$

where $L(P)$ is the loss of path P. So if the loss $L(P)$ can be decomposed into a sum over the edges of P, i.e. $L(P) = \sum_{e \in P} l(e)$, then these update rules have the form of (1) and so they can be efficiently simulated by the edge updates using the Weight Pushing algorithm. Property 4 is also easily satisfied. In most cases, however, the loss is given by

$$L(P) = L(y, \Phi(\boldsymbol{x})_P) = L(y, \textstyle\prod_{e \in P} x_e)$$

for some loss function L, which cannot be decomposed into a sum over the edges in general. Actually, Khardon, Roth and Servedio show that simulating Winnow (a multiplicative update algorithm) with the monomial kernel (which is a path kernel) is $\#P$-hard [KRS01]. Still certain problems can be solved using multiplicative updates of the form of (1). In the subsequent sections we will give some examples. In the full paper we will also discuss in more detail how the efficient disjunction learning algorithm (such as Winnow with one weight per variable) may be seen as examples of the same method.

3 A Dynamic Routing Problem

In this section we regard the digraph G as a probabilistic automaton and consider a dynamic routing problem based on G. At trial $t = 1, 2, \ldots$ we want to send a packet X_t from the source to the sink and assign transition probabilities $v_{t,e}$ to the edges that define a probabilistic routing. Starting from the source we choose a random path to the sink according to the transition probabilities and try to send the packet along the path. But some edges may reject the request with a certain probability that is given by nature and the current trial may fail. The goal is make the success probability as large as possible. Below we give the problem formally.

In each trial $t = 1, 2, \ldots, T$, the following happens:

1. The algorithm assigns transition probabilities $v_{t,e} \in [0, 1]$ to all edges e.
2. Real numbers $d_{t,e} \in [0, 1]$ for all edges are given. The number $d_{t,e}$ is interpreted as the probability that the edge e "accepts" the packet X_t. A path P "accepts" X_t if all the edges e of P accept X_t and the probability of this is

$$\Pr(X_t \mid P) = \prod_{e \in P} d_{t,e}.$$

   We assume that $\Pr(X_t \mid P) > 0$ for at least one path P, so that Property 4 holds. Moreover, this probability is independent for all trials.
3. The probability that the current probabilistic automaton accepts X_t is

$$A_t = \sum_P w_{t,P} \Pr(X_t \mid P),$$

   where $w_{t,P} = \prod_{e \in P} v_{t,e}$ is the probability that the path P is chosen.

The goal is to make the total success probability $\prod_{t=1}^{T} A_t$ as large as possible.

In the feature space we can employ the Bayes algorithm. The initial weight $w_{1,P}$ is a prior of path P. Then the Bayes algorithm sets the weight $w_{t+1,P}$ as a posterior of P given the input packets $X_1, \ldots, X_t$ observed so far. Specifically the Bayes algorithm updates weights so that

$$\begin{aligned} w_{t+1,P} &= \Pr(P \mid X_1, \ldots, X_t) \\ &= \frac{w_{1,P} \Pr(X_1, \ldots, X_t \mid P)}{\sum_P w_{1,P} \Pr(X_1, \ldots, X_t \mid P)} \\ &= \frac{w_{t,P} \Pr(X_t \mid P)}{\sum_P w_{t,P} \Pr(X_t \mid P)} \\ &= \frac{w_{t,P} \prod_{e \in P} d_{t,e}}{\sum_P w_{t,P} \prod_{e \in P} d_{t,e}}. \end{aligned} \tag{3}$$

It is well known that the Bayes algorithm guarantees that

$$\prod_{t=1}^{T} A_t = \sum_P w_{1,P} \Pr(X_1, \ldots, X_T \mid P) \geq \max_P w_{1,P} \Pr(X_1, \ldots, X_T \mid P).$$

This implies that the probability that the Bayes algorithm succeeds to send the all packets $X_1, \ldots, X_T$ is at least the probability for the path P that attains the maximum of the success probability.

It is easy to check that the Bayes update rule (3) has the multiplicative form (1). So we can run the Bayes algorithm efficiently using the Weight Pushing algorithm.

4 On-Line Shortest Path Problem

We consider an on-line version of the shortest path problem. The problem is similar to the previous one, but now $d_{t,e}$ is the distance of the edge e at trial t. Furthermore we assume that the given graph G is acyclic, so that any path has a bounded length.

In each trial $t = 1, 2, \ldots, T$, the following happens:

1. The algorithm assigns transition probabilities $v_{t,e} \in [0, 1]$ to all edges e.
2. Then distances $d_{t,e} \in [0, 1]$ for all edges are given.
3. The algorithm incurs a loss L_t which is defined as the expected length of path of G. That is,

$$L_t = \sum_{P \in \mathcal{P}(G)} w_{t,P} l_{t,P},$$

where

$$l_{t,P} = \sum_{e \in P} d_{t,e} \tag{4}$$

is the length of the path P, which is interpreted as the loss of P.

Note that the length $l_{t,P}$ of path P at each trial is upper bounded by the number of edges in P. Letting D denote the depth (maximum number of edges of P) of G, we have $l_{t,P} \in [0, D]$. Note that the path P minimizing the total length

$$\sum_{t=1}^{T} l_{t,P} = \sum_{t=1}^{T} \sum_{e \in P} d_{t,e} = \sum_{e \in P} \sum_{t=1}^{T} d_{t,e}$$

can be interpreted as the shortest path based on the cumulative distances of edges: $\sum_{t=1}^{T} d_{t,e}$. The goal of the algorithm is to make its total loss $\sum_{t=1}^{T} L_t$ not much larger than the length of the shortest path.

Considering each path P as an expert, we can view the problem above as a dynamic resource allocation problem [FS97] introduced by Freund and Schapire. So we can apply their *Hedge algorithm* which is a reformulation of the *Weighted Majority algorithm* (see WMC and WMR of [LW94]). The Hedge algorithm works over the feature space. Let $w_{1,P}$ be initial weights for paths/experts that sum to 1. At each trial t, when given losses $l_{t,P} \in [0, D]$ for paths/experts, the algorithm incurs loss

$$L_t = \sum_P w_{t,P} l_{t,P}$$

and updates weights according to

$$w_{t+1,P} = \frac{w_{t,P} \beta^{l_{t,P}/D}}{\sum_{P'} w_{t,P'} \beta^{l_{t,P'}/D}}, \tag{5}$$

where $0 \leq \beta < 1$ is a parameter. It is shown in [LW94, FS97] that for any sequence of loss vectors for experts, the Hedge algorithm guarantees[1] that

$$\sum_{t=1}^{T} L_t \leq \min_P \left(\frac{\ln(1/\beta)}{1-\beta} \sum_{t=1}^{T} l_{t,P} + \frac{D}{1-\beta} \ln \frac{1}{w_{1,P}} \right). \tag{6}$$

Since $l_{t,P} = \sum_{e \in P} d_{t,e}$, the update rule (5) is the multiplicative form of (1). So we can employ the Weight Pushing algorithm to run the Hedge algorithm efficiently. Below we give a proof of (6).

Let $W_t = \sum_P w_{1,P} \beta^{l_{1..t-1,P}/D}$, where $l_{1..t-1,P}$ is shorthand for $\sum_{q=1}^{t-1} l_{q,P}$. Note that $W_1 = 1$. It is not hard to see that the update rule (5) implies $w_{t,P} = w_{1,P} \beta^{l_{1..t-1,P}/D} / W_t$. So

$$\begin{aligned} \ln \frac{W_{t+1}}{W_t} &= \ln \left(\frac{\sum_P w_{1,P} \beta^{l_{1..t,P}/D}}{W_t} \right) \\ &= \ln \sum_P \left(\frac{w_{1,P} \beta^{l_{1..t-1,P}/D}}{W_t} \right) \beta^{l_{t,P}/D} \\ &= \ln \sum_P w_{t,P} \beta^{l_{t,P}/D}. \end{aligned}$$

[1] In [FS97] losses of experts are upper bounded by 1 at each trial, while in our case they are upper bounded by D. So the second term of the loss bound in (6) has the factor D.

Since $l_{t,P}/D \in [0,1]$ and $0 \leq \beta < 1$, we have $\beta^{l_{t,P}/D} \leq 1-(1-\beta)l_{t,P}/D$. Plugging this inequality into the above formula, we have

$$\begin{aligned}\ln \frac{W_{t+1}}{W_t} &\leq \ln \sum_P w_{t,P}\left(1-(1-\beta)l_{t,P}/D\right)\\ &= \ln\left(1-(1-\beta)L_t/D\right)\\ &\leq -(1-\beta)L_t/D.\end{aligned}$$

Summing the both sides over all t's, we get

$$\begin{aligned}\sum_{t=1}^{T} L_t &\leq \frac{D(\ln W_1 - \ln W_{T+1})}{1-\beta}\\ &= -\frac{D\ln \sum_P w_{1,P}\beta^{l_{1..T,P}/D}}{1-\beta}\\ &\leq -\frac{D\ln\left(w_{1,P^*}\beta^{l_{1..T,P^*}/D}\right)}{1-\beta}\\ &= \frac{\ln 1/\beta}{1-\beta} l_{1..T,P^*} + \frac{D}{1-\beta}\ln\frac{1}{w_{1,P^*}},\end{aligned}$$

where P^* is any path. This proves (6).

Unfortunately, the loss bound in (6) depends on the depth D of G. Thus the graphs of large depth or for cyclic graphs the bound is vacuous. In some cases, however, we have a bound where D does not appear in the bound although the depth can be arbitrarily large. We give an example of such graphs in Figure 2. The graph has $n+1$ nodes and each node i $(1 \leq i \leq n)$ has two outgoing edges e_i and e'_i to node $i+1$. Let $v_{t,i}$ and $1-v_{t,i}$ denote the weights of these edges at trial t, respectively. Let $d_{t,i}$ and $d'_{t,i}$ be the distances assigned to these edges at trial t. Note that the depth of the graph is n.

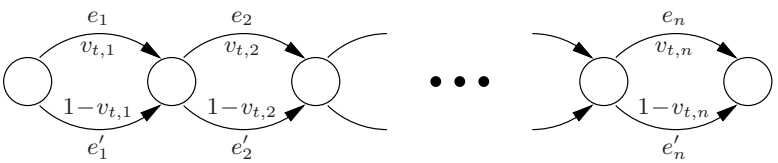

Fig. 2. An example graph whose depth D does not appear in the loss bound

Now we give a loss bound of the Hedge algorithm when applied to this graph. In this case we use the following update rule

$$w_{t+1,P} = \frac{w_{t,P}\beta^{l_{t,P}}}{\sum_{P'} w_{t,P'}\beta^{l_{t,P'}}} \tag{7}$$

instead of (5). The key property we use in the analysis is that $\ln(W_{t+1}/W_t)$ is decomposed into n independent sums:

$$\begin{aligned}
\ln \frac{W_{t+1}}{W_t} &= \ln \sum_P w_{t,P} \beta^{l_{t,P}} \\
&= \ln \sum_{A \subseteq \{1,\ldots,n\}} \left(\prod_{i \in A} \left(v_{t,i} \beta^{d_{t,i}}\right) \prod_{i \notin A} (1 - v_{t,i}) \beta^{d'_{t,i}} \right) \\
&= \ln \prod_{i=1}^{n} \left(v_{t,i} \beta^{d_{t,i}} + (1 - v_{t,i}) \beta^{d'_{t,i}} \right) \\
&= \sum_{i=1}^{n} \ln \left(v_{t,i} \beta^{d_{t,i}} + (1 - v_{t,i}) \beta^{d'_{t,i}} \right).
\end{aligned}$$

Here we used the following formula: $\sum_A \prod_{i \in A} a_i \prod_{i \notin A} b_i = \prod_i (a_i + b_i)$. Using the approximation $\beta^x \leq 1 - (1 - \beta)x$ for $x \in [0, 1]$ as before, we have

$$\begin{aligned}
\ln \frac{W_{t+1}}{W_t} &\leq \sum_{i=1}^{n} \ln \left(1 - (1 - \beta)(v_{t,i} d_{t,i} + (1 - v_{t,i}) d'_{t,i})\right) \\
&\leq -(1 - \beta) \sum_{i=1}^{n} (v_{t,i} d_{t,i} + (1 - v_{t,i}) d'_{t,i}) \\
&= -(1 - \beta) \sum_{i=1}^{n} L_t,
\end{aligned}$$

which ends up in the following tighter bound

$$\sum_{t=1}^{T} L_t \leq \min_P \left(\frac{\ln(1/\beta)}{1 - \beta} \sum_{t=1}^{T} l_{t,P} + \frac{1}{1 - \beta} \ln \frac{1}{w_{1,P}} \right).$$

So why did the ratio $\frac{W_{t+1}}{W_t}$ decompose into a product? This is because the graph of Figure 2 may be seen as sequential composition of n two-node graphs, where there are always two edges connecting the pairs of nodes. As you go from the source to the sink (from left to write), the choices at each node (top versus bottom edge) are "independent" of the previous choices of the edges. In the next section we will make this precise. A sequential composition will cause the kernel computation to decompose into a product (And conveniently logs of products become sums of logs). The graph of Figure 2 represents n sequential binary choices, i.e. a product of Bernoulli variables and this was used to model the BEG algorithm for learning disjunctions [HPW02] (A literal in the disjunction is either on or off). The direct algorithm maintains one weight per disjunction (subset of $\{0, 1, \ldots, n\}$). If the loss of a disjunction is measured with the attribute loss then the loss of a disjunction is the sum of the losses of its literals. We obtain an efficient indirect algorithm by maintaining only one weight per

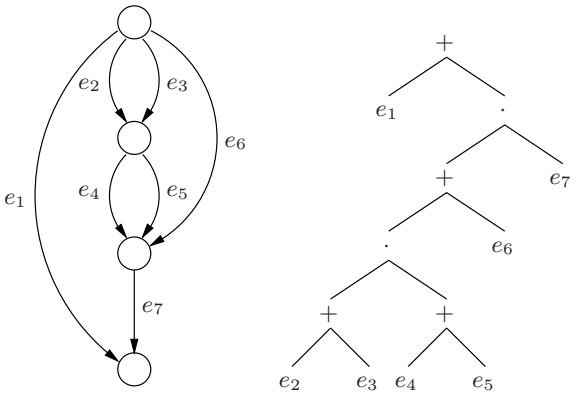

Fig. 3. An example of an SP digraph and its syntax tree

literal and the analysis decomposes as shown above. In this case the Weigh-Pushing algorithm converts the update (7) for the path weights to an update for the edge/literal weights, resulting in the standard BEG algorithm for learning disjunctions [HPW02].

5 Regular Expressions for Digraphs

In this section we consider a digraph as an automaton by identifying the source and the sink with the start and the accept state, respectively. The set of all source-sink paths is a regular language and can be expressed as a regular expression. For example the set of paths of the digraph given in Figure 1 is expressed as

$$e_2e_7 + (e_1 + e_2e_3)e_4^*e_5(e_6e_3e_4^*e_5)^*(e_8 + e_6e_7).$$

The convolution kernel based on regular expressions is introduced by Haussler [Hau99] and is used to measure the similarity between strings. Here we show that the kernel computation gives an efficient implementation for the Weight Pushing algorithm. For the sake of simplicity we assume that the source has no incoming and the sink no outgoing edges, respectively.

5.1 Series Parallel Digraphs

First let us consider the simple case where regular expressions do not have the $*$-operation. In this case the prescribed graphs are series parallel digraphs (SP digraphs, for short). See an example of SP digraphs with its regular expression (in terms of the syntax tree) in Figure 3. Note that any edge symbol e_i appears exactly once in the regular expression that represents an SP digraph. The syntax tree is used to compute the dot product $\Phi(\boldsymbol{v}) \cdot \Phi(\boldsymbol{x})$: Replace the leaves labeled edge e_i by the input product v_ix_i and $+$ by plus $(+)$ and $\circ$ by times $(\cdot)$ (See

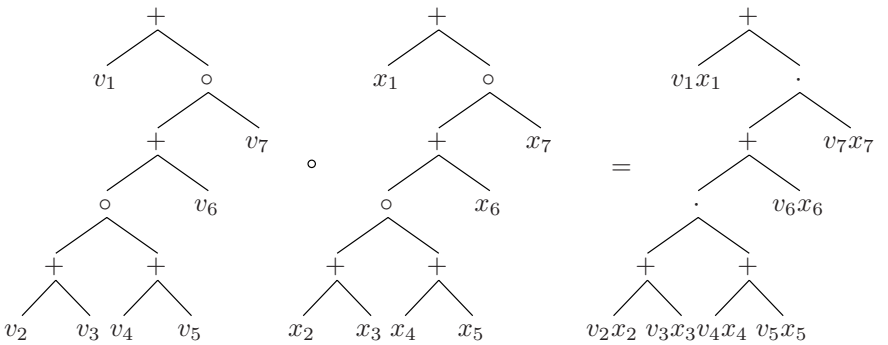

Fig. 4. The dot product $\Phi(\boldsymbol{v}) \cdot \Phi(\boldsymbol{x})$

Fig. 4). Now the value of the dot product is computed by a postorder traversal of the tree. This takes time linear in the size of the tree (number of edges).

Although SP digraphs seem a very restricted class of acyclic digraphs, they can define some important kernels such as the polynomial kernel: $\Phi(\boldsymbol{z}) \cdot \Phi(\boldsymbol{x}) = (1 + \boldsymbol{z} \cdot \boldsymbol{x})^k$, for some degree k. This is essentially the path kernel defined by the regular expression

$$(\epsilon + e_{1,1} + \cdots + e_{1,n})(\epsilon + e_{2,1} + \cdots + e_{2,n}) \cdots (\epsilon + e_{k,1} + \cdots + e_{k,n}),$$

where any set of k edges $e_{1,i}, \ldots e_{k,i}$ happen to receive the same input values z_i and x_i and the edges labeled with ϵ always receive one as input. Note that $\boldsymbol{z}$ does not necessarily represent edge weights that define a probability distribution over the paths. But if all z_i's are non-negative, then the Weight Pushing algorithm when applied to the weights z_i produces the edge weights $v_{j,i} = z_i/(1+z_1+\cdots+z_n)$ for the edges $e_{j,i}$ and and $v_{0,i} = 1/(1+z_1+\cdots+z_n)$ for all ϵ edges. The edge weights $\boldsymbol{v}$ of resulting kernel now satisfy Property 1 and 2, and $\Phi(\boldsymbol{v}) \cdot \Phi(\boldsymbol{x}) = \Phi(\boldsymbol{z}) \cdot \Phi(\boldsymbol{x})/(1 + z_1 + \cdots + z_n)^k$. This kernel (without the ϵ edges) is the path kernel underlying the Normalized Winnow algorithm [HPW02]. Also the case $k = 2$ (related to the kernel associated with Figure 2) is perhaps the simplest kernel of this type:

$$\Phi(\boldsymbol{z}) \cdot \Phi(\boldsymbol{x}) = \sum_{A \subseteq \{1,\ldots,n\}} \prod_{i \in A} z_i x_i = \prod_{i=1}^{n} (1 + z_i x_i).$$

This is the "subset kernel" introduced in [KW97] (also the "monomial kernel" of [KRS01]) which was the initial focal point of this research.

Now we give an efficient implementation for the Weight Pushing algorithm that guarantees normalized updates (1), namely,

$$\prod_{e \in P} \tilde{v}_e = \frac{\prod_{e \in P} v_e b_e}{\sum_{P'} \prod_{e \in P'} v_e b_e} = \frac{\prod_{e \in P} v_e b_e}{\Phi(\boldsymbol{v}) \cdot \Phi(\boldsymbol{b})} \tag{8}$$

for any path P. Each node of the syntax tree corresponds to a regular expression which represents a component of the given SP digraph G. Note that the root of the tree corresponds to the entire graph G. For any component regular expression H of G, let $\mathcal{P}(H)$ denote the set of all paths that H generates. For values $\boldsymbol{x}$ on the edges of H, let $\Phi^H(\boldsymbol{x}) = \sum_{P\in\mathcal{P}(H)} \prod_{e\in P} x_e$. Furthermore, we define the kernel associated with H as

$$K^H(\boldsymbol{v},\boldsymbol{b}) = \Phi^H(\boldsymbol{v})\cdot\Phi^H(\boldsymbol{b}) = \sum_{P\in\mathcal{P}(H)}\prod_{e\in P} v_e b_e.$$

As stated in the beginning of this section, this is recursively calculated as follows.

$$K^H(\boldsymbol{v},\boldsymbol{b}) = \begin{cases} v_e b_e & \text{if } H = e \text{ for some single edge } e,\\ \prod_{i=1}^k K^{H_i}(\boldsymbol{v},\boldsymbol{b}) & \text{if } H = H_1\circ\cdots\circ H_k,\\ \sum_{i=1}^k K^{H_i}(\boldsymbol{v},\boldsymbol{b}) & \text{if } H = H_1+\cdots+H_k. \end{cases}$$

Now we give an update rule for $\boldsymbol{v}$. For any edge e in H, let $\tilde{v}_e^H$ be defined recursively as

$$\tilde{v}_e^H = \begin{cases} 1 & \text{if } H = e \text{ for some single edge } e,\\ \frac{K^{H_i}(\boldsymbol{v},\boldsymbol{b})}{K^H(\boldsymbol{v},\boldsymbol{b})}\tilde{v}_e^{H_i} & \text{if } H = H_1+\cdots+H_k \text{ and } e \text{ is the prefix of some path } P\in\mathcal{P}(H_i),\\ \tilde{v}_e^{H_i} & \text{otherwise, where } H_i \text{ is a child node of } H \text{ for which } e\in P \text{ for some path } P\in\mathcal{P}(H_i). \end{cases}$$

Finally the weights on the edges are given by $\tilde{v}_e = \tilde{v}_e^G$. The next theorem shows that this update assures (8).

Theorem 1. *For any component graph H of G and any path $P\in\mathcal{P}(H)$,*

$$\prod_{e\in P}\tilde{v}_e^H = \frac{\prod_{e\in P} v_e b_e}{K^H(\boldsymbol{v},\boldsymbol{b})}.$$

Proof. We show the theorem by an induction on the depth of the syntax tree.

In the case where H consists of a single edge, the theorem trivially holds.

Consider the case where $H = H_1\circ\cdots\circ H_k$. In this case any path $P\in\mathcal{P}(H)$ is a union $P = P_1\cup\cdots\cup P_k$ for some $P_i\in\mathcal{P}(H_i)$ for $1\le i\le k$. So

$$\begin{aligned} \prod_{e\in P}\tilde{v}_e^H &= \prod_{i=1}^k\prod_{e\in P_i}\tilde{v}_e^H\\ &= \prod_{i=1}^k\prod_{e\in P_i}\tilde{v}_e^{H_i} && \text{by the definition of } \tilde{v}_e^H\\ &= \prod_{i=1}^k\frac{\prod_{e\in P_i} v_e b_e}{K^{H_i}(\boldsymbol{v},\boldsymbol{b})} && \text{by the induction hypothesis}\\ &= \frac{\prod_{e\in P} v_e b_e}{K^H(\boldsymbol{v},\boldsymbol{b})}. \end{aligned}$$

Finally for the case where $H = H_1 + \cdots + H_k$, any path $P \in \mathcal{P}(H)$ is a path in some $\mathcal{P}(H_i)$. Moreover only the prefix edge of P changes its weight. Therefore,

$$\begin{aligned}\prod_{e\in P} \tilde{v}_e^H &= \frac{K^{H_i}(\boldsymbol{v},\boldsymbol{b})}{K^H(\boldsymbol{v},\boldsymbol{b})} \prod_{e\in P} \tilde{v}_e^{H_i} \\ &= \frac{K^{H_i}(\boldsymbol{v},\boldsymbol{b})}{K^H(\boldsymbol{v},\boldsymbol{b})} \frac{\prod_{e\in P} v_e b_e}{K^{H_i}(\boldsymbol{v},\boldsymbol{b})} \quad \text{by the induction hypothesis} \\ &= \frac{\prod_{e\in P} v_e b_e}{K^H(\boldsymbol{v},\boldsymbol{b})},\end{aligned}$$

which completes the proof. □

It is not hard to see that the weights $\tilde{\boldsymbol{v}}$ can be calculated in linear time.

5.2 Predicting Nearly as Well as the Best Pruning

One of the main representations of Machine Learning is decision trees. Frequently a large tree is produced initially and then this tree is pruned for the purpose of obtaining a better predictor. A pruning is produced by deleting some nodes in the tree and with them all their successors. Although there are exponentially many prunings, a recent method developed in coding theory and machine learning makes it possible to maintain one weight per pruning. In particular, Helmbold and Schapire [HS97] use this method to design an elegant multiplicative algorithm that is guaranteed to predict nearly as well as the best pruning of a decision tree in the on-line prediction setting. Recently, the authors [TW99] generalizes the pruning problem to the much more general class of acyclic planar digraphs. In this section we restate this result in terms of path kernels on SP digraphs.

A pruning of an SP digraph G is a minimal set of edges (a cut) that interrupts all paths from the source to the sink. In the example of Fig. 3, all the prunings of the graph are $\{e_1, e_2, e_3, e_6\}$, $\{e_1, e_4, e_5, e_6\}$, $\{e_1, e_7\}$. The key property we use below is that for any path P and pruning R, the intersection always consists of exactly one edge. Let $P \cap R$ denote the edge.

First we describe an algorithm working on the feature space that performs nearly as well as the best pruning of a given SP digraph. The algorithm maintains a weight $w_{t,R}$ per pruning R. Previously an input vector $\boldsymbol{x}_t$ at trial t is given to all edges. Now an input vector $\boldsymbol{x}_t$ is given only to the edges of a single path P_t. In the decision tree example this path is produced by some decision process that passes down the tree. We do not need to be concerned with how the path P_t at trial t is produced. Each pruning R predicts as the edge that cuts the path P_t, i.e. the prediction of R is $x_{t,R} = x_{t,P_t\cap R}$. The algorithm predicts with the weighted average of predictions of prunings, i.e. $\hat{y}_t = \sum_R w_{t,R} x_{t,R}$. Then an outcome y_t is given and the algorithm suffers loss $L_t = L(y_t, \hat{y}_t)$ for a previously fixed loss function L, say, $L(y_t, \hat{y}_t) = (y_t - \hat{y}_t)^2$. Similarly pruning R suffers loss

$L(y_t, x_{t,R})$. Finally the algorithm updates weights so that

$$w_{t+1,R} = \frac{w_{t,R} \exp(-\eta L(y_t, x_{t,R}))}{\sum_R w_{t,R} \exp(-\eta L(y_t, x_{t,R}))}$$

holds for all R.

Now we express this algorithm in terms of path kernels and weight updates on the edges. Consider the dual SP digraph G^D which is defined by swapping the $+$ and $\circ$ operations in the syntax tree. Note that prunings and paths in G are swapped in G^D, namely, a pruning R in G is a path in G^D and the path P_t in G is a pruning in G^D. The algorithm works on the dual graph G^D and maintains weights $\boldsymbol{v}_t$ on the edges of G^D that represent the weight of path R as $w_{t,R} = \prod_{e \in R} v_{t,e}$. At trial t the input vector $\boldsymbol{x}_t$ is assigned to the edges of a pruning P_t. The $x_{t,e}$ for which $e \notin P_t$ are set to one. Let $\boldsymbol{\Phi}^D(\boldsymbol{x}_t)$ denote the vector of all feature values corresponding to the paths in G^D. Then the value of feature/path R is

$$\boldsymbol{\Phi}^D(\boldsymbol{x}_t)_R = \prod_{e \in R} x_{t,e} = x_{t,P_t \cap R} = x_{t,R}.$$

So the prediction $\hat{y}_t$ of the algorithm is expressed as $\hat{y}_t = \boldsymbol{\Phi}^D(\boldsymbol{v}_t) \cdot \boldsymbol{\Phi}^D(\boldsymbol{x}_t)$, which is efficiently computed by a path kernel.

Moreover, since the loss of path R at trial t is $L(y_t, x_{t,R}) = L(y_t, x_{t,P_t \cap R})$, letting

$$b_{t,e} = \begin{cases} \exp(-\eta L(y_t, x_{t,e})) & \text{if } e \in P_t, \\ 1 & \text{otherwise,} \end{cases}$$

we have

$$\prod_{e \in R} v_{t+1,e} = \frac{\prod_{e \in R} v_{t,e} b_{t,e}}{\sum_R \prod_{e \in R} v_{t,e} b_{t,e}}.$$

This is the same form as (8) and hence can be efficiently calculated as shown in the previous section.

Note that in each trial t only the edges in the input path P_t are "relevant" to computing the prediction value $\hat{y}_t$ and the new edge weights $\boldsymbol{v}_{t+1}$. Maintaining $K^H(\boldsymbol{v}_t, \boldsymbol{b}_t)$ for each component expression H, we can compute $\hat{y}_t$ and $\boldsymbol{v}_{t+1}$ in time linear in the size of P_t [TW99], which can be much smaller than the number of all edges. So when the given graph is series-parallel, the implementation we give in this section can be exponentially faster than the original Weight Pushing algorithm.

5.3 Allowing $*$-Operations

Now we consider the general case where regular expressions have $*$-operations. First let us clarify how the digraph is defined by $*$-operations. Let H_1 be any digraph that is defined by a regular expression. Then we define the digraph for $H = H_1^*$ as in Figure 5: add a new source with an ϵ-edge from it to the old

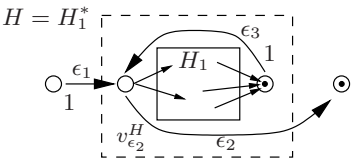

Fig. 5. The digraph defined by H_1^*. The solid box represents H_1, where its source and sink are designated as ◯ and ⊙, respectively

source, add a new sink with an ϵ-edge from the old source to the new sink, and add an ϵ-edge from the old sink back to the old source. For convenience we call them ϵ_1, ϵ_2 and ϵ_3, respectively. All ϵ-edges are assumed to always receive one as input, namely, $b_{\epsilon_i} = 1$. Let $v^H_{\epsilon_i}$ denote the weight for the edge ϵ_i. Clearly it always holds that $v^H_{\epsilon_3} = 1$.

The kernel $K^H(\boldsymbol{v}, \boldsymbol{b}) = \sum_{P \in \mathcal{P}(H)} \prod_{e \in P} v_e b_e$ can be calculated in the same way as stated in Section 5.1 when H is a concatenation or a union. When $H = H_1^*$ for some regular expression H_1, it is not hard to see that

$$K^H(\boldsymbol{v}, \boldsymbol{b}) = v^H_{\epsilon_1} \sum_{k=0}^{\infty} \left(K^{H_1}(\boldsymbol{v}, \boldsymbol{b})\right)^k v^H_{\epsilon_2} = v^H_{\epsilon_1} v^H_{\epsilon_2} / \left(1 - K^{H_1}(\boldsymbol{v}, \boldsymbol{b})\right)$$

if $K^{H_1}(\boldsymbol{v}, \boldsymbol{b}) < 1$ and $K^H(\boldsymbol{v}, \boldsymbol{b}) = \infty$ otherwise.

Similarly the update rule for $\boldsymbol{v}$ is computed through $\tilde{v}^H_e$. The definition of $\tilde{v}^H_e$ is the same as in Section 5.1 when H is a concatenation or a union. When $H = H_1^*$

$$\tilde{v}^H_e = \begin{cases} v^H_{\epsilon_1} & \text{if } e = \epsilon_1, \\ K^{H_1}(\boldsymbol{v}, \boldsymbol{b}) \tilde{v}^{H_1}_e & \text{if } e \text{ is the prefix of some path } P \in \mathcal{P}(H_1), \\ v^H_{\epsilon_2} / \left(1 - K^{H_1}(\boldsymbol{v}, \boldsymbol{b})\right) & \text{if } e = \epsilon_2 \\ \tilde{v}^{H_1}_e & \text{otherwise.} \end{cases}$$

Then we can show that the weights $\tilde{v}^H_e$ satisfy

$$\prod_{e \in P} \tilde{v}^H_e = \frac{\prod_{e \in P} v_e b_e}{K^H(\boldsymbol{v}, \boldsymbol{b})}$$

for any path $P \in \mathcal{P}(H)$.

6 Conclusion

In this paper we showed that path kernels can be used to indirectly maintain exponentially many path weights. Multiplicative updates give factors to the edges and the Weight Pushing algorithm renormalizes the edge weights so that the

outflow out of each vertex is one. We also showed that it is often convenient to express the path sets as regular expressions, leading to efficient implementations of path kernels and the Weight Pushing algorithm. We gave the path kernels that interpret BEG and normalized Winnow as direct algorithms over exponentially many paths. A similar interpretation for Winnow is unknown [HPW02].

Acknowledgment

We thank Fernando Pereira for pointing out that we rediscovered the Weight Pushing algorithm of Mehryar Mohri and Sandra Panizza for fruitful discussions.

References

[FS97] Yoav Freund and Robert 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, August 1997. 80

[Hau99] David Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, Univ. of Calif. Computer Research Lab, Santa Cruz, CA, 1999. 74, 83

[HPW02] D. P. Helmbold, S. Panizza, and M. K. Warmuth. Direct and indirect algorithms for on-line learning of disjunctions. *Theoretical Computer Science*, 2002. To appear. 76, 82, 83, 84, 89

[HS97] D. P. Helmbold and R. E. Schapire. Predicting nearly as well as the best pruning of a decision tree. *Machine Learning*, 27(01):51–68, 1997. 76, 86

[KRS01] Roni Khardon, Dan Roth, and Rocco Servedio. Efficiency versus convergence of Boolean kernels for on-line learning algorithms. In *Advances in Neural Information Processing Systems 14*, 2001. 78, 84

[KW97] J. Kivinen and M. K. Warmuth. Additive versus exponentiated gradient updates for linear prediction. *Information and Computation*, 132(1):1–64, January 1997. 84

[LW94] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Inform. Comput.*, 108(2):212–261, 1994. 80

[Moh98] Mehryar Mohri. General algebraic frameworks and algorithms for shortest distance problems. Technical Report 981219-10TM, AT&T Labs-Research, 1998. 75, 77

[MW98] M. Maass and M. K. Warmuth. Efficient learning with virtual threshold gates. *Information and Computation*, 141(1):66–83, February 1998. 76

[TW99] Eiji Takimoto and Manfred K. Warmuth. Predicting nearly as well as the best pruning of a planar decision graph. In *10th ALT*, volume 1720 of *Lecture Notes in Artificial Intelligence*, pages 335–346, 1999. To appear in *Theoretical Computer Science*. 76, 86, 87

[Wat99] Chris Watkins. Dynamic alignment kernels. Technical Report CSD-TR-98-11, Royal Holloway, University of London, 1999. 74