On Fast and Simple Algorithms for Finding Maximal Subarrays and Applications in Learning Theory

Andreas Birkendorf

Lehrstuhl Informatik II, Universität Dortmund, D-44221 Dortmund, Germany
birkendo@Ls2.informatik.uni-dortmund.de

Abstract. Consider the following problem SUB(k,n).

> Given an array $A = (a_1, \ldots, a_n)$ of real elements a_i and a natural number k, find (at most) k disjoint subarrays $A_1, \ldots, A_k$ in A such that the sum of the elements contained in the subarrays is maximum.

In this paper, we present a simple algorithm, based on Dynamic Programming, solving SUB(k,n) in time $O(kn)$. Extracting the main idea of the dynamic programming scheme, we are able to extend the algorithm such that it is applicable for a wider class of related optimization problems. We show efficient applications of this algorithm in the area of agnostic learning by means of minimum disagreement, such as learning k intervals or identifying objects in a pixel matrix. In particular, the algorithm enables us to generate optimal functions inside a special class of piecewise constant functions. In restricted settings, our algorithm is better by a factor of sample-size-order compared to solutions induced by the dynamic scheme presented in [5].
Furthermore, we develop a generalization of a tree data structure introduced in [1]. Using this tree structure, we can solve corresponding online learning tasks efficiently.

1 Introduction

Consider the following problem.

> Given an array $A = (a_1, \ldots, a_n)$ of real elements a_i and a natural number k, find (at most) k disjoint subarrays $A_1, \ldots, A_k$ in A such that the sum of the elements contained in the subarrays is maximum.

More precisely, this problem, which we denote by SUB(k,n), is given as follows. For a subarray $A' = (a_i, \ldots, a_j)$ of A, let $\|A'\|$ denote the sum $a_i + \ldots + a_j$. Hence, we are searching for $k' \leq k$ subarrays $A_i = (a_{l(i)}, \ldots, a_{r(i)})$, $1 \leq i \leq k'$, $l(i) \leq r(i) \leq l(i+1) - 2$, such that $\|A_1\| + \ldots + \|A_{k'}\|$ is maximum.

Beside the fact that this optimization problem is interesting for its own sake, applications of this problem arise in computational geometry, share analysis, pattern recognition, and learning theory.

For instance, assume that A reflects the development of a special share during a period of n days, i.e., a_i denotes the difference between the share price on day i and day $i-1$. Now, a solution for $\mathrm{SUB}(k,n)$ corresponds to the k periods of time with maximum absolute profit.

This paper deals with efficient algorithms for $\mathrm{SUB}(k,n)$ and related problems in the area of *agnostic learning*. In this learning model, no assumptions concerning the target, that should be learned, is made. Thus, the learner attempts to approximate the unknown target by seeking elements inside a limited hypothesis space, which describe the target in an optimal manner.

A sufficient condition to ensure agnostic learning formally is, e.g., a finite Vapnik-Chervonenkis-dimension or a finite pseudo-dimension of the underlying hypothesis space (see, e.g., [2]). This condition will be guaranteed in the sequel.

The paper is organized as follows.

In Section 2, we present a simple algorithm, named $\mathrm{DYN}_{\mathrm{SUB}}$, solving $\mathrm{SUB}(k,n)$ in time $O(kn)$. The algorithm is based on Dynamic Programming. [1] A slight modification of the algorithm enables us to solve the exact variant of this problem where $k' = k$.

In Section 3, we extract the main idea of the dynamic scheme and formulate a generalized algorithm DYN, which is applicable for a wider class of related optimization problems.

In Section 4, we examine concrete applications of $\mathrm{DYN}_{\mathrm{SUB}}$ and DYN in the area of agnostic learning by means of *minimum disagreement*. We consider simple structured learning tasks in the fields of computational geometry, mathematics, and pattern recognition.

- First, we review the problem of learning k intervals with minimum error, given a sorted sample of size n. Algorithms, presented in [6] and announced in [4], solve this problem in time $O(k^2 n)$ and $O(n \log n)$, respectively. A simple reduction to $\mathrm{SUB}(k,n)$ shows a time bound of $O(kn)$ for this problem using $\mathrm{DYN}_{\mathrm{SUB}}$. Hence, for $k = o(\log n)$ (e.g., k constant), $\mathrm{DYN}_{\mathrm{SUB}}$ beats these algorithms.
- After that, we point out an application in the area of *function learning*. Here, DYN enables us to generate optimal piecewise constant functions with a discrete range according to a given sample. If we measure optimality by the linear loss function, our algorithm is better by a factor of sample-size-order compared to solutions induced by the dynamic programming scheme presented in [5] for this kind of problems.

[1] We learnt from a referee that the same time bound can be obtained from [3] (Theorem 3) by carrying over and generalizing the formalism introduced there. However, we think that our algorithm completes and improves the algorithm presented in [3] in the following ways. Our algorithm is simpler (no preprocessing step has to be performed). Its proof of correctness is more or less evident. Our dynamic scheme can be generalized easily (see Section 3).

– At last, we briefly point out how DYN can be used as a tool to identify two dimensional objects in a pixel matrix.

In Section 5 finally, we design a special data structure, called DYN-tree. A DYN-tree is a generalization of a data structure, introduced in [1], with efficient update facilities. If we consider online variants of the problems, introduced in Section 4, we can solve them efficiently using DYN-trees.

2 The Algorithm $\mathbf{DYN}_{\mathrm{SUB}}$

The general idea of the algorithm $\mathrm{DYN}_{\mathrm{SUB}}$ is to use Dynamic Programming in the following way.

Instead of solving $\mathrm{SUB}(k,n)$ in a direct manner, we first divide it into two disjoint subproblems which are easy to solve using Dynamic Programming. In a final step, an optimal solution for $\mathrm{SUB}(k,n)$ is obtained simply by choosing the optimal subsolution.

Let $\mathrm{SUB}_0(k,n)$ denote the same problem as $\mathrm{SUB}(k,n)$, but with the additional restriction that a_n must *not* be contained in any subarray A_j of the solution. Analogously, let $\mathrm{SUB}_1(k,n)$ denote the problem $\mathrm{SUB}(k,n)$ with the additional demand that a_n must be contained in a subarray A_j of the solution.
Hence, if we define $G_t(k,n)$, $t \in \{0,1\}$, to be the value ("*gain*") of an optimal solution for $\mathrm{SUB}_t(k,n)$, the maximum gain $G(k,n)$ of $\mathrm{SUB}(k,n)$ is given by $G(k,n) = \max\{G_0(k,n), G_1(k,n)\}$.

The following lemma provides a dynamic programming scheme for the computation of these gains.

Lemma 1. *1.* $\forall t \in \{0,1\}, \forall i \in \{0,\dots,k\}, \forall j \in \{0,\dots,n\}:$
$G_t(i,0) = G_t(0,j) = 0.$

2. $\forall i \in \{1,\dots,k\}, \forall j \in \{1,\dots,n\}:$
$G_1(i,j) = a_j + \max\{G_0(i-1,j-1), G_1(i,j-1)\}.$
$G_0(i,j) = \max\{G_0(i,j-1), G_1(i,j-1)\}.$

Proof: Part 1. of the lemma follows by definition of $G_t(\cdot,\cdot)$.

Let S_t, $t \in \{0,1\}$, denote the set of array elements contained in *any* optimal solution for $\mathrm{SUB}_t(i,j)$. Let G_t denote its corresponding gain.
If $a_{j-1} \in S_1$ then $G_1 - a_j$ is the maximum gain of $\mathrm{SUB}_1(i,j-1)$.
If $a_{j-1} \notin S_1$ then $G_1 - a_j$ is the maximum gain of $\mathrm{SUB}_0(i-1,j-1)$.
If $a_{j-1} \in S_0$ then $G_0 - a_j$ is the maximum gain of $\mathrm{SUB}_1(i,j-1)$.
If $a_{j-1} \notin S_0$ then $G_0 - a_j$ is the maximum gain of $\mathrm{SUB}_0(i,j-1)$.
It follows from these four implications that the matrices $G_0(i,j)$, $G_1(i,j)$ can be row-wise (or column-wise) completed as stated in the lemma. ■

```
(* PHASE 1: Computation of Maximum Gains *)
for t = 0 to 1 do
    for i = 1 to k do G(t, i, 0) ← 0;  od;
    for j = 0 to n do G(t, 0, j) ← 0;  od;
od
for i = 1 to k do
    for j = 1 to n do
        G(0, i, j) ← max{G(0, i, j − 1), G(1, i, j − 1)};
        G(1, i, j) ← a_j + max{G(0, i − 1, j − 1), G(1, i, j − 1)};
    od;
od;
(* PHASE 2: Computation of an Optimal Solution via Backtracking *)
i ← k;  if G(0, k, n) < G(1, k, n) then S ← {a_n};  else S ← ∅;  fi;
for j = n − 1 downto 1 do
    if a_{j+1} ∉ S then  (* solve SUB_0(i, j + 1) *)
                        if G(0, i, j) < G(1, i, j) then S ← S ∪ {a_j};  fi;
                   else  (* solve SUB_1(i, j + 1) *)
                        if G(0, i − 1, j) < G(1, i, j) then S ← S ∪ {a_j};
                                                        else i ← i − 1;  fi;
    fi;
od;
output S.
```

Fig. 1. The algorithm $\mathrm{DYN}_{\mathrm{SUB}}$ solving $\mathrm{SUB}(k, n)$ in $O(kn)$ steps.

Equipped with Lemma 1, we can develop the algorithm $\mathrm{DYN}_{\mathrm{SUB}}$ shown in Fig. 1 which solves $\mathrm{SUB}(k, n)$ in $O(kn)$ steps.

In order to achieve an efficient implementation, we first compute the gains $G_t(i, j)$ in a three dimensional table G.

The correctness of PHASE 1 follows immediately from Lemma 1.

In PHASE 2, we create a set S of array elements representing an optimal solution for $\mathrm{SUB}(k, n)$. According to the statements of Lemma 1, a maximum gain of $\mathrm{SUB}_0(i, j)$ is given by $\max\{G_0(i, j-1), G_1(i, j-1)\}$ and a maximum gain $G_1(i, j)$ of $\mathrm{SUB}_1(i, j)$ is given by $a_j + \max\{G_0(i-1, j-1), G_1(i, j-1)\}$.
Hence, starting with $\max\{G_0(k, n), G_1(k, n)\}$, we can construct S by comparing $G_0(\cdot, j)$ and $G_1(\cdot, j)$ for the actual subproblem $\mathrm{SUB}_0(i, j+1)$ or $\mathrm{SUB}_1(i, j+1)$, respectively. Whenever $G_0(\cdot, j) < G_1(\cdot, j)$, we add a_j to our actual solution set S and proceed with the corresponding subproblem $\mathrm{SUB}_t(\cdot, j)$.

Since table G has size $2(k+1)(n+1)$ and each entry is obtained in constant time, PHASE 1 needs $O(kn)$ steps. The second phase has running time $O(n)$ which is evident from Fig. 1. We obtain

Theorem 2. *The problem SUB(k, n) can be solved in $O(kn)$ steps.*

It is possible to sharpen the achieved time bound since w.l.o.g the input of our algorithm is an array $A^* = (a_1^*, \ldots, a_N^*)$, N odd, where $a_1^* > 0$, $a_2^* < 0$, $a_3^* > 0, \ldots$, $a_{N-1}^* < 0$, $a_N^* > 0$. Otherwise, this *normal form* can be easily obtained from $A = (a_1, \ldots, a_n)$ in a linear time preprocessing step.[2]

Corollary 3. *Given an array A of size n, the problem SUB(k, n) can be solved in time $O(n + kN)$, where N denotes the size of the normal form for A.*

Consider the problem $\mathrm{SUB}^*(k, n)$, which is the same as $\mathrm{SUB}(k, n)$, but with the additional demand that *exactly* k subarrays are required. The following slight modification of $\mathrm{DYN_{SUB}}$ leads to the algorithm $\mathrm{DYN_{SUB^*}}$ solving the exact problem in $O(kn)$ steps. We only have to take care of the fact that now, by definition, there exists no solution for the problems $\mathrm{SUB}_t(i, j)$ where $2i > j + t$.

Therefore, we introduce a symbolic value '$-\infty$', s.t. $-\infty < r$ for all $r \in \mathbf{R}$, and associate it with unsolvable problem instances, i.e., $G_t(i, j) = -\infty$ iff $\mathrm{SUB}_t(i, j)$ is unsolvable.

For the algorithm it is sufficient to change the initialization in PHASE 1 as shown in Fig. 2. Other unsolvable instances are detected automatically by the dynamic programming scheme. We obtain

Corollary 4. *The problem SUB$^*(k, n)$ can be solved $O(kn)$ steps.*

(∗ *PHASE 1* ∗)
for $t = 0$ **to** 1 **do**
    for $i = 1$ **to** k **do** $G(t, i, 0) \leftarrow -\infty$; **od**;
    for $j = 0$ **to** n **do** ...(∗ *proceed as in Fig. 1* ∗)

Fig. 2. The algorithm $\mathrm{DYN_{SUB^*}}$ solving $\mathrm{SUB}^*(k, n)$ in $O(kn)$ steps.

3 Generalizing the Dynamic Programming Scheme

In this section, we will generalize $\mathrm{DYN_{SUB}}$ and develop an algorithm DYN that solves a wider class of related optimization problems efficiently. Therefore, let us review the strategy of the dynamic programming scheme induced by Lemma 1 in a more abstract way.

[2] Find longest subsequences $(a_i, \ldots, a_j)$ where all entries are positive (or negative, respectively) and replace these sequences by a single elements $a_l^* = a_i + \ldots + a_j$. Now, drop the first (last) element in A^* if negative. Note that A^* may be the empty set.

Given the input array $A = (a_1, \ldots, a_n)$, $\mathrm{DYN}_{\mathrm{SUB}}$ generates a *decomposition* of A into consecutive blocks B_i, i.e., a set of blocks, such that each a_j of A is contained in *exactly* one block B_i. [3] There are two types t of blocks. Either B_i represents a subarray A_j of $\mathrm{SUB}(k, n)$ (case $t = 1$), or B_i represents a block with elements not involved in a solution of $\mathrm{SUB}(k, n)$ (case $t = 0$).

If we associate with each element a_j, contained in a block of type $t(a_j)$, the gain $g(t(a_j), a_j) \stackrel{\Delta}{=} t(a_j) \cdot a_j$ then $\mathrm{DYN}_{\mathrm{SUB}}$ outputs a decomposition of A into at most k blocks of type 1 with maximum overall gain.

Increasing the number of possible block types and replacing the special gain function by arbitrary loss functions, we obtain the following general optimization problem OAD *(Optimal Array Decomposition)*.

Input of $\mathrm{OAD}(k, n)$ is again an array $A = (a_1, \ldots, a_n)$ containing elements of an arbitrary set S. Furthermore, we have m possible types $T = \{0, \ldots, m-1\}$ of blocks and a loss function $\ell : T \times S \rightarrow \mathbf{R}$. For all inputs, ℓ is evaluable in constant time, e.g., given by a two dimensional table.

The goal is to find a decomposition of A into at most k blocks such that the overall loss $\sum_{1 \leq i \leq n} \ell(t(a_i), a_i)$ is minimum.

Note that the problem structure of OAD is slightly different from SUB. For OAD we are seeking at most k blocks; for SUB we are seeking at most k blocks *of type 1*. But, of course, we can reduce the OAD problem to the SUB problem easily.

Let $L(t, k, n)$ denote the minimum loss of the restricted problem $\mathrm{OAD}_t(k, n)$ where we additionally demand $t(a_n) = t$. Then, analogously to the previous section, $\min_{t \in T}\{L(t, k, n)\}$ defines the minimum loss of $\mathrm{OAD}(k, n)$.

Analogous reasonings as in the proof of Lemma 1 show that the following lemma is valid.

Lemma 5. *1.* $\forall t \in T, \forall i \in \{0, \ldots, k\}, \forall j \in \{0, \ldots, n\}$:

$$L(t, i, 0) = L(t, 0, j) = 0.$$

2. $\forall t \in T, \forall i \in \{1, \ldots, k\}, \forall j \in \{1, \ldots, n\}$:

$$L(t, i, j) = \ell(t, a_j) + \min\{L(t, i, j-1), \min_{t' \in T \setminus \{t\}} \{L(t', i-1, j-1)\}\}.$$

Equipped with Lemma 5, the implemention of DYN is more or less straightforward (see Fig. 3). Output of DYN is the array $(t(a_1), \ldots, t(a_n))$. Note that the running time of DYN is dominated by the computation of the $m(k+1)(n+1)$ values $L(\cdot, \cdot, \cdot)$. The computation of each value needs time $O(m)$.

The main result of this section reads as follows.

Theorem 6. *For m types of blocks, the problem $OAD(k, n)$ can be solved in $O(m^2 kn)$ steps.*

[3] Do not confuse block B_i with a subarray A_j of $\mathrm{SUB}(k, n)$.

```
(* PHASE 1: Computation of Minimum Losses *)
foreach t ∈ T do
  for i = 0 to k do L(t, i, 0) ← 0; od;
  for j = 0 to n do L(t, 0, j) ← 0; od;
od
for i = 1 to k do
    for j = 1 to n do
        foreach t ∈ T do
          L(t, i, j) ← ℓ(t, a_j) + min{L(t, i, j − 1), min_{t'∈T\{t}} {L(t', i − 1, j − 1)}};
        od;
    od;
od;
(* PHASE 2: Computation of an Optimal Solution via Backtracking *)
t(a_n) ← argmin_t{L(t, k, n)}; i ← k;
for j = n − 1 downto 1 do
    t ← t(a_{j+1});
    goal ← L(t, i, j + 1) − ℓ(t, a_{j+1});
    if L(t, i, j) = goal then t(a_j) ← t;
                          else begin
                                 search t' ∈ T \ {t} such that L(t', i − 1, j) = goal;
                                 t(a_j) ← t';
                                 i ← i − 1;
                               end;
    fi;
od;
output (t(a_1), . . . , t(a_n)).
```

Fig. 3. The algorithm DYN solving OAD(k, n) in $O(| T^2 | kn)$ steps.

4 Applications in Agnostic Learning

Since $\mathrm{DYN_{SUB}}$ and DYN provide us with optimal solutions inside a limited (hypothesis) space, we are now prepared to give some applications of these algorithms in the area of agnostic learning by means of minimum disagreement (or minimum loss, respectively).

The first application arises in computational geometry.

We define the problem INT(k, n) as follows.

> For a sequence $S = ((x_1, l_1), (x_2, l_2), \ldots, (x_n, l_n))$ of *positive* points (label $l_i = +1$) and *negative* points (label $l_i = -1$) on the real axis at position x_i, find (at most) k disjoint intervals $I_1, \ldots, I_k$ such that the sum $\sum_{i=1}^{n} l_i \cdot \chi_i$ is maximum, where
> $$\chi_i \triangleq \begin{cases} +1, \text{ if } x_i \in I_1 \cup \cdots \cup I_k, \\ -1, \text{ otherwise.} \end{cases}$$

Hence, we try to find a set of intervals that include positive and exclude negative points as far as possible.
If we assume that the sequence S is sorted, i.e., $x_i < x_{i+1}$, this problem is equivalent to SUB(k,n) with input array $A = (l_1, \ldots, l_n)$. We get the following corollary.

Corollary 7. *For a sorted sequence, the problem INT(k,n) can be solved in time $O(n + kN)$, where N denotes the number of changes of labels in the input sequence, i.e., N is the cardinality of the set $\{1 \le i \le n-1 \mid l_i \ne l_{i+1}\}$.*

The second example demonstrates a mathematical application.

For a set $T = \{y_0^*, \ldots, y_{m-1}^*\}$ of m real values, let PCF_T^k denote the class of piecewise constant functions $f : \mathbf{R} \to T$ where the number of constant pieces is bounded by k.

Given a sequence $S = ((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n))$ of real points x_i and real labels y_i, we define the *loss* $L_\ell(f, S)$ of a function f with respect to S by

$$L_\ell(f,S) \triangleq \sum_{i=1}^{n} \ell(f(x_i), y_i).$$

Examples for $\ell(\cdot,\cdot)$ may be the linear loss $\ell_1(y,y') = \mid y - y' \mid$ or the quadratic loss $\ell_2(y,y') = (y-y')^2$.

For a fixed choice of ℓ, the problem PCF(k,n) is to find a function $f \in PCF_T^k$, where $L_\ell(f,S)$ is minimum.

Obviously, we can solve PCF using DYN in a direct manner. We just feed DYN with the input array $A = (y_1, \ldots, y_n)$ (after sorting the sequence S such that $x_i < x_{i+1}$) and associate with the output array $(t(a_1), \ldots, t(a_n))$ the function

$$f(x) \triangleq \begin{cases} y_{t(a_1)}^*, \text{ if } x \le x_1, \\ y_{t(a_i)}^*, \text{ if } x_{i-1} < x \le x_i, \\ y_{t(a_n)}^*, \text{ if } x_n \le x. \end{cases}$$

We obtain

Corollary 8. *For the function class PCF_T^k, the problem PCF(k,n) can be solved in $(\mid T \mid^2 kn)$ steps.*

If we restrict ourself to the linear loss ℓ_1 and labels $y_i \in T$, an optimal function $f \in PCF_T^k$ is even an optimal function in the class $PCF_{\mathbf{R}}^k$, where we allow arbitrary real values for the k pieces.

The linear loss function ℓ_1 guarantees that extending the output range T from discrete values to $\mathbf{R}$ does not yield an improved solution, i.e., for every function $f^* \in PCF_{\mathbf{R}}^k$ there exists a function $f \in PCF_T^k$ with $L_{\ell_1}(f^*, S) = L_{\ell_1}(f, S)$.[4]

[4] The proof of this statement is based on the well known fact that for any sequence $S = (y_1, \ldots, y_n)$ of real numbers the sum $\sum_{i=1}^n \mid y^* - y_i \mid$ is minimized if we choose y^* as the *median* of S.

In [5], a very general scheme for constructing optimal piecewise functions is presented. This scheme would only achieve a running time of $O(\mid T \mid^2 kn^2)$ for the PCF problem in this restricted setting.

At last, we briefly mention a closely related application of the DYN algorithm in the area of pattern recognition. Consider a pixel matrix M where each pixel is of a certain color. (M represents a digitalized picture, for instance.) Let us take the distinct rows r as input of the DYN algorithm. According to different choices of loss functions and different block types, we obtain optimal block structures in r. The block types T may correspond to a (small) set of colors. Hence, we are able to extract a bounded number of "similar colored" blocks in r. Putting these block structures together for all rows r in M, we may identify ("learn") two dimensional objects in M.

5 An On-Line Scenario

Let us review the *Optimal Array Decomposition* problem (OAD), introduced in Section 3, again. This time, we want to examine OAD in an online environment where the input array grows dynamically.

The *online* OAD(k, n) problem is defined by the following process.

Step 0: Start with the empty array $A^{(0)} = ()$.
Step $1 \leq s \leq n$: Given $A^{(s)} = (a_1, \dots, a_s)$, a new element a^* and an array position p, $1 \leq p \leq s+1$, is presented. Solve the problem OAD$(k, s+1)$ with input array $A^{(s+1)} \triangleq (a_1, \dots, a_{p-1}, a^*, a_p, \dots, a_s)$.

Hence, the task is to generate n solutions $S^{(1)}, \dots, S^{(n)}$ of the OAD problem according a growing input array. Because each new element a^* is inserted arbitrarily into $A^{(s)}$, we have no chance to compute $S^{(1)}, \dots, S^{(n)}$ within a single run of the DYN algorithm, even if we may store further useful information.

Note that we need several subsolutions for the subarrays $(a_1, \dots, a_{p-1})$ and $(a_p, \dots, a_s)$ in order to obtain an optimum for the whole array $A^{(s)}$. But the standard DYN programming scheme provides us only with subsolutions for subarrays $(a_1, \dots, a_j)$, $1 \leq j \leq s$. Of course, we can run the DYN algorithm n times yielding an overall running time of $O(m^2kn^2)$. (Remember that m denotes the number of possible block types $t \in T$.)

In the remainder of this section, we design a data structure, called DYN-tree, solving the online OAD(k, n) problem more efficiently for large n. Given $S^{(s)}$, we can generate $S^{(s+1)}$ in time $O(m^3k^2 \log s)$. Hence, applications using DYN-trees beat "n-times offline algorithms" iff $mk = o(n/\log n)$.

A DYN-tree is a labeled binary tree with efficient update facilities.

In the case $m = 2$, a DYN-tree decreases to a tree data structure described in [1]. However, these trees are constructed in a single run after scanning a fixed sample. Update operations are not considered.

Given $A^{(s)} = (a_1, \ldots, a_s)$, each node v of the DYN-tree $T(A^{(s)})$ corresponds to a subarray $A(v) = (a_{l(v)}, \ldots, a_{r(v)})$ of $A^{(s)}$.

Leaves correspond to the n singletons a_i. Inner nodes v with left child u and right child w are associated with the joint subarray $A(v) = (A(u), A(w))$, where $r(u) = l(w) - 1$. Hence, $A(u)$ and $A(w)$ refer to consecutive subarrays in $A^{(s)}$, and the root r of $T(A^{(s)})$ corresponds to $A^{(s)}$.

Beside the array size $s_v = r(v) - l(v) + 1$, each node v contains for all $t_l, t_r \in T$ and for all $k' \in \{1, \ldots k\}$ the following $1 + mk + m^2k$ losses according to $A(v)$.

- $L^{A(v)}(k)$, the minimum loss of a solution for the problem OAD(k, s_v) with input array $A(v)$.
- $L_L^{A(v)}(t_l, k')$, which is defined as $L^{A(v)}(k')$ but with the additional restriction that the block type of the *first* element is t_l, i.e., $t(a_{l(v)}) = t_l$.
- $L_R^{A(v)}(k', t_r)$, which is defined as $L^{A(v)}(k')$ but with the additional restriction that the block type of the *last* element is t_r, i.e., $t(a_{r(v)}) = t_r$.
- $L_{LR}^{A(v)}(t_l, k', t_r)$, which is defined as $L^{A(v)}(k)$ but with restrictions $t(a_{l(v)}) = t_l$ *and* $t(a_{r(v)}) = t_r$.

Hence, the root r of $T(A^{(s)})$ contains the minimum loss of the online OAD(k, s) problem, namely $L^{A(r)}(k)$. If we want to construct an explicit solution $S^{(s)}$ we may perform a backtracking phase, similar the one in Fig. 3.

The key result is the following lemma which is, loosely speaking, the canonical "two-sided" generalization of Lemma 5. ("Two-sided" in the sense that losses of subarrays are considered where the types of *both* border elements are restricted.)

Lemma 9. $\forall t_l, t_r \in T$, $\forall k' \in \{1, \ldots k\}$, $\forall$ *singletons* a, $\forall$ *nodes* u, v, w*:*

1. $L_{LR}^{(a)}(t_l, k', t_r) = \begin{cases} L(t_l(a), a), & \text{if } t_l = t_r, \\ \infty, & \text{otherwise.} \end{cases}$
2. $L_{LR}^{(A(u),A(w))}(t_l, k', t_r) =$
$$\min \left\{ \begin{array}{l} \min\limits_{t \in T,\, 1 \le i \le k'} \{L_{LR}^{A(u)}(t_l, i, t) + L_{LR}^{A(w)}(t, k' + 1 - i, t_r)\}, \\ \\ \min\limits_{1 \le i \le k'} \{L_L^{A(u)}(t_l, i) + L_R^{A(w)}(k' - i, t_r)\} \end{array} \right\}$$
3. $L_L^{A(v)}(t_l, k') = \min\limits_{t \in T}\{L_{LR}^{A(v)}(t_l, k', t)\}$
4. $L_R^{A(v)}(k', t_r) = \min\limits_{t \in T}\{L_{LR}^{A(v)}(t, k', t_r)\}$
5. $L^{A(v)}(k) = \min\limits_{t \in T}\{L_L^{A(v)}(t, k)\} = \min\limits_{t \in T}\{L_R^{A(v)}(k, t)\}$

(Statements 3., 4., and 5. follow by definition whereas statements 1. and 2. are a straightforward generalization of Lemma 5.)

Lemma 9 implies, that, given two consecutive subarrays $A(u)$ and $A(w)$, we can compute all $O(m^2k)$ losses of the joint array $A(v) = (A(u), A(w))$ in time $O((m^2k) \cdot (mk)) = O(m^3k^2)$.

Let d denote the depth of $T(A^{(s)})$. For each new element a^* and array position p, we update $T(A^{(s)})$ in time $O(m^3k^2d)$ as follows.

First, we search the leaf u in $T(A^{(s)})$ containing the singleton a_{p-1}. (If $p = 1$, the following procedure has to be modified in a straightforward way.) Note that the path P from the root of $T(A^{(s)})$ to u can be found easily due to the structure of the DYN-tree.
After that, we replace the leaf u by a new inner node v with left child u and right child w where w is a new leaf associated with the singleton a^*. Finally, we update the data of w, u and all nodes on P with respect to a^* in a bottom-up pass. The resulting tree is a DYN-tree for $A^{(s+1)} = (a_1, \ldots, a_{p-1}, a^*, a_p, \ldots, a_s)$.

In order to keep the DYN-tree structure balanced ($d = O(\log s)$), we can organize a DYN-tree, for instance, as an AVL-tree. Now, in each phase of rebalancing $T(A^{(s)})$ at most $O(\log s)$ nodes are rearranged and their data has to be updated. Hence, one update step of a DYN-tree needs time $O(m^3k^2 \log s)$ and we obtain the following

Theorem 10. *Using a DYN-tree, we can solve the online OAD(k, n) problem in $O(m^3k^2n \log n)$ steps.*

Online variants of the applications, introduced in Section 4, achieve corresponding running times.

Furthermore, one can think of an online scenario where the input array *shrinks* sequentially by elements a_p. Analogous reasonings as above yield the same time bound for this kind of problem.

Acknowledgements

The author would like to thank Hans Ulrich Simon, Paul Fischer, Norbert Klasner, and some unknown referees for many valuable hints and discussions.

The author gratefully acknowledges the support of Deutsche Forschungsgemeinschaft grant Si 498/3-1 and the support of Deutscher Akademischer Austauschdienst grant 322-vigoni-dr.

References

1. Peter Auer, Robert C. Holte, Wolfgang Maass. *Theory and Applications of Agnostic PAC-Learning with Small Decision Trees.* Proceedings of the 12th International Conference on Machine Learning, 1995.
2. Anselm Blumer, Andrej Ehrenfeucht, David Haussler, Manfred K. Warmuth. *Learnability and the Vapnik-Chervonenkis Dimension.* Journal of the Association for Computing Machinery, 36, 1989, p. 929–965.
3. Truxton Fulton, Simon Kasif, Steven Salzberg. *Efficient Algorithms for Finding Multi-way Splits for Decision Trees.* Proceedings of the 12th International Conference on Machine Learning, 1995, p. 244–251.
4. Michael Kearns, Yishay Mansour, Andrew Y. Ng, Dana Ron. *An Experimental and Theoretical Comparison of Model Selection Methods.* Proceedings of the 8th Annual Conference on Computation Learning Theory, 1995. p. 21–30.
5. Michael Kearns, Robert E. Schapire, Linda M. Sellie. *Towards Efficient Agnostic Learning.* Machine Learning, 17, 1994, p. 115–141.
6. Wolfgang Maass. *Efficient Agnostic PAC-Learning with Simple Hypotheses.* Proceedings of the 7th Annual Conference on Computation Learning Theory, 1994.