Information Processing Letters 115 (2015) 186–192

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Greedily computing associative aggregations on sliding windows[☆]

David Basin[a], Felix Klaedtke[b], Eugen Zălinescu[a,*]

[a] Institute of Information Security, ETH Zurich, Switzerland
[b] NEC Europe Ltd., Heidelberg, Germany

ARTICLE INFO

Article history:
Received 19 May 2014
Received in revised form 24 July 2014
Accepted 14 September 2014
Available online 30 September 2014
Communicated by B. Doerr

Keywords:
Sliding window
Associative aggregation operator
On-line algorithms
Complexity
Optimality

ABSTRACT

We present an algorithm for combining the elements of subsequences of a sequence with an associative operator. The subsequences are given by a sliding window of varying size. Our algorithm is greedy and computes the result with the minimal number of operator applications.

1. Introduction

Problem statement Let $\oplus : D \times D \to D$ be an associative operator over a nonempty set D. Consider a sequence $\bar{a} = (a_1, \dots, a_n)$ of elements in D, with $n \geq 1$. A *window* w in $\bar{a}$ is a pair (ℓ_w, r_w) with $1 \leq \ell_w \leq r_w \leq n$. We call ℓ_w and r_w the w's *left* and *right margin* respectively. We omit the subscript w when it is unimportant or clear from the context. Moreover, we write $\oplus_w(\bar{a})$ for $a_{\ell_w} \oplus a_{\ell_w+1} \oplus \cdots \oplus a_{r_w}$.

We consider the following problem in which the number of applications of the $\oplus$ operator should be minimized.

Input: A nonempty sequence $\bar{a}$ of elements in D and a sequence $\bar{w} = (w_1, \dots, w_k)$ of windows in $\bar{a}$, with $\ell_{w_1} \leq \ell_{w_2} \leq \cdots \leq \ell_{w_k}$ and $r_{w_1} \leq r_{w_2} \leq \cdots \leq r_{w_k}$.
Output: The sequence $(\oplus_{w_1}(\bar{a}), \oplus_{w_2}(\bar{a}), \dots, \oplus_{w_k}(\bar{a}))$.

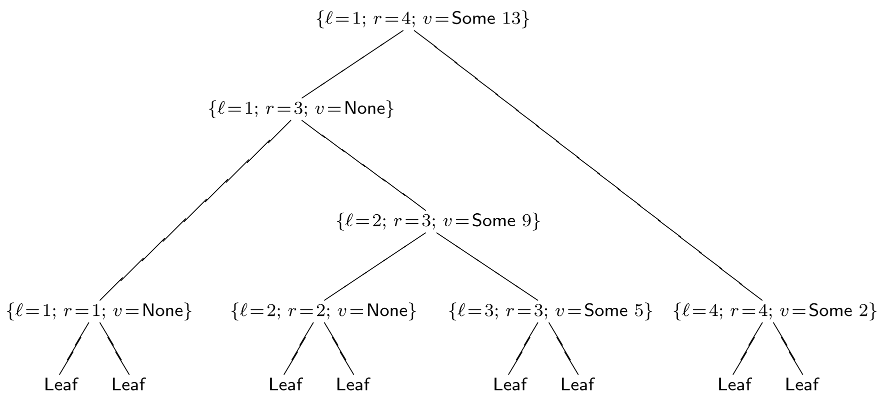
This minimization problem is motivated by settings where $\oplus$'s computation is expensive, for example, when multiplying large matrices, or when taking the union of large finite sets or determining their minimum. This problem arises, for example, when evaluating queries in system monitoring and stream processing, where $\oplus$ is used to aggregate values on windows sliding over data streams.

A straightforward but suboptimal algorithm is to compute $\oplus_{w_i}(\bar{a})$ for each window w_i separately. It is easy to see that this algorithm applies the $\oplus$ operator $\sum_{i=1}^{k}(r_{w_i} - \ell_{w_i})$ times. One can do better by sharing intermediate results between overlapping windows as the following example illustrates.

Example Let D be the domain $\mathbb{N}$ and $\oplus$ integer addition. For the sequence $\bar{a} = (2, 4, 5, 2)$ and the window sequence $\bar{w} = ((1, 3), (1, 4), (2, 4))$, the output is the sequence $(11, 13, 11)$. The straightforward algorithm applies the $\oplus$ operator $2 + 3 + 2 = 7$ times. For this example, the minimal number of $\oplus$ applications is 3, since integer addition is associative and commutative and the windows w_1

☆ This work was partially supported by the Zurich Information Security and Privacy Center (ZISC).
* Corresponding author.

http://dx.doi.org/10.1016/j.ipl.2014.09.009
0020-0190/© 2014 Elsevier B.V. All rights reserved.



$\{\ell=1;\ r=4;\ v=$Some 13$\}$
$\{\ell=1;\ r=3;\ v=$None$\}$
$\{\ell=2;\ r=3;\ v=$Some 9$\}$
$\{\ell=1;\ r=1;\ v=$None$\}$
$\{\ell=2;\ r=2;\ v=$None$\}$
$\{\ell=3;\ r=3;\ v=$Some 5$\}$
$\{\ell=4;\ r=4;\ v=$Some 2$\}$
Leaf

Fig. 1. Instance of the tree data structure used by the algorithm to store intermediate results.

and w_3 contain the same integers. However, the minimal number is 4 if we just exploit the associativity of $\oplus$.

Obviously, when computing $\bigoplus_{w_2}(\bar{a})$ we can reuse the result of the window w_1, since $\bigoplus_{w_2}(\bar{a}) = \bigoplus_{w_1}(\bar{a}) \oplus a_4$. If we compute the intermediate result $h := a_3 \oplus a_4$ when computing the result for the window w_2, we could reuse it for the window w_3, since $\bigoplus_{w_3}(\bar{a}) = a_2 \oplus h$. Note that we do not have h as an intermediate result when computing the results of the previous windows w_1 and w_2 as $(a_1 \oplus a_2) \oplus a_3$ and $((a_1 \oplus a_2) \oplus a_3) \oplus a_4$, respectively. In case we compute the results of the windows w_1 and w_2 as $a_1 \oplus (a_2 \oplus a_3)$ and $a_1 \oplus (a_2 \oplus (a_3 \oplus a_4))$, h is available for the result of the window w_3. However, in this case, the computation of the result of the window w_2 does not use the result of the first window. So how we parenthesize $a_i \oplus a_{i+1} \oplus \cdots \oplus a_j$ is important when computing the result of a window. This choice has an impact on whether we can reuse intermediate results for other windows.

Contributions In this article, we present an efficient algorithmic solution to this problem. Our algorithm, which we present in Section 2 and name SWA, processes the windows iteratively and reuses intermediate results from previously processed windows. SWA is greedy in the sense that it minimizes for each window the number of $\oplus$ applications. In Section 3 we prove SWA's correctness and in Section 4 we show that it has linear running time in the length of the input sequence $\bar{a}$. In Section 5 we prove SWA's optimality with respect to minimizing the number of $\oplus$ applications. We conclude in Section 6 by discussing applications and related work.

2. Algorithm

We present our sliding window algorithm SWA in a functional programming style, close to the OCaml programming language [7].[1] To simplify the exposition, we fix the associative operator $\oplus : D \times D \to D$ and the input sequence $\bar{a} = (a_1, \ldots, a_n)$, i.e., we treat $\oplus$ and $\bar{a}$ as global variables. Our pseudo code can easily be modified so that $\oplus$ and $\bar{a}$ are algorithm parameters. Furthermore, we assume that we can access $\bar{a}$'s element at any position $i \in \{1, \ldots, n\}$ in constant time.

[1] An OCaml implementation is provided as supplementary material on the publisher's website.

SWA uses binary ordered trees to store and reuse intermediate results, which are updated when iteratively processing the window sequence $\bar{w}$. Fig. 1 shows the tree that SWA builds for the window $w_2 = (1, 4)$ for the input from the example in the introduction. Generally, the polymorphic datatype of these trees is

```
type 'a intermediate = 'a option node tree
```

where

```
type 'b node = {ℓ: ℕ; r: ℕ; v: 'b}
type 'c tree =
  | Leaf
  | Node of ('c ∗ ('c tree) ∗ ('c tree))
```

Only the inner nodes of the trees are labeled (cf. the type definition of 'c tree). The content of an inner node (of the type 'b node), which we associate in the following to its subtree t, is a record whose field values are denoted by ℓ_t, r_t, and v_t, respectively. The field values ℓ_t and r_t are elements of $\mathbb{N}$, with $1 \le \ell_t \le r_t \le n$. They describe the elements of $\bar{a}$ that are covered by the tree t and their combination $\bigoplus_{(\ell_t,r_t)}(\bar{a})$ is the field value v_t. If we know that the intermediate result $\bigoplus_{(\ell_t,r_t)}(\bar{a})$ is not reused later, SWA does not store it to reduce memory usage. In this case v_t is actually None; otherwise, v_t is Some $\bigoplus_{(\ell_t,r_t)}(\bar{a})$. We recall that the option type, used in the type definition of 'a intermediate, is defined as

```
type 'a option = None | Some of 'a
```

We lift the $\oplus$ operator in the canonical way to this extended domain. For $t =$ Leaf, we define $\ell_t := r_t := 0$ and $v_t :=$ None. Furthermore, we define the following function for extracting the children of a tree's root:

```
fun children t = match t with
                 | Leaf → error "No children at leaf."
                 | Node (_, t′, t″) → (t′, t″)
```

We first define two basic auxiliary functions for creating and combining trees. The function atomic i builds the single-node tree t with $\ell_t = r_t = i$ and $v_t =$ Some a_i.

```
fun atomic i = Node ({ℓ = i; r = i; v = Some a_i}, Leaf, Leaf)
```

The function combine t' t'' builds the tree t with the left child t' and the right child t'', provided neither t' nor t'' is