Functional Pearl: Maximally Dense Segments

Sharon Curtis
Oxford Brookes University, UK
sharoncurtis@brookes.ac.uk

Shin-Cheng Mu
Academia Sinica, Taiwan
scm@iis.sinica.edu.tw

Abstract

The problem of finding a maximally dense segment (MDS) of a list is a generalisation of the well-known maximum segment sum (MSS) problem, but its solution is more challenging. We extend and illuminate some recent work [2, 3, 7] on this problem with a formal development of a linear-time online algorithm, in the form of a sliding window zygomorphism. The development highlights some elegant properties of densities, involving partitions which are decreasing and all right-skew.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Theory

Keywords program derivation, segment problem, maximum density, sliding window, zygomorphism, right-skew

1. Introduction

The *maximally dense segment* (MDS) problem takes an input list of elements, each of which has an *area* and a strictly positive *breadth*:

$$\begin{array}{ll} area & :: Elem \to \mathbb{R} \\ breadth & :: Elem \to \mathbb{R}^+. \end{array}$$

A *segment* of a list is simply a contiguous subsequence, and the *density* of a (non-empty) list of elements is defined as follows:

$$\begin{array}{l} d \quad :: [Elem] \to \mathbb{R} \\ d\, x = (sum\ (map\ area\ x))/(sum\ (map\ breadth\ x)). \end{array}$$

Densities can be visualised by viewing elements as rectangles, and taking a list of type $[Elem]$ to be a histogram. The density of such a histogram is simply its average height, see Fig. 1, for example.

Merely finding a segment of maximum density is trivial, as a singleton list containing a tallest element will do. However, the problem is made more interesting by putting bounds on the allowed breadths of segments. Since there may be more than one segment with the maximum density, the specification of the MDS problem is relational in nature:

$$mds \quad = \quad max_d \circ \Lambda(bounds? \circ segment).$$

In this specification, and throughout, we will take a relation of type $A \rightsquigarrow B$ to be a subset of the Cartesian product $B \times A$,

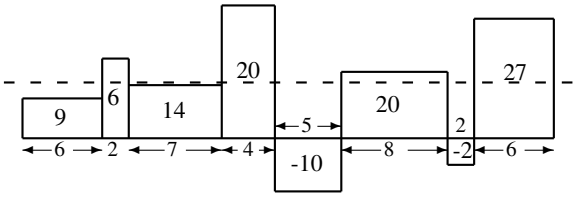

Figure 1. A list of elements, with areas and breadths indicated. The height of the dashed line represents the density of the list.

a function to be a special case of a relation, and $\circ$ is functional or relational composition. Note that for the direction of relational composition to be consistent with that of functional composition, relations take input from the right and output to the left, as do functions. Given relations $prefix$ and $suffix$ which respectively map an input list to a prefix and a suffix, the relation $segment$ is defined by $segment = prefix \circ suffix$. The predicate $bounds$ checks whether the total breadth of a segment lies between a lower bound L and an upper bound U:

$$bounds\ x = L \leq sum\ (map\ breadth\ x) \leq U,$$

and given a predicate $p : A \to Boolean$, the relation $p?$ is given by the relation

$$p? \quad = \quad \{(x, x) |\ x \in A \wedge p\ x\}.$$

The Λ operator converts a relation $R : A \rightsquigarrow B$ to a function $\Lambda R : A \to \mathbb{P}B$, by returning the set of all the values related to the input:

$$\Lambda R\ a = \{b \mid (b, a) \in R\}.$$

Thus the function $\Lambda(bounds? \circ segment)$ takes an input list and returns the set of all segments of that list within the breadth bounds. Then the relation max_d selects a segment which is maximum with respect to function d.

Our aim is a functional program that meets the above specification and computes a maximally dense segment in linear time. As an example, let $(L, U) = (9, \infty)$, and take the list of $(area, breadth)$ pairs $[(9,6),\ (6,2),\ (14,7),\ (20,4),\ (-10,5),\ (20,8),\ (-2,2),\ (27,6)]$ depicted in Fig. 1. The densest segment satisfying the breadth bounds is $[(20,8),\ (-2,2),\ (27,6)]$, with density $45/16 \approx 2.81$.

During this paper, there is just one useful (and easily-proved) property of densities that we shall use: for any lists $x, y :: [Elem]$, and any operator $\oplus \in \{<_d, \leq_d, >_d, \geq_d\}$,

$$x\ \oplus\ (x +\!\!+ y) \quad \Leftrightarrow \quad x \oplus y \quad \Leftrightarrow \quad (x +\!\!+ y)\ \oplus\ y. \qquad (1)$$

As this is the only property of densities we will rely on to develop a linear-time algorithm for MDS, our results are also applicable to other such problems using a different function d satisfying this property.

The MDS problem seems to bear a strong resemblance to the well-known maximum segment sum problem (MSS), a textbook example of program derivation [9], so why is MDS interesting? MDS is interesting to bioinformaticians because finding sections of DNA dense with mutations can provide information about its 3-dimensional structure. However for algorithm designers, MDS is interesting because it turns out to be a lot harder than it looks to come up with an algorithm that is linear, independently of L. Huang [8] presented a variation of MDS without an upper bound on segment length (i.e. with $U = \infty$) and with all elements of breadth 1. He noticed that the optimal segment need not be wider than $2L - 1$, leading to a simple $O(nL)$ algorithm, where n is the number of input elements. Lin et al. [10] introduced a concept called *right-skew partitions* (see Section 4.1) to improve the complexity to $O(n \log L)$. The variation of MDS with an upper bound was first studied by Goldwasser et al. [6], who presented an $O(n)$ algorithm for the case when all elements have a uniform breadth, and an $O(n \log(U - L + 1))$ algorithm for elements of variable breadths. Subsequently, there was a published algorithm that claimed to be linear only to be found wrong later,[1] before both Chung and Lu [2, 3] and Goldwasser et al. [7] refined their algorithms to take linear time for elements of variable breadths. We discovered, however, that both algorithms are still not entirely correct: they both fail for a boundary case (to be shown in Section 3.1). The former could potentially return a invalid result, for which there is an easy fix [1], while the latter loops and it is harder to see whether it is fixable.

We believe that the difficulty is partly due to the complex nature of the problem, and partly due to the absence of a rigorous approach to program construction. The latter two algorithms both maintain invariants that are not explicitly stated and not easy to reconstruct either. The invariants rely on states stored in static variables surviving between subroutine calls, which makes reasoning about them extremely hard. The complicated use of array and index manipulation obscures some beautiful structural properties of segment densities. The MDS problem provides a useful case study of how formal development techniques can help with constructing correct programs.

In Section 2, we review the use of a sliding window technique to solve segment optimisation problems, and show how a computation pattern known as a *zygomorphism* can be used to express such windowing techniques. Section 3 shows how a solution to the MDS problem can be formulated as a zygomorphism. Section 4 is a relatively self-contained section that introduces a very useful partition with respect to densities known as the *DRSP*, and also summarises its properties and shows how to construct such partitions. Section 5 returns to the algorithm development, using the DRSP to provide a more efficient data structure for the MDS algorithm in the case where $U = \infty$. Section 6 is another section concentrating on properties of the DRSP, showing how it can be represented using trees. Returning again to the algorithm development, Section 7 uses a tree representation to further refine the MDS algorithm in the case where $U = \infty$, and Section 8 extends this to the case where the upper bound U is finite. Section 9 concludes. Code and supplementary proofs are available online [12].

2. Techniques for Optimal Segment Problems

Consider more general segment optimisation problems, of the form

$$optseg_{f,p} \quad = \quad max_f \circ \Lambda(p? \circ segment).$$

This specifies a p-satisfying segment of the input sequence that is maximal with respect to the function f. The MDS problem matches this general specification, as $mds = optseg_{d,bounds}$, and there are many other examples of such segment problems in Zantema [15].

When starting a formal development of a specification involving segments, a standard first step is to note that segments are prefixes of suffixes[2]. Thus, within a few short steps, the above specification can be refined to an expression requiring the finding of optimal p-satisfying prefixes of all suffixes of the input:

$$optseg_{f,p} \quad = \quad max_f \circ \mathsf{E}(max_f \circ \Lambda(p? \circ prefix)) \circ \Lambda suffix,$$

where the E operator converts a relation $R : A \rightsquigarrow B$ to a function $\mathsf{E}R : \mathbb{P}A \to \mathbb{P}B$, by applying R in all possible ways:

$$\mathsf{E}R\ as = \{b \mid \exists a \in as : (b, a) \in R\}.$$

2.1 Sliding Windows

To find optimal prefixes of all suffixes, many algorithms, including those by Zantema [15] and those for the MDS problem [3, 7, 10], make use of a *sliding window* approach: as an input sequence s is processed right-to-left, the state information kept during the algorithm is of the form

$$s \quad = \quad u +\!\!+ w +\!\!+ t.$$

Here u is the unexamined portion of the input sequence, w is the "window", and $w +\!\!+ t$ is the current suffix under consideration, from which the optimal p-satisfying prefix is required. Typically, this optimal prefix of $w +\!\!+ t$ is w itself. The window is a function of the processed suffix. For the example above, we have $window\ (w +\!\!+ t) = t$.

The "sliding" characteristic of the window comes from how the window is updated as the algorithm progresses. At each step, a larger suffix of the input s is considered, with the leading edge of the window w moving one step to the left, incorporating the last element of u. The trailing edge of the window (the last element of w) should only move to the left, and never to the right, to ensure an efficient algorithm. Note that if the window data structure can be updated and the optimal prefix extracted in constant (amortized) time then this results in a linear algorithm for the segment problem.

2.2 Zygomorphisms

The technique described above is an example of a *zygomorphism* [11]. To remember which morphism in the zoo of functional algebraic computation patterns is a zygomorphism, it may help to know that "zygo" means "yoked". A zygomorphism is a function which, although not itself a catamorphism[3], can be computed by means of a catamorphism when yoked with a helper function that is a catamorphism. For the sliding window technique, we can use a particular version of a zygomorphism that is a little more specialised than the standard one, and the equation below illustrates the form our zygomorphisms take for lists. Given a helper catamorphism $h :: [A] \to C$, where $h = foldr\ (\otimes)\ c_h$, the function $zygo$ is a zygomorphism if it satisfies the following equations:

$$\begin{array}{lll} zygo & :: & [A] \to B \\ zygo\ [\,] & = & c_z \\ zygo\ (a : x) & = & step\ a\ (zygo\ x, h\ (a : x)). \end{array} \tag{2}$$

Efficient computation of $zygo$ comes from computing $zyh\ x = (zygo\ x, h\ x)$ and then taking $zygo = fst \circ zyh$, since zyh is a

[1] Details of the flawed paper and an argument why it is not linear are given by Chung and Lu [3].

[2] or vice versa, but here we will just stick to prefixes of suffixes.

[3] "Catamorphism" is the categorical name for a *fold* over a data structure.

catamorphism:

$$\begin{array}{l} zyh \qquad\quad :: [A] \to (B, C) \\ zyh\ [\,] \quad\ = (c_z, c_h) \\ zyh\ (a : x) = (step\ a\ (u, v'), v') \\ \quad \textbf{where}\ (u, v)\ =\ zyh\ x \\ \qquad\quad\ v'\ =\ a \otimes v. \end{array}$$

The above equation (2) is a slightly more specialised version of the *syzygy law* from [11], which is the category-theoretic justification for the existence of zygomorphisms.

2.3 Sliding Window Zygomorphisms

Solving optimal segment problems with a sliding window algorithm is possible if $optseg_{f,p}\ (a : x)$ can be expressed this way:

$$\begin{array}{l} optseg_{f,p}\ (a : x)\ =\ optseg_{f,p}\ x\ \ \uparrow_f\ \ optprefix_{f,p}\ w \\ \qquad\qquad\qquad\quad \textbf{where}\ w\ =\ window\ (a : x)\,, \end{array}$$

where $\uparrow_f$ returns the maximum with respect to f, $optprefix_{f,p}$ extracts an optimal prefix of $(a : x)$ from the window w, and the function $window$ can be written as a $foldr$. The window w may be simply a prefix of the input considered so far, or may possibly include some associated data structures.

If this is the case, then (2) holds, the syzygy law is satisfied, and $optseg_{f,p}$ can be expressed as a zygomorphism, which maintains a pair $(optseg_{f,p}\ x, window\ x)$ of values as the algorithm progresses.

3. A Zygomorphism for MDS

Using a sliding window zygomorphism to solve the MDS problem sounds like a good idea, but there are two obstacles.

3.1 Lack of Relational Zygomorphisms

Our specification mds is not a function, but a relation: it is not total (for example when $L > 0$ the empty input sequence $[\,]$ yields no solution), and neither is it a partial function, since many input sequences have more than one segment of maximum density. This is a problem because, as Fokkinga [4] points out, the syzygy law ensures that zygomorphisms exist in the category of sets and (total) functions, but relational zygomorphisms are not guaranteed to exist.

It is tempting to try and refine mds to a function by putting some constraints on the input sequence so that solutions always exist, and using an assymetric operator $\uparrow_d$ which prefers one operand over the other in the event of both segments being equally dense. In that case we would be aiming for an equation for mds to include

$$mds\ (a : x) = \ mds\ x\ \uparrow_d\ mdp\ (trim\ (a : x)),$$

where mdp is a function computing a maximally dense prefix, and $trim$ returns the longest prefix of its input with breadth at most U:

$$trim\ x = \ max_{\#} \circ \Lambda((U \geq_b)? \circ prefix),$$

where $\#$ is a shorthand for $length$, and $\geq_b$, and $\leq_b$, $<_b$, $>_b$, denote comparison of breadths, overloaded to elements, lists, and scalars. If $mdp \circ trim$ is a $foldr$, then mds can be computed using a zygomorphism.

However, putting constraints on the input sequence to attempt to ensure the totality of mds gets messy. It is not as simple as requiring that breadths of elements are no wider than U: for example consider an input sequence $[e_1, e_2, e_3]$ where the total breadth of $[e_1, e_2, e_3]$ exceeds U, but the total breadth of $[e_1, e_2]$ is less than L and $e_3 <_b U$. This input sequence does not have a solution, and indeed input sequences containing segments such as $[e_1, e_2, e_3]$ pose problems for previously published MDS algorithms, either by failing to return a correct answer or going into an infinite loop.

A cleaner approach is to explicitly acknowledge the relational properties of the specification, and aim for a functional version of mds that returns a value of type $Maybe\ [Elem]$. We define:

$$\begin{array}{l} mds_{\mathsf{M}}\ ::\ [Elem] \to Maybe\ [Elem] \\ mds_{\mathsf{M}}\ [\,] \qquad = Nothing \\ mds_{\mathsf{M}}\ (a : x) = mds_{\mathsf{M}}\ x\ \Uparrow_d\ \ wp\ (trim\ (a : x)), \end{array}$$

where wp is a function that returns a suitable prefix (see below), and the operator $(\Uparrow_d)$ lifts $(\uparrow_d)$ to $Maybe$, if the second argument is within bounds:

$$\begin{array}{lll} (\Uparrow_d) & :: & Maybe\ [Elem] \to [Elem] \to Maybe\ [Elem] \\ r & \Uparrow_d\ p & \mid \neg(bounds\ p) = r \\ Nothing & \Uparrow_d\ p & \qquad\qquad\quad = Just\ p \\ (Just\ m) & \Uparrow_d\ p & \qquad\qquad\quad = Just\ (m \uparrow_d p). \end{array}$$

To ensure that mds_{M} returns a value corresponding to mds, we need wp to be able to return a maximally dense prefix of its input. Of the possible prefixes to return, we define a partial function $smsp$ (for "shortest maximally-dense suitable prefix")

$$\begin{array}{ll} smsp & :: [Elem] \to [Elem] \\ smsp & = min_{\#} \circ \Lambda max_d \circ \Lambda((L \leq_b)? \circ prefix). \end{array}$$

Here $min_{\#} \circ \Lambda max_d$ expresses the selection of a shortest list out of all those of maximum density, so $smsp$ returns the shortest prefix that is at least L units wide and has maximum density. Note that $smsp$ does not depend on the upper bound U.

We now define the total function wp, which returns the $smsp$ of the input if its breadth is at least L, and otherwise returns the input unchanged:

$$\begin{array}{lll} wp & :: & [Elem] \to [Elem] \\ wp\ x & \mid & L \leq_b x \qquad = smsp\ x \\ & \mid & \textbf{otherwise} = x. \end{array}$$

It can now be seen that x is in the domain of mds iff $mds_{\mathsf{M}}\ x = Just\ \_$ for some value. From now on we refer to this value as $mds\ x$ for brevity, and when we write $smsp\ x$ we imply that $x \geq_b L$.

3.2 Catamorphism Required for Yoking

The function $smsp$ has some nice properties, for example, it is monotonic on the prefix ordering $\sqsubseteq$ ($x \sqsubseteq y$ if x is a prefix of y):

$$x \sqsubseteq y \ \ \Rightarrow \ \ smsp\ x \leq_d smsp\ y. \tag{3}$$

Unfortunately, $smsp$ is not a $foldr$! Indeed, $smsp\ (a : x)$ might well be longer than $smsp\ x$ — consider the list $x = [(9, 1), (8, 1)]$ and $a = (1, 1)$, where $smsp\ x = [(9, 1)]$, but $smsp\ (a : x) = [(1, 1), (9, 1), (8, 1)]$. Consequently, $wp \circ trim$ is not a $foldr$ either. This is not good news, since we were hoping for a catamorphism to compute mds as a zygomorphism.

All is not lost, however, and we will make use of the following lemma, which states that if the right-hand end of $smsp\ (u +\!\!+ x)$ extends further than $smsp\ x$ (that is, $u +\!\!+ smsp\ x$ is a proper prefix of $smsp\ (u +\!\!+ x)$), the latter is denser than the former:

Lemma 3.1. $u +\!\!+ smsp\ x \sqsubset smsp\ (u +\!\!+ x) \ \ \Rightarrow \ \ smsp\ x >_d smsp\ (u +\!\!+ x)$.

Proof. Denote $smsp\ x$ by x_1. The antecedent implies that there exists some non-empty x_2 such that $smsp\ (u +\!\!+ x) = u +\!\!+ x_1 +\!\!+ x_2$. Since x_1 is a maximum density prefix of x, we have $x_1 \geq_d x_1 +\!\!+ x_2$, which is equivalent to $x_1 \geq_d x_2$ by the density property (1). Since $u +\!\!+ x_1 +\!\!+ x_2$ is the shortest maximum-density prefix of $u +\!\!+ x$, we have $u +\!\!+ x_1 +\!\!+ x_2 >_d u +\!\!+ x_1$, which is equivalent to $x_2 >_d u +\!\!+ x_1 +\!\!+ x_2$ by (1). By transitivity we have $x_1 >_d u +\!\!+ x_1 +\!\!+ x_2$. □

As a special case of Lemma 3.1, if $smsp\ (a : x)$ extends further than $smsp\ x$, then its density will be lower than $smsp\ x$. This will

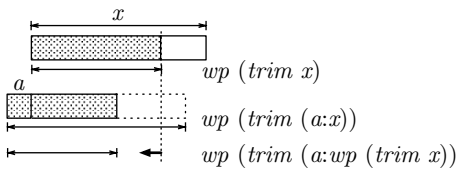

Figure 2. The dashed vertical line marks the trailing edge of the window.

lead to a suggestion for an alternative catamorphism to be yoked to the computation of mds_{M}.

Informally, consider Figure 2: the upper portion of the diagram depicts a sequence x in the case that $trim\ x$ is at least L, and the shaded section is $wp\ (trim\ x)$, the shortest maximally-dense suitable-length prefix of x. The lower portion of the diagram depicts what happens if $wp\ (trim\ (a : x))$, indicated by the dotted outline, extends further than $wp\ (trim\ x)$. By Lemma 3.1, the density of $wp\ (trim\ (a : x))$ is lower than $smsp\ x$, so it looks like we will lose nothing if, instead of computing $wp\ (trim\ (a : x))$, we consider only the prefixes starting from the dotted line, of which the best will be $wp\ (trim\ (a : wp\ (trim\ x)))$.

Thus our window for the zygomorphism is going to be built by repeatedly applying $wp \circ trim$ after adding each element to the leading (left-hand) edge of the window:

$$\begin{aligned} win\ [\,] &= [\,] \\ win\ (a : x) &= wp\ (trim\ (a : win\ x)), \end{aligned}$$

This gives us the catamorphism $win = foldr\ (wp \circ trim \circ (:))\ [\,]$ we need, resulting in the zygomorphism $mds_{\mathsf{M}} = fst \circ zh$ to solve the MDS problem, where

$$\begin{aligned} zh\ \ \ &:: [Elem] \to (Maybe\ [Elem], [Elem]) \\ zh\ [\,] &= ([\,], [\,]) \\ zh\ (a : x) &= (ms \Uparrow_d w', w') \\ &\quad \mathbf{where}\ (ms, w) = zh\ x \\ &\qquad\qquad w' = wp\ (trim\ (a : w)). \end{aligned}$$

Formally, we need to prove:

Theorem 3.2. $mds_{\mathsf{M}}\ (a : x) = mds_{\mathsf{M}}\ x \Uparrow_d win\ (a : x)$.

Proof. Firstly, the case when $a : x <_b L$ is easily dealt with.

When $a : x \geq_b L$, we first consider the case when x is in the domain of mds and $smsp\ (trim\ (a : x)) <_d mds\ x$. We reason:

$$\begin{aligned} & mds_{\mathsf{M}}\ x \Uparrow_d win\ (a : x) \\ = & \quad \{\ \text{since } a : x \geq_b L\ \} \\ & mds_{\mathsf{M}}\ x \Uparrow_d smsp\ (trim\ (a : x)) \\ = & \quad \{\ \text{since } mds\ x >_d smsp\ (trim\ (a : x))\ \text{and} \\ & \qquad smsp\ (trim\ (a : x)) \geq_d win\ (a : x) \text{ by (3) } \} \\ & mds_{\mathsf{M}}\ x \Uparrow_d win\ (a : x). \end{aligned}$$

That leaves two other possibilities: $mds_{\mathsf{M}}\ x = Nothing$ or $smsp\ (trim\ (a : x)) \geq_d mds\ x$. For that case we claim that

$$smsp\ (trim\ (a : x)) = smsp\ (trim\ (a : win\ x))$$

holds, by the following lemma. □

Lemma 3.3. For non-empty u we have:

$$\begin{aligned} & mds_{\mathsf{M}}\ x = Nothing\ \ \vee\ \ smsp\ (trim\ (u +\!\!+ x)) \geq_d mds\ x \\ & \Rightarrow \\ & smsp\ (trim\ (u +\!\!+ x)) = smsp\ (trim\ (u +\!\!+ win\ x)). \end{aligned}$$

Proof. The proof is by induction on x, using Lemma 3.1. □

The assertion made by the whole of Lemma 3.3 is the key invariant of $win\ x$ we maintain to guarantee the correctness of the zygomorphic algorithm above. While we cannot afford to maintain a window that is a maximally dense prefix of the input considered so far, this window does so when it matters: if a maximally dense prefix of $u +\!\!+ x$ is at least as dense as $mds\ x$, this optimal prefix always lies within $u +\!\!+ win\ x$.

Now that mds_{M} is a zygomorphism and the window is constructed by repeatedly calling $wp \circ trim$ in a $foldr$, the next task is to refine $wp \circ trim$ so that it can be computed in constant (amortised) time — no easy task! In the next section we pause in the algorithm development, to explore some elegant properties of sequences and densities; these will be exploited later for the purpose of representing and efficiently maintaining the window.

4. Densities and Partitions

A non-empty list x is called *right-skew* [10] if, for all non-empty lists x_1 and x_2 such that $x_1 +\!\!+ x_2 = x$, we have $x_1 \leq_d x_2$. Thus we define

$$\begin{aligned} & rightskew\ \ :: [Elem] \to Boolean \\ & rightskew\ x = all\ (\leq_d)\ [(take\ n\ x, drop\ n\ x)\ \mid \\ & \qquad\qquad n\ \leftarrow\ [1..\#x - 1]], \end{aligned}$$

where $all\ p$ is defined by $foldr\ (\wedge)\ True \circ map\ p$. An example of a right-skew segment is given in Fig. 3.

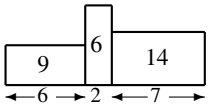

Figure 3. An example of a right-skew segment.

4.1 Decreasing Right-Skew Partition

Any list $x :: [Elem]$ can be partitioned into segments $x = x_1 +\!\!+ x_2 +\!\!+ \ldots +\!\!+ x_n$, such that all the segments are right-skew. (To see this, simply partition x into singleton segments.) What is perhaps surprising is that every list can be *uniquely* partitioned into right-skew segments of strictly decreasing density [10]. That is, for any list $x :: [Elem]$ there is exactly one $xs :: [[Elem]]$ such that $concat\ xs = x$, and $sdars\ xs$, where

$$\begin{aligned} & sdars\ \ \ :: [[Elem]] \to Boolean \\ & sdars\ xs\ = sdec\ (map\ d\ xs)\ \wedge\ all\ rightskew\ xs. \end{aligned}$$

Here, $sdars$ stands for "strictly decreasing, all right-skew", and $sdec$ checks whether a list of numbers is strictly decreasing. We will refer to this partition of a list as its *decreasing right-skew partition* (DRSP). Fig. 4 shows an example of such a partition.

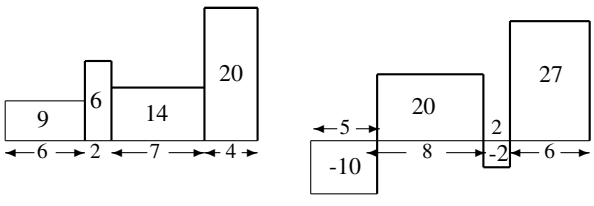

Figure 4. The DRSP of the list illustrated in Fig. 1.

DRSPs have a number of elegant properties which are not too difficult to prove using only the density property (1). Firstly, note that the $sdars$ property is prefix- and suffix-closed: if $xs :: [[Elem]]$ satisfies $sdars$, then so too does any prefix or suffix of xs. This

follows from the fact that taking prefixes or suffixes does not disrupt any right-skew properties nor the decreasing of the densities of the partition components.

In addition, the DRSP of a list has a kind of "rotational symmetry", where given a function $neg :: Elem \to Elem$ that returns an element of the same breadth but negated area, we have

$$\begin{aligned} & map\ (map\ neg) \circ reverse \circ map\ reverse \circ drsp \\ = \quad & drsp \circ reverse \circ map\ neg, \end{aligned} \tag{4}$$

This property can be precisely illustrated by holding Fig. 4 upside-down, and note that the result is still a DRSP.

4.2 Constructing a DRSP

The DRSP of a list can be constructed in several different ways. One method is to repeatedly take the longest right-skew prefix ($lrsp$) of the input list:

$$\begin{aligned} & drsp :: [Elem] \to [[Elem]] \\ & drsp\ x = leftBuild\ lrsp, \\ \\ & lrsp :: [Elem] \to [Elem] \\ & lrsp = max_{\#} \circ \Lambda(rightskew? \circ prefix), \\ \\ & leftBuild :: ([Elem] \to [Elem]) \to [Elem] \to [[Elem]] \\ & leftBuild\ m\ [\,] = [\,] \\ & leftBuild\ m\ x = pre\ +\!\!+\ leftBuild\ m\ (drop\ (\#pre)\ x) \\ & \quad \mathbf{where}\ pre\ =\ m\ x. \end{aligned}$$

Alternatively, since the longest right-skew prefix is also the longest highest-density prefix ($lhdp$),

$$\begin{aligned} lhdp \quad & :: \quad [Elem] \to [Elem] \\ lhdp \quad & = \quad max_{\#} \circ \Lambda max_d \circ \Lambda prefix \end{aligned}$$

a DRSP can be constructed with $drsp = leftBuild\ lhdp$. Dually, via the rotational symmetry property (4), a DRSP can be constructed by repeatedly taking longest lowest-density suffixes, or longest right-skew suffixes.

Alternatively, a DRSP can also be built using a fold

$$\begin{aligned} & drsp :: [Elem] \to [[Elem]] \\ & drsp = foldr\ addl\ [\,], \\ \\ & addl :: Elem \to [[Elem]] \to [[Elem]] \\ & addl\ a\ xs = prepend\ [a]\ xs, \\ \\ & prepend :: [Elem] \to [[Elem]] \to [[Elem]] \\ & prepend\ x\ [\,] = [x] \\ & prepend\ x\ (y : xs) \mid x >_d y = x : y : xs \\ & \qquad\qquad\qquad\quad \mid otherwise = prepend\ (x +\!\!+ y)\ xs. \end{aligned}$$

This fold works by repeatedly adding singleton lists at the front of the partition and merging adjacent segments at the front of the partition where their densities do not strictly decrease. It is not difficult to show that if $sdars\ xs$ holds, so does $sdars\ (prepend\ [a]\ xs)$ for all a. To illustrate, Figure 5 demonstrates building the DRSP of a list $[1, 4, 2, 5, 3]$ (with all elements having breadth 1). The top-left of the diagram indicates that $drsp\ [3] = [[3]]$. Then, doing $addl\ 5$ results in $drsp\ [5, 3] = [[5], [3]]$, since as $[5] >_d [3]$, no merging happens when 5 is added. When 2 is added, is it merged with 5 to form a new sequence with density 3.5. No merging is triggered with the addition of 4 since the densities $[4, 3.5, 3]$ form a decreasing sequence. Adding 1, however, triggers three mergings and all elements are grouped into one segment with density 3.

By doing $prepend$ and $drsp$ backwards, we discover how to remove the leftmost element from a DRSP:

$$\begin{aligned} removel \quad & :: \quad [[Elem]] \to (Elem, [[Elem]]) \\ removel\ ((a : x) : xs) \quad & = \quad (a, drsp\ x +\!\!+ xs). \end{aligned}$$

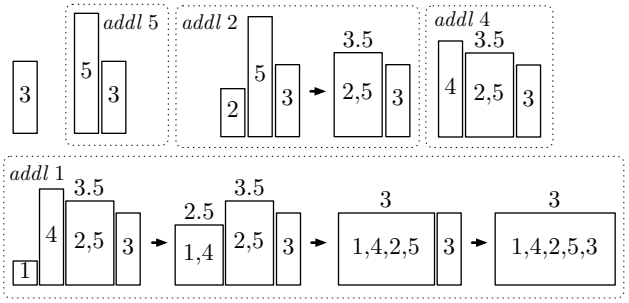

Figure 5. Building DRSP from $[1, 4, 2, 5, 3]$.

Seeing $removel$ as the inverse operation of $addl$, an operational way to understand it is that the calls to $+\!\!+$ in $prepend$, as depicted in Figure 5, forgets the partitions, thus in the second clause of $removel$ we have to make a call to $drsp$ to re-partition x.

All the properties above are symmetrical. We could also have built a DRSP by adding elements from the right:

$$\begin{aligned} & postpend :: [[Elem]] \to [Elem] \to [[Elem]] \\ & postpend\ [\,]\ x \qquad\quad = [x] \\ & postpend\ (xs +\!\!+ [y])\ x \mid y >_d x \quad = xs +\!\!+ [y, x] \\ & \qquad\qquad\qquad\quad \mid otherwise = postpend\ xs\ (y +\!\!+ x). \end{aligned}$$

Therefore, $drsp$ can be defined as a $foldl$ and, by symmetric reasoning, this is how to remove the rightmost element of a DRSP:

$$\begin{aligned} & drsp\ =\ foldl\ addr\ [\,], \\ & addr\ xs\ a\ =\ postpend\ xs\ [a], \\ \\ & remover\ ::\ [[Elem]] \to ([[Elem]], Elem) \\ & remover\ (xs +\!\!+ (x +\!\!+ [a]))\ =\ (xs +\!\!+ drsp\ x, a). \end{aligned}$$

5. The Window and the DRSP

The properties of right-skew segments and the DRSP will be crucial for representing the window for our mds_{M} algorithm. Recall that partway through the computation, the window is a prefix of the input processed so far. As the segments required in the mds specification must be at least L in total breadth, the prefixes obtained from the window must satisfy the same constraint. Thus when the breadth of the window is at least L, we can think of the window w as divided into two parts,

$$w\ =\ c +\!\!+ x$$

where c is the shortest prefix of w whose breadth is at least L, and x contains the remaining elements. The prefix c can be thought of as the "compulsory" part of the window, as any prefix produced from the window must include c.

In our refinement of the window, we will be representing the right-hand side of the window by the DRSP of x. Figure 6 shows the general idea, where $drsp\ x = [x_1, x_2 \ldots x_n]$.

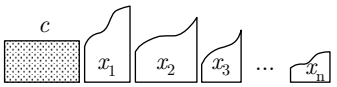

Figure 6. The general plan for the window contents.

When the breadth of the window is less than L, the window will be as a simple list of elements, similar to a prefix of c.

The rest of this section shows why the DRSP holds the information needed by the window.

5.1 Prefixes and Right-Skew Segments

Right-skew segments possess a very useful property: informally, consider a list $z +\!\!+ x$, for an arbitrary list of elements z and a right-skew list x, and also consider all the lists of the form $z +\!\!+ y$, where y is a prefix of x. It turns out that either z or $z +\!\!+ x$ (or both) is maximally dense amongst such lists $z +\!\!+ y$. This means that if we are looking for a prefix of $z +\!\!+ x$ that starts with z and is of maximum density, then the only prefixes that need to be considered are z and $z +\!\!+ x$ — there is no point considering something in the middle.

To be formal, we define *smwr* (shortest maximally-dense with respect to):

$$\begin{array}{l} smwr \;\; :: \; [Elem] \to \mathbb{P}[Elem] \to Elem \\ smwr\ z = min_{\#} \circ \Lambda max_f \\ \qquad \textbf{where}\ f \;\; = d \circ (z +\!\!+) . \end{array}$$

That is, *smwr* z picks, among the given set, the shortest list that has maximum density after being concatenated with z. (We will later express *smsp* in terms of *smwr*.) The above property is a consequence of the following lemma:

Lemma 5.1. $z +\!\!+ smwr\ z\ (\Lambda prefix\ x) = z \uparrow_d (z +\!\!+ x)$ if x is right-skew.

Proof. This is proved by using a generalisation: given a list y, if $x +\!\!+ y_1 \leq_d y_2$ for all y_1, y_2 such that $y_1 +\!\!+ y_2 = y$ and y_2 is non-empty, then we have

$$z +\!\!+ smwr\ z\ (\mathsf{E}(x +\!\!+)\ (\Lambda prefix\ y)) = (z +\!\!+ x) \uparrow_d (z +\!\!+ x +\!\!+ y).$$

This can be proved using a simple structural induction on y, which makes use of the following lemma, which in turn is a consequence of the density property (1). □

Lemma 5.2. Let z, x_1, x_2 be non-empty lists of items such that $x_1 \leq_d x_2$, then at least one of $z +\!\!+ x_1 \leq_d z$ or $z +\!\!+ x_1 <_d z +\!\!+ x_1 +\!\!+ x_2$ is true.

So how does this help with extracting segments of maximum density from the sliding window? Consider a list of the form $z +\!\!+ x$, where z is an arbitrary list of elements and x is an arbitrary non-empty list of elements that can be partitioned into right-skew segments $x = x_1 +\!\!+ x_2 \ldots +\!\!+ x_n$. By applying Lemma 5.1 repeatedly, we can see that finding a maximally dense prefix of $z +\!\!+ x$, of the form $z +\!\!+ y$, only involves considering the prefixes z, $z +\!\!+ x_1$, $z +\!\!+ x_1 +\!\!+ x_2$, and so on, up to $z +\!\!+ x$. More formally, we define $opts = \Lambda(concat \circ prefix)$, that is,

$$opts\ [x_1, x_2, .., x_n] = \{[\,], x_1, x_1 +\!\!+ x_2, ..x_1 +\!\!+ x_2 +\!\!+ .. +\!\!+ x_n\},$$

and then the above mentioned property can be formalised as the following theorem:

Theorem 5.3. $smwr\ z\ (\Lambda prefix\ (concat\ xs)) = smwr\ z\ (opts\ xs)$, if *all rightskew xs* holds.

This justifies the representation of the window: there is an initial "compulsory" prefix c of the window, which must always be included in any candidate for the maximally dense prefix of the window, in order to ensure the total breadth of a prefix is at least L. The rest of the already-trimmed window (the "optional" part) is represented as a DRSP. This means that only the prefixes marked out by the partition component boundaries need be considered as candidates for a maximally dense prefix of suitable length.

5.2 Finding a Maximally Dense Prefix

That a DRSP has decreasing density can be further exploited. Given a sequence of segments $[x_1, x_2 \ldots]$ with decreasing densities, the densities of $[c, c +\!\!+ x_1, c +\!\!+ x_1 +\!\!+ x_2 \ldots]$ must be *bitonic* – ascending and then descending, as depicted in Figure 7. The top part of this diagram depicts the window, where each wavy block denotes a right-skew segment. The blocks in the bottom row represent the densities of c, $c +\!\!+ x_1$, $c +\!\!+ x_1 +\!\!+ x_2$, etc.

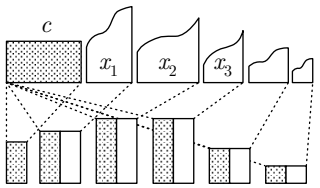

Figure 7. Bitonic property of DRSP.

We can use this bitonic property to define a function $maxchop$, which will identify a densest segment of $[c, c +\!\!+ x_1, c +\!\!+ x_1 +\!\!+ x_2 \ldots]$, choosing the leftmost segment if there is more than one densest segment. The way it does this is that $maxchop\ c$, given a partition, keeps chopping away the rightmost segment of the partition, until more chopping would not improve the density of the remaining elements in c and the partition:

$$\begin{array}{l} maxchop :: [Elem] \to [[Elem]] \to [[Elem]] \\ maxchop\ c\ [\,] = [\,] \\ maxchop\ c\ (xs +\!\!+ [x]) \\ \quad \mid c +\!\!+ (concat\ (xs +\!\!+ [x])) \geq_d x = maxchop\ c\ xs \\ \quad \mid \textbf{otherwise} \qquad\qquad\quad = [\,]. \end{array}$$

To see how $maxchop\ c$ works, it may help to remember that by the density property (1), $c +\!\!+ (concat\ (xs +\!\!+ [x])) \geq_d x$ is equivalent to $c +\!\!+ (concat\ xs) \geq_d c +\!\!+ (concat\ (xs +\!\!+ [x]))$. The bitonic property for a DRSP may then be formally stated by:

Theorem 5.4. $smwr\ c\ (opts\ xs) = concat\ (maxchop\ c\ xs)$ if $sdars\ xs$ holds.

Furthermore, since $maxchop\ c\ xs$ returns a prefix of xs, then as xs is a DRSP, so too is $maxchop\ c\ xs$.

5.3 A Sliding Window Algorithm

Having seen how to represent the window using a DRSP, perhaps at this point it would be a good idea to take stock of how far the algorithm development has got; at this point we do have sufficient information for a (somewhat inefficient) implementation in the case when $U = \infty$.

In Section 3 it was established that

$$\begin{array}{ll} mds_{\mathsf{M}} & :: [Elem] \to Maybe\ [Elem] \\ mds_{\mathsf{M}}\ [\,] & = Nothing \\ mds_{\mathsf{M}}\ (a : x) & = mds_{\mathsf{M}}\ x \Uparrow_d win\ (a : x), \end{array}$$

where the window was constructed by

$$\begin{array}{ll} win & :: [Elem] \to [Elem] \\ win\ [\,] & = [\,] \\ win\ (a : x) & = wp\ (trim\ (a : win\ x)), \end{array}$$

where wp calls *smsp* when the window is at least L units wide, and otherwise leaves the window untouched.

At this point we need to represent the window using a different datatype. Changing datatype representation is a typical case of the worker-wrapper transformation [5]. Let the desired window representation be some type W. We write a function $rep :: [Elem] \to W$ that transforms the list of elements in the window to our representation type W, and we will also write a reverse transformation $abs :: W \to [Elem]$. If $abs \circ rep = id$, we may compute mds_{M} by $fmap\ abs \circ mds'_{\mathsf{M}}$, where $fmap :: (a \to b) \to Maybe\ a \to$

$Maybe\ b$ is the functorial mapping lifting a functon to $Maybe$ type. The function mds'_{M} is defined by:[4]

$$\begin{array}{l} mds'_{\mathsf{M}}\ ::\ [Elem] \to Maybe\ W \\ mds'_{\mathsf{M}}\ [\,] \quad\ = Nothing \\ mds'_{\mathsf{M}}\ (a : x) = mds'_{\mathsf{M}}\ x\ \Uparrow'_d\ win'\ (a : x), \end{array}$$

and $(\Uparrow'_d)$ is simply $\Uparrow_d$ lifted to W:

$$\begin{array}{l} (\Uparrow'_d) :: Maybe\ W \to W \to Maybe\ W \\ r\ \Uparrow'_d\ w = fmap\ rep\ (fmap\ abs\ r\ \Uparrow_d\ abs\ w). \end{array}$$

The function win' is defined below:

$$\begin{array}{l} win' \quad\ :: [Elem] \to W \\ win'\ [\,] \quad = rep\ [\,] \\ win'\ (a : x) = rep\ (wp\ (trim\ (a : abs\ (win'\ x)))). \end{array}$$

Note that for the case where $U = \infty$, no trimming to fit the upper bound U is needed, and thus $trim\ =\ id$.

For our purpose here, let $W\ =\ ([Elem], [[Elem]])$. That is, a window will be represented as a pair (c, xs), where c is the "compulsory part" of the window, and xs is the DRSP of the rest of the elements. If the breadth of the window is less than L, then these elements will be contained in c, with $xs\ =\ [\,]$. To construct rep and abs, we define a function $split :: [Elem] \to ([Elem], [Elem])$ which splits a list into its compulsory and optional parts:

$$\begin{array}{l} split\ x = split'\ ([\,], x) \\ split'\ (z, [\,]) \qquad\qquad\qquad\quad = (z, [\,]) \\ split'\ (z, a : x) \mid (z +\!\!+ [a]) \leq_b L = split'\ (z +\!\!+ [a], x) \\ \qquad\qquad\quad \mid \mathbf{otherwise} \quad\ = (z, a : x), \end{array}$$

and note that it is defined on all lists of elements — when the input is not wide enough, the optional part is simply empty. Now we can define rep and abs:

$$\begin{array}{l} rep \qquad\ = (id \times drsp) \circ split \\ abs\ (z, xs) = z +\!\!+ concat\ xs, \end{array}$$

where $(f \times g)\ (a, b) = (f\ a, g\ b)$. Having defined $split$, note that

Lemma 5.5. If $breadth\ x\ \geq\ L$ and $(z, x')\ =\ split\ x$, then $smsp\ x\ =\ z +\!\!+ smwr\ z\ (\Lambda prefix\ x')$.

Proof. This is an easy calculation from the definitions of $smsp$ (page 3) and $smwr$ (page 6). □

We will use this lemma during the next task, which is to simplify the definition of win'. Thus we calculate starting from the expression $rep \circ wp \circ (a :) \circ abs$, using typical program calculation to push rep to the right. Most of the calculation is routine, and we highlight just the two interesting parts. The theorems established in the previous section are used when rep encounters $smsp$:

$$\begin{array}{ll} & ((id \times drsp) \circ split \circ smsp)\ x \\ = & \quad \{\ \text{let } (z, x') = split\ x, \text{ by Lemma 5.5}\ \} \\ & (z, drsp\ (smwr\ z\ (\Lambda prefix\ x'))) \\ = & \quad \{\ concat \circ drsp = id\ \} \\ & (z, drsp\ (smwr\ z\ (\Lambda prefix\ (concat\ (drsp\ x'))))) \\ = & \quad \{\ \text{Theorem 5.3}\ \} \\ & (z, drsp\ (smwr\ z\ (opts\ (drsp\ x')))) \\ = & \quad \{\ \text{Theorem 5.4}\ \} \\ & (z, drsp\ (concat\ (maxchop\ z\ (drsp\ x')))) \\ = & \quad \{\ \text{a prefix of a DRSP is a DRSP}\ \} \\ & (z, maxchop\ z\ (drsp\ x')) \\ = & ((\lambda(z, x') \to (z, maxchop\ z\ x')) \circ rep)\ x. \end{array}$$

One can see that, now that the optional part is partitioned into a DRSP, to compute the $smsp$ we only need to chop away the less dense segments using $maxchop$.

Some more work needs to be done when rep encounters $(a :)$. To push rep rightwards, we define $split\ (a : x)$ in terms of $split\ x$:

$$\begin{array}{l} split\ (a : x) = \ \mathbf{let}\ (z, w) = split\ x \\ \qquad\qquad\qquad\quad\ (z_1, z_2) = split\ (a : z) \\ \qquad\qquad\qquad\ \mathbf{in}\ (z_1, z_2 +\!\!+ w), \end{array} \tag{5}$$

that is, x is split into (z, w), and we then perform $split$ on $(a : z)$ to find out what suffix of z, if any, is to be added to w. We reason:

$$\begin{array}{ll} & ((id \times drsp) \circ split)\ (a : x) \\ = & \quad \{\ (5)\ \} \\ & (z_1, drsp\ (z_2 +\!\!+ w)) \\ = & \quad \{\ \text{since } drsp \text{ is a } foldr\ \} \\ & (z_1, foldr\ addl\ (drsp\ w)\ z_2). \end{array}$$

In summary, we have derived

$$\begin{array}{l} win'\ [\,] \quad\ = ([\,], [\,]) \\ win'\ (a : x) = \ \mathbf{let}\ (z, xs) = win'\ x \\ \qquad\qquad\qquad\quad\ (z_1, z_2) = split\ (a : z) \\ \qquad\qquad\qquad\ \mathbf{in}\ (z_1, maxchop\ z_1\ (foldr\ addl\ xs\ z_2)), \end{array}$$

which completes the first version of our sliding window algorithm. The window used is of the form $(c, drsp\ x)$.

At present, this is not a very efficient algorithm. It also does not easily extend to the case where the upper bound U is not infinite. In that case, $trim$ is no longer id, and the window needs to be trimmed from the right. The difficulty in that situation is maintaining the DRSP after removing one or more elements from the right-hand side. To illustrate this difficulty, imagine applying the $remover$ function (defined at the end of Section 4) to $drsp\ (xs +\!\!+ [x +\!\!+ [a]])$. This involves creating the of DRSP of x, and overall this requires too much work to be done and yet keep the algorithm linear-time.

Throughout the rest of the paper, we assume the use of a datatype for lists supporting $cons$ and $snoc$ operations, and also area, breadth and density calculations, in (amortized) $O(1)$ time, for example a double-ended queue as in Okasaki [13], annotated with area and breadth information. However we will continue to use the standard notation for lists, for clarity.

In addition, we need a way of manipulating and storing the DRSP such that information about right-skew segments within the partition is not lost once calculated. We will aim to represent a DRSP by a kind of queue that supports $O(1)$ decomposition from both ends, while maintaining the DRSP structure. To do this, we will exploit the hierarchical nature of DRSP structures, by explicitly representing them using trees. The efficiency of the resulting algorithm is then discussed later on in Section 7.2.

6. Tree Representations of the DRSP

Recall that as the window slides from right to left along the input sequence, new elements get added (with $addl$) to the left-hand side of the DRSP section of the window, and when we $trim$ with respect to a non-infinite upper bound U, elements are removed from the right-hand side. The current $remover$ is inefficient in removing elements from the right, because it includes a call to $drsp$ to reform the DRSP at the right-hand side of the window.

What would help is to remember more of the DRSP structure that is observed when adding elements. We will use some hierar-

[4] The transformation is a specical case of Gill and Hutton's formulation [5] by letting $unwrap = (fmap\ rep \circ)$ and $wrap = (fmap\ abs \circ)$.

chical properties of the DRSP to represent it as a tree, which will remember the information required.

At first glance, a DRSP does not seem to have much in the way of hierarchical structure. Consider a list of elements x, decomposed into $drsp\ x\ =\ [x_1, x_2, \ldots x_n]$. This partition cannot directly be decomposed further, because as each segment is already right-skew, the DRSP of x_1 is just $[x_1]$, for example. Instead, we will use the idea of decomposing $tail\ x$ into its DRSP, and use a tree to represent the decomposition structure.

6.1 The DTree Representation

We define a datatype $DTree$ (the letter D comes from the first letter of DRSP), the definition of which is a rose tree with labelled nodes:

$$\begin{aligned}&\mathbf{type}\ DForest = [DTree],\\ &\mathbf{data}\ DTree\ \ = \mathsf{D}\ Elem\ DForest,\end{aligned}$$

and where the tree structure is augmented with area and breadth information (not shown) for efficiently computing densities.

The idea is that if t is a $DTree$ representing the list $(a : x)$, the root of the tree is a, and a pre-order traversal of t returns the list $(a : x)$. Furthermore, the children of the root node are $DTree$s representing the segments of the partition $drsp\ x$. For example, take the list of elements $[1, 4, 2, 5, 3]$ (with all elements having breadth 1), and note that the DRSP of $[4, 2, 5, 3]$ is $[[4], [2, 5], [3]]$. The $DTree$ representing the whole list is illustrated in Figure 8.

Figure 8. The $DTree$ $\mathsf{D}\ 1\ [\mathsf{D}\ 4\ [\,], \mathsf{D}\ 2\ [\mathsf{D}\ 5\ [\,]], \mathsf{D}\ 3\ [\,]]$

Formally, the structure of a $DTree$ can be described by the following (inefficient) function constructing a $DTree$:

$$\begin{aligned}&makeDTree\ \ :: [Elem] \to DTree\\ &makeDTree\ (a : x) = \mathsf{D}\ a\ (map\ makeDTree\ (drsp\ x)).\end{aligned}$$

A datatype invariant satisfied for any tree t of type $DTree$ is as follows:

$$makeDTree\ (pretrav\ t) = t.$$

where $pretrav$ returns a pre-order traversal.

Note that the DRSP of a list x is represented by the $DForest$ returned by $map\ makeDTree\ (drsp\ x)$. For the MDS algorithm, we will need to be able to add elements to the left-hand end of a DRSP forest. In an analogous way to the functions $drsp$, $addl$, and $prepend$ from Section 4, we define functions $drspD$, $addD$, and $prependD$:

$$\begin{aligned}&drspD :: [Elem] \to DForest\\ &drspD = foldr\ addD\ [\,],\\ \\ &addD :: Elem \to DForest \to DForest\\ &addD\ a\ ts = prepend\ (\mathsf{D}\ a\ [\,])\ ts.\\ \\ &prependD :: DTree \to DForest \to DForest\\ &prependD\ t\ [\,] = [t]\\ &prependD\ (t@(\mathsf{D}\ r\ ts))\ (u : us)\\ &\quad \mid t >_d u = t : u : us\\ &\quad \mid otherwise = prependD\ (\mathsf{D}\ r\ (ts +\!\!+ [u]))\ us.\end{aligned}$$

Compared to $prepend$ which uses the list representation of a DRSP and merges segments by concatenation, $prependD$ merges two adjacent segments $[t, u]$ by making u the rightmost subtree of t when $t \leq_d u$. It is a bit counterintuitive that a lighter tree t absorbs a denser entity u, but one way to think of it is that the heavier entity u "sinks" under the lighter t. Figure 9 illustrates building a tree representation of the DRSP of $[1, 4, 2, 5, 3]$, by repeatedly applying $addD$.

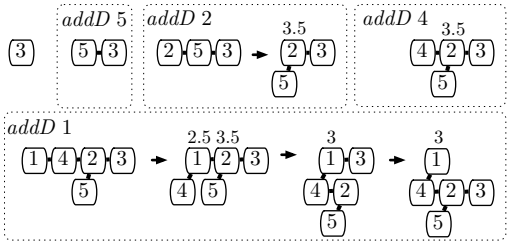

Figure 9. Representing a partition by a $DForest$.

The function $removeD$ removes the leftmost element from a $DForest$, and is analogous to the $removel$ function in Section 4. It is carried out by performing the inverse of $addl$:

$$\begin{aligned}&removeD\ \ :: \ DForest \to (Elem, DForest)\\ &removeD\ ((\mathsf{D}\ a\ ts) : us)\ \ =\ \ (a, ts +\!\!+ us).\end{aligned}$$

6.2 The PTree Representation

Dually, one can also build a DRSP by adding elements from the right. For that we define a dual datatype $PTree$, the letter P coming from the last letter of DRSP:

$$\begin{aligned}&\mathbf{type}\ PForest = [PTree],\\ &\mathbf{data}\ PTree\ \ = \mathsf{P}\ Elem\ PForest.\end{aligned}$$

A tree p of type $PTree$ satisfies a different set of constraints: if p represents the segment $x +\!\!+ [a]$, then the root of p is a, the post-order traversal of p is $x +\!\!+ [a]$, and the children of the root node represent the segments in $drsp\ x$. Thus a $PTree$ can be constructed from

$$\begin{aligned}&makePTree\ \ :: [Elem] \to PTree\\ &makePTree\ (x +\!\!+ [a]) = \mathsf{P}\ a\ (map\ makePTree\ (drsp\ x)),\end{aligned}$$

and the datatype invariant that a $PTree$ should satisfy is

$$makePTree\ (posttrav\ p) = p,$$

where $posttrav$ returns a postorder traversal.

The $PTree$ version of $postpend$ is defined by:

$$\begin{aligned}&postpendP :: PForest \to PTree \to PForest\\ &postpendP\ [\,]\ p = [p]\\ &postpendP\ (us +\!\!+ [u])\ (p@(\mathsf{P}\ r\ ps))\\ &\quad \mid u >_d p = us +\!\!+ [u, p]\\ &\quad \mid otherwise = postpendP\ us\ (\mathsf{P}\ r\ (u : ps)).\end{aligned}$$

and removing an element from the right of a $PForest$ is the mirror image of $removeD$:

$$\begin{aligned}&removeP\ \ :: \ PForest \to (PForest, Elem)\\ &removeP\ (us +\!\!+ [\mathsf{P}\ a\ ps])\ \ =\ \ (us +\!\!+ ps, a).\end{aligned}$$

It is important to note that a $PTree$ records very different information about the DRSP structure than a $DTree$ representing the same segment. Figure 10(a) compares the $DTree$ and $PTree$ (depicted using white text on dark nodes) built out of the same list $[1, 7, 2, 6, 8]$ (with all breadths $= 1$). The $DTree$ shows that $drsp\ [7, 2, 6, 8]\ =\ [[7], [2, 6, 8]]$. The $PTree$, on the other hand, shows that $drsp\ [1, 7, 2, 6]\ =\ [[1, 7, 2, 6]]$.

$DTrees$ are designed to allow easy addition and removal of the leftmost element. It is much harder to add or remove the rightmost element from a $DTree$ while maintaining its structural invariant. Figure 10(b) shows, for example, that removing 9 from a $DTree$

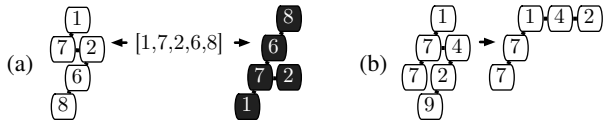

Figure 10. (a) The $DTree$ and $PTree$ built out of $[1, 7, 2, 6, 8]$. (b) Removing the rightmost element in a $DTree$.

for $[1, 7, 7, 4, 2, 9]$ involves lifting both 4 and 2 to the toplevel. For $PTree$ the situation is symmetrical — it is easier to add or remove the rightmost element. The situation is analogous to the dual cons-list and snoc-list representations of a list.

7. MDS without an Upper Bound

En route to developing an efficient algorithm for the case where the upper bound U is finite, in this section we take an intermediate step by upgrading the sliding window algorithm to use a $DForest$ to represent the DRSP part of the window. This exploits the properties established in the previous section to speed up the computation.

7.1 The Single Forest Representation

We perform another data structure transformation on the window datatype, redefining it to be $W = ([Elem], DForest)$. Again, the window is represented by a pair: the compulsory part is a list of elements with breadth at most L, and the optional part is a DRSP represented by a $DForest$. The conversion transformations between the old window type $([Elem], [[Elem]])$ to the new window type $([Elem], DForest)$ are as follows:

$$rep' = (id \times drspD)$$
$$abs' = (id \times map\ pretrav)$$

This conversion requires updating the catamorphism win' (defined in Section 5.3) to use the $DForest$, which requires the updating of $addl$ and $maxchop$. The $addl$ is transformed into a $addD$, and the function $maxchop$ can be trivially altered to a version $maxchopD$ which chops rightmost trees from a $DForest$, rather than rightmost segments from a DRSP (definition of $maxchopD$ omitted). The resulting program is shown in Figure 11, where $(\otimes)$ carries out the job previously performed by win'. In this program, to avoid a profusion of apostrophes to indicate different versions of functions, some names have been re-used (mds_{M}, zh, $\Uparrow_d$, abs, $bounds$) for the updated versions using the new window datatype.

In this algorithm, one round of the window updating (implemented by $\otimes$) is depicted in Figure 12. Each time the leading edge of the window moves one element to the left, the new element is added to the compulsory part of the window. To keep the compulsory part as the shortest prefix with breadth at least L, $split$ may shift some elements (the list c_2 in the definition of $\otimes$) from the compulsory part, which then are added to the DRSP $DForest$ by $foldr\ addD$. The addition may cause one or more segments (each represented by a $DTree$) on the left-hand end of the DRSP $DForest$ to be merged by $prependD$. Afterwards, $maxchopD$ traverses the DRSP from the right-hand end, chopping trees on the way, until it finds the leftmost maximum. This will be one local result to be compared to the global optimal result by $\Uparrow_d$.

Other points concerning the program in Figure 11: both $\uparrow_d$ and $bounds$ are trivially updated to operate on the window datatype, annotated with area and breadth. Also when using double-ended queues to implement lists, $split$ can process elements from the right-end of the input, so each element is examined at most once before being removed.

$$\begin{array}{l}
\textbf{type}\ W = ([Elem], DForest) \\
\\
mds_{\mathsf{M}}\ ::\ [Elem] \to Maybe\ [Elem] \\
mds_{\mathsf{M}}\ =\ fmap\ abs \circ fst \circ zh\ (\Uparrow_d)\ (\otimes)\ (Nothing, ([\,], [\,])) \\
\\
zh\ (\Uparrow_d)\ (\otimes)\ e\ [\,] = e \\
zh\ (\Uparrow_d)\ (\otimes)\ e\ (a : x) = (u \Uparrow_d v', v') \\
\quad \textbf{where}\ (u, v)\ =\ zh\ (\Uparrow_d)\ (\otimes)\ e\ x \\
\qquad\quad\ \ v'\ =\ a \otimes v \\
\\
(\Uparrow_d)\ ::\ Maybe\ W \to W \to Maybe\ W \\
r\ \ \Uparrow_d\ p\ \ \mid \neg(bounds\ p) = r \\
Nothing\ \Uparrow_d\ p\ \ = Just\ p \\
(Just\ m)\ \Uparrow_d\ p\ \ = Just\ (m \uparrow_d p) \\
\\
(\otimes)\ ::\ Elem \to W \to W \\
a \otimes (c, ds) = \textbf{let}\ (c_1, c_2) = split\ (a : c) \\
\qquad\qquad\ \ \textbf{in}\ (c_1, maxchopD\ c_1\ (foldr\ addD\ ds\ c_2))
\end{array}$$

Figure 11. The second algorithm, for $U = \infty$.

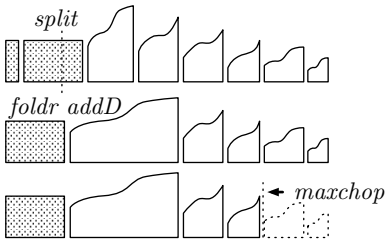

Figure 12. Illustration of one call to $\otimes$ in Figure 11.

7.2 Complexity Analysis

Since each tree and list used is augmented with areas and breadths, computation of densities can be performed in constant time. As the lists of trees are implemented using double-ended queues, each recursive call in $prependD$ takes constant time, as does each addition and removal to the DRSP $DForest$ using $addD$ and $maxchopD$.

We examine the overall running time of the algorithm using the banker's method [14], assigning credits to nodes in trees, and associating each operation with a cost that is to be paid off by credits stored in the data structure. A new node $\mathsf{D}\ r\ [\,]$ created by $addD$ is assigned one unit of credit. Each recursive call $prependD\ (\mathsf{D}\ r\ (ts +\!\!+ [u]))\ us$ uses the credit stored in u. This is possible because all nodes are moved away from the toplevel at most once. We thus have a $DForest$ where all the toplevel nodes have credit 1 and all other nodes credit 0.

Finally, each recursive call of $maxchopD$ removes one tree and uses the 1 unit of credit stored at the root of the tree. For an input of n elements, we need at most n credits to complete the entire operation. Thus the algorithm takes linear time overall.

It is important that $prepend$ and $maxchopD$ work on opposite ends of the DRSP, and thus both operations process each element at most once in the sense above. Had $maxchopD$ started looking for maximum from the left, it would have to go through the prefix of the DRSP over and over, for which we cannot guarantee linear time access.

8. MDS with an Upper Bound

In the previous section we presented an algorithm for the case when $U = \infty$ and $trim = id$. The difficulty with a general U is that before a version of $maxchop$ can be applied, $trim$ now has to remove some elements from the right-hand side of the $DForest$, and this may result in cutting through the middle of a DRSP segment in order to remove the elements to the right (see Figure 13). In this case, the remaining portion of the cut segment may no longer be right-skew, and must be converted to its DRSP.

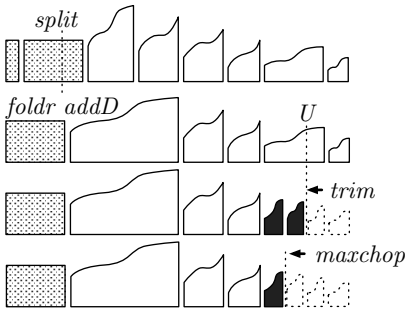

Figure 13. Use of $trim$ with the presence of an upper bound.

A $DTree$ does not have efficient removal at its right-hand end, but a $PTree$ does — so might it be more efficient if we convert the rightmost $DTree$ in the DRSP $DForest$ to a $PTree$ before we start removing elements from its right-hand end?

Recall that in the window updating step, before $trim$ is applied, some elements may need to be shifted from the compulsory part of the window to the DRSP section of the window. The function $trim$ checks through the DRSP from the right-hand end, chopping away segments making the window wider than U. When the limit of the upper bound U cuts through a $DTree$, we convert it to a $PTree$. Then elements in the $PTree$ can be efficiently removed, one by one, until the rightmost element fitting within the bound U is located.

Removing an element from the right-hand end of a $PTree$ may lift more than one $PTree$ to the top level, and therefore what we have in general is not a single $PTree$, but a $PForest$. As a first approximation, the window may be represented by a triple $(c, ds, ps) :: ([Elem], DForest, PForest)$. The forests ds and ps together represent the optional part. The conversion from a $DTree$ to a $PTree$ can be performed in time proportional to the size of the tree, simply by traversing the tree and rebuilding it. To maintain linear-time complexity, we have to make sure that each tree is converted from ds to ps at most once. This scenario is analogous to how a queue can be implemented by using two lists, and amortised linear time access is maintained by moving data from one to another in large chunks. If in the $DForest$ and $PForest$, each element can be lifted to the top level or dropped at most once, then it is still possible to maintain linear time complexity.

This all sounds fine, until one notices the possibility of the situation depicted in Figure 14(a): what if some newly bumped elements are so low in density that not only are all the $DTree$s merged into one tree t, but some prefix ps' of ps needs to be merged as well? To maintain linear time complexity, it is crucial that we do not convert any trees in ps' back to a $DTree$, in case it might later have to be converted to a $PTree$ again. Converting t to a $PTree$ does not help either, since when more elements are added at the left-hand side, adding a tree to the left of ps' takes time proportional to the size of ps', in the worst case. One may imagine some data structure with mixed $DTree$s and $PTree$s, but we have not figured out how to maintain such a structure efficiently.

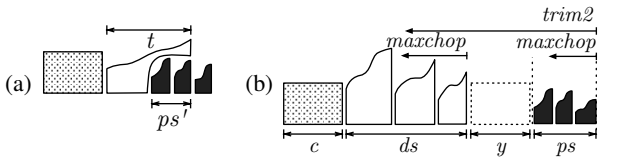

Figure 14.

Our solution, inspired by Chung and Lu [3], keeps the $DTree$s and the $PTree$s apart. Since trees in ds are not allowed to absorb the trees in ps, $ds +\!\!+ ps$ no longer accurately represents the DRSP of the optional part. Instead, the optional part is split into two parts, and we maintain their DRSP separately in ds and ps. Each DRSP has its own bitonic sequence of densities.

The final configuration of the window is shown in Figure 14(b). The window has type $([Elem], DForest, [Elem], PForest)$. Consider a window (c, ds, y, ps). The compulsory part c is still a queue at most L units wide. In each round, the function $trim2$ chops away segments and elements that makes the window wider than U. It should handle both $DTrees$ and $PTrees$, and may need to convert a $DTree$ to some $PTree$s during the chopping. We then perform $maxchop$ on both ds and ps, and pick the better result. The suffix of ds that is chopped away by $maxchop$ shall not be checked again by $maxchop$ in the next round. However, we cannot yet throw those elements away, because we will use them to compute $maxchop$ on ps. The chopped $DTree$s are thus flattened and temporarily stored in y. In effect we are computing two maximally dense prefix problems separately, one on ds with compulsory part c, the other on ps with compulsory part $c +\!\!+ flattenD\ ds +\!\!+ y$, where $flattenD = concat \circ map\ pretrav$.

8.1 Formalising the Twin-Forests Window

The algorithm using two forests motivated in the previous section is rather complex and needs a more formal justification. To prove its correctness, however, we have to restart from the very beginning — like the first algorithm, it is easier to establish the correctness using lists, before introducing trees for efficiency reasons. The list version of the new window is a triple, and the function constructing the window is specified below:

$$\begin{aligned} &win2\ ::\ [Elem] \to ([Elem], [Elem], [Elem]) \\ &win2\ [\,] \qquad = ([\,], [\,], [\,]) \\ &win2\ (a : x) = wp2\ (trim2\ (a \triangleright win2\ x)) \\ &\quad \mathbf{where}\ a \triangleright (w, y, v) = (a : w, y, v). \end{aligned}$$

If we denote the window by (w, y, v), the part w will later evolve into a compulsory part c and a forest $ds :: DForest$, and v into $ps :: PForest$, while y is the section between the two forests. We will discuss the definition of $trim2$ later, but for now, we merely require that it trims the window in a way satisfying the following constraints.

1. If $w +\!\!+ y \leq_b U$, $trim2\ (w, y, v)$ leaves w and y unchanged, but may perform some trimming in v if necessary;
2. otherwise y and v are entirely dropped, and $trim2$ returns some $(w', [\,], v')$ with $w' +\!\!+ v' = trim\ w$, to mimic in advance the behaviour that some $PTree$s might be spun off from the $DForest$. The list w' must be at least L units broad if that is the case for $w' +\!\!+ v'$.

Like wp, the function $wp2$ calls $smsp2$ if the window is at least L units wide, and leaves it unchanged otherwise:

$$\begin{aligned} wp2\ (w, y, v) \mid L \leq_b (w +\!\!+ y +\!\!+ v) &= smsp2\ (w, y, v) \\ \mid \mathbf{otherwise} \qquad\quad &= (w, y, v). \end{aligned}$$

Given a window (w, y, v), $smsp2$ respectively applies $smsp$ to w (thus we need the requirement of $trim2$ above that if the window is at least L units wide, so is w) and applies $smwr\ (w +\!\!+ y)$ to prefixes of v:

$$\begin{aligned} smsp2\ (w, y, v) \ =\ & \mathbf{let}\ w' = smsp\ w \\ & \quad\ v' = smwr\ (w +\!\!+ y)\ (\Lambda prefix\ v) \\ & \mathbf{in}\ (w', [\,], [\,]) \uparrow_d (w', (w - w') +\!\!+ y, v'). \end{aligned}$$

The results are then compared (the operator $\uparrow_d$ is overloaded to windows). If $w' = smsp\ w$ turns out to be better, we throw away the rest of the window and return $(w', [\,], [\,])$. Otherwise the new window is (w', y', v'), where y' stores the part chopped off from w, denoted by $w - w'$.

Finally, define $x \Uparrow_d (w, y, v) = x \Uparrow_d (w +\!\!+ y +\!\!+ v)$. Correctness of the twin-forest window is stated as following theorem which, like Theorem 3.2, allows mds_{M} to be a zygomorphism:

Theorem 8.1. $mds_{\mathsf{M}}\ (a : x) = mds_{\mathsf{M}}\ x \Uparrow_d win2\ (a : x)$.

The proof of this theorem is similar to proving Theorem 3.2 twice, and we need two lemmas similar to Lemma 3.3. Let $(w, y, v) = win2\ x$. The first lemma says that the optimal prefix of $u +\!\!+ w +\!\!+ y$ for some list u, if useful at all, does not extend further than in $u +\!\!+ w$:

Lemma 8.2. Let $(w, y, v) = win2\ x$. We have

$$\begin{aligned} & mds_{\mathsf{M}}\ x = Nothing\ \vee\ smsp\ (trim\ (u +\!\!+ w +\!\!+ y)) \geq_d mds\ x \\ & \Rightarrow \\ & smsp\ (trim\ (u +\!\!+ w +\!\!+ y)) = smsp\ (trim\ (u +\!\!+ w)). \end{aligned}$$

Proof. The proof is by induction on w. □

The next lemma says that the optimal prefix of $u +\!\!+ x$, if useful at all, does not extend further than $u +\!\!+ w +\!\!+ y +\!\!+ v$:

Lemma 8.3. Define $u \bowtie (w, y, v) = u +\!\!+ w +\!\!+ y +\!\!+ v$. We have

$$\begin{aligned} & mds_{\mathsf{M}}\ x = Nothing \vee smsp\ (trim\ (u +\!\!+ x)) \geq_d mds\ x \\ & \Rightarrow \\ & smsp\ (trim\ (u +\!\!+ x)) = smsp\ (trim\ (u \bowtie win2\ x)). \end{aligned}$$

Proof. The lemma is proved by induction on x. □

The assertion made by the whole of the two lemmas above is the crucial invariant that $win2\ x$ always satisfies, to guarantee the correctness of the algorithm, as follows: if the optimal prefix $smsp\ (trim\ (u +\!\!+ x))$ is denser than $mds\ x$, it always lies within either $u +\!\!+ w$ or $u +\!\!+ w +\!\!+ y +\!\!+ v$.

After the correctness proof, we perform a worker/wrapper transformation with $W = ([Elem], DForest, [Elem], PForest)$ and

$$\begin{aligned} rep\ (w, y, v) = \ & \mathbf{let}\ (c, w') = split\ w \\ & \mathbf{in}\ (c, drspD\ w', y, drspP\ v), \\ abs\ (c, ds, y, ps) = \ & (c +\!\!+ flattenD\ ds, y, flattenP\ ps), \end{aligned}$$

where $flattenP = concat \circ map\ posttrav$, and $drspP$ is the $DTree$ equivalent of $drspD$.

8.2 Trimming the Two Forests

The tree version of $trim2$ is a mere case analysis, depending on whether and where the limit of U cuts the window. Let the window be (c, ds, y, ps). If the limit of the upper bound U falls within ps, then $trim2$ calls a helper function $trimP\ b$, which in turn keeps calling $removeP$ to remove the rightmost element until the window fits within the breadth bound b:

$$\begin{aligned} & trimP :: \mathbb{R}^+ \to PForest \to PForest \\ & trimP\ b\ [\,] \qquad\qquad\quad = [\,] \\ & trimP\ b\ ps \mid ps \leq_b b \quad = ps \\ & \qquad\quad \mid \mathbf{otherwise} = trimP\ b\ (fst\ (removeP\ ps)). \end{aligned}$$

$$\begin{aligned} & mds_{\mathsf{M}} :: [Elem] \to Maybe\ [Elem] \\ & mds_{\mathsf{M}} = fmap\ abs \circ fst \circ \\ & \qquad zh\ (\uparrow\uparrow_d)\ (\lambda a\ w \to wp2\ (trim2\ (a \triangleright w))) \\ & \qquad\quad (Nothing, ([\,], [\,], [\,], [\,])) \end{aligned}$$

$$\mathbf{type}\ W = ([Elem], DForest, [Elem], PForest)$$

$$\begin{aligned} a \triangleright (c, ds, y, ps) \ =\ & \mathbf{let}\ (c_1, c_2) = split\ (a : c) \\ & \mathbf{in}\ (c_1, foldr\ addD\ ds\ c_2, y, ps) \end{aligned}$$

$$\begin{aligned} & wp2\ ::\ W \to W \\ & wp2\ w\ =\ \mathbf{if}\ w <_b L\ \mathbf{then}\ w\ \mathbf{else}\ smsp2\ w \end{aligned}$$

$$\begin{aligned} & smsp2\ ::\ W \to W \\ & smsp2\ (c, ds, y, ps) = (c, ds', [\,], [\,]) \uparrow_d (c, ds', y' +\!\!+ y, ps') \\ & \ \mathbf{where}\ cdy = c +\!\!+ flattenD\ ds +\!\!+ y \\ & \qquad ((ds', y'), ps') = (maxchop'\ c\ ds, maxchop\ cdy\ ps) \end{aligned}$$

Figure 15. The final program.

The initial value of b is $U - breadth\ (c +\!\!+ flattenD\ ds +\!\!+ y)$.

If the limit of U cuts through ds, both y and ps can be dropped. The function $trimD$ trims ds from right to left, and possibly splits off a $PForest$:

$$\begin{aligned} & trimD :: DForest \to (DForest, PForest) \\ & trimD\ [\,] = ([\,], [\,]) \\ & trimD\ (ds +\!\!+ [t]) \\ & \quad \mid ds +\!\!+ [t] \leq_b U = (ds +\!\!+ [t], [\,]) \\ & \quad \mid U \leq_b ds = trimD\ ds \\ & \quad \mid \mathbf{otherwise} = (ds, trimP\ (U - breadth\ ds)\ [d2p\ t]). \end{aligned}$$

In the last case, where the limit of U cuts through t, the $DTree$ is converted to a $PTree$ by $d2p$, and $trimP$ is called. A simple linear-time implementation of $d2p$ is given by:

$$d2p = head \circ foldl\ addP\ [\,] \circ pretrav,$$

where $addP\ ps\ a = postpendP\ ps\ (\mathsf{P}\ a\ [\,])$ and $pretrav$ returns a pre-order traversal.

Other parts of the final program is shown in Figure 15. The algorithm body mds_{M} is defined as a zygomorphism, where the function zh is that defined in Figure 11. The function $(\triangleright)$ adds an element to the leading edge of the window and adjusts the compulsory part, which may cause one or more $DTree$ to be merged by $addD$. Since ds and ps both represent DRSPs, $smsp2$ makes two calls to $maxchop$ to compute two optimal prefixes before picking the denser and shorter one. The function $maxchop'$ is a variant of $maxchop$ that returns the chopped elements in a list, and the operator $\uparrow_d$ is overloaded to windows. Note that the list concatenation and tree flattening in the definition of cdy are merely for the sake of comparing densities in $maxchop\ cdy$, and need not be actually performed.

8.3 Complexity Analysis

The complexity analysis follows the same style of that in Section 7.2. This time we allocate 4 credits for each node created by $addD$. Each recursive call of $prependD$ costs 1 credit, thus we have a $DForest$ where all the roots have 4 credits, while the nodes below have 3 credits.

The function $trimD$ either throws away a tree or converts it to a $PTree$. In the former case it uses 1 credit stored on the tree. In the latter case, the conversion takes time proportional to the size of the tree, and uses 1 credit on each node. The remaining 2 credits are transferred to the $PTree$.

Each time $removeP$ decomposes a node $\mathsf{P}\ a\ ps$, the subtrees ps are lifted to the toplevel. We let lifting of each subtree cost 1 credit stored in the tree. Credits stored in the $PForest$ are therefore almost dual to that in the $DForest$: each root in the toplevel (apart from the leftmost root) have credit 1, while the nodes below have 2 credits. The key here is that each node is lifted to the toplevel at most once. Finally, the function $maxchop$ also uses 1 credit stored in the root of a $DTree$ or a $PTree$ before throwing it away. For an input of n elements, we need at most $4n$ credits. Thus the algorithm runs in amortised linear time.

9. Conclusions

Our primary objective was to solve the maximally dense segment problem (MDS) in an elegant and informative fashion. For this, we have formally derived two linear-time algorithms, with non-uniform element breadths. The first algorithm imposes a lower bound on the breadth of feasible segments, while the second imposes an upper bound as well. Both algorithms rely on essentially the same principles as the algorithms of Chung and Lu [3] and Goldwasser et al. [7]. The algorithms are rather complex, which is why a formal development and justification was necessary. Indeed the effort put into a formal development has yielded fruit, resulting in the identification a boundary case for which existing algorithms do not produce a correct result (this is believed to be easily fixable for at least one of the existing algorithms[1]).

For our development, we used a top-down approach and motivated each design choice during the development. We illustrated that a zygomorphism is a natural computation pattern to represent a sliding window technique, and the MDS problem provides an interesting case study, because both the formal specification of the window and the invariants it satisfies are rather tricky. The window possesses a complex structure; it does not always contain the optimal prefix, but it does when it matters. We have formalised the window and its invariants, and proved their correctness.

The use of sophisticated data structures reveals much about the MDS problem. The $DTree$ and $PTree$ structures interact well with the concept of a decreasing right-skew partition (DRSP), and the use of two forests, analogous to a queue, is a new way to organise the window. We believe that the DRSP data structures, both the partitions and the trees representing them, are much clearer than the arrays with pointers used by the previous work in [3, 7], which can now be seen as optimised implementations of trees.

As a secondary objective to solving the MDS problem, we have collected and presented properties of DRSPs, operations on them, and data structures to represent them (Sections 4 and 6). Some of this work is our contribution, and some pieces are extracted from previous work; we feel it is useful to have these DRSP properties collated and stated all together.

In particular, the DRSP properties illuminate the connection between the algorithms of Chung and Lu [3] and Goldwasser et al. [7]. It has not previously been stated, for example, that Chung and Lu's algorithm, based on highest-density segments and taking repeated longest lowest-density suffixes,[5] results in a DRSP in the same way as the fold-like construction of Goldwasser et al.'s algorithm based on right-skew segments.

Like Chung and Lu's approach, our second algorithm unfortunately has to give up some nice DRSP properties. It remains an open problem whether there is a data structure that maintains the DRSP structure yet still allows efficient access from both ends.

[5] Actually Chung and Lu's algorithm uses longest lowest-density prefixes, since it operates from left to right along the input sequence.

References

[1] Kai-Min Chung. Personal communication, 2010.

[2] Kai-Min Chung and Hsueh-I Lu. An optimal algorithm for the maximum-density segment problem. In *Annual European Symposium on Algorithms*, number 2832 in Lecture Notes in Computer Science, pages 136–147, September 2003.

[3] Kai-Min Chung and Hsueh-I Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34(2):373–387, 2004.

[4] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, 7500 AE Enschede, Netherlands, February 1992.

[5] Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[6] Michael H. Goldwasser, Ming-Yang Kao, and Hsueh-I Lu. Fast algorithms for finding maximum-density segments of a sequence with applications to bioinformatics. In *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics (WABI 2002)*, pages 157–171, September 2002.

[7] Michael H. Goldwasser, Ming-Yang Kao, and Hsueh-I Lu. Linear-time algorithms for computing maximum-density sequence segments with bioinformatics applications. *Journal of Computer and System Sciences*, 70(2):128–144, 2005.

[8] Xiaoqui Huang. An algorithm for identifying regions of a DNA sequence that satisfy a content requirement. *Computer Applications in the Biosciences*, 10(3):219–225, 1994.

[9] Anne Kaldewaij. *Programming: the Derivation of Algorithms*. Prentice Hall, 1990.

[10] Yaw-Ling Lin, Tao Jiang, and Kun-Mao Chao. Efficient algorithms for locating the length-constrained heaviest segments, with applications to biomolecular sequence analysis. In *Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*, number 2420 in Lecture Notes in Computer Science, pages 459–470. Springer-Verlag, 2002.

[11] Grant R Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Groningen University, The Netherlands, 1990.

[12] Shin-Cheng Mu and Sharon Curtis. `http://www.iis.sinica.edu.tw/~scm/2010/maximally-dense-segments-code-and-proof`, 2010.

[13] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[14] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[15] Hans Zantema. Longest segment problems. *Science of Computer Programming*, 18(1):39–66, 1992.