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:

\[
\text{area} :: \text{Elem} \rightarrow \mathbb{R} \\
\text{breadth} :: \text{Elem} \rightarrow \mathbb{R}^+.
\]

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

\[
d :: [\text{Elem}] \rightarrow \mathbb{R} \\
d x = (\text{sum}(\text{map area } x)) / (\text{sum}(\text{map breadth } x)).
\]

Densities can be visualised by viewing elements as rectangles, and taking a list of type \([\text{Elem}]\) 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:

\[
\text{mds} = \text{max}_d \ast \Lambda(\text{bounds?} \ast \text{segment}).
\]

In this specification, and throughout, we will take a relation of type \(A \rightsquigarrow B\) to be a subset of the Cartesian product \(B \times A\), a function to be a special case of a relation, and \(\ast\) 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 \(\text{prefix}\) and \(\text{suffix}\) which respectively map an input list to a prefix and a suffix, the relation \(\text{segment}\) is defined by \(\text{segment} = \text{prefix} \circ \text{suffix}\). The predicate \(\text{bounds}\) checks whether the total breadth of a segment lies between a lower bound \(L\) and an upper bound \(U\):

\[
\text{bounds } x = L \leq \text{sum}(\text{map breadth } x) \leq U,
\]

and given a predicate \(p : A \rightarrow \text{Boolean}\), the relation \(p?\) is given by the relation

\[
\text{p?} = \{(x, x) | x \in A \land p x\}.
\]

The \(\Lambda\) operator converts a relation \(R : A \rightsquigarrow B\) to a function \(\Lambda R : A \rightarrow \text{PB}\), by returning the set of all the values related to the input:

\[
\Lambda R a = \{b | (b, a) \in R\}.
\]

Thus the function \(\Lambda(\text{bounds?} \ast \text{segment})\) takes an input list and returns the set of all segments of that list within the breadth bounds. Then the relation \(\text{max}_d\) 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 \((\text{area}, \text{breadth})\) 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 \(x, y :: [\text{Elem}]\), and any operator \(\oplus\in\{<,\leq,\geq,>\}\),

\[
x \oplus (x \oplus y) \iff x \oplus y \iff (x \oplus y) \oplus y. \quad (1)
\]

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 partitioning 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

\[
\text{optseg}_{f,p} = \max f \circ \Lambda(p? \circ \text{segment}).
\]

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 \( \text{mds} = \text{optseg}_{\text{max}_p, \text{anumseg}_{\infty}}, \) 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:

\[
\text{optseg}_{f,p} = \max f \circ \text{E}(\max f \circ \Lambda(p? \circ \text{prefix}) \circ \Lambda \text{suffix}),
\]

where the \( \text{E} \) operator converts a relation \( R : A \rightsquigarrow B \) to a function \( E_R : \text{FA} \rightarrow \text{FB} \), by applying \( R \) in all possible ways:

\[
ER as = \{ b \mid \exists a \in as : (b, a) \in R \}.
\]

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 is processed right-to-left, the state information kept during the algorithm is of the form

\[
s = u + w + t.
\]

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 \( t \) itself. The window is a function of the processed suffix. For the example above, we have \( \text{window} (w + t) = t \).

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 \( h : \text{[A]} \rightarrow C \), where \( h = \text{foldr} \ (\odot) \ c_b \), the function \text{zygo} is a zygomorphism if it satisfies the following equations:

\[
\begin{align*}
\text{zygo} [\ ] &:: [A] \rightarrow B \\
\text{zygo} [\ ] &:: c_a \\
\text{zygo} [a : x] &:: \text{step} (\text{zygo} x, h (a : x)).
\end{align*}
\]

Efficient computation of \text{zygo} comes from computing \( zyh x = (\text{zygo} x, h x) \) and then taking \( \text{zygo} = \text{fst} \circ zyh \), since \( \text{zyh} \) 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:

\[
\begin{align*}
zyh & \quad : [A] \rightarrow (B, C) \\
zyh & \quad = (c_s, c_h) \\
zyh (a : x) & = (\text{step} a (u, v'), v') \\
\text{where} & \quad (u, v) = zyh x \\
& \quad = u \odot v.
\end{align*}
\]

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:

\[
\begin{align*}
\text{optseg}_{f, p} (a : x) & = \text{optseg}_{f, p} \uparrow f \text{ optprefix}_{f, p} w \\
\text{where} & \quad w = \text{window} (a : x),
\end{align*}
\]

where \( \uparrow f \) returns the maximum with respect to \( f \). \( \text{optprefix}_{f, p} \) extracts an optimal prefix of \( (a : x) \) from the window \( w \), and the function \( \text{window} \) can be written as a \( \text{foldr} \). 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 \( \text{optseg}_{f, p} \) can be expressed as a zygomorphism, which maintains a pair \((\text{optseg}_{f, p} x, \text{window} x)\) 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 asymmetric operator \( \vdash \) 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

\[
mds (a : x) = mds x \vdash d \ mdlp (\text{trim} (a : x)),
\]

where \( \text{dlp} \) is a function computing a maximally dense prefix, and \( \text{trim} \) returns the longest prefix of its input with breadth at most \( U \):

\[
\text{trim} \ x = \ \text{max} (\ \text{λ} ((U \geq b) \Rightarrow \text{prefix})).
\]

\[
\text{where} \ # \ \text{is a shorthand for length}, \ \text{and} \ \geq b, \ \leq b, \ < b, \ > b, \ \text{denote comparison of breadths, overloaded to elements, lists, and scalars}. \ \text{If} \ \text{dlp} \ \text{is a foldr}, \ \text{then} \ mds \ \text{can be computed using a zygomorphism}.
\]

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 \([c_1, c_2, c_3]\) where the total breadth of \([c_1, c_2, c_3]\) exceeds \( U \), but the total breadth of \([c_1, c_2]\) is less than \( L \) and \( c_3 < c \). This input sequence does not have a solution, and indeed input sequences containing segments such as \([c_1, c_2, c_3]\) 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 \( \text{Maybe} [\text{Elem}] \). We define:

\[
\begin{align*}
mds_u & : [\text{Elem}] \rightarrow \text{Maybe} [\text{Elem}] \\
mds_u & :: \ [\ ] = \text{Nothing} \\
mds_u (a : x) & = \text{Nothing wp} (\text{trim} (a : x)),
\end{align*}
\]

where \( \text{wp} \) is a function that returns a suitable prefix (see below), and the operator \((\vdash d)\) lifts \((\vdash d)\) to \( \text{Maybe} \), if the second argument is within bounds:

\[
\begin{align*}
\text{wp} & : [\text{Elem}] \rightarrow [\text{Elem}] \rightarrow \text{Maybe} [\text{Elem}] \\
\text{wp} x \ [L < d] & = \text{smsp} x \\
\text{otherwise} & = x.
\end{align*}
\]

It can now be seen that \( x \in \text{domain} \) of \( \text{mds} \) iff \( \text{mds}_u x = \text{Just} \), for some value. From now on we refer to this value as \( \text{mds} x \) for brevity, and when we write \( \text{smsp} x \) we imply that \( x \geq b \).

### 3.2 Catamorphism Required for Yoking

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

\[
x \sqsubseteq y \Rightarrow x \sqsubseteq y \\
\text{(3)}
\]

Unfortunately, \( \text{smsp} \) is not a \( \text{foldr} \). Indeed, \( \text{smsp} (a : x) \) might well be longer than \( \text{smsp} x \) — consider the list \( x = [[9, 1], (8, 1)] \) and \( a = (1, 1) \), where \( \text{smsp} x = [[9, 1]], \text{but} \text{smsp} (a : x) = [(1, 1), (9, 1), (8, 1)] \). Consequently, \( \text{wp} \circ \text{trim} \) is not a \( \text{foldr} \) either. This is not good news, since we were hoping for a catamorphism to compute \( \text{mds} \) 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 \( \text{smsp} (u + x) \) extends further than \( \text{smsp} x \) (that is, \( u + \text{smsp} x \) is a proper prefix of \( \text{smsp} (u + x) \)), the latter is denser than the former:

#### Lemma 3.1.

\( u + \text{smsp} x \sqsubseteq \text{smsp} x \Rightarrow \text{smsp} x \sqsupset u + \text{smsp} x \).

\text{Proof.} \text{Denote} \text{smsp} x \text{ by} x_1 \text{. The antecedent implies that there exists some non-empty} x_2 \text{ such that} \text{smsp} (u + x) = u + x_1 + x_2 \text{. Since} x_1 \text{ is a maximum density prefix of} x \text{, we have} x_1 \geq x_2, x_2 \text{ which is equivalent to} x_2 \geq x_3 \text{ by the density property (1)} \text{. Since} u + x_1 + x_2 \text{ is the shortest maximum-density prefix of} u + x \text{, we have} u + x_1 + x_2 \geq u + x_1 + x_2 \text{, which is equivalent to} x_2 \geq u + x_1 + x_2 \text{ by (1)} \text{. By transitivity we have} x_1 \geq u + x_1 + x_2 \text{.}

As a special case of Lemma 3.1, if \( \text{smsp} (a : x) \) extends further than \( \text{smsp} x \), then its density will be lower than \( \text{smsp} x \). This will
lead to a suggestion for an alternative catamorphism to be yoked to the computation of $\text{mds}_a$.

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

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

$$\begin{align*}
\text{win } [a] &= [a] \\
\text{win } (a : x) &= wp (\text{trim } (a : \text{win } x)).
\end{align*}$$

This gives us the catamorphism $\text{win } = \text{foldr } (wp \circ \text{trim } (\cdot) \cdot) [\cdot]$ we need, resulting in the zygomorphism $\text{mds}_a = \text{fat } \circ \text{zh}$ to solve the MDS problem, where

$$\begin{align*}
\text{zh } : & [\text{Elem}] \to (\text{Maybe } [\text{Elem}], [\text{Elem}]) \\
\text{zh } [\cdot] &= ([\cdot], [\cdot]) \\
\text{zh } (a : x) &= \langle \text{ms } \dashv_d \text{w'}, w' \rangle \\
\text{where } (\text{ms }, w) &= \text{zh } x \\
w' &= wp (\text{trim } (a : w)).
\end{align*}$$

Formally, we need to prove:

**Theorem 3.2.** $\text{mds}_a (a : x) = \text{mds}_x \dashv_d \text{win } (a : x)$.

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

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

$$\begin{align*}
\text{mds } x \dashv_d \text{win } (a : x) \\
= \{ \text{since } a : x \geq_L L \} \\
\text{mds } x \dashv_d \text{smsp } (\text{trim } (a : x)) \\
= \{ \text{since } \text{mds } x \geq_d \text{smsp } (\text{trim } (a : x)) \text{ and } \text{smsp } (\text{trim } (a : x)) \geq_d \text{win } (a : x) \text{ by (3)} \} \\
\text{mds } x \dashv_d \text{win } (a : x).
\end{align*}$$

That leaves two other possibilities: $\text{mds}_a x = \text{Nothing}$ or $\text{smsp } (\text{trim } (a : x)) \geq_d \text{mds } x$. For that case we claim that $\text{smsp } (\text{trim } (a : x)) = \text{smsp } (\text{trim } (a : \text{win } x))$ holds, by the following lemma.

**Lemma 3.3.** For non-empty $u$ we have:

$$\text{mds}_a x = \text{Nothing } \lor \text{smsp } (\text{trim } (u + x)) \geq_d \text{mds } x \Rightarrow \text{smsp } (\text{trim } (u + x)) = \text{smsp } (\text{trim } (u + \text{win } x)).$$

**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 $\text{win } x$ 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 $\text{mds } x$, this optimal prefix always lies within $u + \text{win } x$.

Now that $\text{mds}_a$ is a zygomorphism and the window is constructed by repeatedly calling $wp \circ \text{trim}$ in a foldr, the next task is to refine $wp \circ \text{trim}$ 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

$$\text{rightskew } :: [\text{Elem}] \to \text{Boolean}$$

$$\text{rightskew } x = \text{all } (\leq_d) \left[ \langle \text{take } n x, \text{drop } n x \rangle \mid n \leftarrow [1..\#x-1] \right].$$

where all $p$ is defined by $\text{foldr } (\land \ True \ = \ \text{map } p$. An example of a right-skew segment is given in Fig. 3.

![Figure 3. An example of a right-skew segment.](image)

### 4.1 Decreasing Right-Skew Partition

Any list $x :: [\text{Elem}]$ 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 $x :: [\text{Elem}]$ there is exactly one $x s :: [[\text{Elem}]]$ such that $\text{concat } x s = x$, and $\text{sdars } x s$, where

$$\text{sdars } :: [[\text{Elem}]] \to \text{Boolean}$$

$$\text{sdars } x s = \text {sdec } (\text{map } d x s) \land \text {all rightskew } x s.$$ Here, $\text{sdars}$ stands for “strictly decreasing, all right-skew”, and $\text{sdec}$ 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.](image)

DRSPs have a number of elegant properties which are not too difficult to prove using only the density property (1). Firstly, note that the $\text{sdars}$ property is prefix- and suffix-closed: if $x s :: [[\text{Elem}]]$ satisfies $\text{sdars}$, then so too does any prefix or suffix of $x s$. 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 \( \text{neg} \) \( : \text{Elem} \rightarrow \text{Elem} \) that returns an element of the same breadth but negated area, we have

\[
\text{map} \ (\text{map neg}) \circ \text{reverse} \circ \text{map reverse} \circ \text{drsp} = \text{drsp} \circ \text{reverse} \circ \text{map neg},
\]

(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:

\[
\text{drsp} :: [\text{Elem}] \rightarrow [\text{Elem}]
\]

\[
\text{drsp} \ x = \text{leftBuild} \ \text{lrsp},
\]

\[
\text{lrsp} :: [\text{Elem}] \rightarrow [\text{Elem}]
\]

\[
\text{lrsp} = \max \#_y \cdot \Lambda \text{(right skew?} \cdot \text{prefix)},
\]

\[
\text{leftBuild} : ([\text{Elem}] \rightarrow [\text{Elem}]) \rightarrow [\text{Elem}] \rightarrow [\text{Elem}]
\]

\[
\text{leftBuild} \ m \ [\ ] = [\ ]
\]

\[
\text{leftBuild} \ m \ x = \text{pre} + \text{leftBuild} \ m \ (\text{drop} \ (#\text{pre}) \ x)
 where pre = m \ x.
\]

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

\[
\text{lhdp} :: [\text{Elem}] \rightarrow [\text{Elem}]
\]

\[
\text{lhdp} = \max \#_\Lambda \cdot \Lambda \max \#_d \cdot \Lambda \text{prefix}
\]

a DRSP can be constructed with \( \text{drsp} = \text{leftBuild} \ \text{lhdp} \). Usually, 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

\[
\text{drsp} :: [\text{Elem}] \rightarrow [\text{Elem}]
\]

\[
\text{drsp} = \text{foldl} \ \text{addr} \ [],
\]

\[
\text{addr} \ x \ [\ ] = [\ x,]
\]

\[
\text{addr} \ a \ xs = \text{prepend} \ [a] \ xs,
\]

\[
\text{prepend} :: [\text{Elem}] \rightarrow [\text{Elem}] \rightarrow [\text{Elem}]
\]

\[
\text{prepend} \ x \ [\ ] = [\ x]
\]

\[
\text{prepend} \ x \ (y : xs) | x >a y = x : y : xs
\]

\[
\text{otherwise} = \text{prepend} \ (x + y) \ ys.
\]

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 \( \text{sdars} \ xs \) holds, so does \( \text{sdars} \ (\text{prepend} \ [a] \ xs) \) 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 \( \text{drsp} \ [3] = [3] \). Then, doing \( \text{addl} \ 5 \) results in \( \text{drsp} \ [5, 3] = [5, [3]] \), since as \( |5| > L \ [3] \), no merging happens when 5 is added. When 2 is added, it is 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 \( \text{prepend} \) and \( \text{drsp} \) backwards, we discover how to remove the leftmost element from a DRSP:

\[
\text{removel} :: [\text{Elem}] \rightarrow (\text{Elem}, [\text{Elem}])
\]

\[
\text{removel} \ ((a : x) : xs) = (a, \text{drsp} \ x + xs).
\]
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):

\[
smwr :: [Elem] \rightarrow P[Elem] \rightarrow Elem
\]

where \( f = d \circ (z+) \).

That is, \( smwr \) 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.** \( z + smwr z \) (A-prefix \( x \)) = \(|d| (z + x) \) 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

\[
z + smwr (E(x+) (A-prefix y)) = (z + x) \uparrow_d (z + x + y).
\]

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 \( z + x_1 <_d x + x_1 + x_2 \) 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

\[
L = \Lambda (concat \circ prefix),
\]

\[
L[x_1,x_2,..,x_n] = \{[],x_1,x_1 + x_2,..,x_1 + x_2 + .. + x_n\},
\]

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

**Theorem 5.3.** \( smwr z (A-prefix (concat xs)) = smwr z (opts xs) \), 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 \( [x_1,x_2,..] \) with decreasing densities, the densities of \( [c,c+x_1,c+x_1+x_2,..] \) must be bitonic -- ascending and then descending, as depicted in Figure 7. The top part of this diagram depicts the window, where each right-skew segment. The blocks in the bottom row represent the densities of \( c, c+x_1, c+x_1+x_2, \ldots \).

![Figure 7. Bitonic property of DRSP.](image)

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:

\[
maxchop :: [Elem] \rightarrow [[Elem]] \rightarrow [[Elem]]
\]

\[
maxchop c \ n = \begin{cases} c \conc (concat (xs + [x])) \geq_d x = maxchop c xs \\
\text{otherwise} \end{cases}
\]

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 \( smsp xs \) 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

\[
mds_M :: [Elem] \rightarrow Maybe [Elem]
\]

\[
mds_M [] = Nothing
\]

\[
mds_M (a : xs) = mds_M xs \uparrow_d \text{win} (a : x),
\]

where the window was constructed by

\[
\text{win} :: [Elem] \rightarrow [Elem]
\]

\[
\text{win} [] = []
\]

\[
\text{win} (a : x) = \text{wp} (\text{trim} (a : \text{win} x)),
\]

where \( \text{wp} \) calls \( smsp \) when the window is at least \( U \) 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 \( rep :: [Elem] \rightarrow W \) that transforms the list of elements in the window to our representation type \( W \), and we will also write a reverse transformation \( abs :: W \rightarrow [Elem] \). If \( \text{abs} \circ \text{rep} = \text{id} \), we may compute \( mds_M \) by \( \text{fmap abs} \circ mds_M \), where \( \text{fmap} :: (a \rightarrow b) \rightarrow \text{Maybe} \) →.
Maybe $b$ is the functorial mapping lifting a function to Maybe type.
The function $mds'_c$ is defined by:

\[
\begin{align*}
mds'_c &:: [\text{Elem}] \rightarrow \text{Maybe W} \\
mds'_c [] & = \text{Nothing} \\
mds'_c (a : x) & = mds'_c x \times f\alpha_d \text{ win'} (a : x),
\end{align*}
\]
and $f\alpha_d$ is simply $\alpha_d$ lifted to $W$:

\[
(f\alpha_d) :: \text{Maybe W} \rightarrow W \rightarrow \text{Maybe W}
\]

The function $\text{win'}$ is defined below:

\[
\begin{align*}
\text{win'} :: [\text{Elem}] & \rightarrow W \\
\text{win'} [] & = \text{rep} [] \\
\text{win'} (a : x) & = \text{rep} (\text{wp} (\text{trim} (a : \text{abs} (\text{win'} x)))).
\end{align*}
\]

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

For our purpose here, let $W = ([[\text{Elem}]], [[\text{Elem}]])$. 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 $\text{rep}$ and $\text{abs}$, we define a function $\text{split} :: [\text{Elem}] \rightarrow ([\text{Elem}], [\text{Elem}])$ which splits a list into its compulsory and optional parts:

\[
\begin{align*}
\text{split} x & = \text{split'} ([], x) \\
\text{split'} (z, []) & = (z, []) \\
\text{split'} (z, a : x) & = \begin{cases} 
(z + [a], \le B \text{ with } \text{split'} (z + [a], x) & \text{if otherwise } (z, a : x),
\end{cases}
\end{align*}
\]

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 $\text{rep}$ and $\text{abs}$:

\[
\begin{align*}
\text{rep} = (\text{id} \times \text{drsp}) \circ \text{split} \\
\text{abs} (z, xs) = z + \text{concat} xs,
\end{align*}
\]

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

**Lemma 5.5.** If breadth $x \ge L$ and $(z, x') = \text{split} x$, then $\text{snmsp} x = z + \text{smwr} z (\Lambda \text{prefix} x')$.

**Proof.** This is an easy calculation from the definitions of $\text{snmsp}$ (page 3) and $\text{smwr}$ (page 6).

We will use this lemma during the next task, which is to simplify the definition of $\text{win'}$. Thus we calculate starting from the expression $\text{rep} \times \text{wp} \circ (\text{a} : \alpha) \circ \text{abs}$, using typical program calculation to push $\text{rep}$ 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 $\text{rep}$ encounters $\text{snmsp}:

\[
\begin{align*}
((\text{id} \times \text{drsp}) \circ \text{split}) \circ \text{snmsp} x & = \{ \text{let } (z, x') = \text{split} x, \text{ by Lemma 5.5 } \} \\
(z, \text{drsp} (\text{smwr} z (\Lambda \text{prefix} x')) & = \{ \text{concat} \circ \text{drsp} = \text{id} \} \\
(z, \text{drsp} (\text{smwr} z (\Lambda \text{prefix} (\text{concat} \circ \text{drsp} x')))) & = \{ \text{Theorem 5.3 } \} \\
(z, \text{drsp} (\text{smwr} z (\text{opts} \circ \text{drsp} x'))) & = \{ \text{Theorem 5.4 } \} \\
(z, \text{drsp} (\text{concat} \circ \text{maxchop} z (\text{drsp} x'])) & = \{ \text{a prefix of a DRSP is a DRSP } \}
\end{align*}
\]

The transformation is a special case of Gill and Hutton’s formulation [5] by letting $\text{unwrap} = (\text{fmap} \text{ rep} \circ)$ and $\text{wrap} = (\text{fmap} \text{ abs} \circ)$.

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

Some more work needs to be done when $\text{rep}$ encounters $(a : \alpha)$. To push $\text{rep}$ towards, we define $\text{split} (a : z)$ in terms of $\text{split} x$:

\[
\begin{align*}
\text{split} (a : z) & = \{ \text{let } (z, w) = \text{split} x \} \\
& \quad \{ (z_1, z_2) = \text{split} (a : z) \} \\
& \quad \{ \text{in } (z_1, z_2 + w) \}.
\end{align*}
\]

In summary, we have derived

\[
\begin{align*}
\text{win'} [] & = ([], []) \\
\text{win'} (a : x) & = \{ \text{let } (z, xs) = \text{win'} x \} \\
& \quad \{ (z_1, z_2) = \text{split} (a : z) \} \\
& \quad \{ \text{in } (z_1, \text{maxchop} z_1 (\text{foldr addl dresp } xs z_2)) \}
\end{align*}
\]

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

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, $\text{trim}$ is no longer $\text{id}$, 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 $\text{remover}$ function (defined at the end of Section 4) to $\text{dresp} (xs + [x + [a]])$. 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 $\text{cons}$ and $\text{snoc}$ 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 $\text{addl}$) to the left-hand side of the DRSP section of the window, and when we $\text{trim}$ with respect to a non-infinite upper bound $U$, elements are removed from the right-hand side. The current $\text{remover}$ is inefficient in removing elements from the right, because it includes a call to $\text{drsp}$ to re-form 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-
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:

$$
type DForest = [DTree],
\text{data } DTree = D \text{ Elem } DForest,
$$

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 $DTrees$ 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.

$$
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
\end{array}
$$

Figure 8. The $DTree$ $D1 \ D4 \ [D5 \ []], D2 \ [D5 \ []], D3 \ []$

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

$makeDTree :: [Elem] \rightarrow DTree$

$makeDTree (a : x) = D a (map makeDTree (drsp x)).$

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 $drsp, addD$, and $prepend$ from Section 4, we define functions $drspD, addD$, and $prependD$:

$$drspD :: [Elem] \rightarrow DForest$$

$drspD = foldr addD []$.

$addD :: Elem \rightarrow DForest \rightarrow DForest$

$addD a ts = prepend (D a []) ts$.

$prependD :: DTree \rightarrow DForest \rightarrow DForest$

$prependD t [] = [t]$

$prependD (t@(D r ts)) (u : us)$

$t > a u = f : u : us$

$otherwise = prependD (D r (ts ++ [u])) us.$

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 $t \leq a u$. 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.](image)

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

$removeD :: DForest \rightarrow (Elem, DForest)$

$removeD ((D a ts) : us) = (a, ts ++ us)$.

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:

$$type PForest = [PTree],$$

$\text{data } PTree = P \text{ Elem } PForest.$

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

$makePTree :: [Elem] \rightarrow PTree$

$makePTree (x \oplus [a]) = P a (map makePTree (drsp x)),$

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

$makePTree \ (posttrav p) = p,$

where $posttrav$ returns a postorder traversal.

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

$\text{postpendP :: PForest \rightarrow PTree \rightarrow PForest}$

$\text{postpendP} (p \oplus [u]) (p \oplus (P r ps))$

$\begin{cases} u > a p = us \oplus [u, p] \\ \text{otherwise} = \text{postpendP} (us) \oplus (P r (u : ps)). \end{cases}$

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

$\text{removeP :: PForest \rightarrow PTree, Elem}$

$\text{removeP} (us \oplus [P a ps]) = (us \oplus ps, a)$.

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
for \([1, 7, 7, 4, 2, 9]\) involves lifting both 4 and 2 to the toplevel. For \(P\text{Tree}\) 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 \(D\text{Forest}\) 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 \(W = ([\text{Elem}], D\text{Forest})\). 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 \(D\text{Forest}\). The conversion transformations between the old window type \(([\text{Elem}], [[\text{Elem}]])\) to the new window type \(([\text{Elem}], D\text{Forest})\) are as follows:

\[
\text{rep}' = (id \times \text{drspD}) \\
\text{abs}' = (id \times \text{map pretrav})
\]

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

In this algorithm, one round of the window updating (implemented by \(\odot\)) 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 \(L\), \(\text{split}\) may shift some elements (the list \(c_2\) in the definition of \(\odot\)) from the compulsory part, which then are added to the DRSP \(D\text{Forest}\) by \(\text{foldr addD}\). The addition may cause one or more segments (each represented by a \(D\text{Tree}\)) on the left-hand end of the DRSP \(D\text{Forest}\) to be merged by \(\text{prependD}\). Afterwards, \(\text{maxchopD}\) 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 \(\hat{\text{f}}_d\).

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

Figure 10. (a) The \(D\text{Tree}\) and \(P\text{Tree}\) built out of \([1, 7, 2, 6, 8]\). (b) Removing the rightmost element in a \(D\text{Tree}\).

\[
\text{type} W = ([\text{Elem}], D\text{Forest})
\]

\[
\text{mds}_d :: [\text{Elem}] \rightarrow \text{Maybe} [\text{Elem}]
\]

\[
\text{mds}_d = \text{fmap abs} \circ \text{fst} \circ \text{zh} (\hat{\text{f}}_d) (\odot) (\text{Nothing}, ([], []))
\]

\[
\text{zh} (\hat{\text{f}}_d) (\odot) e [ ] = e
\]

\[
\text{zh} (\hat{\text{f}}_d) (\odot) e (a : x) = (u \hat{\text{f}}_d v', v')
\]

where \((u, v) = \text{zh} (\hat{\text{f}}_d) (\odot) e\)

\[
v' = a \odot v
\]

\[
(\hat{\text{f}}_d) :: \text{Maybe} W \rightarrow W \rightarrow \text{Maybe} W
\]

\[
\text{Nothing} \hat{\text{f}}_d p \ |
\text{Nothing} = \text{Just} p
\]

\[
\text{Just} m \hat{\text{f}}_d p = \text{Just} (m \hat{\text{f}}_d p)
\]

\[
(\odot) :: \text{Elem} \rightarrow W \rightarrow W
\]

\[
a \odot (c, ds) = \text{let} (c_1, c_2) = \text{split} (a : c) \\
\text{in} (c_1, \text{maxchopD} c_1 (\text{foldr addD} ds c_2))
\]

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

Figure 12. Illustration of one call to \(\odot\) 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 \(\text{prependD}\) takes constant time, as does each addition and removal to the DRSP \(D\text{Forest}\) using \(\text{addD}\) and \(\text{maxchopD}\).

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 \(D r [\ ]\) created by \(\text{addD}\) is assigned one unit of credit. Each recursive call \(\text{prependD} (D r [ts ++ [u]])\) 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 \(D\text{Forest}\) where all the toplevel nodes have credit 1 and all other nodes credit 0.

Finally, each recursive call of \(\text{maxchopD}\) 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 \(\text{prependD}\) and \(\text{maxchopD}\) work on opposite ends of the DRSP, and thus both operations process each element at most once in the sense above. Had \(\text{maxchopD}\) 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 \( \text{trim} = \text{id} \). The difficulty with a general \( U \) is that before a version of \text{maxchop} can be applied, \text{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.

\[ \text{split} \]

\[ \text{fold} \text{ addD} \]

\[ \text{trim} \]

\[ \text{maxchop} \]

Figure 13. Use of \text{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 \text{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 \text{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 \((c, ds, ps) :: (\text{Elem}, \text{DForest}, \text{PForest})\). 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 DTrees 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 DTrees and PTrees, but we have not figured out how to maintain such a structure efficiently.

Our solution, inspired by Chung and Lu [3], keeps the DTrees and the PTrees 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 \((\text{[Elem]}, \text{DForest}, \text{[Elem]}, \text{PForest})\). 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 \text{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 PTrees during the chopping. We then perform \text{maxchop} on both \( ds \) and \( ps \), and pick the better result.

The suffix of \( ds \) that is chopped away by \text{maxchop} shall not be checked again by \text{maxchop} in the next round. However, we cannot yet throw those elements away, because we will use them to compute \text{maxchop} on \( ps \). The chopped DTrees 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 \( c + \text{flattenD} ds + y \), where \text{flattenD} = \text{concat} \circ \text{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:

\[
\begin{align*}
\text{win2} :: & \text{[Elem]} \rightarrow \text{([Elem], [Elem], [Elem])} \\
\text{win2} [\text{[]}] & = \text{([[], [], []])} \\
\text{win2} (a : x) & = \text{wp2} (\text{trim2} (a \triangleright \text{win2} x)) \\
\text{where} & a \triangleright (w, y, v) = (a : w \triangleright y, v).
\end{align*}
\]

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

1. If \( w + y \leq L \), \text{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 \text{trim2} returns some \((w', [v'], v')\) with \( w' \triangleright v' = \text{trim} w, v \), to mimic in advance the behaviour that some PTrees might be spun off from the DForest. The list \( w' \) must be at least \( L \) units broad if that is the case for \( w' \triangleright v' \).

Like \text{wp}, the function \text{wp2} calls \text{smssp2} if the window is at least \( L \) units wide, and leaves it unchanged otherwise:

\[
\begin{align*}
\text{wp2} (w, y, v) | L \leq (w + y + v) & = \text{smssp2} (w, y, v) \\
\text{otherwise} & = (w, y, v).
\end{align*}
\]
Given a window \((w, y, v)\), \(\text{smsp}2\) respectively applies \(\text{smsp}\) to \(w\) (thus we need the requirement of \(\text{trim}2\) above that if the window is at least \(L\) units wide, so is \(w\)) and applies \(\text{smwr}\) \((w + y)\) to prefixes of \(v\):
\[
\text{smsp}2 (w, y, v) = \begin{cases} 
\text{let } w' = \text{smsp} w \\
\  \ v' = \text{smwr} (w + y) \ (\text{Prefix} v) \\
\in (w', [ ] ) \ \map{d} (w', (w' - w) + y, v'). 
\end{cases}
\]
The results are then compared (the operator \(\map{d}\) is overloaded to windows). If \(w' = \text{smsp} w\) 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 \(x \map{d} (w, y, v) = x \map{d} (w + y + v)\). Correctness of the twin-forest window is stated as following theorem which, like Theorem 3.2, allows \(\text{mds}_u\) to be a zygomorphism:

**Theorem 8.1.** \(\text{mds}_u (a : x) = \text{mds}_u x \map{d} \text{win2} (a : x)\).

The proof of this theorem is similar to proving Theorem 3.2 twice, and we need two lemmas similar to Lemma 3.3. Let \((w, y, v) = \text{win2} x\). 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 \(u + w + w\):

**Lemma 8.2.** Let \((w, y, v) = \text{win2} x\). We have \(\text{mds}_u x = \text{Nothing} \lor \text{smsp} \ (\text{trim} \ (u + w + y)) \geq d \text{mds} x\)
\[\Rightarrow\]
\(\text{smsp} \ (\text{trim} \ (u + w + y)) = \text{smsp} \ (\text{trim} \ (u + w))\).

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 \(u + w + y + y + v\):

**Lemma 8.3.** Define \(u \map{t} (w, y, v) = u + w + y + y + v\). We have \(\text{mds}_u x = \text{Nothing} \lor \text{smsp} \ (\text{trim} \ (u + x)) \geq d \text{mds} x\)
\[\Rightarrow\]
\(\text{smsp} \ (\text{trim} \ (u + x)) = \text{smsp} \ (\text{trim} \ (u \map{t} \text{win2} x))\).

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 \(\text{win2} x\) always satisfies, to guarantee the correctness of the algorithm, as follows: if the optimal prefix \(\text{smsp} \ (\text{trim} \ (u + x))\) is denser than \(\text{mds} x\), it always lies within either \(u + w\) or \(u + w + y + y + v\).

After the correctness proof, we perform a worker/wrapper transformation with \(W = ([\text{Elem}], \text{DForest}, [\text{Elem}], \text{PForest})\) and
\[
\begin{align*}
\text{rep} \ (w, y, v) & = \begin{cases} 
\text{let} \ (c, w') = \text{split} w \\
\in (\text{drels}D w', y, \text{drelP} v)
\end{cases} \\
\text{abs} \ (c, d, s, y, ps) & = (c + \text{flatten}D ds y, \text{flatten}P ps)
\end{align*}
\]
where \(\text{flattenP} = \text{concat} \circ \text{map} \text{posttrav}\) and \(\text{drelP}\) is the \(\text{DTree}\) equivalent of \(\text{drelSP}\).

### 8.2 Trimming the Two Forests

The tree version of \(\text{trim}2\) is a mere case analysis, depending on whether and where the limit of \(U\) cuts the window. Let the window be \((c, d, s, y, ps)\). If the limit of the upper bound \(U\) falls within \(ps\), then \(\text{trim}2\) calls a helper function \(\text{trimP}\ b\), which in turn keeps calling \(\text{removeP}\) to remove the rightmost element until the window fits within the breadth bound \(b\):
\[
\begin{align*}
\text{trimP} :: \mathbb{R}^+ & \rightarrow \text{PForest} \rightarrow \text{PForest} \\
\text{trimP} \ b \ [ ] & = [ ] \\
\text{trimP} \ b \ ps & \begin{cases} 
ps \leq b \ b = ps \\
\text{otherwise} = \text{trimP} \ b \ (\text{fst} \ (\text{removeP} \ ps))
\end{cases}
\end{align*}
\]

The initial value of \(b\) is \(U - \text{breadth} \ (c + \text{flatten}D ds + y)\).

If the limit of \(U\) cuts through \(ds\), both \(y\) and \(ps\) can be dropped. The function \(\text{trimD}\) trims \(ds\) from right to left, and possibly splits off a \(\text{PForest}\):
\[
\begin{align*}
\text{trimD} :: \text{DForest} & \rightarrow (\text{DForest}, \text{PForest}) \\
\text{trimD} \ [ ] & = ([ ], [ ])
\end{align*}
\]

If \(ds + [ ] \leq b U = (ds + [ ], [ ])
\]
\[
\begin{cases} 
\text{otherwise} = (ds, \text{trimP} (U - \text{breadth} \ ds) [ ] [ dsP ])
\end{cases}
\]

In the last case, where the limit of \(U\) cuts through \(t\), the \(\text{DTree}\) is converted to a \(\text{PTree}\) by \(\text{d2p}\), and \(\text{trimP}\) is called. A simple linear-time implementation of \(\text{d2p}\) is given by:
\[
\begin{align*}
\text{d2p} & = \text{head} \circ \text{foldl} \text{addP} \ [ ] \ \text{pretrav},
\end{align*}
\]

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

Other parts of the final program is shown in Figure 15. The algorithm body \(\text{mds}_u\) is defined as a zygomorphism, where the function \(\text{zh}\) is that defined in Figure 11. The function \(\text{zh}\) adds an element to the leading edge of the window and adjusts the compulsory part, which may cause one or more \(\text{DTree}\) to be merged by \(\text{dadd}\) since \(ds\) and \(ps\) both represent DRSPs, \(\text{smsp}2\) makes two calls to \(\text{maxchop}\) to compute two optimal prefixes before picking the denser and shorter one. The function \(\text{maxchop}\) is a variant of \(\text{maxchop}\) that returns the chopped elements in a list, and the operator \(\map{d}\) is overloaded to windows. Note that the list concatenation and tree flattening in the definition of \(\text{d2p}\) are merely for the sake of comparing densities in \(\text{maxchop} cdy\), 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 \(\text{addD}\). Each recursive call of \(\text{prependP}\) costs 1 credit, thus we have a \(\text{DForest}\) where all the roots have 4 credits, while the nodes below have 3 credits.

The function \(\text{trimD}\) either throws away a tree or converts it to a \(\text{PTree}\). 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 \(\text{PTree}\).
Each time \texttt{remove}\textsubscript{P} decomposes a node \textsubscript{P} a \textit{ps}, the subtrees \textit{ps} are lifted to the toplevel. We let lifting of each subtree cost 1 credit stored in the tree. Credits stored in the \textit{PForest} are therefore almost dual to that in the \textit{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 \texttt{maxchop} also uses 1 credit stored in the root of a \textit{DTree} or a \textit{PTree} before throwing it away. For an input of \textit{n} elements, we need at most 4\textit{n} 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 \cite{Chung92} and Goldwasser et al. \cite{Goldwasser02}. 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 \cite{Chung01}).

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 \textit{DTree} and \textit{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 \cite{Chung92, Goldwasser02}, 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 \cite{Chung92} and Goldwasser et al. \cite{Goldwasser02}. 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,\textsuperscript{3} 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.

\textsuperscript{3} Actually Chung and Lu’s algorithm uses longest-lowest-density prefixes, since it operates from left to right along the input sequence.

References

\begin{thebibliography}{10}
\bibitem{Chung01} Kai-Min Chung. Personal communication, 2010.
\bibitem{Grant90} Grant R Malcolm. \textit{Algebraic Data Types and Program Transformation}. PhD thesis, Groningen University, The Netherlands, 1990.
\end{thebibliography}