Functional Pearl: Finding A Densest Segment

SHARON CURTIS
Oxford University, UK
and SHIN-CHENG MU
Academia Sinica, Taiwan
(e-mail: sharon.curtis@ctsu.ox.ac.uk, scm@iis.sinica.edu.tw)

Abstract

The problem of finding a densest segment of a list is similar to the well-known maximum segment sum problem, but its solution is surprisingly challenging. We give a general specification of such problems, and formally develop a linear-time online solution, using a sliding window style algorithm. The development highlights some elegant properties of densities, involving partitions that are decreasing and all right-skew.

1 Introduction

Processing large datasets, such as DNA sequences, requires efficient algorithms. The data analysis problem we examine here involves finding a segment (contiguous subsequence) of a list of elements with maximum density (to be defined later). While this problem resembles the well-known maximum segment sum problem (MSS), a standard textbook example of program derivation (Kaldewaij, 1990), the internal structure of the segment density problem is more intricate, and finding a linear-time solution is much harder.

Our aim in this paper is the derivation of a linear-time functional program to solve a generalised segment density problem, in an elegant and clear way. We present two instances of such density problems: MMS and MDS; the latter has been solved before (Chung & Lu, 2004; Goldwasser et al., 2005), but in an imperative setting, where it is difficult to see the structure and correctness of the algorithms in amongst the details.

Our algorithm derivation uses a traditional program calculation approach, at least initially. However, as several of the proofs are tricky or sizable, for additional reassurance the theorems and their proofs are also coded into the dependently-typed language/theorem prover Agda (Norell, 2007) and the AOPA library (Mu et al., 2009); the complete proof is around 3500 lines of Agda code.

In this paper, we outline the main proofs, and we also provide a supplement containing full proof details. These documents, along with an implementation in Haskell, are available at (Mu & Curtis, 2014). The paper is structured as follows: in the rest of this section we present two instances of the segment density problem, along with their history and applications, before giving a general specification of such problems in Section 2. The development of the algorithm then proceeds in two stages: in Section 3 we develop the structure of the algorithm, which processes the input list using a sliding-window technique.
The data structure that represents the window, and its crucial properties that allow the algorithm to run in linear time, are presented in Section 4. The resulting algorithm is presented in Section 5 along with an analysis of its performance before we conclude and summarise related work in Section 6.

1.1 Maximum Mean Segment (MMS)

Let input data elements be of type $\text{Elem}$, where each has a weight:

$$\text{weight} :: \text{Elem} \rightarrow \mathbb{R}.$$  

For the MMS problem, a list of elements can be evaluated by calculating its density, which is simply its mean weight:

$$d_{\text{MMS}} :: [\text{Elem}] \rightarrow \mathbb{R}$$

$$d_{\text{MMS}} \text{ seg} = \frac{\sum (\text{map weight seg})}{\text{length seg}}.$$  

We are interested in finding a segment of maximum density, given an input list of elements, but this would be trivial: all that is required is a singleton list containing an element of greatest weight. To make the problem more interesting, we require a function $\text{width} :: [\text{Elem}] \rightarrow \mathbb{R} \geq 0$ on segments, with the property that an empty list has zero width, and adding more elements increases width. Formally:

$$\text{width} [\ ] = 0,$$  \hspace{1cm} (1)

$$\text{width} x < \text{width} (x ++ y) > \text{width} y,$$  \hspace{1cm} (2)

where $x$ and $y$ are any non-empty lists of elements. For example, the function $\text{length}$ is such a width function, but there are other more interesting possibilities (see below).

Having defined a width function, we now introduce the MMS problem: the finding of a maximally dense segment with respect to $d_{\text{MMS}}$, subject to a lower bound on segment width.

One application concerns the analysis of blood glucose data from an individual with diabetes (a disorder of blood sugar control). An answer to the question “When were blood sugar levels worse over the past year?” might prove useful in identifying factors impairing control, and typical data consists of a sequence of timestamped blood glucose readings in chronological order, such as that illustrated in Figure 1. This situation is an instance of the MMS problem: each element is a blood glucose reading, and its weight is how far it deviates from the target range; thus the density of a segment is how far its readings are...
outside the target range, on average. The width of a segment is the time it spans, and it makes sense to put a lower bound on that timespan (for example, a week) as spurious extreme readings can happen occasionally, so it is not the single worst reading that is sought, but an average over a suitably-long period.

We expect that MMS problems can be identified in many other application areas, including analysis of other kinds of medical history data.

### 1.2 Maximum Density Segment (MDS)

The MDS problem is similar: each element has a weight as before, and also a (strictly positive) size:

\[ \text{size} :: \text{Elem} \rightarrow \mathbb{R}^{>0}. \]

The width of a segment is the sum of the sizes of its elements:

\[ \text{width}_{\text{MDS}} :: [\text{Elem}] \rightarrow \mathbb{R}^{>0} \]

\[ \text{width}_{\text{MDS}} = \text{sum} \cdot \text{map size}. \]

Note that the above satisfies the aforementioned width constraints (1) and (2). For MDS, the density of a non-empty segment is calculated by dividing its total weight by its width:

\[ d_{\text{MDS}} :: [\text{Elem}] \rightarrow \mathbb{R} \]

\[ d_{\text{MDS}} \text{ seg} = (\text{sum (map weight seg)})/(\text{width}_{\text{MDS}} \text{ seg}). \]

Densities can be visualised accurately: when each element is depicted as a rectangle as wide as its size, with area corresponding to its weight, then the density of a segment is then simply its average height, e.g. see Figure 2. Again, we set a lower bound on segment width, to make finding a segment with maximum density more interesting than simply finding a tallest element. Thus the MDS problem is to find a maximally dense segment with respect to \( d_{\text{MDS}} \) of sufficient width. For example, the densest segment of the list of elements in Fig. 2, with width at least 9, is the segment \([14, 7, (20, 4)]\), which has density 34/11 ≈ 3.1.

Fig. 2. A list of elements, with weights in bold and sizes as indicated. The density \( d_{\text{MDS}} \) of the whole list is \((9 + 6 + 14 + 20 - 10 + 20 - 2 + 27)/(6 + 2 + 7 + 4 + 5 + 8 + 2 + 6) = 84/40 = 2.1\), which is shown by the height of the dashed line.

The MDS problem has applications in bioinformatics, to the analysis of genetic material to find fragments with high densities of certain characteristics. For example, sections of DNA or RNA that are \textit{GC-rich} – with a high density of bases that are guanine (G) or cytosine (C) – are of interest because they are likely to indicate portions containing genes (Han & Zhao, 2009).
The solution of the MDS problem has a varied history: in (Huang, 1994), a simpler version of the MDS problem was considered, where all elements have unit size. Huang noticed that with a lower bound \( L \) on segment width, a maximally dense segment need not be wider than \( 2L - 1 \), which leads to a simple \( O(nL) \) algorithm, for \( n \) input elements. Later, the complexity of a MDS solution was improved to \( O(n\log L) \) (Lin et al., 2002).

Goldwasser et al. studied a variation of MDS that also used an upper bound \( U \) on segment width (Goldwasser et al., 2002), and they presented an \( O(n) \) algorithm for the simpler case of MDS when all elements have unit size (also an instance of MMS), and an \( O(n\log(U - L + 1)) \) algorithm for elements with variable sizes. Another published algorithm claimed to be linear, but wasn’t; details can be found in (Chung & Lu, 2004). Then both Chung and Lu (Chung & Lu, 2003; Chung & Lu, 2004) and Goldwasser et al. (Goldwasser et al., 2005) produced similar algorithms to take linear time to solve MDS with lower and upper bounds.

## 2 Densest Segment Problem Specification

In this section, we generalise the MMS and MDS problems to produce an abstract specification for the problem of finding a segment with maximum density, subject to a lower bound on segment width.

### 2.1 Initial Specification

We take as given an input list of elements of type \( \text{Elem} \). One standard way to produce all the segments of a list is to take all prefixes of suffixes:

- \( \text{prefixes}, \text{suffixes}, \text{segments} :: [a] \rightarrow [[a]] \),
- \( \text{prefixes} = \text{inits} \),
- \( \text{suffixes} = \text{tails} \),
- \( \text{segments} = \text{concat} \cdot \text{map} \text{prefixes} \cdot \text{suffixes} \).

Later, we will write \( x \subseteq y \) to denote that \( x \) is a prefix of or equal to \( y \), and use \( 
\) for the strict version.

Segment widths are measured using a function \( \text{width} :: [\text{Elem}] \rightarrow \mathbb{R}^\geq \) that abides by the aforementioned width constraints (1) and (2). The lower bound on segment widths is given by \( L : \mathbb{R}^+ \), and the following predicate checks whether a segment is wide enough:

\[
\text{wide} :: [\text{Elem}] \rightarrow \text{Bool}
\]

\[
\text{wide} \ x = \text{width} \ x \geq L.
\]

Densities can be compared using a relation \( (\leq_d) :: [\text{Elem}] \rightarrow [\text{Elem}] \rightarrow \text{Bool} \). The use of the \( \leq_d \) notation is intended to suggest comparison with respect to a density function \( d \), but the existence of a function is not necessary: we just require that \( \leq_d \) is reflexive, transitive and total. As a shorthand, we will write \( x <_d y \) for \( x \leq_d y \wedge y \not\leq_d x \), and write \( \geq_d \) and \( >_d \) for the converses of \( \leq_d \) and \( <_d \). The relation \( \leq_d \) must satisfy the following density property: for any non-empty lists \( x, y :: [\text{Elem}] \),

\[
x \leq_d x \downarrow y \iff x \leq_d y \iff (x \downarrow y) \leq_d y.
\]
Functional Pearl: Finding A Densest Segment

The above must also hold if \( \leq_d \) is replaced with \( <_d \), \( \geq_d \) or \( >_d \).

In particular, the relations used for the MMS and MDS problems are \( \leq_{d_{MMS}} \) and \( \leq_{d_{MDS}} \) respectively, and straightforward arithmetical calculation shows that these both satisfy the above density property.

To specify maximally dense segments, we will use a binary operator \( \Uparrow \), which selects the maximum of two arguments with respect to a total ordering \( \preceq \):

\[
x \Uparrow \preceq y = \begin{cases} x, & \text{if } y \preceq x; \\ y, & \text{otherwise}. \end{cases}
\]

In the case of a tie between the two arguments, \( \Uparrow \) selects the left-hand argument. Later, we will also need an operator \( \Downarrow \), which is defined similarly to break ties in favour of the right-hand argument. We will abbreviate \( \Uparrow_d \) and \( \Downarrow_d \) by \( \Uparrow \) and \( \Downarrow \) respectively, to avoid cumbersome subscripts.

Given a binary operator \( \oplus \) (such as \( \Uparrow \) or \( \Downarrow \)) that selects the maximum of two values, the following function selects the maximum of a list:

\[
\begin{align*}
\max_{\oplus} [x] & = x \\
\max_{\oplus} (x : xs) & = x \oplus (\max_{\oplus} xs).
\end{align*}
\]

We can now specify the problem of finding a densest segment as:

\[
\max_{d} \Uparrow \preceq \cdot \text{filter wide} \cdot \text{segments}. \tag{4}
\]

At this stage, \( \Downarrow_d \) could just as easily have been used, and the choice of \( \Uparrow_d \) will be justified later.

This specification can be a starting point for a calculation to derive a solution. However, during the subsequent development, a number of issues occur, which we find easier to manage if the initial specification is refined first.

2.2 Refinement

Note that the above specification (4) is not a total function, because it may be that no segments of the input are wide enough. Unfortunately this lack of totality forms a major obstacle to deriving an algorithm to solve this problem, as will be discussed later.

Attempting a derivation using relational algebra instead of functional programming calculus does not help, as the requirement for totality remains. To ensure totality, there needs to be unified treatment of wide-enough segments and those that are not wide enough. Using a \( \text{Maybe} \) type is one possibility, but this leads to a very fiddly derivation.

Instead, we use another way to model treating segments differently depending on whether they are wide enough, by generalising \( \leq_d \) to a relation \( \preceq \) with the following properties:

\[
\preceq \text{ is reflexive, transitive, and total;} \tag{5}
\]

\[
\neg \text{wide } x \land \text{ wide } y \Rightarrow x \prec y; \tag{6}
\]

\[
\text{wide } x \land \text{ wide } y \Rightarrow (x \preceq y \Leftrightarrow x \leq_d y), \tag{7}
\]

\[
\neg \text{wide } x \land \neg \text{wide } y \Rightarrow (x \preceq y \Leftrightarrow \text{width } x \leq \text{width } y), \tag{8}
\]

where \( x \prec y \) is a shorthand for \( x \preceq y \land y \npreceq x \). Informally, the requirement (5) ensures that taking a maximum from a non-empty list of segments is always possible, (6) says that
wide-enough segments are always strictly better than those not wide enough, (7) says that amongst wide-enough lists, a denser segment is better, and (8) says that amongst not-wide-enough segments, wider is better (the reason for which will appear later as a consequence of the algorithmic development).

Thus selecting a densest segment can be carried out by $\max_{\preceq} \cdot \text{segments}$, which we will abbreviate to $\max_{\preceq}$. Our new version of the specification is then

$$\text{mds} = \max_{\preceq} \cdot \text{segments}.$$  \hspace{1cm} (9)

This is a total specification, unlike (4). It produces a slightly different output, which can be used to solve the original problem: if the resulting segment is not at least $L$ wide, then there were no segments wide enough. Otherwise, the output is a densest segment with respect to $\leq_d$, which will be the leftmost such segment, as $\preceq$ is used.

### 3 Development

Starting from the $\text{mds}$ specification, we take a standard formal development step to calculate for the cases [ ] and (a : x). After obtaining $\text{mds} [ ] = [ ]$, we then proceed as follows:

$$\text{mds} (a : x)$$

$$= \left\{ \text{definition of } \text{mds} (9) \right\}$$

$$\text{segments} (a : x)$$

$$= \left\{ \text{definitions of } \text{segments and suffixes} \right\}$$

$$\max_{\preceq} \left( \text{concat} \left( \text{map prefixes} \left( (a : x) : \text{suffixes} x \right) \right) \right)$$

$$= \left\{ \text{definitions of } \text{map and concat} \right\}$$

$$\max_{\preceq} \left( \text{prefixes} (a : x) \right) \preceq \text{concat} \left( \text{map suffixes} x \right)$$

$$= \left\{ \text{max distributes over } \preceq \text{ as } \preceq \text{ associative} \right\}$$

$$\max_{\preceq} \left( \text{prefixes} (a : x) \right) \preceq \max_{\preceq} \left( \text{concat} \left( \text{map suffixes} x \right) \right)$$

$$= \left\{ \text{definitions of } \text{segments and mds} \right\}$$

$$\max_{\preceq} \left( \text{prefixes} (a : x) \right) \preceq \text{mds} x$$

$$= \left\{ \text{definition, see below} \right\}$$

$$\text{mdp} (a : x) \preceq \text{mds} x.$$  

Above, we used this definition of a maximally-dense prefix ($\text{mdp}$) with respect to $\preceq$:

$$\text{mdp} = \max_{\preceq} \cdot \text{prefixes}.$$  \hspace{1cm} (10)

Maximally-dense prefixes will feature heavily in the coming calculations, so we now pause the development in order to collect some basic properties of $\text{mdp}$ that we will need later.

#### 3.1 Properties of $\text{mdp}$

Firstly, as can be deduced from its definition, $\text{mdp}$ returns a prefix of its input:

$$\text{mdp} x \subseteq x.$$  \hspace{1cm} (11)
for any list of elements $x$, where $y \sqsubseteq x$ denotes that $y$ is a prefix of $x$. Secondly, this lemma describes the relationship between $mdp$ and segment width:

Lemma 3.1
For any list of elements $x$,

$$\text{width} x \leq L \implies x = mdp x,$$  \hfill (12)

$$\text{wide} x \iff \text{wide} (mdp x).$$  \hfill (13)

The implication (12) expresses that taking $mdp$ of a narrow segment has no effect, and (13) expresses that $mdp$ does not affect whether a segment is sufficiently wide or not.

Proof
The implication (12) is proved using a straightforward induction on $xs$. A straightforward calculation shows that $\text{wide} x \implies \text{wide} (mdp x)$, which together with (12) is sufficient to show (13). (For details, see the supplement.)

The function $mdp$ has two reasonably-obvious monotonicity properties:

Lemma 3.2 (Monotonicity of $mdp$)
For any lists of elements $x, y$ such that $x \sqsubseteq y$, we have that

$$mdp x \sqsubseteq mdp y,$$  \hfill (14)

$$mdp x \preceq mdp y.$$  \hfill (15)

Finally, we have the following lemma that says if $x$ is “sandwiched” in between $mdp y$ and $y$, then the maximally-dense prefixes of $x$ and $y$ are the same:

Lemma 3.3 (Sandwich Lemma)
For any lists of elements $x$ and $y$,

$$mdp y \sqsubseteq x \sqsubseteq y \implies mdp x = mdp y.$$  \hfill (16)

3.2 A sliding window algorithm

Returning to the algorithmic development, above we calculated the following expression, which requires the finding of a densest prefix, for all suffixes of the input:

$$\text{mds} \ [] = []$$

$$\text{mds} \ (a : x) = mdp \ (a : x) \ |
\text{mds} \ x.$$  

Despite being in the form of a foldr, this is inefficient, as $mdp$ would take too long to compute for every suffix of the input. If, however, $mdp$ itself can be expressed as a foldr, then the problem can be solved by performing the paired computation $\langle \text{mds}, mdp \rangle^1$, which can be carried out as a foldr: this would compute an optimal segment $\text{mds} \ x$ and an optimal prefix $\text{mdp} \ x$ for each suffix $x$ of the input. Then the value $mdp \ (a : x)$ would be readily available at each step, when calculating $\text{mds} \ (a : x)$ from $\text{mds} \ x$.

[A technical aside: afficionados of the morphism zoo may like to note that the $\langle \text{mds}, mdp \rangle$ computation illustrates the zygomorphism computational pattern (Malcolm, 1990), where

1 The “split” operator is defined by $\langle f, g \rangle a = (f a, g a)$.}
the value of one expression is computed from a fold that pairs it with a second expression that is directly computable from a fold ("zygo" means "yoked"). The zygomorphic nature of our algorithm is why its development requires a calculus with total expressions, as discussed at the start of Section 2.2. This is because zygomorphisms exist in the category of sets and total functions (Fun), but are not guaranteed to exist in the categories of partial functions (PFun) and relations (Rel).]

Unfortunately, mdp is not a foldr. If it were, then mdp \((a : x)\) could be calculated from \(a\) and mdp \(x\) alone, but consider the list \([([1, 5], (9, 2), (8, 2)])\) of \((\text{weight}, \text{size})\) elements for the MDS problem, with a segment width lower bound \(L = 2\). Here mdp \([([9, 2], (8, 2)] = [[9, 2]],\) but mdp \([([1, 5], (9, 2), (8, 2)]) = [[(1, 5)], (9, 2), (8, 2)],\) which is not computable solely from \((1, 5)\) and \([([9, 2]].\)

One of the surprising features of the mds problem, however, is that mdp not being a foldr does not seem to matter. In the above example, mdp \([([9, 2], (8, 2)]\) is denser than mdp \([([1, 5], (9, 2), (8, 2)])\), and thus the latter cannot be a solution to the whole problem. This suggests that perhaps mdp \((a : x)\) is not needed when \(x\) has a denser shorter prefix. This is the case, and we will prove a slightly generalised version: if mdp \((u ++ x)\) is longer than \(u ++ mdp x\), then mdp \((u ++ x)\) is less dense than mdp \(x\). The following lemma formalises this property, in its contrapositive form:

**Lemma 3.4 (Overlap Lemma)**

For all lists of elements \(u\) and \(x\), we have

\[
\text{mdp } x \leq \text{mdp } (u ++ x) \Rightarrow \text{mdp } (u ++ x) \sqsubseteq u ++ \text{mdp } x.
\]

This means that if mdp \((u ++ x)\) is at least as good (meaning denser, for wide-enough segments) as mdp \(x\), then \(u\) together with mdp \(x\) contains sufficiently many elements to compute it.

**Proof**

We do a case analysis on the width of \(x\).

**Case** \(~\text{wide} x.\)

By (12) we have \(x = \text{mdp } x\). The goal thus simplifies to mdp \((u ++ x) \sqsubseteq u ++ x\), which is true from property (11).

**Case** \(\text{wide } x.\)

By (13) we know that \(\text{wide } (\text{mdp } x)\), thus if mdp \(x \leq mdp \((u ++ x)\)\), it must be the case that \(\text{wide } (\text{mdp } (u ++ x))\) also, and mdp \(x \leq_d \text{mdp } (u ++ x)\). The goal reduces to proving that

\[
\text{mdp } x \leq_d \text{mdp } (u ++ x) \Rightarrow \text{mdp } (u ++ x) \sqsubseteq u ++ \text{mdp } x.
\]

We prove its contrapositive:

\[
u ++ \text{mdp } x \sqsubset \text{mdp } (u ++ x) \Rightarrow \text{mdp } (u ++ x) <_d \text{mdp } x.
\]

Note that since \(u ++ \text{mdp } x \sqsubset \text{mdp } (u ++ x)\), we can let \(\text{mdp } (u ++ x) = u ++ \text{mdp } x ++ z\), for some non-empty \(z\). We first prove that \(u ++ \text{mdp } x ++ z <_d z\):

\[
u ++ \text{mdp } x \sqsubset \text{mdp } (u ++ x)
\]

\[
\Leftrightarrow \quad \{ \text{decomposition of mdp } (u ++ x), \text{as above} \}
\]
Functional Pearl: Finding A Densest Segment

\[ u + + mdp x \sqsubseteq mdp (u + + x) = u + + mdp x + + z \]
\[ \Rightarrow \quad \{ \text{claim, see below} \} \]
\[ u + + mdp x \prec mdp (u + + x) = u + + mdp x + + z \]
\[ \Rightarrow \quad \{ \text{wide}(mdp x), \text{width increases with } + + (2), \text{definition of } \preceq \} \]
\[ u + + mdp x \prec_d u + + mdp x + + z \]
\[ \Rightarrow \quad \{ \text{density property (3)} \} \]
\[ u + + mdp x + + z \prec_d z. \]

The second step is valid because \( \max_{\leq} \) (within the definition of \( mdp \)) chooses the leftmost of the maximal elements. As \( u + + mdp x \), being a proper prefix of \( mdp (u + + x) \), appears to the left of \( mdp (u + + x) \) in the list of all prefixes of \( u + x \), then \( u + + mdp x \) must be strictly lesser if it is not returned.

Then we prove that \( z \leq_d mdp x \):
\[ u + + mdp x + + z = mdp (u + + x) \]
\[ \Rightarrow \quad \{ \text{property of } mdp (11) \} \]
\[ u + + mdp x + + z \sqsubseteq u + + x \]
\[ \Leftrightarrow \quad \{ \sqsubseteq \} \]
\[ mdp x + + z \sqsubseteq x \]
\[ \Rightarrow \quad \{ mdp x \text{ maximum wrt } \leq, \text{ and wide}(mdp x), \text{width increases with } + + (2) \} \]
\[ mdp x + + z \leq_d mdp x \]
\[ \Rightarrow \quad \{ \text{density property (3)} \} \]
\[ z \leq_d mdp x. \]

Thus by transitivity we conclude that \( u + + mdp x + + z \prec_d mdp x \), which is the same as \( mdp (u + + x) \prec_d mdp x. \quad \square \)

The Overlap Lemma (3.4) suggests an alternative to computing \( mdp x \) for each suffix \( x \) of the input list of elements: each new element \( a \) is added to the front of the prefix produced from the previous stage of the computation, and then \( mdp \) of this segment is computed.

[An aside: this strategy is what suggests the use of \( \sqsubseteq \) in the definition (10) of \( mdp \): in the case that there is more than one densest prefix of a segment, \( \max_{\sqsubseteq} \) chooses the shortest, thus leaving fewer elements to be processed later on than if the \( \sqsubseteq \) operator were to be used. In turn, this justifies the use of \( \sqsubseteq \) in the definition (9) of \( mds \) as well.]

Formally, this computation can be expressed as the following function \( wp \):
\[
wp \ [] = []
\]
\[
wp \ (a : x) = mdp \ (a : wp \ x).
\]

Note that unlike \( mdp \), the function \( wp \) is clearly a \textit{foldr}. We now hope to use the value of \( wp \) instead of \( mdp \) for each suffix, in order to compute \( mds \). That is, given the following function \( ms \):
\[
ms \ [] = []
\]
we need to show that \( mds = ms \), which occurs in Section 3.4.

The benefit of using \( ms \) may not be obvious. Recall that earlier, we were unable to use the paired computation \( \langle mds, mdp \rangle \) to compute \( mds \) efficiently because \( mdp \) is not a \textit{foldr}. Now, however, as \( wp \) is a \textit{foldr}, we can use a paired computation \( \langle ms, wp \rangle \), expressible as a \textit{foldr} that makes one call to \( mdp \) in each step. Letting \( mswp = \langle ms, wp \rangle \), a standard \textit{tupling} transformation gives us:

\[
mswp \left[ \right] = ([], [])
mswp \left( a : x \right) = (w' \parallel \leq m, w')
\]

\textbf{where} \((m, w) = mswp x \)

\[w' = mdp \left( a : w \right).\]

This follows the paradigm known as a \textit{sliding window} algorithm (Zantema, 1992): such a computation maintains a segment of the input sequence (the \textit{window}), that “slides” along the sequence at each step of the algorithm, always in the same direction. Above, \( w \) is the window storing the value \( wp x \) from the previous step of the computation, and the application of \( mdp \cdot \left( a : \right) \) produces an updated window \( w' \) that is a prefix of \( a : w \), thus sliding the window to the left. (The name \( wp \) stands for “window processing”.)

[Another aside: this is the point where it is clear how \( \leq \) compares segments that are not wide enough, as expressed earlier in (8). When the algorithm has barely started, and is still examining suffixes of the input that are not wide enough, it is not clear what \( wp \) should return, but for the first suffix \( x \) that is wide enough, \( mds x \) will be \( x \) itself, and therefore \( wp \) will need to retain all the elements of \( x \). This means that on segments that are not wide enough, \( \leq \) should favour wider segments.]

Given suitable list structures to enable constant time computation of \( \parallel \leq \), the above algorithm can be implemented in linear time, provided that the computation of \( mdp \cdot \left( a : \right) \) can be done in amortised constant time. This is our goal in Section 4: choosing suitably efficient data structures for the window.

\section{3.3 Properties of \( wp \) and \( ms \)}

As \( mdp \) returns a prefix of its input (11), a short inductive proof starting from the definition of \( wp \) (18) shows that \( wp x \) also returns prefixes:

\[wp x \sqsubseteq x, \quad (20)\]

for any sequence of elements \( x \).

The following lemma shows that \( wp \) and \( ms \) have similar relationships to the width of segments that \( mdp \) does (see Lemma 3.1).

\textbf{Lemma 3.5}

For any list of elements \( x \),

\[\text{width} x \leq L \quad \Rightarrow \quad x = wp x = mds x, \quad (21)\]

\[\text{width} x \quad \Leftrightarrow \quad \text{width} (wp x) \quad \Leftrightarrow \quad \text{width} (mds x). \quad (22)\]
3.4 Correctness of the sliding window algorithm

Now we carry out our task to prove that

\[ mds x = ms x. \]

Proof

We use an induction on \( x \), and the \([\ ]\) case is immediate from the definitions of \( mds \) and \( ms \). For the \( a : x \) case, the proof obligation (after standard use of the definition of \( wp \) and the induction hypothesis that \( mds x = ms x \)) is

\[
mdp (a : x) \preceq mds x = mdp (a : wp x) \preceq mds x,
\]

(23)

Due to how \( \preceq \) is defined, we start from a case analysis on the width of \( x \).

Case 1: \( width x < L \). By (21), \( x = wp x \), and thus (23) holds.

Case 2: \( width x \geq L \) and \( mdp (a : x) <_d mds x \). For this case, the left-hand side of (23) reduces to \( mds x \). To show that the right-hand side also reduces to \( mds x \), we have to prove that \( mdp (a : wp x) \prec mds x \). We reason:

\[
\begin{align*}
mdp (a : wp x) & < mds x \\
\iff \quad \{ \text{since } mdp (a : x) <_d mds x \} \\
mdp (a : wp x) & \preceq mdp (a : x) \\
\iff \quad \{ \text{monotonicity of } mdp \text{ as in (15)} \} \\
a : wp x & \subseteq a : x \\
\iff \quad \{ \text{wp produces prefixes (20)} \} \\
true.
\end{align*}
\]

Case 3: \( width x \geq L \) and \( mdp (a : x) \geq_d mds x \). This is the difficult case.

Recall from earlier that when comparing two segments which are equal with respect to \( \preceq \), the operator \( \lhd \) chooses its left-hand argument. Thus we need the following to be true:

\[
mdp (a : x) \geq_d mds x \Rightarrow mdp (a : x) = mdp (a : wp x),
\]

(24)

in the circumstance that \( width x \geq L \). If this is true, the two sides of (23) will immediately be equal.

To establish (24) we need a separate lemma, where \( a \) is generalised to a list, and this is Lemma 3.6 below. Then we obtain (24) by letting \( z := [a] \) and \( y := x \) in (25) of the lemma. The condition required is \( mds x \preceq mdp (a : x) \), and this is obtained by using the facts that \( mdp (a : x) \geq_d mds x \) and \( width x \geq L \), together with (22).

\[
\square
\]

Lemma 3.6

For any finite lists of elements \( y \) and \( z \),

\[
mds y \preceq mdp (z ++ y) \Rightarrow mdp (z ++ y) = mdp (z ++ wp y).
\]

(25)

Proof
The proof proceeds by induction on \( y \). The case for \( y = [] \) is routine. For the case \( a : y \), note that our assumption is that

\[
\text{mds} (a : y) \preceq \text{mdp} (z ++ (a : y)).
\]

and for the induction hypothesis, we can assume that

\[
\text{mdsy} \preceq \text{mdp} (v ++ y) \Rightarrow \text{mdp} (v ++ y) = \text{mdp} (v ++ wp y),
\]

for any finite list of elements \( v \).

We need to show that \( \text{mdp} (z ++ a : y) = \text{mdp} (z ++ wp (a : y)) \). One crucial idea is to use the sandwich lemma to turn the proof obligation from an equality to a prefix relation, which then allows us to use the overlap lemma, the key property that guarantees that we need not consider prefixes that are too long. The proof goes as follows:

\[
\begin{align*}
\text{mdp} (z ++ a : y) &= \text{mdp} (z ++ wp (a : y)) \\
&\Leftarrow \quad \text{sandwich lemma (16)} \\
&\quad \text{mdp} (z ++ a : y) \sqsubseteq z ++ wp (a : y) \sqsubseteq z ++ a : y \\
&\Leftarrow \quad \text{by (20), } wp (a : y) \sqsubseteq a : y \\
&\quad \text{mdp} (z ++ a : y) \sqsubseteq z ++ wp (a : y) \\
&\Leftarrow \quad \text{induction, with } v = z ++ [a], \text{ writing } P \text{ for } \text{mdsy} \preceq \text{mdp} (z ++ a : y) \\
&\quad \text{mdp} (z ++ a : wp y) \sqsubseteq z ++ wp (a : y) \land P \\
&\Leftarrow \quad \text{definition of wp} \\
&\quad \text{mdp} (z ++ a : wp y) \sqsubseteq z ++ \text{mdp} (a : wp y) \land P \\
&\Leftarrow \quad \text{overlap lemma (17)} \\
&\quad \text{mdp} (a : wp y) \preceq \text{mdp} (z ++ a : wp y) \land P \\
&\Leftarrow \quad \text{induction} \\
&\quad \text{mdp} (a : wp y) \preceq \text{mdp} (z ++ a : y) \land P \\
&\Leftarrow \quad \text{by monotonicity (15), and } a : wp y \sqsubseteq a : y \\
&\quad \text{mdp} (a : y) \preceq \text{mdp} (z ++ a : y) \land P \\
&\Leftarrow \quad \text{expanding } P \\
&\quad \text{mdp} (a : y) \preceq \text{mdp} (z ++ a : y) \land \text{mdsy} \preceq \text{mdp} (z ++ a : y) \\
&\Leftarrow \quad \text{maximum} \\
&\quad \text{mdp} (a : y) \preceq \text{mdsy} \preceq \text{mdp} (z ++ a : y) \\
&\Leftarrow \quad \text{definition of mds} \\
&\quad \text{mds} (a : y) \preceq \text{mdp} (z ++ a : y) \\
&\Leftarrow \quad \text{assumption} \\
&\quad \text{true}.
\end{align*}
\]
4 Window

Recall the definition of \( wp \) (18), which describes the sliding of the window in the main algorithm:

\[
wp [] = [] \\
wp (a : x) = mdp (a : wp x)
\]

Our goal is to choose a suitable data structure for the window so that the computation of \( mdp \cdot (a : \cdot) \) takes amortised constant time, thus allowing the overall algorithm to be linear.

We will examine \( mdp \) in order to see how to represent the window. In what follows, let the window \( wp x \) at the previous step of the computation be denoted by \( w \), and let \( w' = a : w \).

4.1 Window Header

Recall that the function \( mdp \) takes the best prefix of \( w' \) with respect to \( \langle \leq \|. \rangle \). In the case that \( w' \) is not wide enough (i.e. \( \neg \text{wide } w' \)), then we simply have \( mdp w' = w' \), from Lemma 3.1. In the case when \( \text{wide } w' \), another straightforward step can be taken, given part (6) of the definition of \( \langle \leq \|. \rangle \), which says that wide-enough segments are always better than too-short segments. Thus when \( \text{wide } w' \), it is also the case that \( \text{wide } (mdp w') \), as formalised in Lemma 3.1, (13). This means that when \( \text{wide } w' \), the prefix \( mdp w' \) must include sufficiently many elements that it is wide enough; these elements can be thought of as a compulsory “header” prefix of \( w' \).

Let the function \( hsplit \) be such that it splits a segment into its header prefix, and the rest of its elements.

\[
hsplit :: [Elem] \rightarrow ([Elem], [Elem]),
\]

A possible definition of \( hsplit \) is given later (see Figure 8 in Section 5.2), but for now, we just need the following property: if \( hsplit w' = (h, x) \), we have \( h ++ x = w' \), where \( h \) (the “header”) is the compulsory prefix, and either

- \( width h < L \) and \( x = [] \), or
- \( width (init h) < L \leq width h \) (i.e. \( h \) is the shortest wide-enough prefix).

Furthermore, recall that part (7) of the definition of \( \langle \leq \| \rangle \) says that the better of two wide-enough segments is determined by \( \leq_d \). Thus when \( \text{wide } w' \), taking a maximum prefix using the \( \langle \leq \| \rangle \) operator in \( mdp w' \) is the same as \( \max_d \) (prefixes \( w' \)). This means that for wide-enough \( w' \), we can restrict our attention to prefixes of \( w' \) of the form \( h ++ y \), where \( (h, x) = hsplit w' \) and \( y \sqsubseteq x \), and compare densities with respect to \( \langle \leq_d \).

We thus represent a window as a pair, where the first component is the header \( h \),

\[
\text{type Window} = ([Elem], \ldots),
\]

and the second component is some structure yet to be determined, to represent the rest of the window. From this structure we will need to find the segment that produces the densest prefix of the whole window with respect to \( \langle \leq_d \rangle \), when prepended by the header \( h \).

4.2 Densest Prefixes

Let \( (h, x) = hsplit w' \); our goal is to find a prefix of \( w' \) of the form \( h ++ y \), that is densest with respect to \( \langle \leq_d \rangle \). One possibility is that \( y = [] \) and \( h \) itself is the densest such segment;
otherwise, there must be a non-empty \( y \) such that \( h <_d h + + y \), which is equivalent to \( h <_d y \), by the density property (3). This leads us to investigate non-empty prefixes of \( x \) that are denser than \( h \); indeed perhaps choosing \( y \) to be a maximally-dense prefix of \( x \) might result in a densest prefix \( h + + y \)?

Unfortunately, this is not the case. Intuitively, it seems possible that there might exist a prefix of \( x \) that is slightly less dense than \( y \), but wider, and able to produce a denser prefix overall when prepended by the header \( h \). A concrete example is illustrated in Figure 3: the prefix \( y = [(23, 2), (-3, 1), (61, 3), (12, 2), (69, 4), (43, 4), (88, 6), (97, 10), (82, 5), (77, 5), (60, 5), (60, 7), (45, 3), (-50, 10), (32, 9), (-18, 9), (69, 9), (66, 8)] \) is a maximally-dense non-empty prefix of \( x \) with density 13.5, resulting in a density of 11.1 for \( h + + y \). This can be bettered by extending \( y \) on the right by the elements \( z = [(43, 4), (88, 6), (97, 10), (82, 5), (77, 5), (60, 5)] \), where we have that the density of \( h + + y + + z \) is 12.

This is disappointing, but maybe a densest prefix can still be of use? Although the example above illustrates that \( y \) (a maximally-dense prefix of \( x \)) can be bettered by a prefix wider (longer) than \( y \), perhaps non-empty prefixes shorter than \( y \) can be ruled out? It turns out that this is indeed the case, which will be shown in what follows.

### 4.3 Right-Skew Segments

First, we will need to examine some properties of densest prefixes. Note that if a non-empty list of elements \( y = y' + + y'' \) is a maximally-dense prefix of \( x \), then as \( y' \) is also a prefix of \( x \), we have \( y' \leq_d y \). Then, from the density property, this is equivalent to \( y' \leq_d y'' \). This leads us to the following definition:

**Definition 4.1 (Right-Skew)**

A non-empty list \( z \) of elements is called right-skew if for all \( n \) such that \( 0 < n < \text{length } z \), we have take \( n \) \( z \leq_d \) drop \( n \) \( z \). Let us denote this by the predicate

\[
\text{rightskew } (\leq_d) \quad : \quad \text{Elem} \rightarrow \text{Bool}.
\]

The idea of right-skew segments originated from Lin et al. (2002); informally, a segment is right-skew when chopping it into two always results in a right-hand side that is denser than the left. Please note that density does not increase in a monotonic fashion: although in general, shorter prefixes of right-skew segments are generally less dense than longer prefixes, this is merely a general trend, and should not be relied upon. For example see Figure 4.
From the above reasoning, we have that every densest prefix of a list of elements is right-skew (the converse does not hold):

**Lemma 4.2**
Let \( y \) be a non-empty prefix of a list of elements \( x \), such that \( y \) is maximally dense with respect to \( \leq_d \). Then \( y \) is right-skew.

One property of right-skew segments we will need is that the concatenation of two right-skew segments in the correct order is itself right-skew:

**Lemma 4.3**
If \( z \) and \( z' \) are both right-skew and \( z \leq_d z' \), then \( z ++ z' \) is right-skew.

Returning to the development, recall that given a non-empty list of elements \( y \) that is a maximally dense prefix of \( x \), we wish to rule out the use of prefixes shorter than \( y \). The following lemma allows us to do so:

**Lemma 4.4**
Let \( z_1, z_2, z_3 \) be non-empty list of elements such that \( z_2 \leq_d z_3 \). Then

\[
(z_1 ++ z_2) \downarrow_d (z_1 ++ z_2 ++ z_3) = z_1 \downarrow_d (z_1 +++ z_2 ++ z_3).
\]

In words, this means that the densest among the three lists \( z_1, z_1 ++ z_2 \) and \( z_1 ++ z_2 ++ z_3 \) with respect to \( \downarrow_d \) is either \( z_1 \) or \( z_1 ++ z_2 ++ z_3 \). Thus, as a densest prefix \( y \) of \( x \) is right-skew, and hence any proper division of \( y \) into \( y' ++ y'' \) results in \( y' \leq_d y'' \), Lemma 4.4 allows us to deduce that of the three lists \( h, h ++ y' \) and \( h ++ y \), the densest with respect to \( \downarrow_d \) is not \( h ++ y' \). Thus non-empty prefixes of \( y \) can be eliminated from consideration. This can be shown formally, as an application of the following theorem:

**Theorem 4.5**
Let \( z \) and \( y \) be non-empty lists of elements with \( y \) right-skew. Then

\[
\max_{\downarrow_d} (\text{map} (z++) (\text{prefixes} y)) = z \downarrow_d (z ++ y).
\]

This helps suggest a strategy for structuring the rest of the window, as follows. Having shown that the interior of any right-skew prefix \( y \) of \( x \) can be eliminated from consideration, it seems reasonable that choosing the longest possible right-skew prefix of \( x \) (let us denote this prefix by \( y_0 \)) may help reduce the remaining computation within the rest of the window. Furthermore, having established \( h \) and \( h ++ y_0 \) as possible candidates for the mdp of the window, but with no need to consider prefixes in between, it seems reasonable to try the same step with the rest of the window.
That is, let \( y_1 \) be the longest right-skew prefix of \( x \setminus y_0 \), then applying Theorem 4.5 with \( z = h + + y_0 \) and \( y = y_1 \), removes prefixes between \( h + + y_0 \) and \( h + + y_0 + + y_1 \) from consideration. Similarly, let \( y_2 \) be the longest right-skew prefix of \( x \setminus (y_0 + + y_1) \), and so on, resulting in a partition of \( x \) into right-skew lists of elements \([y_0, y_1, \ldots, y_k]\). Repeated application of Theorem 4.5 leads us to the following theorem:

**Theorem 4.6**

Given \( h : \text{[Elem]} \) and \( xs :: \text{[[Elem]]} \) such that each list in \( xs \) is right-skew, we have

\[
\max \updownarrow_d (\text{map} (\text{h++}) (\text{prefixes} (\text{concat} \, xs))) = \max \updownarrow_d (\text{map} ((h++ \cdot \text{concat}) (\text{prefixes} \, xs))).
\]

The above theorem says that to compute \( \max \updownarrow_d (h + + y_0 + + y_1 + + \ldots + + y_k) \) we only need to consider the ends of each partition: \( h, h + + y_0, h + + y_0 + + y_1, \) etc.

Later on, we will need an alternative version of the above theorem, as follows:

**Corollary 4.7**

Given \( h : \text{[Elem]} \) and \( xs :: \text{[[Elem]]} \) such that each list in \( xs \) is right-skew, we have

\[
\max \updownarrow_h (\text{prefixes} (\text{concat} \, xs)) = \text{concat} (\max \updownarrow_{h++} (\text{prefixes} \, xs)),
\]

where \( \updownarrow_h \) and \( \updownarrow_{h++} \) are defined by

\[
\begin{align*}
x \updownarrow_h y & \equiv h + + x \leq_d h + + y, \\
xs \updownarrow_{h++} ys & \equiv h + + \text{concat} \, xs \leq_d h + + \text{concat} \, ys.
\end{align*}
\]

The above partitioning of \( x \) into right-skew lists will be how the rest of the window is structured, but first we need to formally examine the properties of the partition we have just created, and prove some properties we will need.

### 4.4 Decreasing Right-Skew Partitions

In this section, we present the concept of *decreasing right-skew partitions*, which were proposed by Lin et al. (2002). We also investigate their properties, which show that these partitions are those proposed for the window structure in our algorithm development.

A decreasing right-skew partition (abbreviated DRSP) of a list of elements is a partition of the list into segments, such that each segment is right-skew and the densities of the segments are strictly decreasing from left to right. Formally:

**Definition 4.8 (DRSP)**

Let \( xs \) be a partition of a list of elements \( x \), that is, where \( \text{concat} \, xs = x \). The list \( xs \) is a *decreasing right-skew partition (DRSP)* of \( x \) when

- all (rightskew \( (\leq_d) \)) \( xs \), and
- \( \text{sdec} \, (\leq_d) \) \( xs \),

where \( \text{sdec} \, (\leq) \) is the predicate that expresses that a list is strictly decreasing with respect to a linear ordering \( \leq \), i.e. \( \text{sdec} \, (\leq) [x_1 \ldots x_n] \) holds precisely when \( x_1 \triangleright x_2 \triangleright \ldots \triangleright x_n \).

Note that the right-skew property ensures that each segment of the partition is non-empty. An example of a DRSP can be seen in Figure 5.

The first important property that we need is that DRSPs are unique:
**Functional Pearl: Finding A Densest Segment**

Fig. 5. The (unique) DRSP of the list \( x \) from Fig. 3 is \( \left[ (23, 2), (-3, 1), (61, 3), (12, 2), (69, 4), (43, 4), (88, 6), (97, 10), (82, 5), (77, 5), (60, 5), (60, 7), (45, 3), (-50, 10), (32, 9), (-18, 9), (69, 9), (66, 8) \right] \). The densities of its segments are 13.5, 13.1, 12.8, 12, 10.5 and 2.2 respectively, indicated by the dashed lines representing the average heights of the segments.

**Theorem 4.9 (DRSP uniqueness)**

Let \( x \) be a list of elements. There exists precisely one decreasing right-skew partition of \( x \).

This is a surprising property of such partitions, and indeed we did not find the proof of uniqueness in (Lin et al., 2002) convincing, so have provided our own in the supplement to this paper.

A consequence of DRSP uniqueness that we will need is the necessary existence of a function \( \text{drsp}_{\leq} d : [\text{Elem}] \rightarrow [[\text{Elem}]] \) that produces the unique DRSP, given a list of elements; a definition of such a function will be given later.

The following property notes that a DRSP has a “rotational symmetry”: to visualise this, turn Figure 5 upside-down, and it still depicts a DRSP:

**Theorem 4.10 (DRSP rotation)**

For any non-empty list of elements \( x \),

\[
\text{drsp}_{\leq} d x = \text{reverse} \cdot \text{map reverse} \cdot \text{drsp}_{\geq} d (\text{reverse} \cdot x).
\]

The above property is not needed for the algorithmic development, but it is included here for completeness.

The final property of DRSPs concerns how they can be constructed. We will need some preliminaries. Firstly, the longest right-skew prefix of a list of elements is also the longest densest prefix of the list:

**Lemma 4.11**

Let \( y \) be a non-empty list of elements. Then the longest right-skew prefix of \( y \) is also the longest of the maximally dense prefixes of \( y \).

It is the above lemma that illustrates how the partition of the right-hand side of the window \( x = \text{concat} \left[ y_0, y_1, \ldots, y_k \right] \), as suggested in the previous section, results in a DRSP. Recall that the segment \( y_0 \) is selected as the longest right-skew prefix of \( x \), and thus, by the above lemma, it must also be the longest densest prefix of \( x \). This ensures that the next segment \( y_1 \) must be of strictly lower density than \( y_0 \) (otherwise \( y_0 \) \( +_d \) \( y_1 \) would be at least as dense as \( y_0 \), contradicting \( y_0 \) being the longest of the maximally dense prefixes of \( x \)). Thus \( y_0 >_d y_1 \), and repeatedly selecting longest right-skew segments results in a partition of right-skew segments of decreasing densities.

The above describes one of several possible methods of constructing decreasing right-skew partitions:
Theorem 4.12 (DRSP construction)
The DRSP of a list of elements can be constructed by:

(i) repeatedly taking longest maximum-density prefixes
(ii) repeatedly taking longest right-skew prefixes\(^2\)
(iii) repeatedly taking longest minimum-density suffixes\(^3\)
(iv) repeatedly taking longest right-skew suffixes

The following function illustrates one of the above ways (i) to build a DRSP:

\[
\text{drsp1} :: [\text{Elem}] \rightarrow [[[\text{Elem}]])
\]
\[
\text{drsp1} [] = []
\]
\[
\text{drsp1} x = y : \text{drsp1} (\text{drop} \text{ length} y) x
\]
\[
\text{where } y = \text{max}_d \text{ (prefixes } x)\).
\]

However, for the representation of the window in our algorithm, we are not usually going
to be constructing a complete DRSP from a list of elements; the window will already con-
sist of a header and the DRSP of the remaining elements, and we will want to incrementally
update the partition as new elements are added.

The following lemma allows a DRSP to be updated from the left-hand side:

Lemma 4.13
Let \( z \) be a right-skew list of elements, and \( ys \) a DRSP. Then \( \text{prepend} z \ ys \) is a DRSP, where

\[
\text{prepend} :: [\text{Elem}] \rightarrow [[[\text{Elem}]]) \rightarrow [[[\text{Elem}]])
\]
\[
\text{prepend} [z] = [z]
\]
\[
\text{prepend} z (y : ys) = \text{if } z \leq_d y \text{ then } \text{prepend} (z ++ y) \ ys
\]
\[
\text{else } z : y : ys.
\]

Above, the \( \text{prepend} \) function repeatedly joins the list of elements \( z \) with the leftmost
segment of \( ys \) (justified by Lemma 4.3), until the densities are once more decreasing.

This leads to the following alternative way to build a DRSP, using a function \( \text{addl} \) that
appends one element to the left of an existing DRSP:

\[
\text{drsp} :: [\text{Elem}] \rightarrow [[[\text{Elem}]])
\]
\[
\text{drsp} = \text{foldr addl} [\].
\]
\[
\text{addl} :: \text{Elem} \rightarrow [[[\text{Elem}]]) \rightarrow [[[\text{Elem}]])
\]
\[
\text{addl} a xs = \text{prepend} [a] xs.
\]

Note that the right-skew property that \( \text{prepend} \) requires of its first argument applies, as any
singleton list \([a]\) is right-skew. The function \( \text{addl} \) will be useful in our final code.

4.5 Use of the DRSP
To recap where we have reached in the development: given a list of elements \( w' = a : w \) for
which we want to calculate \( mdp \ w' \), we split the window \( w' \) using \( \text{hsplit} \ w' = (h,x) \), and

\(^2\) A longest right-skew prefixes approach is used in (Goldwasser et al., 2005) and (Lin et al., 2002).
\(^3\) The algorithm in (Chung & Lu, 2004) slides the window from left to right, using a mirror-image
DRSP structure, the construction of which is based on taking longest minimum-density prefixes.
Functional Pearl: Finding A Densest Segment

represent the list \( x \) by its DRSP. The data structure for the window is therefore:

\[
\text{type } \text{Window} = ([\text{Elem}],[[\text{Elem}]]),
\]

where the first component is the compulsory header \( h \), and the second is the rest of the window, partitioned into \( drsp \ \cdot \ x \). Furthermore, Theorem 4.6 established that we don’t need to consider prefixes that end in the interior of right-skew segments in the partition of \( x \). Please note: the lists in the \( \text{Window} \) datatype should be considered as abstract representations at this stage, as we have not yet finalised whether to use cons-lists, snoc-lists, or some other queue representation, as befits a sliding window algorithm. We will make these decisions in Section 5.2, to allow the final algorithm to run in linear time.

Having chosen a data structure for the window, we will need some representation-changing functions to convert between a list of elements and its partition into a DRSP:

\[
\begin{align*}
\text{wbuild} & \quad :: \ [\text{Elem}] \rightarrow \text{Window} \\
\text{wbuild} &= (\text{id} \times \text{drsp}) \cdot \text{hsplit}, \\
\text{wflatten} & \quad :: \ \text{Window} \rightarrow [\text{Elem}] \\
\text{wflatten} (h, xs) &= h ++ \text{concat} \ \cdot \ \text{xs}.
\end{align*}
\]

Above, the function \( \text{wbuild} \) constructs a window from a list of elements\(^4\), while \( \text{wflatten} \) does the opposite, flattening a window back to a list. Thus we have \( \text{wflatten} \cdot \text{wbuild} = \text{id} \).

To work on a more abstract level, we define a function \( \text{wcons} \), so that \( \text{wcons} \ a \) adds an element \( a \) to the left side of the window. It can be seen as \( (a : \cdot) \) lifted to the \( \text{Window} \) datatype, although it performs much more work — repartitioning the header, and updating the DRSP:

\[
\begin{align*}
\text{wcons} & \quad :: \ \text{Elem} \rightarrow \text{Window} \rightarrow \text{Window} \\
\text{wcons} \ a (h, xs) &= (h', \text{foldr addl} \ \cdot \ x x) \\
\text{where} \ (h', x) &= \text{hsplit} \ (a : h).
\end{align*}
\]

We have made progress: the data structure for the window has been chosen, we can construct and update a DRSP by adding elements incrementally to the front of it. Now we just need the \( mdp \) of the window.

### 4.6 Extracting the Densest Prefix

To find the densest prefix with respect to \( \downarrow_{d} \), we need to make use of Theorem 4.6, which guarantees that for a partition of the elements beyond the compulsory header into right-skew segments (such as the DRSP we are using), only the positions of the input corresponding to the ends of these segments need to be considered.

Furthermore, the decreasing densities of the DRSP will enable us to pinpoint exactly where the \( mdp \) of the window elements is to be found, as follows. Empirical examination suggests that the densities of the prefixes \( h \), then \( h \uparrow \uparrow y_0 \) and so on up to \( h \uparrow \uparrow y_0 \uparrow \cdots \uparrow \uparrow y_k \) form a simple hill shape, illustrated by our running example in Figure 6. To be precise: the hill for the window prefix densities of a header & DRSP is bitonic, consisting of a strictly increasing ascent on the left, followed by a (non-strictly) decreasing descent. Either side

\[4\] The “product functor” \( (\times) \) is defined by \( (f \times g) \ (a,b) = (f \ a, g \ b) \).
of the hill can be empty, for example it might just be a gentle decreasing slope. For the moment, please assume that our assertion about this hill shape is true, and we will soon show this formally.

From this hill shape, it is now easy to identify the mdp: the densest window prefix(es) can be found at the top of the hill, and in particular, the densest prefix wrt $\nmid_d$ (which is what mdp uses) will be at the top of the hill, at the left-hand end if it turns out that the top of the hill is a plateau (i.e. more than one prefix having maximal density).

Finding the top of the hill can be achieved by a simple traverse from either the left or right side of the hill; however it will be seen later that starting from the right will have a crucial effect on the overall efficiency of the algorithm, enabling each element to be passed by once only (more details are given in Section 5.2). Here is the definition of a function that, given a header $h$ and a DRSP, chops off segments of the DRSP from the right-hand side, stopping at the left of the top of the hill:

$$
\text{maxchop} :: [\text{Elem}] \rightarrow [[\text{Elem}]] \rightarrow [[\text{Elem}]]
$$

$$
\text{maxchop } h \ [\ ] = [\ ]
$$

$$
\text{maxchop } h \ (\text{xs} \ ++ \ [x]) = \text{if } h \ ++ \ \text{concat} \ \text{xs} \ <_d \ x \ \text{then } \text{xs} \ ++ \ [x] \ \text{else } \text{maxchop} \ h \ \text{xs}.
$$

The maxchop function repeatedly removes a segment $x$ when $h \ ++ \ \text{concat} \ \text{xs} \ <_d \ x$, which from the density property is equivalent to $h \ ++ \ \text{concat} \ \text{xs} \ \geq_d h \ ++ \ \text{concat} \ (\text{xs} \ ++ \ [x])$. In other words, maxchop chops from the right whilst going up the right-hand side of the hill, only stopping if it runs out of hill (the $[]$ case) or if it finds a strict decrease in density, as the given condition $h \ ++ \ \text{concat} \ \text{xs} \ <_d \ x$ is equivalent to $h \ ++ \ \text{concat} \ \text{xs} \ <_d h \ ++ \ \text{concat} \ (\text{xs} \ ++ \ [x])$, from the density property. To illustrate, maxchop applied to our running MDS example in Figure 6 returns the list of segments $[[(23, 2), (-3, 1), (61, 3), (12, 2), (69, 4)], [(43, 4), (88, 6)], [(97, 10), (82, 5), (77, 5)]]$.

The following lemma shows that the list of segments that maxchop returns is strictly ascending in density when prepended by the header $h$, and thus shows formally that we really do have a simple bitonic hill shape as described earlier.

**Lemma 4.14**

Let $h$ be a non-empty list of elements, and $\text{xs} \ ++ \ [x]$ be a list of non-empty segments with strictly decreasing densities. If $h \ ++ \ \text{concat} \ \text{xs} \ <_d x$, then the following list of segments has
strictly increasing densities:

\[
\text{map } ((h++) \cdot \text{concat}) (\text{prefixes } (xs ++ [x])).
\]

We can then use the above lemma to show that \text{maxchop} does indeed compute the \text{mdp} for us, which is what we require:

**Theorem 4.15**

Let \( h \) a non-empty list of elements, and let \( xs \) be a (possibly empty) list of non-empty segments having strictly decreasing densities. Then

\[
(h++) \cdot \text{concat} (\text{maxchop } h \ xs) = \max_{\downarrow} \text{map } ((h++) \cdot \text{concat}) (\text{prefixes } xs).
\]

Later, in our proofs, we will actually use the following corollary, which is the same property stated differently:

**Corollary 4.16**

Let \( h \) a non-empty list of elements, and let \( xs \) be a (possibly empty) list of non-empty segments having strictly decreasing densities. Then

\[
\text{maxchop } h \ xs = \max_{h++} (\text{prefixes } xs),
\]

For convenience, we also define a wrapper function for \text{maxchop} that operates on the whole window:

\[
\text{wmaxchop} :: \text{Window} \rightarrow \text{Window}
\]

\[
\text{wmaxchop } (h, xs) = (h, \text{maxchop } h \ xs).
\]

### 5 Putting Everything Together

Let us remind ourselves of the development so far, the outline of which is summarized in Figure 7.

---

<table>
<thead>
<tr>
<th>( mds )</th>
<th>( ms )</th>
<th>( wp )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( mds = \max_{\downarrow} \cdot \text{segments} )</td>
<td>( ms [] = [] )</td>
<td>( wp [] = [] )</td>
</tr>
<tr>
<td>We proved that</td>
<td>( ms (a : x) = wp (a : x) \downarrow ms x )</td>
<td>( wp (a : x) = mdp (a : wp x) )</td>
</tr>
<tr>
<td>Computing ( ms ) is obtained from</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( (ms, wp) ), which can be expressed as a \text{foldr}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( mdp = \max_{\downarrow} \cdot \text{prefixes} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

\( |_{h++} \) was defined in Corollary 4.7.
In Section 2 the original specification was refined to produce the function \( mds \), then in Section 3, a sliding window algorithm \( ms \) was shown to satisfy that specification, where the value of \( ms \) is obtained from the paired computation \( \langle ms, wp \rangle \). Section 4 was devoted to finding a faster way to compute \( mdp \cdot (a : ) \) in the inductive step for \( wp \), the window-processing function, and as a result we obtained a datatype \( Window \) to use for representing \( wp x \).

In this section we will put the results of Sections 3 and 4 together to calculate our algorithm to solve the density problem. We will then refine the data structure in Section 5.2 to allow efficient implementation of certain operations. For now, using a standard list data structure allows us to see what is going on in the algorithm more easily.

### 5.1 Structuring the Window

Our algorithm for obtaining a maximally dense segment is \( ms = \text{fst} \cdot \langle ms, wp \rangle \), from which we can introduce the window structure as follows:

\[
\text{fst} \cdot \langle ms, wp \rangle = \{ \text{window identity, see page 19} \}
\]

\[
\text{fst} \cdot \langle ms, \text{wflatten} \cdot \text{wbuild} \cdot wp \rangle = \{ \text{identity, composition} \}
\]

\[
\text{fst} \cdot (\text{id} \times \text{wflatten}) \cdot \langle ms, \text{wbuild} \cdot wp \rangle = \{ \text{projections} \}
\]

\[
\text{fst} \cdot \langle ms, \text{wbuild} \cdot wp \rangle.
\]

Our task is now to push \( \text{wbuild} \) into the calculation of \( wp \) so that all the window processing is done with the \( Window \) datatype. Thus we define

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

\[
mwp = \langle ms, \text{wbuild} \cdot wp \rangle,
\]

and we will aim to calculate an inductive definition of \( mwp \).

The base case is straightforward: \( mwp [] = ([], ([], [])) \). For the inductive case we calculate:

\[
mwp (a : x)
\]

\[
= \{ \text{definitions of } ms \text{ and } wp \}
\]

\[
(mdp (a : wp x) \preceq ms x, \text{wbuild} (mdp (a : wp x)))
\]

\[
= \{ \text{since } \text{wflatten} \cdot \text{wbuild} = \text{id} \}
\]

\[
(\text{wflatten } u \preceq ms x, u)
\]

where \( u = \text{wbuild} (mdp (a : wp x)) \).

Consider the above binding \( u = \text{wbuild} (mdp (a : wp x)) \). If we can somehow manage to push \( \text{wbuild} \) to the right and show that \( u = f (\text{wbuild} (wp x)) \) for some \( f \), we will then have an inductive definition for \( mwp \). To do this, since \( \text{wbuild} = (\text{id} \times \text{drsp}) \cdot \text{hsplit} \), it will be helpful to know how \( \text{hsplit} \) interacts with \( (a :) \) and \( mdp \). Firstly, the following lemma shows how \( \text{hsplit} \) and \( (a :) \) exchange:
Lemma 5.1
For any element \( a \) and list of elements \( z \),
\[
hsplit (a : z) = (h', z_1 ++ z_2)
\]
\[\text{where } (h,x_2) = hsplt z \]
\[ (h',z_1) = hsplt (a : h). \]

Secondly, this is how \( hsplt \) interacts with \( mdp \):

Lemma 5.2
For any list of elements \( x \),
\[
hsplt (mdp x) = (h, \max \, \downarrow h (prefixes y))
\]
\[\text{where } (h,y) = hsplt x. \]

The above lemma simply restates the left-hand side involving the input list of elements \( x \), into a form relating to the window data structure, with its compulsory header \( h \). Recall that \( mdp \) chooses an optimal prefix with respect to \( \preceq \). If \( x \) is not wide enough, then both \( y \) and \( mdp x \) evaluate to [ ], and \( h = x \). Otherwise, the right-hand side says that comparison with respect to \( \preceq \) is carried out by comparing densities using \( \preceq_h \) on prefixes within the window structure.

We are now ready to transform \( \text{wbuild} (mdp (a : wp x)) \). Writing \( z \) for \( wp x \), we calculate:
\[
\text{wbuild} (mdp (a : z)) = \{
\text{definition of wbuild } \}
(id \times \text{drsp}) (hsplt (a : z))
= \{
\text{property of hsplt in Lemma 5.2, letting } (h,z') = hsplt (a : z) \}
(id \times \text{drsp}) (h, \max \downarrow h (prefixes z'))
= \{
\text{property of hsplt Lemma 5.1, letting } (h',z_2) = hsplt z \text{ and } (h,z_1) = hsplt (a : h') \}
(id \times \text{drsp}) (h, \max \downarrow h (prefixes (z_1 ++ z_2)))
= \{
\text{since } \text{concat} \cdot \text{drsp} = id \}
(h, \text{drsp} (\max \downarrow h (prefixes (\text{concat} (\text{drsp} (z_1 ++ z_2)))))\)\)
= \{
\text{Corollary 4.7 } \}
(h, \text{drsp} (\text{concat} (\max \downarrow h (prefixes (\text{drsp} (z_1 ++ z_2)))))\))
= \{
\text{drsp} (z_1 ++ z_2) = \text{foldr addl} (\text{drsp} z_2) z_1, \text{ from the definition of drsp in Section 4.4 } \}
(h, \text{drsp} (\text{concat} (\max \downarrow h (prefixes (\text{foldr addl} (\text{drsp} z_2) z_1)))))\))
= \{
\text{definition of wbuild (see Section 4.5), letting } (h',z_2) = \text{wbuild} z \}
(h, \text{drsp} (\text{concat} (\max \downarrow h (prefixes (\text{foldr addl} z_2 z_1)))))\))
= \{
\text{definition of wcons, letting } (h,z_2) = \text{wcons} a (\text{wbuild} z) \}
(h, \text{drsp} (\text{concat} (\max \downarrow h (prefixes z)))\))
= \{
\text{Theorem 4.16 } \}
\]

\( _h \) was defined in Corollary 4.7
\[(h, hsp (concat (maxchop h zs)))\]  
\[= \{ \text{cancellation, as } zs \text{ is a DRSP (see below)} \}\]  
\[(h, maxchop h zs)\]  
\[= \{ \text{definition of } wmaxchop \text{ (see end of Section 4.5)} \}\]  
\[(wmaxchop \cdot wcons a \cdot wbuild) z.\]

The cancellation in the penultimate step is valid because of the following: \(wbuild\) builds a DRSP, and \(wcons\) and \(maxchop\) return DRSPs when given a DRSP as input. Then, as DRSPs are unique, applying \(drsp \cdot concat\) to a DRSP has no effect.

In summary, we have shown above that

\[wbuild \cdot mdp \cdot (a : x) = wmaxchop \cdot wcons a \cdot wbuild. \quad (27)\]

We resume the calculation of the inductive case for \(mwp\):

\[mwp (a : x)\]  
\[= \{ \text{by previous calculation and (27)} \}\]  
\[= (wflatten u \downarrow ms x, u)\]  
\[\text{where } u = wmaxchop (wcons a (wbuild (wp x)))\]  
\[= \{ \text{adding variables } m \text{ and } w \}\]  
\[= (wflatten u \downarrow m, u)\]  
\[\text{where } m = ms x\]  
\[w = wbuild (wp x)\]  
\[u = wmaxchop (wcons a w)\]  
\[= \{ \text{definition of } mwp \}\]  
\[= (wflatten u \downarrow m, u)\]  
\[\text{where } (m, w) = mwp x\]  
\[u = wmaxchop (wcons a w).\]

We have thus constructed an inductive alternative definition for \(mwp\):

\[
mwp :: [\text{Elem}] \rightarrow ([\text{Elem}], \text{Window})
\]
\[
mwp [] = ([], [[], []])
\]
\[
mwp (a : x) = (wflatten u \downarrow m, u)
\]
\[
\text{where } (m, w) = mwp x
\]
\[
u = wmaxchop (wcons a w).
\]

As a maximally dense segment is obtained from \(fst \cdot mwp\), we have arrived at our algorithm, which is summarised in the following section.

### 5.2 Data Refinement and Performance Analysis

In this section we perform some final data structure refinement, and show why the final algorithm runs in linear time (and space). For the reader’s convenience, Figure 8 summarizes the derived program, with an implementation of the function \textit{hsplit}. Here, the type \textit{Window}
Functional Pearl: Finding A Densest Segment

is given more abstractly as

```
type Window = (Header Elem, Parts (Seg Elem))
```

The types `Header`, `Parts`, and `Seg` are respectively the datatypes with which we implement the header, the partition, and each segment in the partition. They were all treated abstractly as lists in previous sections, and they remain to be list-like data structures in the final implementation. For clarity we overload list constants and operations such as `[]`, `(,)`, and `(+ +)` and use pattern matching in the code in Figure 8.

```
ms :: [Elem] → [Elem]
ms = fst · mwp

type Window = (Header Elem, Parts (Seg Elem))

mwp :: [Elem] → ([Elem], Window)
mwp [] = (([],[]))
mwp (a : x) = (wflatten u ≺ m, u)
  where (m, w) = mwp x
        u = wmaxchop (wcons a w)

hsplit :: Header Elem → (Header Elem, [Elem])
hsplit x = split (x, [])

split :: (Header Elem, [Elem]) → (Header Elem, [Elem])
split ([], y) = (([], y))
split (x ++ [a], y) = if width x < L then (x ++ [a], y)
                      else split (x, a : y)

wcons :: Elem → Window → Window
wcons a (h, xs) = (h', foldr addl xs x)
  where (h', x) = hsplit (a : h)

addl :: Elem → Parts (Seg Elem) → Parts (Seg Elem)
addl a xs = prepend [a] xs

prepend :: Seg Elem → Parts (Seg Elem) → Parts (Seg Elem)
prepend x [] = [x]
prepend x (y : xs) = if x ≤ₗ y then prepend (x ++ y) xs
                      else x : y : xs

wmaxchop :: Window → Window
wmaxchop (h, xs) = (h, maxchop h xs)

maxchop :: Header Elem → Parts (Seg Elem) → Parts (Seg Elem)
maxchop h [] = []
maxchop h (xs ++ [x]) = if (h ++ concat xs) ≺ₗ x then xs ++ [x]
                        else maxchop h xs
```

Fig. 8. The derived algorithm.
For all of the lists of elements represented using the datatypes Header, Parts, and Seg, we need to be able to compute their density and width in constant time. This can easily be done by pairing the lists with their current values for weight, length, width, etc., and is considered a separate issue.

The data structure used for the Header datatype depends on the implementation of the hsplit function, as within the algorithm, the header is only altered using hsplit. One possible implementation makes use of an auxiliary function split, which repeatedly removes the rightmost element of the header until it does not exceed the width limit, while retaining as much width as possible. Since each element is initially added to the header on the left and later removed from the right, at most once, hsplit runs in linear time if addition on the left and removal from the right are both amortised constant time operations. This can be done by implementing Header as a simple queue using two lists.

For the Parts and Seg datatypes, we need to examine the operations on the DRSP structure of the window. Elements are added one-by-one on the left by addl, which simply launches prepend. The function prepend makes an indefinite number of recursive calls, and in each call two segments are joined. However, notice that segments in the DRSP are all non-empty, and are never split once joined. Therefore, given an input of length n, there can be at most $O(n)$ joins in total. The function prepend thus runs in linear time, provided that Parts allows addition to and removal from the left in amortised constant time, and that Seg supports list concatenation in (amortised) constant time. The requirement on Seg is easy to fulfill: one may simply use a join list: `data Seg a = Single a | Join (Seg a) (Seg a)`.

The removal of elements from the window is carried out by the maxchop function, which removes segments in Parts from the right. Therefore we need Parts to support amortised constant-time removal from both ends. A number of data structures support such operations, for example, Banker's deques (Okasaki, 1999), or 2-3 finger trees (Hinze & Paterson, 2006).

6 Conclusions

We have derived a linear-time algorithm for solving the generalised segment density problem with a lower bound on segment width. The algorithm scans through the input list using a sliding window, which is split into a header and a partition of right-skew segments with decreasing densities (DRSP), whose properties we exploit to make the linear-time processing possible. While the program itself barely occupies one page, its proof is anything but simple, involving the discovery of intricate properties, and our complete proof of the algorithm uses approximately 3500 lines of Agda code.

Two instances of the density segment problem were presented. While the maximum density segment (MDS) problem has a long history, we believe that the maximum mean segment (MMS) problem is new. These are similar but neither is a generalisation of the other: in MDS, the density function is fundamentally linked to the segment width, but this is not the case for the MMS problem. This shows that, for the densest segment problem, width can be separate from density, allowing the algorithm to analyse data such as blood glucose measurements.

The development of our solution has not been straightforward, and we faced a number of design decisions from the beginning. Should we model the problem functionally, rela-
Functionally, or use total relations? As there were technical complications involving relations and zygomorphisms (see the technical aside in Section 3.2), a functional approach was strongly indicated. Another decision concerned the numerous maximum operators used, where we had to decide which segment to return in case of a tie: shall we use $\uparrow$ or $\uparrow$, and does it matter? It turns out that the choice does matter — the algorithm does not meet that exact specification had we chosen otherwise. Furthermore, the algorithm might not yield a result when there is no segment within the width bound, but using a Maybe type made the reasoning rather cumbersome. Using an extended ordering $\preceq$ allows a much cleaner formulation.

While we had an informal understanding of why this algorithm is correct, formally writing down the properties that make it so turned out to be surprisingly tricky. The correctness of the “outer” algorithm (Section 3.4), treated casually in all previous works, took us a considerable amount of time to formalise. We attempted to come up with a more declarative specification of the prefix returned by $wp$ (for example, we guessed that it is the shortest prefix satisfying certain properties), but none of those specifications were correct. Various possibilities were tried before we reached Lemma 3.6 and its supporting definitions. Afterwards, the proof quickly followed. It is often the case that finding the right thing to prove is harder than producing the proof. It was surprising that in Lemma 3.6 we could prove an equality, rather than merely that the two sides yield segments having the same density. In fact, we needed an equality for the inductive proof to work.

All this hard work was not spent in vain. We re-proved the uniqueness of the DRSP, and presented various properties about it, including how it can be constructed. This fills in more properties about the DRSP than given in previous papers.

We initially set out to solve a more general version of the MDS problem, with an additional upper bound on segment width, making the problem even more intricate. As mentioned in Section 1, an algorithm with a wrong time analysis has been published before. Even the algorithms of Chung and Lu (Chung & Lu, 2004) and Goldwasser (Goldwasser et al., 2005) are not entirely correct: they both fail for a boundary case. The former could potentially return an invalid result, for which there is an easy fix (Chung, 2010), while the latter loops and it is harder to see whether it is fixable.

We believe that the difficulty in developing correct linear-time algorithms is partly due to the complicated nature of the MDS problem, and partly due to the absence of a rigorous approach to program construction. The imperative algorithms of Chung & Lu and Goldwasser both maintain invariants that are neither explicitly stated nor easy to reconstruct. The invariants rely on states stored in static variables surviving between subroutine calls, which makes reasoning about them extremely hard. In addition, their liberal use of array indices 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. We have developed data structures that are used in the sliding window to produce densest segments with an upper bound on their width, and we have some preliminary results on correctness proofs, which will have to be deferred to another paper.

---

7 Namely, for the case when there is no segment in the window whose width is within the bounds.
Acknowledgments

We would like to thank Richard Bird for stimulating discussions of the MDS problem.

References


