Queueing and Glueing for Optimal Partitioning

Functional Pearl

Shin-Cheng Mu
Institute of Information Science, Academia Sinica
scm@iis.sinica.edu.tw

Yu-Hsi Chiang
Dep. of Computer Science and Information Engineering, National Taiwan University
yuhsichiang@gmail.com

Yu-Han Lyu
Dep. of Computer Science, Dartmouth College
yuhanlyu@gmail.com

Abstract
The queueing-glueing algorithm is the nickname we give to an algorithmic pattern that provides amortised linear time solutions to a number of optimal list partition problems that have a peculiar property: at various moments we know that two of three candidate solutions could be optimal. The algorithm works by keeping a queue of lists, glueing them from one end, while chopping from the other end, hence the name. We give a formal derivation of the algorithm, and demonstrate it with several non-trivial examples.

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

Keywords
program derivation, optimal partition

1. Introduction
Consider an algorithm that is divided into a certain number of big steps, where each individual step might make an indefinite number of calls to several basic, constant-time operations. If each of these operations could be applied at most \(O(n)\) times (\(n\) being the size of the input), before which the algorithm must terminate, the algorithm runs in linear time. An example of such an amortising algorithm was presented by Chung and Lu (2004) and Goldwasser et al. (2005) to solve the maximum-density segment problem — given a list of numbers, to compute a segment (a consecutive sublist) of the input that has the largest average. A queue of lists is kept throughout the \(n\) steps of the algorithm. In each step, an indefinite number of lists at one end of the queue are glued together, before some lists at the other end are dropped. The algorithm has a linear time-bound because each of these operations could happen at most a linear number of times.

The algorithm of Chung, Lu, and Goldwasser et al. is not yet applied to solve many other segment problems — as we will explain later, many problems that do satisfy the precondition of their algorithm have simpler solutions. Nevertheless, it turns out that the technique can be applied to solve a number of optimal partition problems — to break an input list into a list of lists that optimises a certain, sometimes complex, cost function. The technique was reinvented a number of times, for example, by Brucker (1995) and Hirschberg and Larmore (1987). Curiously, this technique is not well-known outside the algorithm community.

We think that this pattern, which we nickname the queueing-glueing algorithm, is very elegant, and would like to give it a formal treatment in this pearl.

Example Problems
By the end of this pearl we will demonstrate the algorithm with several problems. The first, one-machine batching, is proposed by Brucker (1995). A list of jobs, each associated with a processing time and a weight, are to be processed on a machine in batches. The goal is to partition the jobs into batches while minimising the total cost. The cost of a job is the absolute finishing time of its batch (thus all jobs in a batch is considered to finish at the same time), multiplied by sum of weights in the batch. Furthermore, each batch incurs a fixed starting time overhead. For an example, consider the jobs with processing time \([2, 2, 1, 5, 3, 2]\), whose weights are all 1, and let the starting overhead be 2. The best way to partition the jobs is in three batches

\[ [[2, 2, 1], [5, 3], [2]] , \]

the first batch finishes at time 2 + (2 + 2 + 1) = 7, the second batch at 7 + 2 + (5 + 3) = 17, and the third batch at 21. The total cost is 7 + 17 + 2 + 21 = 76. To see how a little change of the input could alter the scheduling, notice that, without the last job, it would be preferable to partition the jobs \([2, 2, 1, 5, 3]\) into \([2, 2, 1, [5, 3]]\) (cost: 54) rather than \([2, 2, 1, [5, 3]]\) (cost: 55).

In the second problem, size-specific partitioning, the goal is to partition a given list of positive numbers such that the sum of each segment is as close to a given constant \(L\) as possible. The closeness of a segment to \(L\), however, is measured by the square of difference between \(L\) and the sum of numbers in a partition, and the closeness of a partition is the sum of the closeness of its segments. After solving the problem we will discuss what modifications are needed to use the algorithm to solve the paragraph formatting problem. The goal is similar: to break a piece of text into lines such that the length of each line is close to \(L\), using the same measurement. The difference is that the last line does not count. While it is known that we can format a paragraph in linear time, it is surprising that our algorithm also works for this case.

Outline
In Section 2 and 3 we develop the skeleton of the algorithm and ensure that the main computation can be performed in a fold-like manner. To refine the steps in the fold, we discuss in Section 4 the two-in-three property, a main feature of the type of problems we solve. The queueing-glueing operations, key of this algorithm, are developed in Section 5. As a warning, we will see some modest use of relations in one stage of the development,
which makes a key theorem much easier to prove. The sample problems are solved in Section 6. Programs accompanying this pearl are available at https://github.com/acma/queueing-glueing.

2. From Optimal Segments to Partitions

While the algorithm to be developed in this pearl aims to compute optimal partitions, it is based on computation of optimal segments. We will therefore start with discussing both scenarios. The function computing a non-empty optimal segment of the input, a list of some type E, can be specified by

\( \text{optseg} :: [E] \rightarrow [E] \)
\( \text{optseg} = \text{minBy w \cdot segs} \),

where \( \text{minBy w} :: [[E]] \rightarrow [E] \) computes the minimum with respect to a cost function \( w :: [[E]] \rightarrow \mathbb{R} \). The function \( \text{segs} :: [a] \rightarrow [[a]] \), computing all non-empty segments of its input, can be defined by

\[ \text{segs} = \text{concat} \cdot \text{map} \cdot \text{prefixes} \cdot \text{suffs} \]

where \( \text{prefixes} \) and \( \text{suffs} \) respectively return all the non-empty prefixes and suffixes. The definition of \( \text{segs} \) is given below:

\[ \text{prefixes} :: [a] \rightarrow [[a]] \]
\[ \text{prefixes} [] = [] \]
\[ \text{prefixes} (x:xs) = [x] \cdot \text{map} (\lambda y. \text{prefixes} (xs)) \].

Standard calculation yields that

\[ \text{optseg} = \{ \text{definitions of optseg and segs} \} \]
\[ \text{minBy w \cdot concat \cdot map \cdot prefixes \cdot suffs} = \{ \text{since minBy w \cdot concat = minBy w \cdot map (minBy w)} \} \]
\[ \text{minBy w \cdot map \cdot (\text{minBy w \cdot suffs})} \]

The part \( \text{map (minBy w \cdot prefixes) \cdot suffs} \) computes a list of optimal prefixes, one for each suffix, from which an optimal one is chosen. The \text{scan lemma} then states that, if \( \text{minBy w \cdot prefixes} \) can be computed in a \text{foldr}, \text{optseg} can be computed in a \text{scanr}. Operationally speaking, we compute the optimal prefix for each suffix and cache them in a list, such that each optimal prefix can be computed from the previous one. This is essentially how the classical maximum segment sum problem is solved (Bird 1989).

Optimal partitions can be computed by a similar principle. A list \( \text{split} :: [a] \) is a partition of \( \text{split} :: [a] \) if \( \text{concat} \; \text{split} \; \text{is} \; \text{xs} \) and all elements in \( \text{split} \) are non-empty. We also call each element in \( \text{split} \) a “segment” of the partition. The following function \( \text{parts} \) computes all partitions of its input:

\[ \text{parts} :: [a] \rightarrow [[[a]]] \]
\[ \text{parts} [] = [[]] \]
\[ \text{parts} (x:xs) = (\text{concat} \cdot \text{map} \cdot \text{extparts} \cdot \text{splits}) \; \text{xs} \],

where \( \text{extparts} :: (x, y) = \text{map} (\text{split}) \cdot (\text{parts} \cdot y) \). The function \( \text{splits} \) returns all the \( (y, z) \) where \( y \) is non-empty and \( y + z = \text{x} = \text{xs} \):

\[ \text{splits} :: [a] \rightarrow [[[a], [a]]] \]
\[ \text{splits} [] = [[]] \]
\[ \text{splits} (x:xs) = ([x], x) \cdot \text{map} (\lambda (x, y) \cdot (x \times \text{id}) \cdot \text{split} \cdot \text{xs}) \],

where \( f \times g :: (x, y) = (f \times x, y) \). For example, \( \text{splits} [1, 2, 3] = \) \([[[[1, [2, 3]], [[1, 2, 3]], [[1, 2, 3]]], [[[1, 2, 3]]]]\) and \( \text{parts} [1, 2, 3] \) yields \( \) \([[[[1, 2, 3]], [[1, 2, 3]], [[1, 2, 3]], [[1, 2, 3]]]]\).

Notice that the definition of \( \text{splits} \) is rather similar to that of \( \text{prefixes} \). From now on, a pair of lists \( ([E], [E]) \) will be called a \text{split}.

Some cost functions will be defined on splits rather than segments.

The function computing optimal partitions can be specified by:

\( \text{optpart} :: [E] \rightarrow [[E]] \)
\( \text{optpart} = \text{minBy f \cdot parts} \).

In a simpler scenario, the cost of a partition is the sum of the costs of its parts, that is, \( f :: [[E]] \rightarrow \mathbb{R} \) has the form \( f = \text{sum} \cdot \text{map} \) for some \( w :: [E] \rightarrow \mathbb{R} \). In some applications, \( w \) needs to take the rest of the input into consideration. For such cases we let \( w \) be defined on splits, that is, \( w :: ([E], [E]) \rightarrow \mathbb{R} \), and let \( f \) be

\[ f [] = 0 \]
\[ f (xs : xss) = w (xs, \text{concat} xss) + f xss \].

Either way, the definition allows us to distribute \( (xx) \) into \( \text{minBy} \):

\[ \text{minBy f \cdot map} (\text{split}) : (xx) \cdot \text{minBy} f \]

To find out how to compute \( \text{optpart} \), we calculate, for non-empty \( xs \):

\( (\text{minBy f \cdot parts}) \; xx = (\text{minBy f \cdot concat} \cdot \text{extparts} \cdot \text{splits}) \; xx \)
\( = (\text{since minBy f \cdot concat} = (\text{minBy f \cdot map (minBy f)}) \}
\( = (\text{minBy f \cdot map} \cdot \text{map} \cdot \text{minBy f \cdot extparts} \cdot \text{splits}) \; xx \)
\( = (\text{by (1) and definition of optpart}) \}
\( = (\text{minBy f \cdot map (\lambda (ys, zs) \rightarrow ys \cdot \text{optpart zs}) \cdot splits}) \; xx \)
\( = (\text{introduce } g \text{ using (2), see below}) \}
\( = ((\lambda (ys, zs) \rightarrow ys \cdot \text{optpart zs}) \cdot \text{minBy g \cdot splits}) \; xx \).

In the last step we use the property that for all \( h \) and \( k \),

\[ \text{minBy h \cdot map} k = k \cdot \text{minBy} (h \cdot k) \]

The new cost function \( g \), now defined on splits, is therefore:

\( g :: ([E], [E]) \rightarrow \mathbb{R} \)
\( g (ys, zs) = f (ys \cdot \text{optpart zs}) \)
\( = w (ys, zs) + f (\text{optpart zs}) \),

assuming that \( w \) is defined on splits. We have thus derived:

\( \text{optpart} [] = [] \)
\( \text{optpart} xx = ys \cdot \text{optpart zs} \)
\( \text{where} (ys, zs) = \text{minBy g \cdot splits} \; xx \)

The specification of \( \text{optpart} \) above says that an optimal partition can be built segment-by-segment, each segment being from the presently best split with respect to \( g \). The cost function \( g \) helps to pick the best split \( (ys, zs) \), assuming that we already know how to optimally partition \( zs \).

Comparing with ordinary optimisation problems, one interesting feature of \( \text{optpart} \) above is that \( g \) refers to \( \text{optpart} \) itself, which we certainly want to avoid recomputing. One of the ways to avoid recomputation is to build an array that caches the values of \( \text{optpart} \) for all suffixes of the input.

From now on we denote the input by \( \text{inp} \). Define such an array \( \text{optArr} \) (for brevity we abbreviate the function \( \text{length} \; \# \cdot \text{map} \; \# \cdot \text{split} \):)

\( \text{optArr} = \text{array} (0, \#\text{inp}) \)
\( [(\#\text{ys}, \text{optpart} \; ys) \rightarrow ys \leftarrow (\#\text{split} \; \text{optArr} \cdot \#\text{inp})] \).

The library function \( \text{array} \) builds an immutable, lazy array, whose indices ranges between 0 and \#\text{inp}. Entries of the array are defined by the given assoc-list of indices/values, computed once, and stored. The role of the array is similar to the list in the case of optimal segment problem: to store the result of \( \text{optpart} \) for each suffix.

The \( n \)-th entry of \( \text{optArr} \) is the value of \( \text{optpart} \) on the suffix of \( \text{optArr} \) of length \( n \). We enclose both \( \text{optArr} \) and \( \text{optpart} \) as local definitions, and redefine the latter to fetch entries from the former:

\( \text{opt} :: [E] \rightarrow [[E]] \)
\( \text{opt inp} = \text{optArr} ! \#\text{inp where} \)
\( \text{optArr} = \{ \text{same as above} \} \)
\( \text{optpart} [] = [] \)
\( \text{optpart} xx = ys : \text{optArr} ! (\#\text{xs} \cdot \#ys) \).
where \( \hat{g} k y s = \text{let } z s s = \text{optArr}! (k - \# y s) \ector { w } (y s, \text{concat } z s s) + f z s s \).  
\( y s = \text{minBy} (\hat{g} \# x s) (\text{pref } x s) \).

Compare this definition of \text{oppart} with the previous one. Instead of calling itself, this \text{oppart} looks up the corresponding entry in \text{optArr}. Instead of computing an optimal split, we now compute the optimal prefix of \( x s \), under cost function \( \hat{g} \). The intended relationship between \( g \) and \( \hat{g} \) is, for all \( y s + z s \) that form a suffix of \text{inp},

\[ \hat{g} k y s \equiv g (y s, z s) \equiv (\_ + z s) = \text{inp} \land \# z s = k - \# y s, \]

\[ g (y s, z s) = \hat{g} (y s + z s) y s. \]

In \( \hat{g} \) we no longer need the suffix \( z s \), but instead use a number, the length of \( y s + z s \), to compute the correct index.

Throughout this paper we will sometimes present two versions of functions: one defined on splits, and one accepting a length. As a convention we label the latter by a circumflex (as in \( \hat{g} \)). A property proved on one usually has a counterpart for the other one.

Notice that each entry of \text{optArr} calls \text{oppart} once and accesses only those entries with indices smaller than its own. If we define

\[ \text{oppref} = \text{minBy} (\hat{g} \# x s) (\text{pref } x s), \]

each call to \text{oppart} calls \text{oppref} once. If it turns out that \text{oppref} \( (x : x s) \) can be defined in terms of \text{oppref} \( x s \), that is, \text{oppref} is a foldr, computation of entries of \text{optArr} can be scheduled such that the 0th, 1st, 2nd... entries are computed in turn, until the longest entry, the result, is ready to be fetched. The goal of the next section is to define \text{oppref} \( (x : x s) \) in terms of \text{oppref} \( x s \).

3. Computing Optimal Prefixes Inductively

We wish that the optimal prefix of \( x : x s \) can be computed solely from the optimal prefix of \( x s \). For this to be possible at all for a given cost function \( w \), it had better be the case that, for all \( x \) and \( x s \), the right end of \text{minBy} \( w (\text{pref } x s) \) does not extend further right than that of \text{minBy} \( w (\text{pref } x s) \). Most algorithms, in fact, impose a slightly stronger condition on \( w \):

\begin{definition}[Prefix Stability]
A function \( w :: [E] \rightarrow \mathbb{R} \) is prefix-stable if, for all \( x : x s \) and \( y s \),

\[
\begin{align*}
\text{w } x s & \leq \text{w } (x s + y s) \Rightarrow \\
(\forall y s: w (x s + y s) & \leq w (x s + x y s)). 
\end{align*}
\end{definition}

Prefix-stability guarantees that, if \( x s \) is no worse than \( x s + x y s \), the optimal prefix of \( x s + x y s \) need not extend further to the right than \( x s + x y s \). The suffix \( y s \) may thus be safely dropped, which allows us to compute the optimal prefix in a fold.

Prefix stability is implied by another important property, concavity, discussed a lot in algorithm community (e.g. Hirschberg and Larmore (1987), Galil and Park (1992)).

\begin{definition}[Concavity (for segments)].
A function \( w :: [E] \rightarrow \mathbb{R} \) is concave if, for all \( x s, x s, \) and \( y s \),

\[
\text{w } (w s + x s) + w (x s + y s) \leq w (w s + x s + y s) + w x s. \quad (3)
\]

To see that concavity implies prefix stability, notice that another way to write (3) is \( w (w s + x s) - w (w s + x s + y s) \leq w x s - w (x s + y s) \).

Our cost function \( g \) is defined on splits, and \( \hat{g} \), takes an additional integral argument recording lengths. We extend the notion of concavity and prefix stability to such functions:

\begin{definition}[Concavity (for splits)].
A function \( w :: ([E], [E]) \rightarrow \mathbb{R} \) is concave if, for all \( w s, x s, y s, \) and \( z s \),

\[
\begin{align*}
\text{w } (w s + x s + y s + z s) + w (x s + y s, z s) & \leq \text{w } (w s + x s + y s, z s) + w (x s + y s, z s). 
\end{align*}
\end{definition}

The first arguments to \( w \) are written in boldface font to be compared with (3)—we have merely added the suffixes accordingly.

\begin{definition}
A function \( \hat{g} :: [E] \rightarrow [E] \) is prefix-stable if, for all \( n, x s, y s \),

\[ \hat{g} n x s \leq \hat{g} n (x s + y s) \Rightarrow \\
(\forall y s: \hat{g} (n + w y s) (x s + y s) & \leq \hat{g} (n + w y s) (w s + x s + y s)). 
\]

It turns out that, with definitions of \( f, \hat{g}, \) etc. given in the previous section, concavity of \( w \) implies prefix stability of \( \hat{g} \).

\begin{theorem}
Given a cost function \( w :: ([E], [E]) \rightarrow \mathbb{R} \), and let functions \( f \), \( g \), and \( \hat{g} \) be defined as in Section 2. We have that \( \hat{g} \) is prefix-stable if \( w \) is concave.

Proof. Let \( z s \) be the suffix of the input having length \( n - \# (x s + y s) \) and thus \( y s + z s \) is the suffix having length \( n - \# x s \). We reason (abbreviating \( f \cdot \text{oppart} \text{to} \text{fo} \)):

\[
\hat{g} n x s \leq \hat{g} n (x s + y s) \\
(\forall y s: \hat{g} (n + w y s) (x s + y s) & \leq \hat{g} (n + w y s) (w s + x s + y s)). 
\]

\begin{proof}
We are now ready to derive an inductive definition of \text{oppref} \( x s \), provided that \( w \) is concave. The base case is omitted. In the calculation for the inductive case, we abbreviate \( \hat{g} \# x (x : x s) \) to \( g' \).

The operator \( \left( \_ \right)^{\#} \) denotes binary minimum with respect to \( g' \).

\[
\begin{align*}
\text{minBy} \ g' (\text{pref } x s) & = \{ \text{definitions of minBy and pref s} \} \\
\[x] \hat{g} \# \text{minBy} \ g' (\text{map } x s) (\text{pref } x s) & = \{ \text{let } x s' = \text{minBy} (\hat{g} \# x s) (\text{pref } x s) \text{ and } x s' + y s = x s \} \\
\[x] \hat{g} \# \text{minBy} \ g' (\text{map } x s) (\text{pref } x s) & = \{ \text{by Lemma 6, see below} \} \\
\[x] \hat{g} \# \text{minBy} \ g' (\text{map } x s) (\text{pref } x s) & = \{ \text{definition of minBy, pref s, and function laws} \} \\
\text{minBy} \ g' (\text{pref } x s) & = \{ \text{definition of } x s' \} \\
\text{minBy} \ g' (\text{pref } x : \text{minBy} (\hat{g} \# x s) (\text{pref } x s)) & = \{ \text{definition of } x s' \}. 
\end{align*}
\end{proof}

In the second step, we split \( x s \) into two parts \( x s' + y s \), where \( x s' \) is the optimal prefix from the previous step. Lemma 6 below then allows us to get rid of \( y s \), which will not be part of the optimal prefix of \( x : x s \).

\begin{lemma}
Let \( x s, y s :: [E] \) be such that

\[
x s = \text{minBy} (\hat{g} \# (x s + y s)) (\text{pref } x s + y s). 
\]

If \( \hat{g} \) is prefix-stable, we have that for all \( w s \),

\[
\text{minBy} \ g' (\text{map } w s + y s) (\text{pref } x s + y s)) = \text{minBy} \ g' (\text{map } w s + y s) (\text{pref } x s), 
\]

where \( g' = \hat{g} \# (w s + x s + y s) \).
\end{lemma}
Recall \( \text{optpref} \; x \; s = \text{minBy} \; (\cdot \; s) \; (\text{pref} \; x \; s) \). We have shown that, if \( w \) is concave, we have

\[
\text{optpref} \; [] = [] \\
\text{optpref} \; (x : xs) = \text{minBy} \; (\cdot \; (x : xs)) \; (\text{pref} \; (x : \text{optpref} \; xs)) .
\]

That is, the optimal prefix of \( x : xs \) can be computed from the optimal prefix of \( xs \) by appending \( x \) to its left, enumerate all the prefixes, and pick an optimal one.

While this allows us, as discussed in the end of the last section, to schedule the computation of \( \text{optArr} \), the expression above looks like any of

\[
\text{foldr} \; (\lambda n \rightarrow \text{minBy} \; (\cdot \; n) \cdot \text{pref} \cdot \text{cons}) \; [] .
\]

This does not look particularly efficient. Recall that we are aiming for a linear-time algorithm, while the expression above looks like anything but a constant-time computation. We will seek for ways to speed it up in the next section.

### Optimal Prefix in a Fold

The function \( \text{optpref} \) is almost a fold: it forms a \( \text{foldr} \) together with the function \( \text{length} \). For notational convenience, we design a variation of \( \text{foldr} \) that passes around a length:

\[
\text{foldr} \; g : (N \rightarrow (a, b) \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{foldr} \; g \; f \; e \; a = \text{snd} \; \text{foldr} \\
(\lambda (n, y) \rightarrow (1 + n, f \; (1 + n \; ((x, y),)) \; (0, e)) ,
\]

whose operational meaning is perhaps clearer from its type. Note that \( f \) takes a length and a pair \((a, b)\) where \( a \) is the current element in the list and \( b \) the result of the tail — we find such uncurried algebras sometimes convenient for point-free program calculation. We have, with \( \text{cons} \; (x, y) = x : y \cdot \text{cons} \):

\[
\text{optpref} = \text{foldr} \; (\lambda n \rightarrow \text{minBy} \; (\cdot \; n) \cdot \text{pref} \cdot \text{cons}) [[] .
\]

**Proof of Lemma 6**

For interested readers, the proof of Lemma 6 is given below.

**Proof.** The function \( \text{pref} \) has the following property:

\[
\text{pref} \; (x \; s + y) = \text{pref} \; x \; s + \text{map} \; (x \; s +) \; (\text{pref} \; y) .
\]

To prove the lemma, we reason:

\[
\text{minBy} \; g' \; (\text{map} \; (x \; s +) \; (\text{map} \; (x \; s +) \; y)) = \{ \text{by (5)} \} \\
\text{optpref} \; (x \; s +) = \text{minBy} \; (\cdot \; s +) \; \text{pref} \; x \; s + \text{map} \; (x \; s +) \; (\text{pref} \; y) = \text{minBy} \; (\cdot \; (x \; s +) +) \; \text{pref} \; y)
\]

For the penultimate step to hold we need Lemma 7.

**Lemma 7.** Assume that \( \cdot \) is prefix-stable. Let \( zss \; :: \; [[E]] \) be such that \( (\forall zss : zss \in zss : \; g \; n \; x \; zss = g \; n \; (x + zss)) \), then

\[
(zw + xs) \downarrow g' \; (\text{map} \; (x \; s +) \; zss) = (x + zs),
\]

where \( g' = g \; (n + \# x) \).
Definition 9 (2-in-3 for splits). A function $g : ([E], [E]) \to \mathbb{R}$ is two-in-three for splits if there exists a threshold function $\delta : ([E], [E]) \to \mathbb{R}$ such that

$$\delta (x, y, z) \leq \delta (y, z) \Rightarrow \begin{cases} g (w, x, y, z) \leq g (w, y, x, z) \land g (w, x, y + z) \leq g (w + x, y + z) \land g (w + x, y + z) \leq g (w + x, y + z). \end{cases}$$

Again, we use boldface font for the first arguments of $\delta$ and $g$ to help comparing with Definition 8.

Two-in-three is not an intuitive property. Given a cost function, it is not obvious how to determine whether it is “properly” two-in-three (as opposed to trivially two-in-three as described above), nor is it easy to construct a threshold function. Nevertheless, the following theorem, inspired by a similar property from Brucker (1995), suggests an effective approach to discover threshold functions.

Theorem 10. Given $g : ([E], [E]) \to \mathbb{R}$, if there exists $\delta : ([E], [E]) \to \mathbb{R}$ and $k : [E] \to \mathbb{R}$ such that for all $x, y, z, k$ and $\delta$

$$g (x, y, z) \leq g (x + y, z) \Leftrightarrow \delta (y, z) \leq k (x + y + z), \quad (6)$$

then $g$ is two-in-three with $\delta$ as its threshold function.

Note that $\delta$ is a function that can be computed independently from $xs$, while the function $k$ has all the elements in the three lists, but has no information how they are split.

Proof. Assume that (6) holds. The aim is to prove that

$$\forall w : g (w, x, y + z) \leq g (w, x + y, z) \land g (w + x, y + z) \leq g (w + x, y + z), \quad (7)$$

under the assumption that $\delta (x, y + z) \leq \delta (y, z)$. Consider comparing $k (w + x + y + z)$ and $\delta (x, y + z)$.

Case $\delta (x, y + z) > k (w + x + y + z).$ By (6) it is equivalent to $g (w, x, y + z) \leq g (w + x, y + z)$.

Case $\delta (x, y + z) > k (w + x + y + z).$

$$\Rightarrow \begin{cases} \text{ since } \delta (x, y + z) \leq \delta (y, z) \\ \delta (y, z) > k (w + x + y + z) \Leftrightarrow \delta (y, z) > k (w + x + y + z) \Leftrightarrow \text{ contravariant of (6)} \end{cases}$$

$$g (w + x, y + z) < g (w + x + y + z).$$

In fact, (6) can be extended to guarantee prefix-stability. The following theorem is not needed in this pearl, but recorded for completeness.

Theorem 11. If $g, \delta$ and $k$ satisfy (6) and $k$ is non-decreasing with respect to prefix, that is, $k x \leq k y x + x$ for all $y x$ and $x$, then $\hat{g}$ is prefix-stable.

5. Queueing and Glueing

Let the input be $[x_1, x_2, \ldots, x_n]$. Recall that, if we have $\delta ([x_1, \ldots, x_n]) \leq \delta ([x_2, \ldots, x_n])$, the segment $[x_1, x_2]$ can be treated atomically because an optimal prefix will never end at $x_1$. The same principle applies to larger chunks: if $[x_1, x_2]$ and $[x_2, x_3]$ are both atomic and $\delta ([x_2, x_3]) \leq \delta ([x_2, x_3])$, $[x_1, x_2]$ can be treated as one atomic segment.

That suggests a data type refinement: rather than keeping the previously computed optimal prefix, we will keep a list of segments $xs = [x_1, x_2, \ldots, x_n]$. Such that $\text{concat} xss$ is the optimal prefix, and all segment in $xss$ are atomic. Segments $x_{s1}, x_{s2}$, etc mark the “breakpoints” we need to consider when we add new elements and compute optimal prefixes. Such a list of atomic segments can be built by first turning all the elements into singleton segments, and keeping glueing adjacent segments $x_{s1}$ and $x_{s1+1}$ if $\delta (x_{s1+1}) \leq \delta (x_{s1+1})$. This continues until the $\delta$ values of segments in $xss$ are strictly decreasing. We will then show that for such a sequence of segments, there is a quicker way to find a prefix having minimum cost.

This section constitutes the core of our development. A spoiler for impatient readers: by the end of this section it will be clear that the core computation of or algorithm has the following form:

$$\text{foldr} : (\lambda n \rightarrow \text{minchop} \cdot \text{cons} \cdot (\text{id} \times \text{prepend})) [\text{foldr}]$$

where foldr is a fold similar to foldr but defined on lists of segments, prepend glues adjacent elements until there is no adjacent $x_{si}$ and $x_{si+1}$ such that $\delta (x_{si+1}) \leq \delta (x_{si+1})$, while minchop does the work of $\text{minBy} (\lambda n \cdot \text{prefs})$, but more efficiently. For minchop to work we shall store the segments in a queue, hence the name “queueing-glueing”.

5.1 Glueing Segments

To formalise the data refinement, let $\text{wrap} x = [x]$ and note that

$$\text{prefs} = \text{map} \text{concat} \cdot \text{pref} \cdot \text{map} \text{wrap}, \quad (7)$$

since computing all prefixes of a list $x :: [a]$ is the same as wrapping each element of $x$ as singleton lists, computing all prefixes of the list of type $[a]$, and do a concat for each of the resulting prefix. We calculate:

$$\text{opppref} x = \text{minBy} (\lambda (\# x) \cdot \text{prefs} \cdot \text{map} \cdot \text{wrap}) x$$

The inner minBy $(\lambda (\# x) \cdot \text{prefs}$ now operate on lists of lists. For easy reference later, we give it a name opppref$S$ (where the capital “S” refers to “segments”):

$$\text{opppref} S x = \text{minBy} (\lambda (\# \text{concat} x) \cdot \text{prefs} \cdot \text{map} \cdot \text{wrap} x$$

It can be shown that, for all $h :: [E] \rightarrow \mathbb{R}$, $\text{concat}$ is prefix-stable if $h$ is. Therefore the derivation we did in the previous section can be repeated — if we define the following variation of fold:

$$\text{foldr} : (\lambda n \rightarrow ([a], b) \rightarrow b \rightarrow ([a]) \rightarrow b$$

$$\text{foldr} (\lambda e \rightarrow \text{snd} \cdot \text{fdrd}$$

$$\text{opppref} S x = \text{minBy} (\lambda (\# x) \cdot \text{prefs} \cdot \text{cons} \cdot \text{map} \cdot \text{wrap} x$$

we have that

$$\text{opppref} S = \text{foldr} (\lambda n \rightarrow \text{minBy} (\lambda (\# n) \cdot \text{prefs} \cdot \text{cons}) [\text{foldr}]$$

Relations

The next step is to fuse something that models glueing of segments into the fold. It turns out that it is cleaner, however, if we use relations instead of functions for this stage of development.

Relations are generalisation of functions. A function $A \rightarrow B$ can be seen as a set of pairs $\{(b, a) \mid a : A, b : B\}$ such that, for every $a : A$, there must be a unique $b : B$ such that $(b, a) \in \text{the set}$. For example, the function $(\lambda n \rightarrow n)$ is a set of $(0, 0), (1, 1), (2, 2), (3, 3)$). A relation $R : A \rightarrow B$ is a set $(\{a, b\} \mid a : A, b : B)$ without further restrictions; given $a : A$ there could be zero or more $b : B$ such that $(b, a) \in R$. When $(b, a) \in R$ we say that $b$ maps to $a$. Relations can be used to model model non-deterministic computations: each $b$ such that $(b, a) \in R$ is a possible output of $R$ on input $a$.

Relational program derivation often proceeds by establishing a sequence of inclusions as well as equality:

$$R_1 \supseteq R_2 \supseteq \ldots \supseteq R_n.$$
where $R_1$ is a problem specification and $R_n$ is a refinement. The inclusion guarantees that whatever result $R_n$ returns on a input is a result allowed by $R_1$. This allows the empty relation to be a refinement of any relation. Therefore, in the end of a derivation we often need to check that $R_n$ preserves the domain of $R_1$. The check can be trivial when, for example, $R_1$ is total and $R_n$ is a composition of total functions and thus also total.

Composition of relations is defined by $(c,a) \in R \cdot S$ iff. there exists $b$ such that $(c,b) \in R$ and $(b,a) \in S$. The identity $id$ is the identity of relational composition. The reflexive, transitive closure of a relation $R$ is denoted by $R^*$. For a definition, $R \cdot R^*$ is the least fixed-point of $(RX \to S \cup X \cdot R)$. We thus have (Backhouse 2002):

$$S \cdot R^* \subseteq T \iff S \subseteq T \land T \cdot R \subseteq T \ .$$

Finally, with $foldr^*_R$ defined previously, we have the following fusion theorem:

$$R \cdot foldr^*_R S e \geq foldr^*_R T e' \iff \ (\forall n. R \cdot S n \geq T n \cdot (id \times R)) \land (e', e) \in R \ .$$

\section*{Glueing}

Back to our problem. For this task we use a relation $\text{glue}::[[E]] \to [[E]]$ defined by:

$$\text{glue} \equiv \exists xss, xs, ys, yss. wss = xss + [[xs,ys]] + yss \land wss' = xss + [[xs + ys]] + yss \land \delta (xs, ys + zs) \subseteq \delta (ys, zs) ,$$

where $zs = \text{concat} yss$. That is, $\text{wss}::[[E]]$ is in the domain of $\text{glue}$ if we can find in $\text{wss}$ two adjacent segments $xs$ and $ys$ with $\delta (xs, ys + zs) \subseteq \delta (ys, zs)$. The output $\text{wss}'$ is formed by glueing $xs$ and $ys$ together. There may be more than one such pair and $\text{wss}$ could be mapped to multiple outputs.

The relation $\text{glue}^*$ performs $\text{glue}$ on the input list of segments an indefinite number of times. Our aim now is to introduce $\text{glue}^*$ and promote it into $\text{optprefS}$.

Let us see what properties we have. Define $\text{glue}^0 = id \cup \text{glue} -$ a relation that might perform one $\text{glue}$ or leave the input unchanged. We have that if $hc = h \cdot \text{concat}$ for some $h$, then $\text{minBy} \ h \cdot \text{map} \ \text{glue}^0 = \text{glue}^0 \cdot \text{minBy} \ h \ .$

In words, $\text{glue}$ does not change the result of $\text{minBy} \ h \cdot \text{map} \ \text{glue}^0 = \text{glue}^0 \cdot \text{minBy} \ h \ .$

We would also like to establish some relationship between $\text{prefS}$ and $\text{glue}$. Assume that $\text{glue}$ maps $wss = xss + [[xs,ys]] + yss$ to $wss' = xss + [[xs + ys]] + yss$. where $\delta (xs, ys + zs) \subseteq \delta (ys, zs)$ (with $zs = \text{concat} yss$). We have $\text{prefS} wss = \text{prefS} (\text{init} \ \text{xs}) + $ $\quad (\times s, xss + [[xz]], xss + [[xs, ys]]) \land$ $\quad \text{map} (\times s + [[xs + ys]]) (\text{prefS} yss) \ .$

$\text{prefS} wss' = \text{prefS} (\text{init} \ \text{xs}) + $ $\quad (\times s, xss + [[xz + ys]]) \land$ $\quad \text{map} (\times s + [[xs + ys]]) (\text{prefS} yss) \ .$

One can see that $\text{prefS} wss'$ has one entry less than $\text{prefS} wss$. Furthermore, occurrences of $[[xs,ys]]$ are glued into $[[xs + ys]]$. If we define a relation $\text{rm}::[[[E]]] \to [[[E]]]$:

\begin{align*}
\text{rm} \equiv (\exists xss, xs, ys, yss. wss + xss + [[xs]], xss + [[xs,ys]] + yss \land \\
\quad \text{wss} + xss + [[xs,ys]] + yss \land \\
\quad \delta (xs, ys + zs) \subseteq \delta (ys, zs) ) ,
\end{align*}

where $zs = \text{concat} yss$, we have that $\text{prefS} \cdot \text{glue} \subseteq \text{map} \ \text{glue}^0 \cdot \text{rm} \cdot \text{prefS} \ .$

That is, whatever computed from $\text{prefS} \cdot \text{glue}$ can be computed by taking all prefixes, removing an entry $[[xs]]$, and performing some glueing.

Finally, if $h$ is two-in-three, we have $\text{minBy} \ h \cdot \text{rm} = \text{minBy} \ h \ .$

since the removed entry $[[xs]]$ would not have minimum cost.

Now we are ready to introduce $\text{glue}^*$ and promote it into $\text{optprefS}$:

$$\text{concat} \cdot \text{optprefS} = \text{concat} \cdot \text{foldr}^*_R (\lambda n \to \text{minBy} \ (\text{gc} n) \cdot \text{prefS} \cdot \text{cons} \ ) [ ]$$

$$\subseteq \{ \text{concat} \cdot \text{glue}^* = \text{concat} \}$$

$$\text{concat} \cdot \text{glue}^* \cdot \text{foldr}^*_R (\lambda n \to \text{minBy} \ (\text{gc} n) \cdot \text{prefS} \cdot \text{cons} \ ) [ ]$$

If the fusion succeeds, we are allowed to perform some glueing before $\text{minBy} \ (\text{gc} n) \cdot \text{prefS}$, thereby reduce the number of prefixes to consider. Abbreviating $\text{gc} n$ to $h$, the fusion condition is:

$$\text{minBy} h \cdot \text{prefS} \cdot \text{glue}^* \cdot \text{cons} \cdot (id \times \text{glue}^*)$$

$$\subseteq \text{glue}^* \cdot \text{minBy} h \cdot \text{prefS} \ .$$

It is not hard to see that $\text{glue}^* \cdot \text{cons} \cdot (id \times \text{glue}^*) \subseteq \text{glue}^* \cdot \text{cons}$. We are left with proving that $\text{minBy} h \cdot \text{prefS} \cdot \text{glue}^* \subseteq \text{glue}^* \cdot \text{minBy} h \cdot \text{prefS}$. This property says that the minimum prefix you get by first performing some glueing is a valid one, because you can always get the same result by first computing the optimal prefix before glueing. We prove the property as a lemma below:

\section*{Lemma 12}

$\text{minBy} h \cdot \text{prefS} \cdot \text{glue}^* \subseteq \text{glue}^* \cdot \text{minBy} h \cdot \text{prefS}$ if $h$ is two-in-three.

\section*{Proof}

By (8), the proof obligations are:

$$\text{minBy} h \cdot \text{prefS} \subseteq \text{glue}^* \cdot \text{minBy} h \cdot \text{prefS}$$

$$\text{glue}^* \cdot \text{minBy} h \cdot \text{prefS} \subseteq \text{glue}^* \cdot \text{minBy} h \cdot \text{prefS} \ .$$

The first can be easily discharged since $id \subseteq \text{glue}^*$. To prove the second, we calculate:

$$\text{glue}^* \cdot \text{minBy} h \cdot \text{prefS} \subseteq \{ \text{by (10)} \}$$

$$\text{glue}^* \cdot \text{minBy} h \cdot \text{map} \ \text{glue}^0 \cdot \text{rm} \cdot \text{prefS} \subseteq \{ \text{by (9)} \}$$

$$\text{glue}^* \cdot \text{minBy} h \cdot \text{rm} \cdot \text{prefS} \subseteq \{ \text{by (10)} \} \}$$

$$\text{glue}^* \cdot \text{minBy} h \cdot \text{prefS} \ .$$

\section*{Refining to Functions}

Having just shown that

$$\text{glue}^* \cdot \text{optprefS} \subseteq \{ \text{foldr}_R^* \text{ fusion} \}$$

$$\text{foldr}_R^* (\lambda n \to \text{minBy} \ (\text{gc} n) \cdot \text{prefS} \cdot \text{cons} \ ) [ ] ,$$

we may now choose a particular implementation of $\text{glue}^*$. Define $\text{prefS} :: [[[E]]] \to [[[E]]]$

$$\text{preprend} :: [[E]] \to [[[E]]]$$

$$\text{preprend} [ xs ] = [ [ xs ] ] \land \text{preprend} [ (xs,ys) : xs ]$$

$$\delta xs \leq \delta ys \Rightarrow \text{preprend} (xs + ys : xs)$$

$$\text{otherwise } = xs : ys : xs \ .$$

The function is supposed to be run after $\text{cons}$. It keeps glueing segments on the left-end of the list, until $\delta xs > \delta ys$ where $xs$ and $ys$ are the two leftmost segments.
It is clear that $\text{prepend} \subseteq \text{glue}^*$. However, we will see in Section 6 that some choices of $\delta$ may look up $\text{optArr}$, and performing $\text{prepend}$ on the entire input results in a circular dependency. Therefore, we use the fact that $\text{cons} \cdot (id \times \text{glue}^*) \subseteq \text{glue}^* \cdot \text{cons}$, and refine $\text{glue}^* \cdot \text{cons} \cdot (id \times \text{glue}^*)$. In summary, we have shown that

$$\text{glue}^* \cdot \text{optprefS} \supseteq \text{foldr}_\delta (\lambda n \to \text{minBy} \ (\hat{g} \cdot n) \cdot \text{prefs} \cdot \text{cons} \cdot (id \times \text{prepend})) \ [[].$$

The next thing to do is to refine $\text{minBy} \ (\hat{g} \cdot n) \cdot \text{prefs}.$

It is not hard to show that if $\text{xs}$ is a list of segments with decreasing $\delta$ values, in other words, $\text{xs}$ is completely glued in the sense that glue cannot be applied anymore, $\text{prepend} \ (\text{xs} : \text{xs})$ will also be a list with decreasing $\delta$ values. This will be an invariant: in the body of $\text{optpref}$, the tail of the list of segments we maintain will always be fully glued and be sorted in decreasing $\delta$ values.

Some reflection on the use of relations. Rather than introducing $\text{prepend}$ right in the beginning, we performed a fold fusion with $\text{glue}^*$ and refine glue to $\text{prepend}$ afterwards. In a functional development, we would be attempting to fuse foldr $\ (\text{prepend} \cdot \text{cons}) \ [\]$ into the fold, and the property corresponding to Lemma 12 is $\text{minBy} \ h \cdot \text{prefs} \cdot \text{prepend} = \text{foldr} \ (\text{prepend} \cdot \text{cons}) \ [\] \cdot \text{minBy} \ h \cdot \text{prefs},$ whose proof would tie us into the particular order $\text{prepend}$ glues the segments and would be much more tedious. The advantage of using relations here is that they allow us not to over specify, and focus on the essence that makes the theorem true.

5.2 Finding Minimum

Consider the following list of segments, an output of $\text{prepend}$:

$$\text{xs} = [x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_k],$$

among whose prefixes we want to find one having minimum cost with respect to $\hat{g} \cdot n$. Being results of $\text{prepend}$, the $k$ segments can be seen atomically and we therefore have $k + 1$ candidates to consider. While this is already an improvement over the original $n + 1$ prefixes, the invariant maintained by $\text{prepend}$ allows us to reduce the number of potential candidates even further.

Assume that $\hat{g} \cdot n \ [x_1, x_2, \ldots, x_i] > \hat{g} \cdot n \ [x_1, x_2, \ldots, x_{i+1}]$. Expanding the definitions, we get

$$\hat{g} \ (x_1 + \ldots + x_i) > \hat{g} \ (x_1 + \ldots + x_{i+1}),$$

which, by (6), is equivalent to

$$\delta (x_i, x_{i+1} + \ldots + x_k) > k \ (\text{concat} \ \text{xs}).$$

Recall that, being an output of $\text{prepend}$, the $\delta$ values of segments in the list are strictly decreasing. That means we have, for all $1 \leq j \leq i$, 

$$\delta (x_j, x_{j+1} + \ldots + x_k) > k \ (\text{concat} \ \text{xs}),$$

which is in turn equivalent to that

$$\hat{g} \ n \ [x_1] > \hat{g} \ n \ [x_1, x_2] > \ldots > \hat{g} \ n \ [x_1, x_2, \ldots, x_i] > \hat{g} \ n \ [x_1, x_2, \ldots, x_{i+1}]$$

for all $1 \leq j \leq i$. That is, the costs of all prefixes shorter than $x_1 + \ldots + x_i$ are strictly decreasing when their lengths increase. Therefore, $x_1 + \ldots + x_i$ is a prefix having minimum cost.

To find a prefix having minimum cost, one may therefore start from the right end of the list and keep comparing until seeing a point where $\hat{g} \ n \ [\text{xs}] > \hat{g} \ n \ (x_i + \ldots + x_j)$. When that happens, we know that $\hat{g} \ n \ [x_i] \cdot \text{cons} + [x_i]$ has minimal cost and we may stop.

$$\text{minchop} \ n \ [x_i] = [x_i]$$

$$\text{minchop} \ n \ ([x_i] + [x])$$

$$\hat{g} \ n \ [x_i] \leq \hat{g} \ n \ (x_i + [x_i]) = \text{minchop} \ n \ [x_i] \cdot \text{cons} + [x_i]$$

$$\text{else} = x_i + [x_i].$$

In the end of Section 5.1, we refined $\text{glue}^*$ to $\text{cons} \cdot (id \times \text{prepend})$. This way, the $\hat{\delta}$ values of the list is sorted except the first segment. It is still correct, however, to refine $\text{minBy} \ (\hat{g} \cdot n)$ to $\text{minchop}$. In summary, we now have:

$$\text{glue}^* \cdot \text{optprefS} \supseteq \text{foldr}_\delta (\lambda n \to \text{minchop} \ (1 + n) \cdot \text{cons} \cdot (id \times \text{prepend})) \ [[].$$

An abstract presentation of the queueing-glueing algorithm is shown in Figure 1 as a function that takes $\hat{w}$ and $\hat{\delta}$ as parameters, where we denote the type of elements by $e$. For now, we can read $Q \ (J \ e)$ as $[\ [e] \ ]$. Efficient representation of $J$ and $Q$ will be discussed in Section 5.3.

The function $\text{optpref}$ implements $\text{glue}^* \cdot \text{optprefS} \cdot \text{map} \ \text{wrap}.$ As mentioned before, the results of $\text{optpart}$ for each suffix is stored in the array $\text{optArr}$, which is then referenced by $\text{optpart}$. Instead of prefixes, however, we now store the sequence of segments in $\text{optArr}$. We will see in the next section that $\hat{\delta}$ might refer to $\text{optArr}$.

In these cases we can make $\hat{\delta}$ take the array as an argument.

To compute $\text{optpref} \ (x : x)$, the result of $\text{optpref} \ x$ is fetched from $\text{optArr}$ since, by definition, $\text{head} \ (\text{optArr}! n) = \text{optpref} \ n \ x$ where $x$ is the suffix of $\text{inp}$ having length $n$. We therefore form a chain of dependencies. Let $n$ be the length of the input. In the main function $\text{opt}$ the result is $\text{optArr}! (n - 1)$, etc. The array is thus computed from $\text{optArr}! 0$ back to $\text{optArr}! n$. The function $\text{optpref}$ calls $\text{prepend}$ and $\text{minchop}$ once for each suffix of the input. The function $\hat{\delta}$ in $\text{prepend}$ is a
5.3 Efficiency Analysis and Summary

It is important that minchop processes its input from the right end. Each time \( \delta \) is called at most \( O(n) \) times throughout the algorithm. When we reach a point where \( \delta \) holds, \( \delta \) is dropped and will never be accessed in subsequent steps of the algorithm. Therefore, \( \delta \) can be called at most \( O(n) \) times.

To allow \( \text{prepend} \) and \( \text{minchop} \) to operate on both ends of the sequence of segments, we store the segments in a queue that allows amortised constant-time addition from the left end, and constant-time removal from both ends, hence the name “queue-glueing.” The type of queues is denoted by \( Q \) in Figure 1. A number of data structures support such operations, for example, Banker’s dequesues (Okasaki 1999), or 2-3 finger trees (Hinze and Paterson 2006). As for the segments themselves, the only structural operations (those apart from computation of \( \delta \) and \( \delta \), etc) we perform on them are glueing (\( \delta \)) and conversion back to ordinary lists. Thus they can be represented by join lists:

\[
\text{data} \ J \ a = \text{Singleton} \ a \ | \ (\text{Join} \ (J \ a) \ (J \ a))
\]

Alternatively we can use Hughes lists \( ([a] \rightarrow [a]) \) (Hughes 1986) to achieve constant-time concatenation.

With the above support from data structures, our algorithm runs in linear time overall — provided that our basic operations (length, \( \delta \), and \( \delta \)) can be computed in constant time, which can be done if we store the lengths, values of \( \delta \), and other necessary information along with each segment and prefixes in the array. A more complex implementation of the queueing-glueing algorithm that uses functional queues, join-lists, and uses cached information is given in the code repository accompanying this pearl.

6. Applications

Finally, the promised solutions to the problems given in Section 1.

6.1 One-Machine Batching

In the one-machine batching problem (Brucker 1995), a list of jobs are to be processed on a machine in the order presented (leftmost first). Each Job is associated with a weight indicating its importance, and a processing time (which we will call its span, to be distinguished from absolute time). The attributes can be respectively extracted by the selectors

\[
\text{sp}, \text{wt} :: \text{Job} \rightarrow \mathbb{R}.
\]

A machine processes the jobs in batches. The processing span of a batch is the sum of spans of its jobs, plus a fixed starting-up overhead \( s \):

\[
\text{bspn} :: \text{Job} \rightarrow \mathbb{R}
\]

\[
\text{bspn} = (s+ \cdot \text{sum} \cdot \text{map} \ sp).
\]

Given a list of batches, the finishing time of its last batch is:

\[
\text{ftime} :: \text{Job} \rightarrow \mathbb{R}
\]

\[
\text{ftime} = \text{sum} \cdot \text{map} \ \text{bspn}.
\]

The (absolute) finishing time of a job, however, is not the exact time itself is processed, but the finishing time of its batch. The goal is to minimise the sum of the weight of each job multiplied by its finishing time. The cost function is:

\[
f \left( \text{optArr} \right) = \left[ \sum \text{Job} \rightarrow \mathbb{R} \right.
\]

\[
f \left( \left[ \right] \right) = 0
\]

\[
f \left( \left[ \text{Job} \right] \right) = \text{ftime} \left( \left[ \text{Job} \right] \right) + \text{wts} \cdot \text{ftime} \left( \left[ \text{Job} \right] \right)
\]

where \( \text{wts} = \text{sum} \cdot \text{map} \ \text{wt} \).

For some intuition, consider two extreme solutions. If we let all jobs be in a batch consisting of only itself, we end up wasting too much starting-up overhead. If we include all jobs in one big batch, all the jobs end up having the same longest finishing time. An optimal strategy is usually something in-between.

A key observation to solving this problem is to notice that the function \( f \) can also be computed from left to right and, in this form, one can easily apply Theorem 10 to construct \( \delta \). Define:

\[
\text{weights} :: \left[ \text{Job} \right] \rightarrow \mathbb{R}
\]

\[
\text{weights} = \text{sum} \cdot \text{map} \ \text{wt} \cdot \text{concat},
\]

which computes the sum of all weights of the jobs in batches. It turns out that \( f \) can also be computed by:

\[
f \left( \left[ \text{Job} \right] \right) = \text{bspn} \ \text{wts} \cdot \text{weights} \left( \left[ \text{Job} \right] \right) + f \left( \left[ \text{Job} \right] \right).
\]

It can be verified that \( w \left( \left[ \text{Job} \right] \right) = \text{bspn} \ \text{wts} \cdot \text{sum} \cdot \text{map} \ \text{wt} \left( \left[ \text{Job} \right] \right) \)

is concave. To discover \( \delta \) we reason (abbreviating \( \text{optpart} \) to \( f \)):

\[
g \left( \left[ \text{Job} \right] \right) \leq \text{bspn} \ \text{wts} \cdot \text{weights} \left( \left[ \text{Job} \right] \right) + \text{fo} \left( \left[ \text{Job} \right] \right)
\]

\[
= \text{bspn} \ \text{wts} \cdot \text{weights} \left( \left[ \text{Job} \right] \right) + \text{fo} \left( \left[ \text{Job} \right] \right)
\]

\[
= \text{bspn} \ \text{wts} \cdot \text{weights} \left( \left[ \text{Job} \right] \right) + \text{fo} \left( \left[ \text{Job} \right] \right)
\]

\[
= \text{bspn} \ \text{wts} \cdot \text{weights} \left( \left[ \text{Job} \right] \right) + \text{fo} \left( \left[ \text{Job} \right] \right)
\]

\[
= \text{weights} \left( \left[ \text{Job} \right] \right)
\]

We have thus derived:

\[
\delta \left( \left[ \text{Job} \right] \right) = \text{weights} \left( \left[ \text{Job} \right] \right)
\]

\[
= \text{bspn} \ \text{wts} \cdot \text{weights} \left( \left[ \text{Job} \right] \right) + \text{fo} \left( \left[ \text{Job} \right] \right)
\]

To compute \( \delta \) in constant time, we can decorate each prefix in \( \text{optArr} \) with their \( \text{bspn} \) value and each partition with sum of their weights. Alternatively we can do some preprocessing and create two arrays \( \text{sumsp} \) and \( \text{sumwt} \) storing running sums of spans and weights of all suffixes:

\[
\delta \left( \left[ \text{Job} \right] \right) = \text{weights} \left( \left[ \text{Job} \right] \right)
\]

\[
= \text{bspn} \ \text{wts} \cdot \text{weights} \left( \left[ \text{Job} \right] \right) + \text{fo} \left( \left[ \text{Job} \right] \right)
\]

\[
= \text{weights} \left( \left[ \text{Job} \right] \right)
\]

For \( \text{optArr} :: \left[ \mathbb{N} \ | \ (Q \ (J \ \text{Job})) \right] \), we let \( \text{fc} = \text{f} \cdot \text{map} \ \text{concat} \). However, entries of \( \text{optArr} \) can be aggregated with the costs of partitions, making \( \text{fc} \) a selector. The function \( \delta \) can thus be computed in constant time.

Example For an example, consider a list of jobs with spans \( [12, 7, 18, 6, 12, 3, 4, 1, 12] \), whose weights are all 1. The values of \( \text{optArr} \left( i \right) \), for \( i \in [0..9] \), are shown below (the weights are all 1 and omitted):

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{optArr} \left( i \right) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

The first element of \( \text{optArr} \left( i \right) \) is the working queue. Some of the points to notice: in row 2, \( [12] \) is trimmed off the queue by
minchop after [1] is added. In row 4, [4] and [1] are joined by prepend before [3] is added. In row 6, [12], [3], and [4, 1] are joined into one segment, which is then trimmed off in row 7. The resulting optimal partition is map concat (optArr! 19), that is, \([\{12, 7\}, \{18, 6\}, \{12, 3, 4, 1\}, \{12\}]\).

6.2 Size-Specific Partition and Paragraph Formatting

In the size-specific partition problem, the input to be partitioned is a list of positive natural numbers. To make the problem a bit harder and to relate to the paragraph formatting problem later, we assume that there is a space of size 1 between each adjacent element. The size of a segment \(xs\) is therefore \(\sum xs + \#xs - 1\). The distance to \(L\) of each segment is measured by

\[ w \cdot xs = \left( L - (\sum xs + \#xs - 1) \right)^2, \]

while \(f = \text{sum} \cdot \text{map} \cdot w\). The presence of square allows partitions having more evenly distributed sizes to be favoured more than those in which some segments have a large distance.

One can verify with tedious but elementary arithmetics that \(f\) - \(optpart\) to fo:

\[
\begin{align*}
g (xs, ys) &= w \cdot xs + fo \cdot ys \\
&= (L - (\sum xs + \#xs - 1))^2 + fo \cdot ys \\
&= (L - (\sum (xs + x) - \sum ys + \#(xs + ys) - \#ys - 1))^2 + fo \cdot ys \\
&= \{ \text{let } L' = L + 1, sl \cdot xs = \sum xs + \#xs \} \\
&= (L' - sl \cdot (xs + ys) + s \cdot ys)^2 + fo \cdot ys .
\end{align*}
\]

Therefore we may do a preprocessing that stores \(sum\) and length of all suffixes of the input list.

To discover \(\delta\) we reason:

\[
\begin{align*}
g (xs, ys + zs) &\leq g (xs + ys, zs) \\
\equiv (L' - sl (xs + ys + zs) + sl (ys + zs))^2 + fo (ys + zs) \leq \\
&= (L' - sl (xs + ys + zs) + sl zs)^2 + fo zs \\
\equiv \{ \text{abbreviate } 2 + (sl (xs + ys + zs) - L') \} \text{ to U } \\
&- U + sl (ys + zs) + (sl (ys + zs))^2 + fo (ys + zs) \leq \\
&- U + sl zs + (zs)^2 + fo zs \\
\equiv ((sl (ys + zs))^2 + fo (ys + zs) - ((sl zs)^2 + fo zs)) / \\
(sl (ys + zs) - sl zs) \leq U .
\end{align*}
\]

Thus we have derived

\[
\begin{align*}
\delta (ys, zs) &= ((sl (ys + zs))^2 + fo (ys + zs) - ((sl zs)^2 + fo zs)) / \\
&\qquad (sl (ys + zs) - sl zs) ,
\end{align*}
\]

whose length-accepting equivalent is

\[
\delta n xx = (sj^2 - sk^2 + fc (optArr! n) - fc (optArr! m)) / (sj - sk) \ \\
\text{where } \{ sj = \text{sums}! n; sk = \text{sums}! m; m = n - \#xs \} ,
\]

where the array \(\text{sums}\) stores values of \(sl\) for each suffix, and \(fc\), as in the last section, is conceptually \(f \cdot \text{map} \cdot \text{concat}\) but can be implemented as a constant-time selector.

Paragraph Formatting

The sized partitioning problem appears to be only slightly different from the paragraph formatting problem (Knuth and Plass 1981), often used to demonstrate the use of formal methods. Functional treatment of the problem has been given before by, for example Bird (1986) and de Moor and Gibbons (1999). While it is known that the problem can be solved in linear time, it is perhaps surprising that it can be solved by the queueing-glueing algorithm too.

The input is a list of words which can be abstracted into a list of positive natural numbers denoting the numbers of characters in each word. The goal is to partition the list into lines and minimise waste space. The problem use the same \(w\) for each line, but the last line of a paragraph is not counted. The cost of a paragraph is defined by:

\[
\begin{align*}
f [] &= 0 \\
f [xs] &= 0 \\
f (xs; yss) &= w \cdot xs + f \cdot yss ,
\end{align*}
\]

with the same \(w\) as the previous problem. One can verify that (1) still holds if \((\_f)\) is defined to return the shorter argument in case of a tie. With this cost function, however, a layout putting everything in one single line would have cost 0.

We therefore might want to enforce that each line does not exceed \(L\). The specification becomes

\[\text{optpart} = \text{minBy} f \cdot \text{all} \cdot p \cdot \text{parts},\]

where \(p = \sum xs + \#xs - 1 \leq L\). Brucker (1995) claimed that his algorithm can be adapted to enforce constraints on length of segments simply by having \(\text{minchop}\) chopping off segments that are too long. We noticed, however, that it is not the case. Consider that \(\text{prepend}\) we glued \(xs\) and \(ys\) when \(\delta (xs, ys + zs) \leq \delta (xs + yss, zs)\) because we knew that for all \(ws\):

\[g (ws + xs, ys + zs) \geq g (ws, xs + ys + zs) \lor g (ws + xs, ys + zs) \geq g (ws + xs + yss, zs) .\]

With the constraint \(p\), however, it could be the case that only the second clause holds, but \(ws + xs + ys\) is too long, and \(ws + xs\) cannot be disposed as a potential answer.

With the size constraint \(p\), we can only join \(xs\) and \(ys\) when \(\delta (xs, ys + zs) \leq \delta (xs + yss, zs)\) and

\[\forall ws: \neg (p (ws + xs + ys)) : g (ws + xs, ys + zs) \geq g (ws, xs + ys + zs) ,\]

which by Theorem 10 is equivalent to

\[\forall ws: \neg (p (ws + xs + ys)) : \delta (xs, ys + zs) \leq k (ws + xs + ys + zs) .\]

Furthermore, we may, for each \(xs\), find out the shortest \(w_0\) that falsifies \(p (w_0 + xs + ys)\) by a linear-time preprocessing. If \(\delta (xs, ys + zs) \leq k (ws + xs + ys + zs)\) holds, since \(k\) increases with longer prefixes, the inequality will hold for all prefixes longer than \(w_0\). The first clause of \(\text{prepend}\) thus becomes:

\[\text{prepend} n (xs; yss) : \delta n xx \leq \delta (n - \#xs) ys \land \delta n xx \leq (2 \cdot (\text{sums}! (j - L - 1)) = \text{prepend} n (xs + yss; xss) ,\]

where \(\text{sums}! j = sl (w_0 + xs + yss + \text{concat} xss)\). The function \(\text{minchop}\) only need to be extended with one clause that throws away \(xs\) when \(sl (xs + [xs]) > L\). It can be proved that when \(\text{minchop}\) stops, the segments in the queue still have strictly decreasing values of \(g\), and thus \(\text{minchop}\) is still correct. For more details the reader may see the programs accompanying this pearl.

7. Conclusion

We have presented a derivation of the queueing-glueing algorithm, an algorithmic pattern that has been rediscovered many times without a formal treatment. As we have seen, it offers an elegant linear-time solution to a number of optimal partitioning problems with complex cost functions.

The algorithm presented here is very much inspired by that of Brucker (1995) — with some notable differences, however. Brucker’s algorithm performs \(\text{minchop}\) before \(\text{prepend}\), which, at
least in our implementation, resulted in a circular data dependency. Brucker’s claim that the algorithm can be easily adapted to handle size constraints also appears problematic. This may show that our calculation was not done without merits.

The algorithm belongs to a larger class extensively discussed in the algorithm community — accelerating dynamic programming by using the Monge property. While our focus is limited to concave weight functions, Galil and Park (1992) investigated problems with convex weight functions, and presented algorithms that use stacks rather than queues and run in $O(n \log n)$ time, which can be improved to $O(n \alpha(n))$ using more complex construction (Klawe and Kleitman 1990). A problem less general than Brucker’s was considered by van Hoesel et al. (1994), who presented linear-time algorithms for both concave and convex weight functions satisfying additional monotonicity constraints, respectively using a queue and a stack. These are all interesting directions to investigate.

Acknowledgements The authors would like to thank the referees for their valuable comments. We also thank Jeremy Gibbons for commenting on an early draft of this pearl, and for suggesting the name “queueing and glueing.”

References


