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
yuhsi.chiang@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 *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 $7*3+17*2+21=76$. 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/scmu/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

$$\begin{array}{l} \mathit{optseg} :: [\mathsf{E}] \to [\mathsf{E}] \\ \mathit{optseg} = \mathit{minBy}\ w \cdot \mathit{segs} \ \ , \end{array}$$

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

$$\mathit{segs} = \mathit{concat} \cdot \mathit{map}\ \mathit{prefs} \cdot \mathit{suffs} \ \ ,$$

where *prefs* and *suffs* respectively return all the non-empty prefixes and suffixes. The definition of *prefs* is given below:

$$\begin{array}{ll} \mathit{prefs} & :: [a] \to [[a]] \\ \mathit{prefs}\ [\,] & = [\,] \\ \mathit{prefs}\ (x : xs) & = [x] : \mathit{map}\ (x:)\ (\mathit{prefs}\ xs) \ \ . \end{array}$$

Standard calculation yields that

$$\begin{array}{ll} & \mathit{optseg} \\ = & \quad \{ \text{ definitions of } \mathit{optseg} \text{ and } \mathit{segs} \ \} \\ & \mathit{minBy}\ w \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{prefs} \cdot \mathit{suffs} \\ = & \quad \{ \text{ since } \mathit{minBy}\ w \cdot \mathit{concat} = \mathit{minBy}\ w \cdot \mathit{map}\ (\mathit{minBy}\ w) \ \} \\ & \mathit{minBy}\ w \cdot \mathit{map}\ (\mathit{minBy}\ w \cdot \mathit{prefs}) \cdot \mathit{suffs} \ \ . \end{array}$$

The part $\mathit{map}\ (\mathit{minBy}\ w \cdot \mathit{prefs}) \cdot \mathit{suffs}$ computes a list of optimal prefixes, one for each suffix, from which an optimal one is chosen.

The *scan lemma* then states that, if $\mathit{minBy}\ w \cdot \mathit{prefs}$ can be computed in a *foldr*, *optseg* can be computed in a *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 $\mathit{xss} :: [[a]]$ is a partition of $\mathit{xs} :: [a]$ if $\mathit{concat}\ \mathit{xss} = \mathit{xs}$ and all elements in *xss* are non-empty. We also call each element in *xss* a "segment" of the partition. The following function *parts* computes all partitions of its input:

$$\begin{array}{l} \mathit{parts} \ \ :: [a] \to [[[a]]] \\ \mathit{parts}\ [\,] = [[\,]] \\ \mathit{parts}\ \mathit{xs} = (\mathit{concat} \cdot \mathit{map}\ \mathit{extparts} \cdot \mathit{splits})\ \mathit{xs} \ \ , \end{array}$$

where $\mathit{extparts}\ (\mathit{xs}, \mathit{ys}) = \mathit{map}\ (\mathit{xs}:)\ (\mathit{parts}\ \mathit{ys})$. The function $\mathit{splits}\ \mathit{xs}$ returns all the $(\mathit{ys}, \mathit{zs})$ where *ys* is non-empty and $\mathit{ys} \mathbin{+\!\!+} \mathit{zs} = \mathit{xs}$:

$$\begin{array}{ll} \mathit{splits} & :: [a] \to [([a],[a])] \\ \mathit{splits}\ [\,] & = [\,] \\ \mathit{splits}\ (x : \mathit{xs}) & = ([x], \mathit{xs}) : \mathit{map}\ ((x:) \times \mathit{id})\ (\mathit{splits}\ \mathit{xs}) \ \ , \end{array}$$

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

$$[[[1],[2],[3]],[[1],[2,3]],[[1,2],[3]],[[1,2,3]]] \ \ .$$

Notice that the definition of *splits* is rather similar to that of *prefs*.

From now on, a pair of lists $([\mathsf{E}],[\mathsf{E}])$ will be called a *split*. Some cost functions will be defined on splits rather than segments.

The function computing optimal partitions can be specified by:

$$\begin{array}{l} \mathit{optpart} :: [\mathsf{E}] \to [[\mathsf{E}]] \\ \mathit{optpart} = \mathit{minBy}\ f \cdot \mathit{parts} \ \ . \end{array}$$

In a simpler scenario, the cost of a partition is the sum of the costs of its parts, that is, $f :: [[\mathsf{E}]] \to \mathbb{R}$ has the form $f = \mathit{sum} \cdot \mathit{map}\ w$ for some $w :: [\mathsf{E}] \to \mathbb{R}$. 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, $w :: ([\mathsf{E}],[\mathsf{E}]) \to \mathbb{R}$, and let *f* be

$$\begin{array}{ll} f\ [\,] & = 0 \\ f\ (\mathit{xs} : \mathit{xss}) & = w\ (\mathit{xs}, \mathit{concat}\ \mathit{xss}) + f\ \mathit{xss} \ \ . \end{array}$$

Either way, the definition allows us to distribute $(\mathit{xs}:)$ into *minBy*:

$$\mathit{minBy}\ f \cdot \mathit{map}\ (\mathit{xs}:) = (\mathit{xs}:) \cdot \mathit{minBy}\ f \ \ . \tag{1}$$

To find out how to compute *optpart*, we calculate, for non-empty *xs*:

$$\begin{array}{ll} & (\mathit{minBy}\ f \cdot \mathit{parts})\ \mathit{xs} \\ = & (\mathit{minBy}\ f \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{extparts} \cdot \mathit{splits})\ \mathit{xs} \\ = & \quad \{ \text{ since } \mathit{minBy}\ f \cdot \mathit{concat} = \mathit{minBy}\ f \cdot \mathit{map}\ (\mathit{minBy}\ f) \ \} \\ & (\mathit{minBy}\ f \cdot \mathit{map}\ (\mathit{minBy}\ f \cdot \mathit{extparts}) \cdot \mathit{splits})\ \mathit{xs} \\ = & \quad \{ \text{ by (1) and definition of } \mathit{optpart} \ \} \\ & (\mathit{minBy}\ f \cdot \mathit{map}\ (\lambda (\mathit{ys}, \mathit{zs}) \to \mathit{ys} : \mathit{optpart}\ \mathit{zs}) \cdot \mathit{splits})\ \mathit{xs} \\ = & \quad \{ \text{ introduce } g \text{ using (2), see below } \} \\ & ((\lambda (\mathit{ys}, \mathit{zs}) \to \mathit{ys} : \mathit{optpart}\ \mathit{zs}) \cdot \mathit{minBy}\ g \cdot \mathit{splits})\ \mathit{xs} \ \ . \end{array}$$

In the last step we use the property that for all *h* and *k*,

$$\mathit{minBy}\ h \cdot \mathit{map}\ k = k \cdot \mathit{minBy}\ (h \cdot k) \ \ . \tag{2}$$

The new cost function *g*, now defined on splits, is therefore:

$$\begin{array}{l} g :: ([\mathsf{E}],[\mathsf{E}]) \to \mathbb{R} \\ g\ (\mathit{ys}, \mathit{zs}) = f\ (\mathit{ys} : \mathit{optpart}\ \mathit{zs}) \\ \qquad\qquad = w\ (\mathit{ys}, \mathit{zs}) + f\ (\mathit{optpart}\ \mathit{zs}) \ \ , \end{array}$$

assuming that *w* is defined on splits. We have thus derived:

$$\begin{array}{l} \mathit{optpart}\ [\,] = [\,] \\ \mathit{optpart}\ \mathit{xs} = \mathit{ys} : \mathit{optpart}\ \mathit{zs} \\ \quad \textbf{where}\ (\mathit{ys}, \mathit{zs}) = \mathit{minBy}\ g\ (\mathit{splits}\ \mathit{xs}) \ \ . \end{array}$$

The specification of *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 $(\mathit{ys}, \mathit{zs})$, assuming that we already know how to optimally partition *zs*.

Comparing with ordinary optimisation problems, one interesting feature of *optpart* above is that *g* refers to *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 *optpart* for all suffixes of the input.

From now on we denote the input by *inp*. Define such an array *optArr* (for brevity we abbreviate the function *length* to $\#\cdot$):

$$\begin{array}{l} \mathit{optArr} = \mathit{array}\ (0, \#\mathit{inp}) \\ \quad [(\#\mathit{ys}, \mathit{optpart}\ \mathit{ys}) \mid \mathit{ys} \leftarrow ([\,] : \mathit{suffs}\ \mathit{inp})] \ \ . \end{array}$$

The library function *array* builds an immutable, lazy array, whose indices ranges between 0 and $\#\mathit{inp}$. 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 *optpart* for each suffix. The *n*-th entry of *optArr* is the value of *optpart* on the suffix of *inp* of length *n*. We enclose both *optArr* and *optpart* as local definitions, and redefine the latter to fetch entries from the former:

$$\begin{array}{l} \mathit{opt} \quad :: [\mathsf{E}] \to [[\mathsf{E}]] \\ \mathit{opt}\ \mathit{inp} = \mathit{optArr}\ !\ \#\mathit{inp} \qquad \textbf{where} \\ \quad \mathit{optArr} = \quad \{ \text{ same as above } \} \\ \quad \mathit{optpart}\ [\,] = [\,] \\ \quad \mathit{optpart}\ \mathit{xs} = \mathit{ys} : \mathit{optArr}\ !\ (\#\mathit{xs} - \#\mathit{ys}) \end{array}$$

$$\begin{aligned}\textbf{where } \hat{g}\ k\ ys &= \textbf{let } zss = optArr\,!\,(k - \#ys)\\ &\qquad \textbf{in } w\ (ys, concat\ zss) + f\ zss\\ ys &= minBy\ (\hat{g}\ \#xs)\ (prefs\ xs)\ \ .\end{aligned}$$

Compare this definition of *optpart* with the previous one. Instead of calling itself, this *optpart* looks up the corresponding entry in *optArr*. Instead of computing an optimal split, we now compute the optimal prefix of *xs*, under cost function $\hat{g}$. The intended relationship between g and $\hat{g}$ is, for all $ys \mathbin{+\!\!+} zs$ that form a suffix of *inp*,

$$\begin{aligned}\hat{g}\ k\ ys &= g\ (ys, zs) \quad \textbf{where } (\_ \mathbin{+\!\!+} zs) = inp \wedge \#zs = k - \#ys\ \ ,\\ g\ (ys, zs) &= \hat{g}\ \#(ys \mathbin{+\!\!+} zs)\ ys\ \ .\end{aligned}$$

In $\hat{g}$ we no longer need the suffix zs, but instead use a number, the length of $ys \mathbin{+\!\!+} zs$, 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 *optArr* calls *optpart* once and accesses only those entries with indices smaller than its own. If we define

$$optpref = minBy\ (\hat{g}\ \#xs)\ (prefs\ xs)\ \ ,$$

each call to *optpart* calls *optpref* once. If it turns out that *optpref* $(x : xs)$ can be defined in terms of *optpref xs*, that is, *optpref* is a *foldr*, computation of entries of *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 *optpref* $(x : xs)$ in terms of *optpref xs*.

3. Computing Optimal Prefixes Inductively

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

Definition 1 (Prefix Stablility)**.** *A function* $w :: [\mathsf{E}] \to \mathbb{R}$ *is* prefix-stable *if, for all xs and ys,*

$$\begin{aligned}&w\ xs \leqslant w\ (xs \mathbin{+\!\!+} ys) \Rightarrow\\ &\quad (\forall ws :: w\ (ws \mathbin{+\!\!+} xs) \leqslant w\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys))\ \ .\end{aligned}$$

Prefix-stability guarantees that, if *xs* is no worse than $xs \mathbin{+\!\!+} ys$, the optimal prefix of $ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys$ need not extend further to the right than $ws \mathbin{+\!\!+} xs$. The suffix *ys* 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)):

Definition 2 (Concavity (for segments))**.** *A function* $w :: [\mathsf{E}] \to \mathbb{R}$ *is* concave *if, for all ws, xs, and ys,*[1]

$$w\ (ws \mathbin{+\!\!+} xs) + w\ (xs \mathbin{+\!\!+} ys) \leqslant w\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys) + w\ xs\ \ . \qquad (3)$$

To see that concavity implies prefix stability, notice that another way to write (3) is $w\ (ws \mathbin{+\!\!+} xs) - w\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys) \leqslant w\ xs - w\ (xs \mathbin{+\!\!+} ys)$.

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:

[1] If we replace lists in (3) by indices, we get the *Monge* property, proposed by Gaspard Monge in 1971. Furthermore, prefix stability is also called *Monge monotonoicity*. See Galil and Park (1992).

Definition 3 (Concavity (for splits))**.** *A function* $w :: ([\mathsf{E}], [\mathsf{E}]) \to \mathbb{R}$ *is* concave *if, for all ws, xs, ys, and zs,*

$$\begin{aligned}&w\ (\mathbf{ws} \mathbin{+\!\!+} \mathbf{xs}, ys \mathbin{+\!\!+} zs) + w\ (\mathbf{xs} \mathbin{+\!\!+} \mathbf{ys}, zs) \leqslant\\ &\quad w\ (\mathbf{ws} \mathbin{+\!\!+} \mathbf{xs} \mathbin{+\!\!+} \mathbf{ys}, zs) + w\ (\mathbf{xs}, ys \mathbin{+\!\!+} zs)\ \ .\end{aligned} \qquad (4)$$

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

Definition 4. *A function* $\hat{g} :: \mathbb{N} \to [\mathsf{E}] \to \mathbb{R}$ *is prefix-stable if, for all n, xs, and ys,*

$$\begin{aligned}&\hat{g}\ n\ xs \leqslant \hat{g}\ n\ (xs \mathbin{+\!\!+} ys) \Rightarrow\\ &\quad (\forall ws :: \hat{g}\ (n + \#ws)\ (ws \mathbin{+\!\!+} xs) \leqslant\\ &\qquad\quad \hat{g}\ (n + \#ws)\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys))\ \ .\end{aligned}$$

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

Theorem 5. *Given a cost function* $w :: ([\mathsf{E}], [\mathsf{E}]) \to \mathbb{R}$*, 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 *zs* be the suffix of the input having length $n - \#(xs \mathbin{+\!\!+} ys)$ and thus $ys \mathbin{+\!\!+} zs$ is the suffix having length $n - \#xs$. We reason (abbreviating $f \cdot optpart$ to *fo*):

$$\begin{aligned}&\hat{g}\ n\ xs \leqslant \hat{g}\ n\ (xs \mathbin{+\!\!+} ys)\\ \equiv\ & g\ (xs, ys \mathbin{+\!\!+} zs) \leqslant g\ (xs \mathbin{+\!\!+} ys, zs)\\ \equiv\ & w\ (xs, ys \mathbin{+\!\!+} zs) + fo\ (ys \mathbin{+\!\!+} zs) \leqslant w\ (xs \mathbin{+\!\!+} ys, zs) + fo\ zs\\ \equiv\ & w\ (xs, ys \mathbin{+\!\!+} zs) - w\ (xs \mathbin{+\!\!+} ys, zs) \leqslant fo\ zs - fo\ (ys \mathbin{+\!\!+} zs)\\ \Rightarrow\ & \quad \{\ w \text{ concave, by (4) }\}\\ & w\ (ws \mathbin{+\!\!+} xs, ys \mathbin{+\!\!+} zs) - w\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys, zs) \leqslant\\ & \quad fo\ zs - fo\ (ys \mathbin{+\!\!+} zs)\\ \equiv\ & \quad \{\ \text{definition of } g, \hat{g}, \text{etc. }\}\\ & \hat{g}\ (n + \#ws)\ (ws \mathbin{+\!\!+} xs) \leqslant \hat{g}\ (n + \#ws)\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys)\ \ .\end{aligned}$$

□

We are now ready to derive an inductive definition of *optpref xs*, provided that *w* is concave. The base case is omitted. In the calculation for the inductive case below, we abbreviate $\hat{g}\ \#(x : xs)$ to g'. The operator $(\downarrow_{g'})$ denotes binary minimum with respect to g'.

$$\begin{aligned}& minBy\ g'\ (prefs\ (x : xs))\\ =\ & \quad \{\ \text{definitions of } minBy \text{ and } prefs\ \}\\ & [x] \downarrow_{g'} minBy\ g'\ (map\ (x:)\ (prefs\ xs))\\ =\ & \quad \{\ \text{let } xs' = minBy\ (\hat{g}\ \#xs)\ (prefs\ xs) \text{ and } xs' \mathbin{+\!\!+} ys = xs\ \}\\ & [x] \downarrow_{g'} minBy\ g'\ (map\ (x:)\ (prefs\ (xs' \mathbin{+\!\!+} ys)))\\ =\ & \quad \{\ \text{by Lemma 6, see below }\}\\ & [x] \downarrow_{g'} minBy\ g'\ (map\ (x:)\ (prefs\ xs'))\\ =\ & \quad \{\ \text{definition of } minBy, prefs, \text{ and functor laws }\}\\ & minBy\ g'\ (prefs\ (x : xs'))\\ =\ & \quad \{\ \text{definition of } xs'\ \}\\ & minBy\ g'\ (prefs\ (x : minBy\ (g\ \#xs)\ (prefs\ xs)))\ \ .\end{aligned}$$

In the second step, we split *xs* into two parts $xs' \mathbin{+\!\!+} ys$, where xs' is the optimal prefix from the previous step. Lemma 6 below then allows us to get rid of *ys*, which will not be part of the optimal prefix of $x : xs$.

Lemma 6. *Let* $xs, ys :: [\mathsf{E}]$ *be such that*

$$xs = minBy\ (\hat{g}\ \#(xs \mathbin{+\!\!+} ys))\ (prefs\ (xs \mathbin{+\!\!+} ys))\ \ .$$

If $\hat{g}$ *is prefix-stable, we have that for all ws,*

$$\begin{aligned}& minBy\ g'\ (map\ (ws \mathbin{+\!\!+})\ (prefs\ (xs \mathbin{+\!\!+} ys))) =\\ &\quad minBy\ g'\ (map\ (ws \mathbin{+\!\!+})\ (prefs\ xs))\ \ ,\end{aligned}$$

where $g' = \hat{g}\ \#(ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys)$.

Recall $optpref\ xs = minBy\ (\hat{g}\ \#xs)\ (prefs\ xs)$. We have shown that, if w is concave, we have

$$\begin{array}{ll} optpref\ [\,] & = [\,] \\ optpref\ (x:xs) & = minBy\ (\hat{g}\ \#(x:xs))\ (prefs\ (x:optpref\ xs)) \quad . \end{array}$$

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 *optArr*, the expression in the body of *optpref*,

$$minBy\ (\hat{g}\ \#(x:xs)) \cdot prefs \cdot (x:) \quad ,$$

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 *optpref* is almost a fold: it forms a *foldr* together with the function *length*. For notational convenience, we design a variation of *foldr* that passes around a length:

$$\begin{array}{l} foldr_{\#} :: (\mathbb{N} \to (a,b) \to b) \to b \to [a] \to b \\ foldr_{\#}\ f\ e = snd \cdot foldr \\ \quad (\lambda x\ (n,y) \to (1+n, f\ (1+n)\ (x,y)))\ (0,e) \quad , \end{array}$$

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 $cons\ (x,xs) = x:xs$,

$$optpref = foldr_{\#}\ (\lambda n \to minBy\ (\hat{g}\ n) \cdot prefs \cdot cons)\ [\,] \quad .$$

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

Proof. The function *prefs* has the following property:

$$prefs\ (xs \mathbin{+\!\!+} ys) = prefs\ xs \mathbin{+\!\!+} map\ (xs\mathbin{+\!\!+})\ (prefs\ ys) \quad . \qquad (5)$$

To prove the lemma, we reason:

$$\begin{array}{ll} & minBy\ g'\ (map\ (ws\mathbin{+\!\!+})\ (prefs\ (xs' \mathbin{+\!\!+} ys))) \\ = & \quad \{ \text{ by (5) } \} \\ & minBy\ g'\ (map\ (ws\mathbin{+\!\!+})\ (prefs\ xs' \mathbin{+\!\!+} map\ (xs'\mathbin{+\!\!+})\ (prefs\ ys))) \\ = & minBy\ g'\ (map\ (ws\mathbin{+\!\!+})\ (init\ (prefs\ xs') \mathbin{+\!\!+} [xs'] \mathbin{+\!\!+} \\ & \quad map\ (xs'\mathbin{+\!\!+})\ (prefs\ ys))) \\ = & minBy\ g'\ (map\ (ws\mathbin{+\!\!+})\ (init\ (prefs\ xs')) \downarrow_{g'} (ws \mathbin{+\!\!+} xs') \downarrow_{g'} \\ & \qquad minBy\ g'\ ((ws \mathbin{+\!\!+} xs')\mathbin{+\!\!+})\ (prefs\ ys) \\ = & \quad \{ \text{ see below } \} \\ & minBy\ g'\ (map\ (ws\mathbin{+\!\!+})\ (init\ (prefs\ xs')) \downarrow_{g'} (ws \mathbin{+\!\!+} xs')) \\ = & minBy\ g'\ (map\ (ws\mathbin{+\!\!+})\ (prefs\ xs')) \quad . \end{array}$$

For the penultimate step to hold we need Lemma 7. ☐

Lemma 7. *Assume that $\hat{g}$ is prefix-stable. Let $zss :: [[\mathsf{E}]]$ be such that $(\forall zs : zs \in zss : \hat{g}\ n\ xs \leqslant \hat{g}\ n\ (xs \mathbin{+\!\!+} zs))$, then*

$$\begin{array}{l} (ws \mathbin{+\!\!+} xs) \downarrow_{g'} minBy\ g'\ (map\ (ws \mathbin{+\!\!+} xs\mathbin{+\!\!+})\ zss) \\ \quad = (ws \mathbin{+\!\!+} xs) \quad , \end{array}$$

where $g' = \hat{g}\ (n + \#ws)$.

Proof. Induction on xs. ☐

4. The Two-in-Three Property

Recall that

$$optpref = foldr_{\#}\ (\lambda n \to minBy\ (\hat{g}\ n) \cdot prefs \cdot cons)\ [\,] \quad .$$

In Section 5 we will look into the step function of the fold, to develop efficient ways to compute $minBy\ (\hat{g}\ n) \cdot prefs \cdot cons$. In this section we examine properties of the cost function that make the development possible.

Consider $xs = [x_1, x_2, \ldots x_i]$. To choose an optimal non-empty prefix of $x:xs$ under h, we have in general $i+1$ choices to consider ($[x]$, $[x,x_1]$, $\ldots$, $[x, \ldots x_i]$). It would appear that, for each of the $O(n)$ suffixes of *inp*, choosing an optimal prefix takes $O(n)$ time, resulting in an $O(n^2)$ algorithm.

It turns out, for some cost functions, that not all of these prefixes need to be examined. A distinct feature of the algorithm we are about to develop is the use of the following *two-in-three* property.

Definition 8 (2-in-3 (for segments))**.** *A cost function $h :: [\mathsf{E}] \to \mathbb{R}$ is said to have the* two-in-three *property (for segments) if there exists a function $\delta :: [\mathsf{E}] \to \mathbb{R}$ such that*

$$\begin{array}{ll} \delta\ xs \leqslant \delta\ ys \Rightarrow (\forall ws :: h\ ws & \leqslant h\ (ws \mathbin{+\!\!+} xs)\ \vee \\ \qquad h\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys) & \leqslant h\ (ws \mathbin{+\!\!+} xs)) \quad . \end{array}$$

The function δ is called the threshold *function of h.*

Consider again the list $[x_1, x_2, \ldots x_i]$ and some x to be appended to its left. Among the prefixes to consider are $[x]$, $[x,x_1]$, and $[x,x_1,x_2]$. If h has the two-in-three property and $\delta\ [x_1] \leqslant \delta\ [x_2]$, we know that, whatever x is, either $[x]$ or $[x,x_1,x_2]$ will be no worse than $[x,x_1]$ — although we might not know which one (hence the name “two-in-three”). The sublist $[x_1,x_2]$ may thus be taken *atomically*: we either take them both, or drop them together.

Example and Remark An interesting function that is “reversed” two-in-three is the averaging function:

$$avg\ xs = sum\ xs\ /\ \#xs \quad ,$$

where the threshold function is also *avg*. Indeed, we have

$$\begin{array}{l} avg\ xs \leqslant avg\ ys \Rightarrow \\ \quad (\forall ws :: avg\ (ws \mathbin{+\!\!+} xs) \leqslant avg\ ws\ \vee \\ \qquad avg\ (ws \mathbin{+\!\!+} xs) \leqslant avg\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys)) \quad . \end{array}$$

The only difference here is that the order in the conclusion is reversed to $(\leqslant)$. Assume that $avg\ xs \leqslant avg\ ys$ and we are looking for a prefix of $ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys$ having *maximum* average. If *ws* itself has an average larger than *xs*, glueing them together only makes the average smaller. If *ws* has a small or negative average, we get a better average by including *ys* for a larger denominator. Either way, the prefix having maximum average need not be $ws \mathbin{+\!\!+} xs$. That is why the solution of Curtis and Mu (2015) to the *densest segment problem* (finding a segment of the input that has the largest average) adopts an algorithm similar to ours. Curiously, however, *avg* is *not* prefix-stable, meaning that the classical *scanr* based approach does not give us optimal prefixes. Much of the effort of Curtis and Mu (2015) was in fact proving that one can just go ahead, pretending that *avg* is prefix-stable, and the result will still be correct.

The two-in-three property (and its reversed variant) is only interesting when we cannot determine, before having *ws*, which branch of the disjunction is true. Take, for a counter example, $\delta = h = sum$. If $sum\ xs \leqslant sum\ ys$, it is always the case that, for all *ws*, $sum\ (ws \mathbin{+\!\!+} xs) \leqslant sum\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys)$. Technically *sum* is also (reversed) two-in-three. It will be correct, but an overkill, to use our algorithm on the maximum segment sum problem, for which there exist simpler algorithms.

Generalisation to Splits Functions g and δ are defined on splits rather than segments. Thus we generalise two-in-three to splits.

Definition 9 (2-in-3 (for splits)). *A function* $g :: ([\mathsf{E}],[\mathsf{E}]) \to \mathbb{R}$ *is* two-in-three *(for splits) if there exists a* threshold function $\delta\ ([\mathsf{E}],[\mathsf{E}]) \to \mathbb{R}$ *such that*

$$\begin{aligned}&\delta\ (\mathbf{xs}, ys \mathbin{+\!\!+} zs) \leqslant \delta\ (\mathbf{ys}, zs) \Rightarrow \\ &\quad (\forall ws :: g\ (\mathbf{ws}, xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) \leqslant g\ (\mathbf{ws} \mathbin{+\!\!+} \mathbf{xs}, ys \mathbin{+\!\!+} zs) \vee \\ &\qquad\quad g\ (\mathbf{ws} \mathbin{+\!\!+} \mathbf{xs} \mathbin{+\!\!+} \mathbf{ys}, zs) \leqslant g\ (\mathbf{ws} \mathbin{+\!\!+} \mathbf{xs}, ys \mathbin{+\!\!+} zs))\ \ .\end{aligned}$$

Again, we use boldface font for the first arguments of δ 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* $g :: ([\mathsf{E}],[\mathsf{E}]) \to \mathbb{R}$*, if there exists* $\delta :: ([\mathsf{E}],[\mathsf{E}]) \to \mathbb{R}$ *and* $k :: [\mathsf{E}] \to \mathbb{R}$ *such that for all xs, ys, and zs,*

$$\begin{aligned}&g\ (xs, ys \mathbin{+\!\!+} zs) \leqslant g\ (xs \mathbin{+\!\!+} ys, zs) \equiv \\ &\qquad \delta\ (ys, zs) \leqslant k\ (xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs)\ \ ,\end{aligned} \tag{6}$$

then g is two-in-three with δ *as its threshold function.*

Note that δ 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

$$\begin{aligned}&(\forall ws :: g\ (\mathbf{ws}, xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) \leqslant g\ (\mathbf{ws} \mathbin{+\!\!+} \mathbf{xs}, ys \mathbin{+\!\!+} zs) \vee \\ &\qquad g\ (\mathbf{ws} \mathbin{+\!\!+} \mathbf{xs} \mathbin{+\!\!+} \mathbf{ys}, zs) \leqslant g\ (\mathbf{ws} \mathbin{+\!\!+} \mathbf{xs}, ys \mathbin{+\!\!+} zs))\ \ ,\end{aligned}$$

under the assumption that $\delta\ (\mathbf{xs}, ys \mathbin{+\!\!+} zs) \leqslant \delta\ (\mathbf{ys}, zs)$. Consider comparing $k\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs)$ and $\delta\ (xs, ys \mathbin{+\!\!+} zs)$.

Case $\delta\ (xs, ys \mathbin{+\!\!+} zs) \leqslant k\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs)$. By (6) it is equivalent to $g\ (ws, xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) \leqslant g\ (ws \mathbin{+\!\!+} xs, ys \mathbin{+\!\!+} zs)$.

Case $\delta\ (xs, ys \mathbin{+\!\!+} zs) > k\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs)$.

$$\begin{aligned}&\quad \delta\ (xs, ys \mathbin{+\!\!+} zs) > k\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) \\ \Rightarrow &\quad \{\ \text{since } \delta\ (xs, ys \mathbin{+\!\!+} zs) \leqslant \delta\ (ys, zs)\ \} \\ &\quad \delta\ (ys, zs) > k\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) \\ \equiv &\quad \{\ \text{contrvariant of (6)}\ \} \\ &\quad g\ (ws \mathbin{+\!\!+} xs, ys \mathbin{+\!\!+} zs) > g\ (ws \mathbin{+\!\!+} xs \mathbin{+\!\!+} ys, zs)\ \ .\end{aligned}$$

□

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,* δ *and k satisfy* (6) *and k is non-decreasing with respect to prefix, that is,* $k\ xs \leqslant k\ (ws \mathbin{+\!\!+} xs)$ *for all ws and xs, then* $\hat{g}$ *is prefix-stable.*

5. Queueing and Glueing

Let the input be $[x_1, x_2, ... x_n]$. Recall that, if we have $\delta\ ([x_1], ..) \leqslant \delta\ ([x_2], ..)$, 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 $[x_3, x_4]$ are both atomic and $\delta\ ([x_1, x_2], ..) \leqslant \delta\ ([x_3, x_4], ..)$, $[x_1 .. x_4]$ 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 $xss = [xs_1, xs_2 ... xs_j] :: [[\mathsf{E}]]$ such that *concat xss* is the optimal prefix, and all segment in *xss* are atomic. Segments xs_1, xs_2, 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 xs_i and xs_{i+1} if $\delta\ (xs_i, ..) \leqslant \delta\ (xs_{i+1}, ..)$. This continues until the δ 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:

$$foldr'_{\#}\ (\lambda n \to minchop \cdot cons \cdot (id \times prepend))\ [\,]\ \ ,$$

where $foldr'_{\#}$ is a fold similar to $foldr_{\#}$ but defined on lists of segments, *prepend* glues adjacent elements until there is no adjacent xs_i and xs_{i+1} such that $\delta\ (xs_i, ..) \leqslant \delta\ (xs_{i+1}, ..)$, while *minchop* does the work of $minBy\ (\hat{g}\ n) \cdot prefs$, 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 $wrap\ x = [x]$ and note that

$$prefs = map\ concat \cdot prefs \cdot map\ wrap\ \ , \tag{7}$$

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

$$\begin{aligned}&\quad optpref\ xs \\ = &\quad minBy\ (\hat{g}\ \#xs) \cdot prefs\ \$\ xs \\ = &\quad \{\ \text{by (7)}\ \} \\ &\quad minBy\ (\hat{g}\ \#xs) \cdot map\ concat \cdot prefs \cdot map\ wrap\ \$\ xs \\ = &\quad \{\ \text{by (2), let } \hat{gc}\ n = \hat{g}\ n \cdot concat\ \} \\ &\quad concat \cdot minBy\ (\hat{gc}\ \#xs) \cdot prefs \cdot map\ wrap\ \$\ xs\end{aligned}$$

The inner $minBy\ (\hat{gc}\ \#xs) \cdot prefs$ now operate on lists of lists. For easy reference later, we give it a name *optprefS* (where the capital "S" refers to "segments"):

$$optprefS\ xss = minBy\ (\hat{gc}\ \#(concat\ xss))\ (prefs\ xss)\ \ .$$

It can be shown that, for all $h :: [\mathsf{E}] \to \mathbb{R}$, $h \cdot concat$ 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:

$$\begin{aligned}&foldr'_{\#} :: (\mathbb{N} \to ([a], b) \to b) \to b \to [[a]] \to b \\ &foldr'_{\#}\ f\ e = snd \cdot foldr \\ &\quad (\lambda xs\ (n, y) \to (\#xs + n, f\ (\#xs + n)\ (xs, y)))\ (0, e)\ \ ,\end{aligned}$$

we have that

$$optprefS = foldr'_{\#}\ (\lambda n \to minBy\ (\hat{gc}\ n) \cdot prefs \cdot cons)\ [\,]\ \ .$$

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 $\mathsf{A} \to \mathsf{B}$ can be seen as a set of pairs $\{(\mathsf{B}, \mathsf{A})\}$ such that, for every $a :: \mathsf{A}$, there must be a unique $b :: \mathsf{B}$ such that (b, a) is in the set. For example, the function $square :: \mathbb{N} \to \mathbb{N}$ is a set $\{(0,0), (1,1), (4,2), (9,3) ...\}$. A relation $\mathsf{R} :: \mathsf{A} \rightsquigarrow \mathsf{B}$ is a set $\{(\mathsf{B}, \mathsf{A})\}$ without further restrictions: given $a :: \mathsf{A}$ there could be zero or more $b :: \mathsf{B}$ such that $(b, a) \in \mathsf{R}$. When $(b, a) \in \mathsf{R}$ we say that R maps a to b. Relations can be used to model non-deterministic computations: each b such that $(b, a) \in \mathsf{R}$ is a possible output of R on input a.

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

$$\mathsf{R}_1 \supseteq \mathsf{R}_2 = \mathsf{R}_3 \supseteq .. \supseteq \mathsf{R}_n\ \ ,$$

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

$$\mathsf{S} \cdot \mathsf{R}^\star \subseteq \mathsf{T} \;\Leftarrow\; \mathsf{S} \subseteq \mathsf{T} \wedge \mathsf{T} \cdot \mathsf{R} \subseteq \mathsf{T} \ . \tag{8}$$

Finally, with $foldr'_{\#}$ defined previously, we have the following fusion theorem:

$$\begin{array}{l} \mathsf{R} \cdot foldr'_{\#}\, \mathsf{S}\, e \supseteq foldr'_{\#}\, \mathsf{T}\, e' \Leftarrow \\ \quad (\forall n :: \mathsf{R} \cdot \mathsf{S}\, n \supseteq \mathsf{T}\, n \cdot (id \times \mathsf{R})) \wedge (e', e) \in \mathsf{R} \ . \end{array}$$

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

$$\begin{array}{l} (wss', wss) \in glue \equiv \\ \quad (\exists\, xss, xs, ys, yss :: wss \;= xss \mathbin{+\!\!+} [xs, ys] \mathbin{+\!\!+} yss \wedge \\ \qquad\qquad\qquad\qquad\quad wss' = xss \mathbin{+\!\!+} [xs \mathbin{+\!\!+} ys] \mathbin{+\!\!+} yss \wedge \\ \qquad\qquad\qquad\qquad\quad \delta\ (xs, ys \mathbin{+\!\!+} zs) \leqslant \delta\ (ys, zs)) \ , \end{array}$$

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

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

Let us see what properties we have. Define $glue_0 = id \cup glue$ — a relation that might perform one $glue$ or leave the input unchanged. We have that if $hc = h \cdot concat$ for some h, then

$$minBy\ hc \cdot map\ glue_0 = glue_0 \cdot minBy\ hc \ . \tag{9}$$

In words, $glue$ does not change the result of $minBy\ hc$ since for all $(xss', xss) \in glue$ we have $concat\ xss' = concat\ xss$.

We would also like to establish some relationship between $prefs$ and $glue$. Assume that $glue$ maps $wss = xss \mathbin{+\!\!+} [xs, ys] \mathbin{+\!\!+} yss$ to $wss' = xss \mathbin{+\!\!+} [xs \mathbin{+\!\!+} ys] \mathbin{+\!\!+} yss$, where $\delta\ (xs, ys \mathbin{+\!\!+} zs) \leqslant \delta\ (ys, zs)$ (with $zs = concat\ yss$). We have

$$\begin{array}{l} prefs\ wss = prefs\ (init\ xss) \mathbin{+\!\!+} \\ \qquad\qquad\quad [xss, xss \mathbin{+\!\!+} [xs], xss \mathbin{+\!\!+} [xs, ys]] \mathbin{+\!\!+} \\ \qquad\qquad\quad map\ ((xss \mathbin{+\!\!+} [xs, ys]) \mathbin{+\!\!+})\ (prefs\ yss) \ , \\ prefs\ wss' = prefs\ (init\ xss) \mathbin{+\!\!+} \\ \qquad\qquad\quad [xss, xss \mathbin{+\!\!+} [xs \mathbin{+\!\!+} ys]] \mathbin{+\!\!+} \\ \qquad\qquad\quad map\ ((xss \mathbin{+\!\!+} [xs \mathbin{+\!\!+} ys]) \mathbin{+\!\!+})\ (prefs\ yss) \ . \end{array}$$

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

$$\begin{array}{l} (uss', uss) \in rm \equiv (\exists\, wss, xss, xs, ys, yss :: \\ \quad wss \mathbin{+\!\!+} [xss, xss \mathbin{+\!\!+} [xs], xss \mathbin{+\!\!+} [xs, ys]] \mathbin{+\!\!+} yss \wedge \\ \quad wss \mathbin{+\!\!+} [xss, xss \mathbin{+\!\!+} [xs, ys]] \mathbin{+\!\!+} yss \wedge \\ \quad \delta\ (xs, ys \mathbin{+\!\!+} zs) \leqslant \delta\ (ys, zs)) \ , \end{array}$$

where $zs = concat\ yss$, we have that

$$prefs \cdot glue \subseteq map\ glue_0 \cdot rm \cdot prefs \ . \tag{10}$$

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

Finally, if h is two-in-three, we have

$$minBy\ h \cdot rm = minBy\ h \ , \tag{11}$$

since the removed entry $xss \mathbin{+\!\!+} [xs]$ would not have minimum cost.

Now we are ready to introduce $glue^\star$ and promote it into $optprefS$:

$$\begin{array}{cl} & concat \cdot optprefS \\ = & concat \cdot foldr'_{\#}\ (\lambda n \to minBy\ (\hat{gc}\ n) \cdot prefs \cdot cons)\ [\,] \\ = & \quad \{\ concat \cdot glue^\star = concat\ \} \\ & concat \cdot glue^\star \cdot foldr'_{\#}\ (\lambda n \to minBy\ (\hat{gc}\ n) \cdot prefs \cdot cons)\ [\,] \\ \supseteq & \quad \{\ foldr'_{\#} \text{ fusion, see below }\} \\ & concat \cdot foldr'_{\#}\ (\lambda n \to minBy\ (\hat{gc}\ n) \cdot prefs \cdot glue^\star \cdot cons)\ [\,] \ . \end{array}$$

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

$$\begin{array}{l} \quad minBy\ \hat{h} \cdot prefs \cdot glue^\star \cdot cons \cdot (id \times glue^\star) \\ \subseteq glue^\star \cdot minBy\ \hat{h} \cdot prefs \cdot cons \ . \end{array}$$

It is not hard to see that $glue^\star \cdot cons \cdot (id \times glue^\star) \subseteq glue^\star \cdot cons$. We are left with proving that $minBy\ \hat{h} \cdot prefs \cdot glue^\star \subseteq glue^\star \cdot minBy\ \hat{h} \cdot prefs$. 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:

Lemma 12. *$minBy\ \hat{h} \cdot prefs \cdot glue^\star \subseteq glue^\star \cdot minBy\ \hat{h} \cdot prefs$ if $\hat{h}$ is two-in-three.*

Proof. By (8), the proof obligations are:

$$\begin{array}{l} minBy\ \hat{h} \cdot prefs \subseteq glue^\star \cdot minBy\ \hat{h} \cdot prefs \wedge \\ glue^\star \cdot minBy\ \hat{h} \cdot prefs \cdot glue \subseteq glue^\star \cdot minBy\ \hat{h} \cdot prefs \ . \end{array}$$

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

$$\begin{array}{cl} & glue^\star \cdot minBy\ \hat{h} \cdot prefs \cdot glue \\ \subseteq & \quad \{\ \text{by (10)}\ \} \\ & glue^\star \cdot minBy\ \hat{h} \cdot map\ glue_0 \cdot rm \cdot prefs \\ = & \quad \{\ \text{by (9)}\ \} \\ & glue^\star \cdot glue_0 \cdot minBy\ \hat{h} \cdot rm \cdot prefs \\ = & \quad \{\ glue^\star \cdot glue_0 = glue^\star, \text{ and by (11)}\ \} \\ & glue^\star \cdot minBy\ \hat{h} \cdot prefs \ . \end{array}$$

□

Refining to Functions Having just shown that

$$\begin{array}{cl} & glue^\star \cdot optprefS \\ \supseteq & \quad \{\ foldr'_{\#} \text{ fusion }\} \\ & foldr'_{\#}\ (\lambda n \to minBy\ (\hat{gc}\ n) \cdot prefs \cdot glue^\star \cdot cons)\ [\,] \ , \end{array}$$

we may now choose a particular implementation of $glue^\star$. Define

$$\begin{array}{l} prepend :: [[\mathsf{E}]] \to [[\mathsf{E}]] \\ prepend\ [xs] = [xs] \\ prepend\ (xs : ys : xss) \\ \quad \mid \delta\ xs \leqslant \delta\ ys = prepend\ (xs \mathbin{+\!\!+} ys : xss) \\ \quad \mid otherwise \quad = xs : ys : xss \ . \end{array}$$

The function is supposed to be run after $cons$. 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 $prepend \subseteq glue^{\star}$. However, we will see in Section 6 that some choices of δ may lookup *optArr*, and performing *prepend* on the entire input results in a circular dependency. Therefore we use the fact that $cons \cdot (id \times glue^{\star}) \subseteq glue^{\star} \cdot cons$, and refine $glue^{\star}$ to $cons \cdot (id \times glue^{\star})$. In summary, we have shown that

$$\begin{array}{l} glue^{\star} \cdot optprefS \;\supseteq \\ \quad foldr'_{\#}\ (\lambda n \rightarrow minBy\ (\hat{gc}\ n) \cdot prefs \cdot cons \cdot (id \times prepend))\ [\,]\ \ . \end{array}$$

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

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

Some reflection on the use of relations. Rather than introducing *prepend* right in the beginning, we performed a fold fusion with $glue^{\star}$ and refine $glue^{\star}$ to *prepend* afterwards. In a functional development, we would be attempting to fuse $foldr\ (prepend \cdot cons)\ [\,]$ into the fold, and the property corresponding to Lemma 12 is $minBy\ \hat{h} \cdot prefs \cdot prepend = foldr\ (prepend \cdot cons)\ [\,] \cdot minBy\ \hat{h} \cdot prefs$, whose proof would tie us into the particular order *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 *prepend*:

$$xss = [xs_1, xs_2, \ldots xs_i, xs_{i+1}, \ldots xs_k]\ \ ,$$

among whose prefixes we want to find one having minimum cost with respect to $\hat{gc}\ n$. Being results of *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 *prepend* allows us to reduce the number of potential candidates even further.

Assume that $\hat{gc}\ n\ [xs_1, xs_2, \ldots xs_i] > \hat{gc}\ n\ [xs_1, xs_2, \ldots xs_{i+1}]$. Expanding the definitions, we get

$$\begin{array}{l} g\ (xs_1 +\!\!+ \ldots +\!\!+ xs_i,\ xs_{i+1} +\!\!+ \ldots +\!\!+ xs_k) > \\ \quad g\ (xs_1 +\!\!+ \ldots +\!\!+ xs_i +\!\!+ xs_{i+1},\ xs_{i+2} +\!\!+ \ldots +\!\!+ xs_k)\ \ , \end{array}$$

which, by (6), is equivalent to

$$\delta\ (xs_i, xs_{i+1} +\!\!+ \ldots +\!\!+ xs_k) > k\ (concat\ xss)\ \ .$$

Recall that, being an output of *prepend*, the δ values of segments in the list are strictly decreasing. That means we have, for all $1 \leqslant j \leqslant i$,

$$\delta\ (xs_j, xs_{j+1} +\!\!+ \ldots +\!\!+ xs_k) > k\ (concat\ xss)\ \ ,$$

which is in turn equivalent to that

$$\begin{array}{l} \hat{gc}\ n\ [xs_1] > \hat{gc}\ n\ [xs_1, xs_2] > \ldots \\ \quad > \hat{gc}\ n\ [xs_1, xs_2, \ldots xs_j] > \hat{gc}\ n\ [xs_1, xs_2, \ldots xs_{j+1}] \end{array}$$

for all $1 \leqslant j \leqslant i$. That is, the costs of all prefixes shorter than $xs_1 +\!\!+ \ldots +\!\!+ xs_i$ are strictly decreasing when their lengths increase. Therefore, $xs_1 +\!\!+ \ldots +\!\!+ xs_i$ 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 $\hat{gc}\ n\ xss > \hat{gc}\ n\ (xss +\!\!+ [xs])$. When that happens, we know that $xss +\!\!+ [xs]$ has minimum cost and we may stop.

$$\begin{array}{l} minchop\ n\ [xs] = [xs] \\ minchop\ n\ (xss +\!\!+ [xs]) \\ \quad \mid \hat{gc}\ n\ xss \leqslant \hat{gc}\ n\ (xss +\!\!+ [xs]) = minchop\ n\ xss \\ \quad \mid otherwise \qquad\qquad\qquad\qquad\quad = xss +\!\!+ [xs]\ \ . \end{array}$$

$$\begin{array}{l}
opt :: \forall e \cdot (([e],[e]) \rightarrow \mathsf{R}) \rightarrow (\mathbb{N} \rightarrow [e] \rightarrow t) \rightarrow \\
\quad [e] \rightarrow [[e]] \\
opt\ w\ \hat{\delta}\ inp = map\ concat\ (optArr\ !\ \#inp)\ \mathbf{where} \\
\quad optArr :: \mathsf{Array}\ \mathbb{N}\ [\mathsf{Q}\ (\mathsf{J}\ e)] \\
\quad optArr = array\ (0, \#inp)\ [(\#xs, optpart\ xs) \mid xs \leftarrow ([\,] : suffs\ inp)] \\
\quad optpart\ \ :: [e] \rightarrow [\mathsf{Q}\ (\mathsf{J}\ e)] \\
\quad optpart\ [\,] = [\,] \\
\quad optpart\ xs = yss : optArr\ !\ (\#xs - \#(concat\ ys)) \\
\quad\quad \mathbf{where}\ yss = optpref\ xs \\
\quad optpref \qquad\ :: [e] \rightarrow \mathsf{Q}\ (\mathsf{J}\ e) \\
\quad optpref\ [x] \qquad = [[x]] \\
\quad optpref\ (x : xs) = minchop\ (1+n) \\
\qquad\qquad\qquad\qquad ([x] : prepend\ n\ (head\ (optArr\ !\ n))) \\
\quad\quad \mathbf{where}\ n = \#xs \\
\quad prepend \qquad :: \mathbb{N} \rightarrow \mathsf{Q}\ (\mathsf{J}\ e) \rightarrow \mathsf{Q}\ (\mathsf{J}\ e) \\
\quad prepend\ n\ [xs] = [xs] \\
\quad prepend\ n\ (xs : ys : xss) \\
\quad\quad \mid \hat{\delta}\ n\ xs \leqslant \hat{\delta}\ (n - \#xs)\ ys = prepend\ n\ ((xs +\!\!+ ys) : xss) \\
\quad\quad \mid otherwise \qquad\qquad\qquad\ = xs : ys : xss \\
\quad minchop \qquad :: \mathbb{N} \rightarrow \mathsf{Q}\ (\mathsf{J}\ e) \rightarrow \mathsf{Q}\ (\mathsf{J}\ e) \\
\quad minchop\ n\ [xs] = [xs] \\
\quad minchop\ n\ (xss@(yss +\!\!+ [xs])) \\
\quad\quad \mid \hat{gc}\ n\ yss \leqslant \hat{gc}\ n\ xss = minchop\ n\ yss \\
\quad\quad \mid otherwise \qquad\qquad\quad = xss \\
\quad\quad \mathbf{where}\ \hat{gc}\ n = \hat{g}\ n \cdot concat \\
\quad \hat{g} \qquad :: \mathbb{N} \rightarrow \mathsf{J}\ e \rightarrow \mathsf{R} \\
\quad \hat{g}\ k\ ys = f\ (ys : map\ concat\ (optArr\ !\ (k - \#ys))) \\
\quad f\ [\,] \qquad\quad = 0 \\
\quad f\ (xs : xss) = w\ (xs, concat\ xss) + f\ xss
\end{array}$$

Figure 1. The (abstract) queueing-glueing algorithm.

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

$$\begin{array}{l} glue^{\star} \cdot optprefS \;\supseteq \\ \quad foldr'_{\#}\ (\lambda n \rightarrow minchop\ (1+n) \cdot cons \cdot (id \times prepend))\ [\,]\ \ . \end{array}$$

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

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

To compute $optpref\ (x : xs)$, the result of $optpref\ xs$ is fetched from *optArr* since, by definition, $head\ (optArr\ !\ n) = optpref\ n\ xs$ where *xs* is the suffix of *inp* having length n. We therefore form a chain of dependencies. Let n be the length of the input. In the main function *opt* the result is $optArr\ !\ n$, which in turn depends on $optArr\ !\ (n-1)$, etc. The array is thus computed from $optArr\ !\ 0$ back to $optArr\ !\ n$. The function *optpref* calls *prepend* and *minchop* once for each suffix of the input. The function $\hat{\delta}$ in *prepend* is a length-accepting version of δ, to be defined for each individual problem in the Section 6.

5.3 Efficiency Analysis and Summary

It is important that *minchop* processes its input from the right end. Each time $\hat{gc}\ n\ xss \leqslant \hat{gc}\ n\ (xss \mathbin{+\!\!+} [xs])$ holds, $[xs]$ is dropped and will never be accessed in subsequent steps of the algorithm. When we reach a point where $\hat{gc}\ n\ xss > \hat{gc}\ n\ (xss \mathbin{+\!\!+} [xs])$ holds, we stop and need not look into *xss*. Since each segment can be dropped at most once, and there are at most $O(n)$ segments, *minchop* will be called at most $O(n)$ times throughout the algorithm.

On the other hand, *prepend* operates on the left end of the sequence of segments. Each segment could be glued leftwards (that is, being in the role of *ys* in *prepend* $(xs \mathbin{+\!\!+} ys : xss)$) at most once. Therefore, *prepend* could also be called at most $O(n)$ times.

To allow *prepend* and *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 dequeues (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 δ and $\hat{gc}$, etc) we perform on them are glueing ($\mathbin{+\!\!+}$) and conversion back to ordinary lists. Thus they can be represented by join lists:

$$\mathbf{data}\ \mathsf{J}\ a = \mathsf{Singleton}\ a \mid \mathsf{Join}\ (\mathsf{J}\ a)\ (\mathsf{J}\ a) \ .$$

Alternatively we can use Hughes lists ($[a] \to [a]$) (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, δ, and $\hat{gc}$) can be computed in constant time, which can be done if we store the lengths, values of f, 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

$$sp, wt :: \mathsf{Job} \to \mathbb{R} \ .$$

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:

$$\begin{array}{l} bspan :: [\mathsf{Job}] \to \mathbb{R} \\ bspan = (s+) \cdot sum \cdot map\ sp \ . \end{array}$$

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

$$\begin{array}{l} ftime :: [[\mathsf{Job}]] \to \mathbb{R} \\ ftime = sum \cdot map\ bspan \ . \end{array}$$

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:

$$\begin{array}{ll} f & :: [[\mathsf{Job}]] \to \mathbb{R} \\ f\ [\,] & = 0 \\ \multicolumn{2}{l}{f\ (xss \mathbin{+\!\!+} [xs]) = f\ xss + wts * ftime\ (xss \mathbin{+\!\!+} [xs])} \\ \multicolumn{2}{l}{\quad \mathbf{where}\ wts = sum\ (map\ wt\ xs) \ .} \end{array}$$

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 δ. Define:

$$\begin{array}{l} weights :: [[\mathsf{Job}]] \to \mathbb{R} \\ weights = sum \cdot map\ wt \cdot concat \ , \end{array}$$

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

$$f\ (xs : xss) = bspan\ xs * weights\ (xs : xss) + f\ xss \ .$$

It can be verified that $w\ (xs, ys) = bspan\ xs * sum\ (map\ wt\ (xs \mathbin{+\!\!+} ys))$ is concave. To discover δ we reason (abbreviating $f \cdot optpart$ to fo):

$$\begin{array}{l} \quad g\ (xs, ys \mathbin{+\!\!+} zs) \leqslant g\ (xs \mathbin{+\!\!+} ys, zs) \\ \equiv bspan\ xs * weights\ (xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) + fo\ (ys \mathbin{+\!\!+} zs) \leqslant \\ \qquad bspan\ (xs \mathbin{+\!\!+} ys) * weights\ (xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) + fo\ zs \\ \equiv (fo\ (ys \mathbin{+\!\!+} zs) - fo\ zs)\ /\ (bspan\ (xs \mathbin{+\!\!+} ys) - bspan\ xs) \\ \qquad \leqslant weights\ (xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) \\ \equiv (fo\ (ys \mathbin{+\!\!+} zs) - fo\ zs)\ /\ (sum\ (map\ sp\ ys)) \\ \qquad \leqslant weights\ (xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} zs) \ . \end{array}$$

We have thus derived:

$$\delta\ (ys, zs) = (fo\ (ys \mathbin{+\!\!+} zs) - fo\ zs)\ /\ (sum\ (map\ sp\ ys)) \ .$$

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

$$\begin{array}{l} \hat{\delta} \quad :: \mathbb{N} \to [\mathsf{Job}] \to \mathsf{R} \\ \hat{\delta}\ n\ ys = (fc\ (optArr\ !\ n) - fc\ (optArr\ !\ m))\ / \\ \qquad\quad (sumsp\ !\ n - sumsp\ !\ m) \\ \quad \mathbf{where}\ m = n - \#ys \ . \end{array}$$

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

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

i	$optArr\ !\ i$
0	$[\,]$
1	$[[12]] : optArr\ !\ 0$
2	$[[1]] : optArr\ !\ 1$
3	$[[4], [1]] : optArr\ !\ 1$
4	$[[3], [4, 1]] : optArr\ !\ 1$
5	$[[12], [3], [4, 1]] : optArr\ !\ 1$
6	$[[6], [12, 3, 4, 1]] : optArr\ !\ 1$
7	$[[18], [6]] : optArr\ !\ 5$
8	$[[7], [18, 6]] : optArr\ !\ 5$
9	$[[12], [7]] : optArr\ !\ 7$

The first element of $optArr\ !\ i$ 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* $(optArr\,!\,9)$, 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

$$w\ xs = (\mathsf{L} - (sum\ xs + \#xs - 1))^2 \ ,$$

while $f = sum \cdot map\ w$. 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 *w* is concave. To compute $\hat{gc}$ in constant time, we do some calculation to see what should be cached in preprocessing (again, abbreviating $f \cdot optpart$ to *fo*):

$$\begin{aligned}
& g\ (xs, ys) \\
=\ & w\ xs + fo\ ys \\
=\ & (\mathsf{L} - (sum\ xs + \#xs - 1))^2 + fo\ ys \\
=\ & (\mathsf{L} - (sum\ (xs + xs) - sum\ ys + \\
& \qquad \#(xs + ys) - \#ys - 1))^2 + fo\ ys \\
=\ & \quad \{ \text{let } \mathsf{L}' = \mathsf{L} + 1,\ sl\ xs = sum\ xs + \#xs. \} \\
& (\mathsf{L}' - sl\ (xs + ys) + s\ ys)^2 + fo\ ys \ .
\end{aligned}$$

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

To discover δ we reason:

$$\begin{aligned}
& g\ (xs, ys + zs) \leqslant g\ (xs + ys, zs) \\
\equiv\ & (\mathsf{L}' - sl\ (xs + ys + zs) + sl\ (ys + zs))^2 + \\
& fo\ (ys + zs) \leqslant \\
& (\mathsf{L}' - sl\ (xs + ys + zs) + sl\ zs)^2 + fo\ zs \\
\equiv\ & \quad \{ \text{abbreviate } 2 * (sl\ (xs + ys + zs) - \mathsf{L}') \text{ to } \mathsf{U} \} \\
& -\mathsf{U} * sl\ (ys + zs) + (sl\ (ys + zs))^2 + fo\ (ys + zs) \leqslant \\
& -\mathsf{U} * sl\ zs + (s\ zs)^2 + fo\ zs \\
\equiv\ & ((sl\ (ys + zs))^2 + fo\ (ys + zs) - ((sl\ zs)^2 + fo\ zs))\ / \\
& \quad (sl\ (ys + zs) - sl\ zs) \leqslant \mathsf{U} \ .
\end{aligned}$$

Thus we have derived

$$\begin{aligned}
\delta\ (ys, zs) = {} & ((sl\ (ys + zs))^2 + fo\ (ys + zs) - ((sl\ zs)^2 + fo\ zs))\ / \\
& (sl\ (ys + zs) - sl\ zs) \ ,
\end{aligned}$$

whose length-accepting equivalent is

$$\begin{aligned}
& \hat{\delta}\ n\ xs = (sj^2 - sk^2 + fc\ (optArr\,!\,n) - fc\ (optArr\,!\,m))\ /\ (sj - sk) \\
& \quad \mathbf{where}\ \{ sj = sums\,!\,n;\ sk = sums\,!\,m;\ m = n - \#xs \} \ ,
\end{aligned}$$

where the array *sums* stores values of *sl* for each suffix, and *fc*, as in the last section, is conceptually $f \cdot map\ concat$ 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:

$$\begin{aligned}
& f\ [\,] && = 0 \\
& f\ [xs] && = 0 \\
& f\ (xs : xss) && = w\ xs + f\ xss \ ,
\end{aligned}$$

with the same *w* as the previous problem. One can verify that (1) still holds if $(\downarrow_{g'})$ 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

$$optpart = minBy\ f \cdot all\ p \cdot parts \ ,$$

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

$$\begin{aligned}
& g\ (\mathbf{ws} + \mathbf{xs}, ys + zs) \geqslant g\ (\mathbf{ws}, xs + ys + zs) \lor \\
& g\ (\mathbf{ws} + \mathbf{xs}, ys + zs) \geqslant g\ (\mathbf{ws} + \mathbf{xs} + \mathbf{ys}, zs) \ .
\end{aligned}$$

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 $\delta\ (\mathbf{xs}, ys + zs) \leqslant \delta\ (\mathbf{xs} + \mathbf{ys}, zs)$ and

$$\begin{aligned}
& (\forall ws : \neg\ (p\ (ws + xs + ys)) : \\
& \quad g\ (\mathbf{ws} + \mathbf{xs}, ys + zs) \geqslant g\ (\mathbf{ws}, xs + ys + zs)) \ ,
\end{aligned}$$

which by Theorem 10 is equivalent to

$$\begin{aligned}
& (\forall ws : \neg\ (p\ (ws + xs + ys)) : \\
& \quad \delta\ (\mathbf{xs}, ys + zs) \leqslant k\ (ws + xs + ys + zs)) \ .
\end{aligned}$$

Furthermore, we may, for each *xs*, find out the shortest ws_0 that falsifies $p\ (ws_0 + xs + ys)$ by a linear-time preprocessing. If $\delta\ (xs, ys + zs) \leqslant k\ (ws_0 + xs + ys + zs)$ holds, since *k* increases with longer prefixes, the inequality will hold for all prefixes longer than ws_0. The first clause of *prepend* thus becomes:

$$\begin{aligned}
& prepend\ n\ (xs : ys : xss) \\
& \mid \hat{\delta}\ n\ xs \leqslant \hat{\delta}\ (n - \#xs)\ ys\ \land \\
& \quad \delta\ n\ xs \leqslant (2 * (sums\,!\,j - \mathsf{L} - 1)) = prepend\ n\ (xs + ys : xss) \ ,
\end{aligned}$$

where $sums\,!\,j$ is $sl\ (ws_0 + xs + ys + concat\ xss)$. The function *minchop* only need to be extended with one clause that throws away *xs* when $sl\ (xss + [xs]) - 1 > \mathsf{L}$. It can be proved that when *minchop* stops, the segments in the queue still have strictly decreasing values of *g*, and thus *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 *minchop* before *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

R. C. Backhouse. Galois connections and fixed point calculus. In R. C. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 89–148. Springer-Verlag, 2002.

R. S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6(2):159–189, 1986.

R. S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, April 1989.

P. Brucker. Efficient algorithms for some path partitioning problems. *Discrete Applied Mathematics*, 62(1-3):77–85, 1995.

K.-M. Chung and H.-I. Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34(2):373–387, 2004.

S. Curtis and S.-C. Mu. Calculating a linear-time solution to the densest-segment problem. *Journal of Functional Programming*, 25, 2015.

O. de Moor and J. Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. *Science of Computer Programming*, 35(1):3–27, 1999.

Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49–76, January 1992.

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

R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.

R. J. M. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22:141–144, 1986.

M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal on Discrete Mathematics*, 3(1):81–97, 1990.

D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software – Practice and Experience*, 11(11):1119–1184, 1981.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

S. van Hoesel, A. Wagelmans, and B. Moerman. Using geometric techniques to improve dynamic programming algorithms for the economic lot-sizing problem and extensions. *European Journal of Operational Research*, 75(2):312–331, 1994.