Scale Invariant Rank Operators

Jasper van de Gronde André Offringa

June 20, 2013

1 Introduction

Rank operators are often used for filtering images, and can be considered in the framework of mathematical morphology. They are generalizations of the minimum, maximum and median filters, at each position selecting the r-th value in a sorted list of all values within a certain neighbourhood. In [[TODO ref]] we introduced the "scale invariant rank operator" (henceforth we shall say SIR operator) on binary masks, whose values can be zero or one. This is a highly efficient operator that sets to one all intervals (of any size) that contain a sufficient fraction of ones. This can be interpreted as combining the results of rank operators with linear structuring elements of all possible lengths.

The SIR operator is interesting option when one needs to robustly "grow" regions or close gaps, without knowing in advance what size these regions have. Indeed, in [[TODO ref]] we showed that this can work quite well in practice. However, some issues remained. First of all, the operator could only be applied to binary images, and was limited to 1D signals (rows/columns of images were processed independently). Here we first analyse the operator in more detail, then show how it can be applied to greyscale images (efficiently), and then we show how the SIR operator can be linked to path dilations, relaxing the 1D requirement.

2 Definition and properties

Given a signal $f \in \mathcal{P}(E)$ (with E a chain), the scale invariant rank (SIR) operator ρ can be defined (for $0 \leq s \leq 1$) by

$$\rho_s\{f\} = \bigcup \{[a,b\rangle \mid a,b \in E,\, a \leq b \text{ and } \mu(f \cap [a,b\rangle) \geq s\,\mu([a,b\rangle)\}\,. \tag{1}$$

Note that if E has a largest element, this definition would never set it, but this can easily be fixed by adding another element to E that is larger still. To "count" values this uses a measure μ on subsets of E, which is usually quite easy to define. In particular, if E is the set of all integers $\mathbb{Z}$, we can just define $\mu(S)$ as giving the number of elements in S (for all finite subsets). [[TODO (?) I want some condition that essentially says that the only sets with non-zero measure are intervals and their (countable) unions and intersections.]]

[[TODO This kind of view could be interesting for connecting to hyperconnections. Also, it could be connected to eq. 4 in "Connectivity on Complete Lattices": the connected opening (almost like a "Fuzzy" generalization.]] More generally, we can define (with C a connection class[[TODO ref]])

$$\rho_s\{f\} = \bigvee \{c \mid c \in C \text{ and } \mu(f \wedge c) \geq s\,\mu(c)\}\,. \tag{2}$$

[[TODO I'm not sure going directly to a hyperconnection makes sense, but I might be able to construct a (hyper)connection using a closing(/opening) by reconstruction, using the SIR operator to compute the marker.]]

We will now state some properties of the operator ρ. Note that the subscript s is used only when needed for clarification.

Lemma 1. *The operator ρ is scale invariant, in the sense that if $h : E \to E$ is increasing and a bijection, and d a positive real number such that $\mu(h(S)) = d\,\mu(S)$ for any measurable subset S of*

E, then $\rho_s\{h(f)\} = h(\rho_s\{f\})$. Here h applied to a set is to be interpreted as being applied to all members of the set.

Proof. First note that since h is a bijection, we have that $a \in E \implies h^{-1}(a) \in E$ as well. Now the lemma follows from eq. (1) and some rewriting:

$$\begin{aligned}\rho_s\{h(f)\} &= \bigcup \{[a,b\rangle \mid a,b \in E,\ a \le b \text{ and } \mu(h(f) \cap [a,b\rangle) \ge s\,\mu([a,b\rangle)\} \\ &= \bigcup \left\{[a,b\rangle \mid a,b \in E,\ a \le b \text{ and } \mu(h(f) \cap h([h^{-1}(a), h^{-1}(b)\rangle) \ge s\,\mu(h([h^{-1}(a), h^{-1}(b)\rangle)\right\},\end{aligned}$$

and by replacing a and b by $h^{-1}(a)$ and $h^{-1}(b)$:

$$\begin{aligned}\rho_s\{h(f)\} &= \bigcup \{h([a,b\rangle) \mid a,b \in E,\ a \le b \text{ and } \mu(h(f) \cap h([a,b\rangle)) \ge s\,\mu(h([a,b\rangle))\} \\ &= \bigcup \{h([a,b\rangle) \mid a,b \in E,\ a \le b \text{ and } \mu(h(f \cap [a,b\rangle)) \ge s\,\mu(h([a,b\rangle))\},\end{aligned}$$

making use of the fact that $\mu(h(S)) = d\,\mu(S)$ and $d > 0$:

$$\begin{aligned}\rho_s\{h(f)\} &= \bigcup \{h([a,b\rangle) \mid a,b \in E,\ a \le b \text{ and } \mu(f \cap [a,b\rangle) \ge s\,\mu([a,b\rangle)\} \\ &= h(\rho_s\{f\}).\end{aligned}$$

This concludes the proof. □

Lemma 2. *The operator ρ is increasing: $f \subseteq g \implies \rho\{f\} \subseteq \rho\{g\}$.*

Proof. Assume that $f \subseteq g$, we then prove that any element of $\rho\{f\}$ must also be an element of $\rho\{g\}$. Note that for every $x \in \rho\{f\}$ there is, by definition, an interval $[a,b\rangle$ such that $x \in [a,b\rangle$ and $\mu(f \cap [a,b\rangle) \ge s\,\mu([a,b\rangle)$. Since $g \supseteq f$, we have that $\mu(g \cap [a,b\rangle) \ge \mu(f \cap [a,b\rangle)$. So, if x is in $\rho\{f\}$, then x is also in $\rho\{g\}$, which lets us conclude that indeed $f \subseteq g \implies \rho\{f\} \subseteq \rho\{g\}$. □

Lemma 3. *The operator ρ is "measure extensive": $\mu(f \setminus \rho\{f\}) = 0$.*

Proof. [[TODO This requires that the only sets with non-zero measure are intervals (and their unions/intersections).]] If $\mu(f \setminus \rho\{f\}) > 0$, then there must be some interval $[a,b\rangle$ with non-zero measure such that $\mu((f \setminus \rho\{f\}) \cap [a,b\rangle) = \mu([a,b\rangle) = \mu(f \cap [a,b\rangle)$ [[TODO might be good to mention that the last equality holds because of the first, the fact that $f \supseteq f \setminus \rho\{f\}$, and the fact that the measure cannot be larger than the measure of the interval.]] [[TODO This formulation depends a bit on how the measure is defined.]], thus (for all $s \in [0,1]$) [[TODO Consider how this would work with a Smith-Volterra-Cantor set... Worst-case, restrict it to countable sets (where singletons have non-zero measure), then the operator is plain-old extensive (which is something I may want to mention anyway).]]

$$\mu(f \cap [a,b\rangle) = \mu([a,b\rangle) \ge s\,\mu([a,b\rangle).$$

So if $\mu(f \setminus \rho\{f\}) > 0$, then there exists an interval in $\mu(f \setminus \rho\{f\})$ that satisfies the condition in eq. (1), implying that this interval should have been present in $\rho\{f\}$. This leads to a contradiction, and we can thus conclude that $\mu(f \setminus \rho\{f\})$ must be zero. □

Lemma 4. *The parameter s is inversely proportional to the strength of the operator ρ_s, in the sense that $s \le t \implies \rho_s\{f\} \ge \rho_t\{f\}$, with $\rho_0\{f\} = E$.*

Proof. This follows immediately from the condition $\mu(f \cap [a,b\rangle) \ge s\,\mu([a,b\rangle)$ in eq. (1), which is easier to satisfy for smaller s, in particular

$$s \le t \text{ and } \mu(f \cap [a,b\rangle) \ge t\,\mu([a,b\rangle) \implies \mu(f \cap [a,b\rangle) \ge s\,\mu([a,b\rangle).$$

When $s = 0$ we get

$$x \in \rho_0\{f\} \iff \exists a,b \in E\,(x \in [a,b\rangle \text{ and } \mu(f \cap [a,b\rangle) \ge 0)\,,$$

which is trivially satisfied, due to the non-negativity of measures. This proves the lemma. □

Finally, note that the dual of a traditional rank operator is again a rank operator. This is not true for the SIR operator, as it is always extensive, and the dual of an extensive operator has to be anti-extensive.

3 Computation

For an efficient computation of the scale invariant rank operator we first manipulate the last part of the condition in eq. (1) a bit[[TODO Make this match the definition above.]]:

$$\begin{aligned}
&\mu(f \cap [a,b]) \geq s\,\mu([a,b]) \\
\iff &\mu(f \cap [a,b]) \geq s\left(\mu(f \cap [a,b]) + \mu(f^C \cap [a,b])\right) \\
\iff &(1-s)\mu(f \cap [a,b]) - s\,\mu(f^C \cap [a,b]) \geq 0.
\end{aligned}$$

Now suppose we have a sampled, finite-length, signal $f(x) : \{1,2,3,\cdots,n\} \mapsto \{0,1\}$ and $x \in f \iff f(x) = 1$. We then get (shortening the condition in eq. (1) a bit for brevity)

$$\begin{aligned}
\rho_s\{f\} &= \bigcup \{[a,b] \mid \mu(f \cap [a,b]) \geq s\,\mu([a,b])\} \\
&= \bigcup \left\{[a,b] \mid (1-s)\mu(f \cap [a,b]) - s\,\mu(f^C \cap [a,b]) \geq 0\right\} \\
&= \bigcup \{[a,b] \mid W([a,b]) \geq 0\}
\end{aligned}$$

Where

$$W([a,b]) = \sum_{y \in [a,b]} w(f(y)), \quad w(0) = -s\,\mu(\{y\}) \text{ and } w(1) = (1-s)\,\mu(\{y\}).$$

We can clearly compute all the prefix sums $M(x) = \sum_{y<x} w(f(y))$ in linear time using a simple recurrence relation. We then get:

$$x \in \rho\{f\} \iff \left[\max_{a,b|a\leq x\leq b} M(b+1) - M(a)\right] \geq 0.$$

Furthermore, we can independently optimize a and b, giving:

$$x \in \rho\{f\} \iff \left[\max_{b \geq x} M(b+1)\right] - \left[\min_{a \leq x} M(a)\right] \geq 0. \tag{3}$$

Clearly the maximum and minimum in this equation can be easily computed for all elements in linear time (similar to computing the prefix sums). This means that we can compute $\rho\{f\}$ in linear time. This is essentially the algorithm presented in [[TODO cite A Morphological Algorithm For Improved Radio Frequency Interference Detection]].

3.1 New method

The method discussed above works, but we will now present a method that requires less temporary storage, as well as fewer passes over the data (two instead of three). It is also extremely easy to implement and slightly more general. As a bonus, it shows how the problem we are solving is *very* closely related to the maximum contiguous subsequence sum problem[[TODO reference]], as essentially the same algorithm can be used to solve both. The only down-side is that in its most basic form it is not completely equivalent to the original algorithm (but we will show that the discrepancy is very small, and can be mitigated).

First, observe that instead of eq. (3) we can also use

$$\begin{aligned}
x \in \rho\{f\} \iff &\left[\max_{a \leq x} \sum_{y \in [a,x]} w(f(y))\right] + \left[\max_{b \geq x} \sum_{y \in [x,b]} w(f(y))\right] - w(f(x)) \\
&= L_f(x) + R_f(x) - w(f(x)) \geq 0.
\end{aligned} \tag{4}$$

Here $R_f(x) = L_{f'}(x')$, for some x' and with f' the reverse of f, so if we have an algorithm for $L(x)$ we can also use it for $R(x)$ (the subscripts are dropped whenever context allows it).

Clearly, $L(x)$ can be considered to be zero for $x < 1$ (before the start of the sequence f). Also, if we know $L(x-1)$, then $L(x)$ can be either $L(x-1) + w(f(x))$ or $w(f(x))$, whichever is higher. This leads to algorithm 1, which is almost exactly the same as algorithm 4 presented by Bentley

Algorithm 1 Compute L.

$L \leftarrow 0$
for i=1 to n **do**
  $L[i] \leftarrow \max(0, L[i-1]) + w(f[i])$
end for

[2] for solving the maximum subsequence sum problem. The main differences are that we are not interested in the globally maximal subsequence sum and that we do not consider empty intervals, leading us to omit the `MaxSoFar` variable and to put $w(f[i])$ outside the max.

We can now also compute $R_f(x) = L_{f'}(n+1-x)$, with $f'(x') = f(n+1-x')$, and then $\rho\{f\}$ using $L(x) + R(x) - w(f(x))$ as prescribed by eq. (4). This leads to algorithm 2, which is essentially equivalent to the earlier algorithm based on eq. (3), and even a bit more expensive, as we sum everything twice. However, if it is advantageous to trade memory for a bit of arithmetic, we can easily merge the computation of R with the computation of $\rho\{f\}$, giving an algorithm that only needs the L array for temporary storage, and two passes over the data instead of three. Still, the real interest in the algorithm developed here is in what happens when we try to *independently* compute $\rho\{f\}$ based on L and R, and then try to merge the results.

Algorithm 2 Binary SIR operator.

Compute L and R according to algorithm 1.
$\rho\{f\} \leftarrow$ **false**
for i=1 to n **do**
  $\rho\{f\}[i] \leftarrow (L[i] + R[i] - w(f[i]) \geq 0)$
end for

We can alter algorithm 1 to immediately output a binary mask based on the computed sums, as in algorithm 3. If we then denote the array r corresponding to L by r_L, and similarly for R, then clearly $\rho^-\{f\} = r_L \cup r_R \subseteq \rho\{f\}$. However, as theorem 1 shows, errors only occur sporadically.

Algorithm 3 Single pass of approximate binary SIR operator ρ^-.

$r \leftarrow 0$
$credit \leftarrow 0$
for i=1 to n **do**
  $credit \leftarrow \max(0, credit) + w(f[i])$
  $r[i] \leftarrow (credit \geq 0)$
end for

[[TODO Reminder that we are assuming $E = \mathbb{Z}$.]]

Theorem 1. *For any $x \in \rho\{f\} \setminus \rho^-\{f\}$ we have that $x-1 \in \rho\{f\} \cap \rho^-\{f\}$ and $x+1 \in \rho\{f\} \cap \rho^-\{f\}$.*

Proof. If x is in $\rho\{f\}$ and not in $\rho^-\{f\} = r_L \cup r_R$, we know that $w(f(x)) < 0$, that $L(x) < 0$ and $R(x) < 0$. But we also have that *both* $L(x-1)$ and $R(x+1)$ are positive, otherwise $L(x) = w(f(x))$ and/or $R(x) = w(f(x))$ and we would not have needed *both* $L(x)$ and $R(x)$ to come to a positive score. As both $L(x-1)$ and $R(x+1)$ must be positive $x-1$ and $x+1$ will be in $\rho\{f\}$, as well as in $r_L \cup r_R$. This concludes the proof. □

[[TODO I'm not sure about the relevance of the material in this section after theorem 1]]
[[TODO Demote some theorems to lemmas or corollaries.]]
[[TODO Reminder that n is the number of elements in our finite-length signal.]]

Theorem 2. *The number of elements $|e|$ in $e = \rho\{f\} \setminus \rho^-\{f\}$ is bounded from above by $|e| \leq (2/k)|f|$, with $k = \lceil (1-eta)/eta \rceil$, and by $|e| \leq n/(1+2\kappa+k/2)$, with $\kappa = \lfloor eta/(1-eta) \rfloor$ (also see fig. 1).*

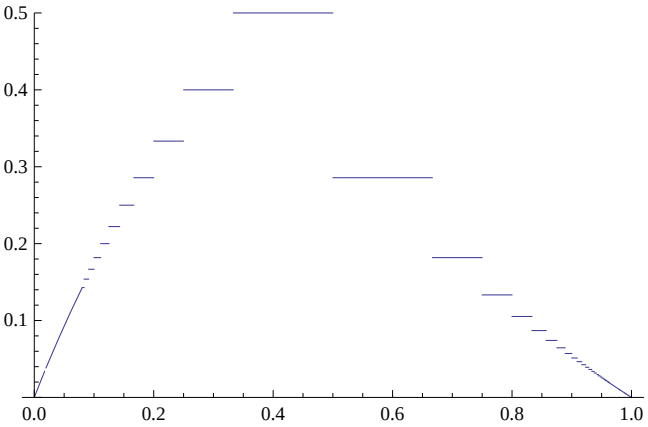


Figure 1: This plots the relative amount of errors $|e|/n = 1/(1 + 2\kappa + k/2)$ against η. The maximum is $|e|/n = 1/2$ for $1/3 \leq \eta < 1/2$.

Proof. To see that the first upper bound is correct, consider a bipartite graph with the elements of f and e as nodes, with an edge between an element x from f and an element x' from e if x is involved in the sum $L(x') + R(x') - w(f(x'))$. Clearly, for every element in e there must be at least $k = \lceil (1 - eta)/eta \rceil$ edges to elements in f. Also, every element in f can only "contribute" to at most two elements in e (one on the left and one on the right), as the error elements necessarily lie at positions where we "restart" the optimal interval in algorithm 1. This means that the number of edges in the graph is bounded from below by $k\,|e|$ and from above by $2\,|f|$. We can now infer that $k\,|e| \leq 2\,|f|$, or equivalently that $|e| \leq (2/k)|f|$. As we trivially have that $|e| + |f| \leq n$, some simple rearranging gives $|e| \leq n/(1 + k/2)$. [[TODO Drawing to illustrate the construction of the graph?]]

For the second upper bound, we use that every error location has to its left and right at least one element from f and $\kappa = \lfloor eta/(1 - eta) \rfloor$ empty positions in between, otherwise L and R would not (both) be negative at the error position (and non-negative at either side). So we have that $(2\kappa + 1)|e| + |f| \leq n$, and thus that $|e| \leq n/(1 + 2\kappa + k/2)$. This concludes the proof. □

Theorem 3. *If $a\eta = b\,(1 - \eta)$ for integer a and b, then algorithm 3 is correct if operated on a signal that has been upsampled by a factor of a. That is, $a^{-1}\rho^{-}\{af\} = \rho\{f\}$ for any f. [[TODO Specify that upsampling should be done by "replication".]]*

Proof. In the upsampled signal, every $x \in f$ is represented by a consecutive elements. Thus whenever we encounter an empty position *credit* is an integer multiple of $(1 - \eta)$. As we deduct $(1 - \eta)$ at each empty position, *credit* remains an integer multiple of $(1 - \eta)$ and we have $L(x) < 0 \implies L(x) \leq \eta - 1$ (and similarly for R). Thus $L(x) < 0$ and $R(x) < 0 \implies L(x) + R(x) - w(f(x)) \leq 2(\eta - 1) - (\eta - 1) = \eta - 1 < 0$, at least for $\eta < 1$ (and if $\eta = 1$ the algorithm is trivially correct). [[TODO Make a little neater.]] □

If we wish to compute an approximation of $\rho\{f\}$ that is a superset instead of a subset, we can slightly change algorithm 3, as in algorithm 4. This works because *credit* corresponds to either $L(x)$ or $R(x)$, and for any element $x \in e$ we must have that $L(x) + R(x) \geq (\eta - 1)$ while both $L(x)$ and $R(x)$ are negative. So if we assume (without loss of generality) that $L(x) \geq R(x)$, then $L(x) + L(x)$ must certainly be larger than or equal to $(\eta - 1)$, giving us $L(x) \geq \frac{1}{2}(\eta - 1)$.

[[TODO Analyze when algorithm 4 makes mistakes.]]

4 Greyscale

[[TODO Different name? Technically the below is probably true for any chain instead of just $\mathbb{R}$.]] The binary case discussed above can be solved quite efficiently by reducing it to finding locally maximal subsequences. However, this clearly does not generalize to the greyscale case just by using different weights for different values, as we would not be able to distinguish between a large

Algorithm 4 Single pass of approximate binary SIR operator ρ^+.

```
r ← new bool[n]
credit ← 0
for i=1 to n do
  credit ← max(0, credit) + w(f[i])
  r[i] ← (credit ≥ ½(η − 1))
end for
```

interval full of small weights and a small interval of large weights. It turns out we can still compute the greyscale SIR operator efficiently though.

In words, the greyscale SIR operator modifies a signal in such a way that at each position we get the highest value for which there is an interval around that position such that at least a certain percentage of that interval has a value greater than or equal to the value in question. From this description it should be clear that the greyscale SIR operator is equivalent to the binary SIR operator applied to all the level sets of a signal. Clearly, if we can find an efficient way of applying the binary SIR operator to all level sets simultaneously, we are in business.

If we use algorithm 3 and denote the credit for value v at position i by $\text{credit}_v(i)$, then we have (with $0 \leq \eta \leq 1$)

$$\begin{aligned} \text{credit}_v(0) &= 0 \\ \text{credit}_v(i+1) &= \max(0, \text{credit}_v(i)) + \begin{cases} \eta & v \leq f(i+1) \\ \eta - 1 & \text{otherwise.} \end{cases} \end{aligned} \tag{5}$$

So clearly, $v \leq w \implies \text{credit}_v(i) \geq \text{credit}_w(i)$. In other words, $\text{credit}_v(i)$ is order reversing in v.

Now imagine that for position i we have built a balanced binary search tree on all values encountered so far, along with their associated credits. As $\text{credit}_v(i)$ is order-reversing in v this tree can also be used to find v^*, the highest value such that its credit is non-negative. Starting at the root and assuming the tree is ordered from left to right (on v), it is sufficient to check whether the credit for the current node is below zero, if it is, we need to look for smaller v, so we start over with the left branch (if there is one, otherwise we are done). Otherwise, v^* is set to the value of the current node and we start over with the right branch (if there is one, otherwise we are done). Note that v^* is always assigned to at least once (as the credit for the current value is always non-negative), and any subsequent assignments always make it larger.

Unfortunately, naively updating such a search tree with the new credits at each position would require $O(n^2)$ time for the entire sequence. So although actually finding the values to output only requires $O(n \log(n))$ time, updating the tree required to enable this would seem to be prohibitively expensive. Fortunately, it is not necessary to update *all* credits at each position, we can defer updating credits until we need to visit the nodes they are stored in, essentially making the updating step no less efficient than a simple insertion.

First imagine that instead of eq. (5) we have the following update rule (so without clamping):

$$\begin{aligned} \text{credit}_v(0) &= 0 \\ \text{credit}_v(i+1) &= \text{credit}_v(i) + \begin{cases} \eta & v \leq f(i+1) \\ \eta - 1 & \text{otherwise} \end{cases}. \end{aligned} \tag{6}$$

Now, in addition to storing a credit, let every node keep track of how much credit it "missed", c_v. Upon accessing a node v, we update its credit to $\text{credit}_v + c_v$, propagate c_v to v's children (this is not considered "accessing" the children) and finally set c_v to zero. Now, if we are inserting/updating a value v', then if $v' \leq v$, add $\eta - 1$ to c for the right child of v, and if $v' \geq v$, add η to c for the left child of v. (Note that it is possible that both children get a value updated.) Only after all this we access a child of v (if needed).

The above procedure is clearly correct if we use eq. (6), but we want to use eq. (5). The trick to fixing this is to make sure we never have values with negative credits in the tree; in that case eq. (6) has the same effect as eq. (5). To accomplish this, after we have found v^* we remove all

larger values from the search tree. The exact details will vary depending on the search structure, but in general it should be possible to do this in $O(n \log(n) + n)$ amortized time. [[TODO The main problem in showing that this is true is that we'd need to show that either it's okay to just replace nodes by their left child whenever we'd otherwise make a left turn (which might violate the structure of the tree) or that there are trees which support deleting a range of length m in $O(\log(n) + m)$ time. The latter should be easy (the C++ standard even requires it from the set data structure), but I haven't found a good reference yet.]] [[TODO One example of a tree that would be suitable is the scapegoat tree as described by Galperin and Rivest [4], this data structure supports a sequence of n insertions and n deletions with $O(\log(n))$ amortized cost per insertion/deletion. (The only reason this is in a todo is that it almost seems too good to be true, but so far I have not found any holes in the analysis, the tree really seems to be almost ideal for this use case.)]] [[TODO Another option might be to use a (deterministic) skip list. Deleting a suffix in a skip list is pretty straightforward.]]

Note that the above has ignored the rotations that might be necessary to keep the search tree balanced. This is because it is sufficient to "access" the nodes involved in the rotation before performing the rotation. Typically most of the nodes will have already been accessed anyway, and otherwise it only adds a constant amount of time to each rotation, so it does not alter the time complexity in any way. [[TODO Something similar holds for scapegoat trees.]]

We started out using algorithm 3, but this algorithm is in general only suited for an approximation of the binary SIR operator. We can modify the greyscale algorithm discussed above so that it becomes exact, without sacrificing its performance (much). The key is to keep track of the credit with which values are removed from the tree during the first pass. As temporary storage we keep an array E of length n, such that $E[i]$ is a [[TODO list/stack/set?]] of values that were removed from the tree at position i. With each value $v \in E[i]$ we associate the "exit credit" $credit_v[i]$ [[TODO using w is not a good idea]] (if we want to be thrifty we can not store the value if its exit credit is $\leq eta - 1$, which should never happen). During the second pass we either use this information on nodes we are removing to see if we should pick a higher value at the current position [[TODO how?]], or we also keep track of these exit credits and combine them in another pass.

[[TODO A third option is to essentially maintain the tree for all positions. This is easier than it sounds, and should not affect the time-complexity.]]

5 Limiting scale invariance

In practice we often deal with streams of data that come in blocks. If we simply apply the SIR operator per block we (unintentionally) limit its scale invariance, and make its behaviour depend on the position in a block. Also, in some cases we might explicitly *want* the operator to be limited to a certain range of scales. For example, we might know that it is impossible (or at least highly unlikely) to have features larger than some predefined size. These issues call for a version of the SIR operator that is limited in its scale invariance in a controlled manner.

The easiest way of limiting the scale invariance of the SIR operator is by limiting how far from the current position we look in eq. (3) or eq. (4). Then both optimizations in that equation could be performed using a normal 1D erosion/dilation, for which many efficient algorithms exist. If we take the algorithm proposed by Dokládal and Dokládalová [3] as an example, we can even modify it slightly to avoid having to keep a running sum for the entire sequence (which becomes problematic if we wish to apply this algorithm to a stream of data).

[[TODO W vs. $W \pm 1$]] First consider $L(x)$ in eq. (4) with a finite "horizon", this can be implemented with a dilation. The basic idea of Dokládal and Dokládalová [3] is to keep a (double-ended) queue of values within range of the structuring element, discarding "useless values". In this case this translates to keeping a queue of the values of $S(a, x) = \sum_{y \in [a,x]} w(f(y))$ for $x - W \leq a \leq x$, discarding those values $S_a(x)$ such that there is also an $S(a', x) \geq S(a, x)$ in the queue for some $a' > a$. Clearly this ensures that the values in the queue are decreasing, and that the first element of the queue is $\max_{x-W \leq a \leq x} S(a, x)$.

To avoid having to add $w(f(x))$ to every element in the queue, we can keep track of the jumps between the consecutive positions in the queue. Take the queue to consist of the positions

$a_1, \cdots, a_m$ and jumps $j_1, \cdots, j_m$, such that $a_i < a_{i+1}$ and $S(a_i, x) = S_i = \sum_{j=i}^{m} j_m$. Clearly, to have decreasing S_i, all jumps except j_m must be positive.[[TODO The indices and limits involved in the sum seem wrong.]] Before we add a position $x > a_m$ and its associated jump, we might need to remove some elements from the back of the queue. If $j_m \leq 0$, then adding the new position would cause the sequence of sums S_i to no longer be decreasing, so position a_m must be removed. If we are removing a_m and $m > 1$, then we need to add j_m to j_{m-1}. This process of removing the last position and adding its associated jump to the jump immediately before it, continues until the last jump is positive or the queue is empty. Only after all this do we add the new position and jump to the queue. Also, if $a_1 < x - W$, remove it from the queue. At all times we keep track of the total of all jumps in the queue.

As $R(x)$ just mirrors $L(x)$ we can use the same method as just described to compute it with a finite horizon. On the other hand, it might make sense to combine the two in one pass. We then have a queue of positions and jumps as above, except that $S(a_i, x) = S_i = \sum_{j=1}^{i} j_m$. We still want all jumps except the last to be positive, but instead of merging a non-positive jump with the one before it, we merge it with the new jump associated with $x + W$.[[TODO Clarify]]

Instead of limiting the "horizon", we can choose to limit the scale invariance of the SIR operator by limiting the size of the intervals we consider. It should be possible to do this rigorously in a way similar to the above, but we would like to argue that this makes little sense. Let r_L and r_R denote the result obtained using the above algorithms for L and R with finite horizon. Let r_H be the exact result obtained by limiting the horizon to W, and r_I the result based on limiting the size of intervals to $W + 1$. We can then see that $r_L \cup r_R \subseteq r_I \subseteq r_H$. Given the simplicity of limiting the horizon we feel that this is the best choice.

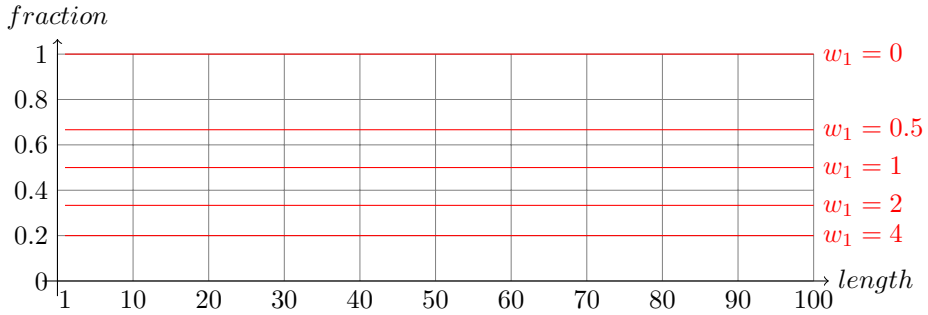
6 Path operators

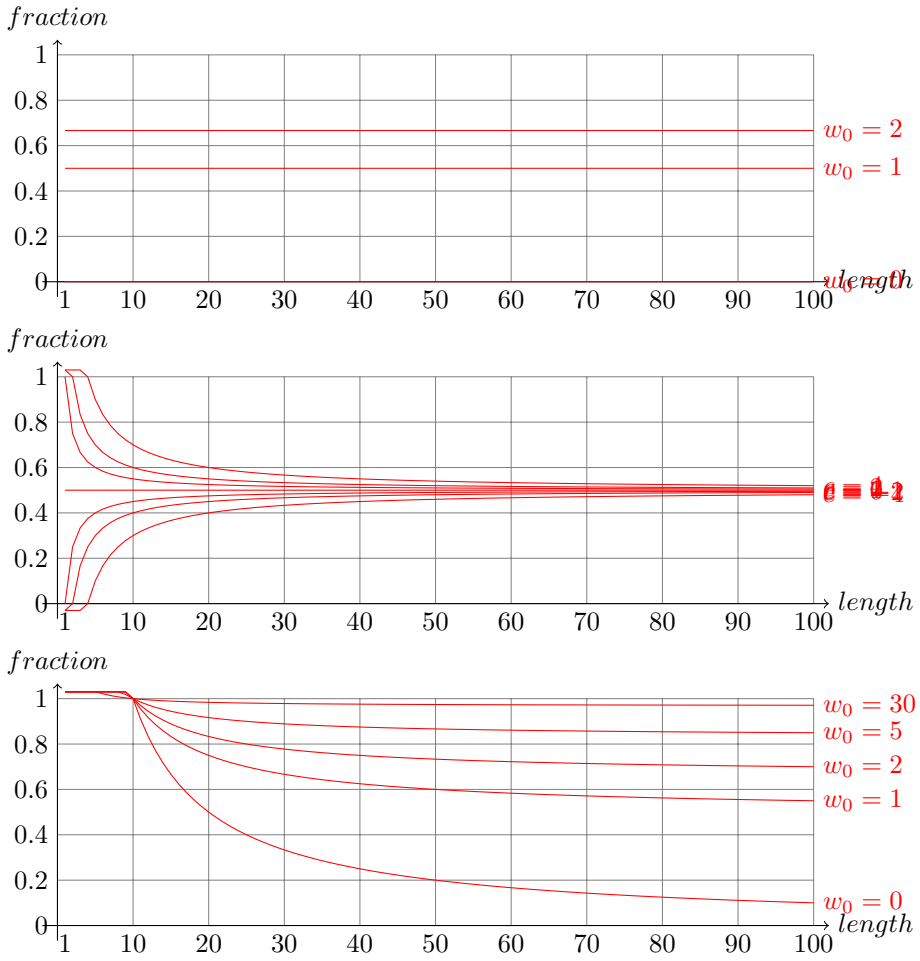
When generalizing the SIR operator to higher dimensional spaces, it is not entirely clear what shapes should be used instead of intervals, let alone whether an efficient algorithm exists. However, one option is to keep using 1D shapes, just not constrained to lie on a straight line. This gets us into the realm of path operators, like path openings and closings.

The trick is now to apply the SIR operator using the same kinds of connectivity as the traditional path opening, rather than just processing images row-by-row. [[TODO It is not too difficult to see that in principle the same algorithms work for the SIR operator as for the path opening, except that we cannot "cap" the length (for a path opening of a specific size we need not consider paths longer than the given size).]] Unfortunately, these algorithms do not have a very satisfactory time complexity. Perhaps it is possible to generalize the approach used above.

$$\text{pathopen}_L\{f\} = \bigcup \{[a, b] \mid \mu(f \cap [a, b]) - \infty\, \mu(f^C \cap [a, b]) \geq L\}$$

$$\text{gen}_L\{f\} = \bigcup \{[a, b] \mid w_1\, \mu(f \cap [a, b]) - w_0\, \mu(f^C \cap [a, b]) \geq c\}$$





7 Generalized scoring functions

Dynamic programming At each position, decide whether to continue or to restart.

Alternatively: No need to subdivide a previously found "optimal" run. If we need to add a previously "aborted" run, then we only need to add it in its entirety. And later we will never have to subdivide it again.

The above is based on finding subsequences whose sum is maximal. However, what if instead of a sum we took some other scoring function? Could we still use the above methods to efficiently compute (locally) maximal subsequences? Essentially the above is a classic application of dynamic programming and Bellman's *principle of optimality* [1]: "An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions."

In our case, we use this principle in two different ways. First of all, when computing the best interval ending or starting at the current position, we rely on the fact that we can decide this locally. Put differently, if we already have the optimal interval for the previous position, then the optimum for the current position can be determined from this and the value at the current position. Secondly, the principle is effectively stretched a bit by making it work "both ways": the optimal interval is the optimal left interval combined with the optimal right interval.

$$a \circ b \leq b \iff a \circ b \circ c \leq b \circ c$$

$$a \circ b \leq b \iff a \circ b \circ c \leq b \circ c$$

This hinges on two properties of the scoring function: the ability to optimize both ends independently, and the ability to efficiently compute those independent optimizations by locally determining whether to expand or restart the interval. Both can be seen to depend upon the property[[TODO Is this true?]]:[[TODO Explain notation.]]

1. $S(u) \leq S(\epsilon) \implies S(uv) \leq S(v)$ and $S(v) \leq S(\epsilon) \implies S(uv) \leq S(u)$
2. $S(u) \geq S(u') \implies S(uv) \geq S(u'v)$ and $S(v) \geq S(v') \implies S(uv) \geq S(uv')$.

For simplicity we assume that $S(A) = S(A')$, with A' being the reverse of A. [[TODO Motivativate necessity, if it is necessary (it is likely slightly stronger than strictly necessary).]]

[[TODO Rewrite to allow empty sets?]]

Algorithm 5 Compute L for (more) general scoring function S.

$L[0] \leftarrow \emptyset$
  $\{\ S(L[0]) = \max_{a|a\leq 0} S(K[a \ldots 0])\ \}$
for i=1 to n **do**
    $\{\ S(L[i-1]) = \max_{a|a\leq i-1} S(K[a \ldots i-1]),$
      and thus, by property 1.:
      $S(L[i-1]K[i]) = \max_{a|a\leq i-1} S(K[a \ldots i-1]K[i])\ \}$
  if $S(L[i-1]K[i]) > S(K[i])$ **then**
    $L[i] \leftarrow L[i-1]K[i]$
  else
    $L[i] \leftarrow K[i]$
  end if
    $\{\ S(L[i]) = \max\left(\max_{a|a\leq i-1} S(K[a \ldots i-1]K[i]), S(K[i])\right) = \max_{a|a\leq i} S(K[a \ldots i])\ \}$
end for

Algorithm 6 Generalized locally maximal scoring subsequences.

Compute L and R according to algorithm 5.
for i=1 to n **do**
    $\{\ S(L[i]) = \max_{a|a\leq i} S(K[a \ldots i])$ and $S(R[i]) = \max_{b|i\leq b} S(K[i \ldots b])\ \}$
  $m, R' \leftarrow R[i]$ $(*$ strip off $K[i]$, which is in both L and R $*)$
  $M[i] \leftarrow L[i]R'$
    $\{$ by property 1., $S(R') = \max_{b|i\leq b} S(K[i+1 \ldots b])$[1] and
      $S(M[i]) = S(L[i]R') = \max_{a|a\leq i} S(K[a \ldots i]R) = \max_{a,b|a\leq i\leq b} S(K[a \ldots b])\ \}$
end for

Theorem 4. *If $S(K) = \sum_{i=1}^{n} K_i + T(n)$, then $T(n)$ has to be linear for $S(K)$ to satisfy property 1..*

Proof. [[TODO]] □

Theorem 5. $S(K) = \sum_{i=1}^{n} w(i)K_i$

Theorem 6. $S(K) = K_1 \bullet \cdots \bullet K_n$

Theorem 7. $S(K) = a_1T_1(K) + a_2T_2(K)$

Theorem 8. $S(K) = a_1(n)T_1(K) + a_2(n)T_2(K)$

References

[1] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[2] Jon Bentley. Programming pearls: algorithm design techniques. *Communications of the ACM*, 27:865–873, September 1984. ISSN 0001-0782. doi: 10.1145/358234.381162.

[3] Petr Dokládal and Eva Dokládalová. Computationally efficient, one-pass algorithm for morphological filters. *Journal of Visual Communication and Image Representation*, 22(5):411–420, July 2011. ISSN 10473203. doi: 10.1016/j.jvcir.2011.03.005.

[4] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 165–174, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics. ISBN 0-89871-313-7.