The Roadmaker's algorithm for the Discrete Pulse Transform

Dirk P. Laurie

Department of Mathematical Sciences
University of Stellenbosch

20 July 2009

Abstract

The Discrete Pulse Transform (DPT) is a decomposition of an observed signal as a sum of pulses, where a pulse is defined as a signal that is constant on a connected set and zero elsewhere. The DPT is by definition obtained via successive application of LULU operators (members of a class of minimax filters studied by C.H. Rohwer). The Roadmaker's algorithm is an efficient way of finding those same pulses, with the aid of operators that act selectively only on those features actually present in the signal.

The original LULU operators act in theory on bi-infinite sequences, though in practice only a finite number of differences must be nonzero. Such a sequence can be seen as a graph in which each member of the sequence is a node, and there is an arc to its successor and its predecessor. The Roadmaker's algorithm is here described in the more general setting of a real-valued function defined on a graph. Doing so has many advantages: the DPT itself becomes a function defined on a tree; the algorithm is easy to visualize; an efficient implementation is possible by merging nodes in the graph; generalizations from the one-dimensional case to the two- and three-dimensional cases, which are important in image processing, are obtained simply by the way in which the arcs of the graph are specified; and the so-called Highlight Conjecture, stated as an open problem in 2006, can be proved.

We pay particular attention to algorithmic details and complexity, including a demonstration that in the one-dimensional case, and also in the case of a complete graph, the Roadmaker's algorithm runs in time $\mathcal{O}(m)$, where m is the number of arcs in the graph.

1 Introduction: the DPT for sequences

The Discrete Pulse Transform (DPT) [6] is a certain well-defined decomposition of a finite sequence $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ as a sum of pulses:

$$\mathbf{x} = \sum_{j=1}^{p} h_j \mathbf{z}_j. \tag{1}$$

Each unit pulse $\mathbf{z}_j$ is a sequence of the same length as $\mathbf{x}$, whose entries equal zero everywhere except on a single contiguous stretch where they equal 1. The number of ones is the *size* of the pulse $h_j\mathbf{z}_j$, and h_j is its *height*. The sign of a pulse is the sign of its height.

The two main applications of the DPT are:

- Recovery of signals made up of a small number of pulses. This is done by selecting the pulses from the DPT according to a criterion appropriate to the application; for example, the sign and approximate size of the pulses in the signal might be specified.
- Filtering of data to remove high-frequency and impulsive noise. This is done by omitting the pulses smaller than some application-dependent size.

The bulk of this introduction contains no new work, but consists of a summary of relevant aspects of the original DPT, as given in [5] and [6]. The length of the introduction is justified by the different point of view, and the fact that the rest of this paper relies heavily on the exposition.

There are of course numerous ways of achieving a decomposition conforming to (1), but the DPT as defined in [6] is the output of a particular algorithm, which makes it well-defined. Strictly speaking, one should even in the case of that algorithm speak of ‘a DPT’ instead of ‘the DPT’, because the algorithm is parametrized by a sequence $\mathbf{p}$ of binary digits (which I here call the *policy* of the DPT), and all the theoretical properties about the DPT proved in [6] require that the policy be fixed in advance. However, in practice the policy almost always consists either of only zeros (the *floor* policy) or of only ones (the *ceiling* policy). One picks one of these two to fix the focus of the discussion, and there is a (usually obvious) way of modifying the result for the other, as well as a (sometimes less obvious) way of modifying the result for any policy. Here, I choose the ceiling policy for this purpose.

We start with so-called LULU filters, introduced by Carl Rohwer in 1989 and developed in a series of papers culminating in the monograph [5]. There are two atomic LULU filters, L_w and U_w, of any given window size w for $w = 1, 2, 3, \ldots$. These are defined by

$$\begin{aligned}(U_w\mathbf{x})_i &= \min_{k=0,1,\ldots,w} \max_{j=0,1,\ldots,w} x_{i+j-k},\\ (L_w\mathbf{x})_i &= \max_{k=0,1,\ldots,w} \min_{j=0,1,\ldots,w} x_{i+j-k}.\end{aligned} \tag{2}$$

The LULU operators have been generalized in various ways:

- Anguelov and Rohwer [1] give an analogue for real-valued functions of a real variable.
- An axiomatic treatment of LULU operators on $\mathbb{Z}^n$, in which graph connectivity is the most important concept, is given by Fabris-Rotelli and Anguelov [2].

A survey of how LULU filters fit into the bigger picture of multiresolution analysis is given by Rohwer and Wild [7].

The LULU filters in principle act on bi-infinite sequences $(x_j : j \in \mathbb{Z})$ rather than finite sequences, and some decision must be made on how to treat the boundaries. In [6] any index $i + j - k$ that falls outside the range $1, 2, \ldots, n$ is simply omitted; this is equivalent to the assumption that $x_j = x_1,\ j < 1$ and $x_j = x_n, j > n$.

Except near the endpoints, it takes $2w$ comparisons to obtain one filtered value directly from the definition. For a sequence of length $n \gg w$, this amounts to approximately $2wn$ comparisons. An idea of Rohwer and L.M. Toerien, implemented by the latter but not published in the open literature, exploits the overlap between adjacent windows to reduce the operation count for one LULU filter of length w to $2n + \mathcal{O}(w)$ comparisons.

For the purpose of the DPT, the LULU filters are ranked as follows:

- When two filters are of unequal window size, the smaller filter precedes the larger.
- When two filters are of equal window size, the policy determines precedence: $p_w = 1$ means "U_w precedes L_w" and $p_w = 0$ means "L_w precedes U_w."

The algorithm in [6] consists of applying the filters in order of precedence. In the case of the ceiling DPT, one computes

$$\begin{aligned} C_0\mathbf{x} &= \mathbf{x}; \\ C_w\mathbf{x} &= L_w U_w (C_{w-1}\mathbf{x}),\ w = 1, 2, 3, \ldots. \end{aligned} \tag{3}$$

For example, when conforming to the ceiling policy, after four stages the sequence has been transformed to $L_2U_2L_1U_1\mathbf{x}$, hence the name LULU.

For other policies, for some values of w one would use U_wL_w instead of L_wU_w. If $C_w\mathbf{x}$ and $C'_w\mathbf{x}$ have been obtained by policies $\mathbf{p}$ and $\mathbf{p}'$ respectively, and $\mathbf{p} \prec \mathbf{p}'$ in the sense of lexicographic comparison, then $C_w\mathbf{x} \leqslant C'_w\mathbf{x}$ term-by-term.

The difference $C_{w-1}\mathbf{x} - C_w\mathbf{x}$, called the w-th *layer* of the decomposition, can be further separated into half-layers:

$$\begin{aligned} H^-_w\mathbf{x} &= C_{w-1}\mathbf{x} - U_wC_{w-1}\mathbf{x}; \\ H^+_w\mathbf{x} &= U_wC_{w-1}\mathbf{x} - C_w\mathbf{x}. \end{aligned} \tag{4}$$

The following facts are proved in [6]. Note that the first two of these allow the decomposition into pulses to be read off from the half-layers.

1. $H_w^-\mathbf{x}$ is a sum of pulses of size w and negative height, such that there are at least w zeros between any two pulses.

2. $H_w^+\mathbf{x}$ is a sum of pulses of size w and positive height, such that there are at least $w+1$ zeros between any two pulses.

3. No two pulses in the DPT overlap partially. Either the support of the smaller pulse is contained in the support of the larger pulse, or the pulses have disjoint support.

4. The total variation of the sequence $\mathbf{x}$, defined as

   $$V(\mathbf{x}) = \sum_{i=1}^{n-1} |x_{i+1} - x_i|,$$

   is conserved by the DPT, i.e.

   $$V(\mathbf{x}) = \sum_{j=1}^{p} |h_j| V(\mathbf{z}_j).$$

The algorithm in [6] does not actually continue to a complete decomposition into pulses, but stops when a non-increasing or non-decreasing sequence (the *trend*) is obtained. The trend depends on the policy; for example, let $\mathbf{x} = (1,1,2,1,2,2)$, then

$$L_1\mathbf{x} = (1,1,1,1,2,2),$$
$$U_1\mathbf{x} = (1,1,2,2,2,2).$$

Since these sequences are monotone, the algorithm stops there. Thus the floor DPT would find the positive pulse $(0,0,1,0,0,0)$ but no negative pulses and the ceiling DPT would find the negative pulse $(0,0,0,-1,0,0)$ but no positive pulses. Often the floor and ceiling DPT's are the same (in that case, the pulses in both half-layers are separated by $w+1$ zeros), but when they differ, the floor policy prefers positive to negative pulses, and vice versa for the ceiling policy.

If it is desired to complete the decomposition of the trend into pulses, one simply prepends x_n or appends x_1 to the sequence $\mathbf{x}$: this forces the trend to be constant. For the floor policy, one would choose the extra element to be the smaller of x_1 and x_n; for the ceiling policy, the larger.

The computation of the DPT up to C_m via the LULU filters requires the application of U_w and L_w for $w = 1,2,\ldots,m$, and therefore takes approximately $4nw + \mathcal{O}(w^2)$ comparisons when using Toerien's implementation.

This can be improved to $\mathcal{O}(n)$ comparisons, which is optimal, by using the so-called Roadmaker's algorithm [4]. Unfortunately [4], which is an extended abstract in a volume of conference proceedings, only gives the complexity analysis, saying "In the full paper we prove that the roadmaker's algorithm is equivalent to the iterated LULU algorithm, in the sense that exactly the same DPT is obtained. Thus any implementation of the roadmaker's algorithm is a way of calculating the DPT." No such full paper was actually written.

Since the Roadmaker's algorithm is easier to understand in the more general setting presented in Section 2, I omit further details at this stage.

The following conjecture is made but not proved in [6].

> **Highlight conjecture.** Given a basis of pulses identified by the DPT, form a new function by any linear combination of those pulses *with heights of the same sign as before*. Then the DPT applied to the new function will identify the same basis of pulses, and recover the new heights.

In [4], it is stated that "The technique described in this talk leads to a proof of the highlight conjecture, but we will not give that proof here."

The present paper paper makes the following contributions:

- It fills the gaps in the extended abstract [4]: the Roadmaker's algorithm is shown to be equivalent to the LULU-defined DPT, and the Highlight conjecture is proved.
- The derivation and proofs are given for the general graph-theoretic case, which subsumes the one-dimensional case, since a sequence can be seen as a graph with arcs $(i, i+1),\ i = 1, 2, \dots, n-1$. The multidimensional case can be obtained almost as simply.
- Considerable attention is given to issues of implementation and complexity. It is shown that the Roadmaker's algorithm has optimal complexity in certain simple cases, including $\mathcal{O}(n)$ complexity in the one-dimensional case.

The bulk of the paper was written after I had attended a short talk by the first author of [2] but before I had a chance to read the manuscript. Some of the concepts and results in Section 2 overlap with [2], although my terminology is quite different. Unlike [2], I do not require that the nodes of the graph can be mapped to a compact subset of the d-dimensional lattice $\mathbb{Z}^d$; however, this property seems never to be actually used in any of the proofs in [2], and their results would therefore remain valid for general graphs. It would be possible to make use of their work in the exposition of the next section, but to keep the present paper self-contained and relatively short, I have not attempted to do so. This paper is not as rigorous as [2] and in some cases, e.g. $(U_w\mathbf{x})_i \geqslant x_i$, I have used as self-evident a statement for which a detailed proof can be found there.

2 LULU operators and the DPT on a graph

As before, the generalized DPT is seen as a decomposition of a finite sequence $\mathbf{x} = (x_1, x_2, \dots, x_n)$ as a sum of pulses:

$$\mathbf{x} = \sum_{j=1}^{p} h_j \mathbf{z}_j,$$

but the semantics is different. The sequence $\mathbf{x}$ is thought of as a real-valued function defined on a connected undirected graph with n nodes, tagged by the numbers $1, 2, \dots, n$. Note that the fact that the node tags are natural numbers is only used for the purpose of storing them as an array (computerese for "finite sequence"). We refer to $\mathbf{x}$ as the value function, and to x_j as the *value* of the node j.

The neighbours of a node are specified by the arcs of the graph. We denote the arc connecting nodes i and j by (i, j).

The term *cluster* of size w is used for a connected subgraph containing w nodes. A unit pulse is a value function equal to 1 on some cluster and to 0 elsewhere. The notions of height and sign are the same as before.

A neighbour of a cluster C is a non-member of C that is connected to a member. We say that an arc (a, b) in the graph is a *border arc* of C if a belongs to C and b is a neighbour of C.

In the course of the construction of the DPT, reduced graphs arise in which all the nodes in a constant-valued cluster have been merged into one node. The *size* of this node is the size of the cluster represented by it. When comparing by value, we say that a node is *greater* or *less* than another; by size, *larger* or *smaller*. We never compare by the numerical value of the tag.

The LULU operators of size w are defined as follows:

$$(U_w\mathbf{x})_i = \min_{W:i\in W} \max_{j:j\in W} x_j,$$
$$(L_w\mathbf{x})_i = \max_{W:i\in W} \min_{j:j\in W} x_j,$$

where W is any cluster of size $w + 1$. In the one-dimensional case, these definitions reduce to what we had before, provided that the implied extra elements at the boundary are supplied explicitly. An efficient way of doing so is pointed out when nodes that represent clusters are discussed in the next section.

With the new definitions of the LULU operators, the ceiling DPT is then defined by (3) and the half-layers by (4) exactly as in the case of a sequence.

For these generalized definitions, it takes $w(N(w,i)-1)$ comparisons to obtain $(U_w\mathbf{x})_i$ directly from the definition, where $N(w,i)$ is the number of clusters of size $w+1$ that contain i. If we make the unrealistic assumption that it is possible to store and retrieve the intermediate maxima over the clusters at no cost, it takes $N(w,i)+w-1$ comparisons. It is not obvious how to exploit overlap between clusters in the absence of additional information on the structure of the graph, and I will not in this paper address the question of the efficient evaluation of LULU operators on a graph in general.

However, the graphs arising by the repeated application of LULU operators and contraction of clusters are intimately related, and we will find that the structure arising in the process can be exploited to obtain the Roadmaker's algorithm. In order to understand this structure, we define some concepts relating to a value function on a graph.

The *sign* of the arc (a,b) is defined as the sign (-1, 0 or 1) of $x_a - x_b$, and we say the arc is negative, neutral or positive, respectively. We can interpret the undirected graph as a directed acyclic graph (DAG), with an arc (i,j) being directed from node i to node j if it is positive. If the arc (i,j) is neutral, there is no corresponding arc in the DAG.

We say that a constant-valued cluster F of size w is a *feature* of $\mathbf{x}$ if in the DAG, the border arcs of F are either all outgoing or all incoming. These cases are respectively called a *bump* and a *pit*. Any neighbour m for which $|y-x_m| = \min_l |y-x_l|$, where y is the constant value on F and l runs through the neighbours, is said to be a *nearest neighbour* of F. The *height* of F is defined as $h = y - x_m$.

The value function $\mathbf{x}$ is said to be of *monotonicity class* (b,p) if it has no bumps of size $\leqslant b$ and no pits of size $\leqslant p$.

The following theorem is the key to the understanding of the DPT and to the derivation of the Roadmaker's algorithm.

Theorem 1 *Let (b,w) be the monotonicity class of $\mathbf{x}$, and let $\mathbf{y} = U_w\mathbf{x}$. If i belongs to a pit P of size w, then $y_i = x_m$, where m is any nearest neighbour of P, otherwise $y_i = x_i$.*

Proof. We first consider the case where $i \in P$. By definition, $x_m = \min_{l\in M} x_l$, where M is the set of neighbours of P. Any cluster W of size $w+1$ containing i must contain at least one neighbour k of P. Therefore

$$\max_{j\in W} x_j \geqslant x_k \geqslant x_m.$$

Since the equality case occurs, namely when $W = P \cup \{m\}$,

$$y_i = \min_{W:i\in W} \max_{j:j\in W} x_j = x_m.$$

For the general case, we prove the contrapositive: if $y_i \neq x_i$, then i belongs to a pit of size w.

Since $(U_w\mathbf{x})_i \geqslant x_i$, $y_i \neq x_i$ means $y_i > x_i$, implying that any cluster of size $w+1$ that contains node i also contains at least one node strictly greater than i. In fact, this is true for any cluster of size $> w$, since we can always delete nodes until a cluster of size $w+1$ is left.

Therefore the largest cluster C containing the node i, such that C has no nodes strictly greater than i, must have size $\leqslant w$. Now pick $k \in C$ so that $x_k = \min_{j \in C} x_j$ and let K be the largest constant-valued cluster containing k. Clearly K is a pit. Moreover, $K \subseteq C$ since any path from k to a node outside C must contain a border node of C, but those are all greater than node i and therefore also greater than k. By the monotonicity of $\mathbf{x}$, K is of size $\geqslant w$, but C is of size $\leqslant w$. Therefore K is a pit of size w and equals C. But $i \in C$, therefore i belongs to a pit of size w. ■

By symmetry, we have:

Theorem 2 *Let (w, p) be the monotonicity class of $\mathbf{x}$, and let $\mathbf{y} = L_w\mathbf{x}$. If i belongs to a bump B of size w, then $y_i = x_m$, where m is any nearest neighbour of B, otherwise $y_i = x_i$.*

It follows by induction that the value function $C_w\mathbf{x}$ defined by (3) has monotonicity class (w, w) and $U_{w+1}C_w\mathbf{x}$ has monotonicity class $(w+1, w)$. The analogues of the four properties given above are:

1. $H_w^-\mathbf{x}$ is a sum of pulses of size w and negative height, and has monotonicity class (w, w).

2. $H_w^+\mathbf{x}$ is a sum of pulses of size w and positive height, and has monotonicity class $(w, w+1)$.

3. No two pulses in the DPT overlap partially. Either the support of the smaller pulse is contained in the support of the larger pulse, or the pulses have disjoint support.

4. The total variation of the sequence $\mathbf{x}$, defined as

$$V(\mathbf{x}) = \sum_{(a,b)} |x_a - x_b|,$$

where the sum runs over all arcs (a, b), is conserved by the DPT, i.e.

$$V(\mathbf{x}) = \sum_{j=1}^{p} |h_j| V(\mathbf{z}_j).$$

In the notation of Theorem 1, the height of the pulse at pit B is $x_i - x_m$. The proofs of the above properties are short and simple enough to be left to the reader.

Property 3 implies that the DPT can be represented as a tree. This is important for the efficient implementation discussed below.

Property 4 can be stated in the equivalent form:

4a. For a given arc (a, b), let $v(\mathbf{x}, a, b) = x_a - x_b$. Then

$$x_a - x_b = \sum_{j=1}^{p} h_j v(\mathbf{z}_j, a, b),$$

and all the nonzero terms in the sum have the same sign as $x_a - x_b$.

3 The Roadmaker's algorithm

The Roadmaker's algorithm is conceptually very simple: construct directly a decomposition into pulses that has the properties of the DPT given above. In effect, these properties, rather than the algorithm in terms of the LULU operators, are taken as the definition of the DPT.

We define operators P_w and Q_w on any sequence $\mathbf{x}$ as follows:

- $P_w\mathbf{x} = \mathbf{y}$, where $y_i = x_m$ if i belongs to a pit of size w with nearest neighbour m, otherwise $y_i = x_i$.
- $Q_w\mathbf{x} = \mathbf{y}$, where $y_i = x_m$ if i belongs to a bump of size w with nearest neighbour m, otherwise $y_i = x_i$.

In general, these operators are not identical to U_w and L_w. By Theorems 1 and 2, however, P_w and Q_w have the same effect as U_w and L_w when restricted to monotonicity classes (b, w) and (w, p) respectively. If we replace U_w by P_w and L_w by Q_w in (3), by Properties 1 and 2 we will obtain the same DPT.

A crucial property of the operators P_w and Q_w is that they can be implemented by treating one feature at a time. That is to say, an algorithm for replacing $\mathbf{x}$ by $P_w\mathbf{x}$ could be:

```
For every pit of size $w$, replace $x_i$ by $x_m$, where $i$ is in the pit and
$x_m$ is a nearest neighbour.
```

To see this, consider any border arc (i, j) where i is in a pit and j is a border node. Then $x_i < x_j$. It follows that j is not a member of any pit. Therefore no pit is adjacent to the nodes that have been changed, and the order in which the pits are filled is immaterial.

Since the operators P_w and Q_w respectively act on only pits and bumps of width w, and leave everything else unchanged, they do much less work than the operators U_w and L_w, which act on everything. Of course, this is only true if we can keep track of where those features are.

Note first that the original graph structure is not needed once the procedure is underway; in fact, it carries excess baggage. It is much simpler to contract neutral arcs as they arise, so that the graph keeps getting smaller, in terms of nodes as well as arcs, during the course of the algorithm. In the process, multiple arcs between the same two nodes may arise, and these can also be combined into a single arc.

To keep track of the fact that nodes and arcs in the derived graphs represent multiple nodes and arcs of the original graph, we need some enrichments of the graph structure.

- With each node a, we associate a positive integer w_a giving the size of the cluster represented by the node. Normally, every node has initial size 1, although cases like the endpoints of a sequence, which are imagined to be bordered by many equal-valued nodes, can be modelled by using a high initial size.
- With each arc (a, b), we associate a positive integer $\kappa(a, b)$ known as its *capacity*. Initially, every arc has capacity 1; when contraction of some arc causes two other arcs to coincide, the capacity of the arc thus formed is the sum of the capacities of those two arcs. The capacity of an arc is not actually needed for the algorithm to work, but appears in the generalized definition of the total variation of x on the the graph, namely

  $$V(\mathbf{x}) = \sum_{(a,b)} \kappa(a, b)|x_a - x_b|,$$

  where (a, b) runs through all the arcs. This quantity is preserved at every step.

In this setting, a feature is simply a node with only incoming or only outgoing arcs, and at least one neighbour. A feature is *flattened* by replacing its value by the value of its nearest neighbour. We say a feature is more *urgent* than another if by the policy of the DPT, it should be flattened first.

At the highest level, this is the Roadmaker's algorithm:

1. Contract all neutral arcs in the graph.
2. Repeat until there are no more features:
   (a) Flatten the most urgent feature and record the corresponding pulse.
   (b) Contract all neutral arcs around that feature.

Since each iteration removes at least one node, there can be no more than $n - 1$ iterations.

Note that in Step 1, the reduced graph is the same irrespective of the order in which the arcs are contracted; and in Step 2, when two features are equally urgent, the order in which they are flattened does not matter since they are both pits or both bumps, and therefore cannot be neighbours of each other.

The Roadmaker's algorithm can be used to prove the following theorem, stated for the one-dimensional case as a conjecture in [6], and described in the multi-dimensional case as "still an open problem" in [2].

Theorem 3 (Highlight Theorem) *Let the DPT of* $\mathbf{x}$ *according to a certain policy be*

$$\mathbf{x} = \sum_{j=1}^{p} h_j \mathbf{z}_j,$$

and suppose that $\mathbf{y}$ *is formed as*

$$\mathbf{y} = \sum_{j=1}^{p} h_j' \mathbf{z}_j, \tag{5}$$

with the same unit pulses $\mathbf{z}_j$ *and where each* h_j' *is nonzero and of the same sign as* h_j*. Then* (5) *is the DPT of* $\mathbf{y}$ *according to the same policy.*

Proof. Let the pulses be numbered in the order in which the Roadmaker's algorithm applied to $\mathbf{x}$ extracts them.

Note first that by Property 4a, the signs of the arcs are not changed by the change of heights, and the structure of the first reduced DAG is therefore also unchanged. This implies that the first feature to be flattened has the same sign and support as before, in other words, the pulse to be subtracted is a multiple of $\mathbf{z}_1$ with the same sign as h_1, which we assume without loss of generality to be positive (the negative case follows by symmetry).

Let a be the node of the DAG representing $\mathbf{z}_1$, and consider a border arc (a, b). By Property 4a,

$$x_a - x_b = \sum_{j=1}^{p} h_j \nu(\mathbf{z}_j, a, b) = h_1 + \sum_{j=2}^{p} h_j \nu(\mathbf{z}_j, a, b)$$

where $x_a - x_b$ and all the nonzero terms in the sum are positive. Therefore

$$y_a - y_b = h_1' + \sum_{j=2}^{p} h_j' \nu(\mathbf{z}_j, a, b) \geqslant h_1',$$

since h_j' has the same sign as h_j.

Let m be a nearest neighbour of a with the original heights. Then $x_a - x_m = h_1$, which implies that $\nu(\mathbf{z}_j, a, m) = 0,\ j = 2, 3, \ldots, m$, since no cancellation is possible among terms of the same sign. Therefore $y_a - y_m = h_1'$, so that $\min_b y_a - y_b \leqslant h_1'$. Thus $\min_b y_a - y_b = h_1'$, which means that the pulse $h_1' \mathbf{z}_1$ consistently extracted and the same m is also a closest neighbour in the case of $\mathbf{y}$.

Since none of the boundary arcs of $\mathbf{z}_1$ is neutral, the following two processes are equivalent:

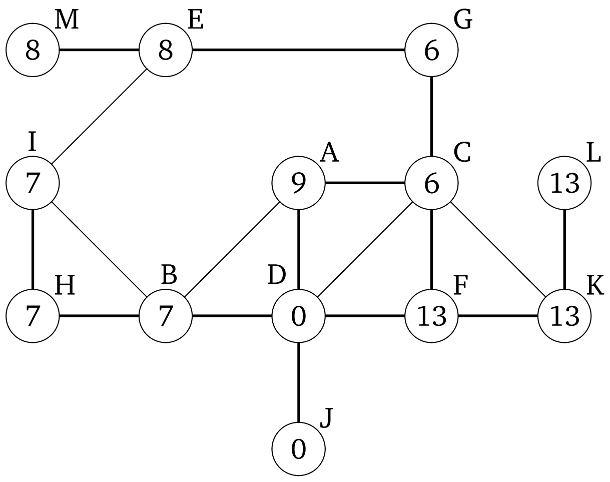
1. Contract all neutral arcs and then flatten the feature at $\mathbf{z}_1$.

2. Flatten the feature at $\mathbf{z}_1$ and then contract all neutral arcs.

Thus after the flattening of the feature at $\mathbf{z}_1$, proceeding with the Roadmaker's algorithm in Step 2 gives the same result as restarting it. When restarting, the pulse $h_2'\mathbf{z}_2$ is consistently extracted. The theorem follows by induction. ■

4 Implementation and efficiency

The high-level description of the Roadmaker's algorithm conceals a lot, although it does identify the most important components of the algorithm. Before going into more detail, it is helpful to look at a small example presented as a graph. The number in each circle is the value at that node.

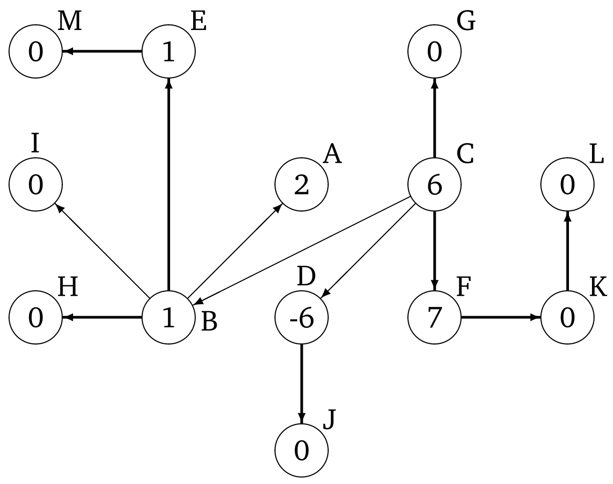


The DPT of this graph is represented by a tree rooted at C. In order to keep the position the nodes the same, the arcs are shown as directed from parent to child. (This is not the same direction as in the DAG of the previous section.) Note that not all of them correspond to arcs in the original graph. The number in each circle is now the pulse height (for the root node this is the same as the value function).

Each nonzero height corresponds to a pulse, and the nodes in the pulse are all the nodes in the subtree rooted at that node. Thus the six pulses in this DPT are:

$$A : +2,\ \mathrm{EM} : +1,\ \mathrm{DJ} : -6,\ \mathrm{FKL} : +7,\ \mathrm{ABEHIM} : +1,\ A \ldots M : 6.$$

The original value at each node equals the sum of the heights of the pulses to which it belongs. It can be recovered from the tree by adding up the heights of the nodes on the path connecting the node with the root.



4.1 Data structures and algorithms

Unless the graph is so small that we do not mind traversing the entire graph at each iteration, we will need several data structures that are initialized by Step 1 and updated by Step 2. In this section I make some suggestions in this regard, using a typewriter typeface for computer code.

These suggestions are not claimed to be the only or the best way of implementing the algorithm; they are, however, sharp enough to demonstrate that the Roadmaker's algorithm can be implemented to take $\mathcal{O}(n)$ time in the one-dimensional case.

Instead of deleting the nodes that are merged into others and creating new nodes for the cluster, it is more efficient to modify the nodes during the course of the algorithm. The nodes that are removed from the graph go into the DPT, and at the end of the algorithm, only one node remains in the graph.

The two pictures above suggest that the original value function and its DPT can be represented by the same data structure. since the array structure provides the option of visiting every node by iteration. Doing so simplifies some of the algorithms.

For every node remaining in the graph we keep the following information:

`neighbors` The set of neighbours in the current graph.

`size` Number of original nodes represented by this node.

`parent` A reference to the node itself.

`value` The value function at this node.

`tag` The index of this node in the array of nodes.

When a node is removed from the graph, two fields change:

`parent` A reference to the node into which this node is merged.

`value` The height of the pulse at this node. This may be zero: when a feature having more one nearest neighbour is contracted, one neighbour survives, and the others go into the DPT as pulses of height zero.

When the DPT has been completed, the semantics of `neighbors` changes too: it now contains the children of the node in the DPT tree. This task can be done in time $\mathcal{O}(n)$ by accessing the nodes of the graph as an array.

```
    for node in graph:
        node.neighbors.clear()
    for child in graph:
        if child is not root:
            node.parent.neighbors.add(child)
```

The above code, like the code below, should be thought of as pseudocode rather than a snippet of program code, although in fact it conforms to the syntax rules of the popular scripting language Python and has been adapted from an actual Python program [3]. The pseudocode is not quite executable Python code, mainly because Python does not allow a reference to a mutable object to be a member of a set. In the actual program, the set members are not the objects themselves but their tags, and access to the array is explicit, e.g. `graph.nodes[i]`. This has been suppressed in the pseudocode.

For the applications, one would replace the heights of unwanted pulses by zero and recalculate the node values. The latter task can be accomplished in time $\mathcal{O}(n)$ as follows.

```
    for child in traverse(graph):
        child.value += child.parent.value
```

Here, `traverse` is an iterator that yields the nodes of the tree except the root node in breadth-first order starting at the root. Such a procedure is feasible because `neighbors` has been changed to describe the tree structure.

Here is the pseudocode for the DPT.

```
def DPT(graph):
    for node in graph:
        if node.parent is node:
            shrink(node)
    while True:
        feature = schedule.pop()
        if feature is None: break
        survivor = nearest(feature)
        height = feature.value - survivor.value
        feature.value = survivor.value
        shrink(survivor)
        feature.value = height
```

nearest(feature) finds a nearest neighbour of feature by examining all the neighbours; we omit the details. Note that feature itself, which represents the feature being flattened, is removed from the graph: its nearest neighbour is the survivor.

The procedure shrink, which shrinks a constant-valued cluster to a single vertex, is the heart of the algorithm in both steps.

```
def shrink(node):
    agenda = queue([node])
    sign = [0,0]
    while agenda is not empty:
        item = agenda.get()
        for child in list(item.neighbors):
            child.neighbors.remove(item)
            if child.parent is not node:
                if node.value == child.value:
                    agenda.put(child)
                    child.parent = node
                    child.value = 0
                    node.neighbors.discard(child)
                    node.size += child.size
                else:
                    insert_arc(node,child)
                    sign[node.value>child.value] = 1
    feature = sign[1]-sign[0]
    if node.neighbors is empty:
        root = node
    elif feature:
        schedule.push(node,node.size,feature)
```

The algorithm is quite straightforward. A first-in first-out queue is used to hold nodes in the cluster. The arc (child,item), where item is the cluster, is deleted and if child is not also in the cluster, (node,child) is inserted in its place. If child is in the cluster, its value is set to 0 (if it is actually the feature, the correct height is later set in the main DPT loop) and it is marked inactive. This is how in Step 1 we know whether a node has already been absorbed into a cluster.

The expression sign[1]-sign[0] evaluates to -1, 0 or 1 according to whether the cluster is a pit, a non-feature or a bump. If this expression is nonzero, the cluster is pushed onto the schedule: more about that later. The sixth line has `list(item.neighbors)` rather than just `item.neighbors` in order to freeze the list: when `item` is `node`, `item.neighbors` changes inside the loop, which would invalidate the iteration.

In addition, `shrink` records which node is the root of the graph. This node is not regarded as a feature.

I have maybe gone into more detail in the code for `shrink` than one would like for mere pseudocode, but it is needed for the complexity analysis.

4.2 Complexity

If we remove the decision-making mechanism from the Roadmaker's algorithm, it can be seen as a special case of the following process.

> Given a general graph, pick an arc arbitrarily and contract it. Continue doing so until no arcs are left.

Contracting an arc is done by deleting all the arcs emanating from the two endpoints and creating arcs from the merged node to all border nodes. In the process, all the original arcs will be examined at least once. Thus, the total complexity of the Roadmaker's algorithm is at least $\mathcal{O}(m)$, where m is the number of arcs in the graph.

It is clear that any fortuitous neutral arcs arising from equal node values might make the algorithm slightly faster (for example by taking advantage of border nodes belonging to more than one cluster member) but never slower. In fact, when all node values are distinct, no cluster ever has more than two nodes, and the Roadmaker's really is the above process with "arbitarily" replaced by "according to the policy".

It is to be expected, therefore, that counting the number of arc deletions will be equivalent to determining the complexity. Since every arc in the graph is eventually deleted, one can equivalently count how many arcs are created, including the arcs originally present. We must also make sure that the decision-making mechanism is not of higher complexity than the graph operations.

The first statement of the innermost loop in the code for `shrink`, which deletes the arc `(child,item)`, will be called the 'counter'. The counter is reached at least once for every call to shrink, and the statements outside the `while` loop, including `schedule.push`, at most once. In the next subsection I demonstrate that schedule operations take amortized time $\mathcal{O}(1)$, which here means an amount of time that, when accumulated over the whole calculation, does not exceed $\mathcal{O}(n)$. Thus the complexity of `shrink` is dominated by how many times the counter is reached.

In the higher-level code for Steps 1 and 2, both loops contain a call to `shrink`. The other statements take time $\mathcal{O}(1)$, except `nearest`, which takes time proportional to the number of neighbours that the feature has. Every feature examined by `nearest` was examined by `shrink` just before it was pushed onto the schedule, and therefore the total complexity of all calls to `nearest` cannot be higher than that of all calls to `shrink`.

It follows that it suffices, from the point of view of complexity, to count how often the counter is reached, i.e. how many arcs are deleted.

Say that there are m_j neighbours of node j, m_j' border nodes in the cluster arising when the node j is shrunk in Step 1 (even the cluster contains only one node and `shrink` has done nothing more than re-insert the arcs it has deleted), and m_j'' border nodes when the cluster represented by node j is shrunk in Step 2. If a node j becomes inactive before it is reached in Step 1, $m_j' = m_j'' = 0$; and for the root node r eventually surviving, $m_r'' = 0$. The total number of arcs ever created is

$$M = m + \sum_j m_j' + \sum_j m_j''.$$

In Step 1, the number of border arcs of a cluster cannot be more than the total number of arcs connected to cluster members, and therefore

$$\sum_j m_j' \leqslant \sum_j m_j = 2m.$$

This bound is reached when there are no neutral arcs.

It is in general difficult to estimate $\sum m_j''$ in advance, since the same node may represent many clusters at different stages. There are nevertheless some cases where this is possible.

The simplest case to analyze is that of an n-cycle when all node values are distinct. Then $m_j'' = 2$ until only three nodes are left, after which only one new arc is created. Since $m = n$,

$$M_{\text{cycle}} = 3m + 2(n-3) + 1 = 5n - 5 = 5(n-1).$$

The one-dimensional case is obtained from an n-cycle by deleting one arc but keeping all the nodes. Therefore

$$M_{\text{sequence}} < 5(n-1).$$

It follows that the Roadmaker's algorithm has complexity $\mathcal{O}(n)$, or equivalently $\mathcal{O}(m)$, in the one-dimensional case.

Another case that can be analyzed is that of a complete graph. In that case, m_j'' assumes the values $n-2, n-1, \ldots, 1, 0$ and

$$\sum_j m_j = (n-1)(n-2)/2 = n(n-1)/2 - n + 1 \leqslant m,$$

and we obtain

$$M_{\text{complete}} \leqslant 4m,$$

with equality only in the trivial case $n = 1,\ m = 0$. Thus the Roadmaker's algorithm has complexity $\mathcal{O}(m)$ in this case.

Since these two cases represent extremes (as few and as many arcs as possible for a given number of nodes), it is tempting to conjecture that the Roadmaker's algorithm always has complexity $\mathcal{O}(m)$, but this is almost certainly not true: see Section 5.

4.3 The schedule

The distinguishing property of the DPT is the order in which features are flattened. This requires a structure rather like a priority queue, onto which features are pushed in any order but from which they are popped in order of urgency.

A priority queue of N items takes $\mathcal{O}(N)$ time to initialize and $\mathcal{O}(\log N)$ per item time to update. This timing are achieved when the queue is organized as a heap. However, it would be the sole reason in the one-dimensional case for achieving only complexity $\mathcal{O}(n \log n)$.

There are two reasons why we can achieve a better update time than in the general situation of a priority queue:

1. The sizes are not general floating-point numbers but integers smaller than n.

2. Any new feature to be pushed is always larger than the feature most recently popped.

To make sure of $\mathcal{O}(1)$ time per item for updating, we can keep the schedule as an array of n deques (double-ended queues). When pushing an item onto the schedule, size and sign are supplied: size determines which deque is being accessed; sign, whether the item goes to the front or rear of the deque. When popping, the policy in effect at the current size is consulted to determine whether the item comes off the front or rear.

Pushing and popping an item requires $\mathcal{O}(1)$ time. However, finding the correct deque may take longer, because the update loop may have to pass through many empty deques before finding a nonempty one. We call this process the *countdown*. Since a new item is always larger than the one most recently popped, a deque is never revisited, and the countdown time is $\mathcal{O}(n)$, giving amortized update time per node of $\mathcal{O}(1)$.

Actually, it is better in terms of storage to restrict the array of deques only to those features smaller than some cutoff size w, and to use a more compact albeit more complicated data structure for the rest. The reason is that the less urgent items are rare: there cannot be more than n/w features of size $\geqslant w$ in the queue at any given moment. It therefore makes sense for the larger items to use a data structure which does not reserve storage for unused items.

A rough rule of-thumb argument shows that $w = \lfloor\sqrt{n}\rfloor$ is a good practical choice. There can be at most n/w entries in the other structure, and there are w deques in the array. The expression $w + n/w$ has a minimum when $w = \sqrt{n}$.

The other structure may be a heap of features, since $n/w \ \log(n/w)$ is much smaller than $\mathcal{O}(n)$ for the recommended value of w. The countdown is also virtually eliminated. The main disadvantage of using a heap is that a comparison routine for entries in the heap is required when updating it.

It is much easier, especially in Python with its built-in dictionary type, to use a hash table of deques, which looks exactly like an array but does not waste storage. The task of accessing the appropriate deque can be hidden at a lower level than the push and pop routines.

One subtle point remains. Not every node pushed onto the schedule should be popped off it. Suppose that the cluster being shrunk contains besides its feature another node that is also on the schedule. The feature at that node is destroyed when the cluster is shrunk. It would be very inconvenient to have to find where it is saved in the schedule, in order to delete it explicitly. An easier way is to interrogate `active(node,size)` before allowing `node` to be popped, where `size` is the size appropriate to the deque in which `node` is stored.

```
def active(node,size):
    return (node.parent is node) and (node.size == size)
```

This test works because the first clause eliminates all of the cluster except `survivor`, and the second clause eliminates `survivor` since during the execution of `shrink(survivor)`, `survivor.size` is increased by at least `feature.size`. If the test fails, the feature is not returned but simply thrown away.

The actual code of the routines implementing the schedule follow the above description so closely that I omit giving pseudocode for them.

5 Examples and remarks

All computer work was done using an implementation in Python 2.5 obtainable from my website [3], running under Ubuntu 8.04 on a Dell Latitude D830 notebook computer, which has a dual-core Intel CPU with 2Gb memory running at 2GHz. The word ‘random’ below actually means ‘pseudorandom using Python’s `randint`].

5.1 Demonstration of algorithmic complexity

Our first example involves one-dimensional graph functions with random integer values in the range 0 to $2^{16} - 1$. The following table reports the number of times that the counter was reached, as well as actual timings in seconds. At the largest value of n, the Python run used 32.1% of the available memory, according to the utility `top`. Thus, $k = 7$ would require swapping on this machine.

n	M	time
11	43	0.001
101	486	0.011
1001	4978	0.047
10001	49961	0.386
100001	499915	3.869
1000001	4999733	39.474

The bound of $5(n - 1)$ on the count can be seen to be quite tight, and the expected $\mathcal{O}(n)$ running time leaps to the eye as soon as the time is long enough to be measured accurately.

Next, complete graphs with random values from the same distribution.

n	m	M	time
8	28	105	0.001
16	120	465	0.005
32	496	1953	0.022
64	2016	8001	0.027
128	8128	32385	0.103
256	32640	129094	0.401
512	130816	521234	1.639
1024	523776	2047188	6.487
2048	2096128	8027651	26.019

It is interesting that up to $n = 128$, the relation $M \leqslant 4m - n + 1$ holds with equality, but not after that, because neutral arcs start appearing in the original graph. If the test is repeated with distinct values, $M = 4m - n + 1$ for all n.

Another interesting phenomenon is that the time per node deletion is much less (about 3 microseconds instead of 8) than in the case of a sequence. The reason is that $n \ll m$ for the complete graph but $n \approx m$ for the sequence. Those parts of the computation that have complexity $\mathcal{O}(n)$ rather than $\mathcal{O}(m)$ are therefore important for a sequence but insignificant for the complete graph.

Finally, random connected graphs more dense than a sequence but less dense than a complete graph were generated for as follows: a random tree containing all n nodes was generated first, and approximately $n^{3/2}/10$ further arcs were created randomly.

n	m	M	time
8	9	40	0.001
16	19	94	0.002
32	44	237	0.004
64	114	817	0.011
128	264	2602	0.019
256	650	12813	0.053
512	1661	57215	0.219
1024	4279	243813	0.902
2048	11281	1207829	4.420
4096	30248	5515890	20.509

For these graphs, the time per node deletion for the larger values of n is a little less than 4 microseconds, which agrees with their intermediate density. What is more interesting is that one can see at a glance that M/m grows steadily: for the last two values, $M \approx m^{3/2}$. This experimental evidence effectively refutes the supposition that $M = \mathcal{O}(m)$.

5.2 Other stopping rules

For most applications, it is enough to stop when all pulses up to a certain size have been found. There is no significant computational advantage in not completing the pulse decomposition since the bulk of the work in the Roadmaker's algorithm is typically done while small pulses are removed. This trivial stopping rule is therfore not very interesting.

In the one-dimensional case, a more useful rule is to stop when the sequence has been reduced to a monotonic trend, because the trend may have a revealing interpretation in the context of the underlying application. The graph-theoretic generalization of a monotonic sequence is a partial ordering: along any path of the graph, the value function is monotonic. This is achieved not by changing the stopping rule, but by changing the definition of a feature:

> A feature is a node with only incoming or only outgoing arcs, and at least two neighbours.

The high-level description of the Roadmaker's algorithm "repeat until there are no more features" remains the same.

5.3 Complexity and optimality

It seems reasonable to suppose that a lower bound for the complexity of calculating the DPT is given by $\mathcal{O}(\max(m, M_{\text{DPT}}))$, where M_{DPT} is the total number of non-neutral arcs in the pulses of the DPT. This would certainly be true if by "calculating" we mean that each pulse should be specified by listing its members and border, since that list contains $\mathcal{O}(M_{\mathrm{DPT}})$ entries, but may be pessimistic if the tree of the DPT is accepted as an adequate description.

To investigate whether this is so, and whether the Roadmaker's algorithm is indeed optimal in the sense of having the same complexity, is at this stage a challenging open problem. Experimental evidence suggests that the answer to both questions is yes.

References

[1] R Anguelov and C H Rohwer. LULU operators for functions of continuous argument. *Quaestiones Mathematicae*, 2009. To appear.

[2] Roumen Anguelov and Inger Fabris-Rotelli. LULU operators and discrete pulse transform for multi-dimensional arrays. *IEEE Trans. Signal Processing*, Submitted.

[3] D. P. Laurie. Python programs for the digital pulse transform. URL=`http://dip.sun.ac.za/~laurie/DPT`.

[4] D. P. Laurie and C. H. Rohwer. Fast implementation of the discrete pulse transform. In T.E. Simos, G. Psihoyios, and Ch. Tsitouras, editors, *IC-NAAM 2006: International Conference on Numerical Analysis and Applied Mathematics 2006*, pages 484–487, Weinheim, 2006. Wiley-VCH.

[5] C. H. Rohwer. *Nonlinear Smoothing and Multiresolution Analysis*. Birkhäuser, Basel, 2005.

[6] C. H. Rohwer and D. P. Laurie. The discrete pulse transform. *SIAM J. Math. Anal.*, 38(3):1012–1034, 2006.

[7] C. H. Rohwer and M. Wild. LULU theory, idempotent stack filters, and the mathematics of vision of Marr. *Advances in imaging and electron physics*, 146:57–162, 2007.