# Image Shrinking And Expanding On A Pyramid*

Jing-Fu Jenq and Sartaj Sahni
University of Minnesota University of Florida

**Abstract**
We develop two algorithms to perform the $q$ step shrinking and expanding of an $N \times N$ binary image on a pyramid computer with an $N \times N$ base. The time complexity of both algorithms is $O(\sqrt{q})$. However, one uses $O(\sqrt{q})$ space per processor while the per processor space requirement of the other is $O(1)$.

**Keywords and Phrases**
Binary image, shrinking, expanding, pyramid computer.

* This research was supported in part by the National Science Fundation under grants DCR-84-20935 and MIP 86-17374.

## 1 Introduction

Let $I[0..N-1,0..N-1]$ be an $N \times N$ image. The *neighborhood* of the pixel $[i,j]$ is the set of pixels

$$nbd(i,j) = \{[u,v] \mid 0 \le u,v < N,\ \max\{|u-i|,\ |v-j|\} \le 1\}$$

The q-step shrinking of $I$ is defined in [ROSE82] and [ROSE87] to be the $N \times N$ image $S^q$ such that

$$S^1(i,j) = \min_{(u,v) \in nbd(i,j)} \{I(u,v)\},\ 0 \le i < N,\ 0 \le j < N$$

$$S^q(i,j) = \min_{(u,v) \in nbd(i,j)} \{S^{q-1}(u,v)\},\ q > 1,\ 0 \le i < N,\ 0 \le j < N$$

The q-step expansion of I is similarly defined to be the $N \times N$ image $E^q$ where

$$E^1(i,j) = \max_{(u,v) \in nbd(i,j)} \{I(u,v)\},\ 0 \le i < N,\ 0 \le j < N$$

$$E^q(i,j) = \max_{(u,v) \in nbd(i,j)} \{E^{q-1}(u,v)\},\ q > 1,\ 0 \le i < N,\ 0 \le j < N$$

Let $B_{2q+1}[i,j]$ represent the block of pixels:

$$\{[u,v] \mid 0 \le u,v < N,\ \max\{|u-i|,\ |v-j|\} \le q\}\}$$

So, $nbd(i,j) = B_3[i,j]$. Rosenfeld [ROSE87] has shown that

$$E^q[i,j] = \max_{[u,v] \in B_{2q+1}[i,j]} \{I[u,v]\},\ 0 \le i,j < N \qquad (1)$$

and

$$S^q[i,j] = \min_{[u,v] \in B_{2q+1}[i,j]} \{I[u,v]\},\ 0 \le i,j < N \qquad (2)$$

For binary images, the max and min operator used in (1) and (2) may be replaced by the logical **or** and **and** operators, respectively. Since the computational aspects of computing $E^q$ and $S^q$ are essentially the same, we shall henceforth deal only with $E^q$. Rosenfeld [ROSE87] introduced the notion of a *coarsely resampled* computation of $E^q$. In this, only a subset of the pixels in the block $B_{2q+1}[i,j]$ are examined when computing $E^q[i,j]$. This subset consists of all pixels of $B_{2q+1}[i,j]$ that are $ck$ ($k = \log_2 q$) distant from $[i,j]$ for $c$ an integer. He showed that the coarsely resampled matrix $E^q$ can be computed in $O(k)$ time on a pyramid with base $N \times N$. His algorithm to compute $E^k$

exactly takes $O(k^2)$ time for one dimensional binary images and uses the tree structure of the pyramid. The generalization of his algorithm to two dimensional binary images on a pyramid takes $O(q)$ time.

Ranka and Sahni [RANK89] develop $O(k)$ time algorithms to compute $E^q$ (exactly) on an $N^2$ PE SIMD hypercube. Their algorithms work for both binary and gray scale images. $E^q$ is easily computed in $O(q)$ time on an $N\times N$ mesh computer. On an $N\times N$ RMESH (reconfigurable mesh with buses), $E^q$ for binary images can be computed in $O(1)$ time [JENQ90].

In this paper, we develop two algorithms to compute $E^q$ for binary images. The algorithms are for a pyramid computer with an $N\times N$ base. Both algorithms have time complexity $O(\sqrt{q})$. One has space complexity $O(\sqrt{q})$ per processor while the per processor space complexity of the other is $O(1)$. These algorithms represent an improvement over the pyramid algorithms of [ROSE87] and demonstrate that $E^q$ can be computed faster on a pyramid than on a mesh.

Because of the similarities in the equations for $E^q$ and $S^q$ (i.e., (1) and (2)), our algorithms as well as those of [ROSE87], [RANK89], and [JENQ90] are easily modified to compute $S^q$ in the same time bounds.

## 2 Preliminaries

A *pyramid computer* (Figure 1) consists of several layers of processors. Each layer is arranged as a square mesh of dimension half that of the layer below it. The lowest layer is called the *base* and the highest layer is called the *apex*. The *apex* has exactly one processor (i.e., it is a 1×1 mesh). The base layer is called layer 0, the one above it is layer 1, and so on. If the base is an $N\times N$ mesh, then layer 1 is an $N/2\times N/2$ mesh; layer 3 is an $N/4\times N/4$ mesh; and so on. So, the total number of layers is $\log_2 N + 1$ (we assume that $N$ is a power of 2). The total number of processors (PEs) in a pyramid with an $N\times N$ base is $(4^{k+1} - 1)/3$, where $k = \log_2 N$. Let $(i,j,l)$ be the index of a PE in the pyramid. $l$ is the level number and $(i,j)$ gives the PE position in the level $l$ mesh. The top left PE in each mesh is $(0,0)$. In addition to being connected to its at most four mesh neighbors in level $l$, PE$(i,j,l)$ is connected to PE ($i$ *div* 2, $j$ *div* 2, $l+1$) in the layer (if any) above it and the PEs $(2i, 2j, l-1)$, $(2i+1,2j,l-1)$, $(2i, 2j+1,l-1)$ and $(2i+1,2j+1,l-1)$ in the layer (if any) below it. Figure 1 shows a pyramid with 4×4 base.

The PE in layer $l+1$ to which PE $(i,j,l)$ is connected is called the *parent* of PE $(i,j,l)$. Note that the apex PE has no parent. The PEs $(2i, 2j,l-1)$, $(2i, 2j+1,l-1)$, $(2i+1,2j,l-1)$, and $(2i+1,2j+1,l-1)$ to which PE $(i,j,l)$ is connected are, respectively, the NW, NE, SW, and SE children of $(i,j,l)$. Note that the base PEs have no children.

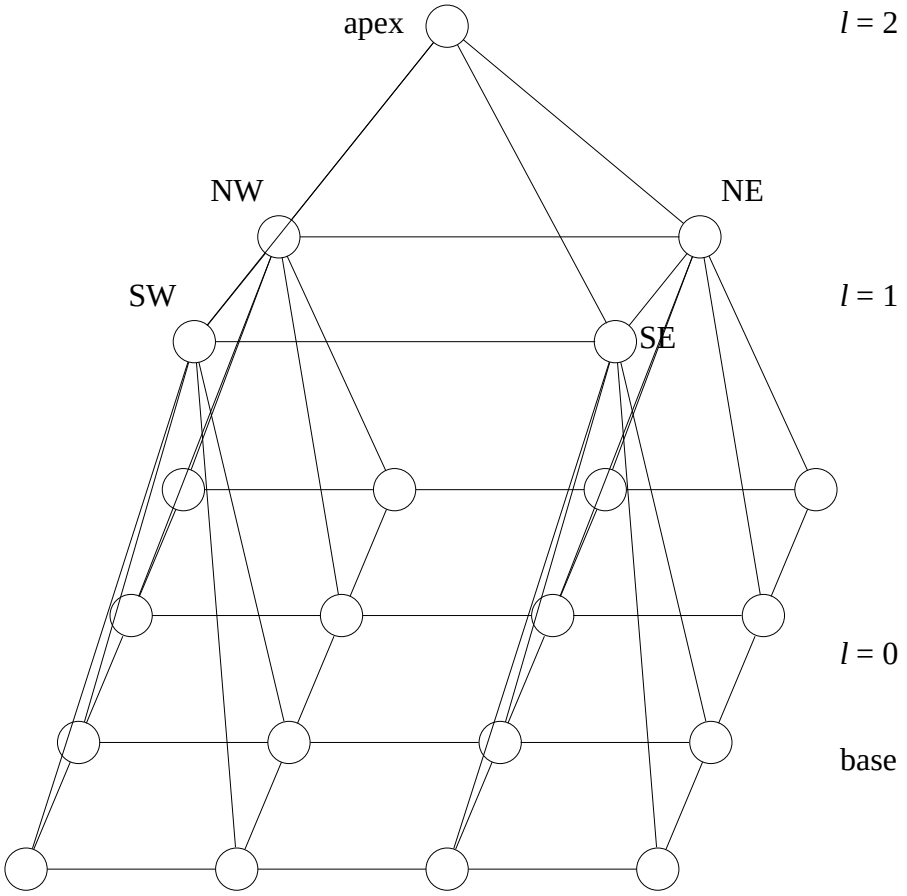



**Figure 1** Pyramid with 4×4 base

We shall use the notation $A(i,j,l)$ to denote the variable $A$ in processor $(i,j,l)$. The notation $B[r](i,j,l)$ denotes the $r$'th element of array $B$ of processor $(i,j,l)$. By the term $N \times N$ pyramid, we mean a pyramid with an $N \times N$ base.

When using (1) to compute $E^q$, it is helpful to decompose the computation into steps as below [RANK90].

$$E^q[i,j] = \max_{[u,j] \in B_{2q+1}[i,j]} \{R^q[u,j]\},\ 0 \le i,j < N$$

where

$$R^q[u,j] = \max_{[u,v] \in B_{2q+1}[i,j]} \{I[u,v]\},\ 0 \le u,j < N$$

$$= \max \{ I[u,v] \mid |j-v| \leq q \},\ 0 \leq u,j < N$$

The computation of $R^q$ may be decomposed into subcomputations as below:

$$left^q[u,j] = \max\{I[u,v] \mid 0 \leq j-v \leq q\}$$

$$right^q[u,j] = \max\{I[u,v] \mid 0 \leq v-j \leq q\}$$

$$R^q[u,j] = \max\{left^q[u,j],\ right^q[u,j]\}$$

Similarly we may decompose the computation of $E^q$ from $R^q$ as below:

$$top^q[i,j] = \max\{R^q[u,j] \mid 0 \leq i-u \leq q\}$$

$$bottom^q[i,j] = \max\{R^q[u,j] \mid 0 \leq u-i \leq q\}$$

$$E^q[i,j] = \max\{top^q[i,j],\ bottom^q[i,j]\}$$

Because of the symmetric nature of the inter-processor connections in a pyramid, the computation of $E^q$ from $R^q$ is very similar to that of $R^q$ from $I$. So, we need to consider only the computation of $E^q$ from $R^q$. In this computation of $E^q$, the computation of $top^q$ and $bottom^q$ are very similar. As a result we need only develop the algorithms to compute $top^q$ from $R^q$. From the definition of $top^q$, it follows that $top^q[i,j]$ is 1 iff $R^q[u,j]$ is a 1 in some row $u$ that is a distance at most $q$ from row $i$ and is above row $i$.

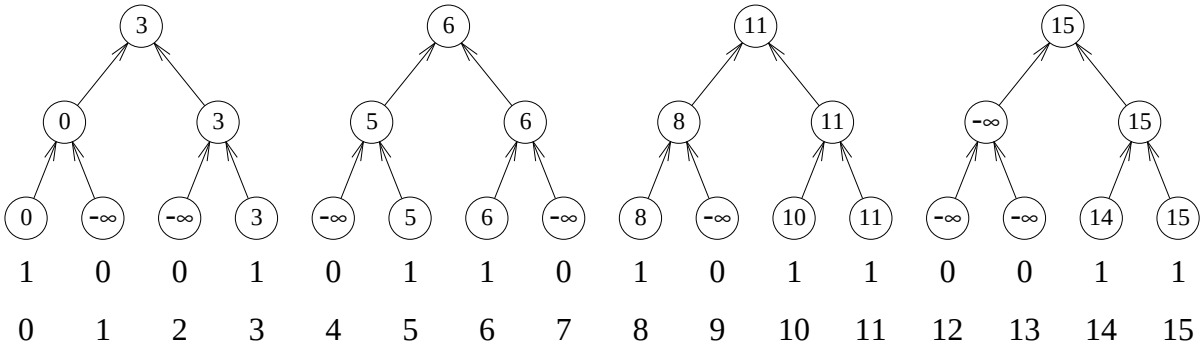
Our algorithms begin with the initial configuaration $R(i,j,0) = R^q[i,j],\ 0 \leq i,j < N$. As a result, $top(i,j,0) = top^q[i,j] = 1$ iff there is a 1 in the $R$ variable of a base processor in column $j$ that is above PE $(i,j,0)$ by at most $q$ rows. Our algorithms seek to determine this for every base PE. Note that since our algorithms are only for binary images, $left^q$, $right^q$, $top^q$, $bottom^q$, $R^q$ and $E^q$ are all binary matrices.

## 3 $O(\sqrt{q})$ Memory Algorithm For $top^q$

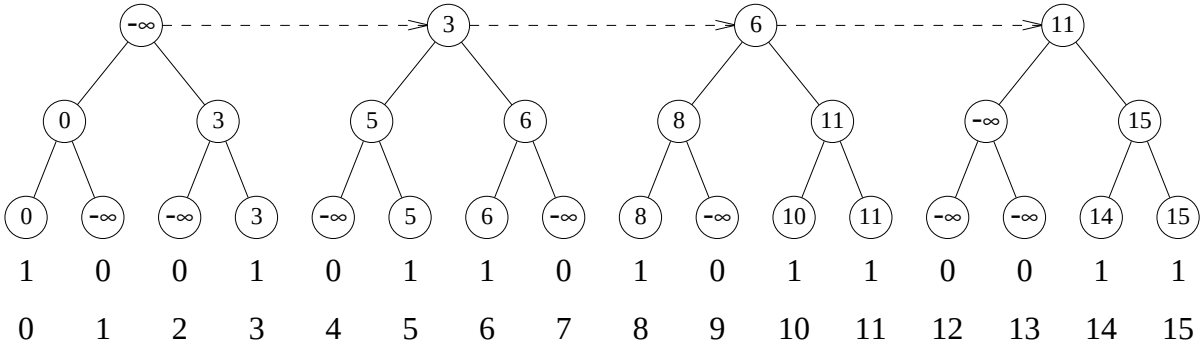
For simplicity, we first assume that $q$ is a perfect square and that $\sqrt{q}$ is a power of two. Later we describe how to modify our algorithm to the case when $q$ does not satisfy these assumptions. Since $\sqrt{q}$ is a power of 2 (by assumption) and $N$ is a power of 2, $\sqrt{q}$ divides $N$. Define $near[i,j]$ to be the largest $u$, $\max\{0, i-q\} \leq u \leq i$, such that $R[u,j]=1$. In case there is no such $u$, let $near[i,j] = -\infty$. From the equation for $top^q$, it follows that $top^q[i,j] = 1$ iff $near[i,j] \neq -\infty$. So, we can concentrate on the development of an algorithm to compute $near[i,j]$.

It is instructive to first consider how $near[i,j]$ may be computed for a single column $j$ using a computer consisting of several binary trees, each having $\sqrt{q}$ leaves. Figure 2(a) shows the case when $\sqrt{q}=4$ and we have four trees each with $\sqrt{q}=4$ leaves. We assume

that, in this computer, nodes at the same level are connected to form a chain. Figure 2(b) shows the chain connections (broken lines) for the root nodes of the four trees. The number below each of the leaf nodes of Figure 2(a) is its $R$ value. The leaf nodes themselves are numbered 0 through 15. Leaf node $i$ represents $R[i,j]$. At the end of the computation, leaf node $i$ is to contain $near[i,j]$.



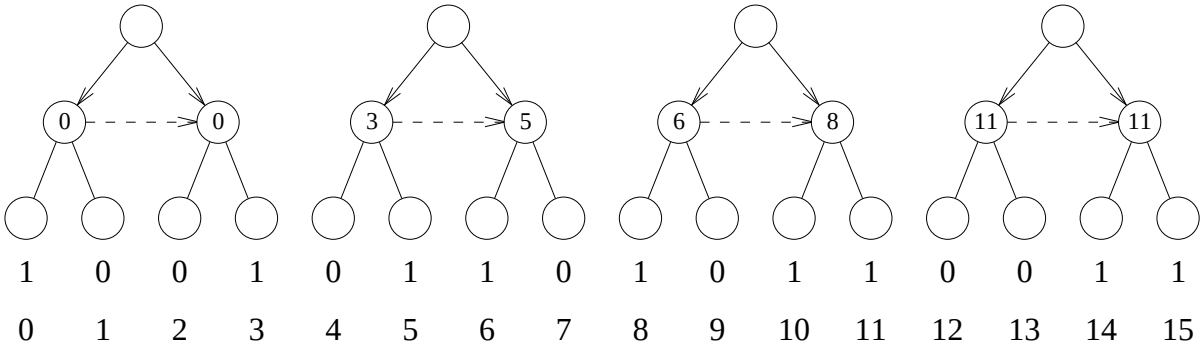

(a) Upward phase



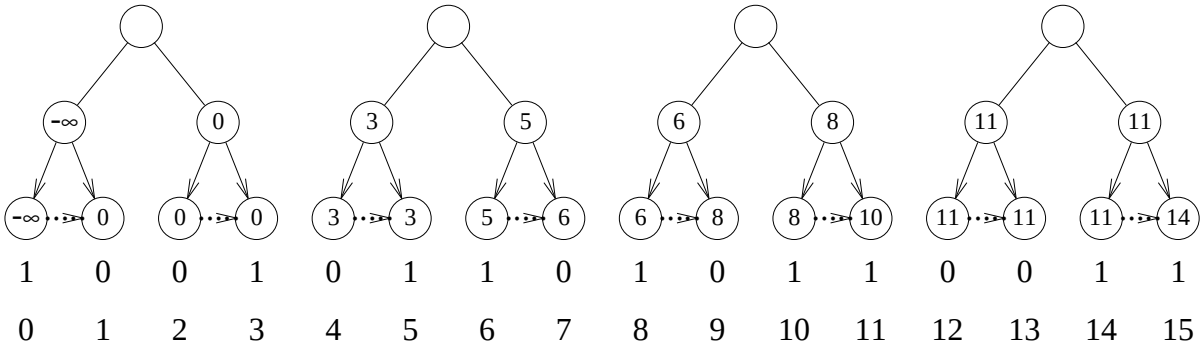

(b) Downward phase, stage 1

**Figure 2** Computation of *near* (continued)

The computation is done in two phases. In the first phase, called the upward phase, each node in each of the trees computes the position of the rightmost 1 (i.e., rightmost $R$ that is one) in the leaves that are in the subtree of which it is root. Let $Rightmost(i,l)$ denote this value for node $i$ on level $l$. In case none of the leaves in the tree with root $(i,l)$ have $R$ value 1, then $Rightmost(i,l)$ is defined to be $-\infty$. We assume that within levels, nodes are numbered from zero left to right. So, the parents of the leaves of Figure




(c) Downward phase, stage 2, iteration $l = 1$




(d) Downward phase, stage 2, iteration $l = 0$

**Figure 2** Computation of *near*

2(a) are numbered 0 through 4. Note that the parent of $(i,l)$ is $(i/2,l+1)$ (to be consistent with level numbers in a pyramid, we assign leaves the level zero).

The value of $Rightmost(i,l)$ is easily computed in $O(\log q)$ time by making an upward pass over the trees as described in Figure 3. The value of *Rightmost* for each node is shown inside the node in Figure 2(a).

The second phase is called the downward phase. In this, each root node first determines the location of the nearest one on its left. This is done by examining the *Rightmost* values of the roots of the trees on its left. Since, eventually, we are interested only in the location of a 1 that is at most $q$ nodes away, each root need only look at the $\sqrt{q}$ roots to the left of it. This is done using the chain connection of the roots as in Figure 4. The result for our example is shown in the roots of Figure 2(b).

---

**Procedure** Rightmost;
{compute $Rightmost(i,l)$, $0 \le i < N$, $0 \le l \le \log_2 \sqrt{q}$ }
$Rightmost(i, 0) := -\infty$, $0 \le i < N$;
**if** $R(i, 0) = 1$ **then** $Rightmost(i, 0) := i$;
**for** $l := 1$ **to** $\log_2 \sqrt{q}$ **do**
$Rightmost(i,l) := \max \{Rightmost(2i,l-1), Rightmost(2i+1,l-1)\}$, $0 \le i < N/2^l$;
**end**;

---

**Figure 3** Computation of $Rightmost(i,l)$, *upwardphase*

---

**Procedure** UpdateRoots
{update *Rightmost* values of roots by looking at}
{roots on left and at most $\sqrt{q}$ distant}
$l := \log_2 \sqrt{q}$; {root level}
$t(i,l) := Rightmost(i,l)$, $0 \le i < N/\sqrt{q}$;
$Rightmost(i,l) := -\infty$, $0 \le i < N/\sqrt{q}$;
**for** $k := 1$ **to** $\sqrt{q}$ **do**
**begin**
$t(i,l) := t(i-1,l)$, $1 \le i < N/\sqrt{q}$;
$Rightmost(i,l) := \max\{Rightmost(i,l), t(i,l)\}$, $1 \le i < N/\sqrt{q}$;
**end**
**end**

---

**Figure 4** Stage 1 of downward phase

Once the value of *Rightmost* has been computed as in Figure 4, stage 1 of the downward phase is complete. Note that now, each tree with $\sqrt{q}$ leaves has all the information it needs to compute the *near* value for its leaves as for any leaf node $(i, 0)$, the *near* value is either the location of the rightmost one in a tree to the left of its own tree or is the location of the rightmost one on its left but in its own tree. The former location is available in the root of the tree. In the second stage of the downward phase, the *Rightmost* values of nodes in each tree are updated so as to yield the location of the rightmost one no more

than $\sqrt{q}$ trees to the left but including leaves in the same tree and to the left of the subtree of which this node is root. This is done using computations that are local to each tree with $\sqrt{q}$ leaves. The steps are given in Figure 5.

---

**Procedure** UpdateRest;
  {update *Rightmost* for all non root nodes}
  $t(i,l) := Rightmost(i,l)$, $0 \le i < N/2^l$, $0 \le l \le \log_2 \sqrt{q}$;
  **for** $l := \log_2 \sqrt{q} - 1$ **downto** 0 **do**
  **begin**
    $Rightmost(i,l) := Rightmost(i/2, l+1)$, $0 \le i < N/2^l$;
    $Rightmost(i,l) := \max\{Rightmost(i,l), t(i-1,l)\}$, $0 \le i < N/2^l$ and $i$ odd;
  **end**
**end**

---

**Figure 5** Update *Rightmost* for non root levels

Figure 2(c) gives the updated *Rightmost* values for the level 1 nodes and Figure 2(d) gives them for the level 0 nodes. It is now easy to obtain $near(i, 0)$ from $Rightmost(i, 0)$. Actually one may directly compute $top^q[i,j] = top^q(i, 0)$ as below:

$top(i, 0) := R(i, 0)$;
**if** $i - Rightmost(i, 0) \le q$ **then** $top(i, 0) = 1$;

Because of the tree connections, the upward phase has complexity $O(\log q)$. Stage 1 of the downward phase uses the chain connections and has complexity $O(\sqrt{q})$. Stage 2 uses both the chain and tree connection and its complexity is $O(\log q)$. The overall complexity is $O(\sqrt{q})$.

The scheme just described is easily adapted to a pyramid. Let $P(j, 0)$ be the set of processors in column $j$ of the pyramid base. Let $P(j,i)$ denotes the set of processors on level $i$ of the pyramid that are connected to the processors in $P(j,i-1)$, $1 \le i \le \log_2 \sqrt{q}$. The processors $S(j) = \bigcup_{i=0}^{\log \sqrt{q}} P(j,i)$ are connected as in the architecture of Figure 2. So, the algorithms of Figures 3-5 can be used to compute $top^q[i,j]$ for any one $j$ value in $O(\sqrt{q})$ time. They cannot be used directly to simultaneously compute $top^q[i,j]$ for all $j$ as the processors in $S(j)$ and $S(r)$ are disjoint only if $\lfloor j/\sqrt{q} \rfloor \ne \lfloor r/\sqrt{q} \rfloor$.

To get around this difficulty, we partition the base columns into $N/\sqrt{q}$ partitions

such that two columns $j$ and $r$ are in the same column partition iff $\lfloor j/\sqrt{q} \rfloor = \lfloor r/\sqrt{q} \rfloor$. Each column partition, $I$, computes its $top^q$ values independent of the other partitions using the processors in $IP(I)=\bigcup_{j\in I} S(j)$. Note that the processor sets $IP$ are disjoint.

Consider any partition of $\sqrt{q}$ columns. Procedure Rightmost (Figure 3) can run in a pipelined manner for all $\sqrt{q}$ columns. First column 0 begins, at the base processors (iteration $l=1$). Once this is complete, column 0 moves on to the $l=2$ iteration and column 1 begins its $l=1$ iteration. Following this, column 0 begins its $l=3$ iteration, column 1 its $l=2$ iteration, and column 2 its $l=1$ iteration. The variable *Rightmost* in levels one through $\log_2\sqrt{q}$ is expanded to an array $Rightmost[0..\sqrt{q}-1]$ with $Rightmost[j]$ being the value computed for base column $j$. Using this pipelining scheme, the *Rightmost* values for all $\sqrt{q}$ base columns and all $\log_2\sqrt{q}$ tree levels can be computed in $O(\sqrt{q})$ time (the computations for column 0 are complete in $\log\sqrt{q}$ steps, those for column 1 are complete one step later, and so on).

Stage 2 of the downward phase (Figure 5) may similarly be pipelined to work in $O(\sqrt{q})$ time. The pipelining of Stage 1 poses a problem. To successfully pipeline this with complexity $O(\sqrt{q})$, it is necessary to decompose this stage into two steps. For this, the roots are partitioned into sets of $\sqrt{q}$ adjacent roots and the algorithm of Figure 6 is used. Note that all communication other than the single inter partition transfer of step 2 is local to a partition. Following step 1, each root in a partition has the location of the rightmost one in a tree to its left but confined to be in its partition. This doesn't cover all trees to its left and at most $\sqrt{q}$ distant as some of these are in a neighboring partition. In step 2, we update using information from the neighboring partition. The values of *Rightmost* computed by Figure 6 may not be the same as those computed by Figure 4 as in Figure 6 each root may look at data from up to $2\sqrt{q}-1$ trees to its left (but at least $\sqrt{q}$ trees to its left) while in Figure 4, each tree looks at exactly $\sqrt{q}$ trees (provided there are this many) on the left. However, the values computed still result in a correct implementation as the values of *top* obtained by using Figure 5 next and the code:

$top(i, 0) = R(i, 0)$;
**if** $i - Rightmost(i, 0) \le q$ **then** $top(i, 0) = 1$;

are unchanged.

The code of Figure 6 can be pipelined to compute *Rightmost* for the roots for all $\sqrt{q}$ base columns of the $N\times\sqrt{q}$ partition in $O(\sqrt{q})$ time. This pipelining is accomplished by first doing the step 1 computation for all $\sqrt{q}$ columns in a pipelined way and then doing the step 2 computation for all $\sqrt{q}$ columns.

---

**Procedure** NewRoots

{new version of Figure 4}

{step 1, update within a partition}

$l := \log_2 \sqrt{q}$; {root level}

$t(i,l) := Rightmost(i,l)$, $0 \le i < N/\sqrt{q}$;

$Rightmost(i,l) := -\infty$, $0 \le i < N/\sqrt{q}$;

**for** $k := 1$ **to** $\sqrt{q}-1$ **do**

**begin**

$y(i,l) := t(i-1,l)$ , $1 \le i < N/\sqrt{q}$, $i \bmod \sqrt{q} = k$;

$Rightmost(i,l) := \max\{Rightmost(i,l), y(i,l)\}$, $1 \le i < N/\sqrt{q}$, $i \bmod \sqrt{q} = k$;

$t(i,l) := \max\{y(i.l), t(i,l)\}$, $1 \le i < N/\sqrt{q}$, $i \bmod \sqrt{q} = k$;

**end**

{step 2}

{transfer across partition boundaries}

$y(i,l) := \max\{Rightmost(i,l), t(i,l)\}$, $1 \le i < N/\sqrt{q}$, $i \bmod \sqrt{q} = \sqrt{q}-1$;

$y(i,l) := y(i-1,l)$, $1 \le i < N/\sqrt{q}$, $i \bmod \sqrt{q} = 0$;

{update within partition}

**for** $k := 1$ **to** $\sqrt{q} - 1$ **do**

**begin**

$y(i,l) := y(i-1,l)$, $1 \le i < N/\sqrt{q}$, $i \bmod \sqrt{q} = k$;

$Rightmost(i,l) := \max\{Rightmost(i,l), y(i,l)\}$, $1 \le i < N/\sqrt{q}$, $i \bmod \sqrt{q} = k$;

**end**

**end**

---

**Figure 6** Pipelineable procedure to update roots

Hence, by using $O(\sqrt{q})$ memory per processor and by pipelining the algorithms of Figures 3, 5, and 6, one can compute $top^q$ for a binary image in $O(\sqrt{q})$ time.

## 4 *O*(1) Memory Algorithm

Let $U(i,l)$ be the rightmost leaf node that has $R$ value 1 (i.e., $R(U(i,l),0) = 1$) in the subtree with root $(i,l)$. So, $U(0,1) = 0$ and $U(1,2) = 6$ in the example of Figure 2(a). Let $(root(i), \log_2(\sqrt{q}))$ be the root of the tree that contains the leaf node $(i, 0)$. In Figure 2(a), $root(3) = 0$ and $root(14) = 3$. The value $Rightmost(i, 0)$ computed by the $O(\sqrt{q})$

memory algorithm of Section 3 is the maximum of the following quantities:

a) $\max\{j \mid (j, 0)$ is a leaf in the same tree as $(i, 0)$, $R(j, 0) = 1$, and $j < i\}$

b) $\max\{U(j, \log_2(\sqrt{q})) \mid j < root(i)$ and $\lfloor j/\sqrt{q} \rfloor = \lfloor i/\sqrt{q} \rfloor\}$ (i.e., maximum over trees to the left of $root(i)$ but in the same $\sqrt{q}$ partition)

c) $\max\{U(j, \sqrt{q}) \mid \lfloor j/\sqrt{q} \rfloor = \lfloor i/\sqrt{q} \rfloor - 1\}$ (i.e., location of rightmost 1 in the $\sqrt{q}$ partition to the left)

d) if $R(i, 0) = 0$ then $-\infty$ else $i$

If we omit the second assignment statement of Figure 5, then only the max of the quantities b), c), and d) is computed as $Rightmost(i, 0)$. Our $O(1)$ memory algorithm computes $N(i, 0) = \max j$ such that $R(j, 0) = 1$ and $(j, 0)$, $j \le i$, is in the same tree as $(i, 0)$ (i.e., the max of a) and d)) by performing $\sqrt{q} - 1$ shifts along columns of the base PEs of the pyramid. This takes $O(\sqrt{q})$ time. Hence, we need consider only the computation of b) and c). The algorithm for b) and c) is obtained from that of Section 3 by omitting the second assignment statement in the **for** loop of Figure 5 and rearranging the remaining computations. The $O(\sqrt{q})$ memory algorithm of Section 3 has the following components:

(a) Phase 1 : Upward phase (Figure 3)

(b) Phase 2, Stage 1, Step 1 : New roots (Figure 6)

(c) Phase 2, Stage 1, Step 2: New roots (Figure 6)

(d) Phase 2, Stage 2 : Update *Rightmost* for non root nodes (Figure 5)

Each of these operates on an $N \times \sqrt{q}$ partition of the pyramid base together with the pyramid ancestors of these base processors. Within an $N \times \sqrt{q}$ partition the processing for the columns is pipelined. First (a) is done for all $\sqrt{q}$ columns, then (b), then (c) and finally (d). $O(\sqrt{q})$ memory is needed, per processor, to save the results of (a), (b), (c), and (d) for the $\sqrt{q}$ columns. To reduce the space requirement to $O(1)$, we cannot perform (a), (b), (c), (d) in this sequence. Rather, we must perform these concurrently.

Each *time step* of our $O(1)$ memory algorithm can be divided into four slots:

**SlotA** In this time slot the assignment statement within the **for** loop of Figure 3 or the first two statements of Figure 3 are computed.

**SlotB** The computations within the **for** loop of step 1 of Figure 6 are done in this time slot.

**SlotC** This is for the work within the **for** loop of step 2 of Figure 6

**SlotD** First assignment statement of **for** loop of Figure 5

For the concurrent computation, each processor employs three variables *V, W,* and

*X*. *V* is used in an upward pass from leaves to root. In this, each node other than the root computes its *U* value as defined at the start of this section. For the root, the *V* value is the max of its *U* value and the value of b). The *W* value for a root is the value of b), and the *X* value for a root is the value of c). Once a root computes a *W* or *X* value, this value is transmitted down the tree to the leaf nodes for use in the computation of *Rightmost*$(i, 0)$.

The work done by each processor in each time slot of a time step is described below:

**SlotA** Leaf nodes set their *V* value to be the value of *Rightmost* as computed by lines 1 and 2 of Figure 3. Nodes other than leaves and roots set their *V* value to be the max of the *V* values of their children. Root nodes set their *V* value to be the max of their children values and their *W* value.

**SlotB** Each root that is not the root of the first tree in its $\sqrt{q}$ tree partition sets its *W* value to be the *V* value of the root on its left.

**SlotC** The first root of each $\sqrt{q}$ tree partition sets its *X* value to be the *U* value of the root on its left. This corresponds to the Figure 6, step 2 transfer across partition boundaries. Remaining roots set their *X* value to be the *X* value of the root on their left. This corresponds to the rightward shifts of the step 2 **for** loop of Figure 6.

**SlotD** Non root nodes set their *W* and *X* values to be those of their parents.

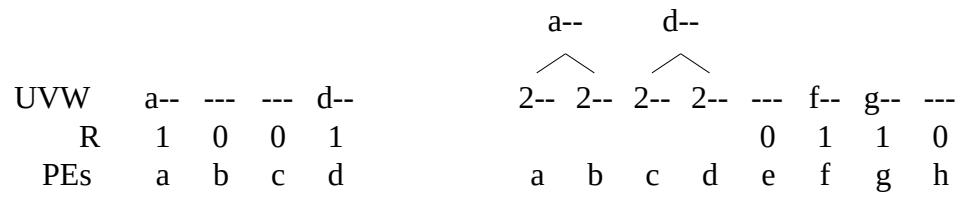
Within each partition of $\sqrt{q}$ trees, the computations are staggered in time. The *j*'th tree of each partition begins its computation in time step *j*. In time step 1, tree 1 begins with base column 1 of the $\sqrt{q}$ columns that it is to work on (see Section 3); in time step 2, it begins working on column 2; in time step 3, it commences on column 3; and so on. Tree 2 begins to work on its column 1 data in time step 2; it starts work on its column 2 data in time step 3; and so on. Generally, tree *j* of a partition begins to work on column *k* of the $\sqrt{q}$ columns assigned to it in time step $j+k-1$.

Let us examine the computations in a single partition of $\sqrt{q} = 4$ trees. Consider the example of Figure 2(a) which shows the *R* values for 16 elements of a single column (say column 1) of the $\sqrt{q}$ columns assigned to this set of trees. The processor indices 0-15 of Figure 2(a) will be mapped to the characters a-p for compactness and we shall use the symbol - to denote $-\infty$. The *V*, *W*, and *X* values of a node will be denoted by a three character string with the first being the value of *V*, the second that of *W*, and the third that of *X*. For first column data these values are drawn from the set {a, b, ..., p, -}. For data from other columns we simply put in the column number (2, 3, or 4).

In time step 1, tree 1 begins. In slot A, the leaves compute their *V* values as indicated in Figure 7(a). In the remaining slots no useful work is done as all involved

variable values are their initial ones which are assumed to be -. In slot A of time step 2, the leaves of tree 1 compute new V values based on column 2 data and the level 1 nodes of this tree compute V values that are the max of their children's step 1 V values. Also, in this slot, the leaves of tree 2 compute V values based on their column 1 data. The new V values for trees 1 and 2 are shown in Figure 7(b). A value of 2 indicates a value based on column 2 data. No useful work is done in the remiaing time slots of step 2.

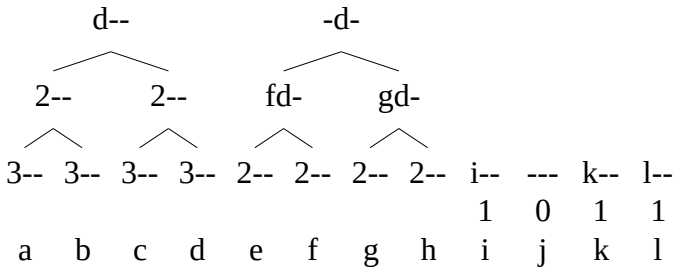
---



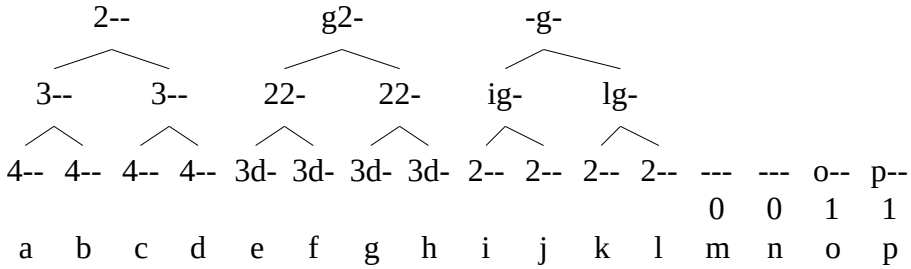

(a) Step 1, slot A, tree 1




(b) Step 2, slot A, trees 1 and 2




(c) Step 3, trees 1, 2 and 3

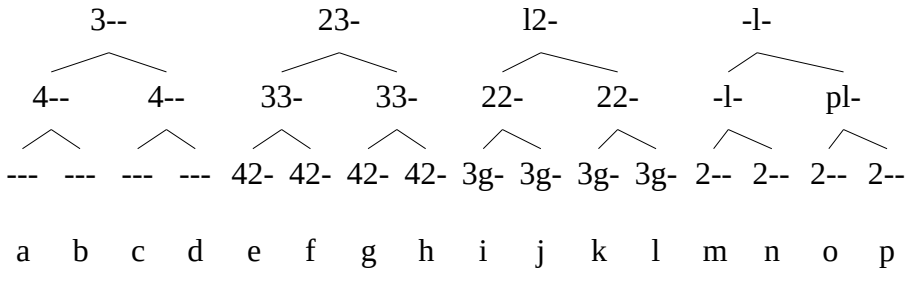



(d) Step 4, trees 1, 2, 3 and 4

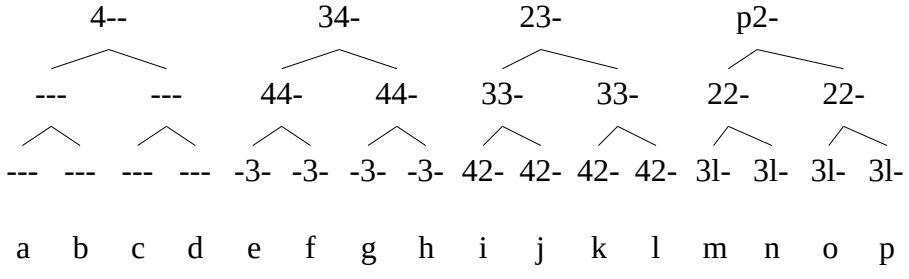
---

**Figure 7** Example for *O* (1) memory algorithm (continued)

In slot A of time step 3, the leaves of tree 1 compute new V values based on column 3 data, the level one nodes compute V values by taking the max of their children's V values from step 2 (these correspond to column 2 data), and the root computes its V value




(e) Step 5, trees 1, 2, 3 and 4




(f) Step 6, trees 1, 2, 3 and 4

**Figure 7** Example for $O(1)$ memory algorithm

taking the max of its children's step 2 V values and its current W value. In tree 2, the leaves compute V values for column 2 data and the level 1 nodes compute these values for column 1 data. Tree 3 is activated in this time step and its leaves compute V values using column 1 data. In slot B of this time step, the root of tree 2 sets its W value to be the V value of the root of the first tree (at this time, this V value is the position of the rightmost 1 in the trees in the same partition and to the left of tree 2). In slot D, the level one nodes of tree 2 set their W values to be that of their parent. The new V and W values are shown in Figure 7(c).

In slot A of step 4, the leaves of tree 1 compute V values for column 4 data, their parents do this for column 3 data, and the root does this for column 2 data. In tree 2, the leaves compute V values for column 3 data, their parents do this for column 2 data, and the root does this for column 1 data. Notice that the root can do this as the W value needed has already been obtained from tree 1. In tree 3, the leaves and their parents compute V values and in tree 4, the leaves become active and compute V values for their column 1 data. In slot B of this step, the roots of trees 2 and 3 get new W values. In the

case of tree 3, this is the position of the rightmost 1 (in column 1) that is in the same partition but in a tree to the left of tree 3. In the case of tree 2, the W value is the location of the rightmost one in the column 2 data examined by tree 1. In slot D, the level one nodes of trees 2 and 3 and the leaves of tree 2 get new W values. The relevant data values are shown in Figure 7(d). Notice that the leaves of tree 2 now know the position of the rightmost 1 in column 1 of tree 1. In the next time step, they will know this for column 2 data, and so on.

Figures 7(e) and (f), respectively, give the data values following steps 5 and 6. Following slot A of step 6, the righmost root in each $\sqrt{q}$ partition has a V value that is the location of the rightmost 1 in the column 1 data assigned to the partition. In slot C of this step, this information is transferred to the first root in the next partition. The value goes into the X variable of this root. Hence, in slot C of step 9 this value reaches the rightmost root of the next partiton. In slot D of this step, the children of this root get this value and in step 10 slot D, the leaves get the value. In step 11, these leaves get this value for column 2 data and in step 13, they get it for column 4 data and the algorithm terminates. In general, the algorithm needs to be run for $2\log_2\sqrt{q} + 3\sqrt{q} - 3$ steps.

Hence, in $O(\sqrt{q})$ time each of the pyramid base processors can get the a), b), c), and d) values and take the max of these to obtain the value of *Rightmost*. So, the $q$-step expansion can be computed in $O(\sqrt{q})$ time using $O(1)$ memory per processor. Some bookkeeping is needed to identify the data column associated with each value. This bookkeeping takes only $O(1)$ space (in the worst case a column identifier is kept with each value though timing information can be used to compute this column identifier).

## 5 General Values of $q$

Our development of Sections 3 and 4 assumed that $q$ was a perfect square and that $\sqrt{q}$ is a power of 2. The algorithms developed are easily applied to arbitrary $q$ by using the least $q' \leq q$ that is a perfect square and such that $\sqrt{q'}$ is a power of 2. The value of *Rightmost* computed using such a $q'$ are limited in that nodes at a distance of at most $2q'$ are examined. However, the check for setting $top(i, 0)$ remains:

$top(i, 0) = R(i, 0)$;
**if** $i - Rightmost(i, 0) \leq q$ **then** $top(i, 0) = 1$;

Hence, *top* is correctly computed for a $q$–step expansion. Since $q' \leq 4q$, the complexity remains $O(\sqrt{q})$.

## 6 Conclusions

By pipelining computations on a pyramid, we have shown how the $q$-step expansion and $q$-step shrinking of a binary image can be computed in $O(\sqrt{q})$ time. Two algorithms for this have been described. One uses $O(\sqrt{q})$ memory per processor and the other $O(1)$. The former uses a simpler pipelining scheme than used by the latter. Our results improve the previous best pyramid algorithms for these problems. These previous algorithms [ROSE87] have complexity $O(q)$.

## 7 References


[JENQ90] J. Jenq and S. Sahni, "Reconfigurable mesh algorithms for image shrinking, expanding, clustering, and template matching", Proceedings of the Fifth International Parallel Processing Symposium, 1991.

[RANK89] S. Ranka and S. Sahni "Hypercube algorithms for image transformations", Proceedings of the 1989 International Conference on Parallel Processing, pp 24-31.

[RANK90] S. Ranka and S. Sahni, *Hypercube algorithms for image processing and pattern recognition*, Springer-Verlag, 1990.

[ROSE82] A. Rosenfeld and A. C. Kak, *Digital picture processing*, Academic Press, 1982.

[ROSE87] A. Rosenfeld, "A note on shrinking and expanding operations in pyramids", Pattern Recognition Letters, 1987, pp 241-244.