Two Algorithms for Computing the Euclidean Distance Transform

Marina L. Gavrilova
Department of Computer Science
University of Calgary
Calgary, Alberta Canada
marina@cpsc.ucalgary.ca

Muhammad H. Alsuwaiyel
Department of Information and
Computer Science
King Fahd University of
Petroleum & Minerals
Dhahran 31261, Saudi Arabia
suwaiyel@ccse.kfupm.edu.sa

Abstract. Given an $n \times n$ binary image of white and black pixels, we present two algorithms for computing the distance transform and the nearest feature transform using the Euclidean metric. The first algorithm is a fast sequentioal $O(n)$ time algorithm. The second is an optimal $O(n)$ time parallel algorithm that runs on a linear array of n processors.

Keywords: Feature transform, Distance transform, Euclidean distance, Parallel algorithms, Image processing.

1 Introduction

Given an $n \times n$ binary image $\mathcal{I}$ of white and black pixels, the distance transform of $\mathcal{I}$ is a map that assigns to each pixel the distance to the nearest black pixel, referred to as *feature*. The feature transform of $\mathcal{I}$ is a map that assigns to each pixel the feature that is nearest to it. The distance transform was first introduced by Rosenfeld and Pfaltz [10], and it has a wide range of applications in image processing, robotics, pattern recognition and pattern matching [8]. The distance metrics used to compute the distance transform include the L_1, L_2 and L_∞ metrics, with the L_2 (Euclidean) metric being the most natural, and rotational invariant.

Several algorithms have been proposed for these metrics. One approach is to grow clusters or neighborhoods around each feature p consisting of those pixels whose nearest feature is p. This approach has been taken in [1, 4] and [12] to obtain sequential and parallel algorithms, respectively. A similar approach that simulates circular waves originating at all features of the image is described in [9]. This method, while as complex as the previous methods, is also not suitable for parallelization.

An alternative approach, pioneered by Rosenford and Pfaltz [10], is based on the idea of dimension reduction. Paglieroni [8] extends this approach to a broader

class of distance functions. Breu *et al* [2] compute the distance transform by computing the Voronoi diagram in $O(n)$ time. They achieve this bound by refining the merge step in the classical divide-and-conquer algorithm for the construction of the ordinary Voronoi diagram, so that the merge step takes $o(n)$ time. A modification of this algorithm that first computes the Voronoi diagram of segments and then obtains the feature transfrom was recently devised in [6]. However, the algorithms in [2, 3, 6], though linear, are computationally expensive compared to those given for other metrics. An attempt to develop a generalized algorithm that is applicable to a wide class of distance transforms has been made in [7]. However, the algorithm given is rather complex and lacks a proof of correctness.
As to parallel algorithms, the cost optimal methods for the EREW PRAM, the mech and the hypercube architectures were proposed in [11, 5].

In this paper, we propose two simple and optimal algorithms for computing the distance transform and the nearest feature transform using the Euclidean metric. We first present a sequential algorithm that processes the rows twice: in a top-down scan and a bottom-up scan. The algorithm maintains a polygonal chain C containing all the necessary information to compute the nearest feature for each pixel of the currently processed row. A marking characteristic of this algorithm is updating the polygonal chain dynamically as the image is swept row by row. The information gathered while processing the previous row is utilized to compute the nearest features for the next row. This results in significant improvement of the algorithm efficiency in comparison with other linear time algorithms.

The sequential algorithm can be parallelized easily by using the method of dimension reduction. Although the mesh seems to be the most natural network architecture for parallelizing the algorithm, it results in waste of resources. This is due to the fact that $O(n)$ processors will end up processing one row in linear time, while one processor is indeed enough, as in the sequential algorithm. This motivates parallelizing the algorithm on a linear array of processors. If the processors are not powerful enough to store one row of the image each, systolic computation can be used to pipeline the pixels and thus to keep all processors busy as much as possible. This results in an $O(n)$ time algorithm with linear total cost. To the authors' knowledge, the linear array was not considered before as a suitable architecture for this problem.

For brevity, in our description of the algorithms, we will confine our attention to computing the feature transform, and the distance transform will not be mentioned explicitly.

2 The sequential algorithm

Let $\mathcal{I}$ be the input $n \times n$ image. It is assumed that $\mathcal{I}$ is an $n \times n$ array I of zeroes and ones, representing white and black pixels, respectively. A pixel will be represented by its coordinates, that is, (i, j) will denote the pixel in row i and column j, where $1 \leq i, j \leq n$. Given a pixel (i, j), $f(i, j)$ will denote the feature that is nearest to (i, j). For simplicity, we will assume that $f(i, j)$ is unique, and hence f is a function from the set of pixels to the set of features. Given a pixel (i, j), $\delta(i, j)$ will denote the *square* of the Euclidean distance between (i, j) and $f(i, j)$.

Consider the pixels in row i, where i is between 1 and n, and let (i', j') be the nearest feature to pixel $p = (i, j)$ among all features in rows $1, 2, \ldots, i$. Clearly, if $i' > 1$, and there is another feature (i'', j') with $i'' < i'$, then (i'', j') cannot be the nearest feature to pixel p. Let S be the set of features on or above row i that are nearest to at least one pixel in row i. Let C denote the polygonal chain whose vertices are the centers of those features in S. It follows that C is monotone with respect to the horizontal line L_i passing by the centers of pixels in row i.

This observation suggests the following approach for finding all nearest features. We perform two sweeps on the image $\mathcal{I}$: one from top to bottom and the other from bottom to top. In the top-down sweep, we compute for each pixel (i, j) its nearest feature $f_{td}(i, j)$ among all features on or above row i and its corresponding $\delta_{td}(i, j)$ value. In the bottom-up sweep, we compute for each pixel (i, j) its nearest feature $f_{bu}(i, j)$ among all features below row i and its corresponding $\delta_{bu}(i, j)$ value. Finally, we set $f(i, j) = f_{td}(i, j)$ if $\delta_{td}(i, j) \leq \delta_{bu}(i, j)$, otherwise we set $f(i, j) = f_{bu}(i, j)$. In each scan, the algorithm maintains a polygonal chain C, implemented as an array that "slides" vertically through the image, top-down in the first scan, and bottom-up in the second scan. This chain contains all the information needed to compute the nearest feature for each pixel in row i above or below row i, depending on the direction of each scan. Since the descriptions of the two sweeps are identical, we will discuss only the top-down sweep. For this reason, we will drop the subscripts from f_{td} and δ_{td}, and simply use f and δ instead.

Now we give a detailed description of the algorithm for implementing the top-down scan. C will be represented by an array of n 2-tuples, such that an entry $C[j]$ is either a feature or $(0, 0)$. If it is a feature, then the center of that feature is a vetex in the chain; otherwise, it is not. The intention is that just *after* row i has been processed, $C[j]$ is a feature if and only if it is the closest feature to *at least* one pixel in row i. In what follows, an entry in array C will be called empty if it contains $(0, 0)$, otherwise, it will be called nonempty.

Suppose that $C[j]$ is nonempty. For fast access to its left and right nonempty neighbors, we will make use of the two functions *left*(p) and *right*(p), which

return, respectively, the two features, if any, that are nearest to feature p to the left and right of p in C. If $C[j]$ is the leftmost nonempty entry in array C, then $\mathit{left}(C[j]) = (0, 0)$. Similarly, if $C[j]$ is the rightmost nonempty entry in array C, then $\mathit{right}(C[j]) = (0, 0)$. Figure 1 provides an example of this representation, as well as the polygonal chain after processing row 4 and before processing row 5.

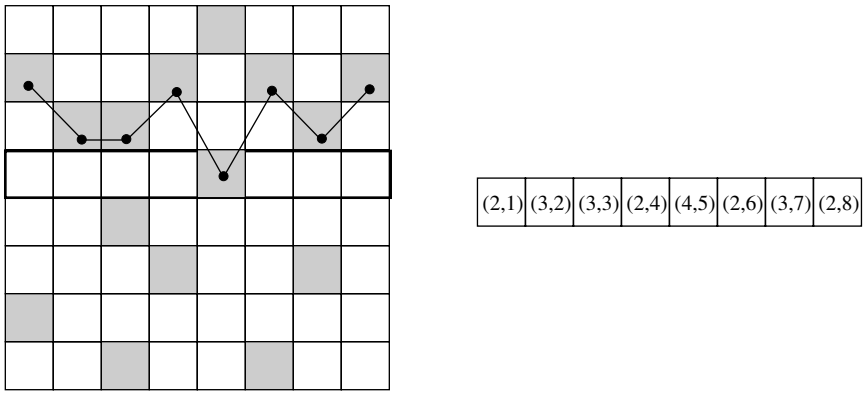

Figure 1: Example of representations and the polygonal chain.

Let p and q be two vertices of the chain. Then, $B(p, q)$ will denote the perpendicular bisector of the line segment $\overline{pq}$. We will denote by V_1 and V_n the two vertical lines defined by the two equations $x = 1$ and $x = n$, respectively. Initially, all entries in C are empty, that is, the chain is empty. When processing the topmost row, for $1 \leq j \leq n$, $C[j]$ is set to $(1, j)$ if and only if pixel $(1, j)$ is a feature. When processing row i, C is updated by setting $C[j]$ to (i, j) if and only if (i, j) is a feature. Next, C is updated further by removing those features that cannot be the nearest to any pixel in row i or below.

There are two tests corresponding to whether a vertex in the chain is extreme (i.e. it has no left or right neighbors) or not. Suppose that $p = (i, j)$ is the leftmost vertex in the chain, and $q = \mathit{right}(p)$. If $B(p, q)$ intersects with V_1 above row i, then p cannot be the nearest feature to any pixel in row i or below. Hence, p should be removed from the chain. This process is applied iteratively until the perpendicular bisector of the leftmost line segment in C does not intersect with V_1 above row i. The same procedure is applied starting from the righmost feature in the chain. In this case, the test is performed against the vertical line V_n. This is illustrated in Figure 2. In this figure, p, q, r and s will be removed from the chain.

The second test to be applied to the chain is concerned with internal vertices of C (unless C consists of two vertices or less). Consider Figure 3(a). In this

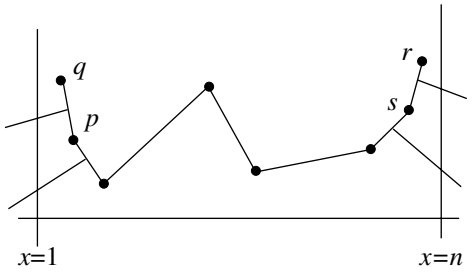

Figure 2: Example of the extreme features test.

figure, $B(p,q)$ and $B(q,r)$ intersect *above* L_i, the line passing by the centers of pixels in row i. As will be shown later, feature q cannot be the nearest to any pixel in row i or below. Therefore, q should be deleted from the chain. After it has been removed, its right neighbor may also be removed, and so on. For instance, in Figure 3(a), feature r will also be deleted. Indeed, it may be the case that after the removal of r, feature q, whose right neighbor has changed in the chain, fails the test, and hence should be removed, as shown in Figure 3(b) This too, may result in a sequence of deletions in the backward direction, which we will refer to as *backtracking*.

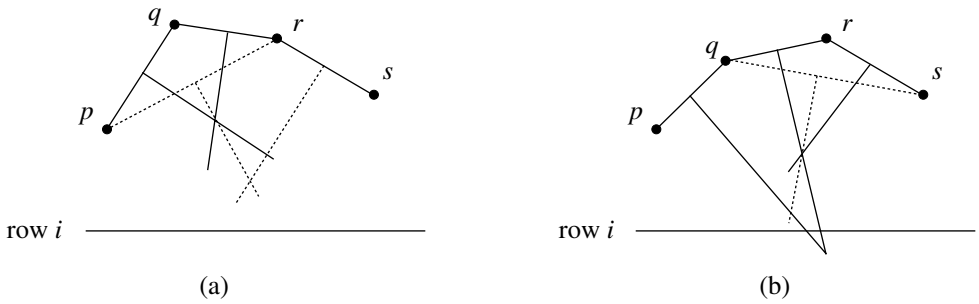

Figure 3: Example of removal of internal features.

After applying both tests, each feature in the chain is the nearest feature of at least one pixel in row i, and the nearest feature of evey pixel in row i can be found in the chain. To do the assignments of features to pixels in row i, the perpendicular bisectors of all line segments in the final chain are computed. These bisectors partition the set of pixels in row i into groups of consecutive pixels, with each group having the same nearest neighbor.

It is interesting to note that two scans of the chain, one forward and another backward may not be enough, that is, a mechanism of either backtracking or

lookahead is needed for a proper maintenance of the chain. Figure 4 shows an instance in which the algorithm that does not implement backtracking every time a feature gets deleted from the chain fails to remove all unneeded features. In Figure 4(a), the algorithm is performing the forward scan, after which r is removed, as the bisectors $B(q,r)$ and $B(r,s)$ intersect above the current row being processed (see Figure 4(b)). This is followed by removing a number of features located between features p and q of the chain during the backward scan, which is shown in Figure 4(c). However, all features between s and t should also have also been removed since they can not be nearest neighbors to any pixel in row i or below. Thus, backtracking should be performed every time a pixel is deleted from the polygonal chain.

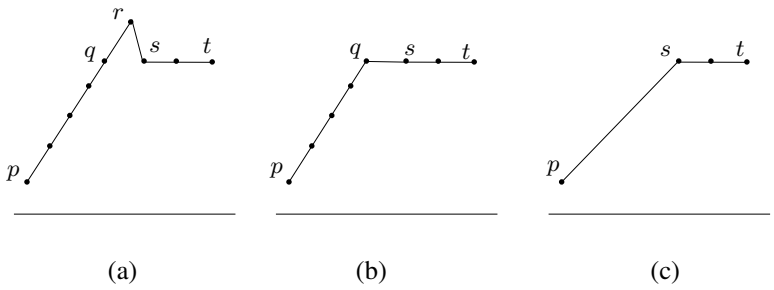

Figure 4: An instance in which the two scans fail to remove necessary features from the chain.

The above description of the algorithm for the top-down scan can be stated more precisely in the following steps:

Step 1 (Initialization) For $1 \leq j \leq n$, if $I[1,j] = 0$ then set $C[j] = (0,0)$, else set $C[j] = (1,j)$. Scan the first row from left to right and right to left to compute $f(1,j), \delta(1,j)$, *left*(j) and *right*(j), for all $j, 1 \leq j \leq n$. Set $i = 2$.

Step 2 (Process row i)

Step 2.1 (Add features in row i to the polygonal chain)

Scan C from left to right. For $j = 1, 2, \ldots, n$, if pixel (i,j) is black , then set $C[j] = (i,j)$.

Step 2.2 (Perform test 1)

If $|C| \leq 1$ then go to Step 3. Let p and q be the two leftmost features in C. While $B(p,q)$ intersects V_1 and q is not the rightmost feature in C do the following: Remove p from C, set $p = q$ and $q =$ *right*(q).

If $|C| \leq 1$ then go to Step 3. Let q and p be the rightmost features in C. While $B(p, q)$ intersects V_n and q is not the lefttmost feature in C do the following. Remove p from C, set $p = q$ and $q = \mathit{left}(q)$.

Step 2.3 (Perform test 2; advance, backtracking whenever it applies.)

If $|C| \leq 2$ then go to Step 3. Otherwise, let p, q and r be the three leftmost features in C, and repeat Step 2.4 until all features in C have been processed.

Step 2.4

If $B(p, q)$ and $B(q, r)$ intersect below row i then
(Advance) Set $p = q$, $q = r$, $r = \mathit{right}(r)$.
else
(Backtrack) Set $q = p$, $p = \mathit{left}(p)$, $\mathit{right}(q) = r$, $\mathit{left}(r) = q$.

Step 3 (At this point, C has been refined. Assign features to pixels.)

Step 3.1 If $|C| = 1$ then let $f(i, j) = p$ for all $j, 1 \leq j \leq n$, where p is the feature in C.

Step 3.2 If $|C| \geq 2$ then let p and q be the two leftmost features in C, set $k_1 = 1$, and repeat Step 3.3 until $q = (0, 0)$ (i.e. the rightmost feature in C has been processed).

Step 3.3 Let x be the intersection point of $B(p, q)$ and row i.
Set $k_2 = \lfloor x \rfloor$.
For $j = k_1, k_1 + 1, \ldots, k_2$, set $f(i, j) = p$.
Set $k_1 = k_2 + 1$, $p = q$, $q = \mathit{right}(q)$.

3 Correctness of the algorithm

In this section, we prove the correctness of the algorithm. The proof is provided for top-down sweep. In the case of bottom-up sweep, the proof is identical.

Lemma 1 All nearest features of pixels on row i can be found in the polygonal chain.

Proof.
As the image is swept from top to bottom, all features on or above row i get inserted into the polygonal chain. Hence, we only need to show that if feature q gets removed from the chain, then it cannot be the nearest feature to any pixel on row i or below. Let q be a feature that has been deleted. We have three cases to consider. If q was replaced by another feature p in the same column then any pixel x on or below row i is closer to p than q. If q was removed because it failed test 1 (see Figure 2), then since C is monotonic with respect to row i,

the center of any pixel x on row i or below belongs to the half plane defined by bisector $B(p,q)$ containing p. That is, x is closer to p than to q. Finally, if q gets removed because it fails test 2 (see Figure 3(a)), then, as shown in the figure, the center of any pixel x on row i or below belongs to either the half plane defined by bisector $B(p,q)$ containig p or the half plane defined by bisector $B(q,r)$ containig r. That is, x is either closer to p or r than to q. □

Note that the converse of Lemma 1 is not true. That is, there are features in the chain that are not assigned by the algorithm to any pixel in row i. These features may be needed when processing subsequent rows. This is illustrated in Figure 5.

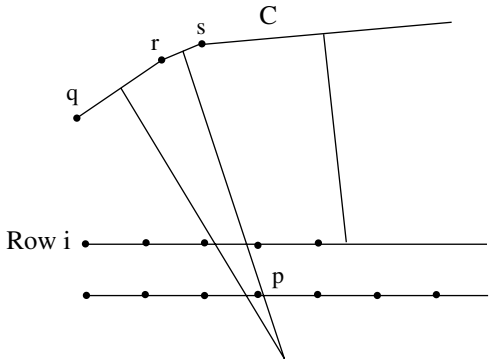

Figure 5: Example of a feature in the chain that is not the nearest feature to any pixel in row i.

Lemma 2 All pixels (i,j), $1 \leq j \leq n$, in row i are assigned their correct nearest feature $f\,(i,j)$.

Proof. By Lemma 1, all nearest features for pixels in row i are found in the polygonal chain. Now, we show that the algorithm assigns to each pixel in row i its nearest feature in the chain. We show that, when processing row i, each pixel located between the bisectors $B(q_{j-1},q_j)$ and $B(q_j,q_{j+1})$ (to the left of $B(q_j,q_{j+1})$ if $j=1$) has q_j as its nearest feature. Suppose there is a pixel p on row i that lies to the left of $B(q_j,q_{j+1})$, but its nearest feature is q_k, for some $k>j$. That is, feature q_k lies to the right of feature q_j in the polygonal chain. The proof is similar if q_k lies to the left of q_j. Since pixel p is closer to q_k than to q_{k-1}, both p and q_k lie in the same half-plane defined by bisector $B(q_{k-1},q_k)$ and containing q_k. Since both q_k and q_{k-1} are above row i and q_k is to the right of p it follows that bisector $B(q_{k-1},q_k)$ intersects row i to the left of pixel p. Consequently, feature q_{k-1} lies above feature q_k (see Figure 6).
By construction, bisectors $B(q_{k-1},q_k)$ and $B(q_{k-2},q_{k-1})$ intersect below row i, for otherwise feature q_{k-1} should have been deleted from the polygonal chain.

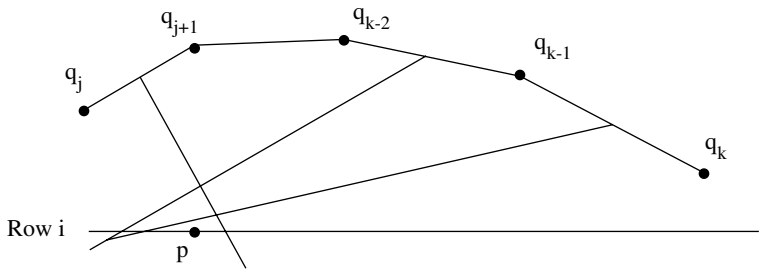

Figure 6: Example of bisectors intersecting to the left of p

It follows that q_{k-2} is above q_{k-1}, and bisector $B(q_{k-2}, q_{k-1})$ intersects with row i to the left of p. Applying the same reasoning iteratively to features $q_{k-2}, q_{k-3} \ldots q_j$, we conclude that bisector $B(q_j, q_{j+1})$ lies to the left of p. This contradicts the assumption that p lies to the left of bisector $B(q_j, q_{j+1})$. It follows that $f(p) \neq q_k$. A similar argument shows that p lies to the right of bisector $B(q_{j-1}, q_j)$ if $j > 1$.

□

As to the time complexity, each of the top-down and bottom-up sweeps costs $O(n^2)$ time, as each row requires $O(n)$ processing time. To see this, observe that when processing any row, each feature is inserted into the chain exactly once and deleted at most once. Updating the *left* and *right* pointers takes $O(n)$ time for the entire row. Finally, the nearest feature to each pixel is assigned exactly once. Hence, we have the following theorem:

Theorem 1 The algorithm described above finds the distance transform and the nearest feature transform of a binary $n \times n$ image in $O(n^2)$ time, which is linear in the input size.

4 The parallel algorithm

The algorithm in the previous section, which we will call ALGORITHM 1, has the following property that makes it very efficient on a sequential machine. The rows are processed sequentially, row i is processed after the completion of processing row $i-1$. To parallelize the algorithm, the idea of dimension reduction is used. Simply stated, all information needed to process each row is made available *a priori*. For this purpose, let $g(i,j)$ denote the nearest feature to pixel (i,j) in column j. Clearly, $g(i,j), 1 \leq i,j \leq n$, can all be computed in $O(n^2)$ with two sweeps over the image: top-down and bottom-up. Hence, the algorithm in the previous section can be modified easily by adding a preprocessing step, which computes all $g(i,j)$'s, and then building the chain for each row from

scrach. We will call this algorithm ALGORITHM 2. Although ALGORITHM 2 performs redundant computations, it can be parallelized easily by processing all rows independently and in parallel. It doesn't seem that the algorithm can be parallelized efficiently on the PRAM. For the PRAM, more efficient algorithms exist that use the scan operator to sweep the image vertically and horizontally without the need for extra data structures as in ALGORITHM 2.

The most natural interconnection network architecture for parallelizing ALGORITHM 2 is a mesh of $n \times n$ procesors. In this case, each row of processors work independently on one row. The cost of implementing the algorithm on this architecture is $O(n^3)$, which is too high in view of the fact that onle one processor is needed to process one row in $O(n)$ time. Decreasing the size of the mesh to $n \times m$ processors, where $1 \leq m < n$ reduces the cost to $O(mn^2)$. Here each m processors work on one row.

If we let $m = 1$, and the processors have enough memory to store one row of the image, then ALGORITHM 2 can be implemented on a linear array of n processors. We only need to describe the preprocessing step, computing the $g(i,j)$'s, as the rest of the algorithm is exactly the same as in ALGORITHM 1, except that rows are processed in parallel. We will limit the description to the top-down evaluation of the $g(i,j)$'s. Let $P_1, P_2, \ldots, P_n$ be the n processors. For the preprcessing step, each pixel (i,j) in row i of the image travels starting from P_i to P_n in a synchronized fashion. First, P_1 computes $g(1,j)$ for all $j, 1 \leq j \leq n$. In the first step of movements, $g(1,1)$ moves into P_2, and $g(2,1)$ is computed. In the second step, $g(1,2)$ moves into P_2, and $g(2,1)$ moves into P_3, and the values of $g(2,2)$ and $g(3,1)$ are computed simultaneously. This pattern of moving the values of $g(i,j)$'s continues until $g(n,n)$ is computed.

The above approach implies a simple systolic computation, in which the the rows are fed to the processors one elements at a time (see Figure 7). In this case, pixel $(1,1)$ is first fed into P_1. Next, both $(1,2)$ and $(2,1)$ are fed simultaneously into P_1 and P_2, respectively. In the third time unit, $(1,3)$, $(2,2)$ and $(3,1)$ are fed into P_1, P_2 and P_3, and so on.

Clearly, the time required for the preprocessing step using pipelining is $O(n)$. This results in an optimal $O(n)$ time algorithm with toal cost in the order of $O(n^2)$.

5 Conclusion

We have presented two algorithms for the computation of the nearest feature transform and the distance transform, one sequential and the other is parallel. The sequential algorithm is a fast linear time algorithm that makes use of the line sweep method to avoid repetitive computations. The parallel algorithm is a time optimal algorithm that uses an array of n processors. In the case when

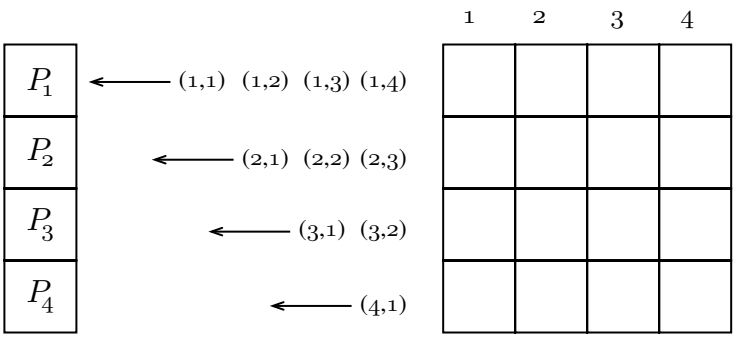

Figure 7: Example of systolic computation in the preprocessing step.

these processors are not powerful enough to hold data of size $O(n)$, they can be used to perform a systolic computation on the input image. Both algorithms are easy to implement, and with minor modifications will work for other metrics.

References

[1] G. Borgefors. Distance transformations in digital images. Computer Vision, Graphics and Image Processing, 34:344-371, 1986.

[2] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman. Linear time Euclidean distance transform algorithms. IEEE Transactions on Pattern Analysis and Machine Intelligence, 17 5:529-533, 1995.

[3] L. Chen and H. Y. H. Chuang. A fast algorithm for Euclidean distance maps of a 2-D binary image. Information Processing Letters, 5 1:25-29, 1994.

[4] P. Danielsson. Euclidean distance mapping. Computer Graphics and Image Processing, 14:227-248, 1980.

[5] A. Fujiwara, M. Inoue, T. Masuzawa, and H. Fujiwara. A simple parallel algorithm for the medial axis transform of binary images. In Proc. IEEE Second International Conference on Algorithms and Architecture for Parallel Processing, pages 1-8, 1996.

[6] W. Guan and S. Ma. A line-processing approach to compute Voronoi diagrams and the Euclidean distance transform. IEEE Transactions on Pattern Analysis and Machine Intelligence, 20 7:757-761, 1998

[7] T. Hirata. A unified linear-time algorithm for computing distance maps. Information Processing Letters, 58: 129- 133,1996.

[8] D. W. Paglieroni. Distance transforms: Properties and machine vision applications. CVGIP: Graphical Models and Image Processing, 54:56-74, 1992.

[9] I. Ragnemalm. Neighborhoods for distance transformation using ordered propogation. Computer Vision, Graphics and Image Processing, 56:399-409, 1992.

[10] A. Rosenfeld and J. L. Pfalz. Sequential operations in digital picture processing. Journal of the ACM, 13:471-494, 1966.

[11] O. Schwarzkopf. Parallel computation of distance transform. Algorithmica, 6:685-697, 1991.

[12] H. Yamada. Complete Euclidean distance transformation by parallel operation. Proc. 7th International Conference on Patter Recognition, pages 69-71, 1984.