The Journal of Supercomputing, 25, 177–185, 2003

Computing the Euclidean Distance Transform on a Linear Array of Processors

MARINA L. GAVRILOVA marina@cpsc.ucalgary.ca
Department of Computer Science, University of Calgary, Calgary, Canada

MUHAMMAD H. ALSUWAIYEL suwaiyel@ccse.kfupm.edu.sa
Department of Information and Computer Science, KFUPM, Dhahran, Saudi Arabia

Abstract. Given an $n \times n$ binary image of white and black pixels, we present an optimal parallel algorithm for computing the distance transform and the nearest feature transform using the Euclidean metric. The algorithm employs the systolic computation to achieve $O(n)$ running time on a linear array of n processors.

Keywords: feature transform, distance transform, Euclidean distance, parallel algorithm, linear array of processors, 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 algorithm was pioneered by Rosenfeld and Pfaltz [9], and it has a wide range of applications in image processing, robotics, pattern recognition and pattern matching. 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 Danielsson [3] and Yamada [11] to obtain sequential and parallel algorithms, respectively. Daniellson [3] describes a sequential nearest-neighbor transform algorithm that is nearly error-free. It runs in $O(nm)$ time, which is proportional to the size of the $n \times m$ image. Yamada [11] provides an exact parallel algorithm that takes $O(\max\{n, m\})$ using $O(nm)$ processors. A similar approach that simulates circular waves originating at all features of the image is described in Ragnemalm [8]. The algorithm is a sequential wavefront algorithm that runs in $O(nm)$ time. This approach, while as complex as the previous one, is not suitable for parallelization.

An alternative approach, pioneered by Rosenford and Pfaltz [9], is based on the idea of dimension reduction. The transform is first computed using the distance function in one lower dimension and then the 2D distance transform is computed.

Breu et al. [2] compute the distance transform by computing the Voronoi diagram in $O(n)$ time. A modification of this algorithm that first computes the Voronoi diagram of segments and then obtains the feature transform was recently devised in Guan and Ma [5]. An attempt to develop a generalized algorithm that is applicable to a wide class of distance transforms has been made in Hirata [6]. Borgefors [1] proposed a sequential algorithm that computes the distance transform in the L_1 metric by performing two scans: one is downward scan from left to right and another is upward scan from right to left.

Some cost optimal parallel algorithms have been also developed. For instance, Schwarzkof [10] suggested an $O(\log n)$ time algorithm for the city block distance transform on the mesh of trees with n^2 processors. Lee and Horng [7] proposed a chessboard transform algorithm. The algorithm runs in $O(\log n)$ time using $n^2/\log n$ processors on the EREW PRAM, in $O(\log n/\log\log n)$ time using $n^2 \log\log n/\log n$ processors on the CRCW PRAM, and in $O(\log n)$ time on an n^2-processor hypercube. Fujiwara et al. [4] developed cost optimal algorithms for the weighted distance transform. It runs in $O(\log n)$ time using $n^2/\log n$ processors on the EREW PRAM, and in $O(\log\log n)$ time using $n^2/\log\log n$ processors on the CRCW PRAM. Cost optimal algorithms for an n^2-processor mesh and an n^2-processor hypercube were also proposed in Fujiwara et al. [4].

In this paper, we propose a simple and optimal parallel algorithm for computing the distance transform and the nearest feature transform using the Euclidean metric. The algorithm builds the polygonal chains C_i for each row i. The polygonal chain contains all the necessary information to compute the nearest feature for each pixel of row i. The method of dimension reduction is used to devise a time optimal parallel 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 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.

2. The approach

This section describes the basis algorithm notions: building the polygonal chain and performing the tests that allow elimination of black pixels that cannot be feature. 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 $f(i, j)$.

2.1. The polygonal chain

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_i denote the polygonal chain for row i, whose vertices are the centers of those features in S. It follows that C_i is monotone with respect to the horizontal line L_i passing by the centers of pixels in row i.

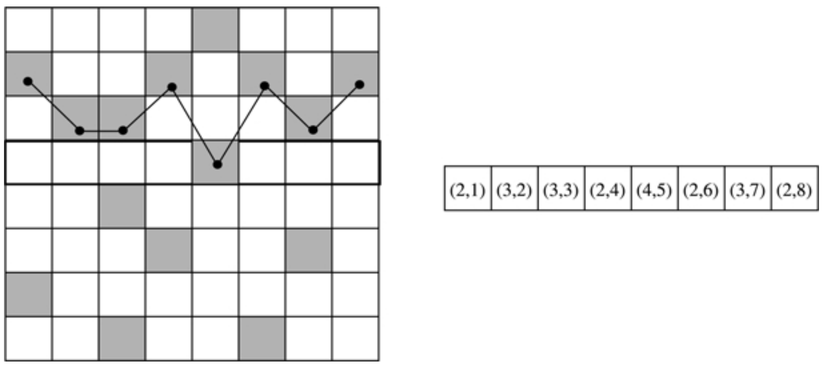
This observation suggests the following approach for finding all nearest features. For each row, we build two polygonal chains $C_{i,\,\mathrm{top}}$ and $C_{i,\,\mathrm{bottom}}$. To build the polygonal chain $C_{i,\,\mathrm{top}}$, we compute for each pixel (i, j) its nearest feature $f_{td}(i, j)$ among all features on or above row i and its corresponding $\delta_{td}(i, j)$ value. To build the polygonal chain $C_{i,\,\mathrm{bottom}}$, 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 $f(i, j) = f_{bu}(i, j)$.

This process can be viewed as performing two scans on the image $\mathscr{I}$: one from top to bottom and the other from bottom to top. The algorithm computes polygonal chains C_i for each row i. Each 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 construction of the polygonal chain for all the pixels on and above row i is identical to the construction of the polygonal chain for pixels on and below row i, we will discuss only the former. For this reason, we will drop the subscripts from f_{td} and δ_{td}, and simply use f and δ instead. We will also drop the subscripts top and bottom and use notation C_i to refer to the polygonal chain.

2.2. Building the chain

Now we give a detailed description of the algorithm for building the polygonal chain for pixels on and above the given row. C_i will be represented by an array of n 2-tuples, such that an entry $C_i[j]$ is either a feature or $(0, 0)$. If it is a feature, then the center of that feature is a vertex in the chain; otherwise, it is not. Suppose that $C_i[j]$ 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_i. If $C_i[j]$ is the leftmost nonempty entry in array C_i, then *left* $(C_i[j]) = (0, 0)$. Similarly, if $C_i[j]$ is the rightmost nonempty entry in array C_i, then *right* $(C_i[j]) = (0, 0)$. Figure 1 provides an example of this representation, as well as the polygonal chain after processing row 4 and before processing row 5.

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_i are empty, that is, the chain is empty. When preprocessing each row, for $1 \leq j \leq n$, $C_i[j]$ is set to $(1, j)$ if and only if pixel $(1, j)$ is a feature. When processing each row, C_i is updated by setting $C_i[j]$ to (i, j) if and only if (i, j) is a feature. Next,



(2,1)	(3,2)	(3,3)	(2,4)	(4,5)	(2,6)	(3,7)	(2,8)

Figure 1. Example of the polygonal chain and its array representation when processing the fourth row.

C_i is updated further by removing those features that cannot be the nearest to any pixel in row i.

2.3. *The extreme feature test*

There is a test corresponding to whether a vertex in the chain is extreme (i.e., has no left or right neighbors) or not. Suppose that $p = (i, j)$ is the leftmost vertex in the chain, and $q = right(p)$. If $B(p, q)$ intersects with V_1 above row i, then p cannot be the nearest feature to any pixel in row i. Hence, p should be removed from the chain. This process is applied iteratively until the perpendicular bisector of the leftmost line segment in C_i does not intersect with V_1 above row i. The same procedure is applied starting from the rightmost 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.

2.4. *The internal feature test*

The second test to be applied to the chain is concerned with internal vertices of C_i (unless C_i consists of two vertices or less). Consider Figure 3(a). In this figure, $B(p, q)$

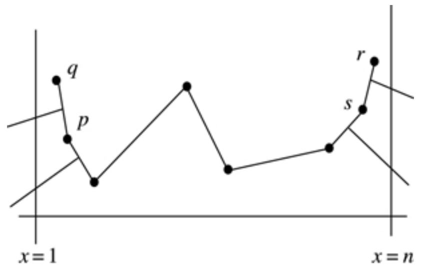


Figure 2. Example of the extreme features test.

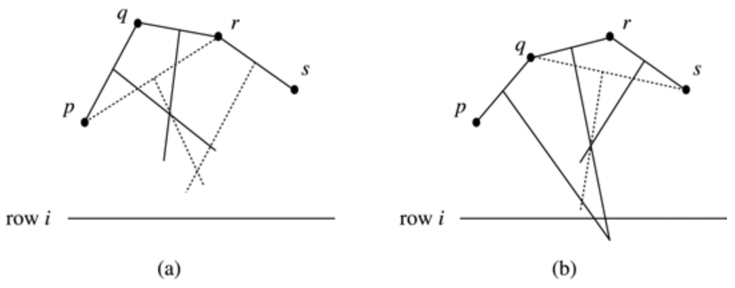


Figure 3. Example of removal of internal features.

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.

After applying both tests the nearest feature of every 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.

Similar technique is applied to compute the nearest feature for each pixel in row i below row i. Then one of the two nearest-features with the smallest distance from a pixel is chosen as the resulting feature.

3. The algorithm

In devising the parallel algorithm, the idea of dimension reduction is used. Note that each row can be processed independently of others, and all nearest-features can be computed in $O(n^2)$ with two sweeps over the image: top-down and bottom-up. Rows can be processed independently and in parallel. The most natural interconnection network architecture for parallelizing the algorithm is a mesh of $n \times n$ processors. In this case, each row of processors works 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 only 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 of 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 the algorithm can be implemented on a linear array of n processors. Let $P_1, P_2, \ldots, P_n$ be the n processors. For the preprocessing step, each pixel (i, j) in row i of the image travels starting from P_i to P_n in a synchronized fashion. This approach

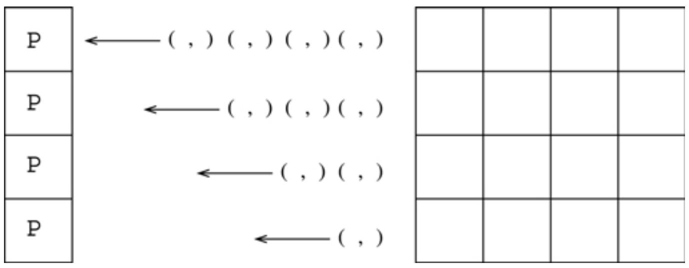


Figure 4. Example of systolic computation in the preprocessing step.

implies a simple systolic computation, in which the rows are fed to the processors one element at a time (see Figure 4). 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.

Next, each of the processors $P_1, P_2, \ldots, P_n$ builds two polygonal chains $C_{i,\ top}$ and $C_{i,\ \mathrm{bottom}}$ for row i, applying the series of extreme and internal feature tests. Each chain corresponds to the top-down and bottom-up scan of the image, and contains information on the feature on and above or on and below the given row. Finally, for each pixel in each row, the nearest between the two features computed on the previous step is selected. Here, once again, each processor works on its own row.

More formally, the algorithm is given as follow:

```
Step 1 (Preprocessing) Apply systolic computation to feed pixels $(i,j)$ to processors
       $P_1, P_2, \ldots, P_n$.
Step 2 (Processing row $i$ by processor $P_i$)
       Step 2.1 Add all nearest black pixels in and above row $i$ (or in and below row $i$
         to the polygonal chain $C_i$, sweeping the row from left to right.
       Step 2.2 (Perform extreme feature test)
         If $|C|_i \leq 1$ then go to Step 3. Let $p$ and $q$ be the two leftmost features in $C_i$.
           While $B(p,q)$ intersects $V_1$ and $q$ is not the rightmost feature in $C_i$,
           remove $p$ from $C_i$, set $p = q$ and $q = right(q)$.
         If $|C|_i \leq 1$ then go to Step 3. Perform the same test for the rightmost
         feature.
       Step 2.3 (Perform internal feature test)
         If $|C|_i \leq 2$ then go to Step 3. Otherwise, let $p$, $q$ and $r$ be the three leftmost
           features in $C_i$, and repeat the following until all features in $C_i$ have been
           processed:
         If $B(p,q)$ and $B(q,r)$ intersect below row $i$ then
           (Advance) Set $p = q, q = r, r = right(r)$.
         else
           (Backtrack) Set $q = p, p = left(p), right(q) = r, left(r) = q$.
Step 3 (Assign features to pixels) For each pixel $(i,j)$ in row $i$, the nearest between
       the two features from the two chains $C_i$ (above and below row $i$) computed
       during the Step 2 is selected. The processing for each row is done by $P_n$.
```

The time required for the preprocessing step using pipelining is $O(n)$. Each row requires $O(n)$ processing time on each processor. 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 between the two feature computed for each pixel in a row is selected once, thus the time required for the third algorithms step is $O(n)$. This results in an optimal $O(n)$ time algorithm.

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 $O(n)$ time, which is linear in the input size, on a linear array of n processors.*

4. The algorithm correctness

In this section, we prove the correctness of the algorithm. The proof is provided for building the polygonal chain containing all nearest features. The prove is given for building $C_{i,\text{ top}}$ polygonal chain. The proof for $C_{i,\text{ bottom}}$ polygonal chain is identical.

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

Proof: 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 $B(p, q)$ containing p or the half plane defined by bisector $B(q, r)$ containing r. That is, x is either closer to p or r than to q. □

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

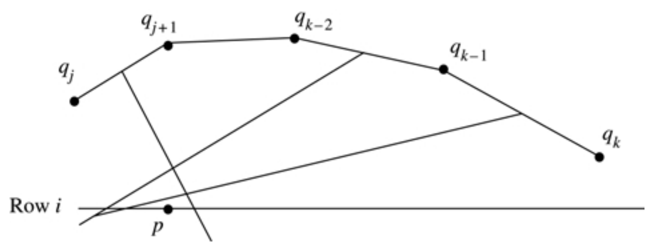


Figure 5. Example of bisectors intersecting to the left of p.

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 5).

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. 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 $j > 1$.

5. Conclusion

We have presented the parallel algorithm for the computation of the nearest feature transform and the distance transform. The algorithm is a time optimal algorithm that uses an array of n processors. In the case when 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. The proof of the correctness is provided. The algorithm is 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. P. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.
4. A. Fujiwara, M. Inoue, T. Masuzawa, and H. Fujiwara. A simple parallel algorithm for the medial axis transform of binary images. In *Proceedings of the IEEE 2nd International Conference on Algorithms and Architecture for Parallel Processing*, 1–8, 1996.
5. 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.

6. T. Hirata. A unified linear-time algorithm for computing distance maps. *Information Processing Letters*, 58:129–133,1996.
7. Y.-H. Lee and S.-J. Horng. Fast parallel chessboard distance transform algorithms. In *Proceedings of the 1996 International Conference Parallel and Distribution Systems*, 488–493, 1992.
8. I. Ragnemalm. Neighborhoods for distance transformation using ordered propogation. *Computer Vision, Graphics and Image Processing*, 56:399–409, 1992.
9. A. Rosenfeld and J. L. Pfalz. Sequential operations in digital picture processing. *Journal of the ACM*, 13:471–494, 1966.
10. O. Schwarzkopf. Parallel computation of distance transform. *Algorithmica*, 6:685–697, 1991.
11. H. Yamada. Complete Euclidean distance transformation by parallel operation. *Proceedings of the 7th International Conference on Pattern Recognition*, 69–71, 1984.