1990

Parallel Heuristics for Determining Steiner Trees in Images

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Lynn TeWinkel

Report Number:
90-1033

PARALLEL HEURISTICS FOR DETERMINING STEINER TREES IN IMAGES

Susanne Hambrusch
Lynn TeWinkel

CSD-TR-1033
October 1990
(Revised May 1991)

Parallel Heuristics for Determining Steiner Trees in Images

*Susanne Hambrusch† and Lynn TeWinkel**

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA.

ABSTRACT

In this paper we consider the problem of determining a minimum-cost rectilinear Steiner tree when the input is an $n \times n$ binary image I which is stored in an $n \times n$ mesh of processors. We present several heuristic mesh algorithms for this NP-hard problem. A major design criterion of our parallel algorithms is to avoid sorting and routing which are expensive operations in practice. All of our algorithms have a $O(n\log k)$ worst-case running time, where k is the number of connected components formed by the entries of value '1'. The main contribution of the paper are two conceptually different methods for connecting components in an image and a method for improving subsolutions by making horizontal and vertical shortcuts.

† Part of the research was done while visiting the International Computer Science Institute, Berkeley, California. Work was supported by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, and by the National Science Foundation under Grant MIP-87-15652.

* This work was supported by the Office of Naval Research under Contract N00014-86-K-0689.

A preliminary version of this paper appeared in the *Proceedings of the 10th International Conference on Pattern Recognition*, June 1990.

1. Introduction

The problem of determining a minimum-cost rectilinear Steiner tree is a fundamental problem in the area of graph algorithms with applications in numerous areas. Since the problem is known to be NP-hard [GJ], many general and problem-specific heuristic approaches have been developed [Be, CSW, Ha, Hw, HVW1, HVW2, LBH, Ri]. In this paper we consider the problem of determining a rectilinear Steiner tree when the input is an $n \times n$ binary image I in which a value of '1' represents a point. The objective is to connect all points by rectilinear segments (i.e., segments that are either horizontal or vertical). We present parallel algorithms for the Steiner tree problem when the $n \times n$ binary image I is stored in an $n \times n$ mesh of processors with one pixel per processor. Our algorithms have an $O(n\log k)$ worst-case time complexity, where k is the number of connected components formed by the entries of value '1', $k \leq n^2$. A major design criterion of our algorithms is to avoid sorting and routing. In practice, both operations are expensive [CSS, RM].

The heart of our algorithms are two conceptually different methods for connecting components. A single application of either method runs-in $O(n)$ time on an $n \times n$ mesh and it does not guarantee that all components are connected with each other. Our algorithms consist of $O(\log k)$ iterations, with each iteration using one of the two methods. Our algorithms are simple and have a small associated constant and, as already stated, do not use sorting or routing operations. They perform connected component computations to achieve global communication and all other steps consist of executing simple operations within a row or column. Note that the connected components can be determined in $O(n)$ time without sorting or routing [CSS, HT].

Another approach to our problem would be to generate from image I a description of the points by their coordinates and to design an algorithm using this description of the input. There are a number of reasons why this approach is inferior to the one of working directly with the image. First, such a conversion makes information that is readily available in the image expensive to retrieve. Furthermore, an algorithm working with the points given by their coordinates is likely to require sorting and routing opera-

tions, as done in problems of a similar nature [MS2, MS3]. Note that an $n \times n$ mesh does not allow for the points of the image to be represented by a graph in the form of an adjacency matrix (since this would require $O(n^4)$ bits). The graph would have to be represented in the form of edge lists, which generally results in complex and routing-dependent algorithms [AH, H, St]. Only few problems on images allow a space-efficient graph-like representation of relevant data and subsolutions [LAN, MS1, MS3].

Throughout we use the following notation. We refer to the points in image I as pixels and to the pixels of value '1' as 1-pixels. Position (0, 0) of the image or the mesh refers to the top leftmost position. Let I^* be the image representing a solution to the Steiner tree problem. Image I^* consists of one connected component which contains I (i.e., if $I(r, c) = 1$, then $I^*(r, c) = 1$) and the number of 1-pixels in I^* should be minimized. The form of connectivity we consider is that of 4-connectivity (i.e., two 1-pixels x and y are in the same connected component if and only if there exists a sequence of 1-pixels from x to y such that two consecutive 1-pixels are horizontally or vertically adjacent). Let the image consisting of the 1-pixels that are in I^*, but not in I, be $I^* - I$. A 1-pixel s, $s \in I^* - I$, is a *Steiner pixel* if s is adjacent to at least three 1-pixels in $I^* - I$. We say that image I^* is *cycle-free* if for any 1-pixel p in $I^* - I$ there exist two other 1-pixels x and y in I^* such that the removal of p disconnects x and y in I^*. Obviously, any solution minimizing the number of 1-pixels in I^* is cycle-free.

Our first method for connecting components is the *Min-Component-Selection (MCS) Method* in which each component selects another component currently at minimum distance. This step resembles a technique used in many parallel algorithms for graph problems [HCS, NS, SV, Ul]. It is well-known that the graph induced by the edges from a component to its selected component contains no cycle. However, the image generated by the MCS method does not necessarily satisfy the corresponding cycle-free property. An interesting part of the MCS method is the detection and removal of 1-pixels belonging to cycles. We also present a number of optimizations that can be applied to the image generated by the MCS method. The optimizations are based on the idea of determining shortcuts. Our algorithm for determining an optimum set of shortcuts requires only simple data movements. Its overall concept and its correctness

are based on a number of non-trivial properties.

The MCS method does not try to position Steiner pixels explicitly. Steiner pixels are created because segments happen to overlap. Our second method for connecting components, the *Steiner-Pixel-Selection (SPS) Method* is based on trying to identify "good" Steiner pixels. In the SPS method a component may not connect to the component currently at minimum distance, but will attempt to connect to a selected Steiner pixel (without exceeding a precomputed maximum distance). Figure 1.1(c) and (d) show an example of how a single application of either method connects the components given in 1.1(a).

As already stated, our algorithms consist of a number of iterations with each iteration applying one of the two methods to the current image. As generally done, we compare the quality of the solutions to the cost of a rectilinear minimum spanning tree [Be, HVW1, HVW2]. Hwang has shown that the cost of a minimum rectilinear Steiner tree is at least $2/3$ of the cost of a minimum spanning tree [Hw]. Our algorithms have been implemented by simulation. Their C-code has been written so that it can easily be translated into MPP Pascal and be put onto the MPP, a 128×128 mesh of processors [Ba, NASA]. For the images considered, the solutions generated by our algorithms are approximately 91% of the cost of a minimum spanning tree which is considered a good performance.

The paper is organized as follows. In Section 2 we discuss the MCS method and the optimizations based on performing shortcuts. Section 3 presents the SPS method for connecting components. Section 4 compares the two methods and describes the performance of our algorithms.

2. Min-Component-Selection Method

Assume image I consists of k connected components $C_1, \cdots, C_k$, $k \geq 2$. The MCS method determines connections between components in three phases. In the first phase, the minimum component selection phase, each component connects to another component at minimum distance. Let CI be the image generated by this first phase. Image CI is not necessarily cycle-free and the second phase, the cycle-removing phase,

eliminates cycles. The third phase is an optimization phase which applies a number of "shortcuts" to reduce the number of 1-pixels further.

Section 2 is arranged as follows. Section 2.1 discusses how components are connected and how cycles are removed in the MCS method. Section 2.1.1 discusses how components at minimum distance are chosen. Segment interaction is dealt with in Section 2.1.2. Section 2.1.3 describes the cycle-removing phase of the MCS method. In Section 2.1.4 details of the implementation of the first two phases of the MCS method are given. Finally section 2.2 describes optimizations that can be applied to the image after the first two phases of the MCS method.

2.1. Connecting Components and Removing Cycles

In the first phase every component C_i determines a component $min(C_i)$ at minimum distance from it. It also determines a sequence of pixels consisting of at most one vertical portion followed by at most one horizontal portion leading from a 1-pixel p_i in C_i to a 1-pixel q_i in $min(C_i)$. Let $S(p_i, q_i)$, or S_i, for short, be this sequence which we call the *segment* from p_i to q_i. A new image CI is created which contains as 1-pixels the 1-pixels in I and the pixels of the segments. Let $G = (V, E)$ be the undirected graph with $V = \{C_1, \cdots, C_k\}$ and $E = \{(C_i, min(C_i)) \mid 1 \leq i \leq k\}$. It is easy to show that G does not contain a cycle and many parallel graph algorithms make use of this property [HCS, NS, SV, UI]. However, image CI does not necessarily satisfy the corresponding cycle-free property. The cycle-removing phase eliminates cycles and changes image CI into image I^*. If image I contains no two components C_i and C_j such that the distance from C_i to C_j is one (i.e.; changing a single 0-pixel into a 1-pixel connects the two components), then the cycle-removing phase guarantees that image I^* satisfies the cycle-free property. As will be discussed later on, we do not detect cycles caused by components that can be connected by a single pixel.

The cycle-free property can be violated in CI in one of two ways. If the image $CI - I$ contains a $c \times c$ block consisting of 1-pixels and the removal of any such 1-pixel does not disconnect image CI, then a cycle is caused by a *thickness* of c, $c \geq 2$. We note that it is possible to connect four components in a cycle-free way such that the

connecting segments contain a 2×2 block of 1-pixels. However, the MCS method forms segments in a way so that every 2×2 block of 1-pixels in image $CI - I$ implies a cycle. Cycles created by a thickness of two or more can easily be detected locally and we refer to them as "local" cycles. If $CI - I$ does not contain a cycle caused by a thickness of c, $c \geq 2$, a cycle can be formed by a sequence of 1-pixels starting at some 1-pixel p in $CI - I$, traversing components and segments, and returning to p. Detecting such a "global" cycle may require global actions. Figure 1.1(b) shows the image of 1.1(a) after the first phase of the MCS method. The segments between components C_3, C_7, and C_9 form, for example, a cycle created by a thickness of 2. The segments between components C_1, C_2, C_4, C_5, and C_6 form a global cycle.

When image I contains components at distance 1, a segment can connect to more than 2 components and this can create a cycle. Figure 2.1 shows an example of such a situation. The image shown represents CI right before the cycle-removing phase. For example, segment S_{10} connects components C_{10}, C_5, and C_6, even though C_{10} intends only to connect to C_5. Throughout the description of the cycle-removing phase we assume that image I contains no two components at distance 1 and thus our claims about image I^* being cycle-free only hold for such an image I. However, the cycles induced by components at distance 1 have, in general, a minimal effect on the total number of 1-pixels used.

2.1.1. Determining Components at Minimum Distance

We now give the precise rules on how the segments which connect the components are determined. Let $row(x)$ (resp. $col(x)$) be the row (resp. col) of pixel x. For two 1-pixels x and y, let $dist(x, y)$ be the minimum number of pixels needed to get from x to y. More precisely, if x (resp. y) is in row $row(x)$ (resp. $row(y)$) and in column $col(x)$ (resp. $col(y)$), then $dist(x, y) = |row(x) - row(y)| + |col(x) - col(y)| - 1$ for $x \neq y$ and 0 for $x = y$. Every component C_i chooses a component $min(C_i)$ at minimum distance. If there is more than one component at minimum distance, then ties are broken in favor of the component with the smallest label. If additional ties need to be broken, then the indices of the endpoints of the segments are used.

Formally, $min(C_i)$, p_i and q_i are chosen so that $p_i \in C_i$, $q_i \in min(C_i)$ and for any 1-pixels p_i' and q_i' with $p_i' \in C_i$ and $q_i' \in C_j$, $j \neq i$:

(i) $dist(p_i', q_i') \geq dist(p_i, q_i)$,

(ii) if $dist(p_i', q_i') = dist(p_i, q_i)$, then $C_j \geq min(C_i)$,

(iii) if $dist(p_i', q_i') = dist(p_i, q_i)$ and $C_j = min(C_i)$, then

if $p_i = p_i'$, then $q_i < q_i'$

else if $q_i = q_i'$, then $p_i < p_i'$

else $\min\{p_i, p_i', q_i, q_i'\} \in \{p_i, q_i\}$

It is easy to verify that rule (iii) ensures that if two components choose each other, the segments chosen by the two components have the same endpoints. These component connection rules will be referred to in later sections.

As mentioned previously, the segments formed by the MCS method consist of at most one vertical portion followed by at most one horizontal portion leading from p_i to q_i. We note that these segments are similar in form to the segments created in the minimum rectilinear Steiner tree algorithm in [HVW1, HVW2]. Given a set of points as input, the algorithm given in [HVW1, HVW2] first determines a minimum spanning tree for these points and then for each edge of the minimum spanning tree, an L-shaped layout. An L-shaped layout consists of a segment with a horizontal and vertical portion, similar to the segments formed by the MCS method.

2.1.2. Segment Interaction

In this section we prove a number of properties concerning the interaction between segments. Knowing how segments can interact is crucial in determining what actions need to be taken by the cycle-removing phase as well as in determining the space requirements of the MCS method. Consider image CI generated as described. For any two segments S_i and S_j the following definitions will be used when characterizing their relationship. When two endpoints of the segments coincide (i.e., $q_i = q_j$, $p_i = q_j$, or $p_j = q_i$), we say that S_i and S_j *share endpoints*. When two endpoints are horizontally or vertically adjacent, we say that S_i and S_j have *neighboring endpoints*. We also say that S_i and S_j have neighboring endpoints when two of their endpoints are diagonally

adjacent, belong to the same component, and one of the endpoints is adjacent to a pixel on the other segment.

We say segments S_i and S_j are *sharing* if there exists at least one 1-pixel that is in both S_i and S_j. Within sharing segments we distinguish between crossing and overlapping segments. When the two segments have at most two 1-pixels in common and, if they have two 1-pixels in common, these pixels are not adjacent to each other, the segments are *crossing*. All other forms of sharing segments are called *overlapping*. Finally, we say segments S_i and S_j are *adjacent* if there exists at least one 1-pixel of S_i that is adjacent to some 1-pixel of S_j and the two segments are not sharing.

The following properties characterize sharing and adjacent segments. Property 2.1 characterizes the endpoints of sharing segments. Properties 2.2, 2.3, and 2.4 deal with overlapping segments with shared endpoints and adjacent segments with neighboring endpoints. All four properties deal with segments in image CI.

Property 2.1. Let S_i and S_j be two sharing segments. Then, S_i and S_j share endpoints.

Proof (by contradiction): Assume that S_i and S_j are sharing segments, but are not sharing endpoints. The four endpoints of the segments can come from either three or four components of I. Let f be the first 1-pixel encountered on S_i that is also in S_j when going from p_i to q_i. Consider first the case when the four endpoints come from three components with p_i and q_j being in the same component. We now have $dist(f, p_i) = dist(f, q_j)$ since p_j chose q_j and since component C_i chose p_i as the first endpoint in its segment. However, this implies that rule (iii) is not satisfied for one of C_i or C_j.

Consider the remaining two cases (i.e., all four endpoints come from different components or q_i and q_j are in the same component). Let l be the last 1-pixel encountered on S_i that coincides with a 1-pixel in S_j when going from p_i to q_i. Let,

$\alpha_1 = dist(p_i, f)$,

$\alpha_2 = dist(l, q_i)$,

$\beta_1 = dist(p_j, l)$,

$\beta_2 = dist(f, q_j)$, and

$z = dist(l, f) + 2$.

See also Figure 2.2. Since p_i chose 1-pixel q_i and not q_j we have:

$$z + \alpha_2 \leq \beta_2 + 1$$

Since p_j chose 1-pixel q_j and not q_i we have:

$$z + \beta_2 \leq \alpha_2 + 1$$

Adding these equations gives $z \leq 1$. If $z = 1$, then for p_i to have chosen q_i and for p_j to have chosen q_j, $dist(f, q_i) = dist(f, q_j)$. This implies, however, that either rule (ii) or rule (iii) was not satisfied for one of components C_i or C_j. Thus Property 2.1 follows. □

Sharing segments with shared endpoints can easily be created. For crossing segments this means that the segments can cross only once and one p-endpoint must coincide with the q-endpoint of the other segment.

We now consider the possible relationships between two adjacent segments. When rules (i)-(iii) generate adjacent segments, they generally have neighboring endpoints. However, it is possible to generate adjacent segments with shared endpoints or with four distinct, non-neighboring endpoints. These types of adjacency can create cycles in the image CI and we briefly discuss them. When two adjacent segments share endpoints, the p-endpoint of one segment coincides with the q-endpoint of the other segment. Examples illustrating this type of adjacency are given in Figures 2.6(a) and 2.7(a).

Consider two adjacent segments, S_i and S_j, with non-neighboring endpoints. Let z be the number of 1-pixels in S_i that are adjacent to a 1-pixel in S_j. When the endpoints of S_i and S_j come from four different components, then z can be arbitrarily large. An example of this configuration with $z = 1$ is between segments S_1 (which connects components C_1 and C_2) and S_6 (which connects components C_6 and C_4) in Figure 1.1(b). Assume now that segments S_i and S_j are adjacent with non-neighboring endpoints and the endpoints come from three different components. If q_i and q_j belong to the same component, one can show that $z = 1$ holds. An illustration of this situation is between segments S_i and S_j in Figure 2.5. If p_i and q_j belong to the same component, there is no bound on the amount of adjacency. Such a relation occurs between segments S_{15}

(which connects components C_{15} and C_{18}) and S_{22} (which connects components C_{22} and C_{15}) in Figure 1.1(b). In this example, $z = 3$. For further details on the relation between adjacent segments we refer the reader to [T].

The next three properties characterize overlapping segments with shared endpoints and adjacent segments with neighboring endpoints. The following definitions are used in the remainder of this paper. A segment S_i with $row(p_i) < row(q_i)$ and $col(p_i) \neq col(q_i)$ is called a "type +" segment. A segment S_i with $row(p_i) > row(q_i)$ and $col(p_i) \neq col(q_i)$ is called a "type -" segment. The first two properties follow immediately from the way segments are determined.

Property 2.2. For every segment S_i there can be at most one other segment S_j such that the vertical portion of S_i is overlapping with (resp. adjacent to) S_j. Furthermore, one of the two segments must consist of a vertical portion only.

Property 2.3. Let S_i and S_j be two horizontally adjacent segments with neighboring endpoints. Let $row(q_i) = row(q_j) - 1$. If both segments contain a vertical portion, then segment S_i is a type + segment and segment S_j is a type - segment. If S_i (resp. S_j) has no vertical portion, then S_j is a type - segment (resp. S_i is a type + segment).

Properties 2.2 and 2.3 imply that adjacent segments can create a thickness of at most 3. Such a situation occurs between segments S_8, S_{10}, S_{11}, and S_{12} in Figure 1.1(b). The next property states that a 1-pixel in image CI can belong to at most 3 mutually overlapping segments.

Property 2.4. There can be at most three segments that are mutually overlapping with each other in the horizontal direction.

Proof (by contradiction): We first show that it is not possible to have three segments of the same type mutually overlapping with shared endpoints. Assume that S_i, S_j, and S_k are three mutually overlapping segments. They can either have their three q-endpoints coincide (i.e., $q_i = q_j = q_k = q$) or they can have two q- and one p-endpoint coincide (i.e., $q_i = q_j = p_k = q$). We only consider the case when three q-endpoints coincide.

The other case is proven by a very similar argument.

W.l.o.g. assume that the three segments are type - segments and that $col(q)$ is the rightmost column to contain a pixel belonging to one of the three segments. The three segments can be ordered so that $row(q) \leq row(p_i) < row(p_j) < row(p_k)$ and $col(q) > col(p_k) > col(p_j) > col(p_i)$. If such an ordering cannot be achieved, the three p-endpoints could not have chosen q. Figure 2.3 shows the position of the endpoints of the segments. Let,

$$v_1 = row(p_i) - row(q),$$
$$v_2 = row(p_j) - row(p_i),$$
$$v_3 = row(p_k) - row(p_j),$$
$$h_1 = col(p_j) - col(p_i),$$
$$h_2 = col(p_k) - col(p_j), \text{ and}$$
$$h_3 = col(q) - col(p_k).$$

Since p_i chose q and not p_j, we have

$$v_1 + h_2 + h_3 \leq v_2.$$

Since p_k chose q and not p_j, we have

$$v_2 + v_1 + h_3 \leq h_2.$$

Adding these inequalities gives $v_1 + h_3 \leq 0$, which is not possible (since $v_1 \geq 0$ and $h_3 \geq 1$). Hence, the three segments cannot have their p-endpoints below $row(q)$. Note that we have $v_1 \geq 0$ and thus the claim holds even when one of the segments is a horizontal segment.

We next show that there cannot exist four segments that are mutually overlapping in the horizontal direction such that two of the segments are type + segments and two of the segments are type - segments. Assume that S_i, S_j, S_k, and S_l are four segments that are mutually overlapping in the horizontal direction such that two segments are type + segments and two segments are type - segments. Since at most one p-endpoint can coincide with q, only two situation are possible: $q_i = q_j = q_k = q_l = q$ or one segment, say S_l, has $p_l = q$ and $q_i = q_j = q_l = q = p_l$. We only consider the first case, since the argument for the second case is similar. W.l.o.g assume again that q is the rightmost pixel in the four segments, as shown in Figure 2.4. Let S_i and S_j be the two

type + segments and S_k and S_l be the two type - segments. We can order S_i and S_j such that $row(q) > row(p_i) > row(p_j)$ and $col(q) > col(p_j) > col(p_i)$. Furthermore, we can order S_k and S_l so that $row(q) < row(p_k) < row(p_l)$ and $col(q) > col(p_l) > col(p_k)$. There are three possibilities (six without considering symmetry) how the four segments can relate and one is shown in Figure 2.4. In this one we have $col(p_i) \leq col(p_k) \leq col(p_j) \leq col(p_l)$. Let,

$$v_1 = row(q) - row(p_i),$$

$$v_2 = row(p_i) - row(p_j),$$

$$v_3 = row(p_k) - row(q),$$

$$v_4 = row(p_l) - row(p_k),$$

$$h_1 = col(p_k) - col(p_i),$$

$$h_2 = col(p_j) - col(p_k),$$

$$h_3 = col(p_l) - col(p_j), \text{ and}$$

$$h_4 = col(q) - col(p_l).$$

Since p_i chose q_i and not p_k, we have

$$h_2 + h_3 + h_4 \leq v_3.$$

Since p_l chose q_l and not p_k, we have

$$v_3 + h_4 \leq h_3 + h_2.$$

Adding these inequalities gives $h_4 \leq 0$ which is not possible (since $h_4 \geq 1$). Note that $v_1 \geq 0$ and thus the claim holds also when one of the segments is a horizontal segment. The other five situations are handled in an analogous manner. Hence, there cannot be four horizontally overlapping segments and Property 2.4 follows. □

Let S_i and S_j be two segments in image *CI*. Then, the relationship between S_i and S_j can be characterized as one of the following. We will refer to these cases by description and by number in the following sections.

(1) S_i and S_j are disjoint

(2) S_i and S_j overlap with shared endpoints

(3) S_i and S_j share endpoints with $p_i = q_j$ and $p_j = q_i$.

(4) S_i and S_j cross with $p_i = q_j$

(5) S_i and S_j are adjacent with neighboring endpoints

(6) S_i and S_j are adjacent with $p_i = q_j$

(7) S_i and S_j are adjacent with non-neighboring endpoints and

(i) the endpoints come from four components

(ii) q_i and q_j are in the same component

(iii) p_i and q_j are in the same component

2.1.3. Removing Cycles

This section describes the cycle-removing phase which generates image I^* from image CI. Recall that the cycle-free property can be violated in CI in two ways. If the image $CI - I$ contains a thickness of c, $c \geq 2$, then image CI contains a local cycle. If the image $CI - I$ contains a 1-pixel p such that there exists a path that starts at p and traverses components and segments and eventually returns to p, then image CI contains a global cycle. Detecting global cycles cannot be done by a simple scanning method. Therefore, if the relation between two segments is such that it could create a global cycle, we take appropriate actions that will destroy the cycle if it should exist. We remove cycles in CI in one of two ways. In many situations we change two interacting segments into overlapping segments with shared endpoints. If doing so does not ensure a cycle-free image, then we remove one of the two segments.

It is clear that case (1) cannot cause any cycles. Case (2) will be shown to cause no cycles in the theorem proven below. Case (3) covers the situation in which pixels p_i and q_j are in one component (namely C_i) and p_j and q_i are in another component (namely C_j). Segments S_i and S_j could both be vertical, both be horizontal, or could both consist of one vertical and one horizontal portion in which case the two segments are not identical. Segments S_3 and S_7 in Figure 1.1(b) illustrate case (3) when the segments are not identical. Obviously, we only want one of the two segments to be in image I^* and therefore we remove either segment S_i or segment S_j.

In case (4), let s be the 1-pixel belonging both to S_i and S_j. There exist two paths from s to $p_i = q_j$ (see segments S_{21} (which connects C_{21} and C_{16}) and S_{16} (which connects C_{16} and C_{17}) in Figure 1.1(b)). We remove one of the two paths by changing

(p_j, q_j) to (p_j, q_i); i.e., $min(C_j)$ is changed from C_i to $min(C_i)$, resulting in overlapping segments with shared endpoints. This change does alter the underlying graph G. However, since the edges (C_j, C_i) and $(C_i, min(C_i))$ were not on a cycle in G, the edges $(C_j, min(C_i))$ and $(C_i, min(C_i))$ cannot be on a cycle in the new graph either.

As for case (5), there can be at most three segments $S_{i,1}$, $S_{i,2}$, and $S_{i,3}$, such that $S_{i,j}$ and $S_{i,j+1}$ are adjacent with neighboring endpoints and a thickness of 3 is created, $1 \leq j \leq 2$. This follows from Properties 2.2 and 2.3. The cycle-removing step changes adjacent segments into overlapping segments with shared endpoints.

Case (6) covers the situation where S_i and S_j are adjacent with $p_i = q_j$. The general approach for eliminating the cycle generated by the adjacency is to change the adjacent segments into overlapping segments with shared endpoints.

Case (7) (i) covers the situation when S_i and S_j are adjacent with non-neighboring endpoints and the endpoints come from four components. The adjacency between the two segments could create a global cycle. In order to avoid this, the cycle-removing phase deletes one of the segments. We do not determine whether the adjacency between the two segments does actually introduce a global cycle, since this could not be done efficiently by local methods. Assume now that segments S_i and S_j are related as described in case 7 (ii) or (iii). An example of case (7) (ii) occurs between segments S_i and S_j in Figure 2.5 and an example of case (7) (iii) occurs between segments S_{15} (which connects C_{15} and C_{18}) and S_{22} (which connects C_{22} and C_{15}) in Figure 1.1(b). Observe that, independent of the adjacency between the two segments, components C_i, C_j, and $min(C_i)$ are in same component in image CI. The general approach for eliminating the cycle generated by the adjacency is to change the q-endpoint of one segment to the q-endpoint of the other segment, resulting in overlapping segments with shared endpoints. If changing endpoints does not eliminate the cycle, one of the two segments is removed. This can happen, for example, when the distance between two of the endpoints is 2, as shown in Figure 2.5.

Assume that image I contains no components at distance 1. Then after the cycle-removing phase has been completed, we have a new image I^* which contains only dis-

joint segments or overlapping segments with shared endpoints. Any two overlapping segments create a Steiner pixel.

Theorem 2.1. Image I^* satisfies the cycle-free property.

Proof (by contradiction): Assume there exist two 1-pixels x and y with $x \in S_i$, and $y \in S_j$ such that there exists a cycle containing x and y. Let Cy be this cycle and let Cy be represented by listing the segments and components traversed. If Cy contains alternating segments and components, then there also exists a corresponding cycle in graph G. Since we know that G is a forest, image I^* cannot contain any such cycles. If the listing of the cycle Cy contains two consecutive segments S_i and S_j, then segments S_i and S_j overlap with shared endpoints. Thus we can replace the sequence S_i, S_j in Cy by S_i, C_γ, S_j, where C_γ is the component containing the shared endpoint. This implies that from Cy we can generate a cycle Cy' such that in Cy' segments and components alternate. Since we know that such a Cy' cannot exist, image I^* cannot contain any such cycles. □

2.1.4. Implementation of the MCS Method

We now discuss some of the details on how to implement the first two phases of the MCS method in $O(n)$ time on an $n \times n$ mesh. Let a *contour pixel* of a component be a 1-pixel adjacent to a 0-pixel. A *left* (*right*, *upper*, *lower*) *contour pixel* of a component is a contour pixel with the 0-pixel to the left (right, top, bottom) of it. A *corner pixel* is a pixel that is part of the vertical and horizontal portion of a segment.

The *min*–components and the segments are determined in the first phase by scanning procedures followed by a connected component computation. The objective of the scanning procedure is to determine, for every contour pixel p of component C_i, a contour pixel q in another component that is at minimum distance (ties are broken as described in rules (i)-(iii)). The scanning starts by every left (resp. right) contour pixel initiating a horizontal scan to the left (resp. to the right). Assume processor u contains contour pixel p of component C_i. Every processor v visited by the left (resp. right) scan initiated at u records that there exists a 1-pixel p belonging to component C_i in the

same row and it also records the distance from u to v. A scan terminates when another 1-pixel or the border of the mesh is reached. After all horizontal scans have been completed, every upper (resp. lower) contour pixel p initiates a forward scan upward (resp. downward). The forward scan determines for every processor v visited the 1-pixel at minimum distance from p and in the same row as processor v (by using the information deposited in v by the previous step). When such a forward scan encounters a 1-pixel or the border of the mesh, it backtracks and selects the overall minimum for 1-pixel p. Pixel p can now determine its pixel q at minimum distance in $O(1)$ time.

After every contour pixel p has determined a contour pixel q at minimum distance, we perform a connected component computation in which each component C_i determines $min(C_i)$, and pixels p_i and q_i. As part of this computation each 1-pixel of C_i is informed of $min(C_i)$, p_i, and q_i. The $O(n)$ connected component algorithms described in [CSS, HT] can easily be modified to accomplish these operations. The final step of the first phase creates image CI. Every 1-pixel p_i of component C_i changes the 0-pixels on the path from p_i to q_i to 1-pixels. We assume that a 1-pixel records the endpoint and component information of each segment it belongs to, resulting in 4 registers needed per segment. This information is used by the cycle-removing phase to determine what actions need to be taken. Since the optimization phase also needs the endpoint and associated component information, it is also necessary to keep this information after the cycle-removing phase. Property 2.4 states that a 1-pixel in image CI belongs to at most three segments. Overall, it is easy to see that the number of registers needed to store all the necessary information about segments is bounded by a constant. We refer to [T] for a more complete discussion on the space requirements of our algorithms.

We next describe the implementation of the cycle-removing phase which involves handling cases (3) - (7). In the implementation, the cases are handled in order with the exception that case (6) is processed between cases (4) and (5). Case (6) is processed out of order to allow groups of adjacent segments to be combined in a way such that outer adjacencies are handled first and inner adjacencies are handled last.

Case (3) handles two components C_i and C_j, with $min(C_i) = C_j$, $min(C_j) = C_i$, $p_i = q_j$, and $p_j = q_i$. This situation can be detected on a local basis by each 1-pixel checking whether the relevant conditions are satisfied. If they are and $C_i < C_j$, then segment S_i is deleted, otherwise S_j is deleted. To delete a segment the p-endpoint starts a scan which deletes the entries about the corresponding segment. If a processor contains no segment entry after such a deletion, its 1-pixel is changed to a 0-pixel.

In case (4) we handle the situation where two segments S_i and S_j cross and $p_i = q_j$. This situation can also be detected on a local basis. Let y be the 1-pixel in both S_i and S_j. To change $S(p_j, q_j)$ to $S(p_j, q_i)$ the pixel representing p_i and q_j initiates a scan which deletes the entries of $S(p_j, q_j)$ between q_j and y. Next the horizontal portion of $S(p_i, q_i)$ between y and q_i is updated to record the additional segment entries for $S(p_j, q_i)$ and all processors containing a pixel of $S(p_j, q_i)$ record the new q-endpoint and corresponding component for $S(p_j, q_i)$.

The next step of the cycle-removing phase handles case (5). Adjacency of segments with neighboring endpoints can be detected locally by considering the endpoints of segments. If the adjacency occurs along the vertical direction, only two segments can be involved (as stated in Property 2.2). Let S_i be the segment that consists of only a vertical portion and let S_j be the segment that has $row(p_j) \neq row(q_j)$. In this situation, q_i and p_j are the neighboring endpoints. We keep segment S_i and delete the vertical portion of S_j, changing $S(p_j, q_j)$ to $S(q_i, q_j)$. If S_j was involved in a case (6) vertical adjacency with a segment S_k, then segment entries of S_k were added to processors containing segment entries from the vertical portion of S_j. These segment entries of S_k are deleted as well and $S(p_k, q_k)$ is changed to $S(q_i, q_k)$.

Assume now that adjacency occurs along the horizontal direction. From Property 2.3 we know that the thickness is either 2 or 3. Let S_i be the segment with the longest horizontal portion involved in creating the thickness. If the thickness is 3, this segment corresponds to the segment whose p-endpoint and q-endpoint are located on the same row. Segment S_i remains and the horizontal portions of the other segments are deleted. Observe that the deletion process may involve segments that are overlapping with other

segments. Let x be a 1-pixel of a segment S_j so that x is vertically adjacent to a 1-pixel of segment S_i. If x is not located at position $(row(q_j), col(p_j))$ or located at position $(row(q_k), col(p_k))$ of a segment S_k which overlaps with S_j, then it and the corresponding entries are deleted. If x is not such a pixel, then it remains a 1-pixel and it initiates an updating of the information recorded about the associated segment. The q-endpoint of any segment that is changed in this process becomes the endpoint of S_i that was one of the original neighboring endpoints.

Case (6) handles the situation where S_i and S_j are adjacent with $p_i = q_j$. This situation can be detected on a local basis by the 1-pixel representing p_i and q_j. In the case of a horizontal adjacency with $p_i = q_j$, S_i is a horizontal segment. See also Figure 2.6(a). We keep segment (p_i, q_i) as well as the vertical portion of segment (p_j, q_j). We delete the horizontal portion of (p_j, q_j), keeping the q-endpoint of segment S_j at 1-pixel $p_i = q_j$. Figure 2.6(b) shows the resulting change in the image. If segment S_j consists of only a vertical portion, then these actions create a 2×2 block of 1-pixels. We remove the 1-pixel at position $(row(q_i), col(p_i))$ to break this cycle.

In the case of a vertical adjacency with $p_i = q_j$, S_j is a vertical segment. See Figure 2.7(a). We keep the segment (p_j, q_j) as well as the horizontal portion of segment (p_i, q_i). We delete the vertical portion of segment (p_i, q_i), keeping the p-endpoint of segment S_i at 1-pixel $p_i = q_j$. Figure 2.7(b) shows the resulting change in the image. If segment S_i consists of only a horizontal portion, then these actions create a 2×2 block of 1-pixels. We remove the 1-pixel at position $(row(q_j), col(p_j))$ to break this cycle.

When case (7) occurs, at least one of the two adjacent segments has $row(p) \neq row(q)$. W.l.o.g let it be segment S_i. The 1-pixel at position $(row(q_i), col(p_i))$ is adjacent to a pixel of segment S_j and it is the job of this 1-pixel to detect case (7) adjacencies. Case (7) (i) covers the situation where the endpoints of segments S_i and S_j come from four different components. In this case we delete the segment S_j if $min(C_i) < min(C_j)$, and we delete S_i otherwise.

The general solution for Case (7) (ii) (i.e., q_i and q_j are in the same component),

is to change the q-endpoint of one segment to the q-endpoint of the other segment. If changing segment $S(p_i, q_i)$ to $S(p_i, q_j)$ causes at least one 1-pixel to change into a 0-pixel, we perform this change. Observe that in this case the cycle is eliminated by changing adjacent segments into overlapping ones. Otherwise, we consider changing $S(p_j, q_j)$ to $S(p_j, q_i)$. If this change in endpoints does reduce the number of 1-pixels, we perform the change in endpoints. It is possible that none of the two possible changes in endpoints reduce the number of 1-pixels. Such a situation is shown in Figure 2.5. In such a case the cycle induced by the two adjacent segments cannot be eliminated by changing endpoints and we remove one of the two segments. The situation for Case (7) (iii) (i.e., p_i and q_j are in the same component) is similar. If changing $S(p_j, q_j)$ to $S(p_j, q_i)$ does not reduce the number of 1-pixels, we delete one of the two segments.

A few additional actions need to be taken during the processing of case (7) adjacencies to ensure that image I^* is cycle-free. Assume that segment S_i is involved in a case (7) configuration with a segment S_j and S_j is deleted. If there exists a third segment S_k, such that S_k shares 1-pixels with S_j and S_k is also adjacent to S_i, then segment S_k is deleted as well. Now assume that segment S_i is involved in a case (7) configuration with a segment S_j and that q_j is changed to q_i. If there exists a third segment S_k, such that S_k shares 1-pixels with S_j and S_k is also adjacent to S_i, then the general solution is to change q_k to q_i as well. If changing q_k to q_i would cause a 1-pixel of S_k to be adjacent to q_j, then segment S_k is deleted instead. Assume now that S_i is involved in a case (7) adjacency with two segments, S_j and S_k, such that S_j and S_k are non-overlapping segments and $p_k = q_j$. Figure 2.8 illustrates this configuration. In this situation we delete segment S_i to prevent a local cycle that could occur in this configuration when q-endpoints are switched during the processing of cases (7) (ii) and (iii).

After the cycle-removing phase has been completed, a 1-pixel in image I^* can belong to at most five different segments [T]. This is also the maximum number of segments a 1-pixel can belong to any time during the first two phases of the MCS method.

As already stated, when image I contains components with $dist(p, q) = 1$, the image generated by the cycle-removing phase is not necessarily cycle-free. Turning image CI into a cycle-free image in these situations can no longer be done efficiently by local scanning methods. Since distances of length 1 do not occur often, the MCS method does not detect cycles caused by components that are distance 1 from other components. The additional effort required to remove single pixels does not seem to be worth the improvement obtained. After image I^* has been generated by the cycle-removing phase, we determine its connected components. Every step of the first two phases of the MCS method is thus either an $O(n)$ time connected component computation or a scanning operation which partially scans a constant number of rows or columns.

2.2. Optimizations

The cycle-removing phase reduces the number of 1-pixels in image I^* by turning adjacent segments into overlapping ones and by eliminating cycles. In this section we describe optimizations that can be applied to image I^* to further reduce the number of 1-pixels. After the cycle-removing phase image I^* may contain segments for which a 1-pixel on the vertical (resp. horizontal) portion is close to another segment so that the number of 1-pixels would be reduced by making a horizontal (resp. vertical) "shortcut". Shortcutting creates overlapping segments from segments that are "not too far apart". We present two techniques for shortcutting and an algorithm for selecting an optimum set of shortcuts. The first technique we describe is a shortcutting technique applied to non-overlapping segments in I^*. (We call a segment non-overlapping if it does not overlap with any other segment.) The second technique we describe is a shortcutting technique applied to overlapping segments in I^*.

Let $C_1, C_2, \cdots, C_k$ be the components of image I and let $C'_1, C'_2, \cdots, C'_l$ be the components of image I^*. Our optimizations do not change any pixels in $C_1, C_2, \cdots, C_k$. Let I^*_{op} be the image after the optimizations described in this section have been applied to I^*. As I^*, image I^*_{op} contains l components and two pixels $x \in C_i$ and $y \in C_j$ are in the same component in I^*_{op} if and only if they are in the same

component in I^*. Figure 2.9 shows an example of how these optimizations can change an image I^*.

We start by describing the shortcutting technique applied to non-overlapping segments in I^*. We first describe how each segment proposes a shortcut. Given all the proposed shortcuts, we give a graph formulation for determining which proposed shortcuts to take and then a dynamic programming formulation of the problem. We finally describe how to use the dynamic programming formulation to obtain an $O(n)$ time mesh algorithm.

Let C_i be a component of image I and let S_i be the segment formed by p_i and q_i. Let N_i be the set containing the non-overlapping segments that have their q-endpoint in C_i. Let O_i be the set containing segment S_i plus the segments that have their q-endpoint in C_i and which overlap with at least one other segment. The main idea of shortcutting is to have every segment S_j in N_i propose a horizontal or vertical shortcut to a segment in $N_i \cup O_i$. Formally, segment S_j proposes a shortcut if there exists a 1-pixel b_j on S_j and a 1-pixel e_j on some segment in $N_i \cup O_i$ such that b_j and e_j are either in the same column or in the same row and $dist(p_j, e_j) < dist(p_j, q_j)$. Pixels b_j and e_j can be viewed as the pixels on the begin and on the end of the shortcut proposed by segment S_j, respectively. Observe that we do not allow segment S_j to shortcut to a segment not in $N_i \cup O_i$. If we would allow S_j to shortcut to such a segment, we could create cycles in image I^*_{op} and/or disconnect components of I^*.

Having each non-overlapping segment determine its best shortcut can easily be done in $O(n)$ time by using simple scanning methods. The difficulty lies in determining which shortcuts to make. Obviously, not all proposed shortcuts can be made since the shortcut made by segment S_j counts on using a certain portion of another segment. Furthermore, if we allow segments to propose shortcuts in all four direction simultaneously, the proposed shortcuts can contain cycles; e.g., segment S_{j_1} proposes a shortcut to S_{j_2}, S_{j_2} proposes one to S_{j_3} and S_{j_3} proposes a shortcut to S_{j_1}. Detecting and handling cycles of this nature could no longer be done by simple local scanning methods. We avoid the creation of cycles altogether by separating the directions in which

shortcuts can be proposed. As it turns out, handling the proposed shortcuts in one direction, say horizontal to the right, is already challenging. The algorithm determining which of the proposed shortcuts should be made requires only simple data movement operations and runs in $O(n)$ time with a small associated constant. The interesting and non-trivial part of this algorithm is the way the selection of shortcuts is done and its correctness.

We next describe how to determine the shortcuts when horizontal shortcuts to the right can be made. The algorithms for the other three directions are analogous. As already stated, in $O(n)$ time every segment can determine whether and how much it gains by making a horizontal shortcut to the right. Let $G=(V, E)$ be the directed, weighted graph in which every segment of I^* corresponds to a vertex and an edge $<i, k>$ implies that

(i) segment S_i proposes a shortcut to segment S_k and

(ii) pixel e_i, which is on the vertical portion of S_k, is not necessary for segment S_k when S_k selects to make its proposed shortcut.

The weight w_i of vertex i corresponds to the number of 1-pixels saved when S_i selects the proposed shortcut; i.e., $w_i = dist(p_i, q_i) - dist(p_i, e_i)$. Every vertex of G has out-degree at most 1 and for every edge (i, k) we have $col(p_i) < col(p_k)$. Thus, G is a directed forest. Since every vertex i in G corresponds to segment S_i, we will no longer distinguish between vertex i and the segment S_i.

The problem of determining which of the proposed shortcuts to take can be formulated as a graph problem as follows. We point out that the graph model is only used in our explanation and that the algorithm does the corresponding actions directly on the image. If we require that, whenever segment S_k selects to make its proposed shortcut, no segment S_i with $<i, k> \in E$ is allowed to take its proposed shortcut, then finding a maximum weighted independent set of G gives the optimum selection of shortcuts. A maximum weighted independent set of G is a subset $W \subseteq V$ such that any two vertices in W are not adjacent in G (†) and $w^* = \sum_{i \in W} w_i$ is a maximum. However, it is not true

(†) If u and v are in W, then neither $<u, v>$ nor $<v, u>$ is in E.

that, if in an optimum solution segment S_k takes its shortcut, none of the segments S_i with $<i, k> \in E$ take their shortcut. An example of this is shown in Figure 2.10. In this figure segment S_k saves 6 pixels by making its shortcut, and segment S_{i_1} saves 3 pixels by making a shortcut to S_k. Assume segment S_k makes its shortcut. Even when S_{i_1} pays for the 2 pixels needed to connect to S_k, it still saves one pixel. We will show that, if in an optimum solution segment S_k takes its proposed shortcut, then at most one segment S_i with $<i, k> \in E$ takes its shortcut (and it pays for the extension of the vertical portion of the modified segment S_k).

Let S_k be any segment. We define $D(k)$, the *diamond* of segment S_k, to be the set containing the positions α with $dist(p_k, \alpha) < dist(p_k, q_k)$. Endpoint q_k is at minimum distance and thus, if an element of $D(k)$ corresponds to a 1-pixel already present in I, this 1-pixel belongs to component C_k. Let the border of $D(k)$ be the set containing the positions α with $dist(p_k, \alpha) = dist(p_k, q_k)$. The border of $D(k)$ contains at least one 1-pixel (namely q_k) belonging to another component. Suppose S_k proposes a shortcut and assume w.l.o.g. that $row(p_k) < row(q_k)$. Let $i_1, i_2, \cdots, i_l$ be the vertices with $<i_j, k> \in E$ (i.e., they correspond to segments shortcutting into S_k) with $row(e_{i_1}) < row(e_{i_2}) < \cdots < row(e_{i_l})$. Let $T(i_j)$ be the set of positions in diamond $D(i_j)$ which are in column $col(p_k)$, $1 \le j \le l$.

Lemma 2.1. Let W be an optimum selection of shortcuts containing the shortcut proposed by segment k. If $i_1 \in W$, then $T(i_1)$ contains pixel b_k. Furthermore, none of $i_2, \cdots, i_l$ is in W.

Proof:. If $T(i_1)$ contains pixel b_k, then segment i_1 reduces 1-pixels by making a shortcut to segment k and to "pay" for the $row(p_{i_1}) - row(b_k)$ pixels needed to extend the vertical portion of segment k. If $T(i_1)$ does not contain pixel b_k, segment i_1 does not gain anything by such an extension. Hence, the first part of the lemma follows.

We next show that $T(i_j) \cap T(i_{j+1}) = \emptyset$; i.e., no two diamonds can share pixels in $col(p_k)$. Let

$$d_1 = col(p_{i_{j+1}}) - col(p_{i_j}) - 1,$$

$d_2 = col(p_k) - col(p_{i_{j+1}}) - 1,$

$f_{i_j} = dist(p_{i_j}, q_{i_j}) - d_1 - d_2 - 2,$

$f_{i_{j+1}} = dist(p_{i_{j+1}}, q_{i_{j+1}}) - d_2 - 1,$

$d_3 = row(p_{i_{j+1}}) - row(p_{i_j}) - f_{i_j} - 1,$ and

$d_4 = dist(e_{i_{j+1}}, q_k),$

as shown in Figure 2.10 for $j = 1$. The entry $f_{i_j} + 1$ represents the number of pixels saved by segment i_j when i_j shortcuts into segment k. Since p_k did not choose $p_{i_{j+1}}$, we have $d_2 \geq d_4$. Pixel $p_{i_{j+1}}$ has its q-endpoint no further away than q_k and thus $f_{i_{j+1}} \leq d_4$. Adding these two inequalities gives $f_{i_{j+1}} \leq d_2$. Since p_{i_j} did not choose $p_{i_{j+1}}$, we have $d_3 \geq d_2 + 1$. Hence, $f_{i_{j+1}} < d_3$. In order for $T(i_{j+1})$ to contain an element also in $T(i_j)$ we need $f_{i_{j+1}} > d_3$ and thus $T(i_j) \cap T(i_{j+1}) = \varnothing$ follows.

Hence, the number of pixels needed to extend any vertical portions for segment i_j is at least f_{i_j} for $j > 1$. Thus, for $j \geq 2$, segment i_j does not reduce 1-pixels by making a shortcut in the case when the shortcut proposed by segment k got selected, and the second part of the lemma follows. □

Determining the optimum selection of shortcuts for one tree of forest G can now be modeled as follows. Let $T = (V_T, E_T)$ be a rooted tree in which the relationship between vertices and edges to segments and shortcuts is as defined for G. Every vertex of T is either red or blue. A blue vertex corresponds to a segment that could gain by taking its shortcut even though its parent takes its shortcut. Because of Lemma 2.1, every vertex has at most one blue child and the root of T is red. Let i be a vertex and k be its parent. Vertex i is a blue vertex if and only if $dist(p_i, b_k) < dist(p_i, q_i)$. Recall that b_k is the pixel on the begin of the shortcut proposed by S_k. Let $w_i' = w_i - dist(e_i, b_k) - 1$, where $w_i = dist(p_i, q_i) - dist(p_i, e_i)$, as already defined earlier. A blue vertex i has two weights, w_i and w_i', associated with it. A red vertex i has one weight, namely w_i, associated with it. We are to determine a subset $W = W_R \cup W_B$ of the vertices, where W_R (resp. W_B) are the red (resp. blue) vertices, such that no two vertices in W_R are adjacent and w^* is a maximum. In order to define w^*, let $W_B =$

$W_B' \cup W_B''$, where W_B' are the blue vertices whose parents are not in W, and W_B'' are the blue vertices whose parents are in W, respectively. Then, the value w^* to be maximized is $\sum_{i \in W_R \cup W_B'} w_i + \sum_{i \in W_B''} w_i'$.

We next give a dynamic programming formulation of this problem. For any vertex i, let T_i be the subtree rooted at i. Let $s(i)$ be the maximum weight achievable for T_i when vertex i is to be included in the solution and let $s'(i)$ be the maximum weight achievable for T_i when vertex i is not present in the solution. For any leaf node i of T we have have

$$s(i) = w_i \text{ and}$$
$$s'(i) = 0.$$

For any interior vertex k with children $i_1, i_2, \cdots, i_l$, where i_1 is either a blue vertex or not existing, we have

$$s(k) = \max\{s'(i_1), s(i_1) - (w_{i_1} - w_{i_1}')\} + \sum_{j=2}^{l} s'(i_j) + w_k \text{ and}$$

$$s'(k) = \sum_{j=1}^{l} \max\{s(i_j), s'(i_j)\}.$$

Obviously, $w^* = \max\{s(r), s'(r)\}$, where r is the root of T, and w^* can be determined in $O(|V_T|)$ sequential time. By using the s and s' entries in a traversal of T initiated at the root r, we can determine a set W achieving weight w^* in additional $O(|V_T|)$ steps.

We now describe how to use the dynamic programming formulation to obtain an $O(n)$ time mesh algorithm. The logic of our algorithm is based on the computation of the s and s' entries. Their computation is done while traversing paths of 1-pixels in the image. Let P be a path from a leaf of T to the root of T and let P' be the sequence of pixels in the image corresponding to path P. The next lemma shows that the number of pixels on P' is $O(n)$. P' consists of horizontal movements (i.e., the shortcuts) and vertical movements (i.e., the portions of the segments between the incoming and outgoing shortcut). A corner pixel is considered to belong to the horizontal movement. The horizontal portions of P' contain a total of at most n pixels (since all shortcuts go from left to right). We note that for any given row the number of pixels belonging to vertical portions of P' is not bounded by a constant. If this property were true, the $O(n)$ length

of P' would follow immediately. Our proof of the $O(n)$ length of a path in the image is based on a non-trivial accounting technique.

Lemma 2.2. The number of pixels on path P' is $O(n)$.

Proof: Let $i_1, i_2, \cdots, i_d$ be the segments on path P' with i_1 being the leaf and i_d being the root. The shortcut proposed by segment i_j, $1 \leq j < d$, leads from pixel b_{i_j} on segment i_j to pixel e_{i_j} on segment i_{j+1}. Let $h_{i_j} = dist(b_{i_j}, e_{i_j}) + 2$ and $v_{i_j} = dist(e_{i_{j-1}}, b_{i_j})$. Segment i_j accounts in P' for a vertical movement of length v_{i_j} and a horizontal movement of length h_{i_j}. We first show how to assign to segment i_j at least $v_{i_j}/8$ pixels belonging to a horizontal movement (not necessarily to the horizontal movement done by the shortcut proposed by segment i_j). Furthermore, no two pixels of a horizontal movement get assigned twice.

Let i_j and i_{j+1} be two consecutive segments on path P'. As defined in section 2.1.2, we say i_j and i_{j+1} are of the same type if either $row(p_{i_j}) < row(q_{i_j})$ and $row(p_{i_{j+1}}) < row(q_{i_{j+1}})$ or $row(p_{i_j}) > row(q_{i_j})$ and $row(p_{i_{j+1}}) > row(q_{i_{j+1}})$. Assume the assignments for $i_1, \cdots, i_{j-1}$ have been made without assigning pixels on horizontal movements to the right of $col(p_{i_j})$.

Consider first the situation when segments i_j and i_{j+1} are of the same type. We then have $h_{i_j} \geq v_{i_j}/2$. This holds since $v_{i_j} < dist(p_{i_j}, q_{i_j})$ and $h_{i_j} \geq dist(p_{i_j}, q_{i_j})/2$ (if the latter were not true, $p_{i_{j+1}}$ would choose p_{i_j} instead of $q_{i_{j+1}}$). See Figure 2.11(a). Thus, we can assign to segment i_j at least $v_{i_j}/2$ pixels belonging to the horizontal movement made by the shortcut of i_j.

Assume now that segments i_j and i_{j+1} are of opposite types. W.l.o.g let $row(p_{i_j}) > row(q_{i_j})$ (which implies $row(p_{i_{j+1}}) < row(q_{i_{j+1}})$). If $h_{i_j} \geq v_{i_j}/2$, we again assign $v_{i_j}/2$ pixels from the shortcut of i_j to segment i_j. Otherwise, we distinguish whether i_{j+1} and i_{j+2} have the same type.

Case 1. Segments i_{j+1} and i_{j+2} have the same type. It then follows from above that $h_{i_{j+1}} \geq v_{i_{j+1}}/2$. We assign $v_{i_{j+1}}/4$ pixels of the shortcut of i_{j+1} to each of segment i_j and segment i_{j+1}. It remains to show that $v_{i_{j+1}}/4 > v_{i_j}/8$. This holds since

$h_{i_j} < v_{i_j} / 2$ and $p_{i_{j+1}} = b_{i_{j+1}}$ imply $v_{i_{j+1}} > v_{i_j} / 2$.

Case 2. Segments i_{j+1} and i_{j+2} are of opposite types. It is easy to see that segments i_j and i_{j+1} together make a horizontal movement of at least $dist(b_{i_j}, q_{i_j})$.

Case 2.1. $row(p_{i_{j+1}}) \geq row(p_{i_j}) - dist(p_{i_j}, q_{i_j})$. Informally, this means that $p_{i_{j+1}}$ does not lie above the top corner of the diamond $D(i_j)$. This implies $v_{i_{j+1}} \leq dist(b_{i_j}, q_{i_j})$. We can thus assign a horizontal movement of length $dist(b_{i_j}, q_{i_j}) / 2$ to each of i_j and i_{j+1} and continue with segment i_{j+2}.

Case 2.2. $row(p_{i_{j+1}}) < row(p_{i_j}) - dist(p_{i_j}, q_{i_j})$. We will go through the argument for the situation when $col(q_{i_j}) \leq col(p_{i_j})$. The other situation is handled in a similar way and is omitted. The goal is again to assign a horizontal movement of length $dist(b_{i_j}, q_{i_j}) / 2$ to each of i_j and i_{j+1}. Let δ be the number of pixels on segment i_{j+1} above row $row(p_{i_j}) - dist(p_{i_j}, q_{i_j}) - 1$, and let ρ be the number of pixels on the horizontal portion of segment i_j, as shown in Figure 2.11(b). If $p_{i_{j+1}}$ and q_{i_j} are in different components, then, $h_{i_j} + \rho \geq 2v_{i_j} + \rho - h_{i_j}$. The right-hand side accounts for a vertical movement of at least v_{i_j} and a horizontal movement of at least $v_{i_j} + \rho - h_{i_j}$ within $D(i_j)$ done by segment i_{j+1}. This contradicts our assumption of $h_{i_j} < v_{i_j} / 2$. Hence, $p_{i_{j+1}}$ and q_{i_j} must belong to the same component. In this case we have $\delta < h_{i_j}$ (otherwise q_{i_j} would be the p-endpoint of this component). Thus,

$$v_{i_{j+1}} \leq dist(b_{i_j}, q_{i_j}) + \delta \leq dist(b_{i_j}, q_{i_j}) + h_{i_j} \leq \frac{3}{2} dist(b_{i_j}, q_{i_j}).$$

We can now assign to each of i_j and i_{j+1} a horizontal movement of length $dist(b_{i_j}, q_{i_j}) / 2$. From the above inequality it follows that the condition on the vertical movement made by segment i_{j+1} is satisfied; i.e., $v_{i_{j+1}} / 8 \leq dist(b_{i_j}, q_{i_j}) / 2$.

Therefore, if path P' makes a total of $k = \sum_{j=2}^{d} v_{i_j}$ vertical movements, then P' makes a horizontal movement of at least $k / 8$. Since the horizontal portions of P' contain a total of at most n pixels, the length of path P' is $O(n)$. □

We point out that the objective was to prove the $O(n)$ length and we omitted the $\lfloor$'s and $\lceil$'s from the analysis. We also did not aim for the tightest bound possible since doing so does not affect the performance of the algorithm.

After all the children of vertex i have computed their corresponding values and the values of $s(i)$ and $s'(i)$ are available in the processor containing b_i, this processor sends these two values to b_k via e_i. If segment i corresponds to a blue vertex, we also send the difference between the two weights. If along the path to b_k the entries from another child of vertex k are encountered, entries are combined; i.e., we start building up the entries $s(k)$ and $s'(k)$. It is easy to show that the entries moving along the longest path are never delayed. Since the length of the longest path is $O(n)$, the $O(n)$ time bound for computing the s- and s'-entries follows. The actual selection of proposed shortcuts is then made by running the just completed data movement backwards. The decision made for the parent, together with the s- and s'-entries, is used to make the decision of a child. Hence, the optimum selection of proposed shortcuts to the right can be made in $O(n)$ time by using simple scanning methods.

After the shortcuts to the right have been selected, the other three directions are handled in a similar way. Again, only segments that do not overlap with other segments are allowed to propose a shortcut. Assume the shortcuts are processed in the order of right shortcuts, left shortcuts, up shortcuts, and then down shortcuts. In order to avoid having initial shortcuts that result in only a small savings, it appears reasonable to require that proposed shortcuts save at least a certain minimum number of pixels. One might require that proposed shortcuts save a fixed amount depending on the size of n and the direction of shortcuts currently being considered. For example, for $n = 128$, one might require that proposed shortcuts to the right must save at least 8 pixels, proposed shortcuts to the left must save at least 5 pixels, proposed shortcuts in the up direction must save at least 3 pixels, and proposed shortcuts in the down direction must save at least 1 pixel. Another possibility is to require that proposed shortcuts save an amount which varies with the length of the segment proposing a shortcut and depends on the direction of shortcuts currently being considered. For example, if segment S_i proposes a shortcut, then a proposed shortcut to the right must save at least

$dist(p_i, q_i) / 4$ pixels, a proposed shortcut to the left must save at least $dist(p_i, q_i) / 4$ pixels, a proposed shortcut in the up direction must save at least $dist(p_i, q_i) / 8$ pixels, and a shortcut in the down direction must save at least 1 pixel. If a segment S_i cannot propose a shortcut of sufficient length for the direction currently being considered, then S_i does not propose a shortcut for that direction.

We conclude this section by sketching similar optimizations that can be applied to overlapping segments. Let C_i be a component of image I, $y \in C_i$, and $OV(y)$ be the set containing the overlapping segments in image I^* that have pixel y as their q-endpoints. Recall that in image I^* there can be at most 5 overlapping segments whose q-endpoint is pixel y. Let $S(p_{i_1}, y)$, $\cdots$, $S(p_{i_5}, y)$ be five overlapping segments. Using an idea similar to the shortcutting described earlier, we allow shortcuts between these overlapping segments. The implementation is now much simpler. Consider two such segments, say $S(p_{i_1}, y)$ and $S(p_{i_2}, y)$ which, w.l.o.g., are both + segments. If $col(p_{i_2})$ is between $col(y)$ and $col(p_{i_1})$, then $S(p_{i_1}, y)$ determines if pixels can be saved by making a horizontal shortcut to $S(p_{i_2}, y)$. If yes, S_{i_1} next checks whether it is possible to change enough 1-pixels into 0-pixels so that it gains by making the shortcut. Observe that the optimizations made by non-overlapping segments may have created additional overlaps with segment S_{i_1} and thus S_{i_1} cannot simply erase itself. Pixel p_{i_1} determines in $O(n)$ time whether it should perform the shortcut. The other cases for performing shortcutting between segments in $OV(y)$ are handled in an analogous way.

3. Steiner-Pixel-Selection Method

In this section we describe our second method for connecting components, the Steiner-Pixel-Selection (SPS) method. In the SPS method every component of image I selects a Steiner pixel to which it attempts to connect by either a vertical or a horizontal segment. The image generated by one application of the SPS method is cycle-free. The SPS method consists of four phases. In the first phase every component C_i in image I selects a Steiner pixel s_i. The second phase establishes a vertical or horizontal connection from a contour pixel in component C_i to the selected Steiner pixel s_i. The third phase handles Steiner pixels that are adjacent to only one other 1-pixel. In such a

case further steps are taken to either assign the component another Steiner pixel or to erase the connection altogether. The final phase makes the image cycle-free. Figure 1.1(d) shows the image obtained by applying the SPS method to the image of 1.1(a).

We next describe how the Steiner pixels are determined. Every component C_i first determines the distance between itself and the component at minimum distance from it. Let $min\_dist\,(C_i) = dist\,(p_i, q_i) - 2$, where p_i and q_i are defined as in the MCS method. Let $w_{i1}, w_{i2}, \cdots, w_{il_i}$ be the contour pixels of component C_i. Every contour pixel w_{ij} of component C_i initiates a constant number of scans. The purpose of these scans is to deposit at every processor that can be reached by a vertical or horizontal segment adjacent to w_{ij} a *scan pair* (w_{ij}, C_i). A contour pixel initiates at most twelve such scans. This happens in the case when component C_i consists of a single pixel, as shown in Figure 3.1. A scan terminates when either a scan pair has been deposited in $min\_dist\,(C_i)$ processors, a scan pair has been deposited at a processor adjacent to the border of the mesh, or a scan pair has been deposited at a processor adjacent to a processor containing a pixel of component C_i. Under these rules at most four scan pairs are deposited at any processor. Observe that a scan cannot encounter a 0-pixel that is adjacent to a 1-pixel belonging to another component (since any 1-pixel of another component is at least $min\_dist\,(C_i) + 2$ positions away).

When all scans have been completed, a processor containing a 0-pixel v contains up to four scan pairs and we next remove multiple entries originating from the same component. Let (w_{ij}, C_i) and (w_{rp}, C_r) be two scan pairs. If $C_i = C_r$, then one of the pairs is deleted. If $dist\,(w_{ij}, v) > dist\,(w_{rp}, v)$, pair (w_{ij}, C_i) is deleted. In case of equality, the pair with the larger w-value is deleted. A processor containing fewer than three scan pairs does not represent a Steiner pixel and it deletes all its scan pairs.

Let p be a processor containing 0-pixel v and m scan pairs $(w_{i_u j_u}, C_{i_u})$, $1 \le u \le m$, $m = 3, 4$; $1 \le j_u \le l_{i_u}$. We next compute M_v, the cost of pixel v as a Steiner pixel. The cost is the arithmetic mean of the distances of the contour pixels to v; i.e., $M_v = \frac{1}{m} \sum_{u=1}^{m} dist(v, w_{i_u j_u})$. Every component C_i next selects, among the proces-

sors containing a scan pair originating from a contour pixel of C_i, the 0-pixel associated with the minimum cost entry. This 0-pixel is the selected Steiner pixel for component C_i. If a component C_i selects pixel v as its Steiner pixel (i.e., $v = s_i$), then for all other possible Steiner pixels t the following rules are satisfied:

(i) $M_t \geq M_v$

(ii) if $M_t = M_v$, then $t > v$.

Assume every component C_i selected its Steiner pixel s_i and that s_i is reached by a scan initiated by contour pixel w_{is}, $1 \leq i \leq k$. Let b_i be the pixel adjacent to w_{is} such that b_i and s_i are in the same row or column. Note that, since we allow up to twelve scans from a contour pixel, w_{is} and s_i may not be in the same row or column. Let $S[b_i, s_i)$ be the segment consisting of the sequence of pixels that connects b_i and s_i and in which b_i, but not s_i, is included. Let $S[b_i, s_i]$ be the segment consisting of the sequence of pixels that connects b_i and s_i and in which b_i and s_i is included. See Figure 3.2(d) for illustration.

The second phase tries to establish the connections from the b_i's to the Steiner pixels. It is easy to see that changing the 0-pixels on $S[b_i, s_i]$ to 1-pixels can create cycles and use more pixels than necessary. In order to avoid cycles, component C_i may end up not connecting to pixel s_i, but to a another pixel on $S[b_i, s_i]$. Let e_i be the pixel to which component C_i ends up connecting. The connections and the end pixels are determined as follows. First all vertical connections are made. If b_i and s_i are in the same column, a scan is initiated at b_i. This scan moves towards s_i and changes 0-pixels to 1-pixels. It terminates when it either reaches s_i (in this case we have $e_i = s_i$) or when a 0-pixel changed into a 1-pixel is adjacent to another 1-pixel. In both cases, e_i is the last pixel changed on the scan. Next, every b_i representing the begin of a horizontal segment starts a scan with the same terminating conditions. One way a horizontal scan can now terminate is by "running into" a vertical connection. Figure 3.2 shows examples of how connections are made.

The third phase of the SPS method handles end pixels that are adjacent to only one other 1-pixel. Note that whenever an end pixel is adjacent to only one other pixel

we have $e_i = s_i$. Every horizontal segment $S[b_i, e_i]$ with e_i adjacent to only one other 1-pixel is extended beyond e_i in an attempt to locate a 1-pixel the component can connect to. Such an extended scan terminates when either a 0-pixel adjacent to a 1-pixel is encountered, $min\_dist(C_i)$ processors have been visited (counting from the begin of the scan at b_i), or the border of the mesh is reached. All 0-pixels traversed by the scan are changed to 1-pixels. An exception is made when the last 0-pixel encountered is adjacent to a 1-pixel belonging to component C_i. In this case the last 0-pixel remains a 0-pixel. The last 0-pixel changed to a 1-pixel becomes the new e_i for component C_i. After the horizontal extensions have been made, portions of horizontal and vertical segments are erased according to the following rules. If $S[b_i, e_i]$ is a horizontal segment in which e_i is still adjacent to only one 1-pixel, then the entire segment is erased. Assume now that $S[b_i, e_i]$ is a vertical segment in which e_i is adjacent to only one 1-pixel. If there exists a 1-pixel s on $S[b_i, e_i]$ that is adjacent to the end pixel of a horizontal segment, then pixel s is made the end pixel for C_i (and 1-pixels between e_i and s as well as 1-pixel s are erased). Should there exist more than one such pixel on the segment, we choose the one closest to e_i. If no such end pixel s exists, the entire segment $S[b_i, e_i]$ is erased.

Let CI be the image created by the second and third phase of the SPS method. Image CI may not be cycle-free and the final phase removes cycles. Cycles can only be created in a very local way, namely in the form of blocks of size 2×2. It is clear that the terminating conditions of the scans used in the second phase do not allow the creation of larger blocks. At the same time, the existence of a 2×2 block does not necessarily imply a cycle. The algorithm checks whether a 2×2 block creates a cycle. If it does, one of the 1-pixels in this block is removed. A 1-pixel that can be removed is a 1-pixel adjacent to only two other 1-pixels. Such a 1-pixel is an end pixel e_i for some component C_i involved in the creation of the 2×2 block. After the 1-pixel is removed, C_i's new end pixel is the pixel of $S[b_i, e_i]$ adjacent to e_i. In Figure 3.2(c) pixel e_1 belongs to a 2×2 block and it is removed during the cycle-removing phase.

Let I^* be the image generated by the fourth phase. We next show that image I^* satisfies the cycle-free property. Every component C_i contains at most one contour

pixel that connects to a Steiner pixel and no pixel on $S[b_i, e_i]$ is adjacent to a 1-pixel belonging to a component, with the exception of pixel b_i. Any cycle caused by the segments $S[b_i, e_i]$ consists thus of horizontal and vertical portions that belong entirely to these segments. We first prove a property about horizontal and vertical segments.

Property 3.1. Let $S[b_i, e_i]$ and $S[b_j, e_j]$ be two vertical (resp. horizontal) segments. Then, no pixel in $S[b_i, e_i)$ can be adjacent to a pixel in $S[b_j, e_j]$. In addition, pixel e_i cannot be adjacent to a pixel in $S[b_j, e_j)$.

Proof (by contradiction): Assume that there does exist a pixel in $S[b_i, e_i)$ that is adjacent to a pixel in $S[b_j, e_j]$. Let a_i be the pixel on $S[b_i, e_i)$ closest to b_i (a_i could be identical to b_i) that is adjacent to a pixel in $S[b_j, e_j]$. If $a_i = b_i$, then a pixel of $S[b_j, e_j]$ is distance one from component C_i. However, phase one deposits scan pairs only at processors within distance $min\_dist(C_i) = dist(p_i, q_i) - 2$ of C_i and phase three extends segments to at most length $min\_dist(C_i)$. Therefore, this is not possible. If $a_i \neq b_i$, then pixel a_i is adjacent to either b_j or e_j. Consider first a possible adjacency with pixel e_j. Since in phases two and three all scans terminate at the first occurrence of an adjacent 1-pixel and all 2×2 blocks of 1-pixels causing cycles are eliminated, it is impossible for a_i to be adjacent to such a pixel. Next consider a possible adjacency with pixel b_j. If a pixel a_i of $S[b_i, e_i]$ is adjacent to b_j, then a_i is distance one from component C_j and as stated previously, this is not possible. The same type of reasoning can be used to show that pixel e_i cannot be adjacent to a pixel in $S[b_j, e_j)$ and thus Property 3.1 follows. □

The only adjacency between two vertical (resp. horizontal) segments $S[b_i, e_i]$ and $S[b_j, e_j]$ that can be possible is between pixels e_i and e_j.

Theorem 3.1. Image I^* satisfies the cycle-free property.

Proof (by contradiction): Assume that image I^* does not satisfy the cycle-free property. As stated previously, every component C_i contains at most one contour pixel w_{is} that established a connection from b_i to e_i. No pixel on $S[b_i, e_i]$ is adjacent to a 1-pixel

belonging to a component, with the exception of pixel b_i. Therefore any cycle consists of 1-pixels that are part of the horizontal and vertical segments. The cycle must contain at least one horizontal segment $S[b_i, e_i]$ such that at least two pixels on this segment belong to the cycle. However, because of Property 3.1 and the fact that only e_i can be adjacent to a vertical segment, no such horizontal segment can exist. Similarly, the cycle must contain a vertical segment $S[b_i, e_i]$ such that at least two pixels on this segment belong to the cycle. But, because of Property 3.1 and the fact that horizontal segments cannot belong to a cycle, no such vertical segment can exist. Hence, image I^* satisfies the cycle-free property. □

The implementation details for the SPS method are straightforward. We only provide details about the last step of the first phase in which every component C_i selects, among the processors containing a scan pair originating from a contour pixel of C_i, the 0-pixel associated with the minimum cost entry. The selection process of Steiner pixel s_i for component C_i is done in two steps. First every contour pixel w_{ij} of C_i selects the minimum cost Steiner pixel containing the scan pair (w_{ij}, C_i), and then the Steiner pixel s_i is determined. This pixel is the best among the pixels chosen by the contour pixels of C_i and it is found by performing a connected component computation. Known connected component algorithms can easily be changed to compute the additional information needed in the Steiner pixel selection process. At the end of the first phase of the SPS method every component C_i has thus selected its Steiner pixel s_i which was reached by a scan originating at contour pixel w_{is}.

The final action in the SPS method (after the four phases have been completed) is a connected component computation to determine the components in image I^*. Every phase can thus be accomplished by either a scanning operation which partially scans a constant number of rows or columns or an $O(n)$ time connected component computation. Hence, the overall running time of the SPS method for connecting components is $O(n)$.

4. Comparisons of the Methods and the Algorithms

In Sections 2 and 3 we presented two methods for connecting components. We now compare these two methods and describe the connected component labeling algorithms based on them. Assume input image I consists of k connected components. One application of the MCS method generates a new image consisting of at most $4k / 5$ components. If no segments were deleted in the cycle-removing phase, then one application of the MCS method would result in at most $k / 2$ components. However, in the processing of case (7) during the cycle-removing phase, segments can be deleted. The worst case scenario is that two overlapping segments are deleted, leaving only one of the original three segments. Five components were initially involved in this configuration and since we have reduced the number of components by one (since the remaining segment connects two components), we obtain the $4k / 5$ bound. However, in practice, most applications of the MCS method will reduce the number of components by at least $1 / 2$. Note that it is also possible that one iteration of the MCS method succeeds in connecting all components.

One application of the SPS method does, in the worst case, connect no components. Possible reasons are that the first phase of the SPS method does not create enough potential Steiner pixels, every component selects a unique Steiner pixel, and components are too close together. Recall that we deposit scan pairs in processors at most distance $dist(p_i, q_i) - 2$ away from a contour pixel of component C_i. By changing the method slightly, it is possible to increase this distance to $dist(p_i, q_i) - 1$ and still obtain a cycle-free image. The proof of the theorem that I^* is cycle-free is more involved in this case and we thus chose to present the version with the $dist(p_i, q_i) - 2$ bound. Our implementation of the SPS method considered both distance bounds.

Even with the possibility that the SPS method does not connect any components, there are a number of situations in which the SPS method outperforms the MCS method. Consider, for example, the point set given in [Hw] for which the cost of a rectilinear minimum Steiner tree is indeed $2 / 3$ of the cost of a minimum spanning tree. This point set is illustrated in Figure 4.1(a). When transforming this example to an

image and running one application of the MCS method on it, we generate the solution shown in Figure 4.1(b) consisting of 27 1-pixels, which corresponds to the minimum spanning tree solution. On the other hand, if we run one application of the SPS method on this image, we generate the solution shown in Figure 4.1(c), which consists of 17 1-pixels, or approximately $2/3$ of the cost of a minimum spanning tree solution and is thus optimal. Note that this example can be made arbitrarily large by replicating the point set shown in Figure 4.1(a) and spacing point sets in the manner illustrated in the figure.

The MCS and SPS methods each have an $O(n)$ asymptotic running time with a small associated constant. The cycle removing phase of the SPS method is simpler than that of the MCS method. This is true since we avoid creating nearly all cycles in the SPS method by separating vertical and horizontal movement. A corresponding approach, however, does not work for the MCS method. Both methods perform connected component computations, scans on rows and columns, and local operations. We refer to [T] for a detailed discussion of the time and space requirements of the mesh implementations of the two methods. Note that the implementation of the MCS method does not perform any of the optimizations described in Section 2.2.

Our algorithms based on the two methods consist of a number of iterations, with each iteration applying one of the two methods. Our first algorithm, referred to as the *MCS algorithm*, uses the MCS method in each iteration.

Recall that in the MCS method Steiner pixels are created and 1-pixels are saved when segments overlap. This segment overlap can occur within an iteration or in multiple iterations. We define this to be *intra-iteration* overlap and *inter-iteration* overlap, respectively. Intra-iteration overlap occurs when segments formed in the same iteration overlap. Inter-iteration overlap occurs when a segment formed in an iteration chooses as its q-endpoint a 1-pixel that was part of a segment in a previous iteration. In the first iteration of the MCS method, many components are connected, which reduces the number of components that can participate in inter-iteration overlap in succeeding iterations. In order to facilitate more inter-iteration overlap, we created our second algo-

rithm, the MCS_SLOW algorithm. In this algorithm, every component chooses its segments as in the MCS algorithm. However, in the first iteration, only approximately half of the segments chosen are actually created. The result is that in succeeding iterations, there are more components to take advantage of connecting to 1-pixels resulting from segments created in the first iteration.

Our third algorithm, referred to as the *MIXED algorithm*, alternates between the SPS and the MCS methods, beginning with an SPS iteration. In the MIXED algorithm, the first iteration of the MCS method forms all of the segments as in the MCS algorithm. We also experimented with another version of this algorithm, one in which we alternated between two applications of the SPS method and one application of the MCS method. However, the solutions produced by this algorithm were inferior to the ones produced by alternating one SPS application with one MCS application. Therefore, we do not consider this version any further.

All three algorithms have a worst-case running time of $O(n \log k)$. It is easy to see that the solutions generated by the MCS and MCS_SLOW algorithms are never worse than the solutions generated by a rectilinear minimum spanning tree algorithm. While we have never experienced the MIXED algorithm to generate a solution of cost larger than that of a minimum spanning tree, this is theoretically possible and we briefly explain how. Assume we have three components, C_1, C_2, and C_3, positioned as shown in Figure 4.2. The SPS method connects the three components by using $8 + 8 + 8 + 1 = 25$ pixels, as illustrated by the shaded squares. A minimum spanning tree would connect the three components by using two edges and only 20 pixels, as illustrated by the empty squares. Hence, assuming we would use the SPS method in each iteration, we could use $1/2$ more pixels than used by a minimum spanning tree algorithm. This behavior could be corrected by depositing scan pairs in processors at most distance $dist(p_i, q_i)/2$ from the contour pixels. However, in practice this happens too infrequently to justify such a restriction.

We have run our algorithms on random images of size 128×128 which consist of m uniformly distributed 1-pixels. Tables 4.1, 4.2, 4.3, and 4.4 summarize our perfor-

mance results for $m = 100$, $m = 250$, $m = 500$, and $m = 750$, respectively. For each value of m, we ran each algorithm on 20 images. The MIXED algorithm shown in the tables uses the SPS method with a $dist(p_i, q_i) - 1$ bound. The last line in both tables gives the average performance for the considered set of data. Recall that k is the number of connected components in the input image I. In actual number of 1-pixels, for images consisting of $m = 100$ 1-pixels, the average minimum spanning tree consists of 983.45 1-pixels, the Steiner tree created by the MCS algorithm consists of 907.9 1-pixels, the Steiner tree created by the MCS_SLOW algorithm consists of 903.6 1-pixels, and the Steiner tree created by the MIXED algorithm consists of 897.75 1-pixels. For images with $m = 250$, these values are 1417.65 1-pixels, 1301.9 1-pixels, 1297.05 1-pixels, and 1289.7 1-pixels, respectively. For images with with $m = 500$ these values are 1869.0, 1702.1, 1696.85, and 1690.15 1-pixels, respectively, and for images with $m = 750$, these values are 2129.05, 1934.3, 1920.55, and 1925.55 1-pixels.

From Tables 4.1, 4.2, 4.3, and 4.4 one can see that for the images we tested our algorithms on, our solutions average about 91% of the cost of a minimum spanning tree solution. In terms of the solutions produced, all of the algorithms performed similarly, with the MCS_SLOW algorithm and the MIXED algorithm performing slightly better than the MCS algorithm. As m increased, the performance of the algorithms generally increased, which can be explained as follows. As m increases, the MCS method needs more segments to connect the components, therefore there is more of a chance that segments overlap, creating Steiner pixels. In addition, as m increases, there are more potential Steiner pixels for the SPS method, which helps increase the performance of the MIXED algorithm as well.

One downfall of the MIXED algorithm is the fact that it takes about twice as many iterations to complete as the other two algorithms. The first two SPS iterations of the MIXED algorithm are the SPS iterations where the most Steiner pixels are formed. One solution to the iteration problem would be to alternate SPS and MCS iterations for four iterations of the algorithm, and then to finish the algorithm using only MCS iterations. This would reduce the number of iterations with little affect on the performance of the algorithm.

The performance of our algorithms compares favorably with the results of other algorithms cited in the literature [HVW1, HVW2, Ri]. These algorithms often have solutions averaging approximately 90% - 92% of the corresponding minimum spanning tree [HVW1, HVW2, Ri]. The algorithms cited in the literature are sequential algorithms. Therefore, if a segment is formed, the algorithm can use this segment immediately in forming succeeding segments. For parallel algorithms such as ours, in one iteration many segments are formed simultaneously, and it is not until the following iteration that the algorithm can use these segments when forming additional connections. Considering these facts, we consider the performance of our algorithms to be good.

References

[AH] M.J. Atallah and S.E. Hambrusch, "Solving Tree Problems on a Mesh-Connected Processor Array", *Proceedings of 26th FOCS*, pp. 222-231, 1985.

[Ba] K.E. Batcher, "Design of a Massively Parallel Processor", *IEEE Transactions on Computers*, Vol. C-29, No. 9, pp. 836-840, September 1980.

[Be] M.W. Bern, "Two Probabilistic Results on Rectilinear Steiner Trees", *Proceedings of 18th Annual ACM Symposium on Theory of Computing*, pp. 433-441, 1986.

[CSS] R.E. Cypher, J.L.C. Sanz, and L. Snyder, "Algorithms for Image Component Labeling on SIMD Mesh Connected Computers", *IEEE Transactions on Computers*, Vol. 39, No. 2, pp. 276-281, February 1990.

[CSW] C. Chiang, M. Sarrafzadeh, and C.K. Wong, "Global Routing Based on Steiner Min-Max Trees", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 9, No. 12, pp. 1318-1325, December 1990.

[GJ] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, W.H. Freeman, 1979.

[H] S.E. Hambrusch, "VLSI Algorithms for the Connected Component Problem", *SIAM Journal on Computing*, Vol. 12, No. 2, pp. 354-365, 1983.

[Ha] M. Hanan, "On Steiner's Problem with Rectilinear Distance", *SIAM Journal on Applied Mathematics*, Vol. 14, No. 2, pp. 255-265, March 1966.

[Hw] F.K. Hwang, "On Steiner Minimal Trees with Rectilinear Distance", *SIAM Journal on Applied Mathematics*, Vol. 30, No. 1, pp. 104-114, January 1976.

[HCS] D.S. Hirschberg, A.K. Chandra, and D.V. Swarte, "Computing Connected Components on Parallel Computers", *Communications of the ACM*, Vol. 22, No. 8, pp. 461-464, August 1979.

[HT] S. Hambrusch and L. TeWinkel, "A Study of Connected Component Algorithms on the MPP", *Proceedings of 3rd International Conference on Supercomputing*, pp. 477-483, May 1988.

[HVW1] J. Ho, G. Vijayan, and C.K. Wong, "A New Approach to the Rectilinear Steiner Tree Problem", *Proceedings of 26th ACM/IEEE Design Automation Conference*, June 1989.

[HVW2] J. Ho, G. Vijayan, and C.K. Wong, "New Algorithms for the Rectilinear Steiner Tree Problem", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 9, No. 2, pp. 185-193, February 1990.

[LAN] W. Lim, A. Agrawal, and L. Nekludova, "A Fast Parallel Algorithm for Labeling Connected Components in Image Arrays", in *Parallel Processing for Computer Vision and Display*, Ed. P.M. Dew, R.A. Earnshaw, T.R. Heywood, Addison-Wesley, 1989.

[LBH] J.H. Lee, N.K. Bose, and F.K. Hwang, "Use of Steiner's Problem in Suboptimal Routing in Rectilinear Metric", *IEEE Trans. on Circuits and Systems*, Vol, CAS-23, No. 7, pp. 470-476, July 1976.

[MS1] R. Miller and Q. Stout, "Geometric Algorithms for Digitized Pictures on a Mesh-Connected Computer", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-7, No. 2, pp. 216-228, March 1985.

[MS2] R. Miller and Q. Stout, "Mesh Computer Algorithms for Computational Geometry", *IEEE Transactions on Computers*, Vol. 38, No. 3, pp. 321-340, March 1989.

[MS3] R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Architectures*, manuscript, (to be published by MIT Press).

[NASA] *MPP Pascal Programmer's Guide*, National Aeronautics and Space Administration - Goddard Space Flight Center, March 1988.

[NS] D. Nassimi and S. Sahni, "Finding Connected Components and Connected Ones on a Mesh-Connected Parallel Computer", *SIAM Journal on Computing*, Vol. 9, No. 4, pp. 744-757, November 1980.

[Ri] D. Richards, "Fast Heuristic Algorithms for Rectilinear Steiner Trees", *Algorithmica*, Vol. 4, No. 2, pp. 191-207, 1989.

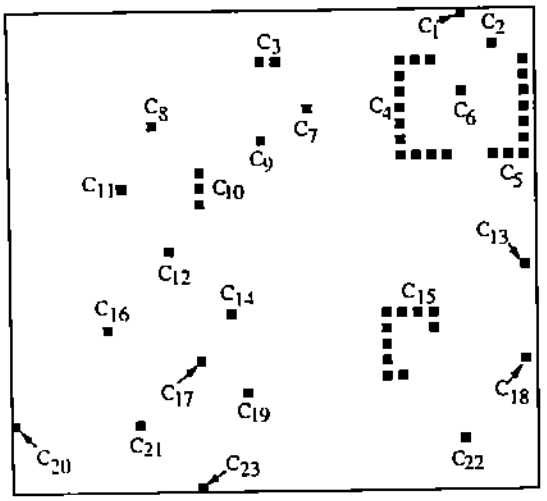
[RM] A. Reeves and C. Moura, "Data Manipulation on the Massively Parallel Processor", *Proceedings of 19th Hawaii International Conference on Systems Sciences*, pp. 222-229, 1986.

[St] Q. Stout, "Tree-based Graph Algorithms for some Parallel Computers", *Proceedings of 1985 International Conference on Parallel Processing*, pp. 727-730, 1985.

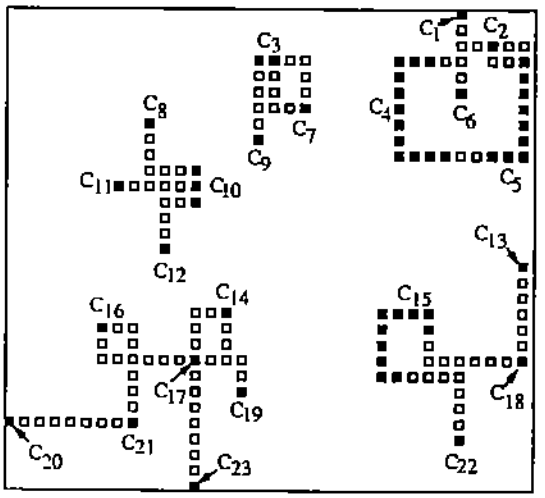
[SV] Y. Shiloach and U. Vishkin, "An $O(\log n)$ Parallel Connectivity Algorithm", *Journal of Algorithms*, Vol. 3, No. 1, pp. 57-67, March 1982.

[T] L. Te Winkel, "Mesh Algorithms for Problems in Image Processing", Ph.D. Dissertation, Purdue University, 1991.

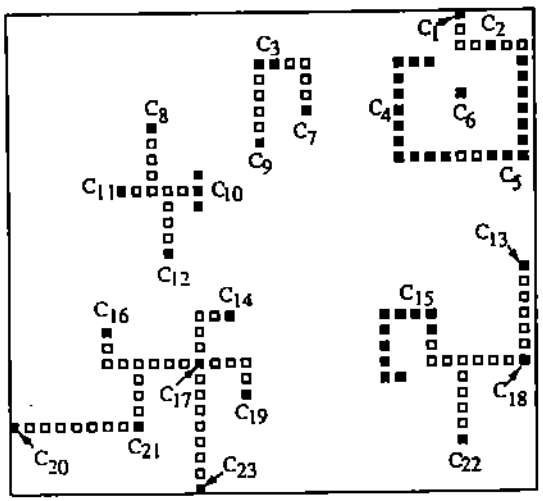
[Ul] J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.



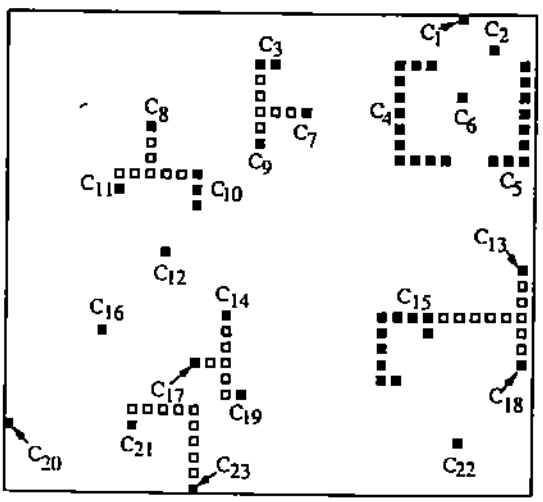
(a) Image I



(b) Image CI



(c) Image I^* using the MCS method



(d) Image I^* using the SPS method

Illustrations for the MCS and SPS Methods
Figure 1.1

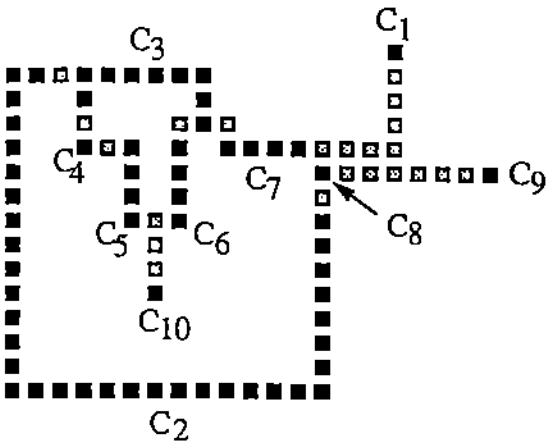
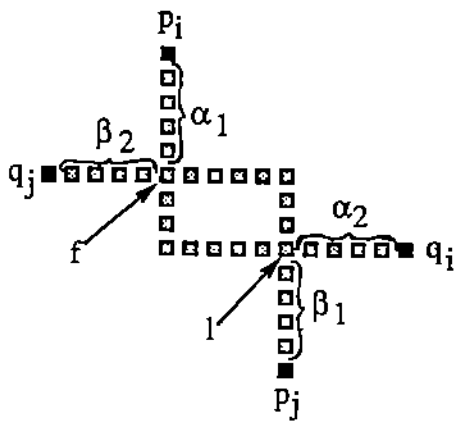
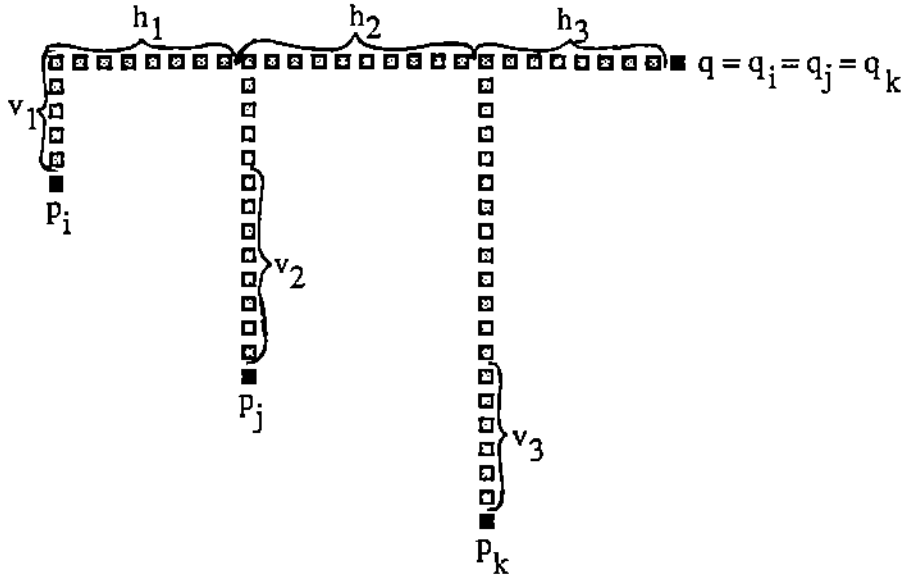


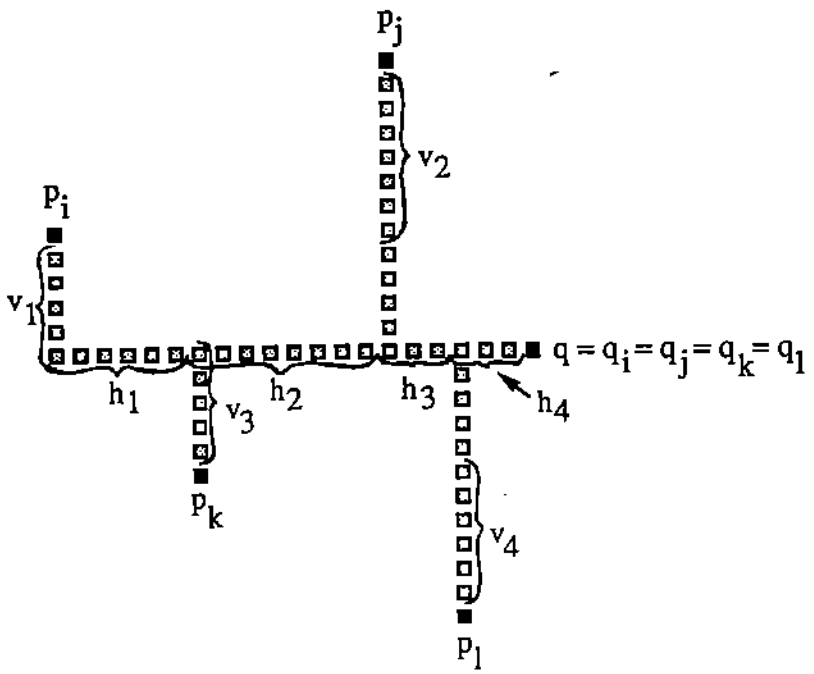
Image *I* contains components at distance 1
Figure 2.1



Segments S_i and S_j are crossing segments with non-shared endpoints
Figure 2.2



Segments S_i, S_j, and S_k are overlapping segments of type -
Figure 2.3



Segments S_i, S_j, S_k, and S_l are overlapping segments with segments S_i and S_j of type + and segments S_k and S_l of type -
Figure 2.4

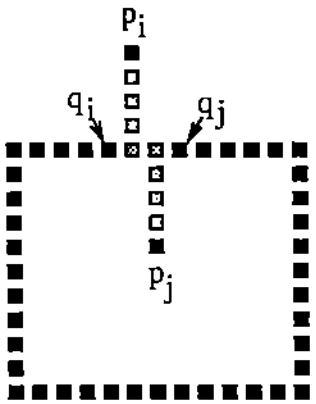
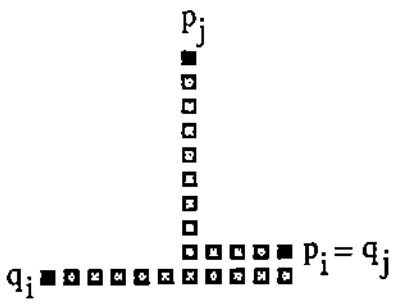
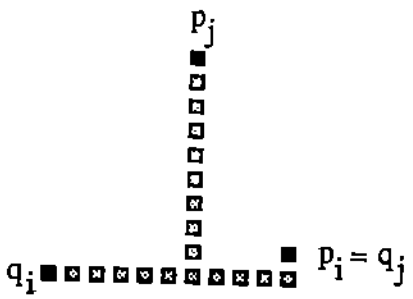


Illustration of case (7) (ii) where changing
q-endpoints does not reduce the number of 1-pixels
Figure 2.5

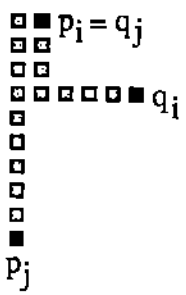


(a) After segments have been created

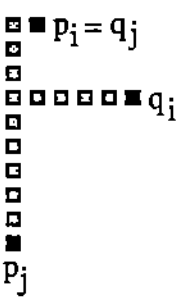


(b) After cycle-removing phase

S_i and S_j are adjacent horizontal segments with $p_i = q_j$
Figure 2.6

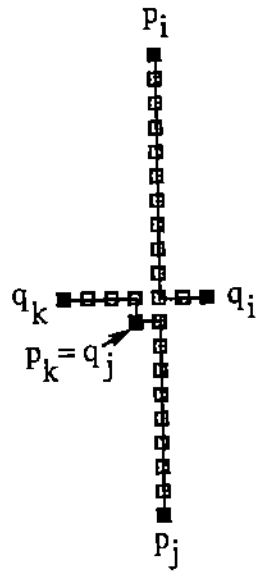


(a) After segments have been created



(b) After cycle-removing phase

S_i and S_j are adjacent vertical segments with $p_i = q_j$
Figure 2.7



Segment S_i is adjacent with non-neighboring endpoints to both segment S_j and segment S_k
Figure 2.8

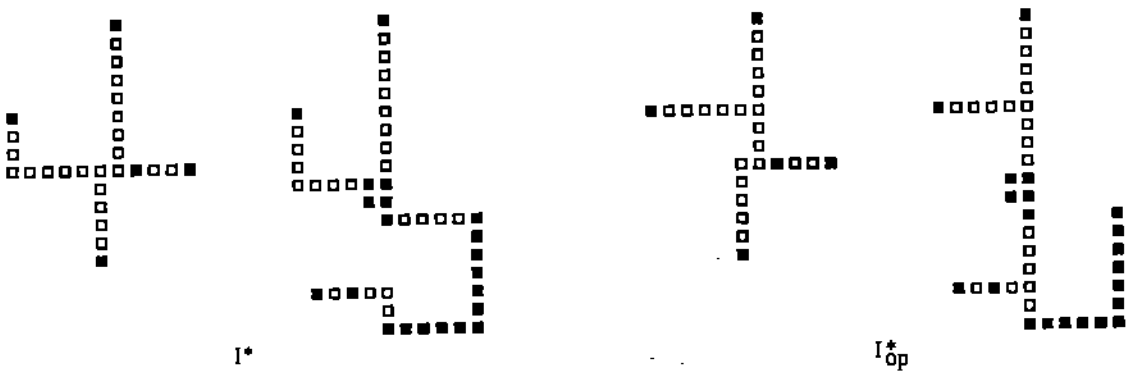
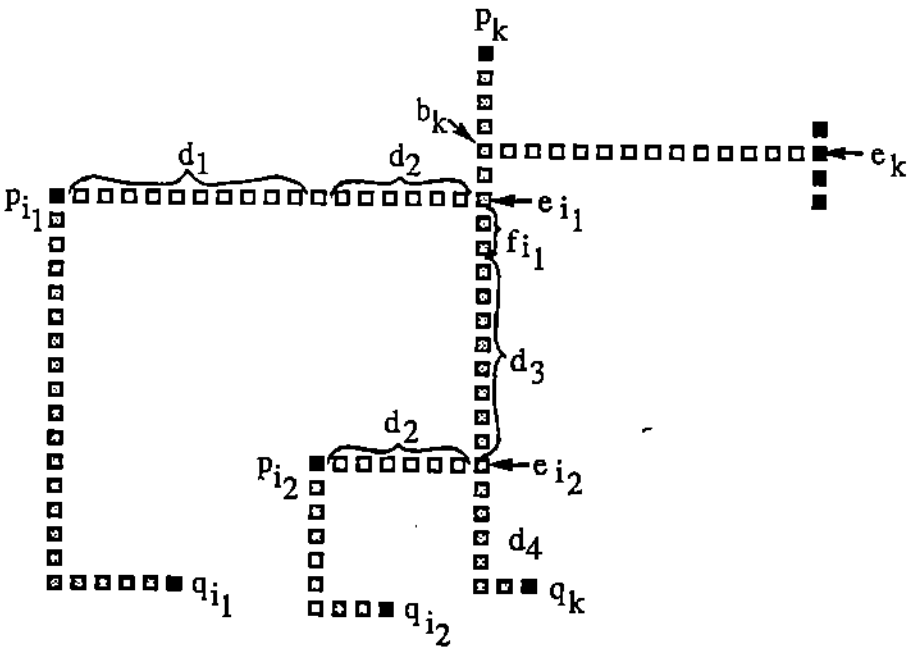
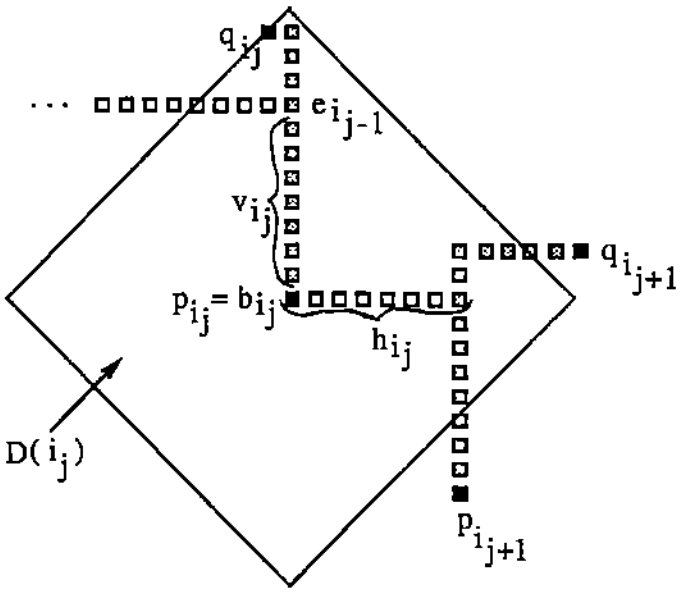


Image I^* uses 45 pixels and image I^*_{op} uses 39 pixels; components are indicated by solid squares, segments by empty squares
Figure 2.9

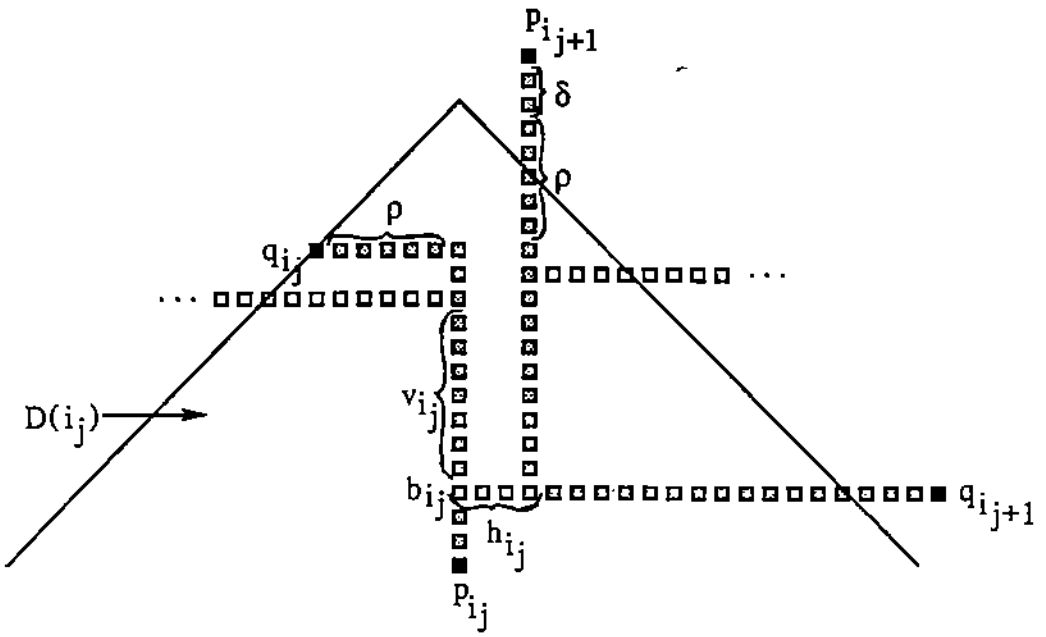


An example of the situation when both segment i_1 and k do better by taking their proposed shortcuts; segments are indicated by filled squares, proposed shortcuts by empty squares

Figure 2.10

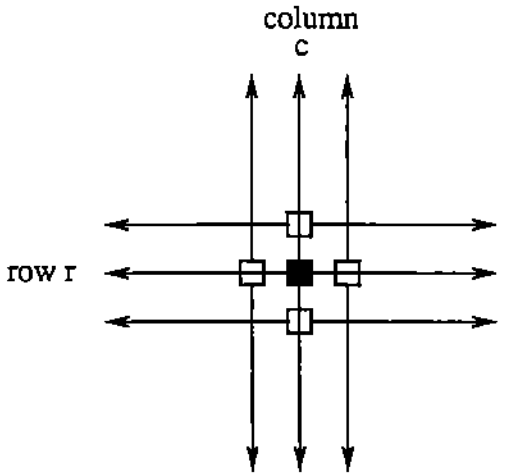


(a) Segments i_j and i_{j+1} have the same type

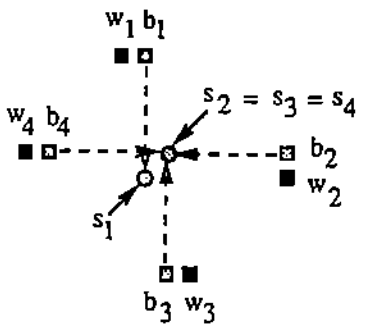


(b) $p_{i_{j+1}}$ lies "above the top of $D(i_j)$ (Case 2.2)

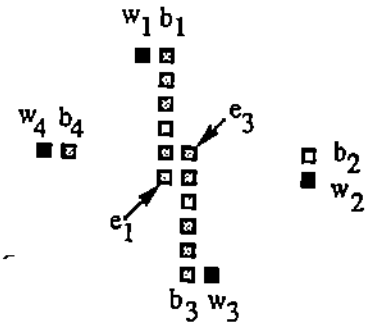
Illustration for the proof of Lemma 2.3
Figure 2.11



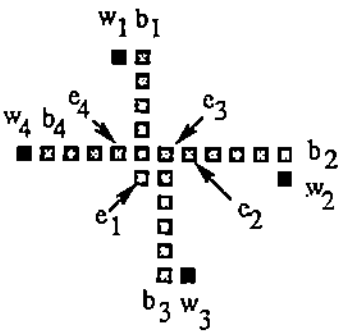
The twelve scans initiated by an isolated 1-pixel
Figure 3.1



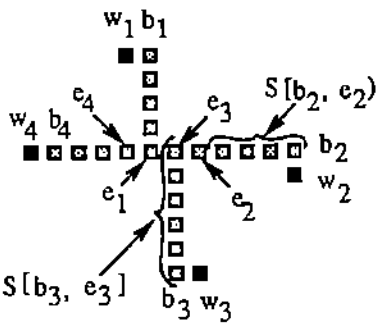
(a) Selection of Steiner pixels



(b) Image after vertical segments have been created

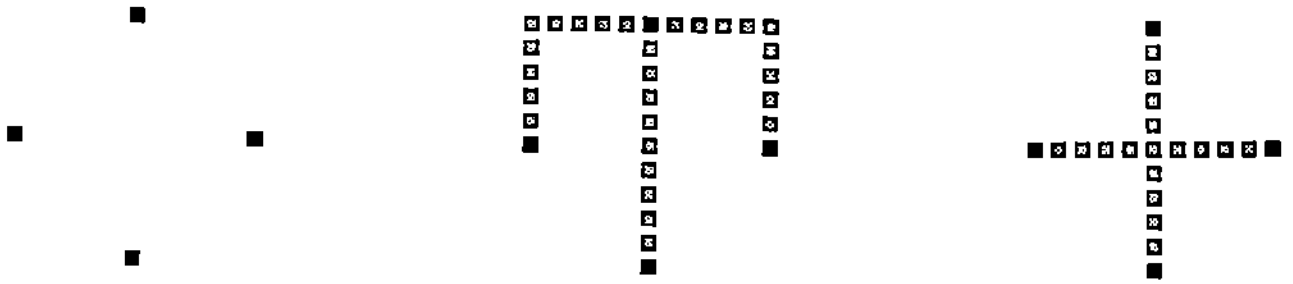


(c) Image after horizontal segments have been created



(d) Image after cycle-removing phase

Illustration of how segments are created in the SPS method
Figure 3.2

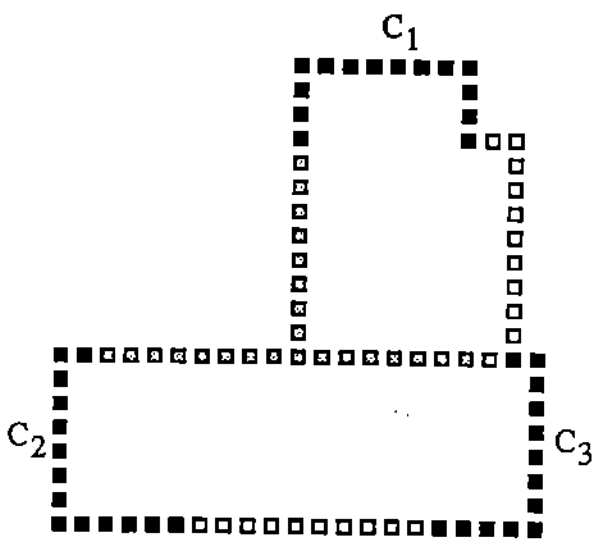


(a) Point set

(b) MCS method and minimum spanning tree solutions consisting of 27 1-pixels

(c) SPS method solution consisting of 17 1-pixels

Example of where the cost of the rectilinear minimum Steiner tree is 2 / 3 of the cost of the minimum spanning tree
Figure 4.1



Example of where SPS method (25 shaded squares) exceeds cost of minimum spanning tree (20 empty squares)
Figure 4.2

	MCS Algorithm		MCS_SLOW Algorithm		MIXED Algorithm	
k	iterations	% of msp	iterations	% of msp	iterations	% of msp
98	4	92.70	5	92.60	8	91.14
97	4	92.78	4	92.88	8	91.98
99	4	91.78	4	91.60	6	91.12
100	4	91.45	4	91.66	6	91.97
99	4	92.50	4	91.88	8	89.31
99	4	90.74	5	90.53	8	90.11
100	4	91.96	4	91.45	8	91.25
99	4	93.63	4	93.19	8	92.86
99	3	92.77	4	92.29	8	90.63
99	4	92.11	4	91.71	6	91.71
99	4	92.05	5	91.05	8	91.25
98	4	90.42	4	90.94	8	91.25
97	4	92.02	5	92.41	8	91.23
96	4	92.04	4	91.14	6	89.43
100	3	91.92	4	90.93	6	91.82
94	4	92.04	5	91.62	8	91.20
99	4	94.23	4	93.61	8	93.08
100	4	92.20	4	90.68	8	90.27
100	4	93.27	5	93.18	8	91.42
99	4	93.83	4	92.37	8	92.89
Averages	3.90	92.32	4.30	91.89	7.50	91.30

Results for random images of size 128×128 with $m = 100$
Table 4.1

	MCS Algorithm		MCS_SLOW Algorithm		MIXED Algorithm	
k	iterations	% of msp	iterations	% of msp	iterations	% of msp
240	4	91.08	5	91.22	8	90.44
243	5	90.88	5	90.61	8	90.07
242	5	90.75	5	90.55	8	89.74
239	5	93.20	5	92.61	10	92.47
242	5	92.26	5	92.04	8	90.97
242	5	91.29	5	90.94	10	90.80
244	4	92.69	4	91.79	8	92.34
245	4	91.84	5	91.29	8	90.11
242	5	91.63	5	91.70	10	90.93
241	5	90.99	5	91.14	10	89.18
242	5	92.35	5	92.28	8	90.96
243	5	92.65	5	92.80	10	92.00
239	5	91.58	5	91.79	8	91.50
241	4	91.95	5	91.13	8	90.18
241	5	91.87	5	91.58	8	91.08
243	5	91.44	5	91.01	8	90.72
242	5	91.57	5	91.30	8	91.36
240	5	93.36	6	91.41	8	92.93
246	5	91.05	5	91.12	10	90.69
242	5	92.39	5	91.69	8	91.20
Averages	4.80	91.84	5.00	91.50	8.60	90.98

Results for random images of size 128×128 with $m = 250$
Table 4.2

	MCS Algorithm		MCS_SLOW Algorithm		MIXED Algorithm	
k	iterations	% of msp	iterations	% of msp	iterations	% of msp
464	5	91.93	5	91.66	10	90.95
469	5	91.12	6	91.02	10	90.43
467	4	90.55	5	89.80	10	89.91
474	5	90.11	6	90.59	10	89.58
471	5	90.90	5	90.69	8	90.49
481	5	91.43	5	91.06	10	90.96
470	5	90.89	6	90.59	10	90.32
463	5	90.21	6	89.63	10	89.21
481	4	90.83	5	90.51	8	90.24
467	5	91.57	5	91.51	10	91.07
473	5	90.83	6	90.61	10	90.08
466	5	90.35	6	89.91	10	89.53
466	5	91.29	6	91.68	10	90.91
479	5	90.62	5	90.83	10	89.94
466	5	91.91	6	91.74	10	91.14
473	5	91.17	5	90.81	10	90.71
471	5	92.56	6	92.56	10	91.75
469	5	90.65	6	90.12	10	90.01
465	5	90.78	6	90.03	10	90.19
474	5	91.78	6	90.50	10	91.25
Averages	4.90	91.07	5.60	90.79	9.80	90.43

Results for random images of size 128×128 with $m = 500$
Table 4.3

	MCS Algorithm		MCS_SLOW Algorithm		MIXED Algorithm	
k	iterations	% of msp	iterations	% of msp	% iterations	% of msp
677	6	90.91	6	90.26	12	90.40
677	5	90.58	5	90.29	10	90.05
685	5	90.92	6	90.17	10	90.36
683	5	91.68	6	91.21	10	91.26
677	5	90.19	5	89.55	10	89.92
680	5	91.31	6	90.85	10	90.85
684	5	90.69	5	90.55	10	90.17
689	6	90.47	6	89.38	12	90.18
680	5	90.79	6	89.70	10	90.46
683	5	91.03	6	90.32	10	90.37
687	5	90.00	5	89.73	10	90.05
679	5	91.57	6	90.52	10	91.09
677	5	91.97	6	91.01	10	91.49
674	6	91.30	6	90.92	10	91.01
692	6	90.63	5	89.74	12	90.26
663	5	91.95	5	91.90	10	91.37
678	5	90.49	6	90.16	10	90.11
694	5	90.15	6	89.01	10	89.74
694	5	89.83	5	89.38	10	89.51
671	5	90.75	6	89.65	10	90.32
Averages	5.20	90.86	5.65	90.22	10.30	90.45

Results for random images of size 128×128 with $m = 750$
Table 4.4