Chapter 3

Research Questions

3.1 Possible Research Areas

As discussed in the introduction to this thesis (Chapter 1), the range of possible Computer Science research questions that arise from the idea of automating space syntax is very wide. These research questions fall into a number of areas:

- using image processing techniques to separate space from non-space in the town plan or aerial photograph and then using polygon approximation algorithms to "accurately" and "efficiently" represent each area by a bounding polygon,
- studying ways to find the convex map (the minimum number of non-overlapping convex polygons that cover the area of the spaces in the deformed grid) of the area under consideration,
- studying ways to find the axial map (the smallest number of axial lines that will cross all of the shared boundaries between the convex spaces in the convex map) of the area, and
- studying the graph theory and other algorithms used in the final stages of applying space syntax.

This breadth of possible research meant that it was thus necessary to concentrate on a subset of the problems. For this reason, the problems of separating space from non-space, determining the convex map and the final analysis stage with its associated algorithms were not considered as part of this research. The decision was made to focus the research for this PhD on the problem of finding the axial lines that cross all of the shared boundaries between the convex polygons in the convex map, that is finding the axial map for a given layout. This problem has not been previously studied although it is a variation on a number of guarding and visibility problems that have been well studied in the literature (see Chapter 2).

The problem which is the focus of this thesis can then be stated as
Axial Line Placement (ALP): Given a convex map of an urban area determine the axial map required to cover the convex map.

This can be restated as
Axial Line Placement (ALP): Given a collection of adjacent convex polygons representing the convex map of an urban area, find the minimum number of maximum length straight line segments contained wholly inside the convex polygons (axial lines) that will cross every adjacency (shared edge) between the polygons.

This problem has two variations

multiple crossings where each adjacency must be crossed by *at least* one axial line.

single crossing where each adjacency must be crossed by *exactly* one axial line.

Neither of these two variations of ***ALP*** have been previously studied and both were thus candidates for the research in this thesis. The fact that many of the guarding and covering or partitioning problems that appear in the literature have been shown to be NP-Complete or NP-Hard indicates that these problems might also be. The focus of research on the problems should thus be on determining if they are also NP-Complete or NP-Hard. If the problems turn out to be NP-Complete or NP-Hard then later research effort could be focussed finding good heuristics. If they can be solved in polynomial time then later research effort could be focussed on finding good algorithms to solve the problems.

The literature also shows that it is worthwhile studying restrictions of the more general problems because even if the more general problem is NP-Complete or NP-Hard a restriction may not be. A number of restrictions of ***ALP*** could thus be considered as potential research areas. In addition, the problem as stated above assumes that the configuration of adjacent convex polygons represents the convex map of some urban layout. If the configuration of convex polygons does not have to represent an urban layout then a slightly different problem results.

A list of some restrictions or generalisations of the original problem that would be interesting areas of research is given below.

1. Restricting the problem to dealing with orthogonally aligned rectangles rather than general convex polygons and restricting the axial lines to be parallel to the edges of the rectangles. In this case there are two subproblems (see Figure 3.1)
   (a) where each adjacency must be crossed by *at least* one axial line.
   (b) where each adjacency can be crossed by *only* one axial line.

A requirement here is that each axial line is maximal in length. In this work maximal is taken as meaning that the axial line crosses as many adjacencies as possible.

2. Restricting the problem to dealing with orthogonally aligned rectangles where the adjacencies between rectangles are cut by axial lines that are not necessarily orthogonal.
   (a) where each adjacency must be crossed by *at least* one axial line.
   (b) where each adjacency can be crossed by *only* one axial line.

   Again a requirement is that each axial line is maximal.

3. Considering the problem of general convex polygons (not restricted to the form that would occur in the town planning domain) where the adjacencies between polygons are crossed by axial lines that are not necessarily orthogonal.
   (a) where each adjacency must be crossed by *at least* one axial line.
   (b) where each adjacency can be crossed by *only* one axial line.

   Again each axial line must be maximal.

Each of these variations of ***ALP*** poses some interesting questions. Is the general problem solvable in polynomial time? Are only specific instances solvable in polynomial time? Is the problem NP-Complete? If so, are there heuristics that offer acceptable solutions?

The actual problems tackled in the research reported in this thesis are discussed in the next section of the document.

3.2 Scope of this thesis

As discussed above there are many problems in this area that could be addressed but tackling all of them would be too much for a single PhD thesis. Thus, this research only considered some of the problems listed in Section 3.1. In this thesis only the variations of the problems where adjacencies may be crossed by more than one axial line (**multiple crossings** above) was studied. The decision to restrict the study to this case was based on the fact that the original urban design problem allows multiple crossings of adjacencies. The problems actually tackled are discussed below.

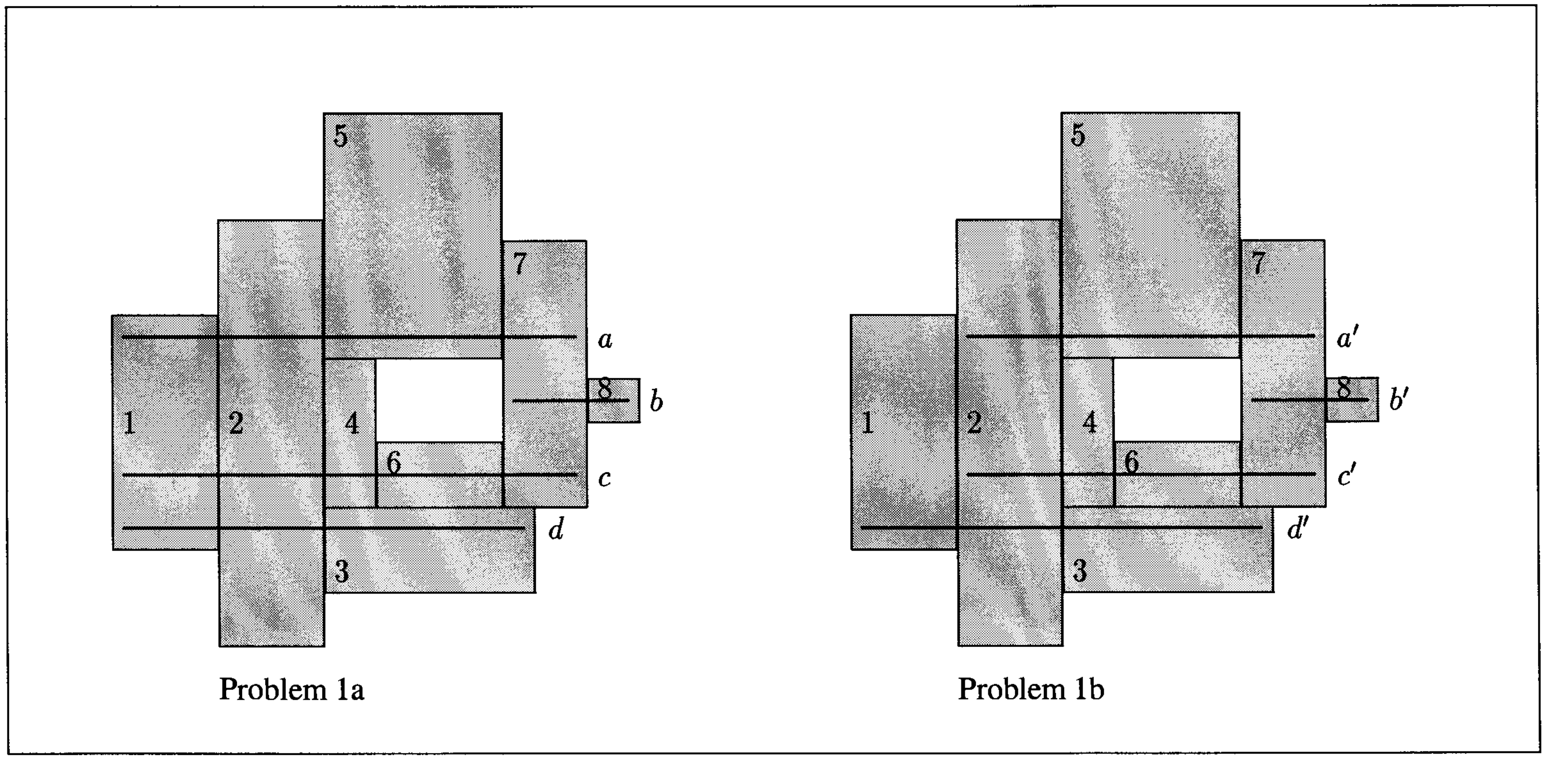

Figure 3.1: A simple configuration showing the two different problems

1. Restricting the problem to dealing with orthogonally aligned rectangles where the adjacencies between rectangles are crossed by axial lines that are restricted to being parallel to the edges of the rectangles (orthogonal) and where each adjacency must be crossed by at least one axial line (1a above).

2. Restricting the problem to dealing with orthogonally aligned rectangles where the adjacencies between rectangles are crossed by lines with arbitrary orientation and where each adjacency must be crossed by at least one axial line (2a above).

3. Considering the problem of general convex polygons (not restricted to the form that would occur in the town planning domain) where the adjacencies between polygons are crossed by axial lines with arbitrary orientation and where each adjacency must be crossed by at least one axial line (3a above).

4. Given a convex map of an urban area determine the minimum number of axial lines required to cover the convex map.

The major emphasis of this thesis is on the first of these problems – orthogonal axial lines crossing the adjacencies between rectangles in configurations of adjacent orthogonal rectangles (see point 1 above). Chapter 4 presents an NP-Completeness proof based on a transformation from vertex cover for planar graphs to show that the problem is NP-Complete and then presents a heuristic algorithm to give a "reasonable" approximation to the exact solution. The chapter concludes by discussing some special cases of the problem that can be solved in polynomial time.

In Chapter 5 the problem of point 2 is considered. Again the problem is shown to be NP-Complete. Future work here would be to develop heuristic algorithms to produce good approximations to the solution. Some heuristics are suggested but they are not fully developed in this research.

Chapter 6 considers the problem in point 3. This problem is a generalisation of 2a and is easily shown to be NP-Complete. Again, future work would be to develop heuristic algorithms to produce good approximations to the solution.

The original ***ALP*** problem (point 4) is a constrained version of that discussed above (point 3) and is considered in Chapter 7 of this thesis.

Chapter 4

Placing orthogonal axial lines to cross adjacencies between orthogonal rectangles

4.1 Introduction

This thesis is concerned with the computational issues that result from attempting to automate the placing of axial lines through the convex spaces in a town plan in the space syntax method [Hillier *et al.*, 1983]. This chapter and chapter 5 concern simplifications of the original problem – the placing of straight lines through collections of orthogonal rectangles. In this chapter the problem is simplified even more, only the problem of placing orthogonal axial lines – axial lines parallel to the Euclidean axis – through a collection of orthogonal rectangles is considered (problem 1a of Chapter 3).

As mentioned earlier (Chapter 2), the problem is similar to many guarding and visibility problems [Bjorling-Sachs and Souvaine, 1991, 1995; Czyzowicz *et al.*, 1994; Gewali and Ntafos, 1993] since axial lines coincide with visibility between two points. The situation can also be envisaged as an art gallery made up of a number of adjacent rooms where the designers wish to position doorways between the rooms in such a way that the minimum number of guards who can only see along a straight line (or laser beams, video cameras, etc.) is required to guard all of the doorways between rooms. To allow easy access between rooms extra doors can be added if a guard's line of sight is blocked by an interior wall. This is equivalent to making the axial lines as long as possible. Another application of this problem is in the design of integrated circuits. Here the problem is the siting of the fewest connecting strips to join all of the components on the chip.

The next section of this chapter presents a formal statement of the problem which is considered. In subsequent sections this problem is shown to be NP-

Complete (Section 4.4) and then a heuristic algorithm that produces a non-redundant solution is presented along with some results of testing this heuristic algorithm on some synthetic test data. The chapter ends with a discussion of some special cases where an exact solution can be found in polynomial time.

4.2 Statement of the Problem

Given a number of adjacent, orthogonally-aligned rectangles, find the fewest orthogonal line segments, contained wholly inside the rectangles, required to cross all of the adjacencies between adjacent rectangles. An additional requirement is that each line segment should cut as many of the adjacencies as possible – the line should be maximal.

The solutions for horizontal line segments (and vertical adjacencies) and vertical line segments (and horizontal adjacencies) are independent and the remainder of this chapter will only discuss the former. The latter problem can be solved by a rotation through 90 degrees.

Depending on how the problem is considered there are 2 similar but distinct problems that can be addressed.

1. The adjacencies between adjacent rectangles can be crossed more than once but every shared boundary must be crossed at least once.

2. Any adjacency between adjacent rectangles has exactly one orthogonal line segment passing through it.

Figure 4.1 shows the difference between these two specifications for a simple configuration of adjacent rectangles. In problem 1 the leftmost adjacency is cut by lines a, c and d. In problem 2, any of a', c' or d' could have cut the leftmost adjacency but only d' actually does. In this thesis only problem 1 is addressed.

In the remainder of this chapter (and thesis) this problem is referred to as ***ALP-OLOR*** Axial Line Placement – Orthogonal Lines and Orthogonal Rectangles.

4.3 Addressing the problem

At first glance this problem would appear to be easy to solve. Given n rectangles, the upper bound on the number of possible adjacencies is $O(n)$ and a simple lower bound for finding the adjacencies can be shown to be $\Omega(n \log n)$.

The upper bound can be easily shown by reducing the rectangles and their adjacencies to the form of a graph where the nodes in the graph represent the rectangles and the edges of the graph represent adjacencies between two rectangles. The graph generated in this way must be planar, thus the maximum number of edges (adjacencies) can be determined from Euler's formula that gives $e \leq 3v - 6$ (e the number

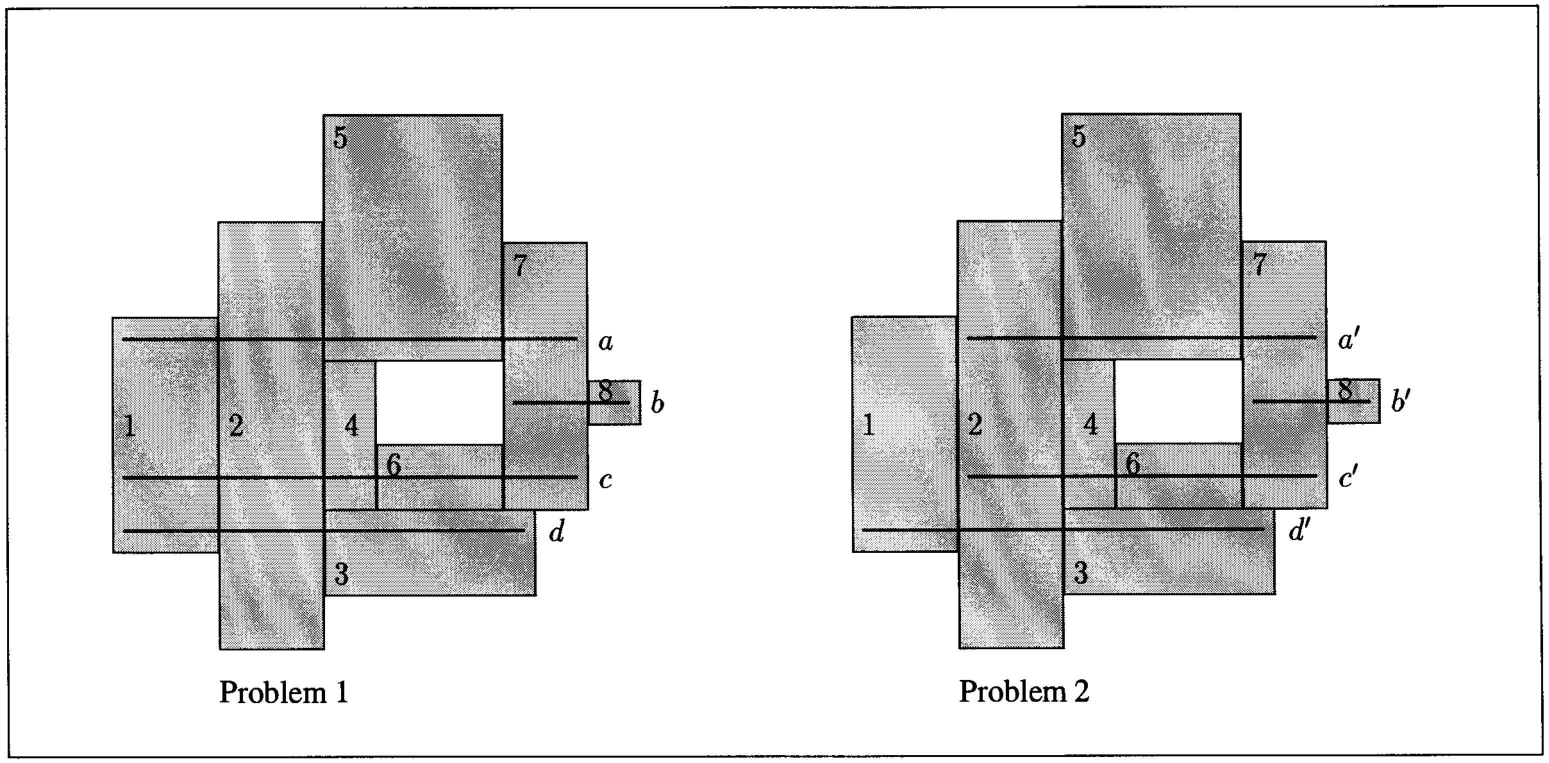

Figure 4.1: A simple configuration showing the two different problems

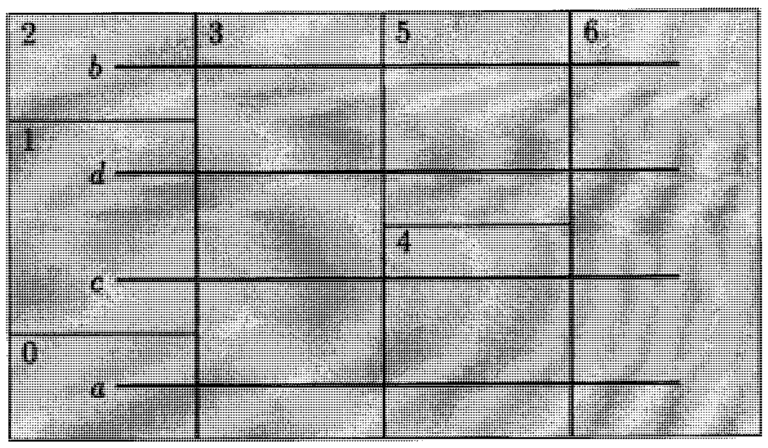

Figure 4.2: A configuration where the solution is not unique

of edges and v the number of vertices). This implies that the number of adjacencies must be $O(n)$.

The lower bound follows by a transformation from ***element uniqueness***. This proof is similar to that by Preparata and Shamos [1985] for intersecting rectangles. ***Element uniqueness*** is defined as: Given n real numbers, decide if any two are equal. ***Element uniqueness*** has an $\Omega(n \log n)$ lower bound [Preparata and Shamos, 1985]. The transformation can be done as follows. Given n real numbers $\{z_1, \ldots, z_n\}$ and an interval $[b, t]$ then for each z_i construct a (degenerate) rectangle defined by bottom left corner (b, z_i) and top right corner (t, z_i). Determining whether any two rectangles are adjacent is now the same as determining whether any two numbers are equal.

The question is then: How easy is it to find the minimum number of axial lines that cross all of the adjacencies in the collection of adjacent rectangles?

The problem is interesting and difficult to solve efficiently because of the issue of choice. The simplest case of choice is illustrated in Figure 4.2. In this case there are seven rectangles and seven adjacencies (0|3, 1|3, 2|3, 3|4, 3|5, 4|6 and 5|6) that must be crossed by axial lines. All but one of the adjacencies can be crossed by the axial lines marked a and b (0-3-4-6 and 2-3-5-6) but the adjacency between rectangles 1 and 3 can be crossed by axial lines c (1-3-4-6) and d (1-3-5-6). Only one of these "choice" axial lines is actually necessary. More complicated choice situations can arise as the number of rectangles to be considered grows. An algorithm to solve the problem must be able to resolve conflicts of this type.

4.4 Proving NP-Completeness of the problem of resolving choice

This section shows ***ALP-OLOR*** is NP-Complete. The proof of this will be accomplished through a transformation from ***vertex cover*** for a planar graph [Garey and Johnson, 1979; Lichtenstein, 1982], to a restricted instance of the problem under consideration, i.e. the problem of choosing the fewest maximal axial lines to cross all the adjacencies in a collection of orthogonal rectangles.

Planar vertex cover is defined as

planar vertex cover
Instance: Planar graph $G = (V, E)$, positive integer $K \leq |V|$.
Question: Is there a vertex cover of size K or less for G, i.e. a subset $V' \subseteq V$ with $|V'| \leq K$ such that for each edge $\{u, v\} \in E$ at least one of u and v belongs to V'?

and ***ALP-OLOR*** can be stated as

ALP-OLOR
Instance: A collection of orthogonal rectangles $R_1 \dots R_n$, where each R_i is adjacent to at least one other rectangle, and a positive integer $O \leq 4n$.
Question: Is there a set P of orthogonal axial lines where each axial line is maximal, each vertical adjacency is crossed at least once by the axial lines in P and $|P| \leq O$?

The transformation from ***planar vertex cover*** [Garey and Johnson, 1979; Lichtenstein, 1982] will be done by mapping vertices in a planar graph to choice axial lines in the problem being considered. Edges in the planar graph will be mapped to adjacencies that are crossed by the choice axial lines. In this mapping an edge between two vertices represents an adjacency that is crossed by two choice axial lines.

This transformation will be done in two steps. First, a planar graph is transformed to a 'stick diagram'. In this 'stick diagram' each vertex in the original graph is mapped to a horizontal line representing a choice axial line and each edge in the original graph is mapped to a vertical line that is cut by the two horizontal lines that represent the two vertices to which the edge is incident. The problem then becomes that of choosing the minimum number of horizontal lines to cut all of the vertical lines.

stick diagram
Instance: A collection H of horizontal lines and U of vertical lines such that each vertical line is cut by exactly two horizontal lines, and a positive integer $S \leq |H|$.

Question: Is there a set of horizontal lines, $H' \subseteq H$, such that every vertical line in U is cut at least once and $|H'| \leq S$?

The reader should note that the idea of converting planar graphs to horizontal and vertical lines has been published elsewhere [Tamassia and Tollis, 1986]. At the time of developing this proof the author of the thesis was not aware of the work of Tamassia and Tollis [1986] whose results are quite similar to those developed in this proof. The transformation below is presented as originally developed because it guarantees that the 'stick diagram' developed by the transformation is in the right form for the second part of the proof. This comment is expanded upon after the transformation process has been presented.

In the second step of the transformation from a planar graph to a configuration of adjacent rectangles, the stick diagram is represented as a collection of adjacent rectangles and horizontal axial lines crossing all of the adjacencies in the collection of rectangles. These axial lines will be of two types "essential lines" which are the only lines to cross a particular adjacency and "choice lines" where a number of lines (none of which are essential) cross some adjacency. Not all of the choice lines are necessary to cross all of the adjacencies in the collection of rectangles. If it is possible to determine in polynomial time a minimal subset of choice lines to cross all of the adjacencies not crossed by essential lines in the diagram then it is possible to solve ***planar vertex cover*** in polynomial time – finding the minimum set of choice axial lines is equivalent to finding the minimum vertex cover of the original graph.

Proving that ***ALP-OLOR*** is NP-Complete is accomplished by means of two theorems – 4.4.1 that shows that ***stick diagram*** is NP-Complete using a transformation from ***planar vertex cover*** and 4.4.2 that shows that ***ALP-OLOR*** is NP-Complete using a transformation from ***stick diagram***.

It is, however, easier to perform the transformation from a planar graph to a stick diagram for a somewhat more restricted form of planar graph – a biconnected planar graph has properties which can be used in the transformation. Therefore it is desirable to prove one other result – vertex cover for a biconnected planar graph is NP-complete. Once it has been shown that this result holds the transformation from ***biconnected planar vertex cover*** to ***stick diagram*** and hence to ***ALP-OLOR*** can be done more easily. This result is addressed in Lemma 4.4.1. The construction presented here is an improvement of a construction first published in Sanders *et al.* [1995] and Sanders *et al.* [1997]. This version of the construction first appeared in Sanders *et al.* [1999] and is similar to that developed by Biedl *et al.* [1997] in proving that vertex cover in cubic triconnected planar graphs is NP-Hard.

biconnected planar vertex cover
Instance: Biconnected planar graph $G = (V, E)$, positive integer $B \leq \mid V \mid$.

```
$G' \leftarrow G$
$j \leftarrow 1$
WHILE there exists a cut vertex $v_j$ in $G'$
    Let $X_j$ and $Y_j$ be any two components created by removing
    $v_j$ from $G'$
    Let $x_j \in X_j$, $y_j \in Y_j$ be such that $x_j$, $y_j$ and $v_j$ lie on a
    face
    Add a triangle graph, $T_j$, to the face in $G'$ containing
    $x_j$, $y_j$ and $v_j$
    Add the edges ($a_j$,$x_j$) and ($b_j$,$y_j$) to $G'$
    $j \leftarrow j + 1$
```

Figure 4.3: Creating a biconnected planar graph

Question: Is there a vertex cover of size B or less for G, i.e. a subset $V' \subseteq V$ with $|V'| \leq B$ such that for each edge $\{u, v\} \in E$ at least one of u and v belongs to V'?

Lemma 4.4.1 ***biconnected planar vertex cover*** *is NP-Complete*

Proof

Clearly ***biconnected planar vertex cover*** is in NP. Given a set of vertices V' such that $|V'| \leq B$ it is possible to check in polynomial time that $|V'|$ is a vertex cover.

Now transform ***planar vertex cover*** to ***biconnected planar vertex cover***. Given a planar graph $G(E, V)$, G can be converted to a biconnected planar graph G' using the algorithm given in Figure 4.3. This is accomplished by appropriate addition of instantiations of "triangle graphs" T_j where each T_j is defined as the vertices a_j, b_j and c_j and the edges (a_j, b_j), (b_j, c_j) and (c_j, a_j). Clearly G' is a biconnected planar graph since

1. after each iteration of the algorithm the vertex v_j is no longer a cut vertex with respect to X_j and Y_j and
2. no new cut vertex is ever added during an iteration of the algorithm.

Thus the algorithm terminates with G' free of all cut vertices. In addition, the triangle graphs T_j that are added during each iteration are added to the face containing the vertices to which they are connected thus maintaining the planarity of the graph. Refer to Figure 4.4 for an example of a triangle graph and to Figure 4.5 for an example of how triangle graphs can be added to a planar graph using the algorithm in Figure 4.3 in order to derive a biconnected planar graph.

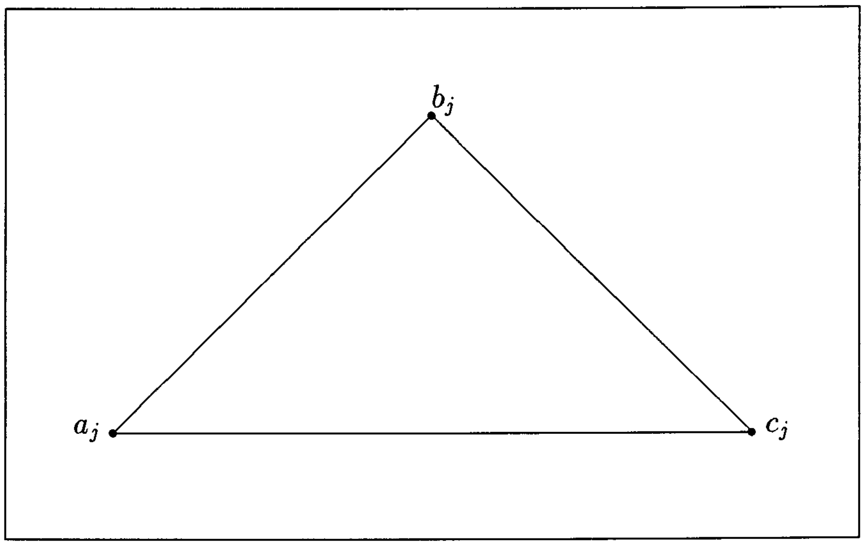

Figure 4.4: An example of a "triangle graph", T_j

To determine the vertex cover for G', the original graph G plus the new edges and vertices must be considered. The structure of the triangle graphs means that for each triangle graph, T_j, exactly two of a_j, b_j and c_j must be in the vertex cover of G'. If k triangle graphs are added then it is trivial to show that ***planar vertex cover*** for graph G with cover size K is true if and only if ***biconnected planar vertex cover*** is true for G' with cover size $B = K + 2k$.

The transformation from G to G' can clearly be accomplished in polynomial time. A cut vertex (articulation point) can be found in polynomial time [Brassard and Bratley, 1996] and there are at most O(n) cuts (n is the number of vertices in G').

Therefore ***biconnected planar vertex cover*** is NP-Complete.

□

biconnected planar vertex cover can now be used to show that ***ALP-OLOR*** is NP-Complete. The proof is accomplished by means of the following two theorems – Theorem 4.4.1 and Theorem 4.4.2.

Theorem 4.4.1 ***stick diagram*** *is NP-Complete.*

Proof

Clearly ***stick diagram*** is in NP – given a set of horizontal lines H' such that $|H'| \leq S$, it is possible to check in polynomial time that every vertical line in U is cut at least once.

Now transform ***biconnected planar vertex cover*** to ***stick diagram***. If $G(V, E)$ is a biconnected planar graph then G can be embedded in the plane (G is planar) and

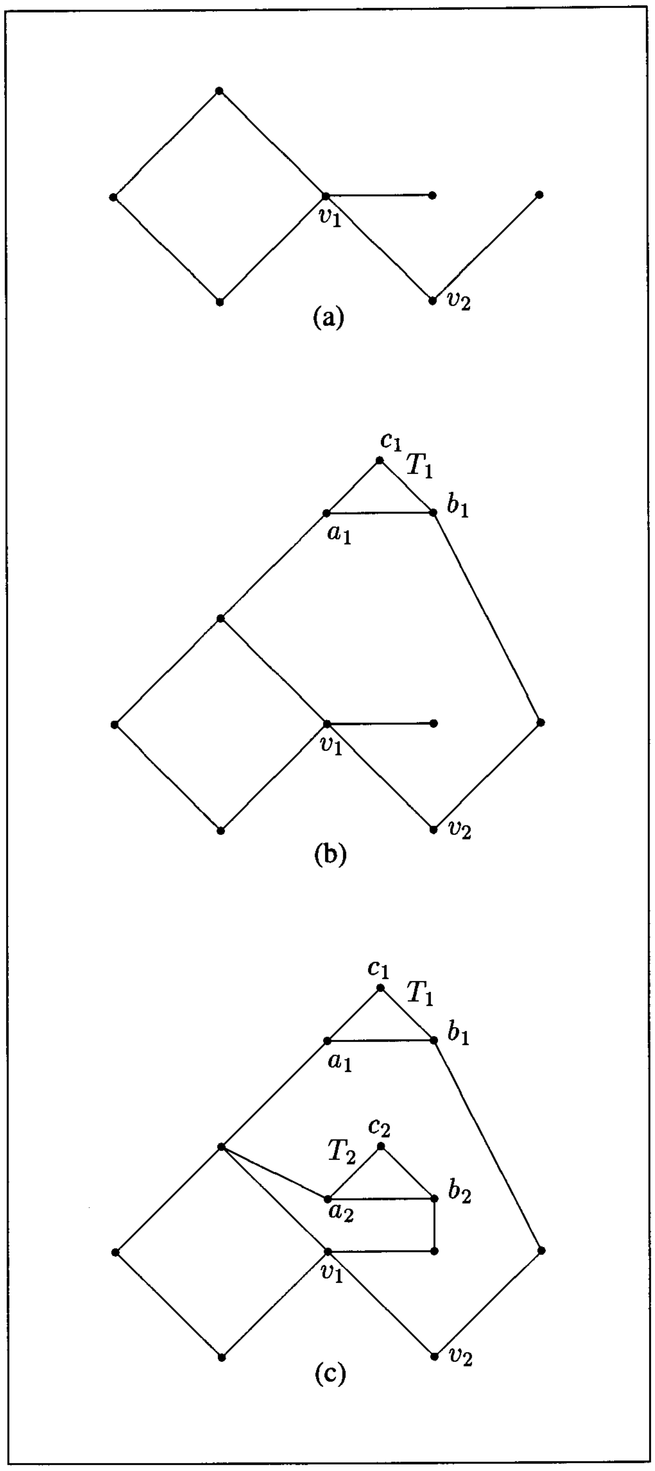

Figure 4.5: An example of adding triangle graphs to a graph to make it biconnected [(a) The original graph, v_1 and v_2 are cut vertices. (b) Graph with T_1 added, v_1 is still a cut vertex. (c) Final biconnected graph]

for every two vertices in G an elementary cycle can be found which contains these vertices (G is biconnected) [Berge, 1962; Harary, 1969]. This implies that there are no vertices of degree 1 in G (other than the case where G is a trivial graph with 2 vertices and 1 edge) and thus that all the faces in G are bounded by cycles [Harary, 1969].

Let F_0 be the exterior face of G and $F_1, \ldots, F_n$ be the interior faces of the graph. The biconnected planar graph G can be transformed to a stick diagram by the following process.

1. Choose C_s as being the cycle bounding any face, F_s of G, that is adjacent to the exterior face F_0 of G.

2. Choose any two vertices x and y joined by an edge A that form part of C_s and are adjacent to the exterior. Represent x and y by horizontal lines in the stick diagram that cut the vertical line representing edge A (see Figure 4.6 (a)).

3. Consider the path from x to y, B, formed by removing edge A from C_s. For the moment, treat B as if it were a simple edge i.e. insert its corresponding vertical line into the stick diagram. This then gives the horizontal lines x and y cutting the vertical lines A and B. The stick diagram is then as shown in Figure 4.6 (b).

4. Break the path B (which was treated as a virtual edge) into its component edges. Let the path B be the sequence of vertices $x, v_0, v_1, \ldots, v_k, y$. On the path B from x to y whenever a vertex v_i is encountered a new horizontal line must be added to the stick diagram. A new vertical line must also be added for each edge encountered. This is done in the following way. Suppose C is the edge joining x to v_0 along the path B. The stick diagram is now altered to include a vertical line for edge C and a horizontal line for vertex v_0. This is shown in Figure 4.6(c).

   In this case, B' represents the original path B minus the edge C which has been included in the stick diagram. A similar operation is applied for all the edges on the path B. Each vertex v_i maps to a horizontal line in the stick diagram and the edge joining it to the previous vertex is a vertical line cut by the two horizontal lines v_{i-1} and v_i. After all the vertices on the path B have been visited, the stick diagram will have the form shown in Figure 4.6 (d). A stick diagram which represents the originally selected closed region F_s of the original graph has now been created.

5. If G only had one interior face then the transformation is complete, otherwise continue with the next step.

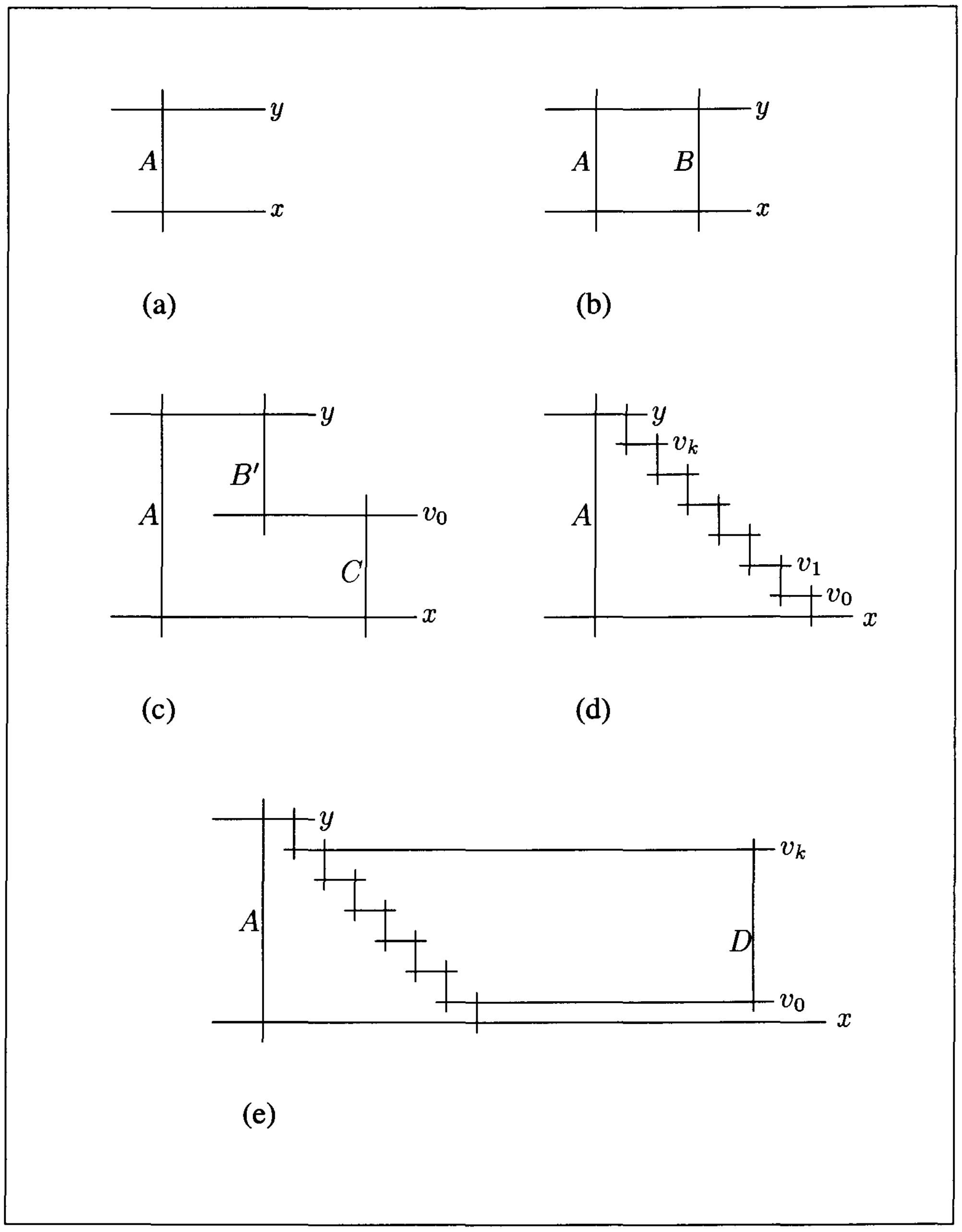

Figure 4.6: Creating a "stick" diagram

6. Let CF (which is used later in the process to store the composite of all the faces considered so far) be F_s. At this stage it will be the first face considered. That is $CF = F_s$ is the face made up of the vertices $x, y, v_0, \dots v_k$ and the edges connecting these vertices

7. While there are still faces in G to consider, repeat the following

   (a) Choose a face $F_m, 1 \leq m \leq n$ that is adjacent to CF. F_m must share a path with CF. C_m, the cycle enclosing F_m is thus made up of two sets of vertices – those that are on the shared path and have already been "visited" (included in the stick diagram) and those that have not yet been visited. The vertices in the latter set make up the path D.

   (b) Treat path D as a single (simple) edge and add it to the stick diagram by extending the horizontal lines representing the start vertex and the end vertex of the shared path to cut a new vertical line representing the path D.

   See Figure 4.6 (e) for an example – in this figure, a new path between v_0 and v_k is being added.

   The path D can then be broken up into its constituent edges in the same fashion as before.

   (c) Grow CF by combining it with F_m and removing the shared path between the faces. In the stage of the process as shown in Figure 4.6 (e), CF would consist of the vertex x, the edge (x, v_0) connecting x to the first vertex in the path D, all of the vertices and edges on the path D, the edge (v_k, y) connecting the last vertex in the path D to y, the vertex y and the edge (y, x). The path from v_0 to v_k through the vertices v_1 to v_{k-1} would have been removed.

This completes the construction of the stick diagram from a biconnected planar graph. A complete example of this is shown in Figure 4.7. First the face represented by $x - y - z - w$ is converted into a stick diagram. Then the face represented by $w - z - p$ is added to the stick diagram and finally the face represented by $w - p - z - y - q$ is added. This gives the complete stick diagram for the original biconnected graph.

Thus it can be seen that if a vertex cover, V', can be found for G then a set of horizontal lines, H', can be found for H – each vertex in G is a horizontal line in H and each edge in G is a vertical line in U. Conversely if a set of horizontal lines H', such that $|H'| \leq S$, could be found to cut each vertical line in U, then a vertex cover, V', for G could be found.

The transformation from ***biconnected planar vertex cover*** to ***stick diagram*** can be accomplished in polynomial time. Each face in the graph G is considered in turn

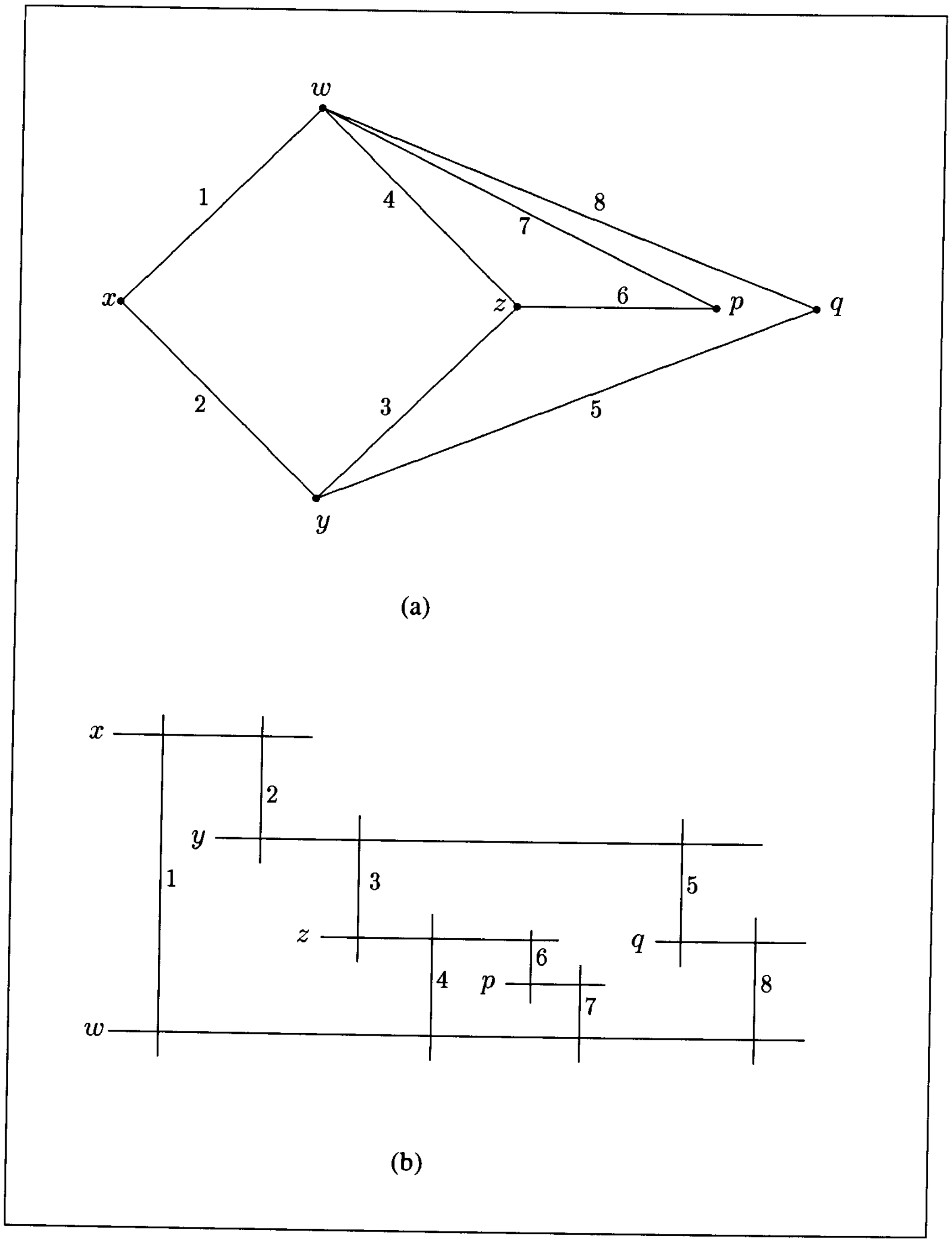

Figure 4.7: An example of the transformation of a biconnected planar graph to a stick diagram

and once only. As the face is considered each edge is added in turn to the stick diagram as a vertical line – this happens once per edge. Horizontal lines are either added to the stick diagram to represent vertices or the horizontal line representing a vertex is extended as necessary. Each vertex can only occur in as many faces as there are in the graph and each vertex in each cycle is only visited once per cycle. Thus the number of operations on vertices is limited by a polynomial expression.

Therefore ***stick diagram*** is NP-Complete.

□

As mentioned earlier Tamassia and Tollis [1986] presented an approach for transforming a planar graph into a configuration of horizontal and vertical line segments – vertices are mapped to horizontal segments and edges to vertical segments. They call this a *visibility representation* of the planar graph where two parallel segments of a set are *visible* if they can be joined by a segment orthogonal to them. They also define *weak-visibility representation* or *w-visibility representation* as a representation where vertices are represented by horizontal segments and edges by vertical segments having only points in common with the pair of horizontal segments representing the vertices they connect. The transformation above produces a representation which is a *w-visibility representation* of the planar graph with the additional property that a left-to-right ordering is imposed on the edges as they are encountered in the faces of the original planar graph (see Figure 4.7). This ordering is important in the second transformation – converting a stick diagram into a collection of adjacent rectangles.

Theorem 4.4.1 shows that a stick diagram can be constructed for any biconnected planar graph. It is now necessary to show that any stick diagram can be represented by a collection of adjacent rectangles whose adjacencies are crossed by essential and choice axial lines. This must be done in a manner that ensures consistency between a minimal selection of choice axial lines crossing rectangle adjacencies and a minimal selection of horizontal lines in the stick diagram. This is considered in Theorem 4.4.2 below. This theorem uses a construction from a stick diagram to produce a collection of adjacent rectangles in which the adjacencies between rectangles are crossed by essential axial lines and choice axial lines. The choice axial lines are directly related to the horizontal lines in the stick diagram. Not all of the choice axial lines are necessary and Theorem 4.4.2 also shows that the problem of choosing the minimum number of such choice axial lines (solving ***ALP-OLOR***) is NP-Complete.

Theorem 4.4.2 ***ALP-OLOR*** *is NP-Complete*

Proof

Clearly ***ALP-OLOR*** is in NP. Given a set of axial lines it is possible to check in polynomial time that each adjacency has been crossed by at least one axial line.

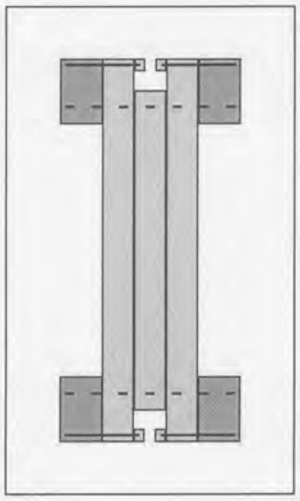

Figure 4.8: The Canonical Unit which produces two choice axial lines (shown as dashed lines)

Now transform ***stick diagram*** to ***ALP-OLOR***.

The transformation from a stick diagram to a collection of adjacent rectangles is done using the canonical choice unit shown in Figure 4.8. In this canonical choice unit, *ccu*, the essential axial lines (shown as solid lines) originating in the four small rectangles and ending in the four darker shaded rectangles are enough to cross all of the adjacencies in the ccu except those between the middle rectangle and the two tall rectangles bordering it. These adjacencies can be crossed by two possible maximal axial lines (shown as dashed lines), only one of which is necessary. Scaling of the canonical choice unit does not change the fact that it always produces these choice axial lines.

This transformation proceeds by replacing each vertical line in the stick diagram by a ccu of an appropriate size. The horizontal lines that crossed through the vertical line are represented by the choice axial lines of the ccu. It is necessary to show that these canonical choice units can be joined together in a fashion which maintains the relation between the horizontal lines in the stick diagram and the choice axial lines in the configuration of adjacent rectangles. There are four ways in which horizontal lines could cut through successive vertical lines or in which choice axial lines could cross successive ccu's (actually there are only two ways, each with a vertical reflection). These are

1. the horizontal line could be the upper (lower) line through one vertical line and the upper (lower) line through the next vertical line (the upper (lower) choice axial line of one ccu is the same as the upper (lower) choice axial line of the next ccu),

2. the horizontal line could be the upper (lower) line through one vertical line

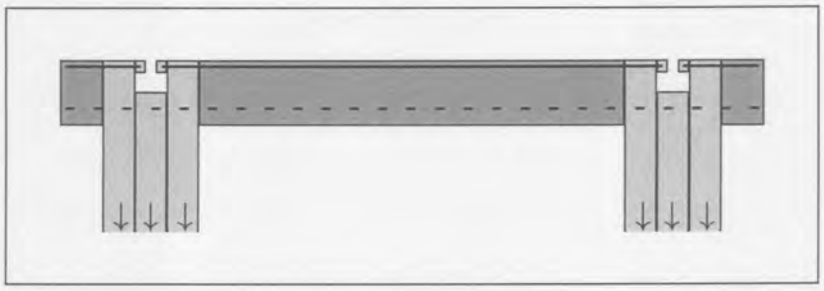

Figure 4.9: Joining the upper choice axial lines of two choice units

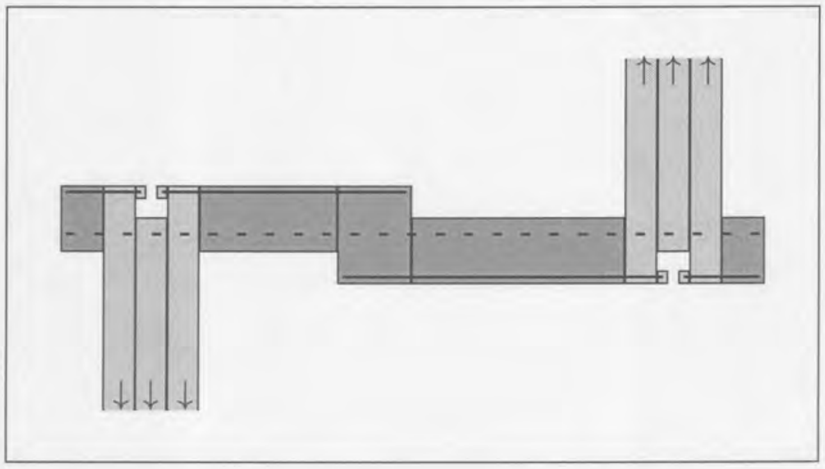

Figure 4.10: Joining the upper choice axial line of one unit to the lower choice axial line of the next unit

and the lower (upper) line through the next vertical line (the upper (lower) choice axial line of one ccu is the same as the lower (upper) choice axial line of the next ccu).

In each of these cases it is possible to connect two ccu's in such a fashion that the relation between lines in the stick diagram and choice axial lines is preserved. This is accomplished by making use of the darker shaded "connector" rectangles of each ccu (see Figure 4.8) and where appropriate making use of "connecting" rectangles. Figure 4.9 shows the construction for case 1 and its reflection. Figure 4.10 shows the construction for case 2 and its reflection.

If all of the vertical lines in the stick diagram are replaced by ccu's, connecting rectangles are added if appropriate and the appropriate changes are made to the connector rectangles then the choices in the original stick diagram can be maintained. An example of converting a stick diagram to a collection of adjacent rectangles is shown in Figure 4.11.

The transformation from ***stick diagram*** to ***ALP-OLOR*** is thus accomplished by inserting an appropriately sized ccu for each vertical line and then joining these up by using the appropriate connecting rectangles working from the leftmost to the rightmost ccu, at each stage connecting the current ccu to those that have already been visited.

This transformation can clearly be done in polynomial time – each vertical line is visited twice, once when it is replaced by a ccu and a second time when it is connected to the ccu(s) to its right in the stick diagram. If the stick diagram can be drawn then a configuration of ccu's can be drawn by scaling the ccu's to be the same size as the vertical lines that they represent. The ccu's (and their connecting rectangles) can thus be drawn as a non overlapping collection of adjacent rectangles – an instance of ***ALP-OLOR***.

It is now necessary to show that it is possible to find a set of horizontal lines H' such that $|H'| \leq S$ for ***stick diagram*** if and only if it is possible to find a set of axial lines axial lines P such that $|P| \leq O$ for ***ALP-OLOR***. Suppose there exists a set of lines H' such that $|H'| \leq S$ for ***stick diagram***. The construction of the collection of adjacent rectangles from the stick diagram changes the horizontal lines in the stick diagram to choice axial lines in the collection of rectangles. It also introduces 4 essential axial lines for every ccu added. These essential axial lines must be in the final solution to ***ALP-OLOR***. There must thus be a solution P ($|P| \leq O$) to ***ALP-OLOR*** with $|P| = |H'| + 4|U| - p$ where p is the number of shared essential axial lines (see Figure 4.9 for an example of how essential axial line sharing can occur). This is because the essential axial lines must be in P and the choice axial lines which correspond to the horizontal lines in H' must also be in P. Conversely if there is a solution P to ***ALP-OLOR*** then there must be a solution $H' = P - \{e \mid e$ is an essential axial line in $P\}$ to ***stick diagram***.

ALP-OLOR is thus NP-Complete.

□

This section shows that the problem of crossing all the adjacencies of a collection of adjacent orthogonal rectangles with the minimum number of maximal-length axial lines is NP-Complete in general. The next section of this chapter (Section 4.5) presents an algorithm that finds an approximate solution to the problem. The next section also includes a discussion of some empirical testing of the heuristic. The following section (Section 4.7) discusses some special cases in which an exact solution to the problem can be found in polynomial time.

4.5 Heuristic Algorithm

Section 4.4 above shows that the problem of crossing all the adjacencies of a collection of adjacent orthogonal rectangles with the minimum number of maximal-length

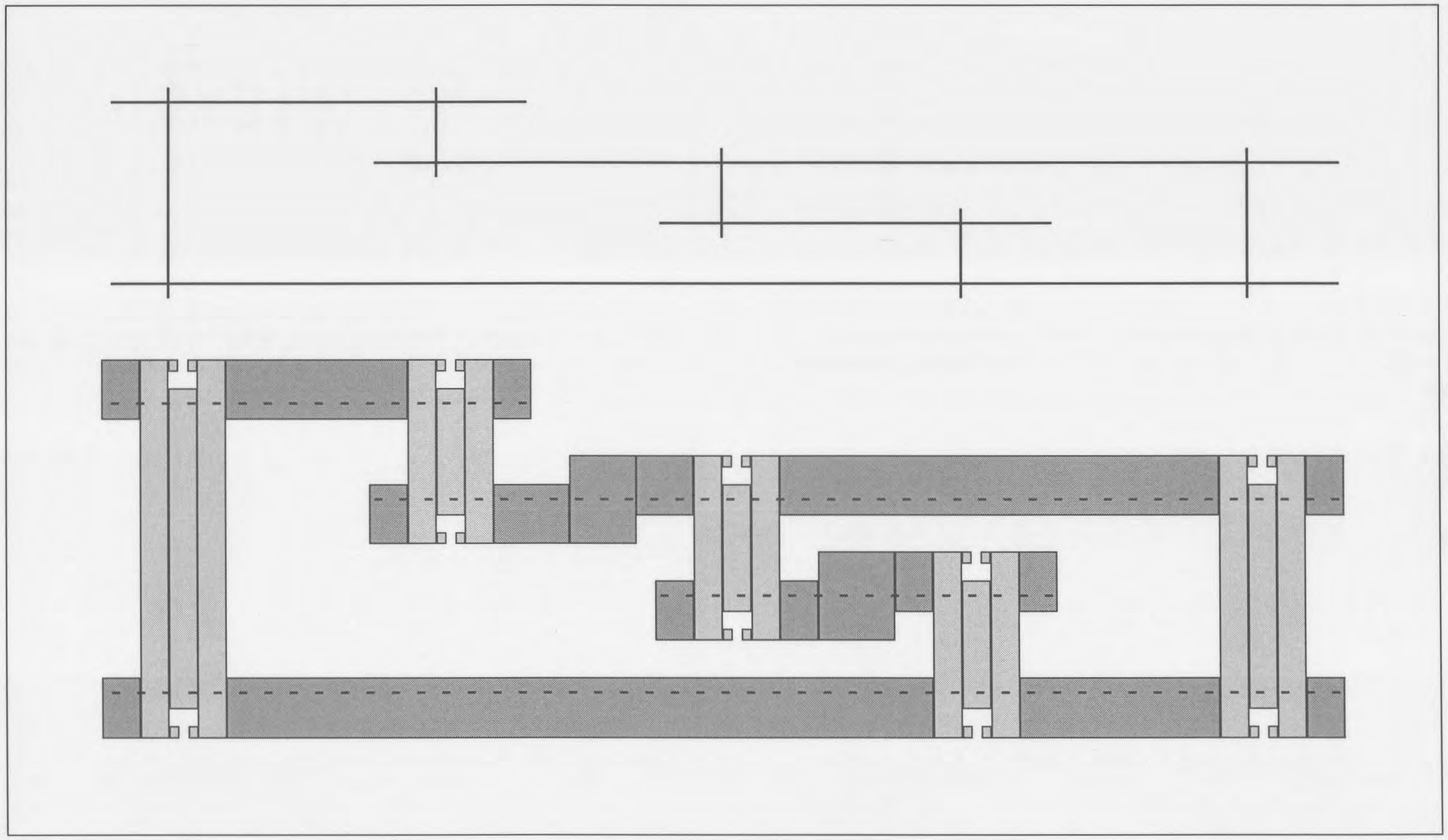

Figure 4.11: An example of converting a stick diagram to a collection of adjacent rectangles

axial lines is NP-Complete in general. A heuristic algorithm to find a non-redundant set of orthogonal axial lines to cross all the adjacencies (a set of lines such that none can be removed without leaving an adjacency uncrossed) is presented in this section. The algorithm has two phases. First the adjacencies among the rectangles are determined (section 4.5.1), and then the axial lines crossing all of the adjacencies are determined (section 4.5.2).

It should be noted that if two rectangles are adjacent then there is an infinite number of orthogonal line segments that could be placed to cross that adjacency. These line segments are all equivalent in the sense that they cross that particular adjacency. Thus in this phase of the research an axial line is defined by a range of y-values through which a line segment parallel to the x-axis could be drawn. The convention used here is that an axial line crossing a given adjacency would be defined by the y-value range of that adjacency and the two rectangles involved. An axial line crossing the adjacencies between a number of rectangles would be given by the common y-value range of the adjacencies between the rectangles and a list of the rectangles concerned.

4.5.1 Determining the adjacencies between the rectangles

In determining the adjacencies between the orthogonal rectangles, horizontal and vertical adjacencies are treated as separate cases. Only the case of vertical adjacencies (and horizontal lines) will be discussed here. Horizontal adjacencies can be treated analogously. An algorithm to determine the adjacencies in a configuration of adjacent orthogonal rectangles is given in Figure 4.12 and discussed below.

Any rectangle R can be defined by the coordinates of its bottom-left and top-right corners. The algorithm thus requires a data structure which contains the rectangle number and these coordinates as well as the ability to keep track of other information calculated in the algorithm. This other information is the number of rectangles which are adjacent to the right hand side of the rectangle being considered and a list of these rectangles. These lists of adjacencies (one list per rectangle) are in fact the required output from this algorithm and are used later in determining the axial lines that must be placed to cross all of the adjacencies. In the algorithm to determine the adjacencies between the rectangles, an array of records is the data structure used. Thus each element of the array, $Rect[i]$, is a record defining a particular rectangle of the configuration and has fields $Rect[i].left$, $Rect[i].bottom$, $Rect[i].right$, $Rect[i].top$, $Rect[i].numadj$ and $Rect[i].adjlist$ to define the bottom-left and top-right corners of the rectangle and to maintain a list of the rectangles which are adjacent to this rectangle on the right.

Left and *Right* are essentially copies of the array *Rect*, but are sorted based on the coordinates of the left and right edges of the rectangles respectively (lines 09 to 12 of the algorithm). These arrays are used to implement a “line sweep” strategy

```
{Get input}
create array $Rect[\ ]$ for the rectangles
{$n$ is the number of rectangles}
FOR $i$ from 1 to $n$
   Input $Rect[i].number$
   Input $Rect[i].left$
   Input $Rect[i].right$
   Input $Rect[i].top$
   Input $Rect[i].bottom$
   Set $Rect[i].numadj$ to be 0 {will be calculated}
   Set $Rect[i].adjlist$ to be Nil {will be calculated}
Create an array $Left[\ ]$ of all the rectangles {a copy of $Rect[\ ]$}
Sort $Left[\ ]$ in ascending order of $Left[i].left$
   {break ties based on increasing $Left[i].bottom$}
Create a list $Right[\ ]$ of all the rectangles {a copy of $Rect[\ ]$}
Sort $Right[\ ]$ in ascending order of the $Right[j].right$
   {break ties based on increasing $Right[j].bottom$}
Set $i \leftarrow j \leftarrow 1$
WHILE $i <= n$ AND $j <= n$
   CASE
      $Left[i].left < Right[j].right$
         increment $i$
      $Left[i].left > Right[j].right$
         increment $j$
      $Left[i].left = Right[j].right$
         WHILE $Left[i].top <= Right[j].bottom$
         AND $Left[i].left = Right[j].right$
            increment $i$
         IF $Left[i].left = Right[j].right$
            THEN
               IF $Left[i].bottom < Right[j].top$
                  THEN
                     add $Right[j].number$ to $Left[i].adjlist$
                     $Left[i].numadj \leftarrow Left[i].numadj + 1$
               IF $Left[i].top <= Right[j].top$
                  THEN
                     increment $i$
                  ELSE
                     increment $j$
```

Figure 4.12: The algorithm for determining the adjacencies between the rectangles

[Manber, 1988; Cormen *et al.*, 1990] for determining which rectangles are adjacent to which other rectangles.

The line sweep (lines 13 to 33) works by comparing the coordinates of the right edges (using *Right*) with the coordinates of the left edges of the other rectangles (using *Left*). Three cases can arise:

1. The left edge of the rectangle being considered in list *Left* is to the left of the right edge of the rectangle being considered in *Right* (lines 16 and 17), in this case these two rectangles cannot be adjacent and so the next rectangle in *Left* must be considered for a potential adjacency. This rectangle's left edge must be further to the right so there is potential for an adjacency to occur.

2. The left edge of the rectangle being considered in list *Left* is to the right of the right edge of the rectangle being considered in *Right* (lines 18 and 19). Again these two rectangles cannot be adjacent and so the next rectangle in *Right* must be considered for a potential adjacency.

3. The left edge of the rectangle being considered in list *Left* and the right edge of the rectangle being considered in *Right* have the same x-coordinate (lines 20 to 33). Thus these two rectangles could be adjacent and it is necessary to determine if they do in fact share a range of y values. This is done by traversing the list *Left* until a rectangle with a top y-value greater than the bottom of the current rectangle from *Right* is found or until the rectangles being considered have different x-values and thus cannot be adjacent (lines 21 and 22).

   In the former case more work needs to be done to test for the adjacency (lines 23 to 33).

   In the latter case no more work will be done in this pass through the loop and the outer `WHILE` will be continued.

   In lines 23 to 33, a test is made to see if the bottom of the left rectangle is below the top of the right rectangle (line 25). If it is, then the right rectangle is adjacent to the left rectangle and this information must be recorded (lines 27 and 28).

   Lines 30 to 33 are then used to determine whether to move along in list *Left* or list *Right*.

The configuration of adjacent rectangles in Figure 4.13 illustrates the working of the algorithm. After input and sorting, the rectangles in the list *Left* would be $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ and the rectangles in the corresponding list *Right* would be $\{1, 2, 3, 5, 7, 6, 8, 4, 9, 10, 11\}$. The `WHILE` loop would begin with $i = 1$ and $j = 1$ and thus would be comparing *Left*[*1*].*left* and *Right*[*1*].*right*. Here

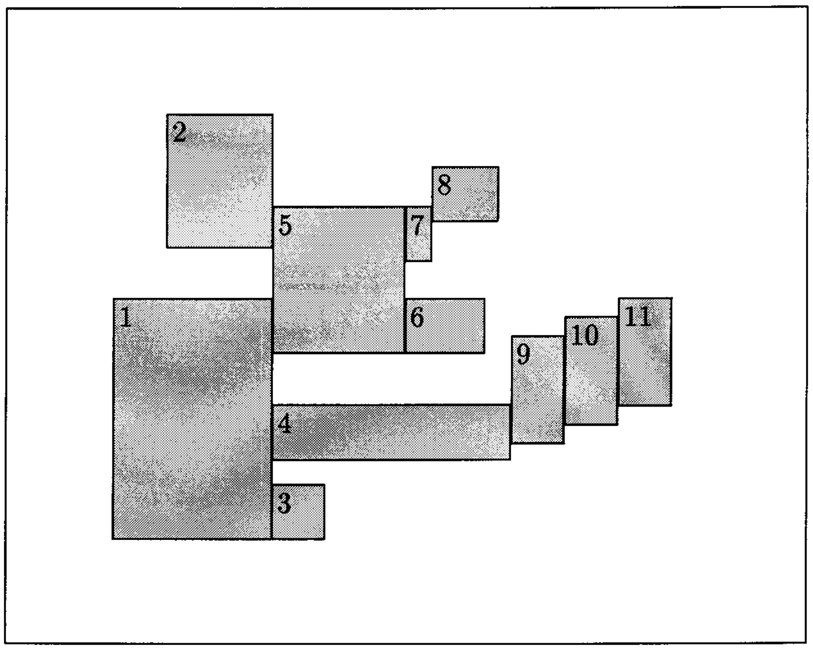

Figure 4.13: A configuration of adjacent orthogonal rectangles

case 1 above holds, the left x-coordinate of rectangle 1 is clearly less than the right x-coordinate of the same rectangle, and i would be incremented. *Left[2].left* and *Right[1].right* would then be compared (the left x-coordinate of rectangle 2 is clearly less than the right x-coordinate of rectangle 1) and again i would again be incremented. Next *Left[3].left* and *Right[1].right* would be compared. Here the left x-coordinate of rectangle 3 is equal to the right x-coordinate of rectangle 1 and case 3 would be executed. The `WHILE` loop in line 21 would not be executed (*Left[3].top* is greater than *Right[1].bottom*) and line 23 would be executed next. *Left[3].left* = *Right[1].right* and so line 25 would be executed. Here *Left[3].bottom* < *Right[1].top* which means that the two rectangles are adjacent and the adjacency list for rectangle 1 must be updated. Lines 30 to 33 determine whether to increment i or j, in this case there is another rectangle, 4, which has the same left x-coordinate as rectangle 3 and in order to consider it i must be incremented to traverse the list *Left*. The algorithm performs in a similar fashion on the remainder of the two lists.

The sorting phase of this portion of the algorithm is clearly $O(n \lg n)$, where n is the number of rectangles. Determining the adjacencies from the sorted lists is $O(n)$ as each list is simply traversed from beginning to end with no backtracking being necessary. The whole algorithm is then $O(n \lg n)$.

4.5.2 Determining the axial lines

The next step in the process is to determine a non-redundant set of axial lines to cross all of the adjacencies in a configuration of adjacent orthogonal rectangles. The algorithm which generates this non-redundant set uses a number of functions. These functions are given in Figure 4.14. The functions are also used by other algorithms discussed in this chapter. The algorithm itself is given in Figures 4.15, 4.16, 4.17, 4.19 and 4.20. The algorithm generates all the possible orthogonal axial lines which cross the adjacencies between rectangles (Figures 4.15, 4.16), determines which lines are essential (are the only lines which cross a particular adjacency) (Figures 4.17), removes any lines which only cross adjacencies crossed by the essential lines (redundant lines) (Figures 4.19) and then resolves the choice conflict (Figures 4.20). The resolving of the choice is done by repeatedly choosing the choice line which crosses the highest number of previously uncrossed adjacencies. In this algorithm, lines are represented as a list of rectangles. For example, the line $1, 2, 3$ means a line which crosses the adjacency between rectangles 1 and 2 and the adjacency between rectangles 2 and 3. Any line, say l, in this algorithm will be represented a data structure with four fields – $l.line$ which is a list of rectangle numbers, $l.bottom$ which is the bottom y-coordinate of the range of y-coordinates that define the line (the bottom of the common y-coordinate range of the adjacencies between the rectangles) and $l.top$ which is the top y-coordinate of the range of y-coordinates that define the line. The fourth field $l.extended$ is used in the algorithm to record whether a line has been extended.

The algorithm uses the adjacency lists for each rectangle created above (Figure 4.12) and visits the rectangles based on the order in the list *Left* used in Figure 4.12. The algorithm also uses a modified adjacency matrix A. Here $A[i, j]$ indicates whether adjacency $i|j$ (where j is any rectangle which is adjacent to i on the right) exists in the configuration of rectangles. $A[i, j]$ also keeps the track of the first line to cross the adjacency (useful for identifying essential lines), the number of lines crossing the adjacency and also a list of these lines. In addition, a set L, the set of candidate lines at any stage of the execution of the algorithm, must be maintained. A set T of temporary candidate lines is created when lines are extended backwards. The algorithm generates as output a set E of non-redundant orthogonal lines which cover all of the adjacencies between rectangles.

The first phase of the algorithm starts by setting the list of candidate lines L to be empty (line 34 of the algorithm in Figure 4.15 and Figure 4.16). The algorithm then generates every possible longest line which crosses the adjacencies between the adjacent rectangles (lines 35 to 75).

This is done by considering each rectangle, *Rect*$[r]$, in turn from the leftmost to the rightmost (in the order defined by the list *Left*$[\,]$ – see line 35) to find the axial lines which cross the adjacencies between that rectangle and each of its right

|| is used to denote concatenation of two strings or lists

minimum and *maximum* are functions which return the smaller of two numbers and the greater of two numbers respectively

interval(a, b) defines a range of y values between the two arguments a and b

overlap(*interval1*, *interval2*) returns true if the two intervals have an overlapping range of y-values, and returns false otherwise

Figure 4.14: Functions used in the algorithms in this chapter

```
Set $L$ to be empty {$L$ is the set of all possible lines}
FOR each rectangle $i$ in $Left[\ ]$
    $p \leftarrow Left[i].number$
    find $r$ such that $Rect[r].number = p$
    FOR each rectangle $j$ in $Left[i].adjlist$
        $q \leftarrow Right[j].number$
        find $s$ such that $Rect[s].number = q$
        $extended \leftarrow$ false
        FOR each line $l$ in $L$
            IF ($overlap(interval(l.low, l.high),$
            $interval(Rect[s].bottom, Rect[s].top))$ = true) AND
            (rightmost rectangle in $l.line$ = $Rect[r].number$
                THEN
                    $l.extended \leftarrow$ true
                    $lnew.line \leftarrow l.line$ || $Rect[s].number$
                    $lnew.low \leftarrow maximum(l.low, Rect[s].bottom)$
                    $lnew.high \leftarrow minimum(l.high, Rect[s].top)$
                    Add $lnew$ to $L$
                    $extended \leftarrow$ true
```

Figure 4.15: Determining all possible orthogonal axial lines – Phase 1 part a

```
51          IF $extended = $ false
            {no candidate line $l$ can be extended into $Rect[s]$}
52             THEN
53                Set $T$ to be empty
                  {$T$ is the set of lines which can be extended
                   backwards, the longest such line will be chosen}
54                Set $t.line$ to be $Rect[r].number \| Rect[s].number$
55                Set $t.low \leftarrow maximum(Rect[r].bottom, Rect[s].bottom)$
56                Set $t.high \leftarrow minimum(Rect[r].top, Rect[s].top)$
57                Add $t$ to $T$
58                For each line $k$ in $L$ which ends in $Rect[r]$
59                   Set $u$ to be equal to $t$
60                   Set $extending$ to true
61                   Set $m$ to point to the second last rectangle in line $k$
62                   WHILE $extending$ AND $m > 1$
63                      find $d$ such that $Rect[d].number = m$th
                        rectangle in line $k$
64                      IF $overlap(interval(u.low, u.high),$
                        $interval(Rect[d].bottom, Rect[d].top))$ = true
65                         THEN
66                            Set $u.line$ to be $Rect[d].number \| u.line$
67                            Set $u.low \leftarrow maximum(Rect[d].bottom, u.low)$
68                            Set $u.high \leftarrow minimum(Rect[d].top, u.high)$
69                            Decrement $m$
70                         ELSE
71                            Set $extending$ to false
72                   Add $u$ to $T$
73                Add the longest line, $long$ in $T$ to $L$
74    FOR each $l$ in $L$
75       IF $l.extended =$ true THEN remove $l$ from $L$
```

Figure 4.16: Determining all possible orthogonal axial lines – Phase 1 part b

neighbours, $Rect[s]$, in turn (see line 38). The algorithm determines whether each of the lines coming into $Rect[r]$ from the left can be extended to cross the adjacency between $Rect[r]$ and $Rect[s]$. This is done by testing if there is an overlap of the y-coordinate range of the adjacency and the line being considered and that the line being considered crosses some adjacency into $Rect[r]$ (line 43). Extending the line (lines 44 to 50) involves appending the rectangle number, $Rect[s].number$, to the line and recalculating the y-coordinate range of the line. Figure 4.13 can be used as an example of how lines can be extended. Suppose that the rectangle being considered is the rectangle labeled 5. This rectangle will have two lines coming into it from the left, the line 1–5 and the line 2–5. It also has two right neighbours 6 and 7. The algorithm will attempt to extend each of these lines into each of these neighbours. The result after considering both neighbours will be to have two new lines in L, 1–5–6 and 2–5–7 with the appropriate ranges of y-coordinates.

If no line can be extended into $Rect[s]$ (line 51) then a new line must be started in order to cross the adjacency between $Rect[r]$ and $Rect[s]$. To make sure that this new line is as long as possible it must also be extended as far to the left (backwards) as possible (lines 51 to 73). This extending backwards of the new line is accomplished by considering all lines which cross into $Rect[r]$ in turn (line 58) and looking at these lines an adjacency at a time (lines 58 to 72) to see whether the new line could cross some of the adjacencies crossed by the line being considered. In this fashion a new line is potentially created for each incoming line. Only the longest line which crosses the adjacency under consideration (between $Rect[r]$ and $Rect[s]$) is chosen (line 73). An example of where no line can be extended can be seen in Figure 4.13 when rectangle 10 is being considered. It has the line 1–4–9–10 coming into it from the left and has rectangle 11 as its right neighbour. The line cannot be extended to cross the adjacency between 10 and 11 so a new line must be created to cross this adjacency. This new line must then be extended backwards to cross the adjacency between 9 and 10. If this was not done then the new line crossing the adjacency between 10 and 11 would not be as long as possible. After this phase of extending backwards the line 9–10–11 is added to L.

The last two lines of the algorithm (lines 74 and 75) remove any lines that have been extended forwards during the calculation of the set of lines as these lines are no longer necessary – every adjacency crossed by one of these lines is also crossed by at least one longer line that was generated when the line was extended.

Figure 4.17 gives the algorithm for finding the essential lines from the set of lines calculated by the algorithm in Figures 4.15 and 4.16. This algorithm works by considering each line in L in turn and marking off in an adjacency matrix C the adjacencies that this line crosses (lines 77 to 79). If any adjacency is only crossed by one line from L then that line must be essential and is added to the set E of essential lines (lines 80 to 84). As an example consider the configuration of rectangles in Figure 4.18. The algorithm in Figures 4.15 and 4.16 would have generated

```
76  Set $E$ to be empty
    {$E$ is the set of essential lines}

77  FOR each line $l$ in $L$
78     FOR each pair of adjacent rectangles $i$ and $j$ in $l.line$
79        Mark in $A[i,j]$ that $l$ crosses the adjacency between $i$ and $j$
80  FOR each adjacency $i|j$ in $A$
81     IF $i|j$ is only crossed by one candidate line $e$
82        THEN
83           Add $e$ to $E$
84           Remove $e$ from $L$
```

Figure 4.17: Finding the essential lines – Phase 2

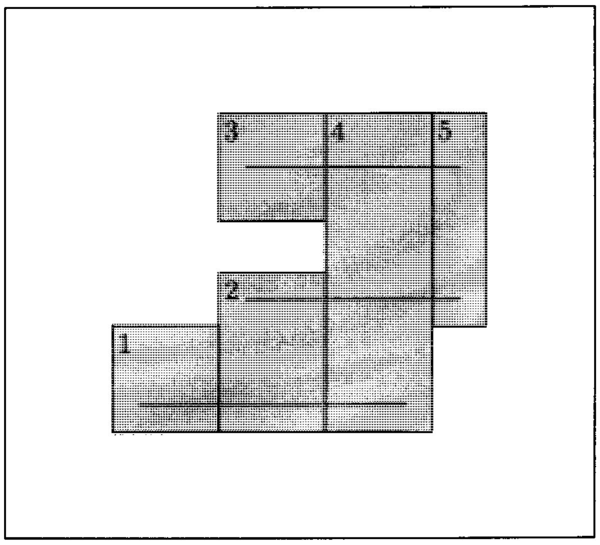

Figure 4.18: A configuration of adjacent orthogonal rectangles

```
85 FOR each line $e$ in $E$
86    FOR each pair of adjacent rectangles $i$ and $j$ in $e.line$
87       Mark in $A$ that $e$ crosses the adjacency between $i$ and $j$
88 FOR each line $l$ in $L$
89    FOR each pair of adjacent rectangles $i$ and $j$ in $l.line$
90       Check if adjacency $i|j$ is crossed by an essential line
91    IF all adjacencies in $l$ are crossed by lines in $E$
92       THEN Remove $l$ from $L$
```

Figure 4.19: Removing Redundant lines – Phase 3

3 lines (1–2–4, 3–4–5 and 2–4–5). The line 2–4–5 would have been created when considering rectangle 4 and attempting to extend the line 1–2–4 into rectangle 5 – this would not be possible and the line 2–4–5 would have been created by first creating a new line 4–5 and then extending that line backwards. In this situation the adjacency between rectangle 1 and rectangle 2 ($C[1, 2]$) is only crossed by the line 1–2–4 so this line must be an essential line. A similar observation applies for line 3–4–5 and the adjacency between rectangles 3 and 4. The line 2–4–5 is in fact redundant and the algorithm in Figure 4.19 describes how this line is identified and removed.

The algorithm considers each essential line in turn and marks off the adjacencies crossed by that line (lines 85 to 87). It then considers each line in L (the set of all possible lines) in turn and determines if all the adjacencies in a given line have been crossed by one or other of the essential lines (lines 88 to 90). If this is the case then the line is redundant. This can be seen in Figure 4.18. The line 2–4–5 crosses the adjacency between 2 and 4 which is also crossed by the essential line 1–2–4 and the adjacency between 4 and 5 which is also crossed by the line 3–4–5. Thus the line 2–4–5 is redundant and is removed from L (lines 91 and 92).

Once the essential lines have been identified and the redundant lines have been removed then there could still be some lines L which are choice lines (see the discussion in Section 4.3 and the Figures 4.2 and 4.11) and only some of these lines are necessary. Phase 4 of the algorithm (Figure 4.20) resolves this choice.

The algorithm first considers each of the remaining lines in L – these are the choice lines – and counts how many adjacencies, which have not already been crossed by essential lines, each line crosses (lines 93 and 94). In Figure 4.2 lines c and d each cross one such adjacency (the adjacency $1|3$). In Figure 4.11 two of the choice lines cross 6 previously uncrossed adjacencies and the other two lines cross 4 each. The algorithm then repeatedly applies the heuristic of choosing the line that

```
93  FOR each remaining line $z$ in $L$ {These are the choice lines}
94     Determine how many adjacencies not crossed by essential
       lines that this line crosses
95  REPEAT
96     Choose the line $y$ which crosses the most previously uncrossed
       adjacencies
97     Add $y$ to $E$
98     FOR each adjacency $a|b$ crossed by $y$
99        FOR each line $t$ which crosses $a|b$
100          Decrement the number of adjacencies crossed by $t$
101          IF the number of adjacencies crossed by $t$ is equal to 0
102             THEN Remove $t$ from $L$
103    Remove $y$ from $L$
104 UNTIL all of the adjacencies have been crossed.
```

Figure 4.20: Resolving the issue of choice – Phase 4

crosses the most previously uncrossed adjacencies and making that an "essential line" (actually adds it to the set of non-redundant lines) (lines 96 and 97). Any lines that cross previously uncrossed adjacencies crossed by the chosen line have their counts reduced appropriately and the process is repeated (lines 98 to 103). When all the previously uncrossed adjacencies have been crossed by lines chosen in this fashion then a non-redundant set of lines has been generated and the algorithm terminates. The heuristic applied means that a minimal set of axial lines is not guaranteed by this algorithm but it seems that the heuristic does produce reasonable approximations in some cases (see Section 4.6.4).

4.5.3 The Correctness of the method

The algorithm in Figures 4.15, 4.16, 4.17, 4.19 and 4.20 generates all the possible orthogonal axial lines that cross the adjacencies. It also extends all lines as far as possible to the left and right. Clearly any line that is the only line to cross a particular adjacency must be in the final set of lines otherwise there would be at least one adjacency that has not been crossed. Also any line that only crosses adjacencies which are crossed by lines which are essential should not be in the final set of lines.

It remains to show that the method for dealing with choice lines does give a non-redundant set of lines. The algorithm repeatedly chooses the choice line that crosses the highest number of adjacencies which are previously uncrossed. This means that the selected line can be treated as essential provided no line selected previously crosses any of those adjacencies. This is clearly the case. In addition, no line selected later can cross only those adjacencies and adjacencies already crossed by the essential lines. A line chosen later has to cross at least one previously uncrossed adjacency. Thus each line selected from the set of choice lines crosses at least one previously uncrossed adjacency and is thus necessary.

It should be noted that this algorithm is not guaranteed to give an optimal solution to the problem. It does, however, give a non-redundant solution from the point of view that removing any line from the final set of lines would leave at least one adjacency uncrossed. Figure 4.21 shows a configuration of rectangles where the heuristic would pick line a first as it crosses the most adjacencies not crossed by essential lines. The algorithm would then pick one of lines b and e, one of lines c and f and one of lines d and g. Thus a solution with four lines would be returned. The optimal solution would be lines b, c and d – only three lines.

4.6 Complexity Argument

4.6.1 Time

If all the rectangles were arranged such that one line could cross all the adjacencies between them then it is clear that this instance of the problem can be solved in linear time. Each rectangle has one adjacent rectangle and one candidate line passes from the rectangle to its neighbour.

The worst case for the algorithm could potentially occur when each rectangle (assuming n rectangles) in the configuration has $O(n)$ neighbours on its right-hand side and also has $O(n)$ lines coming into it from its left-hand side. In this case, if each incoming line could be extended into every right neighbour then $O(n^3)$ work would be required. If none of the incoming lines can be extended into a right neighbour and if for every right neighbour all of the lines have to be extended backwards to find the maximal line then potentially $O(n^4)$ work would be required. It is, however, not possible to construct configurations of rectangles which correspond to these cases.

A configuration of rectangles which would force a lot of work to be done in extending lines forward is shown in Figure 4.22. In this case the rectangles on the left edge of the collection will give rise to $n/2$ lines that must be extended through another $n/2$ rectangles. This means that the portion of the algorithm which generates the lines (lines 35 to 50) is $O(n^2)$ – order $O(n)$ rectangles have $O(n)$ incoming lines which can be extended into 1 neighbour on the right.

The configuration of rectangles in Figure 4.23 shows a situation where $O(n)$ lines would have to be extended forwards and then extended backwards. This configuration would force the algorithm to do a similar amount of work to the case above for generating the lines going into the last large vertical rectangle. This rect-

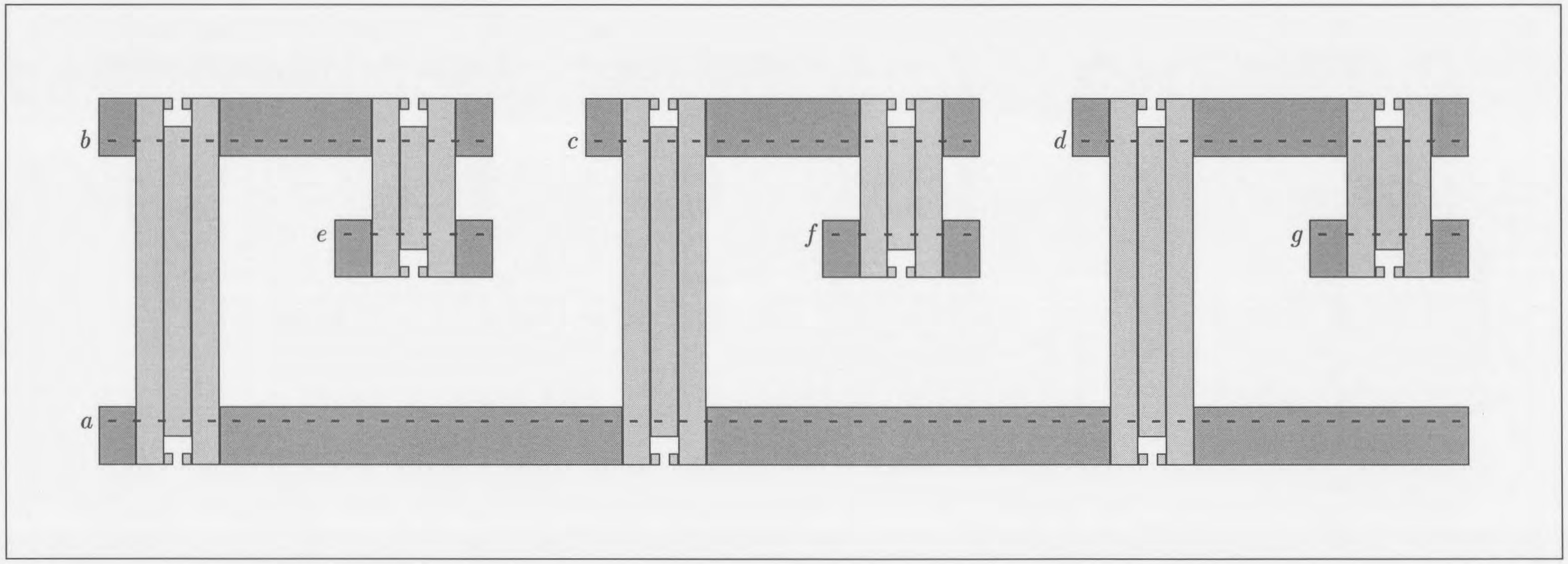

Figure 4.21: An example where the heuristic algorithm would not return an optimal solution

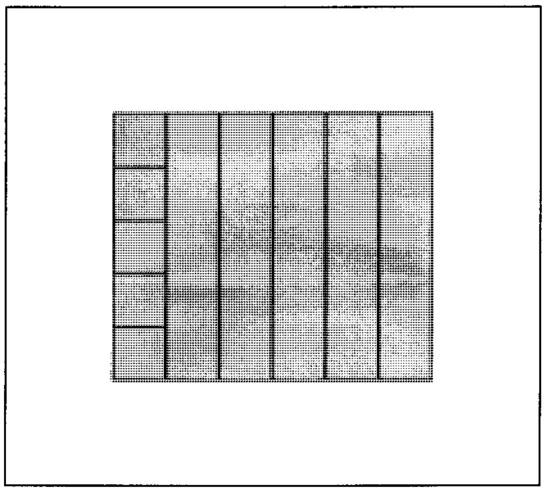
Figure 4.22: A configuration in which there are O(n^2) adjacency crossings

angle then has O(n) incoming lines (the lines originating from the small rectangles on the left of the configuration) and O(n) right neighbours. None of these lines can be extended into any of the small rectangles on the right-hand side of the configuration and so lines must be extended backwards from each of the adjacencies between the right-hand side small rectangles and the rightmost tall rectangle. An additional O(n^2) work would have to be done. Overall the work done is still O(n^2) .

The adjacencies which are crossed by only a single line can be found in O(n^2) time by traversing each line and marking off in the adjacency matrix each adjacency as it is crossed. The first time it is crossed it is marked with the identity of the line that crosses it and subsequent crossings are marked as such (setting the first line field to some flag value). As the number of lines must be less than or equal to the number of adjacencies and the number of adjacencies crossed by any line must be less than or equal to the number of adjacencies, this is clearly O(n^2). Having done this it is easy, O(n^2), to determine which adjacencies are only crossed once and thus to determine the essential lines.

Removing redundant lines can also be done in O(n^2) time – by first marking the adjacencies crossed by the essential lines and then determining which lines in the set of candidate lines only cross adjacencies already crossed by essential lines.

The issue of resolving the choice lines is potentially the most expensive part of the algorithm but, in fact, is also O(n^2). The first step in this phase is to calculate how many adjacencies which are not crossed by essential lines are crossed by each choice line. This is clearly O(n^2) – each adjacency (O(n)) in each line (O(n)) is considered. On each pass through the loop (lines 94 to 104) a number of steps are performed – the line which crosses the highest number of these previously uncrossed adjacencies is selected (O(n)); this line is added to E (O(1)); each adjacency of the selected line is considered and other lines crossing this adjacency have

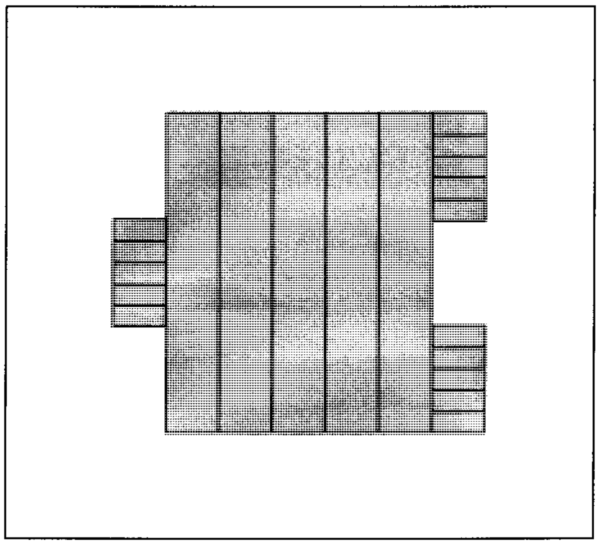

Figure 4.23: A configuration which forces the algorithm to extend O(n) lines backwards

their counts decremented and are removed from L if they no longer cross uncrossed adjacencies (this could be as expensive as O(n^2) if the line selected has O(n) adjacencies and each adjacency is crossed by O(n) lines); finally the selected line is removed from L (O(1)). The work done inside the loop could potentially be as expensive as O(n^2) (lines 98 to 102). This would mean that resolving choice could be as expensive as O(n^3). Again it seems unlikely that a configuration of rectangles which forces this amount of work could be created.

The overall time complexity of the algorithm to produce a non-redundant set of orthogonal axial lines is thus potentially as bad as O(n^4) but this situation is unlikely to occur because of the geometry of the problem.

4.6.2 Space

For each candidate line we store the list of rectangles which are crossed by the line, the list of top and bottom coordinates for each of the rectangles in the line and the final interval for the whole candidate line to date. The final interval can be used each time the line is tested for extending but recalculation must be done when the line is extended backwards. This is O(n^2) space (O(n) lines by O(n) possible crossings of adjacencies). In addition the adjacency matrix could require O(n^2) space.

4.6.3 Bounding the heuristic

The greedy heuristic of choosing the line with the most previously uncrossed adjacencies is analogous to the heuristic of choosing the vertex with the most edges in the original vertex cover problem. This algorithm has been shown not to be an ϵ-approximation [Papadimitriou, 1994] (for any $\epsilon < 1$ the error ratio grows as $\log n$ and thus no ϵ smaller than 1 is valid). The best known approximation algorithm for this problem is based on choosing any edge, say (u, v), in the set of edges, adding both u and v to the set of vertices and repeating until all edges are covered [Papadimitriou, 1994]. This algorithm has an approximation threshold of at most $1/2$ – a solution which is at most twice the optimum solution. Hochbaum [1982] also discusses a heuristic that gives a value that does not exceed twice the optimal value. For unweighted graphs he guarantees a bound strictly less than 2 – a solution strictly less than twice an optimal solution.

4.6.4 Experimental Results

Three algorithms were implemented and tested: the greedy algorithm (most uncrossed adjacencies heuristic); an algorithm based on the random selection of lines and an algorithm that produces the minimum number of orthogonal axial lines required. The algorithm that produces the minimum solution to the problem did an exhaustive search of all the possible ways of selecting subsets of choice lines to find a minimum sized subset. This algorithm was made as efficient as possible by partitioning the choice axial lines into subsets which have in common adjacencies not crossed by essential lines. The solutions for these subsets can then be found independently. This solution works well on average but it is still possible that all the choice lines are in one subset – there are uncrossed adjacencies that are common to all of the choice axial lines.

The test data were generated by first randomly generating a number of rectangles – typically about 30 – placed one on top of another. This configuration was then grown from left to right by randomly generating rectangles which are adjacent to the right hand edges of those that had already been placed in the configuration. The advantage of generating the data in this fashion was that a large number of adjacencies between rectangles in the horizontal direction was guaranteed. All rectangles in the configuration had breadth and height randomly chosen in the range from 5 to 15 units. The final configuration of rectangles was tightly packed and each rectangle could have as many as 4 rectangles adjacent to its right hand edge.

Fifty cases of configurations of 1000 rectangles each were tested, as were 20 cases of configurations of 1500 and 2000 rectangles. In the tests performed the greedy algorithm performs as well as the exact solution in most cases but there were instances of the greedy algorithm requiring an extra line to cross all the adjacencies. The random algorithm ranged in accuracy from producing the same result as the exact solution to requiring six extra lines to cross all adjacencies. Table 4.1 shows the results of the testing of the heuristics on configurations of rectangles of this form.

Number of rectangles	1000	1500	2000
Number of tests	50	20	20
Minimum number of lines	321	477	655
Maximum number of lines	365	527	703
Average number of lines	341	510	680
Standard deviation	10.86	10.81	13.44
Minimum number of essential lines	277	426	569
Maximum number of essential lines	322	476	616
Average number of essential lines	300	450	592
Standard deviation	9.69	14.19	14.24
Minimum number of choice lines	4	4	18
Maximum number of choice lines	27	36	52
Average number of choice lines	14	20	32
Standard deviation	5.83	8.44	8.77
Minimum error for most uncrossed adjacencies heuristic	0	0	0
Maximum error for most uncrossed adjacencies heuristic	1	1	1
Average error for most uncrossed adjacencies heuristic	0.02	0.10	0.05
Standard deviation	0.14	0.31	0.22
Minimum error for random choice heuristic	0	0	0
Maximum error for random choice heuristic	4	6	5
Average error for random choice heuristic	1.26	1.30	1.90
Standard deviation	0.94	1.66	1.55

Table 4.1: Experimental results

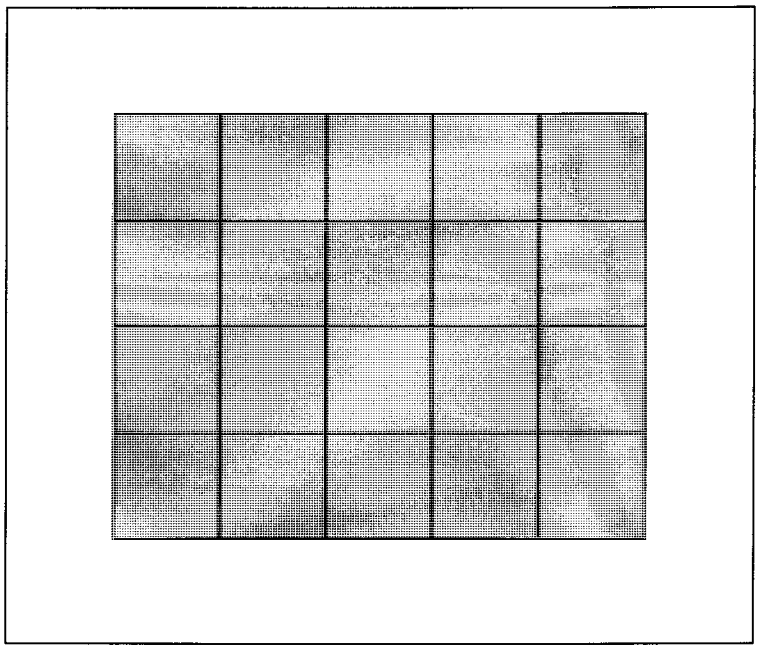

Figure 4.24: A "chequerboard" collection of rectangles

From these results it can be seen that the heuristic of choosing the choice axial line that crosses the most previously uncrossed adjacencies at any stage resulted in a good approximation for the configurations tested. For much larger numbers of rectangles or a different packing method the heuristic might not work as well but configurations of this form were chosen as a reasonable approximation to the type of collections of rectangles that could occur in the problem being studied.

This experimental work showed that although ***ALP-OLOR*** is in general NP-Complete it is possible to get good approximations to the exact solution – at least in the cases tested. The next section of this thesis considers special cases of ***ALP-OLOR*** where exact solutions can be found in polynomial time.

4.7 Special Cases that can be solved exactly in polynomial time

4.7.1 Mapping to interval graphs

ALP-OLOR is in general NP-Complete but there are some cases for which polynomial time algorithms can be obtained. In this section some of these special cases are discussed.

It is clear that any "chequerboard" collections of rectangles (Figure 4.24) can be solved exactly in polynomial time even if there are holes in the chequerboard

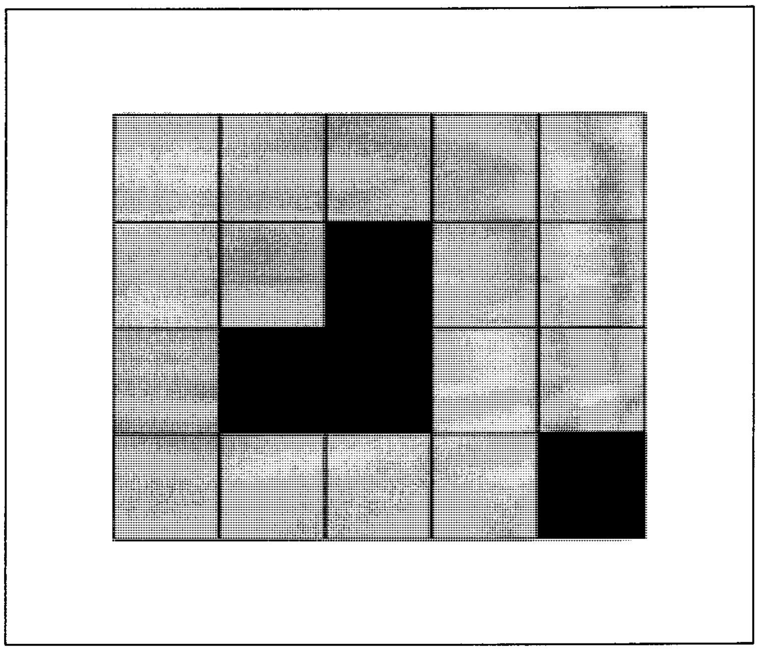

Figure 4.25: A "chequerboard" with holes

(Figure 4.25) [Hedetniemi, 1996]. In fact, this condition can be extended to a more general collection of rectangles.

Suppose that the union of the adjacent rectangles is itself a rectangle as in Figure 4.2 (of Section 4.3), Figure 4.22, Figure 4.24 and Figure 4.26, then the axial line placement problem for orthogonal axial lines and orthogonal rectangles can be solved in polynomial time. This can be shown as follows. First, project each vertical adjacency to a corresponding interval on the vertical line L (see Figure 4.27). Then the problem of finding the minimum number of horizontal lines that intersect the vertical adjacencies (***ALP-OLOR***) is equivalent to that of finding the minimum number of points on L needed such that each interval contains at least one point. This is the problem of finding the independent set of an interval graph which can be solved in linear time [Gavril, 1972; Golumbic, 1980]. The mapping from ***ALP-OLOR*** to vertex cover for interval graphs is also possible for other configurations of adjacent rectangles provided that any vertical adjacencies that produce overlapping intervals when projected onto L can be crossed by a horizontal line that does not leave the union of the rectangles. For example for the configuration of rectangles in Figure 4.28 the mapping would produce a correct answer but for the configuration in Figure 4.29 (and in fact in Figure 4.25) it would not.

There are other configurations of rectangles where solutions to ***ALP-OLOR*** can be found in polynomial time. Some of these are discussed below.

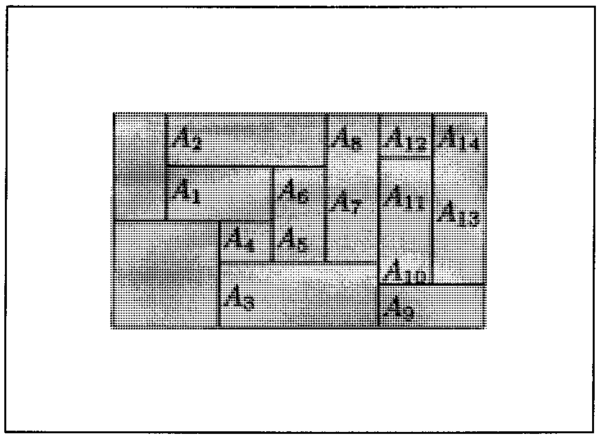

Figure 4.26: A simple configuration of rectangles with a rectangular union

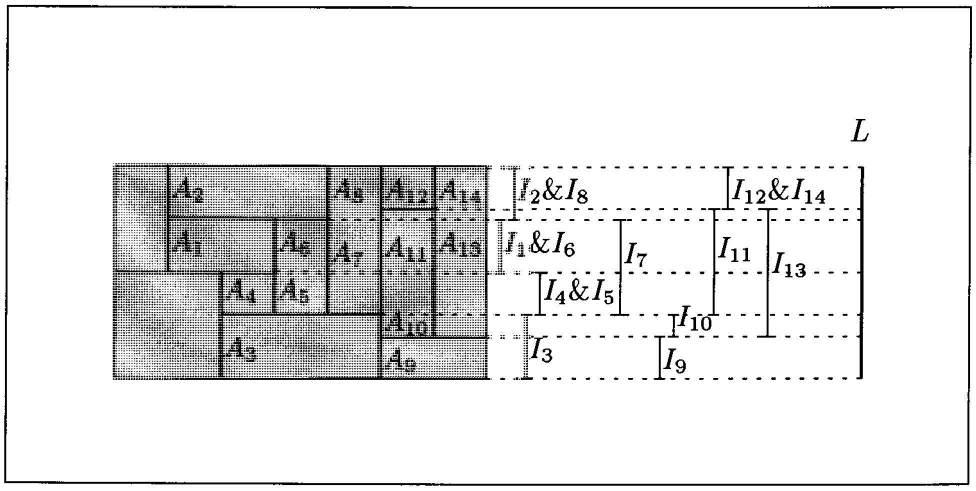

Figure 4.27: Projecting Adjacencies onto Intervals on the line L

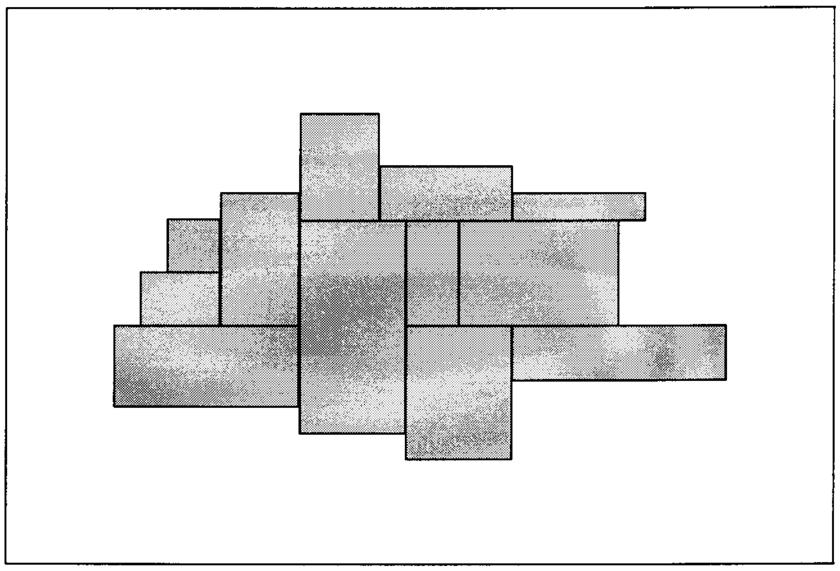

Figure 4.28: A simple configuration of rectangles that can be used in the production of an interval graph

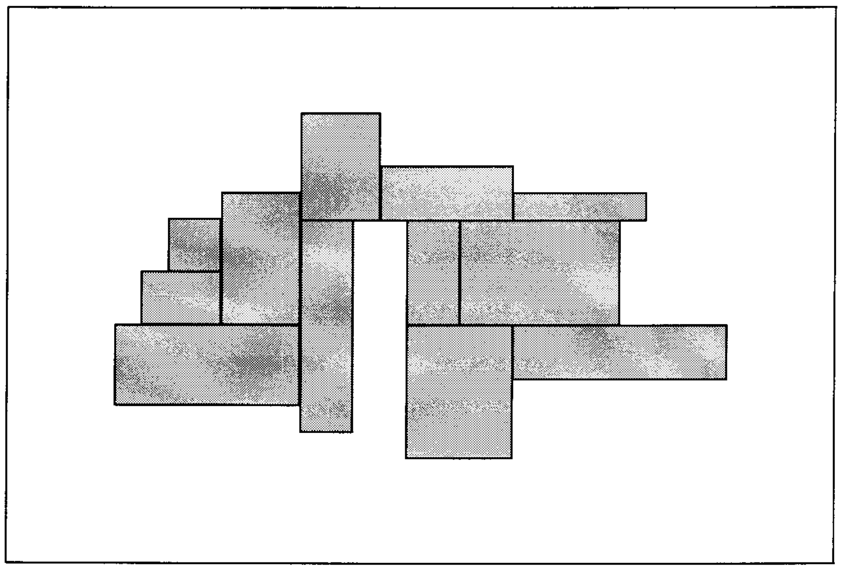

Figure 4.29: A simple configuration of rectangles that cannot be used in the production of an interval graph

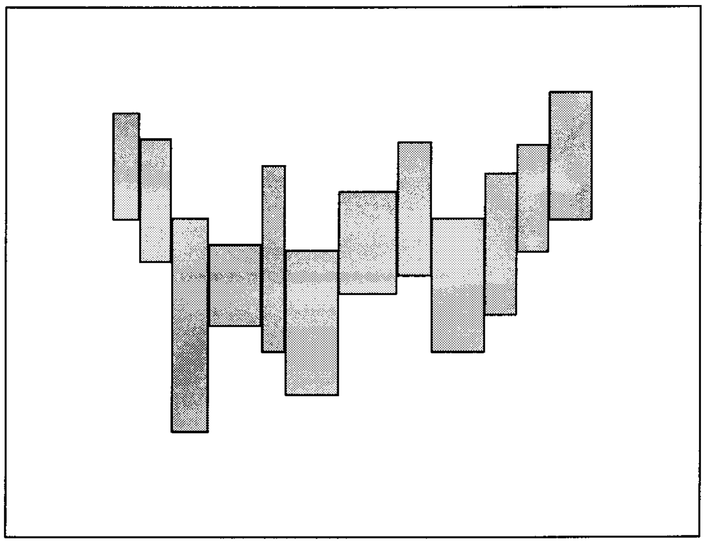

Figure 4.30: An example of a chain

4.7.2 Chains and trees of orthogonal rectangles

This section of the thesis considers algorithms to solve two restricted cases of ***ALP-OLOR*** – chains and trees of rectangles (see Section 4.7.2.1 for definitions of these). These problems cannot be solved by a mapping to an interval graph because the layout of the rectangles could lead to adjacencies that cannot be crossed by a single axial line which remains inside the union of the rectangles being mapped to the same interval. These problems can, in fact, be solved using the heuristic algorithm discussed in Section 4.5 (and discussed briefly in Section 4.7.2.2 below) but this algorithm does a lot of unnecessary work in these cases so better ways of solving these problems are required. In this section of the thesis an $O(n)$ algorithm for orthogonal axial line placement in chains of orthogonal rectangles is given (Section 4.7.2.3) and an $O(n^2)$ algorithm for trees of rectangles is presented (Section 4.7.2.4.

4.7.2.1 Terminology

Definition 4.7.1 *A chain of orthogonal rectangles is any collection of orthogonal rectangles where every rectangle is horizontally (vertically) adjacent to at most one other rectangle at each end.*

An example of a chain is shown in Figure 4.30.

Definition 4.7.2 *A tree of orthogonal rectangles is a collection of adjacent orthogonally aligned rectangles, where each rectangle is joined on the left (right) end to at most one rectangle and on the right (left) to zero or more rectangles.*

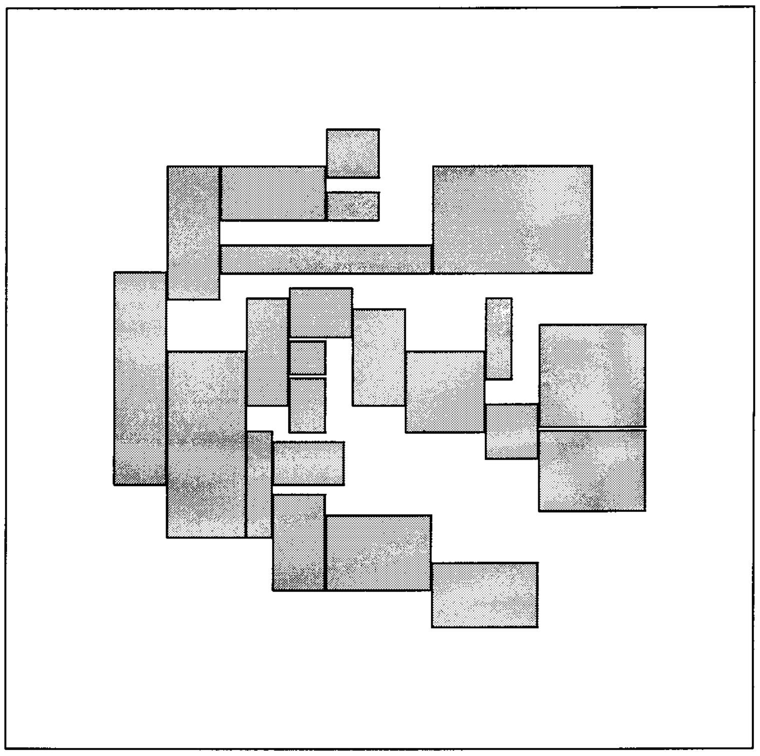

Figure 4.31: An example of a tree of rectangles

A tree is thus a generalisation of a chain – each branch of the tree can be considered as a chain of rectangles. An example of a tree of rectangles is shown in Figure 4.31.

4.7.2.2 The naive algorithm

An $O(n^2)$ algorithm to return a non-redundant set of maximal orthogonal axial lines for ***ALP-OLOR*** was presented in Section 4.5 above. This algorithm has four phases (after determining which rectangles are adjacent). It generates all the possible straight lines which cross the adjacencies between rectangles; it determines which lines are essential (i.e. are the only lines which cross a particular adjacency); it removes any lines which only cross adjacencies crossed by the essential lines (redundant lines); and then it resolves the choice conflict. The resolving of the choice is done by repeatedly choosing the choice line that crosses the highest number of previously uncrossed adjacencies. Each phase is $O(n^2)$ in the worst case.

This algorithm can be applied directly to place the minimal number of axial lines in chains and trees of orthogonal rectangles but it does more work than is necessary – it generates lines which are redundant and then have to be removed to get a minimal set of maximal lines. This is shown in Figure 4.32 where 5 lines are generated in the first part of the algorithm but only 2 lines are actually needed – the lines a–b–c–d–e–f and e–f–g–h–i–j. The reason that these extra lines are

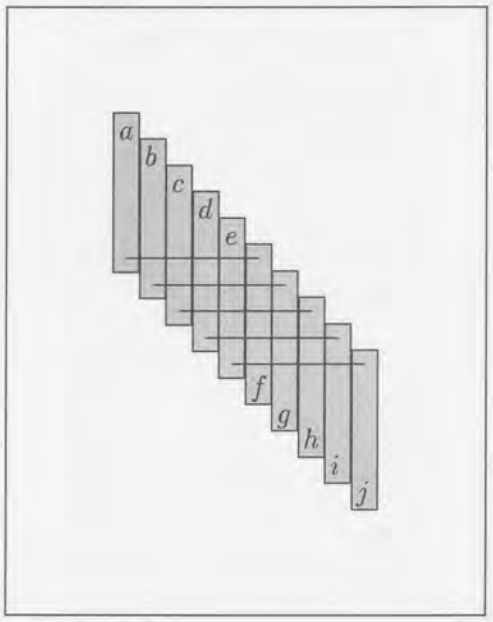

Figure 4.32: A case where more axial lines than necessary are generated

generated is that the algorithm starts from the left and tries to extend any existing line into any rectangle adjacent to the current one. If this can be done it is, but if it cannot be done then a new line is created crossing from the current rectangle into the next rectangle and this new line is then extended as far as it can be to the left before the algorithm continues working towards the right. In Figure 4.32 the line a–b–c–d–e–f cannot be extended into g so a new line f–g is created and this is extended back as far as possible giving the line b–c–d–e–f–g. Similarly for the other lines shown. Phase 3 of the algorithm would identify the two lines which have to be in the solution – the line a–b–c–d–e–f is the only line to cross the adjacency between rectangles a and b and the line e–f–g–h–i–j is the only line to cross the adjacency between i and j. These lines are thus essential. The unnecessary lines would then be removed from the final set of lines in phase 3 of the algorithm. Phase 4 of the algorithm would be unnecessary for both chains and trees of rectangles as no choice lines will ever be generated.

The remainder of this section of the thesis looks at algorithms that are more efficient for solving this problem in the case of chains and trees of orthogonal rectangles. Section 4.7.2.3 gives an $O(n)$ algorithm for placing orthogonal axial lines in chains of orthogonal rectangles and Section 4.7.2.4 gives an $O(n^2)$ algorithm for placing orthogonal axial lines in trees of orthogonal rectangles.

```
{Stage 0}
{Get input}
create array $Rect[\ ]$ of all given rectangles
{$n$ is the number of rectangles}
FOR $i$ from 1 to $n$
    Input $Rect[i].left$
    Input $Rect[i].right$
    Input $Rect[i].top$
    Input $Rect[i].bottom$
    Set $Rect[i].adjbottom$ to be undefined {will be calculated}
    Set $Rect[i].adjtop$ to be undefined

{Stage 1:}
{Define order of rectangles in the chain}
sort $Rect[\ ]$ according to left value of each rectangle
{i.e.  based on $Rect[\ ].left$}

{Stage 2:}
{Determine the extent of the adjacency between each rectangle
 and its right neighbour.}
FOR $i$ from 1 to $n-1$
    $Rect[i].adjtop \leftarrow$ minimum($Rect[i].top$, $Rect[i+1].top$)
    $Rect[i].adjbottom \leftarrow$ maximum($Rect[i].bottom$, $Rect[i+1].bottom$)
```

Figure 4.33: The algorithm for placing orthogonal axial lines in chains of orthogonal rectangles – Stages 0 to 2

4.7.2.3 Orthogonal axial line placement in chains of rectangles

In this section of the thesis an algorithm is presented to solve the orthogonal axial line placement problem in a chain of orthogonal rectangles. The algorithm is shown in Figures 4.33, 4.34, 4.35 and 4.36.

The algorithm takes as input a set of rectangles that are known to represent a chain. Each rectangle is defined by the coordinates of its bottom left and top right corner. The data structure used is an array of records *Rect*[]. Here each record has fields for *left*, *right*, *top* and *bottom* to represent the rectangle's coordinates (minimum x-coordinate, maximum x-coordinate, minimum y-coordinate, maximum y-coordinate) and *adjtop* and *adjbottom* to represent the lowest and highest y-values of the range of y-values that defines the adjacency with the next rectangle in the

```
{Stage 3:}
{Determining the set of forward lines}
{$ForwardLines$ is a list of the lines that have been found in
 this stage}
{The forward sweep is started by initialising the smallest
 common adjacency}
{This is called $CurrentAdj$}
{Initially this is the adjacency between rectangles 1 and 2}
{$Currentline$ is a list of adjacencies that are crossed by the
 line being worked on}

$CurrentAdj.top \leftarrow Rect[1].adjtop$
$CurrentAdj.bottom \leftarrow Rect[1].adjbottom$
Add adjacency 1|2 to the list $CurrentLine$

FOR $i$ from 2 to $n$
   IF ($Rect[i].adjtop$ < $CurrentAdj.bottom$) OR
   ($Rect[i].adjbottom$ > $CurrentAdj.top$)
      THEN
         Add the list $CurrentLine$ to the end of list $ForwardLines$
         $CurrentAdj.top \leftarrow Rect[i].adjtop$
         $CurrentAdj.bottom \leftarrow Rect[i].adjbottom$
         Set $CurrentLine$ to be empty
      ELSE
         IF ($Rect[i].adjtop$ < $CurrentAdj.top$)
            THEN $CurrentAdj.top \leftarrow Rect[i].adjtop$
         IF ($Rect[i].adjbottom$ > $CurrentAdj.bottom$)
            THEN $CurrentAdj.bottom \leftarrow Rect[i].adjbottom$
         Add the adjacency i|i+1 to the end of list $CurrentLine$
IF $CurrentLine$ is not empty
   THEN
      Add the list $CurrentLine$ to the end of list $ForwardLines$
```

Figure 4.34: The algorithm for placing orthogonal axial lines in chains of orthogonal rectangles – Stage 3

```
    {Stage 4:}
    {Determining the set of reverse lines to be stored in list
     $ReverseLines$}

29  Set $CurrentLine$ to be empty
30  $CurrentAdj.top \leftarrow Rect[n-1].adjtop$
31  $CurrentAdj.bottom \leftarrow Rect[n-1].adjbottom$
32  Add the adjacency n-1|n to the end of list $CurrentLine$

33  FOR $i$ from $n-2$ downto 1
34     IF ($Rect[i].adjtop$ < $CurrentAdj.bottom$) OR
       ($Rect[i].adjbottom$ > $CurrentAdj.top$)
35        THEN
36           Add the list $CurrentLine$ to the front of list $ReverseLines$
37           $CurrentAdj.top \leftarrow Rect[i].adjtop$
38           $CurrentAdj.bottom \leftarrow Rect[i].adjbottom$
39           Set $CurrentLine$ to be empty
40        ELSE
41           IF ($Rect[i].adjtop$ < $CurrentAdj.top$)
                THEN $CurrentAdj.top \leftarrow Rect[i].adjtop$
42           IF ($Rect[i].adjbottom$ > $CurrentAdj.bottom$)
                THEN $CurrentAdj.bottom \leftarrow Rect[i].adjbottom$
43           Add the adjacency i|i+1 to the front of list $CurrentLine$
44  IF $CurrentLine$ is not empty
45     THEN
46        Add the list $CurrentLine$ to the front of list $ReverseLines$
```

Figure 4.35: The algorithm for placing orthogonal axial lines in chains of orthogonal rectangles – Stage 4

```
    {Stage 5:}
    {Merge the lines to get the final set of lines}
47  Set $m$ to the number of lines in $\mathit{ForwardLines}$ or $\mathit{ReverseLines}$
    {these will be equal}
48  FOR $i$ from 1 to $m$
49     set $\mathit{FinalLines}[i]$ to be empty
50     WHILE $\mathit{ForwardLines}[i]$ is not empty AND
       $\mathit{ReverseLines}[i]$ is not empty
51        IF (the first adjacency in $\mathit{ForwardLines}[i]$ is to the left
          of the first adjacency in $\mathit{ReverseLines}[i]$)
52           THEN
53              remove first adjacency from $\mathit{ForwardLines}[i]$
54              add this adjacency to end of $\mathit{FinalLines}[i]$
55           ELSE
56              IF the adjacencies are the same
57                 THEN
58                    remove first adjacency from $\mathit{ForwardLines}[i]$
59              remove first adjacency from $\mathit{ReverseLines}[i]$
60              add this adjacency to end of $\mathit{FinalLines}[i]$
61     IF $\mathit{ForwardLines}[i]$ is not empty
          THEN add all remaining adjacencies to $\mathit{FinalLines}[i]$
62     IF $\mathit{ReverseLines}[i]$ is not empty
          THEN add all remaining adjacencies to $\mathit{FinalLines}[i]$
```

Figure 4.36: The algorithm for placing orthogonal axial lines in chains of orthogonal rectangles – Stage 5

chain. The algorithm sorts these rectangles according to the x-coordinate of the left edge of each rectangle. This first stage is dominated by the sorting and thus has a complexity of $O(n \lg n)$. Clearly this stage is unnecessary if the rectangles are given in sorted order as a chain of rectangles.

From this sorted list of rectangles the right adjacencies of each rectangle are found – lines 9 to 11 of the algorithm – by comparing the y-coordinate range of any rectangle with the y-coordinate range of its neighbour on the right – the largest bottom y-value and the smallest top y-value define this adjacency. This information is stored in $Rect[i].adjtop$ and $Rect[i].adjbottom$ for rectangle i. This process takes linear time and forms the second stage of the algorithm.

The third stage involves determining the lines that move forward through the chain. This stage of the algorithm proceeds by traversing the chain of rectangles from left to right keeping track of any common range of y-values ($CurrentAdj$) of the adjacencies that have been considered so far. The existence of such a range implies that an axial line could be placed to cross the adjacencies that share this range. The algorithm also keeps track of which adjacencies between rectangles could be crossed by such a line by maintaining a list of these adjacencies ($CurrentLine$ is the list of adjacencies that share a common range of y-values and each adjacency is given by the rectangles in it in the form $left|right$). The common range starts out as the range of y-values of the adjacency between the first and second rectangles (see lines 12 and 13 of the algorithm). $CurrentLine$ is initialised to $1|2$ (line 14) as rectangle 1 is adjacent to rectangle 2 and a line could be drawn crossing the adjacency between them. This range and the current line are then updated when the next adjacency is encountered (see lines 15-25). If there is an overlap in y-values between the common range and the next adjacency then the current line can be extended (lines 23-25). If there is no overlap then a new line must be started (18 to 21). In this case the common range is reset as the range of the adjacency being considered. Lines 16 to 25 are repeated until the end of the chain is reached i.e. until all the adjacencies have been crossed by a line in the set of forward lines. Lines 26 to 28 make sure that the last line is also added to the set of forward lines. Since each rectangle is visited once by a single line, this stage of the algorithm also takes linear time.

The fourth stage is the same the third stage, except one finds the reverse lines by moving from right to left, instead of left to right (see lines 29 to 46 which are very similar to lines 12 to 28). Note that the set of reverse lines are still arranged so that the leftmost lines come first in the ordering. The adjacencies in any line are also arranged in order from left to right. Clearly the complexity for this stage is the same as the previous stage.

Lastly the forward lines and the reverse lines are merged together to obtain the *maximal* lines that cross every adjacency. This merging is accomplish by noting that any forward line crosses each adjacency that it can exactly once and extends as far

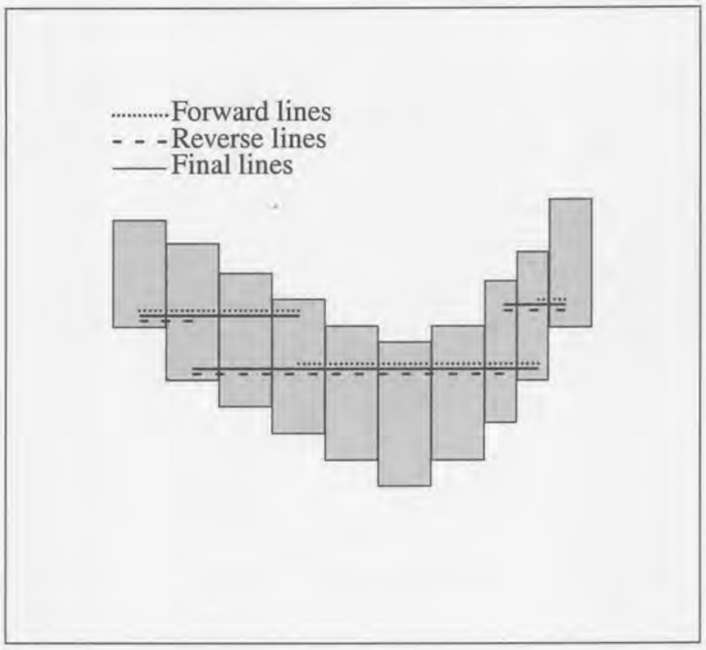

Figure 4.37: A chain of orthogonal rectangles showing the forward and reverse lines and the final maximal lines.

as possible to the right. The process of placing the forward lines thus generates a set containing the minimum number of *non-maximal* lines to cross all of the adjacencies in the chain – any fewer lines and some adjacencies would be left uncrossed, any more lines and one or more adjacencies would have to be crossed more than once. Similarly any reverse line crosses each adjacency that it can exactly once and extends as far as possible to the left. The set of reverse lines is also the minimum number of non-maximal lines required to cross all of the adjacencies. Clearly the two sets must contain the same number of lines. As each forward line extends as far as possible to the right, the forward lines will thus define the rightmost end of some maximal line. Similarly the reverse lines extend as far as possible to the left and define the leftmost extent of the maximal lines. Thus merging any forward and reverse lines that have ranges of y-values which overlap and that cross some common adjacencies will result in a maximal line, through the resulting overlap of y-values, that crosses all the adjacencies crossed by both lines.

The sets of forward and reverse lines are arranged from leftmost to rightmost starting x-value. The merging begins by considering the first (leftmost) lines in each set and generating a new line that crosses all of the adjacencies crossed by those two lines. No other lines need to be considered as no other lines will have an overlap of y-values and share some common adjacencies. Lines 47 to 62 show the detail of how this is accomplished. The merging begins by considering the first adjacency in $ForwardLines[1]$ and comparing it with the first adjacency in $ReverseLines[1]$

(line 51). The leftmost of these adjacencies (found by looking at the x-coordinates of the rectangles concerned) gives the leftmost adjacency in the new final line. This adjacency is removed from the list that it occurred in (both if this was the case) and added to final line which is being created (lines 53 and 54 or lines 56 to 60). The process is the repeated with the first adjacencies of the two new lists. If either list becomes empty then the remaining adjacencies in the non-empty list are copied to the final line which is being built up (lines 61 and 62). This new line is then the first line in the final set of lines – it is maximal because it extends as far as it can to the left and to the right. The other lines in the forward and reverse sets are handled in a similar fashion to produce final lines. An example of a chain of rectangles with forward lines, reverse lines and the resulting final lines is shown in Figure 4.37. This final stage of merging the forward and reverse lines takes $O(n)$ time to complete – each adjacency can appear once in a forward line and once in a reverse line so at most $2n$ adjacencies will be considered for addition into one of the final lines.

The complexity of the entire algorithm is thus $O(n)$.

Empirical tests were done on some different configurations of chains of rectangles [Watts and Sanders, 1997]. The data for the running time of the algorithm, excluding the sorting done in the first stage, verified the theoretical analysis – that is, the data confirmed that the last four stages of the algorithm are indeed linear.

This restricted instance of the problem is, in fact, a special case of the problem to be considered in the next section but it was presented as a separate problem in order to make the algorithm presented in the next section easier to understand.

4.7.2.4 Orthogonal axial line placement in trees of rectangles

The algorithm to find the minimal set of orthogonal axial lines to cross the adjacencies in a tree of orthogonal rectangles is presented in Figures 4.38, 4.39, 4.40 and 4.41. The algorithm takes as input a list of rectangles that represents a tree of orthogonal rectangles and produces a minimal set of maximal orthogonal axial lines.

The algorithm is split into six stages: inputting the rectangle data, finding the adjacencies between the rectangles; defining the order in which the rectangles will be visited; finding the forward lines; finding the leaf lines; and merging the forward lines and leaf lines into the final lines.

The main data structure in the algorithm (as given in Figure 4.38) is an array $Rect$ of records to represent the rectangles. Each rectangle is represented by a record with 8 fields – $left$, $right$, top and $bottom$ to define the rectangle, $parent$ to keep track of the rectangle to the left of the current rectangle, $numadj$ to keep track of the number of rectangles adjacent to the right end of any rectangle, $adjlist$ which is a list of these rectangles and finally $LeafLineNo$ which is used in stages 4 and 5 to keep track of which leaf line crosses the rectangle.

```
    {Stage 0:}
    {Get input}
00  create array Rect[ ] of all given rectangles
    {$n$ is the number of rectangles}
01  FOR $i$ from 1 to $n$
02     Input Rect[$i$].left
03     Input Rect[$i$].right
04     Input Rect[$i$].top
05     Input Rect[$i$].bottom
06     Set Rect[$i$].parent to be undefined {will be calculated}
07     Set Rect[$i$].numadj to be undefined {will be calculated}
08     Set Rect[$i$].adjlist to be Nil {will be calculated}
09     Set Rect[$i$].LeafLineNo to be undefined
       {will be used in stages 4 and 5}

    {Stage 1:}
10  Find adjacencies {using algorithm discussed before}
    {Rect[$i$].parent, Rect[$i$].numadj and Rect[$i$].adjlist are calculated here}

    {Stage 2:}
    {Define the order in which the rectangles will be visited}
11  create array RightList[ ] sorted according to right value of
    each rectangle {i.e.  based on Rect[ ].right}
    {This will be an array of the numbers of the rectangles in
     the order in which they will be visited.}
```

Figure 4.38: The algorithm for placing orthogonal axial lines in trees of orthogonal rectangles – Stages 0 to 2

```
{Stage 3:}
{Find forward lines}

{Set values in root vertex of Interval tree}
high ← Rect[RightList[1]].top
low ← Rect[RightList[1]].bottom
line ← RightList[1]
FOR p from 2 to n do {traverse rectangles in order of right edge}
   k ← RightList[p]
   IF Rect[k].numadj = 0
      THEN
         add k to LeafList
      ELSE
         find vertex z in interval tree such that
         overlap(interval(z.low, z.high),
         interval(Rect[k].bottom, Rect[k].top)) = true
         AND last rectangle in z.line = k
         set extended to be false
         FOR each rectangle r in Rect[k].adjlist do
            IF overlap(interval(z.low, z.high),
            interval(Rect[r].bottom, Rect[r].top)) = true
               THEN
                  {Extend an existing line}
                  Insert a child of z in interval tree with
                     high ← top of overlap
                     low ← bottom of overlap
                     line ← z.line || r
                  set extended to be true
               ELSE
                  {Start a new line}
                  Insert a child of zin interval tree with
                     high ← Rect[r].top
                     low ← Rect[r].bottom
                     line ← r
         IF extended = true THEN delete vertex z
{traverse interval tree to produce a list of forward lines}
i ← 0
FOR each vertex z in the interval tree
   i ← i + 1
   ForwardLines[i].rects ← z.line
   ForwardLines[i].top ← z.high
   ForwardLines[i].bottom ← z.low
```

Figure 4.39: The algorithm for placing orthogonal axial lines in trees of orthogonal rectangles – Stage 3

```
    {Stage 4:}
    {Find leaf lines}

43  FOR each leaf do
44      $k$ ← $RectNo$ of current leaf
45      $CurrentT$ ← $Rect[k].top$
46      $CurrentB$ ← $Rect[k].bottom$
47      $p$ ← $Rect[k].parent$
48      $currentline.rects$ ← $k$
49      $i$ ← 1
50      WHILE $p$ is still defined
51          IF $overlap(interval(Rect[p].top, Rect[p].bottom),$
            $interval(CurrentT, CurrentB))$ = true
52              THEN
                    {extend an existing leaf line}
53                  $currentline.rects$ ← p || $currentline.rects$
54                  $currentline.top$ ← top of overlap
55                  $currentline.bottom$ ← bottom of overlap
56                  $CurrentT$ ← top of overlap
57                  $CurrentB$ ← bottom of overlap
58              ELSE
                    {add current line to set of leaf lines}
59                  $LeafLines[i].rects$ ← $currentline.rects$
60                  $LeafLines[i].top$ ← $currentline.top$
61                  $LeafLines[i].bottom$ ← $currentline.bottom$
                    {start a new leaf line}
62                  $i$ ← $i+1$
63                  $currentline.rects$ ← $p$
64                  $currentline.top$ ← $Rect[p].top$
65                  $currentline.bottom$ ← $Rect[p].bottom$
66                  $CurrentT$ ← $Rect[p].top$
67                  $CurrentB$ ← $Rect[p].bottom$
68          $Rect[k].LeafLineNo$ ← $i$
69          $k$ ← $p$
70          $p$ ← $Rect[k].parent$
```

Figure 4.40: The algorithm for placing orthogonal axial lines in trees of orthogonal rectangles – Stage 4

```
{Stage 5:}
{Merge forward lines and leaf lines to produce final lines}

Set $m$ to the number of lines in $ForwardLines$
FOR $i$ from 1 to $m$
    $FinalLines[i].top \leftarrow ForwardLines[i].top$
    $FinalLines[i].bottom \leftarrow ForwardLines[i].bottom$
    set $FinalLines[i].line$ to be empty
    FOR each rectangle $j$ in $ForwardLines[i].line$ from right end
    to left end
        $FinalLines[i].line \leftarrow j$ || $FinalLines[i].line$
        $k \leftarrow Rect[j].LeafLineNo$
        IF $overlap(interval(ForwardLines[i].top, ForwardLines[i].bottom),$
        $interval(LeafLines[k].top, LeafLines[k].bottom))$ = true
            THEN
                $FinalLines[i].top \leftarrow$ top of overlap
                $FinalLines[i].bottom \leftarrow$ bottom of overlap
                Add all rectangles in $LeafLines[k].line$ to
                $FinalLines[i].line$
                Break out of for loop
```

Figure 4.41: The algorithm for placing orthogonal axial lines in trees of orthogonal rectangles – Stage 5

The adjacencies between the rectangles are found using the algorithm presented in Section 4.5.1. In this stage of the algorithm $parent$, $numadj$ and $adjlist$ are given values.

In stage 2 of the algorithm an additional array, $RightList[]$, is created which stores a list of the indices of the array $Rect[]$ arranged according to the x-coordinate of the right end of the rectangles. This list defines the order in which the rectangles will be visited in stage 3 – finding the forward lines. Stages 0 to 2 are essentially preprocessing to define a tree of orthogonal rectangles.

Stage 3 of the algorithm (as given in Figure 4.39) finds the *forward lines*. These are the lines found by starting at the root rectangle (in this case the leftmost rectangle – with the smallest x-coordinate of its right edge) and working towards the leaf rectangles of the tree (these are rectangles with no right neighbours) considering each rectangle in turn. An interval tree [Cormen *et al.*, 1990] is used to maintain the y-value ranges that can be considered at any stage. An interval tree is a red-black tree where each vertex x contains an (open or closed) interval defined by its low and high endpoints and where the *key* of the vertex is the low endpoint. Insertions into the tree are based on the low endpoint of the interval to be inserted. Thus an inorder traversal of an interval tree would return the intervals in sorted order by low endpoint. In the algorithm above every vertex in the interval tree stores an interval defined by the variables low to $high$ and, in addition, a candidate line ($line$) represented by a list of the rectangles such a line could cross. Each vertex in the interval tree thus represents a line that might need to be considered when attempting to extend lines from the root to the leaf rectangles. Lines 12 to 14 initialise the root vertex of the interval tree to contain the interval represented by the y-value range of the root rectangle and a line that crosses only that rectangle but could be extended into the root rectangle's right neighbours.

After the initialisation each rectangle is considered in turn (lines 15 to 36 of Figure 4.39). Each such rectangle can have at most one rectangle adjacent to it on the left and could have a number of other rectangles adjacent to it on the right. If it has no rectangles adjacent to it on the right then it is a leaf rectangle and its number is added to a list of such leaf rectangles (line 17 to 19). If it has adjacent rectangles on the right then the forward line coming into the rectangle could potentially be extended into these adjacent rectangles (lines 21 to 36). The algorithm searches the interval tree for the line which comes into the rectangle from the left – it will have an interval which has some overlap with the y-value range of the rectangle and the last rectangle through which the line passes will be the current rectangle (line 21). The algorithm then considers each right neighbour of the current rectangle in turn (lines 22 to 35) and determines which adjacencies can be crossed by extensions of this line. There could be more than one possible extension of the line depending on the y-coordinates of the adjacencies being considered and thus the one line coming into a rectangle from the left could become more than one line (see lines 26 to 30) –

one line going into each adjacent rectangle where there is an overlap of the interval covered by the line and bottom and top y-values of the adjacent rectangle. Each of these new lines would result in a new vertex being inserted into the interval tree to store the new interval and detail which rectangles are crossed by the new line. The algorithm also determines for which adjacencies (if any) new forward lines have to be started (lines 32 to 35), i.e. the existing line cannot be extended to cross the new adjacencies (there is no overlap of y-coordinates). Again new vertices are created in the interval tree.

After all the right neighbours have been considered then the vertex in the interval tree representing the line coming into the current rectangle is deleted if the line has been extended (it is no longer needed in this case). If the line has not been extended then it is one of the forward lines and is thus not deleted. The last step in this stage of the algorithm (lines 37 to 42) is to traverse the interval tree and produce a list of the forward lines, represented by a range of y values and a list of rectangles crossed, to cross all of the adjacencies in the tree of rectangles.

Figure 4.42 shows the forward lines generated for part of the tree of Figure 4.31. Note that the line that crosses the adjacencies between rectangles a and b and b and c can be extended in d but not into e. A new forward line is created to cross the adjacency between rectangles c and e. This line can be extended into f but not into g.

Stage 4 of the algorithm (as given in Figure 4.40) works from the leaves of the tree to the root rectangle, hence finding the *leaf lines*. This case is easier than that of finding the forward lines. It is only necessary to check if one of the lines coming into any rectangle from the right can be extended to cross the one adjacency on the left or whether a new leaf line must be created. The algorithm considers each leaf rectangle (from line 19 in stage 3) in turn (lines 43 to 70). The leaf line for any leaf rectangle is first initialised (lines 44 to 48) – this involves finding the interval to be considered, finding the parent of that rectangle and setting the first rectangle in the current line. The algorithm then works up the rectangle tree until the root rectangle is reached (lines 48 to 68). For each rectangle on the path from leaf rectangle to root rectangle it determines if the leaf line coming into a rectangle can be extended into the current rectangle's parent (lines 53 to 57) or if the current leaf line must be terminated and a new leaf line started (lines 59 to 67). This stage of the algorithm also records the number of the last leaf line that crosses from the current rectangle into its parent rectangle (*LeafLineNo* – line 68) – this could be a new leaf line (starts in that rectangle) or one which crosses the rectangle from right to left. This information is used in stage 5 of the algorithm which is described below. Figure 4.42 shows the leaf lines generated for part of the tree of Figure 4.31. Note that the leaf line from rectangle g cannot be extended from e into c and a new "leaf" line has to be created here.

The final stage of the algorithm (stage 5 as given in Figure 4.41) merges the

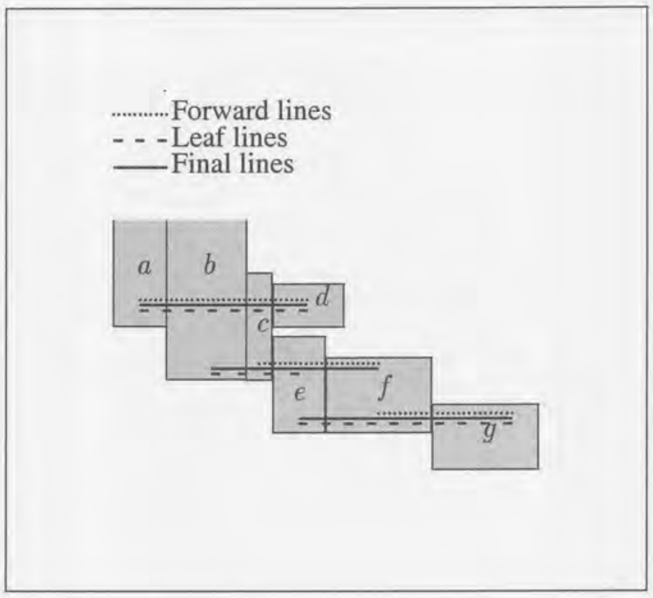

Figure 4.42: An example of placing orthogonal axial lines in a tree of orthogonal rectangles

forward and leaf lines to produce the set of maximal *final lines* which cross all of the adjacencies in the tree of rectangles. Section 4.7.2.3 explained why these lines need to be merged in order to obtain maximal axial lines. The merging here is necessarily more complicated than when working with chains of rectangles because any rectangle could be crossed by more than one forward line and more than one leaf line. Here the algorithm considers each forward line in turn (lines 71 to 84 in Figure 4.41). A new final line is created for each forward line (lines 73 to 75) and the algorithm then considers each rectangle in the forward line in turn starting from the rightmost end of the line – as before forward lines define the rightmost extent of any final line (lines 76 to 84). The leaf line which corresponds to the forward line is found by considering the leaf line (determined in stage 4, line 68) associated with each of the rectangles which make up the forward line in turn, until a leaf line that has an overlapping range of y-values is found (line 79). This leaf line then defines the leftmost extent (leaf lines are always extended as far as possible to the left). The final line is created by using the adjacencies from the forward line and its associated leaf line – the adjacencies from the forward line are used (lines 77 and 78) to define the right end of the final line and when the associated leaf line is found its adjacencies are used to define the left extent of the line (lines 81-84). Figure 4.42 shows the final lines (and forward and leaf lines) generated for part of the tree of Figure 4.31. The forward line c–e–f and the leaf line b–c–e which together result in the final line $b - c - e - f$ is an example of how the merging works. Rectangle f is considered first, the leaf line through f does not have an overlapping y-value range so rectangle e is considered. Here the leaf line which was stored in stage 4 of the algorithm is the line that starts in e and crosses into c. This line does have an overlap so the final line can be generated using the forward line and this leaf line.

This example also illustrates why it is sufficient in stage 4 (line 68) of the algorithm to record only the last leaf line which either starts in or crosses a given rectangle. Leaf lines define the left extent of any final line. If two or more leaf lines start in or cross a given rectangle then the associating of a leaf line and a forward line will not be done when that rectangle is considered. In Figure 4.42, rectangle c has 2 leaf lines which cross it – the line a–b–c–d and the line b–c–e. These leaf lines are paired with their forward lines when rectangle d of forward line a–b–c–d and rectangle e of forward line c–e–f respectively are being considered. Similar arguments apply for other combinations of more than one leaf line either crossing or starting in a given rectangle.

Sanders *et al.* [2000a] show another example of the applying the algorithm to a specific tree.

The algorithm essentially consists of four parts: finding the adjacencies and defining the order of processing, finding the forward lines, finding the leaf lines, and merging the forward lines and leaf lines. To analyse the algorithm each of these stages are considered in turn.

Stage 1 and 2: Find adjacencies and define order The algorithm used to find the adjacencies of the rectangles in stage 1 of the algorithm is as in Section 4.5.1. The algorithm also determines the array $RightList$. This algorithm is dominated by sorting and thus has a complexity of $O(n \lg n)$ where n represents the number of rectangles in the tree.

Stage 3: Find forward lines In essence this phase of the algorithm determines which adjacencies overlap with intervals that have already been inserted into the interval tree – an interval in the interval tree means that there is a forward line which can be considered. There are $n - 1$ adjacencies in a tree of n rectangles and each adjacency only needs to be considered once – only one forward line can cross any adjacency. Also each interval tree operation (adding or deleting here) takes $O(\lg n)$ time [Cormen *et al.*, 1990]. Therefore stage 3 of the algorithm has a complexity of $O(n \lg n)$.

This stage of the algorithm could be done more efficiently as regards time by noting that each rectangle can only have one line coming into it from the left and using a matrix of size $n - 1$ to store the y intervals of these lines. This has the advantage of direct lookup but the disadvantage of always requiring $O(n)$ space – the best case for the interval tree could be much less.

Stage 4: Find leaf lines In this phase of the algorithm the lines starting in the leaves of the tree are extended back towards the root of the tree. Here testing

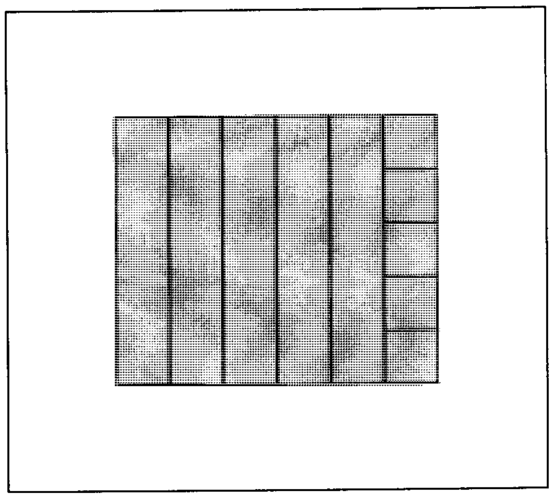

Figure 4.43: A tree with $n/2$ leaves and height also $n/2$

whether the line can be extended is $\mathrm{O}(1)$ (a simple test for overlap) but it is possible that there are $\mathrm{O}(n)$ leaves and that for each of these leaves it is necessary to extend the line through $\mathrm{O}(n)$ rectangles (a tree with approximately $n/2$ leaves and of height also approximately $n/2$ – see Figure 4.43 for a simple example of this case). So the complexity for this part of the algorithm is $\mathrm{O}(n^2)$ in the worst case.

Stage 5: Merge forward lines and leaf lines For a forward line to exist, it must cross at least one adjacency between two rectangles in the tree, and each adjacency is crossed exactly once by the definition of the axial line placement problem. Therefore since there are n rectangles, there are exactly $n-1$ of these adjacencies and there can be a maximum of $\mathrm{O}(n)$ forward lines.

The merging phase of the algorithm considers each forward line in turn and in the worst case works from the rightmost rectangle of the line until a leaf line with overlap is encountered. The algorithm then merges these two lines. Effectively this is merging two lines in a chain of rectangles which is $\mathrm{O}(n)$ (as shown in Section 4.7.2.3).

So since there are a maximum of $\mathrm{O}(n)$ forward lines and merging a forward line with its associated leaf line is $\mathrm{O}(n)$, this part of the algorithm is $\mathrm{O}(n^2)$ in the worst case.

The complexity of the entire algorithm is thus $\mathrm{O}(n^2)$ in the worst case but better performance can be expected from some configurations of input rectangles.

The algorithm was implemented and tested on various configurations of trees. The results of this empirical analysis confirmed the theoretical analysis and also demonstrated that different shapes of the trees of rectangles affect the complexity quite dramatically. In some cases the sorting and finding the forward lines dominate and in other cases finding the leaf lines and merging the forward and leaf lines are the more costly operations. Sanders *et al.* [2000a] presents detailed results of the empirical analysis.

Section 4.7.1 presented some variations of the orthogonal axial line placement problem that can be solved in polynomial time. In this section polynomial time algorithms for the orthogonal axial line placement problem for chains of orthogonal rectangles and trees of orthogonal rectangles have been presented. It is now interesting to consider finding other arrangements of rectangles that are more general than a tree of rectangles that can be solved in polynomial time. This is discussed in the next section of this thesis.

4.7.3 More general cases

The issue to be considered now is whether the results above can be extended to cover the case where each rectangle is joined to at most two rectangles at its left end and two rectangles at its right end (analogously for two rectangles above and two rectangles below). It turns out that this is not the case. This can be seen if the configuration of rectangles shown in Figure 4.44 is considered. This is a configuration which meets the restrictions – each rectangle is adjacent to at most two neighbours to the left and two to the right. Also this configuration of rectangles produces choice – the adjacency between rectangles 4 and 6 can be crossed by the line through $1-3-4-6$ or the line through $1-2-4-6$. Lines $0-1-3-4$ and $1-2-4-5$ are essential lines.

If this configuration of rectangles is treated as a basic building block it is clear that a configuration of rectangles that meets the restriction of at most two adjacent rectangles at each end can be generated and this configuration of rectangles could offer global choice. See for example Figure 4.45. A proof similar to that shown in Theorems 4.4.1 and 4.4.2 can be used to show that this restricted case of the problem is also NP-Complete in the general case.

4.8 Future research

There are a number of interesting research questions that arise from the research discussed in this chapter. Tackling all of these problems is outside the scope of this research. This section of the thesis highlights some of these questions. A more complete coverage can be found Chapter 8 of this document.

As extensions to the work done in this research and discussed in this chapter the following problems are interesting areas of research.

- Reducing the amount of work required by the heuristic algorithm to calculate

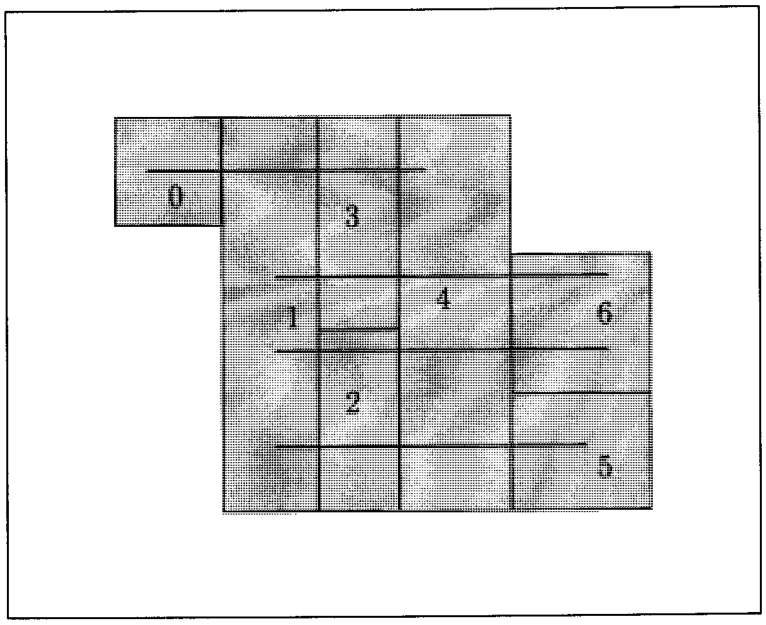

Figure 4.44: Choice introduced where each rectangle has at most 2 left and 2 right neighbours

the non-redundant set of orthogonal axial lines that covers all of the adjacencies in a configuration of adjacent orthogonal rectangles. In particular, attempting to address the issue of redundant calculations which are made for the collection of rectangles shown in Figure 4.22 or similar configurations.

- Developing other heuristics to produce approximate solutions to the exact solution.
- Considering other special cases of the problem that can be solved in polynomial time.

An interesting, but not directly related, area of further research is in the generation of test data. An efficient algorithm for generating configurations of non-overlapping adjacent orthogonal rectangles would be useful in order to test any heuristics that are developed. In addition, generating non-trivial trees of adjacent but non-overlapping orthogonal rectangles proved to be relatively complex in this research and a more efficient way of doing so could be useful for any work at improving the algorithm.

4.9 Conclusion

This chapter addresses the problem of finding the fewest longest orthogonal axial lines that pass through all of the shared adjacencies between adjacent orthogo-

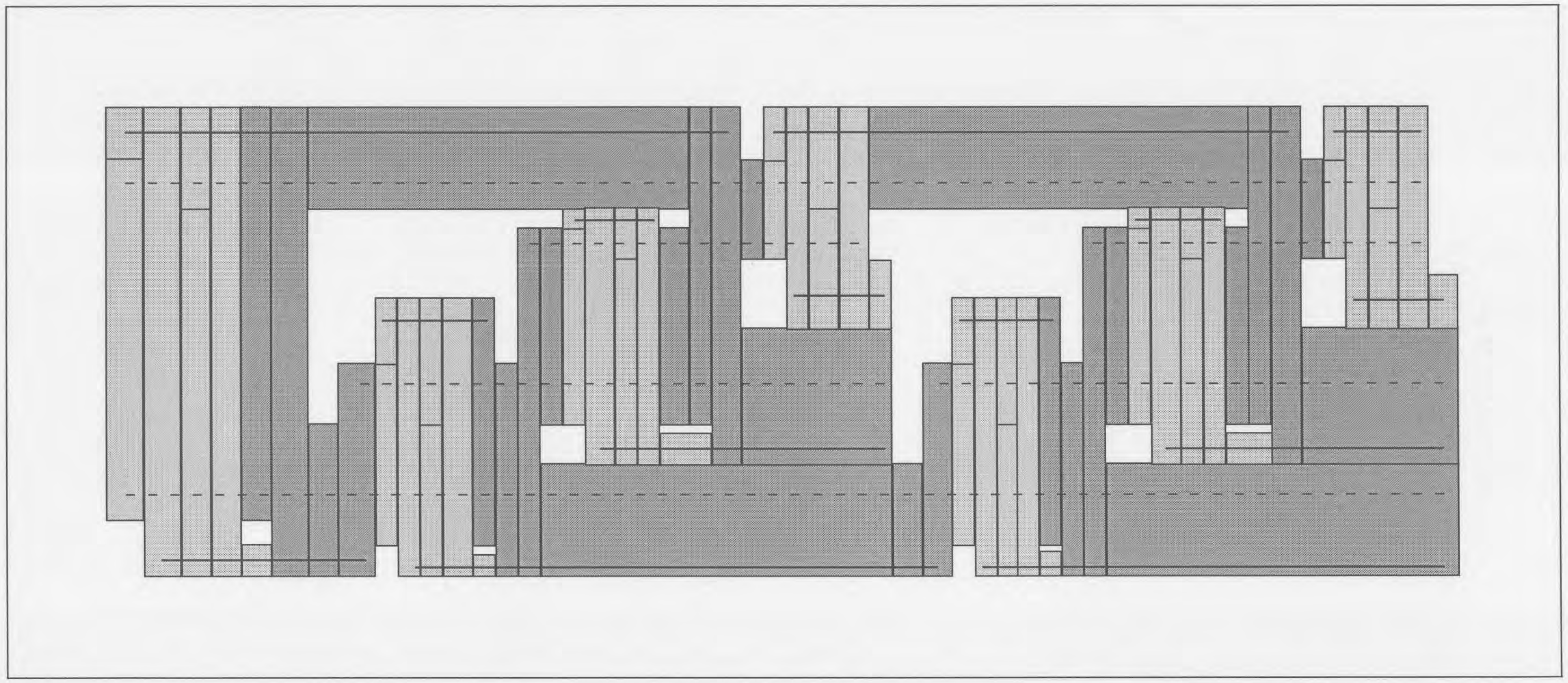

Figure 4.45: An example of choice in configuration where each rectangle has at most 2 left and 2 right neighbours

nal rectangles – ***ALP-OLOR***. The problem is made NP-Complete by the fact that various instances of choice can arise. This chapter of the thesis presents an NP-Completeness proof based on a reduction from ***biconnected planar vertex cover*** to ***stick diagram*** and hence to ***ALP-OLOR***. An $O(n^2)$ algorithm is presented that finds a non-redundant set of lines to cross all of the adjacencies of a collection of adjacent orthogonal rectangles. The algorithm has been shown to produce a very good approximation to the exact solution in all cases tested.

This chapter also presents some restrictions of the orthogonal axial line placement problem for which polynomial time solutions can be obtained – configurations that are mappable to interval graphs, chains of orthogonal rectangles ($O(n)$) and trees of orthogonal rectangles ($O(n^2)$). It is likely that other restrictions exist that are solvable in polynomial time and future research will investigate this area.

The next chapter of this thesis considers the case where the lines which pass through the shared adjacencies between adjacent orthogonal rectangles are no longer restricted to being parallel to the axes – axial lines with arbitrary orientation are acceptable. This problem is also shown to be NP-Complete and once again the use of a heuristic approach to solving the problem is discussed.