A Sequential Detailed Router for Huge Grid Graphs

Asmus Hetzel
Research Institute for Discrete Mathematics
University of Bonn
Lennéstrasse 2, 53113 Bonn, Germany

Abstract

Sequential routing algorithms using maze-running are very suitable for general Over-the-Cell-Routing but suffer often from the high memory or runtime requirements of the underlying path search routine. A new algorithm for this subproblem is presented that computes shortest paths in a rectangular grid with respect to euclidean distance. It achieves performance and memory requirements similar to fast line-search algorithms while still being optimal. An additional application for the computation of minimal rip-up sets will be presented. Computational results are shown for a detailed router based on these algorithms that is used for the design of high performance CMOS processors at IBM.

1 Introduction

As the complexity of high performance designs grows rapidly, routing is often regarded as one of the first tasks to become critical with respect to memory requirements and runtime. Especially in flat design styles there is typically no superimposed structure which makes it possible to reduce the routing task to a sequence of channel or switchbox routing problems for which very efficient algorithms are available. Instead a so called Over-the-Cell-Routing problem arises. For this, more general routing algorithms must be used which are often much more sensitive to the overall complexity of the design.

The probably most common algorithms for Over-the-Cell-Routing are sequential ones. They try to realize one connection after the other according to a priority queue, each time calling some search routine connecting given source and target areas. Previously routed connections are handled as obstacles for later ones. Rip-up and reroute or shove-aside methods are typically used for final wiring completion.

These algorithms have the great advantage to be very general but suffer from two main drawbacks: One is the sequential process which implies some greedy behaviour probably causing problems for the later routed connections. The other are memory and runtime requirements of the underlying path search routine.

In the following sections we focus on both problems. First a new exact algorithm for finding shortest paths within a threedimensional grid is presented that in practice achieves the performance and memory requirements of fast but suboptimal line search algorithms.

The problem of finding a suitable set of connections to be ripped-out and rerouted in order to provide routes to blocked components is discussed in section 3. It is shown that this problem can be tackled with a slight modification of the new path search algorithm. As a result, the problem of finding a minimum weighted rip-up set is shown to be practically feasible even for huge problem instances.

Both algorithms are incorporated in a program package called *XRouter* which was developed in cooperation with IBM-Boeblingen at the Research Institute for Discrete Mathematics in Bonn. *XRouter* has been used successfully for many designs at IBM as for example the IBM S/390 Parallel Enterprise Server - Generation 3 (see [8]). A brief overview of the detailed routing part of *XRouter* is given in section 4. Computational results concerning the detailed routing of several chips are listed in section 5.

2 Shortest Euclidean Length Paths in a Grid

Path search is potentially the most important subproblem to be solved in the type of detailed routers discussed here. More formally, a threedimensional grid $G = (V, E)$ with nodeset V and edgeset E is given as well as source and target areas $S \subset V$, $T \subset V$. The grid describes the still unused parts of the wiring area only. The goal is to find a minimum cost path between S and T according to some cost function on the nodes and/or edges.

This problem has been extensively investigated in the last decades in many different variations. But almost any algorithm belongs to one of the three following categories:

- Node-oriented Dijkstra-like labeling algorithms [4, 7, 9]. These can be implemented for very general cost functions. Their great disadvantage is that even for very restrictive special cases like uniform edge costs the running time and memory requirement is at least linear in $|V|$. Therefore it may be necessary to break up path search problems in huge designs artificially.

- Line search, line expansion or area search algorithms [5, 6]. They have been developed primarily for finding shortest rectilinear paths with respect to euclidean length in the plane which avoid a given set of rectangular obstacles. Most of them can be adopted easily to threedimensional grid problems. Their advantage is the smaller memory requirement and the fast running time at least for "simple" paths with a small number of bends. The drawback is that none of the approaches guarantees to find an optimal solution.

- Instead of using a grid model for the wiring area it can be tried to route only based on geometrical information like the shape of the blockages and the spacing requirements for the current connection. There are very efficient optimal algorithms for the twodimensional case but unfortunately the generalizations to three dimensions are theoretically and practically not better than node oriented algorithms on a full wiring grid (see [1, 3, 2]).

In the following, a new algorithm will be presented to overcome these difficulties. It computes shortest paths with respect to euclidean length within a rectangular grid achieving the performance and memory requirements of fast (but suboptimal) line search algorithms. It is less flexible than general node-oriented labeling algorithms because only euclidean length can be minimized instead of abitrary cost functions. On the other hand it is able to do exact path length optimization even on huge grids.

The algorithm works directly on a description of the grid as a collection of intervals. It is basically a labeling algorithm of Dijkstra-type where node labels $d(v)$, $v \in V$, denoting the length of the shortest path from s to v, are replaced by so-called *potential lengths* $\delta(v)$. This approach was first described by Rubin [11]. Because our algorithm can be interpreted as an adoption of Rubin's one to interval labeling, we will refer often to it. For clarity, we sketch Rubin's algorithm first:

The instance is a path search from $s \in V$ to $t \in V$ in G. We assume that positive edge lengths $l(e)$, $e \in E$, are given and that a lower bound $b(v)$ of the length of a path from v to t is known for all $v \in V$.

0) Set $\mathcal{F} := \{s\}$, $\delta(s) := b(s)$, and $\delta(v) := \infty$ for all $v \in V \setminus \{s\}$.

1) Choose a $v \in \mathcal{F}$ with $\delta(v)$ minimal. Set $\mathcal{F} := \mathcal{F} \setminus \{v\}$.

2) For all $v' \in V$ with $e = \{v, v'\} \in E$ and $\delta(v') > \delta(v)+l(e)-b(v)+b(v')$ set $\mathcal{F} := \mathcal{F} \cup \{v'\}$ and $\delta(v') := \delta(v) + l(e) - b(v) + b(v')$

3) Stop, if $v = t$ or $\mathcal{F} = \emptyset$. Otherwise goto 1).

Finally, the shortest path is constructed as usual by backtracking according to the labels.

This algorithm is not better than a standard Dijkstra-algorithm in the worst case. But due to the more directed search, it performs much faster in practice if the bounds $b(v)$ are good.

We will now use the above algorithm as a basis to derive more efficient labeling techniques on intervals.

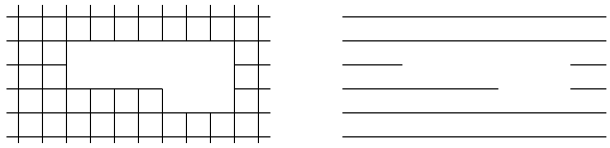

Figure 1: A plane grid and its description with horizontal intervals.

2.1 Notations

Before going into detail, we need to clarify some notations. As before let $G = (V, E)$ denote a threedimensional incomplete grid. G describes the parts of the whole routing area that are usable for the current search. For simplicity we will assume that G is an *induced* grid, i. e. for each edge $e = \{v, v'\}$ of the complete grid with $v, v' \in V$ there is also $e \in E$.

Each $v \in V$ will also be identified with integer coordinates $(x, y, z) \in \mathbb{N}^3$ in threedimensional space. So the L_1-distance of two nodes $v_i = (x_i, y_i, z_i)$, $i = 1, 2$, is well defined as

$$\|v_1 - v_2\|_1 := |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|.$$

It should be clear from the description of the algorithm wether we refer to v as a grid node or as a point in $\mathbb{N}^3$. Note that these coordinates are not derived from a numbering of the gridlines. They reflect the real position of a node on the chip area with the restriction that they must be integral.

Let $z_1 < z_2 < \ldots < z_d$ be all possible z-coordinates of nodes $v \in V$ and G_n, $n = 1, \ldots, d$, the subgrids of G that are induced by all nodes $(x, y, z_n) \in V$. We will refer to G_n as a *plane* in the following.

Furthermore we will assume that G is already organized as a special data structure: For each plane G_n there is a prefered routing direction D_n which is either horizontal (x) or vertical (y). Moreover D_n and D_{n+1} are orthogonal for all $n = 1, \ldots, d-1$. Each plane G_n is described as a set of intervals $\mathcal{I}(G_n)$ extending within preference direction only. An interval $I \in \mathcal{I}(G_n)$ represents the grid nodes it covers (which will be denoted as $V(I)$) and all nodes covered by a single interval are connected by edges (see figure 1). Note that this is sufficient to represent the grid G because edges between nodes in an induced grid are implicitely given between any pair of nodes which are adjacent in the complete grid. The set of all such intervals is denoted by $\mathcal{I}$.

Storing the grid as such a collection of intervals is typically very efficient. It holds for most design styles, that a single plane is mainly used either horizontally or vertically. So for one direction, the number of intervals representing the whole plane is much smaller than the number of grid nodes (see section 5). Additionally, this representation is easy to update during the local routing process.

We want to find a shortest path P between nodes $s, t \in V$ in G with respect to edge lengths $l(e)$, $e =$

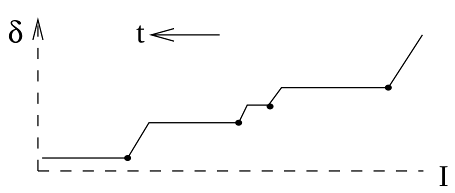

Figure 2: The function $\delta(v)$ on I. Non-redundant labels are drawn as dots. t is assumed to be left of I.

$\{v, v'\} \in E$, as follows:

$$l(e) := \begin{cases} \|v - v'\|_1 + c_n & \text{if } v, v' \in V(G_n) \text{ and} \\ & e \text{ orthogonal to } D_n \\ \|v - v'\|_1 & \text{otherwise} \end{cases}$$

where c_n, $n = 1, \ldots, d$, are nonnegative integers. So we are looking for a shortest path with respect to euclidean distance and additional extra costs for wrong-way jogs within a plane.

Looking at Rubin's algorithm, we can therefore use $b(v) = \|v - t\|_1$ as a lower bound for the length of a path from v to t for all $v \in V$.

For simplicity, we assume that the function $f(v) := \|v - t\|_1$ is monotone upon each interval. This can be achieved by splitting some intervals into at most two parts.

We discuss the algorithm bottom-up beginning with the basic labeling operation:

2.2 Interval Labeling

Throughout the algorithm, there is a set $\Delta(I) \subset V(I) \times \mathbb{N}$ associated with each $I \in \mathcal{I}$. $\Delta(I)$ is the set of labels on I and will be initialized with $\Delta(I) := \emptyset$. A label $(v, \delta) \in \Delta(I)$ indicates that there is a path from s to v in G of potential length δ. This implies that by walking from v to $v' \in V(I)$ upon I, a path of potential length

$$\delta + \|v - v'\|_1 - \|v - t\|_1 + \|v' - t\|_1$$

to any $v' \in V(I)$ exists. We call this the *induced paths* by (v, δ). A label $(v', \delta') \in \Delta(I)$ is said to be *redundant* if

$$\delta' \geq \delta + \|v - v'\|_1 - \|v - t\|_1 + \|v' - t\|_1$$

for some $(v, \delta) \in \Delta(I) \setminus \{(v', \delta')\}$. A redundant label induces no path from s to a $v \in I$ which is shorter than all paths induced by the other labels in $\Delta(I)$.

A *labeling* of I with $(v, \delta) \in V(I) \times \mathbb{N}$ is the following operation:

> Set $\Delta(I) := \Delta(I) \cup \{(v, \delta)\}$. Then remove all redundant labels from $\Delta(I)$.

This operation takes $O(|\Delta(I)|)$ amortized time: Suppose $\Delta(I) = \{(v_1, \delta_1), \ldots, (v_k, \delta_k)\}$ is non-redundant and $\delta_1 < \ldots < \delta_k$. Then for the function $\delta(v)$, $v \in I$, which gives the minimal length of all paths from s to v that are induced by elements of $\Delta(I)$ the following holds: $\delta()$ is monotone and consists of alternating segments with slope 0 and 2 (or -2 if the target is on the other side, see figure 2). So either the new label (v, δ) itself is redundant or there is a possibly empty range of indices $1 \leq m \leq n \leq k$ such that (v_i, δ_i), $m \leq i < n$, becomes redundant when adding (v, δ) to $\Delta(I)$ (see figure 3). Additionally we have $\delta_{m-1} \leq \delta \leq \delta_m$, where $\delta_0 := 0$ if $m = 1$. So assume that $\Delta(I)$ is already non-redundant and organized as a tree with keys δ_i. Then the whole operation can be performed in time $O((l + 1) \log |\Delta(I)|)$ if there are l labels in the initial $\Delta(I)$ which are redundant to (v, δ).

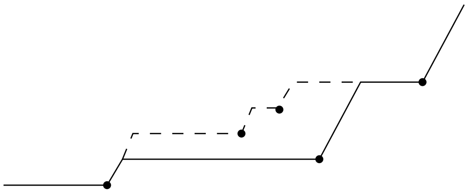

Figure 3: Change of $\delta(v)$ when adding a non-redundant label.

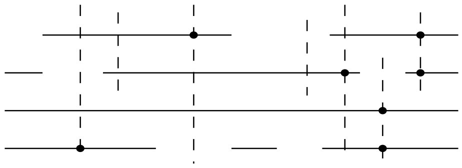

Figure 4: The horizontal intervals $\mathcal{I}(G_n)$ and the vertical intervals $\mathcal{J}$. t is assumed to be left of any interval shown. Non-redundant labeling operations are drawn as dots.

Because this is the only label generating step of the algorithm, the amortized running time is $O(\log |\Delta(I)|)$ if we keep $\Delta(I)$ permanently as an appropriate tree.

2.3 Labeling of Orthogonal Interval Sets

Let $\delta \in \mathbb{N}_0$, G_n a plane, and $\mathcal{J}$ a set of intervals on grid lines orthogonal to D_n. ($\mathcal{J}$ will be generated by projecting some parts of the intervals in planes G_{n-1} and G_{n+1} onto G_n). A labeling from $\mathcal{J}$ to G_n with value δ is defined by:

> For all (I, v), $I \in \mathcal{I}(G_n)$, $v \in V(I) \cap V(J)$ for a $J \in \mathcal{J}$ do: Label I with (v, δ).

More informally: $\mathcal{J}$ describes a set of grid points on G_n that should be labeled with δ if there is an $I \in \mathcal{I}(G_n)$ that covers that point.

Suppose D_n is horizontal and $y_1 < \ldots < y_l$ are the y-coordinates of all $v \in V$. We use standard sweep-line techniques to process all y_i where at least one $J \in \mathcal{J}$ intersects the horizontal line through y_i. Then for each such y_i we get the set $\mathcal{J}_i \subseteq \mathcal{J}$ of intervals that intersect the current horizontal line sorted by x-coordinates. Maintaining the sweepline takes $O(|\mathcal{J}| \log |\mathcal{J}|)$ time overall.

For each $I \in \mathcal{I}(G_n)$ on the horizontal line through y_i we determine the leftmost and rightmost nodes v_l and v_r where any $J \in \mathcal{J}_i$ intersects I. If they exist, we label I with (v_l, δ) and (v_r, δ) as described in the previous section. It is easy to see that all labels generated by other $J \in \mathcal{J}$ which intersect I are redundant. In fact, even only one of both labels considered is not redundant (see figure 4).

Let us assume that all intervals within $\mathcal{I}(G_n)$ on the same y-coordinate have been presorted by x-coordinates. Then the overall running time of this step is

$$O(|\mathcal{J}| \log |\mathcal{J}| + |\mathcal{I}(G_n)|(\log |\mathcal{J}| + \log \gamma)) \qquad (1)$$

where

$$\gamma := \max_{I \in \mathcal{I}(G_n)} |\Delta(I)|. \qquad (2)$$

2.4 Labeling a Plane

Let $\delta \in \mathbb{N}_0$ and G_n be a plane. For each $v \in I \in \mathcal{I}$ with $\Delta(I) \neq \emptyset$ let $\delta(v)$ be defined with respect to $\Delta(I)$ as in section 2.2. Define $\delta(v) := \infty$ for all other $v \in V$.

Labeling on G_n with value δ is the following operation:

1) Mark all $v \in V(G_n)$ with $\delta(v) = \delta$ as unprocessed.

2) As long as there is an unprocessed v: Label all intervals $I' \in \mathcal{I}$ with $v' \in I'$ such that $\{v, v'\} \in E$ and $\delta(v') > \delta(v)$ with the label (v', δ'), where

$$\delta' := \delta + l(\{v, v'\}) - \|v - t\|_1 + \|v' - t\|_1. \quad (3)$$

Set v' unprocessed if $\delta' = \delta$ and $v' \in V(G_n)$. Set v processed.

This is logically the standard labeling step of Rubin's algorithm. But avoiding redundant labeling operations results in a different time bound:

For all $I \in \mathcal{I}(G_n)$, $\Delta(I) \neq \emptyset$, let $J_I \subseteq I$ be the subinterval that covers exactly all $v \in I$ with $\delta(v) = \delta$. It follows from the monotonicity of $\delta(v)$ (figure 2) that there is either no such node (where we define $J_I := \emptyset$) or a single interval covering all of them. We now rewrite the above algorithm as follows:

0) Set $\mathcal{F} := \{I \in \mathcal{I}(G_n) \mid J_I \neq \emptyset\}$.

1) Get $I \in \mathcal{F}$ such that

$$\max_{v \in J_I} \|v - t\|_1 = \max\{\|v' - t\|_1 \mid v' \in J_{I'}, I' \in \mathcal{F}\}$$

and set

$$\mathcal{F} := \mathcal{F} \setminus \{I\}.$$

2) For all $I' \in \mathcal{I}(G_n)$ with $\{v, v'\} \in E$ for a $v \in J_I$, $v' \in I'$: Choose such a v with $\|v - t\|_1$ maximum and label I' with (v', δ') where δ' is defined according to (3). Set $\mathcal{F} := \mathcal{F} \cup \{I'\}$ if $\delta' = \delta$.

3) If $\mathcal{F} \neq \emptyset$ then goto step 1).

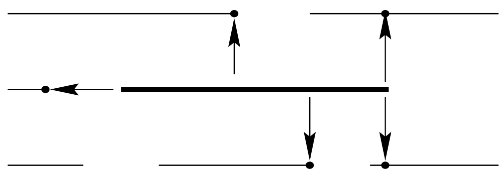

Figure 5: All non-redundant labeling operations from an interval (fat line) to neighboured intervals within the same plane. t is assumed to be left of any interval shown.

4) Let $\mathcal{L} := \{J_I \mid I \in \mathcal{I}(G_n), J_I \neq \emptyset\}$. If G_{n-1} exists, then compute δ' as in (3) for an arbitrary pair of adjacent nodes $v \in G_n$, $v' \in G_{n-1}$. Project all intervals in $\mathcal{L}$ onto G_{n-1} and label from $\mathcal{L}$ to G_{n-1} with value δ' as described in the previous section. Perform analogously if G_{n+1} exists.

The equivalence of both algorithms in terms of resulting (induced) labels on the nodes can be seen as follows: For any two neighboured intervals I, I' on the same plane there is only one non-redundant labeling from I to I' (see figure 5). This is the labeling between nodes with maximum distance to t. We simply avoid doing all other useless labelings. Additionally we label nodes on other planes with the sweepline algorithm of the previous section rather than doing it nodewise.

The running time is bounded as follows: Determining J_I for an interval I is done in time $O(\log |\Delta(I)|)$, if $\Delta(I)$ is always organized as a tree. An interval I will be inserted at most once into $\mathcal{F}$ because of step 1) and the fact that new labels with same value δ can be created only by labeling towards the target node. Therefore the total number of labelings in step 2) is bounded by the number of interval pairs sharing neighboured nodes, namely $O(|\mathcal{I}(G_n)|)$. Organizing the intervals in a suitable data structure enables us to access all neighbours of a given interval in logarithmic time. Thus the steps 0) - 3) can be performed in

$$O(|\mathcal{I}(G_n)|(\log |\mathcal{I}(G_n)| + \log \gamma)) \qquad (4)$$

with γ defined as in (2). The running time of step 4) follows from the previous section.

2.5 Overall Algorithm

Based on the operations defined in the previous sections, the complete algorithm can be formulated as follows:

0) Let $I_s, I_t \in \mathcal{I}$ be the intervals with $s \in I_s$, $t \in I_t$. Set $\Delta(I_s) := \{(s, \|s - t\|_1)\}$, $\Delta(I) := \emptyset$ for all $I \in \mathcal{I} \setminus \{I_s\}$, and $\delta := 0$.

1) Label all planes G_n with value δ as described in section 2.4. Do this according to decreasing distance to the plane containing t.

2) If $\Delta(I_t) = \{(t, \delta)\}$ then stop and return path by backtracking from t to s. If $\delta(v) \le \delta$ for all $v \in V$ ($\delta(v)$ as defined in section 2.2) then stop unsuccessfully. Otherwise compute the next minimal unprocessed label δ' with

$$\delta' := \min\{\delta(v) \mid v \in V, \delta(v) > \delta\},$$

set $\delta := \delta'$ and goto 1).

The ordering of the planes in step 1) guarantees, that processing a plane cannot create labels with value δ on an already processed one. So each plane has to be considered only once.

The correctness of the algorithm follows immediately from the fact that it can be interpreted as Rubin's one with an efficient check for redundant labels on intervals.

The complexity of a single execution of steps 1) and 2) is bounded according to (1) and (4) by

$$O(|\mathcal{I}|(\log|\mathcal{I}| + \gamma))$$

and γ can be bounded roughly by

$$\gamma \le \max\{|V(I)| \mid I \in \mathcal{I}\} \le |\mathcal{I}|.$$

The second inequality holds because otherwise a whole gridline could be removed in a preprocessing step.

If there is a path P of length L from s to t, then at most $detour(P) := L - \|s - t\|_1 + 1$ iterations of steps 1) and 2) will be performed. With another type of analysis it can be shown that the algorithm always performs as least as good as a standard Dijkstra-implementation. This requires only that the intervals in $\mathcal{I}$ are organized in advance as a suitable data structure providing logarithmic query time for the interval covering a given gridpoint. So alltogether we get a running time of

$$O(\min\{(detour(P) + 1)|\mathcal{I}|\log|\mathcal{I}|, |V|\log|V|\}) \quad (5)$$

and similary a memory requirement of

$$O(\min\{(detour(P) + 1)|\mathcal{I}|, |V|\}).$$

The term $detour(P)$ is mainly of theoretical interest, the current implementation behaves in practice like having $detour(P)$ bounded by a very small constant. This holds even in cases where no path exists at all.

Like line-search algorithms this one is theoretically fast for simple paths, i. e. paths with a small detour. But in contrast to them it guarantees optimality.

Of course this algorithm can (and must) be significantly speeded up in practice. The current implementation in *XRouter* uses among others: Fast look-up data structures for intervals, additional labeling avoidance heuristics and buckets for getting the next label δ that should be processed. Moreover it is able to cope with source and target areas covering more than a single gridnode.

3 Rip-Up and Reroute

In almost every sequential router, some strategies to overcome blockages caused by already realized connections are implemented. A common approach is rip-up and reroute. This determines a set of wires to be removed from the design such that previously blocked components can now be joined using the additional free space. Afterwards the ripped-out connections are rerouted on alternate routes if possible.

A crucial step in this approach is to determine the set of connections that should be ripped-out. If each connection is associated with a positive weight, reflecting for example criticality or reroutability, then it is natural to look for a minimum weighted set of connections which must be removed to unblock a certain component. But beside pure heuristics there is mainly one exact algorithm for computing such a minimum weighted rip-up set (MRS) [10]. This paper introduces a hypergraph in which connected unused regions (islands) of the grid form the nodes. Already wired connections form the hyperedges of the graph. A hyperedge joins all nodes such that the boundary of the corresponding island is touched by the connection corresponding to the edge.

There is a serious drawback of this model on large threedimensional grids: An artificial island for each pair of nets crossing each other in adjacent planes must be added. So the hypergraph easily has a size comparable to the number of grid nodes.

Instead it is possible to compute a MRS by using a path search algorithm with a special cost function. Let $G = (V, E)$ denote the whole routing grid including already used areas but without permanent blockages. For each $v \in V$ let $c(v)$ denote the electrical component currently using v, where $c(v) = \emptyset$, if v is unused. Let $w(c(v)) \ge 0$ the weight of $c(v)$, where $w(\emptyset) = 0$. For each edge $e = \{v, v'\}$ let

$$l(e) := \|v - v'\|_1 + \begin{cases} w(c(v)) + w(c(v')), & c(v) \ne c(v') \\ 0, & \text{otherwise} \end{cases}$$

Let $w(c)$, $c \ne \emptyset$, be greater than the euclidean length of any path to be considered during the search. Then it is easily proved, that any shortest path in G between two blocked components according to the cost function l intersects exactly with a MRS.

Moreover the search can be done with a slight modification of the algorithm given in the previous section. We simply add the additional term $w(c(v)) + w(c(v'))$ to the new label whenever we perform a labeling from a node v to a node v' with $c(v) \ne c(v')$. All techniques for non-redundant labeling apply as before with a single exception: The labeling between sets of orthogonal intervals can only be performed as described in section 2.3, if all intervals of the set $\mathcal{J}$ are projections of lines used by the same component. Therefore we have to switch to a pointwise labeling from used intervals I (i. e. $c(v) \ne \emptyset$ for all $v \in I(v)$) to nodes in adjacent planes while still doing the special orthogonal labeling for all other intervals. This causes a moderate increase of running time. It turned out to be small in practice because only few alternate routes using nodes belong-

Table 1: Problem Sizes

Chip	*Nodes*	*Intervals*	*Nets*	*Pins*
C1	83,666,883	1,005,296	35,022	115,238
C2	107,847,621	308,038	9,369	29,126
C3	48,795,267	1,367,530	51,167	192,279
C4	107,847,621	1,145,565	40,898	141,141
C5	107,847,621	2,210,793	81,395	296,225
C6	107,847,621	1,852,115	63,004	221,856
C7	107,847,621	2,367,050	82,763	256,242
C8	84,203,346	2,472,458	67,311	283,511
C9	260,564,164	4,287,534	157,440	546,512
C10	371,822,850	2,445,697	199,579	691,345

Table 2: Local Routing Statistics

Chip	*Searched Paths*	*Search Area (Nodes)* ∅	*Max.*	*Rip-Ups*
C1	80,216	449,673	39,674,500	165
C2	19,757	406,753	23,740,480	73
C3	141,675	268,255	38,364,285	1135
C4	100,865	383,408	34,144,588	736
C5	215,808	286,565	29,922,132	346
C6	158,852	416,646	44,636,072	398
C7	177,064	292,129	26,558,764	387
C8	217,054	419,872	38,566,256	1562
C9	389,074	469,889	84,391,520	1845
C10	460,695	231,332	38,813,610	6727

ing to existing connections are considered until a MRS is found.

This shows that the exact computation of a MRS can be done efficiently without a memory requirement in the order of the grid size.

4 Local Routing Algorithm

As mentioned before, we have incorporated these algorithms into the detailed routing program of *XRouter* package. This program is invoked after a global routing step in which subareas of the chip for the final layout of each net are computed. Detailed routing then tries to assign final routes within these areas sequentially to all connections. Here a connection is a wire joining two already electrically connected components of a net. For multiterminal nets the different connections may be realized at arbitrary times.

The ordering of the connections is dynamically determined according to several criteria: Timing criticality of the net, width and minimal spacing of the wires to be used for the connection, accessibility estimation for the component, estimated length of the connection, etc.

The search for a single connection is done with the shortest path algorithm as described in section 2.

If the program fails to do a connection, then a rip-up and reroute subroutine is invoked immediately. For rip-up, a MRS of connections blocking the path from source to target is determined with the algorithm described in the previous section. Then rerouting the ripped-out connections is tried immediately, which again may cause subsequent rip-ups. Several criteria ensure the termination of this recursive process after a limited number of steps. The initial state before the first rip-up is reconstructed whenever rerouting of any connection fails.

5 Computational Results

XRouter has been run successfully for a large number of designs at IBM. Performance statistics for several ones are listed in tables 1, 2, and 3. The particular chips are mostly components of multiprocessor systems in different CMOS-technologies.

Table 1 shows the complexity of the problem instances. Column 3 illustrates the improvement of an interval description of the grid compared to a nodewise one.

Table 3: CPU-Time(Seconds)

Chip	*Path Search*	*MRS Computation*	*Total*
C1	8226	38	9804
C2	1897	17	2254
C3	7894	91	10034
C4	8938	96	10792
C5	12407	67	16253
C6	16582	552	20142
C7	12042	57	15180
C8	13402	243	18982
C9	37754	691	46818
C10	31059	947	41688

Tables 2 and 3 show performance characteristics of the entire local router and important subroutines. Average and maximum sizes of the areas that have to be considered during a single path search are given in column 3 and 4 of table 2. CPU-times are measured on a RISC/6000 Model 590. The maximum memory requirement of the entire detailed router was at most 220MB for the biggest designs.

6 Summary and Conclusions

We have presented a new path search algorithm that combines the advantages of maze-running as well as line-search algorithms: It is optimal and theoretically fast for simple paths with a small detour. Moreover does the memory requirement mainly depend on the size of an interval representation of the grid which is usually much more compact than nodewise ones.

An additional application to the computation of minimal rip-up sets has been given.

Because of the small memory requirements and fast runtime this has been made a core routine of the detailed routing part of *XRouter* package. With that, even very large processor chips at IBM have been routed within hours using not more than 250 MB memory.

Next generation processors will have approximately four times the complexity of the chips today. Concerning memory requirements, the program is able to handle them without major changes due to the compact interval description of the routing grid. Additional improvements of the labeling process will be investigated in order to improve the runtime on such chips.

References

[1] K. L. Clarkson, S. Kapoor, and P. M. Vaidya. Rectilinear shortest paths through polygonal obstacles in $O(n(\log n)^2)$ time. *Proceedings of the 3rd Annual Symposium on Computational Geometry*, pages 251–257, 1987.

[2] M. de Berg, M. van Kreveld, and B. J. Nilsson. Shortest path queries in rectilinear worlds of higher dimension. *Proceedings of the 7th Annual Symposium on Computational Geometry*, pages 51–60, 1991.

[3] P. J. de Rezende, D. T. Lee, and Y. F. Wu. Rectilinear shortest paths in the presence of rectangular barriers. *Discrete & Computational Geometry*, 4:41–53, 1989.

[4] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 338–346, 1984.

[5] W. Heyns, W. Sansen, and H. Beke. A line-expansion algorithm for the general routing problem with a guaranteed solution. *Proceedings of the 17th Design Automation Conference*, pages 243–249, 1980.

[6] D. W. Hightower. A solution to line-routing problem on the continuous plane. *Proceedings of the 6th Design Automation Workshop*, pages 1–24, 1969.

[7] J. H. Hoel. Some variations of Lee's algorithm. *IEEE Transactions on Computers*, C-25:19–24, 1976.

[8] J. Koehl, U. Baur, B. Kick, T. Ludwig, and T. Pflueger. A flat, timing-driven design system for a high-performance CMOS processor chipset. *this volume.*

[9] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10:346–365, 1961.

[10] M. Raith and M. Bartholomeus. A new hypergraph based rip-up and reroute strategy. *Proceedings of the 28th Design Automation Conference*, pages 54–59, 1991.

[11] F. Rubin. The Lee path connection algorithm. *IEEE Transactions on Computers*, C-23:907–914, 1974.