General Gate Array Routing using a k–Terminal Net Routing Algorithm with Failure Prediction

Ed P. Huijbregts[1] and Jochen A.G. Jess,
Design Automation Section, Eindhoven University of Technology,
P.O. Box 513, 5600MB Eindhoven, the Netherlands

Abstract

In this paper a general approach to gate array routing is presented, based on an abstract routing space model. An efficient k–terminal net maze runner is described. It does not partition nets into 2–terminal net routing problems, but solves the problem by simultaneously growing k search waves. It is shown that the explored routing space diminishes when compared to bi–directional routing schemes. Additional routing space restriction is attained by use of variable search space restriction and due to the introduction of a dynamic routing space partitioning method based on the concept of regions. This concept allows us to determine non routable nets or parts of nets in an efficient way. Is is shown that this new partitioning method may be implemented in any maze runner without increasing the complexity of the maze runner algorithm. Result show a decreas of cpu–time up to 35%. Especially in congested routing space the cpu–time decreases significantly.

1 Introduction

We are developing a generic layout system targeted towards prefabricated devices such as gate arrays and Sea–of–Gates gate arrays [Sle90, Jes86]. The system is generic in the sense that information about the particular technology, the prefabricated geometrical structures and the chosen design strategy is considered as input. Depending on the technology, the number of wiring layers, wire–spacing rules and illegal wire–patterns (e.g. stacked vias) may differ. Prefabricated geometrical structures fall into two parts. Firstly, there is the *master–slice*, the wafer with prefabricated layout patterns such as active areas and possibly global wiring (e.g. clock–lines, power lines). Secondly a *library* of wiring patterns is usually provided with every master slice. When properly mapped onto the master slice, these wiring patterns perform a desired function at their terminal pins. More than one pattern may implement the same function, differing in shape or in the positions where it may be mapped onto the master slice. The wiring patterns heavily depend on the preprocessed structure of the master slice. The master slice determines and fixes the routing area. Possible routing areas are 1) row oriented structures, where routing channels are defined between rows of active area, 2) island oriented structures, where routing areas are defined around islands of active areas and 3) Sea–of–Gates organisation, where the entire space is filled with active area and routing space is defined only after the predefined wire patterns are placed.

We describe the chosen technology, master slice and library of wiring patterns using a specially developed language GADL (Gate Array Description Language [Lip87]). This description is mapped onto

1. This work is supported by Foundation F.O.M., project nr. EEL88.1428

an abstract routing space model (presented further on in this paper) which is stored in a common database [Boo90]. On this database several placement and routing algorithms are defined. Currently two placement algorithms are implemented based on simulated annealing [Ott84] and eigenvalue decomposition [Fra86].

To cope with the diversity of possible routing spaces (variable number of layers, different technologies and master slice structures), we are in need of a flexible router. The router must be capable to route through densely packed areas thereby ensuring that the wiring is distributed smoothly over the area while minimizing the total netlength. To overcome the first problem we have chosen to incorporate a Lee–type maze runner. Although the Lee algorithm [Lee61] stems from the early sixties, it is still used in many routing systems to lay out the wiring. This is mainly because the algorithm will always find a path if one exists and the path it finds will always have minimum possible cost. A drawback of the Lee algorithm is that it searches the entire routing space until a target is reached. To reduce the space searched, many modifications were suggested such as bi–directional search [Poh89], the use of a predicted path cost function to guide the search [Har68, Rub74], or by iterative search space restriction [Tad80]. Several strategies are implemented in our system to limit the routing space. First the routing area is partitioned into several smaller areas or *cells*. A global router plans a route through the cells for each net (see figure 1). Both the partitioner and the global router will try to minimize the total wire length and congestion of nets over the chip area. To route the nets in each cell we developed a new maze runner that routes all terminals of a net at once. We will show that by doing so the explored routing space is reduced. If it does not succeed in routing all nets as indicated during the global routing phase, it will try to find a detour using neighbouring cells as depicted in figure 2

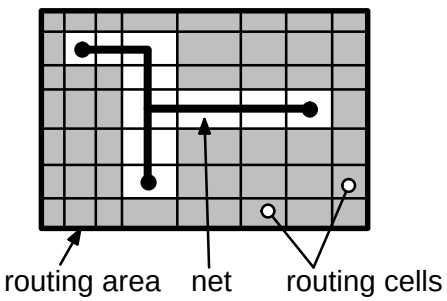

Figure 1: Area partitioning and global routing

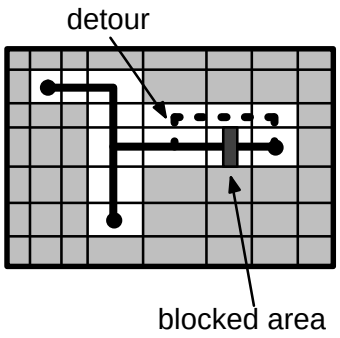

Figure 2: Finding a detour using neighbouring cells.

Essentially this comes down to variable search space restriction as suggested in [Tad80].

A new mechanism is introduced to partition the routing space into so called *unconnected regions*. It enables the identification of non–routable nets or parts of nets without actually trying to route them. Furthermore it offers the opportunity to further reduce the search space. As will be shown this evaluation may be done very fast compared to the actual routing and will decrease the running time of the overall algorithm significantly.

Organisation of paper

The organization of the paper is as follows. In section 2 the routing model will be explained and some definitions are given that are used throughout the paper. Section 3 describes the global routing phase.

In section 4 our *k*–terminal routing algorithm will be described. In section 5 the new failure prediction mechanism is introduced and is described how to benefit from it. Section 6 describes some implementational aspects, and finally results and conclusions are given in section 7 and 8.

2 Layout modelling

Routing grid

The routing space can be seen as a collection of points in $\mathbb{R}^3$. For routing purposes it is fair to restrict the routing space to a discrete3–dimensional space $\mathbb{N}^3$, due to the discreteness of the chip manufacturing equipment. The wiring patterns generated by the router are restricted to be aligned with the Cartesian coordinate axes. This *Manhattan style* layout simplifies the routing problem significantly with respect to both memory usage and implementation of the algorithms.

The above restrictions allow us to represent the routing space as a 3–dimensional undirected *grid graph*. A grid graph G(V,E) is a 3–dimensional $(X \times Y \times Z)$ array of vertices. A vertex v may be denoted by its coordinates (x, y, z), where $0 \leq x \leq X$, $0 \leq y \leq Y$ and $0 \leq z \leq Z$. Points having the same z–coordinate constitute a plane or *layer*. Usually the number of layers Z is small compared to X and Y, giving the routing space a quasi–planar flavour. Two vertices (x_1, y_1, z_1) and (x_2, y_2, z_2) are called *neighbours* if $|x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2| \equiv 1$. We denote the set of neighbouring vertices of a vertex v by $N_v(v)$, where $N_v : V \rightarrow 2^V$. Between each pair of neighbours an edge exists. Thus each vertex may have up to six edges called *north, east, south ,west, up* and *down* as depicted in figure3. The up and down edges are used to represent vias, the contacts between wiring patterns on adjacent layers. From now on an edge between neighbouring vertices v and w is denoted by $e(v,w)$.

Layout modelling

To limit the number of layers needed in the grid graph, only layers in which actual routing may take place are described. To catch the wiring patterns that occur in these layers as defined by the master slice and the library elements, both vertices and edges are assigned status labels using the functions $S_e : E \rightarrow \{\text{INITIAL}, \text{INHIBIT}, \text{IMAGE}, \text{ROUTER}\}$, and $S_v : V \rightarrow \{\text{BLOCK}, \text{FREE}\}$. A vertex is labelled FREE if all one of its edges are labelled either INITIAL or INHIBIT. Otherwise it is labelled BLOCK. The status of an edge may be one of four. INITIAL indicates that the edge is not used in any wiring pattern and thus is free for routing. An edge status INHIBIT indicates that the edge is not used in any wiring pattern and may never be used for routing purposes. Finally, an edge has status IMAGE if it is part of a wiring pattern. Edges belonging to wiring patterns generated during detailed routing are assigned a fourth edge status label ROUTER (see figure4

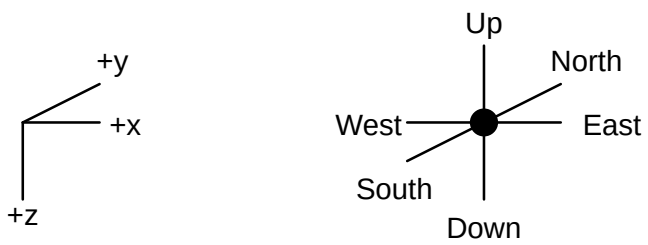

Figure 3: Orientation within grid.

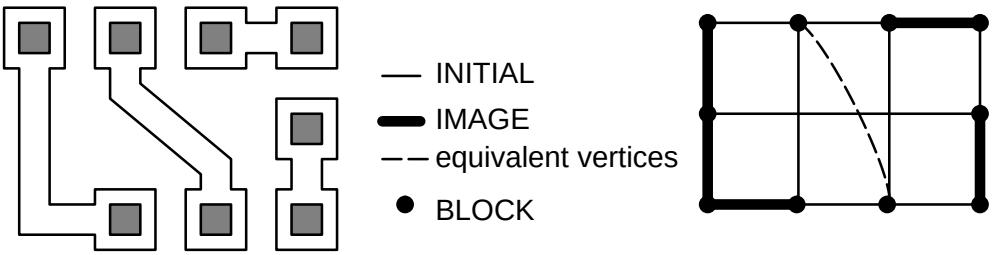

Figure 4: Mask patterns and grid fill–ins

for an example).

Although the wiring patterns generated by the router are restricted to Manhattan style patterns, this does not hold for the wiring patterns that occur on the master slice or in the library. To enhance wiring patterns deviating from the Cartesian coordinate axes, we introduce an equivalence relation between vertices. Two vertices of the grid graph are defined to be electrically equivalent if they represent two positions in the wiring space that are unconditionally on the same potential level, and this relation is not established in the grid graph by some setting of the edge status labels.

All vertices that are connected by edges labelled IMAGE or ROUTER are assumed to be on the same potential level, meaning that a router may connect to one of those vertices to create a connection with all of them. Of course, by definition this also holds for all vertices reachable via the equivalence relation.

To guide the detailed router in its search for minimum cost paths the user may specify a cost for each edge of the grid graph. Therefore an edge labelling function C_e is introduced that assigns a cost to each edge, where $C_e : E \rightarrow \{1, 2, ..., c_{max-1}\}$. The cost $C(P)$ of a path $P(v_1, v_2, .., v_n)$ with $v_i \in V$ and $e(v_i, v_{i+1}) \in E$ is defined as the sum of the costs of all edges $e(v_i, v_{i+1})$ along the path. Define the distance between two sets of vertices U and W as $d(U, W) = \min\{C(P(u, w)) | u \in U \wedge w \in W\}$. Using the above definitions the path consistency property [Rub74] holds which says: Let P_1 be any minimum cost path from u to v, and P_2 be any minimum cost path from v to w. Then P_1P_2 has minimum cost among all paths from u to w that pass through v.

Implementational aspects

Edges are not explicitly described but located at the vertices. Each vertex contains its north, east an down edge. Thus if we want to inspect the north edge of vertex (x,y,z) we simply look at the north field of this vertex, wheres if we want to inspect its south edge we look at the north field of vertex (x,y–1,z).

Since three edges are located at a vertex, it is neccesary to add three edge cost labels per vertex. In practice however the diversity of the occuring triples of cost labels is not very large. Therefore the triples are stored in a table and the index of the triple is stored at the vertex. The gain is twofold. Firstly, only one label per vertex is needed instead of three. Secondly, triples describing the same edge cost labels for the three edges are only stored once in the table.

To store the equivalence relation between vertices efficiently, it is important to know they occur due to predefined wiring pattern. Usually these wiring patterns are repeated several times over the master slice, and thus the number of different equivalence patterns will not be large. Since vertices are denoted by their tuples (x,y,z), an equivalence set is implemented as a circular list of these tuples. This allows

to store these sets as offsets between the equivalent vertices rather then using their absolute coordinates. The circular lists are stored in a table and at the vertices we keep the index of the table in which the corresponding list is held and an offset to correctly access the position in the list, see figure 5

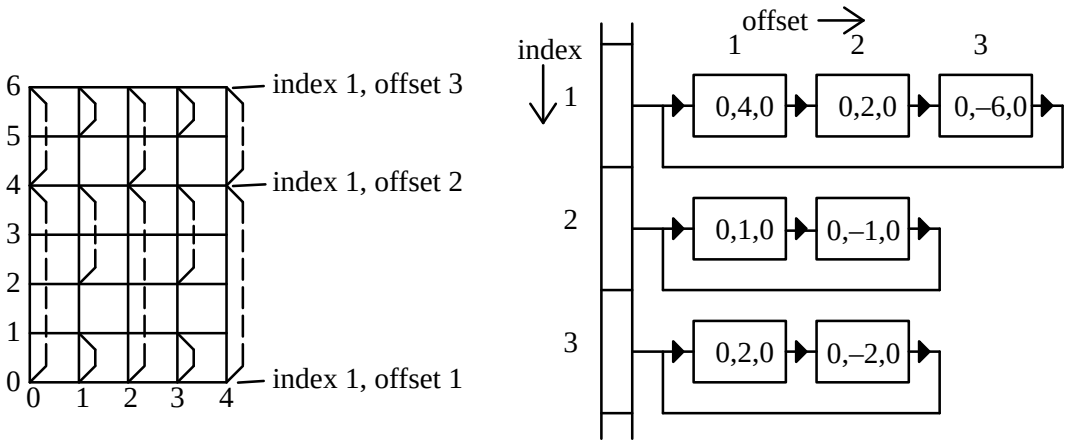

Figure 5: Efficient storing of equivalence relations using a table.

.

3 Global routing

As described previously, the routing space is partitioned into a rectangular grid of cells. Each cell covers a part of the routing space and therefore encloses a number of terminals. A wiring capacity is given for each boundary of a cell, denoting the number of wires the boundary can accomodate. The global routing problem is to assign a route through the cells for each net connecting all cells that enclose its terminals, such that the number of wires passing any boundary does not exceed its capacity. From [Saeijs86] it follows that routing obstacles should preferably be placed at the boundaries of the cells, leaving the center of the cells free for routing. These obstacles may be caused by predefined wiring patterns or local edge cost functions. The definition of the cells is therefore design dependent. It will be clear that both the number of cells, their sizes and boundary capacities are essential for the validity of the global routing result. Using small cells the global router will try to solve the detailed routing problem, whereas few large cells will not help the detailed router much. Thus target cell sizes also depend on (and are determined by) the capacity of the detailed router to solve the routing problem for each cell. Too high capacity values will result in local wire congestion and thus non routable situations. Too low values will result in unnecessary detouring of nets and an increase of the overall net length. We have chosen to estimate the cell boundary capacities by counting the number of vertices where nets are allowed to cross a specific boundary, such that they can at least penetrate the cell over a predefined number of edges. The above definition of the global routing problem makes it easy to incorporate various global routing algorithms with minor modifications [Tin83, Li84, Par89]. Currently, the hierarchical wire router of [Bur83] is implemented.

4 k–terminal nets

An often used way to cope with k–terminal nets is to partition them into several 2–terminal nets that are solved sequentially, e.g. [Poh69]. The resulting connections depend heavily on this partitioning. In

this section an algorithm is presented that overcomes the partitioning step by routing all n terminals at once. We show that all connections found are valid minimum cost paths and that the area explored is always equal to or smaller than the area explored by a bi–directional router.

Search waves

For routing purposes, the vertex labelling function is extended to $S_v : V \rightarrow \{BLOCK, FREE, START, EQ, N, E, S, W, U, D\}$. Here a vertex is labelled START if it denotes a vertex belonging to a terminal of a net routing problem. A vertex is labelled EQ if it is reached during routing through the equivalence relation. A vertex is labelled N (E, S, W, U or D) if it is reached during routing from its north (east, south, west, up or down) neighbour.

Suppose that we want to determine minimum cost paths between vertex v and all other vertices. The router will start from vertex v, called the *root*, and explores the neighbouring vertices of v. Following the definitions given in section 2, a neighbouring vertex w may be used for routing purposes if $S_v(w) = FREE$ and if the edge through which w is reached from v is labelled INITIAL. A neighbour w for which this holds is called a *possible extension* of v. The ordered tuple (v,w) is called an *extension*. Define a function expand(v) that stores all extensions of v into a priority queue keyed by the cost of the relevant edge. (For more detailed information on how an extension is stored in the priority queue the reader is refered to section 6). If all extensions of a vertex are inserted in the priority queue it is said to be *expanded*. By repeatedly extracting from the priority queue the cheapest extension and expanding it, new vertices will be reached. The set of vertices thus reached is called a *search wave*. The set of edges through which a new vertex is reached constitute a minimum cost path from the newly reached vertex to the root. To be able to trace such a path efficiently, a newly reached vertex w changes its status label $S_v(w)$ from FREE to one of N, E, S, W, U, D, representing the direction from which the neighbouring vertex has been reached. Notice that changing the status label of a vertex prevents it from being expanded more than once and hence no selfloops are generated during search wave expansion (see figure 6

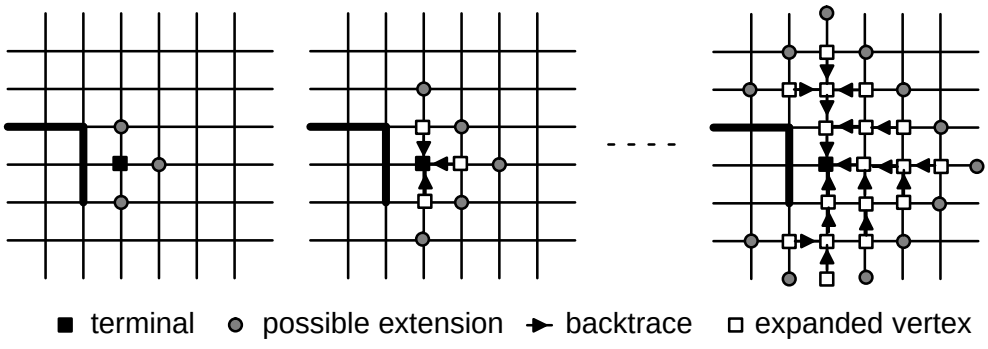

Figure 6: Search wave expansion around one root vertex.

). A special situation occurs if a vertex is reached through the equivalence relation. Such a vertex v is labelled $S_v(v) := EQ$. If we come upon such a vertex when backtracing a path we inspect all equivalent vertices. Only one of them will have a label set to one of N,E,S,W,U,D or START, and from that vertex we continue our trace. For the rest of the paper we will not go into detail with respect to these situations.

Define the *wave front* wf belonging to search sw the set of vertices $v \in sw$ that have at least one extension.

To distinguish between search waves grown around different roots, another vertex labelling function $T_v : V \rightarrow \mathbb{N}$ is defined. It assigns a *terminal number* i to each vertex of search wave sw_i. Whenever an extension $[v,w)$ is extracted from the priority queue, the terminal number of the search wave is propagated according to $T_v(w) := T_v(v)$.

k–terminal routing

Given a routing problem consisting of k terminals $T_1, T_2, ..., T_k$, each terminal consisting of one or more vertices. (From section 2 it follows that storing one vertex for each terminal of a net will suffice. All other vertices of a terminal can be reached by walking along edges labelled ROUTER or IMAGE or through the equivalence relation). The vertex status is set to $S_v(v) := START$, and the terminal number is set to $T_v(v) := i$ for each vertex $v \in T_i, 1 \leq i \leq k$. About each terminal search waves are grown. Whenever two search waves *meet* at some vertex, a minimum cost path is found connecting the two generating terminals. In order for different search waves to meet the above definition of a possible extension should be changed: vertex w is a possible extension of vertex v if the status of the edge through which w is reached is INITIAL and

$$S_v(w) = FREE, \text{ or} \tag{1}$$

$$S_v(w) \in \{START, EQ, N, E, S, W, U, D\} \wedge T_v(v) \neq T_v(w). \tag{2}$$

The second situation occurs if two search waves started from different terminals meet and a connection is found. This implies that search waves started from the same terminal will never meet. Search waves will grow until two of them meet and a minimum cost path is found. The path is traced back and all edges along the path are labeled ROUTER to indicate that a wiring pattern is found. Both merged search waves are deleted, while all other search waves remain unchanged. The vertices on the newly found path, together with the set of vertices of the merged terminals form a new terminal. A search wave is started around this new terminal and all unchanged search waves continue to expand untill another path is found. The above is repeated untill all terminals are connected.

Theorem 1: The presented algorithm will find minimum cost paths.

Proof: Suppose two vertices u and v are to be connected and suppose a search wave sw_u of u already exists. Let $wf_{u,0}$ denote the current wavefront of sw_u. Start a search wave sw_v from vertex v and expand sw_u and sw_v until they meet at vertex x. A minimum cost path $P(v,y)=P(v,x)+P(x,y)$, where $y \in wf_{u,0}$ is found. Since $y \in wf_{u,0}$, a minimum cost path $P(y,u)$ is known. Using the path consistency property it follows that $P(v,u)=P(v,y)+P(y,u)$ is a minimum cost path. ■

Suppose that a routing area is presented by an orthogonal grid with edge cost 1 for all edges. Let a routing problem be defined on the grid that consists of n vertices that are to be connected such that each vertex connects to the set of previously connected vertices. Label the vertices in the order in which they are connected. Let $V_i, 1 < i \leq n$ be the set of vertices $V_i = V_{i-1} \cup P(w, v_i)$, w where is an arbitrary vertex $w \in V_{i-1}$ and $V_1=\{v_1\}$, then first V_1 connects to v_2, V_2 connects to v_3 and in general V_i connects to v_{i+1}.

Theorem 2: For given types of problems the number of vertices reached using a multi–directional router is always equal to or smaller than the number of vertices reached using a bi–directional router.

Proof: Let d_i denote half of the distance between the sets of vertices V_i and $\{v_{i+1}\}$, i.e. $d_i = d(V_i, \{v_{i+1}\})/2$. Because of the order of connection of the vertices for given problems it is obvious that $d_i < d_{i+1}$ for all $1 \leq i < n$. Let the area functions $f_i(x)$ represent the number of vertices reached by a search wave started from i connected vertices with wavefront cost x.

Bi–directional search: Search waves are generated around v_1 and v_2. The search waves meet when they have expanded d_1. The number of vertices reached F_1 is thus $F_1 = 2f_1(d_1)$. Next V_2 connects to v_3 which takes $F_2 = f_2(d_2) + f_1(d_2)$. In the i^{th} step $F_i = f_i(d_i) + f_1(d_i)$ vertices are reached and thus the total number of vertices reached is:

$$N_{bi} = \sum_{i=1}^{n-1} F_i = \sum_{i=1}^{n-1} f_i(d_i) + f_1(d_i). \tag{3}$$

Multi–directional search: Search waves are generated around each vertex. Two of them will meet when they have expanded d_1. The number of vertices reached is then $F_1 = f_1(d_1) + (n-1)f_1(d_1)$. Let $a_0 = 0$, $a_i = d_i - a_{i-1}/2$ and $b_i = d_i + a_{i-1}/2$. A new search wave is started for V_2 and all search waves will expand until the next two meet. Since the other search waves did not change, the distance to be covered is $a_2 = d_2 - d_1/2$ and $F_2 = f_2(a_2) + (n-2)(f_1(b_2) - f_1(d_1))$. The term $-f_1(d_1)$ represents the initial area of an unchanged search wave already taken into account in F_1. In the i^{th} step $F_i = f_i(a_i) + (n-i)(f_1(b_i) - f_1(b_{i-1}))$ vertices are reached and thus the total number of vertices reached is:

$$N_{multi} = \sum_{i=1}^{n-1} F_i = \sum_{i=1}^{n-1} f_i(a_i) + f_1(b_i). \tag{4}$$

To prove that $N_{multi} \leq N_{bi}$ we have to show $f_i(a_i) + f_1(b_i) \leq f_i(d_i) + f_1(d_i)$ for all $1 \leq i < n$. Using area functions of the form $f_i(x) = c_1x^2 + c_2xD_i$, where $D_i = 2\sum_{j=1}^{i-1} d_j$ as depicted in figure 7, it is easy to see that $f_i(a_i) + f_1(b_i) \leq f_i(d_i) + f_1(d_i)$ holds for $1 \leq i < n$ if and only if $c_2D_i - c_1a_{i-1} \geq 0$. Using $a_{i-1} = \sum_{j=1}^{i-1}(-1/2)^{i-1-j}d_j$ this leads to $\sum_{j=1}^{i-1} 2c_2 - (-1/2)^{i-1-j}c_1 \geq 0$ which is true if and only if $2c_2 \geq c_1$, which is the case.

■

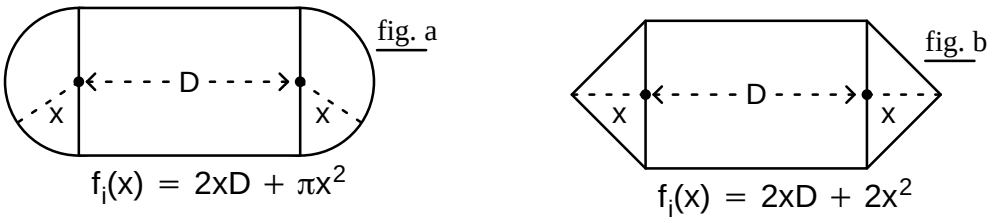

Figure 7: Examples of area functions: a. 2–dimensional Euclidian space, b. 2–dimensional orthogonal grids with edge cost 1.

5 Routing space partitioning

The global router defines a routing problem for each routing cell as described in section 3. Each routing problem consists of a number of nets that are routed one after the other by the detailed router. Suppose we want to route the cell given in figure 8

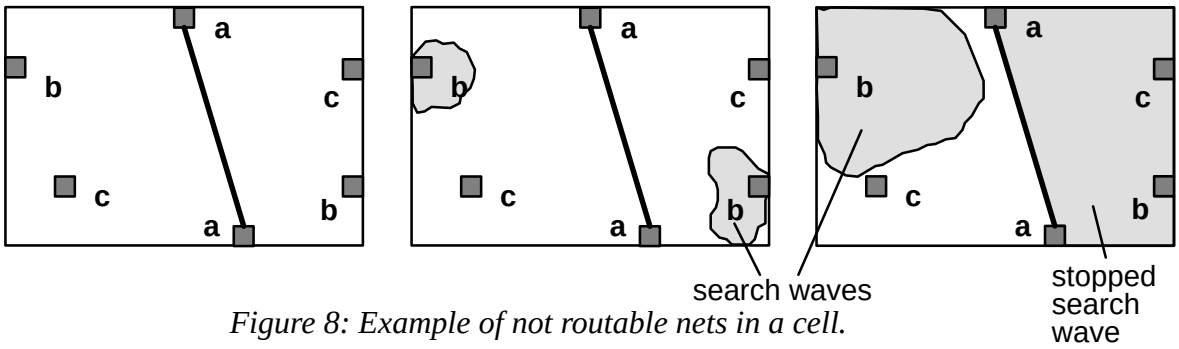

Figure 8: Example of not routable nets in a cell.

. It contains three nets **a**, **b** and **c** that will be routed in this order. Clearly, after net **a** is routed, net **b** and **c** can not be routed. In practical situations however, e.g. if more routing layers are available, this is not that easy to see. Therefore we usually proceed by routing net **b** and **c** which both fail to route. If we had observed that the routing problems of net **b** and **c** are topologically identical, we would not have started routing net **c** since already net **b** failed to route. In this section we describe a mechanism that enables us to recognize these situations and describe how to benefit from this.

The maze runner, as presented in section NO TAG, will grow search waves about each terminal vertex of a net. At each step an extension is added to a search wave and becomes part of the wave front. Search waves will thus grow until two of them meet and a path is found. If no such path exists, some search wave will run out of extensions eventually. This is detected by the algorithm and search wave expansion is stopped. The set of vertices covered by the search wave that ran out of extensions is called a *region*.

Define the set of non–terminal vertices as $V_{non} = \{v \in V | S_v(v) = FREE\}$, and the set of terminal vertices $V_{trm} = V \backslash V_{non}$. Define $R_v : V_{non} \rightarrow \mathbb{N}$ to denote the region of a non–terminal vertex and define a function $g_1 : V \times V \rightarrow B$ as $g_1(v,w) = (\forall_{u \in N_v(v), z \in N_v(w)} \; R_v(u) \neq R_v(z))$. Using the labelling function R_v, all vertices of a stopped search wave are assigned a unique *region number.*

Lemma 1: Let $v, w \in V_{non}$. There exists no path $P(v,w)$ if $R_v(v) \neq R_v(w)$.

Proof: By contradiction. Suppose there exists a path $P(v,w)$ and $R_v(v) \neq R_v(w)$. Then there are two vertices $x, y \in P(v,w)$ such that $N_v(y)=x$, $R_v(x) = R_v(v)$ and $R_v(y) = R_v(w)$. This implies that x must have been a possible extension of the search wave enclosing y at the time of creation of region $R_v(y)$ and should therefore be in the same region. Hence $R_v(v)=R_v(x)=R_v(y)=R_v(w)$ which is a contradiction. ■

Lemma 2: Two terminal vertices $v, w \in V_{trm}$ are not connectable if $g_1(v,w)=true$, i.e. if none of their neighbouring vertices are in the same region.

Proof: By contradiction. Suppose a path $P(v,w)$ exists and $g_1(v,w)=true$. $P(v,w)$ implies the existence of a path $P(x,y)$, where $x, y \in V_{non}$, $x \in N_v(v)$ and $y \in N_v(w)$. Using lemma Lemma 1 this implies $R_v(x)=R_v(y)$, which is in contradiction with $g_1(v,w)=true$, hence no path exists.
■

Define the function $g_2 : 2^V \times 2^V \rightarrow B$ as $g_2(T_i, T_j) = (\forall_{v \in T_i, w \in T_j} \; g_1(v,w))$, which says that two terminals T_i and T_j are not connectable if none of their vertices are connectable.

Corollary 1: A net is not routable if $\exists_{T_i \in T} \forall_{T_j \in T,\ i \neq j}\ g_2(T_i, T_j) = \text{true}$, i.e. there is at least one terminal T_i that cannot be connected to any other terminal.

Define the routability graph as a bipartite graph $G_r = (T_r \cup R_r, E_r)$, where T_r contains a vertex t_i for every terminal, R_r contains a vertex for every region r_j and $E_r = \{e(t_i, r_j) | \exists_{v \in T_i} \exists_{w \in N_v(v)}\ R_v(w) = r_j\}$. The number of edges is bounded by $|E| \leq |T_r||R_r|$. According to corollary 1 a net might be routable if and only if all vertices $t \in T_r$ are pairwise reachable. Notice that no connections can be made in regions containing only one or no terminals. Thus we may restrict routing space by not starting search wave expansion in these regions.

Figure 9

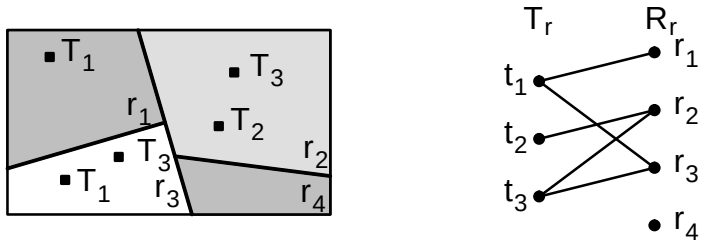

Figure 9: A net routing problem and its matching routability graph.

shows an example of a routing area and the corresponding routability graph. Notice that terminals T_1 and T_3 consist of two electrically equivalent vertices each, i.e. a connection is established if one of the vertices is connected. Using the routability graph we decide that the net might be routable and that no search wave expansion is started in regions 1 and 4.

The complexity of the described tests is as follows. The assignment of a region number to the vertices of a stopped search wave can be done when deleting search waves, because every vertex is visited during search wave deletion. Since the assignment takes constant time the complexity of the algorithm does not change. To construct the graph we have to evaluate the neighbours of all terminal vertices V_m of a net m. The number of neighbours of a vertex is bounded to 6 in our grid graph. Hence, since there are $|V_m|$ terminal vertices this can be done in $O(|V_m|)$.

```
procedure predict_failure()
begin
    construct Gr;
    select a v ∈ Tr ∪ Rr; dfs (v);
    for all t ∈ Tr
        if t is not reached by dfs stop;
    for all r ∈ Rr
        if degree(r) < 2
            delete terminal vertices situated in region r;
end
```

Algorithm 1: The above procedure decides if a net is not routable.

The routability test can be done in $O(|T_r|+|R_r|+|E_r|)$ by doing a depth first search starting from one of the terminal vertices of the graph. If all terminal vertices are reached then the net is routable. Finally, search wave expansion should be started in all regions containing more than one terminal (see algorithm 1). This can be tested in $O(|R_r|)$. Thus the total complexity is $O(|V_m|+|T_r||R_r|)$.

6 Implementational aspects

Priority queue

Whenever a frontier vertex is expanded all its possible extensions are stored in an array A (see also [Hoe76]). Each entry A holds a list of possible extensions. Suppose A is infinitely long. All extensions [w,u) of a search wave with root v are stored in $A[C(P(v,w))+C_e(w,u)]$. We keep an array index n such that for all $n_1 < n$, $A[n_1]$ is empty. If the extension list of $A[n]$ is empty, n is updated by $n := n + 1$. Initially $n=0$. Hence $A[n]$ always holds the list with the cheapest extensions. Suppose that an extension $[w,u)$ is taken from $A[n]$, then $n=C(P(v,w))+C_e(w,u)=C(P(v,u))$. Vertex u becomes a frontier vertex and is expanded itself. All extensions $[u,x)$ are stored in $A[C(P(v,u))+C_e(u,x)]$, which is the same as $A[n+C_e(u,x)]$. Since the cost of an extension is bounded by c_{max}, entries $A[n_1]$ with $n_1 \geq n+c_{max}$ are always empty and we only need an array of length c_{max} by addressing it through $n := n \bmod c_{max}$.

Detection of stopped search waves

Whenever the search wave of a terminal runs out of extensions, the terminal cannot be connected to any other terminal. Hence, the net routing problem cannot be solved and search wave expansion may be stopped. To detect this, an auxiliary counter $aux[i]$ is kept for each terminal T_i. Initially $aux[i] = 0$ for all terminals. Whenever an extension of T_i is inserted into the priority queue, $aux[i]$ is incremented. $aux[i]$ is decremented if an extension of T_i is extracted from the queue. Now suppose that extension [v,w) with $v \in T_i$ is taken from the queue and $aux[i]$ becomes zero. Then it is easy to see that search wave expansion may be stopped if and only if $aux[i]$ remains zero after vertex w is expanded.

Invalid extensions

When two search waves meet a connection is found between the terminals to which the search waves belong. The sets of vertices of the terminals together with the vertices along the path found will form a new terminal. If not all terminals of the current net are connected a new search wave will be generated starting from all vertices of this newly formed terminal. However, extensions belonging to the merged search waves may still reside in the priority queue at this time. Since these search waves are deleted, their extensions are no longer valid. To distinguish between valid and non–valid extensions, we again use the array $aux[\,]$ without violating the function as presented in the previous paragraph. Now, when two terminals T_i and T_j are connected $aux[i]$ and $aux[j]$ are set to a negative value. Thus whenever an extension [v,w) is extracted from the priority queue it is a valid extension if and only if $aux[T_v(v)] \geq 0$. This will only work correctly if the search wave is started around the newly formed terminal is given a new unique terminal number. Since a net with k terminals is connected after $k–1$ connections are found, implying that $k–2$ new terminals are generated, the auxiliary array should have $2k–2$ entries.

An efficient implementation

```
procedure expand (v) /* insert all extensions of v in queue */
begin
    for all neighbouring vertices w of v
        if S_v(e(v, w)) = INITIAL and S_v(w) ≠ BLOCK and T_v(v) ≠ T_v(w)
            put [v,w) in queue; aux[T(v)]++;
end

procedure start_search_wave (v,i) /* start search wave for vertices connected with v */
begin
    S_v(v) := START; T_v(v) := i; expand (v);
    for all neighbouring vertices w of v
        if S_v(w) = BLOCK and S_e(e(v, w)) = IMAGE or ROUTER
            start_search_wave (w,i);
end

procedure extension () /* extract cheapest valid extension from queue */
begin
    repeat
        extract [v,w) from queue;
    until aux[T_v(v)] > 0;
    aux[T_v(v)] := aux[T_v(v)] + 1;
end

procedure traceback (v) /* traceback found path and create new terminal */
begin
    while S_v(v) ≠ START
        let w be the neighbour reached from v according to S_v(v);
        S_e(e(v, w)) := ROUTER;
        S_v(v) := START;
        v := w;
end

procedure delete_search_wave (v) /* delete search wave started around v */
begin
    for all neighbouring vertices w of v
        if w is reached from v according to S_v(w)
            R_v(w) := region;  /* only executed when search wave stopped */
            S_v(w) := FREE;
            delete_search_wave (w);
end
```

```
procedure route (T1, ..., Tk) /* route k-terminal net */
begin
    predict_failure ();
    A := empty_set; new := 1;
    for all Tnew select an arbitrary vertex u
        aux[new] := 0; start_search_wave (u, new); new := new + 1;
    repeat
        [v,w) := extension ();
        if Tv(v) ≠ Tv(w)
            traceback (v); traceback (w);
            Se(e(v, w)) := ROUTER; aux[Tv(v)] := − 1; aux[Tv(w)] := − 1;
            delete_search_waves (v);
            if new = 2k-1 stop;
            else
                aux[new] := 0; start_search_wave (v, new); new := new + 1;
        else
            Sv(w) := backtrace to v; Tv(w) := Tv(v); expand (w);
    until aux[Tv(w)] = 0;
    for all terminals select an arbitrary vertex u
        delete_search_waves (u);
    region := region + 1;
end
```

Algorithm 2: Efficient implementation of the new multiple terminal net router using failure prediction.

7 Results

The presented algorithm was coded in C and implemented on a HP/Apollo DN425 workstation. Table 1 describes the testcircuits taken from the MCNC benchmark set.

circuit	area	#modules	#nets
alu3	74x74	80	90
co14	60x74	79	93
dk17	85x72	109	119
5xp1	80x92	129	136
misex2	115x110	209	233

Table 1: Description of testcircuits.

All circuits are mapped onto a Sea–of–Gates gate array [Vee90] using two routing layers. To exhibit the profit derived from the failure prediction algorithm, each circuit is placed on the gate array such that over 95% of the available area is covered by modules, ensuring congested routing areas. Two placement algorithms are used, one based on eigenvector decomposition [Fra86] (table 2), the other on simulated annealing [Ott89] (table 3).

The routing algorithm is run with and without failure prediction. For each circuit the number of nets that failed to route, the number of nets for which search wave expansion did not start and the cpu–time used are given in columns 3 to 5. Column 6 describes the gain achieved by our algorithm using failure prediction. As the results show our failure prediction algorithm is significantly faster even for relatively small circuits. Notice that the gain increases as the circuit's complexity increases.

circuit	predict	#failed	#not started	time	gain
alu3	no	20	–	104s	
	yes	9	11	87s	16%
co14	no	41	–	63s	
	yes	22	19	53s	16%
dk17	no	52	–	99s	
	yes	22	30	78s	21%
5xp1	no	58	–	147s	
	yes	24	34	118s	20%
misex2	no	134	–	384s	
	yes	25	109	250s	35%

Table 2: Routing results using a placement based on eigenvector decomposition.

circuit	predict	#failed	#not started	time	gain
alu3	no	10	–	101s	
	yes	6	4	89s	12%
co14	no	18	–	53s	
	yes	8	10	46s	13%
dk17	no	22	–	98s	
	yes	10	12	80s	18%
5xp1	no	29	–	186s	
	yes	11	18	134s	23%
misex2	no	112	–	352s	
	yes	30	92	255s	28%

Table 3: Routing results using a placement based on simulated annealing.

8 Conclusions

In this paper a general approach to gate array routing is presented, based on an abstract routing space model. An efficient k–terminal net maze runner is described. It does not partition nets into 2–terminal net routing problems, but solves the problem by simultaneously growing k search waves. It is shown that the explored routing space diminishes when compared to bi–directional routing schemes. Additional routing space restriction is attained by use of variable search space restriction and due to the introduction of a dynamic routing space partitioning method based on the concept of regions. This concept allows us to determine non routable nets or parts of nets in an efficient way. Is is shown that this new partitioning method may be implemented in any maze runner without increasing the complexity of the maze runner algorithm. Result show a decreas of cpu–time up to 35%. Especially in congested routing space the cpu–time decreases significantly.

References

[Boo90] W.A.P.M.P. Boogers, ”Redesign of the GADL compiler,” *M.Sc. thesis*, Eindhoven University of Technology, The Netherlands, 1990.

[Bur83] M. Burstein and R. Pelavin, ”Hierarchical Wiring of Gate–Array VLSI Chips,” *Proceedings European Conefrence on Circuit Theory and Systems*, pp. 198–202, 1983.

[Fra86] J. Frankle and R.M. Karp, ”Circuit placement and cost bounds by eigenvector decomposition,” *Proceedings International Conference on Computer Aided Design*, pp. 414–417, Santa Clara, 1986.

[Har68] P.E. Hart, N.J. Nilsson and B. Raphael, ”A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions of Systems Science and Cybernetics*, vol. SSC–4, no. 2, pp. 100–107, July 1968.

[Hoe76] J.H. Hoel, ”Some Variations of Lee’s Algorithm,” *IEEE Transactions on Computers*, vol. c–25, no. 1, pp. 19–24, January 1976.

[Jes86] J.A.G. Jess and A.G.J. Slenter, ”The prototype of an open design system for gate arrays”, *ESPRIT ’86, Results and Achievements*, pp. 541–550, 1986.

[Lee61] C.Y. Lee, ”An algorithm for path connections and its applications,” *IRE Transactions on Electronic Computers*, vol. EC–10, pp. 346–365, September 1961.

[Li84] J. Li and M. Marek–Sadowska, ”Global Routing for Gate Arrays,” *IEEE Transactions on Computer–Aided Design*, vol. CAD–3, no. 4, pp. 298–307, October 1984.

[Lip87] P.E.R Lippens and A.G.J. Slenter, ”GADL: A Gate Array Description Language,” *M.Sc. thesis*, Eindhoven University of Technology, The Netherlands, 1987.

[Ott89] R.H.J.M. Otten and L.P.P.P. van Ginneken, ”The Annealing Algorithm,” Kluwer Academic pulishers, 1989.

[Par89] T. Parng and R. Tsay, ”A New Approach to Sea–of–Gates Global Routing, ” *Proceedings IEEE International Conference on Computer Aided Design*, pp. 52–55, 1989.

[Poh69] I.S. Pohl, ”Bidirectional and heuristic search in path problems, *Ph.D. dissertation*, Stanford University, Stanford, California, 1969.

[Rub74] F. Rubin, ”The Lee Path Connection Algorithm,” *IEEE Transactions on Computers*, vol. c–23, no. 1, pp. 907–914, September 1974.

[Sae86] R.W.J.J. Saeijs, ”Simultaneous Placement and Global Routing for Gate Arrays,” *M.Sc. thesis*, Eindhoven University of Technology, The Netherlands, 1986.

[Sle90] A.G.J. Slenter, ”A Generalised Approach to Gate Array Layout Design Automation”, *Ph.D. dissertation*, Eindhoven University of Technology, The Netherlands, 1990.

[Tad80] F. Tada, K. Yoshimura, T. Kagata and T. Shirakawa, ”A fast maze router with iterative use of variable search space restriction,” *Proceedings of the ACM*, pp. 250–254, 1980.

[Tin83 B.S. Ting and B.N. Tien, ”Routing Techniques for Gate Array,” *IEEE Transactions on Computer–Aided Design*, vol. CAD–2, no. 4, pp. 301–312, 1983.

[Vee90] H. Veendrick, D. van de Elshout, D. Harberts and T. Brand, ”An Efficient and Flexible Architecture for High–Density Gate Arrays,” *IEEE International Solid–State Circuits Conference*, pp. 86–87, 269, February 1990.