Steiner Tree Optimization for Buffers, Blockages, and Bays

Charles J. Alpert*, Gopal Gandham†, Jiang Hu‡, Jose L. Neves†

Stephen T. Quay† and Sachin S. Sapatnekar‡

*IBM Austin Research Lab, †IBM Microelectronics, ‡Department of ECE, University of Minnesota

Abstract

Timing optimization is a critical component of deep sub-micron design, and buffer insertion is an essential technique for achieving timing closure. This work studies buffer insertion under two types of constraints: (i) avoiding blockages, and (ii) inserting buffers into predetermined buffer bay regions. We propose a general Steiner tree routing problem to drive this application and present a maze-routing based heuristic. We show that the combination of our Steiner tree optimization with effective buffer insertion techniques leads to useful solutions on industry designs.

1 Introduction

It is now widely accepted that interconnect is becoming increasingly dominant over transistor and logic performance in the deep submicron regime. Buffer insertion has become a critical step in modern VLSI design methodologies (see Cong *et al.* [6] for a survey). Indeed, Cong [5] shows that as gate delays decrease while chip dimensions increase, the number of buffers inserted will increase with advancing technologies. He expects that close to 800,000 buffers will be required for 50 nanometer technologies. There is a critical need to automate the entire interconnect optimization process in order to achieve timing closure.

Several works have studied the delay driven-buffer insertion problem. Closed form solutions have been proposed in [1] [3] [4] [7]. Van Ginneken's algorithm [21] has become a classic in the field. His dynamic programming algorithm finds the optimal buffer placement under the Elmore delay model. Several extensions to this work have been proposed (e.g., [2] [14] [15] [16] [19]). All of these works (except for [16] [19]) assume that a Steiner tree is given and that buffers must be placed along the Steiner wires. The works of [16] [19] also perform routing of the tree during buffer insertion but do not consider blockages.

When attempting to insert buffers into a hierarchical design (which are becoming increasingly common due to the complexity explosion), buffers may not be placed on top of pre-existing macros; these regions are called *blockages*. If the existing Steiner tree has been routed almost entirely over blockages, then any buffer insertion algorithm that uses the routing topology fails to find a solution. Figure 1(a) shows an example 2-pin net whose route runs over a large blockage, thereby making buffer insertion infeasible. If one re-routes the tree as in Figure 1(b), then buffers can be inserted, albeit for an additional wire length cost.

In this methodology, the Steiner tree serves as a guide for buffer insertion, but does not represent the final route. The actual routing is performed after buffer insertion. Figure 1(c) shows how the global router may re-route the newly created nets while considering delay, noise, congestion, etc. Without this final step, the regions of the chip without blockages would become unnecessarily congested with interconnect.

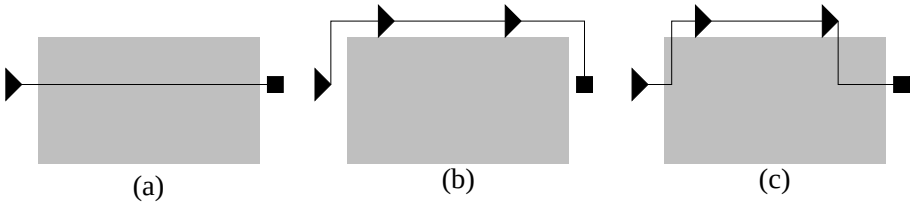

Figure 1: A tree routed over blockage (a) can be re-routed around the blockage, which enables buffer insertion (b), thereby yielding a better solution than in (a). The newly created nets can then be routed over blockage (c).

Figure 2 illustrates an example where the best Steiner tree construction avoids some, but not all, of the blockages. In (a), the existing route is completely blocked for buffering, while in (b), the re-routed tree avoids all blockage, allowing buffers to be inserted. However, the most efficient solution is shown in (c) which avoids only some of the blockage.

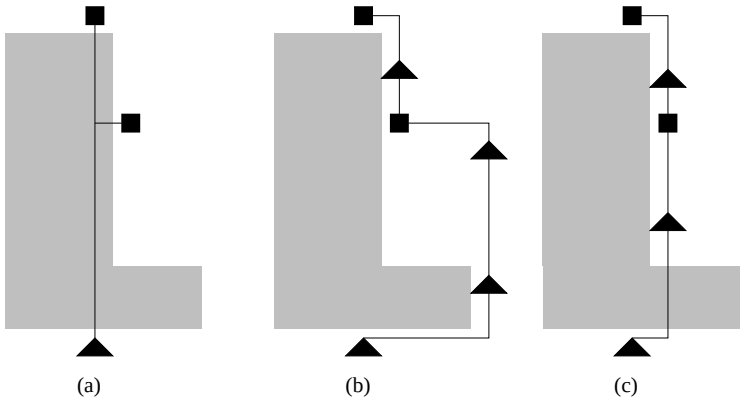

Figure 2: A net routed totally over blockage (a) prevents any buffer insertion; avoiding all blockage (b) requires three buffers; the best solution (c) avoids only some blockage, permitting two buffers to be inserted.

The problem of buffer insertion in the presence of blockage constraints has been recently addressed in [12] and [22]. The method of [12] optimizes the routing tree topology and inserts buffers simultaneously. While it obeys blockage constraints, the method makes no effort to avoid blockages. The approach of [22] allows routing over some blockages while avoiding others. Their algorithm uses maze routing and dynamic programming techniques to find the buffered path with minimum delay (while obeying blockage constraints). However, the algorithm is only applicable to two-pin nets.

In some design methodologies, it may be suitable to pre-allocate space for buffers during floorplanning, rather than trying to squeeze buffers between large blocks during physical design, which can cause both logical and wiring congestion. We call these pre-allocated regions *buffer bays*. For this methodology, the entire layout area is viewed as blockage except for the buffer bays. Figure 3(a) shows an example of a two-pin net that does not cross any buffer bays and is thus totally blocked from buffer insertion. By re-routing the tree through a buffer bay (b), buffers can be suitably inserted (c).

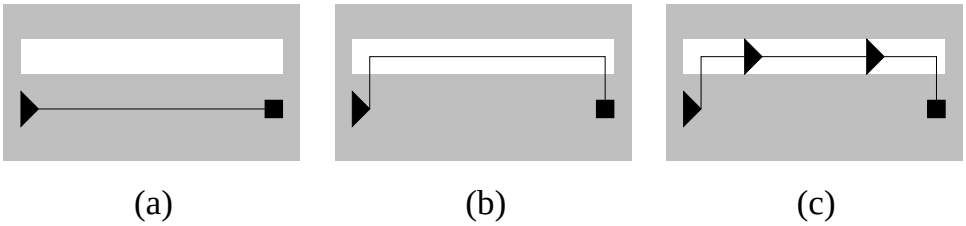

Figure 3: A totally blocked tree (a) can be re-routed through a buffer bay (b) which enables buffer insertion (c).

We make the following contributions:

- We propose a general Steiner tree problem formulation for buffer insertion with either blockage or buffer bay constraints.
- We present a new Steiner tree optimization that derives a heuristic solution to this problem. The algorithm iteratively rips up a sub-path of an existing Steiner tree and uses maze routing to re-connect the two remaining sub-trees.
- We employ a customized grid graph and sparsify it appropriately, so that it can accommodate an efficient solution search without significantly altering the quality of the results. We also utilize a branch-and-bound technique to further improve the computational efficiency.
- We show that for real industry designs, our Steiner tree heuristic, when used with an effective buffer insertion algorithm, results in more useful solutions than a Steiner tree heuristic that does not account for blockages.

In contrast to the works of [12] [19] [22], which simultaneously insert buffers during routing, we first construct the Steiner tree, then inserts buffers. The simultaneous approach is arguably superior considering that one cannot design the best tree until buffer locations are known. However, the simultaneous operations of tree construction and buffer insertion necessitate that the buffering component be somewhat simplistic. The buffer insertion tool that we adopt has a wide user base and has several sophisticated features. It can (i) handle a library of inverting and non-inverting buffers [15], (ii) simultaneously fix noise, slew and capacitance violations, (iii) be run with higher-order gate and interconnect delay computations [2], (iv) trade-off the number of buffers inserted with solution quality, simultaneously perform wire sizing, and (vi) insert buffers to conform with the net's hierarchical structure. It is neither prudent nor necessarily feasible to integrate a simultaneous Steiner tree construction while maintaining both the features and performance of the tool as it currently exists.

2 Problem Formulation

Given a unique source so and a set of sinks SI, a *rectilinear Steiner tree* (RST) $T(V, E)$ is a spanning tree in the rectilinear plane that connects every node in $V = \{so\} \cup SI \cup W$, where W is a set of additional nodes. W typically includes two types of nodes: (i) *internal Steiner nodes* of degree three or four, denoted by the set IN, and (ii) *corner nodes* of degree two that connect a horizontal and vertical edge, denoted by the set CO. We add a third node type to W: a *boundary node* (belonging to the set BY) has degree two, an incident edge lying over blockage, and an incident lying in a blockage-free region. For example, the RST in Figure 4 shows a Steiner tree with source $so = s$ and sinks $SI = \{d, i, k\}$. All other nodes are in W with $b \in IN$, $g, j \in CO$, and $a, c, e, f, h \in BY$.

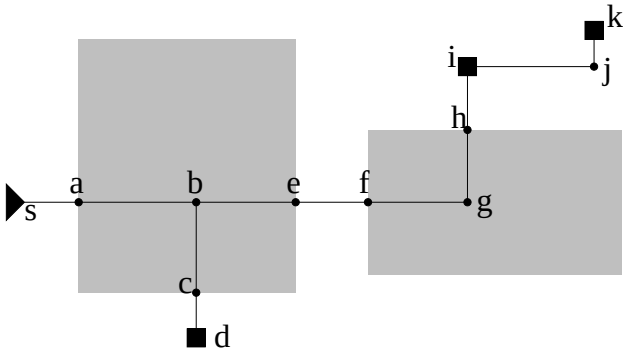

Figure 4: A Steiner tree illustrating different node types.

Definition: A *2-path* of a tree $T(V, E)$ is a path $p(u, v) = \{(u, v_1), (v_1, v_2), \ldots, (v_m, v)\} \in T$ such that $\{v_1, \ldots, v_m\} \subseteq BY \cup CO$ and $u, v \in \{so\} \cup SI \cup IN$.

Every tree T can be uniquely decomposed into a set of 2-paths, e.g., the tree in Figure 4 can be decomposed into four 2-paths: $p(s,b), p(b,d), p(b,i)$ and $p(i,k)$.

A *rectangle* r has a unique *bounding box* $(x_1, y_1), (x_2, y_2)$, where $x_1 \leq x_2$ and $y_1 \leq y_2$. Given a set of rectangles B (i.e., the blockage map), an edge $e \in E$ to said to be *inside* B (denoted by $e \in B$) if there exists a rectangle $r \in B$ such that both endpoints of e lie inside the bounding box of r. Let l_e denote the length of edge e. Our problem formulation is as follows:

Dual Region Rectilinear Steiner Tree Problem: Given a parameter α, a source so, a set of sinks SI, and a set of rectangular blockages B, construct a Steiner tree $T(V, E)$ with $\{so\} \cup SI \subseteq V$ that minimizes

$$cost(T(V,E)) = \sum_{e \in E} l_e + \alpha \sum_{e \in B} l_e \tag{1}$$

The parameter α represents the degree of the penalty for routing over blockage. This problem is NP-Complete by reduction to the Rectilinear Minimal Steiner tree problem (setting $\alpha = 0$). Observe $cost(T(V,E))$ can be expressed as the sum of the costs of all 2-paths in T, where the cost of a 2-path is given by:

$$cost(p(u,v)) = \sum_{e \in p(u,v)} l_e + \alpha \sum_{e \in B \cap p(u,v)} l_e \tag{2}$$

For example, if $\alpha = 1$, then edges that intersect blockage have twice the cost of the other edges. Recall the re-route in Figure 1. If the wire length more than doubles when changing the route from (a) to (b), then (a) is the lower cost solution. The appropriate value for α depends on the technology, though empirically a value of one seems to work well.

An advantage of this cost function is that it can be used to handle both buffer bays and blockages. If B represents a set of buffer bays, then routing over rectangles in B should actually reduce the cost function. Chooses α between -1 and 0 achieves this effect For example, if $\alpha = -\frac{1}{2}$, then the cost of routing outside a buffer bay is twice that of routing inside a buffer bay.

Equation (1) is just one possible objective function. One could also incorporate, e.g., the maximum length over all sub-paths that intersect blockage, the sum of the squared lengths of these sub-paths (scaled), or actual path delays into the objective. More sophisticated objectives may be better suited for buffer insertion, but more difficult to incorporate into an optimization.

3 The Grid Graph Construction

Our Steiner tree heuristic is based on maze routing, which is appealing since it can handle multiple cost functions. Maze routing approaches have been used elsewhere in recent research, e.g., [11] [22], but it

can be inefficient since it may search over numerous locations, many of which do not lead to worthwhile solutions.

A fundamental notion in maze routing is the concept of a *grid graph*, $G(V_G, E_G)$. A grid graph can be viewed as a tessellation of rectangular tiles with V_G being the set of tile centers and E_G being edges that connect tile centers. A grid graph can be uniquely induced by the sets $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ of sorted non-duplicate coordinates. The *induced grid graph* $G(V_G, E_G)$ from X and Y has vertices $V_G = \{(x, y) \mid x \in X, y \in Y\}$, and edges $E_G = \{((x_i, y), (x_{i+1}, y)) \mid 1 \leq i < |X|, y \in Y\} \cup \{((x, y_i), (x, y_{i+1})) \mid 1 \leq i < |Y|, x \in X\}$.

Typically a uniform grid graph is utilized, which forces a routing algorithm to spend an equal amount of time searching each part of the routing area. For our purposes, this is wasteful due to the non-uniform distributions of sinks and blockages. We propose using a non-uniform grid graph, which allows high density channels in difficult routing areas and low density channels elsewhere. Assume that some low-cost RC tree T has already been computed over $\{so\} \cup SI$. Our grid graph is a superset of the *Hanan grid* [9] for T.

We require that no buffer can be placed within a distance of less than M units from a blockage, where the value of M is half the width (or height if greater than width) of the largest buffer. Therefore, we add routes that surround each blockage by this prescribed offset parameter M, as shown in Figure 5. Similarly for buffer bays, the offsets are added internally to each region, allowing sufficient room for buffers.

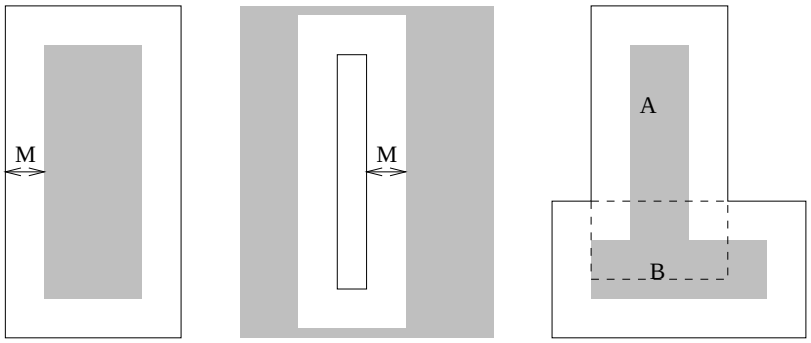

Figure 5: Using M to compute usable tracks (a) around blockages, (b) inside buffer bays and (c) with overlapping blockages. Dashed lines are tracks that are infeasible for buffer insertion.

We construct a grid graph according to the procedure shown in Figure 6. Step 1 initializes sets X and Y to be empty, and Step 2 adds the coordinates of each tree node into X and Y. Step 3 adds the coordinates of the blockages, and Steps 4-5 construct the grid graph induced by X and Y. Finally, Step 6 sets the attribute $blocked(e)$ for each edge e in G. If e overlaps with a blockage in B or does not overlap with a buffer bay in B, then the attribute is set to $true$; otherwise, it is set to $false$. We refer to this grid graph

as the Extended Hanan Grid (EHG). An example grid graph constructed from a 3-pin net and a single blockage is shown in Figure 7. Note that the EHG uniquely depends on the net being optimized, i.e., the Steiner tree heuristic is carried out only on its own customized grid graph.

Procedure Grid_graph(T, B)

Input: Steiner tree $T(V, E)$, set of rectangles B
Output: Grid graph $G(V_G, E_G)$

1. Set $X = \emptyset, Y = \emptyset$.
2. For each $v \in V$ with coordinates (x, y),
   $X \leftarrow x \cup X$, $Y \leftarrow y \cup Y$.
3. For each $r \in B$ with bounding box $(x_1, y_1), (x_2, y_2)$
   If r is a blockage
   $X \leftarrow (x_1 - M) \cup (x_2 + M) \cup X$,
   $Y \leftarrow (y_1 - M) \cup (y_2 + M) \cup Y$.
   If r is a buffer bay
   $X \leftarrow (x_1 + M) \cup (x_2 - M) \cup X$,
   $Y \leftarrow (y_1 + M) \cup (y_2 - M) \cup Y$.
4. Sort the coordinates in X and Y
5. Generate induced grid graph $G(V_G, E_G)$ from X, Y.
6. $\forall e \in E_G$, compute value of *blocked*(e).

Figure 6: The Grid_graph procedure.

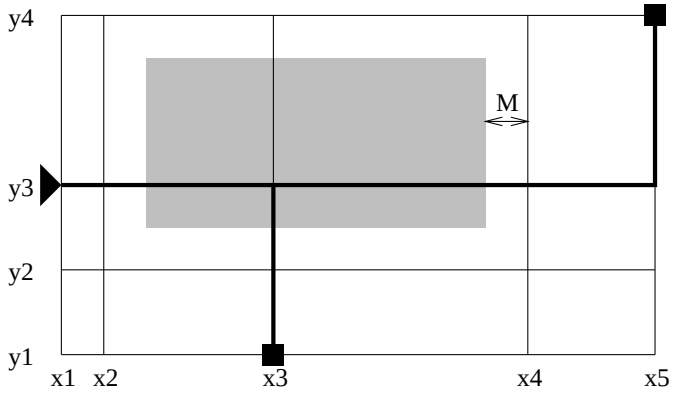

Figure 7: The grid graph constructed according to the Grid_graph procedure in Figure 6 for an example 3-pin net and a single rectangular blockage.

Since the EHG may be sparse in some regions, a natural question to ask is whether any loss in optimality is incurred by considering only tracks on the EHG and neglecting the large spaces off the EHG. It can be shown that *a minimum cost 2-path which connects two disjoint subtrees that are embedded on the EHG also lies on the EHG.* Consequently, it is reasonable to restrict the solution search to the EHG.

4 Algorithm Description

4.1 Overview

Our algorithm first decomposes the existing Steiner tree into disjoint 2-paths and computes each 2-path cost. It then iteratively chooses a poorly routed 2-path, removes it, and re-routes it. The 2-path with the highest cost is not necessarily the most poorly routed path, as the highest cost path could simply be a very long path. We identify poorly routed 2-paths with the highest value of $cost(p(u,v))/l_{p(u,v)}$. Such a 2-path has the highest ratio of wire length routed over blockage to total wire length which implies room for improvement.

Steiner_Tree Algorithm (T, B)
Input: $T(V,E)$, a Steiner routing tree B, rectangles representing blockages or bays **Output:** Re-routed Steiner tree T
1. $G(V_G, E_G)$ = Grid_graph(T, B) (see Figure 6). 2. Compute the set P of disjoint 2-paths in T. Compute cost of each 2-path in P from Equation (2). 3. While $P \neq \emptyset$ 4. Choose $p(u,v) \in P$ with max $cost(p(u,v))/l_{p(u,v)}$. 5. Remove $p(u,v)$ from T and P to get two sub-trees. Label sub-tree containing so as T_s; the other is T_t. 6. Find 2-path $p(q,w)$ = Maze_routing(G, T_s, T_t). 7. Add the edges in the 2-path $p(q,w)$ to T.

Figure 8: The Steiner tree construction algorithm.

A complete description of the algorithm is given in Figure 8. Step 1 computes the underlying grid graph for T and B. Step 2 finds the set of all 2-paths, and Steps 3 and 4 iterate through these 2-paths, each time picking the one with the highest overlap cost. The selected 2-path is removed in step 5, which induces two subtrees T_s and T_t. Step 6 performs the maze routing which returns a minimum cost 2-path between T_s and T_t, and Step 7 re-connects the tree using this 2-path. We now explain the maze routing performed in Step 6.

4.2 Maze Routing

The path re-connecting two subtrees is found via maze routing. The original maze routing algorithm [17] runs on a grid graph and makes point-to-point connections. Each grid edge is assigned a cost such as edge length for unblocked and infinity for blocked edges. Maze routing is equivalent to Dijkstra's shortest path

algorithm [8] applied on the grid graph. The source node is initially assigned zero cost, and then wave expansion proceeds out from the source, labeling all intermediate nodes until the target node is reached. The grid node labels reflect the routing cost from the source. For a linear cost function, maze routing guarantees the least cost path for connecting two points. The primary variation of our algorithm is we wish to find the lowest cost path between subtrees as opposed to unique points. This is accomplished by labeling all nodes in the source tree with zero cost and stopping when any node in the target tree is reached.

Maze_routing(G, T_s, T_t) **Algorithm**

Input: Underlying grid graph $G(V_G, E_G)$.
Disjoint RSTs T_s and T_t embedded in G.

Output: 2-path $p(q, w)$ with $q \in T_s, w \in T_t$

1. $\forall v \in V_G$, set
   $label(v) = \infty$, $visited(v) = false$, $parent(v) = \emptyset$
2. For each node $v \in T_s$
   Set $label(v) = 0$ and set $Q = Q \cup \{v\}$.
3. While $Q \neq \emptyset$
4. Let $v \in Q$ with minimum $label(v)$.
   Delete v from Q. Set $visited(v) = true$.
5. For each u, such that $(u, v) \in E_G$, $u \neq parent(v)$
   $newLbl = label(v) + l_{(u,v)}$.
   If $blocked(u, v)$ then $newLbl = newLbl + \alpha l_{(u,v)}$
6. If $newLbl < label(u)$ then
   $label(u) = newLbl$, $parent(u) = v$.
7. If $visited(u) = false$ and $u \notin T_t$, insert u into Q.
8. Find node $w \in T_t$ such that $label(w)$ is minimum.
9. Find path $p(q, w)$ from w to $q \in T_s$ by tracing back *parent*. Return $p(q, w)$.

Figure 9: Algorithm for maze routing connecting two subtrees

The complete procedure is shown in Figure 9. Step 1 initializes three arrays, *label*, *visited*, and *parent* for each node in the grid graph. The $label(v)$ value is the cost of the best path from a node in T_s to v, the $visited(v)$ value indicates whether v has been explored, and $parent(v)$ is stores the best path to v. Step 2 initializes the labels of all nodes in T_s to zero and puts them into a priority queue Q. Steps 3-7 search the grid graph by iteratively deleting the node v with smallest label from Q and exploring that node. Each neighbor node u of v is explored in Steps 5-6, and the label for u is updated according to length of edge (u, v) and whether edge (u, v) is blocked. If the new label, corresponding to a path to u through v, is less than the previous label, the label is updated and v becomes the parent for u. Steps 8-9 find the node with the smallest label in the target tree, and uncover the path back to the source tree by following the *parent*

structure.

4.3 Complexity Analysis

Given a tree $T(V, E)$ and a set of blockages B, let $n = |V|$ and $k = |B|$. The size of the grid graph is $O((n+k)^2)$ so the procedure Maze_routing has complexity $O((n+k)^2 \log(n+k))$. The number of times this procedure is called is bounded by $O(n)$, which means the complexity for the entire algorithm is thus $O(n(n+k)^2 \log(n+k))$.

5 Improving Efficiency

The high time complexity of the algorithm suggests that one can speed up the algorithm without necessarily sacrificing solution quality. We have employed two speedup techniques, a sparsified grid graph construction and branch-and-bound maze routing, that together improve runtimes by more than a factor of ten.

5.1 Sparsified Grid Graph

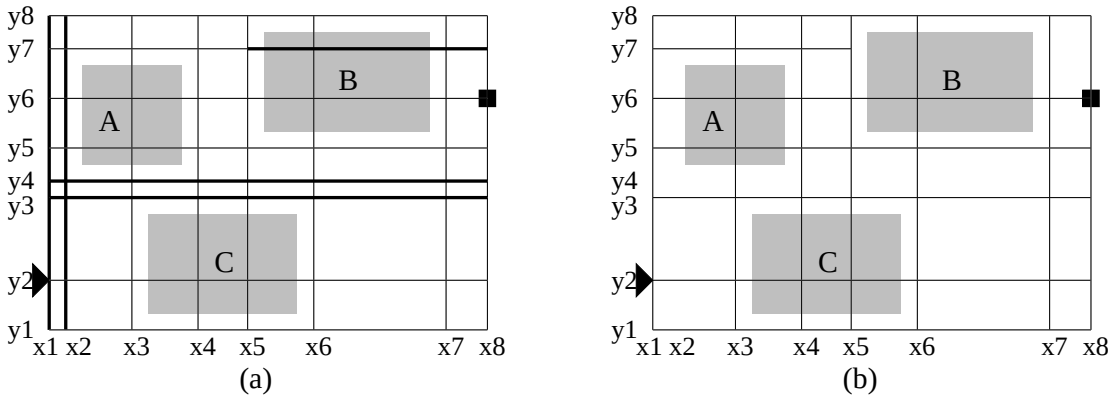

Figure 10: The original grid graph (a) has (shown in bold) two pairs of redundant tracks and a severable track. The sparsified grid graph (b) has neither redundant nor severable tracks.

When $|B|$ is large, the induced grid graph can be dense. Nearby blockages that do not line up can cause several edges in the grid graph to be extremely close together. A routing tree construction could choose any of these edges and result in essentially the same tree. A *track* is a set of edges all with the same x or y coordinate. Given a step size, such as 0.1 mm, two parallel tracks are called *redundant* if they are closer than the step size and if at least one of them does not intersect a net pin (source or sink). Given two redundant tracks a and b, if track a intersects a net pin while b does not, we remove track b. If neither a nor b intersects a net pin, then one track is arbitrarily chosen for removal. Note that the resulting grid

graph is *always* less dense than the original EHG. Figure 10 shows an example of a grid graph (a) before and (b) after sparsification. The pairs of tracks given by coordinates x_1 and x_2 and by y_3 and y_4 are redundant. Since x_1 intersects the source, x_2 is removed. Neither y_3 or y_4 intersect a net pin, so y_4 is randomly removed.

The second sparsification technique severs some tracks span the entire grid graph. For example, in Figure 10(a), the track y_7 is induced by the upper border of the rectangle representing blockage A; thus, a routing path that uses track y_7 results from avoiding blockage A. When the path hits B it can either overlap or circumvent B. If the routing cost according to Equation (2) of circumventing B is less than the cost of overlapping B, then we say the corresponding track is *severable*. The bold part of track y_7 (a) that firsts hits the blockage B can be removed (b). In Figure 10, these techniques reduce the number of grid nodes from (a) 64 to (b) 46.

5.2 Branch and Bound Maze Routing

When expanding the lowest cost node, maze routing cannot distinguish between good and bad global directions. The expansion may proceed in a direction completely opposite the target sub-tree which wastes significant computation time. Branch and bound techniques can prevent some unnecessary expansions.

Recall Steps 3-7 of Figure 8 which iteratively delete and then reconstruct 2-paths. The 2-path $p(u, v)$ removed in Step 5 has $cost(p(u, v))$ which is also an upper bound for the cost of the new 2-path.. Let $upCost$ denote this value. After Step 4 of Figure 9, one can compare $label(v)$ to $upCost$ to determine if node v is worth expanding. If $label(v) > upCost$ then the cost of the path from T_s to v is already higher than the cost of the original 2-path, which makes it wasteful to expand v. Whenever a node $v \in T_t$ is reached, the value for $upCost$ can be replaced by $label(v)$ if this value is less than $upCost$.

The bound can be made even tighter by using a lower bound on the cost of the remaining routing to be done from v to T_t. Let $dist(v, T_t)$ be the Manhattan distance from v to the bounding box of T_t (which can be computed in constant time).[1] Now the test becomes whether $label(v) + dist(v, T_t) > upCost$ holds. If so, node v is not worth further exploration and and Step 7 of Figure 9 is skipped.

Note that this speedup is similar to A^* search [18] in utilizing a estimated cost from wavefront node to target. However, in A^* search this estimated cost is augmented to define node priority in wave expansion instead of serving for cost bounding.

[1] If $\alpha < 0$, then $(1 + \alpha)dist(v, T_t)$ is the lower bound.

6 Experiments

We performed experiments on three designs. Test1 is a small hand crafted test obtained from [20], Test2 is a large macro block, and Test3 is a hierarchical microprocessor design. The comparisons that follow are made between two algorithms, SMT, a Steiner minimal tree algorithm that is used for net analysis within an industrial physical design tool suite, and BBB, our proposed algorithm.

Note that works which perform simultaneous buffer insertion and routing tree construction such as [16] [19] are inappropriate since they do not consider blockage. The work of [22] attacks the right problem space, but it cannot be applied to multi-pin nets. All run times are reported in seconds for an IBM RS6000/595 processor with 512MB of RAM.

6.1 Additional Routing Cost

Our first experiment measures the additional wire length caused by BBB compared to SMT. Since BBB is aware of blockage constraints while SMT is not, BBB should naturally increase total wire length, while decreasing wire length overlapping blockages. Tables 1 and 2 present these results for Test1 and Test2, respectively.

		Avg. wire length			Avg. in-bloc. length		
mode	#net	SMT	BBB	%impr.	SMT	BBB	%impr.
blck	23	21.3	21.7	-1.8%	15.9	5.8	63.5%
bays	30	22.7	22.2	-2.2%	22.2	10.4	52.7%

Table 1: Routing costs of SMT versus BBB for Test1.

For Test1, SMT and BBB were on 23 nets with 7 random blockages inserted and on 30 other nets with 7 random buffer bays inserted. The number of blockages or bays actually used by the BBB grid graph is between 2-7 for all nets. The results were averaged over all nets and are summarized in Table 1. The average wire length increases by only 1.8% for blockages and 2.2% for buffer bays which indicates that BBB is almost as effective as SMT for construction a low wire length Steiner tree. However, the total wire length in blocked regions was reduced by 63.5% for blockages and 52.7% for buffer bays by BBB. The average CPU time to run BBB on a net was less than 0.2 seconds for both blockages and bays.

For Test2, we examined 16 timing critical nets that had differentiating characteristics (number of pins, pin locations, wire length topology, etc.) and ran both SMT and BBB with the 54 blockages that were present in the design. Table 2 presents the results. We observe that BBB results in an average of 2.5% higher than SMT while reducing blocked wire length by 33.3%. The results vary widely for different nets.

net	#pin	$\|B\|$	Wire length			Blocked wire length			CPU
			SMT	BBB	%impr.	SMT	BBB	%impr.	
1	2	26	10.7	12.2	-13.9%	9.3	2.0	78.6%	0.5
2	2	36	9.0	9.0	0.0%	5.2	0.4	92.9%	0.8
3	9	30	14.6	15.1	-3.8%	12.7	4.9	61.4%	1.3
4	9	31	14.6	15.2	-4.6%	12.8	7.1	44.4%	1.3
5	9	47	18.4	18.7	-1.7%	18.2	14.0	23.2%	2.2
6	11	47	17.1	17.6	-2.8%	17.1	2.6	84.9%	2.7
7	17	53	24.1	24.1	-0.1%	22.4	21.9	2.3%	5.8
8	19	47	19.7	20.7	-5.0%	19.7	16.6	16.0%	5.2
9	19	47	20.2	20.8	-3.2%	20.2	17.7	12.3%	5.6
10	25	47	22.2	22.3	-0.3%	22.0	20.9	4.9%	4.7
11	25	47	22.6	22.7	-0.4%	22.4	21.3	4.9%	4.8
12	25	47	23.6	24.1	-2.1%	23.5	14.6	37.8%	5.9
13	29	33	23.3	23.9	-2.8%	15.7	10.9	30.3%	5.4
14	29	33	24.9	25.1	-0.6%	18.4	14.2	22.6%	4.9
15	29	53	30.5	31.4	-3.0%	23.3	11.2	51.8%	9.8
16	29	53	29.0	30.4	-5.0%	19.9	8.6	56.7%	14.0
Ave	18	42	20.3	20.8	-2.5%	17.7	11.8	33.3%	4.7

Table 2: Routing costs of SMT versus BBB for Test2.

For example, reduction in blocked wire length for age length net7, net10 and net11 are limited because the majority of the pins actually lie within blockage.

6.2 Delay Comparisons with Buffer Insertion

To assess the utility of BBB versus SMT trees, buffer insertion must be performed after routing. The next set of experiments were performed on a net by net basis with the following methodology:

1. Compute the SMT tree for the net.
2. Compute the delays to each sink, then compute the slack to the most critical sink based on the required arrival times supplied by the static timing analyzer.
3. Run BBB re-routing.
4. Perform buffer insertion. The tool we employ here has been used in the design of hundreds of ASICs and several microprocessors.
5. Re-compute the slack to the most critical sink. Let $\Delta slack$ denote the difference between this slack and the slack computed in Step 2.

Skipping Step 3 of this methodology yields buffer insertion with the SMT algorithm while including Step 3 yields results for the BBB algorithm.

Average results for Test1 are presented in Table 3 with $\Delta slack$ values presented in picoseconds. Observe that BBB utilizes more buffers than SMT (2.9 versus 2.2 for blockage and 2.3 versus 1.9 for bays) since BBB offers more potential locations for buffers. BBB trees also reduced slack by an additional 337 (768) ps over SMT trees for blockage (bay) mode.

		SMT + BI (Ave.)		BBB + BI (Ave.)		
mode	#net	$\Delta slack$	#buf	$\Delta slack$	#buf	CPU
blockage	23	2064	2.2	2401	2.9	4.0
bays	30	2494	1.9	3262	2.3	4.3

Table 3: Experimental results on average slack improvements for Test1.

Table 4 presents the same experiments for the 16 nets from Test2. For the net by net comparisons, we first took the SMT solution which yielded the best value for $\Delta slack$, then compared it against the BBB solution with the same number of buffers. Thus, each row in Table 4 uses the same number of inserted buffers. Overall, SMT trees resulted in an average slack improvement of 519.4 ps as compared to 694.6 ps for BBB. The runtimes reported are for the combination of BBB plus the buffer insertion step. By comparing these runtimes to those reported for BBB alone in Table 2, we see that the runtimes of BBB do not dominate the buffer insertion runtimes. Note that since buffer insertion is applied only to timing critical nets or those with noise or slew violations, these runtimes are reasonable for practical applications.

net	#pin	SMT $\Delta slack$	BBB $\Delta slack$	#buf	CPU(s)
1	2	1032	1118	2	1.2
2	2	1034	1036	1	1.2
3	9	109	239	2	2.1
4	9	109	236	2	2.2
5	9	190	452	1	2.9
6	11	7	71	1	4.0
7	17	850	1181	2	7.8
8	19	578	1089	2	7.3
9	19	605	880	2	7.8
10	25	277	299	2	7.6
11	25	295	323	2	7.4
12	25	205	228	2	9.0
13	29	223	308	5	24.4
14	29	371	375	4	25.2
15	29	1049	1605	7	35.3
16	29	1376	1674	5	36.6
Ave.	18	519.4	694.6	2.6	11.4

Table 4: Experimental results on slack improvement for Test2.

6.3 Fixing Slew Problems

Finally, we considered the problem of using buffers to fix slew violations. In high performance design, it is common for each gate to have a requirement for the maximum permissible slew rate on the input signal to the gate. Buffers can be used to fix such problems by repowering a degrading signal and sharpening the slew rate. For Test3, microprocessor designers identified 29 non-critical nets that had slew violations. We attempted to fix these violations using the routes provided by both SMT and BBB in conjunction with buffer insertion. The designers also specified several buffer bays; everywhere else was considered completely blocked.

Algorithm	#net	#fixed	#improved	#failed
SMT	24	7	6	11
BBB	24	17	4	3

Table 5: Slew results for SMT and BBB on Test3.

Of the 29 nets, 5 of them had pins nowhere near the designated buffer bays, so neither SMT nor BBB approach would work. The results for the remaining 24 nets are shown in Table 5. Of these 24 nets, BBB was able to successfully re-route and fix 17 of the nets while SMT was only able to fix 7 nets. Of the 7 nets for which BBB failed, BBB was able to improve the slew (but not quite fix it) for 4 nets, while it did not insert any buffers for 3 of the nets. SMT was unable to insert buffers on 11 nets since they did not intersect buffer bays, but it was able to improve, but not fix, the slew on 6 of the nets. Overall, BBB showed that it is better suited for fixing slew violations than a routing algorithm that ignores blockage.

7 Conclusion

We propose a new Steiner tree routing problem for making nets more amenable to buffer insertion in the presence of blockage constraints. The formulation handles either a buffer blockages or buffer bay floorplanning methodology. Our heuristic iteratively deletes and re-routes sub-paths of an existing Steiner tree and can handle complex blockage maps. Several speedup techniques are incorporated so that the empirical run times are practical, though the theoretical time complexity of is high. Experimental results show that our method achieves the objective of avoiding buffer blockages (or seeking buffer bays) and can provide significant improvements in terms of delay and slew when used in conjunction with an industrial buffer insertion tool.

References

[1] C. J. Alpert, and A. Devgan, "Wire Segmenting for Improved Buffer Insertion," *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 588-593, 1997.

[2] C. J. Alpert, A. Devgan and S. T. Quay, "Buffer Insertion with Accurate Gate and Interconnect Delay Computation," *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 479-484, 1999.

[3] C. C. N. Chu and D. F. Wong, "Closed Form Solution to Simultaneous Buffer Insertion/Sizing and Wire Sizing," *Proceedings of the International Symposium on Physical Design*, pp. 192-197, 1997.

[4] C. C. N. Chu and D. F. Wong, "A New Approach to Simultaneous Buffer Insertion and Wire Sizing," *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 614-621, 1998

[5] J. Cong, "Challenges and Opportunities for Design Innovations in Nanometer Technologies", in *SRC Working Papers*, Dec. 1997.

[6] J. Cong, L. He, C.-K. Koh, and P. H. Madden, "Performance Optimization of VLSI Interconnect Layout," *Integration: the VLSI Journal*, vol. 21, pp. 1-94, 1996.

[7] S. Dhar and M. A. Franklin, "Optimum Buffer Circuits for Driving Long Uniform Lines," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 1, pp. 32-40, 1991.

[8] E. W. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, 1, 269-271, 1959.

[9] M. Hanan, "On Steiner's Problem with Rectilinear Distance," *SIAM Journal on Applied Mathematics*, 14(2), pp. 255-265, 1966.

[10] D. W. Hightower, "A Solution to Line Routing Problems on the Continuous Plane," in *Sixth Design Automation Workshop*, pp. 1-24, 1969

[11] S.-W. Hur, A. Jagannathan and J. Lillis, "Timing Driven Maze Routing," *Proceedings of the International Symposium on Physical Design*, 208-213, 1999.

[12] J. Hu and S. S. Sapatnekar, "Simultaneous Buffer Insertion and Non-Hanan Optimization for VLSI Interconnect under a Higher Order AWE Model," *Proceedings of the International Symposium on Physical Design*, pp. 133-138, 1999.

[13] A. B. Kahng and G. Robins, *On Optimal Interconnections for VLSI*, Kluwer Academic Publishers, Boston, 1995.

[14] J. Lillis, "Timing Optimization for Multi-Source Nets: Characterization and Optimal Repeater Insertion," *Proceedings of the 34th IEEE/ACM Design Automation Conference*, pp. 214-219, 1997.

[15] J. Lillis, C.-K. Cheng and T.-T. Y. Lin, "Optimal Wire Sizing and Buffer Insertion for Low Power and a generalized Delay Model," *IEEE Journal of Solid-State Circuits*, 31(3), pp. 437-447, 1996.

[16] J. Lillis, C.-K. Cheng, and T.-T. Y. Lin, "Simultaneous Routing and Buffer Insertion for High Performance Interconnect," *Great Lakes Symposium on VLSI*, pp. 148-153, 1996.

[17] C. Y. Lee, "An algorithm for Path Connection and its Applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346-365, 1961.

[18] N. J. Nilsson, *Problem-solving Methods in Artificial Intelligence*, New York: McGraw-Hill, 1971.

[19] T. Okamoto and J. Cong, "Buffered Steiner Tree Construction with Wire sizing for Interconnect Layout Optimization," *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 44-49, 1996.

[20] W. Thirtle, *personal communication*, 1997.

[21] L. P. P. P. van Ginneken, "Buffer Placement in Distributed RC-tree Networks for Minimal Elmore Delay," *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 865-868, 1990.

[22] H. Zhou, D. F. Wong, I-M. Liu and A. Aziz, "Simultaneous Routing and Buffer Insertion with Restrictions on Buffer Locations," *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 96-99, 1999.