Finding Obstacle-Avoiding Shortest Paths Using Implicit Connection Graphs

S. Q. Zheng, Joon Shink Lim, and S. Sitharma Iyengar, *Fellow, IEEE*

***Abstract*—We introduce a framework for a class of algorithms solving shortest path related problems, such as the one-to-one shortest path problem, the one-to-many shortest paths problem and the minimum spanning tree problem, in the presence of obstacles. For these algorithms, the search space is restricted to a sparse strong connection graph that is implicitly represented and its searched portion is constructed incrementally on-the-fly during search. The time and space requirements of these algorithms essentially depend on actual search behavior. Therefore, additional techniques or heuristics can be incorporated into search procedure to further improve the performance of the algorithms. These algorithms are suitable for large VLSI design applications with many obstacles.**

I. Introduction

Finding shortest paths in the presence of obstacles is an important problem in robotics, VLSI design, and geographical information systems. In VLSI design, circuit components or previously laid out wires are treated as obstacles. Finding an obstacle-avoiding shortest path between a pair of nodes is a fundamental operation used in many layout algorithms. There are two basic classes of shortest path algorithms: maze-running algorithms and line-search algorithms. Maze-running algorithms can be characterized as target-directed grid propagation. The first such algorithm is Lee's algorithm [12], which is an application of the breadth-first shortest path search algorithm. The major disadvantage of the original Lee's algorithm is that it requires $O(mn)$ memory and running time in the worst case for $m \times n$ grid graphs. In addition, each node requires $O(\log L)$ bits, where L is the length of the shortest path from a source node s to a target node t. It is desirable to reduce the memory requirement for each node. More importantly, the size of searched space must be reduced, since the running time is proportional to this size.

There are a large number of variations of the original Lee's algorithm. Interested readers may refer to [17] for a good survey of these algorithms. Akers [1] modified Lee's algorithm by introducing a coding scheme, which requires two bits per node regardless of the value of L. Using the A^* heuristic search idea proposed in [8], Hadlock obtained a shortest path algorithm, called Minimum Detour (MD) algorithm [7]. The search process of the MD algorithm is controlled by a parameter, detour number d, which is used to indicate a lower bound of the shortest path length. The MD algorithm guarantees finding a shortest path in time less than Lee's algorithm. Soukup [20] proposed a heuristic maze-running algorithm that combines depth-first search and breadth-first search. This algorithm executes the depth-first search from s toward t using a "don't change direction" heuristic until an obstacle is hit or the target node t is reached. When an obstacle is hit, the breadth-first search is used for searching around the obstacle until a grid node that directs toward the target node t is found, and this procedure is repeated until the target node t is reached. The paths found by Soukup's algorithm are not necessarily the shortest ones.

All partial paths generated by maze-running algorithms are represented by unit grid line segments. These algorithms are considered memory and time inefficient. Line-search algorithms have been proposed to achieve improved performance. Since such algorithms search a path by locating a sequence of line segments of variable lengths, they save memory and quickly find a simple-shaped path. Some line-search algorithms do not guarantee finding a shortest path. The firsts of such algorithms are reported in [9] and [15]. The line-search algorithm given in [9] is similar to the one in [15]. The difference is that the algorithm in [9] generates fewer trial lines at every level. Several more recent line-search algorithms (e.g., [5], [16,] [18], [21]) are based on computational geometry techniques. Almost all of these algorithms are based on a graph that is sparser than the original grid and contains a path from s to t. Such a graph is named a connection graph in [13]. Wu *et al.* [21] introduced a rather small connection graph, called track graph. The track graph is not a strong connection graph in the sense that it may not contain a shortest path between a source node s and a target node t. However, their algorithm is able to detect the cases where the shortest paths are not contained by the track graph and handle such cases appropriately to obtain shortest paths. The run time and space of their algorithm are $O((e+k)\log t)$ and $O(e+k)$, respectively, where e is the total number of boundary sides of obstacles, t is the total number of extreme sides of all obstacles, and k is the number of intersections among obstacle tracks, which is bounded by $O(t^2)$. In the worst case, $t = (e)$ and $k = (e^2)$.

In this paper, we propose a new approach to solving the problem of finding rectilinear shortest paths in the presence of rectilinear obstacles using connection graphs. Unlike some existing algorithms (e.g., [5], [18], [21]), the connection graph used in algorithms based on this approach is not explicitly constructed prior to the path search process but generated by interrogating a rather small "database" that characterizes the

Manuscript received August 6, 1993; revised September 22, 1995. This work was supported in part by a US Army grant.

The authors are with the Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803 USA.

Publisher Item Identifier S 0278-0070(96)01344-9.

search space in an on-the-fly fashion during the search. The construction of the connection graph is avoided by the use of binary search trees that represent vertical and horizontal boundaries of obstacles and their extensions. These trees are constructed by using a plane-sweeping technique. These data structures allow the fairly simple determination of whether or not a point is in the connection graph. Only searched portion of the connection graph is represented so that further exploration of the graph is possible and a solution, once found, can be retrieved. The implicit representation and demand-driven elaboration of the connection graph open possibilities for incorporating heuristics into search procedures to further improve the overall algorithm performance. The heuristic used in our algorithm is the A^* heuristic search [8]. Like the MD algorithm, it uses the parameter detour length, which is a concept generalized from the detour number of [7], to control the search process. However, the A^* search of our algorithm is more target directed because of the use of an additional "don't change direction" heuristic and the underlying connection graph.

We also show how to use our approach to design efficient algorithms for the problems of finding rectilinear one-to-many shortest paths and rectilinear minimum spanning trees (MST's) in the presence of rectilinear obstacles. These two problems have important applications in VLSI design. Under the linear delay model of wire length minimization, minimum Steiner trees are sought for connecting nets. Since the minimum Steiner tree problem is NP-hard, a common method is to find an MST first and then obtain a near-optimal Steiner tree by modifying the spanning tree. The one-to-many shortest paths problem arises in timing-driven routing. The collection of shortest paths from one point to several other points form a shortest-path tree (SPT), in which paths may have overlapped edges. Such a tree connects a signal net such that the wire lengths for the source to all sinks of the net are minimized. A class of special rectilinear Steiner trees, called the A-trees, were proposed for performance-driven interconnect under the distributed *RC* delay model in [2]. An A-tree is a rectilinear Steiner tree in which the path connecting the source s and any node on the tree is a shortest path. A-trees are a generalization of the rectilinear Steiner arborescences of [19]. It is not known whether or not the minimum Steiner arborescence problem and the minimum A-tree problem are polynomial-time solvable. For the case that no obstacles are present, polynomial-time approximation and heuristic algorithms have been developed for these two problems [2], [19]. When obstacles are present, the problem becomes more difficult. Near optimal A-trees can be obtained by modifying SPT's.

The fact that our algorithms do not explicitly construct connection graphs may have a significant impact. In a large VLSI design with many obstacles, the construction of the entire connection graph could be too costly. The algorithms using our approach have time and space complexities similar to those of existing algorithms in the worst case. In most cases, however, our algorithms should prove significant speed and space improvement. For example, for our one-to-one shortest path algorithm, the time required for obtaining a shortest path is related more to the length of the path (which may be relatively small) than to the size of the entire connection graph (which may be very large).

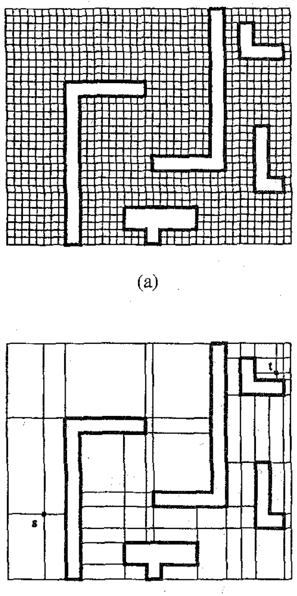


Fig. 1. Grid G and its corresponding G_C for a given pair of s and t.

II. A One-to-One Shortest Path Algorithm

Let R be an mn uniform grid graph that consists of a set of grid nodes $\{(x,y)|x$ and y are integers such that $1 \leq x \leq n, 1 \leq y \leq m\}$ and grid edges connecting grid nodes. The length of grid edges connecting adjacent nodes in R is assumed to be 1. Let $B = \{B_1, B_2, \cdots, B_p\}$ be a set of mutually disjoint rectilinear polygons with boundaries on R. Each polygon in B is an obstacle. Let G denote a partial grid of R that consists of grid nodes that are not contained in the interior of any obstacle in B, and grid edges that are not incident to interior grid nodes of any obstacle in B (see Fig. 1(a)).

Maze-running algorithms use grid G to find a path between a pair of nodes. The main operation of maze-running algorithms is node labeling. Starting from the source node s, nodes of G are labeled in such a way that node v is labeled after at least one of its neighboring nodes has been labeled, until the target node t is labeled. Then, a path from s to t is generated from the node labels. Because of this feature, maze-running algorithms are usually called grid expansion algorithms (they are also called "wave propagation" algorithms). Lee's algorithm [12] is a customized Dijkstra's breath-first search algorithm for uniform grid graphs; Dijkstra's algorithm can be applied to arbitrary graphs. The Minimum Detour (MD) algorithm of [7] is a combination of breath-first search and the A^* search proposed in [8]. Given a source node s and a target node t, we denote the Manhattan distance between s and t by $M(s,t)$. Let P be any obstacle-avoiding rectilinear path from s to t, the detour number $d(P)$ is defined as the total number of nodes on P that are directed away from t. The length of P is $M(s,t) + 2d(P)$. Then, P is a shortest path if and only if $d(P)$ is minimized among all paths connecting s and t. During the grid expansion process of the MD algorithm, detour numbers with respect to t, rather than the distances from the s, are used to label the searched nodes, and those nodes with smaller detour numbers are expanded with higher priority. Since $M(s,t)$ is fixed for a given pair of s and t, the breath-first search in the increasing order of detour numbers ensures that the detour number of t can be correctly computed. After the node labeling process of the MD algorithm, a shortest path from s to t can be easily obtained by backtracking the expanded grid nodes of G in decreasing order of detour number labels of the expanded nodes. Fig. 2 shows how the same example of [20] is solved using these two maze-running algorithms. The grid is in its offset form, i.e., each face defined by four unit segments corresponds to a grid node in the original grid G. The faces labeled by a solid triangle and a white triangle represent the source node s and the target node t, respectively. The faces labeled by solid circles represent a shortest path, and the faces labeled by white circles represent the remaining expanded nodes.

It is a simple fact that to find a path from s to t, we only need to consider a subset of nodes in G. Let V' be a subset of grid nodes of G, and H be a graph such that there exists a subgraph G' of G that spans V', and G' is homeomorphic to H. According to [13], H is called a connection graph for V' in G if all pairs of nodes in V' are connected in $H; H$ is called a strong connection graph for V' in G, if H is a connection graph for V' in G and the lengths of all shortest paths between pairs of nodes in V' are the same in G and H. In what follows, we introduce a strong connection graph G_C for a given pair of nodes s and t in G.

We say that two line segments overlap if they share more than one point. Define a maximal horizontal (respectively, vertical) line segment $l = (u,v)$ in R as a horizontal (respectively, vertical) line segment such that l does not cross any B_j in B, l does not overlap with any boundary of R, and obstacles in B and u and v are the only two points in l that are on the boundaries of R or obstacles in B. Let $L_H^m = \{l | l = (u,v)$ is a maximal horizontal line segment in R such that at least one of its endpoints u and v is a corner of some B_i in $B\}$, and $L_V^m = \{l | l = (u,v)$ is a maximal vertical line segment in R such that at least one of its endpoints u and v is a corner of some B_i in $B\}$. Let $L(R,B)$ be the set of line segments that form the boundaries of R and obstacles in B. Let L_s be the set of all maximal line segments that include s and L_t be the set of all maximal line segments that include t. The nodes of G_C are the intersection points of the line segments in set $L = L(R,B) \cup L_H^m \cup L_V^m \cup L_s \cup L_t$, and the edges of G_C are the subsegments of the segments of L generated by the intersections. The grid graph G_C corresponding to the grid graph G of Fig. 1(a) is shown in Fig. 1(b). Let $e = |L(R,B)|$. Clearly, each of L_H^m and L_V^m contains $O(e)$ line segments. Therefore, $|L| = O(e)$ and the numbers of nodes and edges in G_C are at most $O(e^2)$. Consider any rectilinear obstacle-avoiding path P from s to t. Note that here we are not insisting that P must be on G_C. It is easy to verify that, starting from s, one can "bend" P to obtain a modified path P' in G_C such that the length of P' is no larger than the length of P. This transformation implies the following fact.

Lemma 1: G_C is a strong connection graph for s and t in G.

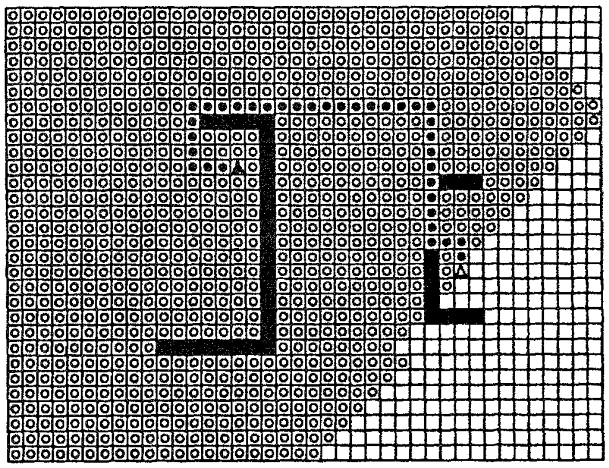
It is important to note that for any problem instance in which the coordinates of the corner points of R and obstacles in B, and a given pair of source and target points (s and t) are not necessarily integers, G_C (and G'_C defined in Section IV) can be used to find optimal solutions. In fact, the algorithms proposed in this paper work for such problem instances.

Based on strong connection graph G_C, we propose a line-search version of the MD algorithm. We first generalize the concept of detour number. Consider a direction assigned to an edge (u,v) of G_C, say, the direction is from u to v. With this direction assignment, we have a directed edge $u \rightarrow v$. We define the detour length of $u \rightarrow v$ with respect to a target node t, denoted by $dl(u \rightarrow v)$, as follows. Let l be the line passing through t and perpendicular to $u \rightarrow v$

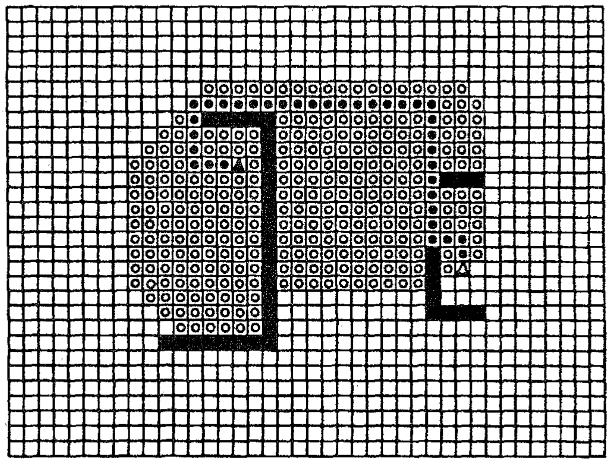
$$dl(u \rightarrow v) = \begin{cases} 0, & \text{if } u \text{ and } v \text{ are on the same side of } l \\ & \quad \text{and} u \text{ is further from } l \text{ than } v; \\ \text{the length of } u \rightarrow v, & \\ & \text{if } u \text{ and } v \text{ are on the same side of} \\ & \quad l \text{ and } u \text{ is closer to } l \text{ than } v; \\ \text{the length of } w \rightarrow v, & \\ & \text{if } l \text{ intersects } u \rightarrow v \text{ at } w. \end{cases}$$

The detour length of a node u with respect to a source node s and a target node t, denoted by $\delta(u)$, is the sum of the detour lengths of all directed edges in any directed shortest path from s to u in G_C. Let P^* be a shortest path from s to t in G_C. Clearly, the length of P^* is equal to $M(s,t) + 2\delta(t)$.

Starting from the source node s, our algorithm explores G_C node by node. A global detour length d, which initially has value 0, is used to control the search process. Each node u is associated with a field $DL[u]$, which contains an upper bound of the detour length $\delta(u)$ of u computed during the execution of the algorithm. Two subsets of nodes of G_C, *VISITED* and *CANDIDATES,* are maintained. Initially, *VISITED* $= \emptyset$. The search starts with *CANDIDATES* containing all neighboring node of s in G_C. The search proceeds as follows: a node u in *CANDIDATES* with the smallest DL value is selected for grid expansion. For all neighboring nodes of u that are not currently in *VISITED,* compute their new DL values and perform *CANDIDATES* update operations using procedure *update* to ensure that they are in *CANDIDATES* with their current smallest DL values. When a node u is detected that $DL[u] = \delta(u)$, it is inserted into *VISITED*, and this equality remains unchanged thereafter. Each node u has another field $PRED[u]$, which links node u to its predecessor in a path from s to u. When the algorithm terminates, the chain of predecessors originating at node t runs backward along a shortest path from s to t. The node set *CANDIDATES* is implemented as a priority queue. The nodes in *CANDIDATES* are ordered in nonincreasing order of their DL values. In case that a tie occurs (i.e., two or more nodes with the same DL value), the nodes with the same DL value are ordered in First-In-Last-Out (FILO) order. This implementation of *CANDIDATES* enforces the invariant that nodes with smaller detour lengths are inserted into *VISITED* before nodes with larger detour



(a)



(b)

Fig. 2. Expanded nodes of Lee's algorithm and the MD algorithm.

lengths. To increase the chance of reaching the target node quickly, a guided depth-first search feature is incorporated into the search process. A procedure *forward* and the ordering maintenance method of *CANDIDATES* effect "don't change direction" search whenever possible. Our algorithm is given below.

```
Algorithm PATHFINDER
begin
   CANDIDATES := $\emptyset$;
   $DL[s] := 0$;
   insert(VISITED, s);
   for each neighbor $u$ of $s$ in $G_C$ do
      $DL[u] := dl(s \to u)$;
      $PRED[u] := s$;
      insert(CANDIDATES, $u$)
   endfor
   repeat
      $u :=$ deletemin(CANDIDATES);
      insert(VISITED, $u$);
      $d := DL[u]$;
      if $u = t$ then stop;
      if $u$ has a neighbor $v$ in $G_C$ such that
            $dl(u \to v) = 0$ and $v \notin$ VISITED then
         begin
            update(CANDIDATES, $u, v, d$);
            for each such neighbor $v$ of $u$ do
               dir := direction of $u \to v$;
               forward($u$, dir, $d$)
            endfor
         end
   else
      for each neighbor $v$ of $u$ in $G_C$ such that $v \notin$
            VISITED do
         update(CANDIDATES, $u, v, DL[u] + dl(u \to v)$)
      endfor
   endrepeat
end
```

The two procedures *forward* and *update* are as follows:

```
procedure forward ($u$, dir, $d$)
begin
   newdl := $d$;
   while newdl $= d$ and $u$ has a neighboring node $v$ in
         $G_C$ such that the direction of $u \to v =$ dir
         and $v \notin$ VISITED do
      newdl := $DL[u] + dl(u \to v)$;
      update(CANDIDATES, $u, v$, newdl)
      if newdl $= d$ then
         begin
            $DL[v] := d$;
            $PRED[v] := u$;
            insert(VISITED, $v$);
            if $v = t$ then stop
         end
      $u := v$
   endwhile
end

procedure update (CANDIDATES, $u, v, dl$)
begin
   if $v$ CANDIDATES and $dl < DL[v]$ then
         delete(CANDIDATES, $v$);
   $DL[v] := dl$;
   $PRED[v] := u$;
   insert(CANDIDATES, $v$)
end
```

Theorem 1: Algorithm *PATHFINDER* finds an obstacle-avoiding rectilinear shortest path from s to t in the presence of rectilinear obstacles.

Proof: By Lemma 1, G_C contains a shortest path from s to t. There are three statements that insert nodes of G_C into *VISITED*. The first one is at the beginning of the algorithm, which inserts only one node, s, into *VISITED*. The second one appears after the *deletemin* operation in the **repeat** loop. The third such statement is in procedure *forward.* Since all the node inserted into *VISITED* by this statement are also inserted into the priority queue *CANDIDATES,* the effect of this statement can be ignored, as far as the correctness of the algorithm is concerned. The reason to include this statement in *forward* is to improve the performance of the algorithm, since the **while** loop implements the "don't change direction" heuristic. In fact, the removal of the **if** statement in *forward* will not affect the correctness of the algorithm, and we may assume that this **if** statement is not present. Also, it is easy to see that once a node is inserted into set *VISITED,* the values of its DL and $PRED$ fields are not changed thereafter. Therefore, we focus our attention on the *insert* operation that follows the *deletemin* operation in the **repeat** loop. Consider the following algorithm:

```
algorithm FMD
begin
  for each node $u$ of $G_C$ do
      $DL[u] := +\infty$;
      $PRED[u] := nil$;
  endfor
  $DL[s] := 0$;
  VISITED := $\emptyset$;
  insert(CANDIDATES, $s$);
  repeat
      $u :=$ deletemin(CANDIDATES);
      insert(VISITED,$u$);
      if $u = t$ then stop;
      for each neighbor $v$ of $u$ in $G_C$ do
        if $DL[v] >, DL[u] + dl(u \rightarrow v)$ then
          begin
            $DL[v] := DL[u] + dl(u \rightarrow v)$;
            $PRED[v] := u$
          end
      endfor
  endrepeat
end
```

It is not difficult to see that algorithm *PATHFINDER* finds a shortest path if and only if FMD algorithm finds a shortest path. Let *length*(u, v) denote the length of the edge connecting two adjacent nodes u and v in G_C. If we replace $dl(u \rightarrow v)$ with *length*(u, v), then FMD algorithm becomes Dijkstra's shortest path algorithm. Since the only difference between FMD and Dijkstra's algorithm is in the metrics used, and both detour length and edge length are nonnegative, by the correctness of Dijkstra's algorithm, we conclude that algorithm *PATHFINDER* correctly computes a shortest path from s to t on G_C. This completes the proof of the theorem. □

III. Algorithm Implementation and Performance

We want to represent G_C implicitly. A basic operation of *PATHFINDER* is for a node u in G_C, find all its neighbors in G_C. We name this operation as neighbor finding in the connection graph. Suppose that we have all the line segments in L available. Partition L into two subsets L_V and L_H, which contain vertical and horizontal segments of L, respectively. The line segments of L can be used to determine the degree of u in G_C. This can be done by the following operation: Find all the line segments in L_V (respectively, L_H) that include u. We can represent L_V by a balanced two-level binary search tree T_V^1 in which each node corresponds to a unique x-coordinate of line segments in L_V. Each node of T_V^1 has a pointer to a balanced binary search tree (secondary structure) for the y-coordinates of lower endpoints of the segments in L_V that have the same x-coordinate. T_V^1 can be easily constructed in $O(|L_V| \log |L_V|) = O(e \log e)$ time and $O(|L_V|) = O(e)$ space. Using T_V^1, all (at most two) vertical line segments in L that include u can be found in $O(\log |L_V|)$ time. Similarly, we can construct a binary tree T_H^1 for horizontal segments in L. Therefore, the operation of finding all the line segments in L that include u can be carried out in time $O(\log |L|)$, which is $O(\log e)$ since $|L| = O(|L(R, B)|) = O(e)$. Then, the problem of finding neighbors of a given node u in G_C can be reduced to the following operation: given a grid node u of G_C and a direction a, find the first line segment in L encountered by a line emanating from u in direction a. For this operation, we can represent L_V (respectively, L_H) by a special balanced binary search tree T_V^2 (respectively, T_H^2) of the structure described in [6] or [14]. The construction of T_V^2 (respectively, T_H^2) requires two steps. The first step normalizes the coordinates of end points of segments in L_V (respectively, L_H) to their ranks, and the second step builds T_V^2 (respectively, T_H^2) using the normalized integer coordinates. Both steps take $O(e \log e)$ time and $O(e)$ space. Using T_V^2 and T_H^2, the above mentioned operation can be carried out in $O(\log e)$ time. Note that the data structure T_V^1, T_V^2, T_H^1, and T_H^2 are static data structures, i.e., once they are constructed, their structures are not changed during subsequent search process.

Based on above discussions, we can use sets L_V and L_H as a database to assist the search process. We can compute L_V and L_H using a straightforward version of the powerful plane-sweep technique developed in computational geometry in $O(e \log e)$ time, using $O(e)$ space. This preprocessing algorithm is similar to the one described in [13, pp. 407].

The operations related to set *VISITED* are the insertion and the operation of testing whether or not a given node of G_C is in *VISITED.* We can represent *VISITED* by a dynamically balanced binary search tree $T_{VISITED}$ using lexicographical order of node coordinates. Each insertion and membership testing operation can be carried out in $O(\log N)$ time, where N is the number of nodes in *VISITED* when algorithm *PATHFINDER* terminates. Since $N \leq O(e^2), O(\log N) = O(\log e)$. Similarly, the set *CANDIDATES* can be implemented using two dynamically balanced binary search trees, one using the DL values as keys (for deletemin), and the other using the node coordinates as keys (for membership testing). An insertion (respectively, deletion) operation on *CANDIDATES* effects two insertion (respectively, deletion) operations, one on each of these two trees. Since every node in *CANDIDATES* has at least one neighbor in G_C that is in *VISITED,* we know that $|$*CANDIDATES* $| \leq 4|$*VISITED*$| = O(N)$. Any of insertion, deletion, deletemin, and membership testing operations on *CANDIDATES* can be done in $O(\log N)$ time, which is no greater than $O(\log e)$. We summarize above discussions by the following theorem.

Theorem 2: Algorithm *PATHFINDER* can be implemented in $O((e+N)\log e)$ time and $O(e+N)$ space, where e is the number of boundary sides of obstacles in B and N is the total number of searched grid nodes of G_C when the algorithm terminates.

The FMD algorithm introduced in the proof of Theorem 1 not only simplifies the proof but also can be used to verify the effectiveness of algorithm *PATHFINDER.* We claim that, in terms of searched space, *PATHFINDER* is more efficient than FMD, and FMD is better than Dijkstra's algorithm.

Let us compare the *PATHFINDER* with FMD. The FMD algorithm is a generalization of the MD algorithm that works on G_C instead of G (and for this reason, the FMD algorithm can be interpreted to stand for Fast Minimum Detour algorithm). To our knowledge, the detour length concept and the FMD algorithm are new, even though they appear to be simple generalizations of detour number and the MD algorithm of [7]. Since *PATHFINDER* uses an additional "don't change direction" heuristic, its searched space is less than that of FMD.

Traditionally, Dijkstra's algorithm is used for the shortest path problem on graphs with edges of variable lengths. For comparisons, Dijkstra's algorithm and the FMD algorithm can be considered as a "coarse-grain" Lee's algorithm and the MD algorithm, respectively. As evidenced by the performances of the fine-grain versions of FMD algorithm and Dijkstra's algorithm shown in Fig. 2, the performance of FMD is always better than Dijkstra's algorithm on G_C due to the difference between metrics used. In terms of the size of searched space, the performance of FMD is better than Dijkstra's algorithm. This is because that the length of a shortest path from s to a node u is always larger than the detour number of u. An extreme case is that the length of a shortest path from s to t is large, but the detour length of t is very small. In such a case, the searched portion of G_C by FMD algorithm can be significantly smaller than that by Dijkstra's algorithm. Therefore, comparing the FMD algorithm with Dijkstra's algorithm (replacing $dl(u \rightarrow v)$ with length (u, v) in FMD) further verifies the advantage of the *PATHFINDER* algorithm. For the same example in Fig. 2, the operations of the *PATHFINDER* algorithm are shown in Fig. 3. The nodes are explored in the direction of edges, and in increasing order of sequence numbers. In Table I, the detour lengths of these nodes are listed with their sequence numbers.

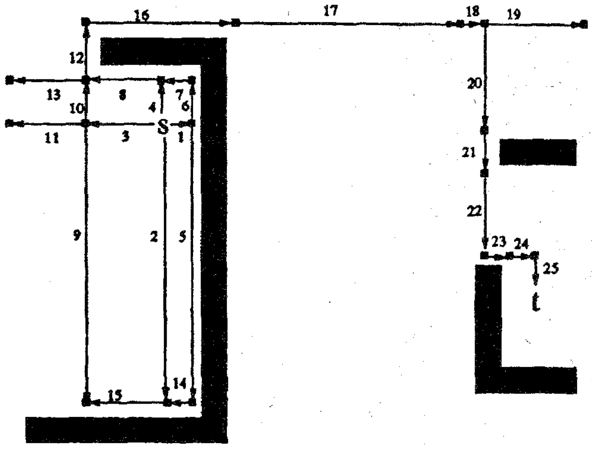


Fig. 3. Extended line segments by the *PATHFINDER* algorithm.

TABLE I
DETOUR LENGTHS OF EXPLORED EDGES OF EXAMPLE SHOWN IN FIG. 3

sq #	dl	sq #	dl	sq #	dl	sq #	dl	sq #	dl
1	0	6	2	11	7	16	7	21	7
2	4	7	3	12	7	17	7	22	7
3	3	8	5	13	9	18	7	23	7
4	2	9	7	14	5	19	9	24	7
5	4	10	5	15	7	20	7	25	7

IV. GENERALIZATIONS

Wu *et al.* [21] considered the problem of finding rectilinear shortest paths from one point in a given point set S to all other points in S (the one-to-many SP's problem) and the problem of finding a rectilinear minimum spanning tree of a set S of points (the MST problem) in the presence of rectilinear obstacles. These two problems can be stated as follows. Let boundary R, a set $B = \{B_1, B_2, \cdots, B_p\}$ of obstacles and grid G be defined as in Section II, and let S be a set of n nodes of G. The one-to-many SP's problem is to find obstacle-avoiding rectilinear shortest paths from a point s in S to all other points in S. The minimum spanning tree considered is defined by treating points in S as nodes and rectilinear shortest paths among them as edges of an implicitly given complete graph. The MST problem is to find a spanning tree of S in this graph that has minimum total edge length. Their approach consists of two phases. In the first phase, a grid-like connection graph, called track graph G_T, is completely constructed. In the second phase, an optimal solution is computed using G_T. For some problem instances, the track graphs are not strong connection graphs for S in G. In such a situation, an obstacle-avoiding optimal solution may not be found in G_T. The second phase is able to detect cases where the optimality can be violated, and handle them appropriately. The construction of G_T takes $O(n\log n + e\log t + k)$ time, and the space required for storing G_T is $O(n+e+k)$, where n is the number of points in S, e is the total number of boundary sides of obstacles, k is the number of nodes in G_T, and t is the total number of extreme sides in the obstacles (for the definition of extreme sides, refer to [21]). The second phase, for either the one-to-many shortest paths problem or the MST problem, takes $O(n\log n + Nlogt)$ time, where N is the total number of nodes of G_T searched when the algorithm terminates. The total time and space complexities of these two algorithms are $O(n\log n + (e+N)\log t + k)$ and $O(e+n+k)$, respectively. For some problem instances, $t = O(e)$ and $k = O(e^2)$, the performance of these algorithms is dominated by the term of $O(k)$. Clearly, the preprocessing time is the bottle-neck of their algorithms, since $N < k$ for most cases. In a large VLSI design with many obstacles, the space requirement for G_T is too costly.

We generalize our connection graph G_C to obtain connection graph G'_C, and show that G'_C can be implicitly represented to solve the one-to-many SP's problem and the MST problem. Let L_H^m and L_V^m be as defined in Section 2. Let L_S be the set of all maximal line segments that include points in S. The connection graph, G'_C, is defined as follows. The nodes of G'_C are the intersection points of the line segments in the set $L = L(R, B) \cup L_H^m \cup L_V^m \cup L_S$, and the edges of G'_C are the subsegments of segments in L generated by their intersections. By Lemma 1, we have the following fact.

Lemma 2: G'_C is a strong connection graph for S in G.

Let L_{SV} and L_{SH} be the subsets of vertical segments and horizontal segments of L_S, respectively. Using the plane-sweeping technique, segment sets $L_V^m \cup L_{SV}$ and $L_V^m \cup L_{SV}$ can be computed in $O((e+n)\log(e+n))$ time, where e is the number of boundary sides of obstacles in B and n is the number of points in S. This can be done by treating each point in S as a degenerated obstacle. We would like to point out G'_C is a superset of the track graph G_T of [21].

Our algorithm for the one-to-many SP's problem is obtained from algorithm FMD given in the proof of Theorem 1 by following modifications. We replace the underlying connection graph G_C with G'_C and replace $dl(u \to v)$ with $\mathit{length}(u, v)$. The termination condition of the new algorithm is that all points of S are included into *VISITED*. Initially, *CANDIDATES* contains one point, s. The step for initializing all nodes in G'_C is not needed because a node is assigned a length value only when it is included into *CANDIDATES*. The data structures used are the same as the ones for *PATHFINDER*. These data structures take $O(e+n)$ space. They support $O(\log(e+n))$-time search and update operations on *VISITED* and *CANDIDATES*, and the neighbor finding operation on implicit G'_C. Therefore, using implicit G'_C, the shortest paths from a point in S to all other points in S in the presence of rectilinear obstacles can be found in $O((e+n+N)\log(e+n))$ time and $O(e+n+N)$ space using implicit G'_C. We summarize the performance of this algorithm in the following theorem.

Theorem 3: Obstacle-avoiding rectilinear shortest paths from a point in S to all other points in S in the presence of rectilinear obstacles can be found by performing Dijkstra's search on implicit G'_C in $O((e+n+N)\log(e+n))$ time and $O(e+n+N)$ space, where e is the number of boundary sides of obstacles, n is the number of points in S, and N is the total number of visited grid nodes of G'_C when the algorithm terminates.

Wu *et al.* also used the track graph G_T to solve the MST problem. Since G_T is not a strong connection graph, their algorithm transforms G_T into a new graph by adding some line segments. The resulting graph, which we denote as G'_T, is a strong connection graph for S. The graph G'_T is explicitly represented, and searched in the second phase. Their search procedure is a modified Kruskal's algorithm [11]. The basic data structures used include a set *VISITED* and a priority queue *CANDIDATES* similar to our algorithm for the one-to-one SP problem. The primitive operations are insertion and membership testing on *VISITED*, insertion and deletemin on *CANDIDATES*, and neighbor finding in G'_T. Therefore, we can use all our data structures for *VISITED*, *CANDIDATES* and implicit representation of G'_C to implement the modified Kruskal's algorithm given in [21]. The following claim directly follows from the analysis of the modified Kruskal' algorithm given in [21].

Theorem 4: Using implicit strong connection graph G'_T, the algorithm of [21] for finding an obstacle-avoiding rectilinear minimum spanning tree of a set S of n points in the presence of rectilinear obstacles can be implemented in $O((e+n+N)\log(e+n))$ time and $O(e+n+N)$ space, where e is the number of boundary sides of obstacles, n is the number of points in S, and N is the total number of visited grid nodes of G'_C when the algorithm terminates.

If $e \geq n$, then the time and space required by our one-to-many SP's algorithm and MST algorithm are $O((e+N)\log e)$ and $O(e+N)$, respectively. If $n \geq e$, then the time and space required by our algorithms are $O((n+N)\log n)$ and $O(n+N)$, respectively. Since the input size is $O(e+n)$, our algorithms can be expected more time and space efficient than the ones given in [21] in most cases.

V. Conclusion

In this paper, we introduced a framework for a class of algorithms for shortest path related problems in the presence of obstacles. There are two major features of such an algorithm.

1) The search space is restricted to a sparse strong connection graph. The connection graph is implicitly represented and its searched portion is constructed incrementally on-the-fly during the actual search process.
2) As a result of (1), the time and space requirements of the algorithm essentially depend on its search behavior. Additional techniques or heuristics can be incorporated into the search procedure of the algorithm to achieve better performance.

The effectiveness of our approach was demonstrated by our algorithm for the one-to-one SP problem. This algorithm combines an implicit strong connection graph G_C with the A^* search method. Since the detour length as a lower bound in our algorithms can be substituted by the number of bends in the rectilinear link metric [3], [10], [22] or the channel wiring density [4], our algorithm can be extended to solve these problems.

To verify the generality of our approach, we also presented algorithms for the one-to-many SP's problem and the MST problem. Our algorithms are based on a strong connection graph G'_C, and they require $O((e+n+N)\log(e+n))$ time and $O(e+n+N)$ space. The dominating term in these complexities is $O(N)$. It can be reduced by replacing G'_C with a sparser strong connection graph that can be represented in an implicit form from which the actual graph is constructable dynamically and efficiently, and/or incorporating effective heuristics into the search process. We would like to point out that the track graph G_T of [21] can be represented implicitly in the same way as our connection graphs G_C and G'_C. By adding procedures for handling special cases, the algorithms of [21] for the one-to-many SP's problem and the MST problem can be transformed into new algorithms that do not require the construction of entire G_T. We omit the descriptions of these transformations. The time and space complexities of these new algorithms are $O(n \log n + (e + N) \log t)$ and $O(n + N)$, respectively, where n is the number of points in S, e is the total number of boundary sides of obstacles, t is the total number of extreme sides of obstacles, and N is the total number of searched nodes of G_T when these algorithms terminate. Since G_T is sparser than G'_C, these new versions of the algorithms given in [21] can be more efficient than the ones presented in Section 4.

Acknowledgment

The authors wish to thank the anonymous referees for their valuable comments and suggestions which made the significant improvement of this paper over the original manuscript possible.

References

[1] S. B. Akers, "A modification of Lee's path connection algorithm," *IEEE Trans. Electron. Comput.*, vol. EC-16, pp. 97–98, 1967.

[2] J. Cong, K.-S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed *RC* delay model," in *Proc. 30th Design Automation Conf.*, 1993, pp. 606–611.

[3] M. T. De Berg, M. J. Van Kreveld, B. J. Nilsson, and M. H. Overmars, "Finding shortest paths in the presence of orthogonal obstacles using a combined L_1 and link metric," in *Proc. Second Scand. Wkshp. Algorithm Theory*, 1990, pp. 213–24.

[4] A. D. Brown and M. Zwolinski, "Lee router modified for global routing," *Computer-Aided Design*, vol. 22, no. 5, pp. 296–300, June 1990.

[5] K. L. Clarkson, S. Kapoor, and P. M. Vaidya, "Rectilinear shortest paths through polygonal obstacles in $O(n(\log n)^2)$ time," in *Proc. Third Annual Conf. Computational Geometry*, 1987, pp. 251–57.

[6] H. Edelsbrunner and M. H. Overmars, "Some methods of computational geometry applied to computer graphics," *Computer Vision, Graphics, and Image Processing*, vol. 28, pp. 92–108, 1984. Circuit Boards," *IEEE Trans. Circuit Theory*, vol. CT-18, pp. 95–100, 1971.

[7] F. O. Hadlock, "The shortest: Path algorithm for grid graphs," *Networks*, vol. 7, pp. 323–34, 1977.

[8] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst., Sci., Cybern.*, vol. SCC-4, pp. 100–107, 1968.

[9] D. W. Hightower, "A solution to line routing problems on the continuous plane," in *Proc. IEEE Sixth Design Automation Wkshp.*, 1969, pp. 1–24.

[10] Y. Ke, "An efficient algorithm for link-distance problems," in *Proc. 5th ACM Symp., Lect. Notes in Computer Sci., 447.* New York: Springer-Verlag, 1990, pp. 213–224.

[11] J.B. Kruskal, "On the Shortest spanning subtree of a graph and the traveling salesman problem," in *Proc. Amer. Mathemat. Soc.*, vol. 7, pp. 48–50, 1956.

[12] C. Y. Lee, "An Algorithm for Path Connections and Its Applications," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 346–65, 1961.

[13] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout.* Reading, England: Wiley, 1990.

[14] E. M. McCreight, "Priority search trees," Tech. Rep. Xerox PARC CSL-81-5, 1981.

[15] K. Mikami and K. Tabuchi, "A computer program for optimal routing of printed circuit connectors," *IFIPS Proc.*, vol. H-47, pp. 1475–78, 1968.

[16] J. S. B. Mitchell, "An optimal algorithm for shortest rectilinear paths among obstacles in the plane," in *Abstracts First Canadian Conf. Computational Geometry*, 1989, p. 22.

[17] T. Ohtsuki, "Maze-running and Line-search Algorithms," in T. Ohtsuki, Ed., *Advances in CAD for VLSI*, vol. 4: Layout Design and Verification. New York: North- Holland, 1986, pp. 99–131.

[18] P. J. Rezend, D. T. Lee, and Y.-F. Wu, "Rectilinear shortest paths with rectangular barriers," in *Proc. Second Annual Conf. Computat. Geom.*, pp. 204–13, ACM, 1985.

[19] S. K. Rao, P. Sadayappan, F.K. Hwang, and P.W. Shor, "The rectilinear steiner arborescence problem," *Algorithmica*, vol. 7, 1992, pp. 277–288.

[20] J. Soukup, "Fast maze router," in *Proc. 15th Design Automation Conf.*, 1978, pp. 100–102.

[21] Y.-F. Wu, P. Widmayer, M. D. F. Schlag, and C. K. Wong, "Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles," *IEEE Trans. Comput.*, vol. C-36, pp. 321–31, 1987.

[22] C. D. Yang, D. T. Lee, and C. K. Wong, "On bends and lengths of rectilinear paths: A graph-theoretic approach," in *Proc. Algorithms and Data Structures, 2nd Wkshp. WADS 191, Lect. Notes in Computer Sci., 519.* New York: Springer-Verlag, 1991, pp. 320–330.

S. Q. Zheng received the M.S. degree in mathematical sciences from the University of Texas at Dallas in 1982, and the Ph.D. degree in computer science from the University of California, Santa Barbara, in 1987.

In 1987 he joined the faculty of the Department of Computer Science, Louisiana State University, Baton Rouge. Currently, he is an associate professor of computer science, and an adjunct associate professor of electrial and computer engineering at LSU. His research interests include VLSI, parallel and distributed computing, computer networks, and computational geometry.

Joon Shink Lim received the B.S. degree from Inha University, Korea, in 1986, the M.S. degree in computer science from University of Alabama at Birmingham in 1989, and the Ph.D. degree in computer science from Louisiana State University, Baton Rouge, in 1994.

His research interests include VLSI, parallel computing and heuristic search methods.

S. Sitharama Iyengar (M'88–SM'89–F'95) received the Ph.D. degree in 1974.

He is the Chairman of the Computer Science Department and a Professor of computer science at Louisiana State University, Baton Rouge. He has directed LSU's Robotics Research Laboratory since its inception in 1986. He has been actively involved with research in high-performance algorithms and data structures since and has directed more than 18 Ph.D. dissertations at LSU. He has served as principle investigator on research projects supported by the Office of Naval Research, the National Aeronautics and Space Administration, the National Science Foundation/LaSer Program, the California Institute of Technology's Jet Propulsion Laboratory, the Department of Navy-NORDA, the Department of Energy (through Oak Ridge National Laboratory, Tennessee), the LEQFS-Board of Regents, and Apple Computers. He has edited a two-volumn tutorial on autonomous Mobile Robots and has edited two other books and more than 150 publications—including 90 archival journal papers in areas of high-performance parallel and distributed algorithms and data structures for image processing and pattern recognition, autonomous navigation, and distributed sensor networks. He was a visiting professor (fellow) at JPL, the Oak Ridge National Laboratory, and the Indian Institute of Science. He is also an Association for Computing Machinery national lecturer, a series editor for Neuro Computing of Complex Systems, and area editor for Journal of Computer Science and Information. He has served as Guest Editor for IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (1988); *Computer Magazine* (1989); IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS; IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING; and *Journal of Computers and Electrical Engineering.*

Dr. Iyengar was awarded the Phi Delta Kappa Research Award of Distinction at LSU in 1989, won the Best Teacher Award in 1978, and received the Williams Evans Fellowship from the University of Otago, New Zealand, in 1991. He was awarded an IEEE Fellowship for his contributions in data structures and algorithms in 1995.