Routing Using Implicit Connection Graphs

S.Q. Zheng, J.S. Lim and S.S. Iyengar
Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803

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 which 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. These algorithms are suitable for large VLSI design applications with many obstacles.

1 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. 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. Almost all of these algorithms are based on a graph that original grid and contains a path from s to t. Such a graph is named a connection graph in [4]. Wu et al. [6] 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. [1, 5, 6]), the connection graph used in an algorithm 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 search space in an on-the-fly fashion during the search. The construction of the complete connection graph is avoided. 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 procedure to further improve the overall algorithm performance. The heuristic used in our algorithm is the A^* heuristic search [3]. Like the minimum detour (MD) algorithm of [2], it uses the parameter detour length, which is a concept generalized from the detour number, 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 [6, 7].

2 A Shortest Path Algorithm

Let R be an $m \times n$ uniform grid graph that consists of a set of grid nodes $\{(x,y)|x$ and y are integers such that $1 \le x \le n, 1 \le y \le m\}$ 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. 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 [4], 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 (resp. vertical) line segment $l = (u, v)$ in R as a horizontal (resp. 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. 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.*

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 $u \to v$. We define the *detour length of $u \to v$ with respect to a target node t*, denoted by $dl(u \to v)$, as follows. Let l be the line passing through t and perpendicular to $u \to v$, and let $|u \to v|$ denote the length of $u \to v$. Define

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

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 *QUEUE*, are maintained. Initially, *VISITED*$=\emptyset$. The search starts with *QUEUE* containing all neighboring node of s in G_C. The search proceeds as follows: a node u in *QUEUE* 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 *QUEUE* update operations using procedure *update* to ensure that they are in *QUEUE* 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. 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* effects "don't change direction" search whenever possible. Our algorithm is given below.

```
procedure PATHFINDER
begin
   $QUEUE := \emptyset$; $DL[s] := 0$; insert(VISITED,s);
   for each neighbor $u$ of $s$ in $G_C$ do
      $DL[u] := dl(s \to u)$; $PRED[u] := s$;
      insert(QUEUE,u)
   endfor
   repeat
      $u := deletemin(QUEUE)$; 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(QUEUE,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(QUEUE,u,v,$DL[u] + dl(u \to v)$)
        endfor
    endrepeat
end
```

```
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(QUEUE,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(QUEUE,u,v,dl)
begin
    if $v \in QUEUE$ and $dl < DL[v]$
    then delete(QUEUE,v);
    DL[v] := dl; PRED[v] := u; insert(QUEUE,v)
end
```

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 (resp. 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 2) 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 (resp. L_H) by a special balanced binary search tree T_V^2 (resp. T_H^2) of the structure described in [6] or [14]. The construction of T_V^2 (resp. T_H^2) requires two steps. The first step normalizes the coordinates of end points of segments in L_V (resp. L_H) to their ranks, and the second step builds T_V^2 (resp. 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 these 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 [4] (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 $N \le O(e^2)$, $O(\log N) = O(\log e)$. Similarly, the set *QUEUE* 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 (resp. deletion) operation on *QUEUE* effects two insertion (resp. deletion) operations, one on each of these two trees. Since every node in *QUEUE* has at least one neighbor in G_C that is in *VISITED*, we know that $|QUEUE| \le 4|VISITED| = O(N)$. Any of insertion, deletion, deletemin and membership testing operations on *QUEUE* 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 1 *Algorithm PATHFINDER finds an obstacle-avoiding rectilinear shortest path from s to t in the presence of rectilinear obstacles 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.*

3 Generalizations

Wu et al. [6] considered the problem of finding rectilinear shortest paths from one point in the 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 2, 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 [6]). The second phase, for either the one-to-many shortest paths problem or the MST problem, takes $O(n \log n + N \log t)$ 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.

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. We have the following fact:

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

Using the same techniques presented in the previous section, an implicit representation of G'_C can be constructed in $O((e + n) \log(e + n))$ time and $O(e + n))$ space, where e is the number of boundary sides of obstacles in B and n is the number of points in S. Using similar data structures that support the operations of the *PATHFINDER* algorithm, we can convert the algorithms of [6] for the one-to-many SP's problem and the MST problem into new algorithms that do not explicitly construct connection graphs. For details of these new algorithms, refer to [7]. We summarize the performances of our algorithms in the following theorem.

Theorem 2 *Using implicit strong connection graph G'_C, the algorithm of [6] for finding obstacle-avoiding rectilinear shortest paths from a point in S to all other points in S, and the algorithm of [6] for finding an obstacle-avoiding rectilinear minimum spanning tree of a set S of points 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 $e >> n$, 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 $n >> e$, 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 [6] in most cases.

References

[1] K. L. Clarkson, S. Kapoor, and P. M. Vaidya, "Rectilinear Shortest Paths through Polygonal Obstacles in $O(n(logn)^2)$ Time", *Proceedings of the Third Annual Conference on Computational Geometry*, pp. 251-57, ACM, 1987.

[2] F. 0. Hadlock, "The Shortest Path Algorithm for Grid Graphs", *Networks*, 7, pp. 323-34, 1977.

[3] P. Hart, N. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems, Science and Cybernetics*, SCC-4(2), pp. 100-107, 1968.

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

[5] P. J. Rezend, D. T. Lee, and Y.-F. Wu, "Rectilinear Shortest Paths with Rectangular Barriers", in *Proceedings of the Second Annual Conference on Computational Geometry*, pp. 204-13, ACM, 1985.

[6] Y.-F. Wu, P. Widmayer, M. D. F. Schlag, C. K. Wong, "Rectilinear Shortest Paths and Minimum Spanning Trees in the Presence of Rectilinear Obstacles", *IEEE Transactions on Computers*, C-36(3), pp. 321-31, 1987.

[7] S.Q. Zheng, J.S. Lim and S.S. Iyengar, "Finding Obstacle-Avoiding Shortest Paths Using Implicit Connection Graphs", to appear in *IEEE Transactions on Computer-Aided Design*.