Robot Path Planning using Intersecting Convex Shapes: Analysis and Simulation

J. SANJIV SINGH AND MEGHANAD D. WAGH, MEMBER, IEEE

Abstract—**An automated path planning algorithm for a mobile robot in a structured environment is presented. An algorithm based on the Quine-McCluskey method of finding prime implicants in a logical expression is used to isolate all the largest rectangular free convex areas in a specified environment. The free convex areas are represented as nodes in a graph, and a graph traversal strategy that dynamically allocates costs to graph paths is used. Complexity of the algorithm and a strategy to trade optimality for smaller computation time are discussed.**

I. INTRODUCTION

THE TASK of moving robots in mapped environments is a two step process: 1) planning paths that are optimal by some criteria and 2) controlling the robot to execute the planned paths gracefully. This paper deals with the former issue by suggesting a new path finding strategy that is computationally efficient and yields near-optimal results.

Earlier research in this area can be summarized by three approaches—Lozano-Perez [1] used a "visibility graph" to set up a configuration space that can be mapped into a graph of vertices (representing corners of obstacles) between which travel is possible in a straight line. Fixed costs are allocated to graph arcs, and the graph is searched for a path between source and destination. A disadvantage of this method is that every time source and destination points are changed, the visibility graph has to be recomputed. Recently it has been shown that the time taken to do this is of $O(n^2)$ [2], [3].

Thorpe [4] imposed a regular grid on the environment, representing grid points not within the boundaries of obstacles as nodes with arcs to the eight neighbor nodes. He used an A^* search to traverse the graph obtained, employing a cost function that kept the path from getting too close to obstacles. Since the graph obtained is very large, heuristics have to be used to find solutions.

Another approach has been to partition free space into convex polygons. This approach capitalizes on the fact that any two points in a convex polygon can be joined with a straight line without leaving the polygon. If convex polygons can be found such that they represent areas free of obstacles, then a robot can travel between two points in that area without colliding into obstacles. Crowley and Chatila suggest breaking up the free area (for traversal) into nonoverlapping convex polygons [5], [6]. Development of the path depends on traversing the connectivity graph that is produced by representing "free" convex polygons as nodes with arcs to nodes representing geometrically adjacent free convex polygons. The problem with a strategy that breaks up space into nonoverlapping areas is that it fails to take full advantage of convexity and consequently misses some straight line paths that may belong to a convex area of which the procedure is not aware. This is a natural consequence of the fact that this method overlooks a considerable number of convex areas in an environment. Further, if paths are not dynamically refitted to be optimal, paths that would be "naturally" straight, turn out to be contrived. This effect is particularly pronounced if there are relatively large free areas with which to contend. However, there is a one-to-one correspondence between the source and destination points in free space and the graph nodes. Thus the method is successful in getting around the high computational expense at the cost of optimality.

Brooks [7] proposed a method that combines the advantages of both the earlier approaches. Instead of determining the corners of objects that are visible, his method isolates free areas in the form of generalized cones. Brook's robot always traverses along the axes of free cones, and generously avoids the obstacles. Optimality is lost, however, because, although the cones overlap, one does not make full use of convexity. Kuan *et al.* [8] further improved Brooks' method by using a mixed representation of free space. Their strategy used cones to represent narrow spaces and nonoverlapping convex polygons for larger free areas. Though their method works well for highly cluttered environments, the drawbacks associated with nonoverlapping areas still remain. It is worth mentioning that none of these methods exploits any benefits from an orderly orientation of the obstacles.

The path planning algorithm discussed in this paper (first presented in [9]) derives benefit from the concept of convexity by identifying *all* the largest rectangular free areas. A graph is created with nodes corresponding to each such convex area. Intersecting convex shapes are represented as adjacent nodes. Path planning is then reduced to finding a route from a source node to a destination node through the graph and choosing the best possible path based on a given cost function.

Several assumptions are made in this paper.

- A circular robot that can turn in place is assumed when optimality is mentioned. This allows one to shrink the robot to a point and to grow the obstacles by the radius of the robot.
- Obstacles are approximated by iso-oriented rectangles in which the edges are parallel to the coordinate axes.

Manuscript received May 30, 1985; revised June 27, 1986. This work was presented in part at the 1986 IEEE International Conference on Robotics and Automation, San Francisco, CA, April 7-10.

J. S. Singh is with the Construction Robotics Laboratory, Porter Hall, Carnegie-Mellon University, Pittsburgh, PA 15213, USA.

M. D. Wagh is with the Department of Computer Science and Electrical Engineering, Lehigh University, Bethlehem, PA 18015, USA.

IEEE Log Number 8714088.

0882-4967/87/0400-0101$01.00

- The cost allocated to a path or a portion of a path is directly proportional to its geometric length.

II. Isolation of Prime Convex Areas

The algorithm presented here tries to isolate the largest "convex" areas that are free of obstacles. In this paper attention is restricted to rectangular convex areas as there are an infinite number of nonrectangular areas. This assumption of iso-oriented rectangles also helps in keeping the computational costs down.

A *convex area* is an area that is free of obstacles and has the property that any two points in that area can be joined by a straight line that lies entirely within that area. A *prime convex area* is an area that is free of obstacles and is not fully incorporated in any other single prime convex area.

Given a map of boundaries and obstacles, an environment is partitioned by the edges of these shapes into a grid of at most $2n + 1 \times 2n + 1$ rectangles where n is the number of such inadmissible areas representing obstacles. Each such rectangle is represented by a pair of binary strings each at most $2n + 1$ bit long. The left substring represents the relative x position and the right the y position. For example, for $n = 2$ as in Fig. 1(a), a partition that is second from the left and third from the top could be represented by the string

$$0\,1\,0\,0\,0 \quad 0\,0\,1\,0\,0.$$

A similar notation can be used for areas made up of several partitions. The string

$$1\,1\,0\,0\,0 \quad 0\,0\,1\,1\,0$$

represents a larger rectangle made up of four partitions:

$$\begin{array}{l} 1\,0\,0\,0\,0 \quad 0\,0\,1\,0\,0 \\ 0\,1\,0\,0\,0 \quad 0\,0\,1\,0\,0 \\ 1\,0\,0\,0\,0 \quad 0\,0\,0\,1\,0 \\ 0\,1\,0\,0\,0 \quad 0\,0\,0\,1\,0. \end{array}$$

The following algorithm may be used to identify *all* (rectangular) prime convex areas by fusing together free rectangular cells from the grid described. This algorithm is similar to the Quine–McCluskey technique [10]–[12] used to identify the prime implicants of a logical expression.

The following six steps describe the method used.

Step 1

Represent *each* horizontal strip by means of a pair of $2n + 1$ length binary strings. The right substring has only one bit set corresponding to the vertical position of the strip. The left substring has those bits set which correspond to the free rectangles in the strip. For example, the fourth strip of Fig. 1(a) has the representation

$$1\,1\,1\,0\,1 \quad 0\,0\,0\,1\,0.$$

Step 2

Find all the contiguous horizontal strips. This is done by breaking up the left substring into contiguous runs of 1's and repeating the right substring in each part. For example, the strip of step 1 can be broken into two contiguous horizontal strips:

$$1\,1\,1\,0\,0 \quad 0\,0\,0\,1\,0$$

and

$$0\,0\,0\,0\,1 \quad 0\,0\,0\,1\,0.$$

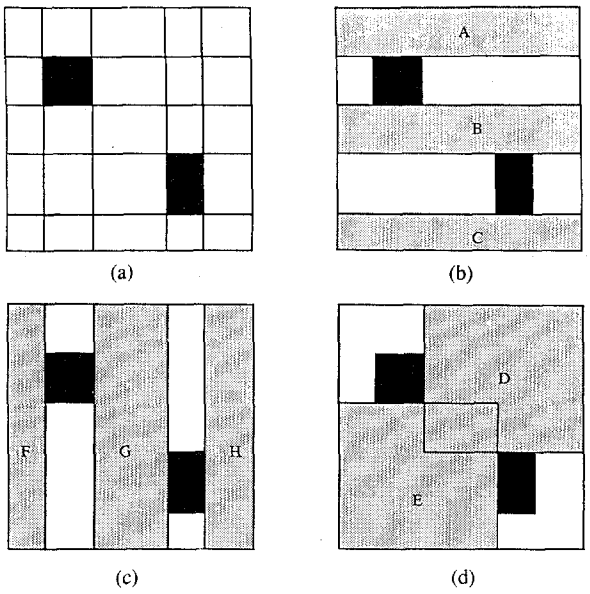


Fig. 1. Typical layout partitioned into a grid because of two obstacles. Layout is shown in (a) and the shaded areas marked A through H in (b), (c), and (d) denote all the prime convex areas for this layout.

Step 3

Make a list of all strings generated by Step 2 such that

1) strings are grouped by the strips that generated them;
2) groups are ordered according to the vertical positions of the generating strips.

Step 4

Generate a new list of strings from the old list of strings based upon the following rules.

1) The new ith group of strings is generated by combining each string from the old ith group with each string from the old $i + 1$th group, $i = 1, 2, \cdots$.
2) Two strings are combined by (logically) ORing the right substrings and (logically) ANDing the left substrings. If the new string has a null (all zero) left substring, discard that string. Otherwise, add it to the new list.
3) Every time a string is added to the new list, check off *all* the strings from the old lists that are covered by the new addition. A string S_1 is said to be covered by a string S_2 if logical ORing of the two strings yields S_2.

Step 5

Repeat Step 4 if the new list generated has two or more groups.

Step 6

A string from any list that is not checked off represents a prime convex area for the layout.

List 1

1 1 1 1 1 1 0 0 0 0 (A)

1 0 0 0 0 0 1 0 0 0 ✓
0 0 1 1 1 0 1 0 0 0 ✓

1 1 1 1 1 0 0 1 0 0 (B)

1 1 1 0 0 0 0 0 1 0 ✓
0 0 0 0 1 0 0 0 1 0 ✓

1 1 1 1 1 0 0 0 0 1 (C)

List 2

1 0 0 0 0 1 1 0 0 0 ✓
0 0 1 1 1 1 1 0 0 0 ✓

1 0 0 0 0 0 1 1 0 0 ✓
0 0 1 1 1 0 1 1 0 0 ✓

1 1 1 0 0 0 0 1 1 0 ✓
0 0 0 0 1 0 0 1 1 0 ✓

1 1 1 0 0 0 0 0 1 1 ✓
0 0 0 0 1 0 0 0 1 1 ✓

List 3

1 0 0 0 0 1 1 1 0 0 ✓
0 0 1 1 1 1 1 1 0 0 (D)

1 0 0 0 0 0 1 1 1 0 ✓
0 0 1 0 0 0 1 1 1 0 ✓
0 0 0 0 1 0 1 1 1 0 ✓

1 1 1 0 0 0 0 1 1 1 (E)
0 0 0 0 1 0 0 1 1 1 ✓

List 4

1 0 0 0 0 1 1 1 1 0 ✓
0 0 1 0 0 1 1 1 1 0 ✓
0 0 0 0 1 1 1 1 1 0 ✓

1 0 0 0 0 0 1 1 1 1 ✓
0 0 1 0 0 0 1 1 1 1 ✓
0 0 0 0 1 0 1 1 1 1 ✓

List 5

1 0 0 0 0 1 1 1 1 1 (F)
0 0 1 0 0 1 1 1 1 1 (G)
0 0 0 0 1 1 1 1 1 1 (H)

Fig. 2. Obtaining all prime convex areas for layout of Fig. 1(a). Prime convex areas remain unchecked and are labeled as A through H.

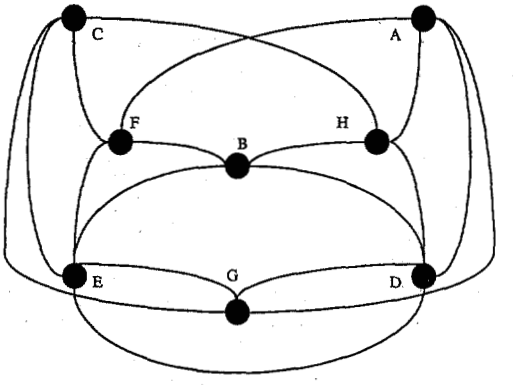
A Quine–McCluskey type of proof [10] can be constructed to show that the algorithm described above does indeed provide *all* the rectangular *prime convex areas*. The prime convex areas corresponding to the layout of Fig. 1, isolated by this procedure, are listed in Fig. 2.

III. Setting Up the Graph

The next step in the path planning is the representation of information about the prime areas (generated in Section II) in a usable data structure. In order to facilitate application of techniques such as orderly graph traversal and backtracking, a graph is set up where each node represents a prime convex area. Two nodes are joined by an arc if the areas they represent intersect and each arc has associated with it information about the geometrical intersection. Fig. 3 shows such a graph obtained from the layout of Fig. 1.

If optimality is not a criterion, traversing the graph is straightforward. Prime convex areas in which source and destination points are located may be determined, and the graph may be traversed from the source node to the destination node using one of a variety of techniques available [13], [14]. However, the consideration of optimality brings about two complications.

- Both the source and destination points may fall inside several different nodes (since the convex areas may intersect). Thus *all* possible paths originating from valid starting nodes (forming set S) and terminating on valid ending nodes (set D) have to be considered.
- Arcs cannot have fixed weights attached to them because any two points in one convex area are not necessarily equidistant to a point in another convex area. The cost of traveling from one node to another is dependent on where the path points are actually located in the convex areas and has to be computed every time an arc between two nodes is chosen.



arc	nodes	area of intersection
a	A - F	1 0 0 0 0 1 0 0 0 0
b	A - G	0 0 1 0 0 1 0 0 0 0
c	A - H	0 0 0 0 1 1 0 0 0 0
d	B - F	1 0 0 0 0 0 0 1 0 0
e	B - G	0 0 1 0 0 0 0 1 0 0
f	B - H	0 0 0 0 1 0 0 1 0 0
g	A - D	0 0 1 1 1 1 0 0 0 0
h	B - D	0 0 1 1 1 0 0 1 0 0
i	G - D	0 0 1 0 0 0 0 1 1 1
j	H - D	0 0 0 0 1 0 0 1 1 1
k	B - E	1 1 1 0 0 0 0 1 0 0
l	C - E	1 1 1 0 0 0 0 0 0 1
m	F - E	1 0 0 0 0 0 0 1 1 1
n	G - E	0 0 1 0 0 0 0 1 1 1
o	D - E	0 0 1 0 0 0 0 1 0 0
p	C - F	1 0 0 0 0 0 0 0 0 1
q	C - G	0 0 1 0 0 0 0 0 0 1
r	C - H	0 0 0 0 1 0 0 0 0 1

Fig. 3. Graph of intersecting prime convex areas for layout of Fig. 1(a) and areas of intersection associated with each arc.

It should be noted here that the isolation of prime convex areas and their representation in a graph needs to be done only once for a given environment and need not be repeated until it changes. Section IV describes a strategy to overcome these difficulties and to choose an optimal path. Complexity issues of the associated algorithm are discussed in Section V.

IV. Dynamic Path Planning

The basic strategy of robot path planning involves traversing the graph generated in Section III from a node containing the source point to a node containing the destination point, and finding the optimal path in terms of graph nodes and consequently a geometric representation of this path. Recall that nodes are connected if the areas they represent intersect. Thus moving from one node to another is geometrically equivalent to choosing a point in the intersection of the two areas. However, the placement of this point within this intersection is dependent on where the path will progress next. A one node look ahead is used in this work and is found to provide reasonable results.

Since graph arcs cannot have fixed weights attached to them, the cost function must dynamically allocate costs to path segments as the path develops. Fig. 4(a) illustrates graph traversal from node X_i to X_{i+1} and then to either node X_{i+2} or to X'_{i+2}. Fig. 4(b) shows the development of the corresponding geometric path. Let a, b, and b' denote areas of intersection of X_i and X_{i+1}; X_{i+1} and X_{i+2}; and X_{i+1} and X'_{i+2}. Also denote by mid(b) and mid(b') midpoints of the two intersections. Assume that the current path has progressed till a point C_i in node X_i.

Assuming that the graph traversal is $X_i \rightarrow X_{i+1} \rightarrow X_{i+2}$,

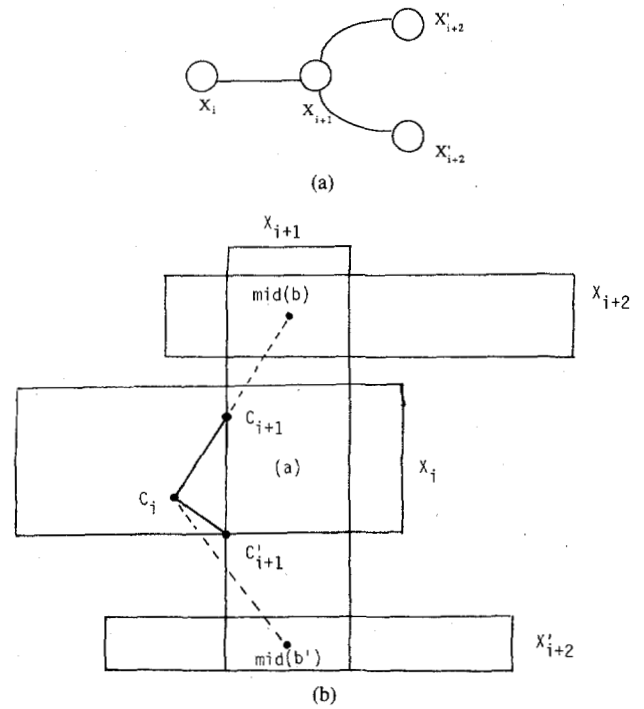


Fig. 4. Development of path segment based upon relative position of a future node. (a) Graph path. (b) Corresponding geometric path.

points C_i and mid(b) are joined by a straight line. If the line intersects area a, then the next path point chosen is the point where the line first meets a. On the other hand, if the line does not intersect a, then the next point chosen on the path is the corner of a that is closest to the line. This second case is illustrated by the graph traversal $X_i \rightarrow X_{i+1} \rightarrow X'_{i+2}$. The point chosen is labeled C_{i+1} and the path segment $C_i \rightarrow C_{i+1}$ is added to the current path. Since the new point C_{i+1} is now in X_{i+1}, a similar procedure could be used to continue path planning till the destination node is reached.

Since a one node look ahead is used, path cost assignments cannot begin until the graph node path progresses at least to the third node. Similarly, when X_{i+1} is the final destination node (and therefore there is no X_{i+2} node), the final destination point itself is used in place of mid(b) in the above description to compute the last two path segments. Hence the graph traversal technique used maintains two concurrent paths, one going through the graph nodes X_i and the other going through path points C_i. The list of C_i lags behind the list of X_i by one step for the reason described above.

An exhaustive graph search for optimal path is performed using a backtracking procedure. It involves extending the path till either a destination node is reached or till a new node cannot be found. This may be due to two reasons.

- Nodes connected to the current node may already be on the path.
- The cost of the path generated so far exceeds the cost of a source-to-destination path already established.

If the current path cannot be extended, the algorithm backtracks by dropping the last node from the current path and continues moving forward along another arc. Once at least one path between the source and the destination is established, a backtrack from a node A to node B implies that all possible ways of going from A to one of the destination nodes have been explored. Knowing the costs of these alternatives, it is possible to obtain the minimum cost of a path from A to a destination node and, consequently, the minimum cost of a path from B to a destination node via arc $B \rightarrow A$.

Since an exhaustive search can be immensely time consuming, a parameter β was introduced to dynamically shrink the graph as the search progresses. Suppose that there are alternate paths of differing costs to go from an intermediate node A to the destination node. If the graph arcs had preassigned weights, one would have been able to eliminate all but the minimum weight path from all future considerations. In this case, however, the costs of these paths depend not only upon the node A but also on the position of the path point in the large area represented by node A. Since the position of this point in node A depends upon how one arrived at node A, one can no longer assume that one would obtain the same path lengths from A to the destination in the next visit to A. All those paths whose lengths are within a factor β (≥ 1) of the minimal path length are allowed to survive for future traversals. The other paths from the particular node A are deleted by removing appropriate arcs originating from A. Fig. 5 shows the process of links being deleted. If one is backtracking from node B to node A, then it means that all possible nonredundant paths from nodes D, E, F, and G to the destination node have been explored. The minimum cost of reaching the destination from each of D, E, F, and G are stored as α_1, α_2, α_3, and α_4, respectively. Note that δ_1, δ_2, δ_3, and δ_4 are the costs of the arcs between B and D, E, F, and G. The pair $\delta_1 + \alpha_1$ that represents the smallest cost is compared with other pairs. For all pairs that have a cost that is greater by a factor β, the corresponding arcs are deleted. This modification is performed while backtracking from node B because that is the only time when the exploration of all the alternatives of going from B to the destination is complete.

The considerations outlined above result in the following graph traversal algorithm.

Dynamic Cost Allocation Graph Traversal Algorithm

initialize
bestcost := ∞, currentcost := 0
unmark all graph arcs
set tcost for all arcs not emanating from a destination node = ∞
set tcost for all arcs emanating from a destination node = 0
{tcost associated with an arc $X_i \rightarrow X_j$ is the tentative minimum cost of reaching a destination node via that arc starting from X_i}

findnewnode [X_{i+1}]
Choose X_{i+1} such that
an arc exists between X_i and X_{i+1}
and X_{i+1} is not on current node path
and X_{i+1} is not in S
and the arc between X_i and X_{i+1} is not marked {the arc has not already been considered and rejected}

moveforward [to X_{i+1}]
Add X_{i+1} to current node path

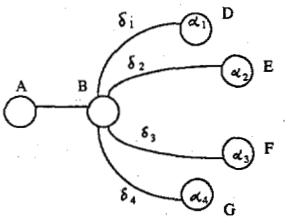


Fig. 5. Deleting links from graph.

Determine C_i {as explained in text}
Add C_i to current point path
Add cost of segment $C_{i-1} \rightarrow C_i$ to current cost
$i := i + 1$

backtrack [from X_{i+1} to X_i]
Unmark all arcs originating from X_{i+1}
Mark arc from X_i to X_{i+1}
min := minimum tcost associated with an arc emanating from X_{i+1} (except $X_{i+1} \rightarrow X_i$)
if min $< \infty$ then delete each arc $X_{i+1} \rightarrow X_j (j \neq i)$ with tcost $> \beta \cdot$ min
and set (tcost of $X_i \rightarrow X_{i+1}$) := m + cost of segment $C_i \rightarrow C_{i+1}$
Reduce the current cost by the cost of path $C_{i-1} \rightarrow C_i$
Remove X_{i+1} from current node path
Remove C_i from current point path
$i := i - 1$

path planning {*main program*}
determine S, D;
if $S \cap D \neq$ nil then
compute straight line path
else
initialize
for every $X_0 \in S$ do
$i := 0$; backtrackflag := false
findnewnode $[X_{i+1}]$
while (newnode exists) or ($i > 0$) do
if backtrackflag = true then
backtrack [from X_{i+1} to X_i]
findnewnode $[X_{i+1}]$
endif
if newnode exists then
moveforward
if (currentcost > bestcost) then
backtrackflag := true
else
findnewnode $[X_{i+1}]$
endif
endif
endwhile
endif

An analysis of the complexity of this algorithm and the results obtained by its simulation are presented in Section V.

V. Analysis and Simulation of the Algorithm

Performance of the algorithm presented in Section IV was evaluated in terms of two characteristics—optimality of the results and speed of computation. Near optimality is obtained by taking into consideration *all* the rectangular prime convex areas and by conducting a complete search of the graph obtained. These very same considerations unfortunately also imply poor execution speed. Since both speed and optimality may not be realized simultaneously, the parameter β is used to systematically trade off optimality for execution speed.

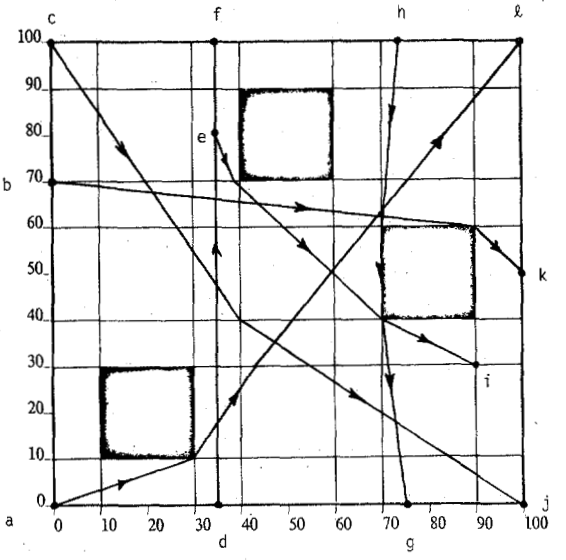


Fig. 6. Sample layout with three obstacles partitioned into 7 × 7 grid by distinct edges of objects and sample paths obtained by procedure of this paper for $\beta = 1$.

Recall now that β determines which graph arcs are deleted while backtracking. If β is very large, no arcs are removed and the search for an optimal path proceeds through all possible travel patterns. On the other hand, if $\beta = 1$ at every backtrack, all but one arc going out from the backtracked node is deleted. The only arc to survive is the one corresponding to the minimum length path from that node to the destination. The computational time in this case is much smaller than in the first case but the results obtained are suboptimal since a highly restricted search is performed.[1] The algorithm described in Section IV was applied to a layout containing three obstacles. β was varied between 1 and 2. Fig. 6 shows the layout and the paths produced between six pairs of source and destination points for $\beta = 1$. Since none of the edges of the different obstacles coincide, they divide the environment into a 7 × 7 grid (determined by the placement of obstacle edges). Table I compares the costs of paths obtained by our routine with the absolute minimum costs of the paths. It also lists the amount of time required by our algorithm (implemented in Pascal on a DEC 2065) to compute each path. As can be seen from the table, the procedure suggested is relatively fast and the paths produced are very close to optimal. It can also be seen that as β increases, paths tend to get closer to optimal but the computational time required increases.

Fig. 7 shows the same paths in a "cluttered" environment obtained by adding more obstacles to the layout of Fig. 6.

[1] Note that the time required to obtain the set of prime convex areas and to set up the graph need not be considered important since these tasks are performed only once for any environment and need to be repeated only if the obstacles are repositioned. The graph search time, on the other hand, is important since it relates to a repetitive task.

TABLE I
COMPARISON OF PATH LENGTHS OBTAINED BY PROCEDURE OF THIS PAPER WITH OPTIMUM PATH LENGTHS FOR LAYOUT OF FIG. 6

Sample Path	Optimal Path Length	Percent Deviation from Optimality $\beta = 1$	$\beta = 1.5$	$\beta = 2$	Computation Time (s) $\beta = 1$	$\beta = 1.5$	$\beta = 2$
a–l	145.64	0.09	0	0	5.09	6.42	7.85
e–i	75.97	0.72	0.72	0.72	4.13	4.56	4.92
c–j	141.42	1.98	1.98	0.31	3.90	5.21	6.42
b–k	104.70	0	0	0	4.97	5.95	7.38
d–f	100.00	0	0	0	0.006	0.006	0.006
h–g	100.62	0	0	0	4.82	6.18	7.62

Time required to set up the graph and the database was 1.42 s.

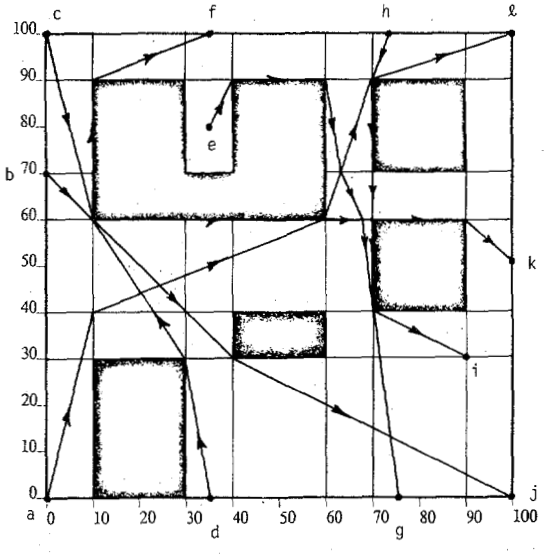


Fig. 7. Sample cluttered layout partitioned into 7 × 7 grid by distinct edges of objects and sample paths obtained by procedure of this paper for $\beta = 1$.

Note, however, that this addition has been done without changing the grid by aligning the new objects with the objects already present. Optimality of the resultant paths and the computational time in this setup are presented in Table II. It can be seen that the parameter β has the same effect in a cluttered environment. A comparison of Table II with Table I shows that the computational times required in a cluttered environment are substantially lower than in a simple environment. This is attributed to a lower number of arcs in the graph. The graph corresponding to the layout of Fig. 6 has 13 nodes and 96 directed arcs whereas that of Fig. 7 has 12 nodes and only 34 directed arcs.

In order to understand the relationship of the computational time to the problem parameters, four different environments with five nontrivial paths in each were simulated. The results shown in Table III indicate that the computational time depends on the number of nodes, the number of arcs, and also the structure of the graph as in any other backtracking algorithm [13]. It is also clear that the time is more strongly dependent on the number of nodes than on the number of arcs.

Upper bounds on the number of nodes and arcs in the graph can be derived as follows. Assume that there are n nonoverlapping objects with no edges aligned. The layout is then partitioned by the edges of the objects into a $(2n + 1) \times (2n + 1)$ grid. If $n \neq 0$, each free area is bounded at least on one side by an obstacle. To estimate the number of prime convex areas bounded either on the east or the west by an obstacle, note that for any pair of obstacles, E and W (with E on the east of W), there can only be *one* prime convex area that is bounded by both the west face of E and the east face of W. This is because if there are two areas A_1 and A_2 enclosed by the same pair of objects E and W with the north edge of A_1 further north of the north edge of A_2, then the object limiting A_2 on the north will be within A_1. A_1 will thus be unable to qualify as a free convex area. Thus A_1 and A_2 must have the same north edges. A similar argument shows that their south edges also coincide. Therefore there is at most one area bounded by the west face of E and the east face of W. The number of ways in which n objects and two north–south boundaries can be paired is $(n + 2)(n + 1)/2$. However, the area enclosed by the pair made up of the east–west boundaries need not be considered since only the areas which are bounded either on east or west (or both) by obstacles are being counted. Thus there are only $(n + 2)(n + 1)/2 - 1 = n(n + 3)/2$ such prime convex areas. The same number of prime convex areas can be accounted for by a north–south argument. Thus the number of prime convex areas is bounded above by $n(n + 3)$ and the number of graph nodes is of $O(n^2)$. Notice that this is a loose upper bound since many pairs of faces counted in the above argument may not indeed give rise to a free convex area and some areas may be counted twice, once in the east–west argument and once in the north–south argument.

Suppose now that a new obstacle with all its edges aligned to earlier edges is added to the setup. It may be paired with all other obstacles, giving rise to possible new convex areas. At the same time, however, it prohibits a pair of obstacles, one on its north–east and the other on south–west, from enclosing a free convex area. Similarly, pairing of obstacles on its south–east with those on its north–west is no more possible. This results in the reduction of many of the free convex areas counted in the earlier argument. A large number of simulations were carried out to establish the dependence of the number of graph nodes on the number of objects thus added to the layout. The results obtained indicate that a random addition of aligned objects almost always results in a graph with number of nodes bounded by the limit based on the number of

TABLE II
COMPARISON OF PATH LENGTHS OBTAINED BY PROCEDURE OF THIS PAPER WITH OPTIMUM PATH LENGTHS FOR LAYOUT OF FIG. 7

Sample Path	Optimal Path Length	Percent Deviation from Optimality			Computation Time (s)		
		$\beta = 1$	$\beta = 1.5$	$\beta = 2$	$\beta = 1$	$\beta = 1.5$	$\beta = 2$
a–l	152.69	3.60	0.01	0.01	1.95	2.70	3.07
e–i	104.53	0.21	0	0	1.29	1.34	1.60
c–j	150.74	0	0	0	0.98	1.30	1.91
b–k	108.28	0	0	0	2.04	2.86	3.35
d–f	123.40	0	0	0	1.80	2.30	2.74
h–g	101.49	0	0	0	1.62	1.62	2.06

Time required to set up the graph and the database was 1.85 s.

TABLE III
DEPENDENCE OF GRAPH PARAMETERS AND PATH COMPUTATION TIME (AVERAGED OVER FIVE NONTRIVIAL PATHS FOR $\beta = 1$) ON NUMBER OF OBJECTS AND SIZE OF GRID OBTAINED BY DISTINCT EDGES OF OBJECTS

Number of Objects	Grid Size	Number of Graph Nodes	Number of Graph Arcs	Average Path Computation Time (s)
3	7 × 7	13	96	4.60
10	7 × 7	12	34	2.27
10	7 × 7	11	42	1.59
26	11 × 11	30	212	46.85

objects with distinct edges. This is also illustrated by the fourth row of Table III. In this layout there are five obstacles with distinct nonoverlapping edges, and consequently the grid created by the distinct object edges is 11 × 11. The number of nodes for this five obstacle layout is bounded by $5(5 + 3) = 40$. The table shows that even after adding 21 additional (aligned) objects to the setup, the graph has only 30 nodes, well within the five obstacle bound.

The number of arcs in the graph could be bound above (very loosely as before) by assuming that the graph is a complete graph, i.e., each prime convex area intersects with every other. Since it is known that the number of nodes in a layout partitioned into $(2n + 1) \times (2n + 1)$ grid by the distinct edges of the obstacles is limited to $n(n + 3)$, the number of *directed* arcs in the complete graph is bound by $n(n + 3)(n^2 + 3n - 1)$.

VI. CONCLUSION

The path planning procedure outlined in this paper differs from earlier methods in that it takes advantage of *all* the free (rectangular) convex areas in a layout. This allows for most near-optimal paths made up of straight line segments to be found efficiently. In addition, if the source and the destination points belong to the same convex area, the optimal path is picked trivially. This procedure calls for the representation of the relationships between convex areas through a graph and use of a backtrack procedure modified for dynamic cost allocation. Because overlap of convex areas is permitted, the graphs generated are a little more complex than the ones used by previous researchers. However, even in the worst case, the number of nodes in our graph is $O(n^2)$, where n is the number of obstacles. If obstacle edges line up at all, as in the case of industrial layouts, the graph complexity decreases drastically. Simulation of the modified backtrack procedure used here indicates that most near-optimal paths can be found relatively quickly. The number of arcs which represent the intersections of convex areas generally decrease if the setup is cluttered with aligned objects. This results in a faster search through the graph and consequently speed improvement for the procedure.

Another advantage of this procedure is the need to maintain a relatively small database which may be precomputed (without knowledge of source and destination points) rapidly from a map of the obstacles. In order to come up with a near-optimal path, a graph node path as well as a geometric path are concurrently developed.

While dynamically establishing the geometric path, this procedure looks ahead only one node in the node path. Consequently the path segment $C_{i+1} \rightarrow C_{i+2}$ is determined independently of the segment $C_i \rightarrow C_{i+1}$. It is therefore possible in certain cases that the path obtained $C_i \rightarrow C_{i+1} \rightarrow C_{i+2}$ is not the optimal path from $C_i \rightarrow C_{i+2}$ (in particular when C_i and C_{i+2} can be joined by a straight line). This drawback can, however, be overcome by refitting the geometric path every time a new point is added to the path.

The speed of the algorithm is governed by both the graph size and the structure. Since the number of graph nodes in an n obstacle layout is bounded by $O(n^2)$, execution speed is roughly of the same order. However, a much lower bound generally holds if many of the objects are aligned. This suggests that the procedure is more suited to situations where the obstacles are positioned in a somewhat regular fashion. Further, parameter β, introduced in the backtracking graph traversal algorithm, allows a systematic trade-off between the path optimality and the execution time.

There is no learning capability that is incorporated in the algorithm as described. However, it can easily be added in the following manner. Procedure findnewnode can be configured to look for the best possible next node to approach the present destination node, rather than the first valid one. This can be done by keeping track of how many times each link of each node was chosen in past successful paths on the way to a given destination. This approach would serve to cut down the graph search considerably because there would be a greater chance that the best paths would be found early in the search. Thus less time would be spent looking at entire paths with costs greater than the best cost. It should be noted, however, that even though this learning capability would improve the computational speed, it would be expensive in terms of its memory requirements.

References

[1] T. Lozano-Perez and M. A. Wesley, "An algorithm for planning collision-free paths among obstacles," *Commun. ACM,* vol. 22, pp. 560–570, Oct. 1979.

[2] L. Guibas and J. Hershberger, "Computing the visibility graph of n line segments in $O(n^2)$ time," *Theoret. Comput. Sc.,* vol. 26, pp. 13–20, Sept. 1985.

[3] E. Welzl, "Constructing the visibility graph of n line segments in $O(n^2)$ time," *Info. Processing Lett.,* vol. 20, pp. 167–171, Sept. 1985.

[4] C. E. Thorpe, "Path relaxation: Path planning for a mobile robot," Carnegie Mellon Univ., Tech. Rep. CMU-RI-TR-84-5, 1984.

[5] J. L. Crowley, "Navigation for an intelligent mobile robot," Carnegie Mellon Univ., Tech. Rep. CMU-RI-TR-84-18, Aug. 1984.

[6] R. Chatila, "Path planning and environment learning in a mobile robot system," presented at the European Conf. on Artificial Intelligence, Orsay, France, 1982.

[7] R. A. Brooks, "Solving the find path problem by good representation of free space," *IEEE Trans. Syst., Man Cybern.,* vol. SMC-13, pp. 190–197, Mar. 1983.

[8] D. T. Kuan, J. C. Zamiska, and R. A. Brooks, "Natural decomposition of free space for path planning," presented at the IEEE Conference on Robotics and Automation, St. Louis, MO, March 1985.

[9] J. S. Singh, "Path planning and navigation for a mobile robot," Master's thesis, Lehigh University, Aug. 1985.

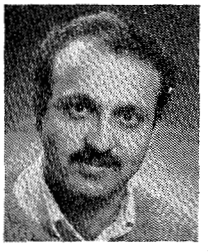
[10] E. J. McCluskey, Jr., "Minimization of Boolean functions," *Bell Syst. Tech. J.,* vol. 35, pp. 1417–1444, Nov. 1956.

[11] W. V. Quine, "The problem of simplifying truth functions," *Amer. Math. Monthly,* vol. 59, pp. 521–531, Oct. 1952.

[12] ——, "A way to simplify truth functions," *Amer. Math. Monthly,* vol. 62, pp. 627–631, Nov. 1955.

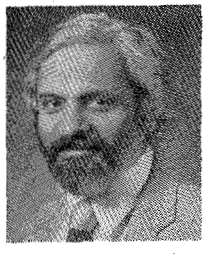
[13] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms.* Rockville, MD: Computer Science, 1984.

[14] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms.* Reading, MA: Addison Wesley, 1968.



J. Sanjiv Singh was born in Barrelli, U.P., India, on March 3, 1962. He received the B.S. degree in computer science from the University of Denver, Denver, CO, in 1983 and the M.S. degree from Lehigh University, Bethlehem, PA, in 1985. He was involved in the development of a mobile factory robot. His thesis described a method to plan optimal paths for such a robot.

He is currently employed as a Research Staff Member at the Construction Robotics Laboratory at Carnegie Mellon University, Pittsburgh, PA, and is primarily involved in the development of navigation systems for NavLab, an autonomous vehicle. His current research interests are in control and artificial intelligence applications to mobile robots.



Meghanad D. Wagh (M'81) received both the B.Tech. and Ph.D. degrees in electrical engineering from the Indian Institute of Technology, Bombay, India.

Currently, he is an Associate Professor of Computer Science and Electrical Engineering at Lehigh University, Bethlehem, PA. Prior to that he was an Assistant Professor of Electrical Engineering at Old Dominion University, Norfolk, VA. His research interests include design of architectures and algorithms for digital signal processing and applications of abstract algebra to problems in electrical engineering.