ROBOT PATH PLANNING USING INTERSECTING CONVEX SHAPES

SANJIV SINGH* - MEGHANAD D. WAGH**

* Civil Engg., Robotics Lab, Carnegie Mellon Univ., Pittsburgh, PA 15213
** Dept. of Computer Science & Electrical Engg., Lehigh Univ., Bethlehem, PA 18015

Abstract. This paper deals with an automated path planning algorithm for a mobile robot in a structured enviornment. The algorithm is based upon finding *all* the largest (prime) free convex areas in the environment and representing this information in the form of a graph. A graph traversal algorithm which exploits back-tracking as well as dynamic cost allocation to graph arcs is presented and simulated. A strategy to trade off the optimality of the results for a smaller computation time is described.

1. Introduction.

Considerable attention has been focused on automatically planning paths for mobile robots between arbitrary source and destination points and a variety of different algorithms are now available to solve this problem[1-5]. Most of these algorithms however, yield either non-optimal solutions or are computationally expensive. A new path finding strategy that is computationally efficient and yields near-optimal results has been reported recently[1]. This paper extends the results presented in that work by studying the trade off between the computation time and the optimality of the paths obtained.

The simplest approach to the path finding problem involves creation of a connectivity graph where each node represents a possible source or destination point and each link has associated with it a predetermined path of corresponding length. Determination of a path between a given pair of source and destrination points can then be done either by searching the connectivity graph for the shortest path, or by looking up a table of precomputed best paths from all possible source destination pairs. However, this method is severely limited- there is no scope to start or to end at a point not included in the initial list of points. Also, modifying this list is time intensive as well as futile for the case where there may be arbitrary starting and ending points. In the past, this method has been used with some amount of success in situations where the robots are only required to repeat a few fixed paths and the probability of adding a new path to the list is very low.

Earlier research on this problem by Ignat'yev[2], and Lozano-Perez[3] reported a technique called V-Graph which uses a graph of vertices between which travel is possible in a straight line. This is essentially a table of which nodes (vertices) are "visible" (can be traveled to in a straight line) from each node. However, every time the source and destination points are specified, the graph has to be augmented with new nodes and new links. Thus, not only does the resultant graph have a large number of links but the establishment of these links is highly complex.

The other approach used more recently[1,4,5] has been to partition the free space into convex polygons since any two points in a convex shape can be joined with a straight line without leaving the shape. If convex shapes 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 non-overlapping convex shapes[4,5]. Development of the path depends on traversing the connectivity graph that is produced by representing free convex polygons as nodes. Nodes corresponding to polygons with common edges are joined by arcs. The problem with a strategy that breaks up space into non-overlapping 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 that the procedure is not aware of. This is a natural consequence of the fact that the need for non-overlapping areas overlooks a considerable number of convex areas in the layout. Further, if the paths are not dynamically refitted to be optimal, paths that would be "naturally" straight, turn out to be quite contrived. This effect is particularly pronounced if there are relatively large free areas to contend with. However, there is a one-to-one correspondence between the source and destination points in free space and the graph nodes, and thus the method is successful in getting around the high computational expense at the cost of optimality.

In this paper, we expand upon the work reported in Singh[1] which exploits the concept of convexity by identifying *all* the largest rectangular free areas. In order to achieve near-optimality, without sacrificing computational efficiency, 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. The cost function used in this paper is the length of the path. In order to improve the computational complexity, and to provide a reasonable data base, the obstacles as well as free areas are chosen

CH2282-2/86/0000/1743$01.00 © 1986 IEEE

to be rectangular in shape. The obstacles are grown in size and equivalently the robot is shrunk to a point to simplify path planning without collisions. The advantages of this method are that it allows the extraction of all near straight line paths, precomputation of the database and graph generated, and a systematic trade-off between the optimality and computation time.

2. Path Planning Using Intersecting Convex Shapes.

For computational simplicity, we restrict our attention only to rectangular free areas (there are an infinite number of nonrectangular free areas). Given a map of the boundaries and the obstacles, the environment is partitioned by the edges of these shapes into a grid of at most $2n+1$ x $2n+1$ rectangles where n is the number of such inadmissible areas representing obstacles. Each such rectangle can be represented by a pair of binary strings each at most $2n+1$ bit long. The left sub-string represents the relative x position and the right the y position. For example, a partition that is second from the left and third from the top could be represented by the string

0 1 0 0 0 0 0 1 0 0

A similar notation can be used for areas made up of several of the rectangles. A string

1 1 0 0 0 0 0 1 1 0

for example, represents a convex area made up of 4 cells:

1 0 0 0 0 0 0 1 0 0,
0 1 0 0 0 0 0 1 0 0,
1 0 0 0 0 0 0 0 1 0 and
0 1 0 0 0 0 0 0 1 0.

A **prime convex area** is a free area which is not part of any other free area. The following algorithm may be used to identify all the prime convex areas by fusing together the free rectangular cells from the grid described above. This algorithm is, in some sense, similar to the Quine-McCluskey technique[6] used to identify *all* the prime implicants of a logical expression.

Prime Convex Area Identification Algorithm

Step 1. Represent *each* horizontal strip (made up of horizontally aligned cells) by means of a pair of $2n+1$ length binary strings. The right sub-string has only one bit set corresponding to the vertical position of the strip. The left sub-string has those bits set which correspond to the free rectangles in the strip.

Step 2. Find all the contiguous free rectangles in each horizontal strip. This is done by breaking up the left sub-string into contiguous runs of 1's and repeating the right sub-string in each part.

Step 3. Make a list of all strings generated by step 2 such that

1. Strings are grouped by identical right sub-strings.
2. Groups are ordered by the position of 1s in the right substrings.

Step 4. Generate a new list of strings from the old list of strings based upon the following rules:

1. The new i-th group of strings is generated by combining each string from the old i-th group with each string from the old i+1-th group. i = 1,2,...
2. Two strings are combined by (logically) OR-ing the right sub-strings and (logically) AND-ing the left sub-strings. If the new string has a null (all zero) left sub-string, 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 OR-ing 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.

The next step in path planning is the setting up of a graph with prime convex areas as nodes. Two nodes are joined by an arc if the areas they represent, intersect. An arc has associated with it information about the area of intersection of the prime areas representing its ends. Fig. 1 shows a typical layout with two objects and the corresponding graph is given in Fig. 2.

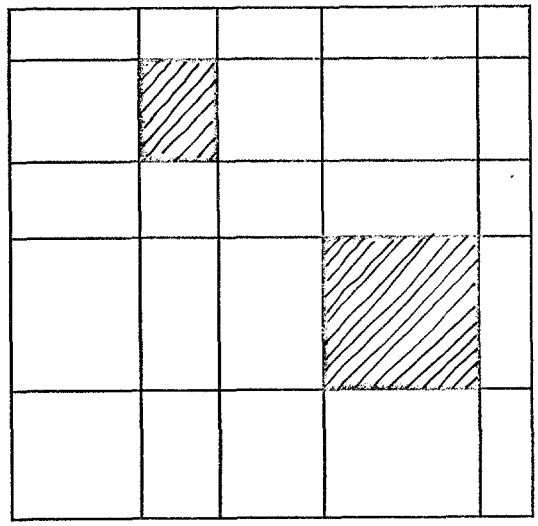


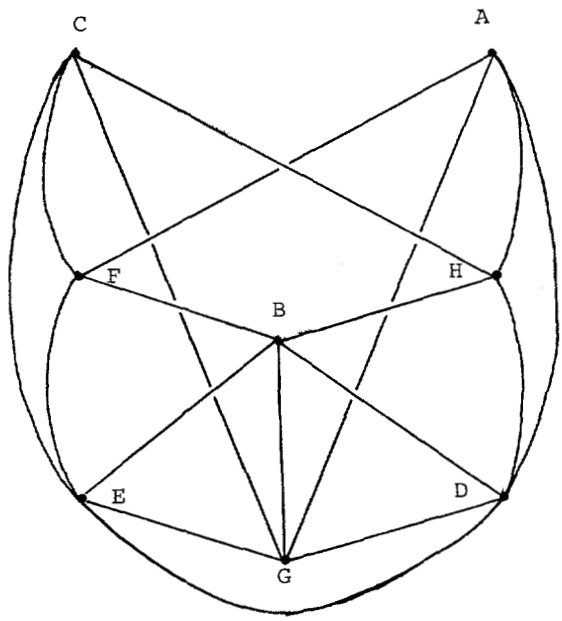
Fig. 1. An environment partitioned by the edges of obstacles

If optimality is not a criterion, the prime convex areas in which the 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[7,8]. The consideration of optimality, however, brings about two complications:

- Both the source and destination points may fall inside several different nodes (since we have intersecting prime convex areas). Thus all possible paths originating from *valid* starting nodes and terminating on *valid* ending nodes 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. This means that the cost of traveling from one node to another is dependent on where the points are actually located in the convex areas and has to be computed every time a path segment is chosen.



graph node	corresponding prime convex area
A	1 1 1 1 1 1 0 0 0 0
B	1 1 1 1 1 0 0 1 0 0
C	1 1 1 1 1 0 0 0 0 1
D	0 0 1 1 1 1 1 1 0 0
E	1 1 1 0 0 0 0 1 1 1
F	1 0 0 0 0 1 1 1 1 1
G	0 0 1 0 0 1 1 1 1 1
H	0 0 0 0 1 1 1 1 1 1

Fig. 2. Graph of intersecting prime convex areas for the layout of Fig. 1

- One should therefore dynamically assign costs to path segments as the actual point path develops. Fig. 3 illustrates the graph traversal from node X_i to X_{i+1} and then to either node X_{i+2} or to X'_{i+2}. 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. For the sake of choosing the next

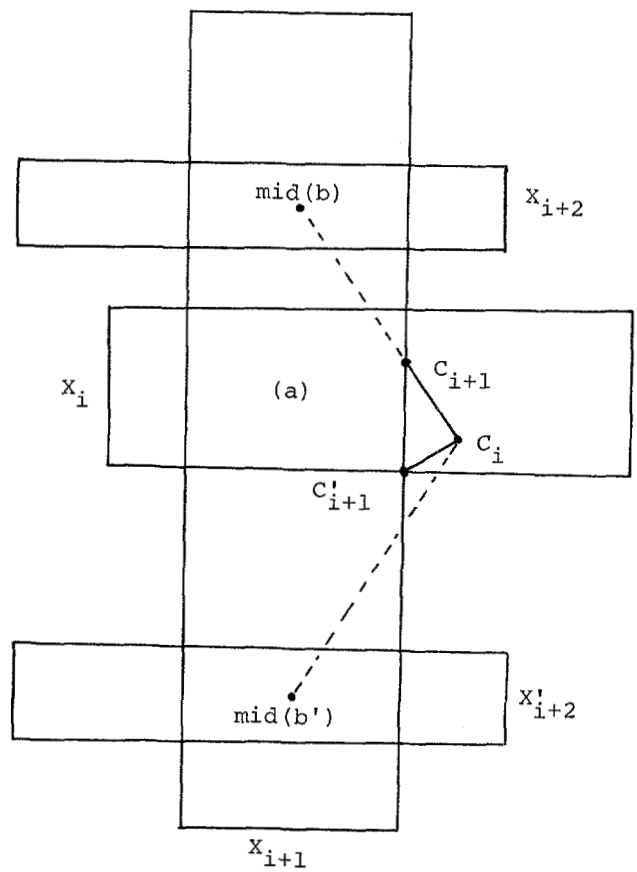


Fig. 3. Development of a path segment based upon relative position of a future node

segment of the shortest path, one should look ahead to nodes to be visited in the future. In the procedure presented here, we look ahead only one node

Assuming that the graph traversal is $X_i \rightarrow X_{i+1} \rightarrow X_{i+2}$, join points C_i and $mid(b)$ 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. For the purposes of this paper, the cost allocated to this segment is merely proportional to its length. 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.

It can be seen that the cost allocation to the developing path lags behind the graph traversal by one step. Consequently, 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. 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. Note that the list of C_i lags behind the list of X_i for the reason described above. Assuming X_i is the current node, C_{i-1} is the current point on the path and, **S** and **D** denote the sets of starting and ending nodes, respectively, the procedure reads as follows:

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 → 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
    Determine C_i       {as explained earlier}
    Add C_i to current point path
    Add cost of segment C_{i-1} → 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} → X_i)
    if min < ∞ then delete each arc X_{i+1} → X_j
           with tcost > β·min   (j ≠ i)
       and set (tcost of X_i → X_{i+1}) := m + cost
           of segment C_i → C_{i+1}

    Reduce the current cost by the
           cost of C_{i-1} → 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 ∩ D ≠ nil then
        compute straight line path
    else
        initialize
        for every X_0 ∈ 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
                      if X_{i+1} ∈ D then
                          copy best path
                          backtrackflag := true
                      else
                          findnewnode[X_{i+1}]
                      endif
                  endif
                endif

            endwhile
```

3. Results of Simulations.

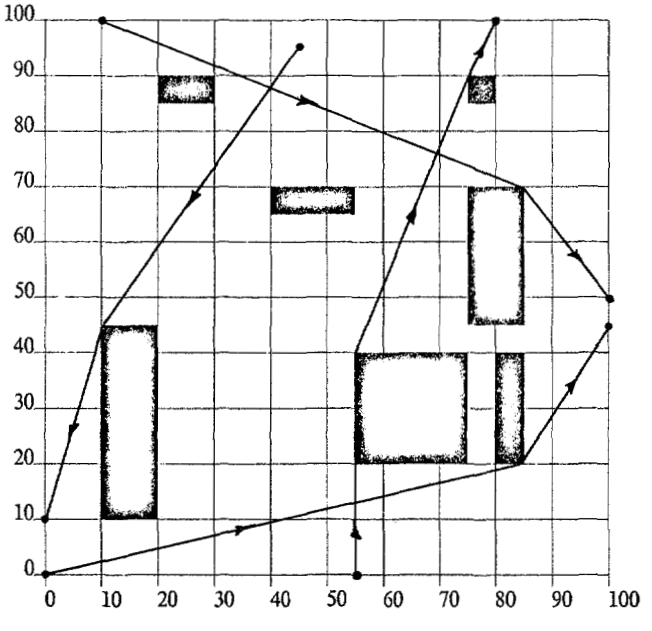
This section describes the results obtained by simulating the algorithms of Sec. 2 for the layout shown in Fig. 4 for the four indicated source destination pairs. Parameter β used in the **backtrack** procedure of the *Dynamic Cost Allocation Graph Traversal Algorithm* was important to strike a balance between the computational time of the algorithm and the optimality of the solution. In the simulations reported, β was varied between 1 and 1.9. The amount of time it took the graph traversal algorithm to come up with a path is shown as a function of β in Fig. 5. Fig. 6 shows the dependance of the deviation from optimality (averaged over all four paths) on β. It may be seen that as β increases, the computation time goes up. At the same time the optimality improves but the small set of data and limited range of β used here does not illustrate this fact fully. The impact of β on the algorithm performance can be understood as follows. When one backtracks from point X_{i+1} to X_i, it is because all the possibilities of reaching the destination from X_{i+1} have been explored. If a particular path, amongst those explored, has a cost that is substantially lower than the others, then obviously the other paths should be discarded.

In the present case, however, there is an additional complication. As depicted in Fig. 3, the actual cost of the path from a node to the destination point is based upon the "point path" and not on the "node path". Thus if one arrives at X_{i+1} from different nodes, the actual point paths to the destination (even though they may go through the same nodes) will have differing costs. In fact, this same consideration keeps us from assigning static costs to the graph arcs. However, it is reasonable to assume that the costs of paths originating from

different points in the same node, going through the same set of nodes and terminating at the same destination point may not differ from one another by large percentages. We use β to estimate the relative cost difference between such paths and discard a node path originating from node X_{i+1} if the cost of its currently associated point path is atleast β times greater than the cost of a point path associated with another node path.

The parameter β thus allows us to adjust for the differences between the point path and the node path. For $\beta = 1$, the algorithm converts to one that identifies a node path with a point path and assumes that the cost of node path is independent of the location of the starting point within the origination node. Conversely, $\beta = \infty$ implies that the node path and the associated point path are independent of each other and the cost of a node path could change drastically if the starting point is located differently within the origination node.

We discard a node path $(X_{i+1}, X_{i+2}, \ldots)$ by deleting the arc from X_{i+1} to X_{i+2}. It should be pointed out here that since the arcs in the original graph are not directed, this deletion should not alter the arc from X_{i+2} to X_{i+1}. In the data structure used in our programs, we achive this by representing each graph arc through two directed arcs. These deletions ensure that as the search for an optimal path progresses, the number of arcs in the graph decrease and the search becomes faster.



Source	Destination
(55, 0)	(80, 100)
(10, 100)	(100, 50)
(45, 95)	(0, 10)
(0, 0)	(100, 45)

Fig. 4. The Simulated layout and the optimal paths

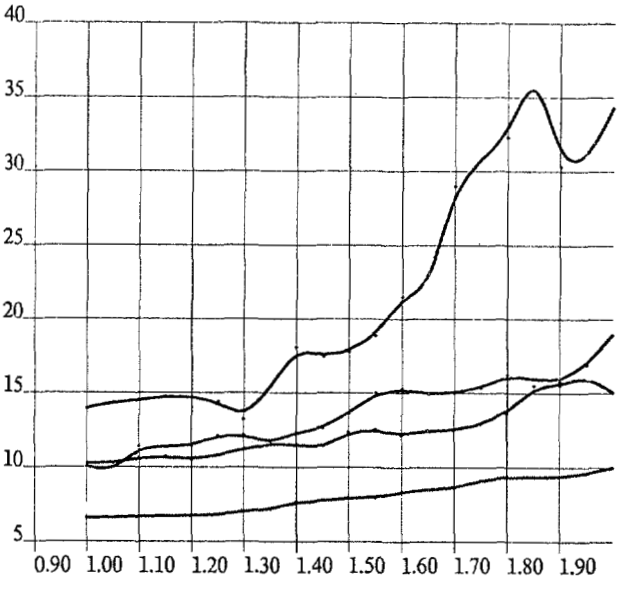


Fig. 5. Computation time for the paths in the layout of Fig. 4

As β aproaches unity, a large number of arcs may be deleted at every backtrack step and therefore the graph size and the computation time reduces quickly. Fig.s 6 and 8 show this predictable trend in the computation time. Similarly as β decreases from ∞, the

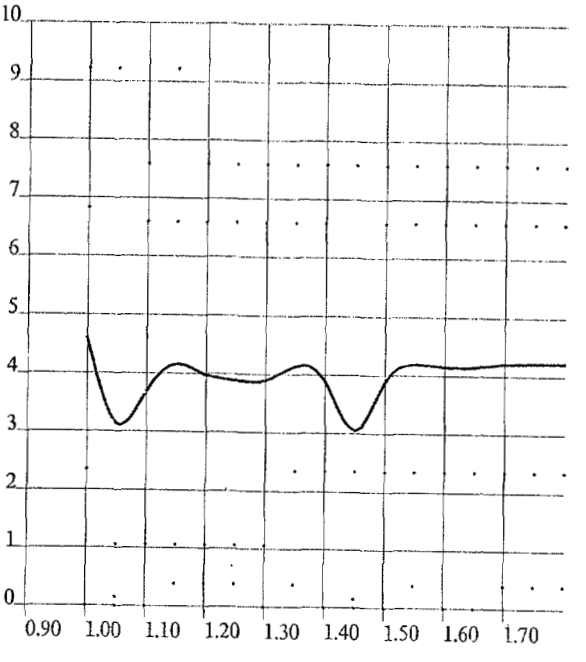


Fig. 6. Average deviation from optimality of the paths in layout of Fig. 4

path obtained from the algorithm moves further and further away from the optimal path. However, the entire procedure being discrete, the percent deviation from the optimality does not exhibit a nice and smooth monotonic characteristics. From the limited number of simulations performed, one may see that for many cases, even for β 1, the deviation from optimality is not very large. However, it should be noted that one may sometimes come across cases for which a decrease of β to such small values may cause a large deviation from optimality. In a realistic application, this may be acceptable since the optimality criterion is not necessarily based only upon the shortest distance. Finally, before this technique is applied, one may have to obtain suitable range of β values for the given layout through typical simulations.

4. Conclusions.

The algorithm presented here appears to hold promise for planning paths of robots that operate in a structured environment. It makes use of *all* the convex areas a robot can travel through freely. As a consequence, the paths obtained are more optimal but because of a larger graph size, take longer to compute. It has been shown that by controlling a particular parameter (β) in the algorithm, one may be able to systematically trade-off the optimality of solution for the computational time. This may prove to be a very useful feature of this algorithm. Finally, it may be noted that if the obstacles are better lined up (as is the case for most industrial set-ups), then both the computation time and the data base size required by this algorithm are both drastically reduced.

References.

[1] S. Singh, "Path Planning and Navigation for a Mobile Robot," Master's thesis, Department of Computer Science and Electrical Engineering, Lehigh University, August 1985.

[2] M. B. Ignat'yev, F. M. Kulakov and A. M. Pokrovsky, "Robot Manipulator Control Algorithms," Tech. Report JPRS 59717, NTIS, August 1973.

[3] T. Lozano-Perez and M. A. Wesley, "An algorithm for planning collision-free paths among obstacles," *Communications of ACM*, vol. 22, pp. 560-570, October 1979.

[4] J. L. Crowley, "Navigation for an Intelligent Mobile Robot," Tech. Report CMU-RI-TR-84-18, Carnegie-Mellon Univ., August 1984.

[5] R. Chatila, "Path planning and environment learning in a mobile robot system," *European Conf. on Artificial Intelligence*, Orsay, France, July 1982.

[6] M. M. Mano, "Digital Design," Prentice-Hall Inc., Engelwood Cliffs, NJ, 1984.

[7] E. Horowitz and S. Sahni, "Fundamentals of Computer Algorithms," Computer Science Press, Rockville, MD, 1984.

[8] D. E. Knuth, "The Art of Computer Programming: Fundamental Algorithms," Vol. 1, Addison-Wesley Publication, Reading, MA, 1968.