OPTIMAL PATH ALGORITHMS FOR AUTONOMOUS VEHICLES

by Johann Borenstein and Yoram Koren

ABSTRACT

Mobile Robots and Automatic Guided Vehicles (AGVs) must avoid collision with obstacles in their way. In order to do so effectively, prior knowledge about the obstacle locations should be available to the obstacle-avoidance algorithm. Proper representation of obstacles within a map stored in the computer can considerably reduce computation time of the algorithm, and is therefore dealt with detail in this paper.

Three path finding algorithms, based on different graph search methods, are explained and discussed. The two preferable algorithms are always able to determine an optimal path in terms of overall distance. Both algorithms have been tested on a prototype mobile robot and have been found to work correctly.

1. INTRODUCTION

In order to travel successfully from one location to another, a mobile robot or A.G.V. must have some knowledge about the obstacles in its surroundings. This knowledge may be obtained with the help of adequate sensors, such that any obstacle would be detected in time and overcome. However, this approach is problematic for the reasons given below:

1. Not all kinds of obstacles are detectable with a certain set of sensors.

2. Once an obstacle has been detected it is not necessarily bypassed by an optimal path.

3. Identification of the obstacle and search for a solution to bypass it will normally take much time, during which the robot is standing still.

For the above reasons, it is suggested to clearly differentiate between two kinds of obstacles:

1. Stationary Obstacles:
   Walls and heavy furniture, such as closets, beds, tables etc. fall into this category. During an initial set-up phase the coordinates of these stationary obstacles are fed into the robot's data base [1], which is also called "map" in this paper.

2. Non-Stationary Obstacles:
   These obstacles result from "disorder" in the environment and must be detected by sensors, in spite of the disadvantages mentioned above. In our application, detected obstacles are added to a temporary map, and consequently processed like stationary obstacles, until the temporary map is eventually erased. There is no definite rule for the time span during which the temporary map is maintained, since this largely depends on the type of non-stationary obstacles which are encountered in a specific application.

In order to add clarity to the explanations, keywords, having a special meaning in the context of this paper, have been printed in capital letters, and, in addition, have been underlined when being defined.

Before proceeding with a detailed explanation of the basic structure of the map and the description of the three suggested path planning methods, some requirements on the motion control must be discussed.

2. MOTION AND CONTROL STRATEGY FOR THE VEHICLE

In order to follow sharp corners as required by the optimal path algorithms, the vehicle must be capable of a purely rotational movement. As a matter of fact, it is desirable that the vehicle performs any motion between two locations A and B in the following manner: at first, the vehicle rotates about its center until it faces exactly the direction of B (pure rotation). Then the vehicle travels straight forward until it reaches B (pure translation), followed by another rotation about its center until it has the required final orientation (pure rotation, again).

Any type of motion between two given locations can be performed in this sequence. The peculiarity of this approach is that it actually uses only two distinct kinds of motion: Either motion in a straight line, where both wheels run at the same angular speed in the same direction, or rotation about the center, where both wheels run at the same angular speed but in opposite directions. This simplification offers numerous advantages:

1. Since in both cases the only task of the controller is to maintain equal angular velocities (measured in encoder pulses per time unit), a relatively simple and effective control system can be utilized. The control system for such a vehicle has been rigorously analyzed in a previous paper [2].

2. Both wheels are either simultaneously running or standing. Therefore, no case may occur where only one wheel is running while the other one is standing, which would inevitably cause severe slippage.

3. The vehicle's path is always predictable in high precision. This is especially important for the path planning algorithm described later, since this algorithm depends heavily on the vehicle's capability to follow a path very close to an obstacle.

4. The platform always travels through the shortest possible distance (straight line or rotation "on the spot").

5. This strategy yields the highest maneuvrability possible for an unidirectional vehicle.

Due to paragraphs 1 and 2 above, the vehicle is extremely accurate. Deviations from a straight line of only 1 - 2cm for 10m travel have experimentally been verified on our mobile robot [3].

Obviously, this motion strategy is less suitable for fast running vehicles, since they would have to stop completely before any change of direction. However, for slow vehicles, especially if operating in cluttered environments, this strategy is perfectly adequate. Also, for slow vehicles, an optimal path in terms of overall distance is almost always optimal in terms of overall time, too.

3. THE MAP

In the robot's memory, a map is maintained which comprises two parts: One is the map of the stationary obstacles, which is taught to the robot at the initial set-up of the system, and the other one is the temporary map which is constantly updated from sensor information. In terms of the following description of the situation, there is no difference between the two maps.

Fig. 1 shows the map of a room with obstacles. The lines in this map represent the boundaries of the obstacles with the addition of half the robot's width and 10 centimeters as a safety factor. This representation is known by the name "configuration space approach" [4,5,6,7] and allows the robot to be thought of as "shrunken" to a point, which may move on the boundaries (including corners). Intersections between boundaries will be called CORNERS and are of special importance to the following sections.

The data for the map is actually stored in form of a table, such as the one shown in Fig. 2 (which is the table representation of the map in Fig. 1), where the coordinates of the CORNERS are noted in columns X and Y. The numbers in column LNKA denote "connectivity" to a neighbouring CORNER, thus defining a border line. Column LNKB in Fig. 2 will be explained later.

It is easily seen, that if a certain path (e.g.: between starting location S and final goal F in Fig. 1) is obstructed by obstacles, then the optimal path, in terms of path length, will lead through certain CORNERS [8] (here, for example, CORNERS 3 and 11) which in this case will be called INTERMEDIATE GOALS. All CORNERS are potential INTERMEDIATE GOALS and only CORNERS may be used as such.

This approach is objected to by some researchers [9,10], claiming that a robotic vehicle approaching an obstacle too closely might run into the obstacle because of the inaccuracies associated with mobile robots. One different approach, where the vehicle's path leads through "adits", artificially inserted via points which are located midway between corners of adjacent obstacles, has been suggested by Crowley [1]. This way, a maximal safety distance between the vehicle and the obstacles is maintained, but the generated path is not optimal in terms of path length. In our case, however, because of the accurate control system and motion strategy (as explained before), travel along obstacle borders is possible without fear of hitting the obstacle.

During the initial set-up phase it is also possible to define FORBIDDEN CORNERS which are not to be used as INTERMEDIATE GOALS. This is neccessary, for example, in order to indicate that a certain CORNER is close to a wall and may therefore not be used for bypassing the obstacle. FORBIDDEN CORNERS are represented by a flag (see column FLAG in Fig. 2). However, in our example there are no FORBIDDEN CORNERS at all and all flags are reset to 0.

In order to determine if a path from one INTERMEDIATE GOAL to another is obstructed, the computer algorithm checks if the required path crosses any one of the boundary lines in the map. This operation is called CROSS-CHECK and its result is positive if there was at least one boundary line which is crossed by the required path. Any rectangular obstacle, such as 1-2-3-4 in Fig. 1, adds 4 boundary lines to the map which must be checked whenever a CROSS-CHECK is performed, thereby increasing the overall computation time.

One problem arises when a CROSS-CHECK, for example for the path from point 1 to point 3, is performed. The (obviously misleading) result would be negative, indicating an unobstructed path.

An elegant solution to overcome the above mentioned problems is the use of diagonal lines to define the obstacles. Fig. 3 shows the same obstacles as Fig. 1, yet now two lines are sufficient to represent a rectangular obstacle, resulting in a considerable decrease in computation time for a CROSS-CHECK. Also, this representation reveals immediately that there is no path from point 1 to point 3 (according to the above example), since now line 2-4 would be crossed.

However, there is one limitation to the diagonal line representation: all obstacles must be represented by convex shapes, such that any two corners of the body may be connected by a straight line without cutting the perimeter of the body. Actually, this limitation is not a severe one at all, since any body may be composed of simpler convex bodies, using FORBIDDEN CORNERS to mask out concave edges. The use of rectangular sided shapes as the basic form is most advantageous since only two lines are needed for its representation. The column headed "LNKB" in Fig. 2 holds the necessary information to define the diagonal lines. For example, the LNKB of point 1 is point 3 as those two points define a diagonal. The LNKB of point 3, is point 1, of course, but this diagonal is already defined and therefore Ø is given as the LNKB of 3. The representation by diagonals also suits non-rectangular obstacles where the entry in the LNKB column is defined as the "connection of the connection" (LNKA of LNKA) of the corner in question.

4. OPTIMAL PATH PLANNING WITH GRAPH SEARCH STRATEGIES

After defining the basic structure of the map, three algorithms for optimal path planning will be discussed. All three algorithms are graph search strategies known from Artificial Intelligence which were adapted and improved for this application [11,12].

In the search-graph to be used by the algorithms, NODES in the search-graph represent CORNERS in the map, and arcs, connecting certain NODES, represent unobstructed direct connections between CORNERS. The only applicable production rule is the CROSS-CHECK which determines whether one NODE is directly

accessible or not. Because of their close relationship, the terms "CORNERS" and "NODES" may be used interchangeably in this text. Practically, a NODE is created by saving some space in memory for a NODE-VECTOR , which holds relevant information about the NODE in pre-defined fields of the vector. NODES are assigned to CORNERS by writing the name of the CORNER into the "C"-field of the NODE-VECTOR. CORNERS are named by numbers, according to their number in the map (see Fig. 1). NODES always have the same name as the CORNERS they are assigned to. NODE 0 is assigned to the initial location S, which is sometimes also refered to as CORNER 0, and the last NODE in the list of NODES is assigned to the goal F, in our example also referred to as CORNER 13 or NODE 13.

Fig. 4 shows an example which will be dealt with by the three algorithms. The special shape of the bodies in this example has been chosen on purpose in order to highlight the features of each one of the algorithms.

I) Heuristic Depth First Search With Hill Climbing

This method always selects the NODE deepest down in the search tree for expansion. In the following, this procedure is described in detail, and major subroutines, indicated by bold-type headings in the respective flow chart (Fig. 5), are referred to by their name in square brackets hereafter.

For our example it is assumed that the search starts at point S and the goal is point F [START, SETPATH]. First, the algorithm checks whether a direct path to F is obstructed, using the CROSS-CHECK. Since the result is positive, the names of all CORNERS that terminate those lines which are cut by a straight line between S and F, are compiled into a list which belongs to the current NODE. These CORNERS are called CUT-CORNERS (CCs). At this stage, the current NODE is NODE 0, which is assigned to point S, and the CCs of NODE 0 would be 1,2,3,4,9,10,11 and 12 [CHECKPATH]. Only the CORNERS in this list are candidates for the optimal path to lead through, therefore the following steps will be restricted to these CORNERS only (similar heuristics have been suggested in [13]).

Out of the list of CCs of NODE 0, all those CCs are selected which may be reached from NODE 0 directly and are not marked USED or FORBIDDEN. Only the selected CCs remain in the list, all others are deleted [FINDNODE]. In the example, only CORNERS 3 and 4 remain in the list, and the algorithm assigns to them new NODES, named NODE 3 and NODE 4 respectively [SETNODE]. Next the COST for each newly created NODE is calculated. Here, the COST is simply the sum of the distance between the current NODE and the new NODE's CORNER, plus the distance between the new NODE's CORNER and the goal. This COST function gives preference to CORNERS close to the straight line connection between the current NODE's CORNER and the goal, but is misleading for certain obstacles, as will be shown later. Out of the newly created NODES the one with the least COST is choosen [CHOOSENODE] to be the next current NODE. Here, the choice is

NODE 3 which now becomes current NODE and the process is repeated, as shown in Fig. 5, until CORNER F is reached. In the example, the selected path is S-3-2-10-9-12-F. Fig. 6 shows the status of each NODE after the goal has been reached. Fig. 7a depicts the resulting search tree and the computed path is shown in Fig. 7b.

Clearly, this is not the optimal path. The algorithm was mislead by the COST function and the oddly shaped obstacle 9-10-11-12: When, for example, examining the path from CORNER 2, the COST function suggested CORNER 10 as the next INTERMEDIATE GOAL, since it is closest to line 2-F. Obviously, this is an unfortunate choice, because of the irregular shape of that obstacle. The same "mistake" was made at NODE 10, when CORNER 9 was suggested for the same reason.

A different search tree results, if, for example, CORNERS 9 and 11 are declared FORBIDDEN. This corresponds to no real situation, but for the sake of the explanation this case may be assumed. Now, the solution develops as follows: From point S through CORNER 3 to CORNER 2. From there CORNER 10 will be choosen because of its lowest COST. Presently, the algorithm is "stuck", because there is no other CORNER to proceed to. Therefore, NODE 10 is flagged "FAIL" and the algorithm steps backward to NODE 2 [NEWTRY]. In case there wasn't a way from CORNER 2 either, the algorithm would backstep once more, to the SOURCE-NODE of NODE 2 (that is: NODE 3) and would try again, until a valid way would be found. However, there is a way to proceed from CORNER 2: to CORNER 12 and from there to F as required. This case is visualized in Fig. 8a and Fig. 8b.

Discussion :

This method is relatively fast, in terms of computing time, since the solution is advanced quickly into the depth of the search-tree. On the other hand, an optimal solution is not guarranted since not all possible paths are checked.

"Hill climbing" (choosing a NODE because of its lower COST) normally advances the solution faster, but there are some extreme cases (such as obstacle 9-10-11-12) where the minimal COST criterium does not yield a local optimal path. For symmetrical obstacles, however, the minimal COST criterium normally does yield an optimal local path.

The main advantage of this method, the heuristic search (which means - in this case - limitation of candidates for INTERMEDIATE GOALS to CROSS-CORNERS only), may allow a considerable reduction in computing time, especially when there are many obstacles on the map. On the other hand, there are some special cases (e.g.: when concave obstacles are involved) where a solution might not be found at all because of the heuristic search. Therefore, when no solution is found at all, the algorithm cancels the heuristic search and starts a new search from the beginning. In this case a solution can be found but computing time increases considerably.

II) Branch and Bound Strategy with Minimum Estimator

The fundamental difference between this method and the preceeding one lies in the fact that here an optimal solution is guaranteed. Unlike the depth-first method where the solution was advanced downwards until it reached the goal or was forced to backstep, here an optimal solution is achieved by advancing the search always from the NODE featuring lowest COST. For this purpose, the accumulated COST is written into each newly created NODE as the solution advances. According to this algorithm the destination NODE will eventually become the current NODE featuring the lowest COST and therefore offering an optimal solution.

Calculation time for this algorithm may be reduced by the use of an estimator for the remaining distance to the goal. In this case, the COST for a new NODE is calculated as the accumulated distance plus the estimate of the distance to the goal. Thus, CORNERS close to the destination are preferred, whereas CORNERS far away might not be selected at all. A good estimate of distance to the goal is the direct distance to the goal, since this estimate always constitutes a lower bound for the actual distance [11]. The branch and bound strategy, as described here, is also known as the A * algorithm [12].

Fig. 9a shows the search-tree for the example in Fig. 4. The number in brackets next to each NODE is the accumulated COST of the NODE, including the distance estimation. The arrows indicate the direction of the advance, and a crossed out arrow shows that a "cheaper" path to the NODE has been found, the COST of which replacing the former COST. Numbers next to arrows indicate the stage at which this path was choosen. The computed (optimal) path is shown in Fig. 9b.

Discussion

Even though the depth-first method normally computes faster, the branch and bound strategy is preferable, since a solution is found in any case, regardless of the type and constellation of obstacles involved. Also, the solution is always optimal in terms of path length.

III) Precomputed Search-Graph With Dynamic Programming

The method described in this section is faster than the two previously described methods, but effective only if the map is mostly static, that is, no unexpected obstacles are added to the map during a run. This is a reasonable requirement for many applications and even if changes to the map do occur, only one computational run (after the change occured) would be required in order to adapt to the new situation.

For a static map, some of the most time-consuming calculations may be performed off-line, and only once, immediately after the map has been defined or updated. The results of this computations (such as distances and possible

connections between CORNERS) are written into NODE-VECTORS in advance (and not during a computational run, once the need arises, as with the other methods). With possible connections compiled into each NODE, a new kind of map, the "MAP of CONNECTIONS" (MC) stands at the algorithm's disposal. A graphical representation of this map is given in Fig. 10., showing the possible connections between CORNERS of the example in Fig. 4. The tabulated form of the MC is shown in Fig. 11, where column "C" is the NODE number (identical with the CORNER number the NODE is assigned to), column "N" is the number of possible connections of this CORNER and column "CONNECTIONS" is the list of CORNERS that CORNER is connected to. The other columns in Fig. 11 will be explained later.

The following computations remain to be done on-line, once a new goal has been defined:

1. Find all CORNERS that connect to the destination and add NODE 13 (assigned to point F) to their list of CONNECTIONS.

2. Find all CORNERS that connect to the present location S and write their names into column "CONNECTIONS" of NODE 0 (which is assigned to point S).

3. Find the optimal path with the help of the dynamic programming algorithm which is explained in the next section.

Dynamic Programming

Dynamic programming is a well-known method used to calculate a minimum cost path between NODES of an interconnected network of NODES, with a given start- and end-NODE and known cost between all NODES. Clearly, the MAP of CONNECTIONS described above fits this requirements. Therefore, dynamic programming is used in order to compute the optimal path in this application.

The algorithm works as follows: In the example of Fig. 4 and Fig. 10, there is a given starting point S and a goal F, and the COST between the CORNERS, in terms of distance, is known. The algorithm proceeds from point S through all possible CONNECTIONS towards point F, while writing into each NODE-VECTOR the accumulated COST for reaching the NODE. If there are several paths leading to a certain NODE, then only the lowest COST is written into the NODE-VECTOR, together with the name of the PARENT NODE (also called "SOURCE", in this text) through which the least expensive path led to the current NODE. Thus, only one

COST and one SOURCE is written into each NODE-VECTOR, especially into the one assigned to point F. The COST represents the minimum COST, and by backtracking through the SOURCE of NODE 13 and the SOURCE of that SOURCE and so on, until NODE 0 is reached, the optimal path is recorded.

The realization of the dynamic programming algorithm in our application is somewhat more complicated, since the order of the NODES is not known in advance. This is so since starting point S and destination F change from run to run. Fig. 12 shows the flow chart for this realization.

Upon reception of a motion command to a new goal, the algorithm checks if the goal may be reached directly [?CUT]. If not, all NODE-VECTORS are initialized in a two step procedure: Firstly, the list of CONNECTIONS for each NODE is reset to the basic status (the one depicted in Fig. 10). Then the CORNERS that may be reached directly from the starting point S are written into the NODE-VECTOR of NODE 0 [RE-INITIALIZE]. Subroutine [START] finds all those NODES that connect to the goal and writes F's NODE number (here: 13) into their CONNECTION list.

Now, starting from NODE 0, the main loop runs once for each current NODE: For all items in the CONNECTIONS list of the current NODE (which are called "LISTED NODES"), the accumulated COST for reaching the LISTED NODE is computed and compared with the COST previously written into the NODE-VECTOR of the LISTED NODE. If the new COST is less, then it is written into the NODE-VECTOR, together with the name of the current NODE, which is the SOURCE-NODE through which the lower COST was achieved (see columns S, for SOURCE, and COST in Fig. 11). If this is the first time COST is computed for a certain LISTED NODE, then the new COST will definitely be less, because all COSTS had been initialized to some very high value during RE-INITIALIZE. Also, the name of the NODE who's COST has just been updated is written into a FIFO-stack (FIFO= First In, First Out), unless it already appears there. This guarantees that each NODE becomes exactly one time current NODE [SETNODE]. Thereafter, the next item from the FIFO-stack is made current NODE and the loop is repeated. When the FIFO-stack is empty, all NODES have been dealt with and the main program is finished. Fig. 11 shows the NODE-VECTORS of all NODES at that stage. Now the algorithm backtraces through all SOURCE entries until NODE 0 and writes them into the CONNECTIONS field of NODE 13 for further processing.

Discussion

This method, based on a modified dynamic programming algorithm, is less general in its application than the two preceding methods. However, for suitable applications this method will compute an optimal path much faster than the two other methods.

5. IMPLEMENTATION

Our mobile robot is controlled by a hirarchical computer system, which comprises three distinct levels. On the lowest level, there are three Z80A-based microcomputers which operate and read various sensors as well as control the motion of the mobile robot). These microcomputers are linked to each other and to the medium level by a network. At the medium level, there is one Z80A-based microcomputer, which coordinates the low-level computers and relays data and commands between them and the highest level.

At the highest level, and not onboard the mobile robot, there is an IBM PC, connected to the medium level by a RS232 serial link. The optimal path algorithm, together with several other high-level tasks, is implemented on this computer. The programming language for the computers on all three levels is FORTH, which suits this application because of its compactness (important at the lower levels), speed, trasportability and extendability.

Presently, the branch and bound strategy is implemented in our mobile robot, since it is the most generally applicable one and our robot is to serve research purposes with varying aspects. However, the first method was tested (as an intermediate step, as it is the easiest for realization) in simulations, whilst the dynamic programming based strategy was actually implemented for some time during the development of our robot. Only after introduction of ultrasonic range finders into our robot (which are used to update the stationary map with information on unexpected obstacles), it became necessary to use a more general algorithm: the branch and bound strategy.

Using the branch and bound algorithm, computation time of the optimal path for the example in Fig. 4 is ca. 2 seconds. For more cluttered maps, with more than 20 or 30 CORNERS, computation time increases to several tens of seconds. This is largely due to the high number of CROSS-CHECKS that have to be performed. It is therefore suggested to recode the (relatively short and simple) subroutine "CROSS-CHECK" in machine-language. It is estimated, that this, or the use of a (commercially available) more sophisticated FORTH compiler, could cut computation time by a factor of 4 to 6.

CONCLUSIONS

Three path planning methods for autonomous vehicles have been explained and discussed, but only the latter two methods guarantee an optimal path. Compared to the branch and bound strategy, the dynamic programming method computes faster for a mostly stationary map with few obstacles. One might argue, though, that in case of a static map, the branch and bound strategy could also benefit from a precomputed MAP of CONNECTIONS, thereby saving computation time. Since the branch and bound strategy does not consider all possible paths, it

should be more effective than the dynamic programming method. Even though the branch and bound strategy needs two distance calculations for each NODE, one for the accumulated distance and one for the distance estimation to the goal, whereas the dynamic programming method needs only one, it seems that the branch and bound strategy computes generally faster. In any case, when many obstacles and a frequently changing map are involved, the branch and bound method should be applied.

Even though the algorithms compared in this paper only solve for the most basic case of an optimal path in a plane and for a circular robot, they can be applied for more general cases (polyhedral robots, three dimensional, with rotation) without any changes to the graph seach algorithm used. What must be changed, however, is the spacial representation of the obstacles. Interesting methods for this are given in [8].

REFERENCES

(1) Crowley, J. L.: "Position Estimation for an Intelligent Mobile Robot". 1983 Annual Research Review , Carnegie-Mellon University, Robotics Institute, 1984, pp. 41-50.

(2) Borenstein, J. and Koren, Y.: A Mobile Platform For Nursing Robots". IEEE Transactions on Industrial Electronics , May 1985, pp.158-165.

(3) Borenstein, J. and Koren, Y.: "Motion Control Analysis of A Mobile Robot". Submitted for publication in: ASME Journal of Dynamics, Measurement and Control .

(4) Brooks, R. A.: "Solving the Find-Path Problem by Good Representation of Free Space". Proceedings of the National Conference of Artificial Intelligence , AAAI-82, August 1982, pp. 381-386.

(5) Crowley, J. L.:"Navigation for an Intelligent Mobile Robot" Carnegie-Mellon University, The Robotics Institute , Technical Report, August 1984.

(6) Lozano-Perez, T.: "Automatic Planning of Manipulator Transfer Movements". IEEE Transactions on Systems, Man, and Cybernetics , Vol. SMC-11, No. 10, October 1981, pp. 681-698.

(7) Jorgensen, C., Hamel, W. and Weisbin. C.: "Autonomous Robot Navigation". BYTE , Jan. 1986, pp. 223-235.

(8) Lozano-Perez, T. and Wesley, A.: "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles". Communications of the ACM , October 1979, pp. 560-570.

(9) Crowley, J. L.: "Dynamic World Modelling for an Intelligent Mobile Robot Using a Rotating Ultra-Sonic Ranging Device". Technical Report of the Laboratory for Household Robotics The Robotics Institute, Carnegie-Mellon University, October 1984.

(10) Thorpe, C. F.: "Path Relaxation: Path Planning for a Mobile Robot". Carnegie-Mellon University, The Robotics Institute Technical Report, April 1984.

(11) Winston, P. H.: " Artificial Intelligence ", Addison-Wesley Publishing Company, Second Printing, 1979, pp. 87-112.

(12) Nielsson, N. J.: " Principles of Artificial Intelligence ". Tioga Publishing Company, Palo Alto, 1980, pp. 61-97.

(13) Keirsey, D.M. et al.: "Algorithm of Navigation for a Mobile Robot". International Conference on Robotics I , Atlanta, March 13-15, 1984, pp. 574-583.

List of Figures

Fig. 1 : Example of a map of stationary obstacles.

Fig. 2 : Table representation of the map in Fig. 1.

Fig. 3 : Representation of obstacles by diagonals.

Fig. 4 : Example map for comparison of the three algorithms.

Fig. 5 : Flow chart for the depth-first search with hill climbing.

Fig. 6 : Status of the NODES after completing the search-tree (depth-first search).

Fig. 7 : Depth-first search:
a) Completed search-tree b) Computed path

Fig. 8 : Depth-first search with FORBIDDEN CORNERS 11,13:
a) Completed search-tree b) Computed path

Fig. 9 : Branch and bound strategy:
a) Completed search-tree b) Computed path

Fig. 10 : MAP of CONNECTIONS, computed off-line.

Fig. 11 : Status of the NODES after completing the run (dynamic programming).

Fig. 12 : Flow chart for the modified dynamic programming algorithm.

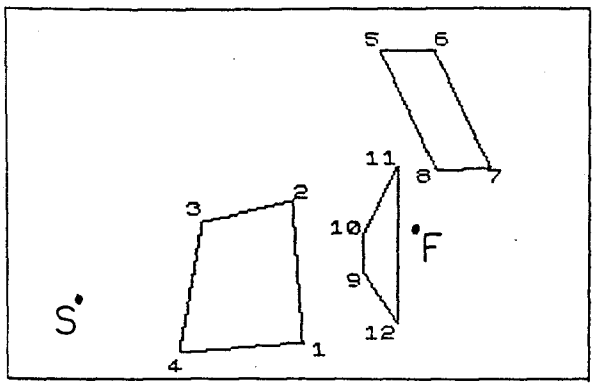


Fig. 1 : Example of a map of stationary obstacles.

POINT	X	Y	LNKA	LNKB	FLAGS
0	0	0	0	0	0
1	125	15	4	3	0
2	120	80	1	4	0
3	80	70	2	0	0
4	70	10	3	0	0
5	160	150	8	7	0
6	185	150	5	8	0
7	210	96	6	0	0
8	186	94	7	0	0
9	152	48	12	11	0
10	152	64	9	12	0
11	168	96	10	0	0
12	168	24	11	0	0

Fig. 2 : Table representation of the map in Fig. 1.

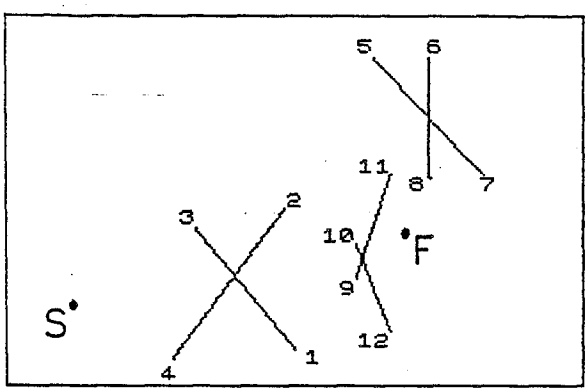


Fig. 3 : Representation of obstacles by diagonals.

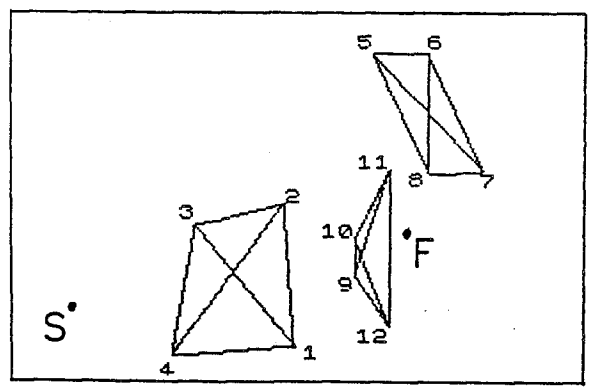


Fig. 4 : Example map for comparison of the three algorithms.

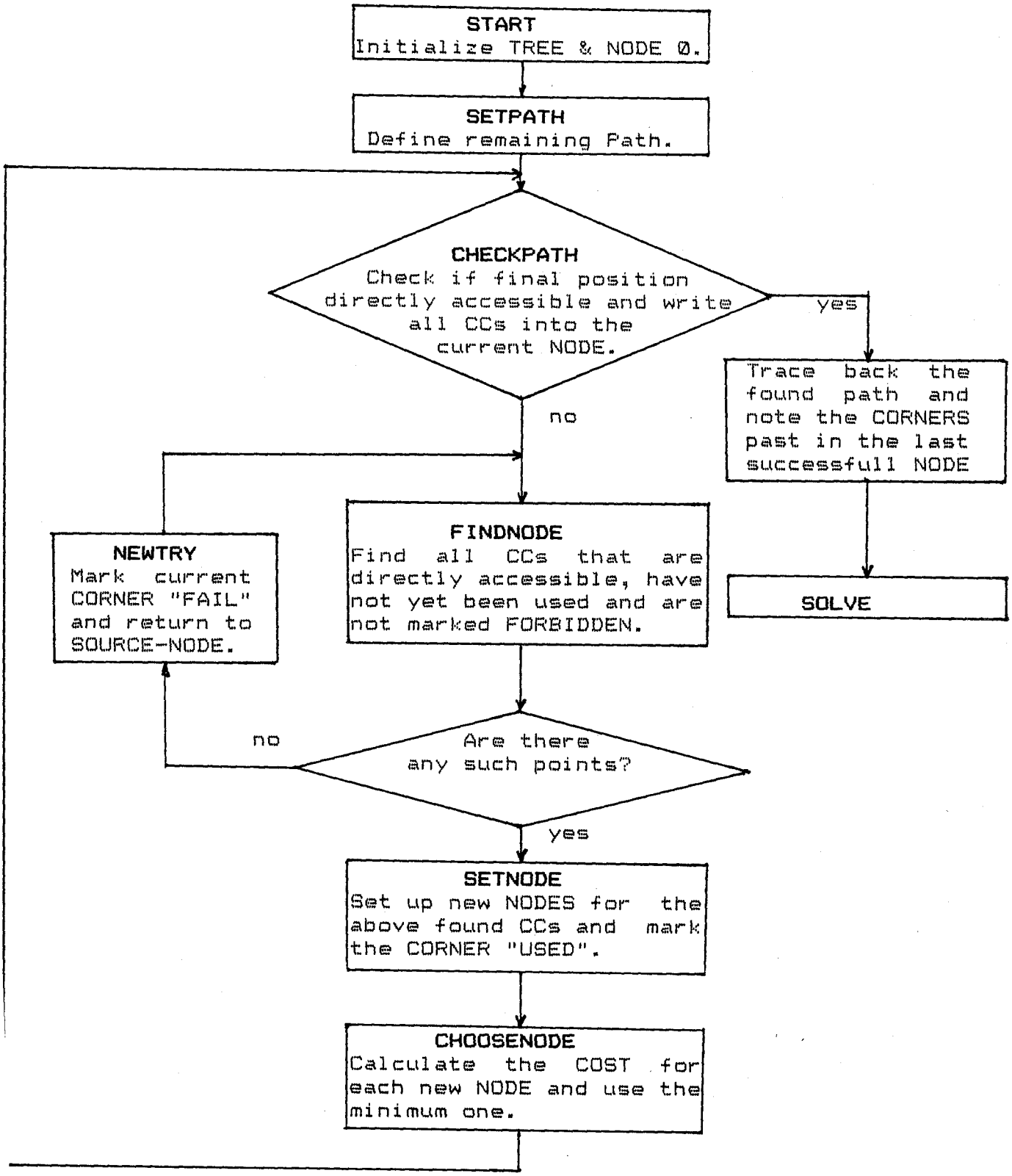
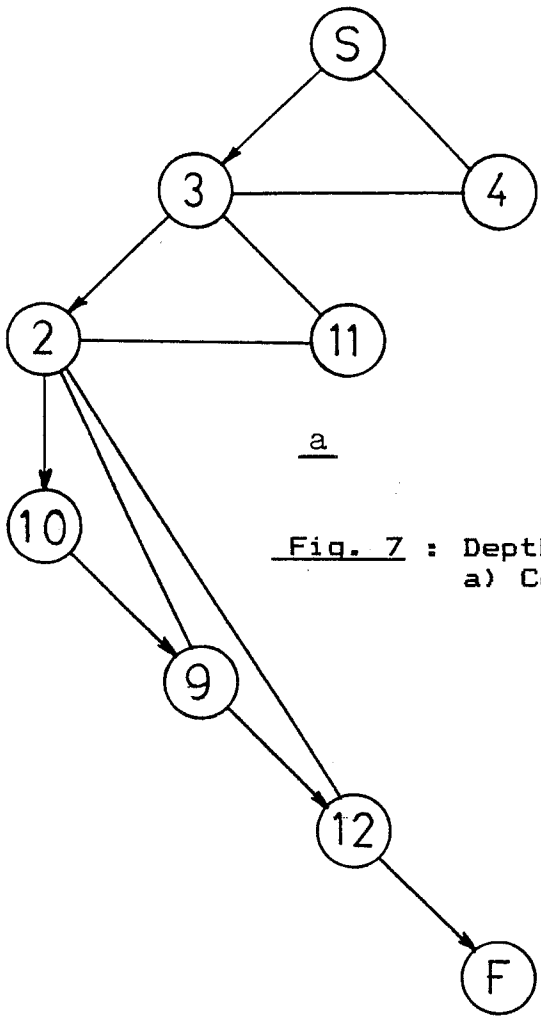


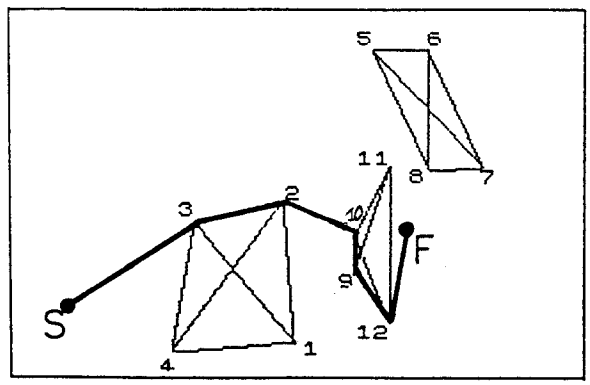
Fig. 5: Flow chart for the depth-first search with hill climbing.

C	Flags	Source	COST	N	CONNECTIONS
0	0			2	4 3
1	0	0	0	0	0
2	1	3	98	4	11 10 12 9
3	1	0	163	3	2 11 4
4	0	3	180	0	
5	0	0	0	0	
6	0	0	0	0	
7	0	0	0	0	
8	0	0	0	0	
9	1	10	46	1	12
10	1	2	59	2	11 9
11	0	10	65	0	0
12	1	9	71	6	12 9 10 2 3 0
13	0	12			

Fig. 6: Status of the NODES after completing the search-tree (depth-first methode).

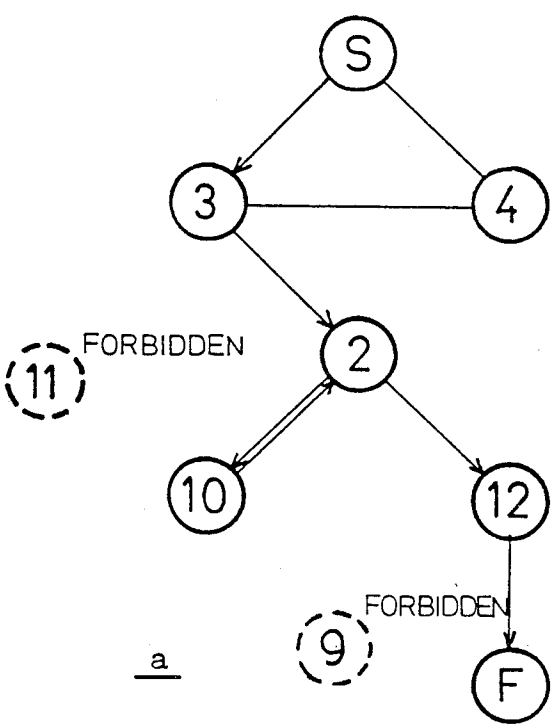


a

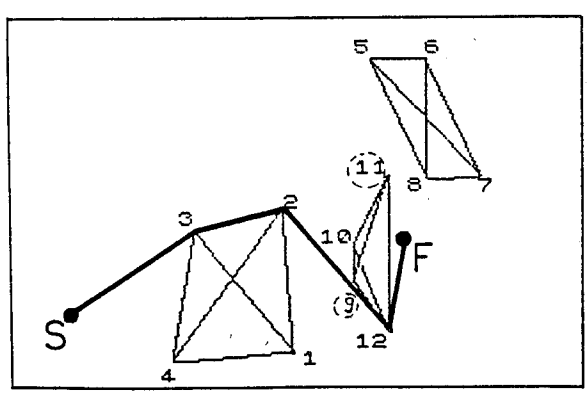


b

Fig. 7 : Depth-first search:
a) Completed search-tree b) Computed path

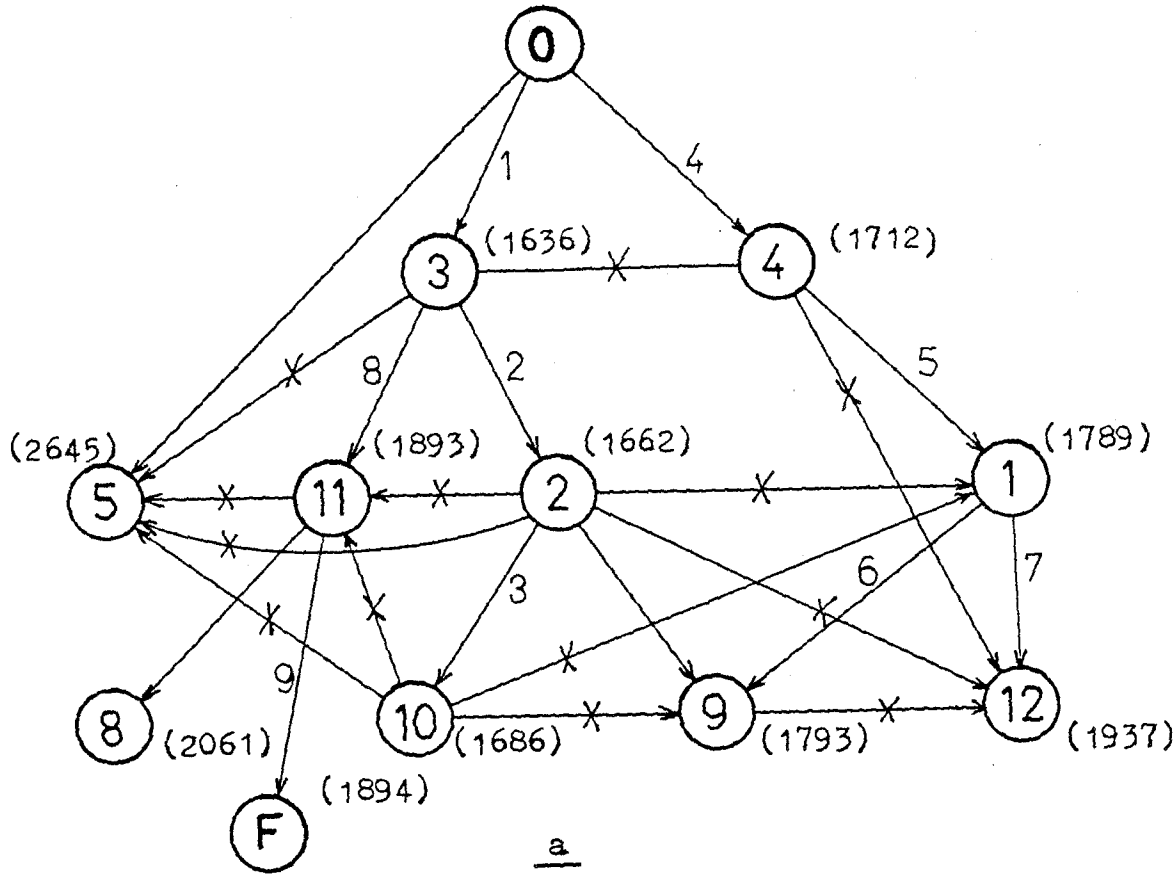


a

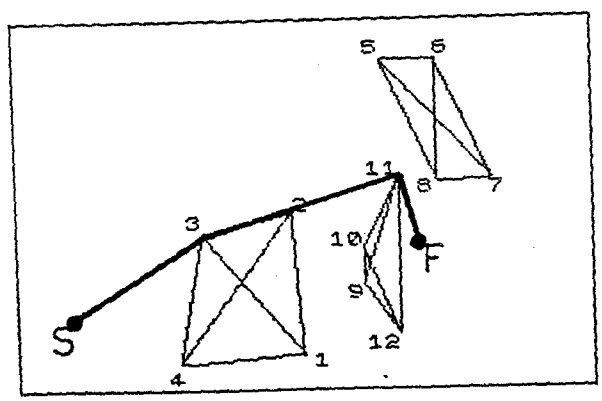


b

Fig. 8 : Depth-first search with FORBIDDEN CORNERS 11,13:
a) Completed search-tree b) Computed path



a



b

Fig. 9 : Branch and bound strategy:
a) Completed search-tree b) Computed path

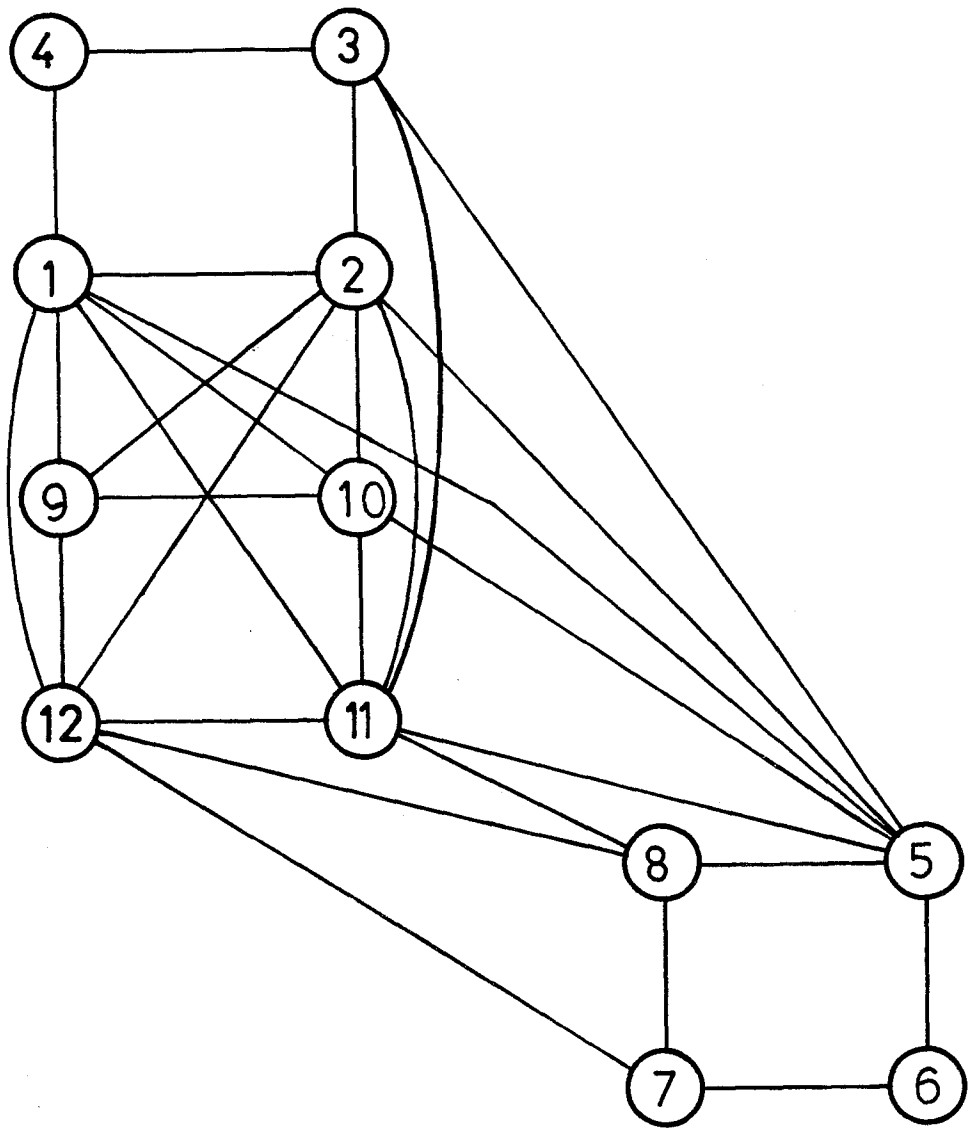


Fig. 10 : MAP of CONNECTIONS, computed off-line.

C	F	S	COST	N	CONNECTIONS
0	0	49	0	3	3 4 5
1	0	4	1061	7	2 4 5 9 10 11 12
2	0	3	1088	7	1 3 5 9 10 11 12
3	0	0	676	4	2 4 5 11
4	0	0	509	2	1 3
5	0	0	1800	8	1 2 3 6 8 10 11 13
6	0	5	2050	2	5 7
7	0	8	2014	4	6 8 12 13
8	0	11	1774	5	5 7 11 12 13
9	0	1	1487	4	1 2 10 12
10	0	2	1445	5	1 2 5 9 11
11	0	3	1593	8	1 2 3 5 8 10 12 13
12	0	1	1500	7	1 2 7 8 9 11 13
13	0	11	1893	4	13 11 3 0

Fig. 11 : Status of the NODES after completing the run (dynamic programming).

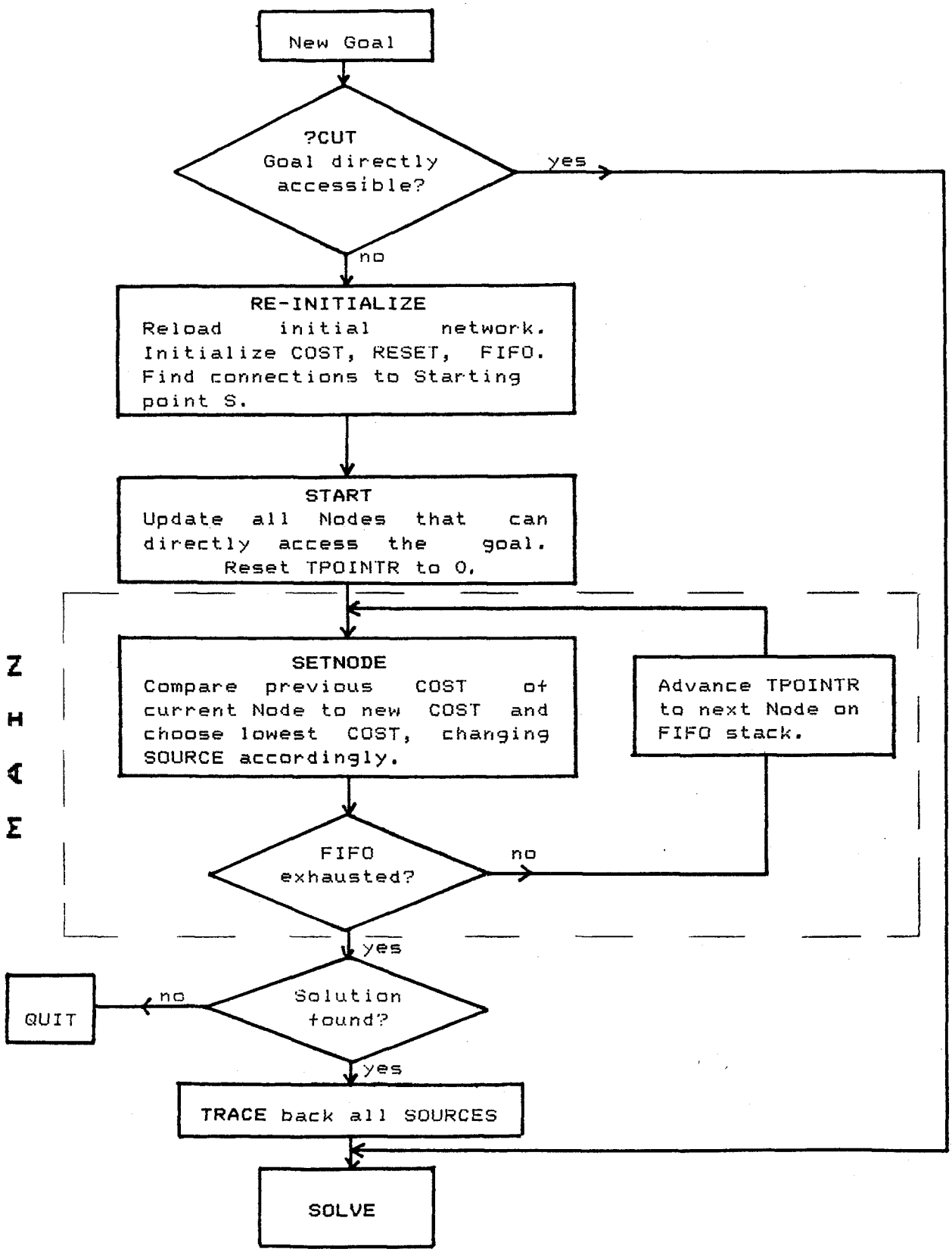


Fig. 12 : Flow chart for the modified dynamic programming algorithm.