

AD-A215 115

NAVAL POSTGRADUATE SCHOOL

Monterey, California

②



THESIS

DTIC
ELECTE
DEC 12 1989
S E D

A COMPUTER SIMULATION STUDY
OF A SENSOR-BASED HEURISTIC NAVIGATION
FOR THREE-DIMENSIONAL ROUGH TERRAIN
WITH OBSTACLES

by

CPT Do Kyeong Ok

June 1989

Thesis Advisor:

Se-Hung Kwak

Approved for public release; distribution is unlimited

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified		1b Restrictive Markings	
2a Security Classification Authority		3 Distribution Availability of Report Approved for public release; distribution is unlimited.	
2b Declassification/Downgrading Schedule			
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)	
6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol (If Applicable) 52	7a Name of Monitoring Organization Naval Postgraduate School	
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000		7b Address (city, state, and ZIP code) Monterey, CA 93943-5000	
8a Name of Funding/Sponsoring Organization	8b Office Symbol (If Applicable)	9 Procurement Instrument Identification Number	
8c Address (city, state, and ZIP code)		10 Source of Funding Numbers	
		Program Element Number	Project No Task No Work Unit Accession No
11 Title (Include Security Classification) A COMPUTER SIMULATION STUDY OF A SENSOR-BASED HEURISTIC NAVIGATION FOR THREE-DIMENSIONAL ROUGH TERRAIN WITH OBSTACLES			
12 Personal Author(s) Ok, Do Kyeong			
13a Type of Report Master's Thesis	13b Time Covered From To	14 Date of Report (year, month, day) June 1989	15 Page Count 123
16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17 Cosati Codes		18 Subject Terms (continue on reverse if necessary and identify by block number)	
Field	Group Subgroup	Sensor-Based Path Plan, Heuristic Navigation, Obstacle Avoidance, Autonomous Vehicle	
19 Abstract (continue on reverse if necessary and identify by block number) A search strategy for autonomous vehicle navigation over three-dimensional digitized terrain containing obstacles is presented and studied in this thesis. The vehicle possesses no a priori information about terrain. Using only information obtained through a sensor which has a limited sensing range, the vehicle navigates a goal utilizing heuristics adopted from human behavior. Simulation results produce a near-optimal path solution in a very short time. Simulation results also prove that this strategy is suitable for real-time navigation under dynamically changing or unknown environments.			
20 Distribution/Availability of Abstract ☒ unclassified/unlimited ☐ same as report ☐ DTIC users		21 Abstract Security Classification Unclassified	
22a Name of Responsible Individual Prof. Se-Hung Kwak		22b Telephone (Include Area code) (408) 646-2168	22c Office Symbol Code 52Kw

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted

security classification of this page

All other editions are obsolete

Unclassified

Approved for public release; distribution is unlimited.

**A COMPUTER SIMULATION STUDY OF A SENSOR-BASED
HEURISTIC NAVIGATION FOR THREE-DIMENSIONAL ROUGH
TERRAIN WITH OBSTACLES**

by

Do Kyeong Ok
Captain, Korean Army
B.S., Korea Military Academy, 1982

Submitted in partial fulfillment of the requirements for
the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

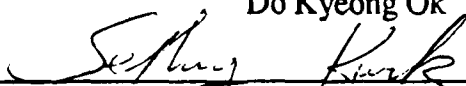
NAVAL POSTGRADUATE SCHOOL
June 1989

Author:

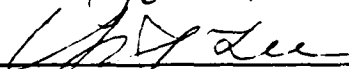


Do Kyeong Ok

Approved by:



Se-Hung Kwak, Thesis Advisor



Yuh-jeng Lee, Second Reader



Robert B. McGhee, Chairman, Department of Computer Science



Kneale T. Marshall, Dean of Information and Policy Sciences

ABSTRACT

A search strategy for autonomous vehicle navigation over three-dimensional digitized terrain containing obstacles is presented and studied in this thesis. The vehicle possesses no a priori information about terrain. Using only information obtained through a sensor which has a limited sensing range, the vehicle navigates a goal utilizing heuristics adopted from human behavior. Simulation results produce a near-optimal path solution in a very short time. Simulation results also prove that this strategy is suitable for real-time navigation under dynamically changing or unknown environments.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification _____	
By _____	
Distribution/ _____	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	BACKGROUND AND BRIEF PROBLEM STATEMENT	1
B.	THESIS ORGANIZATION	3
II.	SURVEY OF PREVIOUS WORK	4
A.	INTRODUCTION	5
B.	PATH PLANNING ALGORITHMS	5
1.	A HIERARCHICAL ORTHOGONAL SPACE APPROACH TO THREE-DIMENSIONAL PATHPLANNING	5
2.	NAVIGATIONS FOR AN INTELLIGENT MOBILE ROBOT	5
3.	ROBOT NAVIGATION IN UNKNOWN TERRAINS USING LEARNED VISIBILITY GRAPHS	6
4.	LEARNED NAVIGATION PATHS FOR A ROBOT IN UNEXPLORED TERRAIN	7
5.	AUTOMATIC PATH PLANNING FOR A MOBILE ROBOT USING A MIXED REPRESENTATION OF FREE SPACE	7
C.	SUMMARY	8
III.	DETAILED PROBLEM STATEMENT	9
A.	INTRODUCTION	9
B.	AUTONOMOUS VEHICLE MODEL	9

C.	TERRAIN MODEL	10
D.	OBSTACLE MODEL	11
E.	SENSOR MODEL	12
F.	SIMULATION FACILITIES	12
1.	LANGUAGE	12
2.	SYMBOLICS LISP MACHINE	12
G.	SUMMARY	13
IV.	NAVIGATION BY PATH PLAN	14
A.	INTRODUCTION	14
B.	CONVENTIONS	14
C.	HEURISTICS	15
D.	PATH PLAN STRATEGY	16
E.	MAIN PATH PLAN	19
1.	Local Cost Function	21
a.	Transitional Cost	21
b.	Rotational Cost	26
2.	Estimation Function	29
3.	Evaluation Function	35
F.	DETERMINATION OF CONTROL MODE	36
G.	DETOUR PATH PLAN	40
1.	Obstacle marker	40
2.	Obstacle Evaluation Function	47
H.	SUMMARY	53
V.	SIMULATION AND EVALUATION	57

A.	INTRODUCTION	57
B.	SIMULATION RESULTS	57
1.	SIMULATION RESULTS WITHOUT EXPLICIT OBSTACLES	58
2.	SIMULATION RESULTS WITH EXPLICIT OBSTACLES	58
C.	COMPARISON WITH A* SEARCH	66
1.	COST OF PATH	67
2.	MAXIMUM NUMBER OF OPEN NODES	67
3.	TIME TO FIND PATH	73
D.	SUMMARY	73
VI.	SUMMARY AND CONCLUSIONS	75
A.	RESEARCH CONTRIBUTIONS	75
B.	RESEARCH EXTENSIONS	75
	APPENDIX – PROGRAM LIST	77
	LIST OF REFERENCES	112
	INITIAL DISTRIBUTION LIST	114

ACKNOWLEDGMENT

Thanks God!. I thought I could not write a thesis because I was a dummy. God encouraged me unceasingly and Se-Hung Kwak, my thesis advisor, helped and taught me with my thesis preparation. James J. Zanolli, my friend, corrected my English.

I greatly appreciate their assistance. Also, I give a lovely thanks to my wife, Hee-Jung.

I. INTRODUCTION

A. BACKGROUND AND BRIEF PROBLEM STATEMENT

Path planning for an autonomous vehicle is, conceptually, classified into two categories. One is to perform path planning without a priori terrain information. The other is to perform path planning with a priori terrain information. The first type of path planning requires single or multiple sensing devices, such as a vision sensor, an ultra-sonic sensor, or a contact sensor. The latter type does not require sensing device.

Path planning with a priori knowledge of the terrain is easily adopted for long range path planning based on a map which contains time-invariant information. Because information about the terrain is available prior to the search, well established graphic search strategy such as the A^* search strategy that minimizes the overall cost can be used to find the optimal path.

However, path planning with a priori information of terrain may not be practically suitable depending on the size of the problem. It usually demands huge resources such as computational time and space even though the problem size is relatively small. Therefore, it is more suitable for off-line path planning than for on-line planning.

Sometimes, a priori information is not available because of technical difficulties or time-variant characteristics of the environment. Such examples include information about deep underwater, a piece of terrain inside enemy occupied territory, and a nuclear power plant disaster. In these cases, path planning without a priori information of terrain is more suitable. Since sensors are carried on board

he vehicle, the vehicle can easily adapt to a dynamically changing environment without using a priori information. Since the vehicle handles only localized information, the requirement for computational resources is usually small and almost constant regardless of the size of the problem.

Finding the optimal path using local information gathered by a sensor is very difficult. Naturally, the most important issue of the path planning without a priori information of terrain is how similar the path obtained by the algorithm is to an optimal path. However, if no a priori information is obtainable, then the path planning without a priori information is the only way to find a reasonable (or a near-optimal) path.

The other issue of the path planning without a priori information of terrain how to overcome a position that produces a local minimum (or maximum) value for an evaluation function whose value is optimized by a search. Though the depth-first search and the hill climb search strategy, which are suitable for the path plan without a priori information of terrain, systematically overcome the local maximum (and minimum) problem with backtracking, their performance is not very good.

A search strategy called PATH PLAN is developed in this study for autonomous vehicle navigation in situations where no a priori terrain information is available. To achieve better performance than that of either the depth-first or the hill climb search strategy, PATH PLAN adopts flavors of the hill climb and the A* search strategy as well as human heuristics. Though it utilizes local information gathered by the sensor, it uses both an estimation function and a cost function like the A* search strategy. However, it does not use the agenda required in the A* search strategy or the hill climb search strategy.

B. THESIS ORGANIZATION

Chapter II reviews previous work on path planning search strategy.

Chapter III presents a detailed problem statement for this thesis. Four models, which are the autonomous vehicle model, the terrain model, the obstacle model, and the sensor model, are introduced. The simulation facilities are also described.

A detailed description of PATH PLAN is presented in Chapter IV. This chapter explains how PATH PLAN, without a priori terrain information, directs an autonomous vehicle over three-dimensional terrain. It also describes how PATH PLAN overcomes local maximum (and minimum) problems and avoids obstacles.

Simulation results are presented in Chapter V. Five simulations were run under various terrain conditions. In order to evaluate the performance of PATH PLAN, paths obtained with PATH PLAN are compared with optimal paths obtained with A* search strategy. Global terrain information was made available to the A* search strategy, but this information was not utilized by PATH PLAN.

Finally, Chapter VI summarizes the PATH PLAN strategy developed in this research and suggests area where further work could be done. An appendix contains the PATH PLAN program which was written in LISP and implemented on a Symbolics 3675 LISP machine with a high-resolution color monitor.

II. SURVEY OF PREVIOUS WORK

A. INTRODUCTION

An autonomous vehicle must be able to reach a goal in an unexplored environment while avoiding collisions with obstacles. An early method for such navigation was invented for the robot SHAKEY by Hart, Nilsson, and Raphael[Ref. 1] and by Nilsson[Ref. 2]. This method assumes that the locations of all obstacles in the environment are known and that the obstacles can be approximated by polyhedral shapes. A visibility graph is created among the vertices of the polyhedra and the vehicle's and the target's positions such that any two connected nodes in the graph are mutually visible. The shortest path in the graph, connecting the robot to its target, is a collision-free path for vehicle. SHAKEY's method of navigation, however, is not always workable in an unexplored environment. An autonomous vehicle will have to move about to locate all the barriers and the obstacles prior to forming its world model. Lozano-Pérez and Wesley[Ref. 3] have extended SHAKEY's method to the problem of collision-free movements of a robot manipulator. Lozano-Pérez[Ref. 4] provides a comprehensive treatment of this problem with many references. Brooks[Ref. 5] describes another algorithm for path finding in a cluttered but known environment.

Other approaches to the navigation problem, based on visual identification of obstacles in a scene, have been discussed by Moravec[Ref. 6; Ref. 7; Ref. 8], Giralt, Sobek, and Chatila[Ref. 9], and Thompson[Ref. 10]. The JPL robot described by Thompson forms a terrain model using vision as the primary source of data. This model consists of a set of nontraversable walls built out of polygonal curves. The

robot HILARE, discussed by Giralt, Sobek, and Chatila, operates along similar lines. A vision-guided robot is Moravec's, which uses stereo vision to locate objects around it. It plans a collision-free path around these objects, follows the path for about a meter, and then restarts with scene analysis. This type of navigation can be used in an unexplored environment but appears to be rather slow.

As mentioned, there are many attempts for path planning. In this chapter, five previous path planning algorithms are introduced. These algorithms are most related to PATH PLAN.

B. PATH PLANNING ALGORITHMS

1. A HIERARCHICAL ORTHOGONAL SPACE APPROACH TO THREE-DIMENSIONAL PATH PLANNING

This algorithm is discussed by Wong and Fu[Ref. 11]. They present a methodology for three-dimensional collision-free path planning in which planning is done in the three-dimensional environment. Collision checking is done in each of the three orthogonal two-dimensional subspaces using primitive path segments. A hierarchical-path search method is used to speed up the search process. Their approach forms the basis for spatial planning in environments where no a priori knowledge is assumed. The three orthogonal two-dimensional projections are readily obtained from three orthogonal cameras in simple environments.

2. NAVIGATIONS FOR AN INTELLIGENT MOBILE ROBOT

This algorithm is described by Crowley[Ref. 12]. He describes a navigation system for a mobile robot equipped with a rotating ultrasonic range sensor. This navigation system is based on a dynamically maintained model of the local environment, called the composite local model. The composite local model integrates information from the rotating range sensor, the robot's touch sensor, and

a pre-learned global model as the robot moves through its environment. He describes techniques for constructing a line segment description of the most recent sensor scan(the sensor model), and for integrating such descriptions to build up a model of the immediate environment(the composite local model). The estimated position of the robot is corrected by the difference in position between observed sensor signals and the corresponding symbols in the composite local model. He also describes a learning technique where the robot develops a global model and a network of places. The network of places is used in global path planning, while the segments are recalled from the global model to assist in local path execution. His system is useful for navigation in a finite, pre-learned domain such as a house, office, or factory.

3. ROBOT NAVIGATION IN UNKNOWN TERRAINS USING LEARNED VISIBILITY GRAPHS

This algorithm is described by Oommen, Iyengar, Rao, and Kashyap[Ref. 13]. They discuss the problem of navigating an autonomous mobile robot through unexplored terrain containing obstacles. They present an algorithm for navigating a robot in unexplored terrain that is arbitrarily populated with disjoint convex polygonal obstacles in the plane. Their algorithm is proven to yield a convergent solution to each path of traversal. Initially, the terrain is explored using a rather primitive sensor, and the paths of traversal made may be near-optimal. The visibility graph that models the obstacle terrain is incrementally constructed by integrating the information about the paths traversed so far. At any stage of learning, the partially learned terrain model is represented as a learned visibility graph, and it is updated after each traversal. They prove that the learned visibility graph converges to the visibility graph with a probability of one when the

source and destination points are chosen randomly. Ultimately, the availability of the complete visibility graph enables the robot to plan globally optimal paths and also obviates further usage of sensors.

4. LEARNED NAVIGATION PATHS FOR A ROBOT IN UNEXPLORED TERRAIN

This algorithm is presented by Iyengar, Jorgensen, Rao, Weisbin[Ref. 14]. They propose a method of robot navigation which requires no pre-learned model, makes maximal use of available information, records and synthesizes information from multiple journeys, and contains concepts of learning that allow for continuous transition from local to global path optimum. Their model of the terrain consists of a spatial graph and a Voronoi diagram. Using acquired sensor data, polygonal boundaries containing perceived obstacles shrink to approximate the actual obstacles surfaces, free space for transit is correspondingly enlarged, and additional nodes and edges are recorded based on path intersections and stop points. Navigation planning is gradually accelerated with experience since improved global map information minimizes the need for further sensor data acquisition. Their method assumes obstacle locations are unchanging, navigation can be successfully conducted using two-dimensional projections, and sensor information is precise.

5. AUTOMATIC PATH PLANNING FOR A MOBILE ROBOT USING A MIXED REPRESENTATION OF FREE SPACE

This algorithm is proposed by Kuan, Brooks, Zamiska, and M. Das[Ref. 15]. They describe a path planning algorithm that uses a mixed representation of free space in terms of two shape primitives: generalized cones and convex polygons. Given a set of polygonal obstacles in space, their planning algorithm first identifies the neighborhood relations among obstacles and uses these relations to localize the

influence of obstacles on free space description and then to locate critical "channels" and "passage regions" in the free space. The free space is then decomposed into non-overlapping geometric-shaped primitives where channels are represented as generalized cones similar to Brooks[Ref. 5]. The passage regions are represented as convex polygons. Based on this mixed representation of free space, their planning algorithm uses two different strategies to path plan trajectories inside the channels and passage regions.

C. SUMMARY

Much research has recently focused on path planning under three-dimensional terrain with dynamically moving obstacles without a priori information of terrain. This chapter surveyed five previous path planning algorithms which are related to PATH PLAN. The detailed problem of PATH PLAN will be introduced in the next chapter.

III. DETAILED PROBLEM STATEMENT

A. INTRODUCTION

A search strategy for this thesis, PATH PLAN, is designed to guide the autonomous vehicle to a certain position under a circumstance that no a priori information of the terrain is available. The autonomous vehicle is modeled as an unmanned vehicle with a sensor which senses the terrain in a limited range, while the terrain is modeled from a three-dimensional digitized terrain database. The entire program is written in LISP, and it is executed on a Symbolics LISP machine.

B. AUTONOMOUS VEHICLE MODEL

The autonomous vehicle is an unmanned vehicle possessing some physical limitations of conventional vehicles and some human-like characteristics. The autonomous vehicle is modeled as follows:

1. It is not capable of locomotion over terrain slope exceeding a specified threshold.
2. It remembers all places it has visited.
3. It consumes a certain amount of energy to move itself to a new position, and it calculates this value.
4. It calculates the distance between its eight neighbor positions and the goal.
5. The size of autonomous vehicle is smaller than the cell size of terrain.
6. It moves to one of its eight neighbor positions at a time.
7. It moves along the direction of its heading.
8. It does not have a priori information about the terrain that will be traversed.

9. It is classified into two types: tank-type and jeep-type. Depending on the vehicle type, different uphill and downhill slope limitations and different rotational energy consumption values are used. Like a real tank and a real jeep, a tank-type vehicle has higher slope limitations than a jeep-type vehicle, and a tank-type vehicle spends more energy to rotate(change its heading) than a jeep-type vehicle does.

C. TERRAIN MODEL

The terrain model adopted for this study is a sample of a Special Defense Mapping Agency-produced digital terrain elevation database that was provided to the Naval Postgraduate School by the United States Army Combat Developments Experimentation Center(CDEC) at Fort Ord, California. The terrain sample covers a one kilometer x one kilometer area of Fort Hunter-Liggett, California. The terrain elevation, sampled at 12.5 meter increments, is available at one foot accuracy, but it is only displayed using ten foot accuracy. To enhance the movement of the vehicle, it is assumed that ten feet in real terrain is five feet in this terrain model. The terrain is modeled as following:

1. It is digitized terrain consisting of 6400 cells grouped in an array of 80 columns x 80 rows.
2. Each cell represents a 12.5 meter x 12.5 meter portion of terrain.
3. The resolution of the terrain elevation is five feet.
4. Different colors are used for representing different elevations; the darker the color, the higher the terrain elevation.
5. The start position, goal position and explicit obstacles are input by the user. The start position is the cell that has a light blue circle containing the letter "S".

The goal position is the cell that has a dark blue circle containing the letter "G".
The explicit obstacles are the cells that are red.

D. OBSTACLE MODEL

Two types of obstacles are considered in this study, explicit and implicit.

An explicit obstacle is anything that can physically block a vehicle movement. Because there is very little relation between an explicit obstacle and vehicle capability, a sensor alone can detect the presence of an explicit obstacle without considering the vehicle status.

Explicit obstacles can be further classified into two sub-classes, static and dynamic, depending on their temporal characteristics. Large man-made structures such as building are examples of static objects, whereas small man-made objects such as other vehicles are as dynamic objects. Human beings and animals can also be considered dynamic objects.

An implicit obstacle is any virtual obstacle derived by the interaction between the vehicle and the terrain. Because the maximum terrain slope that can be traversed by the vehicle is determined by the vehicle capability and direction of movement, the presence of an implicit obstacle depends on the vehicle capability and status. Therefore, a sensor alone can not determine the presence of the implicit obstacles without considering the vehicle status. Excessive slope areas or canyons are examples.

The obstacle is modeled as follows:

1. The size of an obstacle is the same as the cell size of terrain.
2. There is no room for the autonomous vehicle to pass between an obstacle and any obstacle neighboring obstacle located to the north, south, east, or west.

3. There is enough room for the autonomous vehicle to pass between an obstacle and any neighboring obstacle located to the northeast, northwest, southeast, or southwest.

E. SENSOR MODEL

The sensor of the autonomous vehicle functions in a limited sense, like a human eye. It senses, with a limited range, the environment surrounding the vehicle, and it processes the information. Thus, it provides limited but necessary world information for the vehicle. This sensor is modeled as follows:

1. It senses the eight neighbor positions which surround the vehicle position.
2. It measures the distance and the elevation difference between the vehicle position and its eight neighbor positions.
3. It detects the presence of the goal and the presence of explicit obstacles.

F. SIMULATION FACILITIES

1. LANGUAGE

Because system requirements and parameters were not specified well at the beginning of this study, like most artificial intelligence applications, it was expected to be extended and corrected several times. LISP is suited for ill-defined problems not only because LISP is able to represent and manipulate complex interrelationships among symbolic data [Ref. 16], but also because LISP is able to ease program modification and extension. Therefore, LISP was adopted as the implementation language for this study.

2. SYMBOLICS LISP MACHINE

The entire simulation program presented here is written in LISP and is executed on a Symbolics 3675 LISP machine with a color monitor. The Symbolics 3600 family of advanced symbolic processing machines covers a full range of

symbolic processing power and functionality. The Symbolics machines allow users to implement LISP with both speed and efficiency because they are uniquely designed for LISP. The machines are faster and more efficient than conventional computers with Von Neumann machine architecture for implementing applications ranging from Artificial Intelligence, CAD/CAM, high resolution graphics, and expert system research. The power, speed, and flexibility of the Symbolics LISP machines result from the optimized hardware design to match the LISP programming environment[Ref. 17].

G. SUMMARY

This chapter discussed models: the autonomous vehicle, the terrain, the obstacle, and the sensor. For the program implementation, LISP was adopted because of its properties for Artificial Intelligence applications, and the Symbolics 3675 LISP machine was used to execute the programs because of its speed and flexibility for LISP programs. Based on these defined models in this chapter, PATH PLAN will be described in the next chapter.

IV. NAVIGATION BY PATH PLAN

A. INTRODUCTION

This chapter describes PATH PLAN which is a search strategy for guiding an autonomous vehicle over three-dimensional terrain to a goal position. Since a priori information of the terrain is not always available to the autonomous vehicle because of technical difficulties or time-variant characteristics of the environment, PATH PLAN is designed to find a path in the dynamically changing environment without using a priori information of the terrain. Basically, PATH PLAN is based on some heuristics which are adopted from human behavior in order to make the behavior of the vehicle similar to that of a human being. PATH PLAN consists of two main routines: MAIN PATH PLAN and DETOUR PATH PLAN.

B. CONVENTIONS

In order to discuss the PATH PLAN strategy, it is necessary to introduce the following notations:

$P(x,y)$ = position located at x and y in map coordinates.

P_{start} = start position

P_{goal} = goal position

P_k = autonomous vehicle position after its k th movement from start position.

P_n = current vehicle position.

CP_{n+1} = one of candidate positions for P_{n+1} .

$OM(P_n)$ = obstacle marker for P_n .

$D(P_n, P_{n+1})$ = horizontal distance between P_n to P_{n+1} .

$D(P_{n+1}, P_{\text{goal}})$ = horizontal distance between P_{n+1} to P_{goal} .

$PM(P_{n+1})$ = path-marking value of P_{n+1} .

$\hat{E}$ = estimation function.

$\hat{E}(P_{n+1})$ = estimation function of position P_{n+1} .

R = rotational cost to change vehicle heading.

$R(P_n, P_{n+1})$ = rotational cost from P_n to P_{n+1} .

$\hat{R}(P_{n+1}, P_{goal})$ = minimum expected rotational cost from position P_{n+1} to P_{goal} .

$T(P_n, P_{n+1})$ = transitional cost from P_n to P_{n+1} .

C = local cost function.

$C(P_n, P_{n+1})$ = local cost consumed by vehicle to move from P_n to P_{n+1} .

F = evaluation function.

OF = obstacle evaluation function.

$F(P_{n+1})$ = evaluation function of position P_{n+1} .

C. HEURISTICS

Heuristics are any nonnumeric advice about what order to try the successors of a state for further search[Ref. 18]. In order to make the behavior of a vehicle similar to that of a human being attempting to find the goal without a map, the following heuristics were adopted.

1. Move toward the goal whenever possible.
2. Try not to visit the positions which were already explored.
3. Detour a steep slope area.
4. Do not visit positions which make it impossible to return to the current position.
5. If it is in obstacle environment, travel along the obstacles until the environment is cleared.

D. PATH PLAN STRATEGY

Based on given assumptions, it is possible to use a local cost function because an autonomous vehicle can calculate the energy required to move to its next position(the autonomous vehicle model 3) and to use an estimation function because an autonomous vehicle can measure distance between its eight neighbors and its goal(the autonomous vehicle model 4). However, it is impossible to use an agenda because an autonomous vehicle does not store parental information.It stores only the position information of cells that it has visited(the autonomous vehicle model 2). Since there are no classical search strategies, as shown in Table 1[Ref. 18], that fit this problem, a new search strategy, PATH PLAN, that uses both an estimation function and a local cost function without using agenda unlike the A* search strategy was studied.

Figure 1 shows the flow-chart of PATH PLAN. PATH PLAN is divided into two routines. MAIN PATH PLAN and DETOUR PATH PLAN. MAIN PATH PLAN is based on the heuristic 1, 2, 3 and 4. This routine is used until the vehicle encounters obstacles that block its movement. If the vehicle can not move closer to the goal because obstacles block its movement, DETOUR PATH PLAN, based on the heuristic 5 is used until the vehicle clears the obstacles. Since MAIN PATH PLAN and DETOUR PATH PLAN are based on the completely different heuristics, two different evaluation functions were used. They are as follows:

$$F = \begin{cases} C + \hat{E} & \text{for MAIN PATH PLAN.} \\ OF & \text{for DETOUR PATH PLAN.} \end{cases} \quad (4.1)$$

Table 1. CLASSIC SEARCH STRATEGIES

Name of search strategy	Uses agenda?	Uses estimation function?	Uses cost function?	Next state whose successors are found
Depth-first search	no	no	no	A successor of the last state, else a successor of a predecessor
Breath-first search	yes	no	no	The state on the agenda the longest
Hill-climbing (optimization)	yes	yes	no	The lowest-estimation successor of the last state
Best-first search	yes	yes	no	The state on the agenda of the lowest estimation value
Branch-and-bound	yes	no	yes	The state on the agenda of lowest total cost
A* search	yes	yes	yes	The state on the agenda of the lowest sum of estimation value and total cost

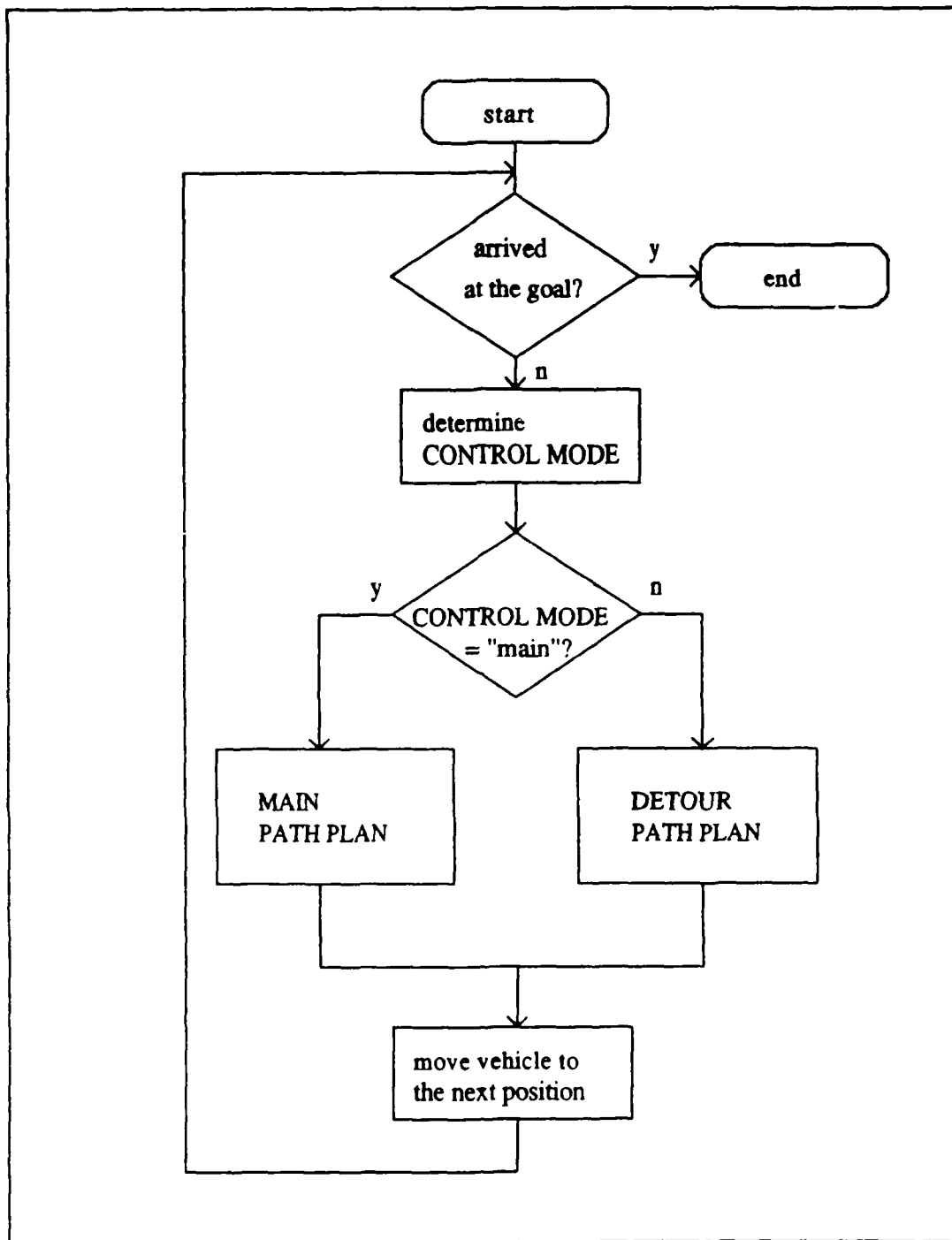


Figure 1. Flow-Chart of PATH PLAN

Before PATH PLAN permits a vehicle to move to a new position, it sets CONTROL MODE either to "main" or to "detour" based on the position and the direction of movement of the vehicle. If CONTROL MODE is set "main", then MAIN PATH PLAN will be executed. Otherwise, DETOUR PATH PLAN will be executed.

In order to guide a vehicle to its goal during the execution of MAIN PATH PLAN, a potential energy concept is introduced. The goal location has the lowest value, and the starting location has a larger potential energy value than that of the goal. In this study, the potential energy is determined by the Euclidean distance between the vehicle current location and the goal. Thus, it may be called distance potential energy. For example, if the distance between the goal and a position is 100 m, then the distance potential energy of the position is 100. This energy unit is also used to calculate the cost(or consumed energy) required to move the vehicle. For instance, the cost to move an autonomous vehicle 12.5m over the flat terrain is 12.5. Thus, the evaluation function returns a single number normalized by the distance potential energy.

E. MAIN PATH PLAN

During PATH PLAN, the MAIN PATH PLAN routine is used until the distance potential energy of the vehicle can not be reduced due to the obstacles: i.e., when the vehicle can not move closer to the goal due to obstacles. Figure 2 shows the procedure in MAIN PATH PLAN used to determine the next vehicle position(P_{n+1}). After evaluating the eight neighbor positions of the current position, a new position is chosen as P_{n+1} . This position was chosen because the value returned by its evaluation function was the lowest among those returned by the eight neighbors of the current vehicle position, P_n . Since some of the eight

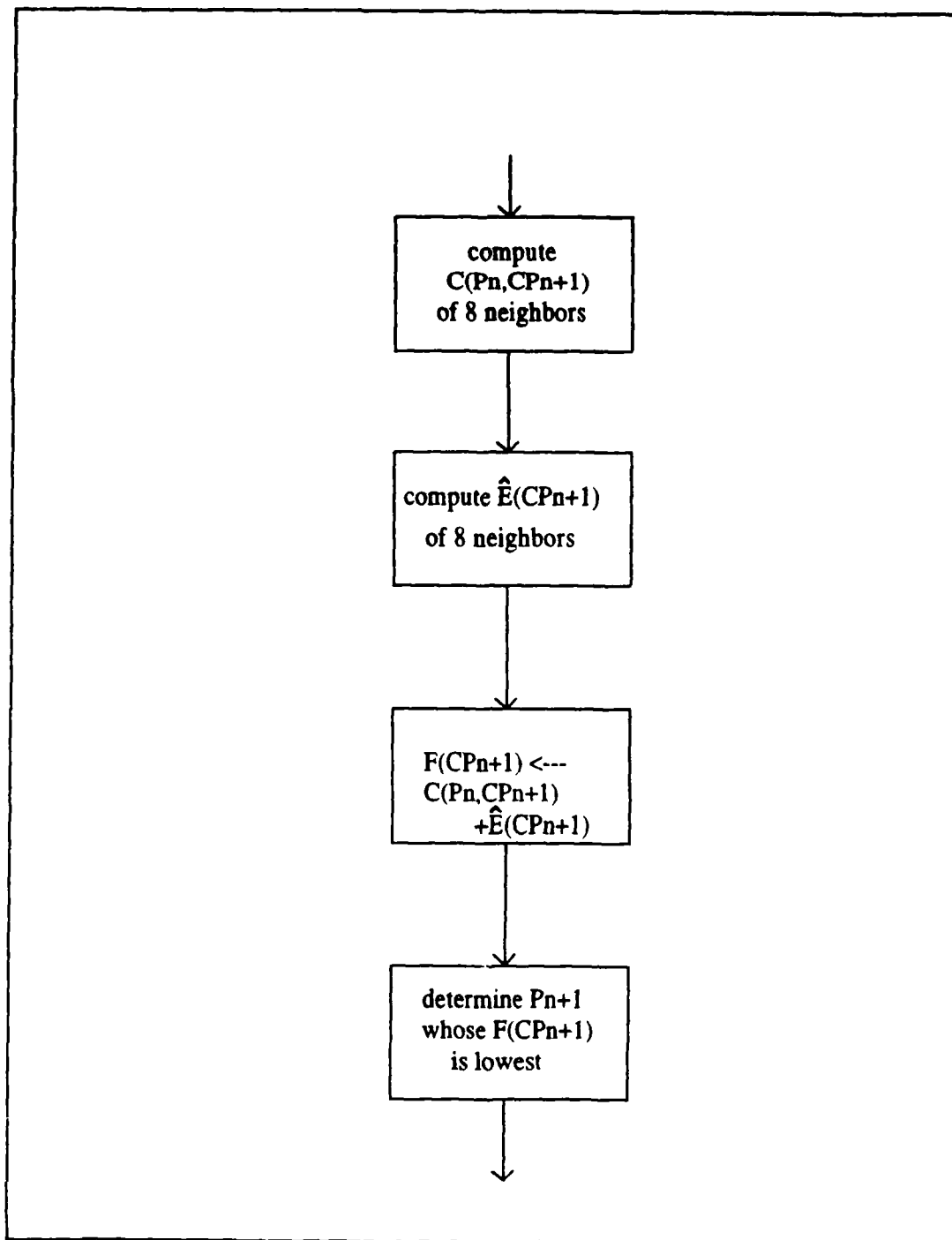


Figure 2. Flow-Chart of MAIN PATH PLAN

neighbors can be obstacles, as shown in Figure 3, their evaluation function definition is different from that of non-obstacle neighbors in order to prevent the vehicle from moving into the obstacle. A very high value, 10,000, is assigned to its evaluation function value when CP_{n+1} is determined to be an obstacle. If CP_{n+1} is not an obstacle, the evaluation function is defined as the sum of the local cost function and the estimation function.

1. Local Cost Function

The local cost function calculates the energy required to move the vehicle from its current position(P_n) to one of its eight neighbors. It does not include accumulated cost during the movement of the vehicle from the start position to its current position.

The local cost is classified into two types: the transitional cost to move an autonomous vehicle over three-dimensional terrain from P_n to CP_{n+1} and the rotational cost required to change the heading of the vehicle. Therefore, the local cost function is the sum of the transitional cost and the rotational cost. The local cost function can be represented as follows:

$$C(P_n, CP_{n+1}) = T(P_n, CP_{n+1}) + R(P_n, CP_{n+1}) \quad (4.2)$$

a. Transitional Cost

The transitional cost is defined as follows:

$$T(P_n, CP_{n+1}) = \text{slope-coefficient} * D(P_n, CP_{n+1}). \quad (4.3)$$

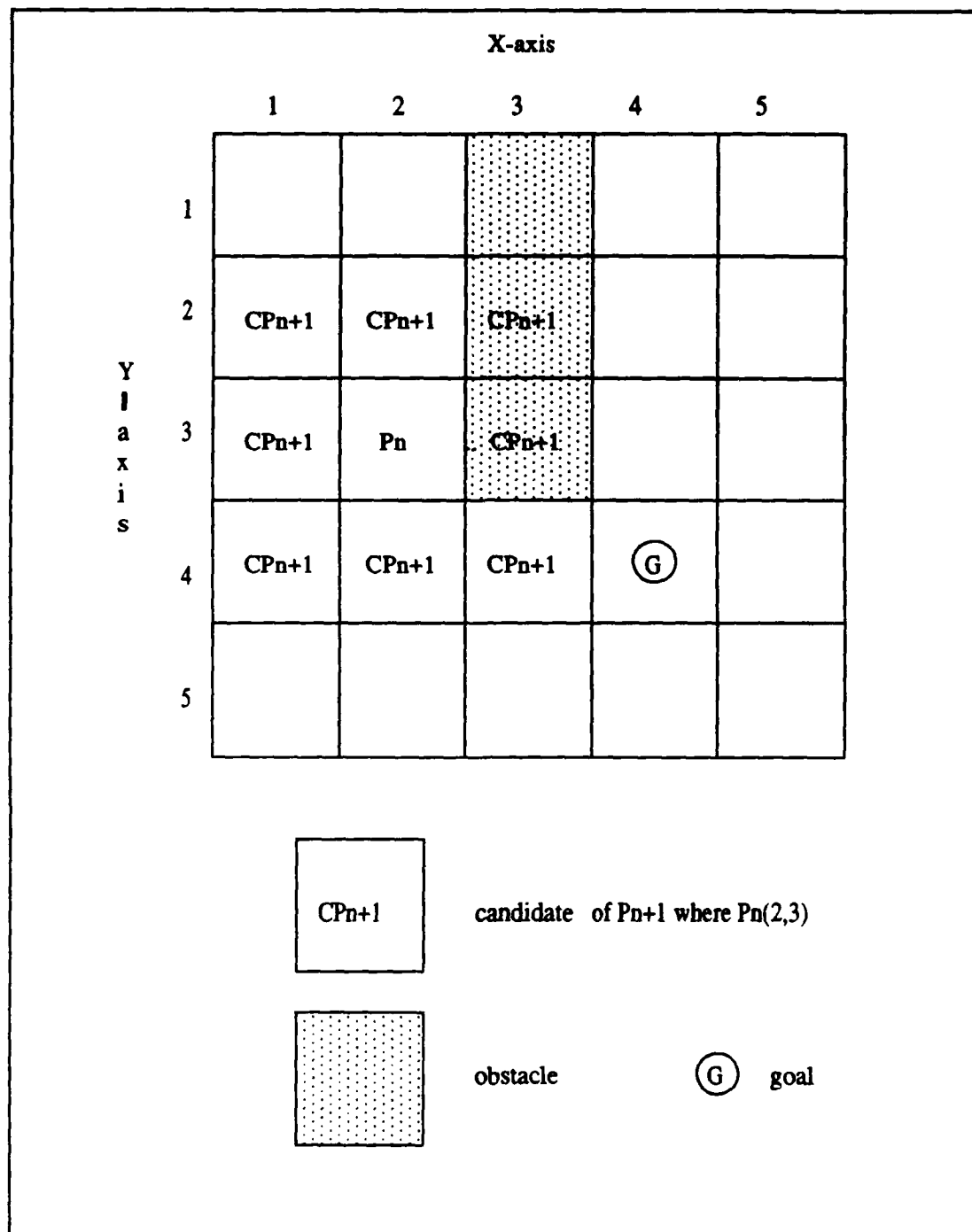


Figure 3. Candidates of P_{n+1}

The slope-coefficient is used to consider the energy needed to move a vehicle to a location with different elevation. For example, the slope-coefficient is 2.0 if twice the energy $D(P_n, CP_{n+1})$, is spent as vehicle moves from P_n to CP_{n+1} . Thus, it is 1.0 if the elevation difference between P_n and CP_{n+1} is zero. The slope-coefficient is generally influenced by the terrain slope-rate, but the slope-coefficient is not exactly proportional to the slope-rate because a vehicle gains energy when traversing a small downhill, and because a vehicle needs extra energy to reduce its speed when traversing a large downhill.

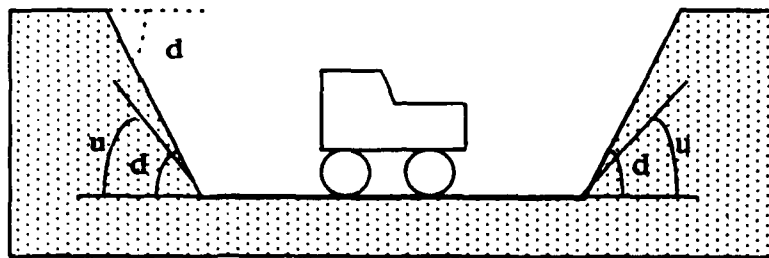
The slope-coefficient is influenced by only the elevation difference between two positions. It is independent of vehicle type. Each vehicle type is restricted to a maximum slope that a vehicle can traverse. For example, a tank-type autonomous vehicle can maneuver on a steep slope area while a jeep-type autonomous vehicle can not.

In this study, the slope-coefficients are selected from the slope-coefficient table for efficiency. Table 2 shows the slope-coefficients that are used for the two types of autonomous vehicles in this simulation study. As shown in the table, two different slope limitations are set according to the vehicle type. Beyond the slope limitations, a very big slope-coefficient, 10,000, is assigned to prevent the autonomous vehicle from moving across steep slope areas. The heuristic 3 is adopted in this way.

Because only localized information is utilized for finding a path, there is a chance to meet local maximum (or minimum) problems. Such problems occur when the uphill slope limitation and the downhill slope limitation of a vehicle are different. Figure 4 shows two examples. Figure 4-a shows one example where a vehicle can not get out of a ditch because its uphill slope limitation is less than its

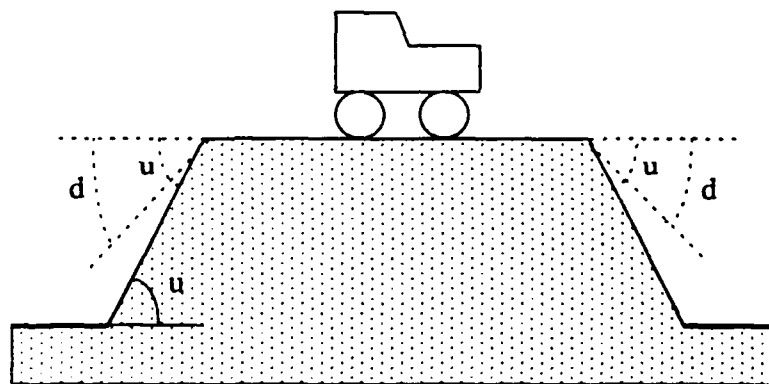
Table 2. SLOPE-COEFFICIENT

slope-rate(r) (ft/m)	slope-coefficient	
	jeep-type vehicle	tank-type vehicle
$r > 0.9$	10,000.0	10,000.0
$0.9 \geq r > 0.6$	10,000.0	2.2
$0.6 \geq r > 0.5$	1.9	1.9
$0.5 \geq r > 0.3$	1.6	1.6
$0.3 \geq r > 0.2$	1.3	1.3
$0.2 \geq r > 0.0$	1.0	1.0
$0.0 \geq r > -0.3$	0.8	0.8
$-0.3 \geq r > -0.5$	1.2	1.2
$-0.5 \geq r > -0.6$	1.5	1.5
$-0.6 \geq r > -0.9$	10,000.0	2.0
$-0.9 \geq r$	10,000.0	10,000.0



d = downhill slope limitation
 u = uphill slope limitation
 $d > u$

a. downhill slope limitation is larger than uphill slope limitation



d = downhill slope limitation
 u = uphill slope limitation
 $d < u$

b. downhill slope limitation is smaller than uphill slope limitation

Figure 4. Local Maximum and Minimum Problem

downhill slope limitation. Figure 4-b shows another example where a vehicle can not descend because its downhill slope limitation is less than its uphill slope limitation. To solve these problems, symmetrical uphill slope and downhill slope limitations are adopted. They prevent a vehicle from being trapped in a place where it can not get out(the heuristic 4). In the implementation, the TRANSITIONAL-COST function returns transitional cost from P_n to CP_{n+1} .

b. Rotational Cost

The rotational cost is the amount of energy needed to change the vehicle heading during movement from P_n to CP_{n+1} . A moving object has the property to maintaining the direction of its movement in accordance with Newton's first law of motion [Ref. 19]. Therefore, the larger the turning angle, the larger the rotational cost. Differing from the slope-coefficient, the rotational cost varies with the type of an autonomous vehicle. Table 3 shows the rotational costs. A 45 degree turning angle means either a 45 degree right turn or a 45 degree left turn. This turn is based on the direction of the movement of the vehicle. Thus, there are only five possible turning angles because an autonomous vehicle can go to only one of its eight neighbors. This rotational cost makes an autonomous vehicle tend to maintain its current direction of the movement.

Figure 5 shows that the white-colored path obtained without considering the rotational cost and the black-colored path obtained with the rotational cost are different in identical environment. When a portion of two paths is overlapped each other, the portion is colored in black because of the drawing sequence. In most cases, the path with the rotational cost is better than the path without the rotational cost. In the implementation, the ROTATIONAL-COST function returns rotational cost.

Table 3. ROTATIONAL COST($R(P_n, P_{n+1})$)

turning angle (degree)	rotational cost	
	jeep-type vehicle	tank-type vehicle
0	0	0
45	1	2
90	3	5
135	7	10
180	10	13



Figure 5. Comparison between Path Applied Rotational Cost and Path not Applied Rotational Cost(Jeep Type Vehicle Case)

2. Estimation Function

The estimation function, $\hat{E}(CP_{n+1})$ is the minimum estimated normalized energy consumed by a vehicle to moving from position CP_{n+1} to P_{goal} . The estimation function is defined by:

$$\hat{E}(CP_{n+1}) = D(CP_{n+1}, P_{goal}) + PM(CP_{n+1}) + \hat{R}(CP_{n+1}, P_{goal}). \quad (4.4)$$

$D(CP_{n+1}, P_{goal})$ is represented in the following manner:

$$D(CP_{n+1}, P_{goal}) = \sqrt{|x_0 - x|^2 + |y_0 - y|^2} \quad (4.5)$$

where $CP_{n+1}(x, y)$ and $P_{goal}(x_0, y_0)$.

Because $D(CP_{n+1}, P_{goal})$ is the horizontal distance from CP_{n+1} to P_{goal} , $D(CP_{n+1}, P_{goal})$ forces a vehicle to move toward the goal. As the position CP_{n+1} gets closer to the goal, $D(CP_{n+1}, P_{goal})$ becomes smaller. Thus, the value of the evaluation function, $F(P_{n+1})$, tends to smaller. Therefore, there is a very high chance for the vehicle to choose the closest position to the goal among its eight neighbors as its next position (the heuristic 1). It is the same idea that a ball has a tendency to roll to the bottom of a funnel (goal) when it is placed anywhere inside of the funnel.

One problem of this type of a search is to meet a local maximum (or minimum) as shown in Figure 6. If $\hat{E}(CP_{n+1})$ were defined as $D(CP_{n+1}, P_{goal})$ only, the vehicle would be trapped in the area surrounded by the high elevation area shown in Figure 6. Let's assume a situation P_{n-1} is located at $P(2,3)$, and P_n is located at $P(3,3)$. The goal position is $P(6,3)$, and the autonomous vehicle is a

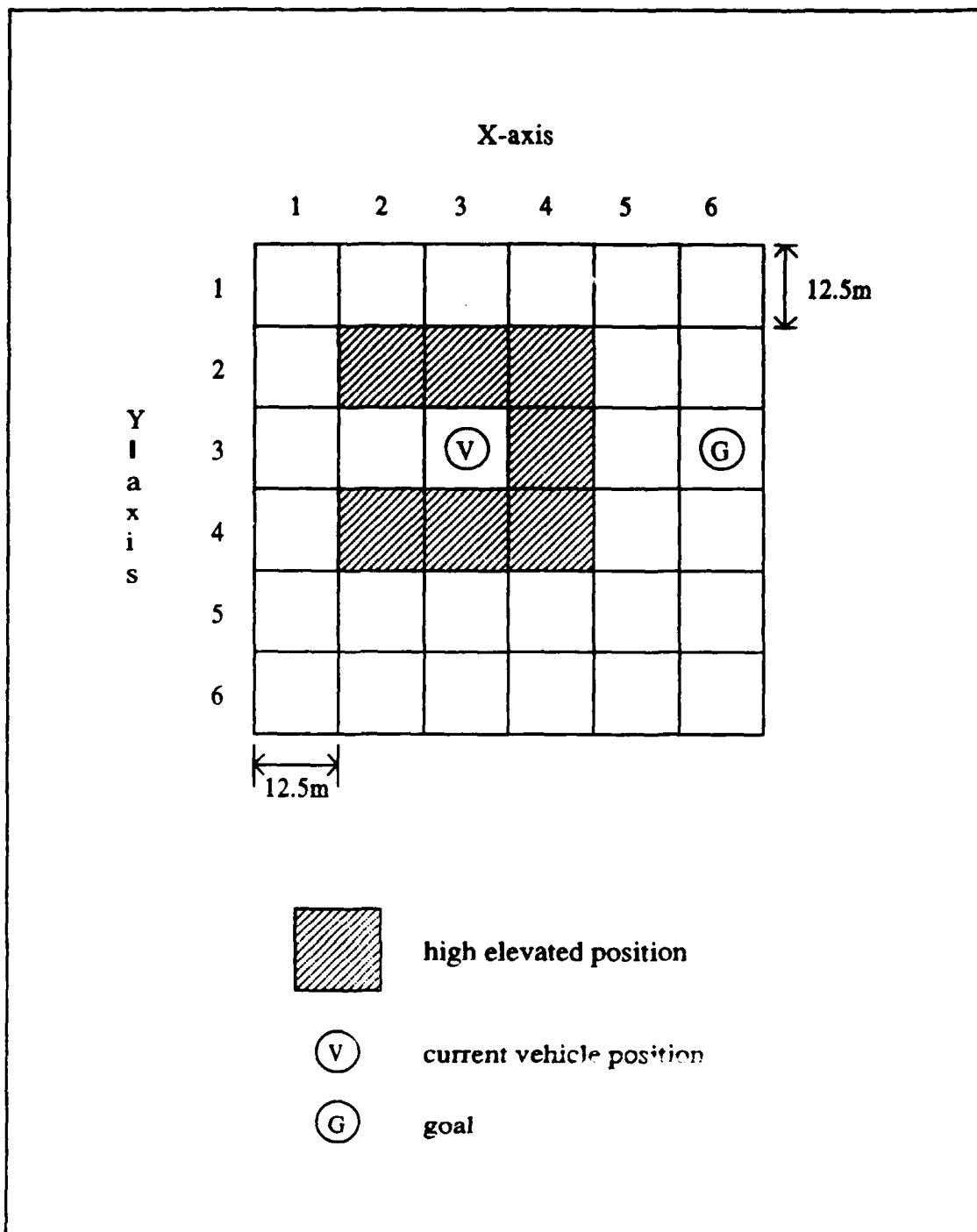


Figure 6. Vehicle Surrounded by the High Elevation Area

jeep-type vehicle. $P(2,2)$, $P(3,2)$, $P(4,2)$, $P(4,3)$, $P(2,4)$, $P(3,4)$, and $P(4,4)$ have higher elevation than the other positions, and the local cost from P_n to any high elevation area is larger than $C(P_n, P(2,3))$ which is 22.5. The local costs and the values of corresponding evaluation function are shown in Table 4. Under this situation, the vehicle will move to $P(2,3)$ because it provides the smallest evaluation function value. This means that P_{n+1} , the next vehicle position, will be $P(2,3)$. When P_{n+1} is chosen as $P(2,3)$, the values of the function of the neighbor positions of P_{n+1} are shown in Table 5. From the table, the next position(P_{n+2}) will be $P(3,3)$ because it has the smallest evaluation function value. Thus, the situation will be exactly the same as that of Table 4. There will be no further change in Table 4 and Table 5 while the vehicle moves back and forth between $P(2,3)$ and $P(3,3)$. In this case, the vehicle never arrives at the goal.

The path-marking value is used to solve this problem. Basically, this provides a way for the vehicle to memorize the positions visited. The path-marking value of each position is initially zero. Whenever the vehicle moves from P_n to CP_{n+1} , a path-marking value, $PM(CP_{n+1})$ which is the same as $C(P_n, CP_{n+1})$, is assigned to CP_{n+1} . When the path-marking value is added to the evaluation function, the evaluation cost of $P(2,3)$ will be increased by 12.5. However, the other positions will have the same evaluation costs as those in Table 4 because they have not been visited by the vehicle. Table 6 depicts this situation. Because of the increased evaluation cost of $P(2,3)$, $P(2,3)$ no longer has the lowest evaluation cost; $P(4,3)$ has the lowest. Thus, the vehicle will choose $P(4,3)$ as the next vehicle position, P_{n+1} . Therefore, the path-marking value helps the vehicle to get out of a trap by making it resist going to the position which has been already visited by the vehicle(the heuristic 2).

Table 4. CASE OF $\hat{E}(CP_{n+1}) = D(CP_{n+1}, P_{goal})$

candidate of P_{n+1} (CP_{n+1})	$C(P_n, CP_{n+1})$		$\hat{E}(CP_{n+1})$	$F(CP_{n+1})$	P_{n+1}
	$T(P_n, CP_{n+1})$	$R(P_n, CP_{n+1})$			
P(2,2)	50.0	7	51.5	108.5	P(2,3)
P(3,2)	50.0	3	39.5	92.5	
P(4,2)	50.0	1	30.0	81.0	
P(4,3)	50.0	0	25.0	75.0	
P(4,4)	50.0	1	30.0	81.0	
P(3,4)	50.0	3	39.5	92.5	
P(2,4)	50.0	7	51.5	108.5	
P(2,3)	12.5	10	50.0	72.5	

* where $P_n = P(3,3)$

Table 5. CASE OF $\hat{E}(CP_{n+2}) = D(CP_{n+2}, P_{goal})$

candidate of P_{n+2} (CP_{n+2})	$C(P_{n+1}, CP_{n+2})$		$\hat{E}(CP_{n+2})$	$F(CP_{n+2})$	P_{n+2}
	$T(P_{n+1}, CP_{n+2})$	$R(P_{n+1}, CP_{n+2})$			
P(1,2)	17.7	1	63.7	82.4	P(3,3)
P(2,2)	50.0	3	51.5	104.5	
P(3,2)	50.0	7	39.5	96.5	
P(3,3)	12.5	10	37.5	60.0	
P(3,4)	50.0	7	39.5	96.5	
P(2,4)	50.0	3	51.5	104.5	
P(1,4)	17.7	1	62.5	81.2	
P(1,3)	12.5	0	63.7	76.2	

* where $P_{n+1} = P(2,3)$

Table 6. CASE OF $\hat{E}(CP_{n+1}) = D(CP_{n+1}, P_{goal}) + PM(CP_{n+1})$

candidate of P_{n+1} (CP_{n+1})	$C(P_n, CP_{n+1})$		$\hat{E}(CP_{n+1})$		$F(CP_{n+1})$	P_{n+1}
	$T(P_n, CP_{n+1})$	$R(P_n, CP_{n+1})$	$D(CP_{n+1}, P_{goal})$	$PM(CP_{n+1})$		
P(2,2)	50.0	7	51.5	0.0	108.5	P(4,3)
P(3,2)	50.0	3	39.5	0.0	92.5	
P(4,2)	50.0	1	30.0	0.0	81.5	
P(4,3)	50.0	0	25.0	0.0	75.0	
P(4,4)	50.0	1	30.0	0.0	81.0	
P(3,4)	50.0	3	39.5	0.0	92.5	
P(2,4)	50.0	7	51.5	0.0	108.5	
P(2,3)	12.5	10	50.0	12.5	85.0	

* where $P_n = P(3,3)$

In order to enhance the tendency for the vehicle to move towards the goal, $\hat{R}(CP_{n+1}, P_{goal})$, the minimum expected rotational cost from the position CP_{n+1} to P_{goal} , is added to the definition of $\hat{E}(CP_{n+1})$. No matter which path the vehicle chooses to follow to the goal, it will consume rotational cost of at least $\hat{R}$, the minimum expected rotational cost. To calculate $\hat{R}$, the turning angle which is necessary to make the vehicle heading align with the direction from CP_{n+1} to the goal is used. If the turning angle is a , the minimum turning angle is derived by following LISP formula:

$$(\text{setf minimum-turning-angle } (* (\text{truncate } (/ a 45)) 45)). \quad (4.6)$$

Using the derived minimum turning angle, $\hat{R}$ is obtained from Table 3. In the implementation, a two-dimensional array, INNER-ARRAY, is declared to store the sum of the $D(P, P_{goal})$ and $PM(P)$ of each position.

3. Evaluation Function

The evaluation function is divided for two cases. For the case that CP_{n+1} is not an obstacle, it is the sum of the cost function and the estimation function. For the case that CP_{n+1} is an obstacle, it is always 10,000 in order to prevent the vehicle from moving to the obstacle. Therefore, The evaluation function of MAIN PATH PLAN is defined as follows:

$$F(CP_{n+1}) = \begin{cases} C(P_n, CP_{n+1}) + \hat{E}(CP_{n+1}) & \text{if } CP_{n+1} \text{ is not an obstacle.} \\ 10,000 & \text{if } CP_{n+1} \text{ is an obstacle.} \end{cases} \quad (4.7)$$

From formulas (4.2), (4.4) and (4.7), the evaluation function of MAIN PATH PLAN, when CP_{n+1} is not an obstacle, is represented as follows:

$$F(CP_{n+1}) = T(P_n, CP_{n+1}) + R(P_n, CP_{n+1}) + D(CP_{n+1}, P_{goal}) + PM(CP_{n+1}) + \hat{R}(CP_{n+1}, P_{goal}). \quad (4.8)$$

In the implementation, the LOCAL-COST-FUNCTION function returns, $C(P_n, CP_{n+1})$, the local cost of CP_{n+1} . The ESTIMATION-FUNCTION function returns $\hat{E}(CP_{n+1})$, the minimum estimated cost of CP_{n+1} . The EVALUATION-FUNCTION function returns $F(CP_{n+1})$, the sum of the local cost and the minimum estimated cost.

F. DETERMINATION OF CONTROL MODE

When the vehicle is blocked by a group of obstacles whose shape is concave, such as that shown in Figure 7, MAIN PATH PLAN, which is based on the heuristic 1, 2, 3 and 4, discussed so far can not find a path effectively. To clear the obstacles, the evaluation function values of all the positions surrounded by the obstacles have to be larger than those of the two corners of the concave obstacle, $P(3,4)$ and $P(12,4)$. This is necessary because MAIN PATH PLAN makes the vehicle move to the position that has the smallest evaluation value among its eight neighbors. When the vehicle makes an initial contact with one of the obstacles, the evaluation function value of $P(3,4)$ or $P(12,4)$ is larger than that of any inside position because of the longer distance from the goal. However, as the vehicle visits the inside positions one by one, their evaluation function values increase because the path-marking value is added as the vehicle visits them. Therefore, the vehicle keeps visiting all the inside positions until all the points have larger evaluation values

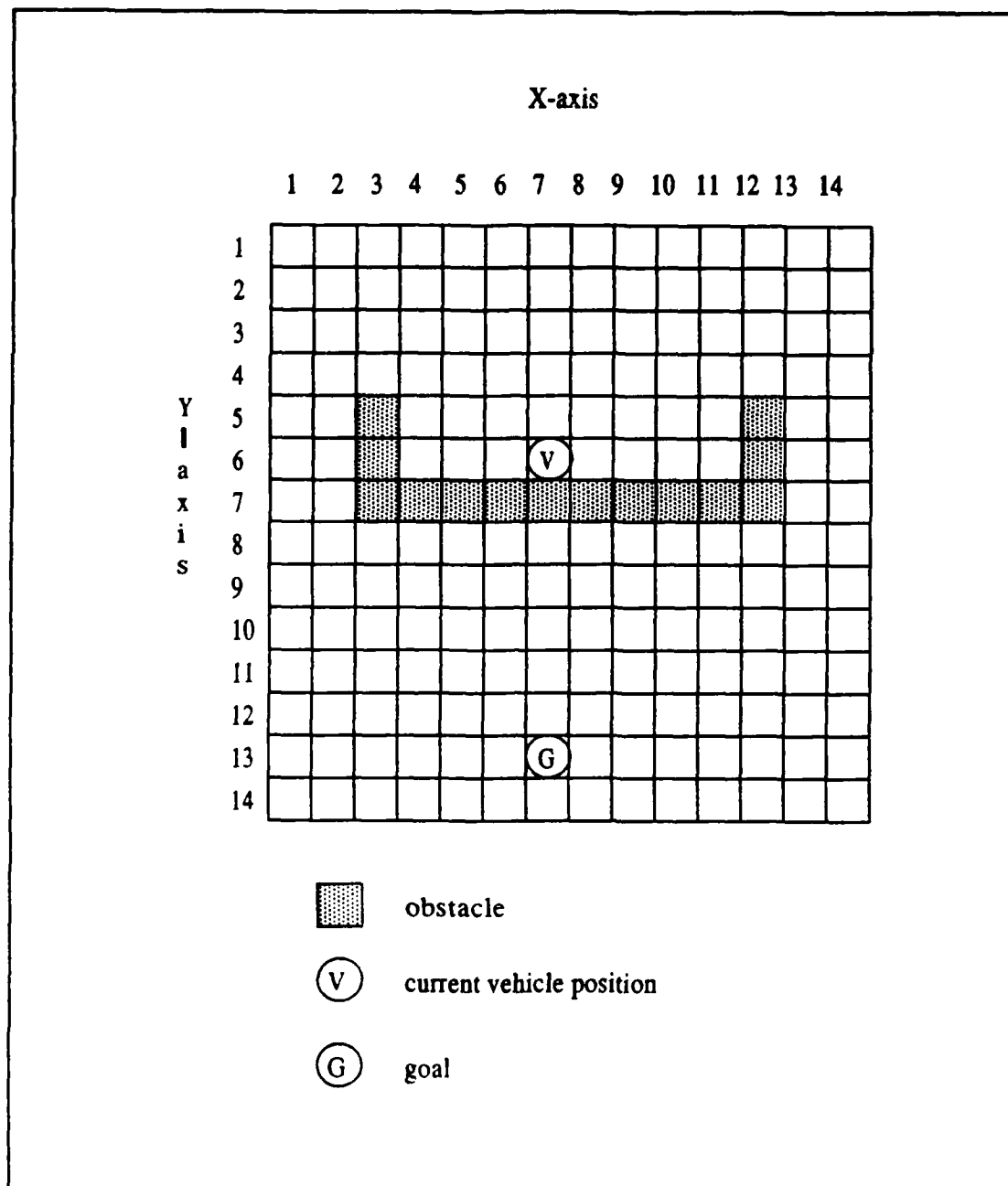


Figure 7. Example for the Problem of MAIN PATH PLAN

than those associated with one of the corner positions, $P(3,4)$ and $P(12,4)$.

To solve this kind of problem, DETOUR PATH PLAN based on the heuristic 5 was introduced. Whenever the vehicle can not move a new position which is closer to the goal than the current vehicle position due to the obstacles, DETOUR PATH PLAN is used, instead of MAIN PATH PLAN to find the next position.

As mentioned earlier, CONTROL MODE is used for selecting the correct routine(MAIN PATH PLAN or DETOUR PATH PLAN) during PATH PLAN. Therefore, it is important to find the conditions when CONTROL MODE is set to "main" or "detour". Figure 8 shows the conditions to set the CONTROL MODE.

The global variable *nearest-distance-before* is used to determine CONTROL MODE. Because it is updated to $D(P_n, P_{goal})$ during MAIN PATH PLAN and it is not updated during DETOUR PATH PLAN, it is used to check if the vehicle moved closer to the goal during MAIN PATH PLAN. It is also used to check if the vehicle cleared the obstacles during DETOUR PATH PLAN. This variable provides an easy check because the vehicle cleared the obstacle when the distance from the current vehicle position to the goal was less than *nearest-distance-before* which was set to the distance between the vehicle position at the last MAIN PATH PLAN execution and the goal.

There are two conditions required to set CONTROL MODE "detour" as follows:

1. vehicle position(P_n) is in contact with the obstacle and
2. $D(P_n, P_{goal})$ is larger than the value of *nearest-distance-before*;
i.e., P_n is not closer to the goal than P_{n-1} .

If the vehicle position, P_n , fails to satisfy either the first condition or the second condition, then CONTROL MODE will be set to "main". Otherwise, it will

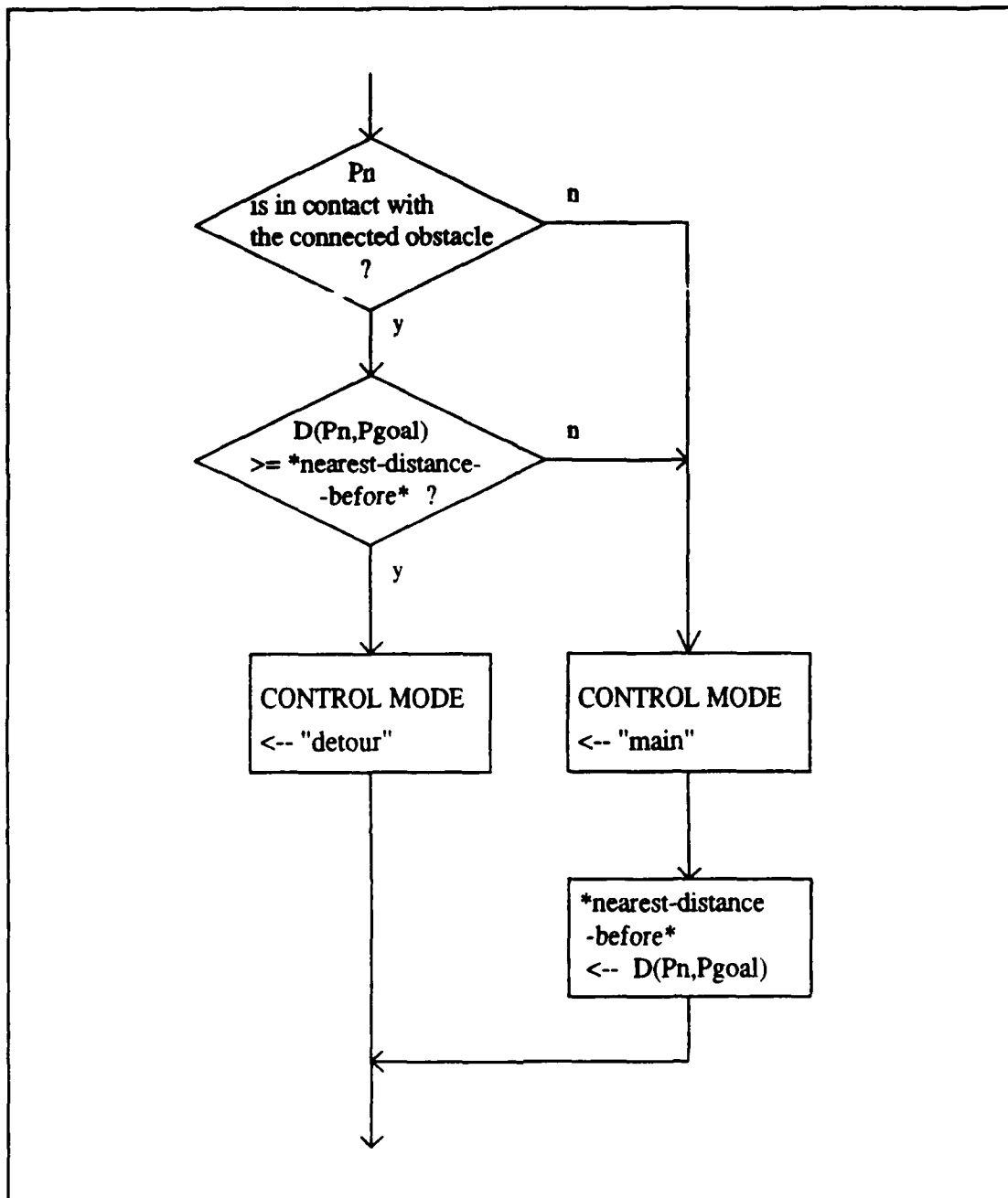


Figure 8. Determination of CONTROL MODE

function is introduced. The candidate possessing the largest function value is chosen as the new obstacle marker. The obstacle marker evaluation function is defined in Table 7. If a candidate is not an obstacle, the function value is zero because the obstacle marker must be placed on one of obstacles. If the candidate is one of obstacles but it is not interlinked with $OM(P_n)$ following the connections among the obstacles, the function value is also zero because the next obstacle marker has to be interlinked with $OM(P_n)$ in order to force vehicle movement along the connected obstacles.

If a candidate is located at the same location of $OM(P_{n-1})$, then the function value will be 5 which is an arbitrary value chosen to give less favor to the location; i.e., this value is low enough for the vehicle to discourage use of this position again as the next obstacle marker. For the other cases, $D(OM(P_n), COM(P_{n+1}))$, the distance between $OM(P_n)$ and a candidate, is the function value of $COM(P_{n+1})$. It is always larger than 5 because the minimum distance between two obstacles is 12.5. Therefore, if one candidate is chosen as the next obstacle marker, then the marker can be interlinked with $OM(P_n)$.

There are several cases to interlink from $OM(P_n)$ to $OM(P_{n+1})$ resulting from numerous connected obstacle shapes. Figure 11 shows five typical cases with respect to the number of steps needed to interlink two obstacle markers. The step means the smallest number of obstacles passed in order to interlink from the old obstacle marker to the new obstacle marker using only four directions which are north, south, east, and west. Figure 11-a shows an example where no step is necessary to interlink the old obstacle marker and the new obstacle marker because they are located at the same position. Figure 11-b shows a one step example, and Figure 11-c shows a two step example. Figure 11-d shows a three step example. In

Table 7. OBSTACLE MARKER EVALUATION FUNCTION

condition of OM(P _n +1)	obstacle marker evaluation function value
is not an obstacle	0
can not interlink with OM(P _n)	0
is same as OM(P _n)	5
the other cases	D(OM(P _n P, OM(P _n +1))

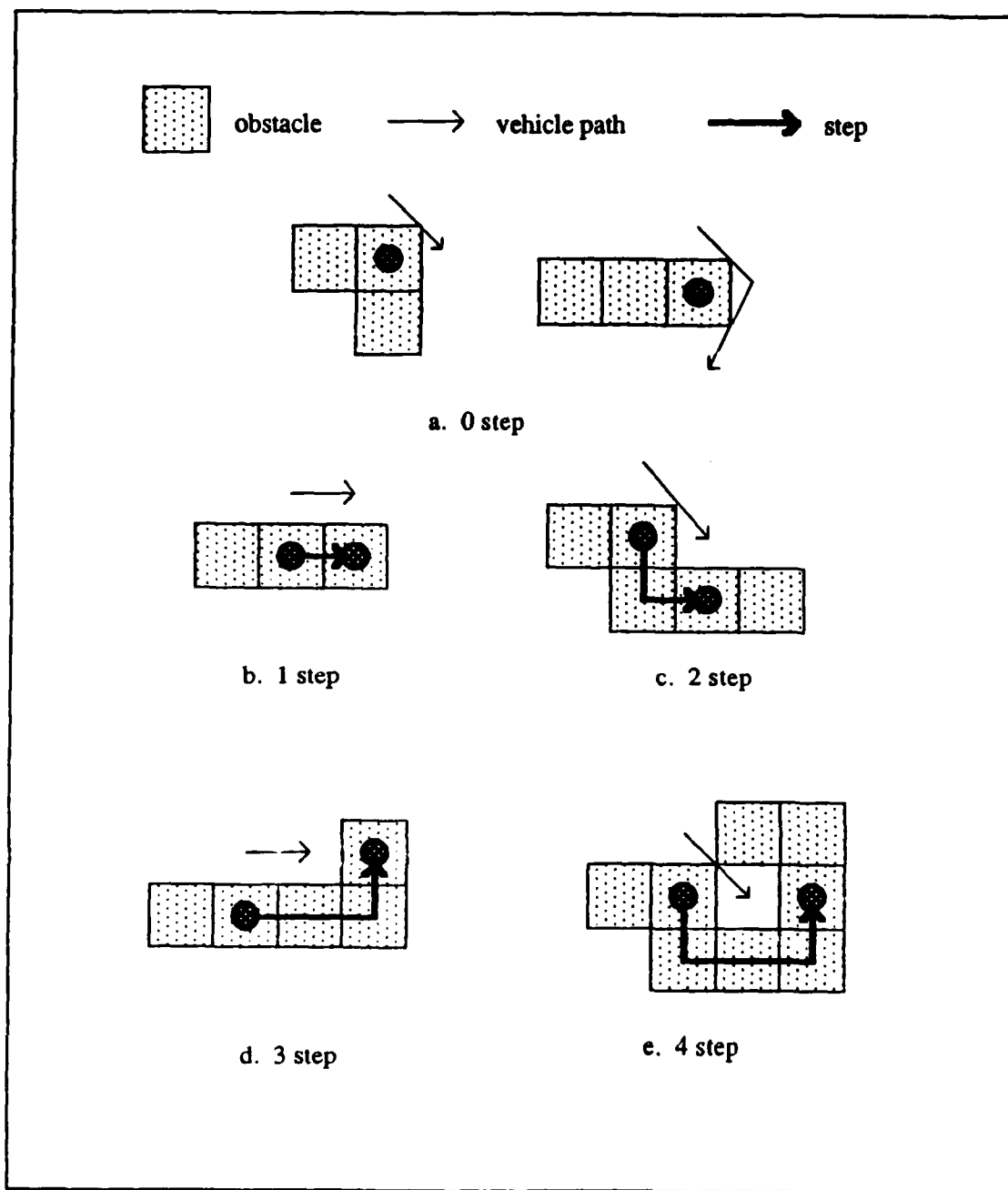


Figure 11. Steps to Interlink $OM(P_{n+1})$ with $OM(P_n)$

this case, there are two possible candidates for the obstacle marker; i.e., one under the new position and the other to the right side of the new position. Though both look good, the latter, which is located farther than the former, is selected as the new obstacle marker in order to help the vehicle advance. Thus, Figure 11-d shows a three step example rather than one step example. Figure 11-e shows a four step example. In this case, there are three possible candidates; i.e., the south, the east, and the north obstacles with respect to the new position. In this implementation, the east obstacle is chosen as the next obstacle marker. This implementation not only reduces the number of steps considered but also simplifies the obstacle marker evaluation function because the east obstacle has the longest distance from $OM(P_n)$ among three candidates. In the implementation, the GET-OBS-MARKER function returns $OM(P_n)$ for P_n .

2. Obstacle Evaluation Function

Based on the heuristic 5, the evaluation function of DETOUR PATH PLAN is adopted to force vehicle movement along and clear of the obstacles when the vehicle can not move closer to the goal due to the obstacles.

The obstacle evaluation function value is shown in Table 8. Like MAIN PATH PLAN, the position that has the lowest obstacle evaluation function value from among the obstacle evaluation function values of the eight neighbor positions will be the next vehicle position. Therefore, if CP_{n+1} is an obstacle(either implicit or explicit), then a very high value , 10,000, is assigned to the evaluation function value to prevent the vehicle from moving to an obstacle. Also, if CP_{n+1} is not in contact with the connected obstacles, then 10,000 is assigned to the evaluation function value in order to prevent the vehicle from moving away from the connected obstacle while detouring connected obstacles. Because the new vehicle position must

Table 8. OBSTACLE EVALUATION FUNCTION

condition of CP_{n+1}	OF(CP_{n+1})
is an obstacle (implicit or explicit)	10,000
is not in contact with the connected obstacles	10,000
OM(CP_{n+1}) is not able to interlink with OM(P_n) no more than 4 steps	10,000
the other cases	R(P_n,CP_{n+1})

have an obstacle marker which is interlinked with the old obstacle marker no more than four steps, 10,000 is assigned to CP_{n+1} if the obstacle marker of CP_{n+1} can not be interlinked with $OM(CP_n)$ no more than four steps. Otherwise, the rotational cost, $R(P_n, P_{n+1})$, is assigned to the obstacle evaluation function value.

The following two examples show that how the obstacle marker and the obstacle evaluation function work for DETOUR PATH PLAN. The first example is shown in Figure 12. Assume that a jeep-type vehicle is located at $P(3,2)$ and has traveled the route shown as an arrow in the figure and $OM(P_n)$ is $P(3,3)$. Obstacles $P(1,3)$, $P(2,3)$, $P(3,3)$ and $P(3,4)$ are connected, and obstacles $P(5,1)$, $P(5,2)$ and $P(4,2)$ are also connected. Table 9 shows that P_{n+1} is $P(4,3)$ whose obstacle evaluation value is the lowest among those of the eight neighbor positions. The obstacle evaluation function values of $P(3,3)$, $P(3,2)$ and $P(4,2)$ are 10,000 because they are obstacles. The obstacle evaluation function values of $P(2,1)$, $P(3,1)$ and $P(4,1)$ are 10,000 because $P(2,1)$, $P(3,1)$ and $P(4,1)$ are not in contact with the connected obstacles. Therefore, $P(2,2)$ and $P(4,3)$ are the only candidates for the next vehicle position because obstacle markers of $P(2,2)$ and $P(4,3)$ can be interlinked to $OM(P_n)$ with no more than four steps. Consequently the next vehicle position, P_{n+1} , will be $P(4,3)$ because $R(P(3,2), P(4,3))$, 1, is less than $R(P(3,2), P(2,2))$, 10. The obstacle marker of $P(4,3)$ is $P(3,3)$, the same as the zero step example in Figure 11-a. If the vehicle is still in "detour" CONTROL MODE after the previous vehicle movement, P_{n+2} will be $P(4,4)$ as shown in Table 10. It is in contact with the connected obstacles, and it requires a smaller rotational cost than $P(3,2)$ does. The obstacle marker of $P(4,4)$ is $P(3,4)$, also the same as a one step example in Figure 11-b. Thus, DETOUR PATH PLAN forces vehicle movement along the connected obstacles.

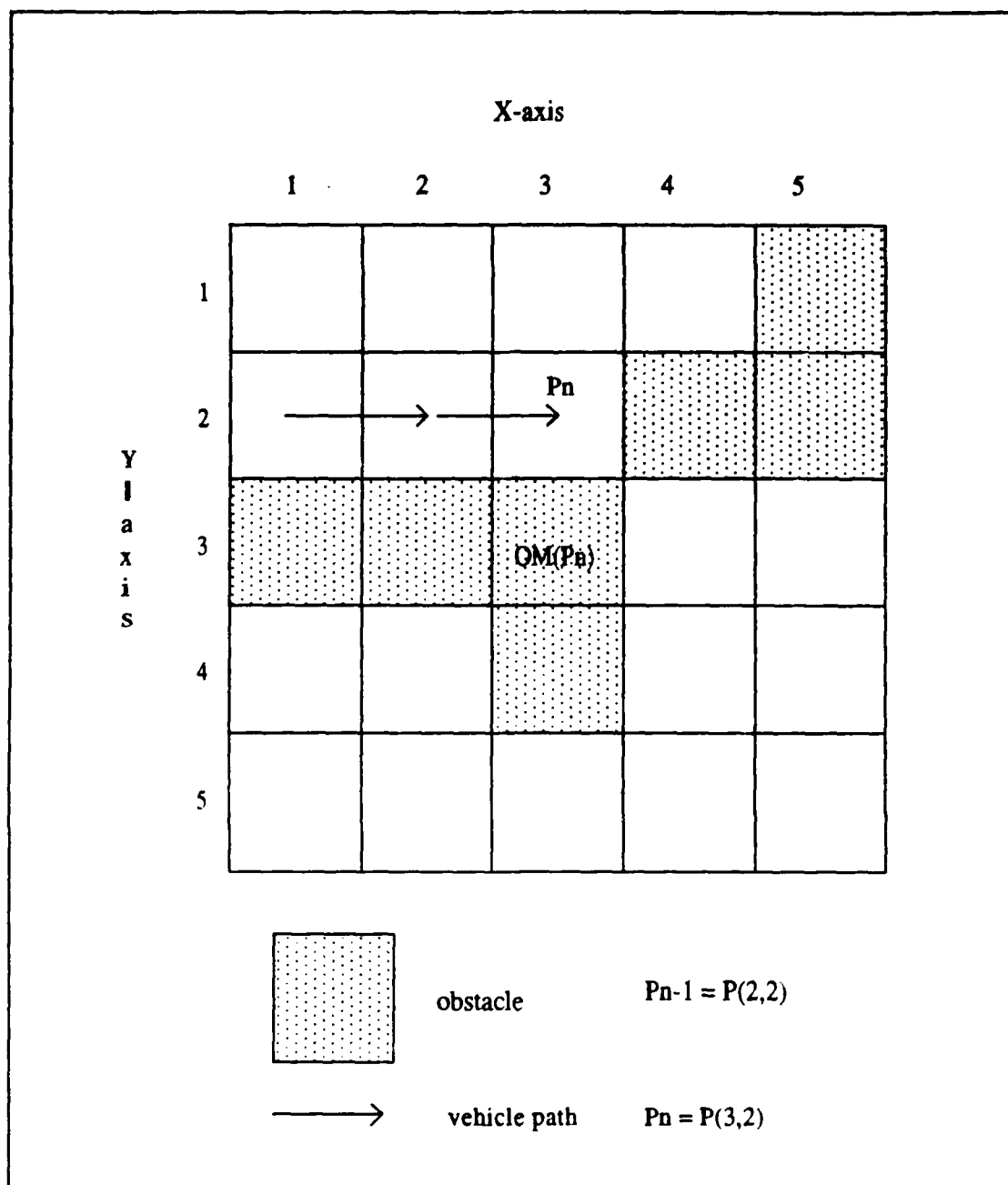


Figure 12. Example of DETOUR PATH PLAN

Table 9. OBSTACLE EVALUATION FUNCTION($OF(CP_{n+1})$) IN FIGURE 12

CP_{n+1}	$R(P_n, CP_{n+1})$	$OF(CP_{n+1})$	P_{n+1}
$P(3,1)$	-	10,000	$P(4,3)$
$P(4,1)$	-	10,000	
$P(4,2)$	-	10,000	
$P(4,3)$	1	1	
$P(3,3)$	-	10,000	
$P(2,3)$	-	10,000	
$P(2,2)$	10	10	
$P(2,1)$	-	10,000	

Table 10. OBSTACLE EVALUATION FUNCTION(OF(Pn+2)) IN FIGURE 12

CPn+2	R(Pn+1,CPn+2)	OF(CPn+2)	Pn+2
P(3,2)	10	10	P(4,4)
P(4,2)	-	10,000	
P(5,2)	-	10,000	
P(3,3)	-	10,000	
P(3,3)	-	10,000	
P(3,4)	-	10,000	
P(4,4)	1	1	
P(5,4)	-	10,000	

Another example is shown in Figure 13. Assume that the vehicle has moved from $P(4,2)$, P_{n-2} , to $P(4,2)$, P_n , via $P(3,3)$, P_{n-1} . The obstacle markers of P_{n-2} , P_{n-1} and P_n are $P(3,2)$, $P(3,4)$ and $P(4,3)$, respectively. If the obstacle evaluation function in Table 8 does not include the third test condition whether the obstacle marker is interlinked with $OM(P_n)$ no more than four steps or not, P_{n+1} can be either $P(4,1)$ or $P(5,2)$ because both $P(3,1)$, the obstacle marker of $P(4,1)$, and $P(5,3)$, the obstacle marker of $P(5,2)$, are interlinked to $OM(P_n)$. Moreover, both $FE(P(4,1))$ and $FE(P(5,2))$ is 1 which is derived from the relational cost in Table 8. Therefore, there is a possibility that the vehicle may backtrack unnecessarily to the positions already visited. However, if the obstacle evaluation function includes the third condition, P_{n+1} becomes $P(5,2)$. $FE(P(4,1))$ becomes 10,000 because $P(3,1)$, the obstacle marker of $P(4,1)$, is not interlinked to $OM(P_n)$ with no more than four steps. Thus, $FE(P(5,2))$ is the lowest among the eight neighbor positions as shown in Table 11. Thus, DETOUR PATH PLAN prevents the vehicle from backtracking unnecessarily.

In DETOUR PATH PLAN, the obstacle evaluation function is the only function to calculate the evaluation value as formula (4.1). In the implementation, the EVALUATION-FUNCTION1 function returns $F(CP_{n+1})$ for DETOUR PATH PLAN.

II. SUMMARY

This chapter discussed PATH PLAN based on human heuristics. Basically, it consists of two parts: MAIN PATH PLAN and DETOUR PATH PLAN. PATH PLAN provides a reasonable path for an the autonomous vehicle traversing over three-dimensional terrain, though the autonomous vehicle does not have the global information of terrain. It also makes the autonomous vehicle avoid obstacles – both

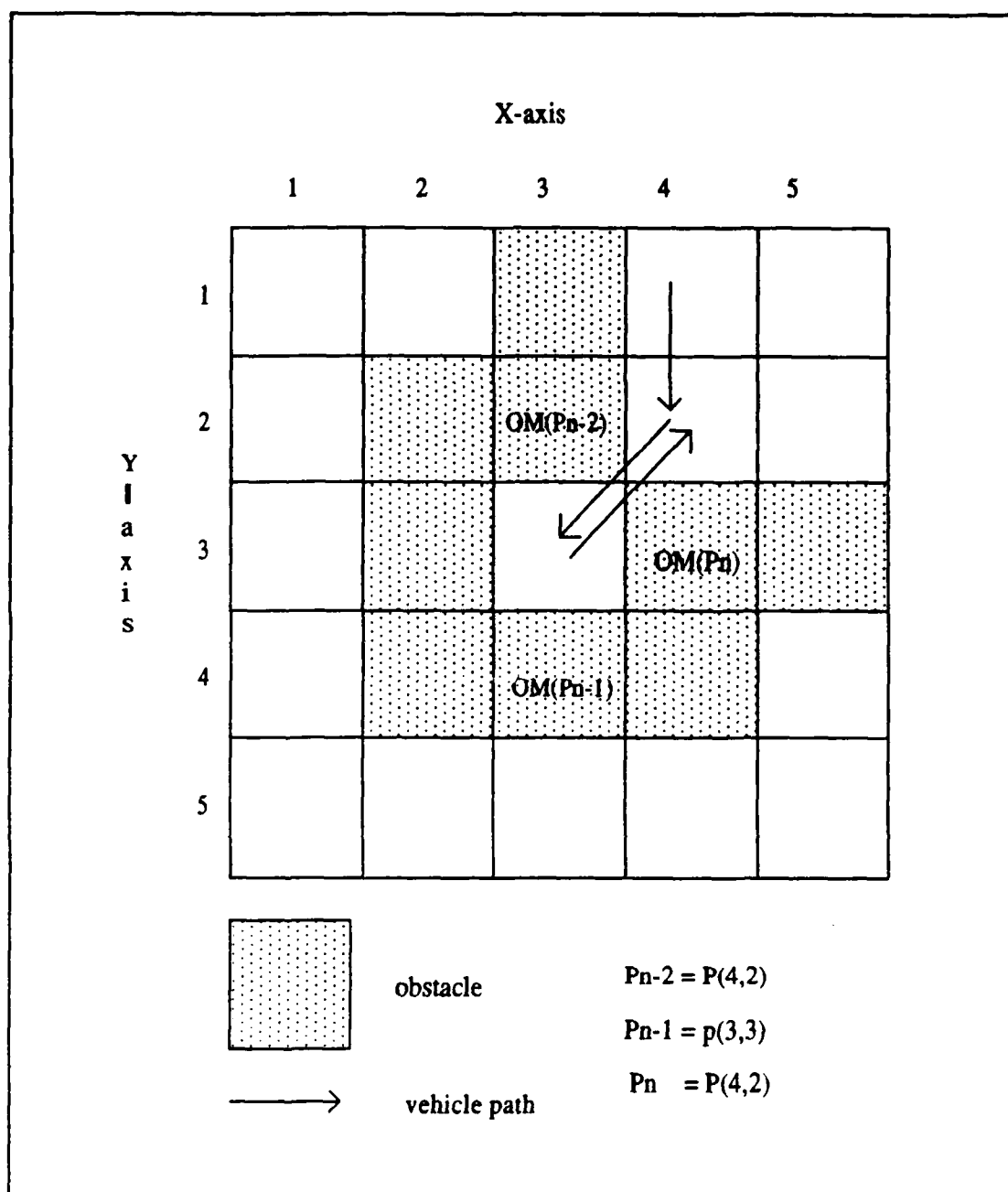


Figure 13. Example of Determination of the Obstacle Marker

Table 11. OF(CP_n+1) IN FIGURE 13

CP_n+1	R(P_n,CP_n+1)	OF(CP_n+1)	P_n+1
P(3,1)	-	10,000	P(5,2)
P(4,1)	-	10,000	
P(5,1)	-	10,000	
P(5,2)	1	1	
P(5,3)	-	10,000	
P(4,3)	-	10,000	
P(3,3)	10	10	
P(3,2)	-	10,000	

explicit obstacles and implicit obstacles – as well as the local maximum (and minimum) problems. The next chapter presents the simulation results and an evaluation of PATH PLAN.

V. SIMULATION AND EVALUATION

A. INTRODUCTION

The last chapter introduced PATH PLAN which provides a reasonable path for an autonomous vehicle traversing three-dimensional terrain without using the global information of the terrain traversed by the vehicle. This chapter presents the simulation results obtained from various terrain conditions encountered on Fort Hunter-Liggett. Five simulation results are shown in this chapter with varying the conditions of terrain and obstacles. In order to evaluate the performance of PATH PLAN, paths derived by PATH PLAN and paths derived by the A* search strategy under the exactly same terrain and conditions are compared. Factors influencing the heuristic power[Ref. 20] were especially noted in this evaluation.

B. SIMULATION RESULTS

Five simulation results are presented in this section. The first three simulation results were obtained without explicit obstacles as changing terrain roughness, and the latter two results were obtained with explicit obstacles as changing obstacle complexity in order to test PATH PLAN on various terrain conditions. Each simulation result shows two paths. One path is obtained with the tank-type autonomous vehicle, and the other path is with the jeep-type vehicle. In order to distinguish two paths, two colors, white and black, are used. The former path is colored in white, and the latter path is colored in black. When a portion of two types of paths are overlapped, the portion is colored in black because the black path denoting the latter path is drawn later than the white path denoting the former path is.

1. SIMULATION RESULTS WITHOUT EXPLICIT OBSTACLES

Three simulation results were obtained without explicit obstacles. The first simulation, which is shown in Figure 14, is performed on a flat area of terrain. The second simulation, which is shown in Figure 15, is performed on a moderately sloped area of terrain. The last simulation, which is shown in Figure 16, is performed on a highly sloped area of terrain. The results of these simulations are summarized in Table 12. The path obtained with the jeep-type autonomous vehicle is slightly different from the path obtained with the tank-type autonomous vehicle because the rotational cost and the uphill and downhill slope limitations are dependent on the type of the vehicle. Generally, the total cost of the path for the jeep-type autonomous vehicle is less than that for the tank-type autonomous vehicle for two reasons. One is that a jeep-type vehicle spends less energy to turn than a tank-type does. The other is that the path for a jeep-type vehicle is usually shorter than the path for a tank-type vehicle because the jeep-type vehicle can more easily correct its heading to the goal direction when its heading is disturbed by obstacles or steep slope than the tank-type vehicle can. Figure 13, 14, and 15 reveal that PATH PLAN finds very reasonable paths for an autonomous vehicles from various terrain conditions without explicit obstacles.

2. SIMULATION RESULTS WITH EXPLICIT OBSTACLES

Two tests were performed on the same terrain while changing the complexity of the explicit obstacle arrangement. One was performed on the terrain with relatively simple explicit obstacle arrangement such as that shown in Figure 17. The other was performed with relatively complicated explicit obstacle arrangement such as that shown in Figure 18. The results of two tests are summarized in Table 13. In Figure 17, the path for the jeep-type vehicle is slightly



Figure 14. Simulation on a Flat Area Without Explicit Obstacles

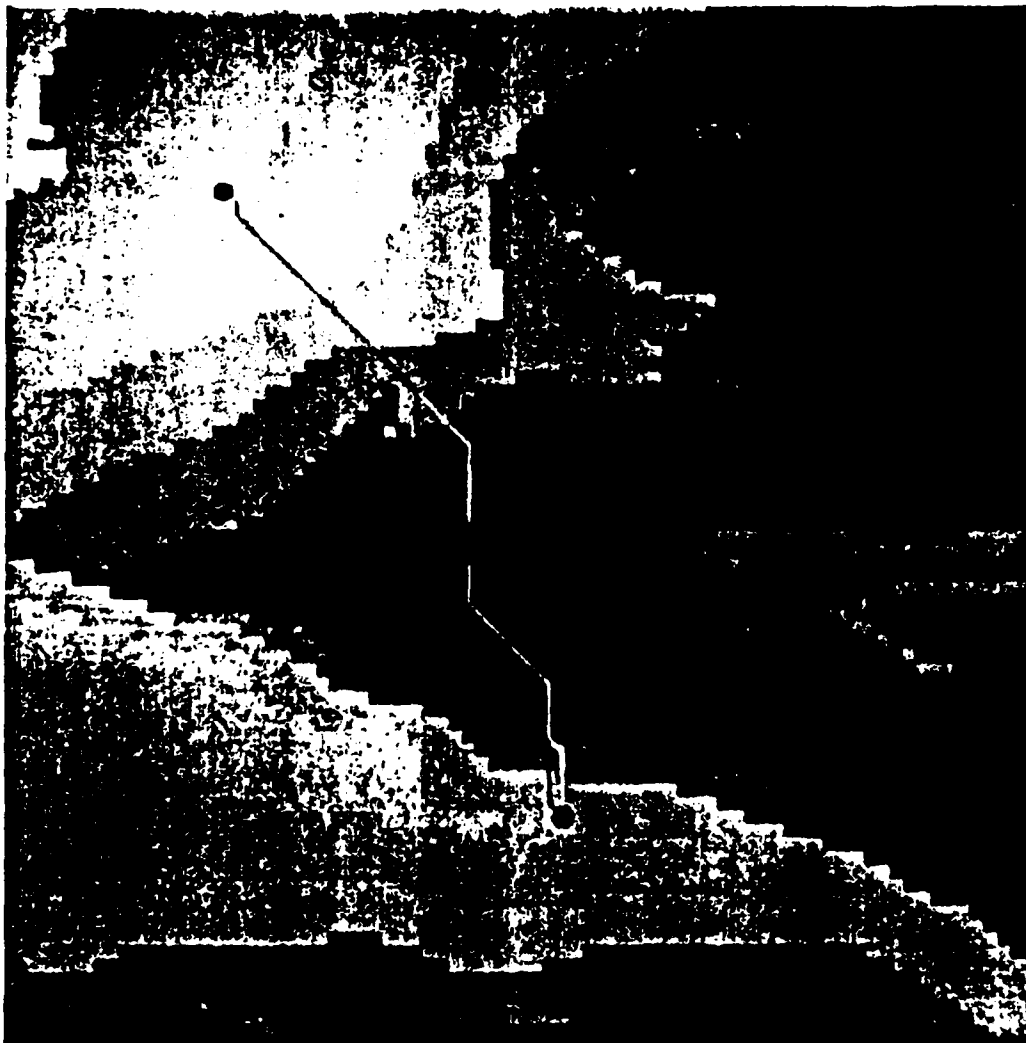


Figure 15. Simulation on a Moderately Sloped Area Without Explicit Obstacles



Figure 16. Simulation on a Highly Sloped Area Without Explicit Obstacles

Table 12. SIMULATION RESULTS WITHOUT EXPLICIT OBSTACLES

simulation	vehicle type	result	
		time required to find a path (second)	cost
simulation in Figure 14	tank	0.86	469.99
	jeep	0.83	466.99
simulation in Figure 15	tank	1.22	786.72
	jeep	1.21	778.10
simulation in Figure 16	tank	1.47	913.75
	jeep	1.35	875.05



Figure 17. Simulation on Terrain With Simple Explicit Obstacle Arrangement



Figure 18. Simulation on Terrain With Complicated Explicit Obstacle Arrangement

Table 13. SIMULATION RESULTS WITH EXPLICIT OBSTACLES

result vehicle type		time required to find a path (second)	cost
simulation			
simulation in Figure 17	tank	6.31	1619.38
	jeep	6.17	1577.57
simulation in Figure 18	tank	8.30	1632.24
	jeep	4.50	1154.78

different from the path for the tank-type vehicle during MAIN PATH PLAN. However, in Figure 18 a portion of two paths is very different because DETOUR PATH PLAN, when DETOUR PATH PLAN is initiated for the vehicles, chooses different obstacle markers whose selections are greatly influenced by the initial contact angle to a group of connected obstacles. Figure 17 demonstrates that PATH PLAN with the moderately complex obstacle arrangement finds a reasonable path that a human being can easily expect. Figure 18, however, demonstrates that PATH PLAN with complicated obstacle arrangement finds a path that a human being can hardly expect in a reasonable amount of time. Thus, the time spent by PATH PLAN is considerably less than the time needed by a human being, who possesses global information, to find a path while avoiding obstacles. PATH PLAN, which does not use the global information of the terrain, can not find an optimal cost path, but, the simulation results show that PATH PLAN always finds a reasonable path on the terrain with complicated obstacle arrangement in relatively short time period.

C. COMPARISON WITH A^* SEARCH

When the global information of the terrain is available, an A^* search strategy can find an optimal cost path to the goal. Though this study was based on local information rather than global information, it is desirable to compare the path found by A^* search with the path found by PATH PLAN in order to evaluate the performance of PATH PLAN. For comparison purposes the heuristic power concept is utilized. The three factors influencing heuristic power are:

1. the cost of the path.
2. the maximum number of OPEN nodes during path search, and
3. the time required to find the path.

The OPEN node used here is defined as the leaf node of the search tree[Ref. 20]. PATH PLAN is compared with A^* search in terms of these three factors. For the evaluation, three tests, which are shown in Figure 19, 20 and 21, were performed on a flat area, a moderately sloped area and a highly sloped area of terrain, respectively. The white colored paths were obtained by A^* search and the black-colored paths were obtained by PATH PLAN. Some portions of the white-colored path are overlapped by the black-colored paths. In order to simplify the comparison, no explicit obstacles were included and a jeep-type autonomous vehicle was chosen.

1. COST OF PATH

This heuristic power factor shows how much a path derived by PATH PLAN is similar to a path derived by A^* search, a minimum cost path. The comparison of PATH PLAN cost with A^* search cost is shown in Table 14. For the case of Figure 19, the difference between PATH PLAN cost and the A^* search cost is less than 1 % of the A^* search cost. For the case of Figure 20, the difference is less than 2 % of the A^* search cost. However, for the case of Figure 21, the difference is about 16% of the A^* search cost. Therefore, it is realized that PATH PLAN usually provides an almost optimized path on flat or moderately sloped terrain and a reasonable path on highly sloped terrain.

2. MAXIMUM NUMBER OF OPEN NODES

This heuristic power factor shows how efficiently PATH PLAN utilizes computer resources during its search operation. The maximum numbers of OPEN nodes during a path search are shown in Table 15. The A^* search strategy uses 99, 129, and 139 maximum number of OPEN nodes for Figure 19, 20, and 21, respectively. On the other hand, PATH PLAN uses only one OPEN node for all the



Figure 19. Simulation on a Flat Area for Comparison Between
PATH PLAN and A* Search



**Figure 20. Simulation on a Moderately Sloped Area for Comparison Between
PATH PLAN and A* Search**



Figure 21. Simulation on a Highly Sloped Area for Comparison Between
PATH PLAN and A* Search

**Table 14. COMPARISON BETWEEN PATH PLAN AND A* SEARCH SPECT TO COST
WITH RESPECT TO COST**

cost	simulation		
	simulation in Figure 19	simulation in Figure 20	simulation in Figure 21
PATH PLAN	552.34	738.13	658.66
A* search	549.34	728.06	567.28
difference (PATH PLAN - A* search)	3.00	10.07	91.38
$\frac{\text{difference}}{\text{A* search}}$	0.006	0.014	0.161

**Table 15. COMPARISON BETWEEN PATH PLAN AND A* SEARCH WITH
RESPECT TO MAXIMUM NUMBER OF OPEN NODES**

maximum number of open nodes	simulation		
	simulation in Figure 19	simulation in Figure 20	simulation in Figure 21
PATH PLAN	1	1	1
A* search	99	129	139
$\frac{\text{PATH PLAN}}{\text{A* search}}$	0.010	0.008	0.007

three tests because it does not need to use an agenda. Therefore, PATH PLAN is remarkably efficient in the use of computer memory.

3. TIME TO FIND PATH

This heuristic power factor shows that how fast PATH PLAN finds a path. The comparison of the time required to find a path by PATH PLAN with the time required to find a minimal cost path by the A^* search is shown in Table 16. Though it takes the A^* search 135.72 seconds, 337.61 seconds, and 229.57 seconds, respectively to find a minimal cost path on Figure 19, 20, and 21, respectively, it takes PATH PLAN less than 2 seconds for each case. The speed difference between two methods is huge. The execution speed of PATH PLAN is over one hundred times as fast as that of the A^* search.

D. SUMMARY

This chapter discussed the simulation results and the evaluation results of PATH PLAN. The simulation results show that PATH PLAN finds a reasonable path from terrain either with or without the explicit obstacles in short time. From the comparison of PATH PLAN with the A^* search, it can be said that PATH PLAN has more heuristic power than A^* search because PATH PLAN wins two comparison criteria areas with great margin even though PATH PLAN can not guarantee finding a optimized cost path. Moreover, A^* search can not find a path without a priori information of the terrain. PATH PLAN, however, can find a path without a priori information.

**Table 16. COMPARISON BETWEEN PATH PLAN AND A* SEARCH WITH
RESPECT TO TIME REQUIRED TO FIND A PATH**

time required to find a path	simulation		
	simulation in Figure 19	simulation in Figure 20	simulation in Figure 21
PATH PLAN	1.10	1.25	1.12
A* search	135.72	337.61	229.57
<u>PATH PLAN</u> A* search	0.008	0.004	0.005

VI. SUMMARY AND CONCLUSIONS

A. RESEARCH CONTRIBUTIONS

The PATH PLAN search strategy makes contributions in five areas. The first contribution is PATH PLAN navigates an autonomous vehicle to a goal while obtaining the local terrain information from the sensor in a dynamically changing environment without a priori terrain information. It is suitable for real-time(on-line) path planning. Thus, PATH PLAN allows an autonomous vehicle to proceed into a hostile or contaminated environment. The second contribution is that PATH PLAN navigates a vehicle to the goal in environments containing obstacles of any shape group. PATH PLAN finds an implicit obstacle by processing terrain information gathered by a sensor, and then it prevents a vehicle from moving into the implicit obstacle. Moreover, it guides a vehicle to avoid the connected explicit obstacles. The third contribution is that PATH PLAN overcomes local maximum and minimum problem, even though only local information of terrain is available. The fourth contribution is that PATH PLAN provides a reasonable path for the vehicle while considering anisotropic three-dimensional terrain cost. This differs from other path planning algorithms that simply consider isotropic terrain cost. The last contribution is that PATH PLAN search strategy requires very short time and a minimum amount of computer memory to find a path because it does not use an agenda.

B. RESEARCH EXTENSIONS

It is desirable to extend this research in four areas. First, PATH PLAN may be designed to use the path-marking value instead of the obstacle marker to avoid

obstacles. Since the path-marking value is used for preventing the vehicle from visiting positions previously visited, a vehicle may be able to avoid the connected obstacles without undesirable backtracking. If this works, PATH PLAN may be simpler than the current version.

Second, PATH PLAN can be designed to distinguish implicit and explicit obstacles during DETOUR PATH PLAN. The current PATH PLAN treats any implicit obstacle as explicit obstacle during DETOUR PATH PLAN. Because an implicit obstacle is considered with respect to the vehicle's approaching direction, PATH PLAN should not use the obstacle marker on the implicit obstacle; it only prevents the vehicle from moving to the implicit obstacle.

Third, PATH PLAN can be extended to consider surface characteristics of terrain. For example, the moving cost on a muddy area is much larger than that on an asphalt-covered area. Moreover, the slope-limitation will be influenced by the surface characteristics of terrain.

Finally, PATH PLAN can be improved by using the realistic data of the autonomous vehicle. The rotational cost, the slope-coefficient and the slope-limitation were estimated in this research. Using realistic vehicle data in PATH PLAN will produce an improved path.

APPENDIX

```
;;; -- Mode: LISP; Syntax: Common-lisp; Package: USER --
```

```
.. *****  
;;  
;; program_id : PATH PLAN  
;; assigned_by : professor Kwak  
;; written_by : Ok, Do Kyeong  
;; date      : 1 FEB 1989  
;;  
;; this program is used for finding a path  
;; under following conditions  
;;  
;;   - three dimensional terrain  
;;     . the size of a pixel is 12.5 meters * 12.5 meters  
;;     . each pixel has its height in feet e.g, 100 ft  
;;     . the size of terrain used is 80 * 80 pixels, 1 km * 1 km  
;;     . each height value is divided by 10  
;;       for example, if some pixel has a 1003 ft height value  
;;         then 1003 DIV 10 = 100  
;;         so, 100 ft will be used in this program  
;;  
;;   - a position represented by (X, Y) format  
;;     . so we can use from (0, 0) to ((- *mapsize* 1), (- *mapsize* 1))  
;;     . any position can be start_position or goal_position  
;;  
;;   - obstacle area can be given by user  
;;  
;;   - boundary (out side of valid position)  
;;  
;;   - one of 8 neighbors and vehicle position  
;;     which has the lowest evaluation function value  
;;  
;;   - CONTROL MODE  
;;     . main  
;;     . detour  
;;  
;; note :  
;;  
;;   - this program is implemented on SIMBOLICS 3675  
;;  
;;   - to run program, enter "(ok-search)"  
;;
```

```

;; - then you can read
;;   "Do you want to take the same map as last e.g, y or off : "
;;   if you want same obstacles,goal,start-position as last run
;;   then type "y"
;;   otherwise type "n"
;; - after displaying "Now, loading data ! please wait a minute"
;;   the map will be displayed on the color monitor located by SYM1
;;
;; - follow the displaying instructions
;;
;; - to erase map on the color monitor, type "(kill)"
;;   but if you want same obstacle,goal and start-position,
;;   then don't use this commander
;;
;; - this program uses two two-dimensional array :
;;   . inner-map array : 80 * 80, contains the distance
;;     from each position to goal
;;     whenever the robot pass some position.
;;     it will be added by the cost to move vehicle
;;
;;   . physical-map array : 80 * 80.
;;     contains each position's height value
;;     obstacle area contains 'obstacle'
;;
;; - the area which the robot can see is limited  $12.5 \cdot (\sqrt{2})$  meters
;;   and it can move 8 directions.
;;   so it is possible to move to one of the nearest
;;     N,NW,W,SW,S,SE,E and NE

```

```

*****

```

```

.....

```

```

:: DEFINE VARIABLES

```

```

.....

```

```

(DEFVAR "green-window")
(DEFVAR "green-window-array")
(DEFVAR "green-window-width")
(DEFVAR "green-window-height")
(DEFVAR "green-window-position")
(DEFVAR "green-window-screen")
(DEFVAR "green-window-pos")

```

```

(DEFVAR *main-screen*)
(DEFVAR *screen-alu*)
(DEFVAR *start-alu*)
(DEFVAR *goal-alu*)
(DEFVAR *black-alu*)
(DEFVAR *path1-alu*)
(DEFVAR *path2-alu*)
(DEFVAR *path11-alu*)
(DEFVAR *path21-alu*)
(DEFVAR *obstacle-alu*)
(DEFVAR *level0-alu*)
(DEFVAR *level1-alu*)
(DEFVAR *level2-alu*)
(DEFVAR *level3-alu*)
(DEFVAR *level4-alu*)
(DEFVAR *level5-alu*)
(DEFVAR *level6-alu*)
(DEFVAR *level7-alu*)
(DEFVAR *level8-alu*)
(DEFVAR *level9-alu*)
(DEFVAR *level10-alu*)
(DEFVAR *level11-alu*)
(DEFVAR *level12-alu*)
(DEFVAR *level13-alu*)
(DEFVAR *level14-alu*)
(DEFVAR *level15-alu*)
(DEFVAR *level16-alu*)
(DEFVAR *level17-alu*)
(DEFVAR *level18-alu*)
(DEFVAR *level19-alu*)
(DEFVAR *level20-alu*)
(DEFVAR *level21-alu*)
(DEFVAR *level22-alu*)

```

```

(DEFVAR *scaled-x*)
(DEFVAR *scaled-y*)
(DEFVAR *xcoord*)
(DEFVAR *ycoord*)
(DEFVAR *x-coord*)
(DEFVAR *y-coord*)
(DEFVAR *answer*)
(DEFVAR *height*)
(DEFVAR *org-height*)
(DEFVAR *goal*)
(DEFVAR *path-list*)
(DEFVAR *first-time*)
(DEFVAR *inner-map*)
(DEFVAR *physical-map*)

```

```

(DEFVAR *x*)
(DEFVAR *y*)
(DEFVAR *old-x*)
(DEFVAR *old-y*)
(DEFVAR *old-position*)
(DEFVAR *old-old-position*)
(DEFVAR *old-direction*)
(DEFVAR *old-old-direction*)
(DEFVAR *x-start*)
(DEFVAR *y-start*)
(DEFVAR *mapsize*)
(DEFVAR *scale*)
(DEFVAR *robot*)
(DEFVAR *input-stream*)
(DEFVAR *start-position*)
(DEFVAR *if-turn-cost*)
(DEFVAR *mode*)
(DEFVAR *nearest-distance-before*)
(DEFVAR *obstacle-mark*)
(DEFVAR *old-obstacle-mark*)
(DEFVAR *total-E* 0)
(DEFVAR *slope*)

```

```

(DEFVAR *sign*)
(DEFVAR *smallest-E*)
(DEFVAR *D*)
(DEFVAR *valid*)
(DEFVAR *last-p*)
(DEFVAR *dir*)
(DEFVAR *temp-queue*)
(DEFVAR *new-queue*)
(DEFVAR *last-evaluation-cost*)
(DEFVAR *evaluation-cost*)
(DEFVAR *method*)
(DEFVAR *current-position*)
(DEFVAR *new-position*)
(DEFVAR *new-dir*)
(DEFVAR *time*)
(DEFVAR *max-open-nodes*)
(DEFVAR *node-num*)
(DEFVAR *min-function-path*)
(DEFVAR *smallest-object*)
(DEFVAR *close-list*)
(DEFVAR *passed-before*)

```

```

.....

```

```

:: DEFINE WINDOW AND COLORS

```

```

.....

```

```

(DEFFLAVOR my-color-flavor()
  (tv:window
   tv:graphics-mixin))

(DEFUN make-window
  (&rest options &key (superior (color:find-color-screen :create-p t))
   &allow-other-keys)
  (apply #'tv:make-window 'my-color-flavor
   :blinker-p nil
   :borders 2
   :save-bits t
   :expose-p t
   :label nil
   :name "Green Window"
   :position (list *x-start* *y-start*)
   ;;upper left position of window
   :inside-width (* *mapsize* *scale*)
   ;;multiply num of pixel by size of pixel
   :inside-height (* *mapsize* *scale*)
   :superior superior
   options))

(DEFUN make-green-window ()
  (SETF *green-window* (make-window))
  (SETF *screen-alu* (SEND color:color-screen
    :compute-color-alu
    tv:alu-seta 0 0 0))
  (SEND *green-window* :set-erase-aluf *screen-alu*)
  (SEND *green-window* :refresh))

(DEFUN create-green-window()
  (SETF *main-screen* (SEND *terminal-io* :superior))
  (make-green-window)
  (SETF *green-window-pos*
    (SEND *green-window* :position))
  (SETF *green-window-screen*
    (SEND *green-window* :screen))
  (init-my-colors)
  'done-init-green-window)

(DEFUN kill ()
  (SEND *green-window* :kill)
  'killed)

```

```

(DEFUN init-my-colors()
  ;; defines colors

  (SETF *start-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 0.0 1.0 1.0))
  (SETF *goal-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 0.0 0.5 1.0))
  (SETF *path1-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 0 0 0))
  (SETF *path11-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 1.0 1.0 1.0))
  (SETF *path2-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 1.0 0 0))
  (SETF *path21-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 0 0 1.0))
  (SETF *black-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 0 0 0))
  (SETF *obstacle-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 1 0 0))
  (SETF *level0-alu* (SEND *green-window-screen*
    :compute-color-alu color:alu-x 1.0 1.0 1.0))
  (SETF *level1-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 238 255) (/ 255 255) (/ 230 255)))
  (SETF *level2-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 207 255) (/ 225 255) (/ 176 255)))
  (SETF *level3-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 186 255) (/ 240 255) (/ 142 255)))
  (SETF *level4-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 160 255) (/ 220 255) (/ 150 255)))
  (SETF *level5-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 142 255) (/ 217 255) (/ 97 255)))
  (SETF *level6-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 138 255) (/ 182 255) (/ 74 255)))
  (SETF *level7-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 220 255) (/ 200 255) (/ 30 255)))
  (SETF *level8-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 230 255) (/ 170 255) (/ 50 255)))
  (SETF *level9-alu* (SEND *green-window-screen*
    :compute-color-alu
    color:alu-x (/ 255 255) (/ 150 255) (/ 68 255)))

```

```

((= *height* 113)
 (box scale scale xcoord ycoord *level11-alu*))
((= *height* 114)
 (box scale scale xcoord ycoord *level12-alu*))
((= *height* 115)
 (box scale scale xcoord ycoord *level13-alu*))
((= *height* 116)
 (box scale scale xcoord ycoord *level14-alu*))
((= *height* 117)
 (box scale scale xcoord ycoord *level15-alu*))
((= *height* 118)
 (box scale scale xcoord ycoord *level16-alu*))
((= *height* 119)
 (box scale scale xcoord ycoord *level17-alu*))
((= *height* 120)
 (box scale scale xcoord ycoord *level18-alu*))
((> *height* 120)
 (box scale scale xcoord ycoord *level19-alu*))))))
(CLOSE *input-stream*))

```

```

(DEFUN display-height-again (mapsize scale)
  ;; used for drawing the map
  ;; which has same obstacles,goal,start-position as last map
  ;; put height-value into physical-map
  ;; display map on the color monitor

```

```

(DO ((y 0 (+ y 1)))
  ((= y mapsize))
  (DO ((x 0 (+ x 1)))
    ((= x mapsize))
    (SETQ *height* (AREF *physical-map* x y))
    (SETQ *x-coord* (* x scale))
    (SETQ *y-coord* (* y scale))
    (COND
      ((EQUAL *height* 1020)
       (box scale scale *x-coord* *y-coord* *level0-alu*))
      ((EQUAL *height* 1030)
       (box scale scale *x-coord* *y-coord* *level1-alu*))
      ((EQUAL *height* 1040)
       (box scale scale *x-coord* *y-coord* *level2-alu*))
      ((EQUAL *height* 1050)
       (box scale scale *x-coord* *y-coord* *level3-alu*))
      ((EQUAL *height* 1060)
       (box scale scale *x-coord* *y-coord* *level4-alu*))
      ((EQUAL *height* 1070)
       (box scale scale *x-coord* *y-coord* *level5-alu*))
      ((EQUAL *height* 1080)
       (box scale scale *x-coord* *y-coord* *level6-alu*))
      ((EQUAL *height* 1090)
       (box scale scale *x-coord* *y-coord* *level7-alu*))
    )
  )
)

```

```

((EQUAL *height* 1100)
 (box scale scale *x-coord* *y-coord* *level8-alu*))
((EQUAL *height* 1110)
 (box scale scale *x-coord* *y-coord* *level9-alu*))
((EQUAL *height* 1120)
 (box scale scale *x-coord* *y-coord* *level10-alu*))
((EQUAL *height* 1130)
 (box scale scale *x-coord* *y-coord* *level11-alu*))
((EQUAL *height* 1140)
 (box scale scale *x-coord* *y-coord* *level12-alu*))
((EQUAL *height* 1150)
 (box scale scale *x-coord* *y-coord* *level13-alu*))
((EQUAL *height* 1160)
 (box scale scale *x-coord* *y-coord* *level14-alu*))
((EQUAL *height* 1170)
 (box scale scale *x-coord* *y-coord* *level15-alu*))
((EQUAL *height* 1180)
 (box scale scale *x-coord* *y-coord* *level16-alu*))
((EQUAL *height* 1190)
 (box scale scale *x-coord* *y-coord* *level17-alu*))
((EQUAL *height* 1200)
 (box scale scale *x-coord* *y-coord* *level18-alu*))
((EQUAL *height* 1210)
 (box scale scale *x-coord* *y-coord* *level19-alu*))
((EQUAL *height* 'obstacle)
 (draw-obstacle *x-coord* *y-coord* *scale* *obstacle-alu*))
)))

```

```

(DEFFLAVOR my-mouse ((m-x 0) (m-y 0) (m-b 0))
 ()
 :initable-instance-variables)

```

```

(DEFMETHOD (read-mouse my-mouse)
 ()
 (tv:mouse-wait)
 (SETF m-x sys:mouse-x)
 (SETF m-y sys:mouse-y)
 (SETF m-b tv:mouse-last-buttons)
 (IF (EQUAL m-b 4)
 (pop-up-my-menu self)
 (LIST m-x m-y m-b)))

```

```

(DEFMETHOD (pop-up-my-menu my-mouse)
 :: if user pressed on the right botton of mouse,
 :: then this menu will be dispalyed

```

```

()
(PRINT my-mouse)
(LET ((my-menu
      (tv:make-window 'tv:momentary-menu
        ':superior *green-window*
        ':label '(:string "Selection")
        ':item-list
          '(("Set Goal Loc" :value 42)
            ("Set Start Loc" :value 43)
            ("End of Operation" :value 40))))))
  (LIST m-x m-y (SEND my-menu :choose))))

```

```

(DEFVAR my-mouse (make-instance 'my-mouse))

```

```

(DEFFLAVOR plotter ((vertex-list nil) (pv-x nil) (pv-y nil) (start-p))
  (basic-plotter)
  :initable-instance-variables
  :readable-instance-variables)

```

```

(DEFFLAVOR basic-plotter((start-x 0) (start-y 0)
  (end-x 0) (end-y 0))
  ()
  :initable-instance-variables)

```

```

(DEFVAR my-plotter (make-instance 'plotter))

```

```

(DEFMETHOD (drawing plotter)
  :: read inputs from mouse
  :: middle botton of mouse —> obstacles
  :: right botton of mouse —> menu

```

```

  ()

```

```

(LET* ((m-v (read-mouse my-mouse))
      (x (first m-v))
      (y (second m-v))
      (f (third m-v)))

```

```

  (cond ((EQUAL f 40)
    :: end of operation
    t)

```

```

    ((EQUAL f 42)
    :: set goal location
    (draw-goal x y) nil)

```

```

((EQUAL f 43)
 ;; set start-position location
 (draw-start x y) nil)

((AND (EQUAL f 2) (NOT (EQUAL pv-x x)) (NOT (EQUAL pv-y y)))
 ;; draw obstacles and set obstacle in *physical-map*
 (SETF *scaled-x* ( (DIV x *scale*) *scale*))
 (SETF *scaled-y* ( (DIV y *scale*) *scale*))
 (draw-obstacle *scaled-x* *scaled-y* *scale* *obstacle-alu*)
 ;;(box *scale* *scale* *scaled-x* *scaled-y* *obstacle-alu*)
 (SETF (AREF *physical-map* (zl:/ *scaled-x* *scale*)
 (zl:/ *scaled-y* *scale*))
 'obstacle)
 nil)
 (t nil))))

```

```

(DEFUN draw-obstacle (x y scale color)
 (box scale scale x y color))

```

```

(DEFUN draw-goal (x y)
 ;; draw goal position on the given map position
 (SETF *goal* ((DIV x *scale*) (DIV y *scale*)))
 (SETQ *x* (+ (* (x-position *goal*) *scale*) (DIV *scale* 2)))
 (SETQ *y* (+ (* (y-position *goal*) *scale*) (DIV *scale* 2)))
 (SEND *green-window* :draw-filled-in-circle *x* *y*
 10 *goal-alu*)
 (LET ((lx (- *x* 5))
 (ly (+ *y* 7)))
 (SEND *green-window* :draw-string "G"
 lx ly (+ 1 lx) ly t '(:fix italic :large)
 *black-alu*)))

```

```

(DEFUN draw-start (x y)
 ;; draw start position mark on the given map position
 (SETF *start-position* ((DIV x *scale*) (DIV y *scale*)))
 (SETQ *x* (+ (* (x-position *start-position*) *scale*)
 (DIV *scale* 2)))
 (SETQ *y* (+ (* (y-position *start-position*) *scale*)
 (DIV *scale* 2)))
 (SEND *green-window* :draw-filled-in-circle *x* *y*
 10 *start-alu*)
 (LET ((lx (- *x* 5))
 (ly (+ *y* 7)))
 (SEND *green-window* :draw-string "S"
 lx ly (+ 1 lx) ly t '(:fix italic :large)
 *black-alu*)))

```

```
.....
;; PATH-PLANNING
.....
```

```
(DEFUN get-map-information ()
```

```
  ;; (get-mapsize)
  (SETQ *mapsize* 80)
  ;; map size is 80 * 80 pixels

  ;; (get-x-y-start-point)
  (SETQ *x-start* 170)
  (SETQ *y-start* 0)

  ;; (get-scale)
  (SETQ *scale* 12))
  ;; each pixel size is 12 * 12
```

```
(DEFUN build-array ()
```

```
  ;; create inner-map and physical-map
```

```
  (SETF *inner-map* (MAKE-ARRAY '(,*mapsize* ,*mapsize*)))
  ;; declare *inner-map* array
```

```
  (SETF *physical-map* (MAKE-ARRAY '(,*mapsize* ,*mapsize*)))
  ;; declare *physical-map* array
```

```
(DEFUN draw-map ()
```

```
  ;; create green window
  ;; and draw the map on the screen
  ;; determine goal,start-position and obstacle-area
```

```
  (create-green-window)
  (FORMAT T "~%Now, data is loading ! please wait a minute")
  (display-height *mapsize* "terrain.data" *scale*)
  (explain-how-to-use-mouse)
  (tv:mouse-set-sheet *green-window*)
  (do ()
    ((drawing my-plotter)
     'done-drawing))
  (tv:mouse-set-sheet *main-screen*))
```

```

(DEFUN get-ready-to-run ()
  (initial *goal*)
  (SETF *total-E* 0)
  (SETQ *old-direction* 100)
  (SETQ *old-old-direction* 100)
  ;; for first step movement, to set turn-cost to 0
  (SETQ *nearest-distance-before*
    (AREF *inner-map*
      (x-position *start-position*)
      (y-position *start-position*)))
  (SETQ *old-position* *start-position*)
  (SETF *mode* 'main)
  (FORMAT T "~% ** select vehicle type. tank-type --> 1 ***")
  (FORMAT T "~% ** jeep-type --> 2 ***")
  (SETQ *robot* (READ))
  (FORMAT T "~% ** apply turn-cost. yes --> on ***")
  (FORMAT T "~% ** no --> off ***")
  (SETQ *if-turn-cost* (READ)))

```

```

(DEFUN ok-search ()
  ;; the highest level function of this program

```

```

  (select-map)
  ;; select old map or new map
  (get-ready-to-run)
  (path-plan-search)

:   (SETF *method* 'A)
:   (print "improved A* search")
:   (imp-A*-search)
:   (SETF *method* 'B)
:   (A*-search)
:   (path-plan-search)

  (dribble)
  'done)

```

```

.....
(DEFUN path-plan-search ()
  (initial *goal*)
  (SETF (AREF *inner-map*
    (x-position *start-position*)
    (y-position *start-position*)))
  (+ (AREF *inner-map*
    (x-position *start-position*)
    (y-position *start-position*)))
    12.5))

```

```

(setf *time* (time
  (DO ((q-element '(0 100 ,*start-position*)
        (expand q-element)))
    ((EQUAL (LAST (CAR (LAST q-element))) *goal*)
      (setf *path-list* q-element))
    (determine-mode (LAST (CAR (LAST q-element))))))
  (print-result)))

```

```

(DEFUN print-result ()
  (print *time*)
  (setf ok-list *path-list*)
  (draw-best-path (CAR (LAST *path-list*)) *level0-alu*)
  (print "DISTANCE FROM START TO GOAL =")
  (prin1 (distance
    (x-position *start-position*)
    (y-position *start-position*) *goal*)))
  (print "D= ")
  (PRIN1 (setf *D*
    (- (CAR *path-list*)
      (distance (x-position *start-position*)
        (y-position *start-position*)
        *goal*)))))

```

```

(DEFUN expand (q-element)
  (if (EQUAL *mode* 'main)
    (add-next-position q-element)
    (add-next-position1 q-element)))

```

```

(DEFUN add-next-position (q-element)
  (SETF *dir* (CAR (CDR q-element)))
  (SETF *current-position* (LIST (CAR (LAST (CAR (LAST q-element))))))
  (SETF *new-position* (best-next-position *dir* *current-position*))
  (path-mark *current-position* *new-position* *new-dir*)
  '(.(+ (CAR q-element)
    (local-cost-function *dir* *current-position* *new-position*))
    *new-dir*
    .(append (CAR (LAST q-element)) *new-position*)))

```

```

(DEFUN add-next-position1 (q-element)
  (SETF *dir* (CAR (CDR q-element)))
  (SETF *current-position* (LIST (CAR (LAST (CAR (LAST q-element))))))
  (SETF *new-position* (best-next-position1 *dir* *current-position*))
  (path-mark *current-position* *new-position* *new-dir*)
  '(.(+ (CAR q-element)
    (local-cost-function *dir* *current-position* *new-position*))
    *new-dir*
    .(append (CAR (LAST q-element)) *new-position*)))

```

```

(DEFUN path-mark (p new-p dir)
  (SETF (AREF *inner-map* (x-position new-p) (y-position new-p))
    (+ (AREF *inner-map* (x-position new-p) (y-position new-p))
      (local-cost-function dir p new-p)))
  (SETF *total-E* (+ *total-E* (local-cost-function dir p new-p))))

```

```

(DEFUN best-next-position (old-direction p)
  ;; return next p
  ;; whose cost is the min of 8 candidate ps' evaluation p cost

  (SETF *smallest-object* (/ (smallest-object p old-direction) 1))
  (COND
    ((EQUAL
      (/ (evaluation-function old-direction p (w-position p)) 1)
      *smallest-object*)
      (SETF *new-dir* 6)
      (w-position p))
    ((EQUAL
      (/ (evaluation-function old-direction p (n-position p)) 1)
      *smallest-object*)
      (SETF *new-dir* 0)
      (n-position p))
    ((EQUAL
      (/ (evaluation-function old-direction p (s-position p)) 1)
      *smallest-object*)
      (SETF *new-dir* 4)
      (s-position p))
    ((EQUAL
      (/ (evaluation-function old-direction p (e-position p)) 1)
      *smallest-object*)
      (SETF *new-dir* 2)
      (e-position p))
    ((EQUAL
      (/ (evaluation-function old-direction p (nw-position p)) 1)
      *smallest-object*)
      (SETF *new-dir* 7)
      (nw-position p))
    ((EQUAL
      (/ (evaluation-function old-direction p (ne-position p)) 1)
      *smallest-object*)
      (SETF *new-dir* 1)
      (ne-position p))
    ((EQUAL
      (/ (evaluation-function old-direction p (sw-position p)) 1)
      *smallest-object*)
      (SETF *new-dir* 5)
      (sw-position p))
    ((EQUAL
      (/ (evaluation-function old-direction p (se-position p)) 1)
      *smallest-object*)

```

```

      (SETF *new-dir* 3)
      (se-position p))))
(DEFUN best-next-position1 (old-direction p)
  ; this is applied if *mode* is 'detour
  ; to select next p
  (COND
    ((EQUAL (/ (evaluation-function1 old-direction p (w-position p)) 1)
      (/ (smallest-eval p old-direction) 1))
      (SETQ *new-dir* 6)
      (get-obs-mark (w-position p))
      (w-position p))
    ((EQUAL (/ (evaluation-function1 old-direction p (n-position p)) 1)
      (/ (smallest-eval p old-direction) 1))
      (SETQ *new-dir* 0)
      (get-obs-mark (n-position p))
      (n-position p))
    ((EQUAL (/ (evaluation-function1 old-direction p (s-position p)) 1)
      (/ (smallest-eval p old-direction) 1))
      (SETQ *new-dir* 4)
      (get-obs-mark (s-position p))
      (s-position p))
    ((EQUAL (/ (evaluation-function1 old-direction p (e-position p)) 1)
      (/ (smallest-eval p old-direction) 1))
      (SETQ *new-dir* 2)
      (get-obs-mark (e-position p))
      (e-position p))
    ((EQUAL (/ (evaluation-function1 old-direction p (nw-position p)) 1)
      (/ (smallest-eval p old-direction) 1))
      (SETQ *new-dir* 7)
      (get-obs-mark (nw-position p))
      (nw-position p))
    ((EQUAL (/ (evaluation-function1 old-direction p (ne-position p)) 1)
      (/ (smallest-eval p old-direction) 1))
      (SETQ *new-dir* 1)
      (get-obs-mark (ne-position p))
      (ne-position p))
    ((EQUAL (/ (evaluation-function1 old-direction p (sw-position p)) 1)
      (/ (smallest-eval p old-direction) 1))
      (SETQ *new-dir* 5)
      (get-obs-mark (sw-position p))
      (sw-position p))
    ((EQUAL (/ (evaluation-function1 old-direction p (se-position p)) 1)
      (/ (smallest-eval p old-direction) 1))
      (SETQ *new-dir* 3)
      (get-obs-mark (se-position p))
      (se-position p))))

```

```

(DEFUN smallest-object (p old-direction)
  ;; get the minimum the cost of
  ;; 4 candidate position evaluation funtion value

  (MIN (evaluation-function old-direction p (w-position p))
        (evaluation-function old-direction p (e-position p))
        (evaluation-function old-direction p (n-position p))
        (evaluation-function old-direction p (s-position p))
        (evaluation-function old-direction p (nw-position p))
        (evaluation-function old-direction p (ne-position p))
        (evaluation-function old-direction p (sw-position p))
        (evaluation-function old-direction p (se-position p))))

(DEFUN smallest-eval (p old-direction)

  (MIN (evaluation-function1 old-direction p (w-position p))
        (evaluation-function1 old-direction p (e-position p))
        (evaluation-function1 old-direction p (n-position p))
        (evaluation-function1 old-direction p (s-position p))
        (evaluation-function1 old-direction p (nw-position p))
        (evaluation-function1 old-direction p (ne-position p))
        (evaluation-function1 old-direction p (sw-position p))
        (evaluation-function1 old-direction p (se-position p))))

(DEFUN evaluation-function (old-direction p new-p )
  (+ (local-cost-function old-direction p new-p)
      (estimation-function old-direction p new-p)))

(DEFUN estimation-function (dir p new-p)
  : for main MODE
  (COND ((EQUAL (watch p new-p) 'obstacle) 100000)
        (T (+ (get-exp-turn dir
                        (x-position new-p) (y-position new-p)
                        (x-position *goal*) (y-position *goal*))
                (AREF *inner-map*
                      (x-position new-p) (y-position new-p))))))

(DEFUN evaluation-function1 (dir p new-p )
  (COND ((OR (NOT (next-by-obstacle new-p))
              (NOT (next-to-new-obs-mark new-p))
              (EQUAL new-p p)) 10000)
        (T (rotational-cost dir p new-p))))

(DEFUN local-cost-function (direction p new-p)
  (+ (transitional-cost p new-p)
      (rotational-cost direction p new-p )))

```


```
(DEFUN imp-A*-search ()
  ;for improved A* search
  (initial *goal*)
  (SETF *close-list* 'nil)
  (SETF *max-open-nodes* 0)
  (setq *time* (time
    (Do ((queue (start-list) (next-step queue)))
      ((EQUAL (arrived-goal queue) 'T)
        (SETF *path-list* (CAR (LAST queue))))
      (setf queue (remove-node queue))))))
  (print *time*)(PRIN1 "MAX-OPEN-NODES")
  (PRINT *max-open-nodes*)
  (draw-best-path (caddr *path-list*) *level0-alu*))
```

```
(DEFUN A*-search ()
  ;for A* search
  (initial *goal*)
  (SETF *close-list* 'nil)
  (SETF *max-open-nodes* 0)
  (setq *time* (time
    (Do ((queue (start-list) (next-step queue)))
      ((EQUAL (arrived-goal queue) 'T)
        (SETF *path-list* (CAR (LAST queue))))
      (setf queue (remove-node1 queue))))))
  (print *time*)(PRIN1 "MAX-OPEN-NODES")
  (PRINT *max-open-nodes*)
  (draw-best-path (CADDR *path-list*) *level0-alu*))
```

```
(DEFUN start-list ()
  (LIST '(
    ,(distance (x-position *start-position*)
      (y-position *start-position*) *goal*)
    100
    ,(distance (x-position *start-position*)
      (y-position *start-position*) *goal*)
    ,*start-position*)))
```

```
(DEFUN num-of-elem (queue)
  (DO ((temp queue (CDR temp))
    (sum 0 (1+ sum)))
    ((atom temp) sum)))
```

```

(DEFUN passed-before (p path-list)
  (SETF *passed-before* 'nil)
  (DO ((temp path-list (CDR temp)))
    ((OR (EQUAL temp 'nil) *passed-before*))
    (IF (EQUAL (CAR p) (CAR temp)) (SETF *passed-before* 'T)))
  *passed-before*)

```

```

(DEFUN arrived-goal (queue)
  (sort queue #'> :key #'car)
  (EQUAL (LAST (CAR (LAST (CAR (LAST queue))))) *goal*))

```

```

(DEFUN draw-step (old-p new-p color)
  ;; draw lines path
  ;; to find the middle point of pixel, (DIV *scale* 2) is required

  (SETQ *old-x* (+ (* (car old-p) *scale*) (div *scale* 2)))
  (SETQ *old-y* (+ (* (car (cdr old-p)) *scale*) (div *scale* 2)))
  (SETQ *x* (+ (* (car new-p) *scale*) (div *scale* 2)))
  (SETQ *y* (+ (* (car (cdr new-p)) *scale*) (div *scale* 2)))
  (SEND *green-window* :draw-line *old-x* *old-y* *x* *y* color))

```

```

(DEFUN draw-best-path (path color)
  (DO ((p1 path (CDR p1)))
    ((EQUAL (CDR p1) 'nil))
    (draw-step (car p1) (car (cdr p1)) color)))

```

```

(DEFUN remove-node (queue)
  ; for improved A* search
  (setf *temp-queue* '())
  (setf *smallest-E* (car (car (last queue))))
  (setf *sign* 'nil)
  (DO ((temp queue (CDR temp)))
    ((OR (null temp) (EQUAL *sign* 'T)))
    (COND ((<= (- (car (CAR temp)) *smallest-E*) *D*)
      (setf *temp-queue* temp)
      (setf *sign* 'T))))
  (setf *new-queue* '())
  (DO ((temp1 *temp-queue* (CDR temp1)))
    ((null temp1)
      (IF (EQUAL (good-node (CDR temp1) (car temp1)) 'T)
        (setf *new-queue*
          (append *new-queue* (list (car temp1))))))
    (SETF *node-num* (num-of-elem *new-queue*))
    (IF (> *node-num* *max-open-nodes*)
      (SETF *max-open-nodes* *node-num*))
    *new-queue*))

```

```

(DEFUN remove-node1 (queue)
  (setf *smallest-E* (car (car (last queue))))
  (setf *sign* 'nil)
  (setf *new-queue* '())
  (DO ((temp1 queue (CDR temp1)))
    ((null temp1))
    (IF (EQUAL (good-node (CDR temp1) (car temp1)) 'T)
      (setf *new-queue*
        (append *new-queue* (list (car temp1))))))
  (SETF *node-num* (num-of-elem *new-queue*))
  (IF (> *node-num* *max-open-nodes*)
    (SETF *max-open-nodes* *node-num*))
  *new-queue*)

```

```

(DEFUN h (queue)
  (DO ((temp queue (CDR temp)))
    ((null temp))
    (print (car (car temp)))
    (princ " " " ")
    (princ (last (car (last (car temp)))))))

```

```

(DEFUN draw-ok (color)
  (DO ((temp *new-queue* (CDR temp)))
    ((null temp))
    (draw-point (x-position (last (car (last (car temp)))))
      (y-position (last (car (last (car temp)))))
      color))
  (SEND *green-window* : draw-filled-in-circle
    (* (x-position
      (last(car(last (car (last *new-queue*))))))
      *scale*)
    (* (y-position
      (last(car(last (car (last *new-queue*))))))
      *scale*)
    7 *obstacle-alu*))

```

```

(DEFUN draw-point (x y color)
  :: draw goal position mark on the given map position

```

```

  (SEND *green-window*
    : draw-filled-in-circle (* x *scale*) (* y *scale*)
    3 color))

```

```

(DEFUN good-node (queue q-element)
  (setf *valid* 'T)
  (setf *last-p* (last (car (last q-element))))
  (DO ((temp queue (CDR temp)))
    ((OR (null temp) (EQUAL *valid* 'nil)))

```

```

      (cond ((EQUAL *last-p* (last (car (last (car temp))))))
            (setf *valid* 'nil) )))
*valid*)

```

```

(DEFUN next-step (queue)
  (SETF *min-function-path* (CAR (LAST queue)))
  (SETF *temp-queue* (DELETE *min-function-path* queue))
  (SETF *close-list*
    (APPEND (LAST (CAR (LAST *min-function-path*))) *close-list*))
  (go-neighbors *min-function-path*))

```

```

(DEFUN select-smaller (x y)
  (IF (> x y) y x))

```

```

(DEFUN get-exp-turn (dir x y x1 y1)
  (COND ((< x x1)
    (COND ((< y y1)
      (COND ((< (- x1 x) (- y1 y))
        (select-smaller (get-turn dir 0) (get-turn dir 1)))
        ((> (- x1 x) (- y1 y))
        (select-smaller (get-turn dir 1) (get-turn dir 2)))
        (T (get-turn dir 1))))
      ((> y y1)
        (COND ((< (- x1 x) (- y y1))
          (select-smaller (get-turn dir 3) (get-turn dir 4)))
          ((> (- x1 x) (- y y1))
          (select-smaller (get-turn dir 2) (get-turn dir 3)))
          (T (get-turn dir 3))))
      (T (get-turn dir 2))))
    ((> x x1)
      (COND ((< y y1)
        (COND ((< (- x x1) (- y1 y))
          (select-smaller (get-turn dir 7) (get-turn dir 0)))
          ((> (- x x1) (- y1 y))
          (select-smaller (get-turn dir 6) (get-turn dir 7)))
          (T (get-turn dir 7))))
        ((> y y1)
          (COND ((< (- x x1) (- y y1))
            (select-smaller (get-turn dir 4) (get-turn dir 5)))
            ((> (- x x1) (- y y1))
            (select-smaller (get-turn dir 5) (get-turn dir 6)))
            (T (get-turn dir 5))))
          (T (get-turn dir 6))))
      (T 0)))

```

99

```

(UNLESS (passed-before (s-position *last-p*) *close-list*)
  (SETF *temp-queue* (append *temp-queue*
    (LIST (LIST (+ (CAR q-element)
      (- *last-evaluation-cost*)
      (SETF *evaluation-cost*
        (estimation-function *dir* *last-p*
          (s-position *last-p*)))
        (local-cost-function (car (cdr q-element))
          *last-p* (s-position *last-p*)))
      4 *evaluation-cost*
      (append (car (last q-element)) (s-position *last-p*)))))))
(UNLESS (passed-before (sw-position *last-p*) *close-list*)
  (SETF *temp-queue* (append *temp-queue*
    (LIST (LIST (+ (CAR q-element)
      (- *last-evaluation-cost*)
      (SETF *evaluation-cost*
        (estimation-function *dir* *last-p*
          (sw-position *last-p*)))
        (local-cost-function (car (cdr q-element))
          *last-p* (sw-position *last-p*)))
      5 *evaluation-cost*
      (append (car (last q-element)) (sw-position *last-p*)))))))
(UNLESS (passed-before (w-position *last-p*) *close-list*)
  (SETF *temp-queue* (append *temp-queue*
    (LIST (LIST (+ (CAR q-element)
      (- *last-evaluation-cost*)
      (SETF *evaluation-cost*
        (estimation-function *dir* *last-p*
          (w-position *last-p*)))
        (local-cost-function (car (cdr q-element))
          *last-p* (w-position *last-p*)))
      6 *evaluation-cost*
      (append (car (last q-element)) (w-position *last-p*)))))))
(UNLESS (passed-before (nw-position *last-p*) *close-list*)
  (SETF *temp-queue* (append *temp-queue*
    (LIST (LIST (+ (CAR q-element)
      (- *last-evaluation-cost*)
      (SETF *evaluation-cost*
        (estimation-function *dir* *last-p*
          (nw-position *last-p*)))
        (local-cost-function (car (cdr q-element))
          *last-p* (nw-position *last-p*)))
      7 *evaluation-cost*
      (append (car (last q-element)) (nw-position *last-p*)))))))
*temp-queue* )

```

```

(DEFUN num-of-steps (path-list)
  (COND ((EQUAL (CDR path-list) nil) 0)
    (T (+ 1 (num-of-steps (CDR path-list))))))

```

```
(DEFUN X-Position (position)
  ;; get X from a specific position
  (FIRST (FIRST position)))
```

```
(DEFUN Y-Position (position)
  ;; get Y from a specific position
  (FIRST (CDAR position)))
```

```
(DEFUN initial (goal)
  ;; assign distance from *goal* to each position
  ;; to *inner-map* array
  (DO ((X 0 (+ X 1))) ;from 0 to mapsize-1
      ((= X *mapsize*)
       ;call function 'sub-initial' mapsize times
       (sub-initial X goal)))
```

```
(DEFUN sub-initial (X goal)
  ;; subfunction of initial
  ;; assign distance from *goal* to each position
  ;; to *inner-map* array
  (DO ((Y 0 (+ Y 1))) ;from 0 to mapsize-1
      ((= Y *mapsize*)
       (assign-distance X Y goal)))
```

```
(DEFUN sensor (P)
  (AREF *physical-map* (x-position p) (y-position p)))
```

```
(DEFUN watch (p new-p)
  ;; this checks implicit and explicit obstacle
  (COND ((EQUAL (slope-coefficient p new-p) 10000)
         (SETF (AREF *physical-map* (x-position new-p)
                       (y-position new-p))
               'obstacle))
        (T (AREF *physical-map* (x-position new-p)
                          (y-position new-p)))))
```

```
(DEFUN get-height-distance (p new-p)
  (/ (- (sensor new-p) (sensor p)) 2))
```

```
(DEFUN div (dividend divisor)
  (truncate (/ dividend divisor)))
```

```
(DEFUN get-slope (p new-p)
  ;; get slope between position and new-position
  ;;
  ;;
  ;;      slope-rate(ft/m)
  ;; diff height(ft)  12.5 m  12.5*(SQRT 2) m
  ;;
  ;;      15      1.2      0.85
  ;;      10      0.8      0.57
  ;;      5       0.4      0.28
  ;;      0       0.0      0.0
  ;;     -5      -0.4     -0.28
  ;;     -10     -0.8     -0.57
  ;;     -15     -1.2     -0.85
  ;;
  ;; you can see return value of each case as follows
```

```
(IF (EQUAL p new-p) 0
    (IF (EQUAL (sensor p) 'obstacle) 0
        (/ (get-height-distance p new-p)
            (distance (x-position p) (y-position p) new-p)))))
```

```
(DEFUN slope-coefficient (p new-p)
  (IF (EQUAL (sensor new-p) 'obstacle) 10000
      (SETF *slope* (get-slope p new-p))
      (COND ((EQUAL *robot* 2) ;jeep-type
              (COND ((> *slope* 0.6) 10000)
                    ((< *slope* -0.6) 10000)
                    ((> *slope* 0.5) 1.9)
                    ((> *slope* 0.3) 1.6)
                    ((> *slope* 0.2) 1.3)
                    ((>= *slope* 0) 1.0)
                    ((> *slope* -0.3) 0.8)
                    ((> *slope* -0.5) 1.2)
                    ((> *slope* -0.6) 1.5)))
              ((EQUAL *robot* 1) ;tank-type
              (COND ((> *slope* 0.9) 10000)
                    ((< *slope* -0.9) 10000)
                    ((> *slope* 0.6) 2.2)
                    ((> *slope* 0.5) 1.9)
                    ((> *slope* 0.3) 1.6)
                    ((> *slope* 0.2) 1.3)
                    ((>= *slope* 0) 1.0)
                    ((> *slope* -0.3) 0.8)
                    ((> *slope* -0.5) 1.2)
                    ((> *slope* -0.6) 1.5)
                    ((> *slope* -0.9) 2.0)))))))
```

```
(DEFUN assign-distance (X Y position)
  ;; assign distance from the given position
  ;; to position (X Y) to the *inner-map* array

  (SETF (AREF *inner-map* X Y) (distance X Y position)))
```

```
(DEFUN distance (X Y position)
  ;; get distance from the given position to position (X Y)

  (SQRT (+ (* (dis-X-from-given-position X position)
              (dis-X-from-given-position X position))
          (* (dis-Y-from-given-position Y position)
              (dis-Y-from-given-position Y position)))))
```

```
(DEFUN dis-X-from-given-position (X position)
  ;; get X-distance from the given position
  ;; X-distance = ABS(position's X-value - X)

  (* (ABS (- (X-position position) X)) 12.5))
```

```
(DEFUN dis-Y-from-given-position (Y position)
  ;; get Y-distance from the given position
  ;; Y-distance = ABS(position's Y-value - Y)

  (* (ABS (- (Y-position position) Y)) 12.5))
```

```
(DEFUN e-position (position)
  ;; get the new position that is located on the
  ;; east-side of given position

  (COND ((/= (x-position position) (- *mapsize* 1))
        (SETQ position '((- (+ (X-position position) 1)
                               (Y-position position)))))
        (T (SETQ position position))))
```

```
(DEFUN w-position (position)
  ;; get the new position that is located on the
  ;; west-side of given position

  (COND ((/= (x-position position) 0)
        (SETQ position '((- (X-position position) 1)
                               (Y-position position)))))
        (T (SETQ position position))))
```

```

(DEFUN n-position (position)
  ;; get the new position that is located on the
  ;; north-side of given position

  (COND ((/= (y-position position) (- *mapsize* 1))
    (SETQ position '((X-position position)
      ,(+ (Y-position position) 1))))
    (T (SETQ position position))))

(DEFUN s-position (position)
  ;; get the new position that is connecting directly and located on the
  ;; south-side of given position

  (COND ((/= (y-position position) 0)
    (SETQ position '((X-position position)
      ,(- (Y-position position) 1))))
    (T (SETQ position position))))

(DEFUN ne-position (position)
  ;; get the new position that is located on the
  ;; northeast-side of given position

  (COND ((AND (/= (x-position position) (- *mapsize* 1))
    (/= (y-position position) (- *mapsize* 1)))
    (SETQ position '((X-position position) 1)
      ,(+ (Y-position position) 1))))
    (T (SETQ position position))))

(DEFUN nw-position (position)
  ;; get the new position that is located on the
  ;; northwest-side of given position

  (COND ((AND (/= (x-position position) 0)
    (/= (y-position position) (- *mapsize* 1)))
    (SETQ position '((- (X-position position) 1)
      ,(+ (Y-position position) 1))))
    (T (SETQ position position))))

(DEFUN se-position (position)
  ;; get the new position that is located on the
  ;; southeast-side of given position

  (COND ((AND (/= (x-position position) (- *mapsize* 1))
    (/= (y-position position) 0))
    (SETQ position '((X-position position) 1)
      ,(- (Y-position position) 1))))
    (T (SETQ position position))))

```

```

(DEFUN sw-position (position)
  ;; get the new position that is located on the
  ;; southwest-side of given position

  (COND ((AND (/= (x-position position) 0)
                (/= (y-position position) 0))
        (SETQ position '((- (X-position position) 1)
                          ,(- (Y-position position) 1))))
        (T (SETQ position position))))

(DEFUN transitional-cost (position new-position)
  ;; returns the cost to move from position to new-position
  ;; cost =
  ;; slope-coefficient * distance between position and new-position

  (* (slope-coefficient position new-position)
     (distance (x-position new-position)
               (y-position new-position) position)))

(DEFUN get-direction (p new-p)
  (COND
    ((EQUAL (x-position p) (x-position new-p))
     (COND ((EQUAL (y-position p) (y-position new-p)) 100)
           ((EQUAL (+ (y-position p) 1) (y-position new-p)) 0)
           ((EQUAL (- (y-position p) 1) (y-position new-p)) 4))))
    ((EQUAL (y-position p) (y-position new-p))
     (COND ((EQUAL (+ (x-position p) 1) (x-position new-p)) 2)
           ((EQUAL (- (x-position p) 1) (x-position new-p)) 6))))
    ((EQUAL (+ (x-position p) 1) (x-position new-p))
     (COND ((EQUAL (+ (y-position p) 1) (y-position new-p)) 1)
           ((EQUAL (- (y-position p) 1) (y-position new-p)) 3))))
    ((EQUAL (- (x-position p) 1) (x-position new-p))
     (COND ((EQUAL (+ (y-position p) 1) (y-position new-p)) 7)
           ((EQUAL (- (y-position p) 1) (y-position new-p)) 5))))))

(DEFUN rotational-cost (old-direction p new-p)
  ;; determine direction from p to new-p
  ;; return turn cost to eval position function
  (get-turn old-direction (get-direction p new-p)))

(DEFUN get-turn (old-direction new-direction)
  ;; return turning-cost as follow

```

```
;;
```

turning-angle	cost(ft)	
	jeep-type	tank-type
0	0	0
45	1	2
90	3	5
135	7	10
180	10	13

```
;;
```

```
(COND ((OR (EQUAL *if-turn-cost* 'on) (EQUAL *mode* 'detour))
```

```
  (COND
```

```
    ((EQUAL *robot* 2)
```

```
      (COND
```

```
        ((= (ABS (- new-direction old-direction)) 0) 0)
        ((= (ABS (- new-direction old-direction)) 1) 1)
        ((= (ABS (- new-direction old-direction)) 2) 3)
        ((= (ABS (- new-direction old-direction)) 3) 7)
        ((= (ABS (- new-direction old-direction)) 4) 10)
        ((= (ABS (- new-direction old-direction)) 5) 7)
        ((= (ABS (- new-direction old-direction)) 6) 3)
        ((= (ABS (- new-direction old-direction)) 7) 1)
        ((> (ABS (- new-direction old-direction)) 7) 0)))
```

```
        ;; for the first movement
```

```
      ((EQUAL *robot* 1)
```

```
        (COND
```

```
          ((= (ABS (- new-direction old-direction)) 0) 0)
          ((= (ABS (- new-direction old-direction)) 1) 2)
          ((= (ABS (- new-direction old-direction)) 2) 5)
          ((= (ABS (- new-direction old-direction)) 3) 10)
          ((= (ABS (- new-direction old-direction)) 4) 13)
          ((= (ABS (- new-direction old-direction)) 5) 10)
          ((= (ABS (- new-direction old-direction)) 6) 5)
          ((= (ABS (- new-direction old-direction)) 7) 2)
          ((> (ABS (- new-direction old-direction)) 7) 0))))
```

```
          ;; for the first movement
```

```
    (T 0)))
```

```
(DEFUN next-by-obstacle (position)
```

```
  ;; if robot is next by obstacle and robot is not on obstacle,
```

```
  ;; then return true
```

```
  (AND (OR (EQUAL (watch position (w-position position)) 'obstacle)
```

```
    (EQUAL (watch position (e-position position)) 'obstacle)
```

```
    (EQUAL (watch position (n-position position)) 'obstacle)
```

```
    (EQUAL (watch position (s-position position)) 'obstacle))
```

```
    (NOT (EQUAL (watch position position) 'obstacle))))
```

```

(DEFUN went-backward (position old-position)
  ;; if new-position is far from goal than old-position's,
  ;; then return true

  (< (distance (x-position position) (y-position position) *goal*)
    (distance (x-position old-position)
              (y-position old-position) *goal*)))

```

```

(DEFUN determine-mode (position)

  (COND
    ((AND (next-by-obstacle position)
          (<= *nearest-distance-before*
              (distance (x-position position)
                        (y-position position) *goal*)))
      (IF (NOT (EQUAL *mode* 'detour))
          (SETQ *old-obstacle-mark*
                (get-first-old-obstacle-position position)))
          (SETQ *mode* 'detour) t)

    (T (SETQ *mode* 'main)
        (SETQ *nearest-distance-before*
              (distance (x-position position)
                        (y-position position) *goal*))
        nil)))

```

```

(DEFUN next-to-obs-mark1 (position)
  ;; if position is an obstacle and is next by old-obstacle-position,
  ;; then return true

  (COND
    ((EQUAL (sensor position) 'obstacle)
      (COND ((EQUAL position (w-position *obstacle-mark*)) t)
            ((EQUAL position (e-position *obstacle-mark*)) t)
            ((EQUAL position (n-position *obstacle-mark*)) t)
            ((EQUAL position (s-position *obstacle-mark*)) t)
            ((EQUAL position *obstacle-mark*) t)
            (T nil))))
    (T nil)))

```

```

(DEFUN next-to-obs-mark2 (position)

  (COND ((next-to-obs-mark1 position) t)
        ((OR (next-to-obs-mark1 (w-position position))
              (next-to-obs-mark1 (e-position position))
              (next-to-obs-mark1 (n-position position))
              (next-to-obs-mark1 (s-position position))) t)
        (T nil)))

```

```

(DEFUN next-to-obs-mark3 (position)
  ;; if position is an obstacle
  ;; and next by obstacle which is next by old obstacle,
  ;; then return true

  (COND ((EQUAL (sensor position) 'obstacle)
    (COND ((next-to-obs-mark1 position) t)
      ((AND (EQUAL (watch position (w-position position))
        'obstacle)
        (next-to-obs-mark2 (w-position position))) t)
      ((AND (EQUAL (watch position (e-position position))
        'obstacle)
        (next-to-obs-mark2 (e-position position))) t)
      ((AND (EQUAL (watch position (n-position position))
        'obstacle)
        (next-to-obs-mark2 (n-position position))) t)
      ((AND (EQUAL (watch position (s-position position))
        'obstacle)
        (next-to-obs-mark2 (s-position position))) t)
      (T nil)))
    (T nil)))

```

```

(DEFUN next-to-obs-mark4 (p)
  ;; if position is an obstacle
  ;; and next by obstacle which is next by old obstacle,
  ;; then return true

  (COND ((EQUAL (sensor p) 'obstacle)
    (COND ((next-to-obs-mark1 p) t)
      ((AND (EQUAL (watch p (w-position p)) 'obstacle)
        (next-to-obs-mark3 (w-position p))) t)
      ((AND (EQUAL (watch p (e-position p)) 'obstacle)
        (next-to-obs-mark3 (e-position p))) t)
      ((AND (EQUAL (watch p (n-position p)) 'obstacle)
        (next-to-obs-mark3 (n-position p))) t)
      ((AND (EQUAL (watch p (s-position p)) 'obstacle)
        (next-to-obs-mark3 (s-position p))) t)
      (T nil)))
    (T nil)))

```

```

(DEFUN next-to-new-obs-mark (position)
  ;; if position is next by obstacle which is next by obstacle
  ;; which is next by old obstacle,
  ;; then return true
  (COND ((AND (EQUAL (watch position (w-position position))
    'obstacle)
    (next-to-obs-mark4 (w-position position))) t)
    ((AND (EQUAL (watch position (e-position position))
    'obstacle)
    (next-to-obs-mark4 (e-position position))) t)

```

```

((AND (EQUAL (watch position (n-position position))
              'obstacle)
      (next-to-obs-mark4 (n-position position))) t)
((AND (EQUAL (watch position (s-position position))
              'obstacle)
      (next-to-obs-mark4 (s-position position))) t)
(T nil)))

```

```

(DEFUN get-obs-mark (position)
  ;; from present position, get best old(longest) obstacle

```

```

(COND
  ((EQUAL (/ (eval-dis-from-obs-mark (e-position position)) 1)
    (/ (max-dis-from-obs-mark position) 1))
    (SETQ *old-obstacle-mark* *obstacle-mark*)
    (SETQ *obstacle-mark* (e-position position)))
  ((EQUAL (/ (eval-dis-from-obs-mark (w-position position)) 1)
    (/ (max-dis-from-obs-mark position) 1))
    (SETQ *old-obstacle-mark* *obstacle-mark*)
    (SETQ *obstacle-mark* (w-position position)))
  ((EQUAL (/ (eval-dis-from-obs-mark (n-position position)) 1)
    (/ (max-dis-from-obs-mark position) 1))
    (SETQ *old-obstacle-mark* *obstacle-mark*)
    (SETQ *obstacle-mark* (n-position position)))
  ((EQUAL (/ (eval-dis-from-obs-mark (s-position position)) 1)
    (/ (max-dis-from-obs-mark position) 1))
    (SETQ *old-obstacle-mark* *obstacle-mark*)
    (SETQ *obstacle-mark* (s-position position)))))

```

```

(DEFUN max-dis-from-obs-mark (position)
  ;; get max distance from old obstacle to possible new old obstacle
  (MAX (eval-dis-from-obs-mark (e-position position))
        (eval-dis-from-obs-mark (w-position position))
        (eval-dis-from-obs-mark (n-position position))
        (eval-dis-from-obs-mark (s-position position))))

```

```

(DEFUN eval-dis-from-obs-mark (p)
  ;; get distance from old obstacle to position
  ;; -----
  ;;   position case      return
  ;; -----
  ;; . impossible to connect
  ;;   to old obstacle      0
  ;; . not obstacle         0
  ;; . old obstacle         5
  ;; . else                 distance from
  ;;                       old obstacle to position
  ;; -----

```

```

(COND ((NOT (next-to-obs-mark4 p)) 0)
      ((NOT (EQUAL (sensor p) 'obstacle)) 0)
      ((EQUAL p *obstacle-mark*) 5)
      (T
       (distance (x-position p) (y-position p) *obstacle-mark*))))

```

```

(DEFUN get-first-old-obstacle-position (position)
  ;; when mode is changed to 'obstacle-path,
  ;; the obstacle which is next position and nearest to goal

```

```

(COND
  ((EQUAL (/ (eval-best-dis-to-goal position (e-position position)) 1)
           (/ (best-dis-to-goal position) 1))
   (SETQ *obstacle-mark* (e-position position)))
  ((EQUAL (/ (eval-best-dis-to-goal position (w-position position)) 1)
           (/ (best-dis-to-goal position) 1))
   (SETQ *obstacle-mark* (w-position position)))
  ((EQUAL (/ (eval-best-dis-to-goal position (n-position position)) 1)
           (/ (best-dis-to-goal position) 1))
   (SETQ *obstacle-mark* (n-position position)))
  ((EQUAL (/ (eval-best-dis-to-goal position (s-position position)) 1)
           (/ (best-dis-to-goal position) 1))
   (SETQ *obstacle-mark* (s-position position))))

```

```

(DEFUN best-dis-to-goal (position)
  ;; get min distance from four neighbors of position to goal

```

```

(MIN (eval-best-dis-to-goal position (e-position position))
      (eval-best-dis-to-goal position (w-position position))
      (eval-best-dis-to-goal position (n-position position))
      (eval-best-dis-to-goal position (s-position position)))

```

```

(DEFUN eval-best-dis-to-goal (p new-p)
  ;; return distance from position to goal
  ;; but position is not an obstacle, then return 10000
  (COND ((NOT (EQUAL (watch p new-p) 'obstacle)) 10000)
        (T (distance (x-position new-p) (y-position new-p) *goal*))))

```

```

(DEFUN get-mapsize ()
  (FORMAT T "~%Enter a mapsize e.g, 80 and RETURN-key : ")
  (SETQ *mapsize* (READ)))

```

```

(DEFUN get-x-y-start-point ()
  (FORMAT T "~%Enter a left upper point of window e.g, 0 0 : ")
  (SETQ *x-start* (READ))
  (SETQ *y-start* (READ)))

```

```

(DEFUN get-scale ()
  (FORMAT T "~%Enter a virtual pixel size e.g, 5 : ")
  (SETQ *scale* (READ)))

(DEFUN select-map ()
  ;; determines whether the user use the old map or not

  (FORMAT T "~%Do you want to take the same map as last e.g, y or n : ")
  (SETQ *answer* (READ))
  (COND ((EQUAL *answer* 'y)
    (kill)
    (create-green-window)
    (initial *goal*)
    (display-height-again *mapsize* *scale*)
    (SETQ *old-position* (LAST *path-list*))
    (draw-goal (* (x-position *goal*) *scale*)
      (* (y-position *goal*) *scale*))
    (draw-start (* (x-position *start-position*) *scale*)
      (* (y-position *start-position*) *scale*)))
    (T (get-ready-to-run))))

(DEFUN explain-how-to-use-mouse ()

  (FORMAT T "~% ")
  (FORMAT T "~% to put obstacles,")
  (FORMAT T "~% 1. move mouse on the place where obstacle will be")
  (FORMAT T "~% 2. press the middle button of mouse")
  (FORMAT T "~% to select goal and start-position")
  (FORMAT T "~% 3. press the right button of mouse")
  (FORMAT T "~% 4. move mouse to SET GOAL LOC to set goal or")
  (FORMAT T "~% SET START LOC to set start-position")
  (FORMAT T "~% 5. press the right button of mouse"))

```

LIST OF REFERENCES

1. Hart, P., Nilsson, N. J., and Raphael, B., "A Formal Basis for The Heuristic determination of Minimum Cost Paths," *IEEE Trans. Sys. Sci. Cyber.*, v. SSC-4(2), pp. 100-107, 1968.
2. Nilsson, N. J., "A Mobile Automation: An Application of Artificial Intelligence Techniques," *Proc. 1st Int. Joint Conf. AI*, pp. 509-520, May 1969.
3. Lozano-Pérez, T., and Wesley, M. A., "An Algorithm for Planning Collision-free Paths among Polyhedral Obstacles," *Comm. v. ACM*-22(10), pp. 560-570, 1979.
4. Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach," *IEEE Trans. Computers*, v. C-32(2), pp. 108-119, 1983.
5. Brooks, R. A., "Solving The Find-path Problem by Good Representation of Free Space," *IEEE Trans. Sys. Man Cyber.*, v. SMC-13, pp. 190-197, 1983.
6. Moravec, H. P., "Visual Mapping by Robot Rover," *Proc. 6th Int. Joint Conf. AI*, pp. 598-600, August 1979.
7. Moravec, H. P., *Robot Rover Visual Navigation*, UMI Research Press, 1981.
8. Moravec, H. P., "Rover Visual Obstacle Avoidance," *Proc. 7th Int. Joint Conf. AI*, pp. 758-790, August 1981.
9. Giralt, G., Sobek, R., and Chatila, R., "A Multilevel Planning and Navigation System for A Mobile Robot," *Proc. 6th Int. Joint Conf. AI*, pp. 335-338, 1979.
10. Thompson, A. M., "The Navigation System of The JPL Robot," *Proc. 5th Int. Joint Conf. AI*, pp. 749-757, August 1977.
11. Wong, E. K., and Fu, K. S., "A Hierarchical Orthogonal Space Approach to Three-Dimensional Path Planning," *IEEE Journal of Robotics and Automation*, v. RA-2(1), pp. 42-53, 1986.
12. Crowley, J. L., "Navigation for an Intelligence Mobile Robot," *IEEE Journal of Robotics and Automation*, v. RA-1(1), pp. 31-41, 1985.

13. Oommen, B. J., Iyengar, S. S., Rao, S. V. N., and Kashyap, R. L., "Robot Navigation in Unknown Terrains Using Learned Visibility Graphs. Part I: The Disjoint Convex Obstacle Case," *IEEE Journal of Robotics and Automation*, v. RA-3(6), pp. 672-681, 1987.
14. Iyenger, S. S., Jorgensen, C. C., Rao, S. V. N., and Weisbin, C. R., "Learned Navigation Paths for a Robot in Unexplored Terrain," *IEEE Computer Society, The second conference on Artificial Intelligence Applications*, pp. 148-155, 1985.
15. Kuan, D. T., Brooks, R. A., Zamiska, J. C., and Das, M., "Automatic Path Planning for a Mobile Robot Using a Mixed representation of Free space," *IEEE Comp. Soc. Conf. on AI applications*. pp. 70-74, 1984.
16. MacLennan, B. J., *Principles of Programming Languages*, Ted Buchholz, 1897.
17. Documentation Group of Symbolics, Inc., *User's Guide to Symbolics Computers*, p. 3, CSA Press, 1986.
18. Rowe, N. C., *Artificial Intelligence Through Prolog*, Prentice-Hall, Inc., 1988.
19. Symon, K. R., *Mechanics*, 3rd ed., Addison-Wesley Publishing Co., 1971.
20. Nilsson, N. J., *Principles of Artificial Intelligence*, Tioga Publishing Co., 1980.

INITIAL DISTRIBUTION LIST

- | | | |
|----|--|---|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, VA 22304-6145 | 2 |
| 2. | Library, Code 0142
Naval Postgraduate School
Monterey, CA 93943-5002 | 2 |
| 3. | Department Chairman, Code 52
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943-5000 | 1 |
| 4. | Curriculum Office, Code 37
Computer Technology
Naval Postgraduate School
Monterey, CA 93943-5000 | 1 |
| 5. | Professor Se-Hung Kwak, Code 52Kw
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 | 5 |
| 6. | Professor Yuh-jeng Lee, Code 52Le
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 7. | Professor Robert B. McGhee, Code 52Mz
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 8. | Professor Neil C. Rowe, Code 52Rp
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 9. | Major Yong Goo Hwang
SMC 2120
Naval Postgraduate School
Monterey, CA 93943 | 1 |

- | | | |
|-----|---|---|
| 10. | Captain Jae Doo Jung
SMC 1504
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 11. | Major Myeong Hung Kwang
SMC 2418
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 12. | Captain Seong Sung Park
SMC 1888
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 13. | Captain Eun Seok Shin
SMC 2961
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 14. | Captain In Sub Shin
SMC 2986
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 15. | Captain Hung Taek Kim
SMC 1930
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| 16. | Tae Sik Yoon
Zip-code 132-062
Dobonggu beon 2 dong
43-7 ho 5 tong 3 ban
Seoul, Korea | 1 |
| 17. | Hae Sung Jang
Zip-code 302-181
ChungNam Daejeon Seogu Naedong
5 Beongi Jugong Apt. 209 dong 508 ho
Seoul, Korea | 1 |
| 18. | Captain Do Kyeong Ok
Zip-code 422-030
Geonggido Bucheon
Namgu Sangdong 245-17
Seoul, Korea | 7 |