A PATH PLANNING AND OBSTACLE AVOIDANCE ALGORITHM FOR AN AUTONOMOUS ROBOTIC VEHICLE

by

Sharayu Yogesh Ghangrekar

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2009

Approved by:

____
Dr. James M. Conrad

____
Dr. Bharatkumar Joshi

____
Dr. Ron Sass

ABSTRACT

SHARAYU YOGESH GHANGREKAR. A Path Planning and Obstacle Avoidance Algorithm for an Autonomous Robotic Vehicle. (Under the direction of Dr. James M. Conrad)

Path planning in robotics is concerned with developing the logic for navigation of a robot. Path planning still has a long way to go considering its deep impact on any robot's functionality. Various path planning techniques have been tried and tested earlier, including probabilistic, integral and genetic approaches. The implementation details of most of these algorithms are proprietary to specific organizations. The requirement of a customized strategy for collision free and concerted navigation of an All-Terrain Vehicle (ATV) led to the activities of this research. As a part of this research an algorithm has been developed and simulated to give a visual effect. The algorithm presented is evolutionary and capable of path planning for ATVs in the presence of completely known and newly-discovered obstacles. This algorithm helps the ATV to maneuver in an open field in a specific pattern and avoid the obstacles, if any, along its path. As part of the research the actual algorithm is implemented and simulated using C and WINAPI. As a result, given the data of known obstacles and the field, the ATV can maneuver in a systematic and optimum manner towards its goal by avoiding all the obstacles in its path. This algorithm can also be deployed on an ATV using real time data from LIDAR and GPS. The logic of the algorithm can be extended for path planning in a completely dynamic environment.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and thank my advisor, Dr. James M. Conrad for his explicit support and help throughout my Master's program all the way till the successful completion of my thesis. I thank him for the utmost confidence and encouragement that he provided in guiding me through my thesis. His advices and teachings will be definitely helpful to me in my future. I am also thankful to Dr. Bharatkumar Joshi and Dr. Ron Sass for accepting to be my committee members and also for their advice and support.

I would like to express my sincere appreciation towards Zapata Engineering for providing me the platform for my thesis. Also, I would like to appreciate the support from Malcolm Zapata, Thomas Meiswinkel and Richard McKinney and for their timely advice.

Last but not the least I would like to thank my husband Yogesh for his love and support and also the support from my parents and family for showing their profound confidence in me and their invaluable help towards the completion of this thesis.

Table of Contents

List of Figures

List of Equations

List of Tables

Chapter 1: Introduction

Autonomous robotics is one of the most key topics of this generation of research. It has a wide range of applications, such as construction, manufacturing, waste management, space exploration, and military transportation. One of the main areas of research, in order to achieve successful autonomous robots, is path planning. Path planning in robotics is defined as navigation that shall be collision free and most optimum for the autonomous vehicle to maneuver from a source to its destination. This thesis concentrates on building a path planning algorithm for an all terrain vehicle (ATV) used for travelling in an open field or forest. The novelty of this algorithm is that it does not simply create a path between a source to its destination, but it makes sure that the vehicle covers the entire field area when navigating from the source to its destination.

Consider a case where a person has to travel from room A to an adjacent room B, wherein he does not know the path from A to B beforehand. Starting at A, the person will have no knowledge at all about the directions to go to B. Initially, he understands the fact that he has to get out of the present room and then go to room B. For this he has to sense an exit from the present room. He uses his eyes to understand the location of the door for room A. His eyes and brain makes him understand that there is no object in his way to the door if he can simply go towards the door in straight direction. He then uses this information collected from his sensory organs to direct himself towards the door and hence towards room B. In other words, he uses his sensory organs (eyes) and control system (brain) to plan his path towards room B.

An autonomous robot is such a person who has no beforehand data about navigation and which needs some algorithm to be used for creating the directions for navigation. Path planning does something similar in case of autonomous robots with the use of electronic sensors and control system (algorithm). The field of robotic mapping addresses the problem of robot navigation where the use of GPS is not available or possible. It considers the ability of a robot to survey its surroundings, build a virtual map and move along an optimal path. The robot uses Laser Detection and Ranging (LIDAR), ultrasound and other sensors for data collection to understand its surroundings. This data is used to build a virtual map of its surroundings and build a path on the fly as the robot proceeds. This is referred to as Mapping. A robot that navigates using this map must be able to accurately calculate its position with respect to the landmarks in the map, and locate itself in this map. This is known as Localization. To maneuver along an optimal path both mapping and localization is necessary. This field is referred to as Simultaneous Localization and Mapping (SLAM) [1, 2].

Mapping starts from a point of zero data. As data is gathered new features are checked for repetition and then added or estimated accordingly. The basics of mapping is comprised of three parts: Data prediction (gather data using sensory inputs), data association (using some algorithm build an estimation of the environment) and map building (build a map using the earlier two steps which the robot can use for its trajectory). Mapping depicts the environment without any person physically measuring the whole area. In short, mapping gives us a virtual or visual representation of the actual environment which shall be used by the robot to do any of its prescribed action.

SLAM is a method which finds if it is possible for a mobile robot to be placed at an unknown location in an unknown environment, and for the robot to incrementally build a consistent map of the environment while simultaneously determining its location within this map. Over the years SLAM has been solved using various algorithms like Extended Kalman Filter (EKF) SLAM, Fast SLAM and Rao-Blackwellized. Most of these algorithms use various probabilistic and state space model based approaches. The need for a probabilistic solution arises because the data obtained from the sensors and from the movement of robot itself can be affected by noise. Furthermore the Gaussian and motion model are amongst few of the models used for robotic motion. Mapping and SLAM both require building a recursive solution, which would continually build a map from the point of start to the final destination. Mapping happens to be one of the integral parts for any autonomous vehicle.

For any ATV, path planning can only happen if the mapping and localization has been finished beforehand, i.e. only when the vehicle has complete knowledge of its surroundings and its own position in this surrounding, it can further plan its way towards the destination. Consider, for example, that a robot is positioned at a corner of a room and the room has only one door (See Figure 1-1). The Goal of the robot is to get out of the room autonomously without clashing with any of the walls. Mapping will give the robot some initial data of the surrounding environment. It will allow the robot to understand that it is positioned in some location totally blocked from all four sides surrounding it from a certain distance, with only one opening. With localization, it understands that, given the above circumstances, it is currently positioned at one of the corners diagonally opposite to the opening from where it has to exit. So we see that the entire journey of the vehicle from its source to destination is comprised of mapping, localization and then path planning. To reach the goal the robot will have to move towards east first then north and then east again to get out through the door. As seen in Figure 1-2, there could be multiple ways to reach the goal, i.e. the robot could go straight diagonally from the corner towards the door or first go north and then towards east. It is the path planning which decides upon these logistics and formulates an algorithm for the robots trajectory.

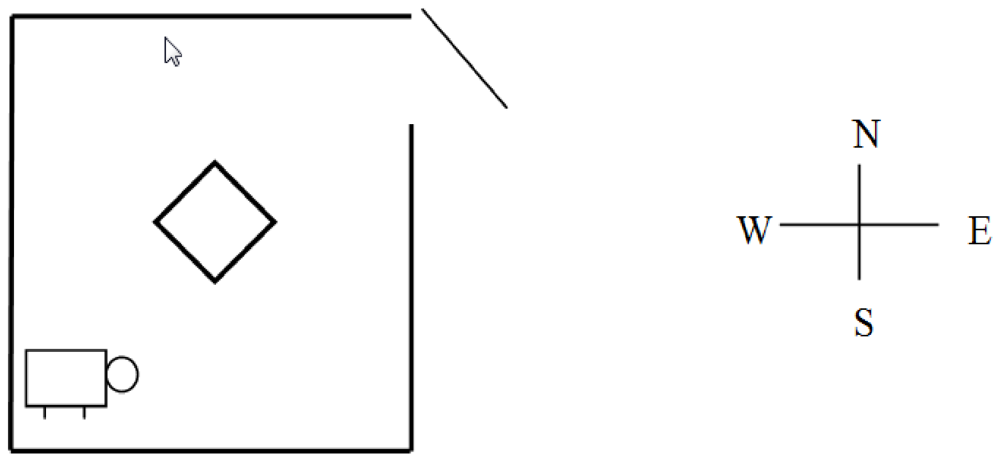


Figure 1-1 : Navigation of robot in a room

In the above example if the robot moves towards east but then does not move towards north, there is a probability of 0.5 that the robot moves north and 0.5 that the robot does not do so. Various such possibilities are shown in Figure 1-2. So for mapping the formulation includes probability distribution for every single state of the robot. In general this functionality is represented as d(s0,a0,s1) – probability of transitioning from state s0 to state s1 when action a0 is applied. For every single possible state its state value is calculated considering how fair or difficult it will be to go to the goal from this state.

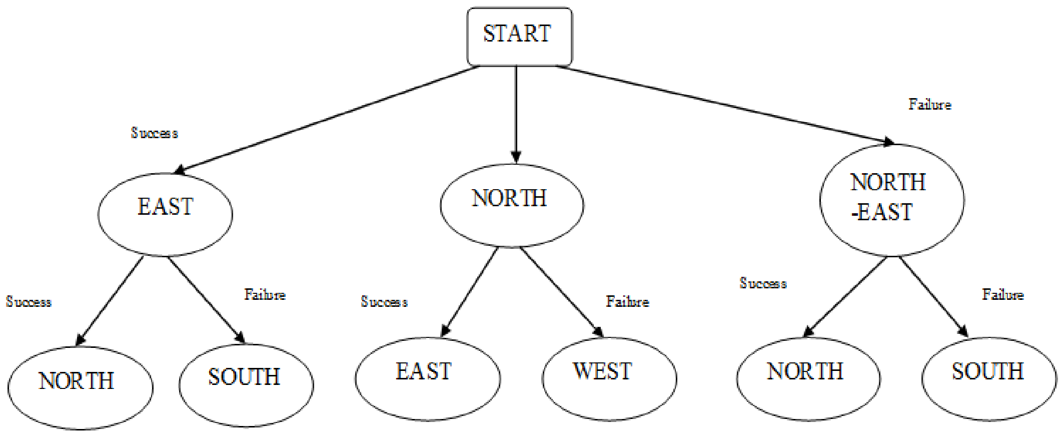


Figure 1-2 : Graphical representation of robot's movement plan

To make the planning robust, algorithms further take into consideration the undeterministic factors especially when in the outdoors using Gaussian noise distribution and Markov models. Hence a significant amount of computation and memory consumption is involved which eventually affects the frequency at which the robot can incorporate sensor data, which in turn implies accuracy.

Using the inputs from mapping and the localization, path planning algorithm further builds the logistics for the vehicle to follow a suitable path to reach its goal. Path planning not only assigns proper directions for the vehicle's trajectory but also handles the obstacles, if any, along its path. As per the algorithm presented in this research, an autonomous vehicle shall maneuver the entire field area in a predetermined fashion. Obstacles, if any, along the path are optimally avoided to resume back to the normal path. A base example would be as shown in Figure 1-3 and Figure 1-4:

Desired path for ATV, which includes an obstacle:

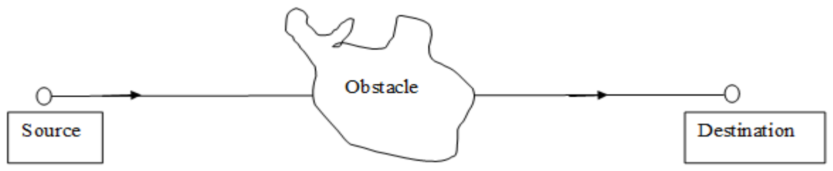


Figure 1-3 : Obstacle in the normal path

Path drawn from the algorithm:

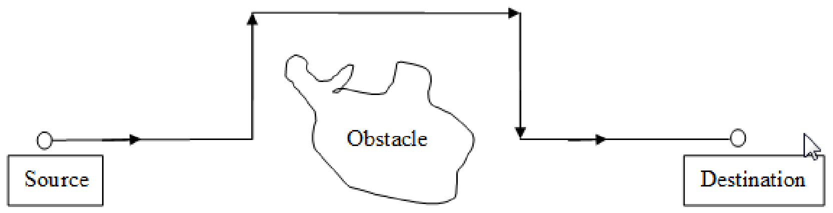


Figure 1-4 : Path around an obstacle

The above example shows the obstacle avoidance part of the algorithm. The algorithm also takes care that the ATV moves around the obstacle and covers the entire field area. Certain assumptions are taken into consideration while developing this algorithm. Typically, a field is represented using a grid. Each grid will be divided into points or nodes. Data of any one point, its surroundings and its goal point is used for planning the path. The algorithm explains how and why these points are created, and how they are used to create the logistics for path planning. In depth reasoning will be provided for every step of the algorithm from its outlining to the developmental stage. The algorithm is comprised of various sub routines such as local path navigation, obstacle detection, initialization which will be elaborated in the following chapters. This approach plans an initial global path or route based on known information and then modifies the plan locally as the robot discovers obstacles with its sensors. The process repeats until the robot reaches the goal or determines that it cannot. The programming is implemented using 'C' language. The scope of this research is limited to the development of a simulation based model of this algorithm.

1.1 Motivation

In July of 2002, nine miners in the Quecreek Mine in Sommerset, Pennsylvania were trapped underground for three and a half days after accidentally drilling into a nearby abandoned mine. A subsequent investigation attributed the cause of the accident to inaccurate maps [7]. Since the accident, mobile robots and SLAM have been investigated as a possible technology for acquiring accurate maps of abandoned mines.

(a) Undersea (b) Underground (c) Other planets

Figure 1-5 : Target Environments for Outdoor Robotic Mapping [7]

Over the years, the basic estimation problem in mapping and path planning is well understood. However, there are still a number of open problems to be addressed. These include computational complexity, linearization effects, association of measurements to features, detection of loops in the robot's path, and maintaining topological consistency as the maps get very large. Typically, for indoor environments, certain features are taken for granted, such as extensive planar regions. Most of these algorithms have been successful for indoor applications; however, for outdoors the results have not been very satisfactory. This is because of complex outdoor environments and dynamic situations as shown in Figure 1-5. Also, when outdoors there are more chances that that the environment will change over time.

This research concentrates on building a fairly optimal and robust path planning algorithm for maneuvering of autonomous vehicles in outdoor environment. It will be based on following main tasks:

- Get a clear picture as far as possible of the surroundings
- Localize the robot inside the map
- The path should be so that the vehicle covers the entire field area
- Algorithm built should be optimum enough for obstacle avoidance

As part of this research, data gathered using LIDAR and processed through filters will be used for building the path. The Open source software tool Dev C++ Integrated Development Environment is studied and worked upon for the entire development. The solution best suited for outdoor environments, especially with dynamically changing factors and unexpected landmarks is worked upon. The vehicle under consideration traverses in the outdoors and is prone to obstacles such as tree locations. The initial map and position built will be used for routing the robot through the open field autonomously. Various factors, such as optimum path planning, self localization of the vehicle in the field map within some predetermined time frame and number and size of the obstacles determine the time frame in which the robotic vehicle shall complete its navigation. In order to cover all possible extreme localization failures, the functionality will be tested by introducing random obstacle cases.

1.2 Current Work

This section addresses the current work that has been done in the field of mapping. The current work in path planning will be discussed in chapter two. The main purpose of robotic mapping is to make the robot's mobility independent of devices such as a GPS in areas especially like underwater and airborne environments. Sensors, lasers, and LIDARs are used for data collection and for map integration.

The paper "*Simultaneous Localization and Mapping [Part I and Part II]*" [1, 2] describes details of SLAM along with a few solution algorithms. Also, it focuses on the recursive Bayesian formula of the SLAM problem, in which the probability distributions or estimates of the absolute or relative locations of landmarks and vehicle pose are obtained.

The paper "*Probabilistic Mapping of an Environment by a Mobile Robot*" [3] analyzes basics of mapping techniques for mobile robots in an indoor environment using a probabilistic model, maximum likelihood estimation, and a two step algorithm based on positioning and mapping respectively.

The paper "*Robotic Mapping: A Survey*" [4] provides a comprehensive introduction to the field of robotic mapping with a focus on the indoor mapping. It describes and compares various probabilistic techniques as they are presently being applied to a vast array of mobile robot mapping problems.

The paper "*Fast SLAM: a Factored Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association*" [7] describes the Fast SLAM technique which samples the potential robotic paths instead of maintaining a parameterized distribution of solutions like the EKF.

A Considerable amount of research work has been done in the area of indoor robotic mapping. This data and research proves crucial for exploring the outdoor tasks.

Algorithms using probabilistic distribution [3], Bayesian approach [11], Gaussian elimination [4], state space matrices [2], and recursive Monte Carlo [10] sampling have been implemented. Most of these algorithms simply distinguish between the scanned and unscanned areas. Stachniss and Burgard investigated occupancy grids [6] computing entropy of each cell in the grid to determine the utility of scanning from certain location. To distinguish color with each new input of data, new histograms are updated to fit changing conditions.

Almost all of the algorithms explored implement a probabilistic model of mapping. Uncertainty and noise factors are the main reasons for this. Also these algorithms model the "uncertainty factor" in autonomous robots using probability theory. An alternative to represent the environment of a robot are coverage maps [6]. The coverage maps store in each cell a posterior about the coverage of that cell.

The basic principle underlying virtually every single successful mapping algorithm is the Bayes Rule:

$$p\,(x|d) = \eta\, p\,(d|x)\, p(x) \qquad [4]$$

As the size of the map increases, the system tends to go slower. The map built can be guaranteed to converge but is subjected to local maxima. Currently some of the map building methods are:

- Line superposition method [32]: Builds a segment oriented geometric map of environment. Speeds up the processing.
- RRT for occupancy grid [32]: Suitable for fusion of different sensors. Most common low level sensor based models of environment.
- Occupancy grid based mapping [32]: Sensor data, gathered from multiple points of view, and is combined by Bayesian approach to allow the incremental updating of occupancy grid. Works for indoor environments and is robust to noise.
- Feature extraction based on neural networks [32]: Based on "*Growing Neural Gas Algorithm*". Here number of neurons (units) and the topology of the network are changed during self organization process.
- Creating map from simple sharp sensors [32]: Sharp sensors are used for local navigation.
- 3D environment mapping [32]: Laser rangefinders are used. Horizontal rangefinder is used for 2D mapping and localization and vertical one for 3D environment information.

The path planning process could be run time or predetermined. The predetermined method builds the path before the next action is decided for the robot. In the run time method the path is built simultaneously along with the performance of any of the actions decided. For the outdoor environments the run time method proves to be more precise considering the impromptu conditions.

Even some minimal amount of noise in path planning can cause a considerably chaotic situation. The probabilistic approaches in mapping takes care for such noises and their counter effect on the robot.

1.3 Organization of Thesis

The thesis is organized into eight major chapters. Chapter 2 explains the concept of path planning and the requirement for this research. Chapter 3 describes the basic algorithm. Chapter 4 explains the reason behind every step of the algorithm and the assumptions upon which it is based. Chapter 5 includes the simulation details, tools used and the correlation between the algorithm and the animation. Chapter 6 tells about study heuristics, where the results and test cases are analyzed. Chapter 7 includes conclusion and future work.

Chapter 2: Path Planning Overview

Research in mobile robotics can be traced back to late 1940s, although most of the effort related to path planning is more recent and has been conducted during the 1980s. Thanks to such fields, such as artificial intelligence, mathematics, computer science and mechanical engineering that theoretical and practical understanding of some issues has received a major boost [13]. Planning refers to a preconceived scheme or method of acting or proceeding. In other words, we can say that it defines an operative intelligence. In this algorithm, path planning is done with respect to a mobile robot or an autonomous all terrain vehicles (ATV), in order to design or scheme its routing. This chapter elaborates the various aspects of path planning in robotics worked upon until now and the reason why this research was evolved.

As the field of mobile robotics diversifies, so does the scope of path planning. Most of the mobile robots are customized for specific operation. Industrial robots are generally customized as per the specific industry and the industrial functionality. Robots used in medical field are customized for the specific surgery or any similar medical operation. Rovers to be used for exploration on other planets are customized for data collection related jobs. Hence all such customizable robots can be classified to have two parts of development. One part deals with the overall development of the robot, while the other deals with structuring the development as per the required customization.

A robot is any machine that resembles a human and does mechanical routine tasks automatically upon a command. Since it is a machine, whether an ATV or a still positioned robot, its functionality depends completely upon the set of instructions given to it. These set of instructions defined for any particular robot, mobile or immobile, defines its intelligence. In other words, since the robot is a machine, we can customize it for any specific functionality that we want it to work for. Currently various customizable robots are present in a successfully working stage [14]. Some such robots are: Pioneer(the Chornobyl reconnaissance robot), Helpmate(an assistant for the elderly and infirm), The Brain Surgeon(a robot to help in surgery), HazBot(a mobile robot for hazardous materials), Dante(the Volcano Explorer), Urbie(the Urban Robot), Underwater Explorer(explores a sunken fishing fleet), Serpentine Visual Inspection Robot(a small, light weight visual inspector), Antarctica 2000 Big Signal(the Nomad Rover hunts for meteorites in Antarctica), Stardust(a mission to collect and return comet dust to Earth), Galileo(a journey to Jupiter) and many more planetary rovers. Mobile robot is another such category with a lot of consideration these days. In this category the robot is suppose to move from one place to another for some specific purpose. This purpose could be mowing a lawn or shifting objects from one room to other in a home or simply exploring an open field. Depending upon the purpose, its navigation details are decided or customized. Development of the navigation details creates the need of a path planning algorithm for such mobile robots or ATVs. This algorithm decides the logic or scheme for navigation.

Present Path Planning Methods

Until now various algorithms have been implemented for the present customizable robots. This section describes a few important algorithms or techniques currently used for path planning.

2.1 Quad tree Multiresolution

This technique is based on the A* search method [16]. A quad tree block is created with zeros representing free space and ones representing obstacles. A minimum cost path is found from the source to the goal. This cost depends on the actual distance travelled and also upon the clearance of the path from the obstacles. This is followed by finding a neighbor using the node expansion process. If any of the horizontal or vertical nodes is a grey node (non obstacle node), a leaf adjacent to this node being expanded is found. As a result, a list of nodes from the quad tree forms a set of paths from source to goal nodes. With this method the number of nodes searched is lower compared to that in the grid search method.

2.2 Evolutionary Algorithm

This algorithm uses an evolutionary navigator to unify the offline [global path] and online [collision avoidance] computation [17]. Nodes are classified as feasible or infeasible depending upon their proximity to any obstacle. An offline algorithm creates a global path from source to goal. Online algorithm creates a sub route in case of facing an obstacle. A chromosome is formed of an ordered list of path nodes. Each node of each chromosome has a feasibility and path cost associated with it. The path cost is the

Euclidian distance from the next node. The chromosomes are further used for crossover (of nodes), mutation (fine tuning co ordinates of nodes), insertion, deletion or swap.

2.3 Use of Laplace's Equation

This method uses Laplace's equation to constrain the generation of a potential function over regions of configuration space of an effector [18]. Use of Laplace equation helps to achieve computation on large parallel architectures. A harmonic function on a domain satisfies the min-max principle, hence creating local minima in regions impossible to traverse, if Laplace equation is imposed. If a function satisfies the Laplace's equation in some region, then any critical point of the function in the interior of that region must be a saddle point, since local extrema of the function are not possible. Every harmonic function satisfies four properties: Analyticity, Polar, and Admissibility and that every critical point must be an isolated saddle point. The neighborhood of a given obstacle has a potential of not only this obstacle but also of all other obstacles. Using superposition the gradient for each maze is calculated. Numerical solutions to Laplace's equation obtained from finite difference methods are well suited for tasks of finding solutions with arbitrary boundary conditions. This technique provides a fast method of creating paths in a robot configuration space.

2.4 Hierarchical Strategy

This technique focuses on finding a three dimensional solution (time being the third dimension) to avoid a collision with moving obstacles [19]. It is assumed that the obstacle cannot accelerate beyond a certain limit. A quad tree hierarchical representation is used for the three dimensional configuration. The 3D space and time is further subdivided into eight subspaces of equal sizes called cells. These cells are termed as vertex, edge, empty and full cells. Initially, the entire universe is treated as a single cell which is represented by an octree containing one node. Depending on violation of the four basic conditions the cell gets further subdivided. A control point (C-point) is used to create the final skeleton of the path. C point comprises of L(x and y) and T (time) points. The L point is assigned as per the nine possible locations around a square. Depending upon the appropriate velocity, the T point is assigned in the search stage. Greater the number of L points in a plane, more is the degree of control attained. The main search procedure uses a priority queue, where the T point component of a C point serves as that point's priority. Once an acceleration value is set it has to be maintained until next L point. This method may not work in a search space.

2.5 Numerical Potential Field Techniques

In this method an entire graph is searched to create the path [20]. This technique is based on the use of multi scale pyramids of bitmap arrays for representing both the robot's workspace and the used configuration space. This method avoids any pre computation otherwise required for creating a global path. This technique provides a solution with three degrees of freedom and two orders of magnitude faster than most of the previous methods. This approach incrementally builds a graph connecting the local minima of a potential function defined over the configuration space, and concurrently searching this graph until a goal configuration is attained. Using the bitmap configuration a workspace pyramid is built. Numerical potential fields are built in two steps: W-potentials (computed for a selected point in a robot) and W potentials at various control points that forms a C potential. Four path planning techniques are put forth in this research. The first one performs a best first search using C potential as cost function and hence is complete in resolution. The second one is based on the Monte Carlo method generating random motions until the local minima are obtained. The third method searches for ‘valleys’ in a C potential. This method is slower in speed and also with less degree of freedom compared to earlier two techniques. The fourth is a constrained motion technique based on ideals of earlier three. It uses the C potential (global minimal) and also the notion of a ‘valley’ while escaping encountered local minima. Experiments show that amongst all the four the random motion method has the best qualities and is highly parallelizable.

2.6 Use of Intersecting Convex Shapes

This technique is based on the Quine-McCluskey method of finding prime implicants in a logical expression [21]. It is used to isolate all of the largest, rectangular, and free convex areas in a specified environment. Convexity is identified with all the largest rectangular free areas. A graph is created with a node corresponding to each of such convex area. This method defines a convex area, as the one that is free of obstacles and has the property that any two points in that area can be joined by a straight line that lies entirely within that area. Each such rectangle is represented by a pair of binary strings each at most 2n + 1 bit long. This algorithm is similar to the Quine-McCluskey technique [l0]-[12] used to identify the prime implicants of a logical expression. Prime convex areas in which the source and destination points are located may be determined, and the graph may be traversed from the source node to the destination node using one of the varieties of techniques available. Since one node look ahead is used, path cost assignments cannot begin until the graph node path progresses at least to the third node. An exhaustive graph search for optimal path is performed using a backtracking procedure. With this technique aligned objects are added randomly. This, results into a graph with the number of nodes bounded by the limit based on number of objects with distinct edges. This method requires relatively small amount of database to be maintained.

2.7 Symbolic and Geometric Connectivity Graph

Two methods based on the A* search algorithm – symbolic and geometric are presented [22]. The symbolic system uses inference rules to analyze and classify spatial relationships within the connectivity graph. The geometric method builds an exact path using connectivity information. The output of symbolic method is a symbolic description of the planned route while the geometric method builds a simple list of coordinate positions. The connectivity graph has adjacency relationships with different regions of free space. Based on number of incident arcs nodes are classified in four different categories. The symbolic or heuristic method represents a different order of traversal amongst the obstacles, that is, the order and side on which the obstacles are traversed is different for each alternative. A tandem process is created between A* search and the inference engine. The geometric system employs a RLC database and a free-space graph. In the geometric approach, a funnel (sequence of vertices) is meant to grow from tile to tile. For a given tile sequence, the ultimate result from such processing is the shortest path within that sequence. Symbolic system is used to exploit resolution hierarchy. Compared to the geometric method the symbolic system is shown to give better speed performance.

2.8 Need of a new Technique

The techniques and algorithms stated above are only a few amongst the many others dedicated for path planning. However the research papers do not elaborate the implementation details for these techniques. The research currently undergoing (behind this algorithm) involves navigation of an ATV in an open field in a predetermined manner. Hence, along with obstacle avoidance, this algorithm has to also take care of following a specific route throughout its navigation. A certain amount of customization was hence required in this algorithm. This customization decided the particular manner in which the vehicle routing will occur. The logic for obstacle avoidance has also been created altogether new so as to match the customization. The requirements for this algorithm can be stated as:

a) A path planning strategy for an autonomous ATV to be used in an open field
b) The navigation of the ATV has to be so as to cover the entire area of the field
c) Obstacles if any (known or newly-discovered) should be avoided in a manner so as to avoid them and then continue on the predetermined navigation route

The reference papers do not describe the intricate details for the implementation of their algorithms into software. The details of logic development behind the algorithm were not found to be described in these papers. The reasoning for questions such as why any particular step is performed is the most important in such path planning techniques. Also, what is important is, understanding how a particular logic has been translated into the actual software.

This algorithm and research takes care of all such details. All the requirements stated in the above paragraph led to the development of this research. This research also describes in detail each and every step behind the logic of the algorithm. Development of simulation to get a visual effect of the algorithm is also a part of this research. As stated above this algorithm is developed and customized so as to be implemented on an autonomous ATV set to navigate in an open field.

This algorithm, developed in two dimensions, is flexible enough to be carried forward and molded as per varied field or ATV or obstacle dimensions.

Chapter 3: Algorithm

The algorithm presented in this chapter basically explains the logic used for path planning from source to destination.

Basic things that this algorithm takes care of are:

a) Build a virtual map of the field

b) Plan a path from source to destination considering known obstacles

c) Plan the path so as to cover the entire field area

d) Detect and avoid newly-discovered obstacles, in order to maintain the decided path

A LIDAR is used on the autonomous vehicle for obstacle detection. The LIDAR will give information, such as how far and at what degree the obstacle is located. This information will be used by the path planning algorithm to modify its path so as to avoid the obstacle and re route the vehicle’s path. For this software, the information related to LIDAR will be taken as input from a file.

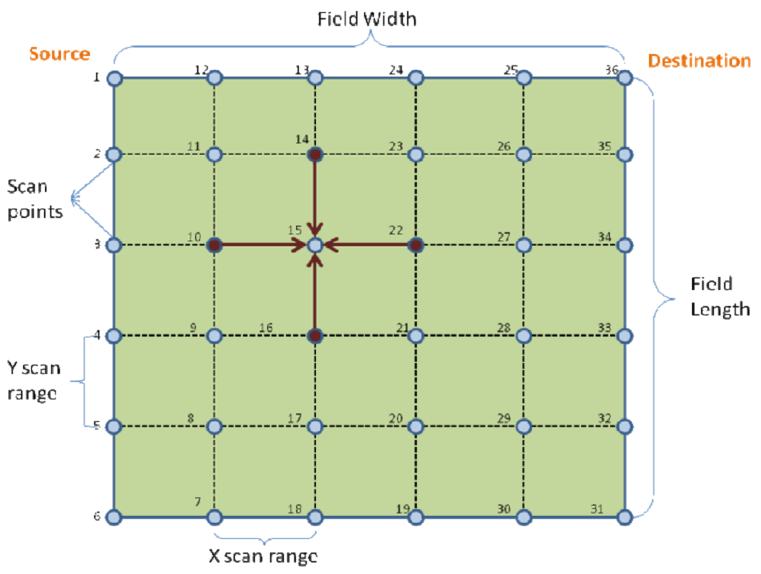


Figure 3-1 : Field Attributes

3.1 Terminology Used

1) Field: Any open space to be explored (forest or farm)
2) Field width: Distance of the field in X direction
3) Field length: Distance of the field in Y direction
4) Source: Start point for the vehicle's trajectory
5) Destination: end point for the vehicle's trajectory
6) Vehicle: Autonomous All Terrain Vehicles (ATV). Any vehicle equipped with LIDAR and other hardware required for computation of the mapping
7) Vehicle width: Maximum distance of the vehicle in direction perpendicular to LIDAR
8) LIDAR range: Maximum distance up to which the LIDAR can scan ahead of it as shown in Figure 3-2
9) Scan Area: 180 degree area in front of the LIDAR

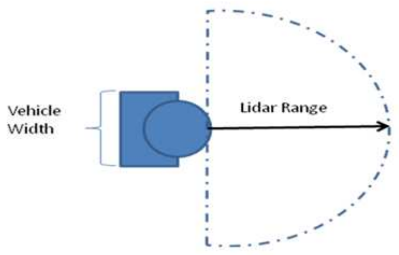


Figure 3-2 : LIDAR Attributes

10) Scan Point: Point on the field where the vehicle will scan the 180 degree area in front of it
11) Number of X scan points: Number of scan points along the width of the field (along the X direction) = Roundup (Field Width/Vehicle Width) +1
12) Number of Y scan points: Number of scan points along the length of the field (along the Y direction) = Roundup (Field Length / [Lidar Range/2] +1)
13) X Scan range: Distance between any two scan points along the X direction = Field Width / (Number of XScanPoints-1)
14) Y Scan range: Distance between any two scan points along the Y direction = FieldLength / (Number of YScanPoints)
15) Total Number of Scan Points = Number of X scan points * Number of Y scan points
16) Goal Point: Scan point with index number consecutively next to the current scan point. In the above example shown in Figure 3-1, scan point 2 is the goal point of 1 and 21 is the goal point of 20
17) Reach Points: All the scan points from where there is direct access (with a

single hop) to the current scan point are called the reach points of the current scan point. These are the scan points with a distance of plus or minus X scan range and Y scan range from the current scan point. Every scan point will have maximum of four points from where it can be reached. In Figure 3-1 10, 22, 14 and 16 are the reach points of 15.

18) Imaginary Square: A virtual square built around each known and newly-discovered obstacle. This square decides a boundary around the obstacle which the ATV cannot cross. The margin kept on each side of the obstacle, to create the boundaries of this virtual square, is so as to keep the ATV at a safe enough distance away from the obstacle. This ensures that the vehicle does not collide with the obstacle. Also at the same time the margin is optimum enough for the ATV to go just around the obstacle.

19) Local path: The sub route other than the main regular path to be followed during navigation. It is created to go around any obstacle.

20) Start point of imaginary square route: Point from where a new local path around an obstacle starts.

21) End point of imaginary square route: Point where the local path around an obstacle finishes.

3.2 Algorithm

1) The field dimensions, vehicle dimensions, LIDAR range and information about any present known obstacles are given.
2) Initially, a virtual map is built using the given data. This map will define the boundary of the field area to be covered and the trajectory of the vehicle required for the navigation considering the known obstacles.
3) The navigation field for this algorithm is considered in two dimensions – X and Y. The X coordinates increment from left to right and the Y coordinates increment from top to bottom.
4) A definite pattern is pre decided for the trajectory (as indicated by the black arrows in Figure 3-3). This pattern is so as to cover the entire field area. In this pattern the vehicle starts its navigation from the top left corner of the field. It maneuvers in straight lines along the length of the field and takes a turn towards right, only when it reaches the top or bottom field limits.

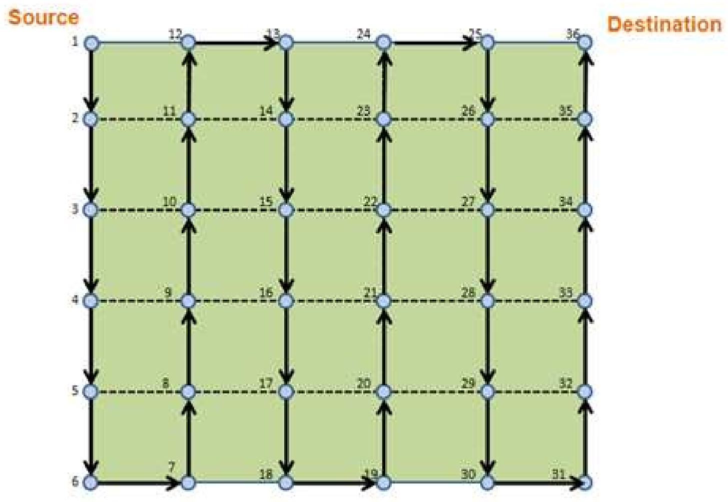


Figure 3-3 Pattern of Navigation in the field

5) Initially, the location (in terms of X, Y coordinates) of the scan points, goal points, and reach points is calculated. Scan points on the boundary will have three reach points while scan points at the corners will have two reach points.

6) The numbering of the scan points is done as per the particular pattern of trajectory required in this algorithm.

7) The vehicle turns or rotations are taken care of by the mechanical section.

8) The navigation starts from source and the ATV moves consecutively from the current scan point to its goal point.

9) In case of no known or newly-discovered obstacles the vehicle continues moving along the specific pattern of trajectory till the last scan point (destination) is reached.

10) For known obstacles, the following information is taken as input:

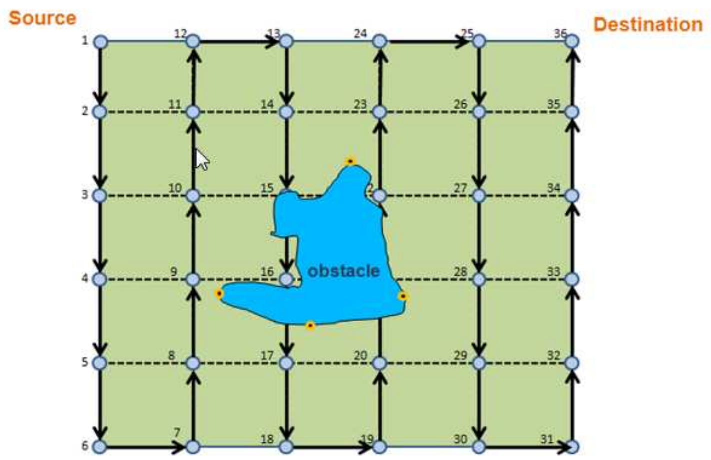
- Number of obstacles
- For each known obstacle:

- Y coordinate of end point of the obstacle in north most direction
- Y coordinate of end point of the obstacle in south most direction
- X coordinate of end point of the obstacle in east most direction
- X coordinate of end point of the obstacle in west most direction

11) Add a distance equal to half the vehicle width to all the above points, and calculate the northwest, northeast, southwest and southeast points.

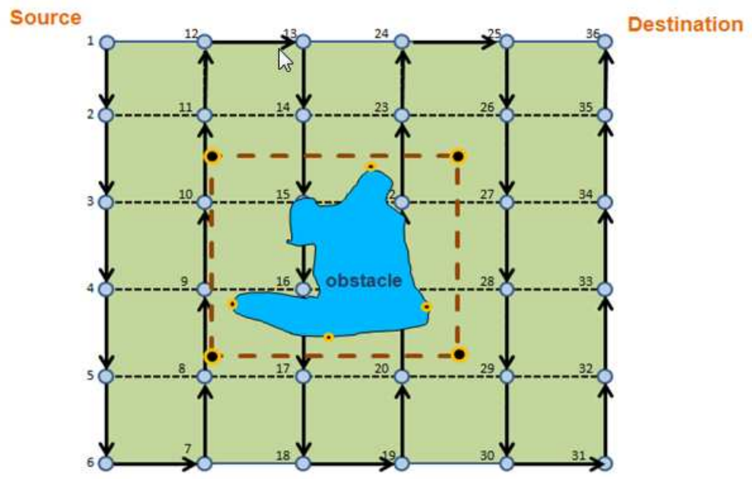
    Draw an imaginary square that include these points as the corner points. See Figure 3-5 for reference.

12) Consider a known obstacle for the given field as shown in Figure 3-4



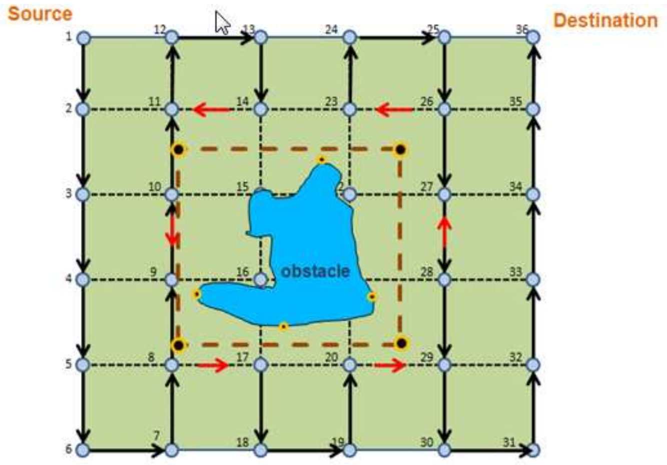
Known obstacle in the field: Points marked in yellow are the extreme north, south, east and west end points of the obstacles given as input for this obstacle

Figure 3-4 Field with Known Obstacles



Using the given extreme points the corner points the virtual square is built surrounding the obstacle. Points inside the imaginary square are discarded (considered unreachable).

Figure 3-5 Imaginary Square around the known obstacles



The northwest, northeast, southwest and southeast points are identified and the local path for navigation around the imaginary square is then formatted.

Figure 3-6 Imaginary squares around know obstacles with local path defined

13) When the vehicle reaches scan point 14, it finds the consecutively next scan point (in this case 15) to be marked as unreachable. Accordingly it understands about the imaginary square ahead and so cannot maneuver in the regular manner as shown in Figure 3-3.

14) Since the immediate next scan point after 14 is found to be not reachable, a search is made for the consecutively next scan points to be set as the new goal point. In this case since 15 and 16 both are unreachable 17 is marked as the new goal point for scan point 14.
15) A local path is now devised for the ATV to go from 14 to 17. This local path around the obstacle can go from the left or the right hand side. The decision to go from left or right is based upon the shortest path criteria. The local path that includes least number of intermediate points to go from the source to its goal is considered to be the most optimum path.
16) In case both the paths have the same number of intermediate points the left hand path is given preference. Since scan points on the left hand of the current scan point are going to be the earlier visited ones, there is probability of having no obstacles in the left local path.

3.2.1 Logic for Local Path Creation

1) As per this algorithm, three thumb rules are checked for in case of local path navigation:
2) *Choose the optimum path (left or right) based on number of intermediate scan points to be traversed*
3) *Are there any common reach points between the goal point and current scan point? If yes, path is created through this common reach point. If no -*

4) *To begin a local path around an obstacle, intermediate goal points are created. Again reach point search based upon direction sequence set for that position and the last visited option is used to traverse the vehicle to it actual goal.*

5) A local path is a path that goes around the obstacle in order to avoid it and then resume the main path. In case of the main path, the ATV maneuvers using the consecutively numbered scan points along the trajectory. Similarly when in a local path, some set of scan point numbers are required to be arranged for the ATV to follow.

6) Hence, as soon as it is understood that the actual goal is not reachable and a local path is required to go to the next possible goal a stack is built up. This stack stores the set of scan point numbers along which the ATV will have to maneuver in order to move around the obstacle.

7) The main task of the algorithm, when a local path is about to start, is to reach the new goal point possibly using the most optimum path.

8) As stated in the fourth point in Section 3.2 the predefined path of navigation for this algorithm is so as to move along the field from left to right. So at any location on the field the scan points to its left are the ones which have been passed upon earlier.

9) Also other than the boundary position when inside the field the ATV always maneuvers in vertical columns alternately going up and down (Figure 3-3)

10) As will be seen in detail in Chapter 4, considering the maximum distance between any two scan points in X direction and the margin used to draw the imaginary square around an obstacle, at least one scan point will go inside the

imaginary square. Hence a situation would never arise such that the source scan point and its goal are horizontally on the same level (having the same Y positions and differing only in X positions).

11) As per this algorithm irrespective of the position of the obstacle in the field, the source scan point and the alternate goal scan point will always differ in their Y positions (irrespective of their X positions).

12) Reach points of every scan point are the most important in developing the local path. Using a chain of reach points a stack is created in backtrack manner from the goal to the source scan point for the local path.

13) Consider for example Figure 3-6. The local path from 14 to 17 consists of scan points 14, 11, 10, 9, 8 and 17. As can be seen from the Figure 3-3, starting from the source, every next point is a reach point of its preceding scan point.

14) All the intermediate scan points in this local path are termed as alternate goals. In the above example 11, 10, 9 and 8 are the alternate goals formed in the local path from 14 to 17.

15) Since the algorithm focuses on having an optimum local path, the local path will go from source to goal from either the left hand or right hand side depending upon which is the shortest path.

16) This algorithm has the logic for obstacle avoidance based on Backtracking, Quadratic Positioning and Direction Sequence cases.

17) Backtracking: To create a local path its intermediate points are decided from the goal scan point towards the source scan point. For local path creation, since the aim is to reach the goal point, a reach point that gives direct access to the goal

point is initially searched. A reach point that is in accordance with the direction sequence from the goal is selected. This assures that the reach point meets the algorithm's criteria for a local path. This reach point is then termed as alternate goal. The same procedure continues till the source becomes a reach point of an alternate goal along the local path.

18) A stack consisting of all such related reach points is made from the goal t the source. The number of elements of this stack hence represents the number of intermediate scan points to go from the source to goal.

19) Two separate stacks of intermediate points are created considering the local path from left and right respectively. The stack counter of these two stacks is compared to decide which stack (left or right local path) has lesser number of intermediate points. The one with a least number of stack counters is marked as the optimum local path to be followed in order to go around the obstacle.

20) Quadratic Positioning and Direction Sequence: As seen in point 'viii', two base cases are set:

    Source Y < Goal Y and Source Y > Goal Y

    In Figure 3-6, for the local path from scan point 14 to 17 Source Y < Goal Y, and from 20 to 23 the Source Y > Goal Y. To create a chain of reach points from the goal to source, alternate goals are created starting from the main goal point. The quadratic case selection for deciding each of these intermediate scan points (reach points) in a local path is based on the position of the alternate goal with respect to the source scan point. Also for each case based on the positioning, preferences are set to choose the reach point on left, right, bottom or top. Eight quadratic cases and specific sequence of four preferences is set for selecting each alternate goal.

21) Two such tables are created considering local path from left and from right

22) Quadratic Cases and sequence to be considered for reach point selection for left local path:

T = Top, L = Left, B = Bottom, R = Right

Table 1 : Case table for left local path

Main case 1	**Ysource < Ytarget**	
Case No	**Condition**	**Sequence**
1.1	(SourceY < AltGoalY) && (SourceX == AltGoalX)	TLBR
1.2	(SourceY < AltGoalY) && (SourceX > AltGoalX)	TLBR
1.3	(SourceY < AltGoalY) && (SourceX < AltGoalX)	LBRT
1.4	(SourceY == AltGoalY) && (SourceX > AltGoalX)	RTLB
1.5	(SourceY == AltGoalY) && (SourceX < AltGoalX)	LTBR
1.6	(SourceY > AltGoalY) && (SourceX == AltGoalX)	BLRT
1.7	(SourceY > AltGoalY) && (SourceX > AltGoalX)	TRLB
1.8	(SourceY > AltGoalY) && (SourceX < AltGoalX)	BRLT
Main case 2	**Ysource > Ytarget**	
2.1	(SourceY < AltGoalY) && (SourceX == AltGoalX)	TLRB
2.2	(SourceY < AltGoalY) && (SourceX > AltGoalX)	RBLT
2.3	(SourceY < AltGoalY) && (SourceX < AltGoalX)	TLRB
2.4	(SourceY == AltGoalY) && (SourceX > AltGoalX)	RBTL
2.5	(SourceY == AltGoalY) && (SourceX < AltGoalX)	LTBR
2.6	(SourceY > AltGoalY) && (SourceX == AltGoalX)	LTRB
2.7	(SourceY > AltGoalY) && (SourceX < AltGoalX)	LTRB
2.8	(SourceY > AltGoalY) && (SourceX > AltGoalX)	BLTR

23) Quadratic Cases and sequence to be considered for reach point selection for right local path:

Table 2 : Case table for right local path

Main case 1	**Ysource < Ytarget**	
Case No	**Condition**	**Sequence**
1.1	(SourceY < AltGoalY) && (SourceX == AltGoalX)	TRBL
1.2	(SourceY < AltGoalY) && (SourceX > AltGoalX)	TLBR
1.3	(SourceY < AltGoalY) && (SourceX < AltGoalX)	TRBL
1.4	(SourceY == AltGoalY) && (SourceX > AltGoalX)	RTLB
1.5	(SourceY == AltGoalY) && (SourceX < AltGoalX)	LTRB
1.6	(SourceY > AltGoalY) && (SourceX == AltGoalX)	BLRT
1.7	(SourceY > AltGoalY) && (SourceX > AltGoalX)	TRLB
1.8	(SourceY > AltGoalY) && (SourceX < AltGoalX)	BLTR
Main case 2	**Ysource > Ytarget**	
2.1	(SourceY < AltGoalY) && (SourceX == AltGoalX)	TLRB
2.2	(SourceY < AltGoalY) && (SourceX > AltGoalX)	RBLT
2.3	(SourceY < AltGoalY) && (SourceX < AltGoalX)	TLBR
2.4	(SourceY == AltGoalY) && (SourceX > AltGoalX)	RBTL
2.5	(SourceY == AltGoalY) && (SourceX < AltGoalX)	LBRT
2.6	(SourceY > AltGoalY) && (SourceX == AltGoalX)	RTLB
2.7	(SourceY > AltGoalY) && (SourceX < AltGoalX)	BRTL
2.8	(SourceY > AltGoalY) && (SourceX > AltGoalX)	BLTR

24) These cases and reach point preference sequence have been set based upon the following criteria :

- Navigation from source to goal from the left hand or right side and
- Depending upon the position of the alternate goal check that the reach point which directs the backtracking path towards the source

25) As per this algorithm, to select the best reach point (alternate goal) in a local path initially the case number is matched depending upon the positioning of

source and alternate goal. As per the case, the best reach point is then selected (depending upon reach ability) based on the preference given in the sequence.

26) A flag is also set to keep track of the last direction used, so that the reach point in opposite direction is not set and thus preventing the local path from going into a loop.

27) For example, if the alternate goal satisfies the 2.8 case (left local path), its Bottom reach point is first checked for reach ability. If so it is returned and marked as the next alternate goal. If not reachable the Left reach point is checked and so on.

28) A sequence of such reach points from the goal to source scan points creates the stack. Based on the least stack counter decision an optimum local path (left or right) is decided. Once the local path is ready the ATV uses this path for navigation from source to goal for every obstacle. Upon reaching the goal navigation along the regular path continues till the final destination.

3.2.2 Avoidance of Newly-Discovered Obstacle

The newly-discovered obstacle avoidance logic is same as that for the known obstacles. Only difference is that it is used to create to local path for newly-discovered obstacles run time when the newly-discovered obstacles are detected during navigation. If the goal point of the current scan point is not reachable due to presence of a newly-discovered obstacle, the goal point is checked for reach ability from any of its other reach points. The data given by the LIDAR scan range as input is considered while taking decisions to reach the goal point from any of its reach point.

3.3 Example

1) Consider the example shown in Figure 3-7 and the local path from scan point 14(source) to scan point 17(goal).

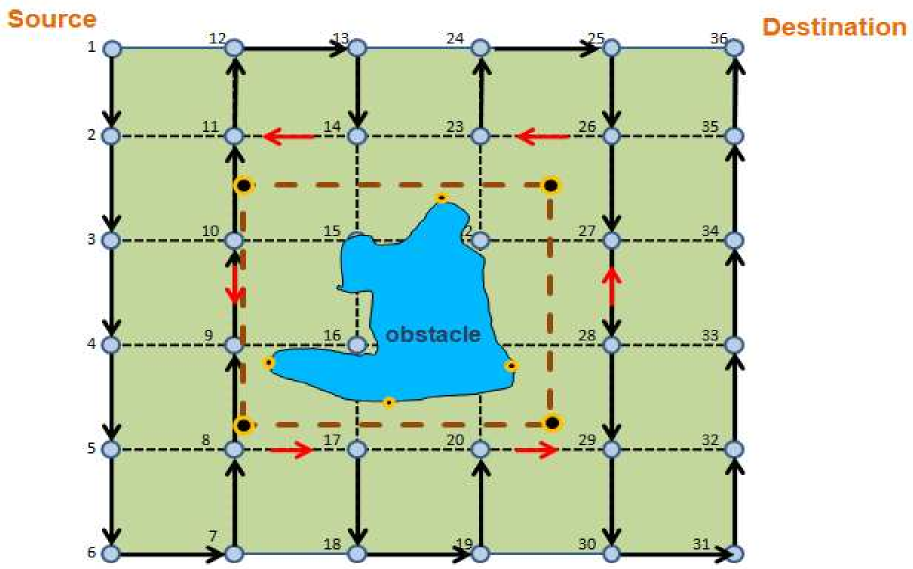


Figure 3-7 : Obstacle and imaginary square

2) Initially, the source Y (Y coordinate of 14) is less than goal Y(Y coordinate of 17). So main case 1(local left path) is considered for deciding the local path.

3) Initially 17 acts as the alternate goal. So we have case 1.1 ((SourceY < AltGoalY) and (SourceX == AltGoalX)). As per the sequence preference is first given to the 'top' reach point of 17. In this case it happens to be scan point 16. Since 16 is not reachable (inside the imaginary square) next preference of the 'left' reach point is checked. Reach point number 8 of alternate goal 17 on its left is reachable and so is set the new alternate goal. This is shown in Figure 3-8.

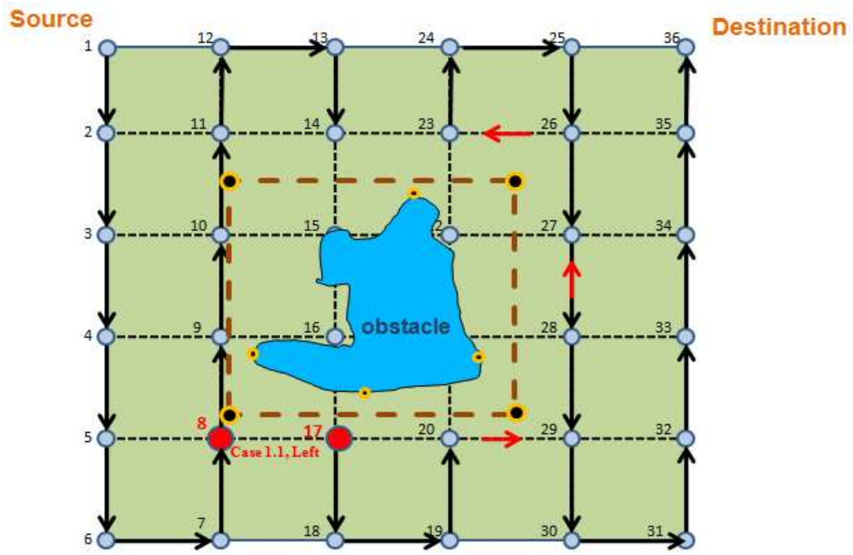


Figure 3-8 : Case 1.1 to decide next scan point to be 8

4) With 8 as the alternate goal and 14 as the source case number 1.2 is considered. As per the sequence preference the 'Top' reach point number 9 of alternate goal 8 is reachable and set as new alternate goal. This is shown in Figure 3-9.

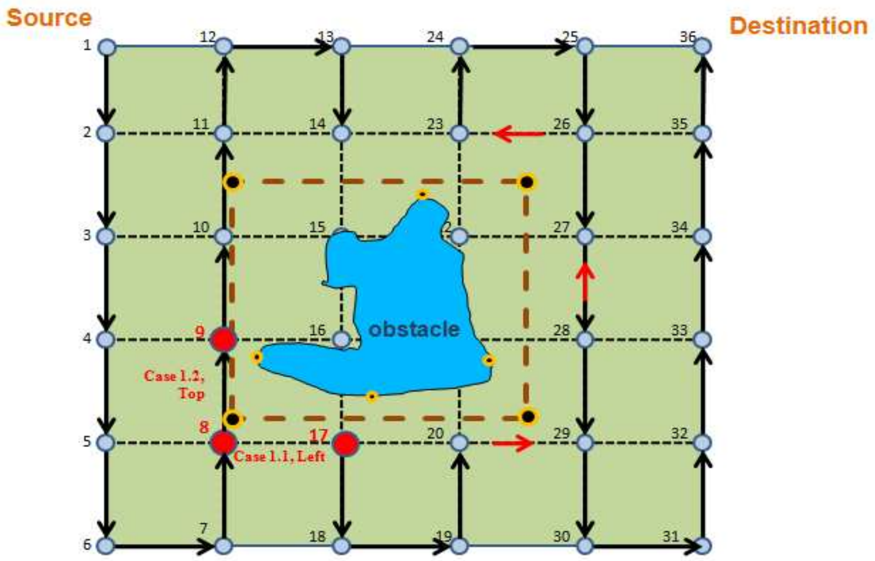


Figure 3-9 : Case 1.2 to decide next scan point to be 9

5) The same case continues for alternate goal points 9 and 10. For both these cases since first reach point preference on 'Top' is available 10 becomes the alternate

goal of 9, and similarly 11 becomes the alternate goal of 10. This is shown in Figure 3-10.

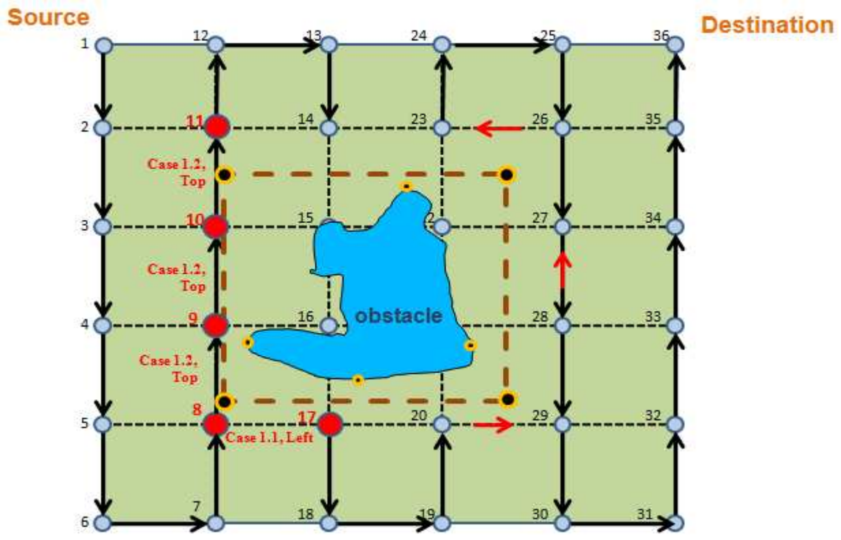


Figure 3-10 : Case 1.2 to decide next scan points to be 10 and 11

6) When 11 is the alternate goal point the case now changes to 1.4. As per the sequence preference the ‘Right’ reach point number 14 of alternate goal 11 is reachable. Also here 14 is the source point.

7) So the reach point chain of sequence (left stack) started from the goal has reached the source and the local path from 14 to 17 from left hand side is created.

8) The same procedure is repeated considering the cases from the table for right hand side local path. This local path creates a stack consisting of scan point 14, 23, 26, 27, 28, 29, 20 and 17. It has a stack counter of 8 compared to the stack counter of the left stack which was 6. The decision to go from left is finalized, and the scan points in the left stack are used for the actual local path navigation. This is shown in Figure 3-11.

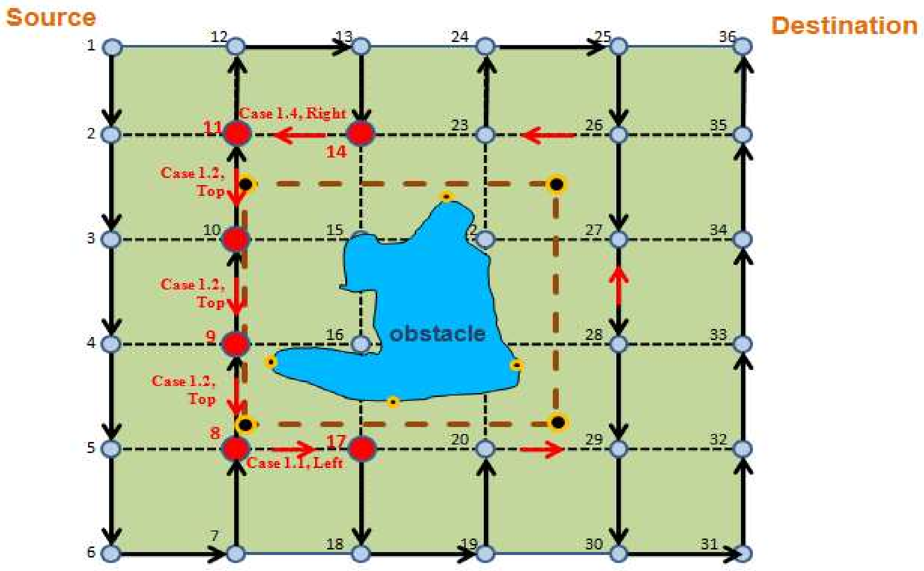


Figure 3-11 : Local path created around obstacle using different cases

3.4 Computational Analysis of the algorithm

3.4.1 Computational Complexity of Obstacle avoidance

1) The obstacle avoidance routine in this algorithm can be categorized to be a modified version of the breadth first search (BFS) technique.

2) The BFS is a graph search algorithm and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal. Consider for example the following tree structure shown in figure 3-12 used to represent a search from source to goal point:

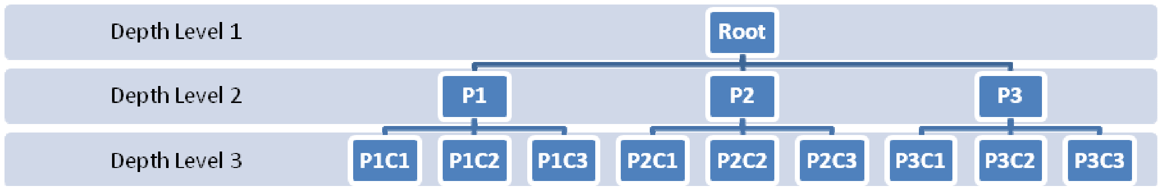


Figure 3-12 : Computational technique used for obstacle avoidance

3) In this example, the search begins from the root at depth level 1. If the goal is not found, the search proceeds to the second level. Here, all the nodes throughout the breadth of the level are searched for from left to right. So P1, P2 and P3 are searched to match with the goal. If not, the search proceeds to the third depth level. Again here all the nodes throughout the breadth of this level (P1C1, P1C2, P1C3 ….. P3C3) are searched.

4) Accordingly, the search complexity of the BFS algorithm is O(b^^{d}) where b is the branching factor (number of children for each node) of the tree(3 in the above example) and d is the depth required to search the goal.

5) The modification done to the BFS in this algorithm is so that at every depth level, the best possible node is selected based on the logic (case tables) developed, and only that node is explored further.

6) So, considering above example, P1, P2 and P3 are searched to match the goal. If none of them is the goal, the best possible node, which will lead to the goal, is considered. For example, if P2 is the best node the child nodes of only P2 are further explored in depth level 3. Likewise, if the goal is found at depth level d the maximum number of nodes that are required to be explored are b*d.

7) In this algorithm the modified BFS is called twice, once for the left and then for the right hand local path. Consider the depth level of search for left local path is l_D and the depth level of search for right local path is $r_{D.}$ In this case complexity would be $(b^* l_D) + (b^* r_D) = b * (l_D + r_D)$.

8) In the above example, shown in figure 3-12, the branching factor is 3. For the tree structure of this algorithm for obstacle avoidance, any parent node is going to be the base scan point and its child nodes will represent its reach points. Accordingly, for this algorithm, the branching factor is four (maximum number of reach points for any scan point).So, the worst case depth for this algorithm would be the total number of scan points (n). In the worst case scenario, all scan points (n) can have depth of d.

9) If only BFS (without any modifications) had been considered for the obstacle avoidance routine in this algorithm, the complexity would have been $O((4n)^{\wedge n})$.

10) For BFS (with the modifications done in this algorithm), the complexity is $O(4(n+n)*n) = O(8n^{\wedge 2})$.

3.4.2 Computational Complexity of Navigation

Overall, there are 10 for loops in the software (considering no obstacles) used to create the trajectory and navigate the vehicle along this trajectory from the source to its destination. Each for loop has a length of n. So, considering a field without any obstacle, the algorithm carries out only the navigation routine with a complexity of O(10n).

3.4.3 Computational Complexity of algorithm

= complexity of navigation + complexity of obstacle avoidance

$= O(10n) + O(8n\hat{}^{2}) = O(10n+8n\hat{}^{2})$

3.4.4 Example to describe the modified BFS in this algorithm

Consider the following example in figure 3-13 to create a left local path from scan point 14 to 17. The following computation is done to develop the stack from left hand side of the obstacle. As described in the algorithm, the stack for the local path is crated from the goal point to the source point. So in this case, the stack creation will start from point 17 and aim to reach till 14.

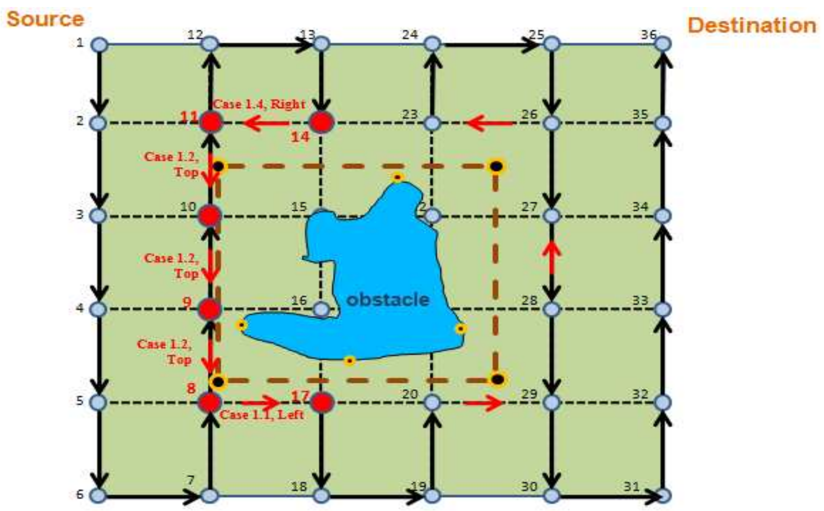


Figure 3-13 : Example for computational technique

Left reach point - 8 is selected as the best node for depth level 2 as per Case 1.1

So, as shown in figure 3-14, the child nodes of only node 8 are explored for depth level 3.

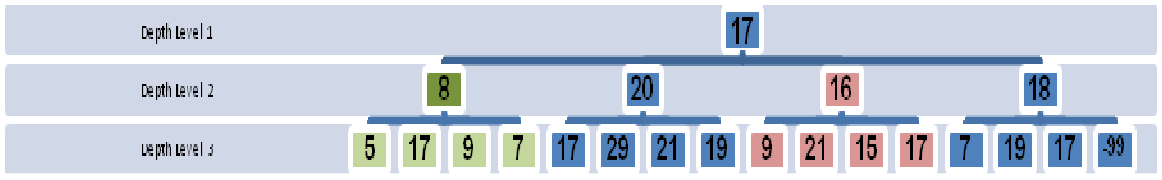


Figure 3-14 : Modified BFS at depth level 2

Top reach point - 9 is selected as the best node for depth level 3 as per Case 1.2

So, as shown in figure 3-15, the child nodes of only node 9 are explored for depth level 4.

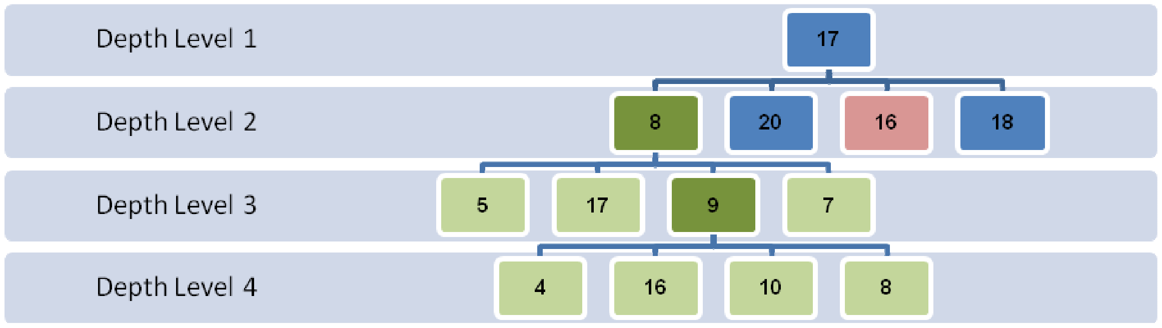


Figure 3-15 : Modified BFS at depth level 3

Top reach point - 10 is selected as the best node for depth level 4 as per Case 1.2

So, as shown in figure 3-16, the child nodes of only node 10 are explored for depth level 5.

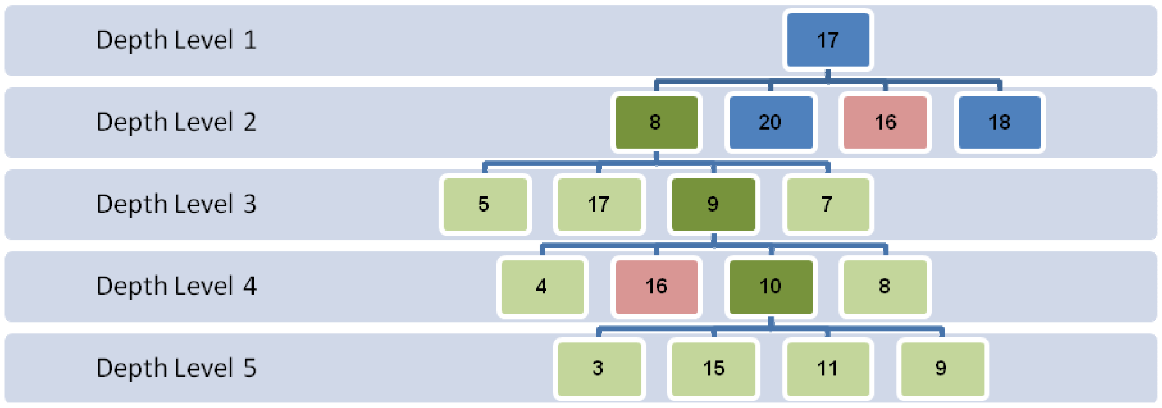


Figure 3-16 : Modified BFS at depth level 4

Top reach point - 11 is selected as the best node for depth level 5 as per Case 1.2

So, as shown in figure 3-17, the child nodes of only node 11 are explored for depth level 6.

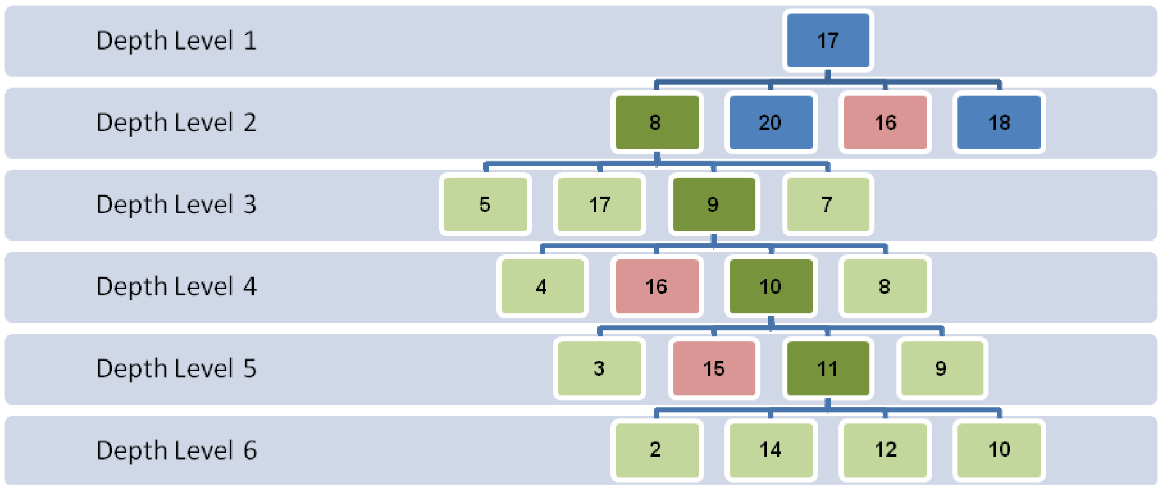


Figure 3-17 : Modified BFS at depth level 5

Top reach point - 14(source) is selected as the best node for depth level 6 as per Case 1.4

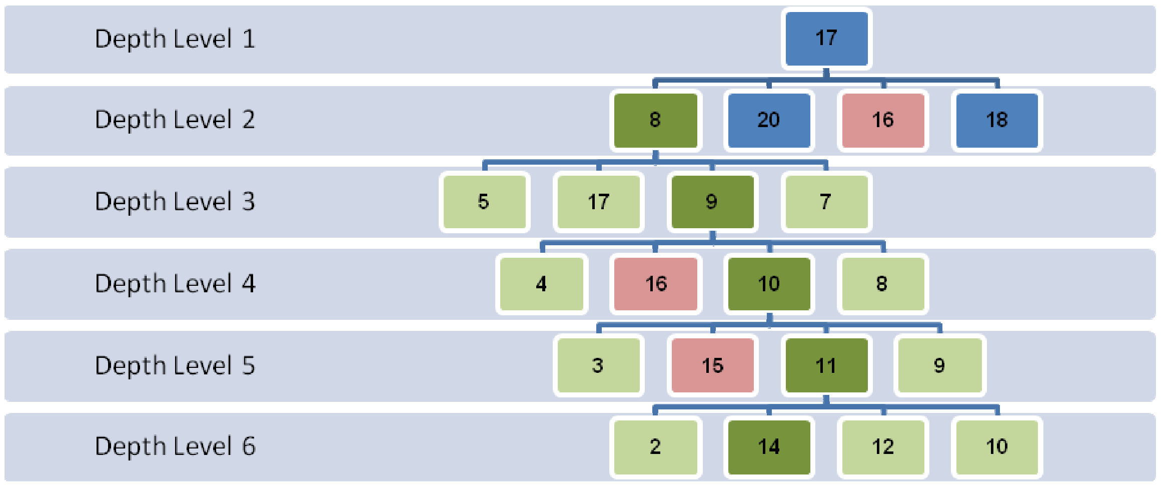


Figure 3-18 : Modified BFS at depth level 6

Note: Nodes in pink color are discarded due to unreachability. Nodes in green color are the ones as best selected as per the case table.

3.5 Data Structure

Structure to store X and Y co-ordinates of any scan point:

```
struct  Coordinates {
                              int X;
                              int Y
                    };

Coordinates Curr, Next, Prev;
```

Structure to store the known obstacle data:

```
struct Obst
{
    Coordinates North, South, East, West;
}
```

Structure to represent various attributes of a scan point:

```
struct ScanPt
{
   Coordinates Source, Goal, Igoal;
   //Coordinates ReachPtCords [4];        // coordinates of reach points
   int ScanPtNumsOfReachPts [4];       // stores scan point number of each of reach points
   int ScanPtNum;                      // number to identify a scan point
   int GoalPtNum;                      // number to identify Goal of current scan point
   int ReachPtAccessibility [10][10];   // identifies accessibility of each reach point
   int NumOfReachPts;                  // total number of reach points for a particular scan point
   int Visited;                   // indicates of the scan point is already visited
   int IsScanPointReachable;          // indicates if the scan point can be reached
   int ObstacleNo;                    // identifies the obstacle which makes this point unreachbale
   int AltGoalPtNum;                  // alternate goal pt in case of known obstacles
   int Weightage;                    //Weightage for every ScanPoint. The ones visited more times get more Weightage
} ScanPtArr [10000], SimulationScanPtArr [10000];
```

3.6 Pseudo Code

The algorithm is categorized into seven routines:

- Ø Initialization
- Ø CreateImaginarySquare
- Ø Newly-discovered Obstacle detection
- Ø Move vehicle
- Ø Local Path Navigation

- Ø Navigate through Field

3.6.1 Initialization

For all scan points, calculates X and Y co-ordinates, goal point, reach points.

```
ScanPtNumber = 1;
iCounter = 1;
Number of XScanPoints = Roundup (FieldWidth/VehicleWidth) +1
Number of YScanPoint: Roundup (FieldLength / [Lidar Range/2]) +1
XScanRange = FieldWidth / (Number of XScanPoints-1)
YScanRange = FieldLength / (Number of YScanPoints-1)
Total Number of Scan Points = Number of XScanPoints * Number of YScanPoints
Temp = Number of YScanPoint;
for (iCountX=0; iCountX< Number of XScanPoints; iCountX ++)
{
        for (iCountY=0; iCountY< Number of YScanPoints; iCountY ++)
        {
                Xcord = (iCountX * XScanRange);
                Ycord = ( iCountY  * YScanRange);
         if ((iCountX is odd)
    {
                ScanPtNumber = iCounter +Number of YScanPoints – (iCountY*2) - 1;
    }
    else
        ScanPtNumber = iCounter;
    ScanPt [ScanPtNumber].ScanPtNum = ScanPtNumber;
        ScanPt [ScanPtNumber].Source.X = Xcord = current.X;
         ScanPt [ScanPtNumber].Source.Y = Ycord = current.Y;
    iCounter++;
    NumOfReachPoints = 0;
}

    for (all scan points)
    {
       Calculate X and Y coordinates of all possible reach points;
      Find the scan point numbers with those coordinates;
      Assign these scan point numbers as the reach points;
}  //end for
For(icount=1;icount<Total Number of Scan Points;icount++)
{
         ScanPt [ScanPtNumber].Goal.X = ScanPt [ScanPtNumber+1].Source.X ;
         ScanPt [ScanPtNumber].Goal.Y = ScanPt [ScanPtNumber+1].Source.Y;
}
```

3.6.2 Create Imaginary Square

Using the data of known obstacles this routine creates a virtual square around the obstacles.

Input: Number of obstacles

Extreme North, South, East, West points of each known obstacle

```
CreateImgSqr (NoOfObst, north, south, east, west);
{

      for (NoOfObst)
```

```
{
  Add a distance = half the vehicle width to each of north, south east, west coordinates;
  North east coordinates = Y coordinate of north and X of east;
  North west coordinates = Y coordinate of north and X of west;
South east coordinates = Y coordinate of south and X of east;
  South west coordinates = Y coordinate of south and X of west;

// determine imaginary scan points
for (i =0; i < = field length; i+=Yscan range)
{
  if ((i>south) && (i<north))
  ImgScanPt1.X = Northwest.X;
  ImgScanPt2.X = Northeast.X;
  ImgScanPt1.Y =ImgScanPt2.Y =i;
}
for (i =0; i < = field width; i+=Xscan range)
{
  if ((i>west) && (i<east))
  ImgScanPt1.X = Northwest.X;
  ImgScanPt2.X = Northeast.X;
  ImgScanPt1.Y =ImgScanPt2.Y =i;
}
for (all scan points in this imaginary square)
{
     IsScanPointReachable = 0;
     ScanPt.ObstNo = sequence number of obstacle;
}
```

3.6.3 Obstacle Detection

This function gets input from LIDAR and calculates obstacle position with respect to scan points.

Input: Current scan point, Next scan point

Output: 1 [If obstacle exists between current scan point and its goal points]

0 [If obstacle does not exist between current scan point and its goal points]

ObstacleDetection (Current scan point, Next scan point)

Detects if an obstacle is present between the current and its next scan point. The LIDAR operation can be used for this function.

3.6.4 Move Vehicle

Based on co-ordinates of previous scan point, current scan point, and goal scan point this function decides the direction [straight, left or right] to be taken to reach the goal point. This function calls mechanical routine to actually move the vehicle to be positioned at the goal point.

Input: Current, Previous, Next Scan point

```
MoveVehicle (startPt, endPt, prevPt)
    {
    if (endPt.X == prevPt.X)
        then, Direction = Straight;
    else
    {
        val = ((endPt.Y- prevPt.Y) / (endPt.X – prevPt.X));
        If (startscan point > goal scan point)
        val = - val;
        if (val == 0)
            Direction = Straight;
        else if (val >0)
            Direction = Right;
        elseif (val <0)
            Direction = Left;
    }
    Call mechanicalMove (Direction, startPt, endPt);
    }
```

3.6.5 Local Path Navigation

This function navigates the vehicle around the obstacle to reach back to its normal path. It calculates intermediate goal points around the obstacle and hence creates a local path.

Input: Goal Scan Point Number

```
LocalPathNav (SourceScanPtNum, GoalPtNum)
    {
        Create two stacks;
        Store the source scan point number;
        Store the goal scan point number in as the first element of both the stacks;
        Find the alternate goals in the left local path;
        Store each of these alternate goals in the left stack;
        Check if reach point of alternate goal is same as the source scan point;
        If so, left local path is completed;
        Repeat the same procedure to create a right local path;
        Based upon the shorter of the two an optimum path is selected;
      Pop the optimum stack scan point numbers and call move vehicle function;
        If a new newly-discovered obstacle is in the path of the local path, then call the local path
function
      again with the new source and goal point numbers;
        Continue the recursive process until the original goal is reached;
  }
```

3.6.6 Find Optimum Intermediate Goal Point

Based on the table of cases above this function finds the most optimum scan point that could form the alternate goal. It calculates intermediate goal points around the obstacle based on the algorithm criteria and hence creates a local path.

Input: Source scan point, alternate goal scan point number

Output: The best possible intermediate reach point

```
FindOptimumLocalGoalPt (int SourcePt, int AltGoalPt)
{
        Based on the source and alternate goal point Y coordinates determine the main case number;
        Based on the main case number, X and Y coordinates of alternate goal and source scan point number the
    particular case number is selected from the table above;
        As per the case number the reach point availability is checked for as per the sequence of preference
    mentioned in the table;
    If the reach point in the sequence is reachable, it is returned,
    Else the next reachable reach point in the sequence is opted;
    If none of the reach points in the sequence are reachable, the particular scan point is marked unreachable
    as well;
}
```

3.6.7 Navigate Through Field

For every scan point in the field, it checks for presence of obstacle. If so, then the vehicle navigates through a local path. Else the vehicle moves along its normal path.

```
NavigateField ()
        {
                for (all scan points)
                 {
                    if (Next scan point is not reachable)
                   LocalPathNav (SourceScanPtNum, Goal.ScanPtNum);
               else
               {
                     if (ObstacleDetection (Current scan point, Next scan point) == 0)//no newly-discovered
obstacle
                    MoveVehicle (currentscanPt, goalscanPt, (current-1) scanPt);
                    else
                    LocalPathNav (SourceScanPtNum, Goal.ScanPtNum);
                  }
            }
        }
```

Chapter 4: Role of Logistics and Assumptions in Path Planning

Decision making is a vital process in any artificial intelligence project. The question why has to be answered for any single step considered. Decision making works in parallel with the role of logistics for this algorithm development. Path planning is only a secondary step to mapping. It can only be done in a known environment. For a robot, SLAM creates this virtual environment. Hence, discrepancies, if any, in mapping are carried forward in the implementation of path planning. Accordingly, utmost care has to be taken during the logic development of the algorithm so as to make the navigation as fault tolerant as possible. Also, being the very first version of this research, there are some basic assumptions that are defined. This chapter elaborates all such assumptions along with the importance of logistics to develop the algorithm.

4.1 Assumptions

There are various assumptions that have been considered before evaluating this algorithm:

1) The field (outdoor operation space, open field or forest) will always be rectangular.
2) There will be no obstacles on the boundary of the field.
3) Information about the location and size of all the existing (known) obstacles will be provided.
4) The Dimensions of the field will be provided.
5) The Dimensions of the vehicle (length and width) to be used for maneuvering shall be provided.
6) The LIDAR has a maximum operating range of 80 meters.
7) A 90 degree turn is decided upon any time the vehicle has to turn.
8) Field is considered only in two dimensions, X and Y. With these assumptions as prerequisites, the algorithm is still capable of detecting new obstacles, avoiding them, and maintaining the predefined route.
9) For any autonomous motion there is a difference between predicted motion and the actual motion. Furthermore, the difference in motion could be in terms of distance travelled, direction of travel, speed of travel, and angle of turn. There are varied factors which affect the intensity of this difference. A few of such factors are the mechanical factors of the vehicle, the sensor inputs to the vehicle, the fault tolerance intensity of the algorithm logic, the external environment or the ground conditions. This algorithm is primararily developed and tested for

the simulation stage. The algorithm focuses on optimum management of the input data, so as to process and provide path planning in a systematic manner. Since the algorithm is for path planning of an autonomous vehicle every single step of the algorithm implementation should have a valid reason.

4.2 Logic

Here we see the details of how very fairly the input resources have been used for the logic development. The reasoning for particular steps, equations and formulas used in the code and algorithm will be explained here.

4.2.1 Data Structure

Three data structures are used: Coordinates, Obstacle and ScanPt.

1) Coordinates stores the Cartesian coordinates of any scan point. This structure is used while referring to the current, next and previous scan points while moving the vehicle.
2) Obst stores all the parameters related to the known obstacles, and its related imaginary square such as the boundary coordinates of these obstacles.
3) ScanPt stores all the parameters related to a scan point. These parameters include the scan point's reach ability, its number, its visited status, number of reach points it has, its actual and alternate goal point number. This structure is used while referring to any particular scan point during simulation as well.

4.2.2 Formulas

- The number of X and Y scan points are calculated using:

$$NoOfXScanPoints = (FieldWidth/VehicleWidth) + 1$$

Equation 4-1

$$NoOfYScanPoints = (FieldLength / (LidarRange/2)) + 1$$

Equation 4-2

This algorithm plans the path for the ATV such that the entire are of the rectangular field is covered in a parallel zigzag manner along the length of the field.

The black arrows shown in Figure 3-3 indicate the direction of navigation in which the ATV should maneuver. The distance between any two adjacent scan points in the X direction is suppose to be so that when the vehicle is at any one of the two adjacent points, half the distance between them is covered. The distance between any two adjacent scan points should be just enough for an entire vehicle width, so that eventually when the ATV maneuvers up and down, the entire field area between these points is covered. Hence, to calculate the distance between any two X scan points the entire field with is divided by the vehicle width. The “+1” in the formula ensures the first and last scan points in X direction. The distance between any two scan points in the Y direction is supposed to be so that the vehicle can maneuver from one point to the next safely with proper obstacle detection. At every scan point the LIDAR scans the area in front of it to check for safety in order to move forward. Consider the case if the distance between two consecutive scan points in Y direction is set to LIDAR range as in Figure 4-1

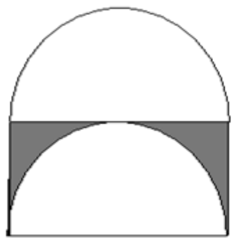


Figure 4-1 : Y scan range = LIDAR range

As seen in this case the shaded area is not covered in LIDAR range to check for presence of any obstacle.

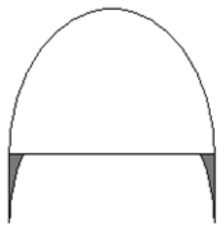


Figure 4-2 : Y scan range = LIDAR range / 2

On the other hand, as seen in Figure 4-2 with a distance of LIDAR range/2 the shaded area which cannot be detected for obstacles is reduced by a large margin. Hence, this algorithm considers the distance between any two scan points in the Y direction to be LIDAR range/2. This distance can be further changed as per the requirement for resolution of obstacle detection. Accordingly the number of scan points in X and Y direction are calculated as shown in equation 4-1 and 4-2.

- The Number of X and Y scan points are then used to calculate the scan range in X and Y direction and the total number of scan points.

$$XScanRange = FieldWidth / (NoOfXScanPoints - 1)$$
$$YScanRange = FieldLength / (NoOfYScanPoints-1)$$
$$TotalNoOfScanPts = NoOfXScanPoints * NoOfYScanPoints$$

Equation 4-3

- The X and Y coordinates for each scan point is calculated as per the distance of the scan point from the source point in X and Y direction respectively.

```
Xcord = (iCountX * XScanRange)
Ycord = (iCountY * YScanRange)
```

Equation 4-4

- The number of reach points for every scan point depends on the scan point's location. If the scan point is on any of the field boundaries, it has 3 reach points. If it is at any of the four corners, it has two reach points. For all the scan point inside the field there are four reach points. All of these three cases for creating reach points are taken care by the following formulas :

```
Left: if(ScanPtArr[].Source.X - XScanRange) >= 0
Right: if ((ScanPtArr[].Source.X + XScanRange) <= FieldWidth
Top: if (ScanPtArr[].Source.Y - YScanRange) >= 0
Bottom: if (ScanPtArr[temp].Source.Y + YScanRange) <= FieldLength
```

Equation 4-5

In case, if the left condition is true, and the scan point has a reach point on its left with a distance of Xscan range away from the current. Similarly the coordinates for reach point on all remaining sides are calculated

- To find the alternate goal point in case if the actual goal point lies inside an imaginary square and is not reachable the formula used is :

```
GoalPtNum = ScanPtArr[iCounter].GoalPtNum;
if (ScanPtArr[GoalPtNum].IsScanPointReachable == 0)
      {
        for(iTemp=GoalPtNum;iTemp<TotalNoOfScanPts;iTemp++)
        {
          if (ScanPtArr[ScanPtArr[iTemp].GoalPtNum].IsScanPointReachable == 1)
          {
            ScanPtArr[iCounter].AltGoalPtNum =  ScanPtArr[iTemp].GoalPtNum;
            break;
          }//end-if
        }//end-for
      }//end-if
```

Equation 4-6

This routine ensures that in case an imaginary square is present in front of a scan point the consecutively next scan point number whichever is reachable is assigned as the alternative goal point.

- To calculate the direction of navigation from one scan point to the next a tangential formula is used :

```
     if((ScanPtArr[prevScanPtNum].Source.X == ScanPtArr[currScanPtNum].Source.X)&&
 (ScanPtArr[currScanPtNum].Source.X != ScanPtArr[goalScanPtNum].Source.X))
     {
        dFactor = ((ScanPtArr[goalScanPtNum].Source.X - ScanPtArr[currScanPtNum].Source.
                (ScanPtArr[currScanPtNum].Source.Y - ScanPtArr[prevScanPtNum].Source.Y));
}
```

Equation 4-7

The direction of navigation is decided based upon on the previous, current and next scan point's X and Y coordinates. The 'if' statement here checks if the current and next scan point does not have the same X coordinate. If so it means it has to divert from its straight path. The dFactor then decides left or right turn. The fact that the X coordinates increase from left to right while the Y coordinates increase from top to bottom is used to decide the direction. In this formula, if the dFactor is greater than zero, it indicates a left turn because as seen from the formulas there are two cases where the dFactor could be positive:

- Ø If the current X is less than goal X: which means the current scan point is on the left of the goal X (considering that the X coordinates increase from left to right) and so to go to the goal the vehicle has to take a left turn
- Ø If the previous Y is less than the current Y: which means the previous scan point is placed above the current Y (considering that the Y coordinates increase from top to bottom). And then the current and goal X are different.

- Another similar formula is used in case if the Y coordinates of previous and current scan points are same and at the same time the Y coordinates of current and goal are not same. Similar logic as mentioned in the above point is used in this to decide the direction of navigation.
- The case table mentioned in chapter 3 is the core for finding the best possible intermediate reach point in a local path. It is represented in the *FindoptimumReachPoint* function.

```
if (SourceY < TargetY)//--------------MAIN CASE 1-----------------------------
    {
      if ((SourceY < AltGoalY) && (SourceX == AltGoalX))//CASE 1.1
      {
              fprintf (fpOut,"\nInside Case 1.1\n");
          //consider reach pt on top
          ReachPt = ScanPtArr [AltGoalPt].ScanPtNumsOfReachPts [2];
          ReachPt = CheckIfTemporaryInAccessible (ReachPt);
          //if top reachpt is reachable AND chk if OffPathDirection is not BOTTOMDIR
          if ((ReachPt! = -99) && (ScanPtArr [ReachPt].IsScanPointReachable ==
                                                   1) && (OffPathDirection! = BOTTOMDIR))
          {
             BestReachPt = ReachPt;
             OffPathDirection = TOPDIR;
             return (BestReachPt);
          }
          else
          {    //consider reach pt on left
              ReachPt = ScanPtArr [AltGoalPt].ScanPtNumsOfReachPts [0];
              ReachPt = CheckIfTemporaryInAccessible (ReachPt);
              //if left reachpt is reachable AND chk if OffPathDirection is not RIGHTDIR
              if ((ReachPt! = -99) && (ScanPtArr [ReachPt].IsScanPointReachable ==
                                   1) && (OffPathDirection! = RIGHTDIR))
             {
                BestReachPt = ReachPt;
                OffPathDirection = LEFTDIR;
                return (BestReachPt);
```

```
            }
            else
            {//consider reach pt on BOTTOM
                ReachPt = ScanPtArr [AltGoalPt].ScanPtNumsOfReachPts [3];
                ReachPt = CheckIfTemporaryInAccessible (ReachPt);
                //if BOTTOM reachpt is reachable
                if ((ReachPt! = -99) && (ScanPtArr [ReachPt].IsScanPointReachable
                                                == 1) && (OffPathDirection! = TOPDIR))
                {
                  BestReachPt = ReachPt;
                  OffPathDirection = BOTTOMDIR;
                  return (BestReachPt);
                }
                else
                {//consider reach pt on right
                    ReachPt = ScanPtArr [AltGoalPt].ScanPtNumsOfReachPts [1];
                    ReachPt = CheckIfTemporaryInAccessible (ReachPt);
                    //if right reachpt is reachable
                    if ((ReachPt! = -99) && (ScanPtArr [ReachPt].IsScanPointReachable
                                                == 1) && (OffPathDirection! = LEFTDIR))
                    {
                      BestReachPt = ReachPt;
                      OffPathDirection = RIGHTDIR;
                      return (BestReachPt);
                    }//end case right
                    else
                    {
                        OffPathDirection = NULLDIR; // if none of the reach points are accessible
                        return (AltGoalPt);//return the same alt goal with offpathdir flag nullified
                    }
                }//end case bottom
            }//end case left
        }///end case top
    }////CASE 1.1
```

The above snippet of code indicates the formulation for only case 1.1. Similar 'if else' statements are used to represent the remaining cases. Depending upon the source and goal positioning the case number is decided and accordingly the reach point preference if selected. For instance consider case 1.1 for example. The source Y coordinate is smaller than the goal Y, which means that the source is above the goal scan point. Further, particularly in case 1.1, the alternate goal (intermediate scan point) and source scan point have same X coordinate and differ only in their Y coordinates. That is, in this case both source and alternate goal are on the same vertical column with the source positioned above the alternate goal. In order to backtrack the local path from goal to source, the first preference is naturally given to the Top reach point to check if there is a direct access to reach the source point. Followed by that, since the left points (in local left path) are given the utmost importance, the reach point on Left is preferred. Again if the left reach point is not reachable then as per the algorithm criteria the Bottom reach point is checked for availability. And then the preference is given to the Right reach point. Accordingly the reach point preference sequence for case 1.1 is ‘TLBR’. Similarly the preferences are set for all of the remaining cases in both left and right local path functions.

4.2.3 Code

The code can be classified into sub categories. One category deals with the logic of algorithm development and the other deals with the simulation development. Chapter 5 elaborates the simulation development part of the software. Here we see how the code is implemented in terms of the algorithm logistics.

- Initially the field, LIDAR and vehicle parameters are read in from an input file in the *ReadInputLineFromFile* function.
- The parameters for all the known obstacles are read in from the input file
- Given the field and vehicle parameters the scan points are created for the entire field in the *CreateScanGoalPts* function
- Also, numbers are assigned to scan points exactly as per the requirement of the predefined trajectory of navigation. The location of the scan points on the field is also considered for numbering. Hence the scan point with a number consecutively next to the current scan point number is assigned as its goal scan point.
    - Ø Initially all the scan points are marked as reachable and not visited
    - Ø Reach points are created

- The coordinates of all the known obstacles is used to create an imaginary square around the known obstacles. This square is created such that it has its boundaries at a distance of half the vehicle width away from the extreme north, south, east and west limits of the obstacle. This gives the vehicle optimum space to maneuver around the obstacle taking care that the ATV does not collide with the obstacle. Accordingly the four corners of the imaginary square are detected
- In case of a known obstacle, the scan points lying inside the imaginary square are now termed as not reachable. The *ScanPtInsideImSqr* function detects and marks all such scan points lying inside imaginary square as unreachable. Also, an alternate goal point is assigned to the last reachable point.
- After all the above assignments are made, the code is now ready to actually navigate the ATV. This is done by the *NavigateField* function. Here the move vehicle function is called to locate the ATV at appropriate scan point by going in appropriate direction.
- The very first thing checked for, in navigated field, is if there are any known obstacles present. If no such obstacle is present, based on the above initializations and assignments the *MoveVehicle* function holds the routine to physically move the vehicle from one scan point to the one having its consecutively next goal. In this manner it eventually continues its navigation till the last scan point (destination). The direction of navigation required to go to the next scan point is also calculated before the ATV is actually moved to the next scan point.
- In case if a known obstacle is present in front of a scan point the *LocalPathNav* function is called. The first step in this function is to find the next best possible

reachable scan point along the same path as it would be without the obstacle. This is termed as the new goal point. As per the algorithm this function maneuvers the vehicle around the obstacle [along the scan points] in an optimum manner such that the path with shorter distance is preferred.

- A reach point is any scan point which is at a distance of X or Y scan range from the current scan point, and hence, has a direct access to the current scan point. Reach points for all scan points are created at the time of initialization itself. Reach points are the most important factors in creating the local path. If the alternate goal point has any reach point which is also the reach point of the source point, the ATV has direct route to go to the alternate goal through this reach point. If not so, local goals (intermediate scan points in the local path) are further created.
- The *FindOptimumLocalGoalPt* function does this routine. It uses the case tables described in chapter 3 to decide for the most optimum alternate (local) goal. The following criteria points are taken care of to find out the most optimum alternate goal point :
  - Ø If not the source scan point itself
  - Ø If reachable (not present in any imaginary square)
  - Ø If it meets the left or right local path direction of routing

This function uses the concept of quadratic positioning of the source and the goal scan points to decide the best possible reach points. The criteria points mentioned above are implemented in the *FindOptimumReachPoint* function and used to decide the reach point preference in each case. The local goals are stored in a stack '*stackScanPtArray*' in the same sequence in which they are created from the *FindOptimumLocalGoalPt*. The process is kept in a recursion until a local goal is found whose reach point is same as the reach point of the source. Once such a local goal is found the stack (left or right) is ready to be used for navigation.

- The *OptimumPathDecision* function creates both the left and right stack as per the local path from left and right hand side respectively around the obstacle. This function then compares the stack length and accordingly selects the path with a shorter path length
- *MoveVehicle* function is called with the scan points in sequence from the selected stack.
- The local navigation function is implemented in a recursive manner. So that in case of facing an newly-discovered obstacle or imaginary square while already navigating in the first local path, its previous local path is stored on a stack and local navigation for the new obstacle is developed. Eventually the ATV routes itself to the original goal. Thereafter the ATV continues its regular path along the consecutive scan points.
- The logic for newly-discovered obstacle avoidance is same but is built run time when the newly-discovered obstacle is detected. The LIDAR data is used for newly-discovered obstacle detection. This data in coordination with the logic for creating the local path is used for newly-discovered obstacle avoidance.

Chapter 5: Simulation and Graphics Details

Simulation is the representation of the behavior or characteristics of one system through the use of another system. It is used in many contexts, including the modeling of a natural or human system, in order to gain insight into its functioning. Other contexts include simulation of technology for performance optimization, safety engineering, testing, training and education. Simulation can be used to show the eventual real effects of alternative conditions and courses of action. The implementation of simulation for this algorithm elaborates on the last sentence. Simulation of this algorithm has helped to achieve a working model for the logic behind the algorithm. In this simulation, a mathematical model of the algorithm has been depicted, primararily to give the essential visual effects for better understanding of this algorithm. Another important reason for simulating this algorithm was to verify the effect of the software output before being actually implemented on an ATV. There are various ways for implementing simulation. This chapter provides details of the simulation technique used for this algorithm, reasons behind it and the tools used for simulation and coding in general.

There are varied tools that are used for simulation graphics today. To list a few are – AC3D, Animation Master (Hash), AutoCAD (Autodesk), FreeHand (Adobe Systems), Inkscape, CorelDraw (CompuServe Incorporated) and Bryce (DAZ 3D Inc.). Few of the animation and 3D simulation graphics tools available as open source studio or suits are: Art of illusion, Blender, Breve, Cal3D, Cairo, Dia, Imgseek, Mesa, OGRE 3D, Open Scene Graph, and GIMP. These tools work based on different programs, such as Visio, GNU, Java, Visual Basic and C++.

5.1 Path of Development of the Research Along With Platform Selection

As a part of this research, the path planning algorithm was developed, implemented in software and also simulated to give its visual effect. The research initiated based on the reasons stated in chapter 2. Initially a rough scope of the entire research was planned and framed. The entire course of research started with development of a pseudo code of the algorithm. The data structure and pseudo code were scrutinized to meet the requirements of the research. Windows operating system and C language was decided upon for the software development of this algorithm. Development of this algorithm is only the first step of a huge research. Going ahead, new additions are possible in terms of algorithm, software and its actual implementation on an ATV. With all these factors Windows OS was an obvious choice, considering following main reasons:

- Windows provide several user friendly software development tools enhancing productivity of the developer.
- Specialized hardware drivers and auxiliary software tools are more readily available for Windows than for any other operating system, providing for easy hardware and software integration
- Familiarity with most of the users

  C language also became an upfront choice due to following reasons:
- Popularity, versatility and portability

- Provides dynamic memory management, interactive execution environment and interactive trace
- Provides powerful debugging facility, flexible input-output and faster execution

Dev C++ studio version 4.9.9.2 is used for software development [23]. Bloodshed Dev C++ is a fully featured Integrated Development Environment (IDE). It uses Mingw ("Minimalistic GNU for Windows") port of GNU Compiler Collection (GCC) as its compiler. Some its outstanding features are:

- Supports GCC based compilers
- Supports Windows API programming
- Integrated debugging
- Project Manager
- Quickly creates Windows, console, static libraries and DLLs
- Makefile creation
- Tool Manager
- CVS support

Dev C++ comes with everything required to compile and link, both console mode and Graphic User Interface (GUI) programs that will run on Windows. It allows the programming to be done in C as well as C++. The explanatory notes mentioned in [25] show the steps required to download, Install, Configure, Compile, Link and Run a program using Dev C++. Overall the main reason for choosing the Dev C++ was the user friendly IDE along with the GUI interface supported by it.

Using Dev C++ and using C language the basic code representing the algorithm was developed. This code was developed using the console output. Starting from navigation in a field with no obstacles, to one with known obstacle and then lastly with newly-discovered obstacles, the *printf* outputs on console depicted the exact journey to the ATV. Once the entire navigation of the ATV as per the code was found exactly in line with the proposed algorithm, the simulation part was considered. The simulation of this algorithm depicts exactly the same logic, as that done by the code with the console output. Only this time the output was in terms of a windows application giving a visual representation of the algorithm.

5.2 Graphics and Simulation Tools Used

An API (Application Programming Interface) is an interface by which the application program accesses operating system and other services. API is a set of routines, data structures, object classes and/or protocols provided by libraries and/or operating system services in order to support the building of applications. The Microsoft Windows API provides services used by all Windows-based applications [26]. It provides graphical user interface (GUI), access to system resources such as memory and devices, display graphics and formatted text, incorporate audio, video, networking, or security. A graphical user interface (GUI) is a human-computer interface (i.e., a way for humans to interact with computers) that uses windows, icons and menus and which can be manipulated by a mouse [28]. An icon is a small picture or symbol in a GUI that represents a program (or command), a file, a directory or a device (such as a hard disk or floppy). In windows, it is the Graphical Device Interface (GDI) which provides the functions and related structures that an application can use to generate graphical output for displays, printers and other devices. These functions are used to draw lines, curves, closed figures, text and bitmap images.

Programming for simulation using windows API gives the application a freedom to return a device context handle to identify a display device. Using the graphical features of windows API programming, such as Bitmaps, brushes, pens, clipping, region, windows color system and coordinate spaces, enhanced the development of simulation for this algorithm.

To create an application a compiler that runs on Microsoft windows application is required. This requirement was completely satisfied by the Dev C++ IDE since it works on windows. Almost all of the structures of Win API are C objects which further made it easy to integrate the basic algorithm representing code for simulation. With Win API, it is possible to achieve full features of animation along with timing controls. For this algorithm, the graphical elements in coordination with timing controls are used to create simulation.

5.2.1 Getting Started with Win API Programming

The following steps will brief about getting started for creating an application using Win API.

1) Open the Dev C++ IDE and create a new project in windows application.
2) C language is selected
3) The '.dev' file is saved in the respective folder
4) The default program is compiled, linked and executed to get a windows application

5.2.2 Some of the Basic Functions and Structures with Win API [29]

1) int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

WinMain is a function equivalent to main ( ) from DOS. Program starts its execution from this function. Its parameters:

HINSTANCE hInstance: Handle to the programs executable module (the .exe file in memory)

HINSTANCE hPrevInstance: Always NULL for Win32 programs

LPSTR lpCmdLine: The command line arguments as a single string

int nCmdShow: Controls how the window being built will be displayed

The object that is displayed on the screen is called as a window. Since there can be varied windows in a program, a control is required to known where they are when, and why. This main window is created using an object that can be called a class (strictly, a structure). In order to create an application a variable of either WNDCLASS or WNDCLASSEX type is required to be declared. Upon declaring a WNDCLASSEX variable, the compiler allocates an amount of memory space for it, as it does for all other variables. Various parameters of the main window such as its style, extra memory, background color and its instance are set inside this structure.

2) *CreateWindowEx( )*

Once the main window is created it is used as a parent to create child windows. This function helps to specify that a window is a child of another window. Various parameters for the child window such as its top, height, width, style, caption, and instance along with a handle of the parent class are passed as parameters.

3) *LRESULT CALLBACK MessageProcedure(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)*

To help the users with computer interaction, the operating system provides a series of objects called Windows controls. Each control creates messages and sends them to the operating system. Four main parts of information related to a message are sent to the OS:

- Identity of the object (handle or HWND) that sends a message.
- A positive natural number (UNIT) corresponding to the particular message to identify what message is being sent
- A 32 bit type word parameter (WPARAM) as an additional information for processing the message
- A 32 bit type long parameter (LPARAM) as an additional information for processing the message

Windows Procedure is actually a function pointer to manage the messages sent. Since it returns a 32 bit integer, it is called as long result (LRESULT).

To manage these messages, they are handled by a function pointer called a Windows Procedure. All the required messages are listed and processed one after another using the switch case structure in this function. Some of the basic messages (cases) used in this algorithm are:

WM_CREATE: Message to create a new window.

WM_TIMER: Message to indicate that the timer has expired

WM_PAINT: Message to request painting certain part of the application's window

WM_MOUSEMOVE: Message given to a window when the mouse moves

WM_LBUTTONDOWN: This Message is given when the left mouse button is pressed while the cursor is in client window area

WM_LBUTTONUP: This Message is given when the left mouse button is released while the cursor is in client window area

WM_CLOSE: Message to terminate the window

WM_DESTROY: Message sent while destroying a window

4) *PostQuitMessage(0)*

Message to close a window.

5) *RegisterClassEx(&WndClass)*

The window class is made available to other controls that are a part of the application using the registration function.

6) *Rectangle(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect);*

Function used to draw a rectangle. The left, top, right and bottom coordinates of the window where the rectangle is to be drawn are passed as its parameters.

7) *Ellipse(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect);*

Function used to draw an ellipse. This ellipse fits in a rectangle. The parameters of this rectangle are passed.

8) *MoveToEx(HDC hdc, int X, int Y, LPPOINT lpPoint);*

Function used to originate drawing of a line. The origin of a line is given in terms of X and Y.

9) *LineTo(HDC hdc, int nXEnd, int nYEnd);*

The end of a line is specified in terms of X and Y parameters.

10) *RGB(BYTE byRed, BYTE byGreen, BYTE byBlue);*

Three separate numerical values ranging from 0 to 255 are passed as parameters to set the required color.

11) *SelectObject():*

    This function selects a new control to a specified device context.

12) *DeleteObject():*

    Deletes a logical pen, brush, font, bitmap, region, or palette, hence setting free all system resources associated with the object. After the object is deleted, the specified handle is no longer valid

13) *TextOut(HDC hdc, int nXStart, int nYStart, LPCTSTR lpString, int cbString);*

    Writes a character string at the location given by X Y coordinates. cbString specifies the length of the string.

14) *SetBkColor(HDC hdc, COLORREF crColor);*

    Function used to highlight the text output

15) *SetTextColor(HDC hdc, COLORREF crColor);*

    Decides the color of the text output

16) BOOL BitBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth, int nHeight, HDC hdcSrc, int nXSrc, int nYSrc, DWORD dwRop);

    Transfers pixels from a specified source rectangle to a specified destination rectangle. It also alters the pixels according to the selected raster operation (ROP) code.

17) *UINT_PTR SetTimer(HWND* hWnd*,UINT_PTR* nIDEvent*, UINT* uElapse*, TIMERPROC* lpTimerFunc)*;*

    Creates a timer with the specific timeout value (uElapse)

18) *CreateSolidBrush(COLOREF crColor);*

    Creates a logical brush with the color specified to be used for painting

19) *TextOut(HDC hdc, int nXStart, int nYStart, LPCTSTR lpString, int cbString);*

Writes a character string at the specified location, using the currently selected font, background color, and text color

20) *LoadBitmap(HINSTANCE hInstance, LPCTSTR lpBitmapName);*

Loads the particular bitmap resource from the executable file for a module

5.2.3 Customization of Programming in Win API for the Algorithm

All the functions mentioned in 5.2.2 for the WinAPI controls are used along with the functions representing the basic algorithm's logic mentioned in chapter 4 to develop the simulation for this algorithm. This section elaborates how the graphical representation of the code is done using the Win API functions. Here the main intention is to display a field, a vehicle, known or newly-discovered obstacles and the navigation of the vehicle as per the algorithm. The field and known obstacles are static objects (not going to change position with respect to time throughout the execution) and hence are painted initially. Handling the coordination between the navigation related functions and Win API functions to accordingly show the required animation on the field, has been the main task of this simulation.

1) A basic windows API program is built and executed to create a window
2) The caption, length and width of the window are set in the *createwindowex* function
3) The background color for the window is set using the *windclass* function
4) A WM_PAINT switch case is added to the windows Procedure and a rectangle is drawn in the window using the 'rectangle' function.
5) As seen in the chapter of logistics, the field, LIDAR, vehicle, as well as the known obstacle parameters are taken as input from a file. The input field length and width

are used to determine the left, right, top and bottom parameters for the ‘rectangle’ function. The window application which represents a rectangular field is drawn inside the entire windows screen. An X offset and a Y offset is set to decide the top left corner of this rectangular field. To draw the width of the rectangle the width (input from read file) is added to the X offset. Similarly the length of field is added to the Y offset. Accordingly the rectangular field is positioned on the windows screen as per the offset values.

6) The ATV is displayed as bitmap image of a small car. A car bitmap image of size 32 * 32 is created and stored in the application folder with a name Car. The same image with inverted colors is stored as another bitmap image with a name CarMask. The car mask image is used to show animation when the vehicle is to be showed as maneuvering from one place to another. An image resource is required to take the bitmap as an input to the simulation. A new resource file is created called as “images.rc”. This file holds the ‘.bmp’ extensions for both the images. Initially two bitmap handles are defined for the Car and the CarMask images as hbmCar and hbmCarMask respectively.

7) WM_CREATE switch case is created to initialize various parameters of the program. The bitmaps of Car and CarMask are assigned to the bitmap handles hbmCar and hbmCarMask respectively. An object reference for the car bitmap is returned to the application. A timer is set using the *settimer* function which decides the speed of navigation of the Car in simulation. The functions to initialize variable, create scan point, reach point, from the data of known obstacles create the imaginary squares, determine the scan points inside the imaginary squares, create

alternate goal points, are all executed in this switch case. Since for this algorithm, the source for navigation in the field is the top left corner of the field, the Car Image's X and Y for the bitmap are set to the offset values.

8) The WM_PAINT switch case is primararily used to display all drawings on the window (field). The drawings include:

- Rectangle function to draw the field. Input parameters are set as Xoffset (Left), YOffset (top), Xoffset+Field width (right), Yoffset + field length (bottom).
- Using the create *solidbrush* function a color is set for the field. The select object function is used to set a new brush color for the field rectangle. Once the rectangle is drawn this object is deleted.
- Rectangles without any solid color fills are drawn around all the known obstacles to represent their respective imaginary squares. Input parameters are set as: KnownObstacle[].ImgSqrNW.X(left), KnownObstacle[].ImgSqrNW.Y(top), KnownObstacle[].ImgSqrNE.X(right), KnownObstacle[].ImgSqrSW.Y(bottom). The imaginary square corners created using the create imaginary square function are used here.
- All scan points are represented as an ellipse in the form of a circle. At all the scan point locations, determined by their respective X and Y locations, an ellipse is drawn with inputs as: (SimulationScanPtArr[].Source.X-4.0)(left), SimulationScanPtArr[].Source.Y-4.0)(top), (SimulationScanPtArr[].Source.X+4.0)(right), (SimulationScanPtArr[].Source.Y+4.0)(bottom). The '+4' and '-4' margins are provided to draw the rectangle of the ellipse around the exact scan point. A new

solid brush with a new color is set for the scan points and deleted after drawing the scan points. Similarly, a new solid brush with a different color is set for the scan points which are marked unreachable (so that all points inside any imaginary square will be seen in a different color to distinguish them as unreachable).

9) All the scan points are numbered just next to the respective circles. The textout function is used to assign numbers to all the scan points. The inputs used are: (SimulationScanPtArr [temp].Source.X-9)(X start position), (SimulationScanPtArr [temp].Source.Y-30.0))(Y start position), sVar(string), strlen(sVar)(length of the string).The scan point numbers created in the *CreateScanGoalPts* function are converted into a string and used as input. The X and Y start positions are set so that the numbers are positioned above just each scan point.

10) Known obstacles are drawn to be in the form of an ellipse. The shape of the ellipse is decided upon the X (west, east) and Y (top, bottom) coordinates of the boundaries of these obstacles given as inputs. Using a new solid brush(new color), the ellipse function is called for all the known obstacles with inputs as: KnownObstacle[].west(left), KnownObstacle[iCounter].north(top), KnownObstacle[iCounter].east(right), KnownObstacle[iCounter].south(bottom).

11) The field diagram is labeled using textout function. A particular highlight color and background color is set for this text using the SetBkColor and the SetTextColor functions.

12) Three switch cases WM_LBUTTONDOWN, WM_LBUTTONUP and WM_MOUSEMOVE are used to draw the newly-discovered obstacle run time during navigation. All of these messages are related to mouse activity. The

WM_LBUTTONDOWN message indicates windows about a regular left hand mouse click on the application window. The X and Y coordinates of the point where the mouse is first clicked are stored. The WM_LBUTTONUP message gives an indication when the mouse click on the application window is released. The WM_MOUSEMOVE message is given every time the mouse is dragged from one X-Y coordinate to another. So if a mouse is clicked on an application window and dragged to some other point on the window and then released, the last X-Y coordinate given by the WM_MOUSEMOVE message will be that for the point of release.

13) Using the above three messages a free line can be drawn on the application window whose starting and end points are recorded. Such a line is used to represent the newly-discovered obstacle. The X coordinate of all the intermediate points are considered to determine the left and right boundaries of the newly-discovered obstacle. Similarly, the Y coordinate of all the intermediate points are considered to determine the top and bottom boundaries of the newly-discovered obstacle. So a specific pattern for drawing the newly-discovered obstacle is defined such that it can be drawn in the form of a line which can extend from left to right, bottom to top or both simultaneously.

14) The WM_LBUTTONUP message also initiates drawing of a line using the *MoveTo* function. The WM_MOUSEMOVE message holds a LineTo function. So every time the line is dragged from the point it was clicked a new X-Y coordinate a new end point for the line is set. The point where the mouse is released is the last point of which the X-Y coordinates are given to the LineTo function.

15) Such a continuous line represents the newly-discovered obstacle. Because this obstacle is drawn run time, as soon as the mouse is released (the line is finished drawing) the imaginary square around this newly-discovered obstacle should be drawn. This imaginary square is drawn using the stored X, Y coordinates for the point of click and those recorded in the WM_MOUSEMOVE routine. This imaginary square is drawn in the WM_LBUTTONUP switch case when all the required X Y coordinates are already stored.
16) The X – Y coordinates at the point of click (WM_LBUTTONDOWN) are stored as the zeroth element of an array. When the line is dragged, every time a new X-Y coordinate is recorded (WM_MOUSEMOVE) it is stored as the consecutively next element of the array. Accordingly, the last element of the array is the X-Y coordinate of the point of release.
17) To draw an newly-discovered obstacle the mouse when once clicked should be dragged continuously (without releasing the *buttondown*) till the desired shape of the obstacle is drawn.
18) Both the X and Y arrays are then subjected to bubble sort in the WM_LBUTTONUP routine to arrange the elements of the array in an ascending manner. Hence, the zeroth element of the X array now holds the least X coordinate recorded throughout the trajectory of the continuous line (leftmost point of the trajectory). The last element of the X array holds the most maximum X coordinate recorded throughout the trajectory of the continuous line (rightmost point of the trajectory). The zeroth element of the Y array holds the least Y coordinate recorded throughout the trajectory of the continuous line (topmost point of the trajectory).

The last element of the Y array holds the most maximum Y coordinate recorded throughout the trajectory of the continuous line (bottommost point of the trajectory).

19) These four elements (leftmost, rightmost, topmost and bottommost) are used to define and draw the boundary of the imaginary square around the newly-discovered obstacle. The left boundary of the imaginary square for newly-discovered obstacle is set as '*Unknown_X_Arr [0] - (VehicleWidth/2)*'. Right boundary is *'Unknown_X_Arr [counterISqr-1] + (VehicleWidth/2'*. Top boundary is *'Unknown_Y_Arr [0] - (VehicleWidth/2)'*. Bottom boundary is *'Unknown_Y_Arr [counterISqr-1] + (VehicleWidth/2)'*.

20) Using the MoveToEx and the LineTo functions a rectangle representing the imaginary square is drawn around the newly-discovered obstacle with the above parameters. The scan points inside the imaginary square for newly-discovered obstacles are marked as unreachable and their respective new goal points are created.

21) The main task in simulation is to show an animated version of the algorithm for the maneuvering of the ATV. This algorithm implements the animation using the timer basis. Every time a timer value is expired the ATV position is updated. That is, its earlier position is erased, a new position is assigned to the ATV and the ATV image is drawn at the new location. This timer value decides the speed of maneuvering of the ATV (car) in the simulation.

22) After every certain time interval the ATV is moved to its next goal scan point. The WM_TIMER switch case takes control of such timer related functions. This

message indicates expiry of the timer and hence a signal to move to the ATV to a new scan point. Three main functions are called upon this timer expiry: *EraseCar* (), *UpdateCar* () and *DrawCar* ().

23) The *EraseCar* () function creates a small rectangle as per the width and height of the car image and fills this rectangle with a solid brush color. For this simulation, this color is kept same as that of the field. Accordingly when the car image is erased its positioned will be seen as a blank on the field.

24) The *UpdateCar* () function is the most important function to represent the path planning navigation in simulation. In this function, the logically next position where the car image is supposed to be displayed (next position of the ATV as per the algorithm) is decided. The navigate field function elaborated in chapter 4 (logistics) is implemented here. The move vehicle or the local path navigation function is called based on the conditions of reach ability of the next point.

25) The *DrawCar* () function initially allocates a block of computer memory that would hold the bitmap image. The CarMask and the Car bitmap objects are selected and the corresponding pixels are transferred from the memory (source) to the application (destination).

26) Since the computation of the new ATV position (in the navigate field function) takes much lesser time than that required for the timer to expire and call the update Car function, by the time the update car function is called the new location of the car Image is ready to b drawn. This pattern of simulation continues till the ATV reaches the final destination of the field.

Chapter 6: Study Heuristics

Given certain conditions, simulation results are always idealistic, unless otherwise provided with some known errors. This chapter analyzes and tabulates the results of the algorithm development along with its deviation, if any, from the expected results. Also, the heuristics behind this research will be elaborated.

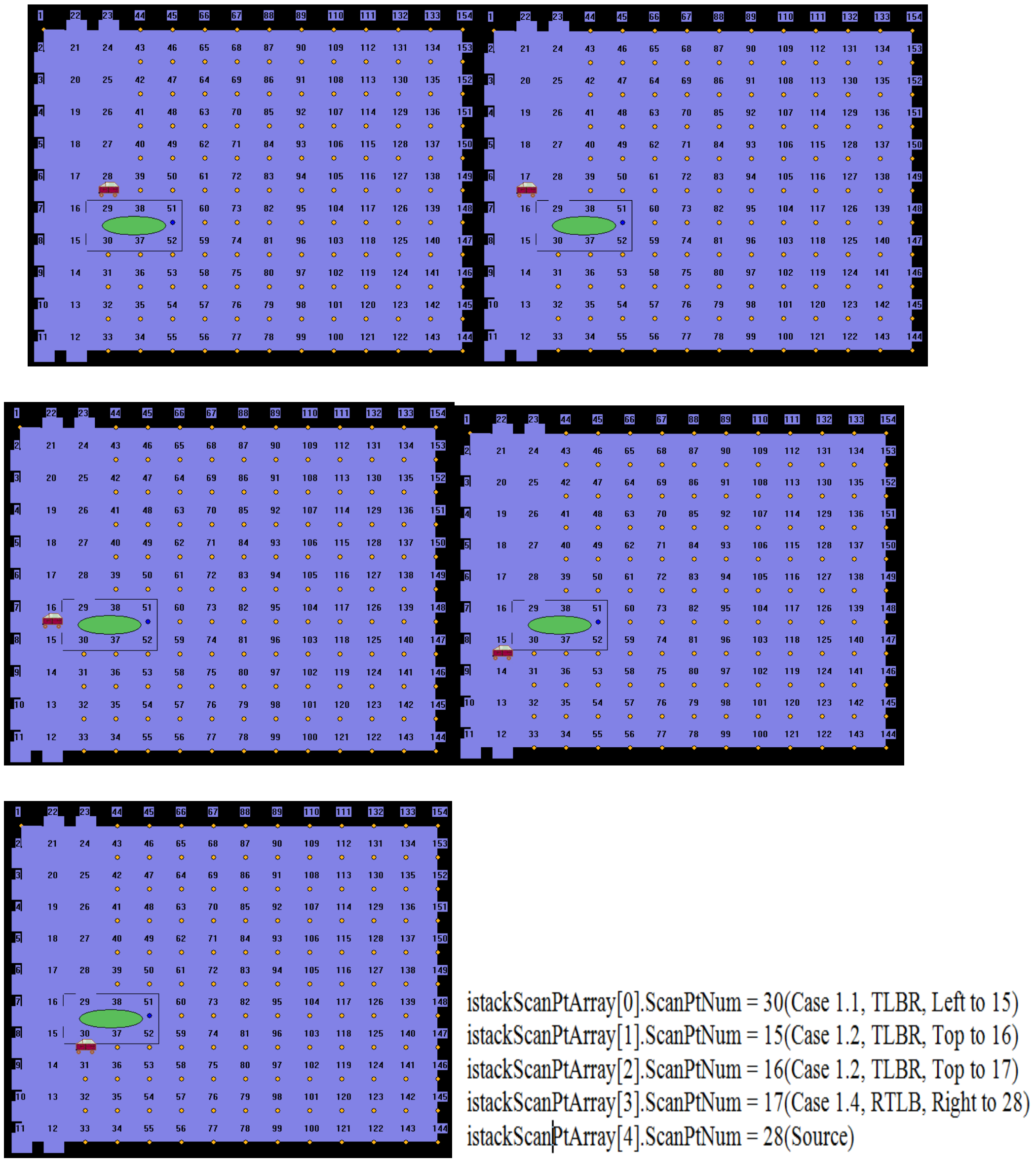
The main challenge while starting this thesis has been to develop a new path planning algorithm, to implement it in software and to simulate it. To do this, it was required to organize the study in such a manner so as to build a clear evidence for the working model of the algorithm. The path planning concepts in various papers studied for this research had some direct contrasts and similarities. Development of an altogether new method for path planning, making it executable and proving that this method is optimum to implement has been the main task.

Software Output

Testing all the different cases in the both the quadratic case tables mentioned in chapter 3 (algorithm) has been one of the major tasks of this research. These tests validated that if the obstacles are positioned at different locations along the ATV's trajectory, the solution comes under one of the 16 cases in the table. Using the preference solution for that case, the navigation is maintained as per algorithm.

Following are the results for a few of the test cases:

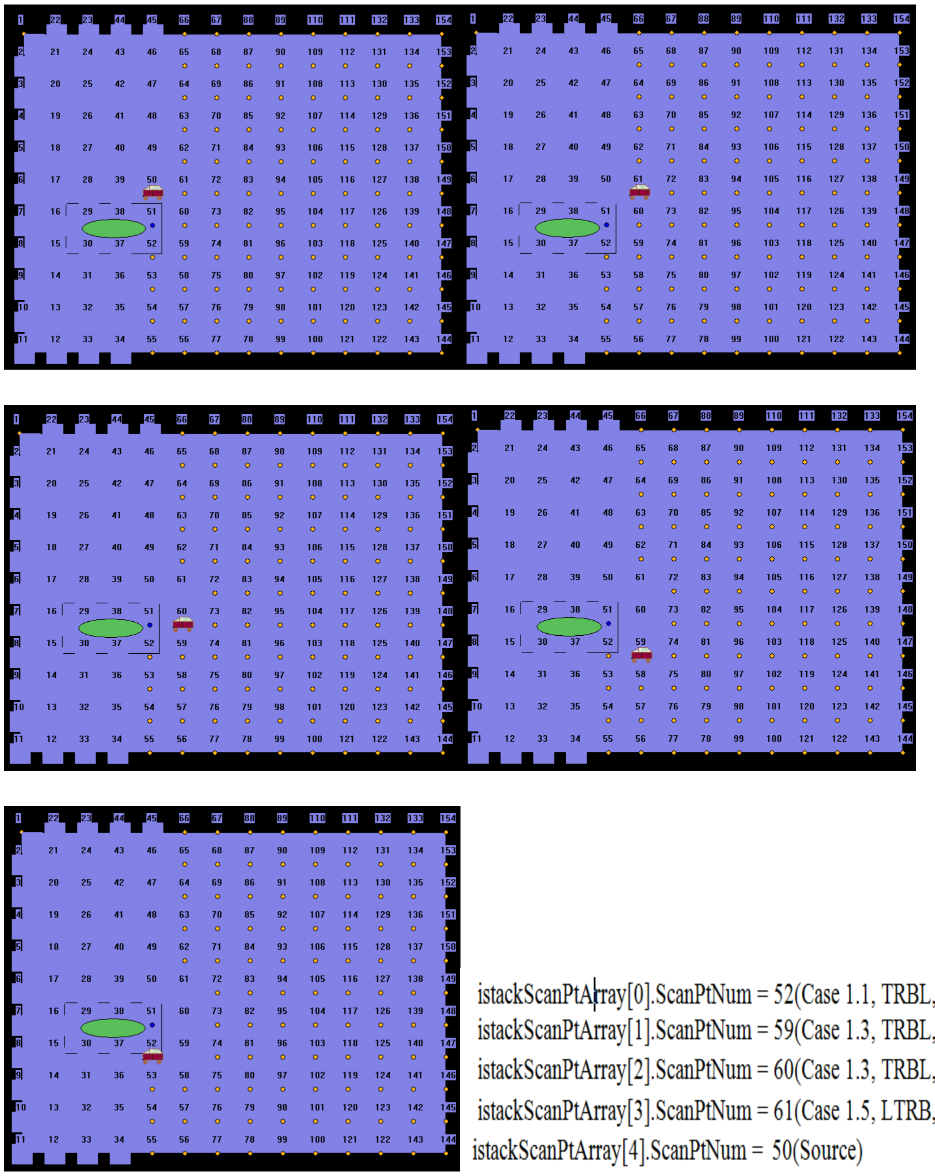
1) Source 28 and goal: 30 (Base case 1). The local path is backtracked from 30 to 28 from left hand side. The stack counter for right hand local path is found to be more (9) compared to the left hand stack counter (5). This can be verified from the simulation output as seen in Figure 6-1.



istackScanPtArray[0].ScanPtNum = 30(Case 1.1, TLBR, Left to 15)
istackScanPtArray[1].ScanPtNum = 15(Case 1.2, TLBR, Top to 16)
istackScanPtArray[2].ScanPtNum = 16(Case 1.2, TLBR, Top to 17)
istackScanPtArray[3].ScanPtNum = 17(Case 1.4, RTLB, Right to 28)
istackScanPtArray[4].ScanPtNum = 28(Source)

Figure 6-1: Simulation output of different scan points when going around the obstacle (left hand path case)

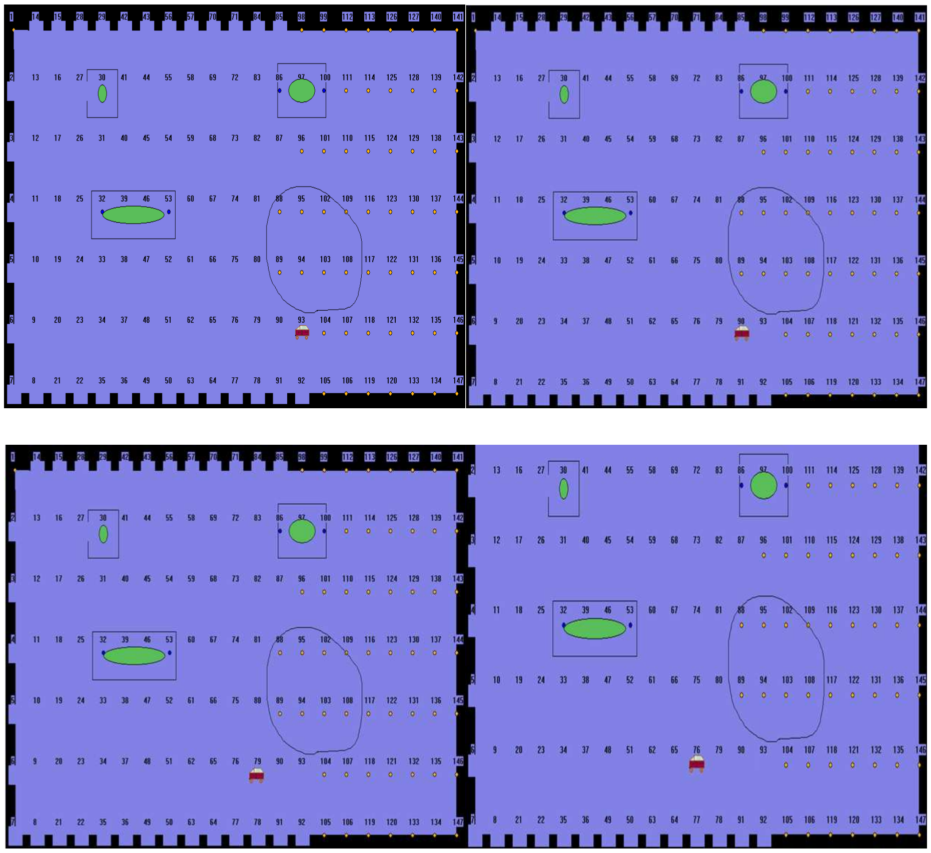
2) Source: 50 and goal: 52(Base case 1). The local path is backtracked from 52 to 50 from right hand side. The stack counter for right hand local path is found to be less (5) compared to the left hand stack counter (9). This can be verified from the simulation output as seen in Figure 6-2.

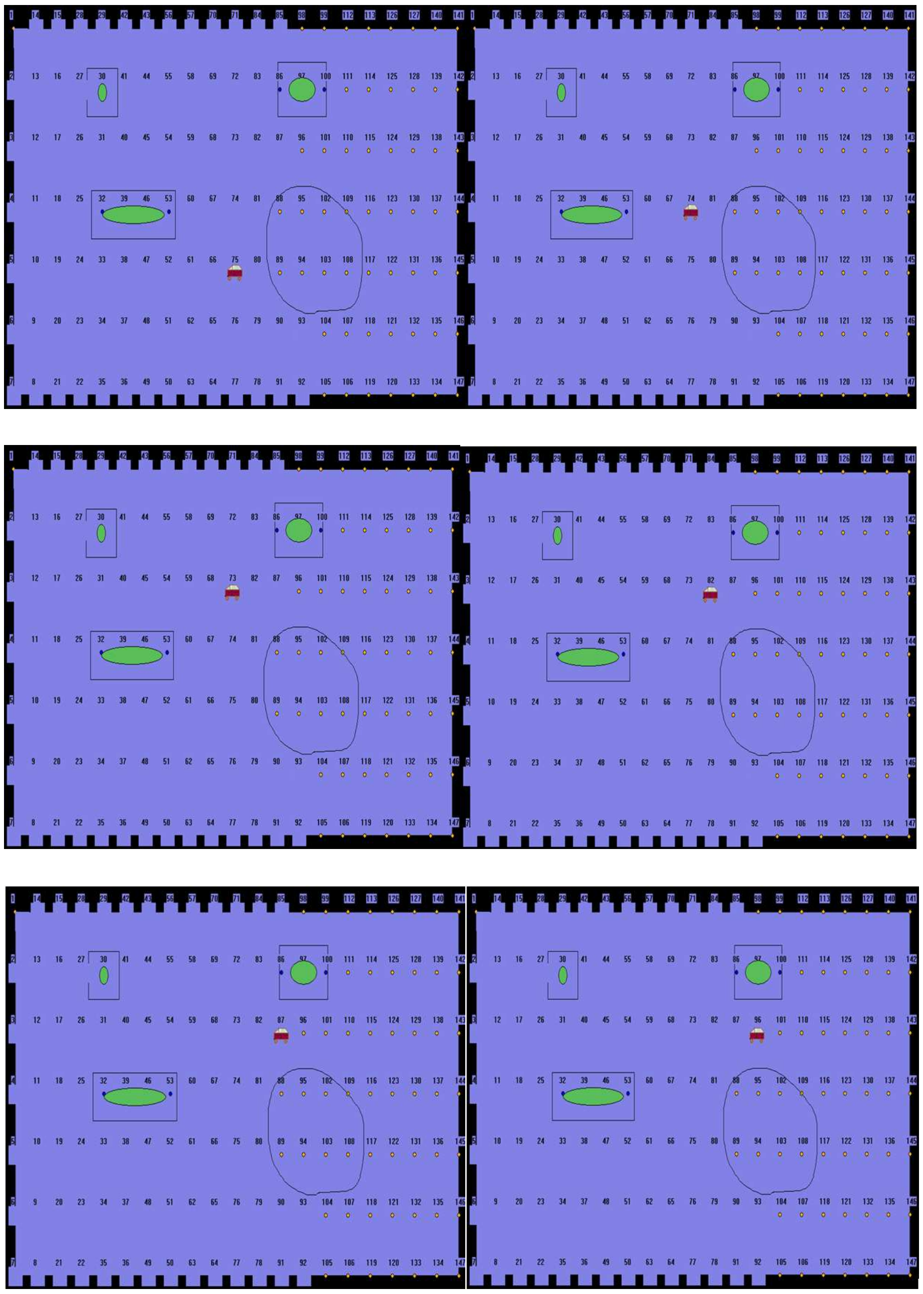


```
istackScanPtArray[0].ScanPtNum = 52(Case 1.1, TRBL, Right to 59)
istackScanPtArray[1].ScanPtNum = 59(Case 1.3, TRBL, Top to 60)
istackScanPtArray[2].ScanPtNum = 60(Case 1.3, TRBL, Top to 61)
istackScanPtArray[3].ScanPtNum = 61(Case 1.5, LTRB, Left to 50)
istackScanPtArray[4].ScanPtNum = 50(Source)
```

Figure 6-2 : Simulation output of different scan points when going around the obstacle (right hand path case)

3) Source: 93 and goal: 96(Base case2). The local path is backtracked from 96 to 93 from left hand side. The stack counter for left hand local path (10) is found to be less compared to the right hand stack counter (12). This can be verified from the simulation output as seen in Figure 6-3.





```
istackScanPtArray[0].ScanPtNum = 96(Case 2.6, LTRB, Left to 87)
istackScanPtArray[1].ScanPtNum = 87(Case 2.8, BLTR, Left to 82)
istackScanPtArray[2].ScanPtNum = 82(Case 2.8, BLTR, Left to 73)
istackScanPtArray[3].ScanPtNum = 73(Case 2.8, BLTR, Bottom to 74)
istackScanPtArray[4].ScanPtNum = 74(Case 2.8, BLTR, Bottom to 75)
istackScanPtArray[5].ScanPtNum = 75(Case 2.8, BLTR, Bottom to 76)
istackScanPtArray[6].ScanPtNum = 76(Case 2.4, RBTL, Right to 79)
istackScanPtArray[7].ScanPtNum = 79(Case 2.4, RBTL, Right to 90)
istackScanPtArray[8].ScanPtNum = 90(Case 2.4, RBTL, Right to 93)
istackScanPtArray[9].ScanPtNum = 93(Source)
```

Figure 6-3 : Simulation output of different scan points when going around the obstacle (base case 1)

Chapter 7: Conclusion and Future Work

An optimum path planning algorithm serves as the basic structure for the navigation of an autonomous ATV. The work is aimed towards navigation in open fields with occasional obstacles. The algorithm and its software have been simulated to give real world application. The following conclusions can be drawn from the presented algorithm:

1) The algorithm presented fulfills its purpose of navigation in a specific pattern, obstacle avoidance and optimum routing
2) All known and newly-discovered obstacles are avoided during navigation so that the vehicle can return quickly to original navigation path
3) The algorithm and the implemented software can be easily customized as per changes in requirements
4) No complex computation and mathematics is involved thus making it easy for future implementations
5) With the concept of an imaginary square around each obstacle, the algorithm ensures that the ATV will keep safe distance from obstacle when travelling around it.
6) A field of any dimension along with any number of obstacles (of any dimensions) can be used as input in the simulation to validate the algorithm

7) The simulation depicts an appropriate representation of the algorithm and its software. It is found to be beneficial to verify the output of the algorithm before being actually implemented on an ATV

As stated in earlier chapters this research is only a part of a large research program which will allow an ATV to navigate in any given terrain. This research creates a base for the navigation part of the entire work. The algorithm presented is so that its software can be used for implementation on an ATV. The presented software has been aimed towards simulation, and hence, would require some obvious extra mechanical and electrical inputs so as to allow it to be used on an ATV. The following points are the possible future work for this algorithm and system:

1) Addition of functionality for inputs from sensors such as LIDARS, GPS and other electrical inputs
2) Addition of functionality for outputs to motor drives and other mechanical ATV parts related to navigation
3) Addition of filters and probabilities to avoid noise effects when actually implemented on an ATV
4) Provision to make the navigation fault tolerant in presence of erroneous sensors or faulty maps
5) Detection of predicted motion and actual motion of the ATV and accordingly adjust the algorithm run time
6) Modification of the coordinate system and units as per the sensor inputs
7) Modification of the algorithm and the software for moving obstacles and multiple ATVs maneuvering simultaneously in the field

8) Modification of the algorithm and the software for any uneven terrain of any shape
9) Ability to get information about height or depth of the obstacle and to make the ATV capable of crossing the obstacle instead of going around it

Bibliography

1. Hugh Durrant-Whyte and Tim Bailey. JUNE 2006. Simultaneous Localization and Mapping - Part I. IEEE Robotics & Automation Magazine 99 TUTORIAL.

2. Hugh Durrant-Whyte and Tim Bailey. JUNE 2006. Simultaneous Localization and Mapping – Part II. IEEE Robotics & Automation Magazine 99 TUTORIAL.

3. Sebastian Thrun, Dieter Fox and Wolfram Burgard. May 1998. Probabilistic Mapping of an Environment by a Mobile Robot. IEEE International Conference on Robotics & Automation Leuven, Belgium.

4. Sebastian Thrun. Robotic Mapping: A Survey. Research sponsored by DARPA's MARS Program (Contract number N66001-01-C-6018) and the National Science Foundation (CAREER grant number IIS-9876136 and regular grant number IIS-9877033).

5. Makarenko, A.A. Williams, S.B. Bourgault, F. Durrant-Whyte, H.F. 2002. An experiment in integrated exploration. Intelligent Robots and System, IEEE/RSJ International Conference.

6. Cyril1 Stachniss and Wolfram Burgard. October 2003. Mapping and Exploration with Mobile Robots using Coverage Maps. IEEURSJ Intl. Conference on Intelligent Robots and Systems Las Vegas. Nevada.

7. Michael Montemerlo, Sebastian Thrun, Daphne Koller and BenWegbreit. 2002. Fast SLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association. National Conference of Association for the Advancement of Artificial Intelligence.

8. http://msdn.microsoft.com/en-us/robotics/default.aspx

9. Microsoft robotic software used for simulation.

10. Borenstein, J. and Koren, Y. May 1990. Real-time Obstacle Avoidance for Fast Mobile Robots in Cluttered Environments. Robotics and Automation, IEEE International Conference.

11. Dieter Fox, Wolfram Burgardy, Frank Dellaert, Sebastian Thrun. 1999. Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. Sixteenth National Conference on Artificial Intelligence (AAAI-99), Orlando, Florida.

12. Wolfram Burgard and Dieter Fox and Daniel Hennig and Timo Schmidt. 1996. Estimating the Absolute Position of a Mobile Robot Using Position Probability Grids. Fourteenth National Conference on Artificial Intelligence (AAAI-96).

13. W. S. Wijesoma, K. W. Lee and J. Ibañez-Guzmán . April 2005. Motion

Constrained Simultaneous Localization and Mapping in Neighborhood Environments. IEEE International Conference on Robotics and Automation Barcelona, Spain.

14. Jean-Claude Latombe. 1991.Robot Motion Planning. Book, Edition: 2, Published by Springer.

15. http://inventors.about.com/gi/dynamic/offsite.htm?site=http://prime.jsc.nasa.gov/ROV/library.html

16. List of few of the varied typed of robots used currently.

17. Colin R. McInnes. 2000. Autonomous Path Planning for On-Orbit Servicing Vehicle. Department of Aerospace Engineering, University of Glasgow, Scotland, UK.

18. Subbarao Kambhampati and Larry S. Davis.1986. Multiresolution Path Planning for Mobile Robots. IEEE Journal of Robotics and Automation, Vol. RA-2, no3.

19. Hoi-Shan Lin Jing Xiaot and Zbigniew Michalewiczt. 1994. Evolutionary Algorithm for Path Planning in Mobile Robot Environment Evolutionary Computation, IEEE World Congress on Computational Intelligence.

20. C. I. Connolly, J. B. Burns, R. Weiss. March 6, 1990. Path Planning Using Laplace's Equation. Robotics and Automation, IEEE International Conference. Amherst, MA.

21. Kikuo Fujimura and Hanan Samet. February 1999. A Hierarchical Strategy for Pat Planning Among Moving Obstacles. IEEE transactions on robotics and Automation, Vol.5, no. 1.

22. JCrGme Barraquand, Bruno Langlois, and Jean-Claude Latombe. 1992. Numerical Potential Field Techniques for Robot Path Planning. IEEE transactions on Systems, Man and Cybernetics, Vol.22, no. 2.

23. J. Sanjiv Singh and Megnad D. Wagh. April 1987. Robot Path Planning using Intersecting Convex Shapes: Analysis and Simulation. IEEE Journal of Robotics and Automation, Vol. FA-3, no. 2.

24. Peter D. Holmes and Erland R. A. Jungert. May 1992. Symbolic and Geometric Connectivity Graph Methods for Route Planning in Digitized Maps. IEEE transactions on Pattern analysis and Machine intelligence, Vol. 14, no. 5.

25. http://www.bloodshed.net/devcpp.html

    Dev C++ IDE used for software development.

26. http://bloodshed-dev-c.en.softonic.com/

Source to Download Dev C++

27. http://csjava.occ.cccd.edu/~gilberts/devcpp5/

    Explanatory notes for getting started with Dev C++

28. http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx

    Microsoft Windows API details

29. http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx

    Microsoft Windows API reference

30. http://www.linfo.org/gui.html

    GUI Definition and information

31. http://www.functionx.com/win32/index.htm

    Win API details

32. Dieter Fox, Wolfram Burgardy, Frank Dellaert, Sebastian Thrun. 1999. Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. Sixteenth National Conference on Artificial Intelligence (AAAI-99), Orlando, Florida.

33. Sebastian Thrun, Wolfram Burgard, Dieter Fox. April 2000. A Real-Time Algorithm for Mobile Robot Mapping with Applications to Multi-Robot and 3D Mapping. IEEE International Conference on Robotics and Automation, San Francisco.

34. http://imr.felk.cvut.cz/Research/mapping/mapping.html

    List of current map building methods