TEMPORAL POTENTIAL FUNCTION APPROACH FOR PATH PLANNING IN DYNAMIC ENVIRONMENTS

by

VAMSIKRISHNA GOPIKRISHNA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2008

To my mother, my father, my teachers and God.

Those who have shaped me into what I am today.

ACKNOWLEDGEMENTS

First and foremost I have to thank my parents and my family for their unwavering faith in my capabilities. Without their support – both emotional and financial – my graduate studies would have been just a pipe dream.

Secondly, I must express my gratitude towards my graduate advisor Dr. Manfred Huber for believing in me and for his invaluable guidance during my thesis research. His encyclopedic knowledge of the subject and immeasurable patience, have made this thesis what it is today. It has been an honor and privilege to work with him over the past two years.

Next, I need to thank the members of my committee, Dr. Gergely Zaruba and David Levine for devoting their time and energy to me in this process.

I would like to thank all of my friends and colleagues, both here at UTA and at the other universities for their support and their help. Specifically, I owe thanks to Srividhya Rajendran, John H.C. Staton, Binu Matthew and Dr. Darin Brezeale for their tips and advice. I would also like to thank Giles D'Silva, not only for sharing his tips for thesis writing with me but also for helping me maintain my sanity for the last frantic months of my thesis research.

I would also like to thank The University of Texas at Arlington's Computer Science and Engineering Department for the most productive and enjoyable two years of my academic life. I would also like to thank Srinivasan T. of Sri Venkateshwara College of Engineering, Dr. Suraishkumar G.K. and Dr. Deepak Khemani of IIT Madras for setting me down path of research.

Last but not the least, I want to thank God. Without his grace, none of this would have been possible.

November 25, 2008

ABSTRACT

TEMPORAL POTENTIAL FUNCTION APPROACH FOR PATH PLANNING IN DYNAMIC ENVIRONMENTS

Vamsikrishna Gopikrishna, M.S.

The University of Texas at Arlington, 2008

Supervising Professor: Manfred Huber

A Dynamic environment is one in which either the obstacles or the goal or both are in motion. In most of the current research, robots attempting to navigate in dynamic environments use reactive systems. Although reactive systems have the advantage of fast execution and low overheads, the tradeoff is in performance in terms of the path optimality. Often, the robot ends up tracking the goal, thus following the path taken by the goal, and deviates from this strategy only to avoid a collision with an obstacle it may encounter. In a path planner, the path from the start to the goal is calculated before the robot sets off. This path has to be recalculated if the goal or the obstacles change positions. In the case of a dynamic environment this happens often. One method to compensate for this is to take the velocity of the goal and obstacles into account when planning the path. So instead of following the goal, the robot can estimate where the best position to reach the goal is and plan a path to that location. In this thesis, we propose such a method for path planning in dynamic environments. The proposed method uses a potential function approach that considers time as a variable when calculating the potential value. This potential value for a particular location and time indicates the probability that a robot will collide with an obstacle, assuming that the robot executes a random walk from that location and that time onwards. The robot plans a path by extrapolating the object's motion using current velocities and by calculating the potential values up to a look-ahead limit that is determined by calculating the minimum path length using connectivity evaluation and then determining the utility of expanding the look-ahead limit beyond the minimum path length. The method is fast, so the path can be re-planned with very little overhead if the initial conditions change at execution time. This thesis will discuss how the potential values are calculated and how a suitable look-ahead limit is decided. Finally the performance of the proposed method is analyzed in a simulated environment.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

There are two approaches to determining how a robot should move in a given environment. In the path planning approach the robot first calculates, based on the obstacle locations, what the best path to the goal location is and then follows it. Path planning generally assumes that the planner has all pertinent information about the world at execution time. If the world suddenly changes, then there is no other option than to scrap the plan generated and plan a new one from scratch. In a reactive control system the robot sets off towards the goal and then modifies it's movements to avoid any obstacles it may encounter. This makes sure that the robot does not need to spend time re-planning in case the environment changes. However the tradeoff is in performance. There is a chance that the robot will end up just tracking the goal, ignoring certain optimal ways to reach the goal.

The environment that a robot is expected to work in can be either static or dynamic. In a static environment the state of the world, i.e. the locations of the goal and obstacle states are fixed. An example for such a static environment is a robot having to navigate in a room with no people in it. The only obstacles are the furniture whose locations are fixed. In a dynamic environment the goal and obstacle locations can change as in the case of a robot having to navigate a busy hallway. The people in the hallway can be considered as obstacles, each of them having their own velocities and trajectories.

In dynamic environments, robot control is usually achieved by means of a reactive system. This is because a path planning system would have to constantly re-plan as the state of the environment keeps changing. This adds significant overhead to the execution, frequently making the use of a reactive system more efficient. However, the tradeoff is in accuracy. Since the robot only knows the location of the robot, the goal, and the obstacles at that instant, it could end up just tracking the goal, following it around the world instead of reaching it.

The goal of this thesis is to propose a method by which path planning can be used in dynamic environments. The method has its roots in Harmonic functions and the potential field approach to path planning. The proposed method selects a certain look-ahead value up to which it plans a path. It then extrapolates the positions of the goal and obstacles up to that time and then calculates a potential value for the remaining locations of the world at each time step up to that look-ahead. The potential value of a particular location at a particular time is the probability that the robot will collide with an obstacle if it takes a random walk from that point and that time onwards. In the case of a goal location, it is zero. In the case of an obstacle location it is one. In the other locations it is the average of the potential value of the current location's possible successor locations. Since our environment is dynamic, the successors of a particular location are the nearby locations and the current location all at the next time step. The potential values are calculated working back from the look-ahead to the initial time step. After calculating the potential values for all locations, the robot follows the path from the start state along the highest negative gradient to get to the goal. The value of the look-ahead selected must be high enough that a solution is found if one exists. The first solution we get may be too risky i.e. the goal could be surrounded closely by obstacles etc. It is possible that a better solution exists which can be found by extending the look-ahead. A way of evaluating a good look-ahead value, using the performance of randomly generated worlds with similar properties of the main environment is discussed in this thesis. For this, the graph connectivity method is used to find the first solution and the potential of the starting location if the look-ahead had been set in such a fashion as to find just the first solution world is calculated. This is compared with the generic potential value generated from random worlds following the same characteristics as the given world. Based on these comparisons, a proper look-ahead value can be decided upon.

The advantage of this method is that it finds the best path within the current look-ahead limit. This is under the assumption that the optimality of the path is determined by the potential values of the locations in the path, where the potential value represents the probability that a robot at that location at that time will collide with an obstacle. The use of graph connectivity ensures at least the shallowest solution will be found. And the comparison with the potential values of the randomly generated worlds allows us to select a good look-ahead value.

The remainder of this thesis is organized in the following fashion. Chapter 2 covers the basic concepts that are utilized in this thesis. It also looks at some of the research done with regards to path planning in dynamic environments. Chapter 3 will present a description of the temporal potential function approach to path planning in dynamic environments. Chapter 4 will discuss the details and module descriptions for an experimental implementation of the proposed method. Chapter 5 will discuss the observations made from the experimental implementation. Chapter 6 will conclude the thesis and discuss possible future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 The Path Planning Problem

2.1.1 Definition of the Problem

The robot navigation or path planning problem is one in which an agent has to navigate an environment containing a number of obstacles to reach a goal location [1]. The path planning problem usually considers the following items: An initial state i.e. the state the robot is initially in, a final state or goal state, i.e. the state the robot has to be in, the state space of the robot, the set of possible actions that the robot can take and a cost function to calculate the efficiency of the path. The state space of an agent is the set of all possible states that the agent can be in. For a robot the velocity and location information can be considered to be its state information. The cost function is a measure that determines the attractiveness of a particular solution. It could be the time taken or the number of turns. A path planner using harmonic functions does not require a start state, and the potential values it calculates are the cost measure as they determine the robustness of the solution.

2.1.2 Applications of Path Planning Algorithms

An algorithm that solves the path planning problem has widespread applications in various areas of industry like robotics, manufacturing plants, drug design, medical surgeries, aerospace applications, warfare and video games. Any real world scenario where an agent has to navigate an environment filled with obstacles can be considered a path planning problem. For example path planning algorithms are used to calculate the motion of robotic arms used in an assembly line. Motion planning software developed by the Fraunhofer Chalmer’s Center is used by the Volvo Cars (in Torslanda, Sweden) assembly plant for the sealing process of their car bodies using programmed robots to function automatically [2] [3]. The motion planning software developed at Kineo CAM is used in the automotive assembly task to insert and remove a windshield wiper motor from a car body cavity [2]. In video games path planning algorithms are used to determine the movement of game characters and objects. Commercial robots like Honda's ASIMO [4] and Sony's AIBO [5] use path planning algorithms to determine how to navigate the world. Path planning algorithms are used by GPS based Satellite Navigation systems to find the best route to a given destination. Path planning algorithms could also be used to simulate vehicles moving at very high speeds involving dynamic constraints, uncertainties and obstacle avoidance. Planning algorithms have also been used in computational biology to solve the docking problem which requires determining if flexible molecules can insert themselves into a protein cavity. Probabilistic Roadmap motion planning techniques (Used in robot motion planning) have also proven successful at studying protein folding pathways and potential landscapes [6].

2.2 Static and Dynamic Environments

The environment a robot needs to work in can be either static or dynamic. In a static environment the variables that define the world are fixed. So the positions of the various objects in the environment need to be given to the robot only once. In a dynamic environment, the locations of the obstacles and goals can change as time passes. So it is necessary for the robot to continually update its representation of the environment. Robot navigation in a dynamic environment has a few issues that need to be addressed. The robot must keep track of all the objects in a dynamic environment. If using a path planner or any other navigation system that needs the obstacle locations before deciding on a navigation strategy, the navigation strategy must be recalculated every time they change. One workaround is to include the dynamics of the environment in its model. Thus the path planner can take into account the motion of the obstacles and the goal when planning a path.

2.3 Modeling the Environment

For any agent to navigate the world it has to have a representation or map of the environment. The map must contain the locations of the various obstacles in the world and must be updated depending upon a change in the environment. To generate a map of the environment, the robot uses sensors to find out the locations of the obstacles in the world and converts that information into an internal representation. Some of the commonly used approaches to modeling the environment are Grid Based Maps, Topological Maps and Feature maps.

2.3.1 Grid based maps

Grid based maps are metric maps that divide the world into grid cells, each of which contains information about the environment [7]. The cell could be an obstacle, free space or a goal. Gird based maps are easy to construct and maintain as their resolution is independent of the complexity of the environment. However the time and space complexity increases as the environment size increases. When navigating an environment represented by a grid based map, the robot has to make corrections for slippage and drift. This problem can be addressed by using dead-reckoning and localization techniques. Grid based maps have been used in the experimental implementation of the proposed method. A grid based representation of an environment with eleven point obstacles and a goal state is shown in Figure 2.1. The red points represent the obstacles and the green point represents the goal state.

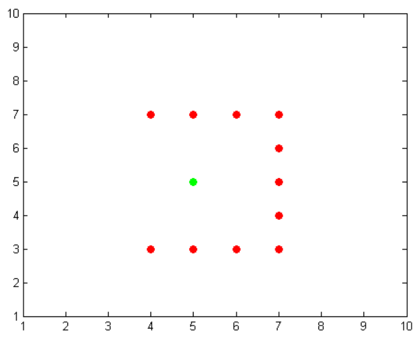


Figure 2.1 Grid based representation of a two dimensional world.

2.3.2 Feature Maps

Feature maps [8] contain representations of the typical features of the local environment of the agent. These features include the agent, straight walls and polyhedral objects. The agent continuously observes these features for local referencing. Sonar or laser sensors can be typically used to build feature maps. SLAM techniques along with kalman filters and particle filters can be used to create feature based maps on-line. These methods provide localization of the agent at virtually any point in the local environment. Their complexity grows with larger environments and number of map objects.

Some of the issues with creating feature maps arise due to limited sensor range, limited field of view, occlusions and noisy data [8]. One method to overcome these is to use Probabilistic frameworks for localization.

Figure 2.2 shows the feature map generated [8] for the Belgioioso Castle in Italy. A manually controlled Pioneer robot (from University of Freiburg) explored this environment with a trajectory of 228m in 16 minutes and 27 seconds. The 227 feature map with an error vector of 576 components was built in 3 minutes 16 seconds.

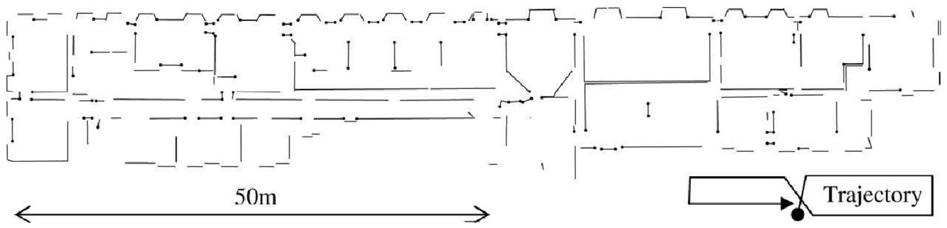


Figure 2.2 Feature Map of Belgioioso castle [8]

2.3.3 Topological Maps

Topological maps are simplified maps that represent the environment using a graph based approach [7]. Nodes in the graph represent important landmarks like doors, entry and exit points etc, and Arcs represent that a non obstructed direct path exists between them. Topological maps are often built upon metric maps by partitioning them into regions separated by critical lines that join critical points. Critical lines can be considered to be doorways and hallways and critical points can be the exit and entry points in the map. There are a few drawback of using topological maps as they suffer from incorrect recognition in situations where places look alike or if the same place has been sensed from different viewpoints by taking different paths. The time complexity for constructing and maintaining topological maps increases as complexity increases. This is due to the increase in the number of critical lines or arcs.

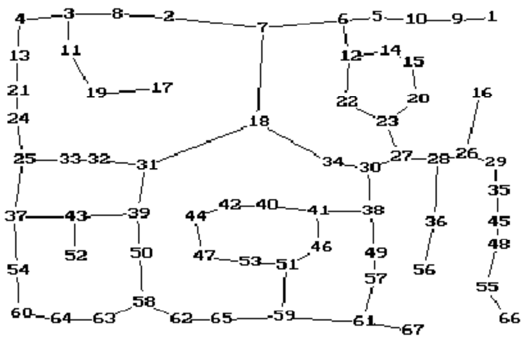


Figure 2.2 Topological graph [7]

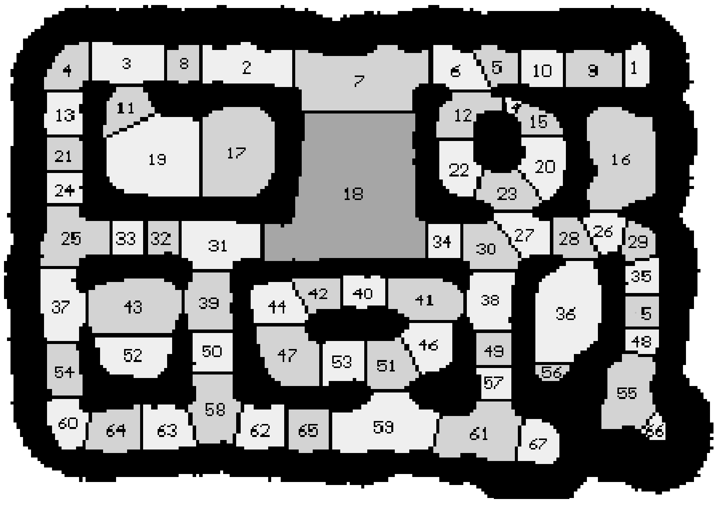


Figure 2.3 Topological region with critical lines [7]

2.3.4 Extending the representation in time.

To capture the dynamics of the environment one approach that can be used is to extend the world in the third dimension. The positions of the obstacles and goal locations are extrapolated up to a certain look-ahead value.

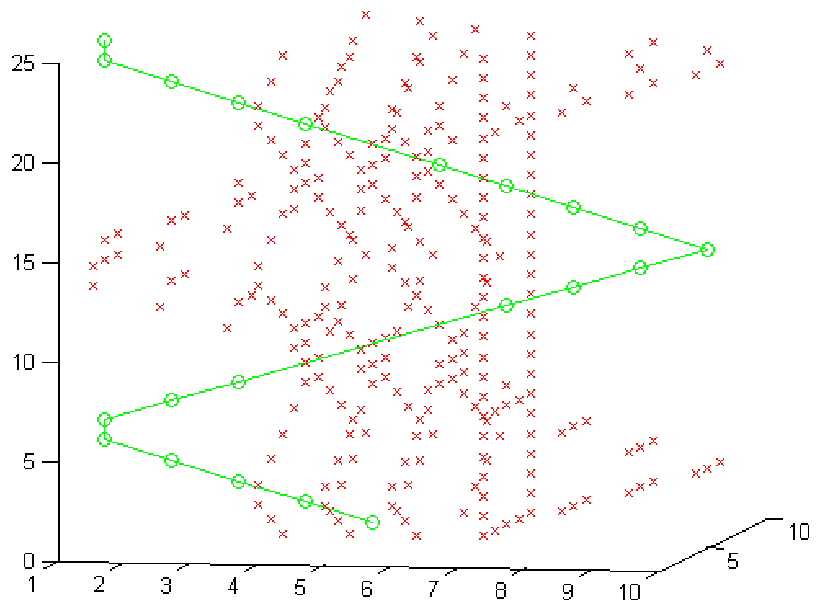


Figure 2.4 Grid based representation of a two dimensional world extended in time

2.3.5 Configuration Space

Configuration space [9] or c-space is the space in which the agent can freely move. C-spaces provide abstraction for complex environments as the robot is reduced to a single point. The number of dimensions of c-space is the degrees of freedom of the agent. The configuration space C is given by

$$C = C_{free} \cup C_{obs} \tag{2.1}$$

where C_{free} is the space where the robot can move freely and C_{obs} is the space occupied by the obstacles and is defined by

$$C_{obs} = \bigcup_{i=1}^{q} C_{obs_i} \tag{2.2}$$

where q is the number of obstacles in this space.

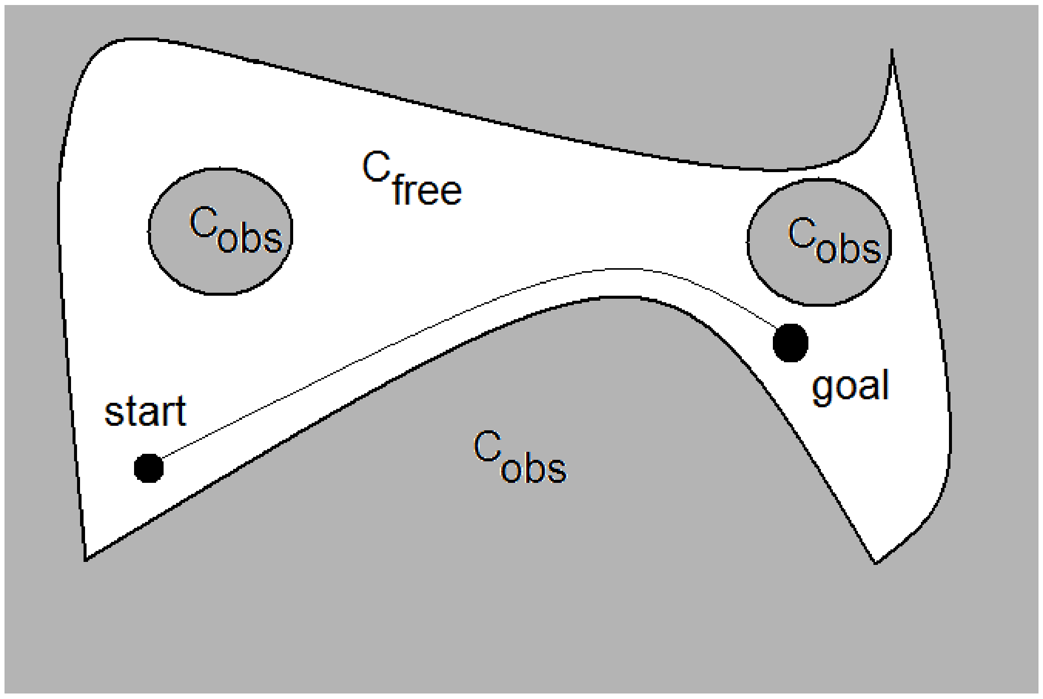


Figure 2.5 Configuration Space

2.4 Dynamic motion planning problem

The dynamic motion planning problem is one in which the robots has to navigate an environment where one or more of the obstacles in the system are moving [10]. Unlike in the static environments the problem can not be solved by merely constructing a geometric path. Instead a continuous function of time specifying the robot's configuration at each instance needs to be generated. One approach is to add a dimension – time – to the robot's configuration space [10]. But unlike the other two, the time dimension is irreversible. The algorithms used for planning in static environments can be used in dynamic environment with some modifications and extensions, which are aimed at taking the specificity of the time dimension into account.

2.5 Path planning vs. Reactive Systems

In a path planning system the path to the goal is usually calculated before the robot sets off. The robot has to follow the given plan to reach the goal. In a dynamic environment the velocities of the obstacles and the goal are taken into account when calculating the goal. If any of the variables describing the environment change during the planning process, the path has to be re-planned. This could result in a lot of overhead if the planning algorithm is going to be used in an environment where the dynamics of the environment constantly change.

In a reactive system, there is no actual planning taking place. The robot moves towards the goal, only changing course to avoid any obstacles it may encounter. In a static environment this will result in the robot reaching the goal. However if the environment is a dynamic one then the robot may end up tracking the goal. If the robot, at each time step tries to reach the goal location then it will end up following the goal across the world instead of actually taking the steps to reach it. This is known as the tracking problem. One solution would be to predict when the robot can reach the goal and the location of the goal at that time and get to that location. This can not be done by a reactive system. However a path planning system can be easily modified to handle this.

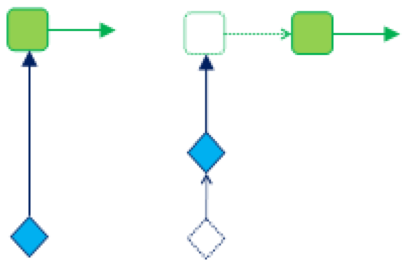


Figure 2.6 The tracking problem

Consider Figure 2.6. The robot (represented by the blue diamond), moves to the current location of goal (represented by the green square). But the square has moved away. This process could possibly repeat forever making sure that the robot never reaches a goal state but ends up tracking it.

A potential field approach combines characteristics of both the path planning system and the reactive system. The potential values are calculated before the robot sets off but no path is calculated. The robot uses the calculated potential values to avoid the obstacles and reach the goal as in a reactive system. To avoid the tracking problem the potential of the obstacle or goal can be specified as a function of time.

2.6 Potential fields and Harmonic Functions

2.6.1 Potential Fields

Potential fields have been proposed for obstacle avoidance by Khatib et al [11]. In the potential field approach the robot in configuration space is acted upon by imaginary forces. The goal produces an attractive force and the obstacles produce a repulsive force. The path the robot follows can be found by calculating the resultant vector of these forces.

The field of artificial forces $\vec{F}(q)$ in c-space are produced by a differentiable potential function $U: C_{free} \rightarrow \mathbb{R}$ where

$$\vec{F}(q) = -\vec{\nabla} U(q) \tag{2.3}$$

Here $\vec{\nabla} U(q)$ represents the gradient vector of U at q. We take the negative as we are performing a gradient descent on the potential field. This is because the obstacles are set at maximum and goals to a minimum potential value.

$$U(q) = U_{attr.}(q) + U_{repl.}(q) \tag{2.4}$$

and

$$\vec{F} = \vec{F}_{attr.} + \vec{F}_{repl.} \tag{2.5}$$

where

$$\begin{aligned} \vec{F}_{attr.} &= -\vec{\nabla} U_{attr.} \\ \vec{F}_{repl.} &= -\vec{\nabla} U_{repl.} \end{aligned} \tag{2.6}$$

define the attractive and repulsive forces.

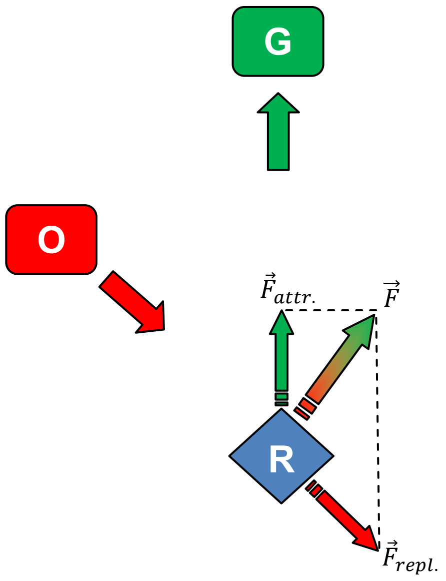


Figure 2.7 Forces due to potential fields

Planners that use potential fields have the chance of getting stuck in local minima instead of reaching the goal. Figure 2.8 shows a function with both local and global minima and maxima. If the below image was the potential function the planner could become stuck in the local minima instead of reaching the goal at the global minima.

A number of approaches have been suggested to solve the local minima problem. One solution is to use vector field histograms (VFH) [11] [12] [13]. We can also place an artificial obstacle at local minima [14]. There has also been some research into designing new potential functions that will avoid the creation of local minima [15] [16]. One such method is to design the potential function as a Harmonic Function.

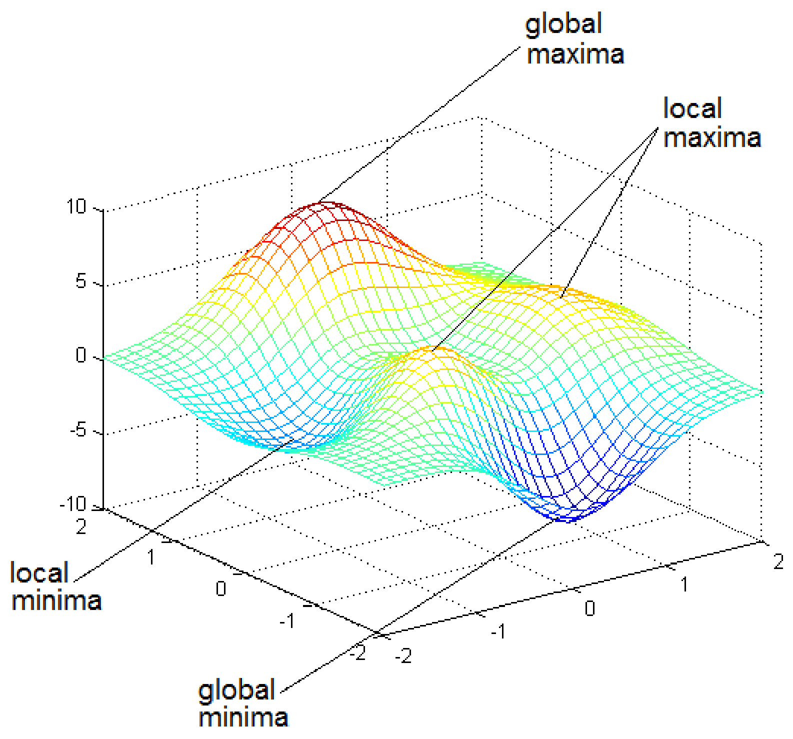


Figure 2.8 Local and Global Minima

2.6.2 Harmonic Function

A harmonic function is a real function with continuous second partial derivatives which satisfies Laplace's Equation [17]. Laplace's equation [18] is the partial differential equation

$$\nabla^2 \phi = 0 \tag{2.7}$$

∇^2 is known as the Laplacian. It is the sum of the second derivatives of the given function. A function ϕ defined on a domain $\Omega \subset \mathbb{R}^n$is said to be harmonic if it satisfies the equation

$$\nabla^2 \phi = \sum_{i=1}^{n} \frac{\partial^2 \phi}{\partial x_i^2} = 0 \tag{2.8}$$

For example, if ϕ is a functions of variables x and y the above equation can be written as

$$\nabla^2 \phi(x, y) = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \tag{2.9}$$

Typically we are given a set of boundary conditions [19] and we need to solve for the (unique) scalar field ϕ that is a solution of the Laplace equation and that satisfies those boundary conditions. Some of the commonly used boundary conditions are discussed in Sub-Section 2.6.6

2.6.3 Characteristics of Harmonic Functions

A Harmonic function's gradient path forms a smooth curve with no local maxima and minima. There may be a few saddle points [20] from which the exit can be found by searching in the neighborhood for a negative gradient. The gradient vector of a harmonic function has zero curl so a gradient descent on this vector always leads to a global minimum. The curves instantaneously tangential to the velocity vectors of the trajectory points (called streamlines) are smooth for any point along the trajectory [2] [21]. Figure 2.9 shows the harmonic function $f(x, y) = x^3 - 3xy^2$.

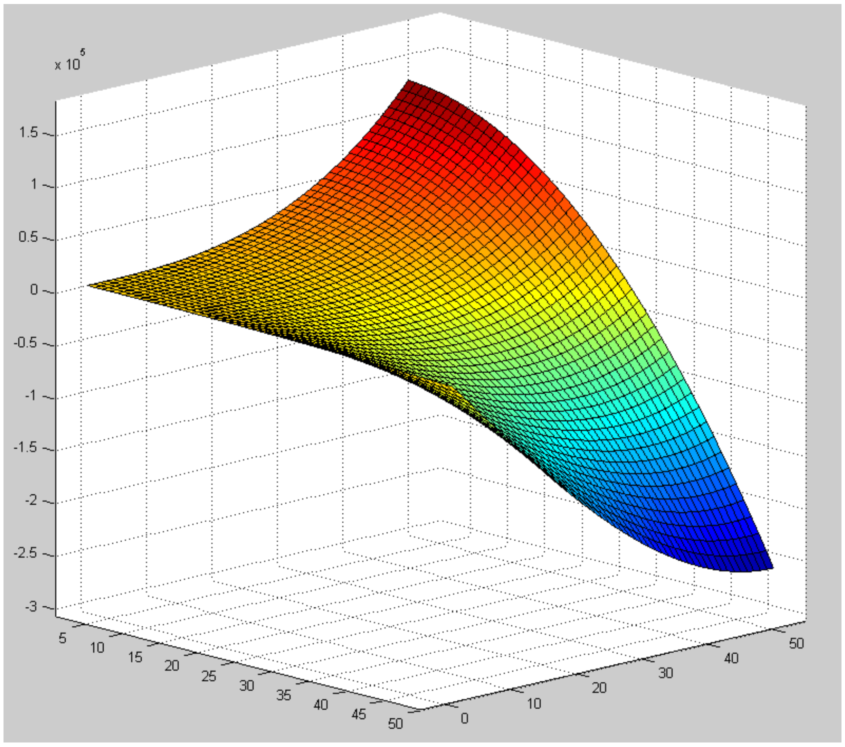


Figure 2.9 The harmonic function $f(x, y) = x^3 - 3xy^2$

2.6.4 Numerical Solutions of Laplace's Equation

Techniques like Jacobi iteration, Gauss-Seidel iteration or successive over-relaxation can be used to calculate the numerical value of the harmonic function at each grid location [21]. These methods require the environment to be discretized, for example by representing it as a grid based map. Although these methods compute function values on a grid, multi-linear functions are used to interpolate between grid points since such functions are harmonic and smooth.

2.6.5 Relaxation Methods

Jacobi iteration for Laplace's Equation replaces every non-boundary value for ϕ with the average of its neighbors' values simultaneously

$$\phi^{(k+1)}\left(x_i, y_j\right) = \frac{1}{4}\left[\phi^{(k)}\left(x_{i+1}, y_j\right) + \phi^{(k)}\left(x_{i-1}, y_j\right) + \phi^{(k)}\left(x_i, y_{j+1}\right) + \phi^{(k)}\left(x_i, y_{j-1}\right)\right] \tag{2.10}$$

Here k is the iteration number. The Jacobi iteration method generally requires a higher number of iterations to converge than Gauss-Seidel or SOR. It is however very effective on SIMD architectures.

Gauss-Seidel iteration is similar to Jacobi except that the iteration numbers for neighboring values are mixed.

$$\phi^{(k+1)}\left(x_i, y_j\right) = \frac{1}{4}\left[\phi^{(k)}\left(x_{i+1}, y_j\right) + \phi^{(k+1)}\left(x_{i-1}, y_j\right) + \phi^{(k)}\left(x_i, y_{j+1}\right) + \phi^{(k+1)}\left(x_i, y_{j-1}\right)\right] \tag{2.11}$$

Successive Over-Relaxation (SOR) converges more rapidly than Gauss-Seidel or Jacobi Iteration. It is therefore one of the most popular methods used. The recurrence relation is given by,

$$\phi^{(k+1)}\left(x_i, y_j\right) = \phi^{(k)}\left(x_i, y_j\right) + \frac{\omega}{4}\left[\phi^{(k)}\left(x_{i+1}, y_j\right) + \phi^{(k+1)}\left(x_{i-1}, y_j\right) + \phi^{(k)}\left(x_i, y_{j+1}\right) + \phi^{(k+1)}\left(x_i, y_{j-1}\right) - 4\phi^{(k)}\left(x_i, y_j\right)\right] \tag{2.12}$$

Here k is the iteration number and ω is the relaxation factor. The value of ω depends upon the properties of the coefficient matrix and determines the speed of convergence. These iterations are repeated until the change in values between iterations drops below a residual value.

2.6.6 Boundary conditions

Solutions to Laplace's equations can be computed using certain conditions called boundary conditions. The commonly used ones are restricted forms of boundary conditions, namely the Dirichlet and Neumann conditions [21].

In the Dirichlet boundary condition the potential of the boundary is set to a constant. This constant is the maximum value in the configuration space. Since all obstacles are also represented by a constant maximum, the potential flow is outward normal to the obstacle surface. The gradient of the function tends to depart from the C-space obstacle boundaries. The Dirichlet boundary condition is represented as

$$\phi|_{\partial\Omega} = c \qquad (2.13)$$

where c is a constant and Ω is the domain.

In the Neumann boundary condition the velocity vectors are forced to be tangential to the obstacle boundary surface. This may cause the agent to stay close to the obstacle surfaces, which may not be preferable in some cases. Since the Neumann condition still requires a source and a sink for the flow, the outer boundary of the C-space map is used as a source. The Neumann condition is represented as

$$\left.\frac{\partial\phi}{\partial\underline{n}}\right|_{\partial\Omega} = 0 \qquad (2.14)$$

One can superpose the Dirichlet and Neumann solutions to obtain a harmonic function that exhibits a behavior somewhere between the two. If ϕ_D is the Dirichlet solution and ϕ_N is the Neumann solution, both in domain Ω, then the new harmonic function can be constructed by taking a linear combination of the two.

$$\phi = k\phi_D + (1-k)\phi_N \qquad (3.1)$$

where $k \in [0,1]$ is the superposition constant. The resulting ϕ is harmonic, has no local minima and guarantees collision free paths.

2.6.7 Harmonic Functions in Path Planning

Harmonic functions have many varied applications in robot control. The most popular application is their use in path planning. If the c-space of the environment can be modeled as a harmonic function, then all the robot has to do is to follow the negative potential gradient to reach the goal and avoid the obstacles. The absence of local maxima and minima and the continuity and differentiability of the harmonic functions guarantees that a planner using harmonic functions generates paths that are correct and complete. Since the gradient is smooth, the paths generated by the planner are well behaved. The negative gradient computed has no local maxima or minima as the potential of a grid location is the average of its neighbors. Thus the only types of critical points that can occur are saddle points [20]. Saddle points are stationary points but not extrema. To escape a saddle point all that the agent has to do is explore the neighborhood for a region of negative gradient. Using harmonic functions we can build a system that combines the characteristics of both a path planning system (since it calculates the potential values of the locations in the world) and a reactive system (since the robot simply follows the path of decreasing potential and avoids the obstacles) [2] [10] [12] [21] [22].

Harmonic functions have also been used along with other methods like probabilistic cell decomposition [23] and probabilistic roadmaps [24] to build path planners. They have also been used to build exploratory systems [25] [26] and to build real time obstacle avoidance systems [27].

2.7 Path planning in Dynamic Environments

Current research in path planning for dynamic environments consists of two approaches. One approach is to build an effective reactive system that moves towards the goal, deflecting from the path when an obstacle comes towards it. One such method was proposed by Ge and Cui of University of Singapore [28]. Their proposed method used a potential field approach. The potential functions they used were defined as follows. The attractive potential was a function of the relative distance and velocity between the robot and the goal. Thus the attractive force acting on the robot is the negative gradient of the attractive potential in terms of both position and velocity. Each obstacle has an influence range specified. The repulsive potential of each obstacle is a function of the relative velocity between it and the robot and the shortest distance between its body and the robot provided that the robot is heading in its general direction (relative velocity is a positive vector) and the robot is within the obstacle's influence range. The repulsive force exerted by each obstacle on the robot is the negative gradient of the obstacle's repulsive potential in terms of both position and velocity. The above described method has a few disadvantages. By definition, the repulsive potential is an undefined value if the distance the robot will travel before coming to a complete stop is greater than the distance between the robot and the obstacle. So there may occur a situation where there is no way to avoid a collision. The robot can also be trapped in local minima. In that case, the robot just waits at that position till the obstacle moves away. If that does not happen for a long time, the planner uses conventional local minimum recovery methods like wall following to escape local minima. Finally there is a possibility that the goal may sometimes be in the obstacles influence radius. In that case the robot will never reach it. One solution is to modify the repulsive potential function to include the relative position and velocity between the goal and the robot. A similar approach has been proposed by Poty, Melchior and Oustaloup of Université Bordeaux [29].

Another approach to path planning in dynamic environments was proposed by Wu, QiSen, Mbede and Xinhan [30]. This method calculates potential field values using harmonic functions and then follows path of negative potential gradient, using fuzzy rules to avoid obstacles.

Finally we could include time as another dimension of the c-space (although a limited one as movement is only possible in one direction). The proposed method discussed in the next chapter belongs to this approach.

CHAPTER 3

TEMPORAL POTENTIAL FUNCTION APPROACH

3.1 Overview

In this chapter we will look at the proposed method for planning a path in a dynamic environment. The method proposed is based on the harmonic function approach and the potential field method of path planning. The way the planner works is as follows. It first extrapolates the world to a certain look-ahead time. This look-ahead time is determined by the following process. The time taken to reach the first goal is found by forward chaining. If the dynamics changes during the forward chaining due to solution not being found for a long time, the process is repeated. Using the time taken to reach the first goal the look-ahead value is estimated based on observation of the potential value curves of a set of randomly generated worlds with the same properties as the given world. These curves are generated during the first execution of the planner. After finding the look-ahead limit the planner calculates the potential values of all the locations in the world at all time steps from the start to that look-ahead time. The robot reaches the goal by simply following the negative potential gradient at each time step. If the robot ends up at an obstacle due to an error in navigation, the potential values are recalculated. If the dynamics of the environment i.e. the velocities of the obstacles or goals changes during runtime, the process is repeated form the start. Once the goal is reached we repeat the process for a new goal location. The entire planning process is illustrated by Figure 3.1.

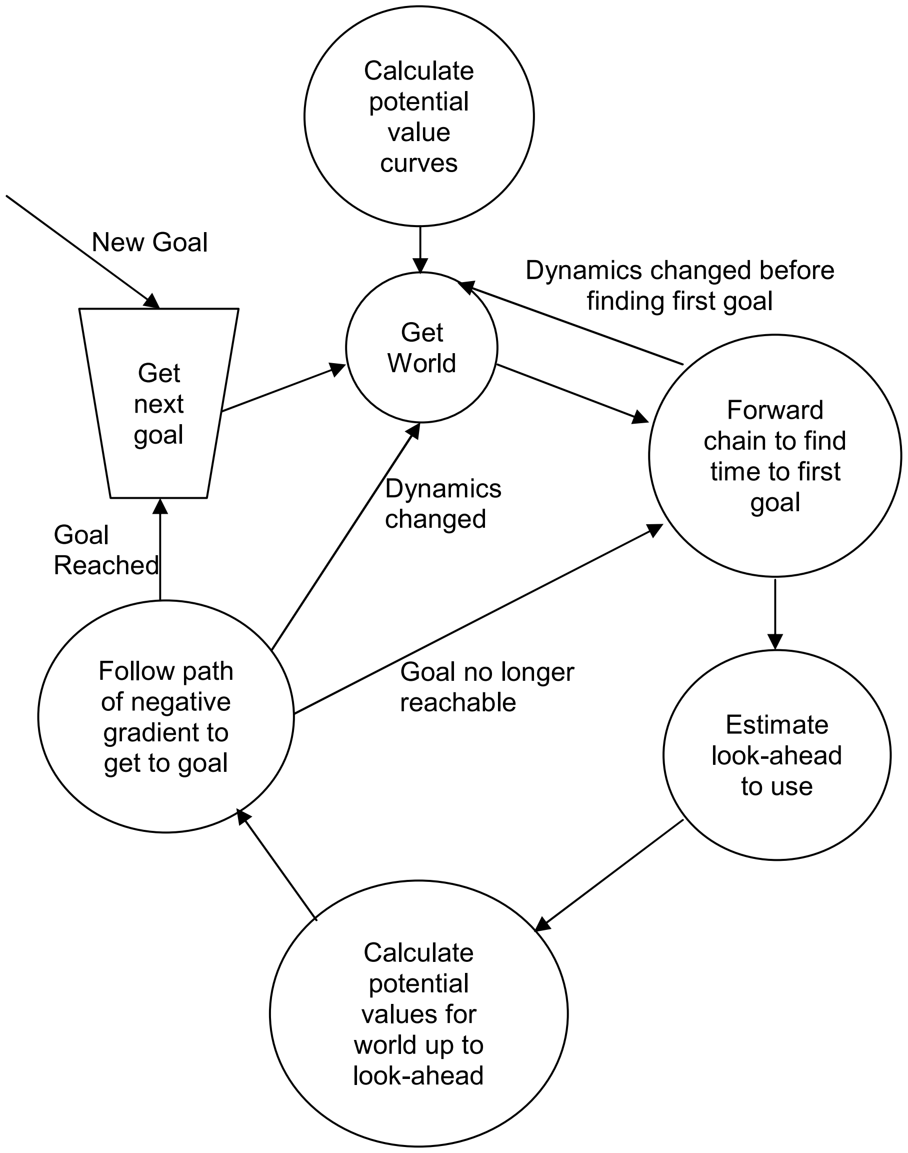

Calculate potential value curves
New Goal
Get next goal
Get World
Dynamics changed before finding first goal
Forward chain to find time to first goal
Dynamics changed
Goal Reached
Follow path of negative gradient to get to goal
Goal no longer reachable
Estimate look-ahead to use
Calculate potential values for world up to look-ahead

Figure 3.1 The Temporal potential function approach.

3.2 Potential Value Calculation

In the potential field method each location in the world is assigned a potential value. In this case the potential value is the probability of a robot at that location colliding with an obstacle if it takes a random walk from that point. This probability is zero for the goal location and one for an obstacle location. For other locations it is calculated by taking the average of the potential values of the neighboring locations. The potential value of a particular location at time t is given by the average of the potential values of that location's successors at time t+1, as shown in Equation 3.1

$$p(x,y,t) = \frac{1}{|S(x,y)|}\left[\sum_{s\in S(x,y)} p(s,t+1)\right] \tag{3.1}$$

The set S is the set of the successors of a particular state i.e. the locations that can be reached in the next time step. For a robot moving at a fixed velocity they are the locations that it can move to in the next time step. The current location itself will become a neighbor in the next time step as the option of just staying at the current location without moving is open to the robot. Thus the set of neighbors or successors S of a location x, y can be defined as the set of the locations that the robot can reach by moving from the current state at a predetermined velocity and the current state itself. This set can also be used to describe a location's predecessors, i.e. the set of possible locations the robot could have been in during the previous time step. The potential of a location, i.e. the probability of a robot at location x, y colliding with an obstacle if it begins a random walk at time t is the average between the potential values of the successors of the location at time t+1 where the successors are the locations the robot can reach in time t+1 and the current location. For example in a world where the robot can move up, down, left and right with a velocity of one grid location per time step, the potential value can be calculated as illustrated in Figure 3.2. As can be seen from the figure we need the values of a location's future neighbors before we can calculate the current probability. This implies that the potential values have to be propagated from the future backwards. As a result a maximum duration or look-ahead value has to be selected in order to facilitate the potential function calculator. For the entire world, we work backwards from the look-ahead time step, calculating the potential value for each location in the world from the values calculated in the previous time step until the initial step is reached.

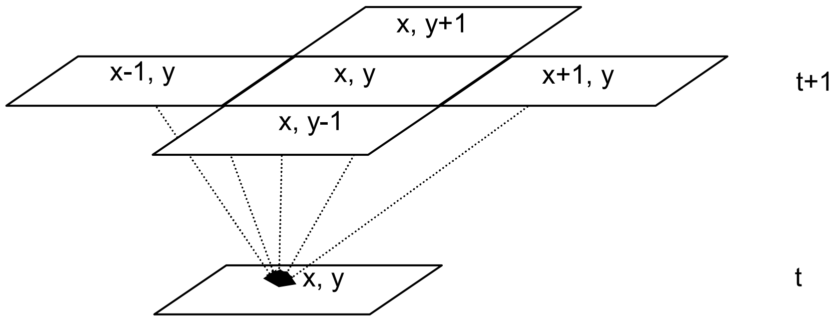


Figure 3.2 Calculation of Potential Value at time t

Once the potential values have been calculated for the entire world the robot then moves from the starting location moving to the neighbor with the lowest potential value at each time step. In effect it follows the negative gradient while moving along the positive time axis. This is repeated until the robot reaches a goal, which indicates a success or the number of steps taken equal the look-ahead, which indicates failure. If the dynamics of the environment (the goal and obstacle velocities) change during run time, then we need to re-plan the path. The selection of the look-ahead value is an important step as without an accurate look-ahead the planner may settle for a sub-optimal solution or not find a solution at all. We will next look at the method with which we can estimate what look-ahead to use.

3.3 Look-ahead value determination

The initial look-ahead value is determined by checking graph connectivity where the world is considered to be a directed acyclic graph. This is done by starting at the initial location and moving ahead one time step at a time to the current location's neighbors. If any of the neighbors is a goal then the number to steps it took to reach it is noted. If the neighbor is a goal or a node that has already been visited then there is no need to expand it. Otherwise the neighbors of the node are expanded next. This is illustrated by Figure 3.3.

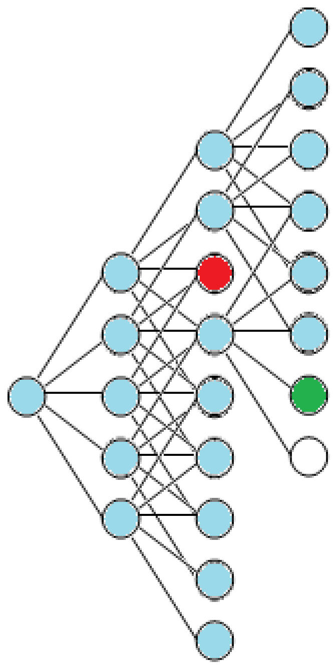


Figure 3.3 Forward propagating to find goal

In the above figure, the blue nodes are the visited nodes, the red nodes are obstacles and the green node is the goal. The nodes are expanded in a manner similar to a breadth first search [1]. Only after all the nodes in a current time step are processed are its descendents processed. This ensures that the first possible solution is obtained.

The look-ahead is initially set to the number of steps needed to find the first goal. Using this we determine the potential of the start state by the method described in the previous section. This potential value is then compared to the expected potential value curve derived for a set of randomly generated environments with the same obstacle density as the current world. We find out the look-ahead value where the current potential value would be obtained in a random world, la_1. We also find the look-ahead value, la_2, where the likelihood of the potential value being below the starting potential value of the first solution drops below a certain percent. This percentage value can be a fixed value for the entire curve or one that increases as the look-ahead increases. We then calculate

$$\begin{aligned} \alpha &= la_2 - la_0 \\ \beta &= la_2 - la_1 \end{aligned} \tag{3.2}$$

where la_0 is the number of the steps after which the first solution was found by forward chaining.

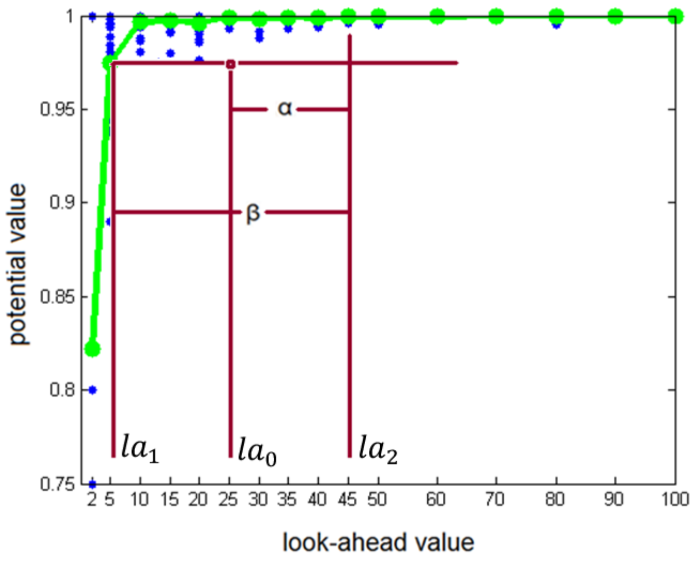


Figure 3.4 Calculating α and β for original look-ahead.

The new look-ahead value is simply the old look-ahead value incremented by α or β, whichever is greater.

3.4 Generating Potential value curves

The potential value curves have to be generated before the planner can be used. These curves have to only be generated once and can then be used for all the runs of the planner, provided that the overall size and characteristics of the world do not change. The first step is to select an obstacle density. The next step is to generate a set of random worlds. These worlds are generated by creating random obstacles until the required obstacle density is obtained. For each of these worlds, a look-ahead value is taken. One of the non obstacle locations in the final time step before the look-ahead is converted into a goal and the potential values of the world are calculated. One of the initial states that are reachable from the goal is chosen as the start state. The potential value for this state is noted. The process is repeated for a number of randomly generated worlds and for various look-ahead values. The values obtained are plotted with the look-ahead values on the x axis and the potential values of the start states on the y axis. Finally the mean values for the potential values for each look-ahead value are calculated and joined by a curve. One such curve obtained is given in Figure 3.5. The blue dots represent the potential values for each random world and look-ahead combination and the green dots represent the means of each set of potential values, grouped by look-ahead times. We also draw a line joining the mean values (potential value curve) for the sake of visualization. These curves are generated for various obstacle densities and are used by the planner as and when needed.

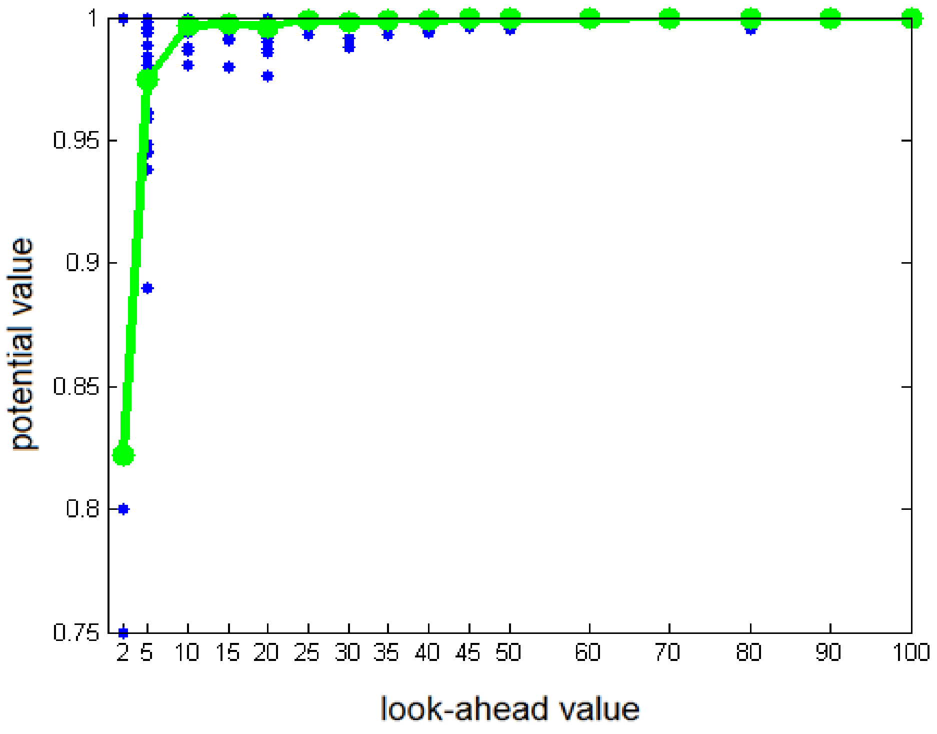


Figure 3.5 Potential Value Curve

In the next chapter we will look at an experimental implementation of the above discussed methods. We will also see some of the issues faced when implementing such a planner.

CHAPTER 4

EXPERIMENTAL IMPLEMENTATION OF PROPOSED METHOD

4.1 Implementation Details

In this chapter we shall discuss the various modules that together, make up an experimental implementation of the proposed method. The language used to implement the method is MATLAB [31]. We shall discuss the program flow in each of the modules and how the data is transferred between the modules. We will also discuss some of the issues that were faced during the implementation process and how they were addressed.

The planner implementation consists of eleven modules which are discussed in the following sections. Section 4.2 will describe the module to generate random worlds. Section 4.3 will describe a module which finds all start states reachable from a given goal state by means of graph connectivity. Section 4.5 will describe the module which calculates the potential values of all the locations in the world. Section 4.6 describes the module which uses the above described modules to generate the potential value curves. Section 4.7 describes a module by which the given two dimensional world is converted into a three dimensional model, where the third dimension is time. This module is used by the forward chaining module described by Section 4.8 to find the first reachable goal location. The module described by Section 4.9 uses the time taken to reach that goal location and the potential value curves obtained from the module described by Section 4.6 to calculate a suitable look-ahead value. The look-ahead value obtained is used by the module described in Section 4.7 to create a three dimensional model of the environment and the locations in this environment have their potential values calculated by the module described in Section 4.5. Once the potential values have been calculated, the module described in Section 4.10 tries to find a path by navigating through locations of least potential value at each increasing time step. If such a path is found then it is displayed by means of the module described in Section 4.11. All the above steps are executed by the module described in Section 4.12 which forms the main planner.

4.2 Random World Generation

We shall now take a look at the module used to generate the random worlds used by the module used to create the potential value curves, described in Section 4.6. First we calculate the exact number of obstacles the world must contain from the density and size information provided. Then we place that many obstacles in the world. For each of these obstacles we select a random direction and velocity according to a given distribution. The obstacles are placed at random locations in the world. The module returns a list of obstacles with their position and velocity information. The worlds generated by the above module will have a set of point obstacles, each with its own initial position and velocity information.

4.3 Backward Chaining to find start state for random goal

This module is used by the module in Section 4.4 to find one of the initial states from which the given goal state can be reached. It uses the concept of graph connectivity to find the set of nodes which are connected to the goal node. We create an empty three dimensional array to flag the nodes already visited. We set the goal node flag to ‘visited’ and add it to a node queue. The following steps are then repeated until the node queue is empty. We de-queue the first node from the queue. We then check that node’s predecessors. If any of the predecessors is an obstacle, we discard it. If any of them have been flagged as visited, we discard it. Otherwise we flag them as visited and en-queue them to the end of the node queue. Repeat the process. After the loop ends we should have a visitation array that contains all the nodes that are connected to the goal node. This module is illustrated by the flowchart given in Figures 4.1 and 4.2.

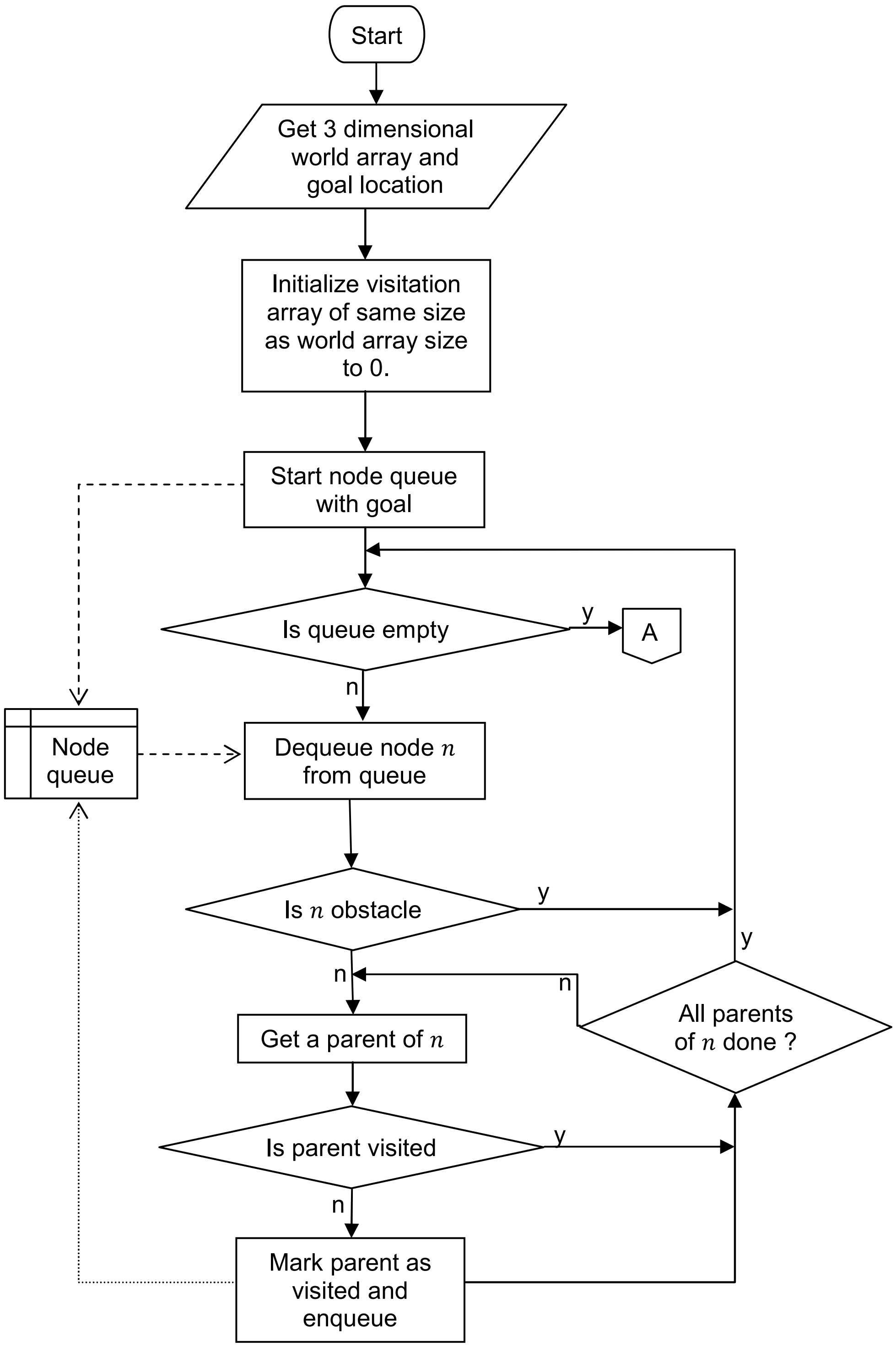

Start
Get 3 dimensional world array and goal location
Initialize visitation array of same size as world array size to 0.
Start node queue with goal
Is queue empty
y
A
n
Node queue
Dequeue node n from queue
Is n obstacle
y
y
n
n
All parents of n done ?
Get a parent of n
Is parent visited
y
n
Mark parent as visited and enqueue

Figure 4.1 Flowchart for Backward Chaining to find Start State, Part 1

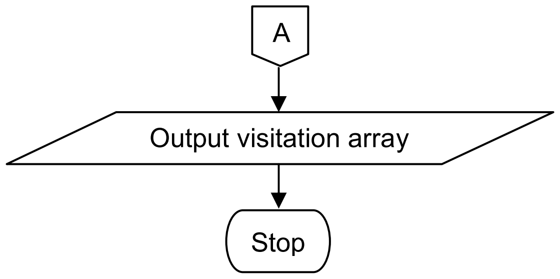


Figure 4.2 Flowchart for Backward Chaining to find Start State, Part 2

4.4 Random World Expansion

In this section we shall see how the random world generated in Section 4.2 is expanded into a three dimensional array, with the third dimension representing time for use by the module described in Section 4.3. The module also needs to fix a starting location for the world. To generate the third dimension values, the module creates a three dimensional array of size $r \times c \times (l+1)$ initialized to -1 where r and c are the dimensions of the world and l is the look-ahead. For every obstacle the module determines the location where it will be at that particular time step and set the potential value to 1. All other locations are marked as -1 and are yet to be processed. A randomly selected location which is not an obstacle at time l is taken as a goal and its potential value is set to 0. The next step is to find a start state from which this goal is reachable. The module to do this is described in Section 4.3. This module returns a set of states that can be reached from the goal states at the various time steps. One of these states in the initial time step is chosen as the start state. The module then returns the three dimensional array and the start state generated. This module is illustrated in the flowchart given in Figures 4.3 and 4.4.

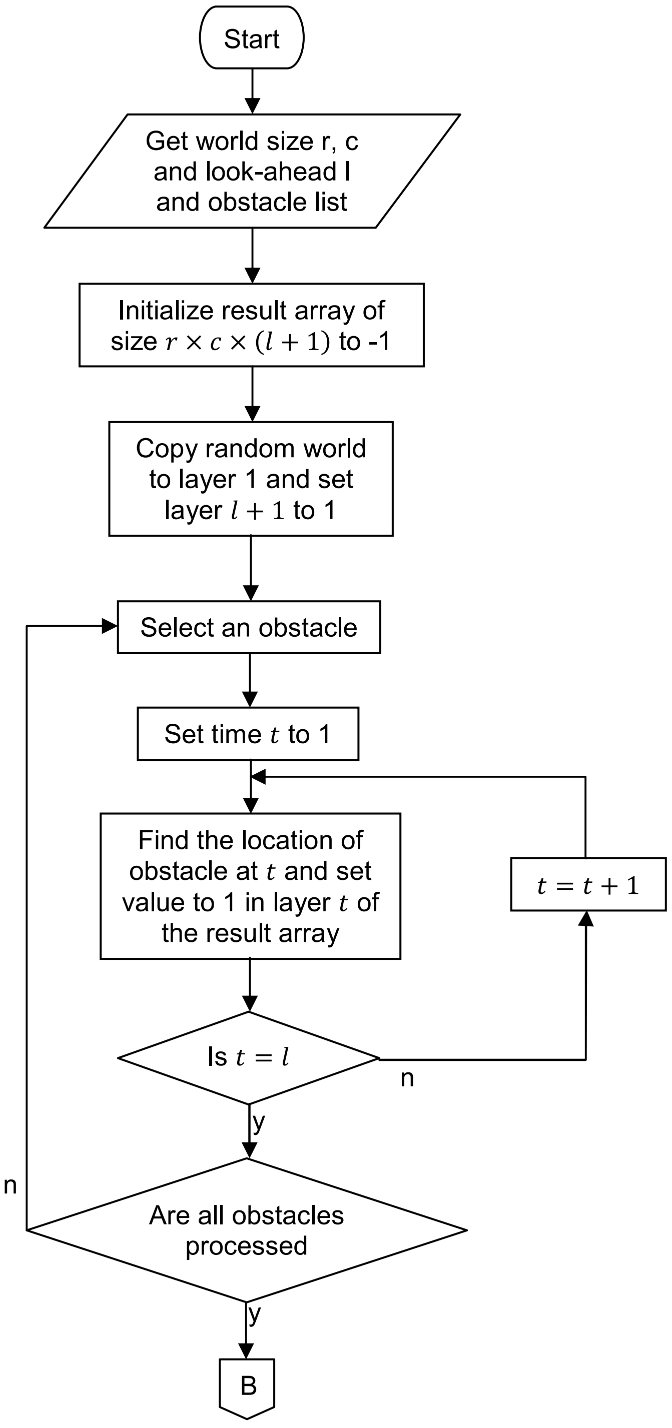

Start
Get world size r, c and look-ahead l and obstacle list
Initialize result array of size $r \times c \times (l+1)$ to -1
Copy random world to layer 1 and set layer $l+1$ to 1
Select an obstacle
Set time t to 1
Find the location of obstacle at t and set value to 1 in layer t of the result array
$t = t + 1$
Is $t = l$
n
y
n
Are all obstacles processed
y
B

Figure 4.3 Flowchart for Random World Expansion, Part 1

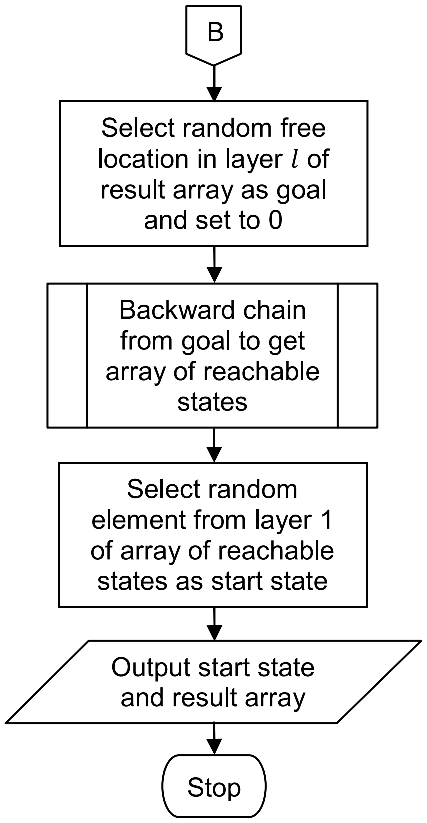

B
Select random free location in layer l of result array as goal and set to 0
Backward chain from goal to get array of reachable states
Select random element from layer 1 of array of reachable states as start state
Output start state and result array
Stop

Figure 4.4 Flowchart for Random World Expansion, Part 2

4.5 Potential Value Calculation

This module implements the calculation of the potential value for the locations in the world as described by Equation 3.1. It takes as input the expanded world array produced by the modules described by Sections 4.4 or 4.7. From the time step at the look-ahead value, the algorithm works its way back in time, processing each location in the world. If the current location has a potential value of -1 i.e. it has been marked as unprocessed then it takes the average of the potential values of the surrounding locations and the present location all at the next time step. If any of these locations does not exist i.e. the current location is on the edge or corner of the world, then the average is computed only for the existing successor locations. It then returns this updated world array, where the value at co-ordinate (x, y, t)

represents the likelihood that a robot will collide with an obstacle provided it takes a random walk from location (x, y) at time step t. This is illustrated by the flowchart provided by Figure 4.5

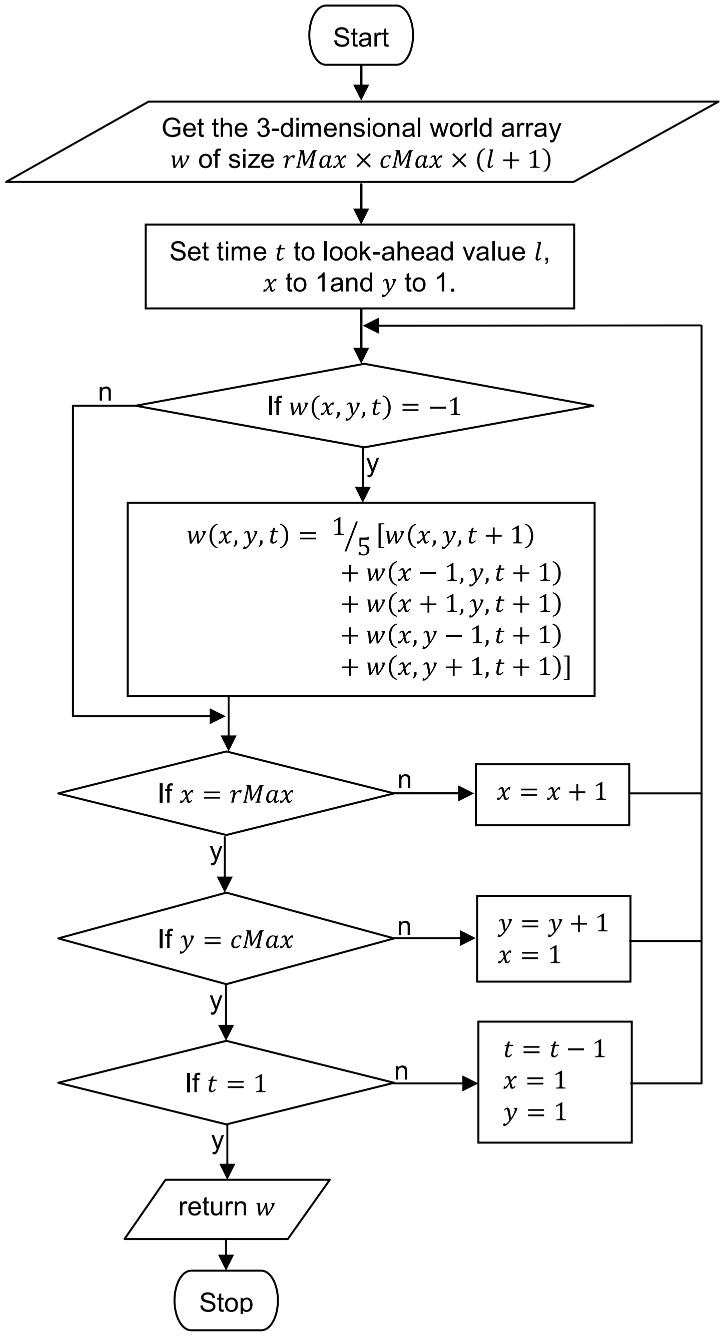

Start
Get the 3-dimensional world array w of size $rMax \times cMax \times (l+1)$
Set time t to look-ahead value l, x to 1and y to 1.
If $w(x,y,t) = -1$
n
y
$w(x,y,t) = {}^1/_5\,[w(x,y,t+1) + w(x-1,y,t+1) + w(x+1,y,t+1) + w(x,y-1,t+1) + w(x,y+1,t+1)]$
If $x = rMax$
n
$x = x+1$
y
If $y = cMax$
n
$y = y+1$ $x = 1$
y
If $t = 1$
n
$t = t-1$ $x = 1$ $y = 1$
y
return w
Stop

Figure 4.5 Potential Value Calculation

4.6 Generation of Potential Value Curves

This module is used to generate the potential value curves needed to find a good look-ahead value to use during the planning stage. The process described below is a very time consuming one and so is done only once. The initial step is to get the size of the world and the obstacle density for which we need to generate the potential value curve. The next step is to generate a set of random worlds of that size and that obstacle density. This is done by the module described in Section 4.2. The initial look-ahead is set to a hundred time steps. The next step is to extend the world in the third dimension (time) and find the obstacle positions up to the look-ahead time step. This is done by the module described in Section 4.4. The module also returns a start state that is reachable from a randomly selected goal state in the time step at the look-ahead limit. Then the potential values for the locations in the world are calculated by the module described in Section 4.5. The potential value of the start state is then stored in an internal array. This process is repeated for decreasing values of the look-ahead, from 100 all the way to 2. Initially the decrements are by 10 until the look-ahead reaches 50 after which the decrements are by 5. This is because there are greater changes in the look-ahead value when the look-ahead value is in the lower ranges than when it is in the higher ranges. This is repeated for all the random worlds generated. The potential values of the starting states of the various worlds are grouped together by the look-ahead times and the mean values of the group are calculated. To better illustrate the behavior of the potential values and to observe the effect of increasing the look-ahead value, the potential values and mean values are plotted on a graph, along with the curve joining the mean values. The potential values and their mean are output for future use. The main planner uses this process to generate the potential curves once for various density values and then uses the values for subsequent runs. The only time this module has to be called again is if the size of the world changes. The module described above is illustrated by the Flowchart given in Figure 4.6

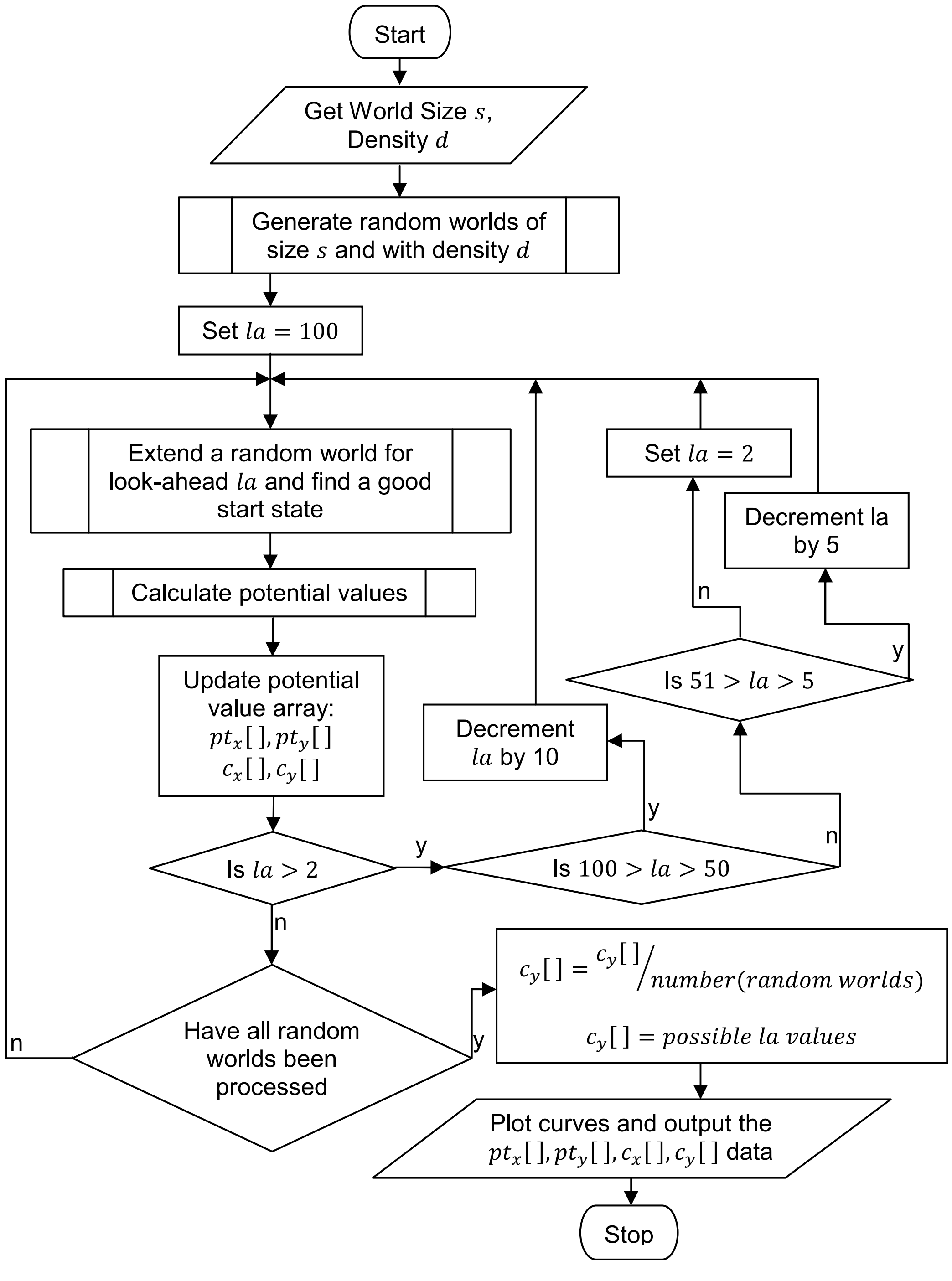

Start
Get World Size s, Density d
Generate random worlds of size s and with density d
Set $la = 100$
Extend a random world for look-ahead la and find a good start state
Calculate potential values
Update potential value array: $pt_x[\,], pt_y[\,]$ $c_x[\,], c_y[\,]$
Is $la > 2$
y
n
Is $100 > la > 50$
y
n
Decrement la by 10
Is $51 > la > 5$
y
n
Decrement la by 5
Set $la = 2$
Have all random worlds been processed
n
y
$c_y[\,] = c_y[\,] / number(random\ worlds)$
$c_y[\,] = possible\ la\ values$
Plot curves and output the $pt_x[\,], pt_y[\,], c_x[\,], c_y[\,]$ data
Stop

Figure 4.6 Flowchart for generating potential value curves

4.7 Extending the world array in time

This module extends the given two dimensional world array into a three dimensional array, with the third dimension representing time. This is similar to the procedure used in Section 4.4. The difference is that instead of randomly selecting a goal state, the goal location at the initial time step is provided. The goal's position just has to be extrapolated across all the time steps. To generate the third dimension values, the module creates a three dimensional array of size $r \times c \times (l + 1)$ initialized to -1 where r and c are the dimensions of the world and l is the look-ahead. For every obstacle find the location where it will be at that particular time step and mark with potential value 1. For the goal find the location where it will be at that particular time step and mark with potential value 0. All other locations still marked as -1 are yet to be processed. The module then returns the three-dimensional world array.

4.8 Forward chaining to find time to reach first reachable goal location

This module is used by the main planner to find the number of time steps needed to reach the first reachable goal location. The method is similar to the backward chaining done in Section 4.3. An empty three dimensional visitation array is created. A node queue is created with the start state in it. Then the following steps are repeated in a loop. De-queue the first location from the queue. If it is a goal state then return the time co-ordinate. If the time co-ordinate limit is more than the forward chaining limit then return a failure message (represented by a value of -1). If it is neither of the two then add the successors of the current node to the end of the queue. Repeat the loop till the queue is empty or a goal state is reached. The module described in Section 4.7 is used to extend the world in three dimensions till the forward chaining limit to make checking for goal and obstacle states easier. The forward chaining limit is used because the dynamics of the worlds may have changed if we have been forward chaining for too long.

4.9 Determining Look-ahead value to use

This module uses the value obtained from the previous module and the set of potential value curves generated in Section 4.6 to determine the amount of time the planner has to look-ahead by. The module gets as an input the current world and the time taken to reach the first goal. The modules described in Section 4.7 and Section 4.5 are used to find the potential value of the start state. This value is compared to the potential value curve generated for the set random worlds of the closest density as the given world. While deciding the closest density it is better to use a conservative estimate i.e. the closest lower value. Assuming that for each look-ahead the probabilities are distributed normally, we try to find the highest look-ahead for which the z-table value falls just below a set threshold, where the threshold is set to 0.05. This value is called la_2. The look-ahead value where the current start potential would lie on the curve is la_1 and the current look-ahead is la_0. The values of α and β are obtained as given by Equation 3.2. The greater of these is added to la_0 to get the new look-ahead.

4.10 Traversing World Array to find the Goal

This module uses the three dimensional array filled with potential values generated by the module described in Section 4.5 to generate a representation of the path that a robot that is using the proposed path planner will navigate. It creates an empty three dimensional array to hold the set of nodes visited during the traversal. Starting from the initial state, the following steps are repeated. The current location at time t is marked as visited. If the current location is a goal location then the visitation array is returned. The potential values of the current location's successors at time $t+1$ are found. The successor with the minimum potential value is found. Current location is changed to that location. The process is repeated until the current location reaches a goal or an obstacle or until the number of time steps taken becomes equal to the look-ahead limit. If the current location is an obstacle then return a failure message. If the current location is at the final time step i.e. at the look-ahead limit, return a failure message. Otherwise return the populated visitation array.

4.11 Display path

The purpose of this module is to display the representation of the path traversed by the previous module in a fashion that can be easily understood by anyone. It just uses the node visitation information given by the module described in Section 4.10 and the three dimensional world array obtained from the module described in Section 4.7. It outputs a 3 dimensional plot where the obstacles are represented by points and the goals motion and the robots path are represented by lines.

4.12 Temporal Potential Path Planning Simulator

This is the main part of the path planner simulator. It consists of two parts, the potential value curve generator and the actual path planner. The first part has to be run only once. It calculates the potential value curves for the given world size and the various densities. The algorithm used to do this is discussed in Section 4.6. The second part is the path planner that is executed for the given sample world. We use the forward chaining discussed in Section 4.8 to find the time taken to reach the first solution. If the forward chaining takes too long to find the solution, we reload the given world and repeat the process. This is because the dynamics of the world may have changed. After obtaining the time required to reach the shallowest goal, we use the potential value curves previously generated to find the best look-ahead value. The module used to do this is discussed in Section 4.9. We then extend the world in the third dimension (time) until the look-ahead value calculated. The module to do this is discussed in Section 4.7. We then calculate the potential values for the world from the look-ahead time limit till the initial state moving backwards, this module is discussed in Section 4.5. Finally the path traversed by a robot using this planner is generated by travelling from the initial state to the neighbor with the least potential value at each time step. The module for this is discussed in Section 4.10. If the traversal does not reach a goal before the look-ahead time limit then an error message is returned. This step is present purely to capture any anomalous error and will never be executed. The path traversed by the robot is displayed by the module described in Section 4.11. The flow chart given in Figure 4.7 best illustrates this method.

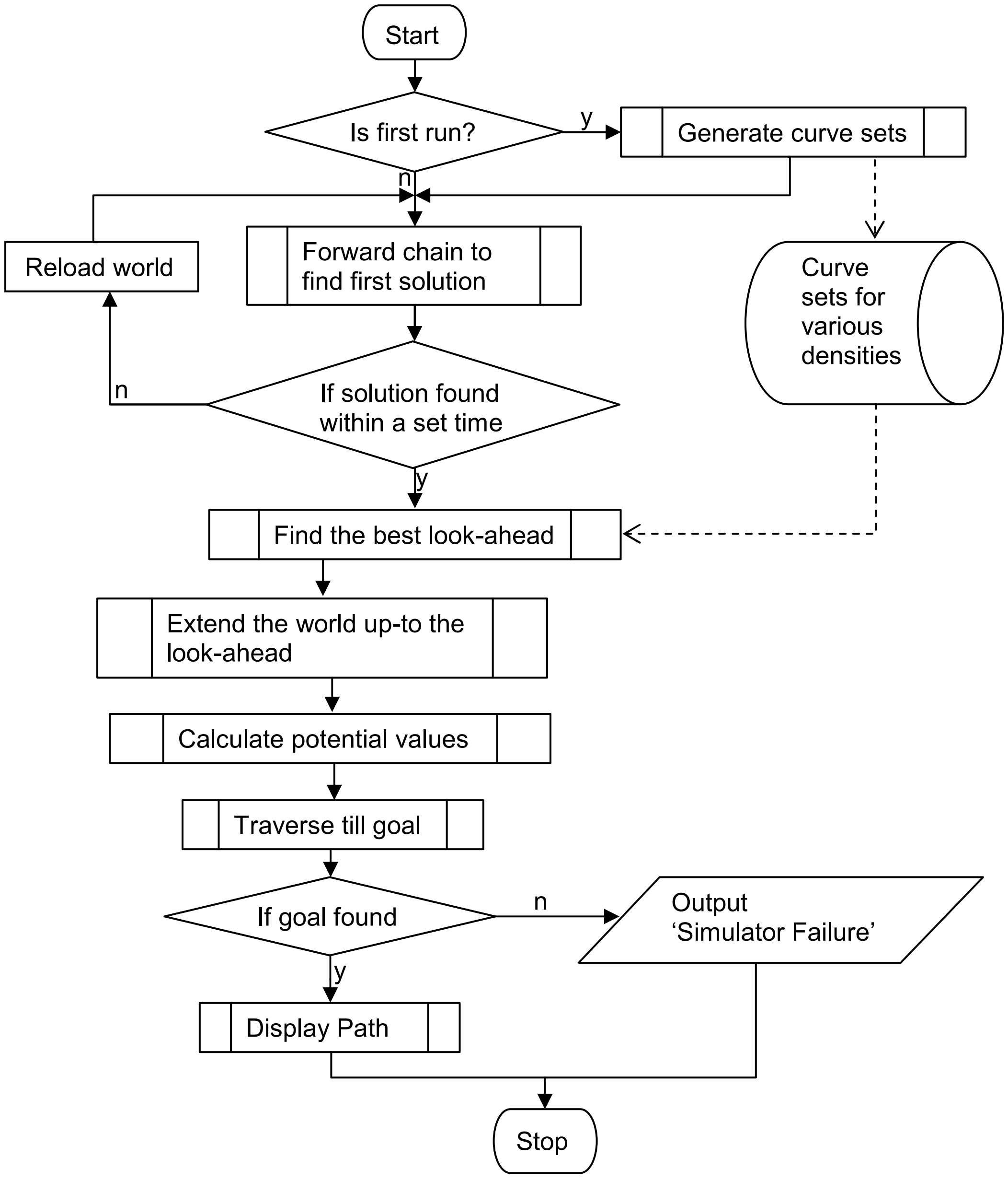

Start
Is first run?
y
Generate curve sets
n
Reload world
Forward chain to find first solution
Curve sets for various densities
n
If solution found within a set time
y
Find the best look-ahead
Extend the world up-to the look-ahead
Calculate potential values
Traverse till goal
If goal found
n
Output 'Simulator Failure'
y
Display Path
Stop

Figure 4.7 Flowchart of the Temporal Potential Function approach

4.13 Additional Notes on Implementation

The modules described in Sections 4.3 and 4.8 use queues to store the list of nodes they can possibly visit. Since the queue data structure is not natively implemented in MATLAB, the Data Structures & Algorithms Toolbox [32] was used along with the updated pointer library for MATLAB 7.0+ [33]. The actual code that was written for this implementation is provided in Appendix A and an explanation of the various symbols used in the flowcharts used in this chapter is given in Appendix B.

CHAPTER 5

EXPERIMENTAL RESULTS AND OBSERVATIONS

5.1 Description of Experiments conducted

In this chapter we shall see some of the experiments conducted with the proposed path planner and the observations made. First a world with known obstacle and goal positions and velocities is taken and the effect of increasing the look-ahead limit is observed. Then a randomly generated world is taken and the potential value when the goal is set to just one step below the look-ahead value is found. (This is the same way the potential value curves are generated). Using this curve the effect the time taken to reach goal has on the potential value can be observed. Next, the potential value curves for various world sizes and obstacle densities are generated and compared. Next, a sample world is taken and the various steps in the path planning process are executed and observed. Finally, observations are made on how the robot picks the best solution within the given look-ahead limit.

5.2 Effect of Look-Ahead on Potential Value

Consider the potential value of a start state in a given world. The world used here is a known world where the solution occurs at time step 12. As can be seen from Figure 5.1, the potential value is 1 until the look-ahead time the planner would have reached the goal. If we had proceeded with a smaller look-ahead, the planner would have ended in a failure. As the look ahead increases, the potential value of the start state decreases until it reaches a minimum and it does not reduce any further after that value. This emphasizes the importance of selecting the correct look-ahead value since if too high a look ahead is selected, the algorithm may perform a lot of unnecessary calculations.

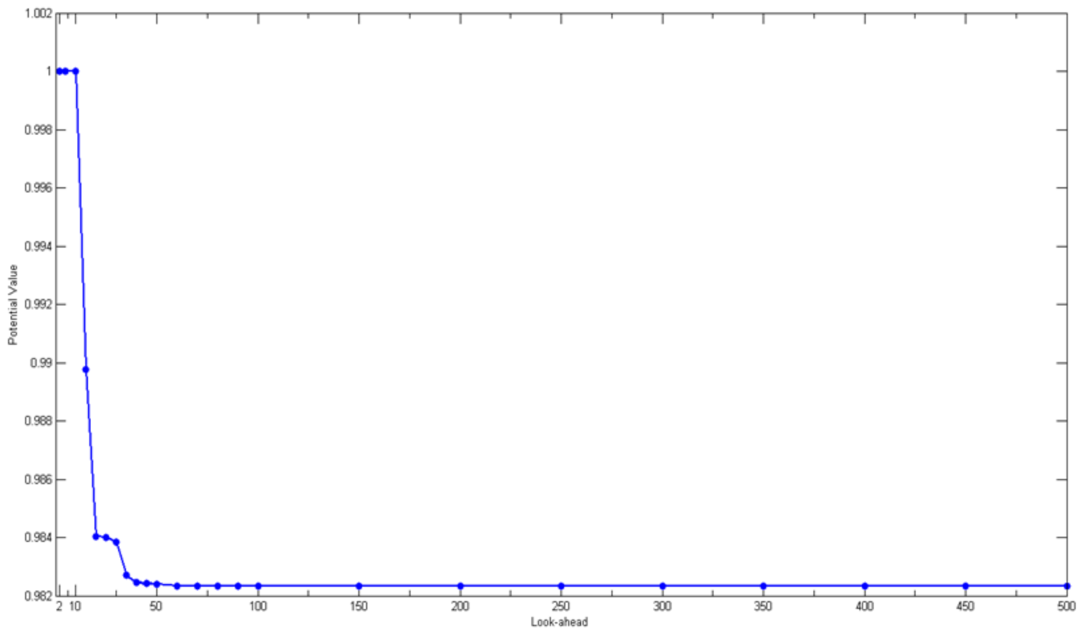


Figure 5.1 Potential values of start state in given world

5.3 Effect of Time to reach goal on Potential Value

The potential value of the start state of a randomly generated world is shown for values of various look-ahead limits in Figure 5.2. For each of these worlds the goal state is set just at the look-ahead value. As can be seen, the potential of the start state increases as the distance to reach the first goal increases. The kinks in the curve are due to small fluctuations in the data value due to the world being generated randomly. So to get a more accurate picture the values for a set of randomly generated worlds is calculated as shown in Figure 5.3. The blue dots in the graph represent the potential values for randomly generated worlds. The green line represents the curve joining the mean of these values. As can be seen, the curve is smoother here. Also, the effect of the goal distance on the potential values can be clearly seen. The longer the robot navigates through the world without reaching a goal, the more likely that it would hit an obstacle.

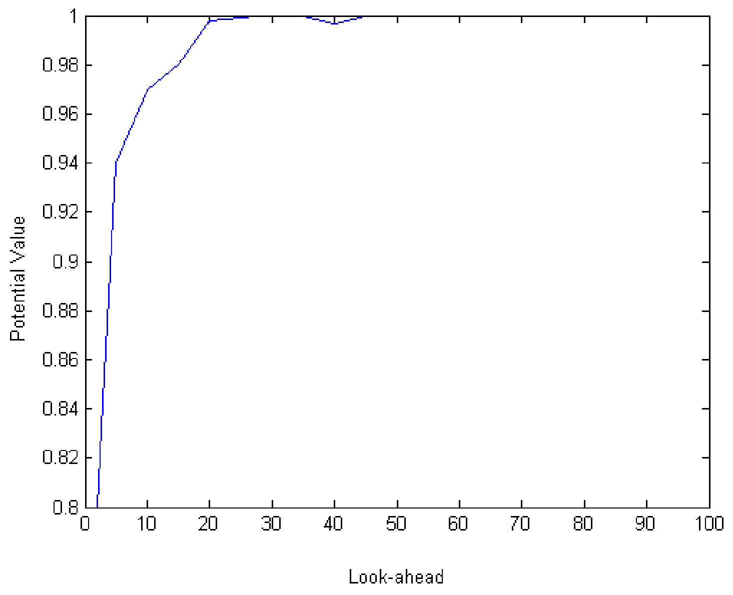


Figure 5.2 Potential value curve of a random 10×10 world with obstacle density 9%

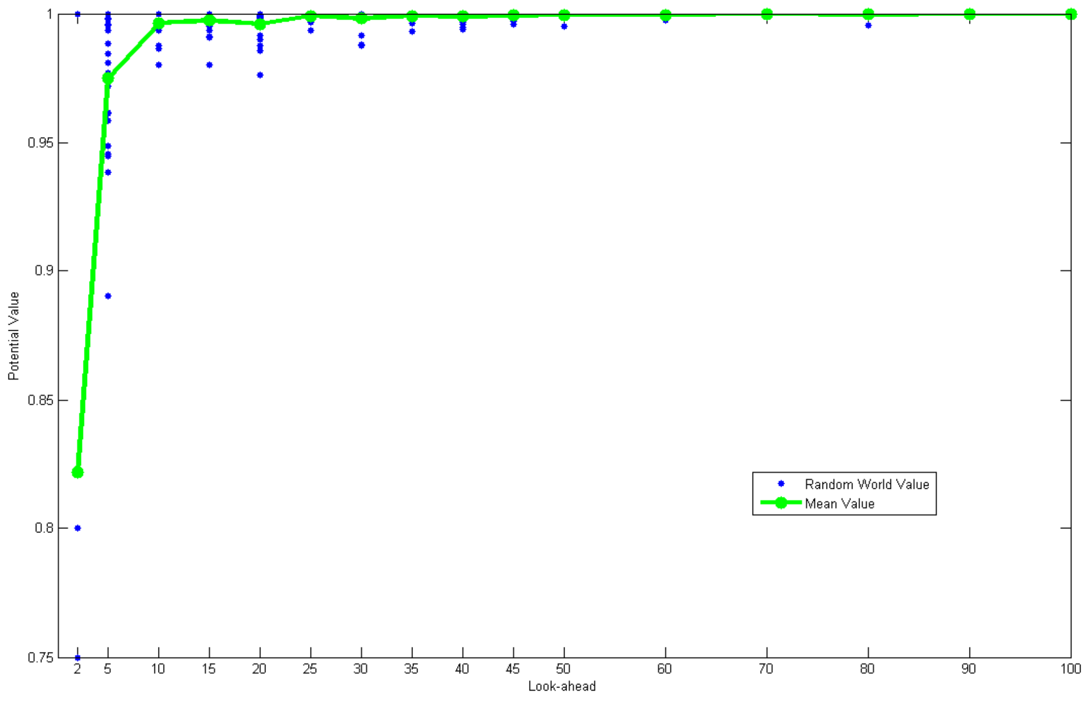


Figure 5.3 Potential value curve of set of random 10×10 worlds with obstacle density 9%

5.4 Effects of World Size on Potential Value

The potential value curves generated by worlds of different sizes and a fixed obstacle density are shown in Figure 5.5. Since the change is greatest in the lower ranges of the look-ahead value, the x-axis (look-ahead value) is taken in logarithmic scale to improve clarity.

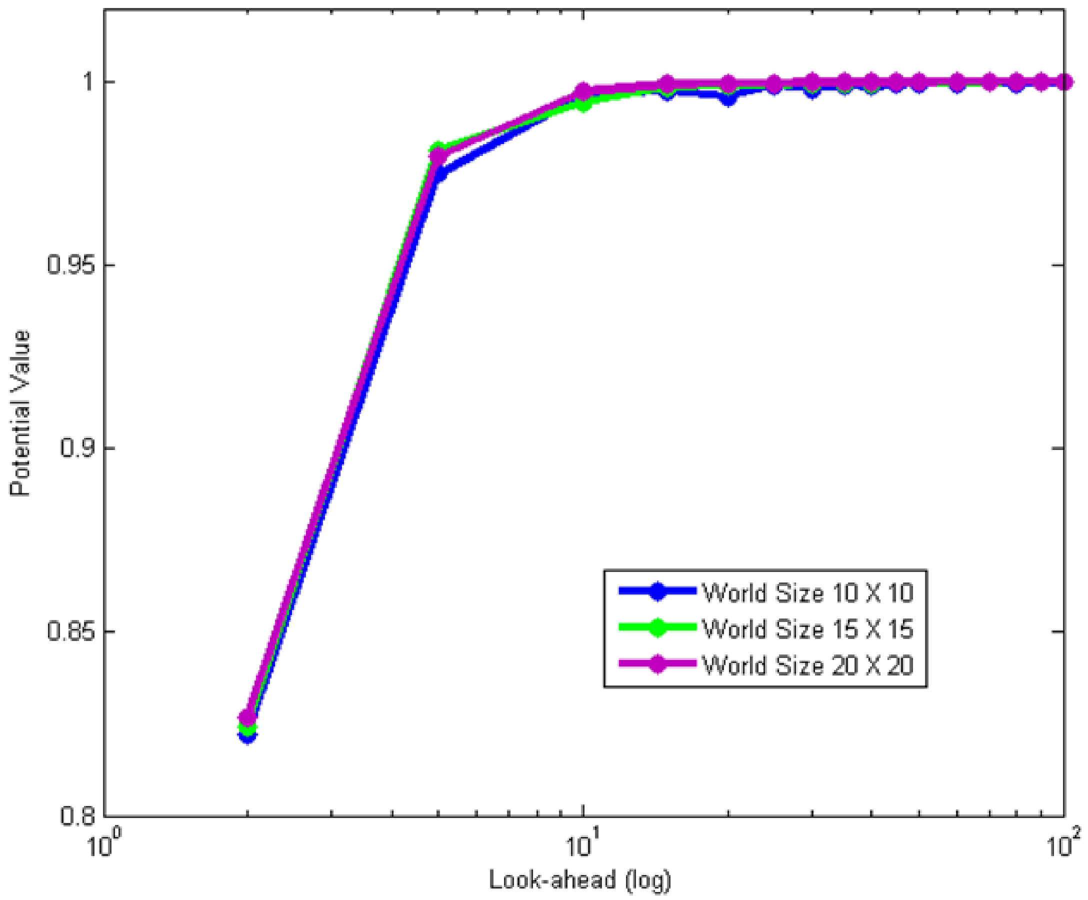


Figure 5.4 Potential curves of set of random worlds of varying sizes with obstacle density 9%

Although the curves are generated separately for each different world size, the curves are remarkably similar. However we only use the curves that have been generated for a particular world size when the planner is planning for that world size.

5.5 Effects of World Density on Potential Value

The potential value curves for random worlds of a fixed size and different densities given in Figure 5.6. The potential values tend to get closer to 1 quicker i.e. at lower look-ahead value as the obstacle density increases. As the number of obstacles increases so too does the probability of a robot colliding with it during a random walk.

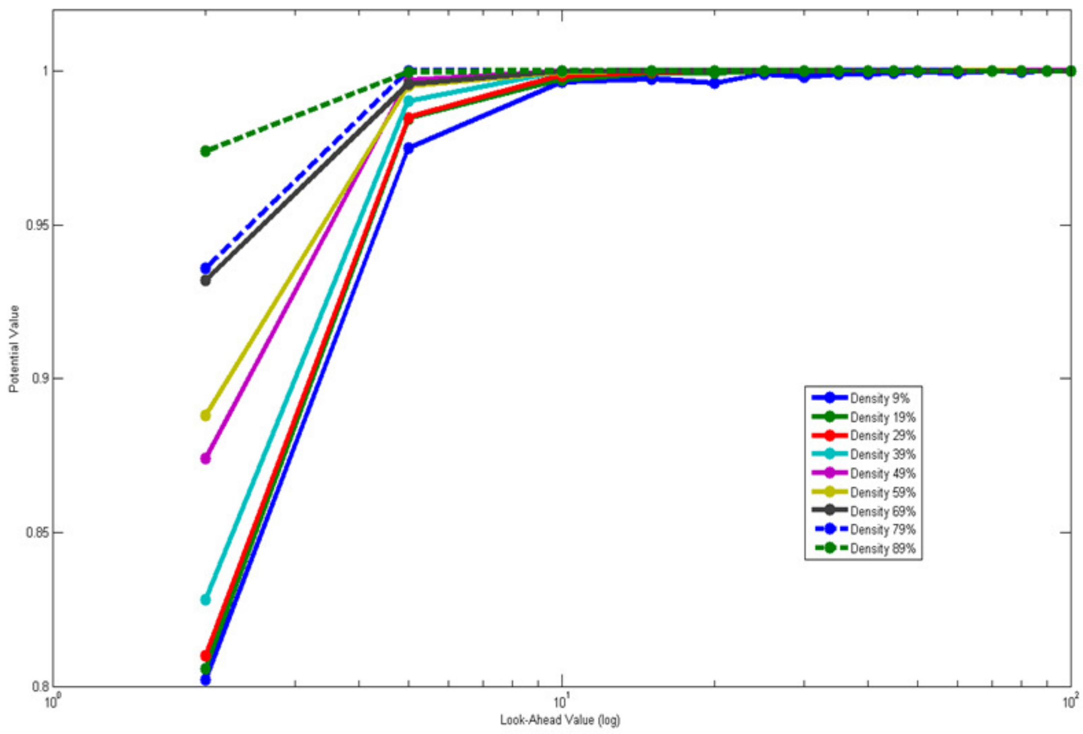


Figure 5.6 Potential curves of sets of random 10×10 worlds of varying obstacle density

5.6 Path planner execution

We shall now look at sample executions of the path planner. The worlds we input for the path planner is similar to the random worlds we generate for the potential value curves. They contain a set of point obstacle each with their own velocities and a goal location with its own velocity.

A two dimensional grid map showing positions of the goal and obstacles in the first world is given in Figure 5.7. The first step is to check if the set of potential value curves exists for the given world's size and density. If they do not exist, then the curves are generated for the world's parameters. The next time a similar world is encountered we can use the previously generated curve. A curve generated in this fashion is shown in Figure 5.3.

The next step is to use forward chaining to find the time required to reach the first goal. After obtaining the number of steps required to reach the first solution, the value is compared to the potential value curve of the closest density lower than the world's density and the correct look-ahead is determined.

After obtaining the look-ahead value the planner builds a three dimensional grid map of the world. The goal and obstacle locations are extrapolated up to the look-ahead value previously obtained. The world thus obtained is shown in Figure 5.8. The red dots represent obstacle positions and the green dots represent the goal positions. The green line shows the path the goal takes. Any position where a goal and obstacle may intersect is also considered as an obstacle.

The next step is to calculate the potential values for each of the cells. This is calculated by the method described in Section 4.5. There is no need to repeat the iteration as the values converge on the first run. This is because changing the value of a location does not change the value of its successor locations. After obtaining the potential value the planner follows the negative gradient in increasing time, to find the path. The path obtained for the example world is given in Figure 5.9. The blue line shows the path taken and the blue dots show the robot position at each time step.

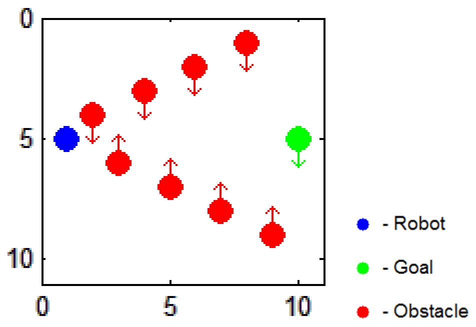


Figure 5.7 Two dimensional representation of world 1.

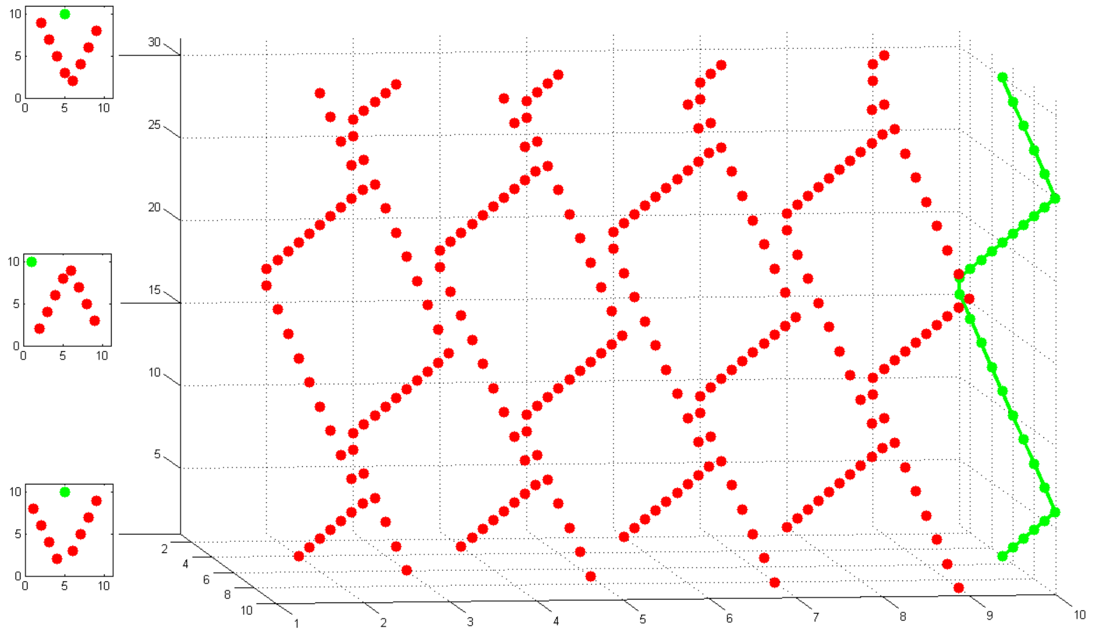


Figure 5.8 Three dimensional representation of world 1.

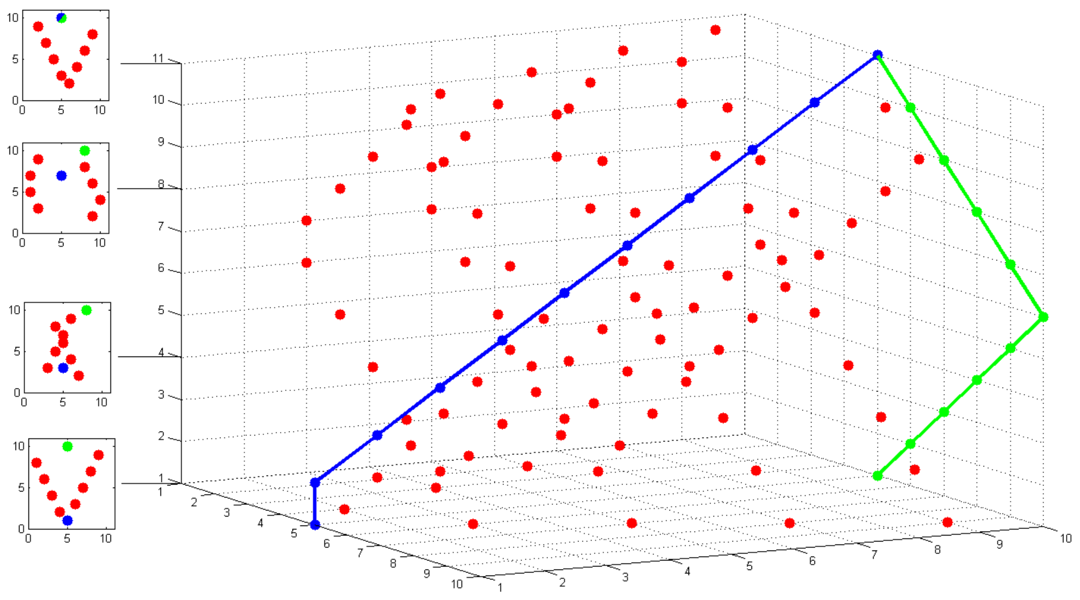


Figure 5.9 Path Traversed by Robot in world 1.

Another sample execution was done for a different world. The two and three dimensional representations of this world are shown in Figure 5.10 and Figure 5.11. The path travelled by the robot is given in Figure 5.12. The blue dots represent the robot and the blue line represents the path it takes. The red dots represent the obstacles. The green dots represent the goal and the green line represents the path it takes.

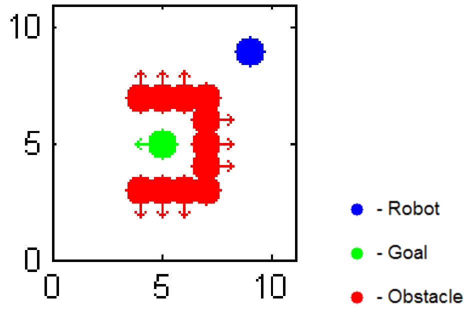


Figure 5.10 Two dimensional representation of world 2.

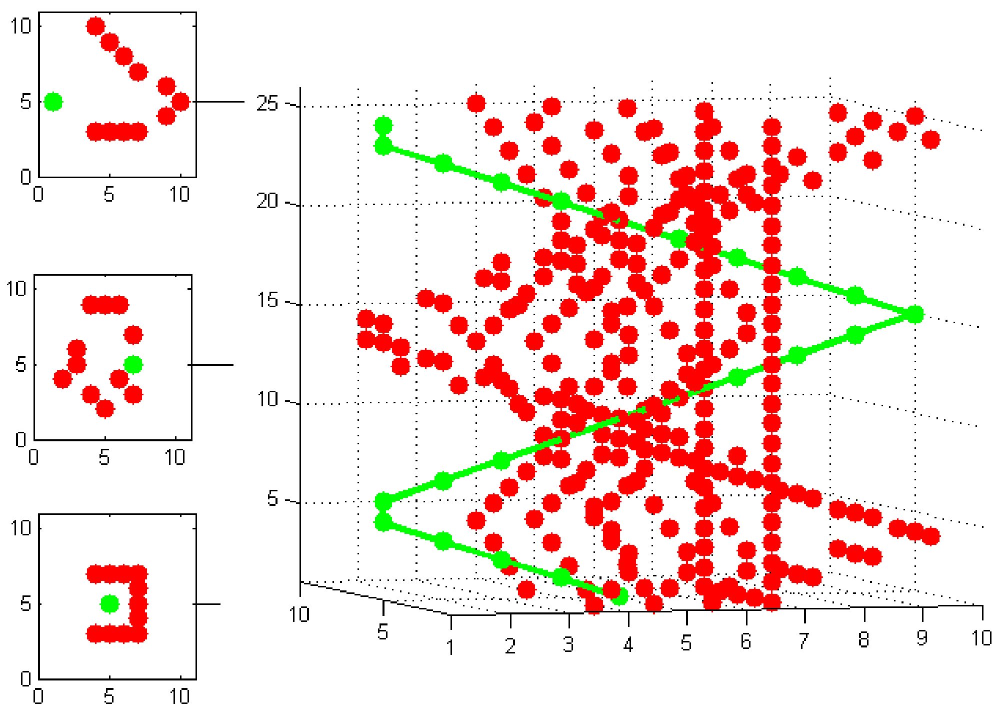


Figure 5.11 Three dimensional representation of world 2.

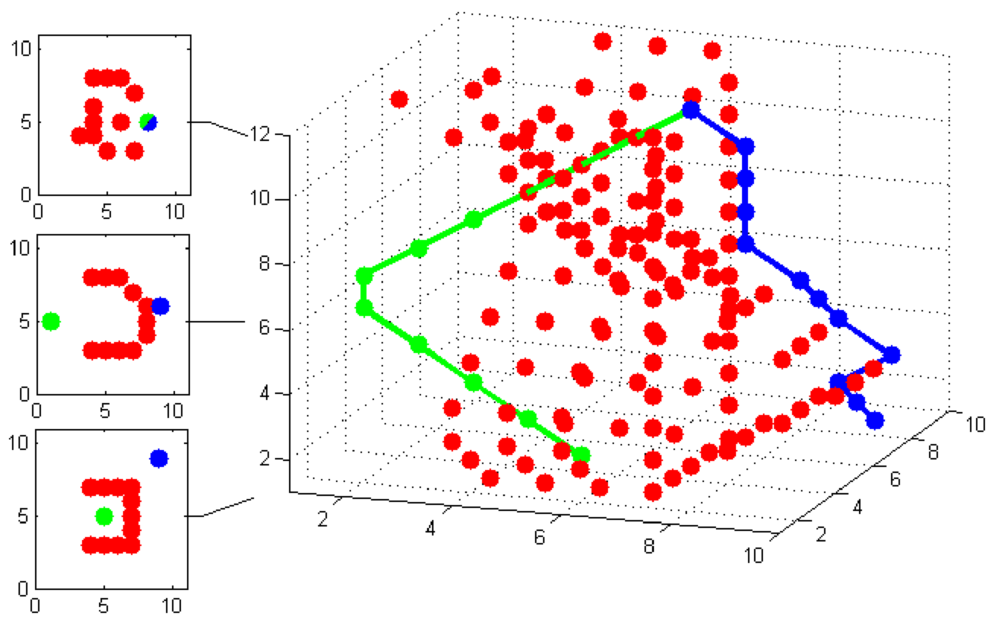


Figure 5.12 Path Traversed by Robot in world 2.

5.6 Avoidance of Suboptimal Goals

The traversal algorithm discussed in Section 3.10 should naturally avoid suboptimal solutions and pick the most optimal solution within the given look-ahead limit. Here optimality is determined as the minimum collision probability in the case of a Random Walk. Consider the following path generated in Figure 5.13. The goal has been deliberately placed in a risky position for the purpose of this experiment. Since this is the only goal location available in the world, the planner has no option but to plan a path to that location.

However consider Figure 5.14. Here another goal has been placed in a much safer position. The path planner now plans a path to the safer goal location, even though it will take an extra 25 time steps more to reach the goal. This is because the free space near the first goal has a high collision probability in case of a random walk because of its proximity to mobile obstacles but the free space near the second goal will have a lower collision probability. The potential values are always calculated such that the robot ends up going to the safest goal within the given look-ahead time

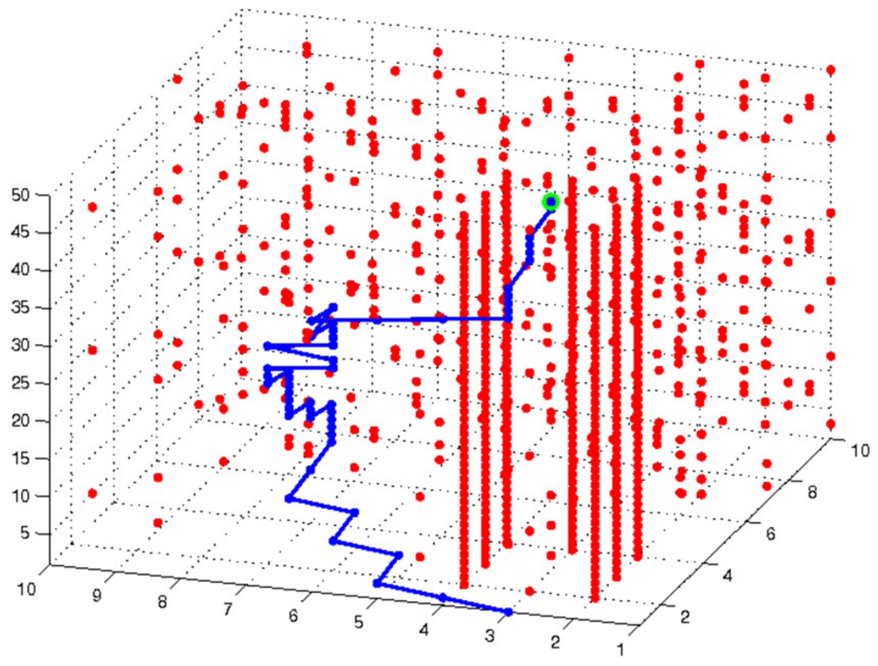


Figure 5.13 Planner forced to pick risky path.

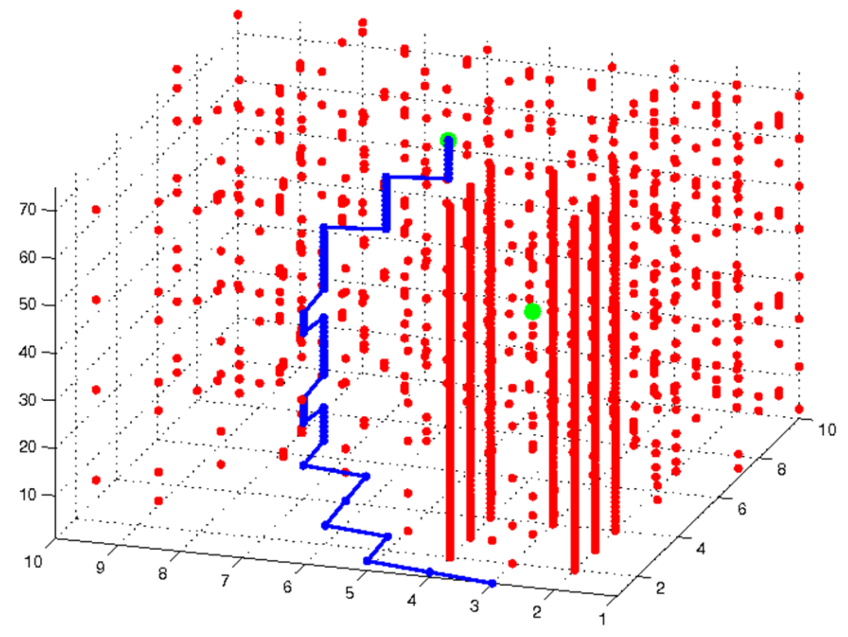


Figure 5.14 Avoiding Sub-Optimal Goal

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis developed and implemented a path planner for use in dynamic environments that uses a harmonic function extended in the time dimension. The following chapter provides conclusion for this work, along with future directions for research.

6.1 Conclusion

Consider for example someone attempting to catch a falling ball, the person automatically calculates where the ball will be when he can catch it and moves his hands to that position instead of trying to move his hands towards the ball. This is in essence the principle behind the proposed approach. The path planner looks ahead by a predetermined amount and calculates the potential values of the world locations from the initial time up to the look-ahead time. The robot then follows the negative potential gradient with respect to time to reach the goal. Since we use forward chaining to find the depth to the first goal we are guaranteed a solution. Also, the usage of potential value curves gives us a good chance of finding a good solution. Since the potential values represent the probability that a robot executing a random walk from that position at that time will collide with an obstacle, following the path of negative potential gradient will mean that the robot will follow the safest path to the goal.

Harmonic Functions were used as the basis for the path planning algorithm because they generate smooth, complete and correct paths. Since the world has been stretched in the time dimension, the multiple iterations of the numerical evaluation of a harmonic function are executed in one pass over the three-dimensional world array.

MATLAB was used to simulate the path planner due to the ease with which large calculations can be done in them. Some external toolboxes were needed for extra functionality like queues.

6.2 Future Work

In the proposed method, if the dynamics of the environment change during runtime, there is no option other than to re-plan from that point on. It remains to be seen if the potential values can be updated on the fly. This could greatly reduce the re-planning time.

APPENDIX A

EXPERIMENTAL IMPLEMENTATION – MATLAB CODE

Random World Generation

gen_random_obs.m

```
function rworlds = gen_random_obs(rows,cols,density,nworlds)
%
%Generated set of random obstacles to use to create random worlds
%
nobs = round((density/100)*rows*cols); %find number of obstacles
for i = 1:nworlds
   w(1:rows,1:cols) = -1;
   for j = 1:nobs
      tf = 1;
      while tf==1 %select empty location
         tr = round(1 + ( rows - 1) .* rand);
         tc = round(1 + ( cols - 1) .* rand);
         tf = 0;
         if w(tr,tc) == 1
            tf = 1;
         end
      end
      w(tr,tc) = 1;
      tdir = round(1 + ( 4 - 1) .* rand);%set a random direction
      switch tdir
         case 1
            td = 'N';
         case 2
            td = 'E';
         case 3
            td = 'W';
         case 4
            td = 'S';
      end
      tv = round((min(rows,cols)-1) .* rand);%set a random velocity
      rw(i).obs(j).r = tr;
      rw(i).obs(j).c = tc;
      rw(i).obs(j).vel = tv;
      rw(i).obs(j).dir = td;
   end
end
rworlds = rw;
end
```

Backward Chaining to find start state for random goal

backward_chain.m

```
function vn = backward_chain(w,tr,tc,la)
%
%Backward chain from goal of given world to find the initial states %that
%are reachable
%
t.r = tr;
t.c = tc;
t.l = la;
q = qu_new;% start node queue
q = qu_enqu(q,t);% enqueue goal
v(1:size(w,1),1:size(w,2),1:la) = 0;
while qu_empty(q) ~= 1
   cu = qu_front(q);
   q = qu_dequ(q);% dequeue goal
   if (w(cu.r,cu.c,cu.l) ~= 1)&&(cu.l~=1)% check if obstacle or initial state
      if v(cu.r,cu.c,cu.l-1) ~= 1
         t.r = cu.r;
         t.c = cu.c;
         t.l = cu.l - 1;
         q = qu_enqu(q,t); % enqueue descendant
         v(cu.r,cu.c,cu.l-1) = 1;% mark descendant as visited
      end
      if (cu.r > 1)&&(v(cu.r-1,cu.c,cu.l-1) ~= 1)
         t.r = cu.r - 1;
         t.c = cu.c;
         t.l = cu.l - 1;
         q = qu_enqu(q,t);
         v(cu.r-1,cu.c,cu.l-1) = 1;
      end
      if (cu.c > 1)&&(v(cu.r,cu.c-1,cu.l-1) ~= 1)
         t.r = cu.r;
         t.c = cu.c - 1;
         t.l = cu.l - 1;
         q = qu_enqu(q,t);
         v(cu.r,cu.c-1,cu.l-1) = 1;
      end
      if (cu.r < size(w,1))&&(v(cu.r+1,cu.c,cu.l-1) ~= 1)
         t.r = cu.r + 1;
         t.c = cu.c;
         t.l = cu.l - 1;
         q = qu_enqu(q,t);
         v(cu.r+1,cu.c,cu.l-1) = 1;
      end
      if (cu.c < size(w,2))&&(v(cu.r,cu.c+1,cu.l-1) ~= 1)
         t.r = cu.r;
         t.c = cu.c + 1;
         t.l = cu.l - 1;
         q = qu_enqu(q,t);
```

```
            v(cu.r,cu.c+1,cu.l-1) = 1;
        end
    end
end
q = qu_free(q);% free queue space
vn = v; % return values
end
```

Random World Expansion

gen_random_world.m

```
function [world st_r st_c] = gen_random_world(rows,cols,obs,lookahead)
%
%generate random world in three dimensions
%
w (1:rows,1:cols,1:lookahead)=-1; % create empty world
w (:,:,lookahead+1) = 1; % set lookahead boundry to 1
for i = 1:length(obs) % extrapolate obatacle locations
    tr = obs(i).r;
    tc = obs(i).c;
    tv = obs(i).vel;
    w(tr,tc,1) = 1;
    for j = 1:lookahead-1
        switch obs(i).dir
            case 'N'
                tr = round(tr - tv);
                if tr < 1
                    tr = 1 + (tr * -1);
                    obs(i).dir = 'S';
                end
            case 'E'
                tc = round(tc + tv);
                if tc > cols
                    tc = cols - (tc - cols);
                    obs(i).dir = 'W';
                end
            case 'S'
                tr = round(tr + tv);
                if tr > rows
                    tr = rows - (tr - rows);
                    obs(i).dir = 'N';
                end
            case 'W'
                tc = round(tc - tv);
                if tc < 1
                    tc = 1 + (tc * -1);
                    obs(i).dir = 'E';
                end
            otherwise
                disp('Not Supported')
        end
        if (w(tr,tc,j+1) == 1)
            switch obs(i).dir
                case 'N'
                    tr = tr + 1;
                    if tr > rows
                        tr = tr - 1;
                    end
                case 'E'
                    tc = tc - 1;
                    if tc < 1
                        tc = tc + 1;
                    end
                case 'S'
                    tr = tr - 1;
                    if tr < 1
                        tr = tr + 1;
                    end
                case 'W'
                    tc = tc + 1;
                    if tc > cols
                        tc = tc - 1;
                    end
                otherwise
                    disp('Not Supported')
            end
        end
        w(tr,tc,j+1) = 1;
    end
end
tf = 1;
tr = 0;
tc = 0;
while tf == 1% select random goal
    tf = 0;
    tflg = 1;
    while tflg == 1
        tr = round(1 + ( rows - 1) .* rand);
        tc = round(1 + ( cols - 1) .* rand);
        tflg = 0;
        if ( w(tr,tc,lookahead) ~= -1 )
            tflg = 1;
        end
    end
    w(tr,tc,lookahead) = 0;
    v  = backward_chain(w,tr,tc,lookahead);
    if any(v(:,1))== 0
        tf = 1;
        w(tr,tc,lookahead) = -1;
    else
        tflg = 1;
        while tflg == 1
            tflg = 0;
            tr = round(1 + ( rows - 1) .* rand);% find stat state
            tc = round(1 + ( cols - 1) .* rand);
            if v(tr,tc,1) == 0
                tflg = 1;
            end
        end
    end
end
world = w;
st_r = tr;
st_c = tc;
end
```

Potential Value Calculation

calc_pot_values.m

```
function res = calc_pot_values(w)
%
%calculate potential values for non obstacle and foal locations in %given world
%
r = size(w,1);
c = size(w,2);
for t = size(w,3):-1:1
    for x = 1:r
        for y = 1:c
            if w(x,y,t) == -1
                n = w(x,y,t+1);
                if x ~= 1
                    n = n + w(x-1,y,t+1);
                else
                    n = n + 1;% boundary condition
                end
                if x ~= r
                    n = n + w(x+1,y,t+1);
                else
                    n = n + 1;
                end
                if y ~= 1
                    n = n + w(x,y-1,t+1);
                else
                    n = n + 1;
                end
                if y ~= c
                    n = n + w(x,y+1,t+1);
                else
                    n = n + 1;
                end
                w(x,y,t) = n / 5;% take average of neighbors
            end
        end
    end
```

```
end
res = w;
end
```

<u>Generate Potential value curves</u>

<u>gen_curves.m</u>

```
function gen_curves(rows,cols,density)
%
%generate potential value curves
%
rw = gen_random_obs(rows,cols,density,25);
g2x = [2 5:5:45 50:10:100];
g2y(1:16) = 0;
g1x(1:400) = 0;
g1y(1:400) = 0;
for ri = 1:25
   g1i = ri;
   disp(sprintf('Density = %d%% World Number = %d Lookahead = 2',density,ri))
   [w st_r st_c]= gen_random_world(rows,cols,rw(ri).obs,2);
   p = calc_pot_values(w);
   g1x(g1i) = 2;
   g1y(g1i) = p(st_r,st_c,1);
   g1i = g1i + 25;
   g2y(1) = g2y(1) + p(st_r,st_c,1);
   for la = 5:5:45
      disp(sprintf('Density = %d%% World Number = %d Lookahead = %d',density,ri,la))
      [w st_r st_c]= gen_random_world(rows,cols,rw(ri).obs,la);
      p = calc_pot_values(w);
      g1x(g1i) = la;
      g1y(g1i) = p(st_r,st_c,1);
      g1i = g1i + 25;
   end
   for la = 50:10:100
      disp(sprintf('Density = %d%% World Number = %d Lookahead = %d',density,ri,la))
      [w st_r st_c]= gen_random_world(rows,cols,rw(ri).obs,la);
      p = calc_pot_values(w);
      g1x(g1i) = la;
      g1y(g1i) = p(st_r,st_c,1);
      g1i = g1i + 25;
   end
end
for i = 1:16
   g2y(i) = mean(g1y((1:25)+25*(i-1)));
end
figure(density)
plot(g1x,g1y,'b.')
hold on
plot(g2x,g2y,'g*-')
set(gca,'xtick',[2 5:5:45 50:10:100])
fign = sprintf('-f%d',density);
```

```
fn = sprintf('graph_%d_%d_%d',rows,cols,density);
print(fign,'-djpeg',fn)
hold off
tmp.ptx=g1x;
tmp.pty=g1y;
tmp.cx=g2x;
tmp.cy=g2y;
fn = sprintf('curveset_%d_%d_%d',rows,cols,density);
assignin('base',fn,tmp);
evalin('base',sprintf('save curveset_%d_%d_%d.mat %s',rows,cols,density,fn));
end
```

Extending world array in time

extend_world.m

```
function world = extend_world(rows,cols,obs,goal,lookahead)
%
%extends the world to the given look-ahead
%
w (1:rows,1:cols,1:lookahead)=-1; %intialize world
w (:,:,lookahead+1) = 1; %initialize loo-ahead boundary
for i = 1:length(obs) %extrapolate obstacle positons
    tr = obs(i).r;
    tc = obs(i).c;
    tv = obs(i).vel;
    w(tr,tc,1) = 1;
    for j = 1:lookahead-1
        switch obs(i).dir
            case 'N'
                tr = round(tr - tv);
                if tr < 1
                    tr = 1 + (tr * -1);
                    obs(i).dir = 'S';
                end
            case 'E'
                tc = round(tc + tv);
                if tc > cols
                    tc = cols - (tc - cols);
                    obs(i).dir = 'W';
                end
            case 'S'
                tr = round(tr + tv);
                if tr > rows
                    tr = rows - (tr - rows);
                    obs(i).dir = 'N';
                end
            case 'W'
                tc = round(tc - tv);
                if tc < 1
                    tc = 1 + (tc * -1);
                    obs(i).dir = 'E';
                end
            otherwise
                disp('Not Supported')
        end
        if (w(tr,tc,j+1) == 1)
            switch obs(i).dir
                case 'N'
                    tr = tr + 1;
                    if tr > rows
                        tr = tr - 1;
                    end
                case 'E'
                    tc = tc - 1;
                    if tc < 1
                        tc = tc + 1;
                    end
                case 'S'
                    tr = tr - 1;
                    if tr < 1
                        tr = tr + 1;
                    end
                case 'W'
                    tc = tc + 1;
                    if tc > cols
                        tc = tc - 1;
                    end
                otherwise
                    disp('Not Supported')
            end
        end
        w(tr,tc,j+1) = 1;
    end
end
%extrapolate goal location
tr = goal.r;
tc = goal.c;
tv = goal.vel;
w(tr,tc,1) = 0;
for j = 1:lookahead-1
    switch goal.dir
        case 'N'
            tr = round(tr - tv);
            if tr < 1
                tr = 1 + (tr * -1);
                goal.dir = 'S';
            end
        case 'E'
            tc = round(tc + tv);
            if tc > cols
                tc = cols - (tc - cols);
                goal.dir = 'W';
            end
        case 'S'
            tr = round(tr + tv);
            if tr > rows
                tr = rows - (tr - rows);
                goal.dir = 'N';
            end
        case 'W'
            tc = round(tc - tv);
            if tc < 1
                tc = 1 + (tc * -1);
                goal.dir = 'E';
            end
        otherwise
            disp('Not Supported')
    end
    if w(tr,tc,j+1) ~= 1
        w(tr,tc,j+1) = 0;
    end
end
world = w;
end
```

Forward chaining to find time to shallowest goal

forward_chain.m

```
function res = forward_chain(rows,cols,obs,goal,start,proplimit)
%
%forward chain to find time to shallowest goal
%
w = extend_world(rows,cols,obs,goal,proplimit);
t.r = start.r;
t.c = start.c;
t.l = 1;
q = qu_new;
q = qu_enqu(q,t);%initalize queue with start state
v(1:rows,1:cols,1:proplimit) = 0;
while qu_empty(q) ~= 1
    cu = qu_front(q);
    q = qu_dequ(q);%dequeue node
    v(cu.r,cu.c,cu.l) = 1;%mark node visited
    %if node is goal return depth
    if w(cu.r,cu.c,cu.l) == 0
        res = cu.l;
        q = qu_free(q);
        clear q;
        return;
    end
    %if chaining has reached propogation limit return reload
    if cu.l == size(w,3)
        res = -1;
        q = qu_free(q);
        clear q;
        return;
    end
    %enqueue non-obstacle sucessor nodes that have not been visited yet
    if w(cu.r,cu.c,cu.l) ~= 1
        if v(cu.r,cu.c,cu.l+1) ~= 1
            t.r = cu.r;
            t.c = cu.c;
            t.l = cu.l + 1;
            q = qu_enqu(q,t);
            v(cu.r,cu.c,cu.l+1) = 1;
        end
        if (cu.r > 1)&&(v(cu.r-1,cu.c,cu.l+1) ~= 1)
            t.r = cu.r - 1;
            t.c = cu.c;
            t.l = cu.l + 1;
            q = qu_enqu(q,t);
            v(cu.r-1,cu.c,cu.l+1) = 1;
        end
        if (cu.c > 1)&&(v(cu.r,cu.c-1,cu.l+1) ~= 1)
            t.r = cu.r;
            t.c = cu.c - 1;
            t.l = cu.l + 1;
            q = qu_enqu(q,t);
            v(cu.r,cu.c-1,cu.l+1) = 1;
        end
        if (cu.r < size(w,1))&&(v(cu.r+1,cu.c,cu.l+1) ~= 1)
            t.r = cu.r + 1;
            t.c = cu.c;
            t.l = cu.l + 1;
            q = qu_enqu(q,t);
            v(cu.r+1,cu.c,cu.l+1) = 1;
        end
        if (cu.c < size(w,2))&&(v(cu.r,cu.c+1,cu.l+1) ~= 1)
            t.r = cu.r;
            t.c = cu.c + 1;
            t.l = cu.l + 1;
            q = qu_enqu(q,t);
            v(cu.r,cu.c+1,cu.l+1) = 1;
        end
    end
end
res = -1; %return reload if queue empty.
q = qu_free(q);
clear q;
end
```

Determining look-ahead to use

find_good_la.m

```
function new_la = find_good_la(rows,cols,obs,goal,start,old_la,curveset)
%
%find a good look-ahead value to use
%
limits = 0.05;%set threshold
w = extend_world(rows,cols,obs,goal,old_la);
p = calc_pot_value(w);
stpot = p(start.r,start.c,1);
ti = 1;
for i = 1:16
   if curveset.cy(i) <= stpot
      ti = i;
   else
      break;
   end
end
la1 = curveset.cx(ti);
la0 = old_la;
la2 = 500;
for i = ti+1:16
   sig = std(curveset.pty((1:25)+25*(i-1)));
   mu = curveset.cy(i);
   z = (stpot - mu)/sig;
   zVal = normcdf(z);% z table value
   if (zVal <= limits)
      la2 = curveset.cx(i);
   else
      break;
   end
end
a = la2 - la0;
b = la2 - la1;
new_la = la0 + max([a b]);
end
```

Traversing world array to find goal

traverse_world.m

```
function [success path] = traverse_world(start,pot)
%
% traverse world
%
res(size(pot,1),size(pot,2))=0;
t = 1;
nr = start.r;
nc = start.c;
res(nr,nc,t)=1;
```

```
while((t<size(pot,3))&&(pot(nr,nc,t)~=0)&&(pot(nr,nc,t)~=1))
   if(nr ~= 1)
      n = pot(nr-1,nc,t+1);
   else
      n = Inf;
   end
   if(nc ~= size(pot,2))
      e = pot(nr,nc+1,t+1);
   else
      e = Inf;
   end
   if(nr ~= size(pot,1))
      s = pot(nr+1,nc,t+1);
   else
      s = Inf;
   end
   if(nc ~= 1)
      w = pot(nr,nc-1,t+1);
   else
      w = Inf;
   end
   st = pot(nr,nc,t+1);
   minv = min([n,e,s,w,st]);
   if (minv == st)
      %do nothing
   elseif((minv == n)&&(nr ~= 1))
      nr = nr - 1;
   elseif((minv == e)&&(nc ~= 10))
      nc = nc + 1;
   elseif((minv == s)&&(nr ~= 10))
      nr = nr + 1;
   elseif((minv == w)&&(nc ~= 1))
      nc = nc - 1;
   end
   t = t + 1;
   res(nr,nc,t)=1;
end
if(pot(nr,nc,t)==0)
   success = 1; %if goal reached
else
   success = 0; %if goal no longer reachable
end
path = res;
end
```

<u>Display path</u>

<u>show_path.m</u>

```
function show_path(world,path)
%
% Display the path to the user
```

```
%
i=1;
for t = 1:size(path,3)
    for r = 1:size(world,1)
        for c = 1:size(world,2)
            if world(r,c,t)==1
                ox(i)=r;
                oy(i)=c;
                oz(i)=t;
                i = i + 1;
            end
        end
    end
end
plot3(ox,oy,oz,'MarkerSize',30,'Marker','.','LineStyle','none','Color',[1 0 0])
grid
hold on
i=1;
for t = 1:size(path,3)
    for r = 1:size(world,1)
        for c = 1:size(world,2)
            if world(r,c,t)==0
                gx(i)=r;
                gy(i)=c;
                gz(i)=t;
                i = i + 1;
            end
        end
    end
end
plot3(gx,gy,gz,'MarkerSize',30,'Marker','.','LineWidth',3,'Color',[0 1 0])
i=1;
for t = 1:size(path,3)
    for r = 1:size(path,1)
        for c = 1:size(path,2)
            if path(r,c,t)==1
                x(i)=r;
                y(i)=c;
                z(i)=t;
                i = i + 1;
            end
        end
    end
end
plot3(x,y,z,'MarkerSize',30,'Marker','.','LineWidth',3,'Color',[0 0 1])
axis([1,size(path,1),1,size(path,2),1,size(path,3)])
view(-16,4)
hold off
end
```

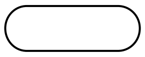
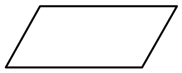
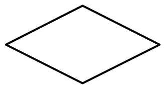
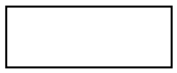
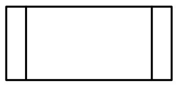
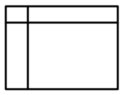
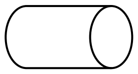
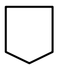
Temporal path planning simulator

planner.m

```
function planner(rows,cols,obs,start,goal)
%
% path planner simulator
%
density = size(obs,2)/(rows*cols);% find obstacle density
ndensity = 9;
for d = 9:10:89
   if d <= density
      ndensity = d;
   else
      break;
   end
end
csetname = sprintf('curveset_%d_%d_%d',rows,cols,ndensity);
if(evalin('base',sprintf('exist(''%s'',''var'')',csetname))==0)% try to load curveset from matlab
runtime memory
   if(evalin('base',sprintf('exist(''%s.mat'',''file'')',csetname))==0)% try to load curveset from
hard drive
      disp('Generating curve set');
      evalin('base',sprintf('gen_curves(%d,%d,%d)',rows,cols,ndensity));% generate curveset
   else
      disp('Loading curve set from file');
      evalin('base',sprintf('load %s.mat',csetname));
   end
   curveset = evalin('base', csetname);
else
   disp('Loading curve set from memory');
   curveset = evalin('base', csetname);
end
clear csetname;
sol1 = forward_chain(rows,cols,obs,goal,start,100);% find time to reach shallowest solution
if (sol1 == -1)
   disp('Run program after reloading world');% Dynamics changed
   return;
end
la = find_good_la(rows,cols,obs,goal,start,sol1,curveset);% estimate lookahead to use
world = extend_world(rows,cols,obs,goal,la);% extend world in time
pot_vals = calc_pot_values(world);% Calculate potential values
[res path] = traverse_world(start,pot_vals);% Follow negative potential gradient
if (res == 0)
   disp('Run program after reloading world');% Dynamics changed or Goal became
unreachable
   return;
end
show_path(world,path)% display path traversed
end
```

APPENDIX B

FLOWCHART SYMBOLS

	**Terminator** – Used to denote start or end of a process
	**I/O** – Used to denote input or output operation
	**Decision** – Used to represent conditional statements
	**Process** – Used to represent internal process
	**Pre Defined Process** – Used to represent processes that are defined elsewhere, i.e. other modules.
	**Internal Storage** – Used to represent values that are stored internally.
	**Direct Access Storage** – Used to represent values that are stored in the main workspace of MATLAB [30]
	**Out of Page Connector** – Used to link to flowchart sections not in current page
——→	**Program Flow** – Represents the direction of control flow in the flowchart
- - - - - - ->	**Data Flow** – Represents the direction of the flow of data in the flowchart

REFERENCES

[1] Russell S., Norvig P. (1995). Artificial Intelligence: A Modern Approach, Second Edition. Englewood Cliffs, NJ: Prentice Hall.

[2] LaValle S. M. (2006). Planning Algorithms. New York, NY: Cambridge University Press.

[3] The Fraunhofer-Chalmers Research Centre for Industrial Mathematics. Automatic Path Planning. http://www.fcc.chalmers.se/geo/intro-1/automatic-path-planning.

[4] Honda Technology Research Institute Company, Limited. (2000 –). ASIMO. http://asimo.honda.com/

[5] Sony Corporation. (1999 – 2006). AIBO. http://support.sony-europe.com/aibo

[6] Song G., Amato N. M. (2001). Using motion planning to study protein folding pathways. Journal of Computational Biology. pp 287–296.

[7] Thrun S., Bücken A. (1996). Integrating grid-based and topological maps for mobile robot navigation. Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence. Vol 2, p 944. Portland, OR.

[8] Rodriguez-Losada D., Matia F., Galan R. (2006). Building geometric feature based maps for indoor service robots. Robotics and Autonomous Systems. vol 54, no 7, pp 546–558.

[9] Lozano-Pérez T. (1983). Spatial planning: A configuration space approach. IEEE Trans. Computers. Vol 32, no 2, pp 108–120.

[10] Latombe J-C. (1991). Robot Motion Planning. Norwell, MA: Kluwer Academic.

[11] Khatib O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. International Journal of Robotics Research. Vol 5, no 1, pp 90–98.

[12] Faria G., Romero R., Prestes E., Idiart M. (2004). Comparing harmonic functions and potential fields in the trajectory control of mobile robots. Proceeding of the 2004 IEEE Conference on Robotics, Automation and Mechatronics. Vol 2, pp 762–767.

[13] Borenstein J., Koren Y., Member S. (1991). The vector field histogram - fast obstacle avoidance for mobile robots. IEEE Journal of Robotics and Automation. Vol 7, pp 278–288.

[14] Park M. G., LEE M. C. (2003). A new technique to escape local minimum in artificial potential field based path planning. KSME International Journal. Vol 17, n 12, pp 1876–1885.

[15] Ge S. S., Cui Y. J. (2000). New Potential Functions for Mobile Robot Path Planning. IEEE Transactions on Robotics and Automation. Vol 16, no 5, pp 615–620.

[16] Zhang P.-Y., Lü T.-S., Song L.-B. (2004). Soccer robot path planning based on the artificial potential field approach with simulated annealing. Robotica. Vol 22, pp 563–566.

[17] Weisstein E. W., Harmonic Function. MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/HarmonicFunction.html

[18] Weisstein E. W., Laplace's Equation. MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/LaplacesEquation.html

[19] The Laplace Equation and Harmonic Functions. MathPages – A collection of HTML lectures on various subjects in mathematics and physics. http://www.mathpages.com/home/kmath214/kmath214.htm

[20] Weisstein E. W., Saddle Point. MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/SaddlePoint.html

[21] Connolly C. I., Grupen R. A. (1993). On the Applications of Harmonic Functions to Robotics. Journal of Robotic Systems. Vol 10, pp 931–946.

[22] Connolly C. I., Grupen R. A. (1994). Nonholonomic Path Planning using Harmonic Functions. University of Massachusetts at Amherst Computer Science Department technical report UM-CS-1994-050.

[23] Risell J., Iñiguez P., (2005) Path Planning using Harmonic Functions and Probabilistic Cell Decomposition. Proceedings of the 2005 IEEE International Conference on Robotics and Automation.

[24] Kazemi M., Mehrandezh M., Gupta K. (2005). Sensor-based robot path planning using harmonic function-based probabilistic roadmaps. Proceedings of 12th International Conference on Advanced Robotics. pp 84–89.

[25] e Silva Jr. E. P., Engel P. M., Trevisan M., Idiart M. A. P. (2002). Exploration method using harmonic functions. Robotics and Autonomous Systems. Vol 40, no 1, pp 25-42.

[26] Trevisan M., Idiart M. A., Prestes E., Engel P. M. (2005), Exploratory navigation based on dynamical boundary value problems. Journal of Intelligent and Robotic Systems. Vol 45, no 2, pp 101-114.

[27] Kim J.O., Khosla P.K. (1992). Real-Time Obstacle Avoidance using Harmonic Potential Functions. IEEE Transactions on Robotics and Automation. Vol 8, no 3, pp 338–349.

[28] Ge S. S., Cui Y. J. (2002). Dynamic motion planning for mobile robots using potential field method. Autonomus Robots. Vol 13, pp 207–222.

[29] Poty A., Melchior P., Oustaloup A. (2004). Dynamic Path Planning for mobile robots using fractional potential field. First International Symposium on Control, Communications and Signal Processing. pp 557-561.

[30] Wu W., QiSen Z., Mbede J. B., Xinhan H. (2001). Research on Path Planning for Mobile Robot among Dynamic Obstacles. Joint 9th IFSA World Congress and 20th NAFIPS International Conference. Vol 2, pp 763–767.

[31] The MathWorks. (2007). MATLAB (v7.5.0 R2007b). http://www.mathworks.com/products/matlab/

[32] Keren Y. (2001). Data Structures & Algorithms Toolbox. http://www.mathworks.com/matlabcentral/fileexchange/212

[33] Zolotykh N. (2007). MATLAB Pointer Library. http://code.google.com/p/pointer/

BIOGRAPHICAL INFORMATION

Vamsikrishna Gopikrishna received his Bachelors in Computer Science & Engineering from Sri Venkateshwara College of Engineering under Anna University, Chennai, India in 2006. He began his graduate career at the University of Texas at Arlington from the Spring of 2007. He received his masters in Computer Science and Engineering in the Fall of 2008. His research interests include Artificial Intelligence, Robotics and Data Mining. His Paper on *TC-ID3: A TESTCODE based ID3 Classifier for Protein Coding Region Identification* was accepted for oral presentation at International Conference on Computational Intelligence for Modeling, Control and Automation held at Sydney, Australia in November 2006, whose proceedings were published by IEEE Computer Society. He has also done independent research on Improvements to A* Algorithm under the guidance of Dr. Deepak Khemani of Indian Institute of Technology, Chennai. He has also an avid coder and software developer having developed a Banking-Customer Care system for Microsoft Student Project Program using Microsoft .NET technologies and a LAN chat system for use in office environment using socket programming in Visual C++ for Pentasoft Technologies, Chennai during In-Plant Training.