Path Planning Algorithms for Autonomous Mobile Robots

MohammadAli AskariHemmat

A Thesis

in

The Department

of

Mechanical, Industrial and Aerospace Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

July 2018

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **MohammadAli AskariHemmat**

Entitled: **Path Planning Algorithms for Autonomous Mobile Robots**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Mechanical Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Suong Van Hoa — Chair
Chair's name

Dr. Javad Dargahi — Examiner
Examiner's name

Dr. Wei-Ping Zhu — Examiner
Examiner's name

_____ — Co-Supervisor
Co-Supervisor's name

Dr. Youmin Zhang — Supervisor
Supervisor's name

Approved by _____
Chair of the MIAE Department

_____ 2018 _____
Dean of Engineering

ABSTRACT

Path Planning Algorithms for Autonomous Mobile Robots

MohammadAli AskariHemmat

This thesis work proposes the development and implementation of multiple different path planning algorithms for autonomous mobile robots, with a focus on differentially driven robots. Then, it continues to propose a real-time path planner that is capable of finding the optimal, collision-free path for a nonholonomic *Unmanned Ground Vehicle* (UGV) in an unstructured environment. First, a *hybrid A* path planner* is designed and implemented to find the optimal path; connecting the current position of the UGV to the target in real-time while avoiding any obstacles in the vicinity of the UGV. The advantages of this path planner are that, using the *potential field* techniques and by excluding the nodes surrounding every obstacles, it significantly reduces the search space of the traditional A* approach; it is also capable of distinguishing different types of obstacles by giving them distinct priorities based on their natures and safety concerns. Such an approach is essential to guarantee a safe navigation in the environment where humans are in close contact with autonomous vehicles. Then, with consideration of the kinematic constraints of the UGV, a smooth and drivable geometric path is generated. Throughout the whole thesis, extensive practical experiments are conducted to verify the effectiveness of the proposed path planning methodologies.

To my loving family

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Dr. Youmin Zhang, for his excellent guidance, caring, patience, and providing me with an excellent atmosphere for doing research. This thesis would not have been possible without his guidance and support. I would like to thank all my fellow researchers and colleagues in Networked Autonomous Vehicles (NAV) Lab at Concordia University. Without their guidance, support and continual encouragements, this thesis would not have been possible. I am especially grateful to Dr. Zhixiang Liu , Ban and Dr. Yiqun Dong. They have proven to be supportive friends as well as thoughtful colleagues with good advice and collaboration. Finally, my deepest and most heartfelt thanks to my family; my parents who provided the best possible environment for me to grow up and supported me in all my pursuit and my brother and sister for all their love and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

LTI	Linear Time Invariant
PID	Proportional —Integral —Derivative
MP	Motion Planning
PP	Path Planning
RTOS	Real Time Operating System
ML	Machine Learning
GD	Gradient Descent
AGD	Adaptive Gradient Descent
AI	Artificial Intelligence
RT	Real Time
PF	Potential Function
APF	Artificial Potential Function
LIFO	Last In First Out
FIFO	First In First Out
BFS	Breadth First Search
DFS	Depth First Search
RRT	Rapidly-exploring Random Tree

Chapter 1

Introduction

The topic of path planning for mobile, car-like, robots has consistently been in the center of attention for the past thirty years. Researches are still proposing new algorithms with higher performance and accuracy. During the last decade, improvements in computational power, easier and cheaper access to hardware and software platforms has helped researches develop innovative algorithms and build on top existing ones. Algorithms that implementing them might have been unfeasible a few years ago, are now being implemented on robots thanks to cheap, fast and affordable hardware.

There are different challenges in *Path Planning* for mobile robots. However, the final goal of all algorithms is to find an optimal and safe path. Optimality of a path can be interpreted differently based on the use case but usually optimality of a path implies how short the path is. A safe path on the other hand is a path that guides the robot to safely travel around obstacles, both static and dynamic ones.

Based on the application, there might be solutions to the path planning problem but they are usually designed with limiting assumptions in mind, assumptions that render the algorithm and solution useless in another scenario or under slightly different assumptions. There have been numerous attempts in the past 20 years to improve the cruise control of cars and not only help the drivers with monitoring and controlling the speed but also with lane changing and navigation. The current solutions usually depend on visual lane finding techniques and driving the car within a lane. However, as soon as the markings on the road

become unclear, due to snow, raining or any other environmental changes, the navigation would fail to find a safe, drivable path.

Automation, transportation and manufacturing industries are undergoing a remarkable revolution thanks to technology advancement in Artificial Intelligence and Machine Learning. The cornerstone of all these advancements lies in the DARPA's 2004, 5005 and 2007 Grand and Urban Challenges [1]. These competitions proved that autonomous driving is possible and since then, there has been a huge spike of interest in this industry. As it is shown in Figure 1.1 the search trend for *Autonomous Cars* has steadily increased [2].

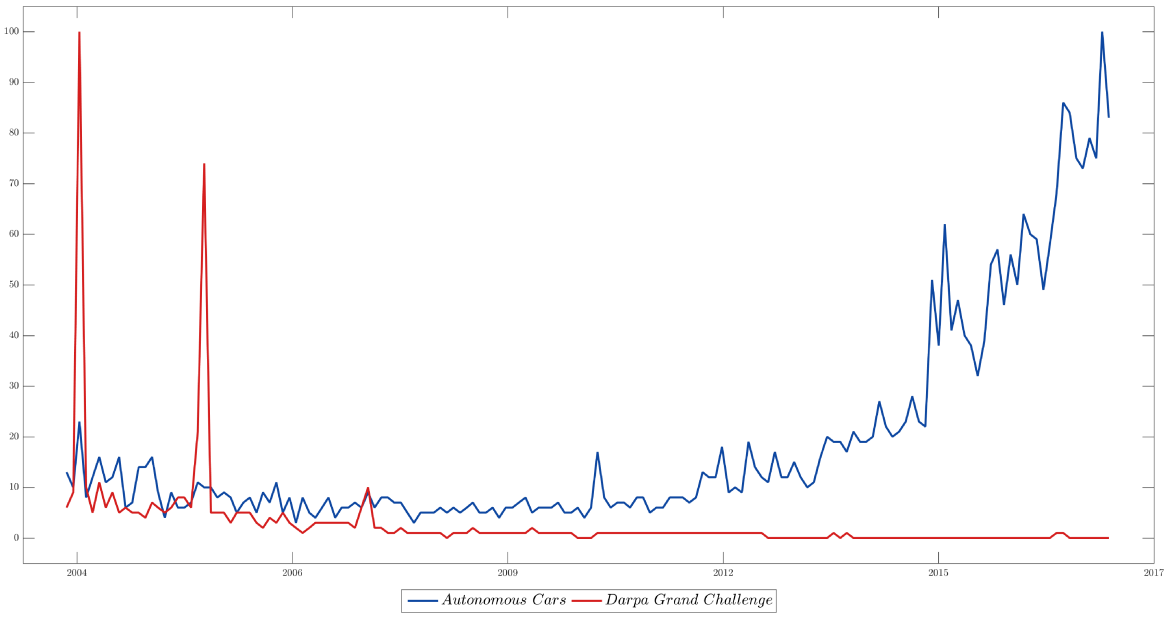


Figure 1.1: Google trends result for the *Autonomous Cars* and *Darpa Grand Challenge* queries

1.1 Motivation and context

Quanser has provided Networked Autonomous Vehicles (NAV) lab with a set of Quanser Ground Vehicles, also known as QGVs. As autonomous driving will play an essential role in the coming years in Automation and Transportation, the aim with this thesis is to provide students and research centers background and working simulations and toolboxes

to perform research on the provided platform.

In this thesis, the focus is to analyze different path planning algorithms which were developed over years and evaluate their strengths and weaknesses. The feedback control approaches to solve the path planning problem has usually failed. The main reason is that feedback control has traditionally approached this problem where the work space and environment is free of obstacles, or the obstacles are static. The attempts to provide a feedback control law in presence of obstacles are usually extremely limited in practice. This work will present such attempts and will discuss why they are so limited and then propose a set of feasible solutions that are relatively easy to implement as base path planner and then expand upon with a more sophisticated algorithm. The proposed algorithms do not depend on strong mathematical background and are fast enough to find collision free paths even in dynamic workspaces.

1.2 Problem description

The goal of this thesis is to provide enough resources to solve a rather generic path planning problem. The path planning problem can be summarized as:

Generating a smooth path in real-time that is drivable by a nonholonomic robot. The smooth, optimal path should start from an arbitrary state x_s and should end in an arbitrary set X_{Goal} while avoiding static and dynamic obstacles. The algorithm should explicitly determine if there is no valid path between initial and goal configurations.

Historically this has been the definition of a path planning problem [25]. The path planners usually assume perfect sensing for localization and exact control. This implies that the workspace and the robot's state is perfectly known at all times. In this thesis a set of similar assumptions are made:

- The localization is a solved problem and we have the exact location and state of the robot and obstacles with high certainty and low latency at all times.

- The size of the obstacles are large enough to be detected by the localization system.
- Dynamic obstacles will only move in a short time span and will stay rested for the majority of the time.

1.3 Structure of the thesis

In Chapter 2, a short background for nonholonomic systems is given and theoretical concepts for controlling such systems are presented. Chapter 3 and Chapter 4 go through different approaches to solve the path planning problem and will discuss how these algorithms have failed to provide a solution for the problem. Chapter 5 will use the theoretical foundation presented in Chapter 4 and proposes a solution. The implementation details and results are also discussed in this chapter. Finally, in Chapter 6 some suggestions for future work is presented and the shortcomings of the proposed algorithm is discussed.

Chapter 2

Mathematical Modeling

2.1 Introduction

In order to design controllers for systems, one should first know how the system functions and behave. The differential drive robot is one of the most common models used in robotics research due to multiple reasons which will become apparent in this chapter. In Section 2.2 the kinematic model of a differential wheeled robot is derived. Then it is shown that it is a driftless control-affine system and then its controllability is discussed [22].

2.2 Kinematic model for differentially driven robots

Differential drive robots are very popular for indoor robotic experiments because they are very easy to build from scratch due to their simple structural design. The kinematic model is also very intuitive and easy to derive and maybe the most appealing reason is the simple control laws required to control such systems. The robot has 2 main wheels, each of which is powered independently using a DC motor. To add stability to the robot and preventing it from falling a third passive wheel, caster wheel, is added to the rear/front of the robot. The steering of the robot is maintained by rotating the wheels at different rates and thus move the robot around the environment.

The movement of the robot in the workspace is the result of combining two basic

motions, pure translation and pure rotation. Assuming non-zero angular velocity, if the velocities have the same magnitude and the same sign $u_r = u_l$, the robot wheel have a pure translational motion and if the signs are opposite $u_r = -u_l$, the robot will have a pure rotational motion.

By controlling the angular velocity of the two wheels one could control the motion of the robot in the working space. So the control vector becomes the angular velocity of the right and left wheels.

$$u = (u_l, u_r) \tag{2.1}$$

We are interested to know where the robot is located in a fixed reference frame and at which direction it is heading to so the state vector, configuration vector, of the system is $[x, y, \theta]^T$. Since there are only two control inputs and the system has 3 states, we have an under-actuated system. Now, the question is which point on the robot should we care about and control, a point at the front, a point on the wheel, or some where else?

In a pure rotational motion $u_r = -u_l$, the middle point of the axle between the two wheels does not move. Assigning the origin of the local coordinate system on the middle point would satisfy the pure rotational motion because that point does not have any translational motions under this condition. So it would be easier to analyze the system by selecting such coordinate system. If the position of this point and the orientation of the local frame with respect to the global reference frame is known at all times one could localize the entire system.

As it can be observed from Figure 2.1 there are only 2 geometrical measurements necessary to construct the kinematic model of a differential drive robot. The vertical distance between the center of the two wheels L and the radius of the wheels r. Assuming two arbitrary velocities for the right and left wheels and by using instantaneous center of velocity the the linear and angular velocity of the mid-point can be easily found.

$$\dot{\theta} = \frac{v_R - v_L}{L} \tag{2.2}$$

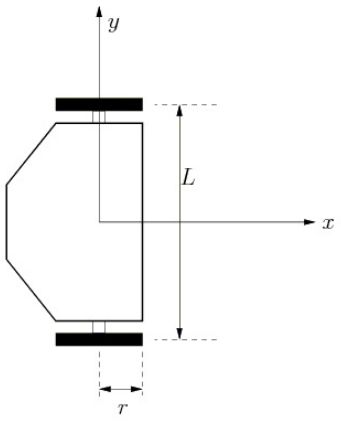


Figure 2.1: Geometry of a generic differentially driven robot [22]

$$v = \frac{v_R + v_L}{2} \tag{2.3}$$

and by assuming the radius of each wheel is r the linear velocity is $v_R = r \times u_R$ and substituting it in equation 3 would yield:

$$\dot{\theta} = \frac{r}{L}(u_r - u_l) \tag{2.4}$$

Now that the angular and linear velocity of the mid-point is known, it is easy to derive the state-transition equation. In order to find the projection of the velocity on the x and y axis simply multiply the linear velocity v, by $\cos\theta$ and $\sin\theta$.

$$\begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= \omega \end{aligned} \tag{2.5}$$

2.3 Controlling a differentially driven robot

Rewriting the kinematic model of the differential drive robot, Equation (2.5), in matrix form would yield

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{2.6}$$

and in vector form it will be

$$\dot{\boldsymbol{q}} = \boldsymbol{A}\boldsymbol{u} \tag{2.7}$$

comparing Equation (2.7) to differential equation model of a linear system

$$\dot{q} = f(q,u) = Aq + Bu \tag{2.8}$$

we notice that it is not a linear system so it must be nonlinear. The general form of a first-order nonlinear equation is

$$\dot{\boldsymbol{q}} = \boldsymbol{f}(\boldsymbol{q},\boldsymbol{u},\boldsymbol{t}) \tag{2.9}$$

and if the system is affine in control input the general form will be

$$\dot{\boldsymbol{q}} = \boldsymbol{f_1}(\boldsymbol{q},\boldsymbol{t}) + \boldsymbol{f_2}(\boldsymbol{q},\boldsymbol{t})\boldsymbol{u} \tag{2.10}$$

Equation (2.10) represents a family of nonlinear differential equations called control affine systems. These systems are linear in action but the states of the system evolve in a nonlinear fashion [22]. If the first term $f_1(q,t)$, is zero, the system becomes a driftless nonlinear control system.

$$\dot{\boldsymbol{q}} = \boldsymbol{f_2}(\boldsymbol{q},\boldsymbol{t})\boldsymbol{u} \tag{2.11}$$

By comparing Equation (2.11) and Equation (2.7) it is trivial that the kinematic model of the differential drive robot is indeed a driftless nonlinear system. The nonholonomic nature of wheeled mobile robots has precise consequences in terms of structural properties of the kinematic model. The first, and most important one, is that in spite of the reduced number of degrees of freedom, wheeled robot is controllable in its configuration space; i.e.,

given two arbitrary configurations, there always exists a kinematically admissible trajectory (with the associated velocity inputs) that transfers the robot from one to the other. Since the kinematic model is driftless, a well known result implies that it is controllable if and only if the accessibility rank condition holds. The motion control problem for wheeled mobile robots is generally formulated with reference to the kinematic model.

There are essentially two reasons for taking this simplifying assumption. First, the kinematic model fully captures the essential nonlinearity of single-body wheeled robots, which stems from their nonholonomic nature. This is another fundamental difference with respect to the case of robotic manipulators, in which the main source of nonlinearity is the inertial coupling among multiple bodies. Second, in mobile robots it is typically not possible to directly command the wheel torques, because there are low-level wheel control loops integrated in the hardware or software architecture. Any such loop accepts as input a reference value for the wheel angular speed, which is then reproduced as accurately as possible by standard regulation actions (e.g., PID controllers). In this situation, the actual inputs available for high-level control are precisely these reference velocities.

Several methods are available to drive a wheeled mobile robot in feedback along a desired trajectory. A straightforward possibility is to first compute the linear approximation of the system along the desired trajectory (which, unlike the approximation at a configuration, results to be controllable) and then stabilize it using linear feedback. Only local convergence, however, can be guaranteed with this approach. For the kinematic model of the unicycle, global asymptotic stability may be achieved by suitably morphing the linear control law into a nonlinear one [10].

In robotics, a popular approach for trajectory tracking is input - output linearizion via static feedback. In the case of a unicycle, consider as output the Cartesian coordinates of a point B located ahead of the wheel, at a distance b from the contact point with the ground. The linear mapping between the time derivatives of these coordinates and the velocity control inputs turns out to be invertible provided that b is nonzero; under this assumption, it is therefore possible to perform an input transformation via feedback that converts the unicycle to a parallel of two simple integrators, which can be globally stabilized with a simple

proportional controller (plus feedforward). This simple approach works reasonably well. However, if one tries to improve tracking accuracy by reducing b (so as to bring B close to the ground contact point), the control effort quickly increases. Trajectory tracking with b (i.e., for the actual contact point on the ground) can be achieved using dynamic feedback linearizion. In particular, this method provides a one-dimensional dynamic compensator that transforms the unicycle into a parallel of two double integrators, which is then globally stabilized with a proportional-derivative controller (plus feedforward). In contrast to static feedback linearizion, no residual zero dynamics is present in the transformed system. However, the dynamic compensator has a singularity when the unicycle driving velocity is zero. This is expected, because otherwise the tracking controller would represent a universal controller. Note that dynamic feedback realizability using the x, y outputs is related to them being flat, the two properties are equivalent.

2.3.1 Accessibility and controllability

For linear systems, $x = Ax + Bu$ where $x \in R^n$ and $u \in R^m$ and the celebrated Kalman rank condition fully characterizes when the system is (globally) controllable (from any point). Our objective here is to come up with similar tests for nonlinear systems. Let us start by making precise the notions of accessibility and controllability [10].

2.3.2 Configuration space

It is easy to imagine the height of the robot does not have any effect on the motion planning algorithm and the generated paths so let's consider the 3-D rigid body of the robot does not have a height, the result would be a 2-D plane. So the motion planning algorithm must generate a path for the new rigid body, the plane, in $\mathbb{R}^2$. As mentioned in Section 2.3 the transformation matrix Equation (2.6) on page 8 could transform any $x, y \in \mathbb{R}$ this would yield a manifold $M_1 = \mathbb{R}^2$. Also one could apply any rotation $\theta \in [0, 2\pi)$ which would yield another manifold $M_2 = \mathbb{S}^1$. So the following manifold covers all possible motions

$$C = \{(x,y,\theta) \mid (x,y) \in \mathbb{R}^2, \theta \in [0,2\pi)]\} = \mathbb{R}^2 \times \mathbb{S}^1 = M_1 \times M_2 \tag{2.12}$$

The new manifold is called the Configuration Space of the system and might be considered as a special case of the state space [22]. The configuration space of the system looks like a torus but the cross section is a square instead of a circle. Topologically speaking, it is important to realize the configuration space is not bounded and it does not have a boundary.

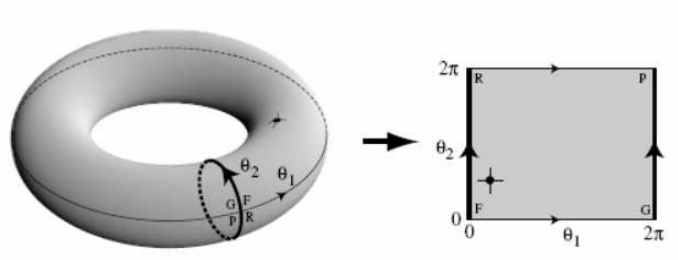


Figure 2.2: A visualization of *Configuration Space* for a double pendulum [5]

Figure 2.2 shows how the systems moves in the configuration space. As it can be seen due to the periodic nature of rotations the system reaches it's starting yaw angle after a complete rotation.

It is crucial to understand the physical space the robot moves in is called the *Work Space* $\mathbb{W}$ and it is a subset of $\mathbb{R}^2$ while the *Configuration Space* is a 3-manifold and it is a subset of $\mathbb{R}^3$ and this is where the state of the system changes. The concept of configuration space might seem to be too abstract and not so useful for motion planning of differential drive robots but this abstraction makes it possible to use similar motion planning algorithms for different problems. The configuration space lets us abstract the motion planning problem from a geometric point of view to a topological one and then use topological tools and find a path and then convert the topological path to a geometric one.

2.3.3 Configuration space obstacles

While defining the *Configuration Space* it was assumed there are no obstacles. But there are such constraints in the configuration space and they should be removed. This removed

section is called the *Configuration Space Obstacles* and the rest of the configuration space is called the *Free Space* and the generated path must solely be in this section.

Let's assume there is some obstacle region O in the *Work Space*, $O \subset W$. Also the robot rigid body $A \subset W$ is defined. If $q \in C$ represents the configuration of the rigid body A, the obstacle configuration space is defined as:

$$C_{obs} = \{q \in C \mid A(q) \cap O \neq \emptyset\} \tag{2.13}$$

This new configuration space is basically the set of all possible configuration of the robot, rigid body, at which it intersects the obstacle region, O. Since the sets $A(q)$ and O are closed sets, the obstacle region is a closed set in C. The rest of the configurations make up the *free space* and it is denoted as $C_{free} = C \backslash C_{obs}$. This *free space* ,$C_{free}$, is an open set. Being an open set means that the rigid body, robot, can come arbitrarily close to the obstacle region and still be in the C_{free} [22].

2.3.4 Definition of a motion planning problem

The classic example of a motion planning problem is the Piano Mover's problem. The problem is to find a collision-free path from some start configuration to a goal configuration for a 3-D rigid body among a known set of obstacles. It is assumed that the rigid body is capable of omnidirectional movements. This problem can be formulated as [22], [5]:

1. A world $\mathcal{W}$ in which either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$.

2. A semi-algebraic obstacle region $\mathcal{O} \subset \mathcal{W}$ in the world.

3. A semi-algebraic robot is defined in $\mathcal{W}$. It may be a rigid robot $\mathcal{A}$ or a collection of m links, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m$.

4. The configuration space $\mathcal{C}$ determined by specifying the set of all possible transformations that may be applied to the robot. From this, $\mathcal{C}_{obs}$ and $\mathcal{C}_{free}$ are derived.

5. A configuration, $q_I \in \mathcal{C}_{free}$ designated as the initial configuration.

6. A configuration $q_G \in \mathcal{C}_{free}$ designated as the goal configuration. The initial and goal configurations together are often called a query pair (or query) and designated as (q_I, q_G).

7. A complete algorithm must compute a (continuous) path, $\tau : [0,1] \rightarrow \mathcal{C}_{free}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$, or correctly report that such a path does not exist.

Other aspects of this problem that might need more attention is that, it is considered that the obstacles are perfectly known and they are stationary. The execution of the planned path is exact. Because the path is planned before execution, it is called offline motion planning [5].

The key issue is to make sure no point on the rigid body hits an obstacle. We use the configuration space concept to represent the configuration of all points on the rigid body and check for possible collisions.

Let's consider the case showed in Figure 2.3. The two squares are stationary and we are trying to pass the rectangle between them. It is really hard to consider all different orientations that the squares or the rectangle can take, and decide if the rectangle can pass through the squares. If there was a way that we could expand the square and shrink the rectangle such that the rectangle becomes a point in space, it would much easier to figure out the possible collision. Because it is just a matter of checking if a point falls into a specific set. The algorithm mentioned below lets us shrink the robot to a point and expand the obstacles, so we don't have to worry about the weird geometry of the obstacle and the robot. It would be much easier to plan a path for a point compared to a 2-D rigid body.

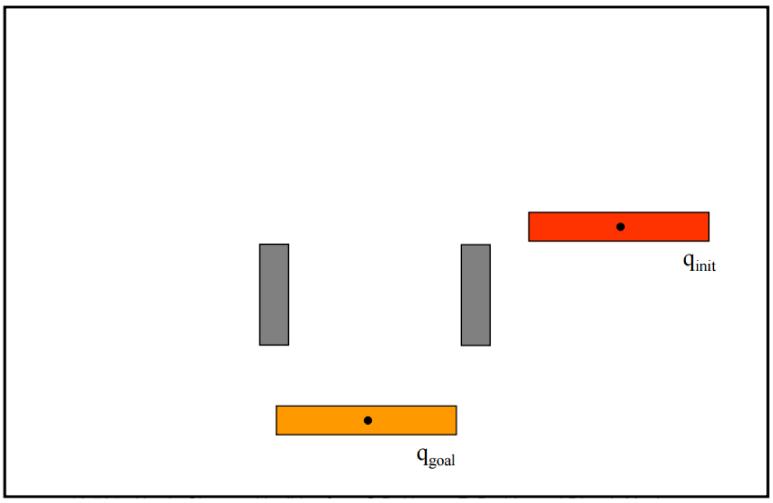


Figure 2.3: A *Work Space* with start and goal states of a 2D rigid body [5]

There are different ways to explicitly model the configuration space obstacle [5] [22]. There is a rather simple method for configuration spaces in $\mathbb{R}^n$ where $n = 1, 2, 3$. For example if we consider a robot which is only capable of pure translation, but not rotation, the configuration space would be $\mathcal{C} = \mathbb{R}^2$. For such a C-space the C-Space obstacle can be found using the *Minkowski Sum*.

$$A \oplus B = \{a + b \in \mathbb{R}^n \mid a \in A, b \in B\} \tag{2.14}$$

The *Minkowski Sum* is a way to add, or subtract, geometrical shapes. A circular rigid body makes the calculation of the Minkowski Sum much more computationally feasible and thus for the sake of simplicity, it is usually considered that the robot is circular and it would drastically improve computing the configuration space obstacle.

According to the definition of the configuration space obstacle, we have to find the configurations at which the robot touches the obstacles. It is very intuitive to see that if the distance between the center of the robot and the perimeter of any obstacle is less than the radius of the robot, there will be a collision. So by just easily expanding the boundary of each obstacle, and shrinking the boundary of the walls in the *Work Space* with respect to the radius of the robot we can find the *Configuration Space Obstacle* [5]. Thus the problem of motion planning for a circular robot in Work Space, is transformed to a simpler problem:

Motion planning for a point in the Work Space.

As we have seen in Section 2.2 on page 5 the differential drive robot has 3 degrees of freedom, two for translation in $\mathbb{R}^2$ and one for rotation and its C-Space is $\mathbb{R}^2 \times \mathbb{S}^1$. Also as mentioned earlier using the *Minkowski Sum* only works for a case where the $\mathcal{C} = \mathbb{R}^n$, where $n = 1, 2, 3$. But if we consider the robot has a fixed orientation the configuration space would be reduced to a 2-manifold. So it is intuitive to reduce the C-Space of the robot for a given orientation, and find the respective C-Space obstacle. Repeating this for different orientations and stacking the respective C-Space would result a 3-manifold which is the C-Space for the robot. As shown in Figure 2.4 each slice represents the configuration space of the robot at one fixed orientation.

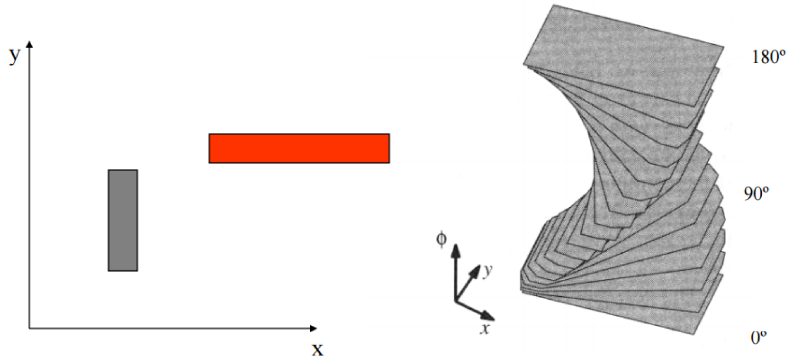


Figure 2.4: A configuration space with start and goal states of a rigid body [5]

Chapter 3

Potential Functions

Previous attempts to solve the path planning problem usually find a collision free path but the proposed algorithms do not provide any guarantees and feasibility if the robot can follow the generated path. The algorithms were usually open loop and there were no answers on what should happen when the robot deviates from the generated path. This was the main motivation for feedback based path planning and eventually potential functions. In this chapter the theoretical background for potential function and their shortcomings are discussed.

3.1 Introduction

Potential Functions are one of the earliest methods of motion planning for mobile robots. Due to ease of implementation and efficiency of the algorithm, potential functions were popular for real-time collision avoidance, specially for the cases where the configuration space is not well defined and the robot does not have a clear model for the configuration space obstacle [22].

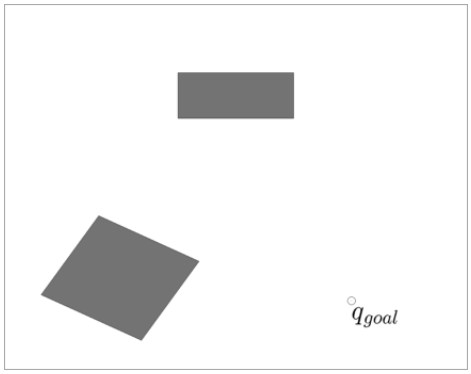
Figure 3.1 shows a discretized work space with the associated potential field. The major concern while working with potential functions is the selection of a differentiable real-valued function $U : \mathbb{R}^n \rightarrow \mathbb{R}$. The function U is illustrated in Figure 3.1d. At any given point in the work space the potential function has a real value which could be considered

as the energy of the moving particle at that point. By measuring the negated gradient of the potential function U, the force applied to the particle at any given point can be calculated and by assigning the value of the negated gradient $-\frac{\partial U}{\partial x}$ and $-\frac{\partial U}{\partial y}$ to any point on the workspace the vector field is generated. This generated vector field would direct any particle from any given start point towards the predefined goal point. An intuitive metaphor to understand this algorithm is to consider the 2-D rigid body of the robot as one single point in the configuration space where according to the position of this particle a specific force drives the point towards the goal.

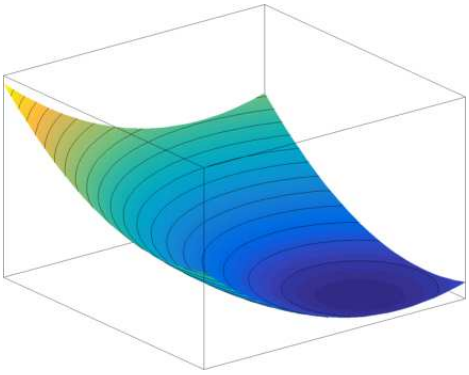
Generally it is considered that the robot mass point is a positively charged particle and the goal configuration is considered to be negatively charged, and thus pulling the robot towards itself. While the obstacles are positively charged, pushing the robot away from the C-Space obstacle. The combination of attractive and repulsive potentials create this force field. As illustrated in Figure 3.2 on page 19 if we consider the robot as a point, it will follow a path downhill towards the goal point, regardless of where the starting configuration is located.

One of the points that make potential fields very interesting is that, this method can be used as a feedback motion planner for any mass point robot. At any given point there is a vector directing the robot towards the goal, so the controller should only have to control the heading of the robot and make sure it follows the right direction. The reference to the controller is the heading, and it could be easily controlled using a PI controller. By constructing a feedback control plan over this continuous space we could generate a trajectory and use trajectory tracking methods and to track it. This would make potential functions a closed loop feedback motion planning algorithm. As shown in Figure 3.2 on page 19 for all points in the work space there is a vector defined that could direct the robot towards the goal.

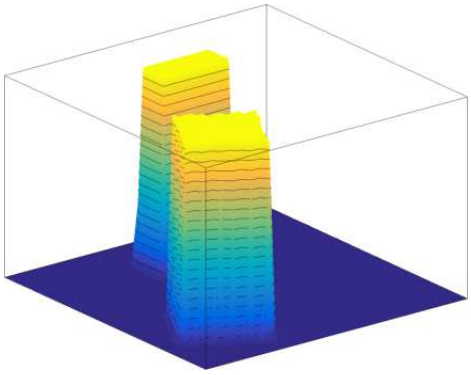
As intuitive and as simple as this method is, it has its own draw backs. Consider the case where the potential function introduces a local minima to the potential field, due to the geometry of the obstacles or simply due to the position of the obstacle relative to the goal. In this case if the starting configuration is close enough to this minima, or the path passes close



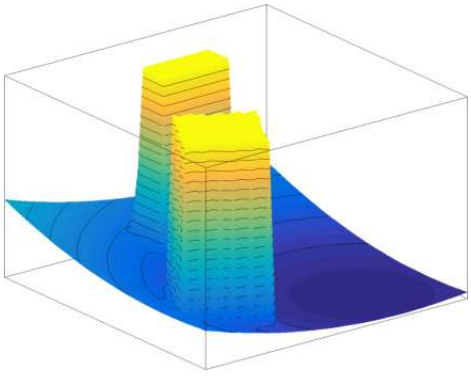
(a) A Work Space with two rectangular obstacles



(b) Attractive potential field



(c) Repulsive potential field



(d) Total potential field

Figure 3.1: The total potential field is simply the sum of the attractive and repulsive fields

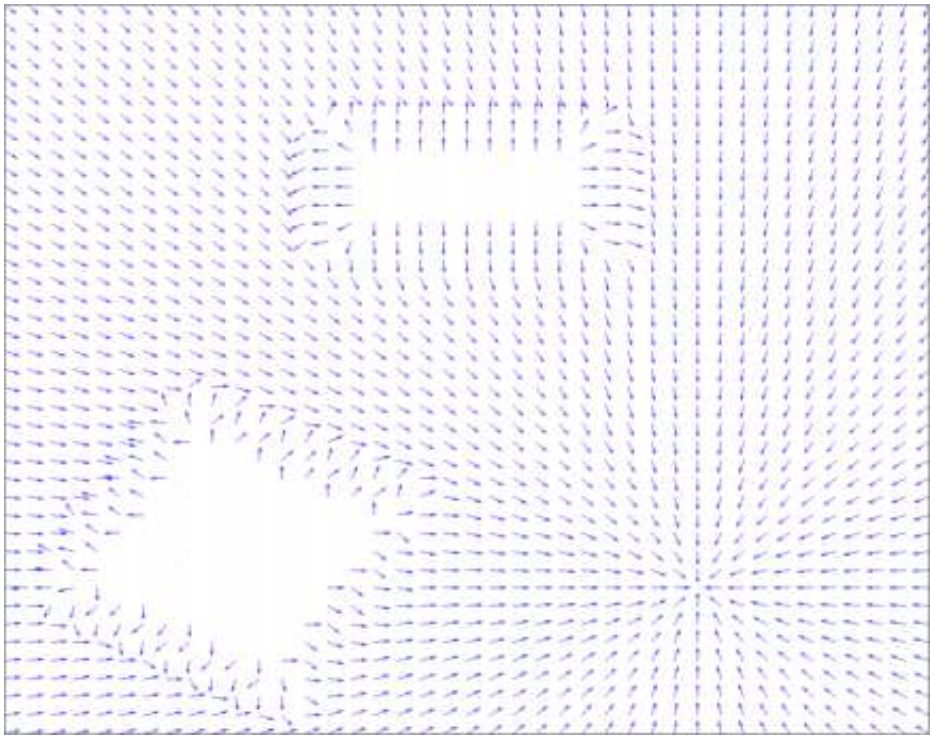


Figure 3.2: Negated gradient vector field

by this local minima, the robot might get trapped in the local minima and will never rich the goal. There are different potential functions other than the attractive/repulsive potential, but almost all of them suffer from the same problem, which is the existence of local minima. That's why potential functions are not considered as a complete motion planner.

3.2 Potential field for $\mathcal{C} = \mathcal{R}^2$

The function defined below was introduced by Khatib [18] and it is probably the most famous potential function for mass point robots in $\mathbb{R}^2$ and even $\mathbb{R}^3$. Let's first construct the artificial forces applied to a point in $\mathbb{R}^2$. The potential should be a differentiable function, $U : \mathcal{C} \to \mathcal{R}$. The artificial forces can be easily found by finding the negated gradient vector:

$$\vec{F}(q) = -\vec{\nabla}U(q) \tag{3.1}$$

where q is the configuration of the point, $q = (x, y)$ and $\vec{\nabla}U(q) = [\frac{\partial U}{\partial x}, \frac{\partial U}{\partial y}]^T$. Notice that

since we are working with just a point the configuration q does not consider the orientation of the robot, θ. The total potential function is constructed as the sum of the attractive and repulsive potential functions:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{3.2}$$

3.3 Attractive potential

The role of the attractive potential is to drive the robot to the goal configuration. Maybe the simplest function that could play the role of the attractive potential is the euclidean distance.

$$U(q) = K_{att} d(q, q_{goal}) \tag{3.3}$$

the value of the function is always positive and somehow represents an error, the distance between current configuration and the desired configuration. This function only has one global minimum at the goal configuration where the potential is zero. The gain K_{att} is used to change the effect of the attractive potential function. If the gain is higher the attractive potential will be higher. The gradient of Equation (3.3) is

$$\begin{aligned} \frac{\partial U(q)}{\partial x} &= \frac{K_{att}(x - x_{goal})}{d(q, q_{goal})} \\ \frac{\partial U(q)}{\partial y} &= \frac{K_{att}(y - y_{goal})}{d(q, q_{goal})} \end{aligned} \tag{3.4}$$

Selecting this gradient function results in a linear change in the force exerted on the robot and when implemented it will result in a non-smooth motion. Also as illustrated in Figure 3.3 on the next page the gradient is not defined if $q = q_{goal}$ and the function becomes non-differentiable. We can simply use a quadratic potential function instead of the conic potential function to have a smooth differentiable function.

$$U_{att}(q) = \frac{1}{2} K_{att} d^2(q, q_{goal}) \tag{3.5}$$

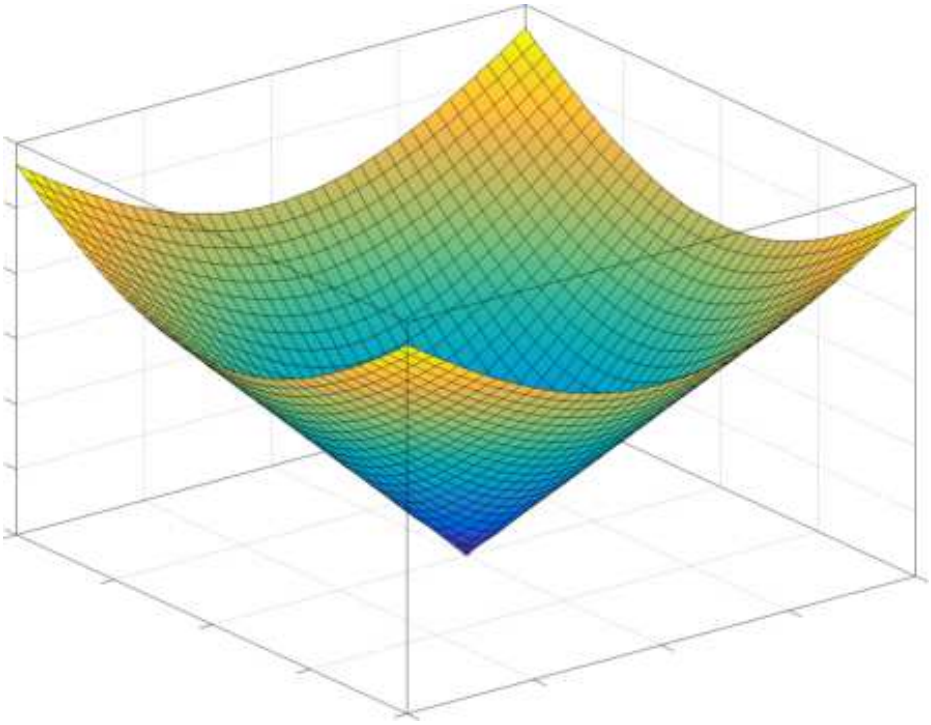


Figure 3.3: Conic potential field

and the gradient would be

$$\begin{aligned} \frac{\partial U_{att}(q)}{\partial x} &= K_{att}(x - x_{goal}) \\ \frac{\partial U_{att}(q)}{\partial y} &= K_{att}(y - y_{goal}) \end{aligned} \tag{3.6}$$

The quadratic potential function also provides a smooth vector field as the robot approaches the goal. When the robot is far away from the goal, the gradient has a higher value and as the robot approaches the goal the gradient decreases. The $\frac{1}{2}$ fraction is added to simplify the gradient function. As it can be seen in Figure 3.4 on the following page the quadratic potential function is smooth and differentiable.

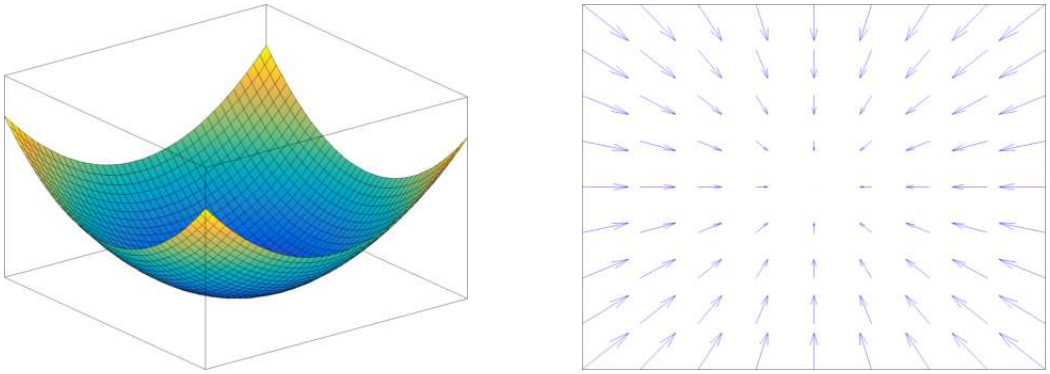


Figure 3.4: Smooth & differentiable attractive potential field

3.4 Repulsive potential

The repulsive potential helps the robot stay away from the obstacles or the work space boundaries. Also it is desirable that the the robot does not get under the influence of the obstacles when it is far from them. It is also assumed the obstacles are convex, if they are not some decomposition algorithm must be used to make all obstacles convex. Equation (3.7) encapsulates this concept. For each obstacle $\mathcal{CO}_\rangle$ the distance function $D_i(q)$ is the distance between the current location of the robot to the closest point on the obstacle.

$$U_{rep,i}(q) = \begin{cases} \frac{1}{2}K_{rep}(\frac{1}{D_i(q)} - \frac{1}{D^*}), & \text{if } D_i(q) \leq D_i^* \\ 0, & \text{if } D_i(q) > D_i^* \end{cases} \tag{3.7}$$

D_i^* is the threshold distance and represents the range of influence of obstacle $\mathcal{CO}_{\rangle\{\}}$. It is interesting to note that this threshold distance is not necessarily similar for all obstacles and it can have different values according to the type of the obstacle. Just like the K_{att}, the effect of the repulsive function could be tuned using a gain K_{rep}. As the distance between the robot and obstacle $\mathcal{CO}_\rangle$ is decreased the value of the potential function is increased and tends to infinity.

The total repulsive potential field is obtained by adding the effect of all obstacles. Given n obstacles the total potential field is

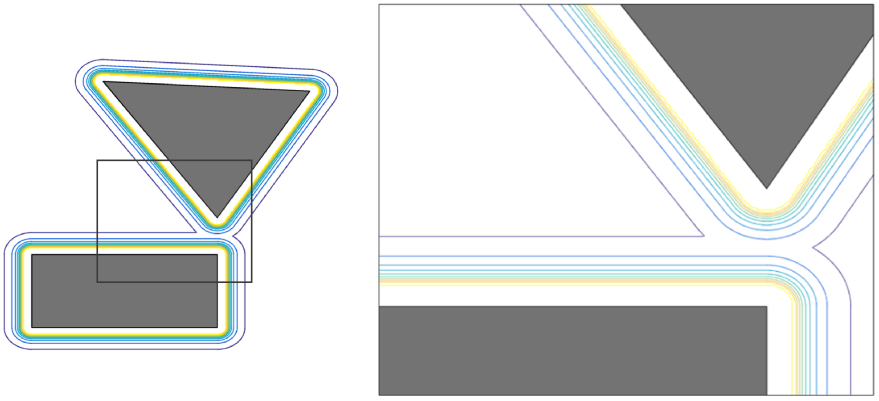


Figure 3.5: Equipotential contour of repulsive function around obstacles

$$U_{rep}(q) = \sum_{i=1}^{n} U_{rep,i} \tag{3.8}$$

The gradient of Equation (3.7) on the preceding page is

$$\nabla U_{rep,i}(q) = \begin{cases} \frac{1}{2} K_{rep} (\frac{1}{D^*} - \frac{1}{D_i(q)}) \frac{1}{D_i^2(q)} \nabla D_i(q), & \text{if } D_i(q) \leq D_i^* \\ 0, & \text{if } D_i(q) > D_i^* \end{cases} \tag{3.9}$$

thus the total force experienced by the robot from the obstacles is

$$F_{rep} = -\sum_{i=1}^{n} \nabla U_{rep,i} \tag{3.10}$$

3.5 Motion planning using APF

Once the attractive and repulsive potential functions are defined it is very straightforward to find the total potential function

$$U_{total} = U_{rep} + U_{att} \tag{3.11}$$

and the vector field, which represents the artificial force experienced by the robot in the work space is

$$\begin{aligned} \nabla U_{total} &= \nabla U_{att} + \sum_{i=1}^{n} \nabla U_{rep,i} \\ F_{total} &= -\nabla U_{att} - \sum_{i=1}^{n} \nabla U_{rep,i} \end{aligned} \tag{3.12}$$

Once the vector field is constructed there are different approaches on how to use this vector field and drive the robot towards the goal [22].

- Consider the vector field as a vector of generalized forces that make the robot move in a certain way according to the current configuration of the robot and the dynamic model of the robot

$$\tau = F_{total}(q) \tag{3.13}$$

- Consider the vector field as a velocity field which describes the velocity of the robot in the configuration space

$$\dot{q} = F_{total}(q) \tag{3.14}$$

In this thesis we only deal with the kinematic model of a robot, so the second approach is more attractive. Once the desired velocity of the robot in the configuration space is known, the robot could be controlled using the kinematic model and Equation (3.14) provides the reference velocity to the controller. The motion planner does not have to provide a trajectory, i.e. a profile of the velocity or acceleration along the path, so it is logical to assume a constant velocity along the path. So, to make things easier it's usually assumed that the final vector field is normalized.

3.5.1 Continuous motion planning

As it was mentioned before, it is assumed the system is a point in the working space and thus there are no constraints on the system. So, the representation of the system is $\dot{x} = f(x,u) = u$, which represents a fully actuated system. Now given:

1. A world $\mathcal{W}$, obstacles $\mathcal{O}$, robot $\mathcal{A}$, and configuration space $\mathcal{C}$

2. An input space U

3. A state transition equation $\dot{q} = -u$

4. An initial configuration $q_{init} \in \mathcal{C}_{free}$ and a goal set $q_{goal} \subset \mathcal{C}_{free}$

the motion planner should return a set of waypoints from the initial configuration to the goal configuration. The input space U, is actually the total vector field over the workspace. Notice that the the goal configuration must be a set of valid configurations. Usually it is reasonable enough to accept a path which makes the system get close enough to the goal. To find the way points the most common choice is the simple numeric integration of state transition equation using the Euler method

$$q_{i+1} = q_i - \alpha U(q_i) \tag{3.15}$$

Equation (3.15) could also be considered as the gradient descent algorithm. Once the initial configuration is known, the robot would move step by step in the direction guided by the force field, which is the negated gradient of the potential field. The only tricky part of this algorithm is selecting how fast the robot should move towards the goal. If the steps that the robot is taking are too big, the robot might pass the goal and/or oscillate around the goal configuration. On the other hand if the steps are too small, it might take a long time for the robot to reach the goal [5].

Algorithm 1 Gradient Descent

$q(0) = q_{start}$

$i = 0$

while $\nabla U_{total}(q(i)) \neq 0$ **do**

$q(i+1) = q(i) - \alpha \nabla U_{total}(q(i))$

$i = i + 1$

end while

The input of the algorithm is the start configuration, a function to calculate the force field related to the current state $q(i)$ and some scalar coefficient α, which decided how far at each step the robot should proceed. The larger this coefficient, the larger the step. It is worth mentioning that α is not necessarily a constant, and it could be dynamically changed. A good approach for changing α is to select a larger value at the beginning and as the robot gets closer to the goal decrease the coefficient [5]. Also, it is almost impossible for the condition $\nabla U_{total}(q(i)) \neq 0$ to ever become true. So a more relaxed condition is usually used $\|\nabla U_{total}(q(i))\| < \varepsilon$, where ε is selected based on the condition of the task at hand, the smaller the ε the closer the robot will be to the goal. That is why the goal configuration is a set rather than a single point. Consider the working space illustrated in Figure 3.6. The goal is to plan a path from q_{start} to q_{goal}.

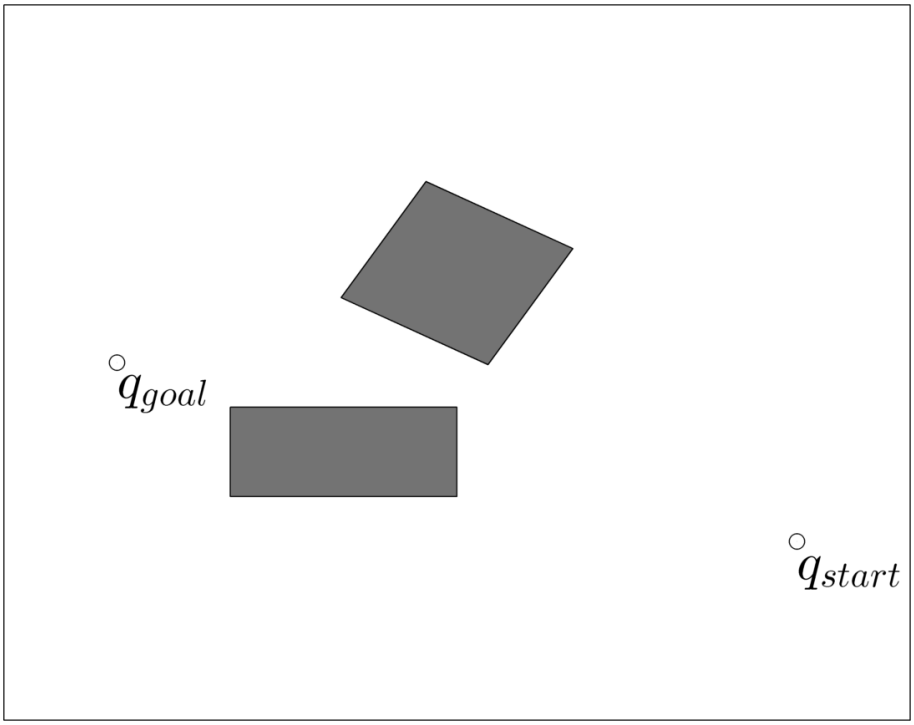


Figure 3.6: A work space with rectangular obstacles

Figure 3.7 shows the planned path using simple gradient descent with a constant step size $\alpha = 0.2$. But as it can be seen there are a lot of oscillations in the generated path. Unfortunately these oscillations are one of the negative points about potential functions.

But fortunately the problem can be usually solved using a more sophisticated optimization method. One of the methods to go around this problem is to update the step size α adaptively. It's very intuitive that when the system gets stuck in these oscillations one of the ways to get out earlier is to have a larger step size or use the information from the previous iteration and decide the step size based on it for the new iteration and thus a smoother path.

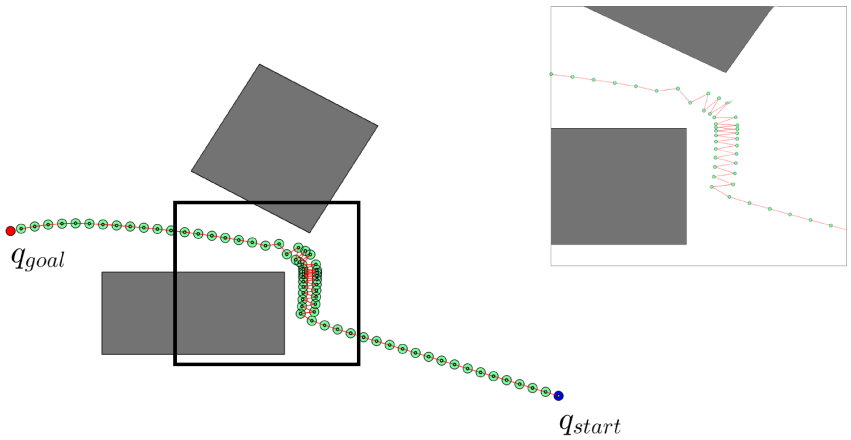


Figure 3.7: Planned path generated by simple integration with oscillations

One of the methods to find the optimized α is to solve the following optimization problem, usually called a line search problem:

$$\underset{\alpha}{\operatorname{argmin}}\{U(q-\alpha\nabla U_{total}(q))\} \tag{3.16}$$

if we consider $-\alpha\nabla U_{total}(q) = \Delta q$ the problem becomes

$$\underset{\alpha}{\operatorname{argmin}}\{U(q+\Delta q)\} \tag{3.17}$$

This basically means find the α which takes the system to the lowest possible potential in the current direction. But the problem is that we don't have an explicit definition of the potential function at hand. The easiest way to solve this problem is to find the Taylor's expansion of the potential function which provides a very good approximation of the value of the total potential at any given point$U$.

$$U(q+\Delta q) = U(q) + b(q)^T \Delta q + \frac{1}{2}\Delta q^T A(q)\Delta q \tag{3.18}$$

where $b(q)$ is the gradient of the potential function and $A(q)$ is the Hessian matrix calculated at q. Equation (3.18) will be minimized by the solution to $A(q)q = b$. As proved here [33] the value of the α which minimizes equation Equation (3.18) can be calculated using equation Equation (3.19)

$$\alpha = \frac{r^T r}{r^T A r} \tag{3.19}$$

where the residual $r = b - Aq$ shows the error between the correct value of b and its' estimated value. Once α is calculated the following algorithm is used to update the state of the system

Algorithm 2 Adaptive Gradient Descent

$q(0) = q_{start}$

$i = 0$

while $\nabla U_{total}(q(i)) \neq 0$ **do**

$b = \nabla U(q_i)$

$A = Hess(U(q_i))$

$r_i = b - A * q_i$

$\alpha_i = \frac{r_i^T r_i}{r_i^T A r_i}$

$q(i+1) = q(i) - \alpha_i \nabla U_{total}(q(i))$

$i = i + 1$

end while

Figure 3.8 shows the same work space as the one in Figure 3.6 but the path is generated using Algorithm 2. Although the generated path is much smoother and there are not that many oscillations, but we have to realize what we are giving to gain this smoother path. In the simple gradient descent algorithm we only had to calculate the gradient and we had full control of the step size α, but here not only we have to calculate the hessian matrix at any given point but we also have to deal with the conservative nature of this algorithm

which chooses much smaller time steps. Figure 3.9 shows a comparison of the value of the step size in the two different methods. As it can be seen the normal implementation of Euler method, general gradient descent, has much less number of iterations, around 150. But The adaptive gradient descent algorithm has gone through around 650 iterations to converge to the goal point. It is an obvious observation when you consider how small the step size α is for a large section of the adaptive algorithm. The average value of α in adaptive algorithm is 0.011 while the constant step size foe the general gradient descent algorithm is 0.1 which is almost 10 times larger. For cases where the normal gradient descent algorithm has a hard time converging to the goal or there are a lot of oscillations it might be better to use the adaptive gradient descent algorithm.

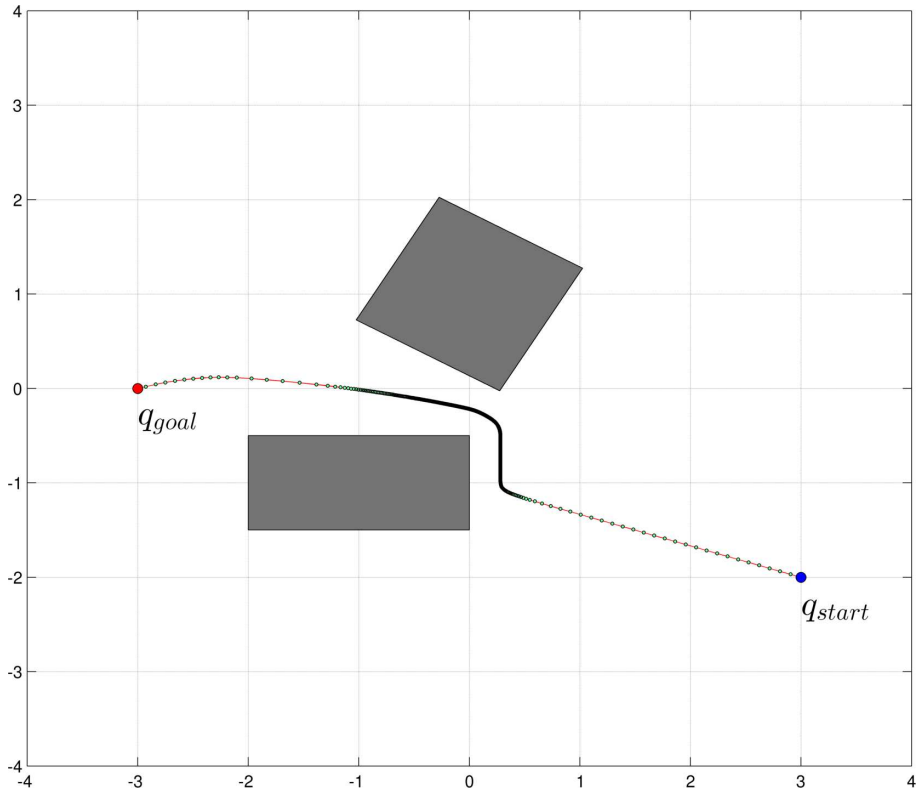


Figure 3.8: Planned path by updating α at each iteration

But this won't stop us from using adaptive gradient descent. Actually the main reason for using gradient descent is cases where the normal gradient descent algorithm does/can not converge to the goal. Consider the same work space illustrated in Figure 3.10 which

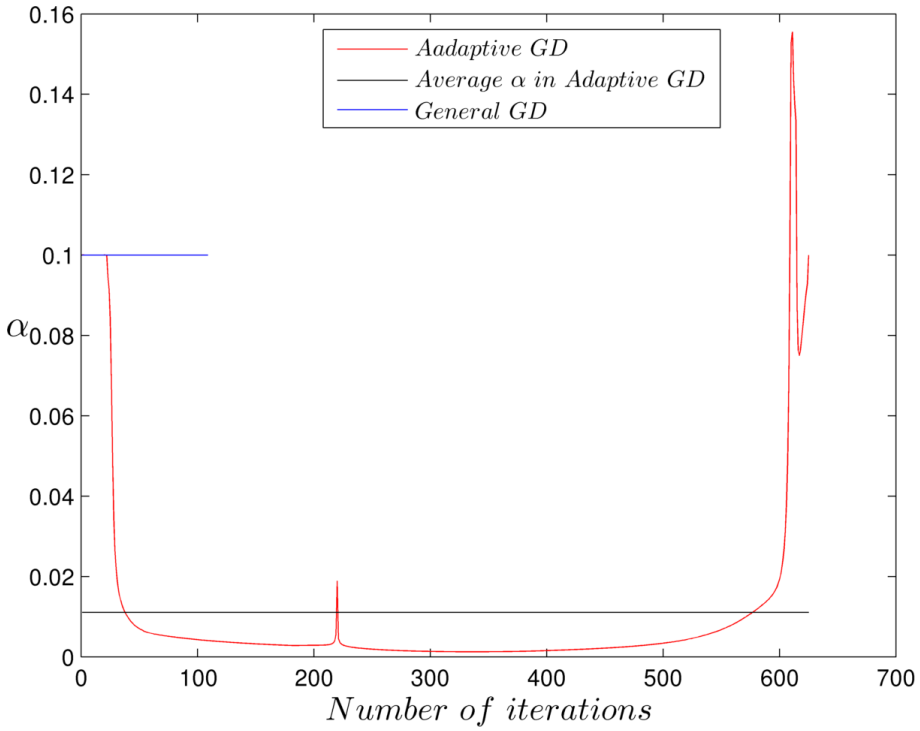
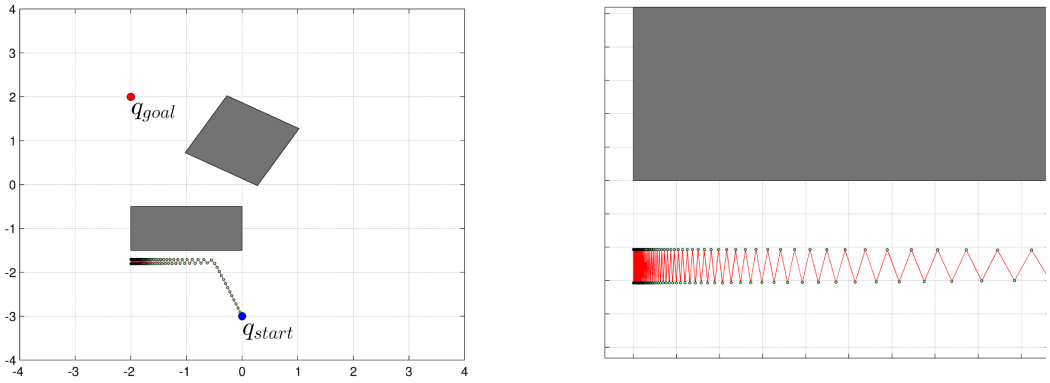


Figure 3.9: Dynamics of α as a function of number of iterations

is the same as Figure 3.6 but with different locations for the start and goal configurations. Running the general gradient descent algorithm on this problem would not generate a path. As shown in Figure 3.10 there are a lot of oscillations close to the obstacle and the algorithm did not converge even after 4000 iterations. But on the same work space and start/goal configurations the adaptive gradient descent algorithm converges and a smooth path is planned. The path is found after almost 1650 iterations.

There are also other algorithms such as conjugate gradient descent, newton's method and also the momentum gradient descent which can solve this problem. In nature they are very similar to the previously discussed algorithms but they have different running times, number of iterations, and behavior depending on the problem.

So far we have solved one of the big problems of potential functions which is numerous oscillations close to the obstacles or between them in a corridor. But there is another challenging problem regarding the potential functions. Consider the work space in figure 3.12, it seems even simpler than the previous work space as there is only one obstacle



(a) Unsuccessful path

(b) Oscillations close to the obstacle

Figure 3.10: General gradient descent on different start/goal configurations might not converge

present. But the location of the start and goal configuration with reference to the obstacle make these type of work spaces interesting and challenging. If we imagine a line between start and goal configurations, it would be completely perpendicular to the obstacle boundaries. At such situations the attractive and repulsive force are co-linear but in opposite directions and this would result an area where these forces are equal but in opposite directions thus they balance each other out. These areas are called local minima and their effect in potential functions have been extensively studied over the years [32] [20] [7].

Figure 3.12 shows the contour of potential function on this work space. The circular lines are equipotential areas. The area shown on the left of the rectangular obstacle is the local minima. If the movement direction of the robot is perpendicular to the side of an obstacle this area appears due to how the forces are defined and the robot will get stuck in this area as shown in Figure 3.13.

There is no deterministic gradient descent algorithm to find a path in such vector fields. The same problem appears in a lot of different areas of engineering. There are different proposed algorithms to solve this problem a few them are mentioned and described here

- Wave-Front planner [5], which probably is the easiest algorithm to solve the local

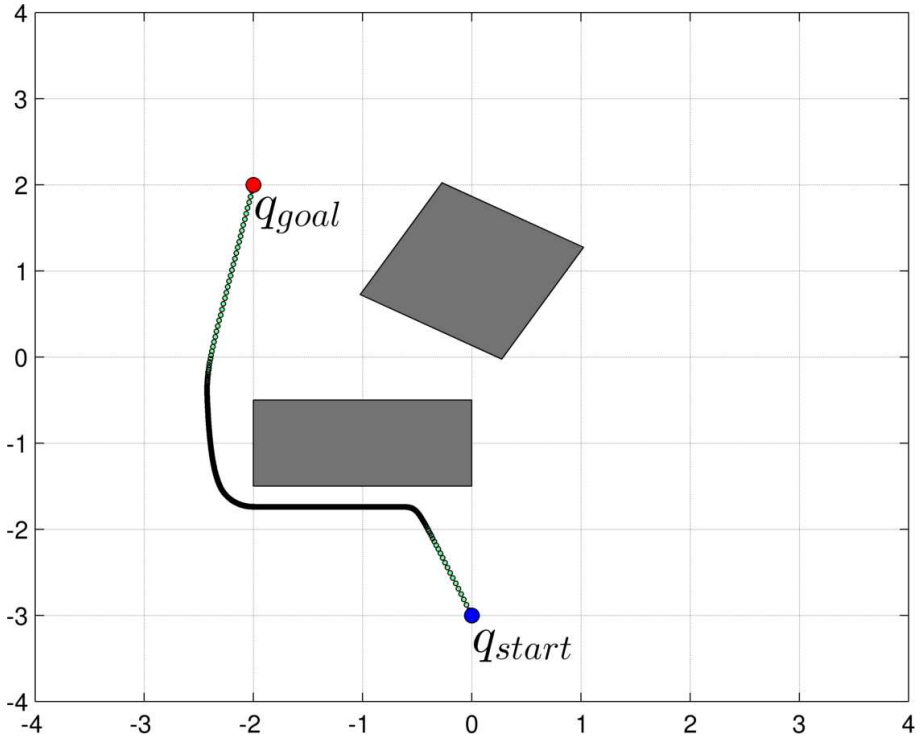
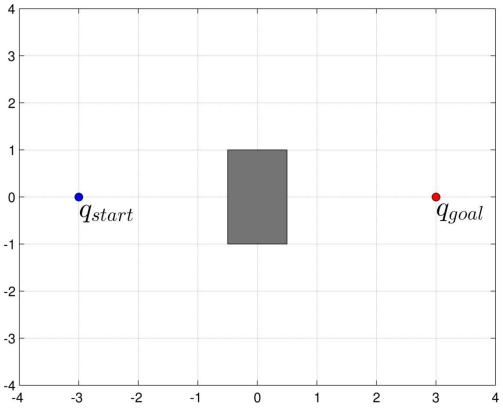


Figure 3.11: Planned path by updating α at each iteration

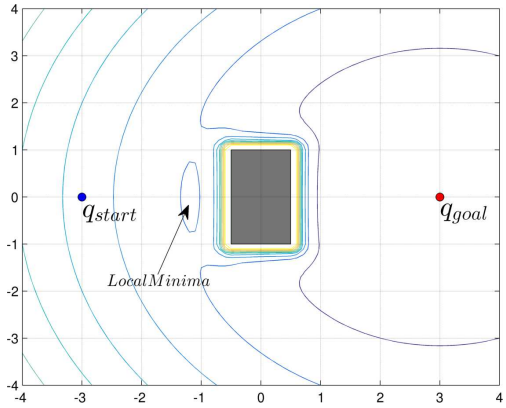
minima problem. The algorithm is described in Subsection 3.5.3, discrete motion planning.

- Rimon and Koditschek [32] developed an analytic method to find a special family of potential functions called *Navigation Functions* which just like potential functions would result in a velocity field but there are no spurious local minima and there is only a single minimum located at the goal. Maybe the most distinctive property of such potential functions is that they must be a *Morse* 1 [28] function to satisfy the single global minimum criteria. A *Morse* function is a function where all critical points the Hessian are nondegenerate. Just like the potential functions this method assumes a repulsive force from the obstacles and an attractive force from the goal. This method is also described in Subsection 3.5.2

- Connolly et. al. [8] proposed a special family of navigation functions which are numerical solutions to Laplace's heat equation and they are usually called *Harmonic*



(a) Even a simple work space can have the local minima problem



(b) The potential function contour over the workspace exposing local minima

Figure 3.12: Local minima in a workspace when the direction of movement is perpendicular to one of the obstacles in the workspace

Potential Functions. Harmonic potential functions hold all conditions for a navigation function except being a *Morse* function due to the possibility of isolated degenerate saddle points.

A function ϕ is called a harmonic function if it satisfies the differential equation

$$\nabla^2\phi = \sum_{i=1}^{n} \frac{\partial^2\phi}{\partial x_i^2} = 0 \tag{3.20}$$

Usually finite element methods are used to solve for the solution of Equation (3.20). In order to solve the equation one must define some conditions on the boundary of the domain over which the function ϕ is defined. Usually either *Dirichlet boundary condition* or *Neumann boundary condition* or a superposition of the two conditions is used depending on the work space. Here lies one of the problem with Harmonic potential functions, they require an explicit boundary of the free space $\mathcal{C}_{free}$ and it's usually avoided in path planning algorithms. Also the numerical solution might be feasible in low dimensions but in higher dimensions it is expensive [22].

- Choset et. al. [6] and also Lavalle [27] [26] have proposed different algorithms where

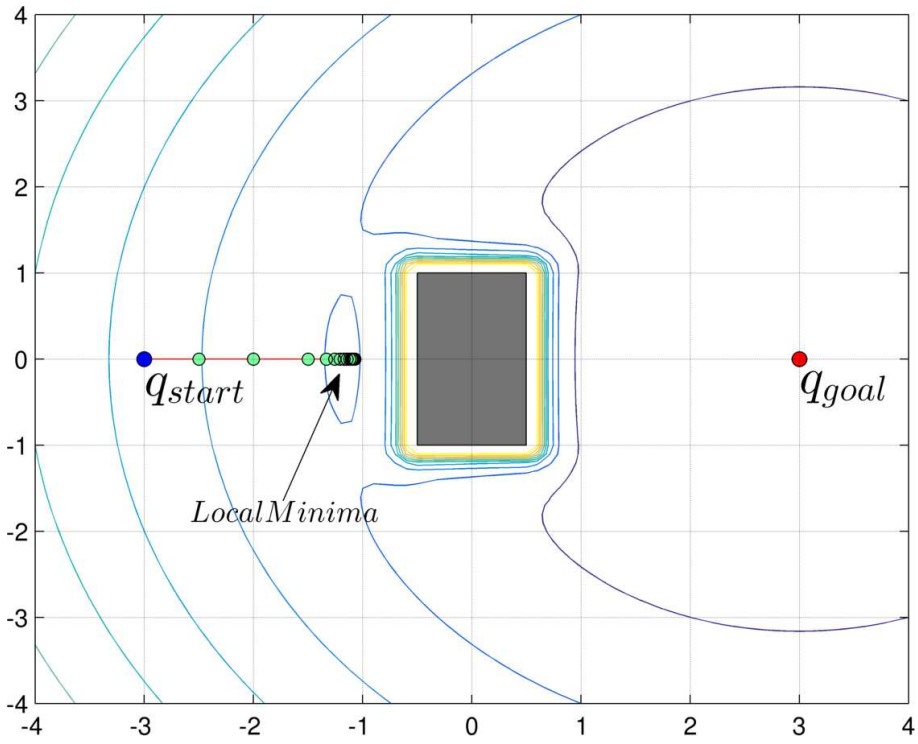
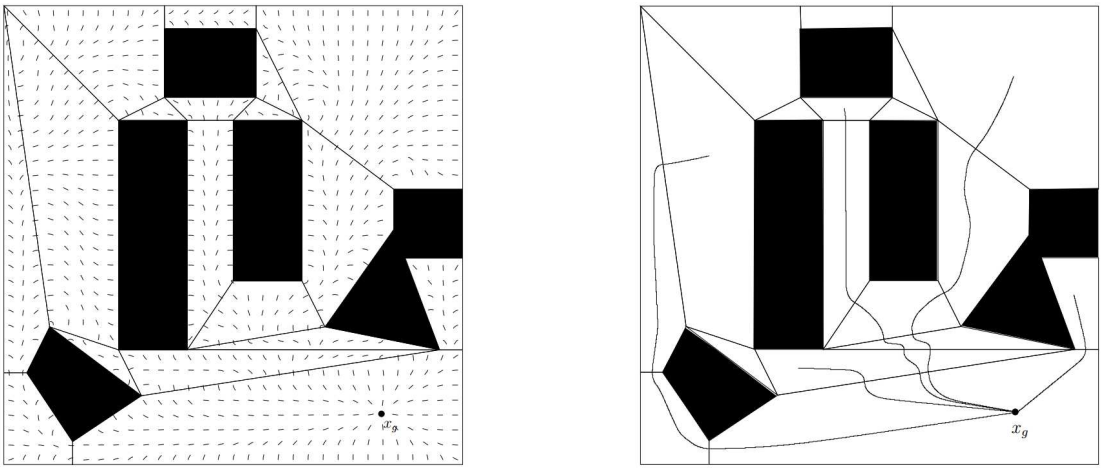


Figure 3.13: There is no deterministic gradient descent algorithm to solve the local minima problem

they use cell decomposition and make convex polygons in the free space $\mathcal{C}_{free}$ and build a vector field and define a control policy on each single cell and a switching strategy to smoothly switch between control policies of each cell. Choset et al have used the *Harmonic potential function* to create this vector field but Lavalle has proposed a very interesting approach where they creates the vector field directly without having to define a potential function based on the distance of edges and vertices to the current location of the robot.

3.5.2 Navigation functions

The major concern with potential functions is the existence of local minima. One of the methods that tries to construct a feedback motion planner over the continuous free space is called Navigation Functions which have been proposed by Rimon and Koditschek [32].



(a) A vector field is constructed directly without a potential function

(b) Trajectories from different initial conditions to the goal

Figure 3.14: Lavalle et. al. [25] proposed an algorithm to use cell decomposition and create convex cells and then construct the vector field directly on each cell

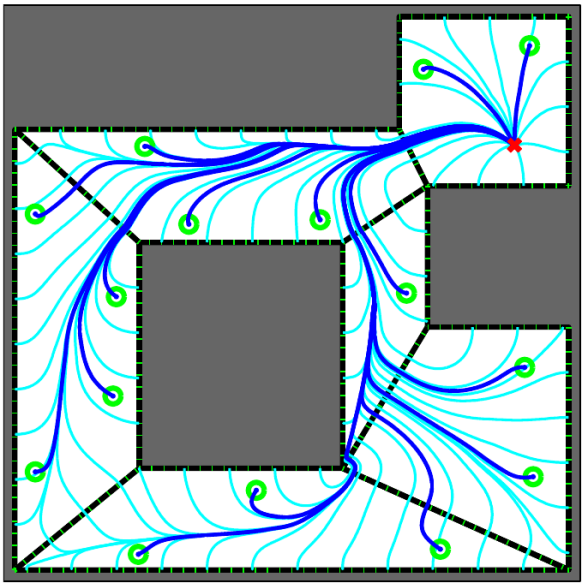


Figure 3.15: Choset et. al. [6] used the weak harmonic potential functions on decomposed cells

They have showed that it is not possible to construct a scalar field free from critical points. But they proposed a set of functions which construct a globally asymptotic scalar potential filed in which the critical points are unstable (i.e. the Hessian is non-singular and the critical point is non-degenerate). Thus, in implementation it is practically impossible for the point-mass robot to get trapped in such unstable critical points.

In the proposed method the motion planning algorithm is abstracted from the geometric space to a topological space, usually this topological space is called a "model space". The obstacle avoidance problem is then equivalent to staying in the same connected section of the free space $\mathcal{C}_{free}$, in which the point-mass robot has started. The model space could be considered any generalized sphere world.

For cases where the obstacle and the work space are not a sphere, a diffeomorphism is used and the geometric complicated obstacles are mapped into simple sphere in the model space. Then a navigation function is constructed on the model space, the motion planner generates a path and then the inverse of the diffeomorphism is used to transform the path from the model space to the real work space. In Subsection 3.5.2.1 the simpler sphere world is considered where there is no need to have a diffeomorphism because every thing is already a simple Euclidean sphere. Then in Subsection 3.5.2.2 a more general and complicated geometry of the work space is considered and it is described how to define the diffeomorphism and its inverse.

3.5.2.1 Navigation functions in a sphere world

A sphere world is defined as a compact, close and bounded, subset of n-dimensional euclidean space $\mathbb{E}^n$ whose boundary is a single $(n-1)-dimensional$ sphere. In this thesis however only the 2-dimensional euclidean space is considered. The space bounded by this 1-sphere called the *workspace* $\mathcal{W}$ and is defined as

$$\mathcal{W} = \{q \in \mathbb{E}^2 \mid \|q\|^2 \leqslant \rho_0^2\} \tag{3.21}$$

where $\rho_0 > 0$ is the radius of the outer sphere bounding the workspace. The center of the

bounding sphere is considered to be at the origin. If there are a total of M obstacles in the working space the number of all spheres would be $M+1$, where the extra 1 is the outer sphere. The remaining M other spheres which bound the obstacles present in the workspace are defined as

$$\mathcal{O}_j = \{q \in \mathbb{E}^2 \mid \|q - q_j\|^2 < \rho_j^2\}, \quad j = \{1,2,...,M\} \tag{3.22}$$

where q_j is the center of each spherical obstacle and its radius is $\rho_j > 0$. Thus the *configuration space obstacle* is defined as

$$\mathcal{C}_{obs} = \bigcup_{j=1}^{M} \mathcal{O}_j \tag{3.23}$$

the free space remains after removing all obstacles from the workspace

$$\mathcal{C}_{free} = \mathcal{W} \setminus \mathcal{C}_{obs} \tag{3.24}$$

Notice that according to the definition of *configuration space obstacle* the boundary of all obstacles are in the free space thus it would be a valid path if the point-mass robot goes on this boundary, which in reality represents scratching the surface of an obstacle. One could assume the workspace as a standard disk and the obstacles as spherical punctures in this disk. This idea is represented in Figure 3.16.

The formal definition [5] of a navigation function is as follows :

Definition 3.5.1.

If $\mathcal{C}_{free}$ is a compact analytical manifold with boundary then the map $\varphi : \mathcal{C}_{free} \to [0,1]$ is called a navigation function if it:

1. *is analytical on $\mathcal{C}_{free}$ (Infinitely differentiable, smooth, or at least C^k for $k \geqslant 2$)*
2. *is polar on $\mathcal{C}_{free}$ (A unique minimum exists at $q_g \in \mathcal{C}_{free}$)*
3. *is Morse on $\mathcal{C}_{free}$ (All critical points are non-degenerate)*
4. *is admissible on on $\mathcal{C}_{free}$ (Uniformly maximal on the boundary of the free space)*

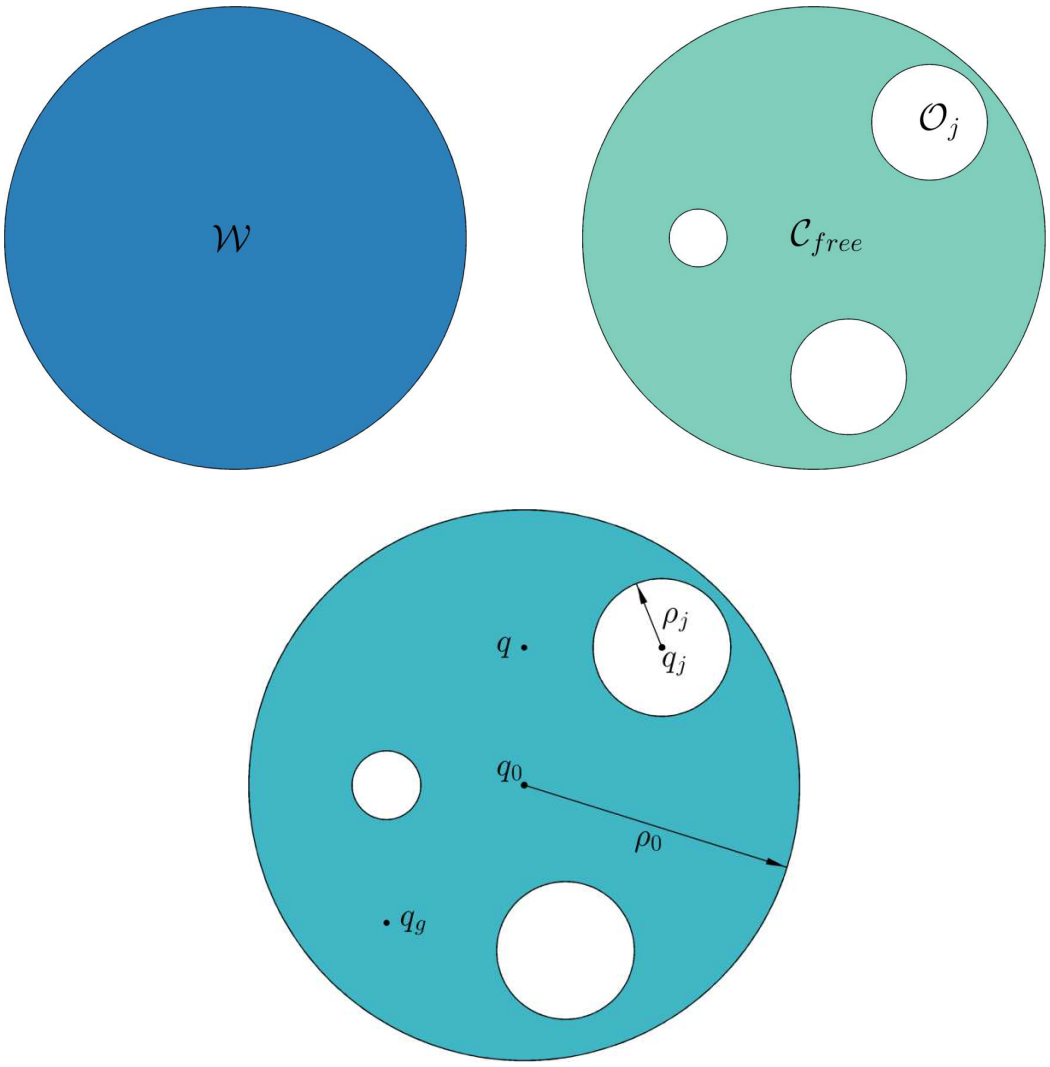


Figure 3.16: Different sets on a sphere world with respective dimensions

The potential function proposed by Koditschek-Rimon has all of the above mentioned properties [1].

The potential that acts as the attractive portion of the navigation function is the simple euclidean distance to the goal $\gamma : \mathcal{C}_{free} \rightarrow [0, \infty)$

[1] They have also proved that all critical points appear close to the boundary of the free space $\mathcal{C}_{free}$. They have shown these critical points would vanish from the free space if an annulus with a width ε is added around all obstacles. It is also proved that there is a formulation to find the minimum value for ε which satisfies this condition

$$\gamma(q) = \gamma_g^k(q), \quad k \in \mathbb{N} \setminus \{0,1\}; \quad \gamma_g(q) = \|q - q_g\|^2 \tag{3.25}$$

where γ is zero at the goal configuration and increases as q moves away from the goal. The repulsive portion is the product of obstacle functions present in the workspace $\beta : \mathcal{C}_{free} \rightarrow [0, \infty)$

$$\beta(q) = \prod_{j=0}^{M} \beta_j(q) \tag{3.26}$$

where β_js are defined as

$$\beta_0(q) = \rho_0^2 - \|q\|^2; \quad \beta_j(q) = \|q - q_j\| - \rho_j^2, \;\; j = 1, 2, ..., M \tag{3.27}$$

the definition of obstacle functions is direct result of the way the configuration space obstacle is defined. The outer sphere which constructs euclidean disk is considered a s the zeroth Obstacle. According to Equation (3.27) the obstacles hold a negative value inside the obstacle, zero on its boundary and positive in the free space $\mathcal{C}_{free}$. Also notice the same thing holds true for $\beta(q)$ as it is the product of the obstacles and not the summation of the obstacles, in contrast with the way repulsive potential was defined in Khatib's potential function.

Using the repulsive and attractive potentials the function $\hat{\varphi}(q) = \frac{\gamma(q)}{\beta(q)}$ is defined. As q approaches the boundary of any obstacle β goes to zero and $\hat{\varphi}$ goes to infinity thus repelling the robot. Also it is only zero at the goal configuration, where γ is zero. As [ref to Robot Navigation Functions on Manifolds with Boundary theorem 4] Koditschek-Rimon have proved there exists a positive integer N such that for every $k \geqslant N$, $\hat{\varphi}$ has a unique minimum at the goal configuration. It is very easy to see that in $\hat{\varphi}$, as k increases the numerator changes more significantly compared to the denominator and thus $\hat{\varphi}$ points toward the goal. It is also worth mentioning that the critical points also move closer to the boundary of the obstacles as k increases because the effect of repulsive function from the obstacles is reduced in further distances [19].

As q approaches the boundaries $\hat{\varphi}$ can have arbitrarily large values. So the diffeomorphism $\sigma : [0,\infty) \to [0,1]$ is introduced to bound $\hat{\varphi}$

$$\sigma = \frac{x}{1+x} \tag{3.28}$$

this diffeomorphism maps the range of $\hat{\varphi}$ to the unit interval. Using this diffeomorphism the values at the boundary of any obstacle is 1 and the goal has a value of zero. But with some ks it might have a degenerate critical point at the goal. So a distortion is introduced to eliminate the degeneracy $\sigma_d : [0,1] \to [0,1]$

$$\sigma_d(x) = x^{\frac{1}{k}}; \;\; k \in \mathbb{N} \tag{3.29}$$

the final function which poses all the conditions of a Navigation Function will be

$$\varphi = (\sigma_d \circ \sigma \circ \hat{\varphi})(q) = \frac{\gamma_g(q)}{(\gamma(q) + \beta(q))^{\frac{1}{k}}} \tag{3.30}$$

which is guaranteed to have a single unique minimum at q_g if k is sufficiently large.

Consider the workspace depicted in Figure 3.17 on the following page. We would like to construct a scalar potential filed on the free space of this workspace. Then use the gradient descent algorithm to find a path starting from any point in the free space toward the goal q_{goal}. As discussed earlier such a potential field can be constructed using Equation (3.30). Figure 3.18 shows how this scalar field develops as the parameter k is increased. Notice that the existence of local minima is apparent where k holds a smaller value but as k increases the scalar field changes and after a large enough k there is no local minima in the free work space.

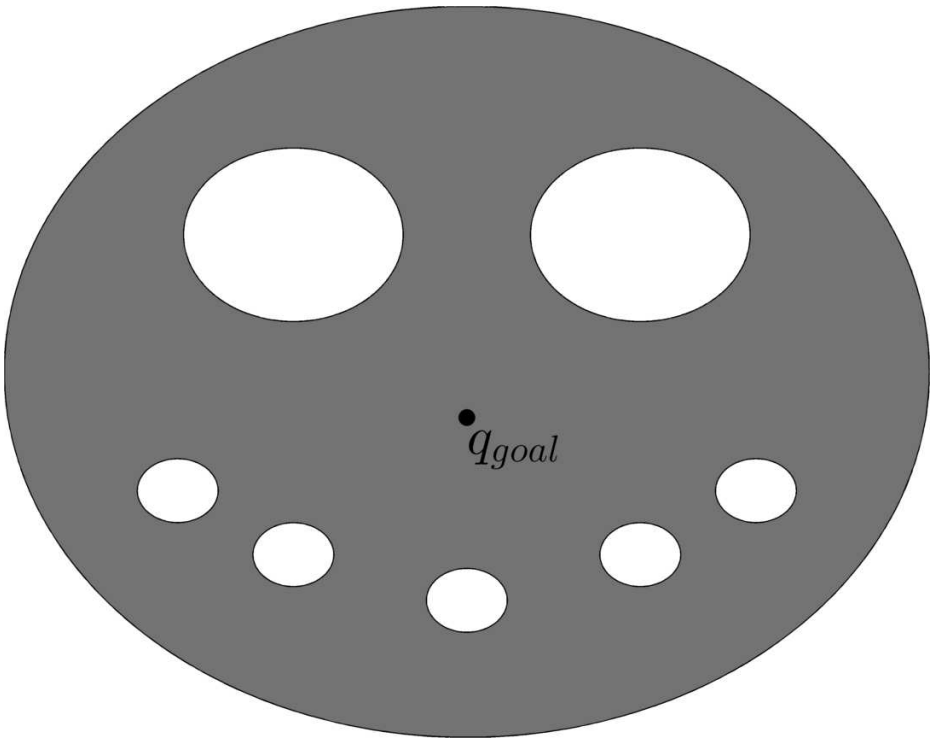


Figure 3.17: A *Sphere World* workspace

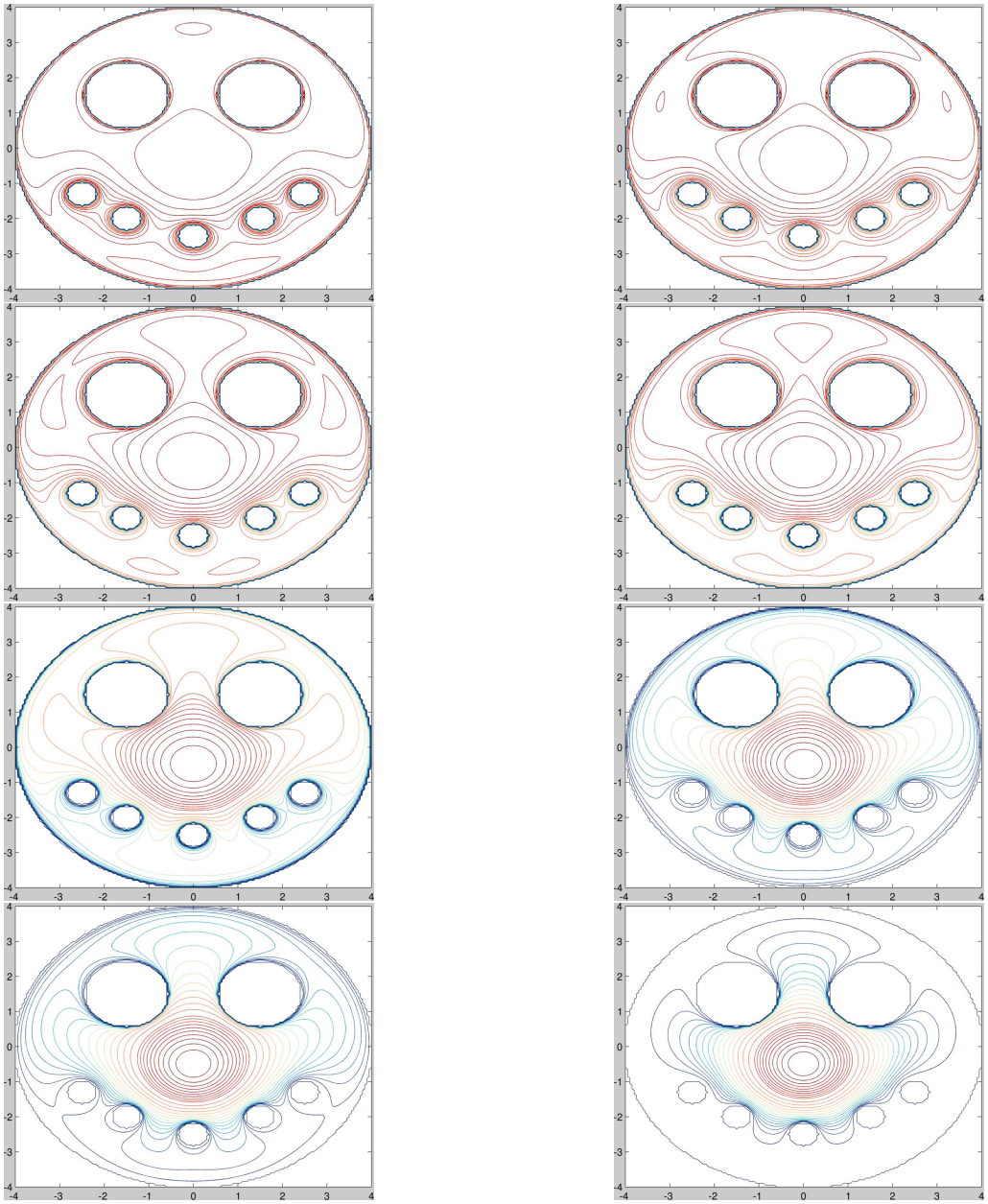


Figure 3.18: Change in the contour lines over free space as K increases

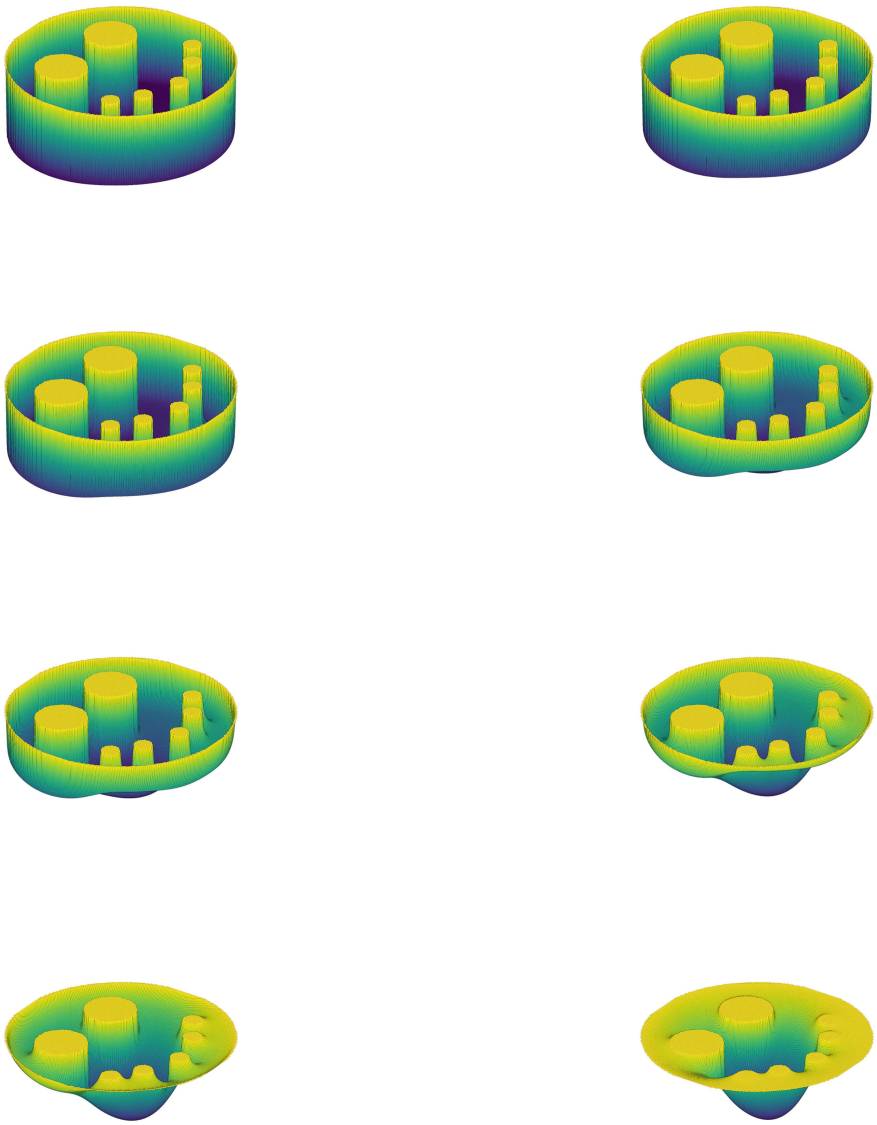


Figure 3.19: Change in the scalar field over free space as K increases

Figure 3.20 shows different paths with different with different initial positions. For this configuration the gain K is set to 7. As it can be seen all the paths are collision free. evolves starting from different positions towards the goal.

3.5.2.2 Navigation functions in a star world

In the previous section it was assumed that we are in a perfectly defined Euclidean Sphere World. This section tries to solve the path planning problem in a more general world, a *Star World*. Let's first mathematically define a Star shaped world. In set theory a star shaped set S is a set where there exists at least one point in the set that is within line of sight of **all** other points of the same set: The construction of analytic diffeomorphisms for exact robot navigation on star worlds [5]:

$$\exists x \text{ such that } \forall y \in S,\ \ tx + (1-t)y \in S\ \ \forall t \in [0,1] \tag{3.31}$$

In the same spirit an obstacle $\mathcal{O}_j$ is considered Star Shaped if there is a point $q_j \in \mathcal{O}_j$ such that for all $q \in \mathcal{O}_j$ the inward gradient $\nabla\beta_j(q)$ satisfies

$$\nabla\beta_j(q).(q - q_j > 0 \tag{3.32}$$

if all obstacles in the free space, $\mathcal{S}_{free}$, are star shaped then, $\mathcal{S}_{free}$ is called a *Star World*.

In the previous section it was shown how to find a navigation function for Sphere Worlds. With a Star World we should first map the Star World to a Sphere World, solve for the navigation function and then use a diffeomorphism to map the navigation function back to the Star World.

Thinking about the definition of a *Star World* you might realize how close they are to a *Sphere World*. A *Sphere World* could be thought of as a homeomorphism of a *Star World* and ice versa [5]. The construction of analytic diffeomorphism for exact robot navigation on star worlds Elon Rimon, Daniel E. Koditschek have shown that given a navigation function in the free configuration space $\mathcal{P}$ of a Sphere World $\varphi : \mathcal{P}_{free} \to [0,1]$ there always exists a mapping which is a diffeomorphism [2] from a Star World to the Sphere World $h : \mathcal{P}_{free} \to \mathcal{S}_{free}$.

Rimon and Koditschek have shown that the construction of analytic diffeomorphisms for

[2] A diffeomorphism is a map between two smooth manifolds. It is an invertible, and thus bijective, mapping. Both the diffeomorphism and its inverse are smooth

exact robot navigation on star worlds could be achieved in two steps. In the first step the the start world is mapped to a homeomorphic sphere world, then the navigation functions are constructed under the sphere World and then they could be *pulled back* into a Star World using this diffeomorphism.

So if $\varphi : \mathcal{P}_{free} \rightarrow [0,1]$is a navigation function on $\mathcal{P}$ and $h : \mathcal{P}_{free} \rightarrow \mathcal{S}_{free}$ is the analytical diffeomorphism then

$$\phi : \varphi \circ h. \tag{3.33}$$

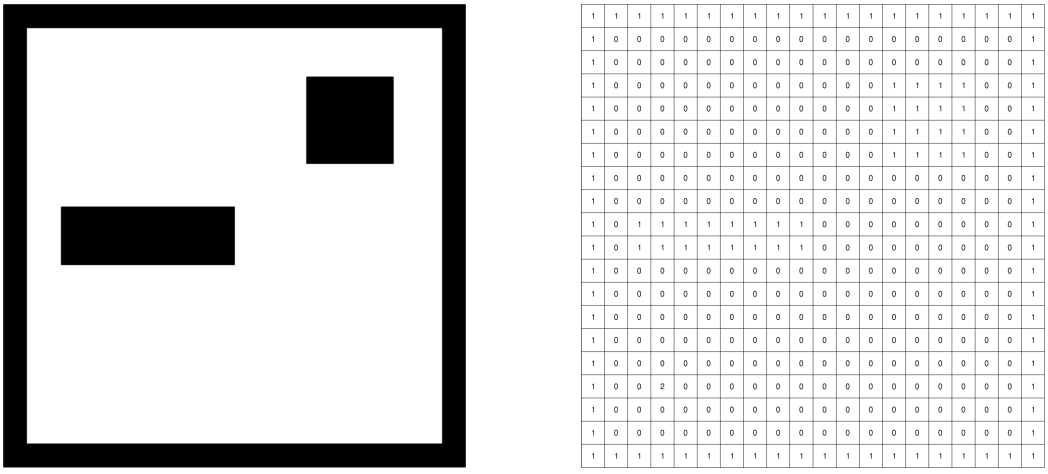
is a navigation function on the *Star World* $\mathcal{S}$. The bijective property of the diffeomorphism guarantees that there is a one-to-one relation between the critical points and obstacles.

3.5.3 Discrete motion planning

As discussed in the previous sections, although the potential function idea is very elegant and simple but it has one major draw back which is the existence of local minima, let alone the oscillations which generally could be solved using the right selection of α_i. There is no simple deterministic algorithm to solve the local minima problem other than the Navigation functions. They provide very beautiful and elegant solution for this problem but at the same time they add a lot of complexity to the the problem. Even a simple work space needs a lot of work to implement the navigation function. But there is a special type of Navigation Functions on spaces which are represented as grids called WaveFront Planner. These family of navigation functions are probably the simplest solution to the local minima problem and are used in a lot of different motion planners and also a lot of video games as the algorithm is very easy to implement and work with. The input to the WaveFront planner is an occupancy grid [3]. Occupancy grids were first popularized by Hans Moravec and Alberto Elfes at CMU. As our initial assumption about the workspace he occupancy grid used in this thesis is a binary map, 0 or 1. Because we have assumed we have perfect mapping information about

[3] An occupancy grid is a discrete probabilistic method to represent a work space. Each cell holds a probability value that shows the certainty if that cell is occupied or not. 1 shows a 100% certainty of an obstacle on that cell and 0 shows a 100% certainty that it is a cell free of obstacles

the workspace. Figure 3.20 shows a Work Space and its corresponding occupancy grid.



(a) A simple work space

(b) Occupancy grid

Figure 3.20: (a) Shows a simple work space with 2 rectangular obstacles and (b) Shows the occupancy grid representation of the same work space. Cells that lie on an obstacle have a value of 1 and the free cells have a value of zero

The goal cell has a value of 2. The wave Front planner starts from the goal, then finds all zero valued cells around the goal and change their value to 3. In the next step all zero valued cells adjacent to cells with a value of 3 are update to have a value of 4. This process continues until the whole gird is covered. The value of a cell holds could be interpreted as a cost function, the cost it takes to move from the goal cell to the current cell. The wave front algorithm is described below

Algorithm 3 Wave Front Algorithm

Label the goal cell in the occupancy grid with a 2

i = 2

Find all zero valued cells neighboring the cell with value i

Update the label for all the zero valued cells to i+1

i = i+1

Go to step 3

The only question that remains unanswered is how the neighboring cells are found. Consider Figure 3.21 where two types of possible neighborhoods are shown. If the dynamics of the robot allows movements in diagonal direction usually the 8-neighborhood method is chosen and the cell *M* has 8 children in the graph induced by Moore connectivity. If the movement of the robot is limited the 4-neighborhood method is chosen and the cell *M* has 4 children in the graph induced by the Von Neumann connectivity.

(a) Von Neumann connectivity inducing 4-neighborhood for the central cell

(b) Moore connectivity inducing 8-neighborhood for the central cell

Figure 3.21: Comparing a 4-neighborhood and an 8-neighborhood connectivity

Figure 3.22 on the following page shows how the wave front planner grows on the work space shown in Figure 3.20 if the 8-neighborhood connectivity is chosen.
As it might not be clear how the wave is propagating through the workspace Figure 3.23 shows the same wave propagation where the cells having similar colors also have the same cell value.

Once the wave front has expanded all the cells in the grid, a simple gradient descent algorithm can be implemented to find a path from any given start point towards the goal. That is a very positive point about Wave-Front algorithm. For a given work space and static obstacles the algorithm runs only once but it will work for any start configuration in the free work space. That's why a lot of researchers categorize this algorithm as a Feed Back motion planning algorithm because a new control input is provided based on the last position of the system. The accuracy of this algorithm depends on how small each cell is. On the other hand the smaller the size of a cell, the longer it would take for the algorithm

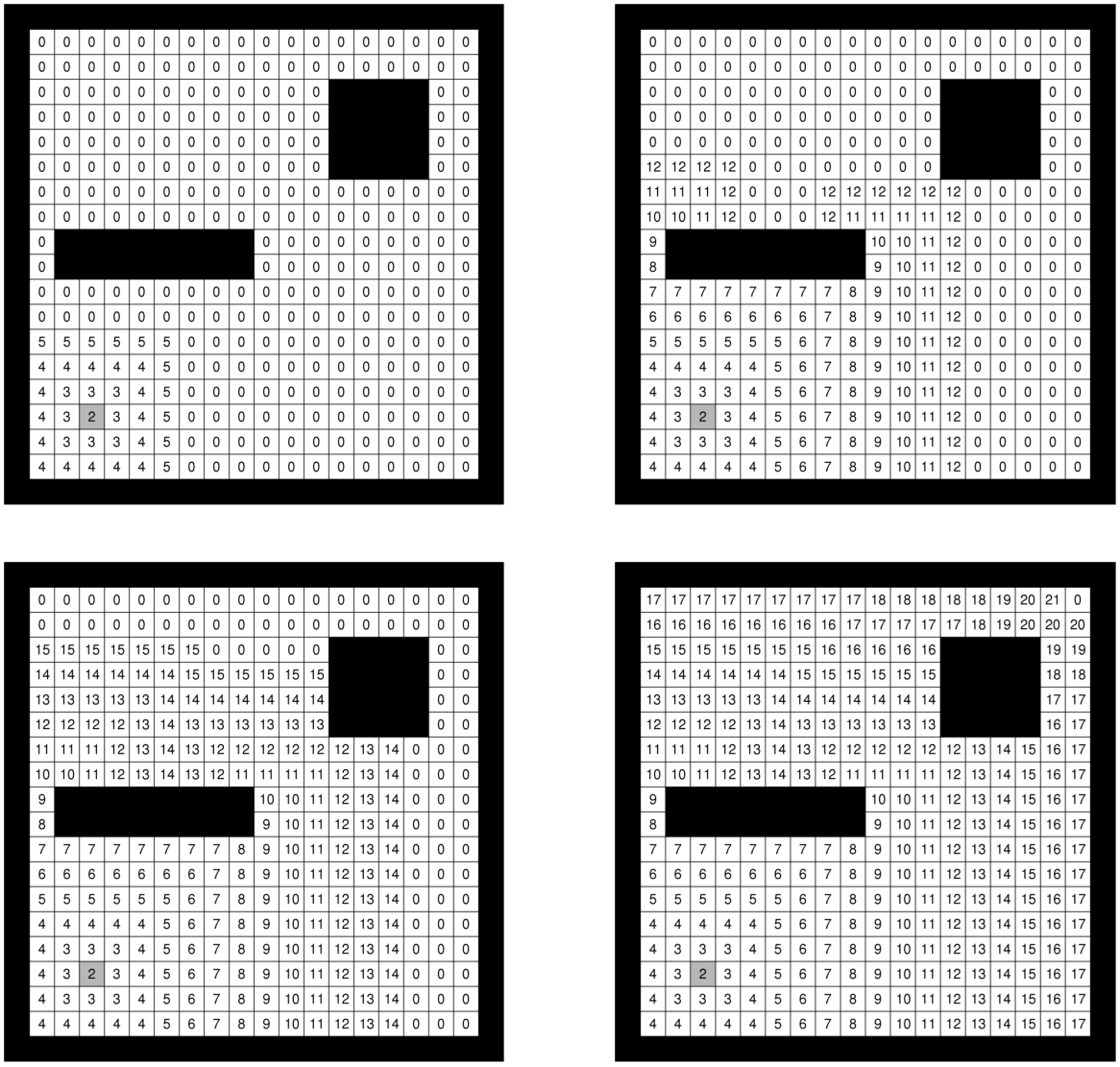


Figure 3.22: Wave front planner growing inside a workspace

to go through the whole grid. Figure 3.24 shows the final path generated using the gradient descent algorithm starting from an arbitrary configuration.

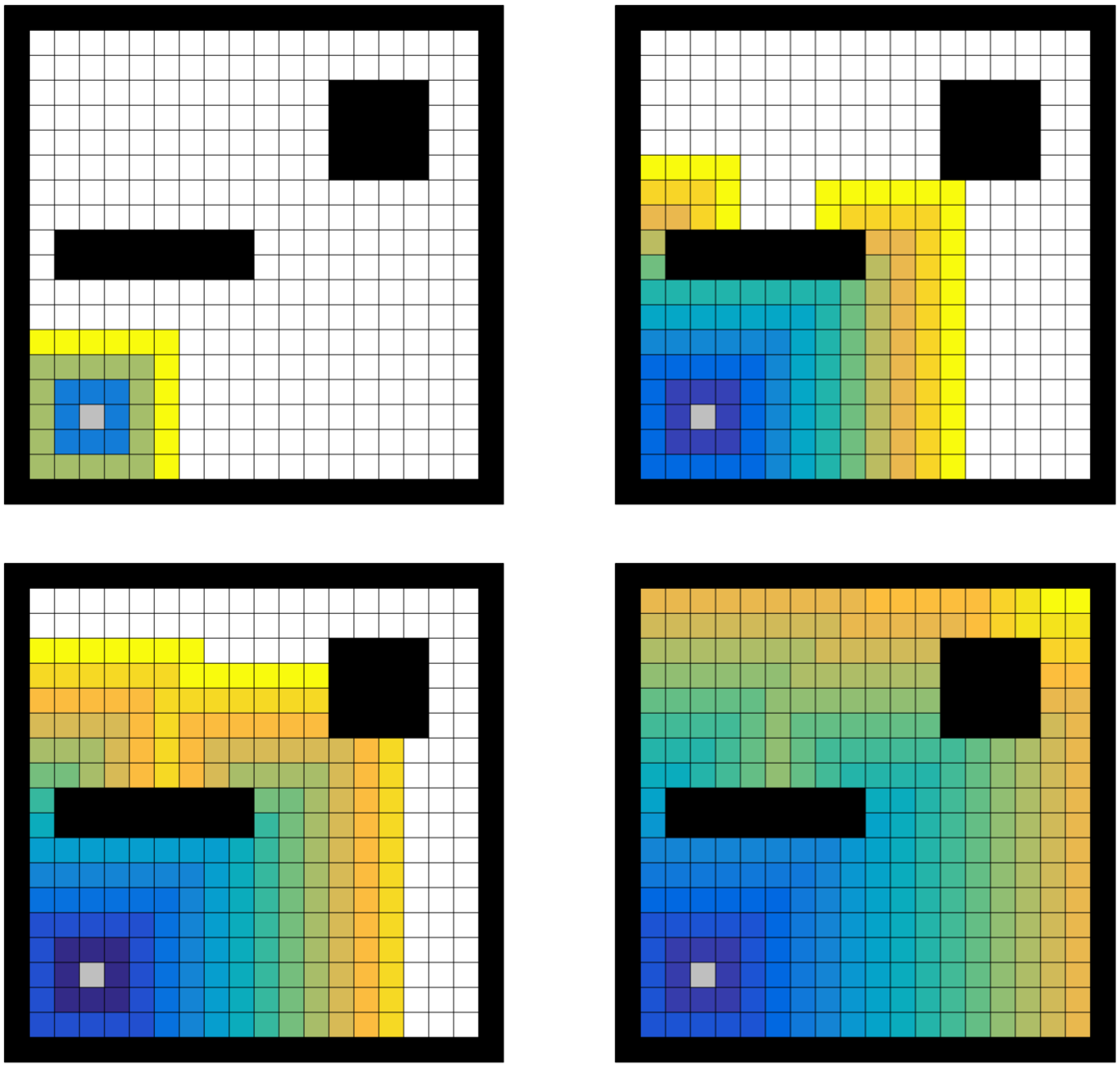
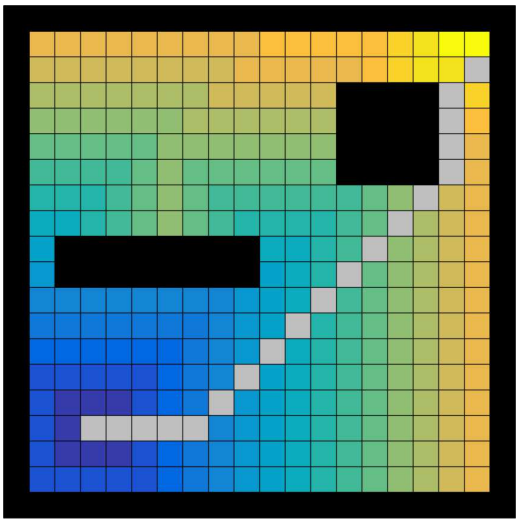
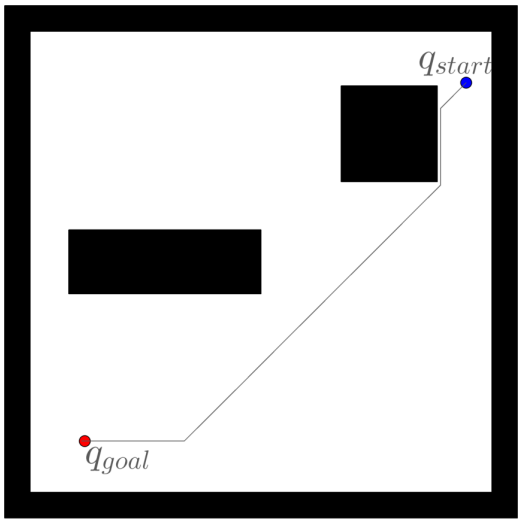


Figure 3.23: Wave front planner growing inside a workspace different colors represent different cost of a cell



(a) A path found using gradient descent on the wave front grid



(b) The same path on the work space created by connecting gray cells on the grid

Figure 3.24: A path found using gradient descent is shown on the grid, and the same path is shown on the work space

Chapter 4

Heuristic-Based Path Planning

In this chapter the basic terminology and mathematical background necessary to understand graphs is covered. This first part will give enough intuition to introduce search algorithms and then discuss and compare their differences.

4.1 Graph basics

In heuristic-based Path Planning the problem is usually represented as a graph and then a graph-searching algorithms is used to solve the path planning problem. Such search algorithms have been used extensively to solve different engineering problems such as routing of telephone traffic, navigation through mazes, layout of printed circuit boards, etc. In general there are 2 main different approaches to solve a graph-search problem, *mathematical approach* and *heuristic approach*. The mathematical approach usually considers the abstract properties of a graph rather than the computational feasibility of the solution while *heuristic approaches* usually use a special knowledge about the domain of the problem to improve the efficiency of the solution.

A graph, in the most common sense, is an ordered pair $G = (V, E)$ such that V is a set of vertices $\{v_i\}$, or nodes, and E is a set of edges $\{e_{ij}\}$, or arcs. In such a graph each edge is related with two distinct vertices. If e_{mn} is an element form the set of edges then there exist an arc from node v_m to v_n and the node v_n is a *successor* of v_m. There is usually a label

associated with each edge. Depending on the application the label might have different names, such as *cost*, *length* or *capacity*.

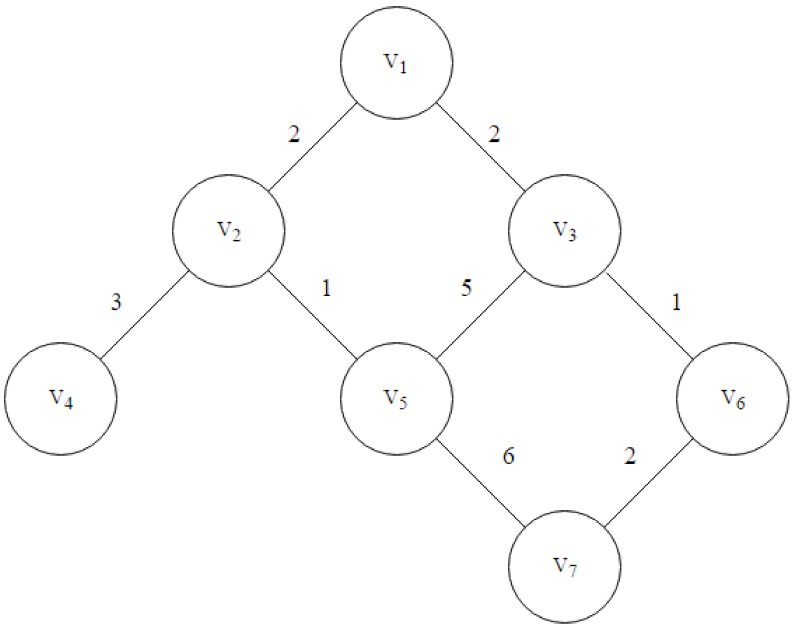


Figure 4.1: A graph with 7 nodes and 8 edges. The label represents the cost of moving from node V_i to V_j.

The nodes that share an edge are called adjacent. All nodes adjacent to a node are called it's neighborhood [1] . The graphs can have two different types depending on the type of the edge

- In an **undirected** graph movement on an edge is bidirectional. i.e. If exist a path from v_i to v_j, the same path can be used to go from v_j to v_i.
- In a **directed** graph an edge from v_i to v_j can not be used to go from v_j to v_i. [2]

In the graph search domain it becomes much easier to think of a graph as a space that goes through different states. Let's assume all search algorithms have a search agent

[1] This is also sometime referred to as the *Branching Factor* of a node. The *Branching Factor* in a graph is the number of edges going out of that node.

[2] Such maps are usually used in games where the character enters a room through a door but it is not possible to get out of the room from the same door!

that starts from a specific node, *Initial State* x_s and then traverses the graph by checking different vertices, nodes, and finally ends up inside the *Goal Set* X_G. At each vertex there are just a few actions that the search agent can take to move on to the next vertex. Lavalle [22] has beautifully defined the state space, X, of a graph as:

- A nonempty *state space* X, which is a finite or countably infinite set of *states*.
- For each state $x \in X$, a finite action space $U(x)$.
- A state transition function f that produces a state $f(x,u) \in X$ for every $x \in X$ and $u \in U(x)$. The state transition equation is derived from f as $x' = f(x,u)$.
- An initial state $x_{Initial} \in X$.
- A goal set $X_{Goal} \subset X$.

Let's consider the following gird as a search problem. We can simply represent a grid as a graph.

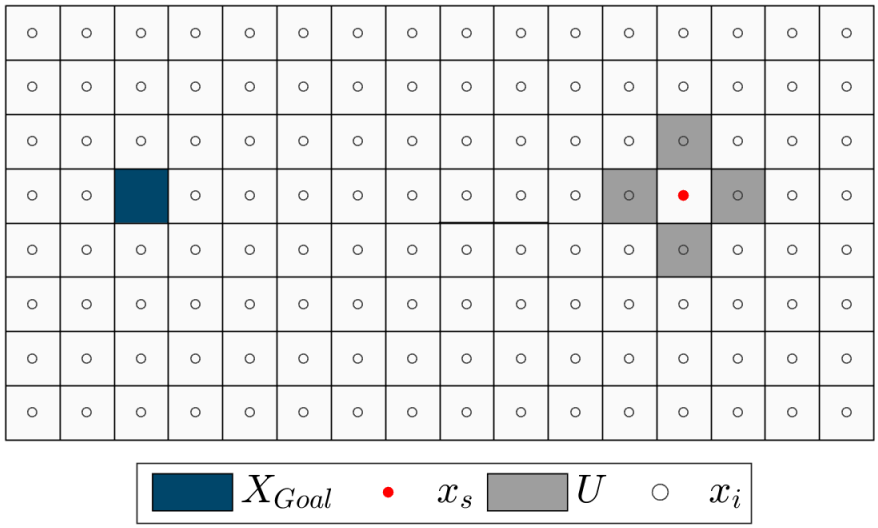


Figure 4.2: A grid represented as a graph

As shown, the center of each cell on the grid can represent a node of a graph. The search problem is then how to find a path from x_s to the goal set, X_{Goal}. $\forall x_i \in X$ we have $U(x) =$

$\{(0,1),(1,0),(0,1),(1,0)\}$ as the action space and the transition equation is $f(x,u) = x+u$ where $x \in X$ and $u \in U$. In this graph it is assumed that the search agent cannot traverse diagonally and each vertex has a degree [3] of 4. If we were to allow the diagonal movements the action space would be $U(x) = \{(0,1),(1,1),(1,0),(1,1),(0,-1),$
$(1,-1),(-1,0),(-1,1)\}$.

In the discrete search algorithms that are covered in this chapter there is a repeating scenario. At each step of the search, each vertex can be in one of these three different states: *unexplored*, *explored* or inside the *frontier* list [4]. The search algorithm starts the search from a specific node, *Initial* or *start state* x_s. Then the search agent starts the search by applying the transition function. When the transition function is applied on a node that node is marked as *explored* and it is added to the *Closed Set C*, specifying that this node is already explored by the search agent. The nodes that are adjacent to the explored nodes are then added to the *Open Set O*. The nodes in the open set are also called the frontier and they specify the nodes that have the potential to be selected as the next node to be explored by the search agent. The order in which these frontier nodes are explored depends on the search algorithm. The search will continue as long as there are still nodes in an *unexplored* state.

The frontier is an ordered set of nodes that creates a data structure. The main difference between search algorithms is how new elements enter this data structure and how old elements leave it. There are 3 main different data structures:

- Stack: A stack is a **LIFO**, last in first out, data structure. When the search agent is exploring new nodes, it will pick the most recent node from the data structure and apply the transition function on it.

- Queue: A queue is a **FIFO**, first in first out, data structure. When the search agent is exploring new nodes, it will pick the oldest node from the data structure and apply the transition function on it.

[3] The degree of a vertex is equal to the number of adjacent vertices.
[4] LaValle [22] refers to these states as *Unvisited*, *Dead* and *Alive*

- Priority Queue: Each node is given a priority based on some type of criteria and then it is added to the list. When the search agent is exploring new nodes, it will pick a node that has the minimum priority from the list. Unlike a stack and a queue the order at which the nodes are added to the priority queue does not affect when they are picked out of the queue.

Considering any given node in the graph G and a starting node x_s, there are different algorithms that can be used to find the goal set. In this chapter we will cover the following algorithms:

- Breadth First Search
- Depth First Search
- Dijkstra's Algorithm
- A* Search

Considering these search algorithms as a Path Planner we expect to get a path from node x_s to X_G as the output. This path should be a set of nodes $v_0, v_1, ..., v_k$ where each node $v_i + 1$ is a *successor* of node v_i and $v_k \in X_{Goal}$. Usually in path planning problems we are interested in a more particular path, a path that is shorter or more cost effective than all other feasible paths. Such a path is called an *optimal path*. The cost for an edge could be integrated in a graph by giving each edge a specific weight based on a metric, i.e. euclidean distance between two nodes. The cost of an edge between any two nodes v_i and v_j is represented by $h(v_i, v_j)$. There are three criteria that should be considered when we are comparing the search algorithms; *Completeness*, *Time Complexity* and *Space Complexity*.

- **Completeness:** An algorithm is complete when, if there exists a solution, it guarantees to find a solution within a finite amount of time.
- **Time Complexity**: The time complexity of an algorithm is the worst-case amount of time that it takes to run the algorithm. We express the time complexity of each

algorithm in terms of the maximum branching factor (b) and the maximum length of the path (m).

- **Space Complexity**: The space complexity of an algorithm is the worst-case amount of space that it takes to run the algorithm. We also express the space complexity of each algorithm in terms of the maximum branching factor (b) and the maximum length of the path (m).

4.2 Breadth first search

The Breadth First Search (BFS) algorithm was introduced by Lee [24] in 1971. Lee has compared the algorithm to *A computer model of waves expanding from a source under a form of straight-line geometry. Those cells having the same cell mass may be thought of as the location of the wave front at the nth unit of time.*. This description is very similar to the Wave front planner algorithm which was discussed in section 3.5.3. They are actually the same thing but Wave Front planner is a stripped down version of BFS. The main difference is that in Wave front planner the algorithm goes through the whole grid and gives a value to each cell and then a gradient descent algorithm finds the final path. But BFS is searching for the goal point and once it reaches the goal the search stops. Then another method is used to back propagate through the graph and find the path.

BFS starts at the initial start vertex x_s in the graph and explores the neighbor vertices first before moving to the next level vertices. A **Queue** is used as the data structure to store the nodes and the frontier is a list of vertices $[v_0, v_1, v_2, ..., v_n]$, where $v_0 = x_s$. The search agent always selects the earliest element that was added to the frontier.

In the first step, v_0 is selected as the starting point and tested for being a goal. If v_0 was not a goal. The state transition function is then applied to this node and the output nodes of the transition function are added to the end the frontier list. The search agent would then iteratively explore all nodes until all nodes are explored or the agent reaches a node inside the X_{Goal}.

BFS is guaranteed to find the path that involves the fewest arcs and it is a complete algorithm if the branching factor for all nodes was finite. The time complexity is $O(b^m)$ because every node in the tree has to be examined. BFS's Space complexity is $O(b^m)$ because the whole frontier has to be stored in the memory. This should not be surprising. As it can be seen in Algorithm 4 BFS does not check for the cost of an edge during the search, i.e. BFS assumes the input graph is unweighted.unweighted graphs

Algorithm 4 Breadth First Search Algorithm

```
O = Empty Queue
C = {}
O:push(x_s)
while True do
    x_i = O:pop();
    if x_i ∈ X_G then
        return
    end if
    C:add(x_i);
    for all u ∈ U(x_i) do
        x_succ = f(x_i, u)
        if x_succ ∉ C then
            if x_succ ∉ O then
                O:push(x_succ)
            end if
        end if
    end for
end while
```

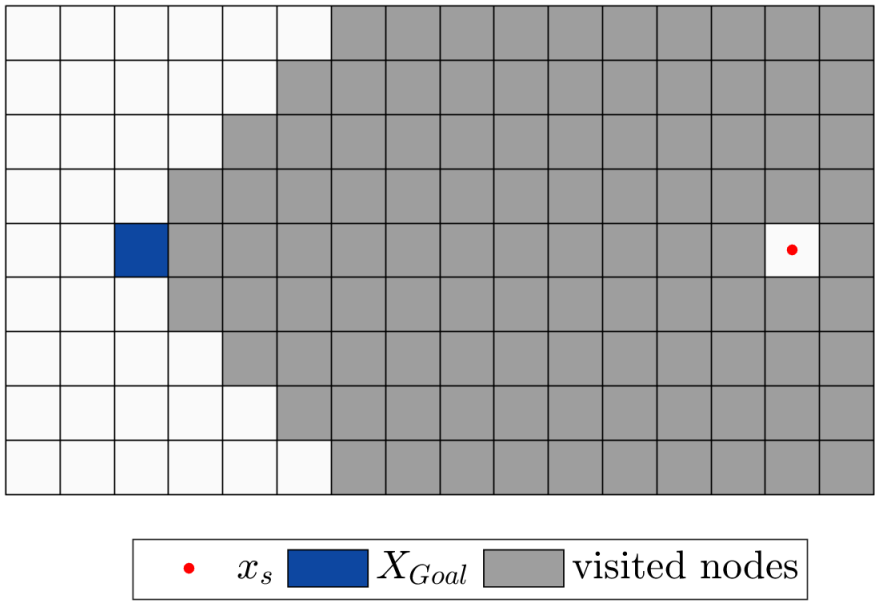


Figure 4.3: BFS state space

4.3 Depth first search

Depth first search starts at a given vertex x_s in the graph and explores along each branch until it reaches a dead end. The frontier is a stack of vertices; $[v_0, v_1, v_2, ..., v_r]$ and $\{n_1, n_2, ..., n_k\}$ is v_s's neighbors. In the first step, v_s is selected as the starting point and tested for being a goal. If v_s was not a goal, all its neighbors are added to the frontier stack, so the frontier will be $[n_1, n_2, ..., n_k, v_1, v_2, ..., v_r]$. v_1 is only selected when all the paths of v_s's neighbors are explored. Exploration is done iteratively until it reaches the goal, v_g or every vertex has been visited.

If the graph is acyclic and finite, DFS can be considered as a complete algorithm and guarantees to find a solution if one exists. DFS has the time complexity of $O(b^m)$ and its space complexity is $O(bm)$. Where b is the maximum branching factor and m is the maximum length of the path.

Algorithm 5 Depth First Search Algorithm

O = Empty Stack
$C = \{\}$
O:push(x_s)
while True **do**
    x_i = O:pop();
    if $x_i \in X_G$ **then**
        return
    end if
    C:add(x_i);
    for all $u \in U(x_i)$ **do**
        $x_{succ} = f(x_i, u)$
        if $x_{succ} \notin C$ **then**
            if $x_{succ} \notin O$ **then**
                O:push(x_{succ})
            end if
        end if
    end for
end while

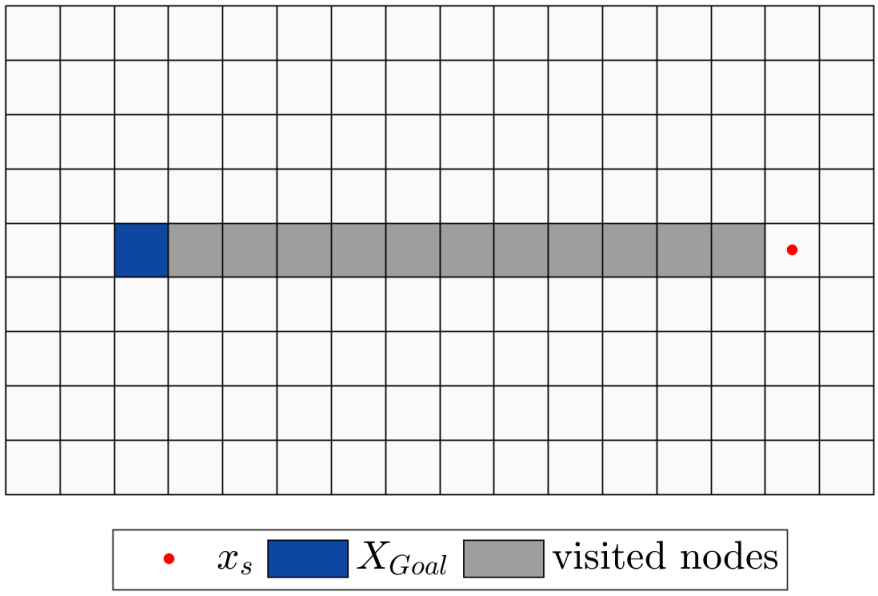


Figure 4.4: DFS state space

4.4 Dijkstra search

A very natural question that might pop up on any one's mind is that, If we already have some type of information about the goal can we make the search faster, *Can we check for a smaller number of cells?* A* is the answer to this question.

Consider a graph G and a starting node x_s and the goal set X_{Goal}. In BFS or Wave front planner all neighboring nodes are blindly visited. But what if we could just visit a more promising node in the neighborhood instead of all nodes? We must have a criteria or a metric to choose the most promising nodes from the neighboring nodes. A very common and logical metric could be the distance between any of the neighboring nodes and the goal node. It might seem that if we always choose the closest node to the goal there is a higher chance of being on the shortest path towards the goal. Following the closest cell to the goal does not always necessarily results the shortest path. This algorithm is called greedy search. Greedy search does not take into account how many steps it has moved to reach the goal. It just cares about the best immediate action it can take, no matter what might be the result of this action.

Now, let's take a look at what would happen instead of caring about the distance to goal, we try to move in such a way that we always maintain the minimum distance to the starting node. At each step the neighboring nodes are pushed into a data structure. Then we check which node in the data structure is still unvisited and has the shortest distance to the start node. As it can be seen in figure ??? this algorithm results in checking a large number of nodes, but it is guaranteed to generate the shortest possible path, because we always go to a node that is the closest to the start point. This algorithm is actually very famous and it's called Disktra's algorithm, named after famous Danish Computer Scientist Edsger W. Dijkstra.

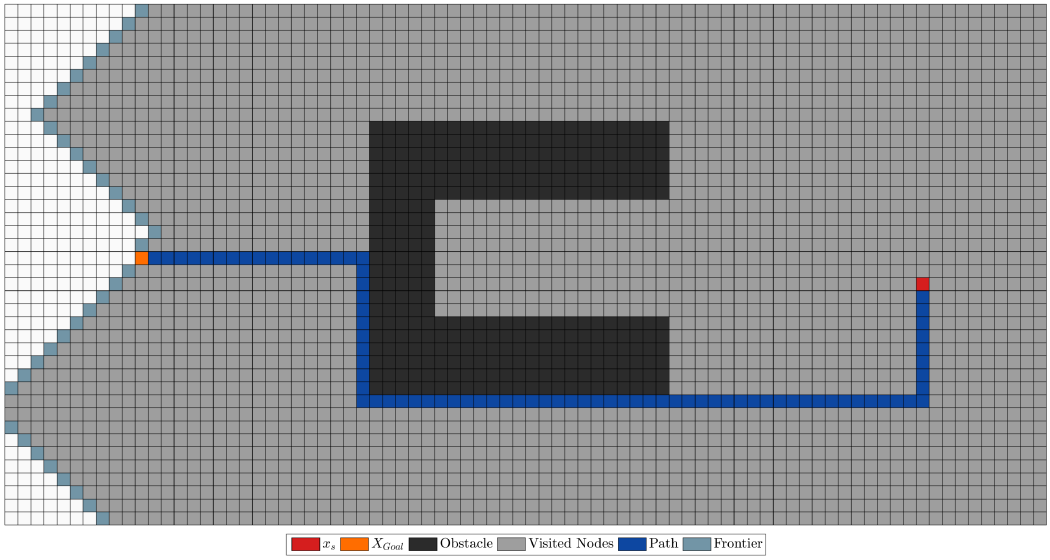


Figure 4.5: Dijkstra state space

4.5 A* search

What if we could take into account the number of steps the search agent has taken so far and also the the distance it still has to cover to reach the goal. That's exactly what A* [5]

[5] **A note on the A* name:** In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of Dijkstra's algorithm. This algorithm was called A1. In 1967 Bertram Raphael made dramatic improvements upon this algorithm, but failed to show optimality. He called this algorithm A2. Then in 1968 Peter E. Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only

does. A* is at the sweet spot between Dijkstra and Greedy search. If we think of Dijkstra as an improvement over BFS, A* is an improvement over Dijkstra.

A* search is a deterministic [6] heuristic based search that was introduced in 1968 [15]. A* is considered to be optimal if the following three conditions apply:

- costs are positive,
- Branching factor, b is finite,
- $h(x_i)$ is non-negative and is an *underestimate* of the shortest path from x_i to the goal set X_{Goal}.

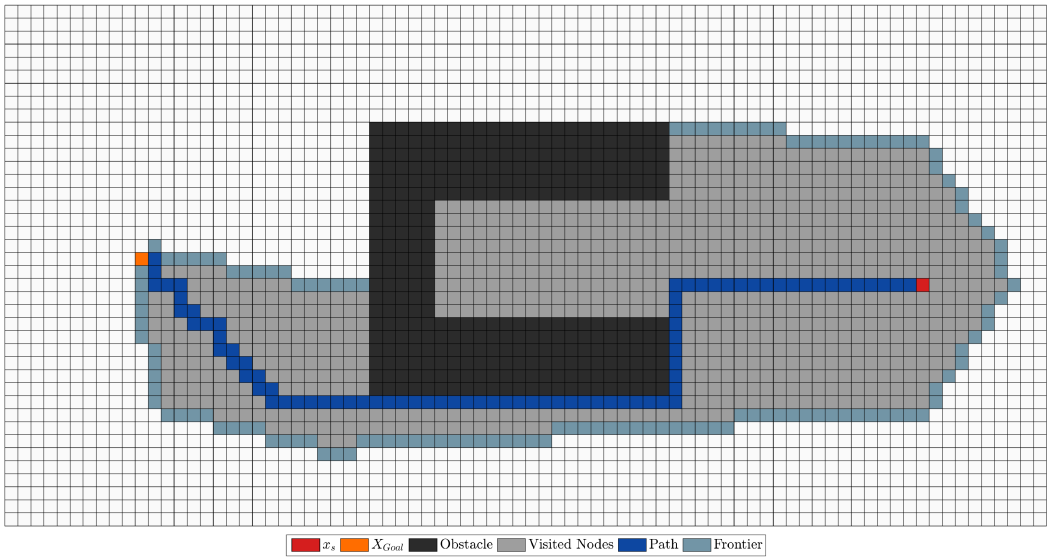


Figure 4.6: A* state space

Figure 4.6 shows the same work space as shown in the previous two search algorithms. As it can be seen $A*$ visits a lot less number of nodes but also like other algorithms it finds the shortest path. In the coming sections we will discuss how and why $A*$ explores fewer nodes compared to other algorithms.

minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions. He thus named the new algorithm in Kleene star syntax to be the algorithm that starts with A and includes all possible version numbers or A*

[6] The other technique is a non-deterministic randomized algorithm developed by Lavalle et al and Kavraki et al [17] [23] [22]

Algorithm 6 A* Search

Require: $x_s \cap X_{Goal} \in X$

1: $O = PriorityQueue()$
2: $C = \emptyset$
3: $O.push(x_s)$
4: **while** $O \neq \emptyset$ **do**
5: $x_i \leftarrow$ O.pop()
6: $C.push(x_i, f(x_i))$
7: **if** $x_i \in X_{Goal}$ **then**
8: **return**
9: **else**
10: **for** $u \in U(x_i)$ **do**
11: $x_{succ} \leftarrow f(x_i, u)$
12: **if** $x_{succ} \notin C$ **then**
13: $g_{tentative} \leftarrow g(x_i) + cost(x_i, x_{succ})$
14: **if** $x_{succ} \notin O$ **or** $g < g(x_{succ})$ **then**
15: $g(x_{succ}) \leftarrow g_{tentative}$
16: $h(x_{succ}) \leftarrow FindHeuristic(x_{succ}, x_{Goal})$
17: $f(x_{succ}) \leftarrow g(x_{succ}) + h(x_{succ})$
18: $O.push(x_{succ}, f(x_{succ}))$
19: **end if**
20: **end if**
21: **end for**
22: **end if**
23: **end while**

4.5.1 Heuristics

Heuristics are a set of strategies that are used to solve a problem faster or find a more optimal solution. The data structure that is used is a priority queue. Because it can be used to associate a node with the heuristic of that node. One can think of the heuristic as an estimate that helps the search agent to decide which node will result in a more optimal path and fewer explored/visited nodes during its exploration. A* uses a type of distance to goal and traveled distance from the starting node as a strategy to explore fewer nodes, this is the heuristic that is used in A*. At each node x_i the search agent looks up in the data structure and picks the node that has the lowest heuristic value. The heuristic is calculated as

$$f(x_i) = g(x_i) + h(x_i), \tag{4.1}$$

where $g(x_i)$ is the cost from the starting node, x_s, to the current node x_i. It is very straight forward to calculate $g(x_i)$ for any x_i.A search heuristic $h(x_i)$ is an estimate of the cost of the shortest path from node x_i to the goal set, X_{Goal}. This is the tricky part, because if we already knew the length of the shortest path we did not have to search the graph. This is where we have to do our best. As it will become clear choosing a good heuristic function is very crucial however most implementations use an approximation that is computationally non-intensive. It is a very common practice ti play with the heuristic function to get a more optimal solution depending on the current problem. Depending on the value of $h(x_i)$ the behavior of the search drastically changes:

- $h(x_i) = 0$, will result in a search that only $g(x_i)$ plays a role and practically $A*$ is converted to Dijkstra.
- $h(x_i) < RealExactCosttogoal$, then it is guaranteed that $A*$ will find the optimal path. The smaller the $h(x_i)$ is compared to the real cost, the more nodes are expanded by the search agent which essentially makes the search slower.
- $h(x_i) = RealExactCosttogoal$ will result in the best and most optimal path. No extra node will be explored by the search agent.

- $h(x_i) > RealExactCosttogoal$ will break the guarantee that the final found path will be optimal. This heuristic might be useful in cases where the optimality of the path is not important and we are more interested in finding any path and also considering a fast search speed. If the difference is too large A* acts like the greedy search.

This is what makes $A*$ so popular and powerful. By just playing with the heuristic and exploiting the trade off between speed and accuracy we can get totally different results based on the assumptions and requirements of the application.

A very crucial point that is usually overlooked is the scale of $g(x_i)$ and $h(x_i)$ functions. Extra care should be taken when defining these two functions and it should be checked that the units of the two functions match. As an example if $g(x_i)$ gives the number of nodes between x_i and x_s while $h(x_i)$ gives the euclidean distance between x_i and goal set, adding the two functions will create a number that doesn't really represent the heuristic and $A*$ will not perform as expected.

As mentioned there is no way to tell the shortest distance so we have to use some kind of estimation. The most popular estimations are [7]:

- Manhattan Distance. This is the standard heuristic used for square grids. The reason is obvious in Figure 4.7. The search agent has only expanded the nodes on the path and no extra node is expanded. This is a case where the heuristic is exact and always returns that exact distance to goal. Adding an obstacle in the field would result in a totally different behavior.

- Euclidean distance. This is simply the norm between current vertex and the goal set. $d(x_i, X_{Goal}) = inf\{||x_i - a|| : a \in X_{Goal}\}$. This estimation is usually used in graphs that have 8-neighborhood connectivity and the search agent can traverse the graph diagonally. As shown in Figure 4.8 the search agent has expanded a lot of extra nodes. This is also expected, because Euclidean heuristic assumes the search agent can move at *any* angle. Obviously this is not the case and this heuristic returns values

[7] Please note that these methods are **only** valid for grid with square cells

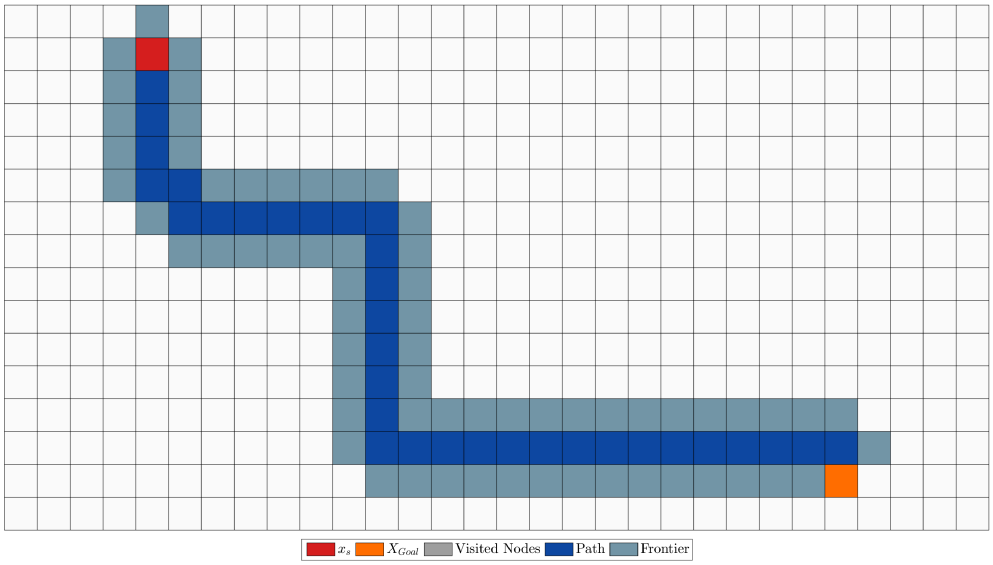


Figure 4.7: A* Search using manhattan heuristic

that are smaller than the exact distance. As discussed earlier this will result in more node expansion and a longer search time. But the final found path is still guaranteed to be the shortest path.

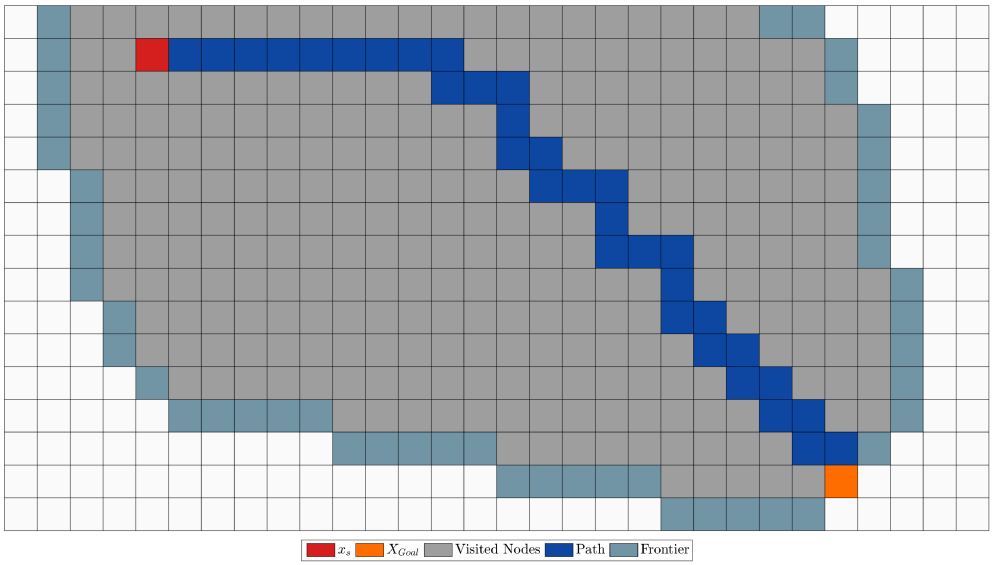


Figure 4.8: A* Search using euclidean heuristic

There are other heuristics such as Chebyshev and Octile but they are not discussed in this thesis.

4.5.2 Admissibility

As it was mentioned in Subsection 4.5.1 if the heuristic overestimates the cost of reaching the goal the optimality guarantee of the path will be broken and the heuristic is called non admissible. On the other hand an admissible heuristic will result in an optimal path. Basically $h(x_i)$ is a lower bound on the cost of getting from x_i to the goal. This is also another powerful feature of A* algorithm. The designer has the power to decide if they prefer an optimal path or they are just looking for *a* path and not necessarily the optimal one.

4.6 Hybrid A*

All algorithms that have been covered in this thesis can find a path, some of them even find the optimal path. But none of them guarantee that the generated path is actually drivable by the robot! Another issue that was never discussed is that what if we want the robot to arrive at the goal set in a specific angle? None of the current algorithms address these two problems. In this section we will first address the second problem and then discuss possible solutions for the first one.

Other than the dimension of the robot another geometric property of the robot plays a significant role in the generated path. That property is the minimum turning radius. The minimum turning radius is the radius of the smallest arc that a robot, or a car, can make. This radius has a great significance on the path and it's length. In 2007, Dolgov and Thrun used the hybrid A* algorithm and came in second in the DARPA Urban Challenge [29, 12, 11, 13]. In the publications related to the search algorithm it is shown that one can use a combination of discrete workspace and a continuous action space. The continuous actions space would solve the problems that raise with dynamical systems and their inherit constraints and the discrete workspace lets us use the discrete algorithms that have been working and tested for decades, algorithms like *BFS*, *DFS*, *A** and *Dijsktra*. In Subsections 4.6.1 and 4.6.2 the details of Hybrid A* will be discussed in detail.

4.6.1 Dubin's path

So far we have always assumed the robot can move from point a to any other point b in a straight line. But in reality there are nonholonomic constraints that does not necessarily allow movements on a straight line. Consider the scenario show in Figure 4.9 where the robot starts from x_s and the heading is $\frac{\pi}{2}$ and has to arrive at the goal set with the same $\frac{\pi}{2}$ heading. We are also interested in the *shortest* path between the two configurations.

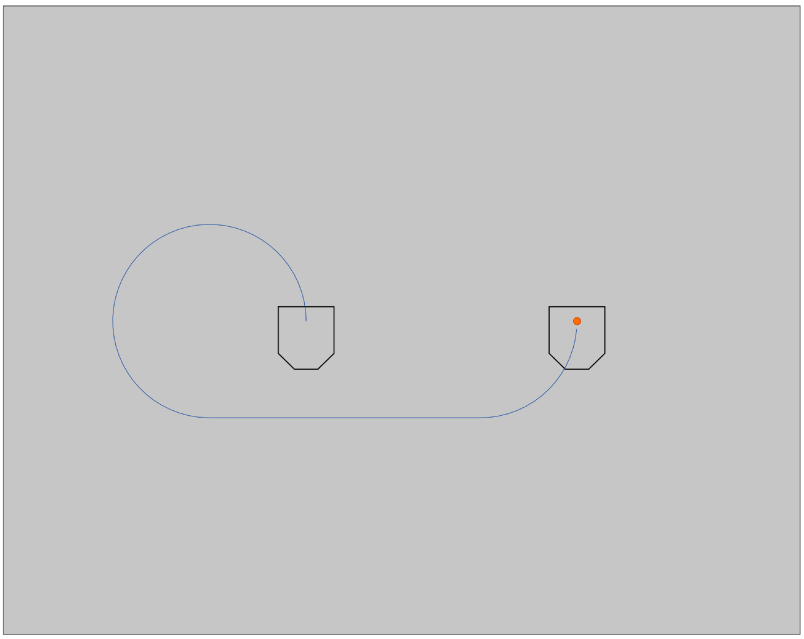


Figure 4.9: Dubin's path

Due to holonomic constraints of the system lateral movement is not possible but it is also not trivial to find such a path. In 1957, Dubins published a paper and showed there exist an analytical solution to find the shortest path necessary to connect any configuration x_s to any other configuration in X_{Goal} [14]. In order to find such a path, Dubin has simplified the model of a car. A Dubin's car is a car that can only move forwards at a unit velocity. This means the car cannot move backwards and it doesn't ever accelerate or brake. Such a car essentially has only 3 type of movements:

- Turning left at maximum

- Going straight
- Turning right at maximum

all the paths that a Dubins car can trace is a combination of such movements. Dubins has given a name for each one of these movements:

- Turning left at maximum : **L**
- Going straight : **S**
- Turning right at maximum : **R**

and he has proved that all shortest paths between any two configurations can be described by only 6 combination of such movements and they are **RSR**, **LSL**, **RSL**, **LSR**, **RLR** and **LRL**. The idea of a Dubins path is used in the *Hybrid A** algorithm to drive the car at towards the specified algorithm in the goal set.

4.6.2 Action space

In previous algorithms the action space was very simple, just up, down, left and right movements. Actions that do not take into account the dynamics of the robot and possible existing constraints. This will result in discrete and underivable paths. Doglov et al. [12] have shown that unlike discrete algorithms one can assume that the search agent can reach any continuous point on the grid and not necessarily the vertex. Transitioning from one node to another in this continuous workspace give us the ability to incorporate the non-holonomic nature of the dynamical system. This means that just like the conventional A* the 2D (x, y) search space is discretized but unlike A* which only allows the search agent to expand nodes on the centers of cells, hybrid A* lets the search agent at any point in a cell and then associates the (x, y, θ) of the expanded node to the cell.

Algorithm 7 Hybrid A* Search

Require: $x_s \cap X_{Goal} \in X$

$O = PriorityQueue()$
$C = \emptyset$
$O.push(x_s)$
$x_{old_succ} = (0,0)$
while $O \neq \emptyset$ **do**
    $x_i \leftarrow$ O.pop()
    $C.push(x_i)$
    if $x_i \in X_{Goal}$ **then**
        return
    else
        for $u \in U(x_i)$ **do**
            $x_{succ} \leftarrow f(x_i, u)$
            if $x_{succ} \notin C$ **then**
                $g_{tentative} \leftarrow g(x_i) + cost(x_i, x_{succ})$
                if $x_{succ} \notin O$ **or** $g_{tentative} < g(x_{succ})$ **then**
                    $g(x_{succ}) \leftarrow g_{tentative}$
                    $h(x_{succ}) \leftarrow FindHeuristic(x_{succ}, x_{Goal})$
                    $tempF(x_{succ}) \leftarrow g(x_{succ}) + h(x_{succ})$
                    if $x_{old\_succ} == x_{succ}$ **then**
                        if $tempF(x_{succ}) > f(x_{succ})$ **then**
                            continue
                        end if
                    end if
                    $O.push(x_{succ})$
                end if
                $x_{old\_succ} = x_{succ}$
            end if
        end for
    end if
end while

4.6.3 Node expansion

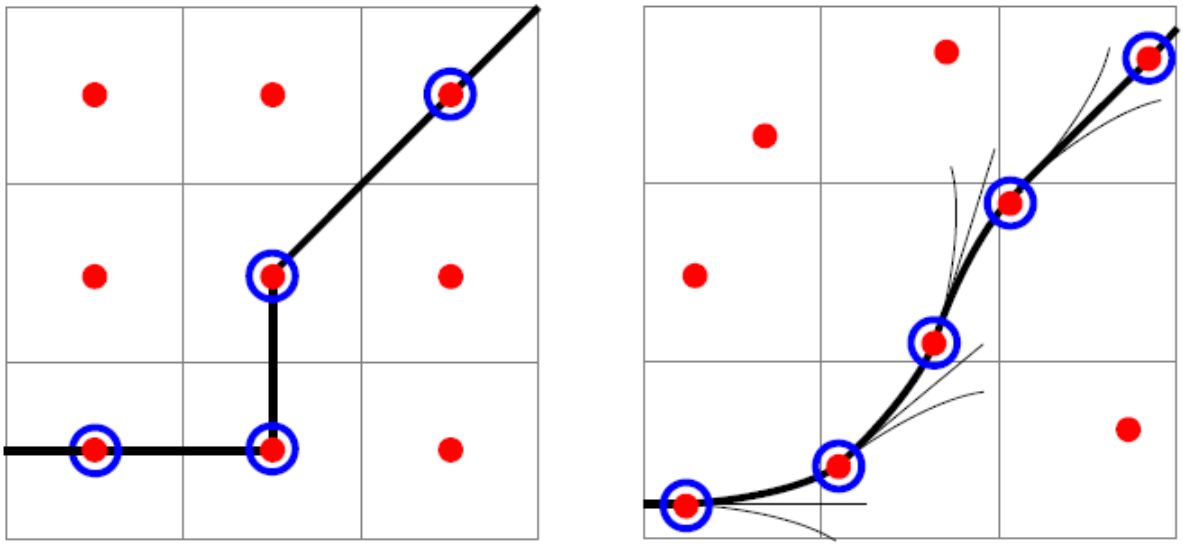
As shown in Figure 4.10, *Hybrid A** uses a continuous action space to explore new nodes. In order to explore new nodes the continuous transition function is applied to the current state and the new successor nodes are found. The state at node v_i is characterized by (x, y, θ) where x and y represent the Cartesian coordinate of the node and θ shows the heading of the search agent, it is safe to assume that in Hybrid A* the search agent is a car/robot. Using the idea of the Dubins car each node is expanded by driving the car using three different scenarios:

- Constant velocity and maximum steering to left.
- Constant velocity and no steering.
- Constant velocity and maximum steering to right.

that's why there are only three lines out of each node in Figure 4.10. Each arc represents the path the robot can drive in a given time dt with the minimum turning radius. Each one of these actions is applied only for a fixed amount of time. The time depends on the granularity of the gird and it is usually selected in such a way that it gives enough time to the search agent to drive to a different cell, this will speed up the search. This idea of combining Dubins paths and continuous action space guarantees that the path generated by the search algorithm is drivable by the robot/car.

In A* the cost of actions in action space was simply the distance between two neighboring cells. In Hybrid A* the cost is the length of the arc that connects two nodes. As it will be apparent later, in order to get a smoother path it would be a good idea to give additional costs when a change in driving direction occurs, the penalty for changing is usually a constant value. As shown in Algorithm 7 just like normal A* when the search agent reaches a cell that is not in the closed set the expansion continues, this means that the cell has never been expanded. Then there will be two scenarios:

- The cell is not in open list (The cell is not expanded through any other node)



(a) A generic implementation of A* only explores centers of cells and the f score is associated with the center of each cell

(b) Hybrid A* can visit any point inside a cell and the f score is associated with each continuous state rather than the center of a cell

Figure 4.10: Generic discrete A* vs hybrid continuous A* [12]

- The cell is in open list, but the tentative cost is lower than the current cost of the cell. (This means that the current cell was already in the frontier but the cost of old path is higher than the tentative cost of the current path)

in both cases the search continues. Figure 4.10 reveals yet another critical difference between A* and Hybrid A*. During node expansion it is very likely that there are multiple successor nodes in the same gird cell. This adds another level of complexity to the algorithm. This is important because we are still associating continuous vertices with discrete grid cells and in cases where there are multiple vertices in the same cell the algorithm should first calculate the f score for all successor nodes residing in the same grid cell and then push the node that has a smallest f score into the priority queue.

4.6.4 Analytical expansion

Continuous node expansion resolves the problem of underivable paths, but we still do not have a solution for reaching exact goal configurations. The current solution will generate a drivable path from start configuration to the goal set but it does not guarantee that it can reach a specific configuration in the goal state. In order to overcome this problem Dogolv has proposed to use Dubins path algorithm to reach any goal configuration. This means that during node expansion the algorithm should try to find a drivable obstacle free path from the current node configuration to the goal configuration. On the other hand finding the obstacle free Dubins path for every single node expansion is a computation intensive operation. Instead of looking for such a path at every node expansion, the operation is done every *n*th iteration and as the search agent gets closer to the goal the frequency is increased because it is more likely to find a path when the search agent is closer to the goal configuration.

4.6.5 Heuristics

As it was discussed in Subsection 4.5.1 selecting the right heuristic is crucial for an optimal search and further more if the heuristic is not admissible the search can not even find the optimal path. The nature of search algorithms discussed so far doesn't allow for integrating the dynamic and geometry of the search agent to generate a drivable path that follows the non-holonomic constraints of the system. In those algorithms the heuristics could be a simple euclidean norm because it is assumed that the search agent can move in any direction. In Hybrid A* a more sophisticated heuristic is used which takes into account the dynamic and non holonomic constraints of the system. The Hybrid A* heuristic is a combination of two different heuristics, a constrained heuristic and an unconstrained one. These heuristics capture two very different part of the problem. The constrained heuristic assumes there is no obstacle in the environment and embodies the non-holonomic and dynamical constraints of the system while the unconstrained heuristics neglects the vehicle constraints and only takes into account the configuration space and the work space. The final heuristic is the minimum

of one of the two heuristics. This will guarantee the admissibility of the algorithm and also takes into account the dynamics of the system.

4.6.6 Constrained heuristics

The constrained heuristic neglects any and all obstacles in the workspace and only focuses on the dynamics and constraints of the system. At each node expansion a Dubins path is found from the expanded node to the goal state. Then the length of this path is used as one of the possible heuristics.

The current heading and geometrical properties of a system do not have any effect on a normal euclidean heuristic. But this heuristic considers the current heading, goal configuration and also the geometrical properties of the robot, such as minimum turning radius, into account. This heuristic cannot ever over estimate the length of the optimal path and usually performs much better than a simple euclidean distance because it is closer to the real path and thus less nodes are expanded.

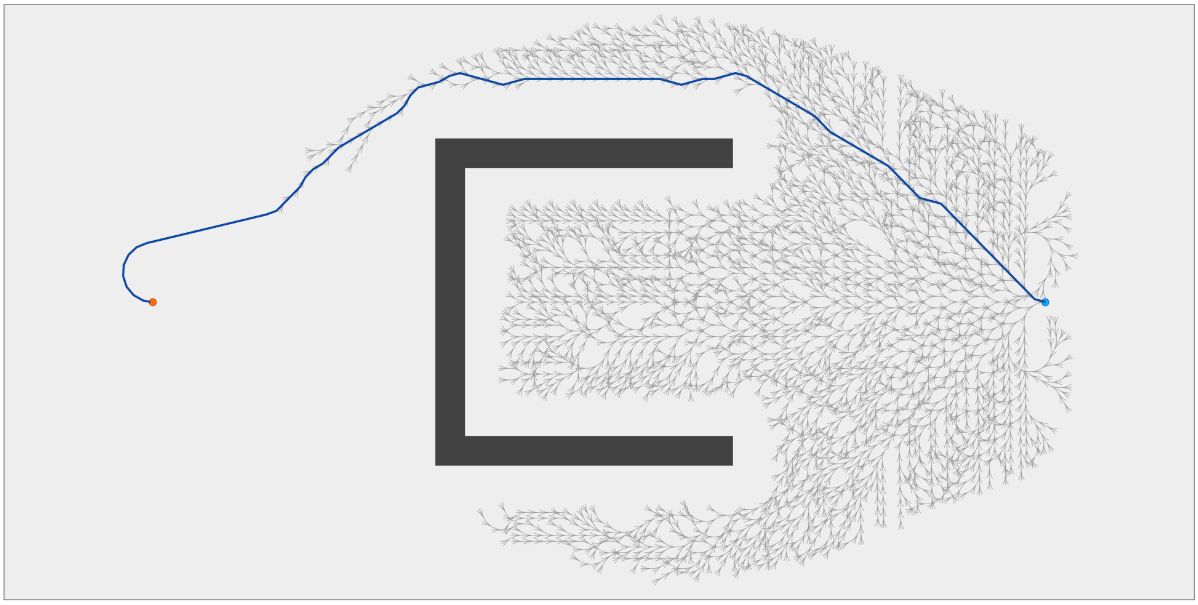


Figure 4.11: Euclidean heuristic, visited nodes: 3011

Path Length: 7.567432 m

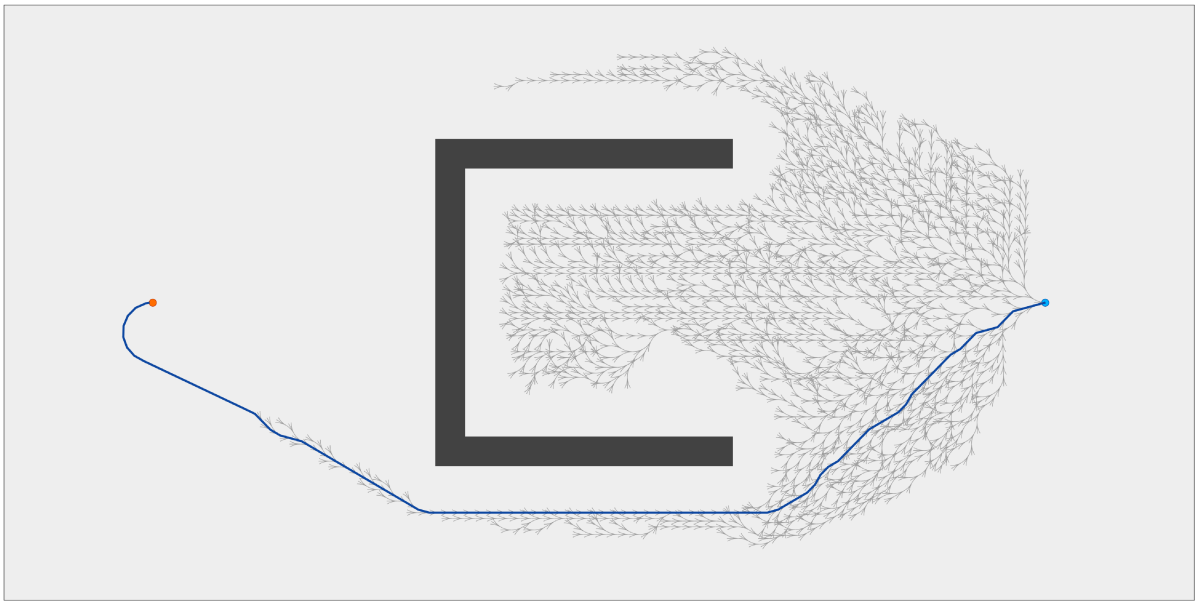


Figure 4.12: Constrained Dubins heuristic, visited nodes: 2213

Path Length: 7.408262 m

Figure 4.12 and Figure 4.11 clearly show the difference and the effect that the unconstrained heuristic plays in Hybrid A* search. The number of visited nodes is reduced by *26.5%* and the length of the path is reduced by around *3%*. The improvement in the length of the path might not be significant but the reducing the number of visited nodes significantly speeds up the search, which is the main bottle neck in real time applications.

4.6.7 Unconstrained heuristics

Unconstrained heuristics disregards the non-holonomic constraints of the system and instead focuses on available the obstacles in the work space. This heuristic should give us a very good estimate of the length of a path while considering the available obstacles. But how is that even possible, if we want to know the length of the path we should first search and then find the length of the path, how can we find the path without searching? The answer is we don't know the length of the path and we must do the search but we need a fast search! The fastest search algorithm that was covered in this thesis was $A*$ and that

is exactly what we are going to use. At each node expansion we will first search the discrete work space without considering the constraints on the system and use the result of the search as the heuristic for the Hybrid A*.

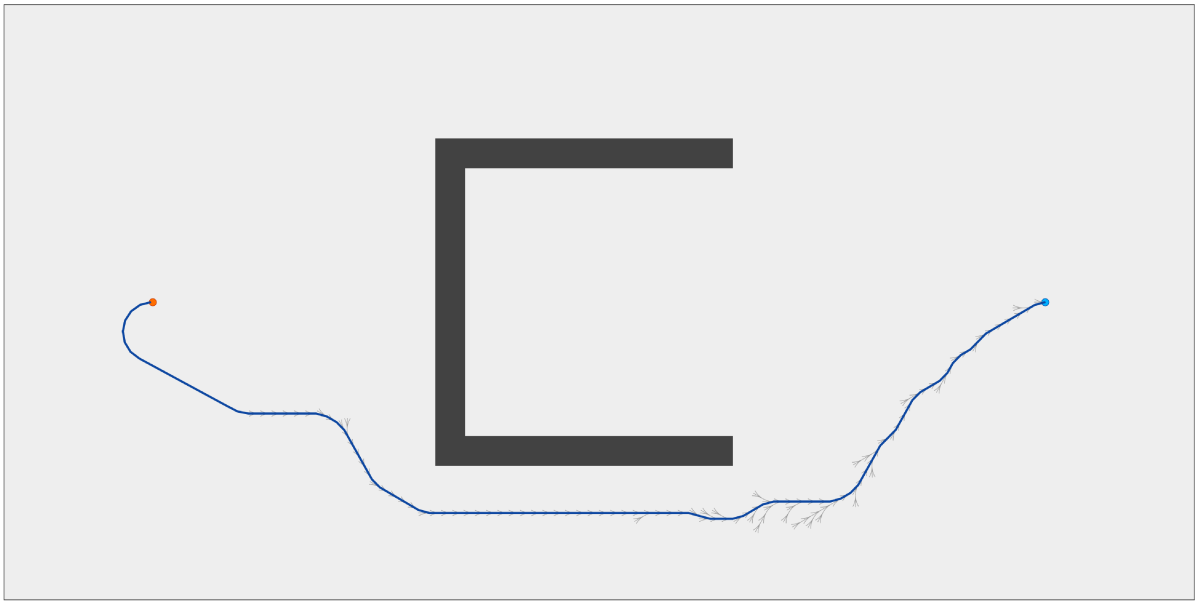


Figure 4.13: Unconstrained Heuristic, Visited Nodes: 107

Path Length: 7.617800 m

The algorithm to find the heuristics will be [8]

Algorithm 8 Hybrid A* Heuristics

1: $DubinsPath = FindDubinsPath(x_j, X_{Goal})$

2: $DubinsPathLength = FindDubinsPathLength(DubinsPath)$

3: $AStarPath = FindAStarPath(x_j, X_{Goal})$

4: $AStarPathLength = FindAStarPathLength(AStarPath)$

5: $heurstics(x_j) = max(AStarPathLength, DubinsPathLength)$

As shown in Figure 4.13 the number of visited nodes has drastically decreased to 107 nodes. This is a huge improvement! The constrained heuristics will guide the search agent away from dead ends and obstacles There is a caveat though, something which might not

[8] Note that this algorithm soubd be executed for *every* single node expansion

be clear. At each iteration we are running Algorithm 8 which means a complete normal A* search using Euclidean heuristics. This will significantly slow down the search algorithm. There are some workarounds for this issue though. One could run the A* search for every single cell in the workspace prior to the Hybrid A* search and then save the result in a look up table, then during the Hybrid A* search the search agent will have to lookup that table for the A* path result, this will make Hybrid A* a perfect and powerful solution for path planning of mobile robots and cars.

Chapter 5

Implementation & Results

In the previous sections we discussed how to find a path for a mass point robot, a rigid body and a non-holonomic robot. In this section we will discuss how to integrate all of the previous steps to generate a path and successfully follow it. Hybrid A* is the most complicated and involved algorithm discussed in this thesis. In this chapter only the results and details of implementing Hybrid A* will be discussed.

As discussed in the introduction in this work we assume:

- We have a complete and correct map of the environment
- The goal is to find a path, a set of waypoints, from a start configurations to a goal set
- Minimal effort is done on the path tracking section

In the following section the overall structure of the experiment and lab equipments will be discussed.

The developed *Hybrid A* path planner* is able to generate a path from any initial start configuration to a the goal set with a maximum frequency of 20HZ.

5.1 General structure of experiments

5.1.1 Software

All tests done in this work were designed and implemented in MATLAB 2015b using QUARC Real-Time Control Software [4]. The toolbox provides a soft Real-Time control environment on Windows through MATLAB and Simulink.

5.1.2 Hardware

All tests were performed on a Windows 7 Dell Precision Tower 3000 Series work station.

5.1.3 Mobile robot

The test platform is a research platform designed and developed by Quanser, marketed as QGV (Quanser Ground Vehicle). TODO: add reference to Quanser manual for QGV

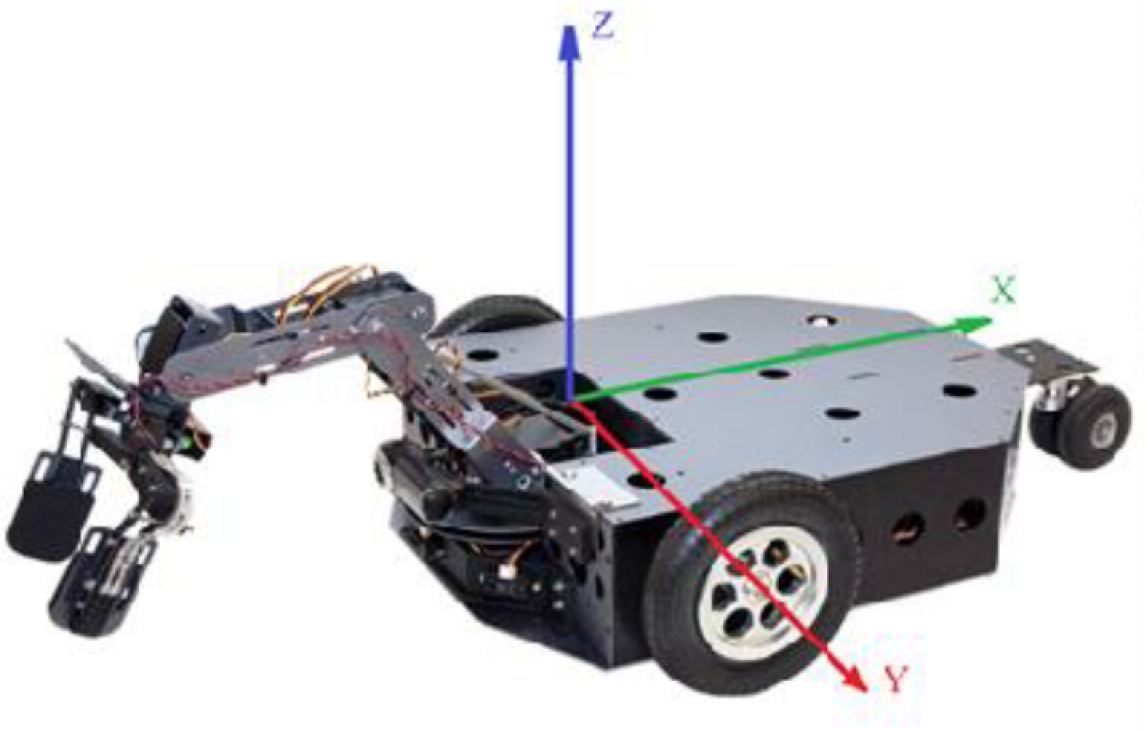


Figure 5.1: Quanser Ground Vehicle, QGV

5.2 Path planner structure

As shown in Figure 5.2 the inputs to the path planner are the occupancy grid, start configuration x_i and the goal set X_{Goal}. Once the inputs are prepared a check is done to make sure the goal and start configurations and set reside within the occupancy grid. Then the input is fed to the search algorithm. The search then performs Hybrid A* and generates a drivable path. Usually the path is not smooth. The path is then passed to another algorithm to smoothen the path. The final smooth path is guaranteed to be drivable by the robot and is very close to the most optimal path.

In the following sections each section of the path planner is explained in more details.

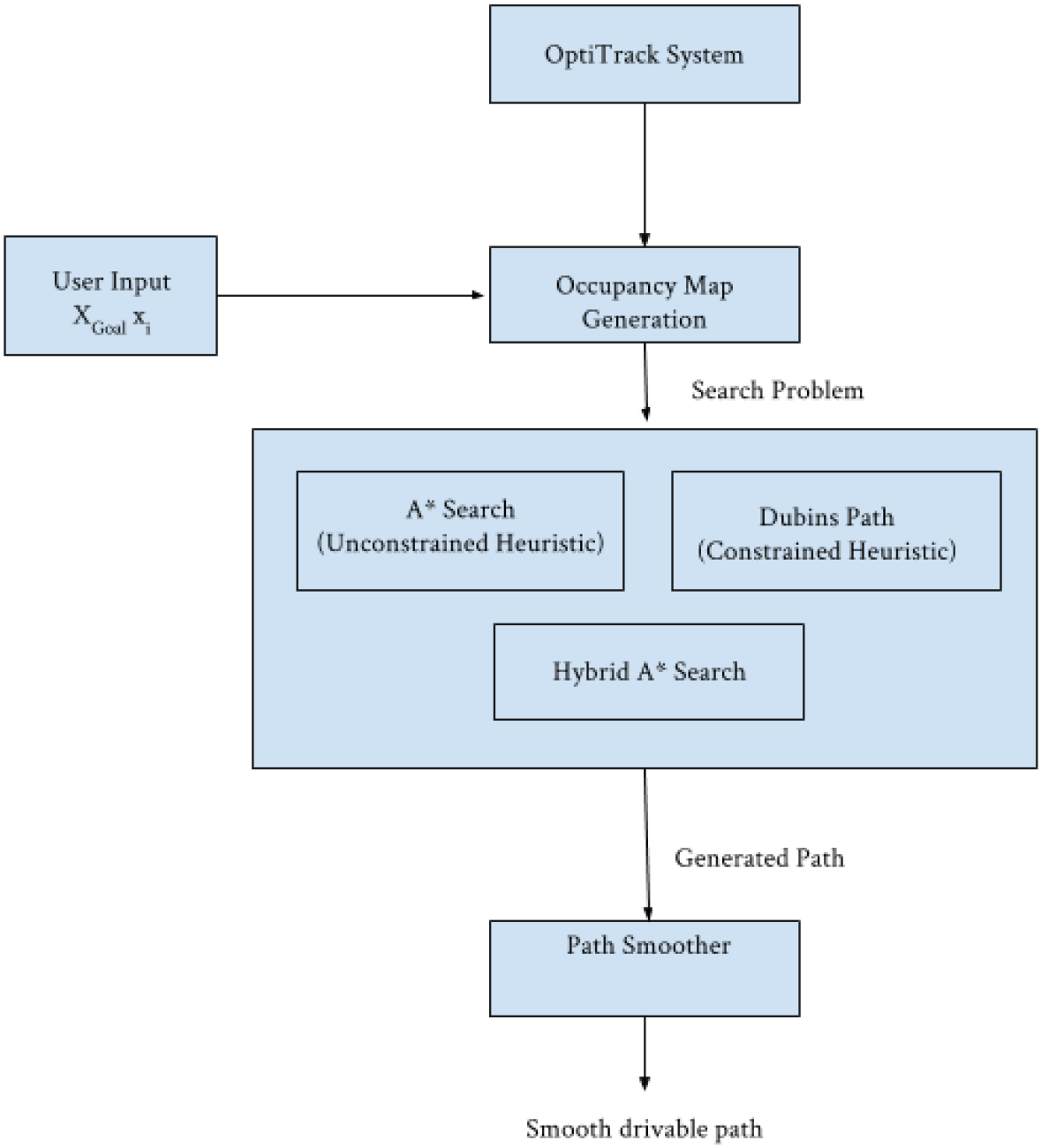


Figure 5.2: Hybrid A* path planner

5.2.1 Localization

All tests were performed in a controlled in door environment. The OptiTrack [3] motion capture system was used to localize the robot and the obstacles. A set of 24 *Flex 3* cameras are used for localization. The cameras emit IR light which is then reflected off of markers mounted on the QGV and the obstacles back to the cameras' sensors. The sensors then estimate the location of all bodies in the workspace. The output of the cameras is then made available in MATLAB through the Quarc Real-Time Control Software.

Figure 5.3: Flex 3 OptiTrack motion capture system

5.2.2 Occupancy grid

Using the output of the OptiTrack system an occupancy grid is generated. The occupancy grid is a **4x 4** (m x m) grid with a resolution of **5 cm**. This will create an **80 x 80** pixel grid, a total of 6,400 cells. The grid is represented by a simple MATLAB matrix. If the value of a cell is 1 that cell is occupied, other wise the cell is free. The occupancy grid is passed to the search algorithm along with start state and the goal configuration.

5.2.3 Search

The workspace shown in Figure 5.4 is selected for the search and all tests are performed in this workspace. The blue dot represents the start state and the orange dot represents the goal state. The final goal is to find a drivable path for th vehicle shown in Figure 5.1.

The properties of the configuration space have not been discussed yet. The first question to ask is if it is a 3D space or a reduced 2D space? It is much easier and computationally

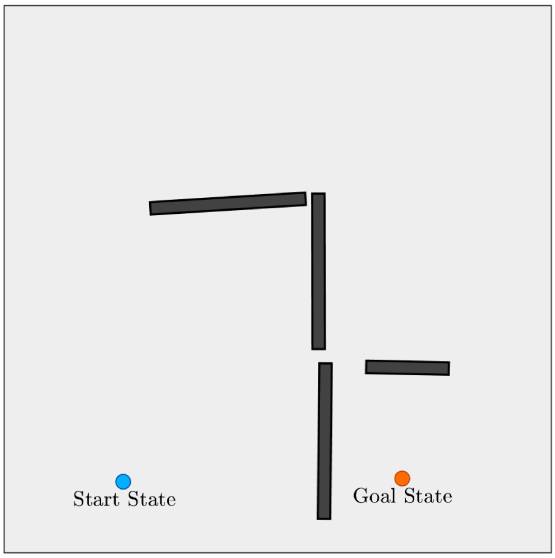


Figure 5.4: Work Space

less intensive if we assume the robot is a circle and then find the Minkowski sum of the circle, representing the robot, and all other obstacles in the workspace, thus representing the 2D work space with a 2D configuration space in $\mathbb{R}^2$. But there is a more elegant way to achieve the same behavior. One could use the idea of potential functions [16] and create a similar space on top of the current gird and then use the magnitude of the potential function at each cell is the extra cost associated with nodes in that cell. This will make the algorithm even more powerful. Consider a case where there are several obstacles in the work space but some obstacles are much more critical than the others, i.e. it *might* be OK if we hit some specific obstacles in the work space with the cost of totally avoiding some other critical obstacles. This behavior is not feasible through canonical specification of a workspace but adding different costs to different cells, depending on their proximity to the obstacles, makes this behavior possible and it also seems like a very natural solution to the problem and at same time very intuitive.

Figure 5.5 is a grid representation of the workspace illustrated in Figure 5.4. The yellow cells are cells that have a very high cost and they will be the very last cells expanded by the search algorithm. The gradient in color shows that the blue cells and dark purple cells are also costly but they do not cost as much as the yellow cells. All the white cells

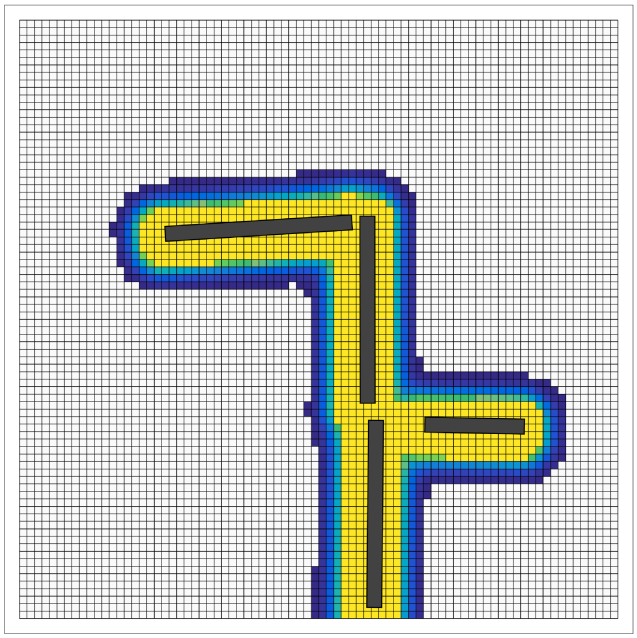


Figure 5.5: Cost map of a work space

have a cost of zero. This will make the cells close to the obstacles have a very high cost and as we get away from the obstacle the cost is reduced. Then after a certain distance away from the obstacles, this should be configurable threshold and it should depend on the size and geometry of the mobile robot, all other cells have a cost of zero.

We have a well defined grid, the start state is the blue dot and the starting heading of the QGV is $-\pi$. The goal is the orange dot with the same heading as the starting state. Using the algorithm discussed in Algorithm 7 we will get the following path.

Figure 5.7 shows the same work space and the path that Hybrid A* algorithm has found for the depicted QGV. As it can be seen the path that Hybrid A* has generated is not smooth. In the next section it will be discussed how this path is smoothed.

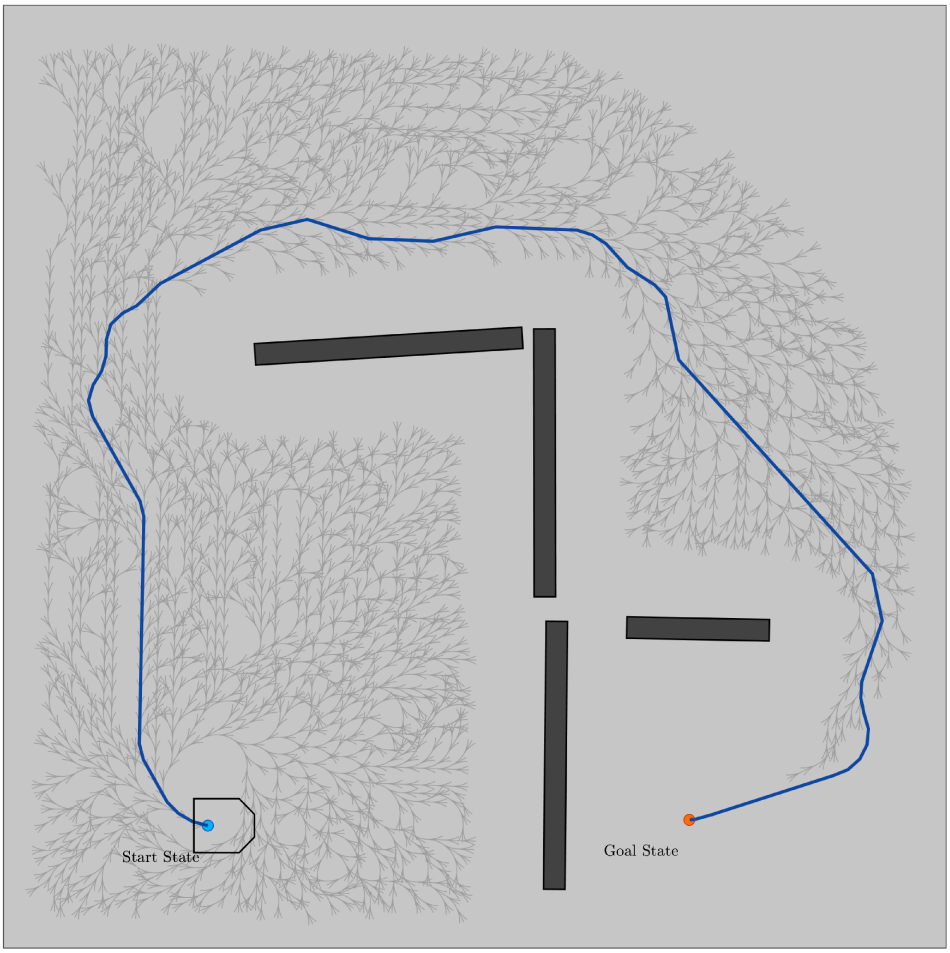


Figure 5.6: Hybrid A* path

5.3 Path smoothing

Although the paths generated by Hybrid A* are drivable there are a lot of redundant changes in steering that will generate large control inputs that some times even saturate the actuators. Usually an algorithm is used to relocate the way points ever so slightly to attain a higher degree of smoothness in both the control efforts and the generated path. Dimitry Doglove that worked on the DARPA's grand challenge on self driving car has proposed an algorithm to smoothen such paths [12].

Once the Hybrid A* algorithm is finished it will generate a series of waypoints as the final path. This sequence of waypoints could be presented as $X_i = (x_i, y_i), i \in [1, N]$ where

N is the total number of waypoints in the path. Defining the following variables will help us express the final cost function:

- O_i the location of the closest obstacle to the ith waypoint.
- $\Delta X_i = X_i - X_{i-1}$ the displacement vector at the ith waypoint.
- $\Delta\phi_i = |tan^{-1}\frac{\Delta y_{i+1}}{\Delta x_{i+1}} - tan^{-1}\frac{\Delta y_i}{\Delta x_i}|$ the change in the tangential angle at the ith waypoint.

The first term of the cost function is $P_{obstacle}$ which penalizes based on how close a waypoint is to an obstacle. It was [12] found that a simple quadratic function works well as a cost function.

$$P_o = w_o \sum_{i=1}^{N} \sigma_o(|X_i - O_i| - d_{max}) \tag{5.1}$$

where d_{max} is the maximum distance that an obstacle can affect the cost of the path. Notice that d_{max} is constant for all obstacles and waypoints. The weight w_o coefficient determines how important this cost function is. If $|X_i - O_i| > d_{max}$ we simply ignore this term because it is too far from an obstacle to have a significant effect. Otherwise the gradient of this term will be:

$$\begin{aligned} \frac{\partial \sigma_o}{\partial x_i} &= 2(|x_i - O_{x_i}| - d_{max})\frac{x_i - O_{x_i}}{|x_i - O_{x_i}|} \\ \frac{\partial \sigma_o}{\partial y_i} &= 2(|y_i - O_{y_i}| - d_{max})\frac{y_i - O_{y_i}}{|y_i - O_{y_i}|} \end{aligned} \tag{5.2}$$

The second term of the cost function is the $P_{curvature}$. At each waypoint we must check that we are not creating a path that the robot cannot drive. This means that the instantaneous curvature at each waypoint should be in the drivable range of the robot. For each waypoint i, there are two other points that affect the curvature, $i-1$ and $i+1$. The tangential angle at each waypoint with respect two the other two waypoint can be calculated as

$$\Delta\phi_i = cos^{-1}\frac{\Delta X_i^T \Delta X_{i+1}}{|\Delta X_i||\Delta X_{i+1}|} \tag{5.3}$$

As mentioned earlier this term is to ensure drivability of the path which means that at each waypoint the curvature should be larger than the maximum drivable curvature. So we can define the curvature term as:

$$P_c = w_c \sum_{i=1}^{N-1} \sigma_c(\frac{\Delta\phi_i}{|\Delta X_i|} - \kappa_{max}) \tag{5.4}$$

where κ_{max} is the maximum drivable curvature and the quadratic function σ_c. We can then define the curvature of the path as $\kappa_i = \frac{\Delta\phi_i}{\Delta X_i}$. The derivative of this curvature at each on of the 3 waypoints can be calculated as

$$\begin{aligned}
\frac{\partial\kappa_i}{\partial X_i} &= \frac{-1}{\Delta X_i}\frac{\partial\Delta\phi_i}{\partial cos(\Delta\phi_i)}\frac{\partial cos(\Delta\phi_i)}{\partial X_i} - \frac{\partial\phi_i}{(\Delta X_i)^2}\frac{\partial\Delta X_i}{\partial X_i} \\
\frac{\partial\kappa_i}{\partial X_{i-1}} &= \frac{-1}{\Delta X_i}\frac{\partial\Delta\phi_i}{\partial cos(\Delta\phi_i)}\frac{\partial cos(\Delta\phi_i)}{\partial X_{i-1}} - \frac{\partial\phi_i}{(\Delta X_i)^2}\frac{\partial\Delta X_i}{\partial X_{i-1}} \\
\frac{\partial\kappa_i}{\partial X_{i+1}} &= \frac{-1}{\Delta X_i}\frac{\partial\Delta\phi_i}{\partial cos(\Delta\phi_i)}\frac{\partial cos(\Delta\phi_i)}{\partial X_{i+11}}
\end{aligned}$$

The third and last term that was used to smoothen the path is $P_{Smoothnessterm}$. This term penalties the displacement vectors between two waypoints. The waypoints that are further from their neighborhood waypoints are given a larger cost. Also the waypoints that change the heading of the path are given a higher cost.

$$P_s = w_s \sum_{i=1}^{N-1} \sigma_c(\Delta X_{i+1} - \Delta X_i)^2 \tag{5.5}$$

The proposed algorithm consists of running gradient decent on the following cost function:

$$P = P_o + P_c + P_s \tag{5.6}$$

and adjust the location of the way points according to the 3 mentioned terms. The final gradient descent algorithm is shown in Algorithm 9 on the next page. The output of this algorithm will be a list of waypoints that geometrically is much smoother and it is still drivable by the robot. An important note that must be mentioned is that in most gradient

descent algorithm there is a stop threshold. This threshold determines when the algorithm should stop. The proposed algorithm uses an iterator and when the iterator is reached a certain number the optimization stops.

Algorithm 9 Gradient descent on waypoints

$iterations$ = 500

$i = 0$

while $i < iterations$ **do**

  for all $x \in X$ **do**

    $correctionAtThisWaypoint = (0,0)$

    $correctionAtThisWaypoint = correctionAtThisWaypoint - P_o(x_j)$

    $correctionAtThisWaypoint = correctionAtThisWaypoint - P_c(x_{j-1}, x_j, x_{j+1})$

    $correctionAtThisWaypoint = correctionAtThisWaypoint - P_s(x_j, x_{j+1})$

    $x_j = correctionAtThisWaypoint + x_j$

  end for

  $i = i + 1$

end while

The smoother algorithm can be applied on any set of way points and thus it can be applied on the way points generated by other algorithms. Figure 5.7 represented a path found using Hybrid A* but it is not a smooth path. Once the smoothening algorithm is applied on this path it will generate a much smoother path as the output. The output of this algorithm is depicted in Figure 5.7. The difference between the two paths is more obvious in Figure 5.8.

We now have a path but we how should it be tracked? So far we have only discussed the path generation and path tracking was not discussed at all. In the next section we will cover how to track a path and control the robot to follow the path as closely as possible.

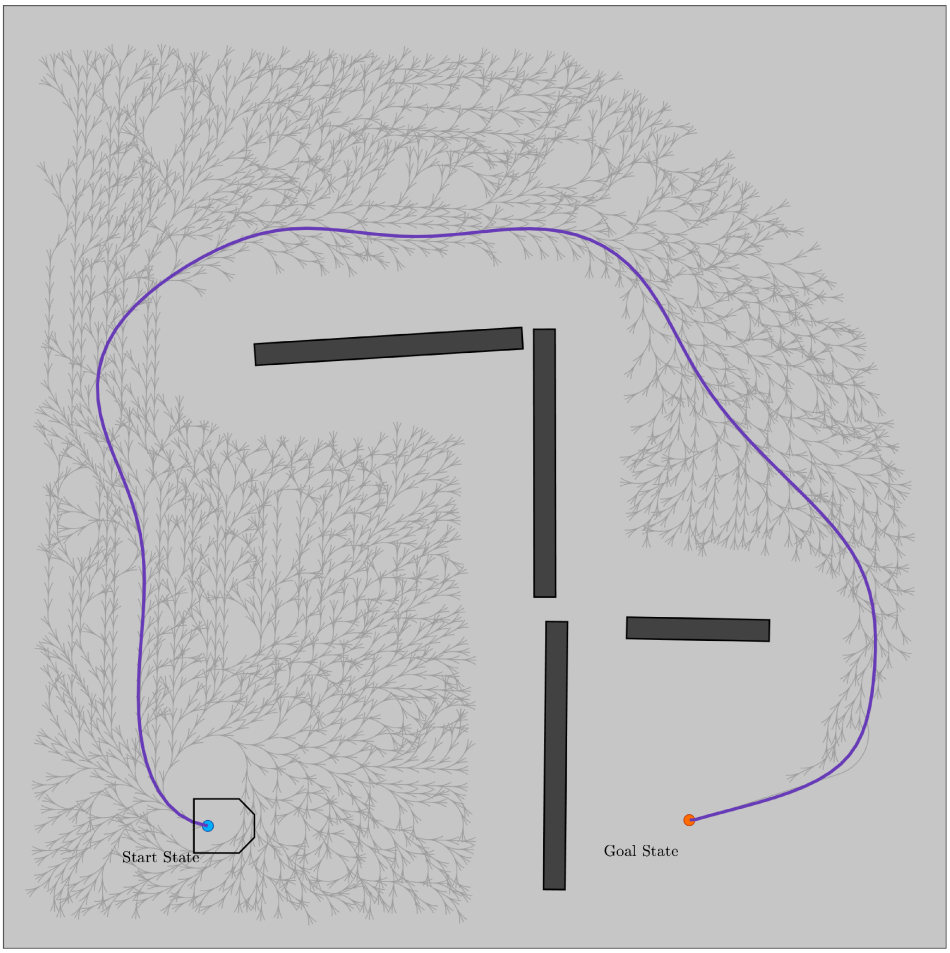


Figure 5.7: Smooth Hybrid A* path

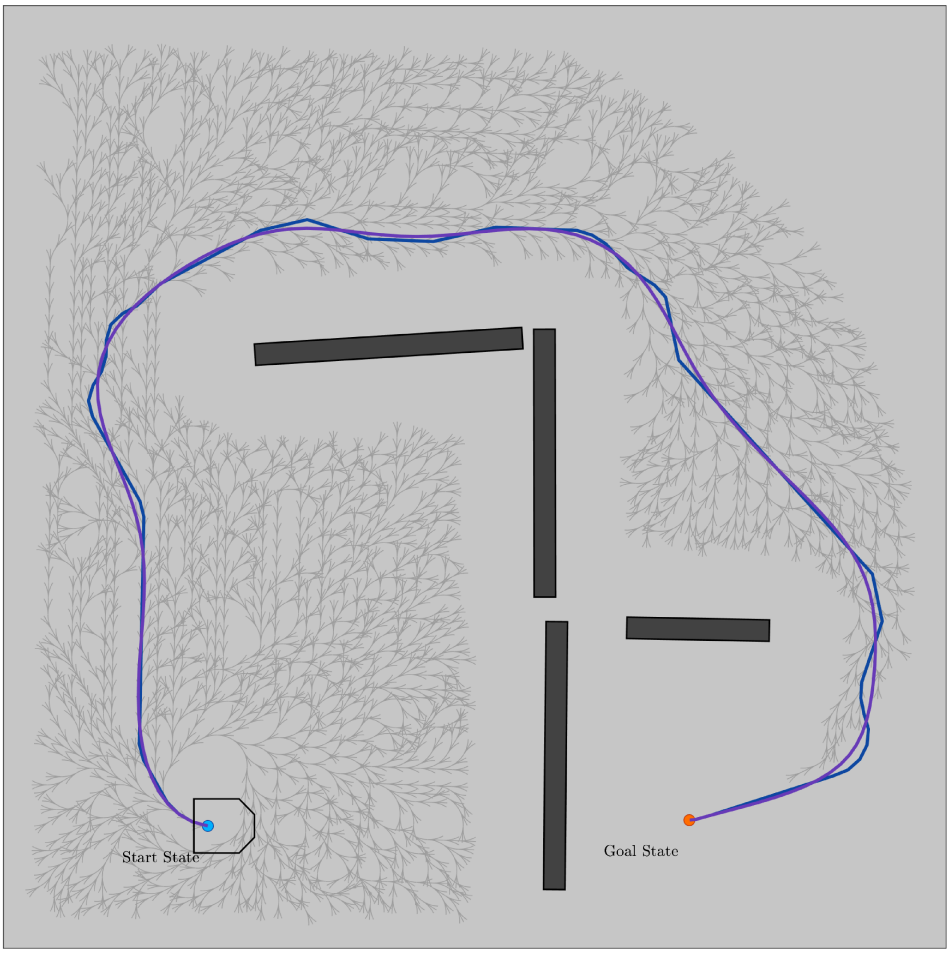


Figure 5.8: Hybrid A* path vs a smooth hybrid A* path

5.4 Path tracking

As mentioned in the introduction the goal of this thesis was to generate a drivable path but the path tracking are relatively simple and promising. In this section we will cover the a simple geometrical algorithm for path tracking.

The paths generated so far are simple geometric waypoints that are connected together with straight lines. The only assumption in path generation is a constant velocity of the vehicle and the minimum turning radius. There are a lot of algorithms that can use way points to create speed and acceleration profiles but they are usually complicated and add extra time and computation power to the algorithm. Maybe the most famous algorithm to generate a smooth path using waypoints is spline which tries to create a continuous *profile* for velocity and acceleration. The major concern with splines is that it will try to insert or remove waypoints from the given list of waypoints and there is no natural way to integrate the algorithm with the configuration space and the existence of obstacles in the work space. This means there should be additional steps and operations performed on the path generated by splines to verify that the path still goes through the free space.

The idea of generating a set of waypoints and tracking the waypoints is one of the earliest problems in video game development [31]. Like all other problems in robotics, game developers have already come up with a smart, efficient and easy to implement algorithm for this problem. The *pure-pursuit* algorithm are among the most common algorithms of path tracking for mobile robots or even UAVs [30]. The algorithm and its variations are so efficient and powerful that they were used by multiple different teams in the 2007 DARPA grand challenge, MIT [21] , Stanford [34] and Carnegie [35]. The Stanford team even won the competitions with this algorithm with a slight change in both name [1] and logic. The algorithm for mobile robot and car like vehicles was first introduced by Craig Coulter [9] at Carnegie Mellon University. If the speed of the vehicle is restricted, such as an indoor lab environment, the problem of controlling the steering of a robot can be approached as a pure kinematic problem. The performance of the algorithm has been compared to other modern

[1] They renamed the algorithm after their car *Stanley*!

control algorithms.

Tracking Method	Robustness to Disturbances	Path Requirements	Cutting Corners	Overshooting	Steady State Error	Good Applications
Pure Pursuit	Good	None within reason	Significant as speed increases	Moderate as speed increases	Significant as speed increases	Slow driving and/or on discontinuous paths
Stanley	Fair	Continuous curvature	No	Moderate as speed increases	Significant as speed increases	Smooth highway driving and\or parking maneuvers
Kinematic	Poor	Continuous through 2nd derivative of curvature	No	Moderate as speed increases, significant during rapidly changing curvature	Significant as speed increases	Smooth parking maneuvers
LQR with FF	Poor	Continuous curvature	No	Significant during rapidly changing curvature	Minimal until much higher speeds	Smooth high speed driving, urban driving at speed
Preview	Fair	None within reason	Moderate in rapidly changing curvature and/or speed	Moderate in rapidly changing curvature and/or speed	Minimal until much higher speeds	Highway driving at relatively constant speed

Figure 5.9: Empirical comparison of tracking results [25]

Pure-Pursuit is a geometric tracking algorithm that finds the arc that following it will move a vehicle from its current position to an arbitrary goal position. The way the algorithm works is very intuitive because it tries to replicate how humans actually track a specific path. The algorithm first needs a goal point at some distance a head of its current position. That distance between the current position and the *current* goal is called the *look a head distance*. The vehicle is then assumed to chase such a goal point that is always ahead of the vehicle by a *look a head* distance, the vehicle is *pursuing* this current goal point. This is very similar to they way that the humans drive. We also look at some specific point in front of us and that point changes as we drive in different conditions. Please note that the current and the look ahead points are changing continuously and the only constant part of this algorithm is the look a head distance.

Consider the diagram depicted in Figure 5.10, the following equation holds about the geometry of the problem

$$\begin{aligned} x^2 + y^2 &= l^2 \\ x + d &= r \end{aligned} \tag{5.7}$$

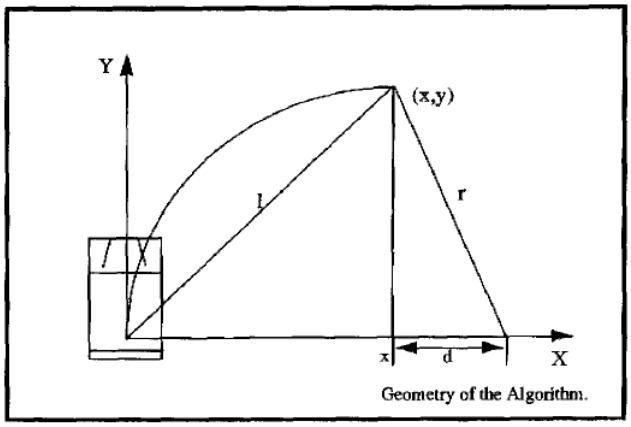


Figure 5.10: Geometry of Pure-Pursuit algorithm

The goal is to relate the curvature of the arc with the lateral x offset. Following simple geometry we will have

$$\begin{aligned} d &= r - x \\ (r-x)^2 + y^2 &= r^2 \\ 2rx &= l^2 \\ r &= \frac{l^2}{2x} \\ \kappa &= \frac{2x}{l^2} \end{aligned} \tag{5.8}$$

Just like the algorithm itself implementing the pure-pursuit algorithm is very straight forward.

5.5 Dynamic path planning

In all previous sections the assumptions was that the work space is static. The position and the number of the obstacles do not change. The following sections will discuss the results produced in a dynamic environment.

As illustrated in figure Figure 5.11 the algorithm is very similar to static search. The main difference is the fact that the search is done continuously to take into account obstacles

that move in the work space. This means that the search algorithm should be fast enough to account for the movement of obstacles in the work space and generate a safe and drivable path in a timely fashion.

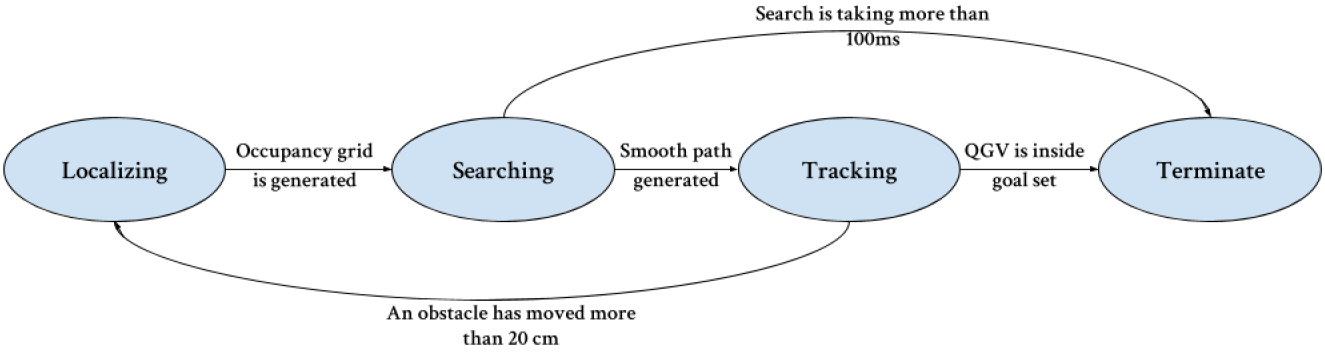


Figure 5.11: Hybrid A* search in a dynamic environment

The diagrams in the next pages illustrate the workspace, the generated path and the path that the QGV has tracked with the mean squared error from the original path.

As it can be seen from different configurations the Dynamic Hybrid A* path planner has successfully found a path and most importantly the QGV has tracked the path using pure pursuit algorithm very closely with minimal cross track MSE error. The most significant downside to the pure pursuit algorithm is that it doesn't get close enough to the goal configuration. Figure 5.20 shows that the QGV does not reach the goal configuration as expected. The reason lies in how the waypoint that should be tracked by the *Pure Pursuit* algorithm is update. When the QGV reaches a specific way point the algorithm should update and find the new way point that is in a certain look-a-head distance. The logic that updates the waypoints is very simple. It checks how close the QGV is to the current way point. We call this distance *pp-update-waypoint*. If this distance is too small the controller becomes unstable and if it is too big the side track error becomes large. Usually this distance is selected in comparison with the resolution of the grid. In this experiment this distance is the same as the resolution of the grid, which is *5 cm*. When the QGV is in a 5 cm radius of the current waypoint the algorithm updates the next, to be tracked, way point. When the QGV reaches

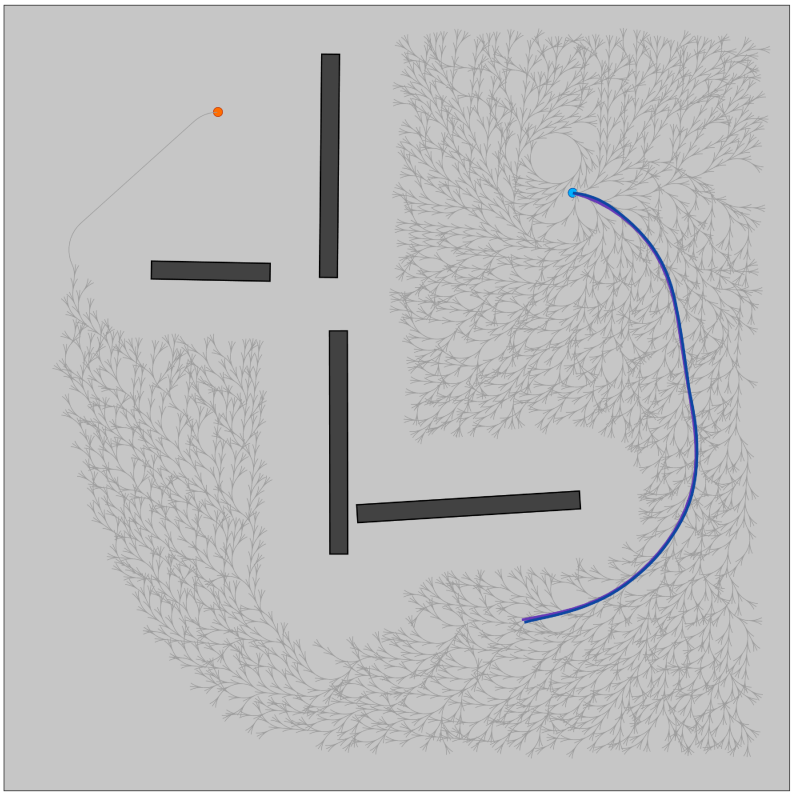


Figure 5.12: Dynamic Hybrid A* - configuration 1

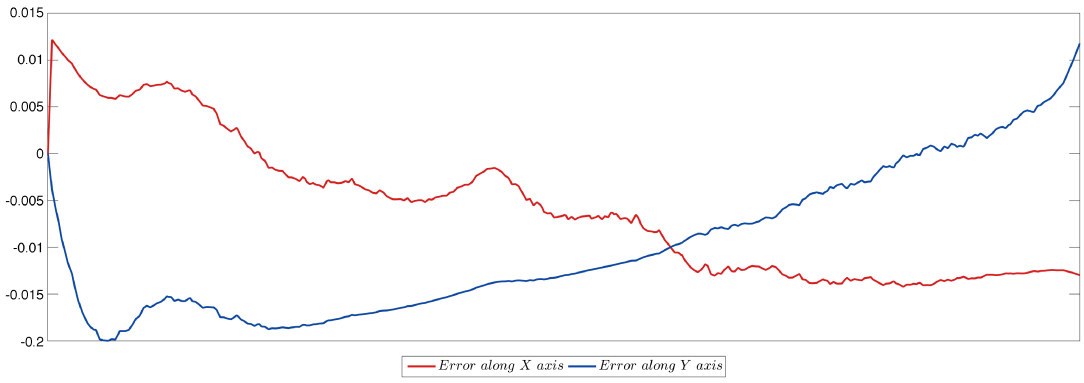


Figure 5.13: A depiction of cross-track errors along X & Y axes in (m) at configuration 1

the end of the path there is not more waypoints and it has to stop. That's why the tracked path doesn't quit reach the goal state but one could argue it is within the goal set!

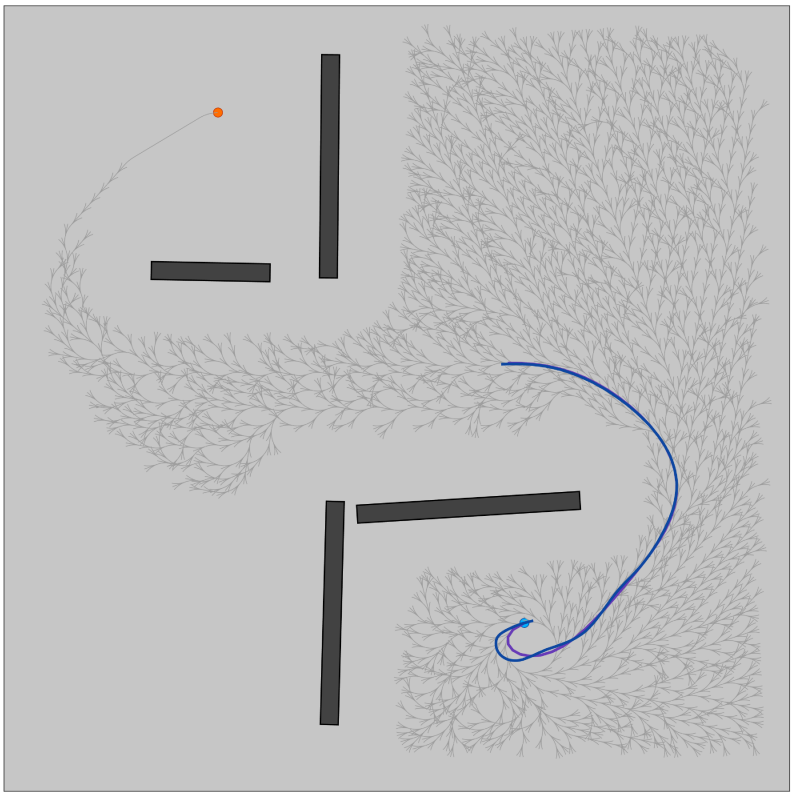


Figure 5.14: Dynamic Hybrid A* - configuration 2

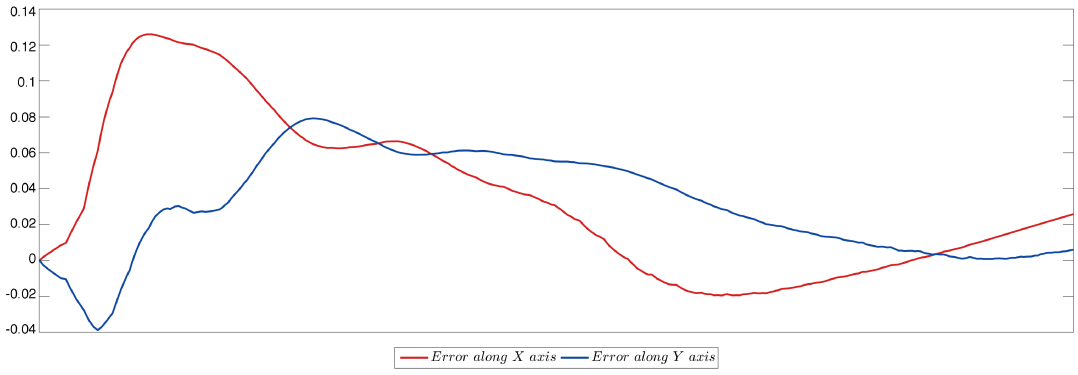


Figure 5.15: A depiction of cross-track errors along X & Y axes in (m) at configuration 2

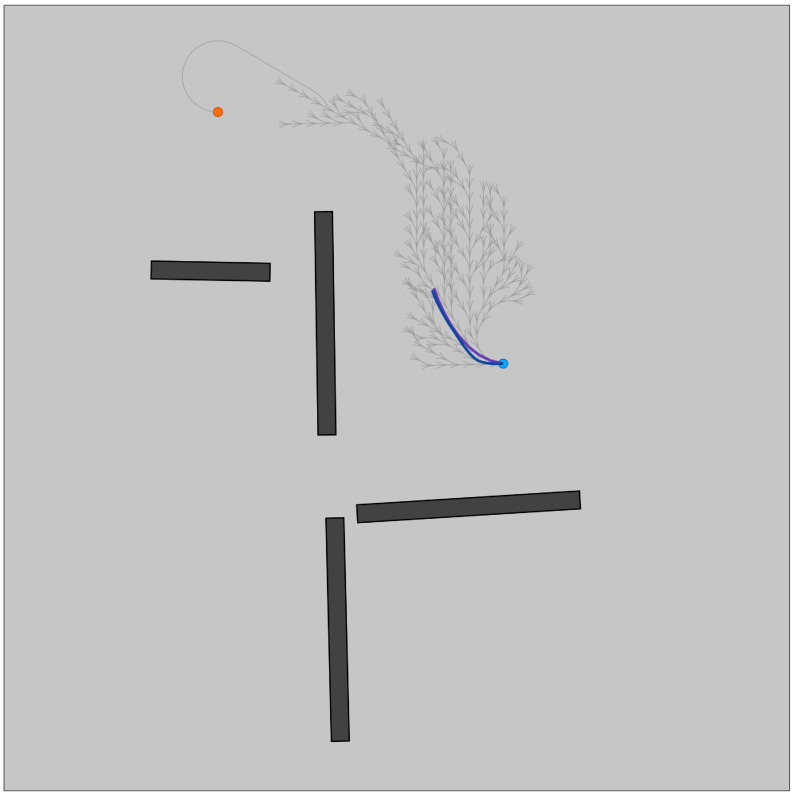


Figure 5.16: Dynamic Hybrid A* - configuration 3

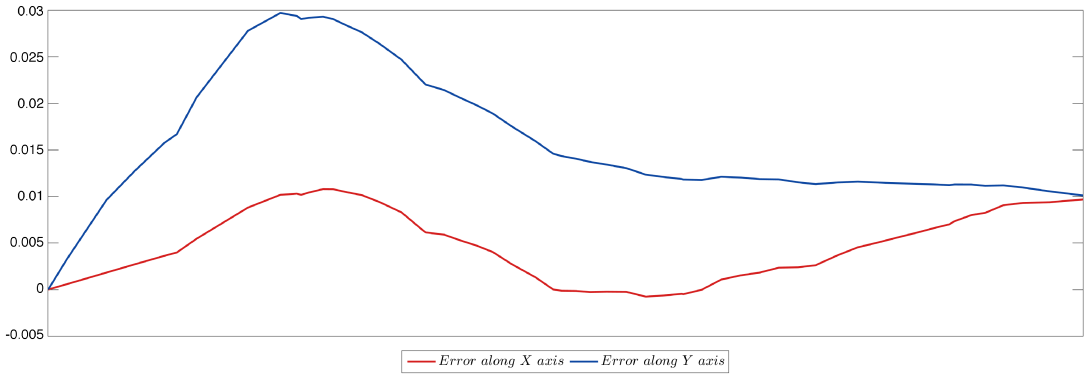


Figure 5.17: A depiction of cross-track errors along X & Y axes in (m) at configuration 3

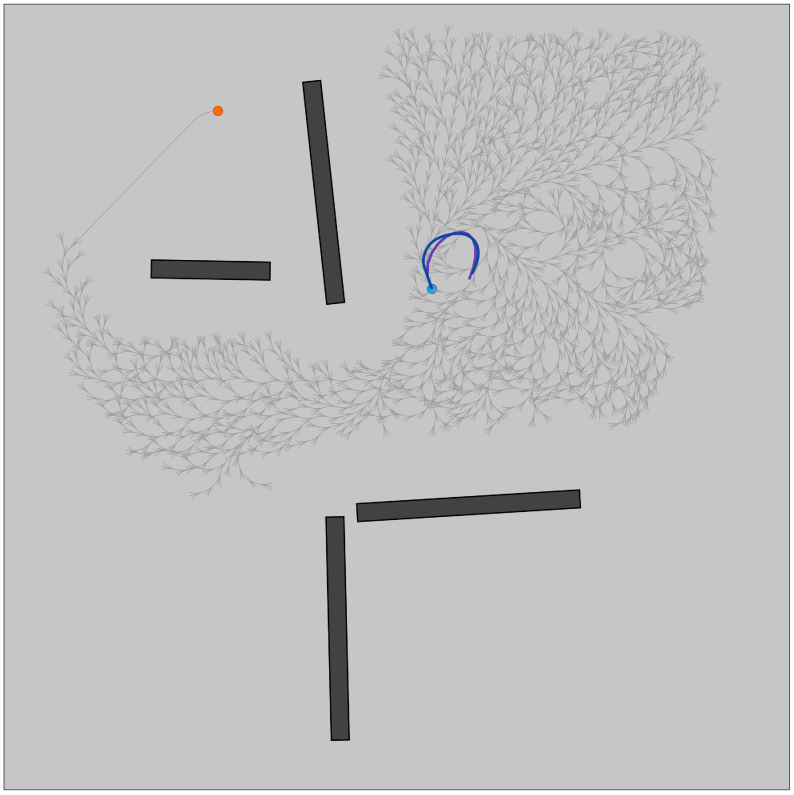


Figure 5.18: Dynamic Hybrid A* - configuration 4

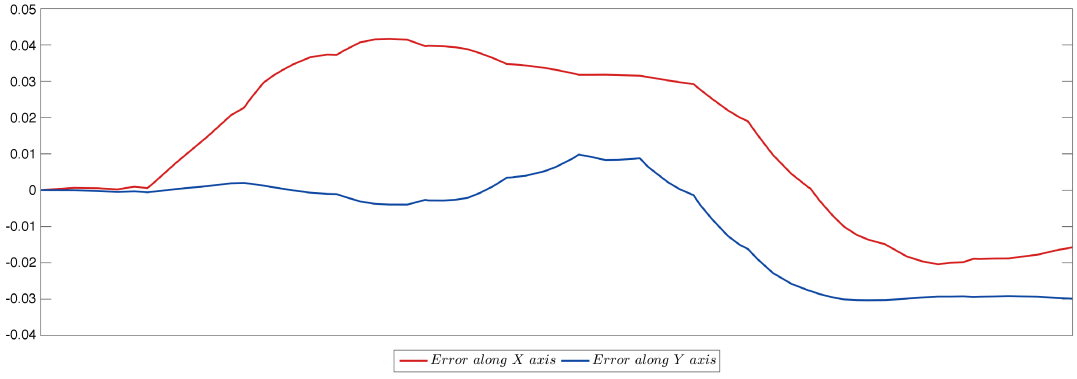


Figure 5.19: A depiction of cross-track errors along X & Y axes in (m) at configuration 4

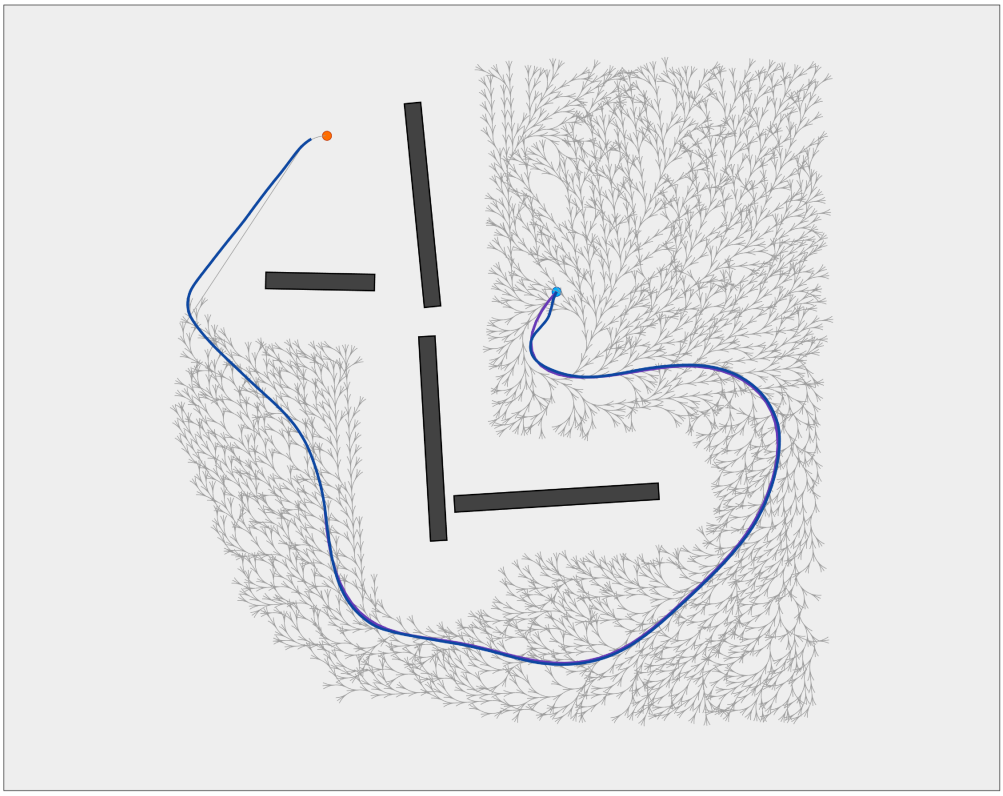


Figure 5.20: Dynamic Hybrid A* - configuration 5

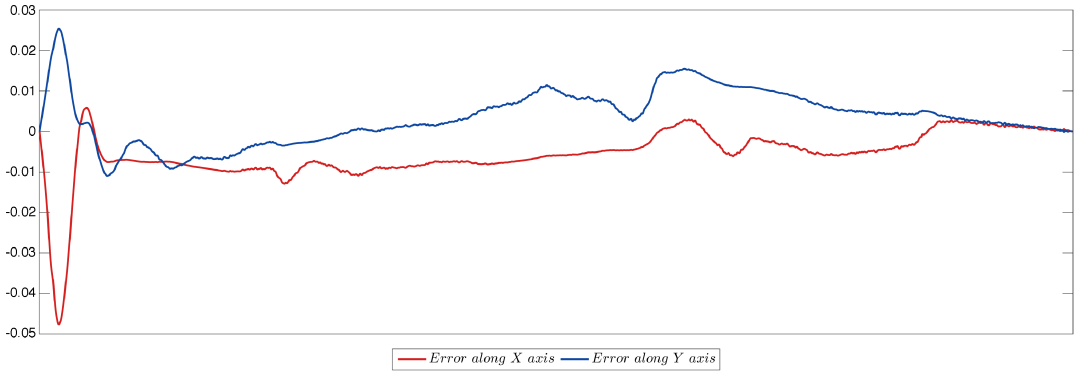


Figure 5.21: A depiction of cross-track errors along X & Y axes in (m) at configuration 5

Chapter 6

Conclusions & Future Works

6.1 Conclusions

The research in the area of path planning for mobile robots, car like and differential drive, will continue to grow based on the milestones achieved by the teams in the DARPA Urban Challenge and Grand challenge. There are many ways to solve the navigation problem based on the environment that the robot has to work under. In this work different approaches for path planning for car like robots were discussed and the positive and negative points of each algorithm was discussed. The implementation details of the Hybrid A* was discussed in details and the results were presented. The Hybrid A* is a fast planner that could be even used in dynamic environments. The algorithm was developed using MATLAB/Simulink and the Quarc real time package and it was tested on the QGV platform in an indoor environment.

The discussed planner addressed the problem of finding a smooth drivable path for car like robots. The algorithm can uses the non-holonomic nature of the system to create the action space and the constraints of the system are respected in all stages of the algorithm, node expansion, heuristic estimates and analytic expansion. This results in a smooth drivable path, which is the most significant characteristic of this algorithm. None of the other algorithms discussed in this work can generate such paths.

In order to provide a simulation environment for learning and teaching purposes a MATLAB toolbox was designed and the source code is freely available under `https://gitlab.com/AliAskari/HybridAStar`. The toolbox provides an easy interface to setup a configuration space and the user can apply multiple path planning algorithms on the configuration space.

6.2 Future works

6.2.1 Dynamic path planning

The trigger to start a new search was 20cm. This is a relatively high trigger specially in an indoor environment. The main problem introduced with smaller triggers are jerky maneuvers by the QGV. Each small movement of an obstacle will trigger the search and each search will generate a slightly different path. The next step to resolve such problems is to use a multi level path planner. The higher level path planner will always provide a smooth path and the lower level path planner will perform a search when an obstacle has moved. Then the result of the lower level path planner should be integrated with the previously generated path in the higher level path planner. This also allows the use of variable resolution grids. The higher level path planner can generate a grid where the resolution of the cells varies based on how close they are to the free space and the obstacles. This will allow a more smooth transition between generated paths when an obstacle moves in the work space.

6.2.2 Localization

In this work the localization problem was not discussed at all. It was a solved problem from the beginning. The set of OptiTrack cameras can be used to generate a perfect high resolution gird and configuration space. In the next step of the work on board sensors can be used to generate a map of the environment in real time and integrate the localization problem with the path planning problem.

6.2.3 Acceleration and velocity profiles

In both discrete and analytical node expansion it was assumed the system will always have a constant velocity and thus no acceleration. The generated paths also are suitable for a constant velocity along the path. The problem that arises is that the drive is not necessarily comfortable for the possible passengers in some part of the path, specially in sharp turns. An addition to hybrid A* algorithm would be to generate a velocity and acceleration profile based on the current geometric path while considering the forces felt by the passengers. This is an area where a lot of self driving car companies are spending their development and research efforts on.

Bibliography

[1] The darpa urban challenge. Accessed: 2015-09-01. URL: http://archive.darpa.mil/grandchallenge/. 2

[2] Google trends. Accessed: 2017-02-01. URL: https://trends.google.com. 2

[3] Optitrack motion capture system. Accessed: 2015-09-01. URL: http://optitrack.com/. 82

[4] Quarc real-time control software. Accessed: 2015-09-01. URL: http://www.quanser.com/Products/quarc. 79

[5] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005. ix, 11, 12, 13, 14, 15, 25, 26, 31, 37, 44

[6] D. C. Conner, A. A. Rizzi, and H. Choset. Composition of local potential functions for global robot control and navigation. *Proceedings of the IEEE/RSJ Intl. Conference on Intelligent Robots and Systems*, 2003. x, 33, 35

[7] D. C. Conner, A. A. Rizzi, and H. M. Choset. Construction and automated deployment of local potential functions for global robot control and navigation. 2003. 31

[8] C. I. Connolly and R. A. Grupen. Applications of harmonic functions to robotics. 1992. 32

[9] R. C. Coulter. Implementation of the pure pursuit path tracking algorithm. 1992. 91

[10] Canudas de Wit, Carlos, Siciliano, Bruno, Bastin, and Georges. *Theory of Robot Control*. Springer, 1996. 9, 10

[11] D. Dolgov and S. Thrun. Autonomous driving in semi-structured environments: Mapping and planning. pages 3407–3414, 2009. doi:10.1109/ROBOT.2009.5152682. 67

[12] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel. Practical search techniques in path planning for autonomous driving. 2008. x, 67, 69, 72, 85, 86

[13] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel. Path planning for autonomous vehicles in unknown semi-structured environments. *The International Journal of Robotics Research*, 29(5):485–501, 2010. doi:10.1177/0278364909359210. 67

[14] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. 79:47–516, 1957. 68

[15] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. 4:100107, 1968. 62

[16] M. A. Askari Hemmat, Z. X. Liu, and Y. M. Zhang. Real-time path planning for nonholonomic unmanned ground vehicles. 2017. Won best student paper award. 83

[17] LE. Kavraki, P. Svestka, JC. Latombe, and MH. Overmars. Probabilistic roadmaps for path planning in high-dimensionalconfiguration spaces. 12:566580, 1996. 62

[18] O. Khatib. The international journal of robotics research. *Journal of Field Robotics*, 5(1):90–98, 1986. 19

[19] D. E. Koditschek and E. Rimon. Robot navigation functions on manifolds with boundary. 11(4):412–442, 1990. 39

[20] B. H. Krogh. A generalized potential field approach to obstacle avoidance control. *SME Conf. Proc. Robotics Research: The Next Five Years and Beyond*, 1984. 31

[21] Y. Kuwata, J. Teoy, and Frazzoli E. et. al. Motion planning in complex environments using closed-loop prediction. 91

[22] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. ix, 5, 7, 8, 11, 12, 14, 16, 24, 33, 53, 54, 62

[23] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. 20:378400, 2001. 62

[24] C. Y. Lee. An algorithm for path connections and its applications. 3:346–365, 1961. 56

[25] S. R. Lindemann. *Smooth Feedback Planning*. PhD thesis, University of IllinoisUrbana, 2008. ix, xi, 3, 35, 92

[26] S. R. Lindemann and S. M. LaValle. Computing smooth feedback plans over cylindrical algebraic decompositions. 2008. 33

[27] S. R. Lindemann and S. M. LaValle. Simple and efficient algorithms for computing smooth, collision-free feedback laws over given cell decompositions. 28(5):600–621, 2009. 33

[28] J. W. Milnor. *Morse Theory*. Princeton University Press, 1963. 32

[29] M. Montemerlo, Becker, and S. Thrun. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008. 67

[30] S. Park, J. Deyst, and J. P. How. Performance and lyapunov stability of a nonlinear path-following guidance method. *Journal of Guidance, Control, and Dynamics*, 30(6):17181728, November 2007. 91

[31] C. W. Reynolds. Steering behaviors for autonomous characters. 1999. 91

[32] E. Rimon and D. E. Kodischek. Exact robot navigation using artificial potential functions. *IEEE Transactions on Robotics and Automation*, 8(5):501518, 1992. 31, 32, 34

[33] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. 1994. 28

[34] S. Thrun, M. Montemerlo, H. Dahlkamp, and et. al. Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, 23(9):661692, 2006. 91

[35] C. Urmson, C. Ragusa, and D. et. al. Ray. A robust approach to high-speed navigation for unrehearsed desert terrain. *Journal of Field Robotics*, 23(8):467508, 2006. 91