UNIVERSITY OF CALIFORNIA
SANTA CRUZ

LIDAR A*

An Online Visibility-Based Decomposition and Search Approach for Real-time Autonomous Vehicle Motion Planning

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Engineering

by

Po-Jen Wang

June 2020

The Thesis of Po-Jen Wang
is approved:

Prof. Roberto Manduchi, chair

Prof. Michael Wehner

Prof. Narges Norouzi

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Table of Contents

Experiments and Results

List of Figures

Abstract

LIDAR A*, An Online Visibility-Based Decomposition and Search Approach for Real-time Autonomous Vehicle Motion Planning

by

Po-Jen Wang

Motion planning is the task of finding a sequence of feasible motions for a robot to transform from an initial state to a goal state avoiding collisions. Modern motion planner algorithms for non-holonomic vehicles typically rely on graph search or sampling-based techniques. However, graph search methods quickly become ineffective as the computational complexity scale exponentially to map size. Sampling-based methods have the shortcomings of creating highly suboptimal paths or having high run-times. This thesis introduces a new hierarchical motion planner, LIDAR A*, that features an online visibility-based decomposition and search process. It incrementally analyzes regional maps with locally simulated 2D LIDAR scans to search for the shortest opening gateway sequence toward the goal. These sequential gateways are used to guide the simulation of a robust local motion planner to generate a smooth and kinematic friendly path. Experiments show this method significantly reduces the number of nodes minimizing computation time while generating a near-optimal path compared to the recently proposed RRT-based algorithms.

Acknowledgments

It is my pleasure to acknowledge those individuals who made the completion of this thesis possible. I would also like to recognize those who were instrumental to my robotics journey. First of all, I would like to express my deepest appreciation to my supervisor, Professor Roberto Manduchi at U.C. Santa Cruz, who gave me the opportunity to pursue the topic and explore new innovative solutions to resolve the challenging problems in robotics motion planning. His immense knowledge, guidance, and insightful inputs has upscaled the quality of the research, and helped overcoming obstacles and obtaining impactful results. In addition to the committee chair, I would like to thank the committee members: Professor Michael Wehner and Professor Narges Norouzi for helping me construct a strong foundation in robotics and AI, and providing continuous support and generous assistance during the thesis research.

My deep gratitude also extends to my family — my parents, Yun-Long Wang and Li-Hon Ko; my grandma Lan-Xin Fan; my sisters, Li-Jen Wang and Kuan-Jen Wang; and my brothers-in-law, Sertac Cakici and Jack Lee. Their unconditional love, commitment, and joint effort in supporting my education and helping the family, especially in some of the most difficult times, has helped me physically and spiritually through every step of my career. I wouldn’t be able to complete this thesis without their hard work and dedication. Finally, I wish to sincerely acknowledge my undergraduate advisor, Professor C.T. Lin, who previously taught robotics at

California State University, Northridge. Professor Lin introduced me and sparked my interest in autonomous driving technologies that later turned into an award-winning project and shaped my career path. I’m extremely grateful for his inspiration and long lasting influence in my life.

Chapter 1

Introduction

1.1 History of Autonomous Vehicles

Grey Walter was a well-known robotics pioneer and the creator of some of the first ever electronic autonomous robots. He constructed his robot *Tortoises* (Figure 1, 2) in the late 1940s and described their abilities to sense light, hunger, touch, dance a jig, and feed themselves by returning to a charging station[1]. These mobile machines simulated the cognitive process of living creatures using only analog circuitries, demonstrating the first level of artificial intelligence. Grey Walter's work was considered a milestone in the history of autonomous vehicles and has inspired numerous researchers in building more advanced robotics autonomy technologies[1].

Today modern autonomous mobile robots are tasked with handling more complicated missions; for example, warehouse robots must autonomously transport materials to the designated area with high precision[2]. Planetary exploration rovers are tasked with safely traversing through rugged terrains avoiding dangers and collecting valuable information[3]. To further accelerate the developments of self-driving technologies, United States's Defense Advanced Research Projects Agency (DARPA) created the famous DARPA Grand Challenge in 2004, 2005, and the DARPA Urban Challenge in 2007. This was to challenge research organizations to

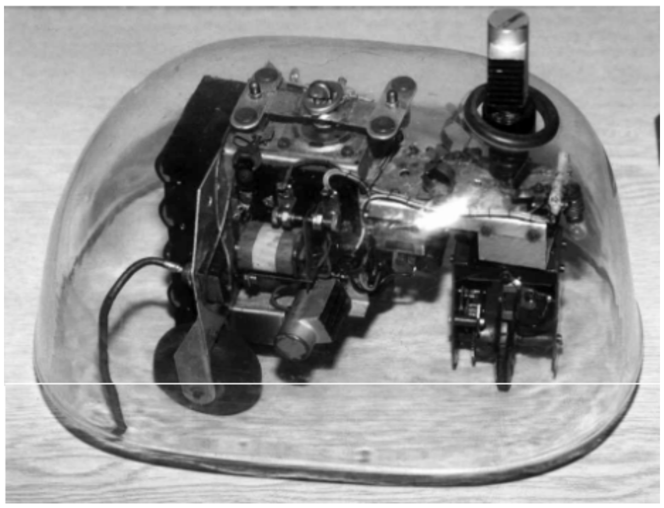

Figure 1. Grey Walter's Tortoises Robot. On display in the Science Museum, London

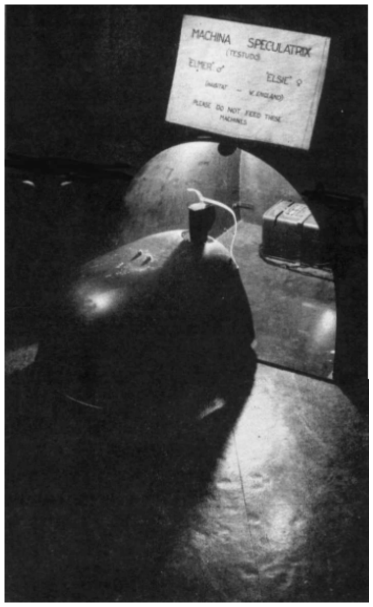

Figure 2. Tortoise return to recharge battery

build vehicles capable of autonomous navigation through a 142-mile off-road route in the desert as well as self-driving in an urban environment following traffic regulations[4, 5]. According to the U.S. National Highway Traffic Safety Administration (NHTSA), motor vehicle crashes cost $836 billion in economic activity, loss of life, or injuries in 2010. 94% of these serious crashes were due to human errors which can be eliminated with automation[6]. Another study shows self-driving can potentially save up to 50 minutes each day in work commute or accessing entertainment[7]. The advancement in self-driving technologies can bring significant benefits to the society in terms of safety, economy, and mobility[6].

The recent race to full-autonomy self-driving cars between auto-makers and tech giants has set timelines of delivering the self-driving systems. Some predict autonomous vehicles will be safe and reliable by 2025, while shared autonomous vehicles and rides may become common in the 2030s and 2040s[8].

1.2 Problem Statement

One of the important topics in autonomous driving is the path and motion planning of the vehicle. Motion planning, in a broad sense, is the task of finding a sequential configuration that transforms a robotic system, such as robotic arm, mobile robot, or Unmanned Aerial Vehicle (UAVs), from a start state to a goal state in the workspace[9]. Motion planning for autonomous mobile robots has to deal with trajectory planning and obstacle avoidance in the configuration space. It is a classic and thoroughly researched topic in robotics, which a number of effective and diverse approaches have been proposed in the last few decades. The optimality and robustness of these motion planning methods can be evaluated based on three major metrics: path quality, intelligence level, and computational efficiency.

1.2.1 Path Quality

Robotic systems can exhibit different levels of motion constraints based on the kinematic model of the system. For example, in the previously discussed non-holonomic mobile robot applications, planetary rovers and self-driving cars must travel in segmented sequences of circular paths at varying radii in which motions toward the vehicles' side directions cannot be performed. UAV systems, on the other hand, do not have such limitations thanks to extra degrees of freedom in the system. Therefore, the ideal mobile robot motion planner should generate an optimal path that is a smooth and continuous curved path that closely matches the reachable sequence of configurations based on the vehicle's kinematic model, reducing control errors caused by kinematic constraints when following the path. This optimal path, at the same time, should also minimize path length to reduce the cost traversing toward the goal while maintaining safe clearances to surrounding obstacles. Paths built with segmented straight lines and sharp angles are less desirable for mobile robots to follow and are considered suboptimal for robotics motion planning.

1.2.2 Intelligence Level

Another important metric in evaluating motion planning methods is the algorithm's intelligence level in handling complex obstacle arrangements. A robust motion planning algorithm must be able to deal with a variety of obstacle arrangements at different levels of difficulty. Examples of simple obstacle avoidance tasks may include lane following with lightly curved path or avoiding a small number of obstacles in open areas. At an intermediate level of difficulty, robots may be required to navigate through a large number of random obstacles, zigzag switchbacks, or avoid local minimum situations. In the most challenging scenarios, robots may be tasked to perform the following: finding a path within a large complicated maze with multiple passages lead to dead-ends; finding and entering a small choke point that leads to the goal; maneuvering through zigzag switchbacks with very limited obstacle clearance; or having a combination of the above troublesome scenarios mentioned above.

1.2.3 Computational Efficiency

The last metric used in evaluating motion planning algorithms is their computational efficiency. Autonomous vehicles operating in dynamic environments such as self-driving cars can navigate at high speeds. Their abilities to correctly react to emergency situations not only depend on the intelligence of the system but also how fast a decision or an emergency maneuver can be executed. Therefore, it is crucial for a motion planning algorithm to be highly computationally efficient that allows real-time operation on the selected system platform. In most cases, the ideal computation time should be in the millisecond range. For local motion planners, the computational efficiency of the method will also affect the length of the calculated path which represents how far a robotic vehicle can foresee and plan ahead to handle future problems. Frequently, trade-offs between speed and path quality are carefully considered and taken to satisfy their minimum requirements for the application.

In summary, this thesis aims to solve the classic motion planning problem which, given an autonomous vehicle's kinematic model and start configuration of on a large obstacle map, finds a motion feasible path in real-time that can sequentially translate the vehicle to reach a goal destination avoiding obstacle collision on all parts of the vehicle. The proposed algorithm will be tested and compared to some of the recent notable motion planning algorithms based on their path qualities, intelligence levels, and computational efficiencies.

1.3 Thesis Structure

This thesis will begin with the introduction of the important robotics concepts related to autonomous robots such as map representation and configuration spaces, followed by literature review of the notable path and motion planning algorithms. The thesis then presents the proposed methodology of LIDAR A* and validates this method with experimental results. It finishes with a discussion of findings and proposes future work for improvements and possible extensions.

1.4 Map Representation

Two major frameworks frequently used in map representation are metric and topological frameworks[10, 11]. In metric framework, environment objects are described with fine-grained precise coordinates. The commonly used two-dimensional obstacle grid map is an example of using this framework. On the other hand, topological framework uses a graph to describe the relations between significant landmarks in the environment. In this representation, graph nodes are significant landmarks and edges indicate paths and distances between these landmarks[10].

1.5 Configuration Space

Configuration space is a key concept in robotics motion planning. Perez established and extensively used this concept in solving spatial motion planning problems in the 1980s[12, 13, 14]. A configuration q is defined to be a state of a system in the workspace. A configuration space C is a space or a subset of workspace that includes all possible configurations $A(q)$ in the workspace based on the kinematic model of the robotic system. When obstacles are present, the configuration space C can be further divided into obstacle space $C_{obstacle}$ and free space C_{free}. Obstacle space $C_{obstacle}$ is the region where parts of the robot would collide with obstacles, and free space C_{free} is the region where the robot is free of collision which $C_{free} = C - C_{obstacle}$.

In the following example (Figure 3), assume the robot has a rectangular shape and can move in horizontal and vertical directions (x, y) but can not perform rotation, obstacle space $C_{obstacle}$ (gray and black areas) and free space C_{free} (white areas) can be determined by wrapping the robot's contour at static state around the obstacle's contour. When the robot's motion has a higher degree of freedom such as (x, y, θ) or with accelerations, or when vehicle shape is complex, precise boundary of obstacle space $C_{obstacle}$ and free space C_{free} can be difficult to compute[13]. Using the static state of the robot to determine obstacle space $C_{obstacle}$ and free space C_{free} can significantly simplify the motion planning problem, though it is not a true representation of actual obstacle and free space regions[13].

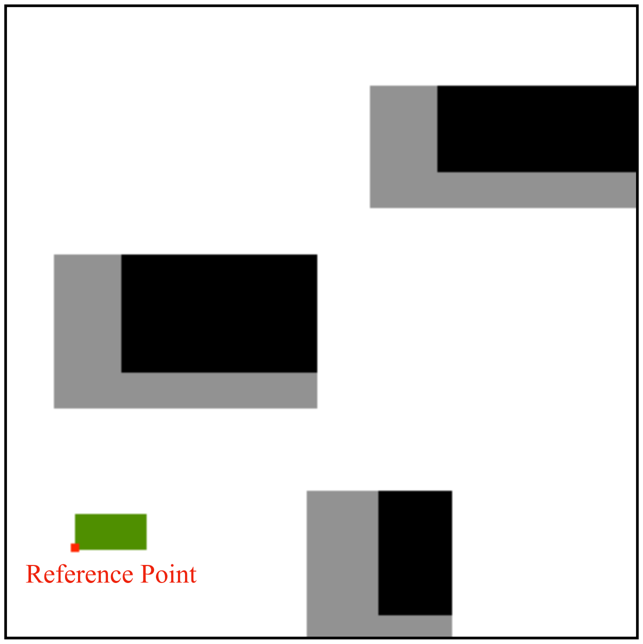

Figure 3. Configuration Spaces. Obstacle Space (Black and Gray) and Free Space (White Area)

Chapter 2

Literature Review

2.1 Notable Path and Motion Planning Methods

Motion planning can be applicable to numerous domains such as robotic arms, mobile robots, UAVs, or video game agents. It is a widely researched topic which a variety of different approaches have already been proposed. Each approach has its own strengths and weaknesses, and may perform better or worse in certain domains. This section is devoted to survey the popular and influential algorithms that are related to mobile robot motion planning, starting with the classic methods followed by more modern approaches. These methods are grouped into similar approaches.

2.2 Geometric-Based Methods

2.2.1 Visibility Graph

Visibility graph was studied by Nilsson, Perez, and Wesley[14, 32]. It is an important property in solving computational geometry problems. When obstacles on a planar map are given in the forms of polygons, a visibility graph can be constructed by checking and adding edges between all possible pairs of obstacle polygon vertices and the start and goal vertices when these are directly insight with each other without getting blocked by obstacles. Perez applied graph search on visibility graphs to find the shortest path between a start vertex and a goal vertex, and shifted the resulting shortest path vertex positions away from their obstacle positions to accommodate for the robot's size[14]. This approach was used on the Shakey robot project at Stanford Research Institute (SRI) in the 1960s[15]. A more optimized Reduced Visibility Graph was later introduced that removes edges going toward instead of around obstacles which makes path planning more efficient than the original visibility graph[29].

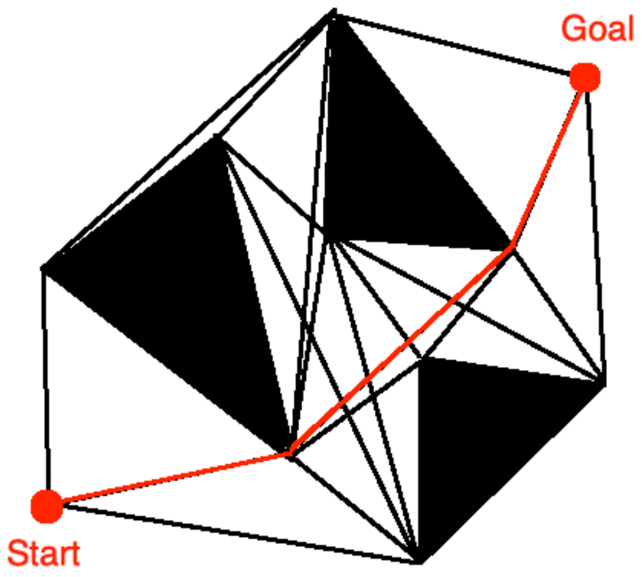

Figure 4. Path Planning with Visibility Graph

2.2.2 Voronoi Diagram

A class of geometric motion planning algorithms is based on the generalized Voronoi diagram (GVD). Voronoi diagram has the property of subdividing a map into regions of Voronoi regions, where each point within a Voronoi region is closest to one of the given set of sites which the region encloses as shown in Figure 5[39]. Common boundaries of Voronoi regions are the sequential points that are equidistant to two sites on the map that maximize the obstacle distances. Path planning algorithms based on Voronoi diagrams construct a graph along the Voronoi region common boundaries and apply a search to determine an obstacle free path that maximizes obstacle distances[32].

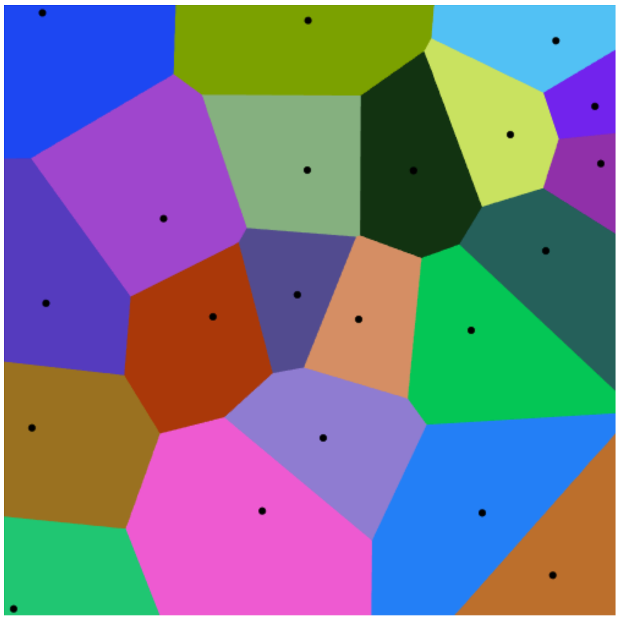

Figure 5. Voronoi Diagram. Voronoi diagram partitions the map into Voronoi regions given a set of sites. The common regional boundaries are constructed into a graph for path planning

2.2.3 Cell Decomposition

Another class of geometric-based methods for motion planning can be generalized as a cell decomposition approach that can be traced back to Brooks and Perez in 1985[32, 38]. When polygon obstacles are given on a planar map, the obstacle polygon vertices are used to subdivide the map into smaller obstacle free regions. These obstacle free regions are constructed into a graph which a graph search algorithm is applied for finding consecutive neighboring regions that contain the start and goal locations[32, 38]. An example of cell decomposition method is the Trapezoidal Cell Decomposition that extends horizontal or vertical lines on the obstacle vertices to form regional common boundaries as shown in Figure 6. This decomposition method is a type of exact cell decomposition. Hierarchical Path-Finding A* or HPA* also uses cell decomposition for its map abstraction process and will be discussed in section 2.4.3.

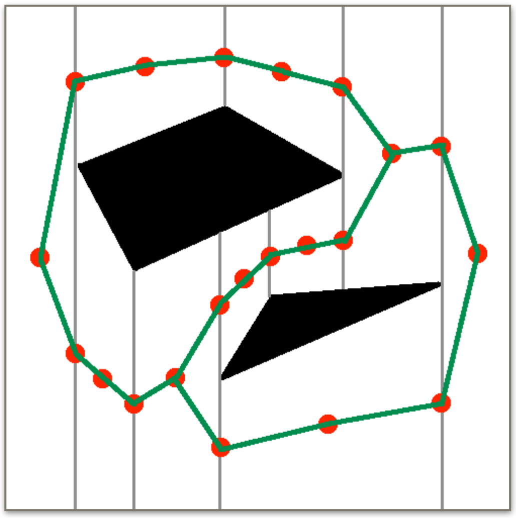
Figure 6. Trapezoidal Cell Decomposition. A roadmap determined by Trapezoidal Cell Decomposition

2.3 Artificial Potential Field Methods

Artificial Potential Field is a class of motion planning algorithm that is based on generating a potential field map. The map is built by inserting attractive forces at the goal location and repulsive forces at the obstacle locations in the workspace[42]. The potential field map can guide a robot toward its goal location by summing the attractive and repulsive forces at each cell of the grid map. The strength of attractive and repulsive forces are inversely proportional to its distances[42].

2.3.1 Vector Field Histogram

Vector Field Histogram (VFH) is a fast online local obstacle avoidance algorithm introduced by Borenstein and Koren[16]. It finds a desired direction or vector to travel using real-time data collected from onboard ultrasonic or LIDAR sensors. The algorithm first constructs a high level two-dimensional Cartesian histogram grid map that is always centered at the robot's position. The values of cells are continuously updated by adding repulsive forces to cells where an obstacle is present from sensor readings. At the same time, an attractive force is applied at the goal location. The algorithm then divides the two-dimensional Cartesian histogram grid map into a number of angular sectors whose obstacle densities are computed using the accumulated cell values inside the sector regions. Finally, an angular sector with low obstacle density is selected as the ideal heading vector for motion output.

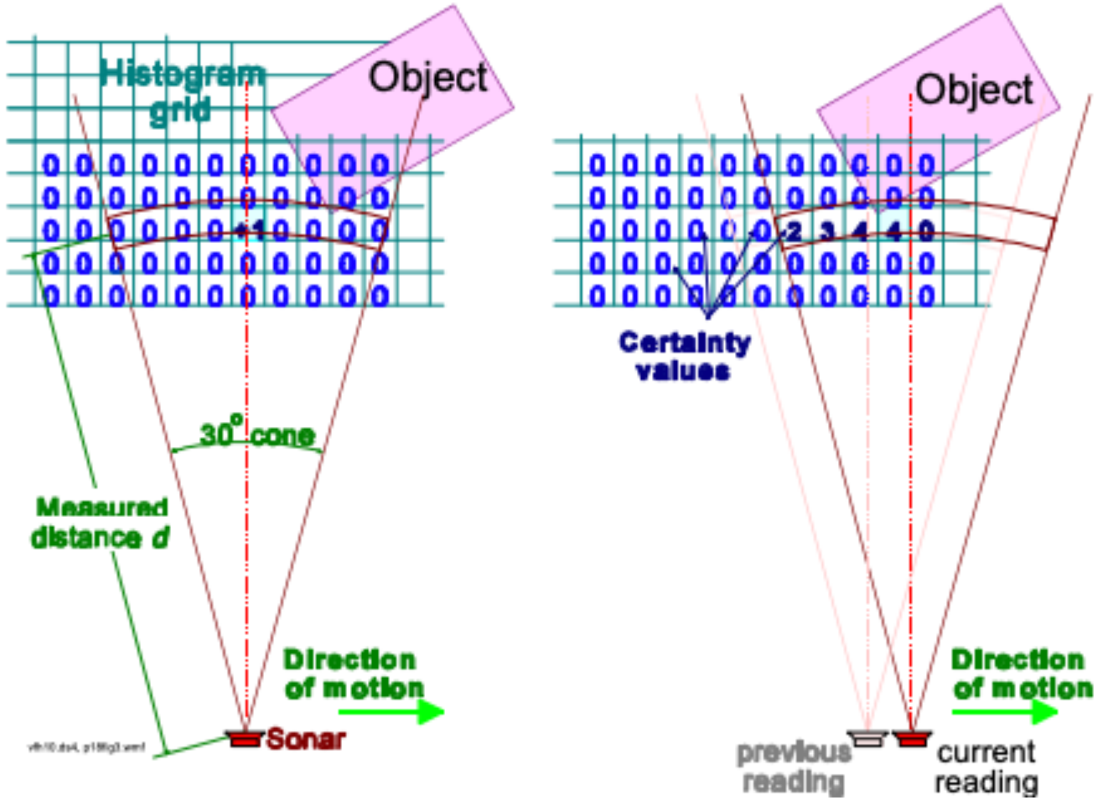

Figure 7. Vector Field Histogram adds repulsive forces to the histogram grid cells when obstacle is present

2.4 Graph Search-Based Methods

2.4.1 Dijkstra's Shortest Path

Dijkstra's shortest path algorithm is a well known algorithm for finding the shortest path between two nodes in a weighted graph. It was invented and published by Dutch computer scientist Edsger Dijkstra in 1959[17]. Dijkstra's algorithm is also known for being a single-source shortest path algorithm which simultaneously computes the shortest path to every other remaining vertex in the graph with a given source vertex[18]. Dijkstra's algorithm will keep track of lists of visited and unvisited vertices in the graph; and for every vertex, it would also track the current shortest distance from the source vertex and last vertex it previously visited (parent vertex).

Dijkstra's algorithm begins by assigning the distance of the source vertex to 0 and distances of all other vertices to infinity. Then, starting from the source vertex, it selects and visits one of the unvisited neighboring vertices with the shortest known distance from the source vertex. This vertex is then moved from the unvisited vertex list to the visited vertex list and will not be visited again. At this current vertex, the algorithm calculates the distances of all of its unvisited neighbors from the source vertex. At every unvisited neighboring vertex, if the calculated distance is smaller than the known shortest distance, it updates the known shortest distance and the parent vertex. This process is repeated until all vertices are visited.

2.4.2 A* Search and Heuristic

A* or A-Star search algorithm is another popular shortest path search algorithm developed by Hard, Nilsson, and Raphael as part of the Shakey project at Stanford Research Institute (SRI) in 1968[15, 19]. A* search algorithm can be seen as an optimized version of the Dijkstra's algorithm[20]. It also searches for the shortest path from a start vertex to a goal vertex in a weighted graph, but introduces heuristics to guide the search process that significantly reduces the number of vertices visited and improves computational performance[19, 20].

Heuristic is a technique that is commonly used in solving self-exploratory problems such as graph search. By using known information or previous experience to perform informed search to find a solution, the search process intelligently avoids searching through the unfavorable regions in the search space, thus rapidly speeding up the search process[19]. Due to applying educated guesses, using heuristics in a search problem means that the algorithm can no longer guarantee an optimal solution but will likely generate a satisfactory solution at a lower run-time[21].

Unlike Dijkstra's shortest path algorithm that keeps track of the shortest distances from the source vertex to every remaining vertex in the graph, A* search algorithm calculates a total cost *f(x)* at each of the current neighboring vertex during the search process. The total cost *f(x)* is equal to the sum of the vertex cost *g(x)* and a heuristic cost *h(x),* which is an estimated future cost for reaching the goal. The euclidean (straight line) distance toward the goal is a commonly used heuristic for path planning problems. Other heuristics such as Manhattan distance is also a popular option. The total cost *f(x)* at neighboring vertices is then used to decide which vertex to traverse next in the search process.

The algorithm of A* search is summarized as follow: first, A* search algorithm will initialize two lists of open vertices and closed vertices and calculate the heuristic value of the start vertex. Beginning at the start vertex, its total cost *f(x)* is calculated with a distance cost *g(x)* equal to 0. Then for each neighboring vertex, if the vertex is neither in the open vertex list nor in the closed vertex list, it is added to the open vertex list. The total cost *f(x)* at each neighboring vertex is also calculated by summing the vertex distance cost and the heuristic cost.

$$f(x) = g(x) + h(x)$$

If the calculated total cost *f(x)* is smaller than the known vertex total cost *f(x)*, the known total cost *f(x)* is updated and its parent vertex is set to the current vertex. After all neighboring vertices are analyzed, the current vertex is added to the closed vertex list and the algorithm traverses to the open vertex with the least total cost *f(x)* by setting this vertex to be the current vertex. This process is repeated until the current vertex is the goal vertex, which means the goal has been reached.

2.4.3 HPA* and Hierarchical Graph

When dealing with a large complicated graph, a technique to reduce graph complexity, improve graph clarity and efficiency is to perform hierarchical decomposition to get a hierarchical graph or AH-graph[22]. In a hierarchical graph, a group of elements in a lower hierarchical level graph are abstracted as an element of a graph at a higher hierarchical level maintaining element relationship[22]. A hierarchical graph can have multiple levels. It uses hierarchy to represent the same environment at different levels of details and can potentially reduce exponential complexity problems to linear ones[22, 23].

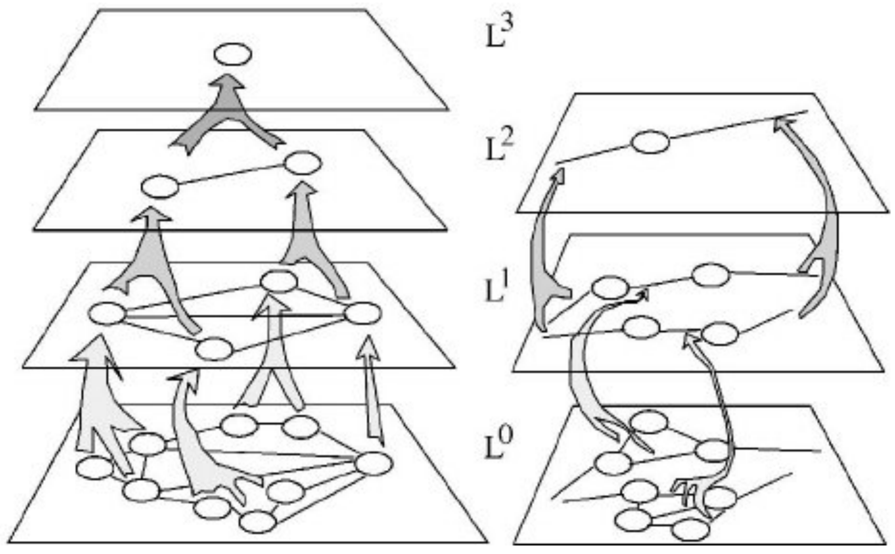

Figure 8. Two Examples of Hierarchical Graph. Node abstraction on the left and edge abstraction on the right

HPA* or Hierarchical Path-Finding A* is an algorithm developed by Botea and Muller that utilizes hierarchical approach to reduce search complexity in path planning problems on a grid-based map[24]. The algorithm decomposes the map and constructs a higher hierarchical level graph called *abstract graph* with the following process: it first abstracts the original grid map into a number of equally sized square sub-grid called *clusters* (Figure 7). The common boundary between two adjacent clusters is called *boarder*. Each cluster has four boarders at the top, right, bottom, and left positions. Within a common boarder, *entrances* are identified when there is an acceptable obstacle free segment that may be adjacent to an obstacle, or an obstacle free corner of the cluster. For each cluster, depending on the size of the entrance, one or two nodes per entrance are inserted into the abstract graph at their respective locations. Then they are connected to their respective pair of nodes which is on the adjacent cluster that shares the common boarder. These connections that link across adjacent clusters are called *inter-edge* and their lengths are set to 1. For every possible pair of nodes at the four boarders within a cluster, the algorithm searches their optimal path inside the cluster. When a path is found, an edge called *intra-edge* is inserted to the abstract graph with a length equal to the path distance. When no path can be found, the pair of entrances remain unconnected.

The above procedures complete the construction of the abstract graph. This graph has fewer nodes compared to the original grid map and is usually pre-processed. The online search for the shortest path is usually performed using

Dijkstra's or A* algorithms on the abstract graph to calculate the final shortest path. Hierarchical Path-Finding is popular in the video game community. It is likely due to its fast run-time on the simplified abstract graph and that in-game static map can be easily pre-processed. Its use on partially dynamic maps can be performed by reprocessing individual cluster connections when obstacle information inside the cluster is changed. The following images (Figure 9) show the construction and search of an abstract graph in HPA*[25].

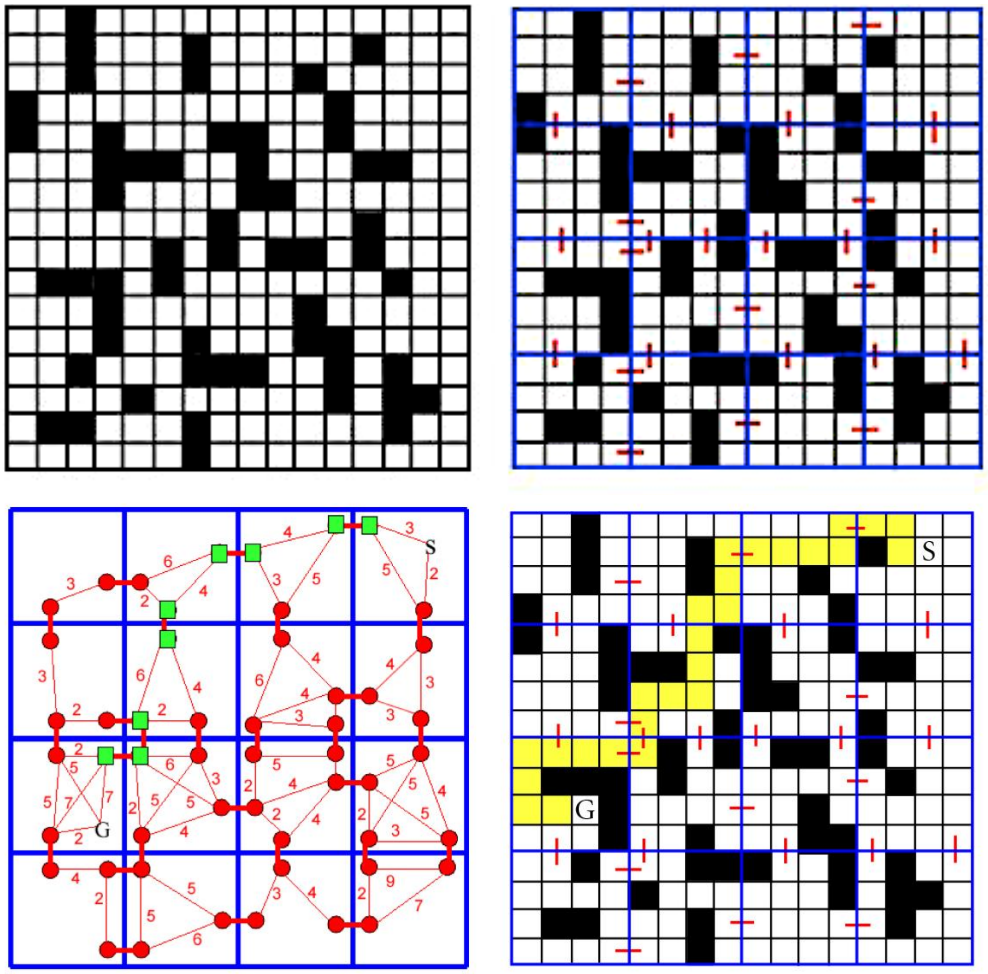

Figure 9. HPA* Map Abstraction into Clusters and Inter-Edges. Shortest Path Search on Abstract Graph

2.4.4 Hybrid A*

Hybrid A* algorithm is another important variant of the original A* algorithm developed by Dolgov, Thrun, Montemerlo, and Diebel[26, 27]. It was specifically designed for path and motion planning of non-holonomic autonomous vehicles. The algorithm was the primary approach implemented on Stanford Racing Team's self-driving car *Junior* and was validated on their successful entry at the 2007 DARPA Urban Challenge. Hybrid A* follows the general concept of A* search algorithm that expands search tree by identifying open nodes and traversing to the one with the least total cost *f(x)*. Major improvement over the original algorithm is that nodes in Hybrid A* are based on the robotic car's reachable continuous states (x, y, θ) simulated using the vehicle's kinematic model. It avoids using discrete cell positions on a grid map as shown in Figure 10. The resulting path is a continuous curved path that matches the motion of the vehicle, compared to the segmented straight line path generated from A* search algorithm. Hybrid A* expands forward and backward nodes with varying steering angles with extra penalty for backward motion.

The search algorithm uses two heuristics. The first one is referred to as *non-holonomic-without-obstacles.* It is a cost pre-computed offline by setting a goal state (x, y, θ) of (0, 0, 0) and calculating the shortest path from every possible locations within a fixed neighboring region of this goal state assuming no obstacle is present. This heuristic will eliminate search branches that approach the goal with invalid angles. The second heuristic cost is based on a more traditional shortest path search

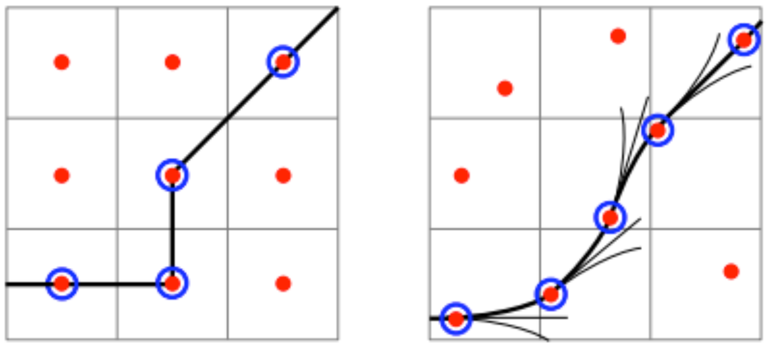
Figure 10. State Representation of A* (Left) and Hybrid A* (Right)

algorithm on a grid-based obstacle map ignoring vehicle's kinematic constraints and guiding the vehicle toward the goal. This heuristic is computed in real-time and the method was not clearly identified in the paper. Possible options for the search algorithm are Dijkstra's, A*, or flowfield.

Hybrid A* uses Voronoi diagram as the graph's cost function which is defined as follows:

$$\rho_{\mathcal{V}}(x,y) = \left(\frac{\alpha}{\alpha + d_{\mathcal{O}}(x,y)}\right)\left(\frac{d_{\mathcal{V}}(x,y)}{d_{\mathcal{O}}(x,y) + d_{\mathcal{V}}(x,y)}\right) \frac{(d_{\mathcal{O}} - d_{\mathcal{O}}^{max})^2}{(d_{\mathcal{O}}^{max})^2},$$

Where d_O and d_V are the nearest obstacle distance and edge of the Generalized Voronoi Diagram (GVD). The Voronoi Field is computed online with an obstacle map as input.

It was observed that a number of community's implementations of Hybrid A* algorithm use only three possible forward steering angles, maximum left, straight, and maximum right, in generating forward nodes. After the search is completed, due to not using a high resolution of steering angles in generating the vehicle's subsequent state branches, the immediate resulting path of Hybrid A* is usually suboptimal that includes unnecessary steering. Stanford's team applied post-smoothing using non-linear optimization on the output nodes and used non-parametric interpolation to add new intermediate nodes to smooth out the path.

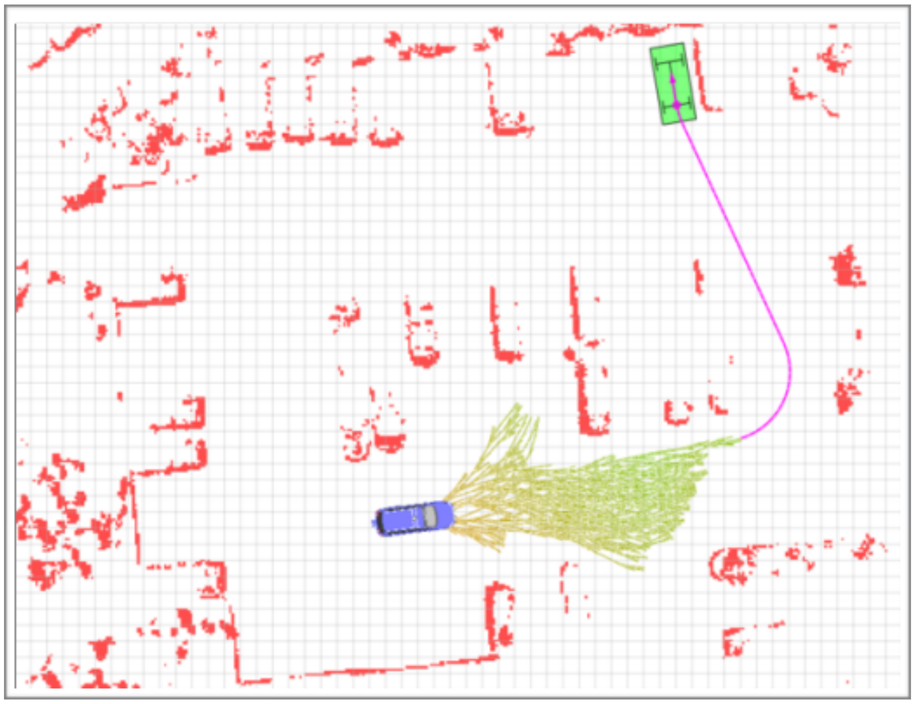
Figure 11. Hybrid A* Search Tree Expansion

2.5 Sampling-Based Methods

Sampling-based algorithms is another class of solutions that is widely applied to solve robotic motion planning problems including robotic arm and mobile robot. This method characteristically finds a path by generating random sample nodes in the search space and attempts to connect it to the current built tree that leads to the goal[30]. In some methods of this type, such as informed RRT* discussed in section 2.5.3, random node generation may be biased toward a region of the search space, similar to using a heuristic in A* search algorithm.

2.5.1 Rapidly Exploring Random Tree (RRT)

Rapidly Exploring Random Tree or RRT is a classic sampling-based path planning algorithm developed by LaValle and Kuffner in 1998[28, 34]. RRT grows a tree by generating a random node, which is a state q of configuration, from the search space and finds the nearest node from the current built tree. A possible new node is calculated at a position that moves the nearest node toward the random node's direction by a set distance. If no obstacle obstruction is found in between the two nodes, the new node is inserted (with an edge) to the current built tree. This process is repeated until a tree branch reaches the goal. Though RRT can rapidly explore the search space and converge to a path, the solution is usually not optimal[34]. The resulting solution paths may also be inconsistent between calculations because of the algorithm's random nature.

Algorithm RRT

```
 1: function RRT ($q_{init}$, $q_{goal}$)
 2:     $G$.init($q_{init}$)
 3:     while $q_{new} \neq q_{goal}$ do
 4:         $q_{rand} \leftarrow$ randomNode()
 5:         $q_{near} \leftarrow$ neartestNode($q_{rand}$)
 6:         $q_{new} \leftarrow$ newNodePos($q_{near}$ $q_{rand}$)
 7:
 8:         if collisionCheck($q_{new}$) is false
 9:             $G$.addNode($q_{new}$)
10:             $G$.addEdge($q_{near}$, $q_{new}$)
11:         end if
12:     return $G$
```

2.5.2 RRT*

RRT* is a powerful improved variant of RRT developed and popularized by Karaman and Emilio in 2011 that converges toward a more optimal path compared to the original RRT algorithm[30, 34]. RRT* inherits the concept of RRT with addition of two new features that simplify the tree structure while growing tree nodes. First, RRT* records the distance from the root to each node on the tree and this distance is considered as the cost of the node. When a new node is determined from the nearest tree node, neighboring nodes within a circular radius of the new node are analyzed to determine the new node's potential parent. RRT* calculates and picks the node with the least cost as the parent and connects the new node to the parent. The next step will again examine the neighboring nodes. Neighboring nodes are checked whether they would have a lower cost when reconnected to the new node instead of their original parents. When lower cost is found, the nodes are rewired to the new node. The select parent and rewire processes reorganize the tree nodes every time a new node is sampled and inserted, ensuring shortest paths while growing the tree. RRT* generates smoother tree branches and finds a more optimal path toward the goal compared to the original RRT though it takes longer time to converge[30, 34]. Once an initial solution is found, continuing to run the algorithm will add more nodes filling out the search space. It gradually optimizes the path and theoretically reaches the shortest possible path if the run-time is infinite[30, 34].

Algorithm RRT*

```
 1: function RRT ($q_{init}$, $q_{goal}$)
 2:     $G$.init($q_{init}$)
 3:     while $q_{new} \neq q_{goal}$ do
 4:         $q_{rand} \leftarrow$ randomNode()
 5:         $q_{near} \leftarrow$ neartestNode($q_{rand}$)
 6:         $q_{new} \leftarrow$ newNodePos($q_{near}$, $q_{rand}$)
 7:         if collisionCheck($q_{new}$) is false
 8:             $Q_{neighbors} \leftarrow$ findNeighbors($G$, $q_{new}$, RAD)
 9:             $q_{minCost} \leftarrow$ findParent($Q_{neighbors}$)
10:             $G$.addNode($q_{new}$)
11:             $G$.addEdge($q_{minCost}$, $q_{new}$)
12:             $G$.rewire($Q_{neighbors}$, $q_{new}$)
13:         end if
14:     return $G$
```

2.5.3 Informed RRT*

Informed RRT* was developed by Gammell in 2014 and it further improves the speed of RRT*[31]. Informed RRT* defines an elliptical region which is a prolate hyperspheroid informed subset of the search space that has a higher probability of containing better or shorter solution paths. New samples will be created within this region reducing the search space thus improving the convergence time.

2.6 Discussion

This chapter presents various popular motion planning algorithms and they were divided into four major categories: geometric-based methods, artificial potential field, graph search, and sampling-based methods. This section discusses the algorithms' strengths and weaknesses in different scenarios. One major drawback of using the classic visibility graph is that it is computationally expansive in a complex and large obstacle environment[32]. Similar to many geometric-based methods, it also requires prior knowledge of the obstacle polygonal information which may not be suitable for real-time applications that rely on discrete obstacle sensor data[30]. Voronoi diagram is another geometric-based approach and it can be generated efficiently. Nevertheless, the path is less efficient as it maximizes obstacle distances and cannot be easily travelled by non-holonomic vehicles[27]. Artificial potential field algorithms are straightforward and provide fast run-time. However, they have difficulties avoiding local minimums situation, proceeding into narrow passages, and only restricted to local obstacle avoidance[33, 42]. The graph search algorithms are efficient in solving low dimensional problems but scale poorly to problem size[24]. Sampling based methods such as the variants of RRTs are recently emerging techniques. They scale well to high dimensional problem but suffer from low convergence time[34]. When comparing the algorithms' path qualities, the majority of the above algorithms produce segmented straight line paths which are suboptimal for non-holonomic vehicles to travel. The exceptions are the variants of curvature-based A* or RRT* which have further trade-offs for speed. Despite post-smoothing may be performed to tackle this problem, it poses new potential problems since a great portion of the smoothing functions do not consider the presence of the surrounding obstacle. The resulting smoothed path may no longer guarantee collision free, especially in obstacle clustered areas.

Chapter 3

Methodology

3.1 LIDAR A* and Hierarchical Path Planning

To create a robust motion planner, one should improve upon the issues discussed above. Given a two dimensional obstacle grip map, the algorithm's objective is to create a kinematically friendly, intelligent, and computationally efficient real-time motion planner that determines a path from a start to a goal location without collision. This thesis presents a new method, LIDAR A*, that utilizes a two level hierarchical architecture that divides the tasks into high level global trajectory planning and low level simulation-based motion generation. At high level, LIDAR A* incrementally explores the map from the start region and searches the consecutive neighboring regions for the shortest sequence of opening gateways that leads to the goal region. At low level, a sequence of vehicle motions is simulated using the robot's kinematic model and gateway information to produce a final simulated path toward the goal position.

3.2 Shortest Opening Gateway Sequence Search

Dijkstra's and A* are two commonly used global path planners. However, their computational complexity scales exponentially with map size, resulting in unacceptable run-time in large maps. One common solution is to reduce map resolution (increase map decomposition cell size) that has the trade-off of losing map details. Another solution is through hierarchical map abstraction and decomposition as previously discussed in section 2.4.3. HPA* has to pre-process the high level abstract graph before a search can take place. Unlike HPA*, LIDAR A* introduces a new novel technique that the map abstraction and decomposition processes are done concurrently as the search in the high level abstract graph is in progress. LIDAR A* uses A* search with euclidean distance toward the goal as the heuristic function *h(x)*, though similar technique can be applied to Dijkstra's algorithm when optimal gateway sequence must be guaranteed.

Beginning at a robot's start configuration $q(x, y, \theta)$, LIDAR A* abstracts and decomposes and its local regional map into an abstract graph node. The region is identified by simulating a 181 degree 2D LIDAR scan (referred as scan) at a scan configuration of map position, orientation, and a variable LIDAR detection distance (x, y, θ, k). The simulated scan data is analyzed to determine the local opening gateways' positions and orientations used to enter neighboring unscanned regions where subsequent scans are performed. This process represents the identification of neighboring open nodes in high level A* abstract graph search. When no neighboring opening gateway can be identified, the node is closed. Because of the visibility feature of LIDAR and design of scan data clustering process, opening gateways in LIDAR A* are guaranteed traversable and visible from local scan configuration. Contrasting to HPA* which the shortest path searches must be performed on all possible pairs of entrances within a cluster to construct its abstract graph, LIDAR A* completely eliminates those in-cluster searches simplifying the process. The euclidean distance between two neighboring gateway locations is used as the cost $g(x)$ between the two nodes. Once the total costs $f(x) = g(x) + h(x)$ of all neighboring gateways or adjacent abstract graph nodes are determined from a scanned region, the gateway node with the least cost is selected to proceed A* search process. The old node is then closed, and an imaginary boundary is drawn at the old gateway location on the map to block and prevent the search from getting trapped in a cycle. A new scan is then performed at the selected gateway's configuration. A* search process on the abstract graph will continue to expand and explore the map until the goal is within the current scanned region. The shortest opening gateway sequence is therefore determined.

The process of finding opening gateways within a scanned region requires three steps: simulating LIDAR data with cache optimization, clustering obstacle data into groups, and generating opening gateways from the obstacle group edges.

3.2.1 Cached Simulated LIDAR Scan

The purpose of simulating a LIDAR scan to represent a grid map region is to first, reduce the data size from a 2 dimensional area to a small fixed number of polar coordinates, and second, utilize the visibility property of LIDAR. In the experiment setup, a scan contains 181 data points in the form of (θ, d) that reports the nearest visible obstacle distance d at an incrementing angular scanning direction θ with respect to the local LIDAR's frame. It has an angular resolution of 1 data per 1 degree angular increment (from 0 to 180 degree). This map format conversion technique greatly lowers the map data size while retaining a satisfactorily detailed view of the obstacle contour looking from the LIDAR's configuration. The reduced number of map data is favorable for more efficient analysis of the obstacle map, as long as the processing time for the format conversion justify the trade-off.

One way to simulate LIDAR scan data is to replicate LIDAR's detection mechanism. Starting from the minimum scanning direction or 0 degree cartesian direction of the local LIDAR's scanning frame, it checks the map cell outward incrementally until it detects an obstacle cell or reaches the maximum detection distance. It reports the distance, then moves on to the next detection direction by incrementing the detection angle by a step of the angular resolution or 1 degree. Although this method may be useful in generating a more realistic LIDAR data when noise can be more accurately simulated and inserted, it is less efficient because map cells are accessed out of order from memory which causes frequent CPU cache misses. Notice that the input 2 dimensional obstacle map may either be a predetermined static map or a single frame of a real-time dynamic map that already contains sensor noise. In the second case, adding additional noise might add unnecessary complications.

The second proposed LIDAR simulation method is more optimized for cache hits with speed improvement but does not mimic LIDAR’s detection mechanism. Since a scan in LIDAR A* has an angular scanning range of 181 degree, it forms a maximum scanning region of a half circle with a radius equal to detection distance. The goal is to find the limits of the smallest possible bounding box on the map that tightly encloses this half circle which can be in any global orientation from 0 to 359 degree on the map as shown in the green rectangle box in Figure 12. This bounding box is determined by first calculating the maximum scan detection positions *A* and *B* on the map, which are at LIDAR’s 0 and 180 degree angular scanning directions.

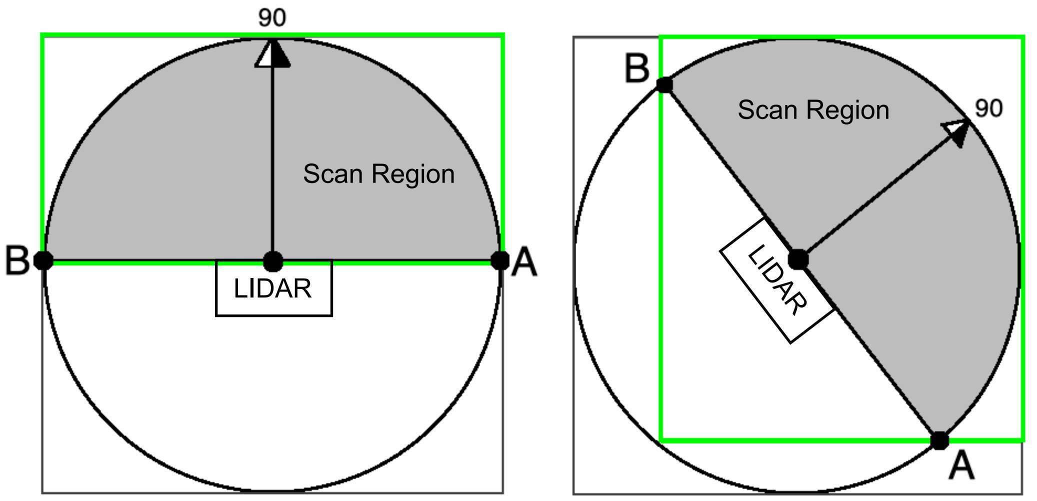

Figure 12. Smallest Bounding Box (Green Rectangle) for Simulated LIDAR Scan

Depending on what quadrant the scan's orientation lies in, the global position values (*x, y)* of *A* and *B* and the LIDAR's detection distance are used to set up the respective top, right, bottom, and left limits of the smallest bounding box. Compared to using a bounding box that encloses the entire full circle based on only the LIDAR's detection distance, this method improves the speed by 27% to 50% because the region is reduced from a full square to a smaller rectangle.

Once the bounding limits are identified, the map cells within the bounding box limits are accessed in row-major order (for C++), which is best for CPU cache hits since data is accessed in order from memory. If an obstacle cell is detected while looping through the bounding box region, LIDAR A* finds the cell's local distance and global direction with respect to the scan's position. Then it updates the known minimum obstacle distance at that direction when the current detected obstacle distance is smaller. Since the cell distances and directions with respect to a scan's position are independent of obstacle information, a table of cell distances and directions is predetermined offline and is directly accessed when an obstacle cell is detected, which further enhances the speed of the algorithm. The local simulated LIDAR scan can be obtained once all cells within the bounding box are analyzed and global directions that are outside of the LIDAR's angular scanning range based on the current scan orientation are cropped out.

3.2.2 Obstacle Data Clustering

The next step is to cluster these scan data into groups and label each individual obstacle data point with a group number. Notice that the cluster used in this and following sections refers to the process of grouping data together instead of the sub-grid map cluster used in HPA*. Using clustered LIDAR data to achieve local obstacle avoidance was studied by Wang and Peng[35, 36, 37]. The objective is to cluster the scan's polar coordinate data into groups using a *grouping radius* such that:

> Rule: An obstacle data can be apart from its nearest obstacle data within the same obstacle group by a maximum distance of *grouping radius*.

Since the 2D scan data are collected by incrementing the scan direction and report the nearest obstacle distance at each scan direction, this rule guarantees that:

1. There is no obstacle free passage that is greater than the size of *grouping radius* in between the directions of the right edge and the left edge of an obstacle group.
2. There is an obstacle free passage that is greater than the size of *grouping radius* between the directions of the left edge of one obstacle group and the right edge of the next (angular) obstacle group, and the opening to this free passage is always visible from LIDAR's configuration

The second property is the general rule for identifying opening gateways within a scanned region using a *grouping radius* of robot width plus additional clearance. However, there are some exceptions and will be discussed in the next section 3.2.3.

Peng proposed a clustering method that checks the distance between every two consecutive LIDAR scan data. If the distance is smaller than the grouping radius, the data are clustered together[37]. This method has potential flaws for ensuring obstacle clearance. The reason is that it is possible for a data point P_2 to have a larger scanning direction, $\theta_2 > \theta_1$, while having a closer distance to data point P_0 compared to P_1 when P_1 is the next consecutively scanned data point of P_0 as illustrated in Figure 13. When P_1 and P_0 have a distance equal to robot width, the robot assumes there's a safe opening when the gap between P_0 and P_2 does not provide sufficient obstacle clearance.

An improved method is described as follows: Every obstacle datapoint O_n in a scan S has obstacle direction, distance, and label of $O_n(\theta_n, d_n, l_n) \in S$, a minimum and a maximum angular scan directions of θ_{min} and θ_{max} where $\theta_{min} \leq \theta_n \leq \theta_{max}$, and a *grouping radius* of r. The algorithm first initializes the label l_n for every data point O_n in S to 0 or *unknown* and initializes a *newLabel* variable to 1. Then, for every data point O_n in S, if l_n is *unknown*, it sets data label l_n to *newLabel* and increments *newLabel.* Then it identifies a group of obstacle data $O_c(\theta_c, d_c, l_c)$ based on their directions that need to be checked for distance D_c, which is the distance between O_n to O_c. When obstacle distance D_c is less than *grouping radius* $d_n < r$, it will select $O_c(\theta_c, d_c, l_c)$ where $\theta_n < \theta_c \leq \theta_{max}$ which is the remaining obstacle data from O_n in S. When obstacle distance is greater than *grouping radius* $d_n > r$, it will select $O_c(\theta_c, d_c, l_c)$ where $\theta_n < \theta_c \leq \theta_n + asin(r / d_c)$ which is an angular range that contains all possible O_c

that can have a distance less than *grouping radius,* $D_c < r$. The upper limit $\theta_n + asin(r / d_c)$ is the tangent line of a circle centered at O_c with a radius r as shown in Figure 13. When D_c is found to be less than r and the group label l_c is *unknown,* l_c is set to l_n . If $D_c < r$ but the group label l_c is known and $l_c \neq l_n$, an inner loop is applied on S to iteratively merge all obstacle data in S that has label l_c to l_n .

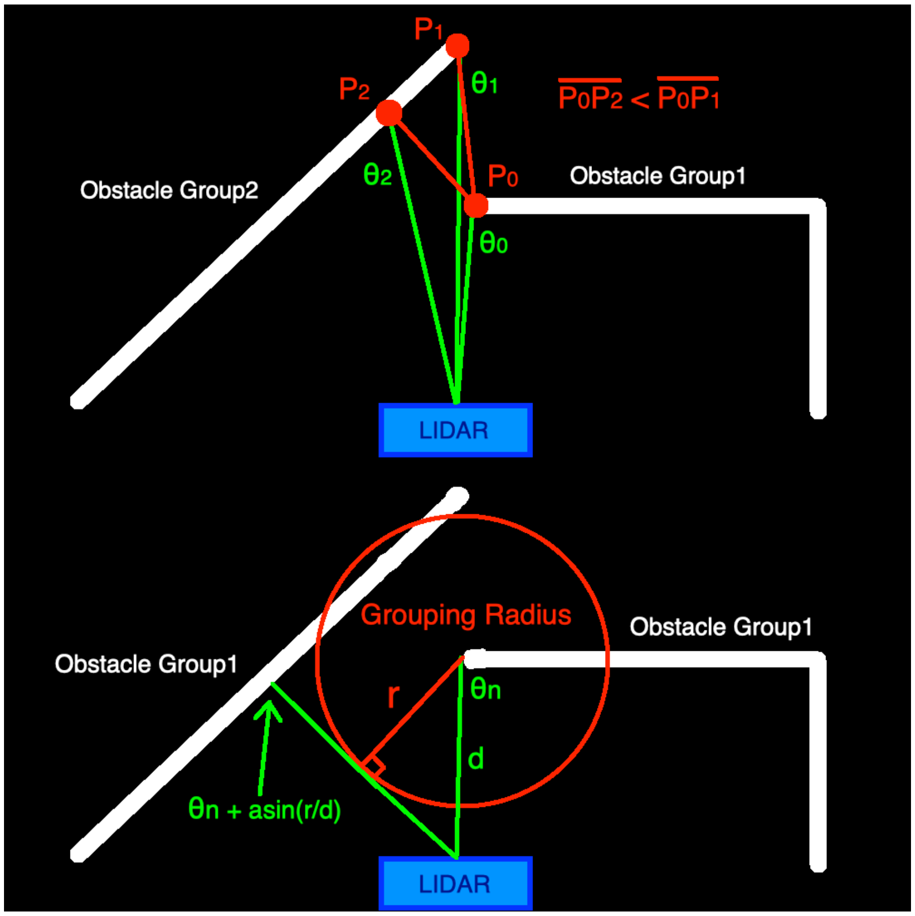

Figure 13. Obstacle Clustering. Failure condition for checking consecutive data point distance. Proposed method uses a grouping radius on every data point and check distances to data points between directions from Θ_n+1 to $\Theta_n + asin(r / d)$

The number of O_c is inversely proportional to the obstacle distance d_c which can be seen in the upper limit of the angular direction check range $\theta_n + asin(r / d_c)$. The run-time of this algorithm is dynamic. When a scan returns a more clustered obstacle data, the algorithm takes a longer time to cluster data because more distance checks are required on obstacle data with shorter distance.

A *grouping region* is formed by overlapping the areas of the grouping radii within an obstacle group. A *grouping region* of an obstacle group does not overlap

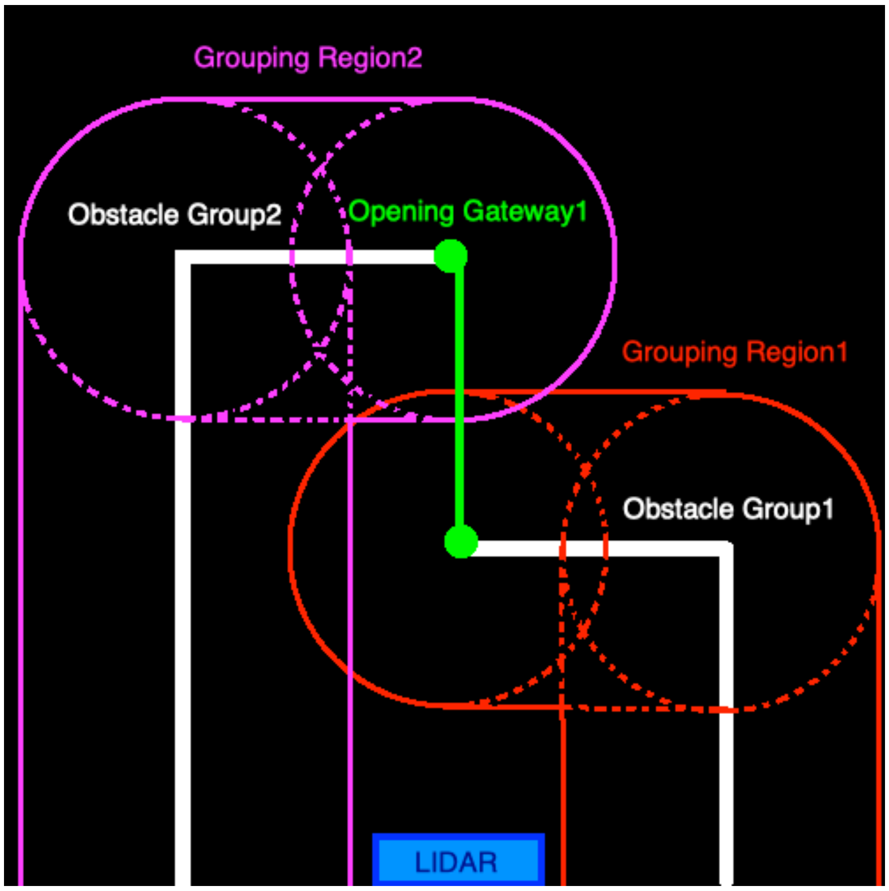

Figure 14. Grouping Regions and Opening Gateway. Grouping Region does not overlap with data in other obstacle group. Passage (Opening Gateway) is identified between the obstacle edges of the two consecutive obstacle groups

with any obstacle data in another obstacle group creating obstacle clearance that is greater than or equal to the *grouping radius* as shown in Figure 14. A passage (Opening Gateway1) is determined between the left edge of Obstacle Group1 and the right edge of Obstacle Group2 that guarantees obstacle clearance and is always visible from LIDAR's point of view.

Algorithm ClusterData

```
 1: function Cluster (scan, groupRad, maxRange)
 2:     initLabel(label, 0) //0 means "unknown"
 3:     newLabel = 1
 4:     for i = 0 to 180  //$O_n$
 5:         if  scan[i] < maxRange //distance filter
 6:             if label[i] is 0
 7:                 label[i] ← newLabel
 8:                 newLabel++
 9:             end
10:             if distance[i] < groupRad
11:                 checklimit ← 180
12:             else
13:                 checklimit ← i + asin(groupRad/distance[i])
14:             end
15:             for j = i+1 to checklimit  //$O_c$
16:                 if  label[i] ≠ label[j] and dis(scan[i], scan[j]) < groupRad
17:                     if label[j] is 0
18:                         label[j] ← label[i]
19:                     else
20:                         mergeGroup(label, i ,j)
21:                     end
22:                 end
23:
24:         end
25:     return  label
```

3.2.3 Opening Gateways

Passages are located between the left edge of one obstacle group and the right edge of the next obstacle group and LIDAR A*'s clustering algorithm can help ensure minimum obstacle clearance. Nevertheless, not all passages identified are actually traversable because of the passages' positions and orientations. This part of the process aims to remove the non-traversable passages to determine the final set of opening gateways for exploring and traversing the map.

There are two types of passages that are non-traversable. The first one is *sideway passage. Sideway passage* is identified when an obstacle group's side (Left and Right) edges are entirely enclosed by another obstacle group's side edges. This condition occurs when a pair of obstacle groups have one placed in front of the other as shown in Figure 15. These passages are either non-traversable or entering/exiting within its own region which defeat the purpose of connecting to neighboring regions. In this case the obstacle group in the back is removed leaving the front obstacle group to create possible passages with other adjacent obstacle groups. The possible passages using the rear obstacle group will be identified in subsequent scans.

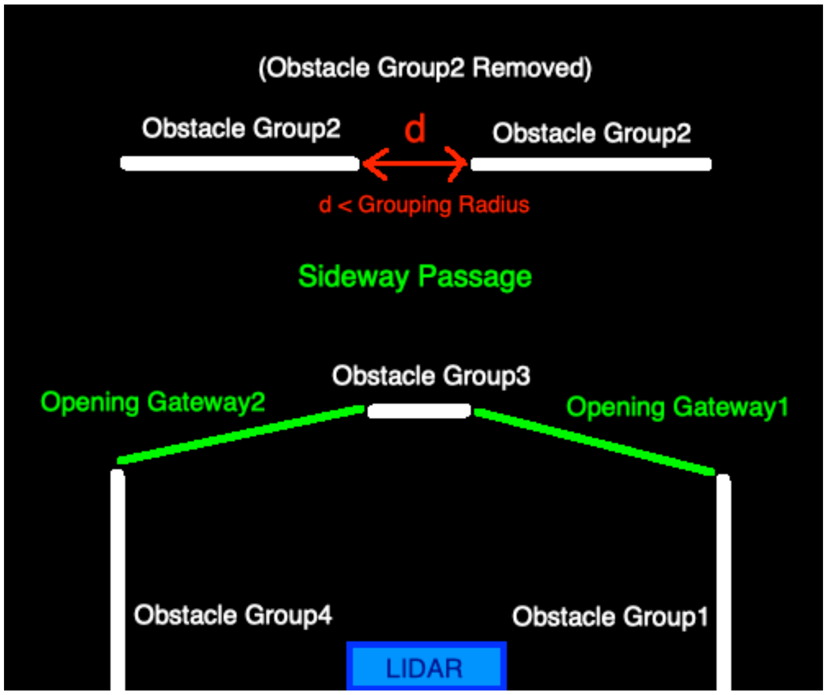

Figure 15. Sideway Passage. Obstacle group3 is in front of Obstacle group2 creating a sideway passage. Obstacle group2 is removed at the current scan location to determine valid Opening Gateway1 and Opening Gateway2

The second type of non-traversable passage has the following general orientation as in Figure 16. These passages are unreachable because parts of its adjacent obstacles are blocking its path toward the passage. These passages are identified and filtered out by making sure when a passage's left edge is near the 0 degree heading or when the passage's right edge is near the 180 degree heading, there must be sufficient clearance in the vertical direction.

After these types of invalid passages are successfully filtered out, the remaining passages are considered valid opening gateways for entering the neighboring unscanned regions. The global coordinates of the adjacent left and right obstacle edges, as well as their mid points are recorded and used for motion simulation of the robotic vehicle. A new scan will take place at an orientation perpendicular to the line connecting the two adjacent obstacle edges of the selected opening gateway, facing toward the unscanned region.

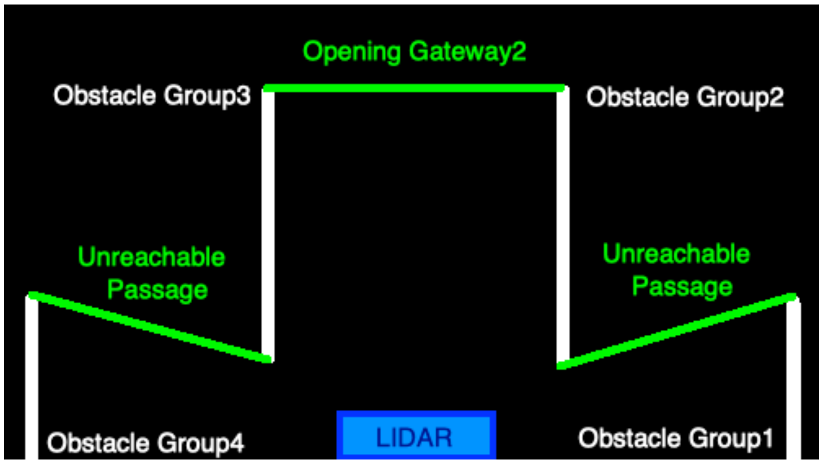

Figure 16. Unreachable Passage. Unreachable passages may occur when they are near the 0 or 180 degree of the LIDAR's frame

3.3 Motion Simulation

LIDAR A* utilizes a hierarchical architecture that divides the tasks into high level shortest opening gateway sequence search and low level motion planning. The generated gateway sequence is constructed with seamless transition between simulated LIDAR scans with scan configurations close to the ideal robot configurations when traversing the gateways. Low level motion planning is tasked with building a path that smoothly maneuvers within each local region and transitions to neighboring regions using the given gateway information.

Motion planner algorithms worked in the lower level hierarchical architecture are often given a suboptimal path or a sequence of waypoints from the high level path planner to determine obstacle free motions. Picture skiing down a slope with surrounding obstacles and trying to follow a suboptimal path. The task of smoothing out a suboptimal path under motion constraints while ensuring obstacle clearance can be challenging. Imagine instead of a path, a sequence of marked gateways with ideal traversing orientations given to follow. With better knowledge of the drivable regions that provide additional room for path optimization, the task of maneuvering around clustered obstacles becomes considerably easier. Similar to Hybrid A* that ensures motion feasibility, LIDAR A* uses online simulation with vehicle kinematics to plan a path based on sequential states of a non-holomonic car-like vehicle. When a collision is identified during the simulation process, a replanning technique is proposed in section 3.3.3.

3.3.1 Kinematic Model

The kinematic model used on the non-holonomic car-like vehicle simulation is specified in LaValle's *Motion Planning* book[41]. No dynamic model was used to avoid over-complicating the system. A configuration of a car is denoted $q = (x, y, \theta)$, the distance between the front and rear axles is denoted L, the steering angle of the front wheel is denoted ϕ, and speed of the vehicle is denoted u_s. The configuration transition equations of a car-like vehicle are:

$$\begin{aligned} \dot{x} &= u_s \cos\theta \\ \dot{y} &= u_s \sin\theta \\ \dot{\theta} &= \frac{u_s}{L} \tan\phi \end{aligned}$$

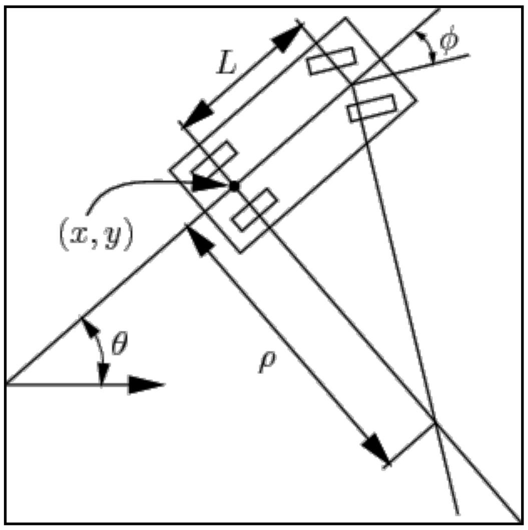

Figure 17. Car-Like Vehicle Kinematics

The simulation was completed with a constant speed in terms of map cell distance over time, a *dt* of 0.2 second, and a steering angle limit of -40 and +40 degrees.

3.3.2 Obstacle Avoidance

The calculated opening gateways sequence contains position information about the adjacent left obstacle edge, the adjacent right obstacle edges, and their midpoints between the two obstacle edges. A simple but effective motion planning algorithm which is the simplified version of Wang's motion planner algorithm was used to control the vehicle[35, 36]. This algorithm considers the current approaching gateway and the next follow-up gateway and converts their global map coordinates to the local coordinates of the current vehicle's frame in direction and distance format.

Let $G_L(\theta_L, d_L)$, $G_M(\theta_M, d_M)$, $G_R(\theta_R, d_R)$ denote the current left edge, mid point, and right edge of an opening gateway, and $G_{LN}(\theta_{LN}, d_{LN})$, $G_{MN}(\theta_{MN}, d_{MN})$, $G_{RN}(\theta_{RN}, d_{RN})$, for the next opening gateway respectively.

Traversing a gateway has an important property that the angle between the left and right obstacle edges, θ_L and θ_R, will gradually increase to 180 degree as it approaches and passes through the gateway. The speed of increase depends on the orientation of the gateway as well as the vehicle's orientation when it approaches the gateway. In Nilsson's visibility graph, Perez shifts away and traverses toward the shifted obstacle polygonal vertex position that is visible from the current vertex.

Since opening gateways are generated with similar visibility property, LIDAR A* first shifts the current obstacle edge directions, θ_L and θ_R, away from the obstacle edge locations to create necessary clearance to accommodate the size of the vehicle. It then checks for multiple thresholds on the value of $\theta_L - \theta_R$ to identify the relative position and orientation between the vehicle and the gateway within the current scanned region and apply policies for motion generation.

When $\theta_L - \theta_R \leqq 0$, this indicates a front obstacle is partially blocking the view of another rear obstacle and a path maneuvering around the front obstacle is expected. The algorithm outputs θ_L or θ_R that has the respective smaller obstacle distance d_L or d_R as the output steering angle. It helps the vehicle take the shortest path around the corner of the front obstacle.

When $0 < \theta_L - \theta_R \leqq$ *MidThreshold,* this indicates the vehicle is still away from the opening gateway that causes a small difference between the two gateway edge directions. In this case, the algorithm outputs θ_M as steering angle that aims toward the center of the opening gateway. The vehicle will tend to stay in the safer center path of the current scanned region.

When *MidThreshold* $< \theta_L - \theta_R \leqq$ *CheckOffThreshold,* this indicates the vehicle is near the current opening gateway. It can begin steering toward the next opening gateway without going out of the angular range limits of the current opening gateway to explore a more efficient path toward the next region. The algorithm outputs the next mid gateway θ_{MN} as steering angle if $\theta_R \leqq \theta_{MN} \leqq \theta_L$. If $\theta_{MN} < \theta_R$ or $\theta_{MN} > \theta_L$, the algorithm outputs either θ_L or θ_R whichever is closer to θ_{MN}.

When θ_L - θ_R > *CheckOffThreshold,* this indicates the vehicle is at the current opening gateway. It checks-off the current gateway and moves on to the next two consecutive opening gateways.

Figure 18 demonstrates the partially blocking view condition when $\theta_L - \theta_R \leqq 0$. Figure 19 to Figure 23 show the motion generation of a car-like vehicle traversing through three sequential opening gateways. The steering angle of the vehicle is determined by thresholding the value of $\theta_L - \theta_R$.

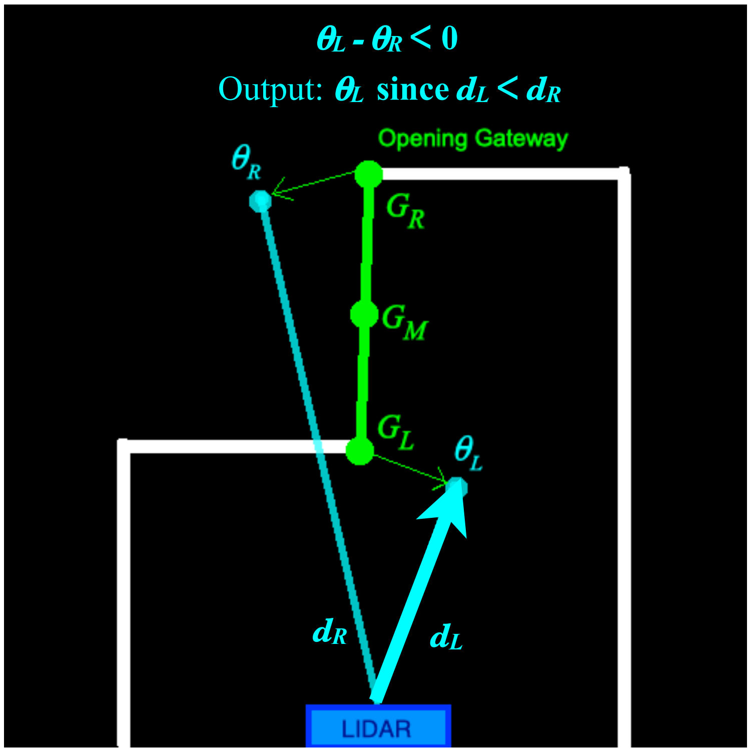

Figure 18. Motion Planning: when $\theta_L - \theta_R \leqq 0$, the algorithm outputs θ_L or θ_R that has the respectively smaller obstacle distance d_L or d_R as the output steering angle

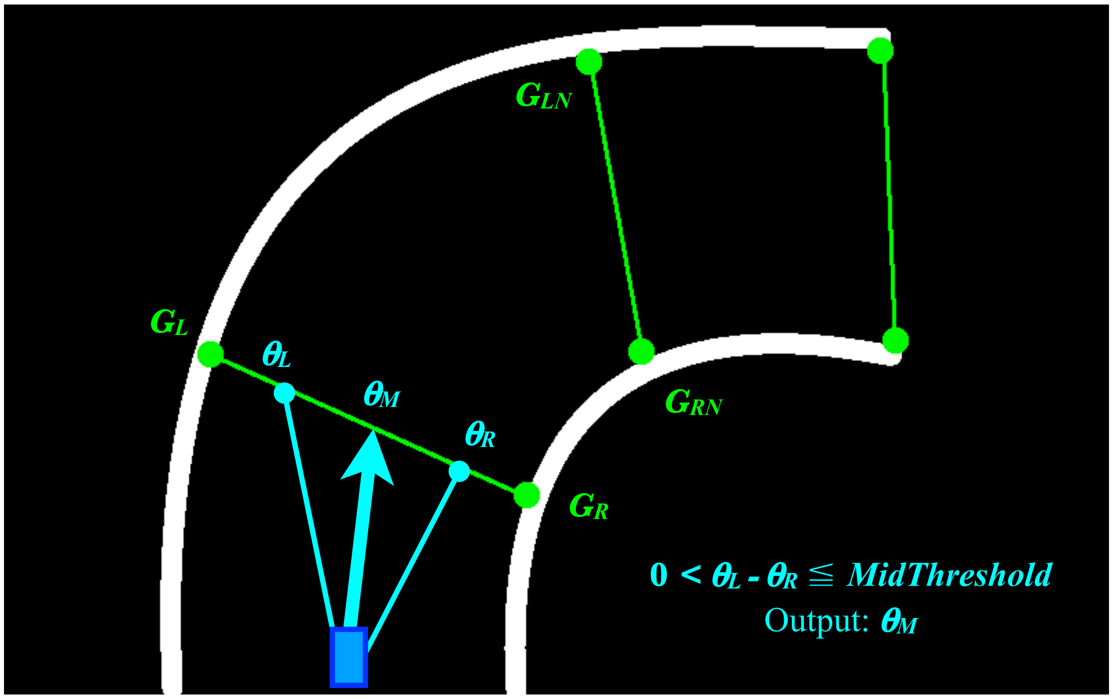

Figure 19. Motion Planning: when $0 < \theta_L - \theta_R \leqq$ *MidThreshold*, The algorithm outputs θ_M that aims toward the center of the opening gateway

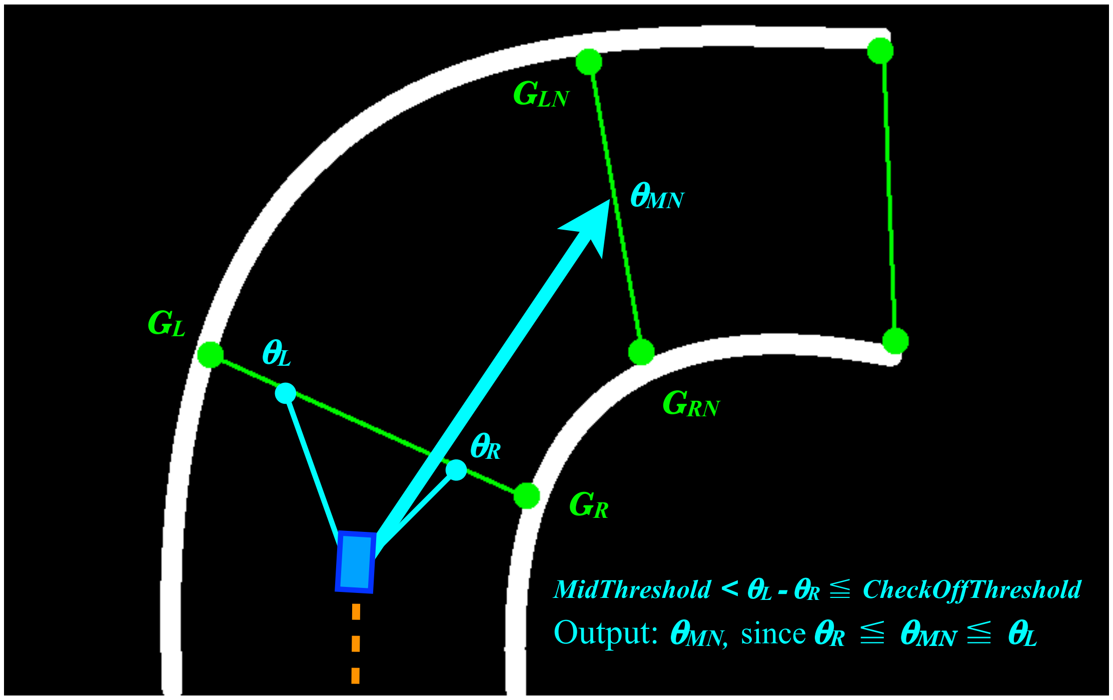
G_{LN}
θ_{MN}
G_L
θ_L
G_{RN}
θ_R
G_R
$MidThreshold < \theta_L - \theta_R \leqq CheckOffThreshold$
Output: θ_{MN}, since $\theta_R \leqq \theta_{MN} \leqq \theta_L$

Figure 20. Motion Planning: when *MidThreshold* $< \theta_L - \theta_R \leqq$ *CheckOffThreshold*, the algorithm outputs the next mid gateway θ_{MN} to generate a more efficient path toward the next gateway

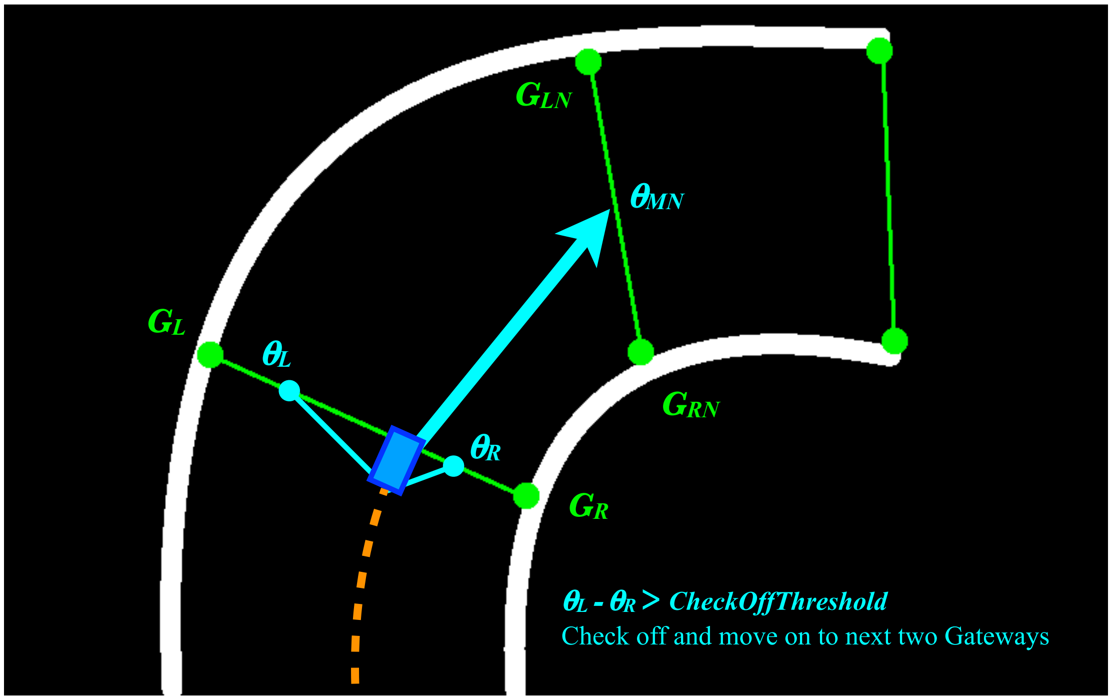
G_{LN}
θ_{MN}
G_L
θ_L
θ_R
G_{RN}
G_R
θ_L - θ_R > CheckOffThreshold
Check off and move on to next two Gateways

Figure 21. Motion Planning: when θ_L - θ_R > *CheckOffThreshold*, the algorithm checks-off the current gateway and moves on to the subsequent two opening gateways

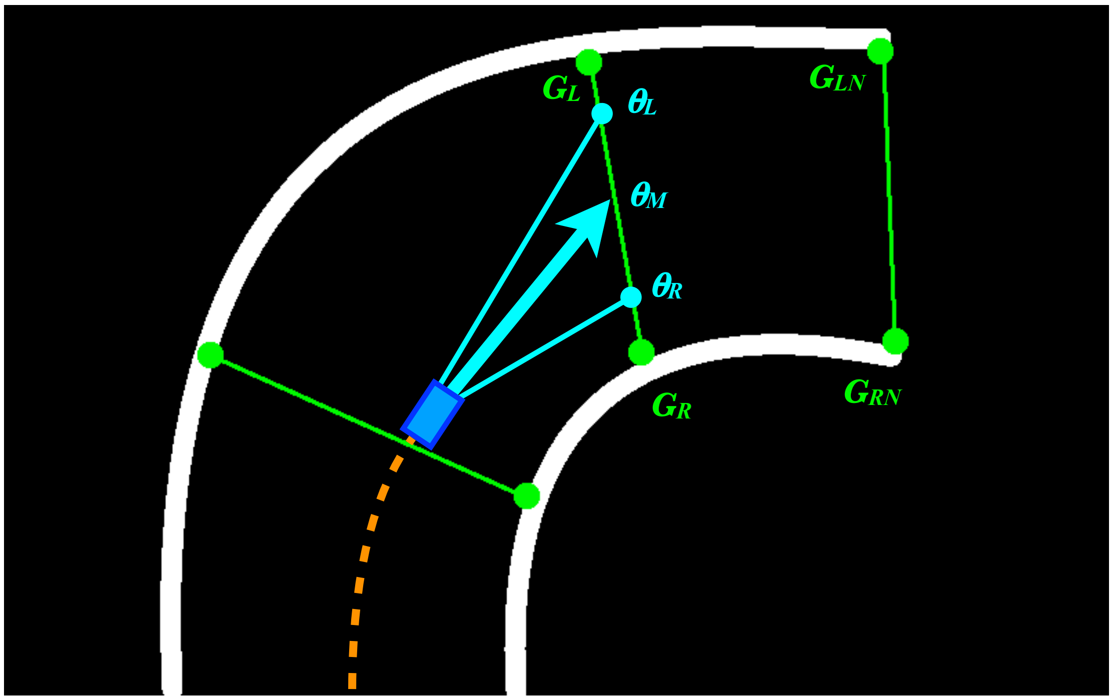

Figure 22. Motion Planning: Upon checking off the old gateway, the algorithm again outputs θ_M that aims toward the center of the new current opening gateway

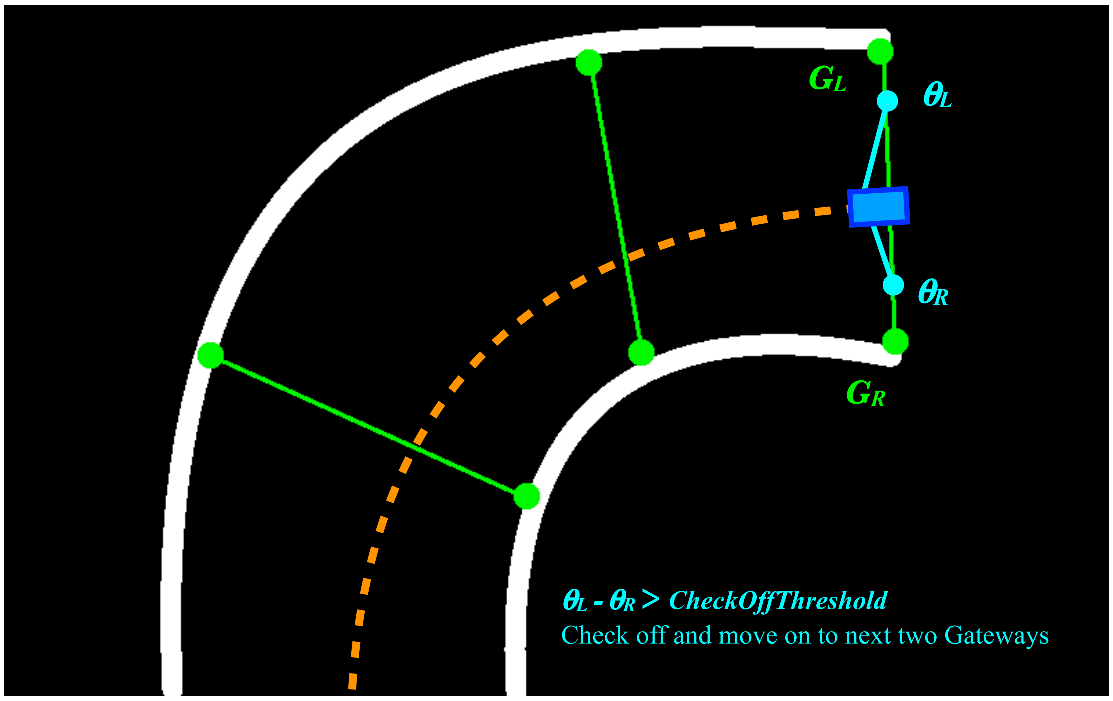
G_L
θ_L
θ_R
G_R
$\theta_L - \theta_R >$ CheckOffThreshold
Check off and move on to next two Gateways

Figure 23. Motion Planning: The vehicle finished traversing a sequence of three opening gateways

At the end of the gateway sequence, LIDAR A* outputs the goal direction as the vehicle steering angle when the goal is directly in sight. Then it reaches the goal and completes the simulation. Since motion control of the vehicle is based on its continuous local feedback of the stationary opening gateway edges, the above policies generate a continuous and smooth path that is near optimal from the start to goal locations on the map.

3.3.3 Collision Check and Intermediate Opening Gateway

The proposed local motion planner strictly uses the opening gateways' obstacle edge information to simulate a motion path ignoring the rest of obstacle information. It greatly simplifies the motion planning processes but comes with side-effects. Even though obstacle avoidance based on maneuvering around obstacle edges is an effective approach, in certain obstacle settings, there is still a possibility that the algorithm simulates a path that grazes against other parts of the obstacle before reaching the current gateway location as shown in Figure 24.

A simple solution to this problem is to perform collision check through each time step of the simulation process within the current region while traversing toward the current gateway location. When a collision is detected, a new opening gateway analysis is performed at the original scan location using a reduced scan detection distance that is slightly larger than the detected collision distance. Using this reduced detection distance will determine a new intermediate opening gateway that has one of the adjacent obstacle edges causing the previously detected collision. This intermediate opening gateway is inserted into the gateway sequence and a new path is simulated avoiding the original collision.

There are several approaches for detecting collisions. Since LIDAR A* already has the ability to simulate local LIDAR scan data, collision check can be easily accomplished by simulating a scan at the vehicle's configuration on the map using a detection distance that is just enough to cover the entire local contour of the vehicle. This scan data is compared to a predetermined vehicle contour mask which is also a simulated LIDAR scan to check for any interference.

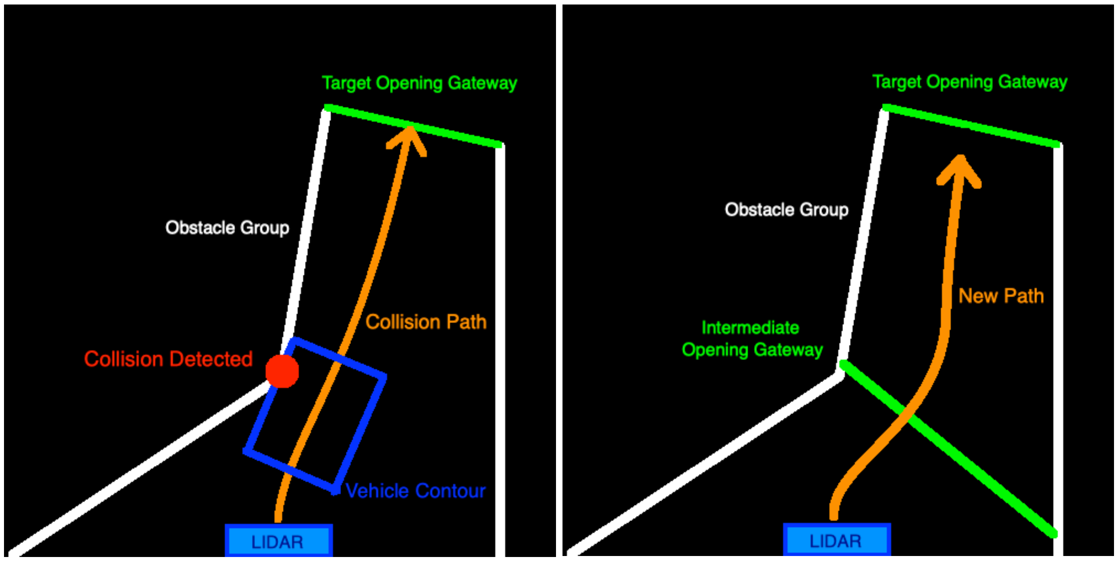

Figure 24. Collision Detection and Intermediate Gateway. When a collision is detected while traversing toward the current gateway, an intermediate gateway is identified using a smaller LIDAR detection distance. This gateway is inserted to replan a collision-free path

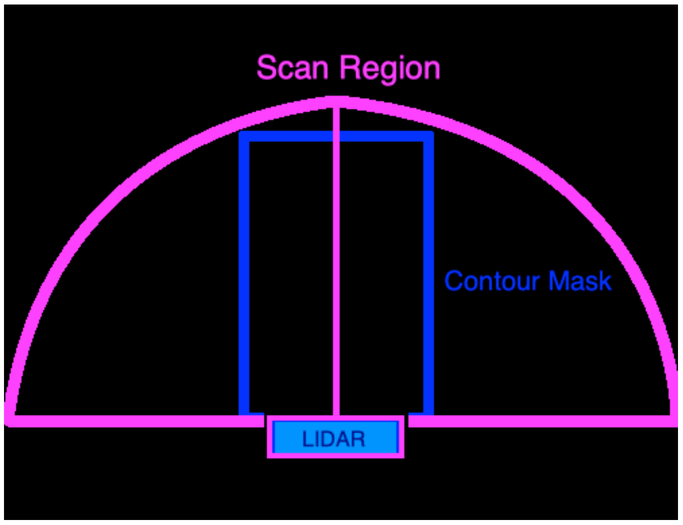

Figure 25. Collision Detection. Collision can be detected by checking for interference between the scan region and the vehicle contour mask

3.3.4 Dynamic LIDAR Detection Distance

It is crucial to select an appropriate LIDAR detection distance in LIDAR A* that best works for the obstacle setting at the current scan location, as it will affect the accuracy and efficiency of the algorithm. Because of the visibility property of LIDAR scan, using a larger than ideal detection distance may waste computational resources scanning regions that are not visible at the LIDAR configuration, making the algorithm less efficient. Using a larger than ideal detection distance may also generate opening gateways that are further away from the scan location increasing the risk of creating a collision path in low level motion planning in a clustered environment. This condition can be observed in the previously discussed collision path example (Figure 24). The added collision check and replanning technique is a fail-safe for not using an appropriate detection distance.

The LIDAR detection distance can not be too short either, especially in more open areas. LIDAR A* strictly relies on obstacle edges for path and motion planning. It can be problematic when no obstacle can be identified at the current scanned region. The most effective solution is to increase the detection distance until some obstacles are detected or until the detection distance can cover and reach the goal location. In the second case there is no obstruction in between the current and goal locations and the vehicle can simply travel toward the goal. Using a shorter than ideal detection distance may also reduce the algorithm's efficiency. When a shorter detection distance is used in decomposing a map, a scan covers a smaller region, resulting in more scans and more clustering processes required to complete the high level gateway search which penalizes the algorithm's efficiency. To best optimize the accuracy and efficiency of LIDAR A*, a dynamic detection distance shall be implemented that vary depending on obstacle complexity at each gateway region. The general rule is to use a larger detection distance in a more open area and a shorter detection distance in a clustered area that produces ideal opening gateways without causing a collision path.

Chapter 4

Experiments and Results

4.1 Software Implementation

To evaluate the accuracy and efficiency of LIDAR A*, the algorithm was coded in C++ with OpenCV for map and decision visualization. The program ran on a 2019 MacBook Pro with a i7-8850H CPU clocked at 2.60GHz. The input map is a regular BMP image that can vary in sizes. Black pixels in the image represent obstacle free areas and white pixels represent obstacle areas. To maintain sufficient map detail, there is no minimum size for obstacle size and an obstacle can be as tiny as a single obstacle pixel. The experiments first tested the shortest opening gateway sequence search followed by low level motion simulation. The computation times are recorded excluding the time spent on graphing the decisions. Collision detection is turned off to examine the potential collision path condition in low level motion planning.

4.2 Simulated LIDAR Scan, Clustering, Opening Gateways

Figure 26 shows the result of a typical abstraction and decomposition process on a regional map. The blue contour at the bottom center shows the robotic vehicle and the LIDAR's configuration which has a scanning orientation pointing toward the up direction in the regional map. With a LIDAR detection distance about half the width of this image, the algorithm correctly generated simulated LIDAR scan data at the closer edge of obstacle contour observed from LIDAR's position. These data are being clustered into two obstacle groups, Obs1 and Obs2, and the individual obstacle data in the same group is labeled and drawn using the same group color. The opening gateways G1 and G2 are then identified between the left and right obstacle edges of two consecutive obstacle groups. They may also be formed between the LIDAR's maximum detection position at 0 or 180 degree direction and the nearest obstacle edge. Notice because there is an immediate follow-up dead-end at G2 opening gateway, G2's search branch will quickly be terminated as there is no neighboring region to be further explored. The search will proceed on G1 because it provides a subsequent neighboring region.

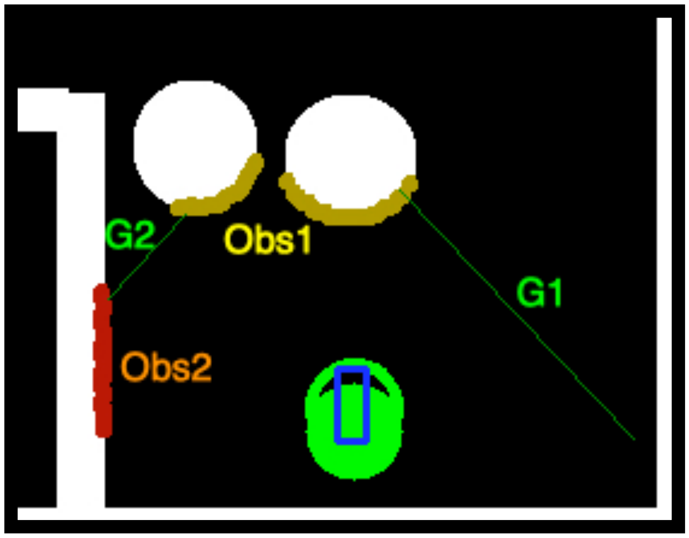

Figure 26. LIDAR Scan, Obstacle Groups, Opening Gateways

4.3 Shortest Gateway Sequence Search and Motion Simulation

Figure 27, 28, 29 show the complete search process of the shortest opening gateway sequence search and the final result from low level motion simulation. During the search process, neighboring regions of the current scanned location or neighboring abstract graph node were added to open list and their total costs in A* search were determined. The neighboring region in the open list with the least cost was selected to proceed where a new scan was performed at the selected opening gateway location pointing toward the unscanned region. The bright blue line in the image demonstrates the opening gateway's selection process in the high level A* abstract graph search. The bottom image of Figure 29 shows the complete low level simulated motion path which is based on the vehicle’s kinematic model. This simulated path is drawn using dark blue color and is the sequential configuration of the vehicle's contour from the start location to goal location. As shown in this Figure, the proposed local motion planner calculated a near-optimal path using obstacle edge information from the predetermined sequential gateways. The search was completed with 17 simulated LIDAR scans that represent 17 decomposed regions or 17 abstract graph nodes. It took an impressively low run-time of 16ms to compute the final simulated path, which has a path distance of 792 pixels.

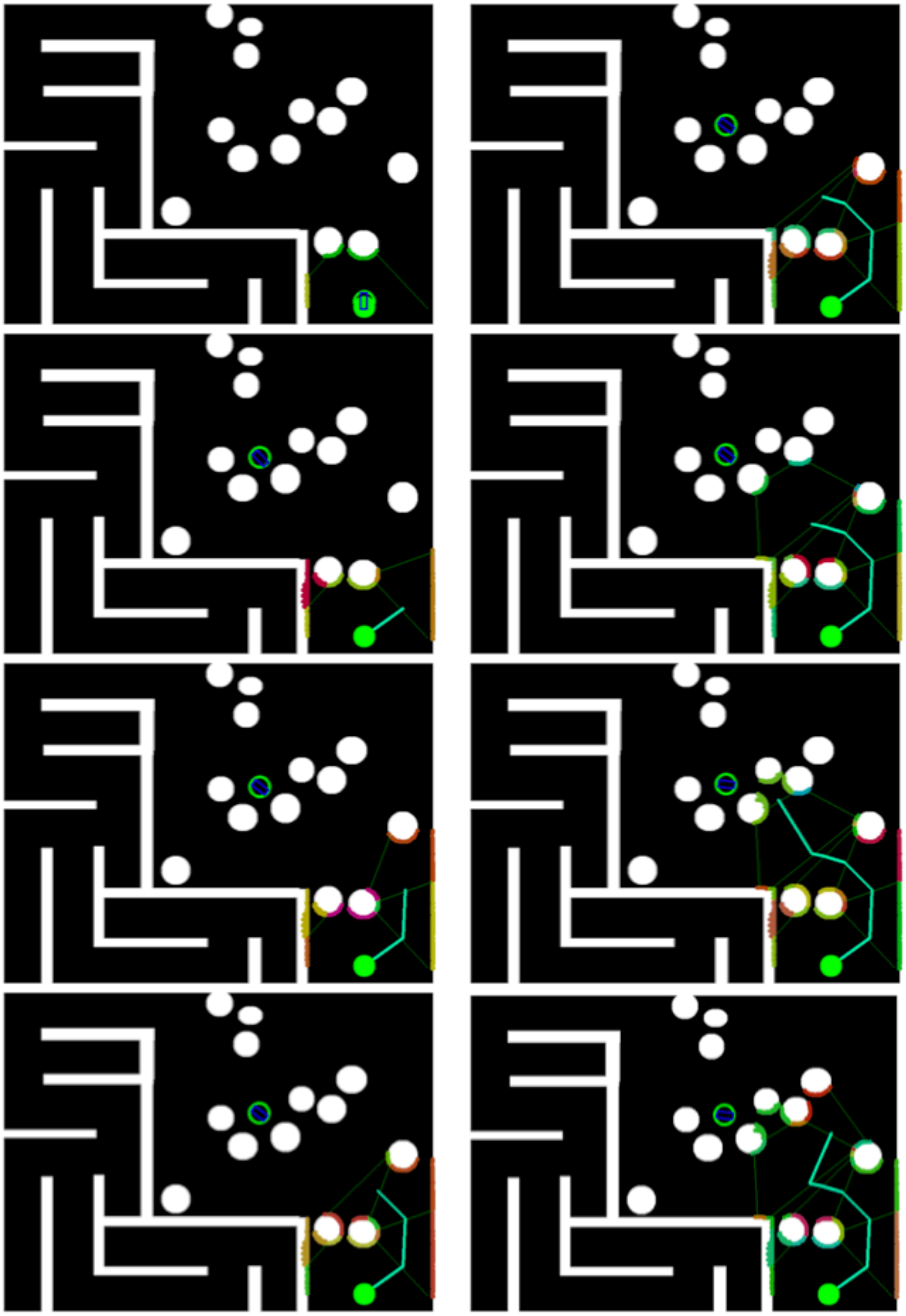
Figure 27. Shortest Opening Gateway Sequence Search 1-8

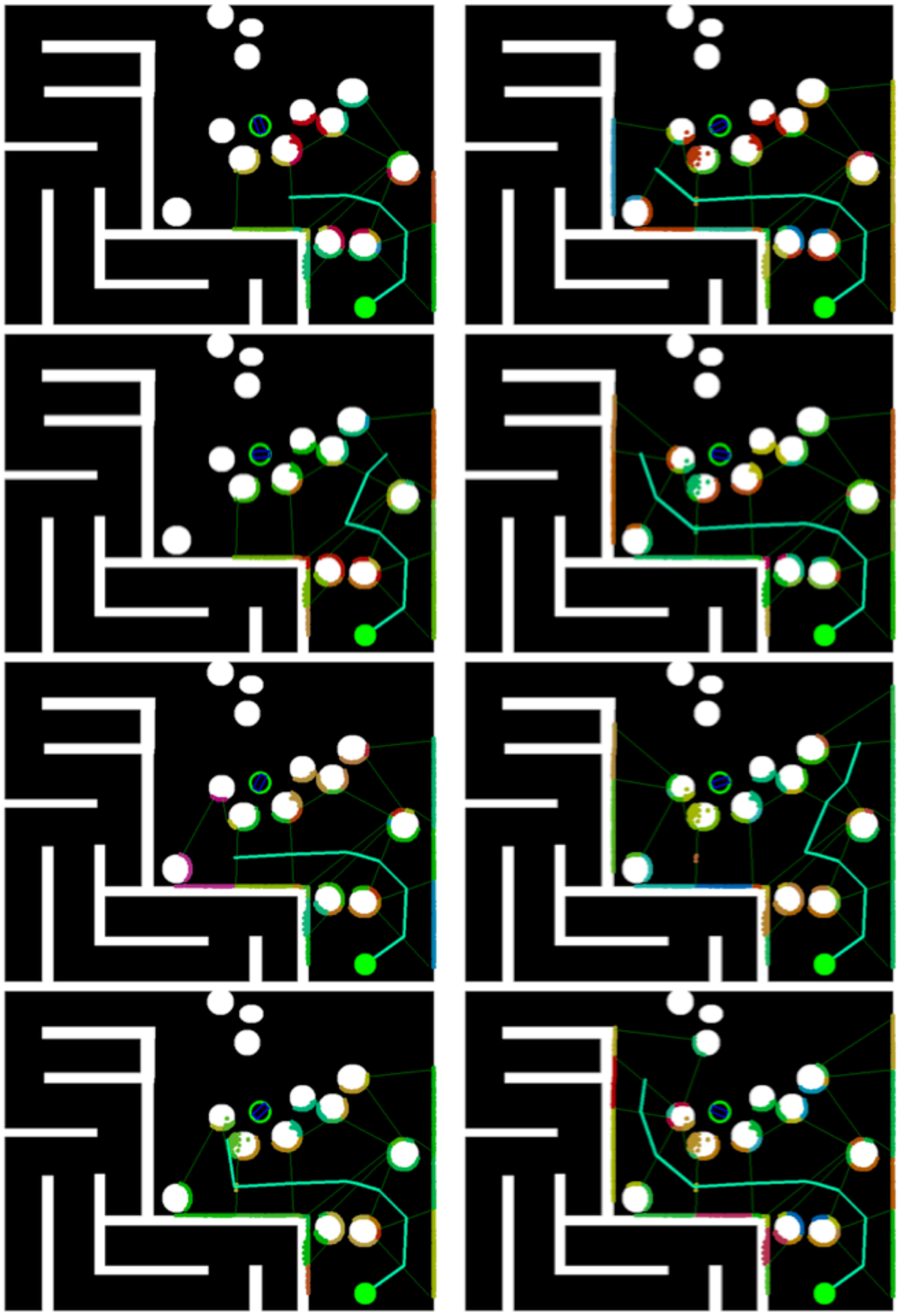
Figure 28. Shortest Opening Gateway Sequence Search 9-16

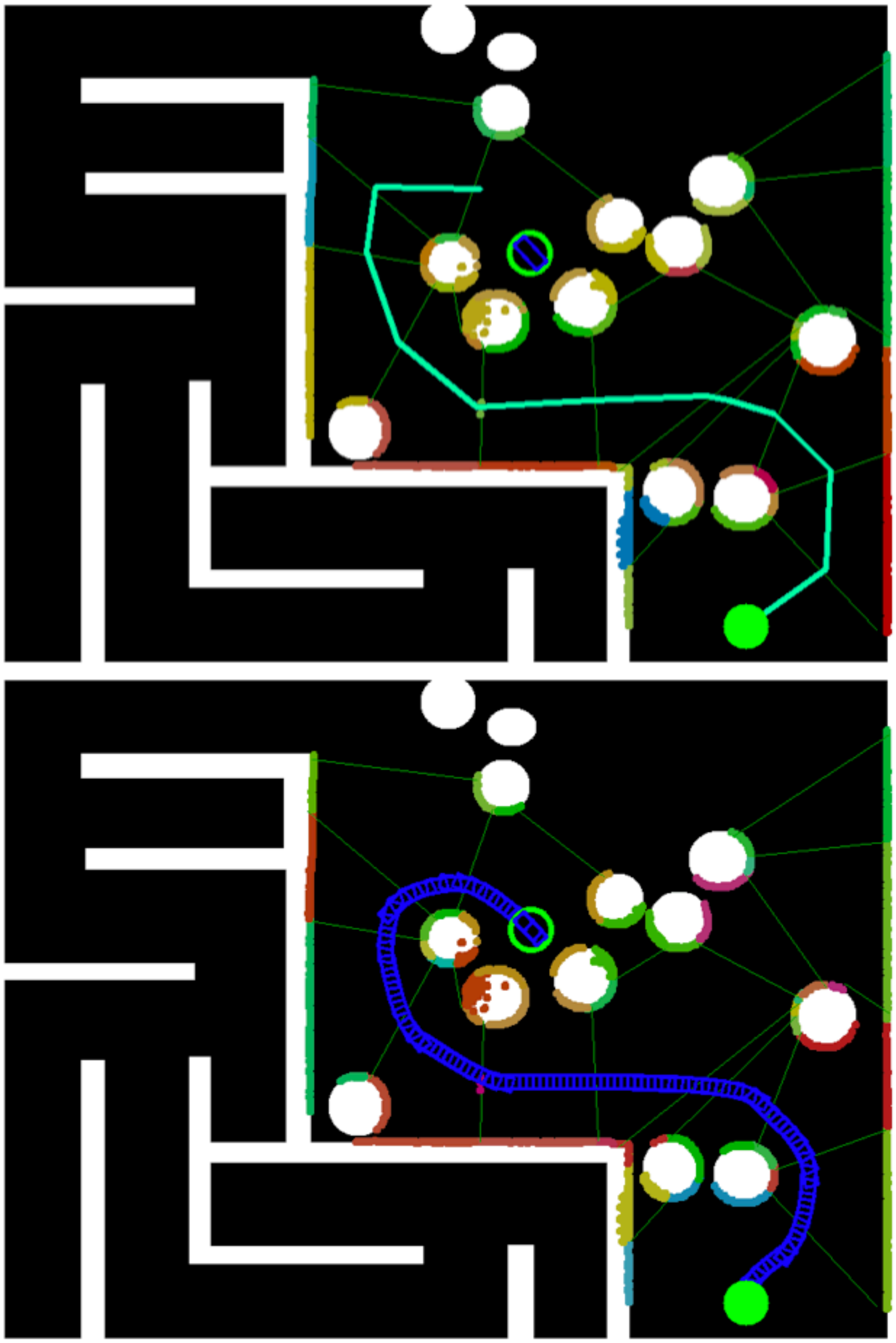
Figure 29. Shortest Opening Gateway Sequence and Simulated Motion Path

4.4 Comparison to RRT, RRT*, Informed RRT*

LIDAR A*'s performance was compared to some of the recently proposed algorithms for robotics path and motion planning, RRT, RRT*, and Informed RRT*. The algorithms were tested on a square maze map that has a size of 1000 pixels * 1000 pixels. Because the produced paths in sampling-based algorithms may vary between each calculation, a total of 10 experiments were conducted and their average performances were calculated. The RRT-based algorithms tested were based on kyk0910's implementation of the algorithms[40]. The results are summarized in the following path length and computation time tables.

Path Length (Pixels)	RRT	RRT*	Informed RRT*	Lidar A*
1	4548	3648	3716	4164
2	4851	3612	3804	4164
3	4842	3611	3789	4164
4	4723	3625	3811	4164
5	4704	3605	3762	4164
6	4738	3629	3738	4164
7	4768	3623	3783	4164
8	4398	3649	3721	4164
9	4882	3643	3773	4164
10	4679	3617	3731	4164
Avg. Length	4713.3	3626.2	3762.8	4164

Computation Time (ms)	RRT	RRT*	Informed RRT*	Lidar A*
1	43.4	1098	719	57.9
2	38.8	1025	584	56.2
3	39.3	931	684	54.4
4	47.7	961	661	57.3
5	31.5	1158	626	55.9
6	31.4	865	455	57.2
7	35.2	1109	616	55.1
8	33.9	845	639	59.1
9	45.3	1053	539	53.8
10	35.5	1010	670	55.1
Avg. Time	38.2	1005.5	619.3	56.2

4.5 Analysis

Three metrics of path quality, intelligence level, and computation efficiency are used to evaluate the performances of LIDAR A*, RRT, RRT*, and Informed RRT*. Results are shown in Figure 30, 31, 32, 33. When analyzing the results of RRT, RRT is very computationally efficient and can determine a path in less than 40ms. However, it generated low quality paths that feature inefficient paths averaging 4713 pixels in length with frequent unnecessary sharp turns that can be difficult for non-holonomic car-like vehicles to follow. RRT* algorithm, with the added select parent and rewire processes, produced much more efficient paths than the original RRT, averaging 3626 pixels in length which is near the shortest possible path toward the goal. However, it took an average of 1 second to compute, which can be too slow for applications that require real-time operation at this map size. Informed RRT* output slightly longer paths with close to 40% reduction in computation time which is a worthy trade-off and an improvement over RRT*.

The paths from RRT* and Informed RRT* tend to stay very close to obstacles. This issue can be addressed by pre-processing the map and finding the free space C_{free} to add necessary clearance for the vehicle. When dealing with the classic local minimum problem, RRT based algorithms should experience no difficulties because this obstacle arrangement would not prevent the tree from eventually growing out of the minimum region to find a solution. However, RRT based algorithms may have a difficult time efficiently finding smaller passages due to its reduced probability of adding samples in the specific passage region within the search space.

When comparing RRT based algorithms to LIDAR A*, the proposed LIDAR A* method has the smoothest path of the four algorithms with a slightly longer but safer average path length of 4164 pixels. Online motion simulation that follows the vehicle's kinematic model contributes to this result. It also has a very efficient planning time of 56.2ms that is sufficient for real-time operation. When encountering local minimum problems, LIDAR A* will eventually terminate the search branch in the minimum region when there's no neighboring region to be explored and start finding alternate branches that lead to the goal. The algorithm shouldn't have an issue finding small passages either as long as the passages have enough obstacle clearance to be considered opening gateways. They will be added and treated as any other open nodes in A* search algorithm and be explored when they have the least cost. The proposed method of LIDAR A* has the best all rounded performance in path quality, intelligence level, and computation efficiency.

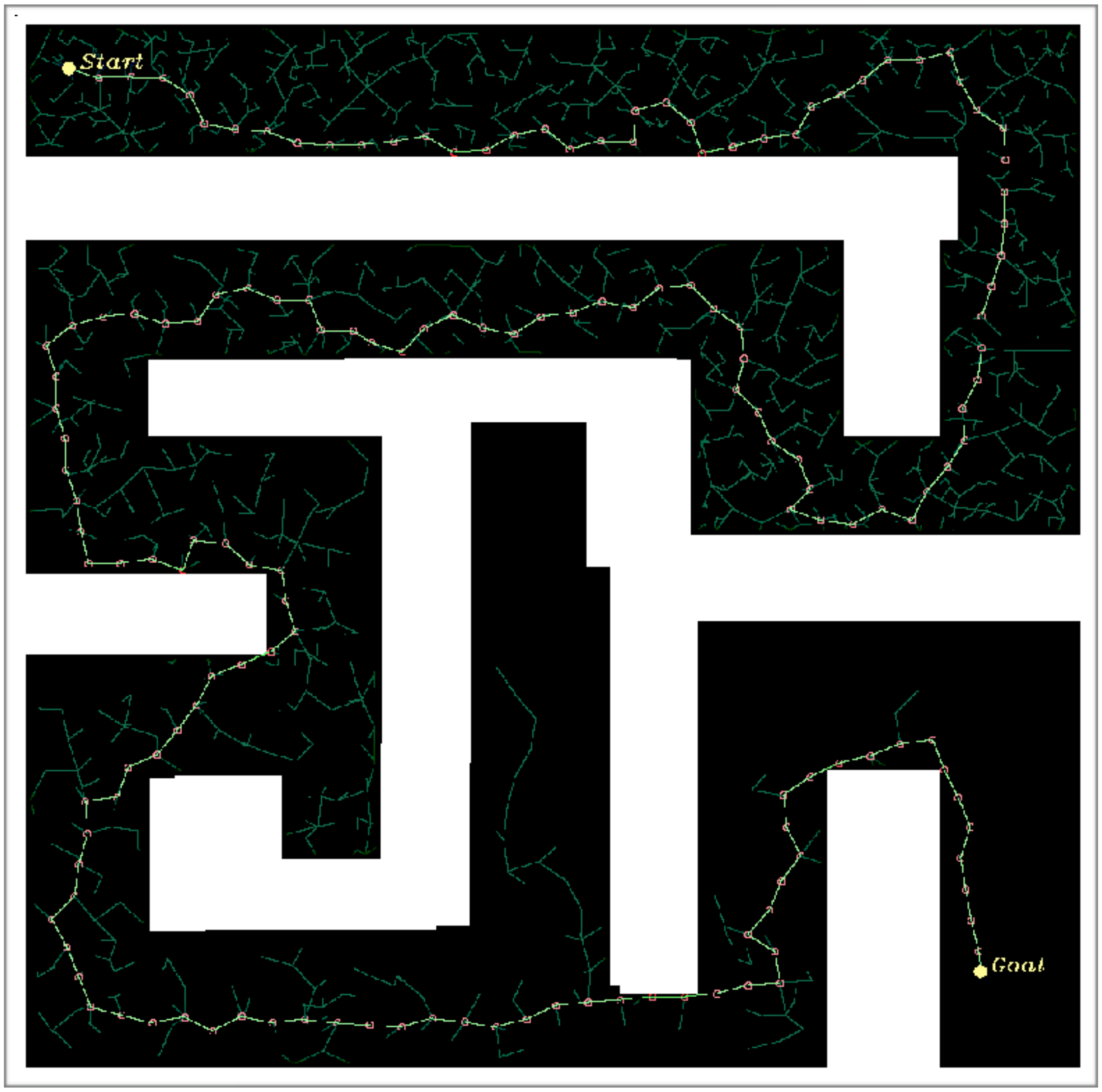

Figure 30. RRT on Maze Test. Average path length of 4713.3 pixels, Average computation time of 38.2ms

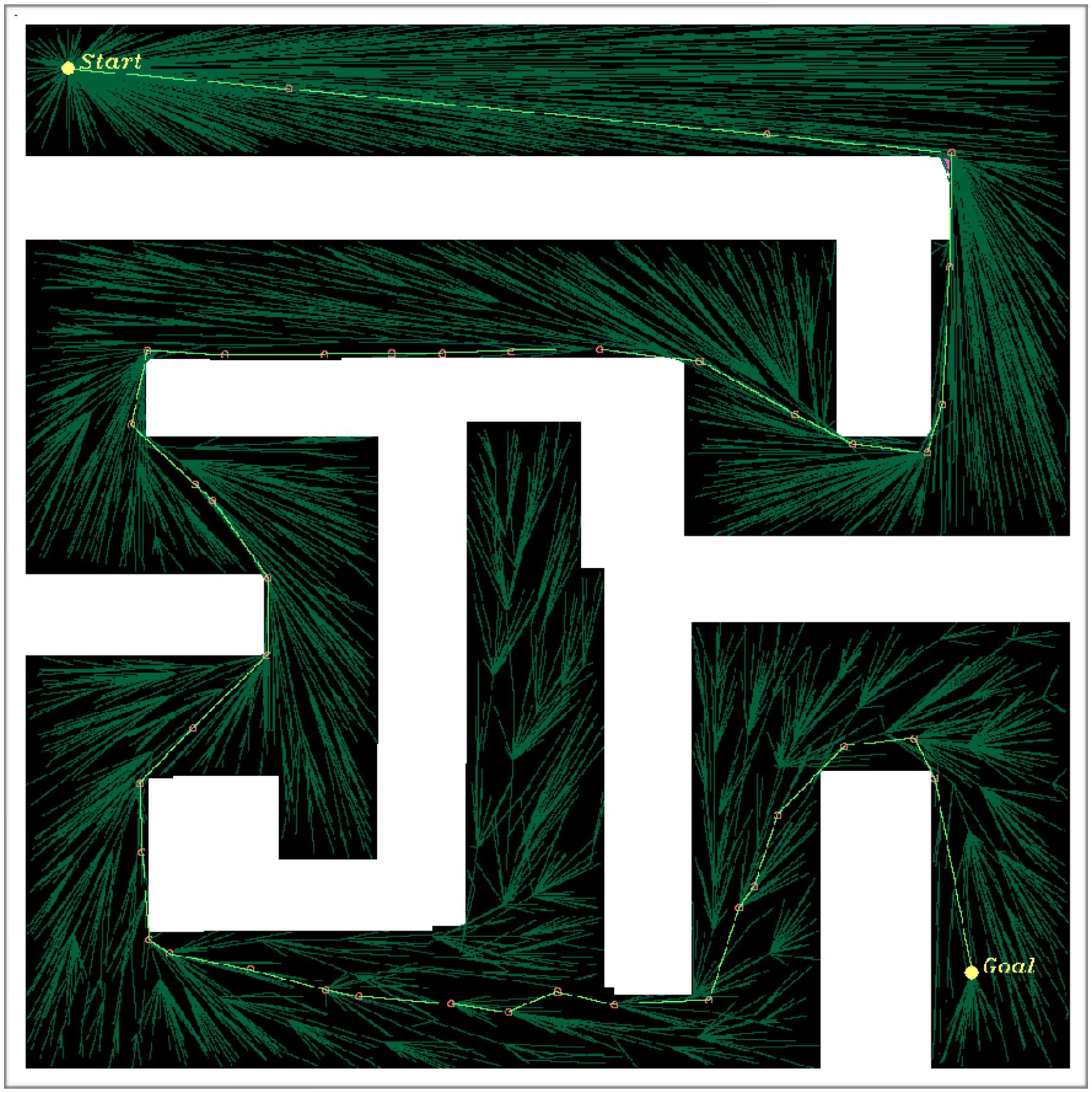

Figure 31. RRT* on Maze Test. Average path length of 3626.2 pixels, Average computation time of 1005.5ms

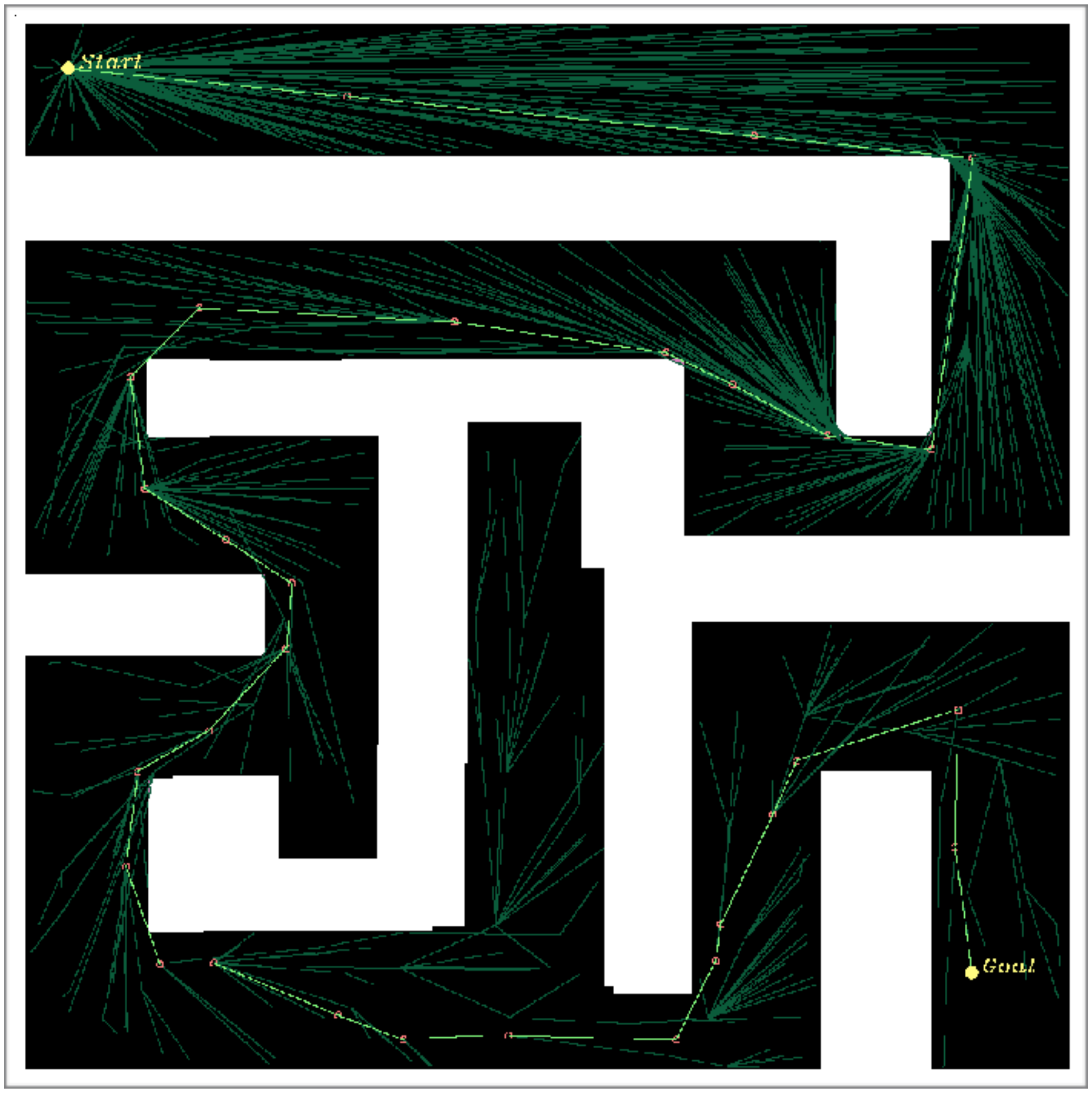

Figure 32. Informed RRT* on maze test. Average path length of 3762.8 pixels, Average computation time of 619.3ms

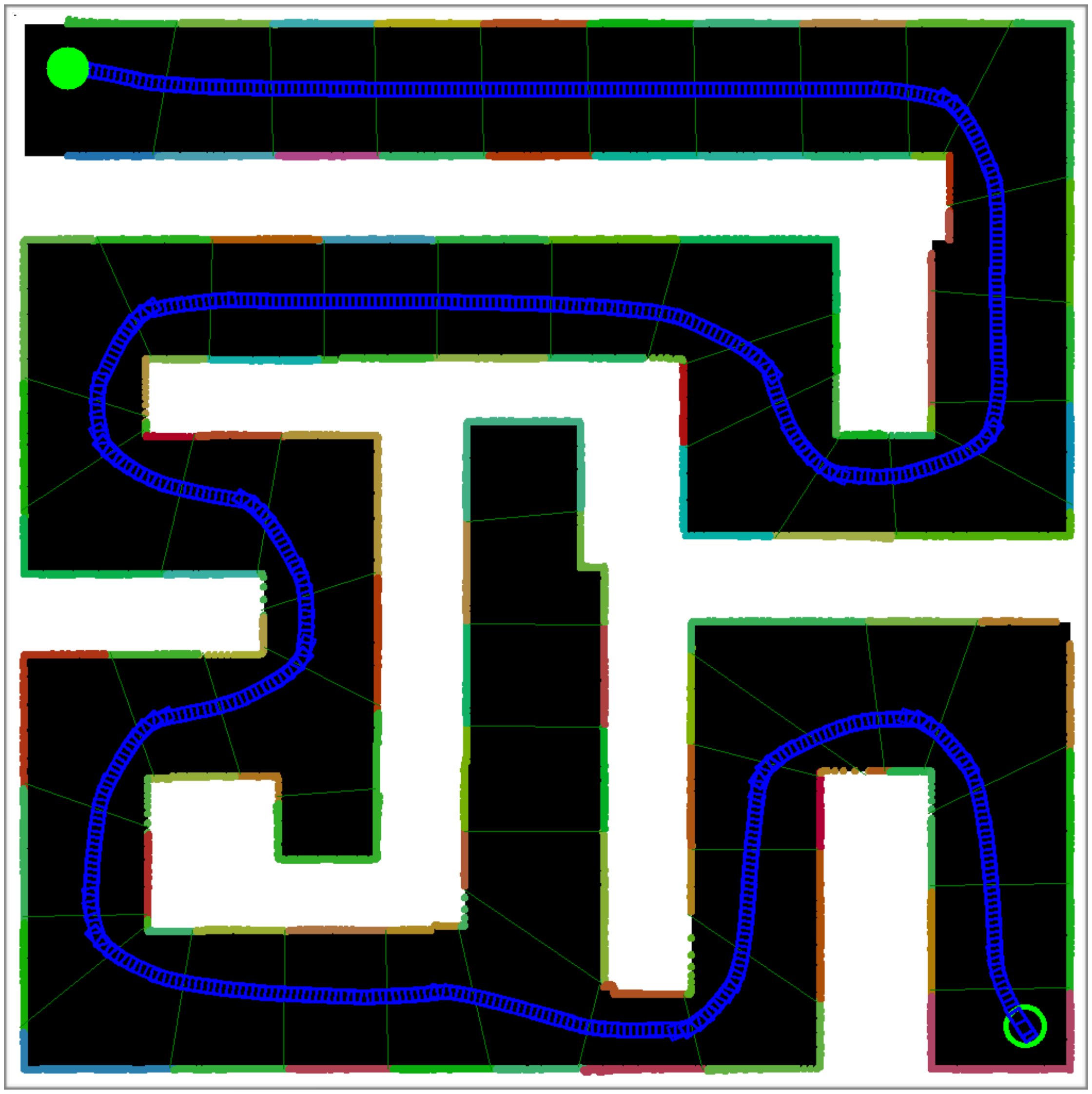

Figure 33. LIDAR A* on maze test. Path length of 4164 pixels, Average computation time of 56.2ms

4.6 Large Dynamic Map Test

LIDAR A* was also put into an extreme test of a large 1280*1024 pixels complex and dynamic maze to examine its ability to deal with difficult scenarios and to react to dynamic obstacle information. The program was coded to be able to add new obstacles in real-time using mouse cursor. Obstacles were intentionally added to actively close opening gateways and create narrow passages as shown in Figure 34, 35, 36. LIDAR A* was able to handle the change in map information and recalculate a smooth path in real-time. In the end, it took 56 ms to calculate a path of 5700 pixels in length. The trade-off for using a heuristic can also be observed in this experiment. Since LIDAR A* uses A* with euclidean distance toward the goal as heuristic, the algorithm attempted to find a path toward the goal (up) direction when there is a shorter path to the right of the vehicle's starting position. It shows that A* doesn't guarantee the shortest path as a trade-off for speed.

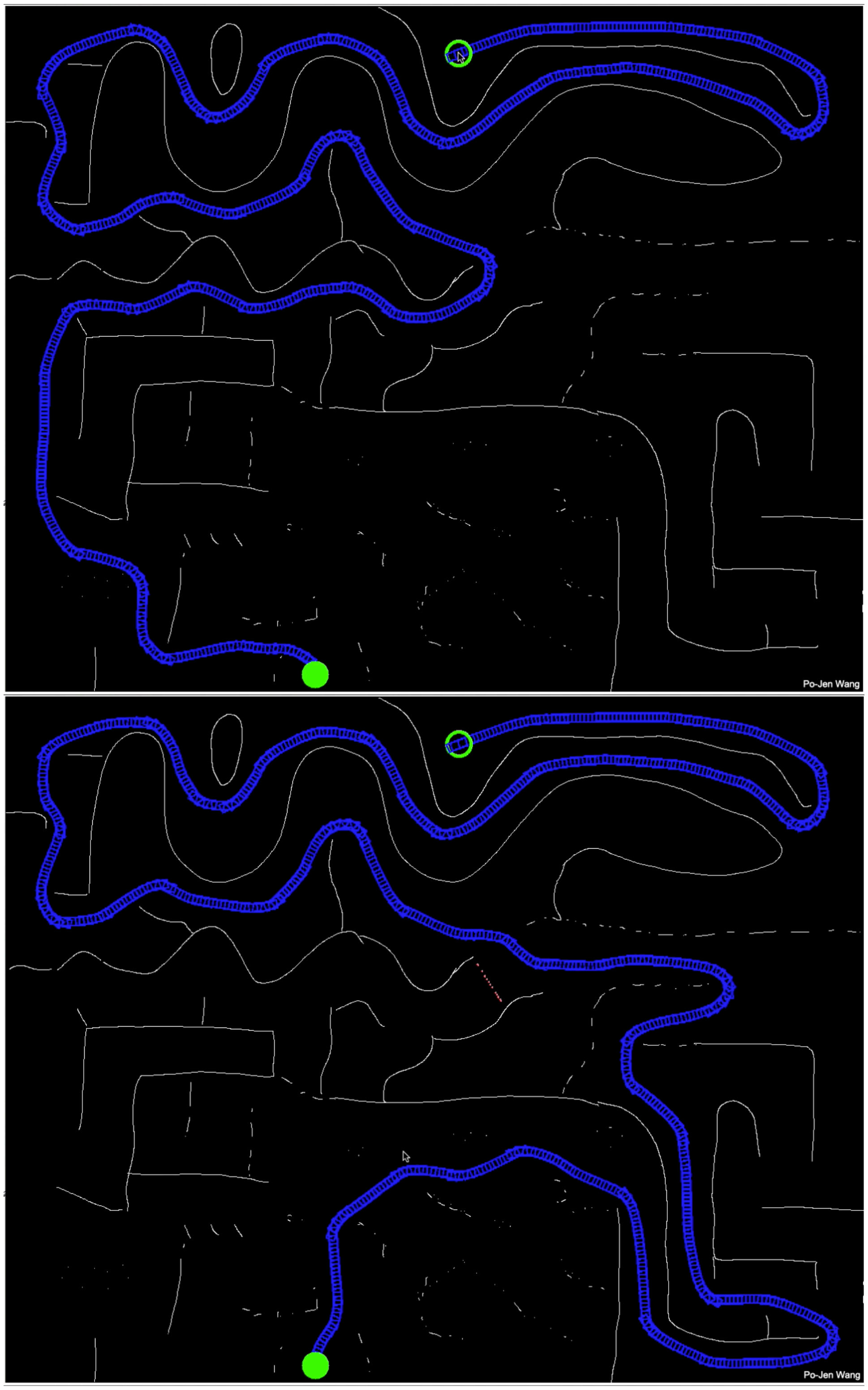

Figure 34. Dynamic Map Test 1. LIDAR A* updates the path in real-time as new obstacles were added to the map using the mouse cursor

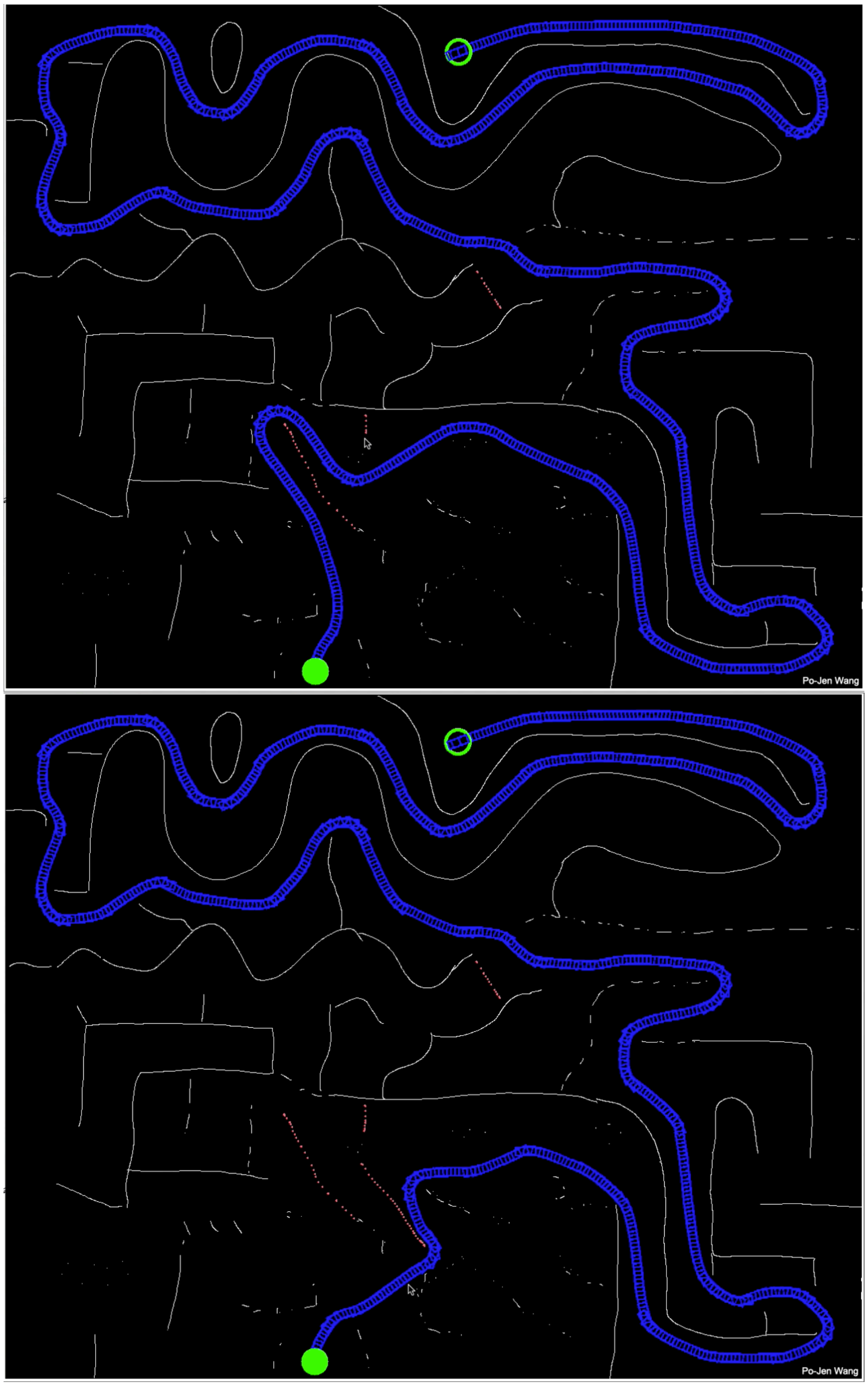

Figure 35. Dynamic Map Test 2

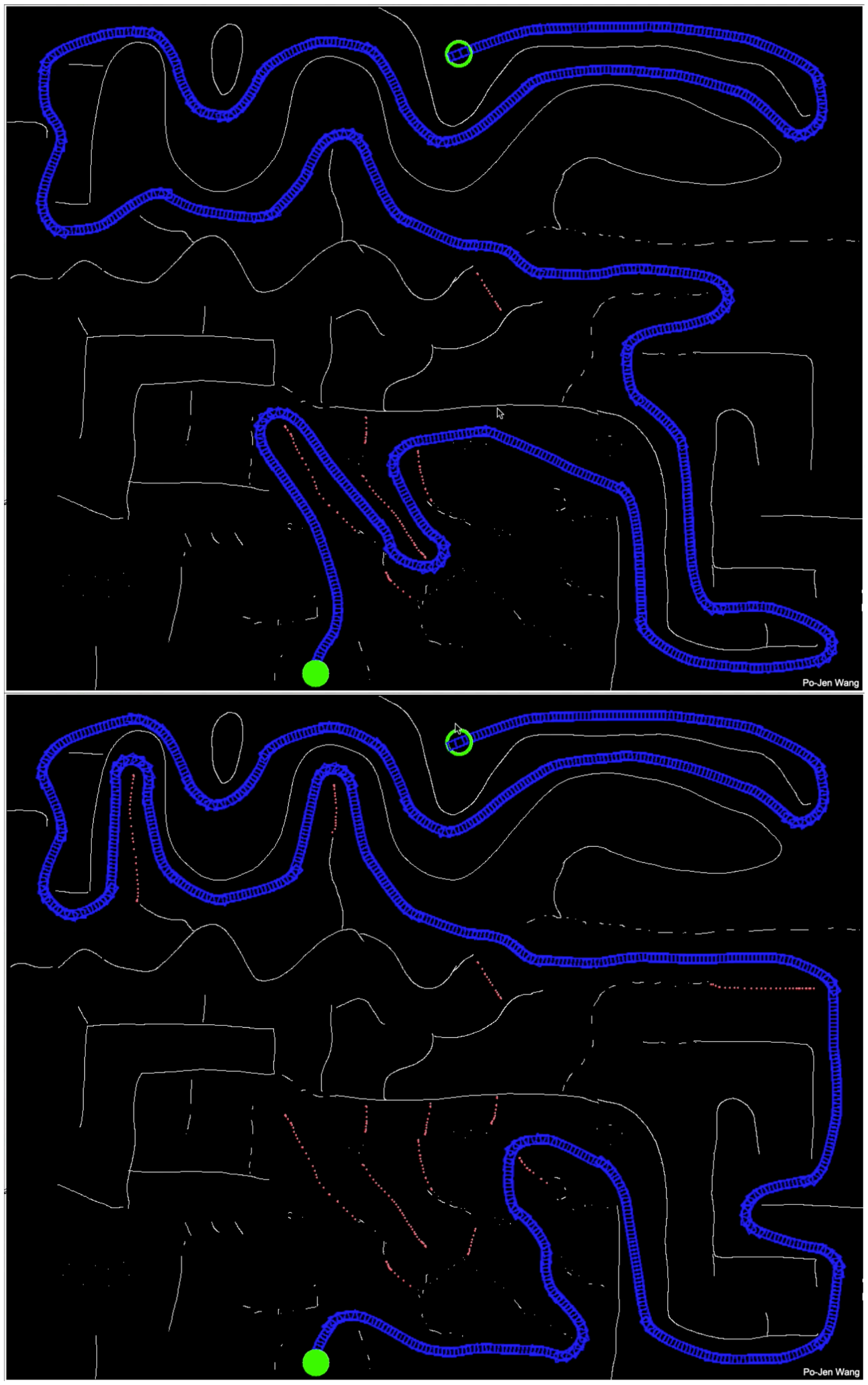

Figure 36. Dynamic Map Test 3

4.7 Discussion and Future Work

When designing the LIDAR A* algorithm, significant effort was put into maximizing the efficiency of the algorithm while maintaining a good quality of drivable path. The algorithm first divides a 2 dimensional map into local regions of simulated LIDAR scans with reduced data size of 181 polar coordinates. The scanned data are analyzed to find its possible transitions to neighboring regions. Through levels of abstractions, a regional map is abstracted and decomposed into a single node on an abstract graph where the search actually takes place. The significant reduction in the number of nodes helps the algorithm drastically improve its computational efficiency, overcome the curse of problem size in large graph search problems, and achieve long distance motion planning on a large high resolution map.

LIDAR A* has other advantages. First, the online decomposition and search process requires explorations to only a portion of the map necessary to complete the gateway search to reach the goal. It avoids decomposition on the entire map compared to HPA* and many other decomposition-based algorithms. Second, when the entire map is being abstracted and decomposed with static obstacles, the opening gateway graph may potentially be reused as an offline map which makes the online decomposition and search process unnecessary for some applications. Third, the generated opening gateway sequence can easily be utilized by other local motion planners not restricted to the one presented in this thesis. For example, one can use machine learning to train a model that specializes in traversing sequential opening gateways and use it for low level motion planning. Finally, when using a small grouping radius, the clustering algorithm becomes a segmentation tool for 2D LIDAR data that can be used to identify and track dynamic objects and make predictions.

LIDAR A* also has some shortcomings. The most challenging part of LIDAR A* is to correctly identify the valid traversable passages (opening gateways) in regions that are near the edge of the LIDAR's angular visible range (0 and 180 degree directions) where parts of the map are out of sight. The algorithm must ensure the passages to these opening gateways are reachable based on the vehicle's kinematic model and deal with possible corner cases.

In conclusion, LIDAR A* is a powerful global motion planner that features a hierarchical structure that efficiently performs online decomposition and search of consecutive neighboring regions to produce sequential gateways. The gateways guide the simulation of a simple yet robust local motion planner to generate a kinematic friendly path. Its highly optimized hierarchical approach significantly reduces the number of nodes required by the high level search, allowing the algorithm to produce a near optimal path on a highly complex large obstacle map and achieve real-time operation.

Bibliography

[1] Owen Holland. Exploration and high adventure: The legacy of Grey Walter. Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences 361, October 2003

[2] Ronald C. Arkin, Robin R. Murphy. Autonomous Navigation in a Manufacturing Environment. IEEE Transactions on Robotics and Automation Vol6. No. 4 August 1990

[3] Eric Krotkov, Reid Simmons. Perception, Planning, and Control for Autonomous Walking With the Ambler Planetary Rover. Intl. Journal of Robotics Research, Vol. 15, No. 2, pp. 155-180, April, 1996

[4] Sebastian Thrun, Michael Montemerlo, Hendril Dahlkamp. Stanley: The robot that won the DARPA Grand Challenge. Journal of Field Robotics 23(9), 661-692, 2006

[5] Martin Buehler, Karl Iagnemma, Sanjiv Singh. The DARPA Urban Challenge, Autonomous Vehicles in City Traffic. Springer 2009

[6] Automated Vehicles for Safety. National Highway Traffic Safety Administration

[7] Michele Bertoncello, Dominik Wee. Ten ways autonomous driving could redefine the automotive world. McKinsey&Company

[8] Todd Litman, Autonomous Vehicle Implementation Predictions. Implications for Transport Planning, Victoria Transport Policy Institute. June 5, 2020

[9] Gregor Klančar, Andrej Zdešar, Sašo Blažič, Igor Škrjanc. Wheeled Mobile Robotics, From Fundamentals Towards Autonomous Systems. Page 161-206. 2017

[10] Sebastian Thrun, Jens-Steffen Gutmann, Dieter Fox, Wolfram Burgard, Benjamin J. Kuipers. Integrating Topological and Metric Maps for Mobile Robot Navigation: A Statistical Approach AAAI-98 Proceedings 1998

[11] Sebastian Thrun. Learning metric-topological maps for indoor mobile robot navigation Artificial Intelligence, Volume 99, Issue 1, Page 21-71, Feb 1998

[12] Tomas Lozano-Perez. Spatial Planning: A Configuration Space Approach, IEEE Transactions on Computers, Vol. C-32, No. 2, Feb 1983

[13] Karl Kurzer. Path Planning in Unstructured Environments. A Real-time Hybrid A* Implementation for Fast and Deterministic Path Generation for the KTH Research Concept Vehicle. KTH Royal Institute of Engineering Sciences. 2016

[14] Tomas Lozano-Perez and Michael A. Wesley. An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles. IBM Thomas J. Watson Research Center. 1979

[15] Nilsson, N.J. A Mobile Automaton: An Application of Artificial Intelligence Techniques. Stanford Research Institute. Jan 1969

[16] J. Borenstein, Y. Koren, The Vector Field Histogram - Fast Obstacle Avoidance For Mobile Robots. IEEE Journal of Robotics and Automation Vol 7, No 3, pp. 278-288, June 1991

[17] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1(1):269–271, 1959

[18] Deep Medhi, Karthik Ramasamy. Routing Algorithms: Shortest Path, Widest Path, and Spanning Tree. Network Routing, Algorithms, Protocols, and Architectures. A volume in The Morgan Kaufmann Series In Networking, 2018

[19] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions of Systems Science and Cybernetics, Vol. SSC-4 No.2, July 1968

[20] Steven M. LaValle. Searching for Feasible Plans. Planning Algorithms. Cambridge University Press, 2006

[21] E. A. Silver. An Overview of Heuristic Solution Methods. The Journal of the Operational Research Society Vol. 55, No. 9 pp. 936-956, Sep., 2004

[22] Cipriano Galindo, Juan-Antonio Fernandez-Madrigal, Javier Gonzalez, Improving Efficiency in Mobile Robot Task Planning through World Abstraction, IEEE Trans. On Robotics, Vol. 20, No. 4, 2004

[23] Juan-Antonio, Fernandez-Madrigal, Javier Gonzalez, Multihierarchical Graph Search. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 24, No. 1. January 2002

[24] Adi Botea, Martin Muller, Jonathan Schaeffer. Near Optimal Hierarchical Path-Finding. Department of Computer Science, University of Alberta Edmonton. 2004

[25] Le Minh Duc,[1] Amandeep Singh Sidhu,[1] and Narendra S. Chaudhari. Hierarchical Pathfinding and AI-Based Learning Approach in Strategy Game Design. International Journal of Computer Games Technology, Cyber Games and Interactive Entertainment. 2006

[26] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, James Diebel. Practical Search Techniques in Path Planning for Autonomous Driving. Stanford University. 2008

[27] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo and James Diebel. Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments. The International Journal of Robotics Research. 2010

[28] LaValle, Steven M. Rapidly-exploring random trees: A new tool for path planning. *Technical Report*. Computer Science Department, Iowa State University. October 1998

[29] Howie M. Choset, Seth Hutchinson, Kevin M Lynch, George Kantor, Wolfram Burgard, Lydia E. Kavraki, Sebastian Thrun. 5.1 Visibility Maps: The Visibility Graph, Principles of Robot Motion: Theory, Algorithms, and Implementation. The MIT Press. 2005

[30] Sertac Karaman, Emilio Frazzoli. Sampling-based Algorithms for Optimal Motion Planning. International Journal of Robotics Research. 2011

[31] Jonathan D. Gammell. Siddhartha S. Srinivasa, Timothy D. Barfoot. Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsodal Heuristic. 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014), pp. 2997-3004, 14-18. Sept. 2014

[32] Oded Maron, Tomas Lozano-Perez, Visible Decomposition: Real-Time Path Planning in Large Planar Environments. Artificial Intelligence Lab. NE43-755. 1996

[33] Qidan Zhu, Yongjie Yan, Zhuoyi Xing. Robot Path Planning Based on Artificial Potential Field Approach with Simulated Annealing, Sixth International Conference on Intelligent Systems Design and Applications

[34] Iram Noreen, Amna Khan , Zulfiqar Habib, A Comparison of RRT, RRT* and RRT*-Smart Path Planning Algorithms, IJCSNS International Journal of Computer Science and Network Security, VOL.16 No.10, October 2016

[35] Po-Jen Wang, Nicholas R. Keyawa, and Craig Euler. Radial polar histogram: obstacle avoidance and path planning for robotic cognition and motion control, Proc. SPIE 8301, Intelligent Robots and Computer Vision XXIX: Algorithms and Techniques, 83010Y, January 23, 2012

[36] Nicholas R. Keyawa, Po-Jen Wang, RED RAVEN, RED Robotic Autonomous Vehicle Engineered at Northridge, IGVC Design Report, 2011 http://www.igvc.org/design/2011/CSU-Northridge%20-%20Red%20Raven.pdf

[37] Yan Peng, Dong Qu , Yuxuan Zhong, Shaorong Xie, Jun Luo. The Obstacle Detection and Obstacle Avoidance Algorithm Based on 2-D Lidar, Proceeding of the 2015 IEEE International Conference on Information and Automation. Lijiang, China, August 2015

[38] Rodney Brooks, Tomas Lozano-Perez. A Subdivision Algorithm in Configuration Space For Find-path With Rotation. IEEE Transactions on Systems, Man and Cybernetics, SMC-15(2):224-233, 1985

[39] Howie M. Choset, Seth Hutchinson, Kevin M Lynch, George Kantor, Wolfram Burgard, Lydia E. Kavraki, Sebastian Thrun. 5.2.1 GVD Definition, Principles of Robot Motion: Theory, Algorithms, and Implementation. The MIT Press, 2005

[40] ky0910, sampling-based-planners. GitHub repository. https://github.com/kyk0910/sampling-based-planners

[41] Steven M. LaValle. 13.1.2.1 A simple car. Planning Algorithms. Cambridge University Press, 2006.

[42] Rymsha Siddiqui. Path Planning Using Potential Field Algorithm. Medium. July 2018