PATH PLANNING AND FOLLOWING FOR AUTONOMOUS VEHICLES AND ITS APPLICATION TO INTERSECTION WITH MOVING OBSTACLES

By

MINCHEUL KIM

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2017

To my wife, Minji Kim

ACKNOWLEDGMENTS

I thank my advisor Dr. Carl Crane for his kind help and advice during my graduate education. I also thank Dr. Gloria Wiens, my committee member, for her help and advice.

I also thank my colleagues and friends at CIMAR member, Neal Patrick and John Esposito. As working on the FDOT project together, they gave their first steps and guidance on my research journey. We went very well and had a good time.

I also thank Minji Kim, my wife. Thanks to her dedicated help for two years, my research ended well.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

PATH PLANNING AND FOLLOWING FOR AUTONOMOUS VEHICLES AND ITS APPLICATION TO INTERSECTION WITH MOVING OBSTACLES

By

Mincheul Kim

May 2017

Chair: Carl D. Crane III
Major: Mechanical Engineering

The algorithm to find the optimal route to the destination is the most necessary part for an autonomous vehicle. Much research has been conducted for the optimal path search problem. These researches can be divided into offline path planning and online path planning.

However, each path planning algorithm has conflicting prerequisites, advantages, and disadvantages. Although offline path planning guarantees optimal path to the destination, it cannot run without a prior information about the world before the path search. Online path planning is suitable for dynamic environments, but cannot guarantee an optimal result. Therefore, there is a need for an algorithm that allows the autonomous vehicle to reach its destination successfully in a dynamically changing traffic environment, taking advantage of two different algorithms above.

First, existing algorithms for path finding will be reviewed. Among them is the A* algorithm which is widely used for optimized path search, and its related search will be discussed. Second, mapping algorithms for applying a path search algorithm will be discussed. Finally, a review of the studies designed to avoid moving obstacles will be presented.

Based on the established studies, this research will introduce the dynamic local search area and utilize the offline path planning efficiency and the dynamic environment correspondence function of online path planning. This local search area is defined as a limited search range centered on the autonomous vehicle and guarantees the optimal route to the destination through the route search to the replaced temporary target within this range. In addition, we will introduce an algorithm to track the optimized path, which is designed to perform the action to detect the obstacles while following the path and to respond appropriately to evade the obstacles.

The developed algorithms have been tested through simulations. The results of the tests and future improvements to this research are presented.

CHAPTER 1
INTRODUCTION

Motivation

Since the production of the first gasoline car developed in Germany in 1885, the car made our life more enriching thanks to the remarkable speed of development. With faster speeds and increased loads, one can send more loads faster and further through it. According to a survey in 2014, there are 7.9 million vehicles in the United States and the rate of growth is gradually increasing.

The advancement of technology has further advanced to the point that it allows the car to move without human command. We call this an autonomous vehicle. Autonomous vehicles are also called unmanned vehicle, driverless cars, self-driving cars, or robotics cars, which means an automobile that can operate automatically without human control. Autonomous vehicles travel only by designating its own position and surrounding environment with sensors, such as radar, Light Detection and Ranging(LIDAR), Global Positioning System(GPS), Inertial Navigation System(INS), and vision cameras. Indeed, there is already a growing number of autonomous vehicles in the military, as well as Google cars, which are now leading the automotive technology, or Tesla's cars, now in commercial production.

However, irrespective of the technical development of these vehicles, traffic accidents caused by collisions between cars are not easily reduced. According to the data for 2015, the total number of drivers who died in the United States is 35,092. This is a phenomenon that cannot be solved even in a conventional vehicle operated by a human driver, and an autonomous vehicle that will appear in the future also has the same problem that must be solved.

As we have already seen, there is always the risk of unexpected movement of moving obstacles like conventional cars that people are driving. In most cases, an operation to avoid such moving obstacles is performed at the discretion of the driver who operates the vehicle. Such a driver 's judgment may be accumulated based on experience, or it may cause a momentary reflex to take action to move in the opposite direction to the current operation. However, it is difficult to safely avoid moving obstacles in this way. Even now, when the era of autonomous vehicles is about to arrive, computers cannot behave like human beings or experience immediate reflexes, or hunches. The computer also cannot assure that the identified obstacles will anticipate the risk of the collision and make the appropriate evasive maneuver.

Now, to develop effective algorithms for autonomous vehicles that are adaptive to dynamically changing traffic conditions, it is important to know how the various objects are expected to be positioned and how they behave and how efficient paths are performed within the optimized time and search space. To do this, a brief review of the existing path planning algorithms is now presented.

Path Planning Algorithm

The path planning algorithm can be divided into an offline path planning algorithm and an online path planning algorithm.

The offline path planning algorithm is, in other words, the global path planning algorithm. Offline path planning needs a priori information about the whole world, and after calculating the most optimal path, it communicates the path to the robot and the robot follows the path with the help of the path tracking algorithm. Therefore, the offline pass algorithm is suitable for creating the most optimal algorithm for reaching the goal. However, this offline path algorithm needs to be given prior information completely for

path planning, and the larger the search range, the greater the computational speed and space that is required. Also, it cannot appropriately respond to a dynamically changing environment.

The online path planning algorithm is, in other words, a local path planning algorithm. This is done in real time. This algorithm contrasts with the above case where the robot has a prior map information and is used when an unexpected obstacle appears or when the surrounding environment changes. For this path planning, the robot needs to detect the environment using onboard sensors in real time. Thus, the online path planning algorithm is an effective algorithm in a changing dynamic environment. However, due to limited computational capability (memory size, time efficiency, sensor range limitation), the algorithm uses a simplified algorithm or heuristic algorithm for path finding. Therefore, the online path planning algorithm is unlikely to produce an optimized algorithm.

It is necessary to combine the two algorithms properly to search the path to reach the goal in the dynamic environment including the moving obstacles. In the next sections, the problem will be defined and an approach will be devised to solve the problem.

Research Goal

It is important to find a generic and reliable path planning algorithm that can be applied to any type of autonomous vehicle to reach the goal in a rapidly changing traffic environment. The purpose of this study is to simulate various situations that are difficult to realize and to apply the appropriate algorithm.

Problem Statement

The goal of this research is to devise an algorithm that produces an optimal path in time to reach the goal in a dynamic environment with moving obstacles. The following elements are given or assumed to achieve this goal:

- The autonomous vehicle must autonomously calculate the route after receiving only the information of the destination without the input of any external computation.
- The position and orientation of the vehicle shall be measured in the global coordinate system using GPS and INS. These global coordinates are used to convert local information into global information.
- The sensor for recognizing the environment of the car is the LIDAR or vision camera mounted on the car. In this case, since each sensor has a limited range of sensor detection and the vehicle can obtain necessary information only within the range.
- Autonomous vehicles must be driven by realistic inputs. It must be manipulated only with the steering input to manipulate the heading of the car, and with the throttle and brakes responsible for the acceleration/deceleration of the car. Sudden side movements or jumps are prohibited.
- An autonomous vehicle must select the best route to reach its destination using the information obtained.

Development

In the above problem statement, the solution to arrive at the destination is as follows. First, autonomous vehicle modeling will be applied to the algorithm and position and speed information of the vehicle will be obtained. The onboard sensors observe the surrounding terrain information and the behavior of other vehicles moving around. Based on this information, the world environment is modeled by a grid and the cost of each grid cell is calculated. The cost map is searched for the optimal route to the goal. The path planning algorithm creates an optimal path to the goal based on the cost map and hands it to the autonomous vehicle. The path following algorithm is applied from the

starting point to the goal point based on the path. The following presents a novel idea to solve this problem:

- A new path planning algorithm has been devised to allow the autonomous vehicle to reach its destination in a dynamic environment. This algorithm searches the optimized route within a limited range based on the limited local range of each sensor. It then repeats the same route search according to the movement of the vehicle to arrive at the destination. To find optimal path in the range, an improved A* algorithm is used.

- A new cost calculation algorithm is introduced to reflect the information of the observed dynamically moving vehicle to the above improved path finding algorithm. This cost is determined by the current location and velocity of the obstacle, and this cost is fused with the cost map, affecting the path tracking behavior of the autonomous vehicle.

- A simulation environment is created that can test the above algorithm. This allows for visualization of the input and output in a GUI environment. A keyboard or mouse may be used for the operation of the autonomous vehicle, and a mouse may be used for the destination input.

CHAPTER 2
REVIEW OF LITERATURE

In this section, a review of path planning algorithms that have been studied for autonomous vehicles will be presented. First, the offline algorithm that plans from the given starting point to the destination point will be discussed, the most common of which is the A* algorithm. Next, algorithms which act to avoid dynamic obstacles will be reviewed

Path Finding Algorithm

Dijkstra Algorithm

The Dijkstra algorithm is an algorithm for finding the shortest path between nodes in a graph [Dijkstra 1959]. A graph is a type of data structure that consists of vertices and edges connecting vertices. This graph data structure is widely used in path finding algorithms because it can be applied to cities and roads problems.

This Dijkstra algorithm is applied to negative weightless graphs and is used for most problems finding the fastest possible path with the least possible cost. The Dijkstra algorithm is a method of determining the shortest distance by calculating the distance by visiting the neighboring node by looping.

However, the disadvantages of the Dijkstra algorithm are as follows. The Dijkstra algorithm nodes all distances that can travel between paths. So, all directions of the origin node must be searched in the network. Therefore, to apply this, the space that needs to be searched will become larger and the time to execute the algorithm will become longer. In addition, making the network can be slow due to various variables such as motor vehicle congestion and work attendance. Therefore, extended version of the Dijkstra algorithm, such as the A* algorithm were developed.

A* Algorithm

The A* algorithm is the search algorithm proposed in 1968 by Peter Hart et al. The A* algorithm finds the shortest path from the given starting node to the target node in a graphical data structure. The A* algorithm uses a heuristic estimation function to estimate the best path when passing through each node. Although this A* algorithm does not always guarantee an optimal path, the optimal path can be detected by appropriately selecting the above heuristic estimator function. The following is a cost evaluation function of the A* algorithm.

$$f(n) = g(n) + \mathrm{h(n)} \tag{2-1}$$

$f(n)$: cost evaluation function for nth node

$g(n)$: cost of the current state

$\mathrm{h(n)}$: cost when moving from the current state to the goal state

When finder moves from the current state to the next state, it first searches for the point where f is minimized. Therefore, the performance is clearly depending on the heuristic function h, and in the case of $f = g$, it is the same as the Dijkstra algorithm.

Hereinafter, the execution process of the A* algorithm is briefly as follows:

1. Put adjacent nodes found in the starting rectangle into the open list.
2. Find the lowest f cost in the open list and select it as the current node.
3. Take it out of an open list and put it into a closed list.
4. For the adjacent node to the current node, if it is inaccessible or is on a closed list, ignore it, otherwise continue. If it is not in the open list, add it to the open list and make the parent of this node the current node. Record f, g, h costs for this node. If it is already in the open list, use g cost to find out if this node is better, and if its g cost is smaller, it means that it is a better length, so change the parent node to the node. Then recalculate the g and f costs of the node
5. Add a target node to an open list. If open list became empty (in this case, it fails to find the target node, so there is no route to reach), we should stop.

6. Save the path (going from the target node to the parent node until we get to the starting node, this is the path we are looking for).

Other Path Planning Algorithms

A. Stentz commented about the D* algorithm in his book [Stentz 1994]. This algorithm was introduced to improve the problem of the existing A* algorithm. The A* algorithm has a problem that it takes a long time because it stops the process when the path search is blocked and calculates the path again from that point. The D* algorithm introduces the concept of *backpointer* to optimize the search speed and space by recalculating the path back to the previous *backpointer* state when the path becomes clogged.

P.Shamsinejadsm saw that an exhaustive, complete, comprehensive search technique cannot navigate the optimal path within a given time [Shamsinejadsm 1959]. He distinguishes between global and local path navigation. The global method is suitable for obstacles to be static and the search target to find the optimal path. On the contrary, the local method is suitable for dynamic dynamical environments requiring only information of local obstacles. However, the path created by the local method is only valid for the local optimal path.

P.Shamsinejadsm devised a local path search based on a genetic algorithm to obtain high accuracy such as global route search, and at the same time to achieve proper execution speed such as local path search. A genetic algorithm is an algorithm based on the biological genetic theory of nature. It is a method to obtain an optimal solution through repetition of selection/crossover/mutation/substitution processes.

He expected the speed of navigation to be very important, and found genetic structures to be appropriate for complex pathways. He also found that the length of the

chromosome fixed in the path search through the genetic algorithm is simpler and enables faster genetic function, but it does not show the complicated path as a disadvantage. However, he assumed that the chromosomes can be stretched freely.

By applying this genetic algorithm, E. Zawodny devised an algorithm that ensures that multiple vehicle can be positioned with each other maintaining the line of sight without obstacles interfering [Zawodny 2003]. This algorithm was compared with the exhaustive search algorithm, and it produced the optimum path for the shorter execution time and thus it is very helpful for the location selection for direct communication among multiple vehicle.

Mapping Algorithm

An autonomous vehicle must design a map to apply the path planning algorithm with given information of the surrounding environment or measured information with its onboard sensors. Sometimes there is a lack of the priori information or limitation of measured sensor data. Past researches on how to map the real world so that the path planning algorithm can be applied is now reviewed.

Occupancy Grid

The Occupancy Grid is an algorithm that creates a map by utilizing the position of a robot together with indeterminate sensor data information.

The concept of Occupancy grids was developed in 1989 by Carnegie Mellon University to provide accurate path mapping and navigation for autonomous robots. This concept incorporates various environmental variables as a background to the occupancy grid [Elfes 1989]. In other words, the robot can fuse each piece of information to one large world map, depending on the sensor data it collected without the precompiled designated path. The occupancy grid map represents 2D or 3D spatial

information as a set of individual cells called a grid. This grid contains a specific state variable that indicates whether an obstacle is present in the grid, which is either occupied or empty, and the two states combined to yield a probability of 1. The judgment of these characteristics is obtained by applying the Bayes Law through continuous observation. Therefore, the robot can perform the obstacle avoidance using the A* algorithm with probability that each cell has an obstacle.

The Occupancy grids discussed above are models in a static environment. Therefore, Occupancy grids are not suitable for applying to the dynamic of the real world. The concept of a temporal occupancy grid was further developed following the traditional occupancy grid to solve the issue.

The Temporal Occupancy Grid (TOG) proposed by Daniel Arbuckle and others in 2002 is designed to classify the characteristics of objects that temporarily occupy space in the environment [Arbuckle 2002]. TOG is method that can classify the occupancy of grid cells according to time, including time data in the existing spatial coordinate information. In other words, traffic patterns, including the space occupied by fast moving objects and spaces occupied by slower objects, can be obtained differently through TOG.

As TOG is different from Occupancy Grid, the way to distinguish between static obstacles and dynamic obstacles is as follows. Static obstacles are assigned a high occupancy probability over all time scales. Conversely, a cell with a high expectation of a dynamic obstacle would have a high occupancy probability for a short time scale, or a low occupancy probability for a long-time scale.

Traversability Grid

There are cases in which such occupancy grid and temporal occupancy grid are applied as actual cases. In the Urban Navigator of the University of Florida, it created a grid-based map through a smart arbiter which fuses sensor data [Crane 2005]. This concept - the so-called ‘Traversability Grid’- focuses on describing the condition of the terrain, and the class of obstacles, according to the fluctuating cycle in real time [Crane 2006]. The map consists of a 121x121 set of horizontal and vertical cells, each of which represents 0.5m of the real world. The central cell (60,60) showed the measured center position of the car, and the north was always aligned with the grid. Therefore, the map could track information 30m by 30m, and all smart sensor components needed to be synchronized with latitude/longitude, heading information received from the Global Position Sensor(GPOS) component. Costs are given from 2 to 12; 2 is a road that cannot be traversed, 12 represents the optimal path, and 7 represents the normal state, which may or may not be good for the sensor. These values are also colored and comprehensively described in a GUI. These grids are then sent to smart arbiter, which combines this information into a single cost map.

Avoid Obstacles

So far, methods for world modeling for application to the optimal path algorithm have been discussed. The next thing is to see how moving objects can be avoided in the world when they are encountered.

Avoiding moving obstacles is more difficult than avoiding static obstacles. Because the position and velocity information on moving obstacles is constantly changing, it is difficult to devise an algorithm that calculates the path in real time,

keeping pace with the dynamically changing variables. Thus, most past studies are focused on estimating or measuring the path of moving obstacles.

Previous studies have introduced probabilistic models to explain the dynamic behavior of obstacles to solve this problem. Another approach was the space-time concept. Space-time theory was created by adding additional time dimensions to existing spatial dimensions. In space-time theory, obstacles have been replaced by static obstacles, and the dynamic obstacle avoidance process has been simplified to avoid static obstacles in space-time. This theory could be applied to real-time avoidance [Fujimura 1989, Erdmann 1986, Shin 1990]; or to Virtual Force Field (VFF) and Vector Field Histogram (VFH) theories have been developed to avoid real-time obstacles [Borenstein 1989, Borenstein 1991].

Yung and Ye introduced Collision Zones, which would never go through autonomous vehicles in transit, to ensure that there is no collision [Yung 1998]. Unlike the Collision Zones, the potential field theory introduces the force field theory, rather than defining the area where the robot should not go.

Potential Field

Among the various algorithms to avoid and respond to obstacles, is the potential field theory. The potential field fills the environment space of a robot with a virtual force field. In this virtual force field, the goal creates the force to pull the robot, and the obstacle creates the force to push the robot.

Because of its simplicity and mathematical beauty, the potential field theory is widely used in the path planning of mobile robots. However, Ge and Cui have raised the question that most studies related to potential fields focus only on motion planning in a static environment consisting of static goal and obstacles [Ge 2002]. They have found

that the problem of the traditional attractive potential field was only considering the relative distance of the robot to a fixed spatial goal.

They presented a new attractive potential field that is different from the traditional theory; this new theory is determined by the relative position and velocity between the target and the robot when determining the force acting on robot. The result of the virtual forces coming from this new potential field allows the robot to take appropriate action in response to the goal or obstacles. The advantage of this theory is that it does not require preemptive knowledge of the path of obstacles. Instead, the path is only planned by real-time measurement of the information of the obstacles.

Ge calculated the potential field according to the following equation:

$$U_{att}(p, v) = \alpha_p ||p_{tar}(t) - p(t)||^m + \alpha_v \left||v_{tar}(t) - v(t)|\right|^n \quad (2\text{-}2)$$

Where $p_{tar}(t)$ and $p(t)$ is the position of the robot and target at time t, and $v_{tar}(t)$, $v(t)$ is the velocity of the robot and target at time t. Therefore, $||p_{tar}(t) - p(t)||$ is a Euclidean distance between robot and target at time t, $||v_{tar}(t) - v(t)||$ is the relative velocity between robot and target at time t. And α_p, α_v are scalar positive parameters, and m, n are positive numbers.

Therefore, the force acting on the robot is the gradient value of the potential field, and this potential field can be altered in real time depending on the relative distance between the robot and the target, and the relative speed. Therefore, by calculating the virtual forces in the potential fields for all obstacles, the sum of these virtual forces influences the robot to judge and adjust its movements accordingly. This theory is worthy as it shows the proper relationship between the position and speed between the robot and the obstacle.

Avoidability Measure

Based on potential field, Ko and Lee proposed a technique for avoiding moving obstacles and applying them to a robot movement [Ko 1996]. They proposed a concept called Avoidablility *Measure* (AVM), which defines the conditions associated with the collision of a robot and an obstacle pair. Ko introduced the concept of AVM to avoid real-time obstacles. AVM is the inverse of the probability that a vehicle will strike an obstacle; it therefore represents how easily the robot can avoid obstacles.

In AVM, the distance between the robot and obstacles can be used to detect collisions between them; the probability of a collision can be determined also by the relative velocity between the robot and the obstacle that is also approaching (or moving away) from that distance. Therefore, they set the distance and the traveling speed to state variables to account for the possibility of avoiding collisions.

$$d_{o,r}(t) = \left| |P_o(t) - P_r(t)| \right| - r_o \tag{2-3}$$

$$v_{o,r}(t) = \dot{P}_o(t) \cdot \frac{P_o(t) - P_r(t)}{||P_o(t) - P_r(t)||} \tag{2-4}$$

$P_o(t)$, $P_r(t)$ represent the position of the robot and obstacle at time t. r_o represents the radius of an obstacle. So AVM increases as the distance $d_{o,r}(t)$ or $v_{o,r}(t)$ increases.

While there are many functions that can satisfy AVM, they propose a function called *Virtual Distance Function* (VDF) to express the pulling power of robots.

VDF, which is a function of the relative distance and velocity between the robot and the obstacle producing the virtual distance $vd_{o,r}$ that the robots needs to maintain as the threshold value, so that it can avoid obstacles.

$$vd_{o,r}\left(d_{o,r}(t), v_{o,r}(t)\right) = \frac{\alpha}{\beta - v_{o,r}(t)} \cdot d_{o,r}(t) \tag{2-5}$$

where

$$\alpha > 0, \quad \beta > \max\{v_{o,r}(t)\} > 0, \quad (\alpha, \beta \in \mathcal{R}) \tag{2-6}$$

VDF created a virtual potential field that emits a force that pulls the robot from the target. At each sampling time this potential field is updated and the force acting on the robot is determined by the gradient of this virtual potential field. This algorithm is suitable for avoiding moving obstacles because it considers its motion as well as distance from the obstacle.

CHAPTER 3
DYNAMIC LOCAL WINDOW PATH PLANNING ALGORITHM

Figure 3-1 shows the progress of the algorithm for searching the optimal path in a dynamic environment with moving obstacles. This algorithm is based on a graph structure. All coordinates of a map are defined as nodes with individual characteristics.

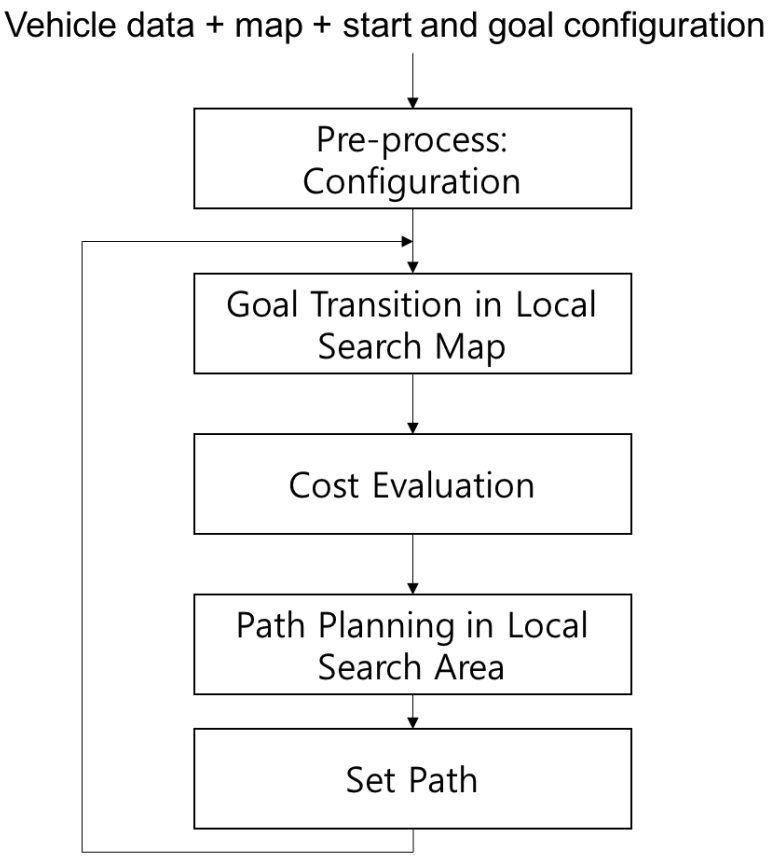

Figure 3-1. Flowchart of the dynamic local window path planner

When a goal point is assigned to the autonomous vehicle, it fuses terrain data with visual camera data, LIDAR, GPS and INS. In addition, the vehicle transmits the velocity and position information of the individual moving objects to an algorithm for searching the final path, together with the terrain data. This information is converted into

cells with individual costs, and the set of these cells is transformed into a single cost map and used to create the input values of the path tracking algorithm described below.

This path finding algorithm first creates a temporary goal point within a certain space, within the local coordinate system of the autonomous vehicle, to reach from the starting point to the goal point; then an optimal path is set up to the temporary goal point. While the vehicle is moving along that path, the algorithm recalculates the optimal path by setting a new temporary goal point. Through such a process, the car ultimately reaches the initial goal point. Before applying this algorithm, the process of modeling the world to apply it to the algorithm will be presented. Then each step of the algorithm will be discussed in turn.

World Modeling

World modeling defines the simplified real-time world, kinematics, and dynamics of autonomous vehicle in that world before applying path search algorithm. The following steps are performed before the algorithm executes.

Two-dimension Grid Modeling

This step simplifies the world to a set of grids in a two-dimensional coordinate system, and displays all the objects in the world, including cars and obstacles, as a grid with coordinates. Each object has a center of mass at its position, and it is assumed that all physical elements such as mass, force, and inertia are applied only to that point.

This grid has information such as the environment variable, obstacle recognition, autonomous vehicle recognition, etc., and has a cost value related to the information; this information can also be changed in real time.

Variables of Vehicle in Coordinates

There are many variables associated with modeling the vehicle. They have been simplified to a series of moving points in a two-dimensional coordinate system. Therefore, this work will look at the modeling of a mobile robot that exhibits the most similar forms and behaviors to a conventional car-like vehicle in the algorithm application.

The first is circular modeling. Circular modeling can be best explained by thinking of a robot as a circle; in other words, a single circle that draws a certain radius around a center of mass is composed of one robot. Since this circular modeling does not need to consider the rotation according to the heading, it has the advantage of simplifying a calculation. However, the simulation to be implemented is an automobile, and rotation is a key characteristic that cannot be excluded, so excluding it is not an effective way to produce a model.

Pasha mentioned kinematic limitations of car-like robots in his research [Pasha 2003]. He saw that a robot moving in a plane had a total of three degrees of freedom; an x-axis and y-axis movement, and an angle about an axis perpendicular to the plane. Therefore, the robot has three environmental variables (x, y, θ). On the other hand, the input to this robot is velocity; which is again divided into linear velocity and angular velocity, and the input variable is (v, ω). The number of this input variable is smaller than the number of environment variable, so the robot can be called as nonholonomic robot.

The characteristic equation model that can be applied to this nonholonomic robot is the automobile modeling defined in the coordinate system proposed by Ahmad Abu Hatab [Ahmad Abu Hatab 2013]. He declared two coordinate systems, the first being a

fixed global coordinate system in which the robot will travel. The other coordinate system is the local coordinate system that moves with the robot.

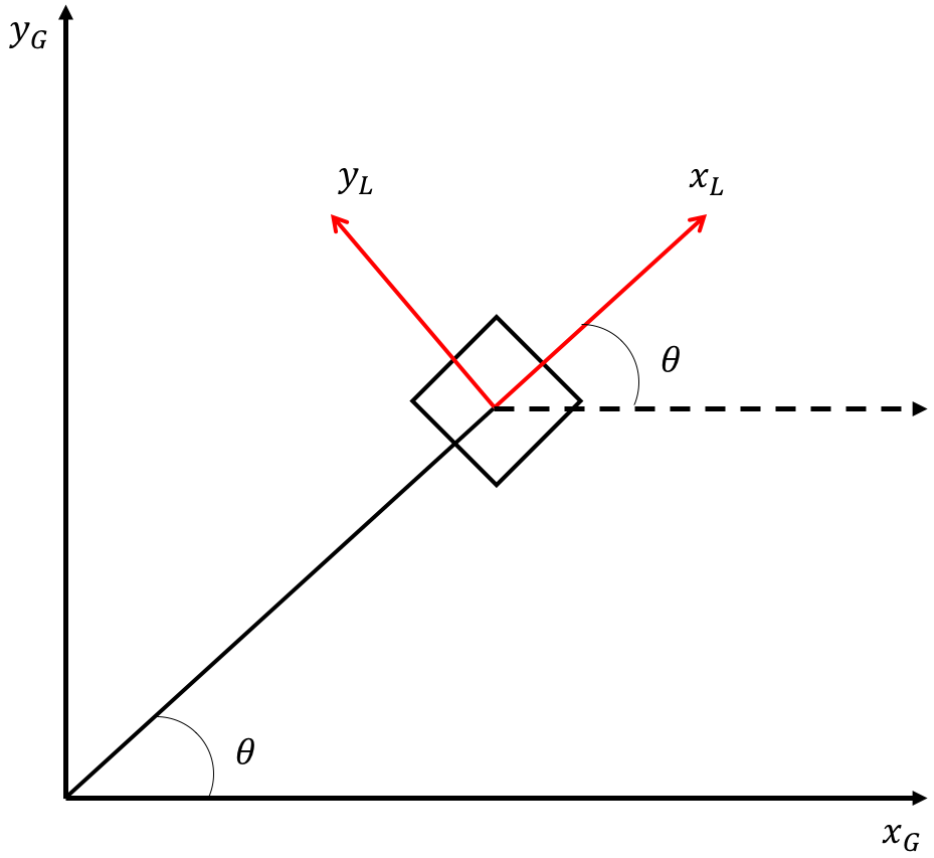

Figure 3-2. Relation between global and local coordinate system

The coordinate system defined above is shown in figure 3-2 above. This global coordinate system (X_G, Y_G) and the local coordinate system (X_L, Y_L) can be replaced by the following rotation matrix, which receives the input value as direction of the robot, θ.

$$(X_G, Y_G) = R(\theta) \cdot (X_L, Y_L) \tag{3-1}$$

$$R(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3-2}$$

Kinematic Equation and Types of Vehicles

Automobile behavior refers to the basic behavior that is achieved through the manipulation of a vehicle. The first thing, steering, is a circular input about the steering axis, which changes the heading of the vehicle. The second thing, throttle and brakes, are responsible for accelerating and decelerating the vehicle. At this time, the maximum value of the acceleration rate is fixed; and it is assumed that the deceleration rate is a bit larger than the maximum acceleration because of a safety concern.

The following is a matrix equation that shows how the three coordinate values that make up the vehicle change, according to the vehicle's linear velocity and angular velocity.

$$\dot{q} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos\theta & 0 \\ sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3-3}$$

The first and second elements of the vector $\dot{q}$ indicate the amount of change in the linear velocity of the vehicle, and are the derivative values of the position information respectively. The third element is the change in angular velocity of the vehicle.

The position and heading of the vehicle (x, y, θ) are changed according to the linear velocity and the angular velocity (v, ω), which are input values of the vehicle.

$$\theta_{t+\Delta t} = \theta_t + \omega \Delta t \tag{3-4}$$

$$X_{t+\Delta t} = X_t + vcos(\omega \Delta t) \tag{3-5}$$

$$Y_{t+\Delta t} = Y_t + vsin(\omega \Delta t) \tag{3-6}$$

where Δt is the time the input was applied.

An entity is an object that has default values for the above physical variables and behavior. This entity can be inherited and define two details shapes. The first is a player to apply this path search algorithm, and the other is a moving obstacle.

Player. The player is the model of the autonomous vehicle that is to be implemented. This player can receive prior environmental data. In addition, environmental data can be received through a local sensor, and information (position, speed) of moving obstacles around the object can be obtained with the same sensor. The player can also fuse the obtained information into a single large world map. The player can calculate the optimal route from the start point to the goal point with this fused world map and has a primitive driver and tracking algorithm that can follow the optimal route to reach the goal.

Moving Obstacle. A moving obstacle is any object that moves at a speed and direction in the world. This movement may follow a given goal with a scenario, or roam freely. However, in a typical intersection environment, most of the moving obstacles are assumed to be conventional vehicles, and the actions that these vehicles can take are limited; so, they cannot move around freely. Therefore -when applying this algorithm- moving obstacles are designed to have the following environment variables - position, velocity and dependent variable - start direction and arrival direction.

Pre-process

Once the model has been implemented to apply the algorithm, the algorithm can be started. The algorithm goes through a pre-process step before execution. In this step, it verifies whether the input value to be executed by the algorithm is valid, and then calculates various pre-computable variables necessary for executing the algorithm.

The input value of this algorithm is fused map data. This fused map is a single grid map in which two different maps are combined into one. One map is a cost map, consisting of the environmental cost values. There costs are considered as static values. Another map is a moving obstacle map, which is a dynamic map based on the predicted path generated by the observed moving obstacle. And the output value of this algorithm is a set of points that make up the optimized path.

Goal Transition in Local Search Area

Once the model has been implemented for application to the algorithm above, the algorithm can be started. This step includes setting a limited search range and temporary destination conversion in the limited search range of the vehicle before the route search for the optimal path planning.

Temporary Local Search Area Building

Arras proposed two modeling and planning steps for robot local path planning and obstacle avoidance [Arras 2002]. In the modelling stage, the shape and the dynamic of the robot are considered. In the planning stage, the path of the goal point branches from the point composed of (x, y) coordinates. Further, Arras performed a path search in the robot's local frame, rather than performing the calculation in the global frame. This is because, in applying the same search algorithm, doing it in a global frame makes the computation more complicated, as well as increases the difficulty of immediately responding to dynamically changing variables.

In this research, the path planning algorithm is influenced by the research of Arras, so that the robot's search radius is centered on its current position to perform the estimation, within a limited range. This search scope is typically set in a grid-like shape, which confines the range, so that it is detectable for the sensor.

Goal Transition

This path-planning algorithm is utilized to identify the optimal path within a limited search range, as presented above. Therefore, there is a need to shift the initial goal to a more limited, temporary, search range, which is further revised in each pass-planning phase.

The initial goal is transformed into a temporary changed goal for each pass planning phase. To set this temporary goal, we must check whether there is an initial goal point within the limited search range set above. If the initial goal point is outside the search range, the temporary goal point is set on the boundary of the search range, where Manhattan distance between the initial goal and boundary of the search area is the smallest.

The Manhattan distance is a concept developed by Hermann Minkowski in the 19th century, a method for estimating the distance between two points in Cartesian coordinates [Krause 1987]. The Manhattan distance d is the sum of the lengths of the vectors p and q projected onto points on a coordinate axis in the Cartesian coordinate system.

Let (p, q) are vectors, $p = (p_1, p_2, \cdots p_n)$, $q = (q_1, q_2, \cdots q_n)$

$$d = \left||p - q|\right| = \sum_{i=1}^{n} |p_i - q_i| \tag{3-7}$$

For example, if the position of two points on the 2D plane are (p_1, p_2) and (q_1, q_2), the Manhattan distance is $|p_1 - q_1| + |p_2 - q_2|$.

Conversely, if the initial goal is above the search range boundary or within the search range, the temporary goal point is set as the initial goal point and no more goal transitions are made.

The following figures depict how a goal transition occurs.

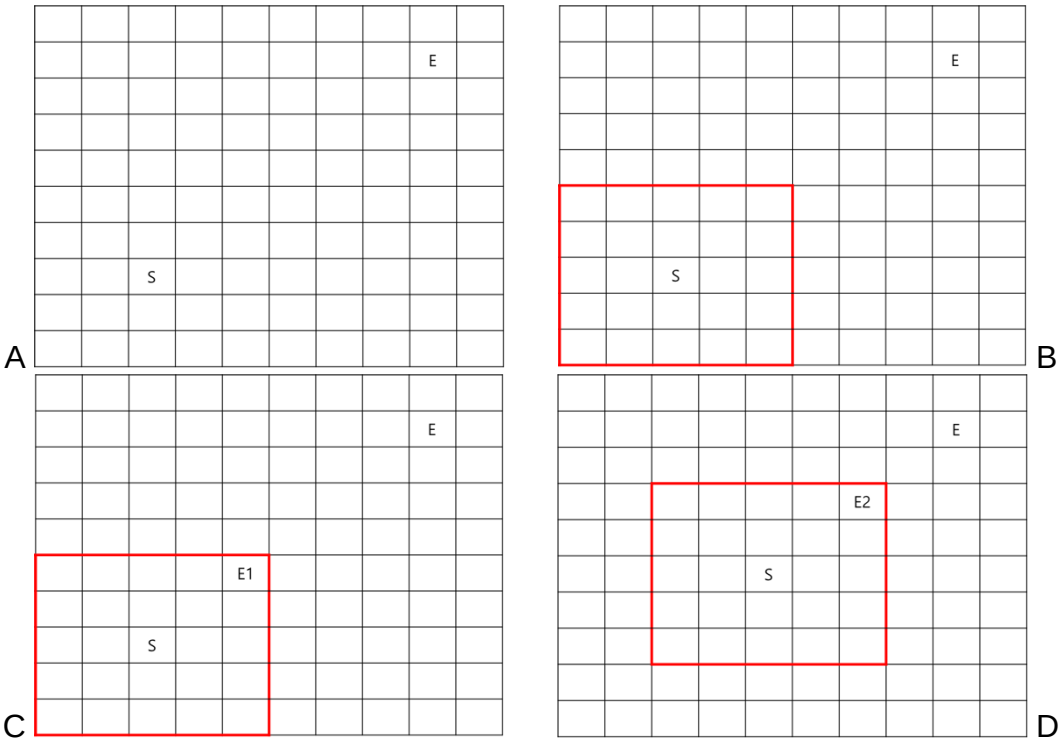

Figure 3-3. The process of goal transition. A) Initial State, B) Define local search area, C) Transition from initial goal to temporary goal D) Next step.

Figure 3-3-A is the first step of the algorithm. S is the starting point, the current location of the player, and E is the given initial goal. In the case of an offline global path algorithm, there will be increased time and space required to find the optimal path from S to E.

Figure 3-3-B shows how to limit the search space. This limited search space is called a local search area, and it is extended to a certain area around the current position of the player. In general, this search area can be set at most detectable sensor area of the player.

Figure 3-3-C is the transition of the given initial goal E, into the local search area defined above. Given the confinement to 2D space, a point on the boundary of the local area, which has nearest distance to the given initial goal, was chosen.

Figure 3-3-D depicts the process of the search algorithm after the player moves; the algorithm is re-called again, by the iterative process to be described later. Thus, an updated search area will be set around the new location of the player.

The local search area of a player may be a polygon, or a curvature shape including a call or a circle, as the local sensor (such as a LIDAR) has a radially extending sense area. However, the algorithm depicted limited the local position to a square of certain size, for simplification of operation. The goal transition is a step that is performed every time an algorithm is executed, until an initial goal exists outside the local search area. If the initial goal is reached within this local search area, there is no need to perform a goal transition again and the step is skipped.

Cost Evaluation

If the initial goal has transitioned to the local search area, the cost for all environmental variables within the local search range, will be calculated and fused a map. This map depicts a set of two-dimensional, individual grid cells, declared in the previous world modeling. The grid cell has the following characteristics:

Table 3-1. Characteristic of grid cells.

Type	Description
Int[][] Position	The position of this grid in global frame
Int cost	The cost of this grid
Boolean isStartCell	Whether this grid is the starting point
Boolean isFinishCell	Whether this grid is the goal point

Cost will be used as the input value of the heuristic function, when determining the route to the player using the A* algorithm. This algorithm designs the path toward the lowest cost, through computations involving heuristic function values.

Environmental Cost

The environmental cost is the value for the environmental variable used in the existing offline path-planning. In other words, this value is a statistically considered variable in the path search, i.e. the cost associated with the terrain elements. This cost may be pre-entered the vehicle in advance, or it may be computed through visually and spatially analyzed information, via a terrain detection sensor mounted on the vehicle. Importantly, this value is a static value, meaning it does not change between path searches. The following is an example of this kind of environment variable in intersection:

Travel lane: The area in which the vehicle can move. Generally, it refers to well-cleaned asphalt or concrete roads and is the preferred route for vehicles to move. A grid with this property has a low cost, which causes the vehicle to move toward the grid.

Verge: This area occupies most of the simulated intersection model and is restricted for entry, due to high cost values. In the real world, this area would be full of static objects, such as buildings. The reason for giving a high cost to this area, is that the collision between the structures and vehicles may be catastrophic to progress.

Shoulder: This area is an edge, or boundary of a travel lane, which has higher cost than the travel lane, but less cost than the verge. Therefore, movement to a shoulder is not impossible.

Center line: This area controls the direction of movement of the vehicle at the road and has a role of dividing the lane. A vehicle moving in a lane cannot cross the opposite lane beyond this center line.

The following table shows static-cost values given for the above-mentioned environmental variables, as designed to apply the discussed algorithm to this simulation.

Table 3-2. Cost value to environmental type.

Type	Cost	Description
Travel lane	1	Movable area
Verge	99	Unmovable area
Shoulder	4	Not preferred to travel, but can
Center line	99	Cannot across

* Not real data.

Path Planning in Local Search Area

This step is the process of deriving the optimal path from the above-defined local search range, to the replaced temporary goal. Here, the fused map is a cost grid map, including: the fusing grid cell, the starting point, and the temporary goal. These converses the terrain information collected from the sensor.

Initialize Planning

This step receives the fusing map and initializes variables and is the first function declared when the path-planning algorithm is executed. Through checking the start point, converted goal, local search area and fused cost map, values are confirmed as correctly declared and maximum step is specified. This limitation is necessary to stop the route search algorithm immediately after determining that the number exceeds the allowable number of search units. It then declares an array that can temporarily store the path, found through path search. If the validity of the above input values is

confirmed, and the output variables are created, the following step-by-step search is performed.

Step Finding

It is an essential function of path search through A* algorithm; most of the various A* algorithms are this part. For each step, a neighbor node, accessible from the current node, is put into the open list, based on the A* algorithm. Then, the heuristic function selects a grid having the lowest cost value, as a target node to be next, returning the type of the target node. The current node is then included in the closed list and excluded from the search. The return value of this step is the grid type, which will go to the check arrival state step, to determine whether to continue or abort the algorithm.

Check Arrival State

This step determines whether to continue the route search, according to the above step-by-step search results. The result of iterative execution is shown as the type of the corresponding grid. If the result is a temporary goal destination, it is judged as arrived, and the search function is terminated. If not, this trigger is increased by one and the neighbor grid is searched again. If this trigger exceeds the maximum search range, the algorithm determines that there is no way to reach that temporary goal, and the algorithm terminates (returns no route).

Set Path

In this step, the optimal path from the start point to the temporary goal point is found through the above algorithm and placed in an array, so that the autonomous vehicle can follow the path. Having received the grid at the current location, and the grid at the goal point, confirming that the path search is complete, the result is stored in an array. This result then goes backwards from the temporary goal to the start point

because the path is based on the A* algorithm result. So, when put in an array, the path reverses to go into the time of the array. The final data then becomes a set of waypoints of the optimal path from the start point to the temporal goal point.

The key point of this step is to extract only the required waypoints out of the optimal path. When the path is searched through the first A* algorithm, the result is generated by the step of each visited cell. Therefore, there is a potential for several problems, such as: programming efficiency, storage space and the behavior which must be followed whilst following a path, to create a grid passes through all points. In addition, unexpected problems can occur, due to sudden speed increase/decrease, frequent route changes and rotation switching. Therefore, it is efficient to give only the waypoint that the vehicle must pass.

The process of specifying key waypoints is as follows:

1. Put the first $t = 1$ data at the beginning of the array.
2. Put the data of the next step $t = 2$ into the next array. Comparing this with $t = 1$, we calculate whether the position at $t = 2$ moved horizontally or vertically.
3. Now, from $t = 3$, if the movement of the corresponding step and the movement of the previous step differs from the movement of all the steps and the movement of the previous step, the coordinates of the current step are stored. Every time a direction change (vertical-horizontal, horizontal-vertical) occurs, the coordinates are stored.
4. Since the last coordinate is the result value, it must be inserted at the end of the array.
5. Consequently, the array is the set of the coordinates of the start point + the coordinates of the next step point + the coordinates of breaking point + goal point.

Examples: If the number of grids in optimal path between the start point and the goal point is 120, then the traditional A* algorithm has a total of 120 path points, because each cell is 1 in size. To check all 120 points, the speed of the car between

point checks exceeded the check time, so it may not be possible to check properly, as there may be trouble circling previous points. Therefore, extracting only a fraction of the waypoints that have a decisive influence on the optimal path, and giving it to the vehicle, helps to use more efficient path space.

Iterative Running

After each successful search of the best path, the temporary destination is compared with the initial destination, whenever the algorithm is re-executed. At this point, iterative execution continues, if the temporary goal is different from the initial destination. This initial iteration is done after the first delay $d_0(ms)$ has elapsed, since the original algorithm was fully executed and the local search area was created.

Now, depending on the current speed of the car, the interval at which the next iteration is executed differs. At slow speeds, it takes a long time to move the path, so update time may be slow. At faster speeds, the update time should be shorter because it takes a shorter time to move the path. The calling cycle of the repeated function is given by:

$$rate = \frac{||Velocity_{Current}||}{||Velocity_{Max}||} \tag{3-8}$$

$$delay = d_0 - (rate * d_0) \tag{3-9}$$

The result of this iterative execution, optimal path and $delay$ are given to the player. The player follows the path using the path tracking algorithm. The player then readjusts local search area and temporarily goal after $delay$(ms). With these information, the path finding algorithm is repeated, until the player reaches the initial goal.

CHAPTER 4
PATH TRACKING ALGORITHM

This algorithm defines how the autonomous vehicle executes the input value, when a certain optimal path is created for the above pathfinding algorithm.

The optimal path contains information of the route given to the autonomous vehicle with a set of the coordinates: the coordinates of waypoints that must be passed, the coordinates of the start point, and the coordinates of the goal point. The following steps are taken by the robot, to the goal along the route obtained through the path finding algorithm.

Target Waypoint Selection

The autonomous vehicle compares its position with a set of waypoints that they have entered and determine its target point. First, the vehicle sets $count$ to 1 to obtain the first one of the waypoints of the input route. After the vehicle calculates the distance from its current position to the goal destination, it also calculates the distance between the $count$ point of the waypoint and the goal destination.

$$distToGW_{count} = \sqrt{(Goal.x - Waypoint_{count}.x)^2 + (Goal.y - Waypoint_{count}.y)^2} \quad (4\text{-}1)$$

$$distToGP = \sqrt{(Goal.x - Point.x)^2 + (Goal.y - Point.y)^2} \quad (4\text{-}2)$$

$distToGW_{count}$ is the Euclidean distance from the $count$-th waypoint to goal destination, and $distToGP$ is the Euclidean distance from the current vehicle position to the goal destination.

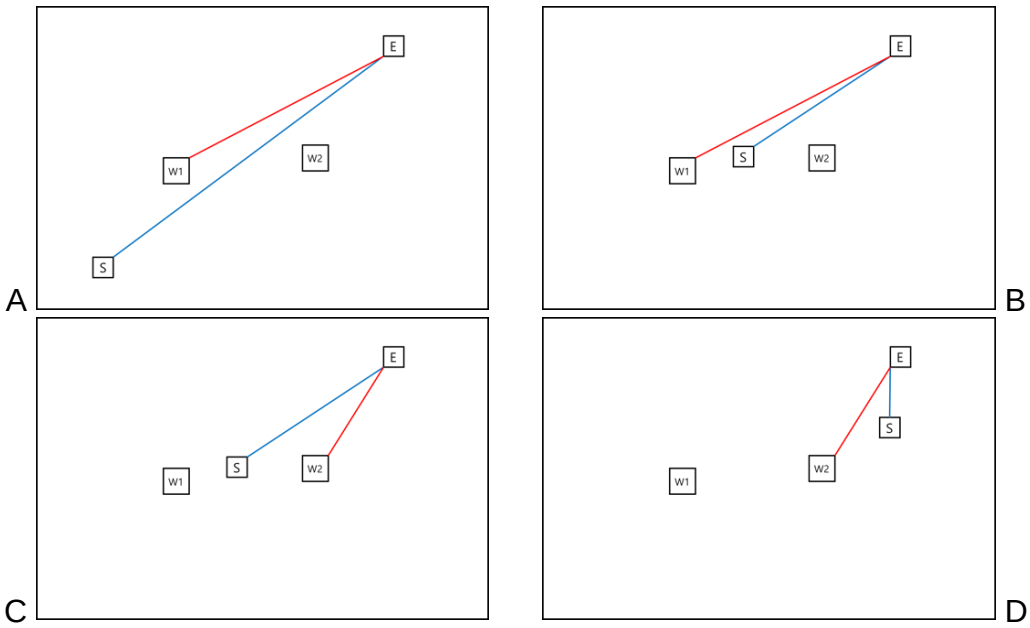

Figure 4-1. The process of target waypoint selection. A) Initial State (calculating distances), B) Passed first waypoint, C) Recalculating distances D) Passed second waypoint

Figure 4-1-A is the state when the autonomous vehicle received the first mission waypoint. Here, S means the autonomous vehicle and E means the (temporal) goal point. W1 means first mission waypoint, and W2 means second mission point. The blue solid line represents the distance from the current vehicle position to the goal, $distToGP$, and the solid red line represents the distance from the current mission waypoint to the goal, $distToGW_{count=1}$. The mission waypoint that the vehicle must follow is the first waypoint, because the goal point distance from the mission point, $distToGW_1$ is shorter than the distance from the current point to the goal point, $distToGP$. Figure 4-1-B is the state after passing the first mission waypoint. From here, the distance from the current vehicle to the goal, $distToGP$ is shorter than the distance from the first waypoint to the goal, $distToGW_1$. Therefore, $count$ is incremented by 1, then the car selects the next level of mission point. Figure 4-1-C likewise checks the next mission waypoint. It

recalculates again $distToGW_2$, $distToGP$ and compare two of them. Because $distToGP$ is longer than $distToGW_2$, the vehicle goes towards the second waypoint. Figure 4-1-D is the final appearance. Figure 4-2 shows whole process of target waypoint selection.

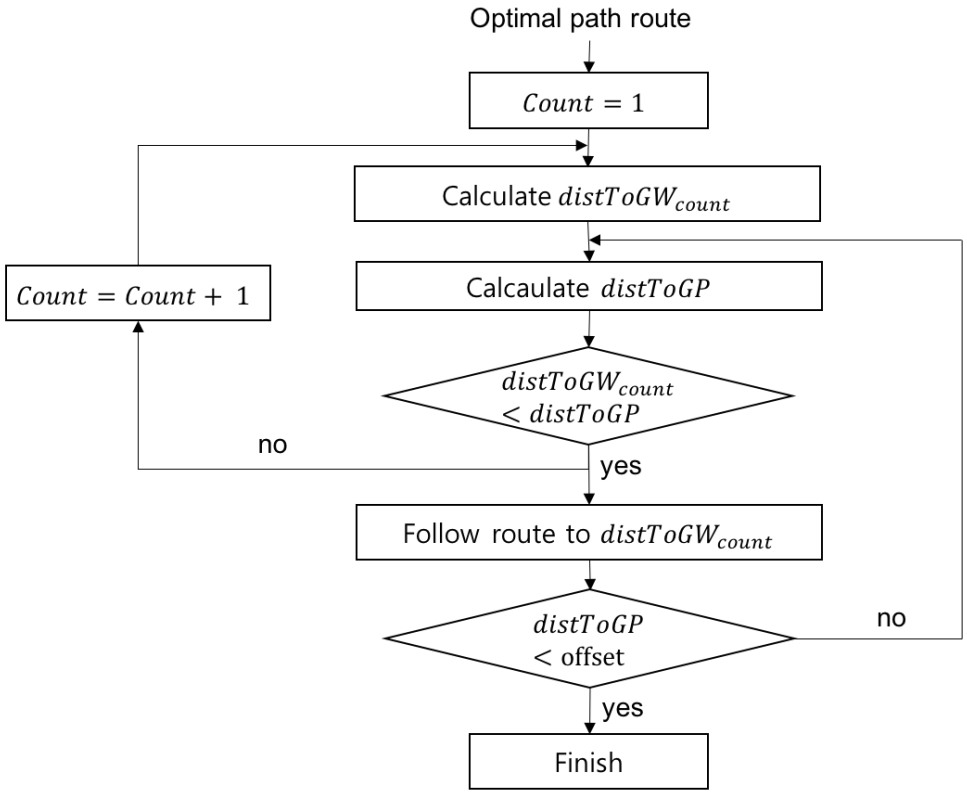

Figure 4-2. Flowchart of target waypoint selection.

Following Target Waypoint

Once a target waypoint is selected, a kinematic method is needed that allows the robot to move to the waypoint. Coulter at Carnegie Mellon university devised a kinematic model algorithm for robot to destination [Coulter 1992]. The robot computes the arc from the present position to the destination position as much as it should go, and calculate the look-ahead distance to pursue the destination.

Based on the method, the tracking algorithm for this autonomous vehicle is as follows:

1. Determine current position and heading of the vehicle
2. Determine the position of the waypoint
3. Covert the position of the waypoint into the coordinate system of the vehicle
4. Calculate the angle and look-ahead distance between the vehicle and waypoint.
5. Compare the heading of the vehicle and angle
6. Calculate steering and throttle input

The vehicle calculates the angle, $angle_{to\_Target}$ between its position coordinates, $(x_{vehicle}, y_{vehicle})$ and the waypoint, $(x_{waypoint}, y_{waypoint})$. This angle is obtained on a global coordinate basis as

$$angle_{to\_Target} = tan^{-1}(\frac{y_{waypoint} - y_{vehicle}}{x_{waypoint} - x_{vehicle}}) \quad (4\text{-}3)$$

$angle_{to\_Target}$ is the heading that the vehicle should aim for to reach the waypoint. $angle_{to\_Target}$ is compared to the heading which the vehicle is currently facing and if there is a difference, the vehicle will begin to rotate within the maximum rotation angle to the target waypoint. This rotation is satisfied if it falls within the allowable offset and the alignment of the heading and the acceleration of the vehicle are made simultaneously.

The look-ahead distance L between the position of the current vehicle and the position of the waypoint is then calculated. If the waypoint is not the goal point, the robot accelerates to the maximum speed.

$$L = \sqrt{(x_{waypoint} - x_{vehicle})^2 + (y_{waypoint} - y_{vehicle})^2} \quad (4\text{-}4)$$

After calculating the angle to rotate and the look-ahead distance to go, the vehicle then generates the steering and throttle inputs based on these values and moves to the waypoint.

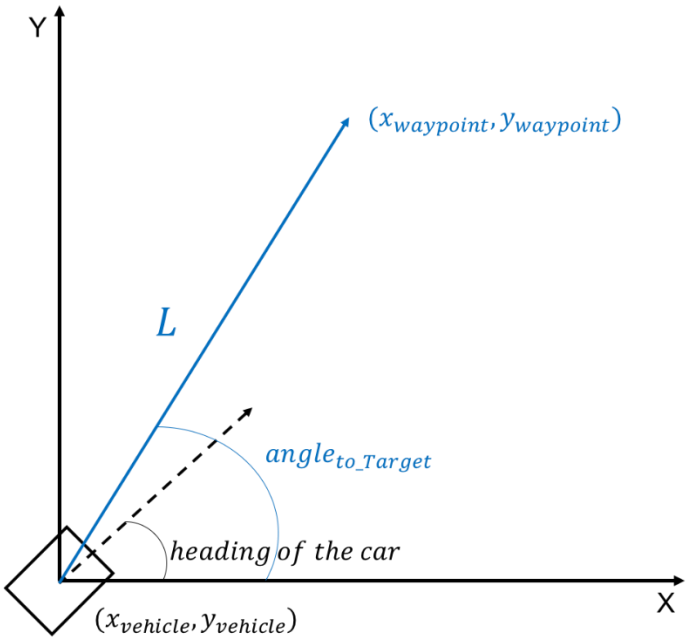

Figure 4-3. Geometry of the following target waypoint.

Since the robot is moving in real time, it is possible that it has already passed its mission waypoint. Therefore, there is a circular offset around each waypoint, wherein if the robot enters the range, it is considered to have arrived at the waypoint, prompting the robot to go towards the following waypoint.

Finally, if the waypoint to which the robot should go is the goal point, the robot approaches a relatively smaller range than the offset above and stops when it arrives within the offset. All execution sequences are shown in Figure 4-2. As the robot follows the path of environmental variables, dynamic obstacle detection and avoidance movements should be performed. The following is an algorithm that avoids collision and detection while the robot follows the path.

Collision Detection and Response

Collision detection and avoidance are essential to ensure safe car behavior at an intersection. Unrecognized static obstacles, or moving obstacles, can pose a major threat for catastrophic disaster to vehicle operation in real-time situations.

The detection of the autonomous vehicle depends on the moving obstacle expected-cost value of the local search range, computed in real time, with the above algorithm. The Probabilistic Moving Obstacle Grid (PGMO) method is used to determine the range of risk generated by moving obstacles.

Probabilistic Moving Obstacle Grid

There is a need to assign a cost to the obstacles, as if there are only stationary obstacles (such as the environmental variables above), one can simply give them a high cost, to defer entry to that area. However, in the case of moving obstacles, it is impossible to assign a cost simply because they change their positions in real time. Therefore, the following cost allocation method was devised.

Probability Grid of Moving Object (PGMO) is a method to calculate the cost for moving obstacles, as measured on the local sensor of the vehicle. The PGMO places a high cost on the grid corresponding to the location of the measured obstacle. Further, the magnitude and direction of the obstacle’s velocity also affects the cost. The faster the speed and the longer the distance traveled by the obstacle, the larger the area occupied by the obstacle's future path becomes. Therefore, the cost range is broadened by the speed of the obstacle (the magnitude of velocity), and widened toward the heading of the obstacle (direction of the velocity).

The following is a method to calculate the risk area of dynamic obstacles where the position (x, y), the magnitude of velocity $CURRENT_{VELOCITY}$ and direction of the velocity θ are measured

First, there are three risk areas for the obstacle. The risk area is a set of cells with the values of the lowest risk cost, the medium risk cost, and the highest risk cost, respectively. The risk area of the moving obstacles are as follows:

$$Risk\_area = \{Lowest\ risk, Medium\ risk, Highest\ risk\}$$

The next step sets the size of the individual risk areas that this moving obstacles is expected to occupy.

$$width = safe\_dist + \left(\frac{|Velocity_{Current}|}{|Velocity_{Max}|} \times safe\_dist \times i_{risk}\right) \quad (4\text{-}5)$$

$width$ is the breadth of the risk area and

$$height = safe\_dist + \left(\frac{|Velocity_{Current}|}{|Velocity_{Max}|} \times safe\_dist \times j_{risk}\right) \quad (4\text{-}6)$$

$height$ is the vertical part of the risk area. $safe\_dist$ is the minimum safe distance to avoid collision and has the same value $safe_{dist} = 20$ for all obstacles in this simulation. $Velocity_{Current}$ is the current velocity of the observed obstacle, and $Velocity_{Max}$ is the maximum velocity that this moving obstacle can have. Finally, (i_{risk}, j_{risk}) is a constant according to the risk level, ranging from the lowest risk to the highest risk:

$$i_{risk} = \{0, 0.5, 1\},\ j_{risk} = \{0, 1, 2\} \quad (4\text{-}7)$$

In other words, at the lowest risk, the risk area is equal to the $safe\_dist$ because $i_{risk}, i_{risk} = 0$. At the highest risk, the vertical length of risk area is much longer than the horizontal length of it because we will pace the heading axis of the moving obstacle equally on the $height$ axis.

Through the obtained values $width$ and $height$, the risk area around the position of the moving obstacle is represented as a set of cells having the following coordinates.

$$risk_x = \{risk_{x1}, risk_{x2}, \cdots, risk_{width}\} = \sum_{i=1}^{width} \left(x - \frac{width}{2} + i\right) \quad (4\text{-}8)$$

$$risk_y = \{risk_{y1}, risk_{y2}, \cdots, risk_{height}\} = \sum_{j=1}^{height} \left(y - \frac{safe_{dist}}{2} + j\right) \quad (4\text{-}9)$$

$risk_x$ is a set of x points that are uniformly extended on both sides around the position of the obstacles, and $risk_y$ is a set of y points whose forward coordinate are more deflected than the backward coordinates around the position of the obstacle. Because the obstacle is expected to move in the direction of velocity, sudden back warding is impossible after the time has passed.

Now based on the heading of the observed moving obstacles, we rotate the upper risk area. $Trans_{risk_X}$ is a set of x coordinates of the risk after the rotation.

$$Trans_{risk_X} = \sum_{i=1}^{width} (risk_{xi} - x) \times \cos(\theta) - \left(risk_{yi} - y\right) \times \sin(\theta) + x \quad (4\text{-}10)$$

And $Trans_{risk_y}$ is a set of y coordinates of the risk after the rotation.

$$Trans_{risk_y} = \sum_{j=1}^{height} (risk_{xi} - x) \times \sin(\theta) + (risk_{yi} - y) \times \cos(\theta) + y \quad (4\text{-}11)$$

The area of risk thus rotated is cost-adjusted for each risk level.

Table 4-1. Risk cost to moving obstacle.

Risk level	Risk cost	i_{risk}	j_{risk}
Lowest risk	16	0	0
Medium risk	13	0.5	1
Highest risk	10	1	2

* Not real data.

The following is a series of steps on how a cost is awarded by the PGMO method.

1. The onboard sensors of the vehicle determine the position and velocity at the time of detection of moving obstacles in the sensor range.

2. The current position of the obstacle and the minimum safety distance are given highest risk-cost to the adjacent cell. The ratio of the current speed to the maximum speed of the obstacle is calculated according to the direction of the obstacle, to obtain the danger area $Area_{risk}$ corresponding to each risk.

3. Considering the heading of the obstacle, adjust the risk-area for each of the above risks, to more heading. The risk level is consisted of following: Lowest risk cost, Medium risk cost, and Highest risk cost.

4. From the lowest risk to highest risk, the cost is given to the risk-area that has been modified to suit the heading above.

Figure 4-4 displays the above procedure.

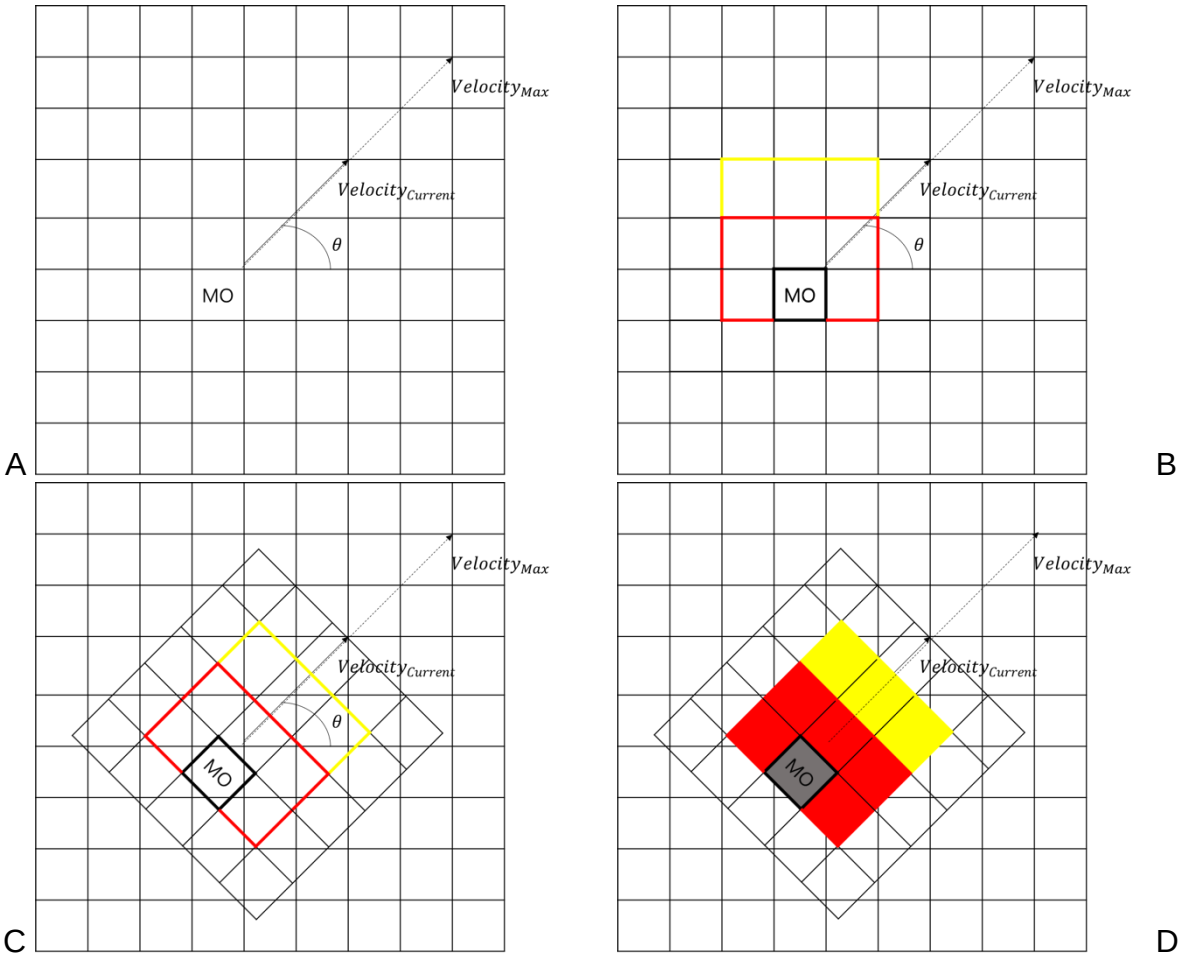

Figure 4-4. The process of cost evaluation. A) Initial state (measuring moving obstacle), B) Set risk areas to risk levels, C) Rotate risk areas D) Set cost to each risk areas

The path of each moving obstacle is predicted according to the position and velocity observation which defines the risk area of the moving obstacle. The information about this risk area will be included in the moving obstacle map used to account for the behavior of the autonomous vehicle between path tracking.

There are many theories for detecting the potential collision of objects: Bounded Box theory, Axis Across, Triangle Centroid Segments, and so on. The method used herein, is designed as follows. The vehicle leaves its current position and the acceleration goes towards the heading; a vehicle moving in real time will soon be in a certain range at the end of the heading direction. A straight line is drawn from the center of mass of the vehicle, to the direction of the heading. This straight line can be assumed to be the future positions of the vehicle. Then, a set of coordinates is specified from the start point of this straight line (car center of gravity), to the end. In this research, twenty discretized detection points, with initial position of the vehicle $[x, y]$ and direction of the vehicle θ, were specified:

$$senseLine = \{[x_1, y_1], [x_2, y_2], \cdots, [x_i, y_i]\} \tag{4-12}$$

$$x_i = x + i \cdot \cos(\theta) \tag{4-13}$$

$$y_i = y + i \cdot \sin(\theta) \tag{4-14}$$

where $i = 20$.

A discretized search point was selected, because the search was within the cell of the grid. Figure 4-5 shows details.

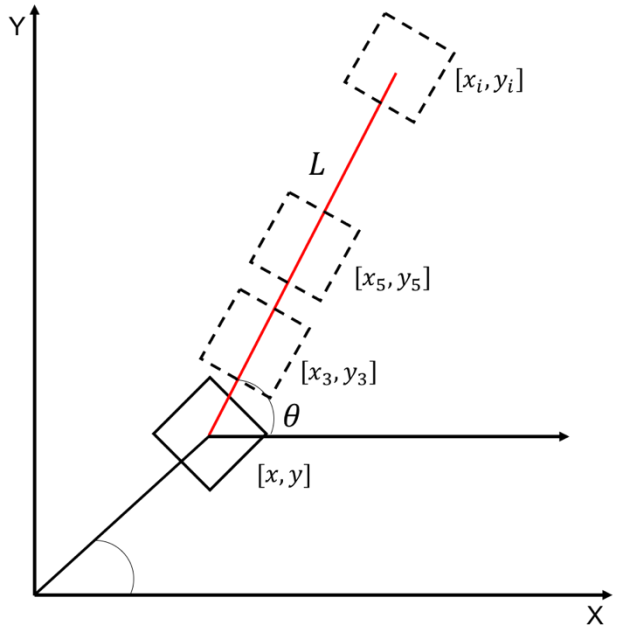

Figure 4-5. The search range line and its discretized search points.

At every reference time, the car initializes a series of maximum risk values and compares 20 detection points to the maximum value declared above, therefore redefining the largest value, as the maximum risk value. When the vehicle moves along the path it takes from above, it compares the maximum value.

$$\text{max\_risk} = Max\left\{\max_{init},\ senseLine_i\right\} \tag{4-15}$$

The behavior of the vehicle based on max_risk level follows:

- If the max_risk value does not change, it means that there is no dynamic obstacle at least to a certain distance in the heading direction. The car runs normally.
- If the max_risk value indicates lowest risk level, there is no dynamic obstacle up to a certain distance in the car heading direction, but the dynamic obstacle is less likely in the future. The car does not accelerate and remain current velocity.
- If the max_risk value indicates medium risk level, there is a higher probability that there will be more moving obstacles in this search line. The car decelerates.
- If the max_risk value indicates highest risk level, there will be dynamic obstacles in this search line. The car stops immediately.

CHAPTER 5
SIMULATION RESULTS

Software

In this research, the simulation was made to verify the pathfinding algorithm. It would be better if it was applied to the real environment, that is, to the autonomous vehicle, but first it is tested in simulation to examine the suitability, feasibility, and constraints of these algorithms for the simulator.

This simulation implements a simple crossing of a cross-shaped two-lane crossing and allows for the implementation of adjustable moving obstacles. An autonomous vehicle is also implemented to apply the corresponding path planning algorithm. Additional functions, algorithms, and GUIs were also written using Java's awt and Swing native packages.

This simulation is written in Java under the OSX environment. There are many programming languages such as C / C++, FORTRAN, which guarantees high speed, and MATLAB, which is easy to express mathematical process. However, Java is used here because Java is an object-oriented programming language so it has many strengths. For example, autonomous vehicles and conventional cars in simulation have common attributes of the same vehicle. In other words, it is advantageous to declare a certain entity and to easily implement various transformations, cars or moving objects , with the framework. This leads to improved performance through reusability of program sources and optimization of memory space.

Configuration

The simulation environment needs to be set before testing the algorithm.

First, the environment variables are set up to which the simulation will be applied. The world model in which the simulation will take place is an intersection where two lanes intersect each other.

The top of the map points northward. The absolute north of all simulators is -90 degrees absolute. This value decreases in the counterclockwise direction, and increases in the clockwise direction. Thus, the north is -90 degrees, the west is -180 degrees, the south is -270 degrees, and the east is -360 degrees or -0 degrees. To maintain a sequence of angles, the angle beyond 360 degrees automatically adds -360. Its size is scaled to 600px by 600px.

The discretized unit time is declared to measure the execution time of the simulation. This Unit Time(UT) is added by 1 every time the simulation is updated in the main program, and follows the following equation:

$$1UT = The\ time\ it\ took\ for\ the\ frame\ to\ be\ updated\ once$$

Results are shown below. Figure 5-1 is a visualized map of the intersection where this experiment is performed. Figure 5-1 consists of several environment variables. First, the gray space is the travel lane, which means that the car can travel. And the space marked in green is the verge, which means that the car cannot go. At the bottom of the map, the current player object is shown as a white object.

Figure 5-2 shows the map to be applied to the path search algorithm by converting each element of the above visualized map into cost. The travel lane has the lowest cost and is marked in green and the verge part has the highest cost and is marked in black. And the part that was not shown in the visualized map is the shoulder part, it has medium cost, and the cost map is yellow.

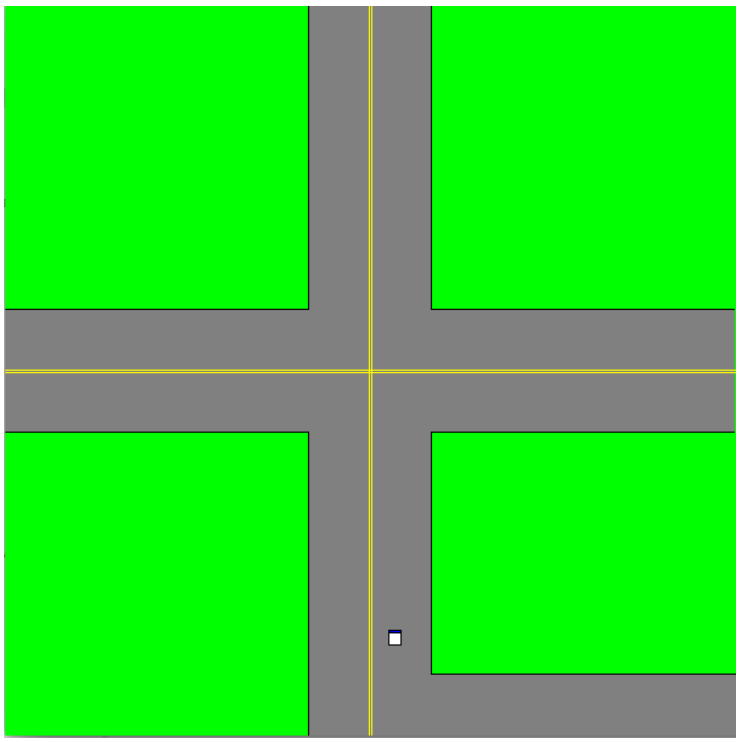

Figure 5-1. Visualized map.

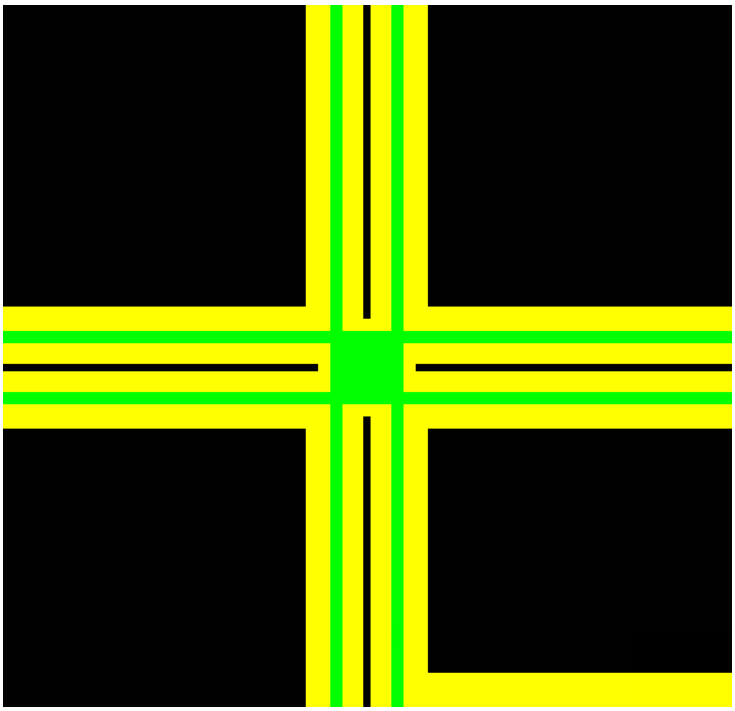

Figure 5-2. Cost map.

Path Planning without Moving Obstacle

In the first simulation, the algorithm is tested to find the optimal path from the obstacle-free state to the destination. Performance improvements through comparison with traditional off-line path planning results will also be discussed.

Scenario I: Path Planning without Moving Obstacle with Initial Search Area

The following values are declared for simulation.

- Location of the car in the map: $(320, 520)$
- Maximum speed of car: $0.4px/UT$
- Maximum acceleration of the car: $0.01925px/UT^2$
- Maximum angular velocity of the car: $0.52deg/UT$
- Position of destination in the map: $(331, 75)$
- Area of local search area: $(60, 60)$

Figure 5-3-A shows that the path planning algorithm to the destination is executed in real time. Figure 5-3-B shows the result of the position shift change to destination. From the start of the first search algorithm to the generation of the first route, the speed of 301 UT was taken, and the arrival time to the last destination through the new route search through the moving and iterative algorithm call was 1696 UT. A total of 94 search algorithms were called, and the search time of the average search algorithm was 4.1170 UT. Figure 5-3-C and Figure 5-3-D are graphs showing changes in velocity and heading per UT.

Table 5-1. Results of Scenario I.

Type	Description
Final arrival time	1696 UT
First path find completion time	301 UT
Number of search algorithm calls	94
Average path finding time	4.1170 UT

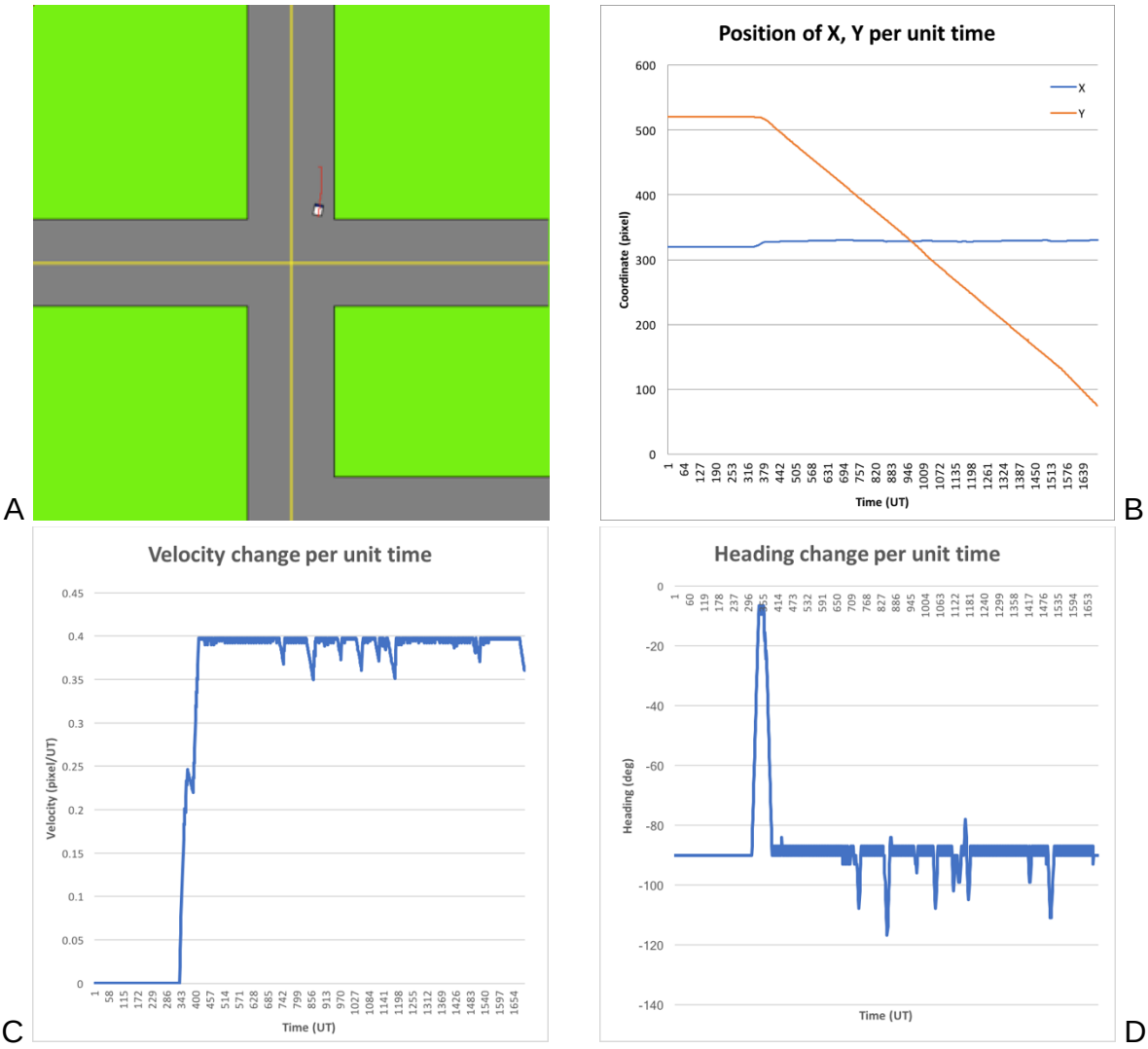

Figure 5-3. Results for scenario I. A) Execution scene, B) Position changes C) Velocity changes D) Heading changes

Scenario II: Path Planning without Moving Obstacle with Off-line Path Planning

The following values are declared for the simulation.

- Location of the car in the map: $(320, 520)$
- Maximum speed of car: $0.4px/UT$
- Maximum acceleration of the car: $0.01925px/UT^2$
- Maximum angular velocity of the car: $0.52deg/UT$
- Position of destination in the map: $(331, 75)$

Figure 5-4-A shows the result of a series of processes that the route searches for the car to the destination using the conventional off-line path planning algorithm and then the car arrives at the destination along the route when the route search is completed. Figure 5-4-B shows the result of the position shift change to destination. It took 2776 UT from the start of the off-line search algorithm to the path generation.

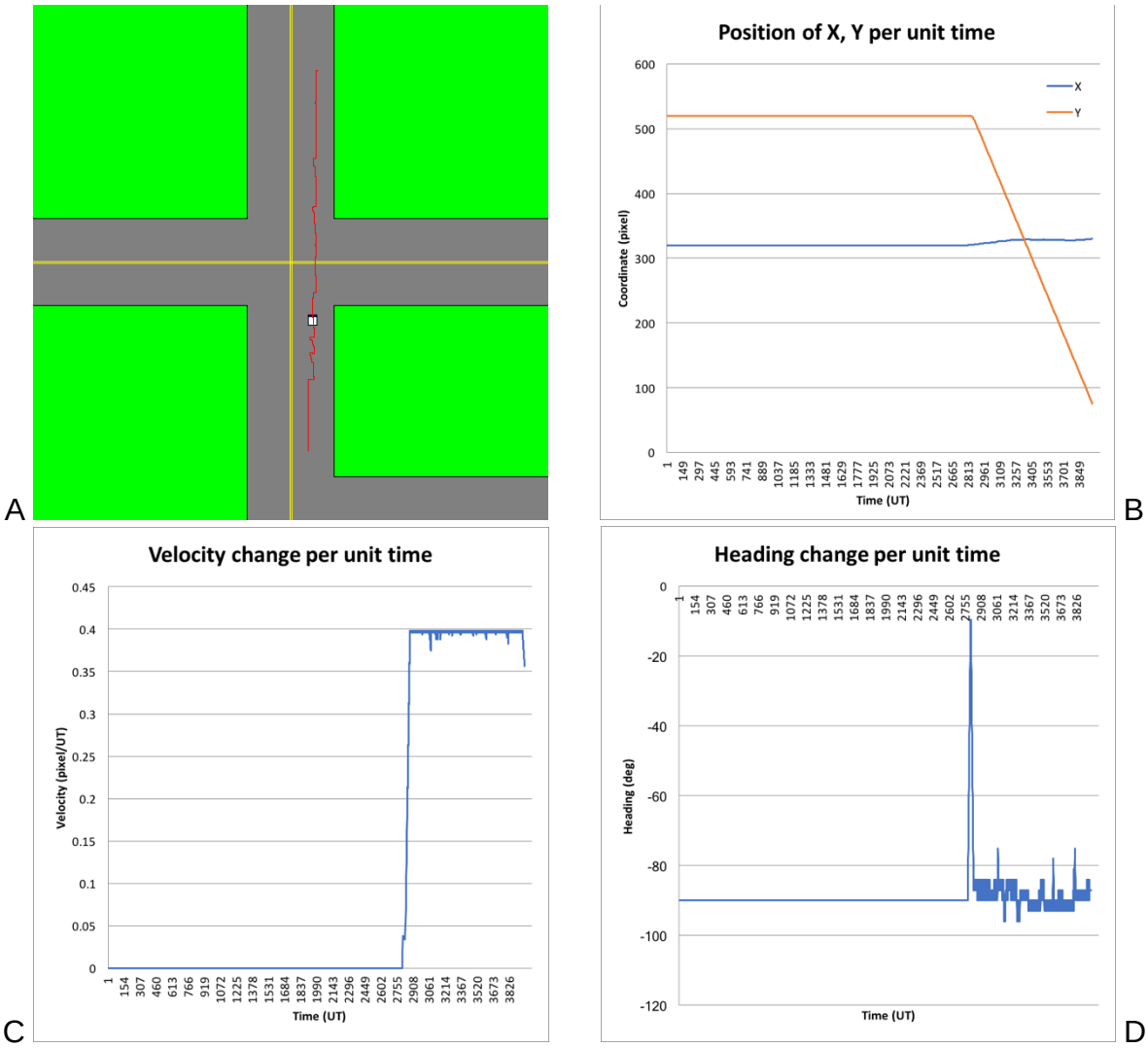

Figure 5-4. Results for scenario II. A) Execution scene, B) Position change C) Velocity change D) Heading change

After receiving this optimal route, the car moved to its destination and its arrival time was 5750 UT. Figure 5-4-C and Figure 5-4-D are graphs showing changes in speed and heading per unit time.

Table 5-2. Results of Scenario II.

Type	Description
Final arrival time	3971 UT
First path find completion time	2776 UT
Number of search algorithm calls	1
Average path finding time	2776 UT

Comparing the two results of Scenario I, II, the algorithm developed in this research reached the destination at about 40% faster than the existing offline path planning algorithm. Also, the generation time of first optimal route of this algorithm was about 9.22 times faster than that of the conventional algorithm. Therefore, we can see that the algorithm of this research is more efficient than the off-line pass planning algorithm to reach the destination under obstacle-free environment.

Path Planning with Moving Obstacles

In the second simulation, we will look at the implementation of our research-based path search algorithm in an environment with obstacles.

Scenario III: Path Planning behind a Moving Object

The following values are declared for the simulation.

First, the attributes for autonomous vehicle are determined as follow:

- Location of the car in the map: $(320, 500)$
- Maximum speed of car: $0.4px/UT$
- Maximum acceleration of the car: $0.01925px/UT^2$
- Maximum angular velocity of the car: $0.52deg/UT$
- Position of destination in the map: $(330,100)$

Next, the attributes for moving obstacle are determined as follow:

- Range of initial position in the map: $(320,490)$

- Direction of heading: North
- Range of speed of velocity of the moving obstacle: $0.15px/UT$
- Maximum acceleration of the moving obstacle: $0.01925px/UT^2$
- Maximum angular velocity of the moving obstacle: $0.52deg/UT$

Figure 5-5-A is the first step in this scenario. A moving obstacle moving from south to north and it located above an autonomous vehicle. The autonomous vehicle leaves from the south of the intersection, enters the intersection, and moves towards the north lane. In Figure 5-5-B, the blue solid line represents the vehicle's obstacle search line, and area of risk generated by the moving obstacle can be identified.

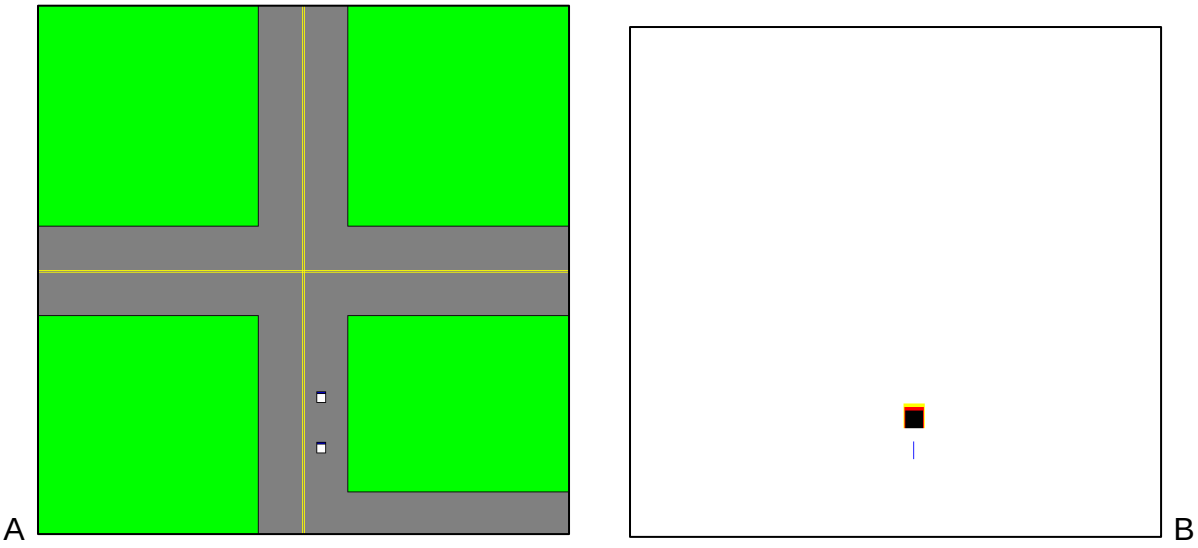

Figure 5-5. Initial state for scenario III. A) World map, B) Moving object map

Figure 5-6-A shows the application of a dynamic window path finding algorithm to find the optimal path to the destination in local search area. As Figure 5-6-B, since there is no currently detected obstacle in front of the vehicle, the autonomous vehicle begins to move along the searched path.

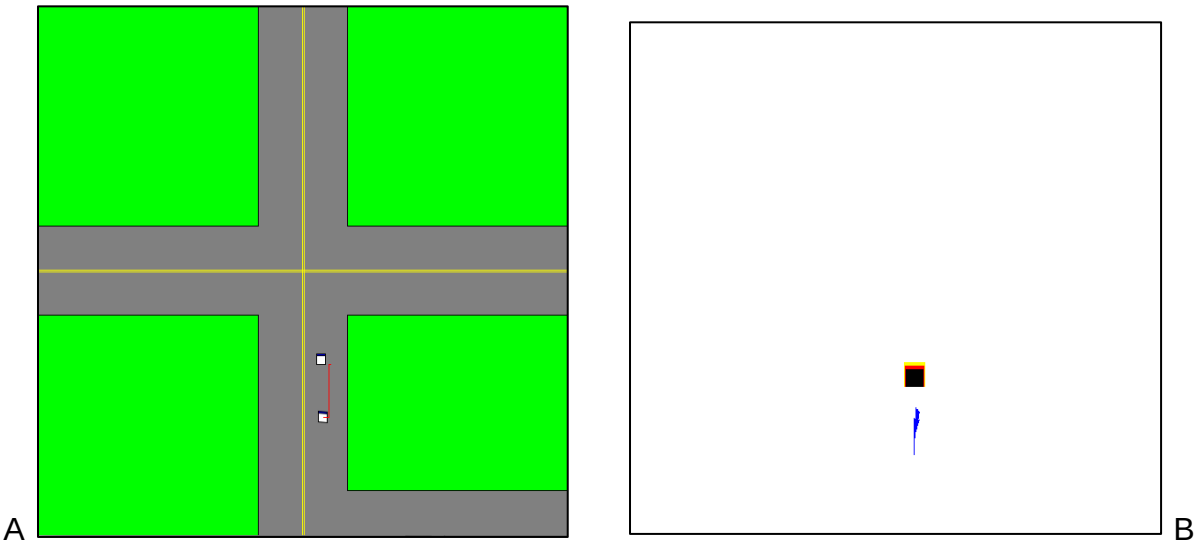

Figure 5-6. Before detecting the obstacle. A) World, B) Moving object map

In Figure 5-7-A and Figure 5-7-B, the autonomous vehicle detected an obstacle ahead when it moving to its destination through continuous route search. When examining the maximum value at the obstacle search stage, the value is the highest risk cost. Then the autonomous vehicle decelerates rapidly to avoid collision with the preceding obstacle.

In Figure 5-8-A and Figure 5-8-B, the autonomous vehicle continues to move behind the obstacles in front. Since this intersection is currently assumed to be single lane, autonomous vehicle cannot attempt to overtake the obstacle, because the risk area created by the obstacle occupies most of the lane. Therefore the autonomous vehicle advances to the speed of the preceding obstacle until it reaches the destination.

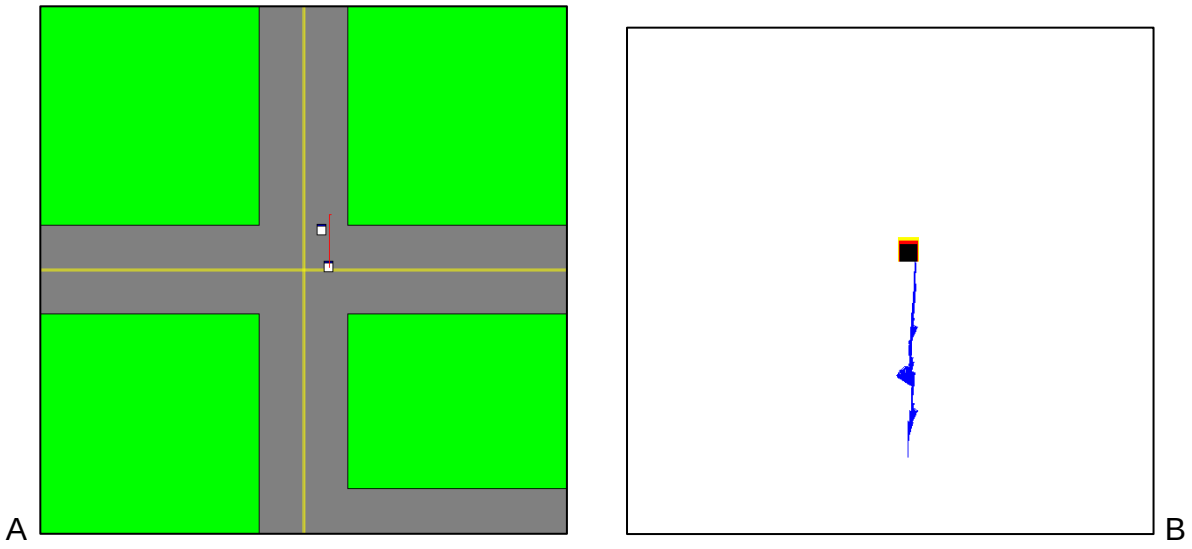

Figure 5-7. After detecting the obstacle. A) World map, B) Moving object map

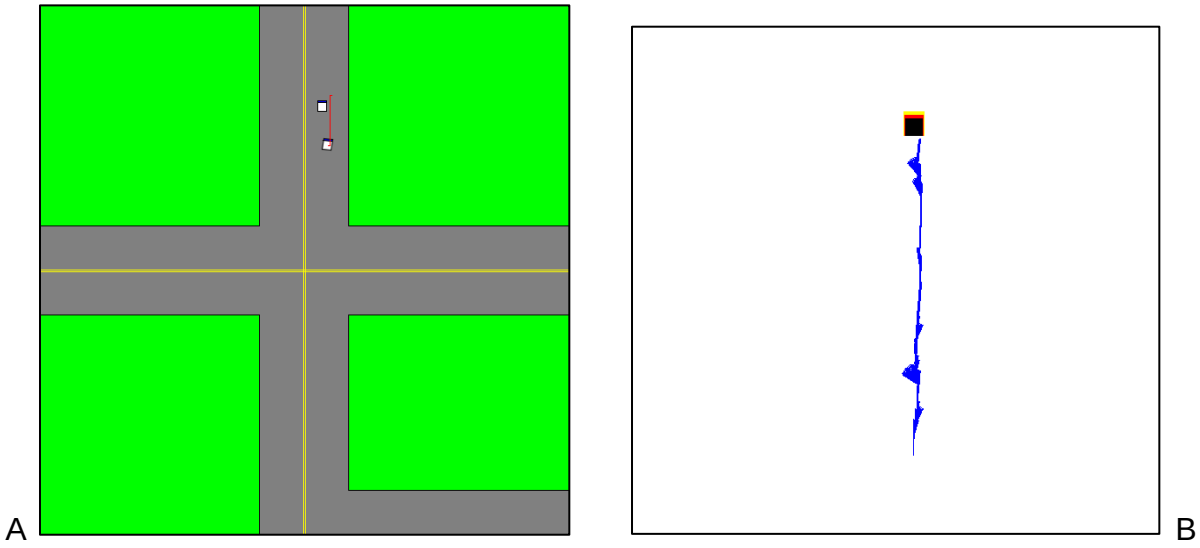

Figure 5-8. Following the obstacle. A) World map, B) Moving object map

Table 5-3. Results of Scenario III.

Type	Description
Final arrival time	2405 UT
First path find completion time	301 UT
Number of search algorithm calls	40
Average path finding time	3.9 UT

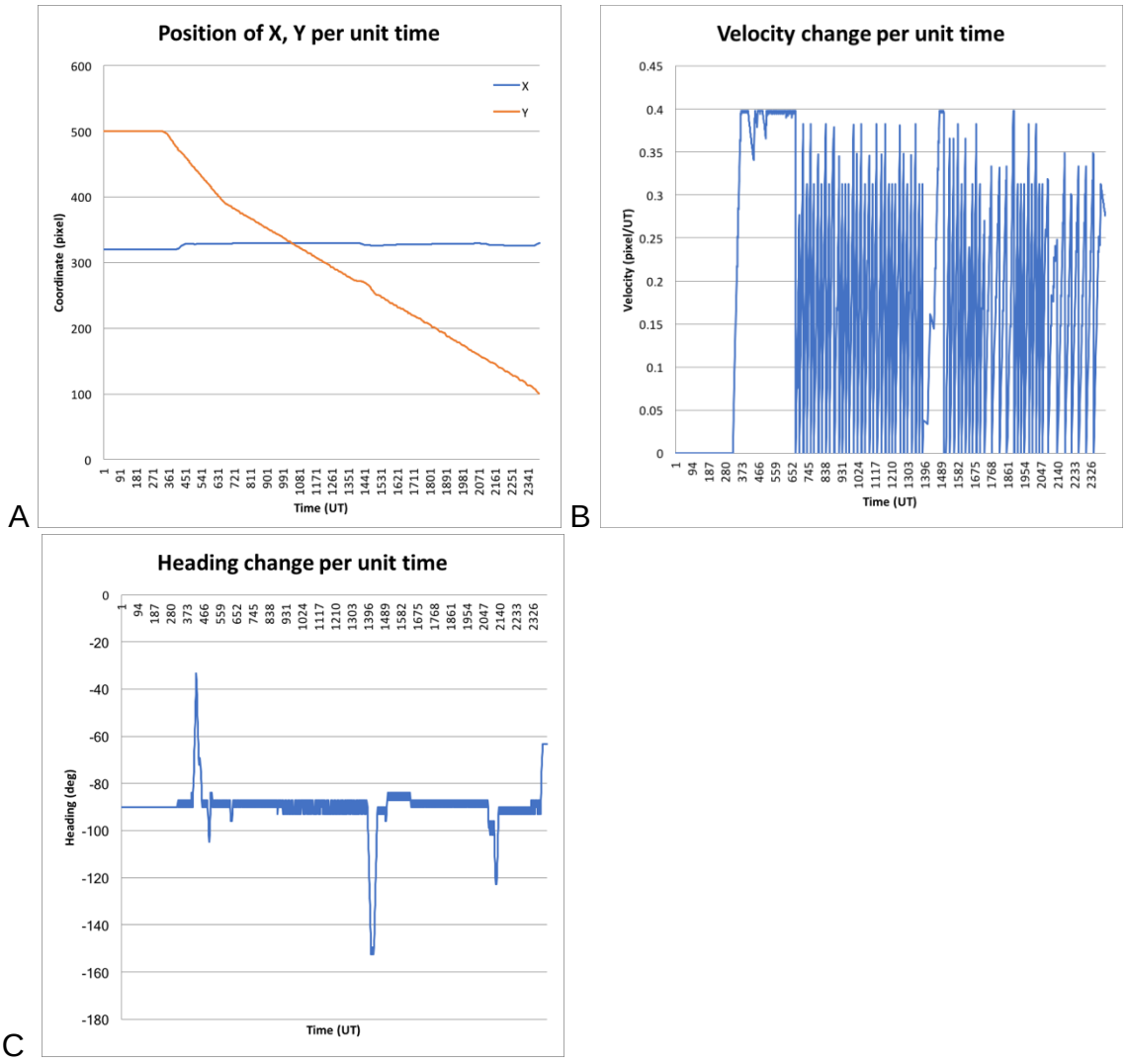

Figure 5-9. Results for scenario III. A) Position change, B) Velocity change C) Heading change

According to Figure 5-9-B, the autonomous vehicle has detected an obstacle ahead of it at approximately 689UT. Since the real of the moving obstacle has the highest risk level, the behavior corresponding to the highest risk level according to the collision detection algorithm of the autonomous vehicle is immediate deceleration. When the vehicle is immediately decelerated, the distance between the preceding obstacle and the autonomous vehicle becomes farther away, so the autonomous vehicle starts to accelerate again. Because of this, the graph of velocity change shows

that the speed of autonomous vehicle is constantly fluctuating up and down. In future research, the collision detection and response algorithm should be improved so that the autonomous vehicle can keep up with the speed of the moving obstacle without any rapid rate of acceleration or deceleration.

Scenario IV: Path Planning with Three Crossing Moving Objects

The following values are declared for the simulation.

First, attributes for autonomous vehicle are determined as follow:

- Location of the car in the map: $(320, 400)$
- Maximum speed of car: $0.4px/UT$
- Maximum acceleration of the car: $0.01925px/UT^2$
- Maximum angular velocity of the car: $0.52deg/UT$
- Position of destination in the map: $(200, 277)$

Next, attributes for first moving obstacle are determined as follow:

- Initial position in the map: $(25, 325)$
- Direction of heading: East
- Speed of velocity of the moving obstacle: $0.4px/UT$
- Maximum acceleration of the moving obstacle: $0.01925px/UT^2$
- Maximum angular velocity of the moving obstacle: $0.52deg/UT$

Next, attributes for second moving obstacle are determined as follow:

- Initial position in the map: $(580, 275)$
- Direction of heading: West
- Speed of velocity of the moving obstacle: $0.2px/UT$
- Maximum acceleration of the moving obstacle: $0.01925px/UT^2$
- Maximum angular velocity of the moving obstacle: $0.52deg/UT$

Finally, attributes for third moving obstacle are determined as follow:

- Initial position in the map: $(370, 325)$
- Direction of heading: East
- Speed of velocity of the moving obstacle: $0.4px/UT$
- Maximum acceleration of the moving obstacle: $0.01925px/UT^2$
- Maximum angular velocity of the moving obstacle: $0.52deg/UT$

Figure 5-10-A is the first step in this scenario. Moving obstacles are three in total, two vehicles moving from west to east and one vehicle moving from east to west. The autonomous vehicle leaves from the south of the intersection, enters the intersection, and moves towards the west lane. In Figure 5-10-B, the blue solid line represents the vehicle's obstacle search line, and two different areas of risk generated by moving obstacles can be identified.

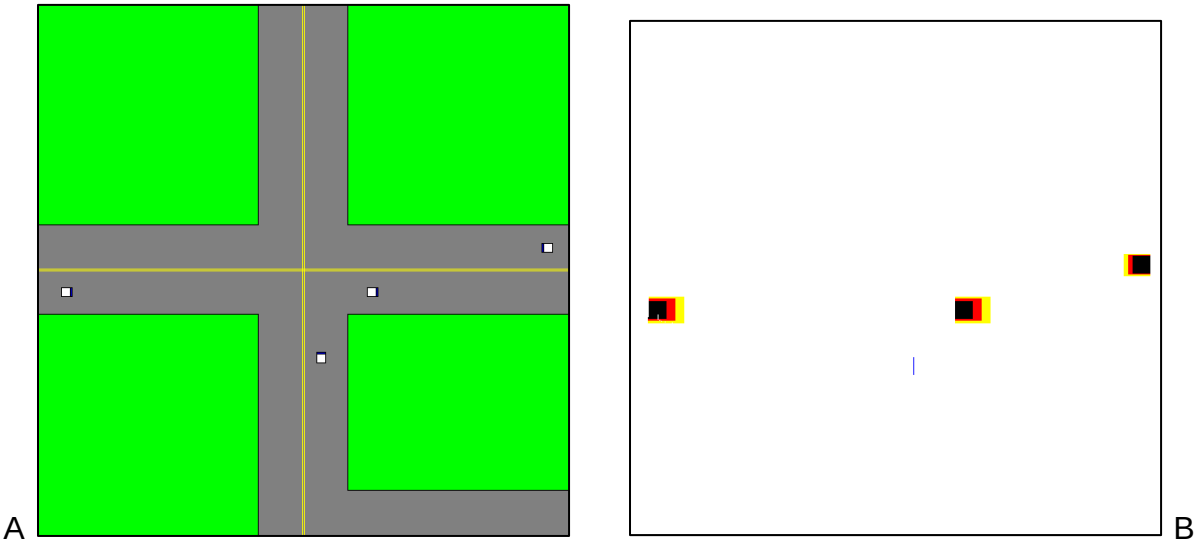

Figure 5-10. Initial state for scenario IV. A) World map, B) Moving object map

Figure 5-11-A shows the application of a dynamic window path finding algorithm to find the optimal path to the destination in local search area. This car is currently in suspension. This is because the obstacle detection line of the autonomous vehicle is included in the risk grid generated by the moving obstacle according to the moving obstacle map of Figure 5-11-B. Since the maximum value of this obstacle detection line indicates a high risk, the car will decelerate.

Figure 5-12-A and Figure 5-12-B show an autonomous vehicle finally moving out of the intersection and moving toward its destination. When examining the maximum

value at the obstacle search stage, the autonomous vehicle starts accelerating along the original path because there is no moving obstacle.

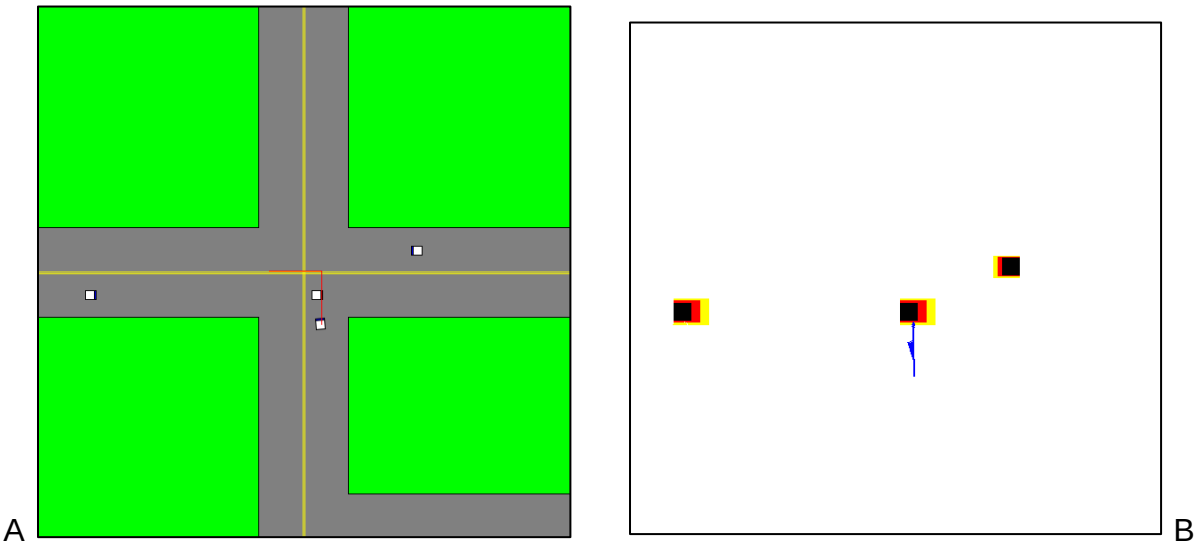

Figure 5-11. Before entering the intersection. A) World, B) Moving object map

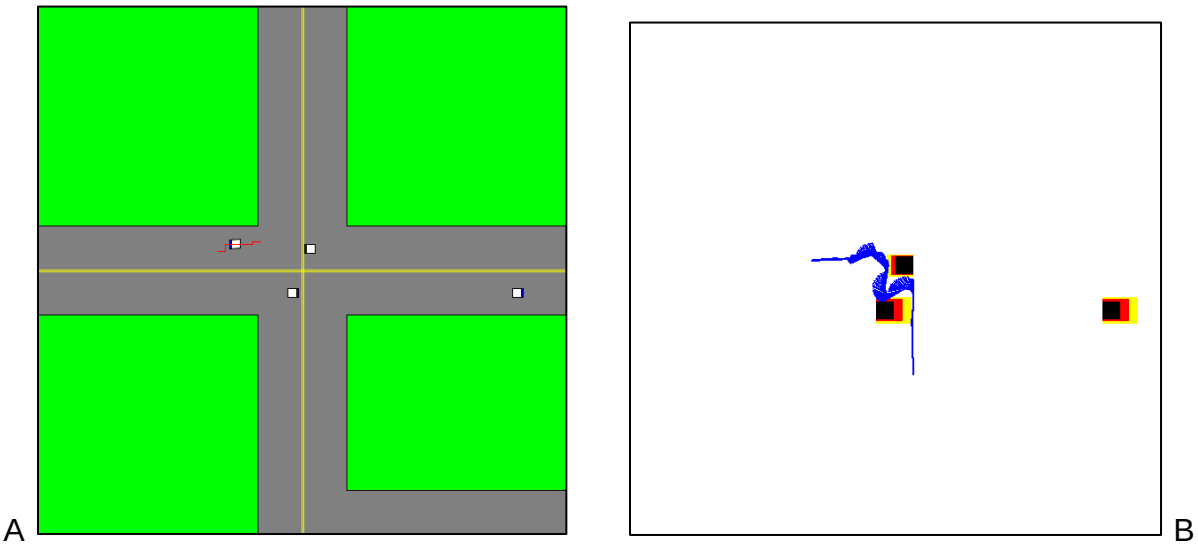

Figure 5-12. After exiting the intersection. A) World map, B) Moving object map

Table 5-4. Results of Scenario IV.

Type	Description
Final arrival time	1341 UT
First path find completion time	559 UT
Number of search algorithm calls	34
Average path finding time	16.94 UT

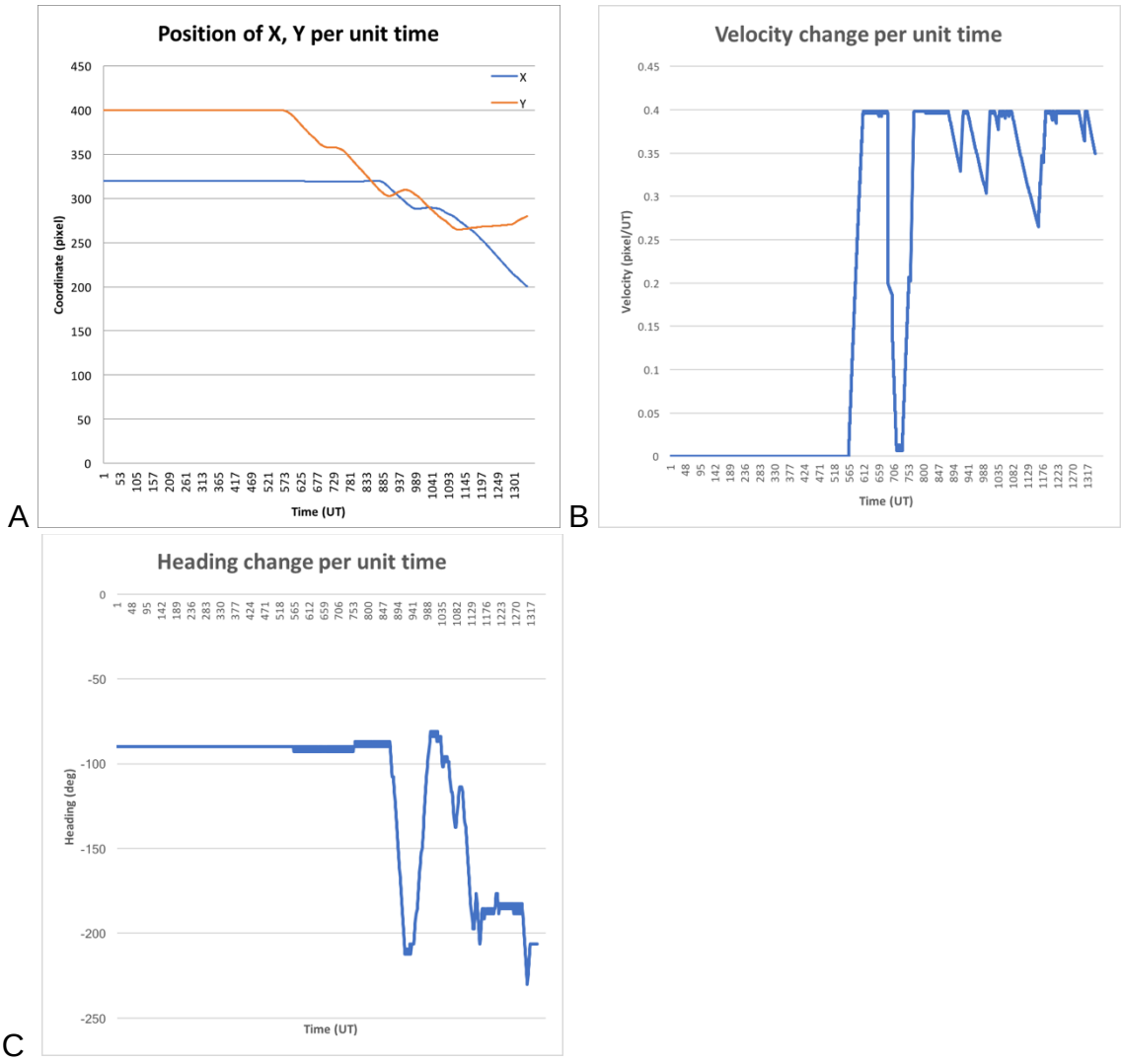

Figure 5-13. Results for scenario IV. A) Position change, B) Velocity change C) Heading change

According to Figure 5-13-B, the autonomous vehicle stopped at about 715 UT because the obstacle moving east blocked the autonomous vehicle's path. Around 757 UT the autonomous vehicle continues to follow the path, and it started left turn from about 925 UT according to Figure 5-13-A and Figure 5-13-C.

Scenario V: Path Planning Coming in Front

The following values are declared for the simulation.

First, attributes for autonomous vehicle are determined as follow:

- Location of the car in the map: $(320, 500)$
- Maximum speed of car: $0.4px/UT$
- Maximum acceleration of the car: $0.01925px/UT^2$
- Maximum angular velocity of the car: $0.52deg/UT$
- Position of destination in the map: $(324,100)$

Next, attributes for moving obstacle are determined as follow:

- Range of initial position in the map: $(275,50)$
- Direction of heading: East
- Range of speed of velocity of the moving obstacle: $0.4px/UT$
- Maximum acceleration of the moving obstacle: $0.01925px/UT^2$
- Maximum angular velocity of the moving obstacle: $0.52deg/UT$

Figure 5-14-A is the initial state in this scenario. A moving obstacle advancing from north and it will take left turn in the intersection to go east. The autonomous vehicle leaves from the south of the intersection, enters the intersection, and moves towards the north lane. In Figure 5-14-B, the blue solid line represents the vehicle's obstacle search line, and area of risk generated by the moving obstacle can be identified.

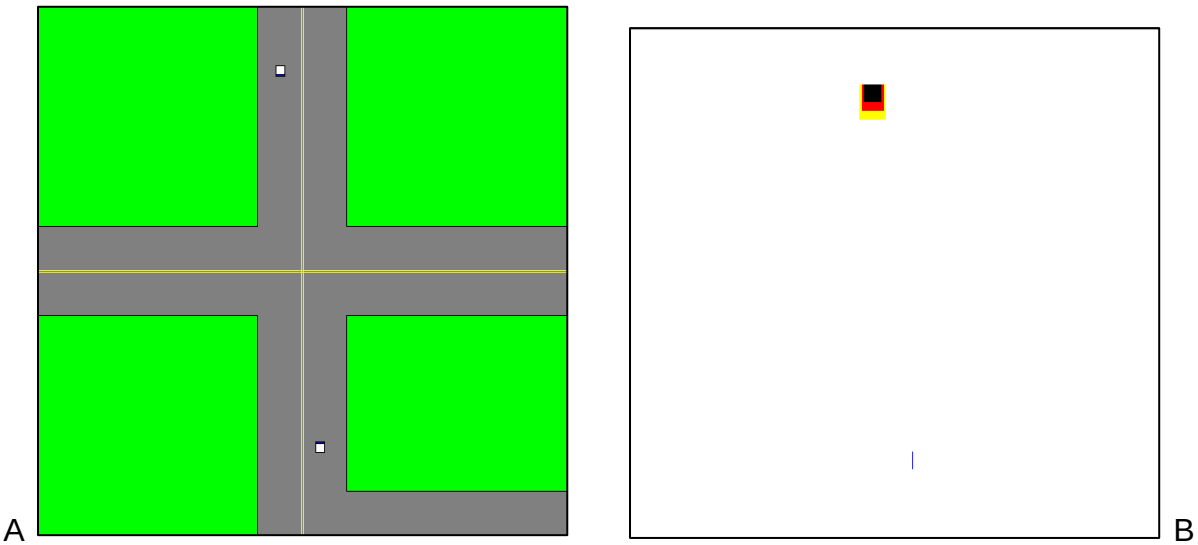

Figure 5-14. Initial state for scenario V. A) World map, B) Moving object map

Figure 5-15-A shows the application of a dynamic window path finding algorithm to find the optimal path to the destination in local search area. As Figure 5-15-B, since

there is no currently detected obstacle in front of the vehicle, the autonomous vehicle begins to move along the searched path.

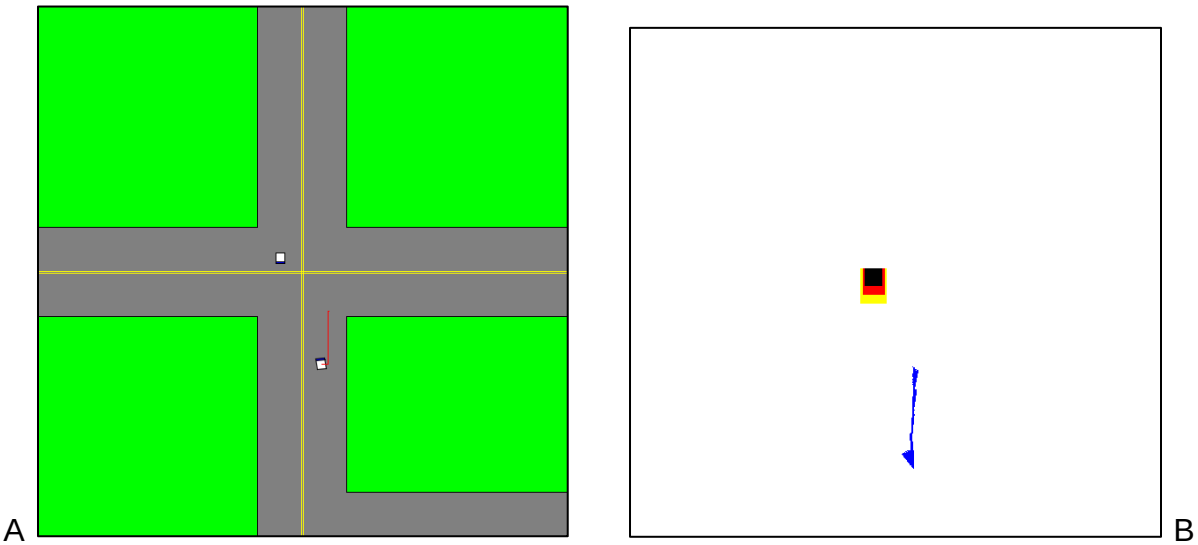

Figure 5-15. Before detecting the obstacle. A) World, B) Moving object map

In Figure 5-16-A and Figure 5-16-B, the autonomous vehicle detected an obstacle ahead when it moving to its destination through continuous route search. When examining the maximum value at the obstacle search stage, the value is start from low risk cost. The autonomous vehicle then slows down its velocity. Because the obstacle is turned left and move to eastbound, examined maximum risk values increasing. So, depends on the risk value, the autonomous vehicle decelerates rapidly to avoid collision with the preceding obstacle.

In Figure 5-17-A and Figure 5-17-B, the autonomous vehicle continues to move behind the obstacles. Since there is no obstacle to observe in front, there is no risk value ahead the vehicle. Therefore, the autonomous vehicle start to accelerate along the path until it reaches the destination.

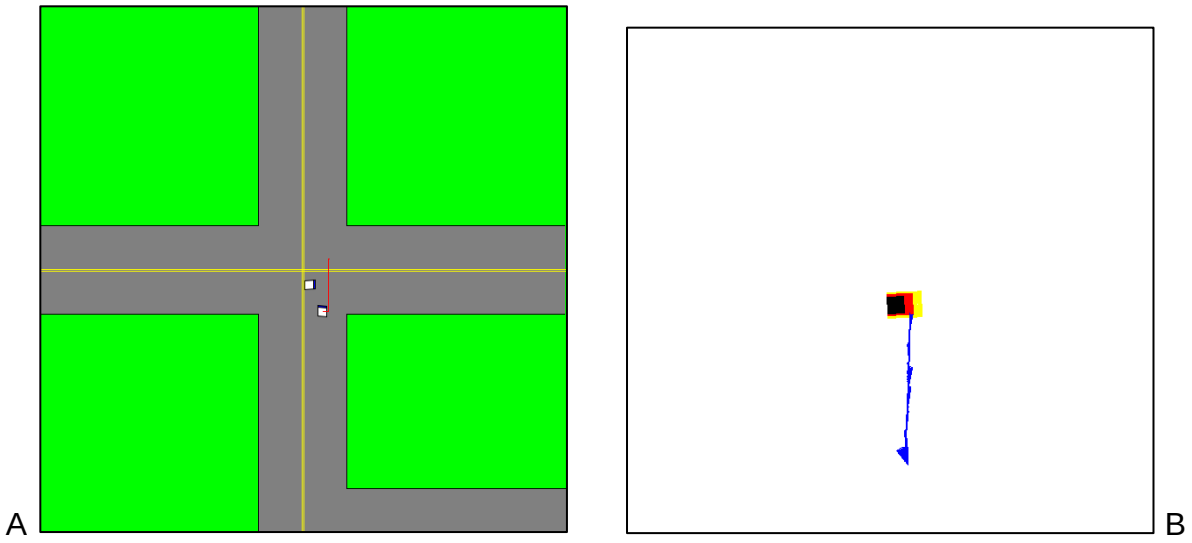

Figure 5-16. After detecting the obstacle. A) World map, B) Moving object map

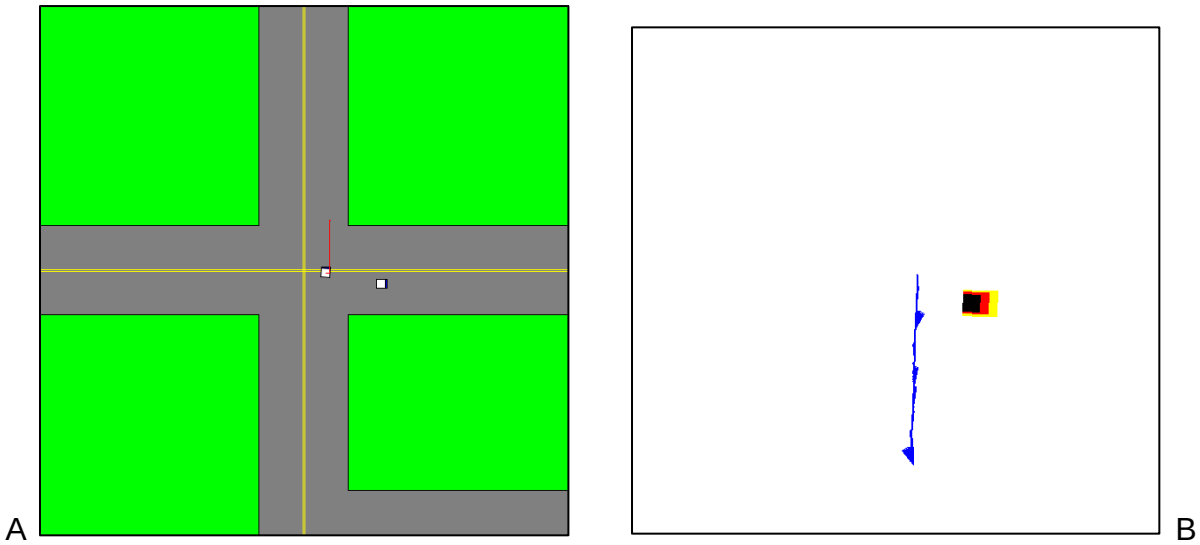

Figure 5-17. Following the path behind obstacle. A) World map, B) Moving object map

Table 5-5. Results of Scenario V.

Type	Description
Final arrival time	2405 UT
First path find completion time	301 UT
Number of search algorithm calls	40
Average path finding time	3.9 UT

According to Figure 5-18-B, the autonomous vehicle stopped at about 769 UT, because the obstacle left from the north to the east in the intersection blocked the

autonomous vehicle's path. After approx. 865 UT, the autonomous vehicle, with obstacles completely exiting the intersection, begins to move north towards the path again.

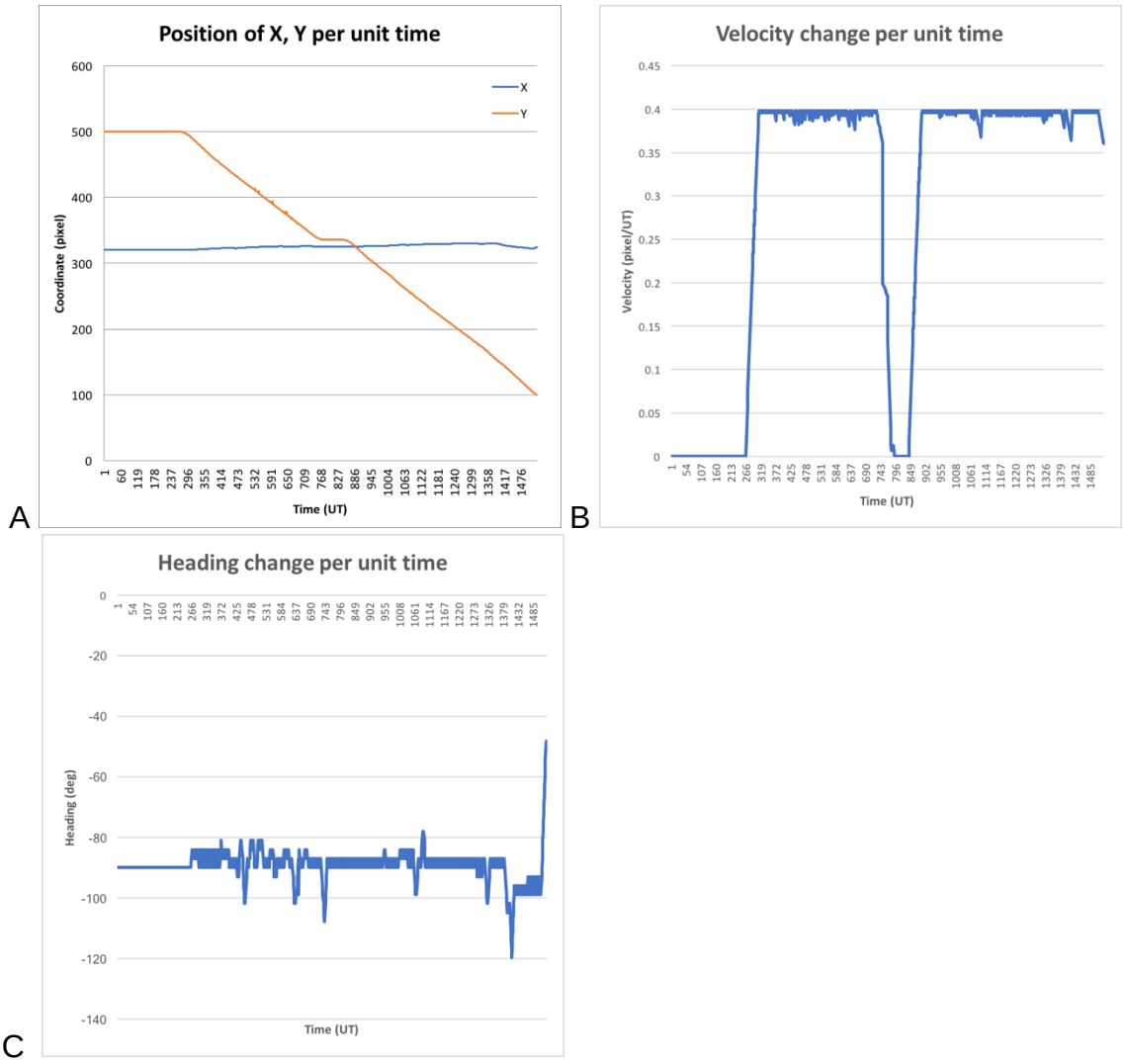

Figure 5-18. Results for scenario V. A) Position change, B) Velocity change C) Heading change

CHAPTER 6
CONCLUSION AND FUTURE WORK

Conclusion

In this research, an algorithm was developed for an autonomous vehicle in an intersection with dynamic obstacles to find an optimal path without obstacles to reach the destination. Previous research was examined that calculated an optimal path to reach the destination. Also, the methods of modeling the real world for applying the path search algorithm were analyzed. Finally, some methods for dynamic obstacle avoidance were also examined.

To solve the previous problems, a path planning algorithm was devised along with a path tracking algorithm for the autonomous vehicle. First, the path planning algorithm repeatedly searches the optimal path within a limited range to reach the destination. This limited range is based on sensors mounted on the vehicle, and the destination has been converted to a temporary destination so that a partial optimization path can be searched within this range. The optimal path to reach the converted destination is calculated by the A* algorithm based on the cost within the limited search range.

The cost for path planning could be divided into a static environmental factor and dynamic environmental factor. The static environmental factor was calculated based on the observed terrain environment around the vehicle, and the dynamic environmental factor was calculated according to the cost based on different risk values within a risk area generated by the expected path of the observed dynamic obstacle. This area of risk varied with the velocity and position of the dynamic obstacle.

Each of the optimal routes calculated based on the cost map is then handed over to the autonomous vehicle, and the vehicle carries out various process to apply the path tracking algorithm along the route and to reach the destination. The behavior of an autonomous vehicle to reach the destination was analyzed, starting with selecting the mission waypoint that the car will compare to the current location and given route to reach the goal. How autonomous vehicles can take optimal behaviors to avoid collisions was also examined, depending on the observed moving obstacles and the range of risk they generate.

Through simulation, the performance difference between the path planning algorithm designed in this study and the conventional offline path planning algorithm was analyzed. As a result, it is confirmed that the path planning algorithm of this study guarantees the autonomous vehicle to move to the optimal route to the destination in a shorter time. It is also confirmed through various scenarios that this algorithm moves along the optimal path to the destination in the direction of minimizing collision with obstacles, in accordance with the environment of various dynamically varying intersections.

Future Work

In this research, the focus was on an algorithm for the path search of an autonomous vehicle moving on a 2D plane. In addition to modeling the world in a 2D plane, one can consider a 3D model that considers the elevation of the terrain and obstacles. In fact, the height of the terrain can be measured through sensors. Using the data, the estimation of movement and behaviors of autonomous vehicle and obstacles will change. Then a more complex algorithm of optimal path search should be considered.

Also, in this simulation, the behavior of an automobile or an obstacle that responds to a signal of a traffic signal or an algorithm for adjusting a virtual traffic signal is not realized. In the future, when an autonomous vehicle becomes available, the role of the smart signal in the intersection to control the autonomous vehicle will also become important. Therefore, it will be necessary to study algorithms that can organically combine the control of this smart signal and the behavior of autonomous vehicles.

The model of the intersection used in this study was a simple intersection of transverse and longitudinal two lanes crossing each other. In real life, however, there may be various roundabouts of three-lane or four-lane crossing intersections, as well as T-shaped crossroads, as well as complex intersections found on highways or rotary-type intersection. A more sophisticated path planning algorithm is needed for autonomous vehicles in accordance with these various intersection type.

Finally, a program called ROS has been spreading around the world for autonomous robots. ROS is a cutting-edge open source software robot control program of the current generation, and many companies, organizations and schools are utilizing this project.

There are many researches on autonomous mobile robots through ROS. Future work will simulate multiple obstacle modules through local world, and each obstacle will create an autonomous vehicle, mechanical design, it is desirable that objects can implement motion predicted by implementing a virtual sensor or the like.

APPENDIX A
STRUCTURE OF CODE

The Java 1.8 version was used to simulate this algorithm because the programming language is multi-platform capable, program coding is simple, and the GUI for input and output is easy to handle.

This simulation can be divided into four packages – simulation (root), Aster, entity and util are described in Table 1 below. Each package and its contents are recorded in the following table.

Table A-1. Packages contained in the implementation of the algorithm.

Package	Description
simulation	Contains main file & GUI
simulation.AStar	Contains A* algorithm and fused map
simulation.entity	Contains entity, extended entity
simulation.util	Contains utility files

Table A-2. Classes contained in package simulation.

Class	Description
Main	Contains main program
moMap	Contains costs of moving obstacle in map
WorldPanel	Implementing GUI, input event (mouse, keyboard)

Table A-3. Classes contained in package simulation.AStar.

Class	Description
GridCell	Define grid cell
HuristicAStar	Define improved A* algorithm for path finding
Map	Fusing map

Table A-4. Classes contained in package simulation.entity

Class	Description
Entity	Define original object
Obstacle	Inherit Entity, to define moving obstacle
Player	Inherit Entity, to define autonomous vehicle

Table A-5. Classes contained in package simulation.util

Class	Description
Clock	Define unit time for simulation
Vector2	Contains vector calculation of mathematic methods
Task	Define iterative running for path planning algorithm

APPENDIX B
SOURCE CODE

Player.java

```
/**
* Initializes a new Player instance.
*/
public Player() {
		this.rotation = DEFAULT_ROTATION;
		this.thrustPressed = false;
		this.thrustBackPressed = false;
		this.rotateLeftPressed = false;
		this.rotateRightPressed = false;
		this.animationFrame = 0;

		this.xCoordinationDestine = 0.0;  // goal initialize
		this.yCoordinationDestine = 0.0;  // goal initialize
		this.missonState = true;  // check mission completed
		this.destRotation = DEFAULT_ROTATION;

		tempX = new int[50000];
		tempY = new int[50000];
		check = new boolean[50000];
		letstart = false;

		missionX = new int[50000];
		missionY = new int[50000];

		missionStep = 0;

		this.whois = 33;

   finderPlayer = new HuristicAStar();

		  try {
			out = new BufferedWriter(new FileWriter("out.txt"));
		} catch (IOException e) {
			System.err.println(e);
			e.printStackTrace();
		}
}

public void update(Main main) {

		/**
		* 020617
		* Check collision
		*/
		int mouseX = (int)this.getPosition().x;
		int mouseY = (int)this.getPosition().y;
		double longX = Math.cos(rotation);
		double longY = Math.sin(rotation);

		max = 0;
		for(int i=0;i<20;i++)
			{
			realtoX[i]=mouseX+(int)(longX * (i+1));
			if(realtoX[i]<0){realtoX[i]=0;}
			else if(realtoX[i]>599){realtoX[i]=599;}
```

```
                realtoY[i]=mouseY-1+(int)(longY * (i+1));
                if(realtoY[i]<0){realtoY[i]=0;}
                else if(realtoY[i]>599){realtoY[i]=599;}
                main.map.gridCell[realtoX[i]][realtoY[i]].isSensor = true;
                }

                for(int i = 0;i<20;i++)
                {
                max = Math.max(max, (int)main.map.gridCell[realtoX[i]][realtoY[i]].type);
                }

        /**
        * Select behavior of vehicle
        */
                if(this.missonState == false && this.autonomousMode ==1)
                        // Mission point Set + Autonomous Mode = 1
                {
                        this.stageOne(main);

        }else if(this.missonState == false && this.autonomousMode ==2 && this.letstart)
                        {
                                this.stageTwo(main);   // path following algorithm
                }

                else{
                }
        }
}

public void fnAcceleration(Vector2 velocity, double rotation, double rate, double coefficient)
        {
                velocity.add(new Vector2(rotation).scale(rate*coefficient));
        if(velocity.getLengthSquared() >= MAX_VELOCITY_MAGNITUDE * MAX_VELOCITY_MAGNITUDE) {
                velocity.normalize().scale(MAX_VELOCITY_MAGNITUDE);
                        }
        }

public void fnDeceleration(Vector2 velocity, double rotation, double rate, double coefficient, double minVelocity)
        {
                        //velocity.add(new Vector2(rotation).scale(THRUST_MAGNITUDE));
                        if(velocity.getLengthSquared() < minVelocity*minVelocity){}
                        else{
                        velocity.add(new Vector2(rotation).scale(-rate*coefficient));
                        if(velocity.getLengthSquared() == 0.0){
                                setVelocity(new Vector2(rotation).scale(getNowVelocity()));
                        }
                }
        }

public void stageOne(Main game){
        // Calculate to go heading
        double dx = this.getPosition().x - this.getXCoordinationDestine();
        double dy = this.getPosition().y - this.getYCoordinationDestine();
        double rad = Math.abs(Math.toDegrees(Math.atan2(dx, dy) + 1.57079632));

        double temp_rotation = Math.abs(Math.toDegrees(rotation));

        for(int i = 0; i < game.entities.size(); i++) { // 0, 1, 2, 3
                Entity a = game.entities.get(i);
                for(int j = i+1; j <= game.staticentities.size() + 1; j++) {
                        Entity b = game.entities.get(j);
                        if(a == game.player) {
```

```
                                        double dxx = this.getPosition().x - b.getPosition().x;
                                        double dyy = this.getPosition().y - b.getPosition().y;
                                        double radd = Math.toDegrees(Math.atan2(dyy, dxx));
                                        if(radd>0){radd=radd-360;}
                                        if(rad != temp_rotation) {
                                rotate((rad-temp_rotation)>0 ? -ROTATION_SPEED : ROTATION_SPEED);
        setVelocity(new Vector2(rotation).scale(Math.sqrt(velocity.getLengthSquared())));
                                        }
                                }
                        }
        }

        // Calculate toGO destance
        this.destLength = Math.sqrt(dx*dx + dy*dy);
        // destLength : changing distance between car and destination
        // gotoDestLength : fixed goTO distance

        if(gotoDestLength == 0.0){
                gotoDestLength = destLength;
        }else{}

        if(destLength > 0 && (destLength / gotoDestLength < 0.3) ){
                if(velocity.getLengthSquared() == 0.0){
                        setVelocity(new Vector2(rotation).scale(getNowVelocity()));
                }
        }

        else if(destLength > 0 ){
                // accelerate
                this.fnAcceleration(velocity, rotation, THRUST_MAGNITUDE, 1.0);
        if(velocity.getLengthSquared() >= MAX_VELOCITY_MAGNITUDE * MAX_VELOCITY_MAGNITUDE) {
                        velocity.normalize().scale(MAX_VELOCITY_MAGNITUDE);
                }
        }
}

public void stageTwo(Main game){
        //checker = 1;  //

        for(int i = 0;i<this.missionStep;i++){
        //System.out.println(i+"th mission waypoint : "+this.missionX[i]+", y : "+this.missionY[i] + ", check : " + this.check[i]);
        }

        //System.out.println(checker + "(=checker)번째 경로 실행!(좌표 x:"+missionX[checker]+", y : "+missionY[checker]+")" + ",
checker = "+check[checker]);
        //System.out.println(checker + "(=checker)번재 현재 위치!(좌표 x:"+this.getPosition().x+", y : " + this.getPosition().y+")");
        // calculate distance between waypoint and current position
        /**
        * get mission waypoint data
        */
        double dx = this.missionX[checker] - this.getPosition().x;
        double dy = this.missionY[checker] - this.getPosition().y;
        if(checker ==1){}

        /**
        * Process to select next mission waypoint
        */
        this.destLength = Math.sqrt(dx*dx + dy*dy);
        // System.out.println("destLEngth = "+destLength);

        double rad = Math.toDegrees(Math.atan2(dy, dx)) ;
```

```
		if(rad>0){
			rad = rad - 360;
			}

		// Calculate toGO heading
		double current_rotation = Math.toDegrees(rotation);

		/**
		* compare distance between waypoint and current position to goal
		*/
		double CtoDx = this.missionX[missionStep-1] - this.getPosition().x;
		double CtoDy = this.missionY[missionStep-1] - this.getPosition().y;
		double ChtoDx = this.missionX[missionStep-1] - this.missionX[checker];
		double ChtoDy = this.missionY[missionStep-1] - this.missionY[checker];
		double CtoDLength = Math.sqrt((CtoDx*CtoDx)+ (CtoDy*CtoDy));
		double ChtoDLength = Math.sqrt((ChtoDx*ChtoDx)+ (ChtoDy*ChtoDy));
		//System.out.println("CtoDLength : "+CtoDLength+", "+"CHtoDLength : "+ChtoDLength);

		if((CtoDLength < ChtoDLength) && this.check[checker] == false)
		{
			checker++;
		}else{

		if(this.check[checker] == true)
		{
if(this.currentStage == false){this.currentStage = true;System.out.println(currentStage);}
			if(destLength > 3.0)
			{
				if(rad != current_rotation) {
				rotate((rad-current_rotation)<0 ? -ROTATION_SPEED : ROTATION_SPEED);
		setVelocity(new Vector2(rotation).scale(Math.sqrt(velocity.getLengthSquared())));
				}
				velocity.add(new Vector2(rotation).scale(THRUST_MAGNITUDE));
		if(velocity.getLengthSquared() >= MAX_VELOCITY_MAGNITUDE * MAX_VELOCITY_MAGNITUDE) {
				velocity.normalize().scale(MAX_VELOCITY_MAGNITUDE);
				}
			}else
			{
				setVelocity(new Vector2(rotation).scale(0));
			}

			if(this.max == 16){
				velocity.add(new Vector2(rotation).scale(-THRUST_MAGNITUDE*5));
				setVelocity(new Vector2(rotation).scale(0));

			}
		}else   // checker is false
		{  if(this.currentStage == true){
			this.currentStage = false;
		}
			if(destLength < 5.0)
			{
				checker++;
			}else
			{
				if(rad != current_rotation) {
			rotate((rad-current_rotation)<0 ? -ROTATION_SPEED : ROTATION_SPEED);
		setVelocity(new Vector2(rotation).scale(Math.sqrt(velocity.getLengthSquared())));
					}
				if(Math.abs(rad - current_rotation)<ROTATION_SPEED*30)
				{
					if(this.max < 10)    // no moving obstacle
```

```
                                {
                                        velocity.add(new Vector2(rotation).scale(THRUST_MAGNITUDE));
                if(velocity.getLengthSquared() >= MAX_VELOCITY_MAGNITUDE * MAX_VELOCITY_MAGNITUDE) {
                                                velocity.normalize().scale(MAX_VELOCITY_MAGNITUDE);
                                        }
                                }else if(this.max < 13)   // low risk
                                {
                                        velocity.add(new Vector2(rotation).scale(0));
if(velocity.getLengthSquared() >= MAX_VELOCITY_MAGNITUDE * MAX_VELOCITY_MAGNITUDE/4) {
                                                velocity.normalize().scale(MAX_VELOCITY_MAGNITUDE/2);
                                        }
                                }else if(this.max < 16)   // medium risk
                                {
                                        velocity.add(new Vector2(rotation).scale(-THRUST_MAGNITUDE));
if(velocity.getLengthSquared() >= MAX_VELOCITY_MAGNITUDE * MAX_VELOCITY_MAGNITUDE/9) {
                                                velocity.normalize().scale(MAX_VELOCITY_MAGNITUDE/3);
                                        }
                                }
                                else if(this.max == 16){  // high risk
                                        velocity.add(new Vector2(rotation).scale(-THRUST_MAGNITUDE*5));
                                        setVelocity(new Vector2(rotation).scale(0));
                                        }
                                }
                        }
                }
}

public double getCurrentVelocity(){
        double currentVelocity;
        double currentVX = this.getVelocity().x;
        double currentVY = this.getVelocity().y;
        currentVelocity = Math.sqrt(currentVX*currentVX + currentVY*currentVY);
        return currentVelocity;
}
```

Obstacle.java

```
public Obstacle(Random random, int positionX, int positionY, double velocity, int direction, int destination) {
        // Position
        this.position = new Vector2(positionX, positionY);
        this.direction = direction;
        this.destination = destination;

        // Velocity
        this.rotation = -Math.PI / 2.0 * direction;
        INIT_VELOCITY = velocity;
        this.velocity =  new Vector2(rotation).scale(INIT_VELOCITY);  // 121916

        this.whois = 56;
passSignalN=passSignalW=passSignalE=passSignalS = false; // Hasn't yet pass any signal
        }



public void update(Main main) {
        // Distance between signals and obstacle
        this.distToSignalS = Math.sqrt((this.getPosition().x - 325.0)*(this.getPosition().x - 325.0)+ (this.getPosition().y -
350.0)*(this.getPosition().y - 350.0));
        this.distToSignalE = Math.sqrt((this.getPosition().x - 350.0)*(this.getPosition().x - 350.0)+ (this.getPosition().y -
275.0)*(this.getPosition().y - 275.0));
        this.distToSignalN = Math.sqrt((this.getPosition().x - 275.0)*(this.getPosition().x - 275.0)+ (this.getPosition().y -
250.0)*(this.getPosition().y - 250.0));
```

```
        this.distToSignalW = Math.sqrt((this.getPosition().x - 250.0)*(this.getPosition().x - 250.0)+ (this.getPosition().y - 325.0)*(this.getPosition().y - 325.0));

        /**
        * 192016 Check Obstacle passing signals
        */

        checkPassSignalSremain(300,350,350,350, this.position);
        checkPassSignalEremain(350,350,250,300, this.position);
        checkPassSignalNremain(250,300,250,250, this.position);
        checkPassSignalWremain(250,250,300,350, this.position);

        checkPassSignalSout(250,300,350,350, this.position);
        checkPassSignalEout(350,350,300,350, this.position);
        checkPassSignalNout(300,350,250,250, this.position);
        checkPassSignalWout(250,250,250,300, this.position);

        /**
        * 01.10.17 Tue
        * set cost of moving obstacle to map
        */
        Vector2 AsPosition = this.getPosition();
        int intAsPositionX = (int)AsPosition.x;
        int intAsPositionY = (int)AsPosition.y;

        /*
        * Set range of risk areas
        */
        int fourtyRe = (int)(80*normVelocity/MAX_VELOCITY);
        int twentyRE = (int)(30*normVelocity/MAX_VELOCITY);
        int tenRe = (int)(20*normVelocity/MAX_VELOCITY);

        int xLRedge = intAsPositionX -fourtyRe;
        if(xLRedge<1){xLRedge=0;}

        int xRRedge = intAsPositionX +fourtyRe;
        if(xRRedge>598){xRRedge=598;}

        int yURedge = intAsPositionY -fourtyRe;
        if(yURedge<1){yURedge=0;}

        int yDRedge = intAsPositionY +fourtyRe;
        if(yDRedge>598){yDRedge=598;}

        int xLRedgeT = intAsPositionX -twentyRE;
        if(xLRedgeT<1){xLRedgeT=0;}


        int xRRedgeT = intAsPositionX +twentyRE;
        if(xRRedgeT>598){xRRedgeT=598;}

        int yURedgeT = intAsPositionY -twentyRE;
        if(yURedgeT<1){yURedgeT=0;}

        int yDRedgeT = intAsPositionY +twentyRE;
        if(yDRedgeT>598){yDRedgeT=598;}


        int xLRedgeTT = intAsPositionX -tenRe;
        if(xLRedgeTT<1){xLRedgeTT=0;}

        int xRRedgeTT = intAsPositionX +tenRe;
```

```
		if(xRRedgeTT>598){xRRedgeTT=598;}

		int yURedgeTT = intAsPositionY -tenRe;
		if(yURedgeTT<1){yURedgeTT=0;}

		int yDRedgeTT = intAsPositionY +tenRe;
		if(yDRedgeTT>598){yDRedgeTT=598;}

		double Nrotation = rotation - (Math.PI/2);

		/*
		* Low risk
		*/
		int velX2 = 20+(int)(this.CURRENT_VELOCITY / MAX_VELOCITY * 20);
		int velY2 = 20+(int)(this.CURRENT_VELOCITY / MAX_VELOCITY * 40);

		int[] testX2 = new int[velX2];
		int[] testY2 = new int[velY2];
		int[] TranTestX2 = new int[velX2];
		int[] TranTestY2 = new int[velY2];

		for(int i=0;i<velX2;i++){
			for(int j=0;j<velY2;j++){
				testX2[i]=intAsPositionX-(velX2/2)+i;
				testY2[j]=intAsPositionY-10+j;
			}
		}
		for(int i=0;i<velX2;i++){
			for(int j=0;j<velY2;j++){
			TranTestX2[i] = (int)(((testX2[i]-intAsPositionX)*Math.cos(Nrotation))-((testY2[j]-
intAsPositionY)*Math.sin(Nrotation))+intAsPositionX);
		TranTestY2[j] = (int)(((testX2[i]-intAsPositionX)*Math.sin(Nrotation))+((testY2[j]-
intAsPositionY)*Math.cos(Nrotation))+intAsPositionY);
			if(TranTestX2[i]<0){TranTestX2[i]=0;}
			else if(TranTestX2[i]>599){TranTestX2[i]=599;}
			if(TranTestY2[j]<0){TranTestY2[j]=0;}
			else if(TranTestY2[j]>599){TranTestY2[j]=599;}
			main.map.gridCell[TranTestX2[i]][TranTestY2[j]].type = 10;
		//	game.map.gridCell[TranTestX2[i]][TranTestY2[j]].cost = 10;
			if(TranTestX2[i]==(int)main.getPlayer().getPosition().x && TranTestY2[i]==(int)main.getPlayer().getPosition().y)
			{
				max = 16;
			}
		}
		}

		/*
		* medium risk
		*/
		int velX1 = 20+(int)(this.CURRENT_VELOCITY / MAX_VELOCITY * 10);
		int velY1 = 20+(int)(this.CURRENT_VELOCITY / MAX_VELOCITY * 20);

		int[] testX1 = new int[velX1];
		int[] testY1 = new int[velY1];
		int[] TranTestX1 = new int[velX1];
		int[] TranTestY1 = new int[velY1];

		for(int i=0;i<velX1;i++){
			for(int j=0;j<velY1;j++){
				testX1[i]=intAsPositionX-(velX1/2)+i;
				testY1[j]=intAsPositionY-10+j;
			}
```

```
		}
		for(int i=0;i<velX1;i++){
			for(int j=0;j<velY1;j++){
			TranTestX1[i] = (int)(((testX1[i]-intAsPositionX)*Math.cos(Nrotation))-((testY1[j]-
intAsPositionY)*Math.sin(Nrotation))+intAsPositionX);
		TranTestY1[j] = (int)(((testX1[i]-intAsPositionX)*Math.sin(Nrotation))+((testY1[j]-
intAsPositionY)*Math.cos(Nrotation))+intAsPositionY);
					if(TranTestX1[i]<0){TranTestX1[i]=0;}
					else if(TranTestX1[i]>599){TranTestX1[i]=599;}
					if(TranTestY1[j]<0){TranTestY1[j]=0;}
					else if(TranTestY1[j]>599){TranTestY1[j]=599;}
					main.map.gridCell[TranTestX1[i]][TranTestY1[j]].type = 13;
			//		game.map.gridCell[TranTestX1[i]][TranTestY1[j]].cost = 13;

			}
		}

		/*
		* high risk
		*/
		int[] testX = new int[20];
		int[] testY = new int[20];
		int[] TranTestX = new int[20];
		int[] TranTestY = new int[20];

		for(int i=0;i<20;i++){
			for(int j=0;j<20;j++){
				testX[i]=intAsPositionX-10+i;
				testY[j]=intAsPositionY-10+j;
			}
		}
		for(int i=0;i<20;i++){
			for(int j=0;j<20;j++){
				TranTestX[i] = (int)(((testX[i]-intAsPositionX)*Math.cos(Nrotation))-((testY[j]-
intAsPositionY)*Math.sin(Nrotation))+intAsPositionX);
		TranTestY[j] = (int)(((testX[i]-intAsPositionX)*Math.sin(Nrotation))+((testY[j]-
intAsPositionY)*Math.cos(Nrotation))+intAsPositionY);
					if(TranTestX[i]<0){TranTestX[i]=0;}
					else if(TranTestX[i]>599){TranTestX[i]=599;}
					if(TranTestY[j]<0){TranTestY[j]=0;}
					else if(TranTestY[j]>599){TranTestY[j]=599;}
					main.map.gridCell[TranTestX[i]][TranTestY[j]].type = 16;
			//		game.map.gridCell[TranTestX[i]][TranTestY[j]].cost = 16;

			}
		}

		CURRENT_VELOCITY = Math.sqrt(this.velocity.getLengthSquared());

		/**
		* Adjust heading to center of lane
		*/
		adjustLaneCenter(this.direction, this.destination,
					(int)this.position.x, (int)this.position.y, this.rotation);


		if(max == 16)
		{
			setVelocity(new Vector2(rotation).scale(0));
		}
}
```

HuristicAStar.java

```
public GridCell[] findPath(Map map)
  {
            minCost = 10;
            map.game.getPlayer().letstartTrigger=true;

     for(int i=map.game.getPlayer().SlidingWindowXL;i<=map.game.getPlayer().SlidingWindowXR;i++){
       for(int j=map.game.getPlayer().SlidingWindowYU;j<=map.game.getPlayer().SlidingWindowYD;j++){
          minCost = Math.min(map.gridCell[i][j].getCost(),minCost);
       }
     }

            this.map = map;
     GridCell.reset();
     pass = 0;
     count = 0;

     tempX = new int[150000];
     tempY = new int[150000];
     check = new boolean[150000];
     missionStep = 0;
     missionDirection = 0;
     edge = new Vector<GridCell>();
     done = new Vector<GridCell>();

     state = NOT_FOUND;
if(GridCell.getStartCell() == null){
       return null;
     }
     if(GridCell.getFinishCell() == null){
       return null;
     }
edge.addElement(GridCell.getStartCell());

     while(state==NOT_FOUND && pass<maxSteps){
                    pass++;
            state = step();  //step() : finding function
            }
       if(state==FOUND){
            main.getPlayer().passFound = true;
            setPath(map);
       }
       else if(state == NO_PATH){
            }  // pass(+1) exceed max_pass
       else{
            }

                                return null;
}

public void setPath(Map map){    // Present completed path

            /**
             * 012317
             * Initialzied variables
             */
            main.getPlayer().checker = 0;
            try {
                                main.getPlayer().out.write("path found!!");
                    main.getPlayer().out.newLine();
                    } catch (IOException e) {
                                e.printStackTrace();
```

```
                    }

GridCell.setShowPath(true);
boolean finished = false;
GridCell next;

GridCell now = GridCell.getFinishCell();
GridCell stop = GridCell.getStartCell();

while(!finished){

   main.world.pass = this.count;
   main.costMap.pass = this.count;
   main.getPlayer().pass = this.count; // 112916
   main.world.trigger = 1;

   tempX[count] = now.getPosition().x;
   tempY[count] = now.getPosition().y;
   if(count==0){check[count]=true;}
   else{
   check[count] = false;}
   next=map.getLowestAdjacent(now);
   now=next;

   now.setPartOfPath(true);

   if(now == stop){
        finished = true;
   }
   count++;

}

for(int i = count;i>0;i--){
   /**
    * AStar -> WorldPanel
    */
   main.world.tempX[this.count - i]= this.tempX[i];
   main.world.tempY[this.count - i]= this.tempY[i];
   main.world.check[this.count - i] = this.check[i];

   main.costMap.tempX[this.count - i]= this.tempX[i];
   main.costMap.tempY[this.count - i]= this.tempY[i];
   main.costMap.check[this.count - i] = this.check[i];
}

main.getPlayer().letstart = true;

for(int i=count;i>=0;i--){
if(i==count-1){main.getPlayer().missionX[this.missionStep]=this.tempX[count-1];
                                                main.getPlayer().missionY[this.missionStep]=this.tempY[count-1];
                                                main.getPlayer().check[this.missionStep]=this.check[count-1];
                                                this.missionStep++;
                                                main.getPlayer().missionStep = this.missionStep;
                                                }
else if(i==count-2){main.getPlayer().missionX[this.missionStep]=this.tempX[count-2];

        main.getPlayer().missionY[this.missionStep]=this.tempY[count-2];
                                                                        main.getPlayer().check[this.missionStep]=this.check[count-2];
                                                                        this.missionStep++;
                                                                main.getPlayer().missionStep = this.missionStep;
```

```
                        if(this.tempX[count-1]==this.tempX[count-2]){missionDirection = 0
                        else{missionDirection = 1
        }
    else{
        int checkDirection;
        if(this.tempX[i]==this.tempX[i+1]){
                checkDirection = 0;

        else{
                checkDirection = 1;
                }

        if(checkDirection==missionDirection){}
        else{
                main.getPlayer().missionX[this.missionStep]=this.tempX[i];
                                main.getPlayer().missionY[this.missionStep]=this.tempY[i];
                                main.getPlayer().check[this.missionStep]=this.check[i];
                missionDirection = checkDirection;
                this.missionStep++;
                main.getPlayer().missionStep = this.missionStep;
        }

        if(i==0){
                main.getPlayer().missionX[this.missionStep]=this.tempX[0];
                                main.getPlayer().missionY[this.missionStep]=this.tempY[0];
                                main.getPlayer().check[this.missionStep]=this.check[0];

                                this.missionStep++;
                                main.getPlayer().missionStep=this.missionStep;
        }
    }
  }
    }
```

Map.java

```
int ROAD = 1;
int FRINGE = 4;
int GRASS = 99;

public void mapInitialize(){
        for(int i=0;i<w;i++){
      for(int j=0;j<h;j++){
        gridCell[i][j].cost = GRASS;  // fringe
      }
    }

        for (int i = 0;i<w;i++){
                        for (int j=250;j<350;j++){
                                gridCell[i][j].cost = FRINGE; // road horizontal
                        }
                }

                for (int i=250;i<350;i++){
                        for (int j=0;j<h;j++){
                                gridCell[i][j].cost = FRINGE; // road vertical
                        }
                }

                for (int i=350;i<600;i++){
                        for (int j=550;j<h;j++){
                                gridCell[i][j].cost = FRINGE;
                        }
```

```
                }

  for (int i = 270;i<280;i++){
                        for (int j=0;j<h;j++){
                                gridCell[i][j].cost = ROAD; // road horizontal
                        }
                }

                for (int i=320;i<330;i++){
                        for (int j=0;j<h;j++){
                                gridCell[i][j].cost = ROAD; // road vertical
                        }
                }

                 for (int i = 270;i<280;i++){
                                for (int j=0;j<h;j++){
                                        gridCell[j][i].cost = ROAD; // road horizontal
                                }
                        }

                        for (int i=320;i<330;i++){
                                for (int j=0;j<h;j++){
                                        gridCell[j][i].cost = ROAD; // road vertical
                                }
                        }

                        for (int i=280;i<320;i++){
                                for (int j=280;j<320;j++){
                                        gridCell[j][i].cost = ROAD; // road vertical
                                }
                        }

                                for (int i=0;i<260;i++){
                                        for (int j=297;j<303;j++){
                                                gridCell[i][j].cost = GRASS; // west
                                        }
                                }
                                for (int i=340;i<600;i++){
                                        for (int j=297;j<303;j++){
                                                gridCell[i][j].cost = GRASS; // east
                                        }
                                }
                                for (int i=297;i<303;i++){
                                        for (int j=0;j<260;j++){
                                                gridCell[i][j].cost = GRASS; // north
                                        }
                                }
                                for (int i=297;i<303;i++){
                                        for (int j=340;j<600;j++){
                                                gridCell[i][j].cost = GRASS; // south
                                        }
                                }
}
```

LIST OF REFERENCES

A. Elfes. (1989). Using occupancy grids for mobile robot perception and navigation. *Computer(Long. Beach. Calif)., 22(6)*.

A. Pasha. (2003). Path planning for nonholonomic vehicles and its application to radiation environments. Master's Thesis, *University of Florida*

A. Stentz. (1994). Optimal and efficient path planning for partially-known environments. *Proc. 1994 IEEE Int. Conf. Robot. Autom.*, 3310-3317.

C. Coulter. (1992). Implementation of the pure pursuit path tracking algorithm. *Report CMU-RI-TR-92-01, Carnegie Mellon University, Pittsburg PA*

C. D. Crane., D. G. Armstrong., M. Ahmed., S. Solanki., D. Macarthur., E. Zawodny., S. Gray., T. Petroff., M. Grifis., C. Evans. (2005). Development of an integrated sensor system for obstacle detection and terrain evaluation for application to unmanned ground vehicles. *Proc. SPIE 5804, Unmanned Ground Vehicle Techonology, 7 156,*

C. D. Crane., D. G. Armstrong., R. Touchton., T. Galluzzo., S. Solanki., J. Lee., D. Kent., M. Ahmed., R. Montane., S. Ridgeway., S. Velat., G. Garcia., M. Griffis., S. Gray., J. Washburn., & G. Routson. (2006). Team CIMAR's NaviGATOR: An Unmanned Ground Vehicle for the 2005 DARPA Grand Challenge. *The 2005 DARPA Grand Challenge, 23*, 311-347.

C. L. Shin., T. T. Lee., & W. A Gruver. (1990). A unified approach for robot motion planning with moving polyhedral obstacles. *IEEE trans. Syst., Man, Cybern, 20(4)*, 903-915

D. Arbuckle., A. Howard., & M. Mataric. (2002). Temporal occupancy grids: a method for classifying the spatio-temporal properties of the environment. *IEEE/RSJ International Conference on Intelligent Robots and System, 1*, 409-414.

E. F. Krause. (1987). Taxicab Geometry: An Adventure in Non-Euclidean Geometry. *Dover Publication*

E. W. Dijkstra. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik, 1,* 269-271.

E. Zawodny. (2003). Multiple vehicle positioning simulation and optimization. Master's Thesis, *University of Florida*

J. Borenstein., & Y. Koren. (1989). Real-time obstacle avoidance for fast mobile robots. *IEEE trans. on System, Man, and Cybernetics, 19(5)*, 1179-1989

J. Borenstein., & Y. Koren. (1991). The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE trans. on Robotics and Automation, 7(3)*, 278-288

K. Fujimura., & H. Samet. (1989). A hierarchical strategy for path planning among moving obstacles. *IEEE trans. On Robotics and, 5(1)*, 61-69

K. O. Arras., J. Persson., N. Tomatis., & R. Siegwart. (2002). Real-time obstacle avoidance for polygonal robots with a reduced dynamic window. *Proceedings 2002 IEEE International Conference on Robotics and Automation, 3*, 3050-3055.

M. Erdmann., & T. Lozano-Perez. (1986). On multiple moving objects. *Proc. 1986 IEEE Int. Conf Robotics Automat*, 1419-1424

Nak Young Ko., & Bum Hee Lee. (1996). Avoidability measure in moving obstacle avoidance problem and its use for motion planning. *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 3*, 1296-1303

N. C. Griswold., & J. Eem. (1990). Control for mobile robots in the presence of moving objects. *IEEE trans. Robotics and Automation, 6(2)*, 263-268

N.H.C. Yung., & C. Ye. (1998). Avoidance of moving obstacles throught behavior fusion and motion prediction. *SMC'98 Conference Proceeding. 1998 IEEE International Conference of Systems, Man, and Cybernetics, 4*, 3424-342 9.

P. Shamsinejad., M. SaraeeBailey., & F. Sheikholeslam. (2010). A new path planner for autonomous mobile robots based on genetic algorithm. *Int. Conf. Comput. Sci. Inf. Technol. 3*, 115-120.

R. D. Ahmad Abu Hatab. (2013). Dynamic Modelling of Differential-Drive Mobile Robots using Langrange and Newton-Euler Methodologies: A Unified Framework. *Adv. Robot. Autom*, 2(2)

S. Ge., & Y. Cui. (2002). Dynamic Motion Planning for Mobile Robots Using Potential Field Method. *Auton. Robots, 13(3)*, 207-222.

BIOGRAPHICAL SKETCH

Mincheul Kim was born in 1987 in Daegu, South Korea. He completed bachelor's degrees in Korea Military Academy. As an officer after his graduation, he served as a platoon leader in the DMZ and as a web programming officer in Headquarter of Republic of Korea Army. For his future career, he pursues his master's degree in the University of Florida supported by the government. Under the direction of his advisor, Carl D. Crane III, he participated in various classes and projects. His current research goal is to create autonomous vehicles for army. After graduation, he will continue his military service and will go to the institute for weapons development research. His interests are related to real-time path planning, artificial intelligence, and recognition.