2018 IEEE Intelligent Vehicles Symposium (IV)
Changshu, Suzhou, China, June 26-30, 2018

Graph Search Based Local Path Planning with Adaptive Node Sampling

Rie Katsuki, Tsuyoshi Tasaki, and Tomoki Watanabe

Abstract— **This paper describes a graph search–based local path planner with an adaptive node sampling according to positions of obstacles. Randomly sampled nodes of a graph in a traversable region for finding a local path can generate a winding path due to connection between the randomly sampling nodes. Node sampling with constant intervals can fail to find a proper path in a case that an interval between adjacent obstacles are smaller than the sampling interval. To solve these problems, our path planner changes node sampling strategy according to obstacle positions; it samples nodes along a reference path, e.g. the center line of the lane, if there are no obstacles, but densely samples nodes around obstacles. After sampling, the planner searches the graph using Dijkstra's algorithm for finding an optimal trajectory. For efficient search for the optimal trajectory, the planner firstly generates a trajectory approximated by piecewise linear lines with minimum cost, and fine-tunes it by adjusting node positions with curved edges having smaller cost. We test the planner through simulations in which an ego vehicle drive along a reference path when there are no obstacles, and densely sample around obstacles to improve robustness against obstacle locations.**

I. Introduction

To reduce traffic accidents, reduce driving burden, and support transportation of the elderly, expectations for automatic driving are increasing. For the safety automatic driving system that does not collide with obstacles, local path planner is one of the key module. Local path planner generates trajectories of several tens of meters in length. The planners generate a trajectory along the street when there are no obstacles present and an avoidance trajectory if obstacles are present in the path of travel.

Pure Pursuit [1] is a well-known path-tracking method. It focuses solely on tracking a reference path and does not aim for generating trajectories that depart from the reference path to avoid obstacles. Werling et al. have proposed a method for reactively avoiding obstacles [2]. This method allocates terminal nodes at constant intervals along the lateral direction of the road, and then connects an ego vehicle with each terminal node using quintic polynomials to generate trajectories with sigmoidal shape. The method selects the trajectory having minimum cost. The ego vehicle can track trajectories very well. On the other hand, it does not focus on optimization of serially connecting plural nodes shown in Fig. 1 although the yellow trajectory is needed when the ego vehicle avoids plural obstacles.

Graph search–based methods have been proposed for generating trajectories by serially connecting nodes of the graph. To find a trajectory the rapidly-exploring random tree

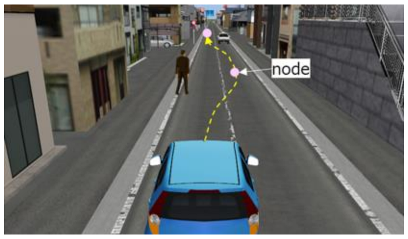

Fig. 1 Two obstacle avoidance path

(RRT) samples nodes randomly across the traversable region [3], while the lattice planner samples nodes with constant intervals [4]. Both methods connect two nodes using a curve and choose a trajectory with minimum cost. Such graph search–based methods are widely used in experiments using actual autonomous vehicles.

RRT-based methods tend to generate winding trajectories due to connection between the randomly sampled nodes. To avoid winding trajectories, Kuwata et al. proposed closed loop RRT (CL-RRT) [3], which build a closed loop using RRT module, controller, and a vehicle model. The controller computes control signals for acceleration and steering to follow a zigzag trajectory generated by the RRT module. Non-holonomic responses of the vehicle model to these control signals smooths the zigzag trajectory. Although this method modifies a zigzag trajectory to a gentle curve, the trajectory is winding inherently. To reduce the winding, Ma et al. combined RRT and an off-line template set [5]. Example of the templates are "Go straight", "Turn left", and so on.

The inverse path generation method [4] [6] in the lattice planner calculates polynomials regarding trajectories through convergence calculations. This method is quite robust against obstacle position, but the planner still fails to generate a trajectory in a case that an interval between adjacent obstacles are smaller than the sampling interval. However, if the lattice planner uses narrow sampling intervals, computation times become enormous.

In applications that can specify a canonical path beforehand, a reference path is often used for such purpose. For example, an automatic driving system often uses a center line of a lane as a reference path. Schwesinger et al. have used a reference path for guidance of node sampling [6]. Their method samples nodes to form a tree from the ego vehicle as a root node towards the reference path. Since the tree area is smaller than the full traversable region, the method can sample at narrower intervals than does the lattice planner method.

Rie Katsuki and Tomoki Watanabe are with Media AI Laboratory, Corporate Research & Development Center, Toshiba Corporation , Kanagawa, Japan (phone: +81-44-549-2393; fax: +81-44-520-1267; e-mail: {rie.katsuki, tomoki8.watanabe}@toshiba.co.jp).

Tsuyoshi Tasaki was with Toshiba Corporation, Kanagawa, Japan. He is now with Meijo University, Aichi, Japan(e-mail: tasaki@meijo-u.ac.jp).

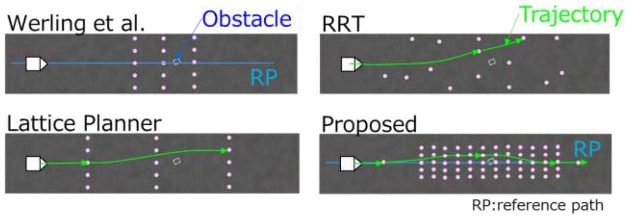

Fig. 2 Sampling strategies of local path planner

If a reference path is available, following the reference path is sufficient for safe driving in areas without obstacles, but dense sampling of nodes is necessary around obstacles. We propose an adaptive sampling method that samples along a reference path if there are no obstacles but densely samples around obstacles. Sampling along a reference path saves time for graph search and leads to comfortable driving along the reference path. Densely sampling around obstacles improves robustness against obstacle locations. Fig. 2 shows our sampling strategy compared with those of other methods.

The remainder of this paper is organized as follows. Section II describes the problem setting, and Section III describes the proposed algorithm. To evaluate the algorithm, Section IV presents the simulation results. Our conclusions are given in Section V.

II. Problem Setting

In this section, we define a problem we consider and coordinate systems.

A. Overview of the Local Path Planner

Fig. 3 shows an overview of an autonomous driving system we consider, which consists of four parts: a perception part that understands surrounding road environment, a navigation part that plans motions of an ego vehicle to reach a goal, a control part that calculates control signals, and an actuator that drives the ego vehicle.

The navigation part consists of three planners, which are described in TABLE I. This paper focuses on the local path planner. Fig. 4 shows the interface for the local path planner.

The reference path represents a canonical path, which is set at the center of a lane with legal speed, or is obtained from a driving log by a human driver. As shown in Fig. 5, a reference path is a series of waypoints shown in blue points, each having a position $(x(j), y(j))$, a rotation $\theta(j)$, and a velocity $v(j)$

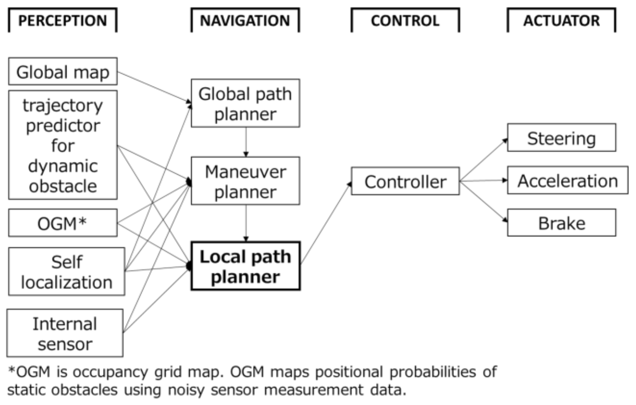

Fig. 3 Overview of the autonomous driving system

TABLE I Path planner roles

Planner	Role
Global path	Route search from start to destination to output a route
Maneuver	Judgment for lane changes, passing, stopping, etc. to output reference path
Local path	Trajectory generation for following the reference path and collision avoidance

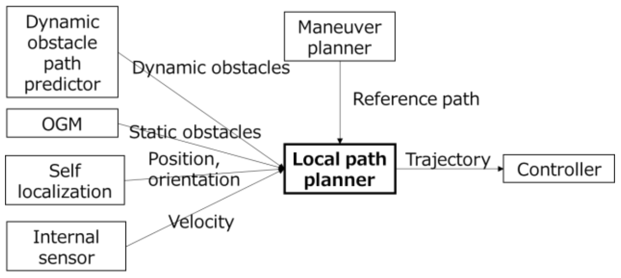

Fig. 4 Interface for the local path planner

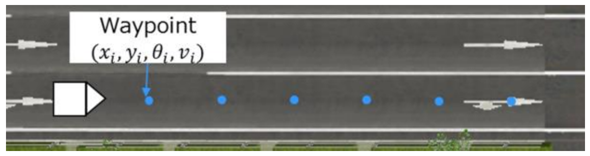

Fig. 5 A reference path

in the world coordinate system, where j represents an index to a waypoint. Since a road with multiple lanes has multiple choice of a reference path, the maneuver planner choose a reference path according to the surrounding road environment.

The local path planner receives information about obstacles and an ego vehicle and then generates trajectories along the reference path when there are no obstacles present and an avoidance trajectory if obstacles are present on the reference path. The local path planner also outputs a series of waypoints as a trajectory.

B. Coordinate System

To handle vehicle motion along a reference path easily, we use the coordinate system [2] shown in Fig. 6. The origin is set on the reference path. s and d axes are longitudinal and lateral direction of the reference path, respectively. The graph at the upper right shows a path in x–y (world) coordinates, and the graph at the lower right shows the path in s–d coordinates.

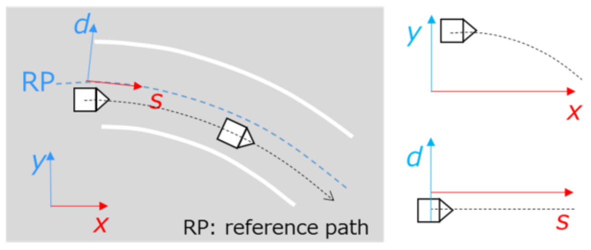

Fig. 6 The s-d coordinate system

III. Proposed Method

This section describes our local path planning algorithm including the adaptive node sampling.

The main function in local path planning is PlanLocalPath(), which is executed repeatedly. Algorithm 1 describes PlanLocalPath(). In line 1, a starting waypoint $\mathrm{w_s}$ and a terminal waypoint $\mathrm{w_g}$ define a region for local path planning. We set a nearest waypoint of a reference path from an ego vehicle for $\mathrm{w_s}$.We set a waypoint of a reference path that L_{lp} meter away from $\mathrm{w_s}$ for $\mathrm{w_g}$. L_{lp} is a user-defined length of a local path. SearchGraph() performs the adaptive node sampling and outputs a piecewise-linear approximated trajectory with the minimum cost, $\boldsymbol{w}(i)$. In the adaptive node sampling, the planner samples nodes along a reference path if there are no obstacles but densely samples around obstacles. ConnectNode() generates a smooth trajectory, $\boldsymbol{p}(t)$, from the approximated trajectory, $\boldsymbol{w}(i)$.

The local path planner is repeatedly executed to deal with changes in surrounding road environment. Repeated execution of the current local path planning algorithm has the following two problems: (1) ConnectNode() outputs $\boldsymbol{p}(t)$ with sigmoidal shape for obstacle avoidance. Local path planner reruns before an ego vehicle following $\boldsymbol{p}(t)$ departs from a reference path. At the next planning, ConnectNode() revises $\boldsymbol{p}(t)$ with sigmoidal shape, so the ego vehicle cannot depart from the reference path. (2)When a relative position and/or orientation between an ego vehicle and an obstacle changes, the look ahead point [1] that is a target point for path following swings as shown in (a) of Fig. 7. The swing results in a jerky steering.

To solve these problems, PlanLocalPath() replaces the reference path with the last trajectory(line 4), which the local path planner uses to generate new trajectories. An ego vehicle can depart from the reference path and the look ahead point do not swing as shown in (b) of Fig. 7.

Algorithm 2 describes the graph search. The search space is defined in a three-dimensional *s–d–t* space, where *t* represent time and is introduced for checking collision with moving obstacles. ConnectNode() calculates $v(t)$ for output of our local path planner. $\mathbf{G}$ denotes a graph consisting of nodes,$\mathbf{V}$, and edges, $\mathbf{E}$.

SearchGraph() adds occupied cells in OGM and predicted path of box-shaped moving obstacles into the search space(line2). We often encountered almost region is not drivable when the search space consider an error ellipse obtained from covariance matrix of predicted path as an obstacle. SearchGraph() adds a predicted path with no errors to ensure drivable region. SearchGraph() determines an obstacle region O depending on the current velocity of the ego vehicle, e.g. O should have wide region around an obstacle for a large ego-motion (line 2). It samples nodes along the reference path at line 3, and samples around obstacles at line 4.

The density of node sampling is a dominant factor in the trajectory shape, not the type of sampling (random sampling or uniform sampling); we introduce uniform sampling simply. We selected sampling density through a trial-and-error adjustment.

SearchGraph() samples nodes shown in Fig. 8. A line of nodes parallel to t-axis overlaps due to a top view. It includes the waypoints of the reference path in O except for the waypoints colliding with an obstacle. SearchGraph() forms an edge at line 5. As Fig. 8 shows, the edge connects adjacent nodes as well as distant nodes to form a trajectory with various curvatures.

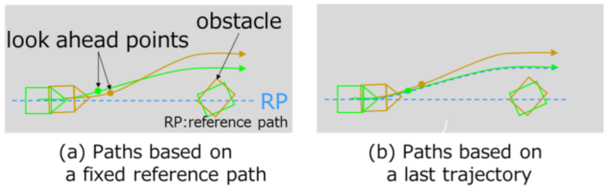
look ahead points
obstacle
RP
RP:reference path
(a) Paths based on a fixed reference path
(b) Paths based on a last trajectory

Fig. 7 Swings of look ahead points

Algorithm 1 PlanLocalPath()

```
input: road map, reference path, position, orientation, and velocity of ego vehicle, OGM, trajectory of moving obstacle(s)
output: $\boldsymbol{p}(t)$ as a series of waypoints of a trajectory
1: Set starting waypoint $\mathrm{w_s}$ and terminal waypoint $\mathrm{w_g}$
2: $\boldsymbol{w}(i)$ =SearchGraph()
3: $\boldsymbol{p}(t)$ = ConnectNode()
4: Replace reference path with $\boldsymbol{p}(t)$
5: return $\boldsymbol{p}(t)$
```

Algorithm 2 SearchGraph()

```
input: road map, reference path, position, orientation, and velocity of ego vehicle, OGM, trajectory of moving obstacle(s), $w_s$, $w_g$
output: approximated trajectory, $\boldsymbol{w}(i)$
1: Set s–d–t search graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$
2: Set obstacle region O in and around obstacles. If no
   obstacles, O is null
3: Sample nodes from $w_s$ to $w_g$ along reference path
   including the current position of the ego vehicle for
   the outside of O
4: Sample nodes around obstacles in O
5: Make linear edges, $\boldsymbol{E}$
6: set node number $i$=0
7: Set a node of current position of the ego vehicle into
   $V(i)$
8: Add $V(i)$ in $\boldsymbol{S}$
9:  Repeat
10:     $\boldsymbol{V}(i+1)$= nodes that are connected to $V(i)$
11:     Get edges $\boldsymbol{E}(i+1)$ connecting $V(i)$ and
          $\boldsymbol{V}(i+1)$
12:     Calculate trajectory costs $\boldsymbol{C_t}(i+1)$
13:     $V^*$= $V(i+1)$ with minimum $C_t(i+1)$
14:     Add $V^*$ to $\mathbf{S}$
15:     $i$=$i$+1
16:   until $V(i)$ reaches $w_g$
17: $\boldsymbol{w}(i)$= Prunes intermediate nodes that do not
      collide with an obstacle from $\boldsymbol{S}$
18: return $\boldsymbol{w}(i)$
```

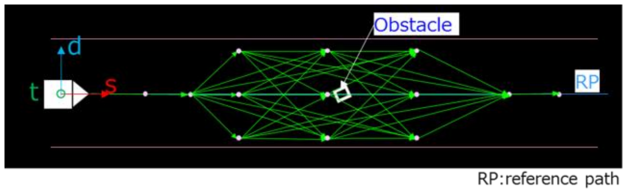

Fig. 8 Nodes and edges

Each node $V(i)$ has a property (s(i), d(i) , $t(i)$, $x(i)$, $y(i)$, $\theta(i)$). (s(i), d(i) , $t(i)$) stand for a position in *s-d-t* coordinate system, ($x(i)$, $y(i)$, $\theta(i)$) stand for a position and orientation in *x-y* coordinate system. For the sake of a vehicle motion along a reference path, (i) is same as an orientation of the waypoint of the reference path.

SearhGraph() gets $\boldsymbol{E}(i+1)$ that do not collides with obstacles. It locates $\boldsymbol{E}(i+1)$ on OGM and then it checks whether $\boldsymbol{E}(i+1)$ are on occupied cells or not. It randomly set waypoints of a moving obstacle within error ellipses, and then it checks collisions between the waypoints and $\boldsymbol{E}(i+1)$.

Various search method such as Dijkstra's algorithm [8] and A* [9] have proposed. Heuristics(a cost between current node and goal node) of A* family [9] [10] [11] can reduce searching time. If a local path planner plans only path and then combines specified velocity, its heuristics is only a distance between a current node and a goal node, thus it can estimate the heuristics easily. In contrast, if a local path planner plans trajectory with comfortable ride, its heuristics has plural parameters such as acceleration, steering, distance, and so on, thus it is difficult to specify values of the heuristics. Our costs as shown below have plural parameters, therefore we select Dijkstra's algorithm.

Dijkstra's algorithm connects nodes with minimum trajectory cost until a node reaches a goal. We calculate the trajectory cost $C_t(i)$ using Eq. (1).

$$C_t(i) = \sum_{i=0}^{n} C_e(i), \tag{1}$$

where $C_e(i)$ is an edge cost for an edge, $E(i)$. Since computing edge costs for various shape of curve edges is time consuming, SearchGraph() uses linear edges. We calculate the edge cost, $C_e(i)$, as a weighted sum of 5 types of costs as shown in Eq. (2);

$$C_e(i) = w_v C_v(i) + w_\psi C_\psi(i) + w_{\varepsilon d} C_{\varepsilon d}(i) + w_{\varepsilon t} C_{\varepsilon t}(i) + w_\rho C_\rho(i) , \tag{2}$$

where

$C_v(i)$: cost for acceleration

$C_\psi(i)$: cost for yaw angular speed

$C_{\varepsilon d}(i)$: cost for lateral offset from the reference path

$C_{\varepsilon t}(i)$: cost for time deviation from the reference path

$C_\rho(i)$: cost for distance from the closest obstacle

w_v, w_ψ, $w_{\varepsilon d}$, $w_{\varepsilon t}$, w_ρ : weights

To improve driving comfort, we introduce $C_v(i)$ and $C_\psi(i)$. $C_v(i)$ works for avoiding quick acceleration. Since Dijkstra's algorithm needs costs relating to distance, Eq. (3) multiplies $a(i)$ by $\mathrm{d_E}(i)$, where $d_E(i)$ represents a distance between the two connected nodes and $a(i)$ is a magnitude of acceleration between the two connected nodes computed by Eq. (4)-(7).

$$C_v(i) = a(i) \times d_E(i), \tag{3}$$

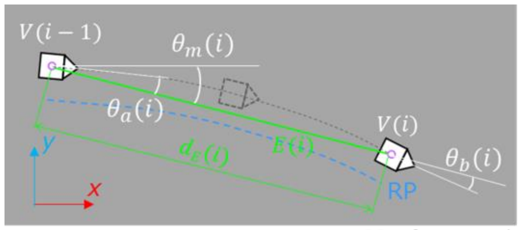

Fig. 9 Orientations for calculation of yaw angular speed $\omega(i)$

$$a(i) = \left\| \frac{v(i)-v(i-1)}{\Delta t(i)} \right\|, \tag{4}$$

$$v(i) = \frac{d_E(i)}{\Delta t(i)}, \tag{5}$$

$$d_E(i) = \sqrt{(x(i)-x(i-1))^2 + (y(i)-y(i-1))^2}, \tag{6}$$

$$\Delta t(i) = t(i) - t(i-1) \tag{7}$$

Jerk (the derivative of acceleration with respect to time) is also one index for comfortable driving. A jerk value of 0 represents constant acceleration (constantly increasing or decreasing velocity). When we set jerk instead of acceleration for $C_v(i)$, our local path planner outputs a trajectory that constantly increasing or decreasing velocity until a speed limit once it starts speed up/down. Since large difference of velocity makes driving comfort decrease, we introduce acceleration.

$C_\psi(i)$ works for avoiding abrupt steering. Since our method connects two nodes with a linear edge, we use an approximated cost using an angular velocity of the ego vehicle as shown in Eqs. (8) – (13) and Fig. 9. $\theta_m(i)$ approximates an orientation of an ego vehicle at a center point of $\boldsymbol{E}(i)$. $\Delta\theta_a(i)$ calculates a difference of orientations of an ego vehicle between $\theta(i-1)$ and $\theta_m(i)$. $\Delta\theta_b(i)$ calculates a difference of orientations of an ego vehicle between $\theta_m(i)$ and $\theta(i)$.

$$C_\psi(i) = \omega(i) \times d_E(i), \tag{8}$$

$$\omega(i) = \frac{|\Delta\theta(i)|}{\Delta t(i)}, \tag{9}$$

$$\Delta\theta(i) = |\Delta\theta_a(i)| + |\Delta\theta_b(i)|, \tag{10}$$

$$\Delta\theta_a(i) = \theta_m(i) - \theta(i-1), \tag{11}$$

$$\Delta\theta_b(i) = \theta(i) - \theta_m(i), \tag{12}$$

$$\theta_m(i) = \tan^{-1}\left(\frac{y(i)-y(i-1)}{x(i)-x(i-1)}\right) \tag{13}$$

We introduce $C_{\varepsilon d}(i)$ and $C_{\varepsilon d}(i)$ to follow the reference path, and we introduce $C_\rho(i)$ to safely avoid obstacles.

ConnectNode() fine-tunes terminals and generates curve trajectories. The algorithm for ConnectNode() is similar to Algorithm 2, with three differences: (i) the algorithm samples nodes around waypoints only (line 2 to 4), (ii) generates curved edges (line 5), and evaluate trajectory cost with different cost function.

We introduce Werling's method for generating curved edges and computing edge costs [2]. This method can generate human-like and physically feasible trajectories. It first defines s-d coordinate system as shown in Fig. 6, and generates lateral trajectories (trajectories in *t-d* plane) and longitudinal trajectories (trajectories in *t-s* plane), respectively, using quintic polynomials based on Kelly and Navy's method [6]. It next chooses a set of a lateral trajectory and a longitudinal trajectory having minimum cost. It combines them to output a trajectory in the world coordinate system. Edge cost is a sum of a cost of a lateral trajectory and a cost of a longitudinal trajectory. The two costs are calculated using jerks and deviations from a reference path.

IV. Experiments

We evaluated the proposed method through simulations to compare trajectories with those resulting from the CL-RRT [3] and lattice planner [4].

As shown in Fig. 10, a collision detection module adds a margin based on the size of the box-shaped ego vehicle and changes the ego vehicle in a point mass. The module sets enlarged obstacles into an occupancy grid map (OGM), and then checks whether the ego vehicle will collide with an occupied cell. We add small change of occupied area at each path planning to simulate sensing errors or quantization errors of OGM.

A. *Evaluation of Obstacle Avoidance Trajectories*

We assume one-lane roads with obstacles as shown in Fig. 11. There are ten courses and the lengths of all courses are 1 kilometer. We set randomly the shapes of the roads, number of obstacles, sizes of the obstacles, and locations of the obstacles. The three methods generated trajectories for the five course.

Fig. 12 shows trajectories when an obstacle is not located inside the lane. The trajectories are structured as waypoints at intervals of 0.1 second. Our method generates a trajectory along the reference path because of node sampling along a reference path. Lattice planner chose an almost linear trajectory with minimum cost. The trajectory of CL-RRT is a little bit winding and near a boundary of the lane.

Fig. 13 shows the number of obstacles that the three methods could avoid. The total number of obstacle is 89. CL-RRT and proposal method avoided 87 obstacles. Lattice planner fails when obstacles are located with narrow intervals shown in Fig. 14. The trajectory of lattice planner shows last successful trajectory. On the other hand, our method could generate trajectory because of densely sampling of nodes around obstacles. CL-RRT also could generate trajectory because of random sampling of nodes. CL-RRT also fails to generate trajectory when it unfortunately cannot add nodes inside the lane shown in Fig. 15. Our method output colliding path two times at ConnectNode() due to small sampling region. We should enlarge a sampling region for ConnectNode() to solve it.

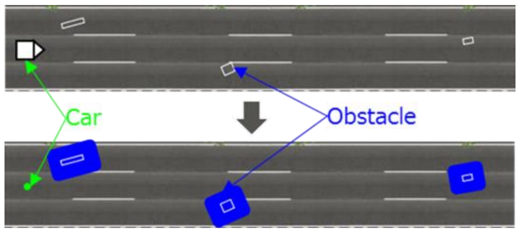

Fig. 10 Collision check

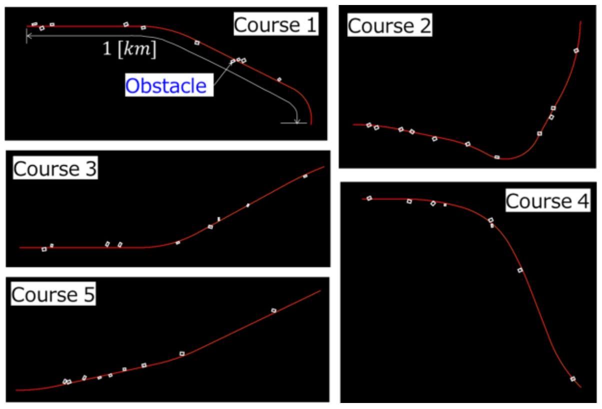

Fig. 11 A part of the ten courses

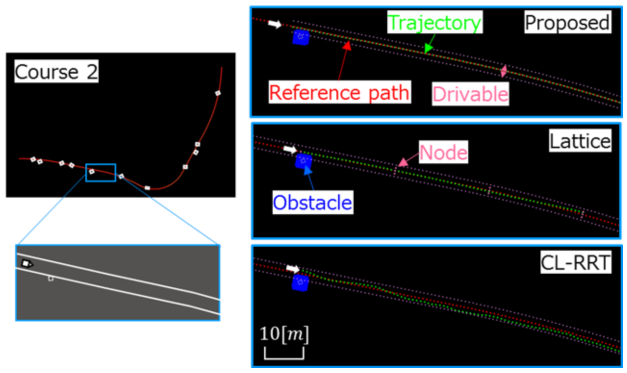

Fig. 12 Trajectories when an obstacle is not located inside the lane

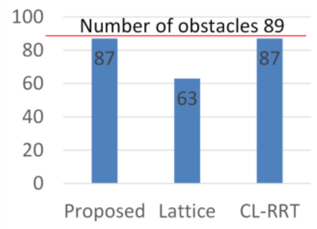

Fig. 13 The number of avoided obstacles

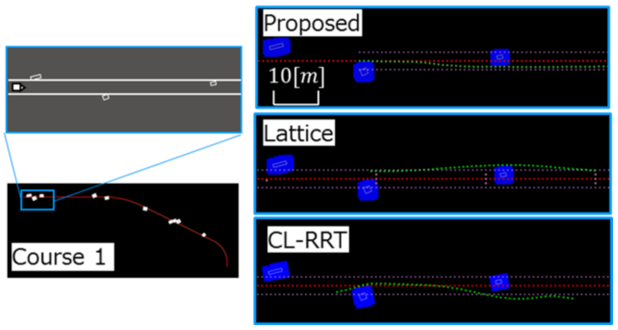

Fig. 14 Trajectories of course 1

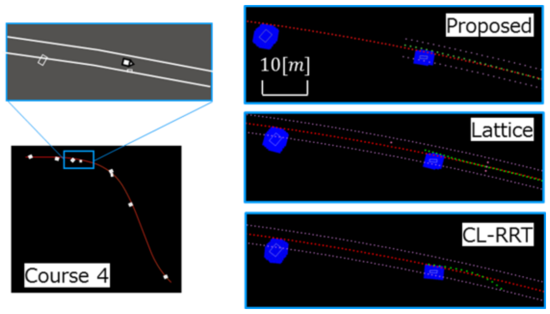

Fig. 15 Trajectories of course 4

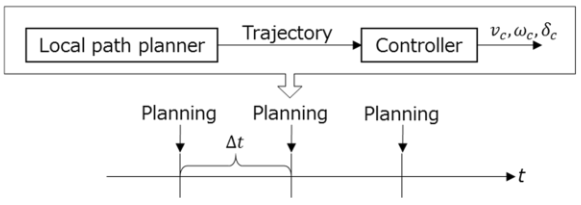

Fig. 16 Cyclic planning

In order to evaluate comfort driving, we calculated lateral accelerations of trajectories. The local path planner and the controller run at intervals of Δt (0.1 second), shown in Fig. 16. We implemented the controller based on Pure Pursuit in Autoware [12] [13]. The controller outputs v_c (velocity), ω_c (angular velocity), and δ_c (steering). We calculated a_ω(lateral acceleration) using Eq. (14).

$$a_\omega = v_c \times \omega_c \tag{14}$$

We define 3m/s^2 (about 0.3G) as a threshold of uncomfortable [14]. a_ω of our method exceeded more than 3m/s^2 88 times (1.1%). a_ω of lattice planner exceeded 53 times (0.6%), a_ω of CL-RRT exceeded 2300 times (58.2%) shown in Fig. 17. Lattice planner can generate smooth paths, while we cannot calculate a_ω when obstacles are located with narrow intervals due to abort of generating paths. Random sampling and low number of nodes leads to large a_ω of CL-RRT.

B. *Evaluation of Computational time*

TABLE II describes computational time. The time of our method is less than 1 minute despite the 21000 nodes due to cut-down of the nodes at SearchGraph(). Lattice planner needs more than 1 minute due to accessing each cell in OGM to calculate distances between static obstacles and a path. Calculating the distances using identified static obstacles can reduce the computation time. Reference [15] introduces 0.65 second. CL-RRT breaks sampling after Δt (0.1 second).

V. Conclusion

We proposed a novel sampling method that samples along a reference path and around obstacles. Ego vehicles drive along a reference path when there are no obstacles, and densely sample around obstacles to improve robustness against obstacle locations. The simulation result shows that the number of avoiding obstacle of our method is larger than lattice planner, in addition, trajectories of our planner are smoother than CL-RRT. In future work, we will adjust a density and a region of a node sampling to balance a computation time and quality of a path. We also will evaluate our method for moving obstacles. Collision checks with many moving obstacles need computation time. Reducing of the number of collision check and low density of node sampling corresponding a covariance of a moving obstacle have a possibility of saving of the computation time.

TABLE II Computational time

	Proposed	Lattice	CL-RRT
The number of nodes	21000	9	117
Computational time[sec]	44	114*	0.2

CPU : 3.0 GHz, Intel$^{(R)}$ Core$^{(TM)}$i7
* 0.65second at [15]

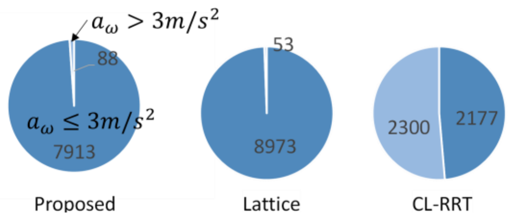

Fig. 17 Lateral accelerations of three methods

Acknowledgment

We would like to thank Yusuke Tezuka and Shouta Terui belonging to Toshiba Information Systems Corporation for their constant support.

References

[1] R. Craig Coulter, "Implementation of the Pure Pursuit Path Tracking Algorithm. Tech. Report, CMU-RI-TR-92-01, Robotics Institute, Carnegie Mellon University, January, 1992

[2] M. Werling et al., "Optimal trajectories for time-critical street scenarios using discretized terminal manifolds," The International Journal of Robotics Research, vol. 31, No. 3, pp. 346–359, March. 2012.

[3] Yoshiaki Kuwata, Justin Teo, Gaston Fiore, Sertac Karaman, Emilio Frazzoli, and Jonathan P. How, "Real-Time Motion Planning With Applications to Autonomous Urban Driving," IEEE Transactions on Control Systems Technology, vol. 17, no. 5, Sep. 2009.

[4] Wenda Xu, Junqing Wei, John M. Dolan, Huijing Zhao and Hongbin Zha, "A Real-Time Motion Planner with Trajectory Optimization for Autonomous Vehicles," Proceedings of IEEE International Conference on Robotics and Automation, pp. 2061-2067, May 2012.

[5] Liang Ma, Jianru Xue, Kuniaki Kawabata, Jihua Zhu, Chao Ma, Nanning Zheng, "A Fast RRT Algorithm for Motion Planning of Autonomous Road Vehicles," IEEE 17th International Conference on Intelligent Transportation Systems (ITSC), October 2014.

[6] Alonzo Kelly and Bryan Nagy, "Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control," The International Journal of Robotics Research, vol. 22, No. 7–8, pp. 583–601, July 2003.

[7] Ulrich Schwesinger, Martin Rufli, Paul Furgale and Roland Siegwart, "A Sampling-Based Partial Motion Planning Framework for System-Compliant Navigation along a Reference Path," Proceedings of IEEE Intelligent Vehicles Symposium (IV) pp. 391–396, June 2013.

[8] Dijkstra, E.W., "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik 1, pp.269-271, 1959.

[9] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov et al., "Junior: The Stanford Entry in the Urban Challenge," Jounal of Field Robotics, vol. 25, no.9, pp. 569–597, Sep. 2008.

[10] Ji-Wung Choi and Kalevi Huhtala, "Constrained Path Optimization with B´ezier Curve Primitives," Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, Sep. 2014.

[11] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz and Sebastian Thrun, "Anytime dynamic A*: An anytime, replanning algorithm," Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling, June, 2005.

[12] "Autoware," https://www.autoware.ai/

[13] Hiroki Ohta, Naoki Akai, Eijiro Takeuchi, Shinpei Kato and Masato Edahiro, "Pure Pursuit Revisited: Field Testing of Autonomous Vehicles in Urban Areas," IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications, 2016

[14] "Automatically Commanded Steering Function (ACSF) - Transport - Vehicle Regulations - UNECE Wiki," https://www2.unece.org/wiki/pages/viewpage.action?pageId=25265606

[15] Matthew McNaughton, Chris Urmson, John M. Dolan and Jin-Woo Lee, "Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice," Proceedings of IEEE International Conference on Robotics and Automation, pp. 4889-4895, May 2011.