Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice

Matthew McNaughton, Chris Urmson, John M. Dolan, and Jin-Woo Lee

***Abstract*— We present a motion planner for autonomous highway driving that adapts the state lattice framework pioneered for planetary rover navigation to the structured environment of public roadways. The main contribution of this paper is a search space representation that allows the search algorithm to systematically and efficiently explore both spatial and temporal dimensions in real time. This allows the low-level trajectory planner to assume greater responsibility in planning to follow a leading vehicle, perform lane changes, and swerve around obstacles in the presence of other vehicles. We show that our algorithm can readily be accelerated on a GPU, and demonstrate it on an autonomous passenger vehicle.**

I. INTRODUCTION

A viable autonomous passenger vehicle must be able to plot a precise and safe trajectory through busy traffic while observing the rules of the road and minimizing risk due to unexpected events such as sudden braking or swerving by another vehicle, or the incursion of a pedestrian or animal onto the road. The planner must be able to produce a plan within a small fixed time window. In this paper we are concerned with the production of precise short-term plans of the vehicle location, valid for the next several seconds.

The state lattice[2] is a method for inducing a discrete search graph on a continuous state space while respecting differential constraints on motion. It was demonstrated in the 2007 DARPA Urban Challenge[8], where it was used to plan motions in parking lots. The lattice planner formulation was not readily applicable to on-road driving scenarios due to the high density of vertices and edges that would have been necessary to represent paths conforming to lanes. In this paper we adapt the lattice to planning dynamically-feasible motions in structured environments such as roads with moving traffic, by conforming the lattice to the environment.

Our planner also uses a novel spatiotemporal search graph that combines precise constraint satisfaction along selected spatial dimensions with resolution-equivalent pruning of states along temporal dimensions, with the effect that a large number of variations in time and velocity can be examined without an undue increase in the size of the state space.

Typical efforts to construct a planner that can operate intelligently in complex environments rely on a decomposition of the planning solution into a hierarchy of planners working on successively more concrete representations of the search space, and finer discretizations in time. Each planner in the hierarchy must have a model of the other planners. These models are necessarily flawed or else the decomposition would be redundant. Mismatches in expectations of behavior between the planners can cause instability in planned motions, especially when higher-level planners give commands to lower level planners that turn out to be infeasible. Our proposed planning framework mitigates this problem by assuming more responsibility at the lower levels — those that can decide whether an action is feasible before attempting to execute it.

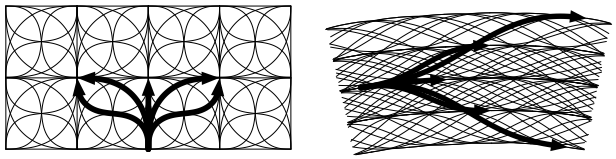


Fig. 1. Left: regular state lattice in an unstructured environment. The same five paths are rigidly transformed to create a graph of kinematically feasible actions. Right: state lattice conformed to a structured environment. Each path must be optimized to fit its endpoints.

We demonstrate that our planner is able to, for example, decide whether to distance-keep behind slower traffic or whether to change lanes to advance past it without specific behavioral instruction by planning within a unified optimization framework.

II. RELATED WORK

We adapt the state lattice of Pivtoraiko et al.[2] to a structured environment. Their state lattice is a search graph where vertices representing kinematic states of the robot are connected by edges representing paths that satisfy the kinematic constraints of the robot. The vertices are placed in a regular pattern such that the same paths, albeit rigidly translated and rotated, can be used to connect all vertices to form a graph dense enough that a feasible path to the goal is likely to exist as a sequence of edges in the graph, as Figure 1(Left) shows. This state lattice is suitable for unstructured environments where every global robot heading is a priori equally likely to be in the final solution. For a robot driving on a public road, as also noted in [9], the selection of feasible headings is not only highly constrained, but the a priori likelihood that they would be a part of the solution varies with location in the environment.

Therefore, the state lattice must be adapted to the environment such that only states that are a priori likely to be in the optimal path are represented (Figure 1(Right)). The authors in [9] show a method for constructing such a lattice for an urban road. Their contribution is a closed-form calculation to

M. McNaughton C. Urmson, and J. Dolan are with the Robotics Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA USA

J.-W. Lee is with General Motors Research and Development, Warren MI USA

This research is supported by General Motors Company.

satisfy the endpoint constraints on a state lattice conformed to a road, for forms of robot kinematics including typical cars. Compared to their work, we use the method of [6] to construct the lattice, and we propose a special augmentation of the lattice with time and velocity dimensions in order to plan in a dynamic environment. The authors of [9] do not provide an evaluation of the number of lattice edges that can be generated per second, so we are unable to compare our approach to constructing the lattice with theirs. However, as our experimental results show (Section V), the time taken by our approach to construct the lattice forms a small part of the overall time taken to search for an optimal plan through the lattice, when considering dynamic obstacles.

Ziegler and Stiller[10] propose an augmentation of a conformal state lattice with time and velocity dimensions in order to plan with moving traffic. Their approach required trajectories connecting lattice points to start and end at one of a fixed set of times T and velocities V. The use of fixed values in T and V makes it difficult to plan to merge between two vehicles traveling with velocities not contained in V. They used quintic splines with a point mass to connect endpoints, requiring additional effort to track the path using the vehicle kinematics. Compared to their work, we use a more efficient method of augmenting the state lattice with time and velocity dimensions, and we use realistic vehicle kinematics to construct the actions. They propose that energy cost terms useful for tuning vehicle behavior, such as squared jerk of the path, are difficult to compose search heuristics for, and therefore propose an exhaustive, rather than heuristic search in order to find the optimal path. We make the same observation and likewise perform an exhaustive search in our planner.

Werling et al.[3] propose a planner that samples a large number of swerve actions that take the vehicle away from the lane center-line and then back towards it. We propose an approach similar in spirit, but search over a regular lattice structure that can express more complex maneuvers.

Lee and Litkouhi[7] propose a real-time path planning algorithm for smooth lane changing. They use a polynomial equation and find a closed-form solution with path continuity in the second degree. This approach provides a quick and smooth trajectory, however, it has difficulty in generating a path for multiple obstacle avoidance.

In summary, the contribution of this paper is a planner for an autonomous vehicle that can plot a course through structured road environments with dynamic obstacles, while reducing the modeling error inherent to hierarchically decomposed planners. We demonstrate real-time performance of the planner on a real robot.

III. Method

Our lattice is constructed around a lane center line defined as a sampled function $[\, x(s)\ y(s)\ \theta(s)\ \kappa(s)\,]$ of arc length s, also known as station. We define points $p(s,\ell)$ on the road at lateral offset ℓ, or latitude, from the center line as $p(s,\ell) = [x(s,\ell)\ y(s,\ell)\ \theta(s,\ell)\ \kappa(s,\ell)]$ where

$$\begin{array}{rcl} x(s,\ell) &=& x(s) + \ell\cos(\theta(s)) \\ y(s,\ell) &=& y(s) + \ell\sin(\theta(s)) \\ \theta(s,\ell) &=& \theta(s) \\ \kappa(s,\ell) &=& (\kappa(s)^{-1} + \ell)^{-1}. \end{array} \tag{1}$$

$\kappa(s)$ is the curvature of the path, such that $\theta(s,\ell) = \int_0^s \kappa(s,\ell)$. To construct our lattice we define a discrete grid (i,j) and a linear mapping $(s(i), \ell(j))$ from discrete grid points to station and latitude using a simple linear multiplication $s(i) = a_s i$, $\ell(j) = a_\ell + b_\ell j$, so that station is monotonic increasing starting from zero and moving right in the coordinate frame, and latitude may be positive (left) or negative (right) w.r.t. the center line.

We compute paths between vertices in our conformed lattice using the iterative numerical method of [6], also used by the Tartan Racing team[1][4] in the 2007 DARPA Urban Challenge. A path is defined as a cubic polynomial spiral, that is, the curvature of the path is a cubic polynomial function of arc length

$$\kappa(s) = a + bs + cs^2 + ds^3,$$

for which parameters can be found to define a path connecting any pair of endpoints (x, y, θ, κ). Following [5], we use the parameterization $\mathbf{p} = [p_0\ p_1\ p_2\ p_3\ s_f]$, so that

$$\kappa(s) = a(\mathbf{p}) + b(\mathbf{p})s + c(\mathbf{p})s^2 + d(\mathbf{p})s^3,$$

and s_f is the arc length of the curve between the boundary constraints. The coefficient functions $z(\mathbf{p})$ are selected so that

$$\begin{array}{rcl} \kappa(0) &=& p_0 \\ \kappa(s_f/3) &=& p_1 \\ \kappa(2s_f/3) &=& p_2 \\ \kappa(s_f) &=& p_3. \end{array}$$

The resulting polynomials are

$$\begin{array}{rcl} a(\mathbf{p}) &=& p_0 \\ b(\mathbf{p}) &=& -\frac{11p_0 - 18p_1 + 9p_2 - 2p_3}{2s_f} \\ c(\mathbf{p}) &=& \frac{9(2p_0 - 5p_1 + 4p_2 - p_3)}{2s_f^2} \\ d(\mathbf{p}) &=& -\frac{9(p_0 - 3p_1 + 3p_2 - p_3)}{2s_f^3}. \end{array}$$

Since $p_0 = \kappa_0$ is given, we only need to find $\mathbf{p} = [p_1\ p_2\ p_3\ s_f]$. Deriving the coefficients from the parameters this way rather than the simpler $p_0 + p_1 s + p_2 s^2 + p_3 s^3$ ensures that the elements of $\mathbf{p}$ (except for s_f) are close in magnitude, adding numerical stability to the optimization. We use a simple bicycle model kinematics where

$$\begin{array}{rcl} \mathrm{d}x_{\mathbf{p}}/\mathrm{d}s &=& \cos\left[\theta_{\mathbf{p}}(s)\right] \\ \mathrm{d}y_{\mathbf{p}}/\mathrm{d}s &=& \sin\left[\theta_{\mathbf{p}}(s)\right] \\ \mathrm{d}\theta_{\mathbf{p}}/\mathrm{d}s &=& \kappa_{\mathbf{p}}(s), \end{array}$$

indicating the dependence on the parameters by the subscript $\mathbf{p}$, so that the state of the system after traveling a distance s

is

$$\begin{aligned} x_{\mathbf{p}}(s) &= \int_0^s \cos\left[\theta_{\mathbf{p}}(s)\right] \mathrm{d}s \\ y_{\mathbf{p}}(s) &= \int_0^s \sin\left[\theta_{\mathbf{p}}(s)\right] \mathrm{d}s \\ \theta_{\mathbf{p}}(s) &= a(\mathbf{p})s + b(\mathbf{p})s^2/2 + c(\mathbf{p})s^3/3 + d(\mathbf{p})s^4/4 \\ \kappa_{\mathbf{p}}(s) &= a(\mathbf{p}) + b(\mathbf{p})s + c(\mathbf{p})s^2 + d(\mathbf{p})s^3. \end{aligned} \tag{2}$$

The endpoint of the path is $[x_{\mathbf{p}}(s_f)\ y_{\mathbf{p}}(s_f)\ \theta_{\mathbf{p}}(s_f)\ \kappa_{\mathbf{p}}(s_f)]$, and we wish to find parameters that make this equal to $\mathbf{x_{des}}$. Both $\theta_{\mathbf{p}}(s)$ and $\kappa_{\mathbf{p}}(s)$ can be evaluated in closed form, but the expressions for $x_{\mathbf{p}}(s)$ and $y_{\mathbf{p}}(s)$ are Fresnel integrals and must be integrated numerically. We use Simpson's method to evaluate the integrals.

We use a shooting method to solve for the parameters satisfying the endpoint constraints. We evaluate the Jacobian of the endpoint state vector $\mathbf{x}_{\mathbf{p}}(s_f) = [x_{\mathbf{p}}(s_f)\ y_{\mathbf{p}}(s_f)\ \theta_{\mathbf{p}}(s_f)\kappa_{\mathbf{p}}(s_f)]$ with respect to the parameter vector $\mathbf{p} = [p_1\ p_2\ p_3\ s_f]$,

$$\mathbf{J}_{\mathbf{p}}(\mathbf{x}_p(s_f)) = \left[\frac{\mathrm{d}\mathbf{x}_{\mathbf{p}i}(s_f)}{\mathrm{d}\mathbf{p}_j}\right],$$

and follow the gradient to minimize the error between $\mathbf{x}$ and $\mathbf{x_{des}}$. Other authors computed the Jacobian numerically using central differencing[5]. By using small integration steps, this method can handle inequality constraints such as saturation of the steering angle at points along the path, but we assume that in highway driving scenarios such limits are not reached. Therefore, since we assume no inequality constraints need be enforced during the integration, we can use fewer integration steps and calculate the Jacobian using a symbolic differentiation with respect to the parameters of the expression used to calculate the endpoint.

With the Jacobian we use Newton's method for root-finding to generate a sequence of estimates $\{\mathbf{p}_i\}$ for $\mathbf{p}$ which causes $\mathbf{x}$ to equal $\mathbf{x_{des}}$

$$\begin{aligned} \Delta\mathbf{x} &= (\mathbf{x_{des}} - \mathbf{x}_{\mathbf{p}_i}(s_f)) \\ \Delta\mathbf{p} &= \mathbf{J}_{\mathbf{p}_i}(\mathbf{x}_{\mathbf{p}_i}(s_f))^{-1}\Delta\mathbf{x} \\ \mathbf{p}_{i+1} &= \mathbf{p}_i + \Delta\mathbf{p} \end{aligned}$$

This iteration proceeds until $\Delta\mathbf{x}$ is deemed sufficiently small, or a maximum number of iterations is reached.

The initial guess $\mathbf{p_0}$ is obtained from either a precomputed look-up table or the $\mathbf{p}$ vector derived for a similarly situated pair of endpoints from the previous planning cycle. Our approach is fast in practice despite the apparent computational expense compared to e.g. the quintic splines used by other researchers[3], since only a couple of iterations are typically required to re-converge. In addition, our paths are consistent with the kinematic constraints of the robot, sparing the complexity of verifying whether the generated path can actually be followed. We also have greater flexibility in selecting end point constraints, such as ending paths with an arbitrary curvature.

A. Paths and Trajectories

We define a path τ_p as a continuous curve through the state space $[x\ y\ \theta\ \kappa]$ defined by a start state $\mathbf{x_0} = [x_0\ y_0\ \theta_0\ \kappa_0]$ and a set of cubic polynomial spiral parameters $\mathbf{p} = [p_1\ p_2\ p_3\ s_f]$, using Equation 2 to drive to an end state $\mathbf{x_f} = [x_f\ y_f\ \theta_f\ \kappa_f]$. We write $\tau_p(s)$ for the point at the arc length s along the path from $\mathbf{x_0}$. The set of all $\{\tau_p\}$ therefore contains all possible paths that may be derived by the method of the previous section, irrespective of the time or velocity at which they are traversed. Since our paths are defined as functions of curvature w.r.t. arc length, they can be readily followed by the vehicle at any speed, subject to constraints on curvature rate with respect to time, lateral acceleration, and the like.

For each path τ_p we can define a family of trajectories τ_r by the addition of a time and velocity component to obtain a curve through the six-dimensional state space $\hat{\mathbf{x}} = [x\ y\ \theta\ \kappa\ t\ v]$. To obtain a trajectory τ_r from a path τ_p, we use a starting time and velocity $[t_0\ v_0]$ and apply a constant acceleration a over the course of the path. Then

$$\begin{aligned} \tau_r(s) = [\ & x_{\mathbf{p}}(s) & y_{\mathbf{p}}(s) & \theta_{\mathbf{p}}(s) \\ & \kappa_{\mathbf{p}}(s) & t_0 + t(s, v_0, a) & v(s, v_0, a)\ ], \end{aligned}$$

where for constant acceleration a, $v(s, v_0, a)$ is the velocity obtained at the distance s along the trajectory:

$$v(s, v_0, a) = \begin{cases} \sqrt{v_0^2 + 2as} & \text{if } v_0^2 + 2as \geq 0 \\ \text{undefined} & \text{otherwise,} \end{cases}$$

and $t(s, v_0, a)$ is the time taken to traverse the distance s:

$$t(s, v_0, a) = \begin{cases} s/v_0 & \text{if } a = 0 \\ \dfrac{v(s, v_0, a) - v_0}{a} & \text{if } a \neq 0, v(s, v_0, a) \in \mathbb{R} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The set τ_r therefore contains all possible trajectories that may be traversed by the vehicle using our method. Again, $\tau_r(s)$ denotes the point at an arc length s along the trajectory.

We define cost functions for paths and for trajectories. The cost function $c_p(\tau_p)$ for paths is composed of terms that do not require the time and velocity of the states along the path to be evaluated, such as those for avoiding static obstacles. The cost function $c_r(\tau_r)$ is composed of c_p plus terms involving time, such as those penalizing lateral acceleration. By sampling points $\mathbf{x}$ along a path, and then sampling the time and velocity components at the same points to form sample points $\hat{\mathbf{x}}$ along the trajectories defined by that path, we are able to inexpensively evaluate many trajectories for each path.

In the next section we describe how trajectories are arranged into a lattice, and how the lattice is searched.

B. Spatiotemporal Lattice

Since we are searching in a dynamic environment, we must consider both time and space. The state lattice is a proven method for systematically searching through static environments; however, naively adding time and velocity dimensions can cause an unacceptable blowup in the size of the search space. Vertices in the state lattice for static spaces are normally defined by the vehicle state vector $[x\ y\ \theta\ \kappa]$, such that the vehicle traverses paths that satisfy starting

and ending boundary constraints coincident with the lattice vertices. The values of all state variables associated with the lattice vertices are fully specified before the edges are evaluated. If this approach is taken with the time-enhanced state vector $[x\ y\ \theta\ \kappa\ t\ v]$, the number of lattice vertices and edges would be too large to be tractable for driving applications.

The size of the velocity dimension for a freeway driving application would be at least the number of distinct velocity values taken on as it passes over each discrete station value used in the lattice. For a vehicle with a reasonable acceleration time of 10 s to reach 100 km/h from a stop, driving on a lattice with 5 m longitudinal spacing, the number of discrete velocity values needed for the lattice would be equal to the number of meters travelled, divided by the longitudinal spacing, while accelerating from a standing start up to full speed. Assuming a constant acceleration so that $v = at$, $p = vt = \frac{1}{2}at^2$, and with final time $t_f = 10$s to reach 100 km/h

$$a = v(t_f)/t_f \quad (3)$$

$$= \frac{100\text{km/h}}{10\text{s}} = 2.78\text{m/s}^2, \quad (4)$$

$$p(t_f) = \left(\frac{1}{2}\right)\left(2.78\frac{m}{s^2}\right)(10s)^2 = 139m, \quad (5)$$

meaning that about $139/5 \approx 28$ distinct values would be needed for a velocity dimension in the state lattice in the naïve case. The use of more gentle accelerations would only increase this number. The time dimension would multiply the size of the lattice by a similar factor. For a 100-meter planning horizon with a reasonable 40-centimeter lateral discretization on a 7.2 meter-wide road, the total size of the lattice would be on the order of 20 longitudinal increments $\times$ 20 lateral increments $\times$ 30 velocity increments $\times$ 30 time increments $\approx$ 360 000 vertices. Assuming a modest 7 paths to shift latitude with 5 accelerations each, we get $\approx$ 12 million trajectory edges to evaluate at each planning cycle. This space, while not intractable, is too large for real-time planning.

To address the graph-size problem, we first observe that in any graph search through a continuum, one can either specify the graph before edge evaluations begin, or specify it as the search proceeds. In our approach, we partially construct the graph before we search, and specify the rest as the search proceeds. In particular, we delay the assignment of values to the time and velocity components of the lattice vertices. Rather, we assign to each vertex a range of times and velocities $[t_i, t_{i+1}) \times [v_j, v_{j+1})$ which the state vector may take on, with the actual values being assigned during the evaluation of trajectories (Figure 2). Only a few such divisions are necessary, compared to the tens required should each vertex be assigned a point value as in Equation 3. With each vertex is associated a state vector $(x, y, a, \theta, \kappa, [t_i, t_{i+1}), [v_j, v_{j+1}))$, where a is the index of the incoming acceleration profile, and is included in the lattice for reasons discussed later. As in Equation 1, the static state variables of the lattice are specified uniquely by the road coordinates (s, ℓ), so the actual dimensionality of the state vector can be simplified to the five-dimensional $(s, \ell, a, [t_i, t_{i+1}), [v_j, v_{j+1}))$.

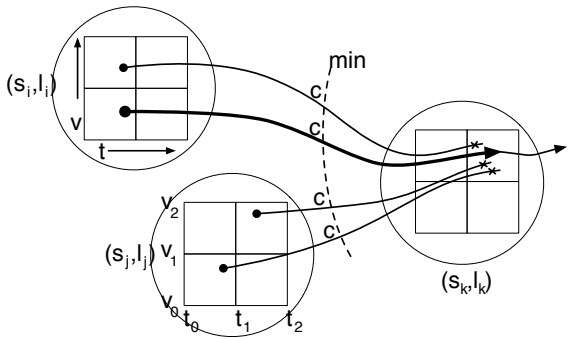


Fig. 2. Multiple trajectories converge into a single lattice vertex. Each vertex in the lattice is identified with a quadrant, corresponding to a pose on the road and a range of times and velocities. In each vertex, the ending time and velocity value from the incoming trajectory with the minimum cost is assigned to the lattice vertex's (t, v) coordinates. These values are used in turn for outgoing trajectories from the vertex.

We refer to the starting state as $\hat{\mathbf{x}}_\mathbf{0} = [x_0, y_0, \theta_0, \kappa_0, t_0, v_0]$ and we can assume $t_0 = 0$ without loss of generality. Since the starting state $\hat{\mathbf{x}}_\mathbf{0}$ is not necessarily coincident with a vertex on the regular lattice structure, an additional source vertex n_0 is constructed with the state vector equal to $[x_0, y_0, \theta_0, \kappa_0, 0, v_0]$. We choose a subset S of the states on the regular lattice S close to n_0, and plot trajectories $\{\tau_r\}$ s.t. $\tau_r(0) = \hat{\mathbf{x}}_\mathbf{0}$ and $\tau_r(s_f) \in S$, with one trajectory for each (endpoint, acceleration) pair. For each of the trajectory endpoints $\hat{\mathbf{x}}_\mathbf{f}$, we identify the associated lattice vertex $n = (x, y, \theta, \kappa, a, [t_i, t_{i+1}), [v_j, v_{j+1}))$ such that $t(\hat{\mathbf{x}}_\mathbf{f}) \in [t_i, t_{i+1})$ and $v(\hat{\mathbf{x}}_\mathbf{f}) \in [v_j, v_{j+1})$. For each vertex in the search a cost is maintained representing the minimum known cost to reach that vertex from the source vertex. At the start of the search they are initialized to $\hat{c}(n) \leftarrow \infty$. Since only one of these trajectories will end at each vertex, we can assign the cost $\hat{c}(n) = c(\tau_r)$ and specify the time and velocity values $t(n) \leftarrow t(\hat{\mathbf{x}}_\mathbf{f})$, $v(n) \leftarrow v(\hat{\mathbf{x}}_\mathbf{f})$ for the vertex. These will be the values used for trajectories outgoing from this vertex later in the search. Subsequently, the search proceeds as a dynamic program. Since the vehicle is constrained to move forward, that is, strictly increasing in station, we can evaluate edges in the graph starting at the same station simultaneously, and groups of these can be evaluated in increasing order of station. Figure 3 outlines the algorithm. In summary, from each lattice vertex that has been assigned a finite cost and a definite time and velocity, we evaluate a set of outgoing trajectories representing the product of possible paths to other (x, y, θ, κ) states on the road and a range of possible accelerations to take on the way.

C. Why include acceleration in the state space?

Consider two trajectories τ_1, τ_2 starting at the same lattice vertex and using the same underlying path, but with differing accelerations. If their final states $\hat{\mathbf{x}}_\mathbf{1}, \hat{\mathbf{x}}_\mathbf{2}$ land in the same $[t_i, t_{i+1})$ and $[v_j, v_{j+1})$(Figure 4), and acceleration were not included as a dimension in the lattice, then only one of the

for each station $s = 1 \cdots \#s$
  for each vertex n at station s
    if $\hat{c}(n) \neq \infty$
      Form the vector
        $\hat{\mathbf{x}}_\mathbf{n} = [x(n), y(n), \theta(n), \kappa(n), t(n), v(n)]$
      for each acceleration a and path parameter $\mathbf{p}$
        Form the trajectory $\tau_r(\mathbf{p}, t(n), v(n), a)$
        Evaluate the trajectory cost $c(\tau_r)$
        Identify the lattice vertex $n_f(\tau_r(s_f))$ —
          —at the end of the trajectory
        if $\hat{c}(n) + c(\tau_r) < \hat{c}(n_f)$
          $\hat{c}(n) \leftarrow \hat{c}(n) + c(\tau_r)$
          $t(n_1) \leftarrow t(\tau_r(s_f))$
          $v(n_1) \leftarrow v(\tau_r(s_f))$
          $\text{incoming}(n_1) \leftarrow n$ // *backtrace info*
        end if // $\hat{c}(n)$
      end for // a
    end if // $\neq \infty$
  end for // *vertex*
end for // *station*

Fig. 3. The dynamic programming search algorithm for the spatiotemporal lattice.

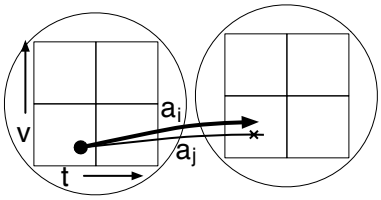


Fig. 4. If acceleration is not a dimension in the lattice, multiple trajectories proceeding from the same starting lattice vertex and differing only in their acceleration profiles may interfere by ending in the same vertex.

accelerations would be represented in the result. By including acceleration in the lattice we found that we could represent a greater diversity of trajectories, and importantly, final trajectories with more consistent acceleration profiles, than if we simply refined the discretization of the t and v dimensions by the amount required to overcome this problem.

D. Cost function

Each trajectory is assigned a cost representing the relative desirability of having the vehicle follow that trajectory. The cost function includes terms to avoid obstacles, as well as physical limitations on the vehicle's performance, such as the rate of change of path curvature with respect to time. We also use terms representing behavioral preferences, for example to minimize the magnitude of lateral acceleration undergone during a maneuver, which promotes passenger comfort. We can also use cost terms to represent behavioral preferences, such as a desired lane to drive in. The cost function is evaluated in two parts. Since multiple trajectories use the same underlying path, the cost function terms depending only on (x, y, θ, κ) are evaluated before the various trajectories using the path. Then the terms depending on a, t, v are evaluated with each trajectory.

E. Picking the best final state

The best plan is represented by the continuous sequence of trajectories through the lattice to the final vertex with the smallest cost. However, freeway driving on the time scale of seconds is a continuous process with no final goal, so it is not obvious how to specify the final, or sink vertex of the lattice. It is not sufficient to simply search to the furthest station of the lattice, since a road blockage could prevent any path from reaching the end. Nor can we simply search to the furthest time value that appears in the lattice, since that implies driving as slowly as possible. Safety requires us to select a minimum planning horizon t_h in terms of time, to ensure that the vehicle is following a plan that is guaranteed to be feasible (given reasonable behavior of other traffic) within the range limits of the sensors while the vehicle is traveling at maximum speed, and the time required for a controlled emergency stop. We balance our desire for the vehicle to make rapid forward progress at a reasonable cost with the recognition that sometimes road conditions make this impossible. We do this by selecting the final vertex n_f to be the one that minimizes

$$\arg\min_{n_f} \hat{c}(n_f) - k_s(s(n_f)) + k_t(t(n_f)), \tag{6}$$

a weighted sum of the trajectory cost to reach the vertex n_f, with a bonus for driving further and a penalty for taking extra time. We then trace backward through the lattice from n_f to the start state, reconstructing the trajectory. Weights are tuned manually to achieve the desired balance of forward progress against low trajectory cost.

F. World Representation

A trajectory is evaluated by sampling (x, y, t) points along its length and calculating the cost terms at each sample. Intersections with static obstacles are computed by discretizing the (x, y) space and rendering the obstacles into a look-up table. Intersections with moving obstacles are similarly computed by discretizing the three-dimensional (x, y, t) space and rendering the moving obstacles into the table at their predicted future locations.

The vehicle will always remain roughly parallel to the road, even when changing lanes. Therefore we can expand obstacles in a rectangular pattern complementary to the shape of the vehicle, and perform a single look-up at each sample. A more complex convolution is not necessary.

IV. GPU Acceleration

Contemporary graphics processing units (GPUs) such as those sold by Nvidia are well suited to search on a state lattice such as the one we have described in previous sections. Briefly, GPUs run thousands of threads at once, each executing a simple C/C++ program that uses a few dozen registers at once. The speed at which an individual thread is run is sacrificed in order to increase overall throughput. All threads must run the same program, and groups of 32

threads must actually run at the same instruction counter, or else sacrifice throughput by running in serial, leaving some processing elements idle. Memory operations performed by groups of threads are much faster when they affect contiguous memory locations. These performance characteristics are well suited to dynamic programming on a regularly structured graph, where threads each evaluate one edge, using the same algorithm. We now argue that an exhaustive search is necessary anyway for a driving application.

Typical state lattice planners for static domains are implemented using a best-first search over the graph such as A* or D*-lite. These algorithms are appropriate when a reasonable heuristic estimate can be obtained for the cost-to-go to reach the goal from any vertex. In highway driving, however, an effective heuristic combining the many kinds of cost terms, along with the selection of the final state, is unknown. The worst case of the search is likely to occur - that all vertices in the graph would have to be expanded in order to find the optimal path. Since driving is a real-time application where the vehicle cannot stop and deliberate on its next action, we are concerned with minimizing the worst-case time to find a solution. Although some driving applications have used anytime algorithms[8] to obtain a suboptimal path that can be refined as it is followed, even these algorithms must find a feasible path before it can be refined to an optimal path, and it is still not guaranteed that all vertices would not be expanded before such a path is found. Therefore, since all vertices must be examined in the worst case, we must search the entire graph, which indicates a simple dynamic programming algorithm.

V. Experimental Results

We implemented our planner on an Nvidia GeForce GTX 260 installed with an Intel Core 2 Quad processor. The planner was run on Boss, our autonomous SUV. To compare the performance improvement gained by using the GPU, the planner is also implemented to run on a normal CPU and run in simulation on a single core. The planner runs on a 10 Hz update cycle, providing curvature, velocity, and acceleration commands to a controller which converts them into steering and throttle actuation commands to the vehicle. Since Boss cannot be run on public roads, nor at speeds greater than 30 mph, our tests were limited to small private roads with low speeds and minimal traffic. The values we assigned to parameters controlling the size of the search lattice are given in Table I. We found that these values could generate acceptable plans within the allowed planning latency. Table II displays the time taken in each phase of the search by each of the GPU and CPU. The GPU provides a considerable speedup overall even accounting for the fact that only one core is used in the CPU implementation, although the GPU is much faster at certain tasks. Both implementations are reasonably well-optimized, so this comparison is strongly suggestive of the relative merits of the platforms for our algorithm.

We tested several scenarios demonstrating the range of abilities of the planner. Figure 5 demonstrates that the planner can choose an appropriate speed to round a tight corner. The planner penalizes trajectories with higher lateral accelerations up to a hard limit of 0.3 g. At the first sample in the figure, the robot is traveling at 33 km/h. It slows to 13 km/h with a lateral acceleration of 0.11 g to round the corner at the tenth sample and accelerates back to 32 km/h at the last sample.

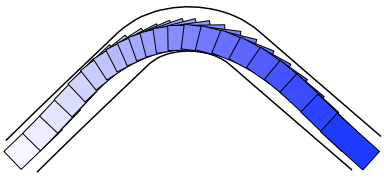
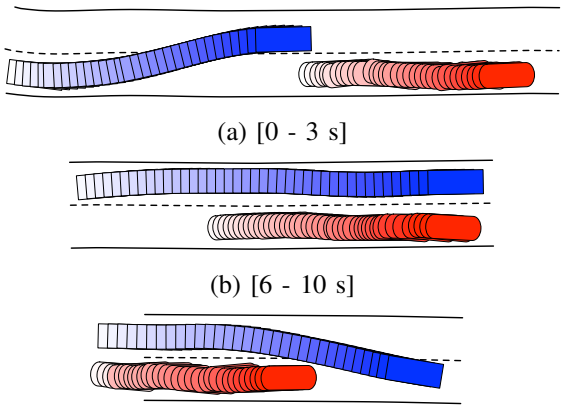


Fig. 5. With a constraint that penalizes lateral accelerations, the planner automatically slows the vehicle in anticipation of a tight turn. The distance between successive samples shows the deceleration coming into the turn and acceleration beginning near the apex of the turn. Samples are spaced at 300 ms intervals.



(a) [0 - 3 s]

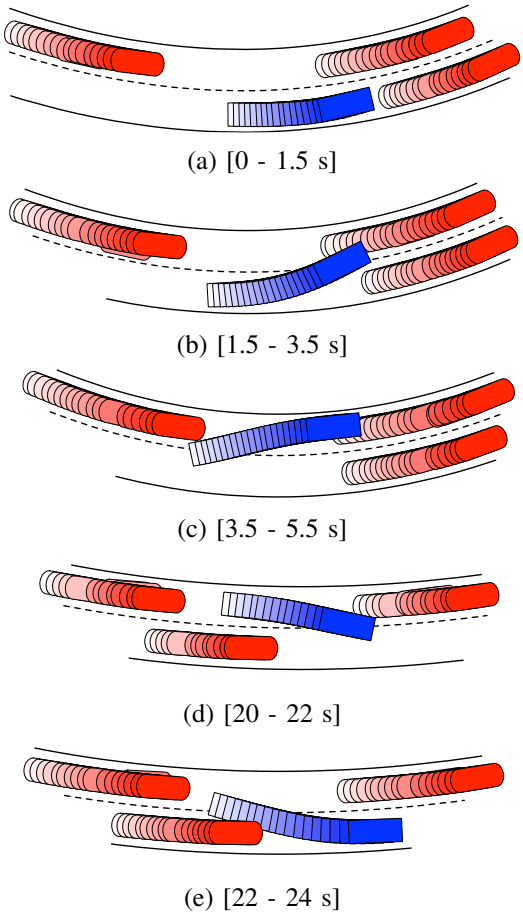
(b) [6 - 10 s]

(c) [10 - 14 s]

Fig. 6. In a test conducted at 30 km/h, the robot passes a slow-moving human-driven vehicle in the right lane by (a) moving into the left lane, then (b) remaining there until it passes the other vehicle, and (c) finally merging in front of it. Samples are spaced at 100 ms intervals.

Figure 6 demonstrates a passing scenario executed on the robot. The right lane is given a slightly lower cost to traverse than the left lane, encouraging the robot to stay in the right lane. The robot prefers to travel at approximately 35 km/h, but a human-driven vehicle traveling at 20 km/h is ahead of it in the same lane. Since the bonus awarded for driving further by the final cost coefficient k_s (Equation 6) exceeds the slight penalty for driving in the left lane, the planner selects a trajectory that moves into the left lane, passes the slower vehicle, and merges back in front of it.

Figure 7 shows a more complex version of the behavior in Figure 6, where the planner must wait for a space to open before merging into the passing lane. Due to practical constraints, we demonstrated this behavior in simulation. Part (a) of the figure shows that the exploration of times and velocities in the search allows a distance-keeping behavior to emerge. We create a region of high cost behind obstacle vehicles, so that the optimal path keeps the vehicle just out of the high-cost regions.



(a) [0 - 1.5 s]

(b) [1.5 - 3.5 s]

(c) [3.5 - 5.5 s]

(d) [20 - 22 s]

(e) [22 - 24 s]

Fig. 7. In a simulation, the planner (a) distance-keeps behind a slow-moving vehicle until a gap between faster vehicles in the next lane opens up alongside. It (b) and (c) merges into the gap and remains there until it passes the slower-moving vehicle, whereupon it (d) and (e) merges back in front of it. In this scenario, the robot's desired speed is faster than all other vehicles. Samples are spaced at 100 ms intervals.

Parameter	Value
Station increments	7
Latitude increments	19
Accelerations	7
Outgoing paths	7
Time discretizations	3
Velocity discretizations	3
Total trajectories evaluated per cycle	$\approx 400\ 000$

TABLE I

PARAMETERS OF THE LATTICE USED IN THE EXPERIMENTS

Search Phase	GPU Time	CPU Time	Speedup
Plan trajectories from source pose onto lattice	2 ms	12 ms	6
Update all paths in lattice	0.1 ms	4 ms	40
Plan all trajectories coming out of a single station	2 ms	42 ms	21
Whole planning cycle	45 ms	650 ms	15

TABLE II

TIME TAKEN ON THE CPU AND GPU FOR STAGES OF THE PLANNING CYCLE

VI. CONCLUSIONS

Our planner is able to generate reasonable plans in real-time for a variety of traffic situations. The planner can use a GPU to exploit the parallel structure of the search space and significantly reduce the planning latency.

The cost function plays an important role in shaping the final behavior, and more work is necessary to develop cost functions for more complex behaviors.

An obvious next step in the development of the planner is to investigate more sophisticated acceleration profiles while maintaining execution efficiency. Using a constant acceleration over the course of the path can lead to execution errors, since vehicles typically cannot change acceleration abruptly. At low speeds, constructing paths using a cubic polynomial spiral while staying strictly within curvature rate overly limits the maneuverability of the vehicle. A refinement to the types of actions considered may be necessary at low speeds.

We have not yet looked at reachability in the lattice. If there are large portions of the lattice which are never part of the solution, they may be trimmed in order to increase trajectory density in other parts.

REFERENCES

[1] D. Ferguson et al. Motion planning in urban environments. *Journal of Field Robotics*, 25(11-12):939–960, 2008.

[2] M. Pivtoraiko et al. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 26(1):308–333, March 2009.

[3] M. Werling et al. Optimal trajectories for dynamic street scenarios in a frenét frame. In *Proceedings of ICRA*, pages 987–993, 2010.

[4] T. Howard et al. State space sampling of feasible motions for high-performance mobile robot navigation in complex environments. *Journal of Field Robotics*, 25(1):325–345, June 2008.

[5] T. Howard. *Adaptive Model-Predictive Motion Planning for Navigation in Complex Environments*. PhD thesis, Carnegie Mellon University, 2009.

[6] A. Kelly and B. Nagy. Reactive nonholonomic trajectory generation via parametric optimal control. *The International Journal of Robotics Research*, 22(1):583 – 601, July 2003.

[7] J.-W. Lee and B. Litkouhi. Control and validation of automated lane centering and changing maneuver. In *ASME Dynamic Systems and Control Conference*, 2009.

[8] M. Likhachev and D. Ferguson. Planning long dynamically-feasible maneuvers for autonomous vehicles. In *Proceedings of Robotics: Science and Systems IV*, Zurich, Switzerland, June 2008.

[9] M. Rufli and R. Siegwart. On the design of deformable input-/state-lattice graphs. In *Proceedings of ICRA*, Anchorage, Alaska, 2010.

[10] J. Ziegler and C. Stiller. Spatiotemporal state lattices for fast trajectory planning in dynamic on-road driving scenarios. In *Proceedings of ICRA*, 2009.