Space-Time Graph Planner for Unsignalized Intersections with CAVs

Caner Mutlu, Ionut Cardei, and Mihaela Cardei

Florida Atlantic University, Boca Raton FL 33431, USA
Department of Electrical Engineering and Computer Science
{cmutlu,icardei,mcardei}@fau.edu
http://www.eecs.fau.edu

Abstract. Emerging autonomous intersection management systems control the entry order and trajectory for connected and autonomous vehicles ready to traverse a road intersection. They aim to compute trajectories that are safe and optimal in order to reduce congestion, environmental impact, and to cut travel time. We propose a novel approach for computing the fastest waypoint trajectory using search in a discretized space-time graph that produces collision-free paths with variable vehicle speeds complying with traffic rules and vehicle dynamics constraints. The resulting trajectories allow high levels of intersection sharing, high evacuation rate, with a low algorithm runtime even with large scenarios with 1200 vehicles (5.5 seconds on a laptop).

Keywords: space-time graph · intersection management · autonomous cars

1 Introduction

There are about 15 million road intersections in the continental US alone, and 44% of the road incidents occur at intersections [1]. Time waste and fuel consumption [2] at intersections also have negative societal impact. In contrast to traditional signaled intersections, signal-free intersections controlled by Autonomous Intersection Management Systems (AIMS) do not employ semaphores and provide conflict-free intersection transit for Connected Autonomous Vehicles (CAVs). CAVs talk to AIMS on a network, such as cellular, vehicle-to-vehicle (V2V), vehicle-to-instructure (V2I), and others [3].

A CAV contacts the nearby AIMS and sends an admission request with its predicted arrival time at the intersection entry and its intended intersection exit lane. The AIMS considers new requests, the trajectories of CAVs already in transit, and computes control commands for each new vehicle. The computation can be centralized in the infrastructure or it can be distributed among coordinating CAVs. Possible AIMS goals include minimizing exit time for each vehicle in a fair first-in/first-out (FIFO) way or maximizing the global exit flow, across all vehicles.

Management of CAVs in an unsignaled AIMS environment can be categorized under two broad problem classes, “vehicular scheduling” and “vehicular control” [4]. Both can be addressed separately, but a true optimal trajectory would require solving both concurrently. A vehicle control system formulates a solution describing commands for vehicle actuators, such as steering, throttle, braking. Some examples use model predictive control [5] or optimal control [6]. On the other hand, a vehicle scheduler is a trajectory planner that gives a sequence of waypoints (location + time) that must be traversed by the CAV. At a minimum, it just provides an entry sequence to the intersection, enough for vehicles to avoid collisions if they stick to their desired lane. Reservation-based systems [6, 7] schedule vehicles to leave the intersection in FIFO order, underutilizing shared intersection space. Solutions that solve a discrete optimization problem [8] are limited by the exponential growth of the search space - unfeasible for realistic large scenarios with hundreds of vehicles. Graph search methods model the intersection as a graph in 2D. Depth first spanning tree (DFST) methods [4, 9] do conflict analysis and determine an entry order that increases parallel access to the intersection.

In this paper we are concerned with the problem of finding the shortest collision-free space-time trajectory through an intersection, constrained by traffic rules and vehicle limitations. Such a trajectory can then be passed to the CAV's navigation unit to generate actuator commands.

We propose a solution – the Fastest Trajectory Planner algorithm – that a) models the intersection road map as a discretized graph G_u; b) expands G_u's vertices and edges to the time dimension into a space-time graph G_t so that a G_u vertex or an edge used at a particular time corresponds to vertices and edges in G_t that will be removed from G_t for subsequent vehicles; c) finds the fastest variable-speed trajectory complying to constraints using a shortest path algorithm in the space-time graph. Our algorithm was inspired by our earlier work on drone traffic management [10–13], with addition of the variable speed capability and vehicle dynamic constraints. The algorithm has a low runtime complexity and scales well: scenarios with 1200 vehicles at a 4-lane 4-way intersection are solved in about 5.5 seconds on a typical laptop, with code in Python.

This paper continues with related work in Section 2, the problem statement in Section 3, the proposed algorithm in Section 4.4, a performance evaluation in Section 5, and conclusions (Section 6).

2 Related Work

Papers [4, 14] study global optimality for vehicular scheduling problems in an AIMS. Their method models vehicles with vertices and they build a Conflict Directed Graph where edges map from pair-wise path conflicts. An Improved Depth First Search Spanning Tree is used to design a conflict-free passing order through the intersection. A second algorithm uses a complementary Coexisting Undirected Graph built from non-conflicting vehicle pairs to compute the Minimum Clique Cover. That gives an optimal passing order with the minimum evacuation time.

The conflict-duration approach in [15] builds a Gantt-chart inspired conflict-duration diagram. Its axes are conflict locations and timing stamps. The conflict-duration diagram registers double or triple conflicts between vehicles. By considering the physical size (L x W) of each vehicle, the total duration where a physical conflict persists between two or three vehicles is represented on the conflict-duration diagram as overlapping time duration at a particular conflict point. By removal of the overlapping time region, through rescheduling speed profile of one or more vehicle(s), a conflict can be prevented between any pair of vehicles.

In our prior work on drone traffic management, we developed the concept of shortest path search in a space-time graph for vehicle trajectory planning. We initially formulated the point-to-point trajectory planner for drone package delivery in [10] using multi-source/multi-destination BFS on the space-time graph. The planner computes shortest space-time paths with edges traversed in one time unit and no constraints on vehicle dynamics. We improved that approach in [12] with a batch scheduling method that has a lower complexity and better results.We addressed in [11] the problem of energy-constrained drone package delivery with multiple warehouses and customers using a multi-source A* algorithm running on the space-time graph. More recently, [13] presents a multi-source/multi-destination search algorithm for the fastest trajectory between two disjoint groups of vertices in the space-time graph. This is suitable for drone planning when the operator has multiple drones available stationed through the network and has to deliver packages to multiple customers.

Our contribution in this paper differs from prior work with space-time graphs by complying to vehicle dynamic constraints and by allowing multiple possible times for space-time edge traversal, necessary for supporting variable average edge velocities. The collision constraints and resource sharing rules are different from drone scenarios.

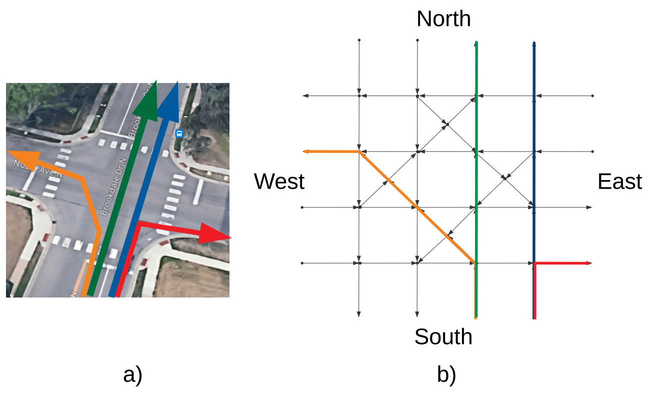


Fig. 1: (a) a 4-lane, 4-way intersection. (b) the graph for the intersection traffic road network. Paths possible from the two South entry vertices are highlighted with different colors.

3 Problem Statement

Fig. 1a illustrates a typical 4-lane, 4-way intersection, with legal trajectories highlighted. Vehicles moving in perpendicular directions have paths that intersect at shared points. The shared space forces vehicles to serialize their passage in order to avoid collisions. An AIMS runs the algorithms presented in Section 4.4. It uses the graph representation for the road network described below.

We model a road map as a directed graph with tuple $G = (V, E, pos)$, with edge set E and vertex set V, as in Fig. 1b. A segment corresponds to a directed edge $(u, v) \in E$.

If a road segment (u, v) is bidirectional, then $(v, u) \in E$, too. Function $pos : V \to \mathbb{R}^2$ defines the position of a vertex.

A collision between two vehicles occurs when their trajectories overlap in space and time. Vehicles can safely occupy the same space at different times sufficiently separated.

A vehicle's admission request is as an object $Request(src, dst, t_s, T_{max})$, where src, $dst \in V$ are the source and destination vertices, $t_s \in \mathbb{R}_{\geq 0}$ is the vehicle arrival time at the intersection, and $T_{max} \in \mathbb{R}^+$ is the maximum allowed trajectory duration. $T_{max} = \infty$ if a vehicle may take indefinitely to exit the intersection. The intersection manager accepts requests over a finite time interval $[0, t_{end}]$.

Vehicle movement on the road is limited by the maximum legal speed s_{max}, maximum acceleration a^+_{max}, maximum deceleration a^-_{max}, and vehicle length, packaged all in an object $Cons(s_{max}, a^+_{max}, a^-_{max}, L)$.

The computed trajectory between the $req.src$ and $req.dst$ is defined by a list of space-time waypoints that the vehicle must reach: $Tr(times, positions, velocities)$ indicating the time at each waypoint in $times = (t_0, t_1, ..., t_{m-1})$, the position of each waypoint $positions = (p_0, p_1, ..., p_{m-1})$, and a velocity vector for each waypoint.

A Tr object for which a solution cannot be found has no waypoints: $Tr((), (), ())$, where $()$ is the empty sequence. Otherwise, that is a *valid* trajectory.

We define the problem of finding trajectories for vehicles on a traffic map as follows:

Problem Definition *Given a road network graph $G = (V, E, pos)$, vehicle constraints $Cons$, and a list of vehicle admission objects $(Request)_i$ over a time interval $[0, t_{end}]$, the* **Fastest Trajectory Planning problem** *is finding a trajectory with m_i waypoints for each vehicle i through the road network that has the earliest arrival time t_{m_i-1}, subject to these conditions:*

1. *vehicles move on edges in E,*
2. *there are no collisions between any two vehicles on the road network,*
3. *vehicle constraints as defined by $Cons$ are satisfied at all times.*

The problem objective is locally greedy. An algorithm that globally minimizes the maximum delay is NP-complete because of the combinatorial explosion of the number of ways in which vehicle moves can be sequenced over time edges and space-time edges.

Performance Metrics.

The intersection trajectory planner accepts a sequence of N requests $reqs$ and produces a sequence of N Tr objects, from which n are valid: $trj_i = Tr(times_i, positions_i, velocities_i)$, $i = 0, ..., N-1$, and with attribute $times_i = (t_0, t_1, ..., t_{m-1})_i$, for m_i waypoints. We define the following performance metrics for a planning solution:

Definition 1. *The* **trajectory delay** *for a* valid *trajectory i is the difference between trajectory arrival time at destination and the request start time: $delay = t_{m-1} - t_0$.*

The following metrics apply to a batch $reqs$ of N *Requests* resulting in n valid trajectories that complete in $t_{evac} = \max_i t_{m_i-1}$.

Definition 2. *The* **average trajectory delay** *is* $delay_{avg} = \frac{1}{n}\sum_i delay_i$ *over valid trajectories. The* **maximum trajectory delay** *is* $delay_{max} = \max_i delay_i$

Definition 3. *The* **request admission ratio** *is the fraction of valid trajectories vs. the total number of requests submitted,* $adm = n/N$ *.*

Definition 4. *The* **traffic flow rate** *is the number of vehicles that reach their destination vertex per time unit. This is the exit rate from the intersection. The traffic flow rate over a time period of duration* t_{evac} *is* $flow_T = n/t_{evac}\ [s^{-1}]$.

We make the following assumptions to design our algorithm:

1. The waypoint trajectory is converted by the CAV's own control systems to commands for actuators (throttle control, braking, steering) to maintain a trajectory with high fidelity.
2. Without loss of generality, all vehicles have the same dynamic constraints.
3. The optimization objective for the planning algorithm is to minimize the delay of each request while preserving the *first in - first out* order at intersection entry lanes. Minimizing the travel time reduces the overall utilization of shared intersection resources, such as graph edges and vertices, contributing to increased traffic flow.

4 The Space Time Graph Methodology for Trajectory Search

The proposed solution, Fastest Trajectory Planner (FTP), is inspired from the Space-Time graph planner for the drone delivery problems introduced in articles [10–13]. In contrast to our earlier work, the new algorithm works for autonomous cars carrying people and goods, supports multiple average speeds on graph edges, enforces dynamic vehicle constraints (e.g. min/max acceleration), and applies collision avoidance rules specific to road vehicles.

4.1 Collision Avoidance and Graph Representation

Fig. 2 illustrates several collision scenarios on graph G. Fig. 2a shows two vehicles moving on different edges in G_u towards the same vertex. Fig. 2b shows the red vehicle moving on an edge towards a vertex v occupied by the blue vehicle standing still. Fig. 2c shows the red vehicle on edge (u, v) and the blue vehicle on edge (v, u) moving towards each other. A vehicle (red) can stand still in the middle of an edge (u, v) while the blue vehicle comes barreling towards it from vertex u, Fig. 2d.

At the same time, two vehicles moving on the same long edge in the same direction, with similar speeds should be perfectly fine, with no collision.

Three salient observations are apparent:

(a) Graph edges and vertices occupied by a vehicle at a time are resources that must be allocated to vehicles with mutual exclusion on that time.
(b) Time-dependent allocation of resources for one request controls allocation for other vehicles, hence their movement, collision avoidance, and performance metrics.

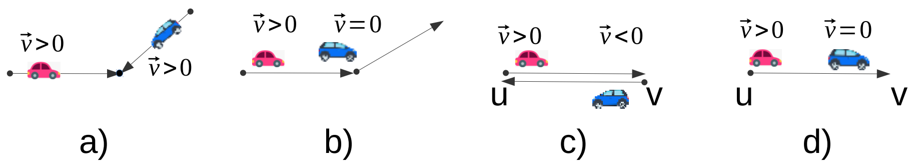


Fig. 2: Examples of collisions scenarios.

(c) The original graph G derived from the road map has insufficient space and time resolution for an adequate fine-grained granularity to achieve effective resource reuse.

The basic approach of our proposed algorithm is summarized here:

1. Discretize the original road map graph G to a spatial *unit graph* G_u that has fine-grained "granularity", e.g. 5 m.
2. From G_u, build a *space-time graph* G_t, with *time edges* for each vertex for a vehicle that stands there still for a time unit, and *space-time* edges for a vehicle that moves from one vertex to another during one or more time units.
3. Time variable availability of edges and vertices in G_u is modeled by existence of edges in the space-time graph.
4. A vehicle trajectory is expressed by a path in this space-time graph. Edges and vertices in G_t that form a trajectory and their adjacent neighbors are "allocated" exclusively to a vehicle's trajectory and removed from G_t. A trajectory (vehicle) cannot use vertices and edges in G_t already allocated to other trajectories.
5. The fastest ending (shortest) path in the space-time graph is a good approximation to the fastest trajectory in the original graph G.

Next are the key ideas underlining the Fastest Trajectory Planner's algorithm.

4.2 Discretized Graph and Discrete Time

The original road map graph G is discretized with unit length D (e.g. 5 m) so that each original edge $(u, v) \in E$ is split into smaller edges of length D and at most one shorter edge (w, v) at the end of (u, v). The new discretized graph is denoted by $G_u(V_u, E_u, pos)$, with $V \subseteq V_u$. $pos(u)$ represents the position of vertex u, as before. The size bound of the discretized graph is about the order of $|V_u| = \Theta(D^{-1}|V|)$, and $|E_u| = \Theta(D^{-1}|E|)$. Since $V \subseteq V_u$, the route planning problem on G is equivalent to the same problem on graph G_u. However, an optimal solution for G_u is suboptimal for the problem in G because of space discretization error.

Discretizing the graph allows one to run the planner in discrete time, with time units of δ_t (e.g. 1 second) expressed in time *ticks*.

We allow an edge to be traversed in discrete multiples of δ_t: $\{\delta_t, 2\delta_t, ..., p\delta_t\}$. This is called a *slow fragment* and p is called the *edge time multiplier*. We also allow for up to q consecutive edges to be traversed in just one δ_t time interval. We call this a *fast fragment* and q is the *edge speed multiplier*. The *unit edge speed* is $s_u = \frac{D}{\delta_t}$. The set of possible average speeds on edges in E_u of length D is $\{s_u, 2s_u, 3s_u, ..., q\, s_u\} \cup \{\frac{s_u}{2}, \frac{s_u}{3}, ..., \frac{s_u}{p}\}$, where constants $p, q \in \mathbb{N}^+$ are selected such that $qs_u \leq s_{max}$ and $\frac{s_u}{p}$ exceeds the minimum possible vehicle speed allowed. In our simulations $p = 2$ and $q = 4$. These parameters also affect runtime complexity, as discussed later.

4.3 The Space-Time Graph

We assume the planner computes trajectories for a sequence of requests over a finite time horizon $H > 0$, with $H = \min\{\{req_i.tf\}_{i=0..n-1} \cup \{T_{max}\}\}$. The discrete time horizon is defined as $K = \left\lfloor \frac{H}{\delta_t} \right\rfloor$, where T_{max} is the maximum simulation time.

The **space-time graph** G_t is built from the discretized unit graph G_u as follows. Each vertex $u \in V_u$ converts to K space-time vertices $(k, u) \in V_t$. Time edges in E_t are defined as $((k, u), (k + 1, u))$ for all $0 \leq k < K - 1$ and $u \in V_u$. Space-time edges are defined as $((k, u), (k + 1, v))$ for all $0 \leq k < K - 1$ and $(u, v) \in E_u$. A space-time edge is added for each edge in the discretized graph G_u and each time unit. The size of G_t is given by $|V_t| \in \Theta(K\delta_t^{-1}|V|)$, and $|E_t| \in \Theta(K\delta_t^{-1}|E|)$. The space-time graph for a very simple G_u is shown in Fig. 3a.

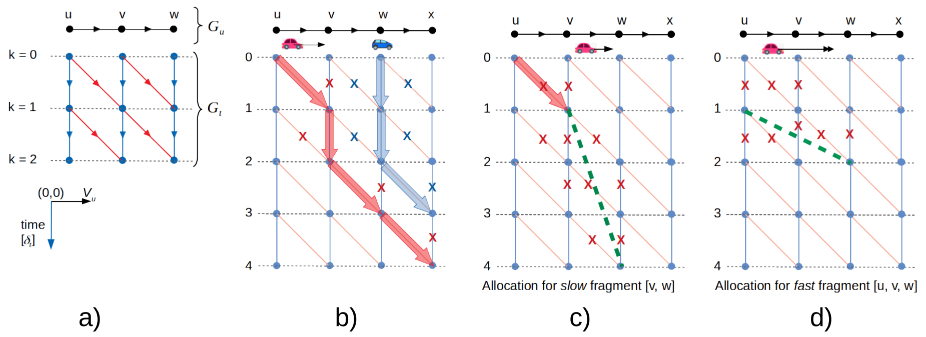

u
v
w
x
G_u
G_t
k = 0
k = 1
k = 2
(0,0)
V_u
time
[δ]
0
1
2
3
4
Allocation for *slow* fragment [v, w]
Allocation for *fast* fragment [u, v, w]
a)
b)
c)
d)

Fig. 3: a) Space-time graph G_t is derived from G_u by "extending" (u, v) edges in time (red) and by adding time edges (blue) for each vertex. b) space-time paths for two vehicles (red and blue wide arrows). Edges crossed with an **X** are pruned from G_t after admission to prevent collisions. c) a *slow fragment* (green dashed line) of one G_u edge traversed in more than one single tick at half the normal edge speed. d) a *fast fragment* (green dashed line) of more than 2 edges in one tick leads to higher speeds.

4.4 The Fastest Trajectory Planner Algorithm

The key ideas behind the use of the space-time graph are:

1. A shortest path for $Request(src, dst, t_s, T_{max})$ in the discretized graph G_u between vertices src and dst can be found by computing the "earliest ending" path in the space-time graph G_t between any space-time vertices (k_s, src) and (k_f, dst), with $t_s \leq k_s\delta_t \leq k_f\delta_t \leq t_s + T_{max}$, such that k_f is the minimum such value that is possible. This is the same as the multi-source, multi-destination shortest path problem in the space-time graph.

2. A space-time edge $e = ((k_u, u), (k_u + 1, v))$ is traversable at time k_u from u to v only if $e \in E_t$. We remove (prune) space-time edges from G_t to prevent collisions when ulterior requests (with later start times) are computed. Fig. 3b shows shortest space-time paths for two vehicles. Blue's path $u \to x$ is computed first. The algorithm finds a path with the edges covered by the wide blue arrow and, then, it prunes from G_t the edges marked with **X** signs from G_t. The path computation for the red vehicle will not *see* those deleted edges and it will find the space-time path drawn with the wide red arrows. The edges marked with **X** signs will be pruned from E_t. Space-time edges in G_t used by a solution path are removed from G_t
3. Support for multiple speeds (and dynamic constraints) is added by modifying the Dijkstra algorithm to consider during the "vertex expansion step" space-time path fragments (i.e. subpaths) corresponding to multiple traversal times and multiple space-time edges.

No optimality with constraints: since our planner enforces dynamic constraints (min/max acceleration) involving successive edges, we cannot prove that it finds the fastest path. Without constraints on acceleration, it does.

Several functions implement the planning algorithm.

4.5 planRequests: Top Level Algorithm

The entry to the planner is a function called *planRequests*, implemented in Algorithm 1. Function *planRequests($G_u, reqs$)* computes a trajectory for each $Request$ in list $reqs$, in the given input order.

Algorithm 1 : compute N admssion from list *requests*.

```
1: function planRequests($G_t, requests$)                 ▷ Process a list of requests
2:     $trjs = [\,]$                                              ▷ empty list
3:     for all $reqinrequests$ do
4:         $path \leftarrow$ computePath($G_u, req$)          ▷ plan one path: list of $V_t$ vertices
5:         $trj \leftarrow$ convertToTrajectory($G_u, req, path$)     ▷ convert path to Tr
6:         $trjs.append(trj)$
7:     return $trjs$                    ▷ all trajectories, including $Tr((), (), ())$ for failed ones
```

Line 4 calls function *computePath(G_t, req)* to compute the space-time path for the current request req. That is the main part of the planning algorithm. It returns in variable $path$ a list of vertices (waypoints) in G_t that defines a space-time trajectory in format $[(k_0, v_0), (k_1, v_1), ...]$, indicating that the vehicle must be at vertex v_0 at tick k_0, at v_1 at tick k_1, etc. This works also when multiple space-time edges are traversed in one tick.

4.6 computePath: the Shortest Space-Time Path Algorithm

The *computePath(G_t, req)* function call (Algorithm 2) computes the fastest space-time path from vertex $req.src$ to vertex $req.dst$ using only available edges in the space-time graph G_t. This algorithm runs a multi-source/multi-destination version of Dijkstra's shortest path algorithm. Once a space-time vertex (k_u, u) is reached our algorithm explores all feasible path fragments that start from time tick k_u like this:
• *slow fragments* (u, v) with exactly two space vertices that can be traversed in $1, 2, ..., q$ ticks; the average speed on this fragment does not exceed s_u (Fig. 3c).

• *fast fragments* $(u, ..., v)$ with three or more space vertices that can be traversed in exactly 1 tick; the average speed on this fragment may exceed D/δ_t (Fig. 3d).

A path fragment is *feasible* if it satisfies intersection/traffic lane constraints, its space-time edges are available in G_t, and if speed, acceleration/deceleration constraints (including vs. the previous fragment) are satisfied.

The priority queue that orders space-time vertex expansion holds objects of type $QueueEntry(k1, k0, priorVelocity, fragment, priorQe)$. $fragment$ is a list of V_u vertices $[u, ..., v]$ that forms a subpath in G_t traversable from tick $k0$ at u, arriving at v on tick $k1$. The fragment is constrained by available space-time edges in G_t and vehicle constraints. $priorQe$ is the currently expanding $QueueEntry$ object. $priorVelocity$ is the average 2D velocity vector across the $priorQe.fragment$ fragment.

QueueEntry objects in the priority queue created on line 5 are ordered by their $k1$ attribute, the arrival at their fragment's end vertex.

Algorithm 2 computes the shortest space-time path for one *Request*.

```
1:  function computePath($G_t, req$)
2:      $ks \leftarrow \lfloor req.ts/\delta_t \rfloor$ and $kf \leftarrow \lfloor req.tf/\delta_t \rfloor$  ▷ arrival ticks; last allowed exit time in ticks
3:      $explored \leftarrow \{(ks, req.src)\}$
4:      $queue \leftarrow$ new $PriorityQueue()$
5:      $queue.enqueue$(new $QueueEntry(ks, ks, (0.0, 0.0), [req.src], None)$)
6:      $ktime \leftarrow ks$                          ▷ ktime keeps the current exploration time tick
7:      $path \leftarrow [\,]$
8:      while $path == [\,]$ and $queue.size > 0$ and $ktime \leq kf$ do
9:          $qe \leftarrow queue.dequeue()$                 ▷ ordered by fragment end time, qe.k1
10:         continue if impossible to reach $req.dst$ from $qe.fragment[0]$ by tick $tf$
11:         update $ktime$ from $qe.k1$
12:         $nextQes \leftarrow discoverFragments(G_t, req, explored, qe)$
13:         if $nextQes.size > 0$ then                      ▷ if fragments were found
14:             for all $nqe$ in $nextQes$ with $nqe.k1 \leq kf$ do
15:                 if $req.dst \in nqe.fragment$ then            ▷ reached destination?
16:                     $path \leftarrow$ extract path from nqe and its predecessors
17:                     prune space time edges in $path$ from $G_t$
18:                     break
19:                 else
20:                     $queue.enqueue(nqe)$
21:     return $path$
```

Function $discoverFragments$ computes the feasible fragments consisting of *feasible* edges in the space-time graph: available in G_t and that satisfy the constraints on vehicle dynamics and intersection lanes (line 12). Line 16 checks if the destination vertex was reached. If so, it computes $path$ from the chain of queue entries, going backwards in time towards the root queue entry. In case of failure to find a path, the function returns the empty list [ ]. The call to $discoverFragments(G_t, req, explored, qe)$ explores the current space-time vertex from $(qe.k1, qe.fragment.last)$ and returns new $QueueEntry$ objects for the shortest *feasible* fragments with duration between 1 and maximum q ticks. Its algorithm runs a Breadth-First Search starting from space-time vertex $(qe.k1, qe.fragment.last)$ in G_t with a search radius limited to q ticks.

Each shortest feasible fragment $[u, ..., v]$ returned by function *discoverFragments*, with u at $k0$ and v and $k1$ has these properties:

- it has no cycles if it is longer than 2 vertices;
- it forms no cycles going back on the queue entry chain history;
- has only space-time edges available in G_t and that comply with constraints;
- there is no other faster fragment with the same space endpoints u and v.

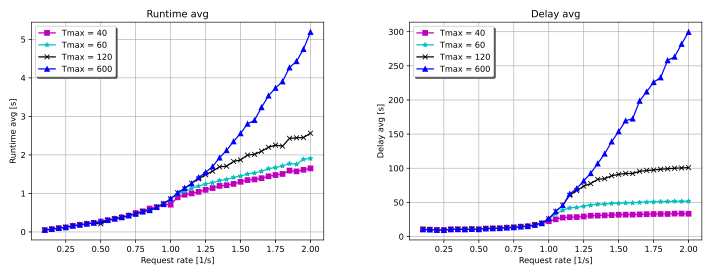
Function $discoverFragments$ uses the space-time vertices in parameter $explored$ and updates it during search with each encountered space-time vertex.

Runtime Complexity Analysis

In the following, f is the maximum number of exits reachable from any entry vertex, typically 2-4.The constrained *effective* average out-degree for exploration in V_u is $b \gtrsim 1$. In the 4-lane 4-way example from Fig. 1, $b = 1.0833$ and $f = 1$. Search in the space-time graph search will now branch only on time edges.

Other parameters for runtime complexity include (with typical values): the total number of requests to consider N: $10^1 - 10^3$, the discrete time horizon for trajectory computation $K = \Theta\left(H\delta_t^{-1}\right)$: $10^1 - 10^3$, the edge time multiplier p: 4 - 8, and the edge average speed multiplier q: 1, 2.

The runtime of $convertToTrajectory$ is $O(K)$. The time complexity of the top-level $planRequests$ algorithm is $O(NfK(p + b^{q+1} + \log_2 fK))$, with a heap priority queue. For intersections parameters f, p, q have moderate values and can be considered constant. In that case, the runtime is $O(NfK \log_2 fK)$ and does not seem to depend directly on the road map's graph topology, but on the discretization granularity D.



(a) Algorithm runtime. (b) Average trajectory delay.

Fig. 4: Algorithm runtime and average trajectory delay.

5 Performance Evaluation

We evaluate the performance of the Fastest Trajectory Planner for the 4-lane, 4-way intersection in Fig. 1b. $|V_u| = 36$, $|E_u| = 56$ edges, with average out-degree 1.55. Legal lanes restrict the *effective* edge out-degree during search to $b = 1.0833$. The space edge discretization length $D = 10m$ and the time tick unit is $\delta_t = 1s$. The speed multiplier $q = 2$ and the edge time multiplier $p = 4$. The top acceleration/deceleration is $2m/s^2$, consistent with a comfortable ride with enough braking ability.

Vehicle admission requests (random source/destination, no U-turns) are generated from time 0 to $t_{end} = 600s$ with a rate that varies from 0.1/s ($N = 60$) to 2/s ($N = 1200$) in 0.05 s ($\delta N = 30$) increments. All charts have on the horizontal axis this independent variable. The maximum allowed trajectory delay $T_{max} \in \{40, 60, 120, 600\}$s.

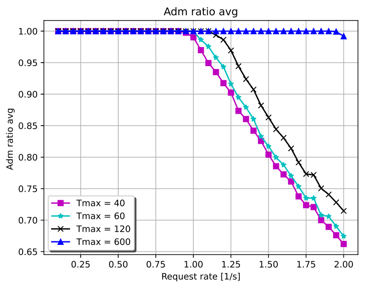
Saturation starts when the request rate approaches 1/s, $N = 600$ requests. When resource availability drops, vehicles experience higher waiting time, and metrics start deteriorating. This congestion behavior, with an inflection point, is common in scheduling with resource sharing. After congestion starts, requests will be rejected, bringing relief for resource contention. Most relief is seen for scenarios with $T_{max} < 600$

The average runtime is shown in Fig. 4a. It is proportional to the trajectory duration and it has the $N \log_2 N$ asymptotic trend. The worst running time was for a request rate of 2/s ($N = 1200$), with 4.323 ms/request, and a 5.5s total.

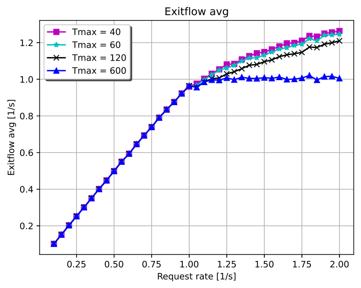
Fig. 4b shows the average trajectory delay. It has a very slow growth under 1 request/s, followed by a sharper growth above 1/s, when congestion begins that tapers off, converging to T_{max}. Note the a lower T_{max} value causes more requests to be dropped. This is evident in the request admission ratio chart from Fig. 5a. The admission ratio stays at 100% for all scenarios before congestion begins (at 1/s - 1.1/s). After that, the admission ratio starts a linear drop, delayed by a higher value for T_{max}.

Fig. 5b shows the evolution of the intersection exit flow rate vs. request rate. It follows the identity function before the congestion threshold, for all T_{max} values. It is constant for $T_{max} = 600s$ after that since the admission ratio is 100% up to the end and no requests are dropped. However, for $T_{max} < 600s$ the admission ratio is less than 100%, allowing only shorter trajectories. That causes a higher exit flow rate.

Finally, in the maximum traffic flow regime, we counted on average 14 vehicles present at the same time in the intersection. This high resource utilization should lead to superior traffic flow rates compared to alternatives.



(a) Request admission ratio.



(b) Average intersection exit flow rate (vehicle exits/second).

Fig. 5: Request admission ratio and intersection exit flow rate.

6 Conclusions

This paper proposes a novel algorithm for the Fastest Trajectory Planning problem for intersections with CAVs. The algorithm uses a shortest path search in a space-time discretized graph derived from the original road network graph. The algorithm enforces vehicle constraints (e.g. acceleration/deceleration) and it has a low runtime compared to that reported for state of art algorithms using Conflict Detection Graphs [4, 14]. The algorithm scales well with the number of admission requests and with the traffic graph size, the main limitation being the maximum path duration parameter T_{max}.

Future research directions include improving the search algorithm with A* and local search heuristics that reorder vehicle advance at each search step.

References

1. Choi, E.-H. Crash Factors in Intersection-Related Crashes: An On-Scene Perspective, HS-811 366. 2010.
2. Wang, J., Guo, X., and Yang, X.: Efficient and Safe Strategies for Intersection Management: A Review Sensors 21, no. 9: 3096 (2021)
3. Kiela, K. et. al : Review of V2X-IoT Standards and Frameworks for ITS Applications, Applied Sciences 10, no. 12: 4314 (2020).
4. Chen, C., Xu, Q., Cai, M., Wang, J., Wang, J., Li, K.: Conflict-free cooperation method for connected and automated vehicles at unsignalized intersections: Graph-based modeling and optimality analysis. IEEE Transactions on Intelligent Transport. Sys., 23(11), 21897 (2022)
5. He, X., Liu, X., Liu, HX.: Optimal vehicle speed trajectory on a signalized arterial with consideration of queue, Transp. Res. C, Emerg. Technol., vol. 61, pp. 106120, (2015)
6. Zhang, Y., Malikopoulos, A., Cassandras, C. G.: Decentralized optimal control for connected automated vehicles at intersections including left and right turns, in Proc. IEEE 56th Annu. Conf. Decis. Control (CDC), pp. 44284433, (2017).
7. Xu, B., Ban, X. J., Bian, Y., Wang, J., Li, K.: V2I based cooperation between traffic signal and approaching automated vehicles, in Proc. IEEE Intell. Vehicles Symp. (IV), (2017)
8. Xu, Xi, Zhang, Y., Li, L., Li, W.: Cooperative driving at unsignalized intersections using tree search, IEEE Trans. Intell. Transp. Syst., vol. 21, no. 11, pp. 45634571, (2019).
9. B. Xu et al., Distributed conflict-free cooperation for multiple connected vehicles at unsignalized intersections, Transp. Res. C, Emerg. Technol., vol. 93, pp. 322334, Aug. 2018.
10. Steinberg, A., Cardei, M., Cardei, I.: UAS Path Planning using a Space-Time Graph, IEEE SysCon, (2020).
11. Papa, R., Cardei, I., Cardei, M.: Energy-constrained drone delivery scheduling. In Combinatorial Optimization and Applications: 14th International Conference, COCOA 2020, Dallas, TX, USA, December 1113, 2020, Proceedings 14 (pp. 125-139). Springer (2020)
12. Steinberg, A., Cardei, M. Cardei, I.: UAS Batch Path Planning With a Space-Time Graph, in IEEE Open Journal of Intelligent Transportation Systems, vol. 2, pp. 60-72, doi: 10.1109/OJITS.2021.3070415 (2021)
13. Papa, R., Cardei, I., Cardei, M.: Generalized Path Planning for UTM Systems With a Space-Time Graph, in IEEE Open Journal of Intelligent Transportation Systems, vol. 3, pp. 351-368, doi: 10.1109/OJITS.2022.3171502 (2022)
14. Chen, C. et al.: A Graph-based Conflict-free Cooperation Method for Intelligent Electric Vehicles at Unsignalized Intersections, IEEE Int. Intelligent Transportation Sys. Conf. (2021).
15. Deng, Z., Shi, Y., Han, Q., Lu, L., Shen, W.: A Conflict Duration Graph-Based Coordination Method for Connected and Automated Vehicles at Signal-Free Intersections, Appl. Sci. (2020).