Graph-Based Modeling, Scheduling, and Verification for Intersection Management of Intelligent Vehicles

YI-TING LIN, HSIANG HSU, SHANG-CHIEN LIN, CHUNG-WEI LIN, and IRIS HUI-RU JIANG, National Taiwan University, Taiwan
CHANGLIU LIU, Carnegie Mellon University, USA

Intersection management is one of the most representative applications of intelligent vehicles with connected and autonomous functions. The connectivity provides environmental information that a single vehicle cannot sense, and the autonomy supports precise vehicular control that a human driver cannot achieve. Intersection management solves the fundamental conflict resolution problem for vehicles—two vehicles should not appear at the same location at the same time, and, if they intend to do that, an order should be decided to optimize certain objectives such as the traffic throughput or smoothness. In this paper, we first propose a graph-based model for intersection management. The model is general and applicable to different granularities of intersections and other conflicting scenarios. We then derive formal verification approaches which can guarantee deadlock-freeness. Based on the graph-based model and the verification approaches, we develop a centralized cycle removal algorithm for the graph-based model to schedule vehicles to go through the intersection safely (without collisions) and efficiently without deadlocks. Experimental results demonstrate the expressiveness of the proposed model and the effectiveness and efficiency of the proposed algorithm.

CCS Concepts: • **Computer systems organization → Embedded systems**; *Embedded and cyber-physical systems*;

Additional Key Words and Phrases: Conflict resolution, connected and autonomous vehicles, cycle removal, intersection management, verification

ACM Reference format:
Yi-Ting Lin, Hsiang Hsu, Shang-Chien Lin, Chung-Wei Lin, Iris Hui-Ru Jiang, and Changliu Liu. 2019. Graph-Based Modeling, Scheduling, and Verification for Intersection Management of Intelligent Vehicles. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 95 (October 2019), 21 pages.
https://doi.org/10.1145/3358221

1 INTRODUCTION

Intersection management is one of the most representative applications of intelligent vehicles with connected and autonomous functions. The connectivity provides environmental information that a single vehicle cannot sense, and the autonomy supports precise vehicular control that a human

This article appears as part of the ESWEEK-TECS special issue and was presented at the International Conference on Embedded Software (EMSOFT) 2019.
Authors' addresses: Y.-T. Lin, H. Hsu, S.-C. Lin, C.-W. Lin, and I. H.-R. Jiang, National Taiwan University, No. 1, Sec. 4, Roosevelt Road, Taipei, 10617, Taiwan; emails: {r07943102, r07943109, r07943106}@ntu.edu.tw, cwlin@csie.ntu.edu.tw, huirujiang@ntu.edu.tw; C. Liu, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213, USA; email: cliu6@andrew.cmu.edu.

1539-9087/2019/10-ART95 $15.00
https://doi.org/10.1145/3358221

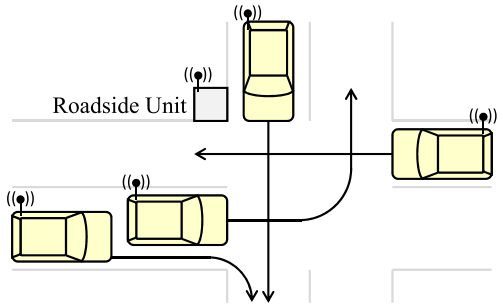

Fig. 1. Intersection management is crucial for safety and traffic efficiency. With connected and autonomous functions, the safety and traffic efficiency can be improved.

driver cannot achieve. As shown in Figure 1, an intersection represents the potential path conflicts between vehicles and significantly influences traffic flows, and thus intersection management is crucial for safety and traffic efficiency. It solves the fundamental conflict resolution problem for vehicles—two vehicles should not appear at the same location at the same time, and, if they intend to do that, an order should be decided to optimize certain objectives such as the traffic throughput or smoothness.

Traditionally, the intersection management focuses on adapting the lengths of traffic signals to real-time traffic conditions such as queue lengths, flow speeds, and information from neighboring intersections [5]. With the development of connected and autonomous functions, intelligent intersection management, where physical traffic signals turn into cyber ones, has become a popular research topic in recent years. Most related works design protocols between vehicles and a centralized intersection manager which receives requests from vehicles, decides their passing order, and provides confirmations or instructions to them [3, 4, 11, 12, 16, 26, 27]. In the control and robotics domains, there are also related works focusing on discrete-event control and conflict resolution [2, 7, 9, 22] in a centralized setting. Besides these centralized approaches, it is also possible for vehicles to communicate with each other and decide the passing order by themselves distributively [18, 20]. In general, a centralized approach has better controllability, reliability, and overall system awareness, but a distributed approach is more adaptive, especially when the infrastructure cost is a concern.

Especially, Liu et al. formulated the high-level control (passing order) problem as a cycle removal problem in a graph model [18]. A conflict graph is first derived and available to all vehicles. Based on the conflict graph, each vehicle follows a cycle removal algorithm to remove cycles from the graph so that *the passing order can be obtained from the remaining acyclic graph*. The concept here is essential as connected and autonomous vehicles will still be manufactured by different original equipment manufactures (OEMs). As a result, governments and OEMs need to define and agree on how to resolve conflicts in advance so that each OEM can design their vehicles accordingly. Here, "how to remove cycles in a graph" is exactly the agreement which should be defined and verified (to be collision-free and deadlock-free).

Regarding cycle detection or cycle removal, it is a traditional problem in graph theory [8, 14, 25], and it has been widely used in the domain of operating systems or distributed systems [6, 19, 23]. However, there are some automotive features that are different from existing problems. For example, one can delete a thread to remove a cycle, which implies a deadlock, in an operating system, but we cannot remove a vehicle from an intersection. On the other hand, regarding verification or its corresponding formal models in an intersection scenario, Ahmane et al. used a timed Petri net to model an intersection controller [1], and Wu et al. also used a Petri net to model

a two-way intersection controller [24], but both of them did not provide follow-up model checking. Zheng et al. considered message delays or jamming attacks in an intelligent intersection and verified the corresponding intersection manager and its timed automata to be deadlock-free [26].

The work [18] gives the insight into using a graph model to solve the intersection management problem, but there are still some limitations:

- It models the intersection as a single conflict zone in the cycle removal process, but we will show that it limits the modeling expressiveness, solution space, and thus solution quality.
- Same as many existing works, it considers only the first few vehicles approaching the intersection, and the solution quality can be further improved by considering more vehicles, especially as the communication range of connected vehicles can be up to few hundred meters.
- It applies a simple rule-based heuristic to remove cycles, and the solution quality can also be further improved by a more sophisticated algorithm.

To address these limitations, the main contributions of this paper are as follows:

- We propose a graph-based model for intersection management. The model is more general and applicable to different granularities of intersections and other conflicting scenarios.
- We develop a centralized cycle removal algorithm for the graph-based model to schedule vehicles to go through the intersection safely (without collisions) and efficiently without deadlocks. The algorithm is sufficiently efficient to consider more conflict zones and more vehicles in real time.
- We derive formal verification approaches which can guarantee deadlock-freeness.
- Experimental results demonstrate the expressiveness of the proposed model and the effectiveness and efficiency of the proposed algorithm.

It should be mentioned that, although our cycle removal algorithm is centralized, it is very lightweight so that it is possible for a vehicle to perform the computation.

The rest of this paper is organized as follows: Section 2 presents our graph-based model and problem formulation. Section 3 demonstrates our verification approaches. Section 4 describes our scheduling algorithm based on cycle removal. Section 5 provides experimental results, and Section 6 concludes this paper.

2 MODELING AND PROBLEM FORMULATION

In this section, we define the problem formulation, and the notation is summarized in Table 1.

Intersection. There is one intersection. Within the intersection, the *trajectory* from a *source lane* to a *destination lane* is fixed.

Conflict Zone. A conflict zone is the crossing location of two trajectories, and two vehicles cannot be at (occupy) the same conflict zone at the same time. There are n conflict zones, $\Xi_1, \Xi_2, \ldots, \Xi_n$, in the intersection. This model allows us to consider different granularities of an intersection. For example, as shown in Figure 2, the intersection can be modeled by 1, 4, 16, and 24 conflict zone(s), and much more alternatives are possible.

Vehicle. All vehicles are connected and autonomous. Each vehicle has a fixed route—it fixes its source lane (before the intersection) and destination lane (after the intersection), and it does not change lanes before and after the intersection. Each vehicle periodically broadcasts the Basic Safety Message (BSM) including the position, speed, etc., and its source and destination lanes (note that, given the source and destination lanes, its trajectory within the intersection is given and fixed). Two vehicles have a *conflict* at Ξ_j if and only if Ξ_j is on the both trajectories.

Table 1. The Notation

Index	i, i'	the index of a vehicle
	j, j', j''	the index of a conflict zone
	(i, j)	the index of a vertex
	k	the index of an edge
Given (Input)	G	a timing conflict graph
	m	the number of vehicles
	n	the number of conflict zones
	Δ_i	the i-th vehicle
	Ξ_j	the j-th conflict zone
	$v_{i,j}$	the (i, j)-th vertex
	e_k	the k-th edge
	a_i	the earliest arrival time of Δ_i
	$p_{i,j}$	the vertex passing time of $v_{i,j}$
	w_k	the edge waiting time of e_k
Output	G'	an acyclic timing conflict graph
	$s_{i,j}$	the vertex entering time of $v_{i,j}$

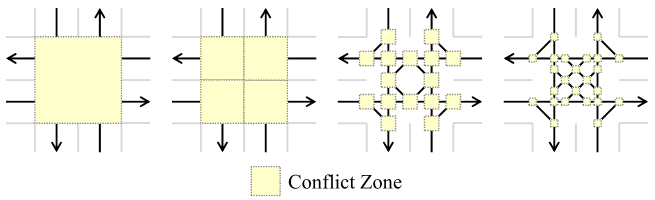

Fig. 2. The model allows us to consider different granularities of an intersection. The intersection can be modeled by 1, 4, 16, and 24 conflict zone(s). and much more alternatives are possible.

Intersection Manager. The intersection is associated with a periodic task, intersection manager, for the intersection management. For each period, the intersection manager receives the information from m vehicles, $\Delta_1, \Delta_2, \ldots, \Delta_m$, within its communication range. Based on the received information, the intersection manager assigns a time window to each vehicle at each conflict zone on the trajectory of the vehicle.

Timing Conflict Graph. A directed timing conflict graph $G = (V, E)$ is constructed by the following rules:

- There is a *vertex* $v_{i,j}$ if and only if Ξ_j is on the trajectory of Δ_i.
- There is a *Type-1 edge* $(v_{i,j}, v_{i,j'})$ if and only if the next conflict zone of Ξ_j on the trajectory of Δ_i is $\Xi_{j'}$.
- There is a *Type-2 edge* $(v_{i,j}, v_{i',j})$ if and only if Δ_i and $\Delta_{i'}$, on the same source lane and with the order where Δ_i is in front of $\Delta_{i'}$, have a conflict at Ξ_j.
- There are two *Type-3 edges* $(v_{i,j}, v_{i',j})$ and $(v_{i',j}, v_{i,j})$ if and only if Δ_i and $\Delta_{i'}$, on different source lanes, have a conflict at Ξ_j.

Note that the vertex set is a subset of the Cartesian product of the sets of vehicles and conflict zones. An example and its timing conflict graph are shown in Figure 3(a) and (b), respectively. Here, we can also show the expressiveness of our models with conflict zones and timing conflict graphs. Another example and one of its solutions are shown in Figure 4(a) and (b), respectively, where the two vehicles can enter the intersection at the same time. However, as shown in Figure 4(c) and (d),

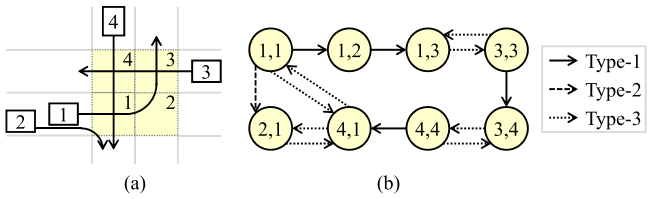

Fig. 3. (a) An example and (b) its timing conflict graph.

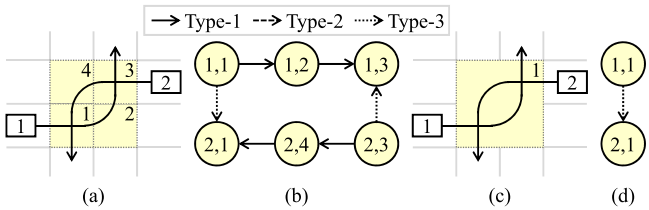

Fig. 4. (a) An example and (b) one of its solution where the two vehicles can enter the intersection at the same time. (c) If the intersection is modeled by only one conflict zone (or a vertex in the timing conflict graph represents a vehicle only, not the product of a vertex and a conflict zone), (d) its expressiveness is limited, and the two vehicles cannot enter the intersection at the same time.

if the intersection is modeled by only one conflict zone (or a vertex in the timing conflict graph represents a vehicle only, not the product of a vertex and a conflict zone) [18], its expressiveness is limited, and the two vehicles cannot enter the intersection at the same time.

Earliest Arrival Time. Each vehicle Δ_i is associated with a_i, the earliest arrival time for Δ_i to arrive at the first conflict zone on its trajectory, without being delayed by any other vehicle (*i.e.*, no vehicle is in front of Δ_i before the intersection). It can be either computed or provided by Δ_i or computed by the intersection manager.

Edge Waiting Time. Each edge $e_k = (v_{i,j}, v_{i',j'})$ is associated with w_k, the waiting time "length" from Δ_i leaving Ξ_j to $\Delta_{i'}$ entering $\Xi_{j'}$, without being delayed by any other vehicle. For a Type-1 edge e_k (where $i = i'$), w_k is the time from Δ_i leaving Ξ_j to Δ_i entering $\Xi_{j'}$; for a Type-2 or Type-3 edge e_k (where $j = j'$), w_k is the time from Δ_i leaving Ξ_j to $\Delta_{i'}$ entering Ξ_j. In practice, the waiting time of a Type-2 edge e_k is smaller than that of a Type-3 edge $e_{k'}$ as vehicles from the same source lane can perform better in vehicle-following.

Vertex Passing Time. Edge vertex $v_{i,j}$ is associated with $p_{i,j}$, the time "length" for Δ_i from entering Ξ_j to leaving Ξ_j.

Vertex Entering Time. Each vertex $v_{i,j}$ is associated with $s_{i,j}$, the time for Δ_i to enter Ξ_j, which implies that the earliest time for Δ_i to leave Ξ_j is $s_{i,j} + p_{i,j}$. If a timing conflict graph G' is *acyclic*, the vertex entering time of each vertex is assigned as follows[1]:

- As the graph is acyclic, the assignment can follow a topological order. If there are multiple options, a Type-1 edge has a higher priority than a Type-2 or Type-3 edge.
- If $v_{i,j}$ is the first conflict zone on the trajectory of Δ_i,

$$s_{i,j} = \max \Bigg\{ a_i, \max_{k \mid e_k = (v_{i',j'}, v_{i,j}) \in G'} \{ s_{i',j'} + p_{i',j'} + w_k \}, \max_{k' \mid e_k = (v_{i',j'}, v_{i,j}) \in G', e_{k'} = (v_{i',j'}, v_{i',j''}) \in G', e_k \neq e_{k'}} \{ s_{i',j''} - w_{k'} + w_k \} \Bigg\}. \quad (1)$$

[1]As there is dependency between vehicles, the vertex entering time of each vertex cannot be given as an input.

Note that $j = j'$ is always true in this case. The last maximum term is to make sure that $\Delta_{i'}$ leaves $\Xi_{j'}$ for $\Xi_{j''}$ so that Δ_i can enter Ξ_j. For easier understanding, we can also set the intersection-entering point of each source lane as a conflict zone so that it is the first conflict zone of the trajectory of each vehicle from the source lane.

- Otherwise,

$$s_{i,j} = \max \left\{ \max_{k \mid e_k = (v_{i',j'}, v_{i,j}) \in G'} \{s_{i',j'} + p_{i',j'} + w_k\}, \right.$$
$$\left. \max_{k' \mid e_k = (v_{i',j'}, v_{i,j}) \in G', e_{k'} = (v_{i',j'}, v_{i',j''}) \in G', e_k \neq e_{k'}} \{s_{i',j''} - w_{k'} + w_k\} \right\}. \quad (2)$$

Note that either $i = i'$ or $j = j'$ is always true in this case. If $i = i'$, the last maximum term is not needed.

Problem Formulation. Given a conflict graph G, the earliest arrival time a_i of each vehicle Δ_i, the edge waiting time w_k of each edge e_k, and the vertex passing time $p_{i,j}$ of each vertex $v_{i,j}$, the problem is to

(i) Compute an acyclic subgraph G' of G, where
- For each vertex v_i in G, v_i is also in G',
- For each Type-1 edge e_k in G, e_k is also in G',
- For each Type-2 edge e_k in G, e_k is also in G',[2] and
- For each pair of vertices $v_{i,j}$ and $v_{i',j}$ in G, there exists a path either from $v_{i,j}$ to $v_{i',j}$ or from $v_{i',j}$ to $v_{i,j}$ in G',

(ii) Guarantee no deadlock,

(iii) Assign the vertex entering time $s_{i,j}$ of each vertex $v_{i,j}$ (as the paragraph above), and

(iv) Minimize

$$\max_{v_{i,j}} (s_{i,j} + p_{i,j}), \quad (3)$$

which is the total time needed for all vehicles to go through the intersection.

The item (i) is the safety (*collision-freeness*) property to guarantee an order for vehicles having a conflict. The item (iii) follows the order to schedule vehicles, and the item (iv) is the objective function. The item (ii) is the liveness (*deadlock-freeness*) property. To this point, we have not detailed how to guarantee no deadlock—it will be demonstrated in Section 3.

2.1 Assumptions and Variants

We assume that the communication is perfect in this paper. To mitigate communication failure or data corruption, one approach is to increase the edge waiting time of an edge, *i.e.*, being more pessimistic. No matter how, each vehicle should still have basic safety features, such as a collision avoidance system which is not based on wireless communication between vehicles. A detailed analysis for the optimal setting of edge waiting time is probabilistic, also involving some trade-off analysis (if it is too pessimistic, the overall performance will be bad; however, if it is too optimistic, the overall performance will also be bad due to frequent emergency braking by basic safety features). Besides, we can also consider the vehicle dynamics by setting the given variables such as the edge waiting time and the vertex passing time, but we mainly focus on the application-level modeling, scheduling, and verification in this paper.

[2]We do not consider overtaking in this paper; otherwise, we can relax the constraint to potentially remove Type-2 edges.

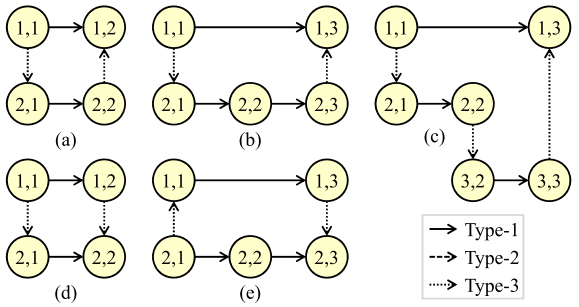

Fig. 5. (a)–(c) Examples with deadlocks and (d)–(e) examples without deadlocks.

As defined, the intersection manager is a periodic task for the intersection management. For each period, the intersection manager receives the information from m vehicles and solves the problem for all of the m vehicles. The vehicles coming in the next period (or the vehicles not in the communication range in this period) are considered in the next period. One alternative is to leave some vehicles unscheduled and then schedule them in the next period—this is a more complicated setting.

It is also possible to solve the problem in a distributed setting. Same as the previous work [18], the key point in a distributed setting is to have an agreement (between vehicles) on how to remove cycles. With the agreement, each vehicle remove cycles separately and periodically. If the computation is fast enough and the communication is perfect, the whole system can reach a stable state. Again, each vehicle should still have basic safety features, if the computation and communication conditions cannot be guaranteed.

3 DEADLOCK-FREENESS VERIFICATION

Collision-freeness has been guaranteed by the passing order and scheduling in the problem formulation. In this section, we will demonstrate two approaches based on graphs and Petri nets which can guarantee deadlock-freeness. Each of them can serve as a routine for the scheduling in Section 4 to verify deadlock-freeness for G'. Although the graph-based verification is more straightforward with the proposed graph model, the Petri-net-based verification can be an alternative with existing verification methodologies and tools such as PIPE2 [10].

3.1 Graph-Based Verification

Having no cycle in G' or G does not guarantee deadlock-freeness,[3] and some examples are shown in Figure 5(a)–(c).[4] In Figure 5(a) and (b), there are deadlocks because Δ_2 is waiting Δ_1 to leave Ξ_1, but Δ_1 is waiting Δ_2 to enter and leave Ξ_2 and Ξ_3, respectively. In Figure 5(c), there is a deadlock because Δ_2 is waiting Δ_1 to leave Ξ_1, and Δ_3 is waiting Δ_2 to enter and leave Ξ_2 (from Ξ_1), but Δ_1 is waiting Δ_3 to enter and leave Ξ_3 (from Ξ_2). On the contrary, in Figure 5(d), there is no deadlock as Δ_1 enters Ξ_1 and Ξ_2 before Δ_2. In Figure 5(e), even if Δ_2 enters Ξ_1 first, Δ_2 enters Ξ_2 after that so that Δ_1 is able to enter Ξ_1 (after Δ_2) and Ξ_3 (before Δ_2) without a deadlock.

As illustrated above, having no cycle in G' cannot verify that there is no deadlock. Therefore, we introduce *resource conflict graphs* as follows:

[3]This is the reason that we need the item (ii) in the problem formulation.
[4]To demonstrate the examples concisely, the examples in Figure 5 are not associated with any intersection modeling in Figure 2.

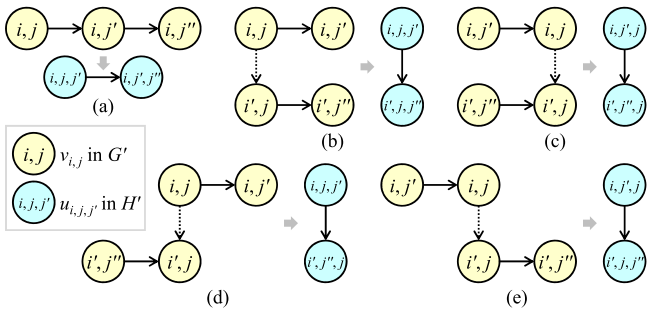

Fig. 6. The construction rules of resource conflict graphs.

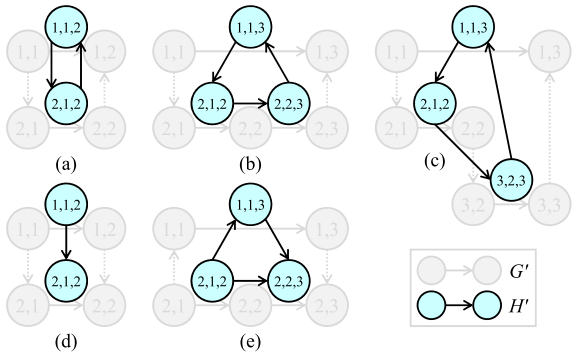

Fig. 7. The resource conflict graphs of the examples in Figure 5.

Resource Conflict Graph. The directed resource conflict graph H' of G' is constructed by the following rules:

- There is a *vertex* $u_{i,j,j'}$ if and only if there is a Type-1 edge $(v_{i,j}, v_{i,j'})$ in G'.
- If there are edges $(v_{i,j}, v_{i,j'})$ and $(v_{i,j'}, v_{i,j''})$ in G', then there is an edge $(u_{i,j,j'}, u_{i,j',j''})$ in H' (illustrated in Figure 6(a)).
- If there are edges $(v_{i,j}, v_{i',j})$, $(v_{i,j}, v_{i,j'})$, and $(v_{i',j}, v_{i',j''})$ in G', then there is an edge $(u_{i,j,j'}, u_{i',j,j''})$ in H' (illustrated in Figure 6(b)).
- If there are edges $(v_{i,j}, v_{i',j})$, $(v_{i,j'}, v_{i,j})$, and $(v_{i',j''}, v_{i',j})$ in G', then there is an edge $(u_{i,j',j}, u_{i',j'',j})$ in H' (illustrated in Figure 6(c)).
- If there are edges $(v_{i,j}, v_{i',j})$, $(v_{i,j}, v_{i,j'})$, and $(v_{i',j''}, v_{i',j})$ in G', then there is an edge $(u_{i,j,j'}, u_{i',j'',j})$ in H' (illustrated in Figure 6(d)).
- If there are edges $(v_{i,j}, v_{i',j})$, $(v_{i,j'}, v_{i,j})$, and $(v_{i',j}, v_{i',j''})$ in G', then there is an edge $(u_{i,j',j}, u_{i',j,j''})$ in H' (illustrated in Figure 6(e)).

The general concept of the last four rules is that, if there is an edge $(v_{i,j}, v_{i',j})$ in G', then there is an edge from each vertex (which corresponds to an edge in G') involving $v_{i,j}$ to each vertex (which corresponds to an edge in G') involving $v_{i',j}$ in H'. It implies that, if Δ_i enters Ξ_j before $\Delta_{i'}$ enters Ξ_j, then Δ_i must leave Ξ_j before $\Delta_{i'}$ enters Ξ_j. The resource conflict graphs of the examples in Figure 5 are shown in Figure 7. We can observe that they are cyclic in Figure 7(a)–(c), while they are acyclic in Figure 7(d)–(e).

Theorem 3.1. *H' is cyclic if and only if G' has a deadlock.*

Proof. From left-hand side (LHS) to right-hand side (RHS): If there is a cycle in H', we assume the cycle as $((i_0, j_0, j_0'), (i_1, j_1, j_1'), \ldots, (i_k, j_k, j_k'), \ldots, (i_l, j_l, j_l'))$, where $(i_l, j_l, j_l') = (i_0, j_0, j_0')$. By the construction rules of H', for any pair of (i_k, j_k, j_k') and $(i_{k+1}, j_{k+1}, j_{k+1}')$, at least one equality of $j_k = j_{k+1}, j_k = j_{k+1}', j_k' = j_{k+1}$, and $j_k' = j_{k+1}'$ is true. Assume that it is equal to j^* in the true equality. By the definition of a conflict zone (that two vehicles cannot be at the same conflict zone at the same time), Δ_{i_k} must leave Ξ_{j^*} before $\Delta_{i_{k+1}}$ enters Ξ_{j^*}. This means that (i_k, j_k, j_k') blocks $(i_{k+1}, j_{k+1}, j_{k+1}')$, and thus, considering $0 \le k \le l-1$, the cycle forms a deadlock.

From RHS to LHS: If there is a deadlock, without loss of generality, we assume that Δ_i cannot move from Ξ_j to $\Xi_{j'}$. The conditions that Δ_i cannot move from Ξ_j to $\Xi_{j'}$ include[5] (1) Δ_i cannot move from another conflict zone $\Xi_{j''}$ to Ξ_j, (2) another vehicle $\Delta_{i'}$ scheduled to enter Ξ_j earlier cannot enter Ξ_j, (3) another vehicle $\Delta_{i'}$ scheduled to leave Ξ_j earlier cannot leave Ξ_j, (4) another vehicle $\Delta_{i'}$ scheduled to enter $\Xi_{j'}$ earlier cannot enter $\Xi_{j'}$, and (5) another vehicle $\Delta_{i'}$ scheduled to leave $\Xi_{j'}$ earlier cannot leave $\Xi_{j'}$. By the construction rules of H', each of the conditions constructs an edge to vertex (i, j, j') in H'. Repeating applying the same conditions, those edges must form a cycle[6] since the numbers of vehicles and conflict zones are finite. □

By Theorem 3.1, H' is acyclic if and only if G' has no deadlock (deadlock-freeness). Note that we construct H' from G'. After the construction, we do not need G' in the verification.

3.2 Petri-Net-Based Verification

We also use another model, Petri net [21], to verify if G' is deadlock-free. A Petri net is suitable for modeling a distributed dynamic system, where its concurrent and distributed properties can model vehicles' behavior at an intersection. A Petri net is defined as a 5-tuple $\Pi = (P, T, I, O, M_0)$, where P is a finite set of places. T is a finite set of transitions. $I : P \times T \to \mathbb{N}$ is an input function that defines arcs from places to transitions, where $\mathbb{N}$ is the set of natural numbers. $O : T \times P \to \mathbb{N}$ is an output function that defines arcs from transitions to places. $M_0 : P \to \mathbb{N}$ is the initial marking.

In our model, a place $q_{i,j}$ represents the state of Δ_i at Ξ_j; a place $q_{i,\text{in}}$ represents the state of Δ_i at its source lane, a place $q_{i,\text{out}}$ represents the state of Δ_i at its destination lane, and a place $q_{i,i',j}$ represents the available state of Ξ_j between Δ_i and $\Delta_{i'}$. A transition $t_{i,j,j'}$ represents that Δ_i moves from Ξ_j to $\Xi_{j'}$. An arc runs from a place to a transition or vice versa.

Petri Net Construction. Given G', a Petri net Π is constructed by the following rules:

- For every vertex $v_{i,j}$ in G',
  $-q_{i,j} \in P$.
- For every Δ_i, assume that Ξ_j and $\Xi_{j'}$ are the first and last conflict zones on the trajectory of Δ_i, respectively,
  $-q_{i,\text{in}} \in P$ and $q_{i,\text{out}} \in P$.
  $-t_{i,\text{in},j} \in T$ and $t_{i,j',\text{out}} \in T$.
  $-I(q_{i,\text{in}}, t_{i,\text{in},j}) = 1$ and $I(q_{i,j'}, t_{i,j',\text{out}}) = 1$.
  $-O(t_{i,\text{in},j}, q_{i,j}) = 1$ and $O(t_{i,j',\text{out}}, q_{i,\text{out}}) = 1$.
  $-M_0(q_{i,\text{in}}) = 1$.
- For every Type-1 edge $(v_{i,j}, v_{i,j'})$ in G',
  $-t_{i,j,j'} \in T$.
  $-I(q_{i,j}, t_{i,j,j'}) = 1$.
  $-O(t_{i,j,j'}, q_{i,j'}) = 1$.

[5] If all of the conditions are false, then Δ_i can move from Ξ_j to $\Xi_{j'}$. A similar claim is not true for G', so having no cycle in G' cannot guarantee deadlock-freeness.
[6] Though it may not go back to (i, j, j').

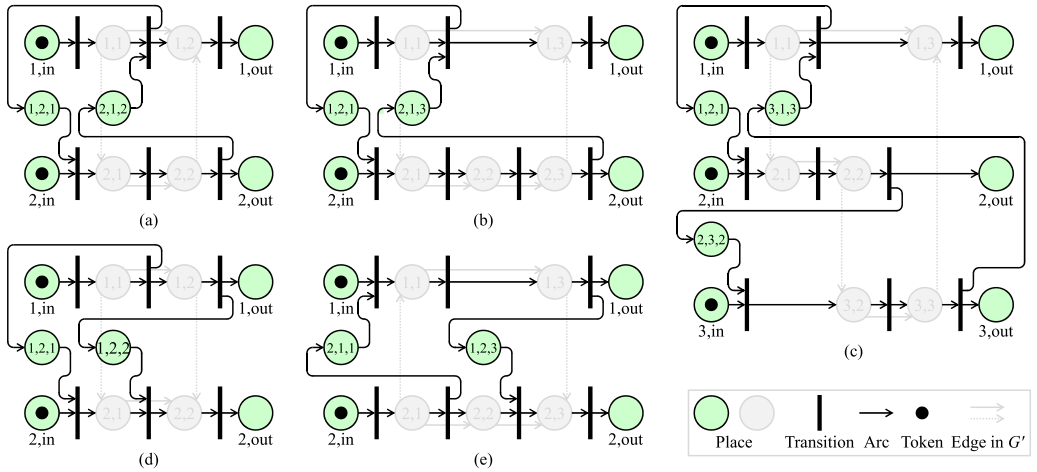

Fig. 8. The Petri nets of the examples in Figure 5. The edges in G' do not belong to the Petri nets (for reference only).

- For every Type-2 or Type-3 edge $(v_{i,j}, v_{i',j})$ in G', assume that $\Xi_{j'}$ is the previous conflict zone of Ξ_j on the trajectory of $\Delta_{i'}$ (if Ξ_j is the first one, then j' = "in") and $\Xi_{j''}$ is the next conflict zone of Ξ_j on the trajectory of Δ_i (if Ξ_j is the last one, then j'' = "out"),
  $-q_{i,i',j} \in P$.
  $-I(q_{i,i',j}, t_{i',j',j}) = 1$.
  $-O(t_{i,j,j''}, q_{i,i',j}) = 1$.

The corresponding Petri nets of the examples in Figure 5 are shown in Figure 8. In the constructed Petri net, initial marking represents that every vehicle is at its source lane. When we perform execution on the Petri net, the flow of tokens represents the movement of vehicles from source lanes to destination lanes. If $q_{i,\text{out}}$ has a token, then we verify that Δ_i can pass the intersection. If $q_{i,\text{out}}$ has a token for all i, then we verify that all vehicles can pass the intersection. However, if no transition can be fired before $q_{i,\text{out}}$ has a token, then Δ_i cannot pass the intersection, and there exists a deadlock. It can be seen that there are deadlocks in Figure 8(a)–(c), while there is no deadlock in Figure 8(d)–(e).

Theorem 3.2. *The Petri net Π has a deadlock if and only if G' has a deadlock.*

Proof. From LHS to RHS: By definition, Π has a deadlock if there is no more transition which can be fired. In our Petri net model, when there is a deadlock, there is at least one place $q_{i,i',j}$ which will never receive a token. Note that $q_{i,i',j}$ represents the available state of Ξ_j between Δ_i and $\Delta_{i'}$, and $q_{i,i',j}$ will receive a token if and only if Δ_i leave Ξ_j. Therefore, if $q_{i,i',j}$ will never receive a token, it means that Δ_i cannot leave Ξ_j before $\Delta_{i'}$ enters Ξ_j, which forms a deadlock.

From RHS to LHS: If G' has a deadlock, without loss of generality, we assume that Δ_i cannot move from Ξ_j to $\Xi_{j'}$. In our Petri net model, if Δ_i cannot move from Ξ_j to $\Xi_{j'}$, there is a place $q_{i',i,j'}$ which will never receive a token. Therefore, there will be no more transition which can be fired after all other fireable transitions are fired, which causes a deadlock in Π. □

By Theorem 3.2, Π has no deadlock if and only if G' has no deadlock (deadlock-freeness), and the Petri-net-based verification can provide the same verification results as the graph-based verification.

4 SCHEDULING AND CYCLE REMOVAL

4.1 Motivations

To decide the passing order for vehicles to go through the intersection without traditional traffic signals, there is a naive implementation that schedules the vehicles based on their earliest arrival times. We call this greedy method as a *First-Come-First-Serve* approach. However, the First-Come-First-Serve approach ignores much key information, such as the interactions between vehicles and conflict zones, and thus leads to extra delay in many cases. To address this problem, with the graph-based model in Section 2 and the verification approaches in Section 3, we can decide the passing order for vehicles to go through the intersection safely and efficiently by removing all cycles in the graph.

The most common method to detect and remove cycles in a directed graph is the Depth-First Search (DFS) algorithm [15]. There is a cycle in a graph only if a *back edge*, which is an edge from a vertex to itself or its ancestors, is found during the DFS traversal of the graph. Then, the method can remove any edge in the cycle to avoid having cycle in the graph. However, without optimization objective, the DFS method may not remove "good" edges to perform optimization. Furthermore, to decide a passing order, we cannot remove some edges because of the safety property (item (i)) in our problem formulation, and thus the direct use of a DFS method is not feasible. On the other hand, the minimum feedback arc set problem, a special case of our problem, is NP-hard [14] and has not known to be approximable within a constant [13].

Our objective is to minimize the total time needed for all vehicles to go through the intersection, equivalent to the leaving time of the last vehicle. To remove cycles while considering the edge costs, finding a minimum spanning tree (MST) of the graph can be a potential solution, and one approach is the Kruskal's algorithm [17]. The Kruskal's algorithm repeatedly chooses a minimum-cost edge which does not form any cycle with those already-chosen edges. Kruskal also proposed the backward version of the original one, and it repeatedly removes a maximum-cost edge whose removal does not disconnect the graph. Inspired by this method, we do intend to remove the edge which results in the largest delay to the objective. This can remove cycles and benefit the objective minimization at the same time.

4.2 Overview

Based on our graph model, we develop a cycle removal algorithm. First, we compute the vertex entering time of each vertex without considering Type-3 edges. Next, the costs of Type-3 edges are estimated by their impacts on the objective. Then, we remove a Type-3 edge which has the largest cost from the graph. The impact of $(v_{i,j}, v_{i',j})$ on the objective is measured by considering $(v_{i,j}, v_{i',j})$ when recomputing the vertex entering time of each vertex. Repeating those steps, we can remove cycles and compute the vertex entering time of each vertex in the graph. It should be noted that, sometimes, we cannot remove an edge because of the last constraint of the item (i) in the problem formulation. In this case, we divide the problem into sub-problems and solve the sub-problems.

4.3 Definitions

We first provide some definitions which will be used in our algorithm as follows.

Edge State. There are four possible states for an edge:

- An edge is **ON** if it has been decided to be kept (in G'). By the item (i) in the problem formulation, a Type-1 or Type-2 edge is always **ON**. When discussing the graph G', we only consider **ON** edges.
- An edge is **OFF** if it has been decided to be removed.

- An edge is **UNDECIDED** if it is going to be decided in the current sub-problem.
- An edge is **DONTCARE** if it is not considered in the current sub-problem.

Vertex State. There are three possible states for a vertex:

- A vertex is **BLACK** if its vertex entering time has been scheduled. If $v_{i,j}$ is **BLACK**, then each edge $e_k = (v_{i,j}, v_{i',j'})$ or $(v_{i',j'}, v_{i,j})$ must be **ON** or **OFF**. On the other hand, if $e_k = (v_{i,j}, v_{i',j'})$ is **ON**, then $v_{i,j}$ must be **BLACK**.
- A vertex is **GRAY** if its vertex entering time can still be influenced by Type-3 edges. If $v_{i,j}$ is **GRAY**, then for each Type-1 or Type-2 edge $e_k = (v_{i',j'}, v_{i,j})$, $v_{i',j'}$ must be **BLACK**. When we remove edges, we only estimate the cost of an edge $e_k = (v_{i,j}, v_{i',j'})$, where at least one of $v_{i,j}$ and $v_{i',j'}$ is **GRAY**.
- A vertex is **WHITE** if its vertex entering time can be influenced by any type of edges.

Vertex Slack. The vertex slack is the maximum time which can be delayed at the vertex without increasing the objective. We consider **ON** edges only. Similar to the computation of the vertex entering time, if G' is acyclic, we follow a reverse topological order and compute the vertex slack of each vertex $v_{i,j}$ as follows:

- If Ξ_j is the last conflict zone on the trajectory of Δ_i,

$$slack\ [v_{i,j}] = \min \left\{ \max_{v_{i',j'} \in G'} (s_{i',j'} + p_{i',j'}) - (s_{i,j} + p_{i,j}), \min_{k | e_k = (v_{i,j}, v_{i',j'}) \in G'} \{slack\ [v_{i',j'}]\} \right\}. \tag{4}$$

- Otherwise,

$$slack\ [v_{i,j}] = \min_{k | e_k = (v_{i,j}, v_{i',j'}) \in G'} \{slack\ [v_{i',j'}]\}. \tag{5}$$

Edge Cost. The edge cost of a Type-3 edge is the delay time of the objective caused by this edge if we keep it. For the edge $e_k = (v_{i,j}, v_{i',j})$, $(s_{i,j} + p_{i,j} + w_k)$ and $s_{i',j}$ are the vertex entering times of $v_{i',j}$ with and without considering e_k, respectively. If the delay time caused by e_k is larger than the slack of $v_{i',j}$, the objective will increase if we keep e_k. The edge cost of a Type-3 edge $e_k = (v_{i,j}, v_{i',j})$ is defined as follows:

$$cost\ [e_k] = (s_{i,j} + p_{i,j} + w_k) - s_{i',j} - slack\ [v_{i',j}]. \tag{6}$$

Note that, although the edge cost may be negative, the objective will never decrease.

4.4 Cycle-Removal-Based Scheduling

To solve the cycle removal problem, we follow the steps listed in Algorithm 1. First, based on the problem formulation in Section 2, Type-1 and Type-2 edges must be included in G'. Thus, we set the states of all Type-1 and Type-2 edges to **ON** and the states of all Type-3 edges to **UNDECIDED**.

Next, we apply Algorithm 2 to compute the vertex entering times and slacks of vertices. At this moment, the graph G' contains only Type-1 and Type-2 edges and thus is an acyclic graph. According to its topological order, we compute the vertex entering time of each vertex by Equations (1) and (2) and the leaving time of the last vehicle. We also compute the slack of each vertex according to reverse topological order by Equations (4) and (5).

Then, we decide which edges to be removed by Algorithm 3. First, in the process of *Find-Leaders*, a vertex $v_{i,j}$ is identified as a *leader vertex* if Δ_i is the first vehicle of its source lane and Ξ_j the first conflict zone on the trajectory of Δ_i. Second, an **UNDECIDED** edge $e_k = (v_{i,j}, v_{i',j})$, *i.e.*, a Type-3 edge, is identified as a *candidate edge* if $v_{i,j}$ or $v_{i',j}$ is a leader vertex. Third, we compute the edge

ALGORITHM 1: Cycle-Removal-Based Scheduling

Input: G
Output: G'
Initialization;
for each vertex $v_{i,j} \in V$ **do**
  $state[v_{i,j}] \leftarrow$ WHITE;
  $slack[v_{i,j}] \leftarrow \infty$;
end
for each edge $e_k \in E$ **do**
  if e_k is a *Type-3 edge* **then**
    $state[e_k] \leftarrow$ UNDECIDED;
  else
    $state[e_k] \leftarrow$ ON;
  end
end
Update-Time-Slack (G);
Remove-Type-3-Edges (G, 0, m);
Output the resultant graph as G';

ALGORITHM 2: Update-Time-Slack

Input: G
Initialization;
Topological-Sort (G);
for each vertex $v_{i,j}$ in topological order **do**
  $s_{i,j} \leftarrow a_i$;
  for each edge $e_k = (v_{i',j'}, v_{i,j}) \in E$ **do**
    Compute $s_{i,j}$ by Equation (1) or (2);
  end
end
$maxLeavingTime \leftarrow \max_{v_{i,j}}(s_{i,j} + p_{i,j})$;
for each vertex $v_{i,j}$ in reverse topological order **do**
  $slack[v_{i,j}] \leftarrow maxLeavingTime - s_{i,j} - p_{i,j}$;
  for each edge $e_k = (v_{i,j}, v_{i',j'}) \in E$ **do**
    $slack[v_{i,j}] \leftarrow \min\{slack[v_{i,j}], slack[v_{i',j'}]\}$;
  end
end

cost of each candidate edge by Equations (6). Fourth, we try to remove Type-3 edges in descending order of edge cost. Removing edge $e_k = (v_{i,j}, v_{i',j})$ means its reverse edge $e_{k'} = (v_{i',j}, v_{i,j})$ must be included in G' and cannot be removed. As a result, we temporarily set the state of e_k to **OFF** and $e_{k'}$ to **ON**. Then, we verify deadlock-freeness for the current G' by the verification approaches in Section 3. If G' is not deadlock-free, we recover e_k and remove $e_{k'}$ by exchanging their states and verify deadlock-freeness for G' again. If G' is deadlock-free after we decide the states of e_k and $e_{k'}$, we update the states of related vertices, identify newly set **GRAY** vertices as *leader vertices*, and recompute vertex entering times and slacks. Then, we perform the same process to the next highest cost edge.

ALGORITHM 3: Remove-Type-3-Edges

Input: G, i_{start}, i_{end}
Initialization;
for each vertex $v_{i,j} \in V$ **do**
  if $order[\Delta_i] \geq i_{start}$ **then**
    $state[v_{i,j}] \leftarrow$ WHITE;
  end
end
for each *Type-3 edge* $e_k = (v_{i,j}, v_{i',j}) \in E$ **do**
  case *1:* $order[\Delta_i], order[\Delta_{i'}] < i_{start}$ **do**
    do nothing;
  case *2:* $i_{end} \leq order[\Delta_i], order[\Delta_{i'}]$ **do**
    $state[e_k] \leftarrow$ DONTCARE;
  case *3:* $i_{start} \leq order[\Delta_i], order[\Delta_{i'}] < i_{end}$ **do**
    $state[e_k] \leftarrow$ UNDECIDED;
  case *4:* $i_{start} \leq order[\Delta_i] < i_{end} \leq order[\Delta_{i'}]$ **do**
    $state[e_k] \leftarrow$ ON;
  case *5:* $i_{start} \leq order[\Delta_{i'}] < i_{end} \leq order[\Delta_i]$ **do**
    $state[e_k] \leftarrow$ OFF;
end
$f_{fail} \leftarrow$ FALSE;
$LeaderVertices \leftarrow$ Find-Leaders (i_{start}, i_{end});
while $LeaderVertices \neq \emptyset$ **do**
  $CandidateEdges \leftarrow$ Find-Candidates ($LeaderVertices$);
  for each edge $e_k = (v_{i,j}, v_{i',j})$ *in* $CandidateEdges$ **do**
    $cost[e_k] \leftarrow s_{i,j} + p_{i,j} + w_k - s_{i',j} - slack[v_{i',j}]$;
  end
  $e_{max} \leftarrow$ Find-Max-Cost-Edge ($CandidateEdges$);
  $e_{max'} \leftarrow (v_{i',j}, v_{i,j})$ when $e_{max} = (v_{i,j}, v_{i',j})$;
  $state[e_{max}] \leftarrow$ OFF;
  $state[e_{max'}] \leftarrow$ ON;
  if VerifyGraph (G) is FALSE **then**
    $state[e_{max}] \leftarrow$ ON;
    $state[e_{max'}] \leftarrow$ OFF;
    if VerifyGraph (G) is FALSE **then**
      $f_{fail} \leftarrow$ TRUE;
      break;
    end
  end
  $LeaderVertices \leftarrow$ Update-Leaders ($LeaderVertices$);
  Update-Time-Slack (G);
end
if f_{fail} is TRUE **then**
  $i_{mid} \leftarrow \frac{1}{2}(i_{start} + i_{end})$;
  Remove-Type-3-Edges (G, i_{start}, i_{mid});
  Remove-Type-3-Edges (G, i_{mid}, i_{end});
end

However, sometimes G' may have a deadlock no matter we remove either e_k or $e_{k'}$. The reason is that the previous assignments of edges conflict with the decision of choosing e_k or $e_{k'}$. Backtracking the already removed edges is a solution for resolving the dilemma. Unfortunately, the backtracking suffers from a long runtime of finding a valid assignment. Therefore, instead of backtracking the removed edges, we divide the original problem to sub-problems. We partition all vehicles into two parts according to the ascending order of their earliest arrival times. The first part contains vehicles ordered before i_{end}, the second part contains the rest. Consider each pair of vehicles Δ_i and $\Delta_{i'}$, where Δ_i is in the first part, while $\Delta_{i'}$ the second. If Ξ_j is a common conflict zone on the trajectories of Δ_i and $\Delta_{i'}$, we assume that Δ_i will pass zone Ξ_j before $\Delta_{i'}$. The assumption implies the state of edge $(v_{i,j}, v_{i',j})$ is **ON** and the state of edge $(v_{i',j}, v_{i,j})$ is **OFF**. Therefore, when solving the sub-problem associated with the first part, we consider only Type-3 edges in between two vehicles belonging to the first part. For both Δ_i and $\Delta_{i'}$ in the first part, the state of their Type-3 edge $(v_{i,j}, v_{i',j})$ is set to **UNDECIDED** (to be decided in the current sub-problem); for both Δ_i and $\Delta_{i'}$ in the second part, the state of their Type-3 edge $(v_{i,j}, v_{i',j})$ is set to **DONTCARE** (ignored in the current sub-problem). After we have solved the sub-problem associated with the first part, we turn to the sub-problem associated with the second part based on the result derived in previously solved sub-problems. We set the Type-3 edges within the second part to **UNDECIDED** and keep those in the first part unchanged. These procedure is repeated until there are no **UNDECIDED** edges and all the vertices are **BLACK**. Finally, we obtain an acyclic graph G' and schedule the vertex entering time of each vertex in G' by Equations (1) and (2).

THEOREM 4.1. *Our scheduling algorithm always finds a feasible solution.*

PROOF. A feasible solution should satisfy items (i) and (ii) in the problem formulation. Type-1 and Type-2 edges must be included in G', and they do not generate cycles or deadlocks. For Type-3 edges between any pair of $v_{i,j}$ and $v_{i',j}$, only one edge (either $e_k = (v_{i,j}, v_{i',j})$ or $e_{k'} = (v_{i',j}, v_{i,j})$) is selected by our algorithm. If we cannot determine all Type-3 edges at a time, the original problem of all vehicles is recursively divided into two sub-problems according to the ascending order of their earliest arrival times. For Type-3 edges in between two parts, we select only Type-3 edges from the first part to the second part. Hence, only Type-3 edges within one sub-problem have to be discussed. Every time, including a Type-3 edge in one sub-problem is verified by our verification approaches in Section 3 to guarantee cycle-freeness and deadlock-freeness. In the worst case of sub-problem division, each sub-problem solves only one vehicle. In this case, no Type-3 edges exist in between two vertices belonging to the same vehicle. As a result, the resultant G' is guaranteed to be acyclic and deadlock-free. □

THEOREM 4.2. *The time complexity of our scheduling algorithm is* $O(E^2 \log V)$.

PROOF. Our scheduling algorithm (Algorithm 1) contains three parts: vertex and edge state initialization, updating vertex entering times and slacks (Algorithm 2), and Type-3 edge removal (Algorithm 3). Vertex/edge state initialization can be done by graph traversal in $O(V + E)$ time. Vertex entering times and slacks can be computed in $O(V + E)$ time based on topological sort and graph traversal. For Type-3 edge removal, assume the induced subgraph for a sub-problem covers a vertex subset $V_s \subseteq V$ and an edge subset $E_s \subseteq E$. The running time of each sub-problem is dominated by the while loop and sub-problem division in Algorithm 3. The while loop examines each Type-3 edge at most once, and the verifier takes $O(V + E)$ time. Thus, the while loop takes a total of $O(E_s(V + E))$ time. The recurrence for the running time $T(V_s, E_s, V, E)$ of Algorithm 3 can be written as $T(V_s, E_s, V, E) = T(\frac{V_s}{2}, \alpha E_s, V, E) + T(\frac{V_s}{2}, \beta E_s, V, E) + O(E_s(V + E))$, where $\alpha + \beta \leq 1$. In the base case, every sub-problem contains only one vehicle, and $T(1, E_s, V, E)$ takes $O(E)$ time. The

overall running time $T(V, E, V, E)$ of Algorithm 3 is $O(VE + E(V + E) \log V)$. Therefore, our scheduling algorithm takes $O(E^2 \log V)$ time. □

In practical cases, the number of vehicles near an intersection is less than 100, and the experimental results will show the efficiency applicable in real time.

5 EXPERIMENTAL RESULTS

We implemented both of the verification approaches and scheduling algorithms in the C++ programming language. The experiments were run on a macOS mojave notebook with 2.3 GHz Intel CPU and 8 GB memory. The traffic is generated at every source lane by Poisson distribution where the parameter of Poisson distribution λ is set to as 0.1, 0.3, 0.5, 0.6, and 0.7. The higher λ, the higher traffic density. When $\lambda = 0.1$, the average time interval between two incoming vehicles is 10 seconds, while it is 2 seconds when $\lambda = 0.5$. The respective edge waiting time of a Type-1 edge, Type-2 edge, and Type-3 edge is 0.1, 0.2, and 0.2, respectively. The minimum time for a vehicle to pass a conflict zone is set to 1 second, which means a vehicle takes 1 second to pass a conflict zone without considering other vehicles.

5.1 Scheduling Effectiveness and Efficiency

In the first experiment, a four-way intersection is considered. For each direction, there is only one incoming lane and one outgoing lane. Four conflict zones are generated according to the crossing locations of four incoming lanes. Two traffic settings are generated. In the first setting, the earliest arrival time of the last vehicle is 30 seconds, meaning that the intersection manager is required to have a communication range covering all vehicles that will arrive in 30 seconds. In the second setting, the earliest arrival time of the last vehicle is 60 seconds. For each vehicle, the probability of going straight, taking a right turn, or taking a left turn is generated by a uniform distribution.

As listed in Tables 2 and 3, the proposed scheduling algorithm is compared with three approaches: (1) 3D-Intersection, (2) First-Come-First-Serve, and (3) Priority-Based. In the 3D-Intersection approach, vehicles do not consider the conflicts with vehicles on other lanes so that a vehicle is delayed only by vehicles on the same lane and in front of it. Thus, the 3D-Intersection approach provides a lower bound for the objective (T_L), although it may not be collision-free. The First-Come-First-Serve approach was introduced in Section 4.1. The distributed priority-based approach in [18] is modified to fit in our graph-based model and problem formulation. The Priority-Based approach iteratively decides the passing order of vehicles by their priorities, and the priorities may change after each iteration. In our experiment, for every 1.0 second, the priorities are updated according to the newly estimated earliest arrival times to intersection.

All approaches are evaluated by two criteria: (1) the leaving time of the last vehicle T_L and (2) the average delay time of all vehicles T_D. T_L is equivalent to the total time needed for all vehicles to go through the intersection. On the other hand, since the 3D-Intersection approach provides the lower bound of T_L, the average delay time of all vehicles T_D is computed as the average of the difference between each vehicle's leaving time and its leaving time in the 3D-Intersection solution. The average delay time of the 3D-Intersection approach itself is always 0.

We demonstrate the effectiveness and efficiency of our algorithm by changing (1) the traffic density and (2) the communication range.

Different Traffic Densities. Table 2 shows the impact of traffic density on scheduling, and Figure 9 shows the corresponding bar charts. Note that 3D-Intersection provides the lower bounds of T_L and T_D. When λ is 0.1, all approaches can achieve the optimal solution on T_L and T_D due to low traffic density. However, when λ becomes higher, the T_L and T_D of the First-Come-First-Serve approach increase rapidly, and our algorithm can always achieve better results than the

Table 2. Results Under Different λ When the Earliest Arrival Time of the Last Vehicle is 30 seconds, Where T_L, T_D, and RT are the Leaving Time of the Last Vehicle, the Average Delay Time of All Vehicles, and the Runtime, Respectively (all units are in second)

		3D-Intersection			First-Come-First-Serve			Priority-Based			Ours		
λ	m	T_L	T_D	RT	T_L	T_D	RT	T_L	T_D	RT	T_L	T_D	RT
0.1	11	33.40	0	0.001	33.40	0.00	0.003	33.40	0.00	0.009	33.40	0.00	0.002
0.3	34	40.70	0	0.003	50.70	5.85	0.005	44.50	3.17	0.007	40.80	2.23	0.015
0.5	58	42.40	0	0.006	82.40	19.58	0.009	68.20	10.62	0.013	60.40	6.91	0.057
0.6	66	40.50	0	0.009	90.39	24.65	0.011	70.10	12.31	0.020	68.70	13.65	0.119
0.7	77	46.10	0	0.010	90.20	23.68	0.013	74.90	13.44	0.024	72.80	13.46	0.174

Table 3. Results Under Different λ When the Earliest Arrival Time of the Last Vehicle is 60 seconds, Where T_L, T_D, and RT are the Leaving Time of the Last Vehicle, the Average Delay Time of All Vehicles, and the Runtime, Respectively (all units are in second)

		3D-Intersection			First-Come-First-Serve			Priority-Based			Ours		
λ	m	T_L	T_D	RT	T_L	T_D	RT	T_L	T_D	RT	T_L	T_D	RT
0.1	25	66.30	0	0.002	68.80	0.48	0.005	68.80	0.48	0.008	66.90	0.32	0.006
0.3	66	68.80	0	0.009	89.19	10.84	0.013	73.50	2.36	0.015	71.10	1.78	0.070
0.5	104	74.00	0	0.015	131.10	26.75	0.020	105.30	12.30	0.052	98.40	11.80	0.229
0.6	129	71.50	0	0.026	149.20	37.62	0.033	133.00	27.64	0.091	116.90	20.77	0.626
0.7	157	72.90	0	0.039	176.50	54.67	0.049	157.80	38.49	0.157	139.50	34.22	1.825

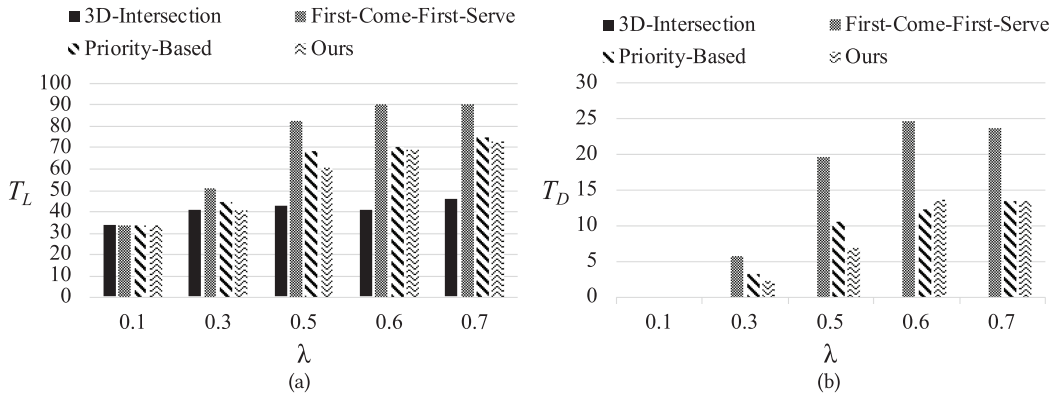

Fig. 9. Comparison of (a) T_L and (b) T_D under different λ when the earliest arrival time of the last vehicle is 30 seconds.

First-Come-First-Serve approach. This is because our algorithm considers more vehicles and their interactions, *i.e.*, a global view, and provides a systematic approach to optimize the objective. Only few cases, e.g., $\lambda = 0.6$ or 0.7 when the earliest arrival time of the last vehicle is 30 seconds, the priority-based approach achieves better T_D than ours. The main reason is that its frequent updates (every 1.0 second) on the earliest arrival times can sometimes mend the lack of a global view. If the update is not fast enough, its effectiveness will decline.

Different Communication Ranges. The communication range of an intersection manager is an important factor. To show the flexibility of communication ranges of our algorithm, we compare

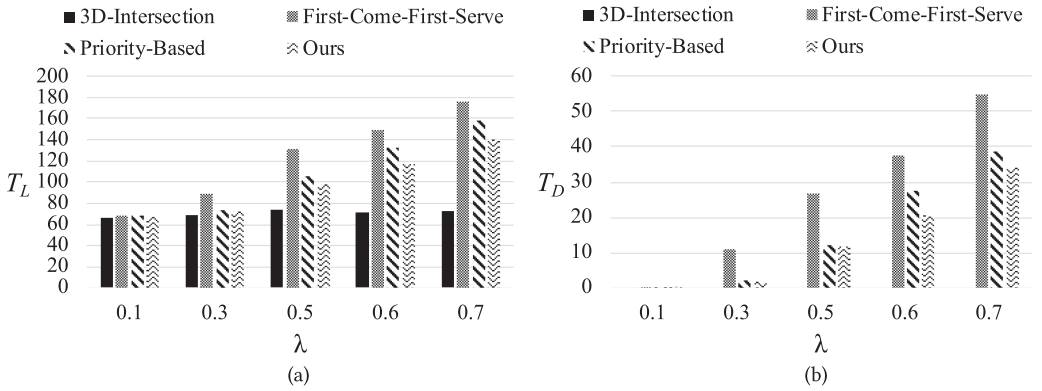

Fig. 10. Comparison of (a) T_L and (b) T_D under different λ when the earliest arrival time of the last vehicle is 60 seconds.

Table 4. Verification Runtimes When the Earliest Arrival Time of the Last Vehicle is 60 seconds (all units are in second)

λ	m	Graph-Based Verification	Petri-Net-Based Verification
0.1	25	6.00e-06	1.50e-05
0.3	66	2.80e-05	3.60e-05
0.5	104	1.30e-04	1.30e-04
0.6	129	1.50e-04	1.80e-04
0.7	157	3.10e-04	3.10e-04

Table 2 with Table 3 (also Figure 9 with Figure 10) to observe the results generated by different communication ranges under same λ. In Table 2, the communication range of the intersection manager covers all vehicles that will arrive in 30 seconds. In Table 3, the communication range of the intersection manager covers all vehicles that will arrive in 60 seconds. As the communication range becomes twice larger, the T_L of our algorithm also becomes approximately twice larger, which means different communication ranges do not affect the solution quality of our algorithm.

Overall, the proposed scheduling algorithm always achieves better solutions than the First-Come-First-Serve approach under different scenarios. Our algorithm is sufficiently efficient for real-time use even when the number of vehicles reaches 100, which can be completed in around 1 second.[7] As the number of vehicles exceeds 100, the runtime grows up. However, the number of vehicles in an intersection will not exceed 100 in most cases. Even if the number of vehicles is large, we can still split the traffic and schedule the front vehicles first because it is impossible for 100 vehicles to go through the intersection in 1 second. Besides, Table 4 shows the runtimes of two verification approaches when the communication range of the intersection manager covers all vehicles that will arrive in 60 seconds. In general, both approaches do not play a major role in the overall runtime, and the graph-based verification is faster than the Petri-net-based verification. This is because the graph-based verification matches the currenct intersection model better, but we believe that the Petri-net-based verification can potentially be integrated with existing verification methodologies and tools.

[7]It is believed that an intersection manager has much better computational capability than a current vehicle.

Table 5. Results of the Proposed Algorithm Under Different Numbers of Conflict Zones, Where T_L, T_D, and RT are the Leaving Time of the Last Vehicle, the Average Delay Time of All Vehicles, and the Runtime, Respectively (all units are in second)

		1 Conflict Zone			4 Conflict Zones			16 Conflict Zones		
λ	m	T_L	T_D	RT	T_L	T_D	RT	T_L	T_D	RT
0.1	11	33.40	0.09	0.004	33.40	0.00	0.002	34.50	0.72	0.004
0.3	34	50.30	6.29	0.016	40.80	2.23	0.014	44.20	3.05	0.020
0.5	58	77.70	16.87	0.092	60.40	6.91	0.057	51.40	5.13	0.103
0.6	66	89.00	24.03	0.188	68.70	13.65	0.119	55.80	6.34	0.134
0.7	77	100.70	28.84	0.284	72.80	13.46	0.174	64.20	9.37	0.769

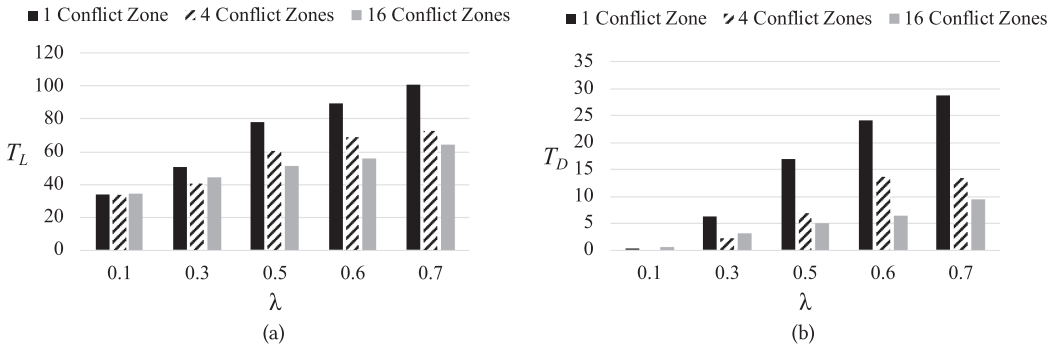

Fig. 11. Comparison of (a) T_L and (b) T_D under different number of conflict zones.

5.2 Modeling Expressiveness

In the second experiment, we show the expressiveness and generality of our modeling for different granularities of an intersection. The four-way intersection is modeled by 1 (like the previous work [18]), 4, and 16 conflict zone(s) as shown in Figure 2. As shown in Table 5 and Figure 11, when λ is low, different granularities of an intersection lead to near-optimal solutions because of few conflicts between vehicles. However, when λ becomes higher, the intersection modeled by 4 conflict zones always has better solutions than that modeled by 1 conflict zone. Similarly, intersection modeled by 16 conflict zones has better solution than those modeled by 1 and 4 conflict zone(s) in most cases. The results are consistent with the expectation as discussed in Figure 4. The finer granularity of an intersection, the more delicate intersection modeling and solution space, and thus the better scheduling results. It should be mentioned that we provide general modeling, scheduling, and verification for intersection management, and they can further assist intersection designers (*i.e.*, governments or city planners) to design intersections (*e.g.*, the number of conflict zones, the passing speed, the safety gap, the communication range, etc.).

6 CONCLUSION

In this paper, we propose a graph-based model for intersection management. The model is very general and applicable to different granularities of intersections and other conflicting scenarios. We derived formal verification approaches which can guarantee deadlock-freeness. Based on the graph-based model and the verification approaches, we developed a cycle removal algorithm to schedule vehicles to go through the intersection safely (without collisions) and efficiently without deadlocks. The algorithm is sufficiently efficient to consider more conflict zones and more vehicles

in real time. Experimental results demonstrated the expressiveness of the proposed model and the effectiveness and efficiency of the proposed algorithm.

ACKNOWLEDGMENT

The authors gratefully acknowledge the support from Ministry of Education (MOE) in Taiwan (Grant Numbers: NTU-107V0901 and NTU-108V0901), Ministry of Science and Technology (MOST) in Taiwan (Grant Numbers: MOST-108-2636-E-002-011, MOST-106-2628-E-002-019-MY3, and MOST-108-2221-E-002-099-MY3), MediaTek Inc. (Grant Numbers: MTKC-2018-0167 and MTKC-2019-0070), Global Unichip Corp., Synopsys Inc., and TSMC Ltd.

REFERENCES

[1] M. Ahmane, A. Abbas-Turki, F. Perronnet, J. Wu, A. El Moudni, J. Buisson, and R. Zeo. 2013. Modeling and controlling an isolated urban intersection based on cooperative vehicles. *Transportation Research Part C: Emerging Technologies* 28 (2013), 44–62.

[2] H. Ahn and D. Del Vecchio. 2016. Semi-autonomous intersection collision avoidance through job-shop scheduling. In *ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*. 185–194.

[3] S. R. Azimi, G. Bhatia, R. R. Rajkumar, and P. Mudalige. 2013. Reliable intersection protocols using vehicular networks. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. 1–10.

[4] S. R. Azimi, G. Bhatia, R. R. Rajkumar, and P. Mudalige. 2014. STIP: Spatio-temporal intersection protocols for autonomous vehicles. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. 1–12.

[5] L. Chen and C. Englund. 2016. Cooperative intersection management: A survey. *IEEE Transactions on Intelligent Transportation Systems* 17, 2 (2016), 570–586.

[6] E. G. Coffman, M. Elphick, and A. Shoshani. 1971. System deadlocks. *Comput. Surveys* 3, 2 (1971), 67–78.

[7] A. Colombo and D. Del Vecchio. 2015. Least restrictive supervisors for intersection collision avoidance: A scheduling approach. *IEEE Trans. Automat. Control* 60, 6 (2015), 1515–1527.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[9] E. Dallal, A. Colombo, D. Del Vecchio, and S. Lafortune. 2013. Supervisory control for collision avoidance in vehicular networks using discrete event abstractions. In *American Control Conference*. 4380–4386.

[10] N. J. Dingle, W. J. Knottenbelt, and T. Suto. 2009. PIPE2: A tool for the performance evaluation of generalised stochastic Petri nets. *ACM Performance Evaluation Review* 36, 4 (2009), 34–39.

[11] K. Dresner and P. Stone. 2008. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research* 31 (2008), 591–656.

[12] Q. Jin, G. Wu, K. Boriboonsomsin, and M. Barth. 2012. Advanced intersection management for connected vehicles using a multi-agent systems approach. In *IEEE Intelligent Vehicles Symposium*. 932–937.

[13] V. Kann. 1992. On the approximability of NP-complete optimization problems. *Ph.D. Thesis, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm* (1992).

[14] R. M. Karp. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations. The IBM Research Symposia Series*. 85–103.

[15] J. Kleinberg and E. Tardos. 2006. *Algorithm Design*. Pearson, Addison Wesley.

[16] H. Kowshik, D. Caveney, and P. R. Kumar. 2011. Provable systemwide safety in intelligent intersections. *IEEE Transactions on Vehicular Technology* 60, 3 (2011), 804–818.

[17] J. B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *American Mathematical Society* 7, 1 (1956), 48–50.

[18] C. Liu, C. Lin, S. Shiraishi, and M. Tomizuka. 2018. Distributed conflict resolution for connected autonomous vehicles. *IEEE Transactions on Intelligent Vehicles* 3, 1 (2018), 18–29.

[19] D. A. Menasce and R. R. Muntz. 1979. Locking and deadlock detection in distributed data bases. *IEEE Transactions on Software Engineering* SE-5, 3 (1979), 195–202.

[20] R. Naumann, R. Rasche, J. Tacken, and C. Tahedi. 1997. Validation and simulation of a decentralized intersection collision avoidance algorithm. In *International Conference on Intelligent Transportation Systems*. 818–823.

[21] J. L. Peterson. 1977. Petri nets. *Comput. Surveys* 9, 3 (1977).

[22] S. Reveliotis and E. Roszkowska. 2011. Conflict resolution in free-ranging multivehicle systems: a resource allocation paradigm. *IEEE Transactions on Robotics* 27, 2 (2011), 283–296.

[23] M. Singhal. 1989. Deadlock detection in distributed systems. *Computer* 22, 11 (1989), 37–48.

[24] J. Wu, F. Perronnet, and A. Abbas-Turki. 2014. Cooperative vehicle-actuator system: A sequence-based framework of cooperative intersections management. *IET Intelligent Transport Systems* 8, 4 (2014), 352–360.
[25] D. H. Younger. 1963. Minimum feedback arc sets for a directed graph. *IEEE Transactions on Circuit Theory* 10, 2 (1963), 238–245.
[26] B. Zheng, C. Lin, H. Liang, S. Shiraishi, W. Li, and Q. Zhu. 2017. Delay-aware design, analysis and verification of intelligent intersection management. In *IEEE International Conference on Smart Computing (SMARTCOMP)*. 1–8.
[27] F. Zhu and S. V. Ukkusuri. 2015. A linear programming formulation for autonomous intersection control within a dynamic traffic assignment and connected vehicle environment. *Transportation Research Part C: Emerging Technologies* 55 (2015), 363–378.

Received April 2019; revised June 2019; accepted July 2019