ILS-YOLO: An Improved List Scheduling Algorithm for YoloLite DAGs in Vehicular Edge Computing

Xianglin Xiao
Department of information engineering Sichuan Vocational and Technical College of Communication, Sichuan
E-mail: sndmhk3878263@outlook.com

Keywords: vehicular edge computing (VEC), YoloLite, real-time vehicle detection, task offloading, DAG scheduling, Q-learning, low latency

Received: September 17, 2025

Real-time vehicle detection is critical for intelligent transportation systems, yet deploying lightweight models like YoloLite on resource-constrained vehicular units leads to prohibitive latency. To address this challenge, this paper introduces ILS-YOLO, an Improved List Scheduling algorithm for YoloLite Directed Acyclic Graphs (DAGs) in Vehicular Edge Computing (VEC). The algorithm models the YoloLite inference pipeline as a DAG of sub-tasks and introduces two key innovations: a Unified Scheduling Table to mitigate resource fragmentation on multi-core edge servers and a Transfer Scheduling Table to accurately model link contention and communication overhead. Performance is evaluated through extensive simulations using synthetic DAGs with varying structures (up to 200 nodes) and Communication-to-Computation Ratios (CCRs), alongside a case study using parameters derived from a complex perception task and real-world traffic data. Results show that ILS-YOLO significantly reduces end-to-end detection latency, achieving a speedup of up to 2.5× over baseline scheduling heuristics for large, computation-intensive DAGs (n=200, low CCR). This work presents a robust and efficient scheduling solution that makes low-latency, high-accuracy vehicle detection feasible in practical VEC environments.

Povzetek: Študija predstavi algoritem ILS-YOLO, ki z izboljšanim razporejanjem podopravil v VEC-okolju močno zmanjša zakasnitev pri YoloLite zaznavi vozil in doseže do 2,5-kratno pohitritev glede na klasične heuristike.

1 Introduction

Automobile edge detection systems have gained prominence with the rapid development of intelligent transportation systems and autonomous driving technologies [1]. These systems demand real-time and precise vehicle detection to enable safe and efficient traffic management and autonomous navigation. However, the computational requirements of advanced detection algorithms often surpass the capabilities of vehicle onboard units. Vehicular Edge Computing (VEC) emerges as a solution, allowing vehicles to offload computation-intensive tasks to nearby edge servers. This paradigm enhances computational power and reduces latency, making real-time vehicle detection feasible. Recent works like Sihai Tang et al. and Yu-Jen Ku et al. have demonstrated the potential of VEC in improving multi-vehicle perception and minimizing end-to-end delay [2,3]. A review of traditional edge detection methods reveals their applications in various fields including autonomous vehicles, where edge detection and segmentation are crucial for detecting motion and tracking video.

Traditional vehicle detection methods primarily relied on local processing within the vehicle's onboard unit. While this approach ensures data privacy and offline functionality, it struggles with the computational demands of modern deep learning models. Studies such as Li et al. highlight the limitations of traditional methods in achieving real-time performance [4]. Local processing often results in significant latency, especially during peak traffic periods or complex traffic scenarios, hindering the system's ability to provide timely and accurate detection results. Traditional edge detection algorithms generally adopt differential methods for calculation. Common first-order differential edge detection operators include Robert, Sobel, Prewitt, and Kirsch operators, which are simple and fast in calculation but less accurate in positioning [5]. Second-order differential operators, such as Canny, Log, and Laplacian operators, locate edges accurately but are sensitive to noise [6]. For noise-polluted images, filtering is usually performed before differential edge detection to minimize noise impact, but this can also blur image edges to some extent. The Marr operator combines noise filtering with edge extraction, but it has poor noise resistance when the template is small and is time-consuming when the template is large [7].

Current optimization strategies in vehicle detection technology have been advanced through various approaches. Satzoda and Trivedi focused on the dynamics of rear lights for vehicle detection at night, improving detection accuracy under low-light conditions

[8]. Yan et al. combined HOG with AdaBoost classification to achieve real-time vehicle detection, enhancing the speed and accuracy of the detection process [9]. Concurrently, advanced deep reinforcement learning techniques, such as attention-augmented multi-agent algorithms, are being employed to manage complex resource allocation in NOMA-based vehicular systems [10]. Furthermore, these optimization strategies are being extended to emerging collaborative scenarios, such as UAV-assisted vehicle platooning, to ensure efficient task processing and resource management across multiple platforms [11]. These strategies collectively contribute to more efficient, accurate, and real-time vehicle detection systems.

This study addresses these challenges by proposing the Improved List Scheduling for YoloLite DAGs (ILS-YOLO) algorithm. The ILS-YOLO algorithm models the YoloLite inference process as a Directed Acyclic Graph (DAG) and introduces a Unified Scheduling Table and a Transfer Scheduling Table. These innovations aim to optimize resource utilization and communication efficiency, minimizing end-to-end detection latency. Unlike existing methods that often treat applications as atomic tasks, our approach provides a more fine-grained and efficient scheduling strategy by considering the pipeline's dependencies and workload characteristics.

The remainder of this paper is organized as follows: Section 2 describes the problem and formulates the mathematical model. Section 3 details the proposed ILS-YOLO algorithm. Section 4 presents the numerical experiments and discusses the results. Finally, Section 5 concludes the paper and suggests directions for future research.

.

2 Related work

2.1 Task offloading in vehicular networks

Task offloading is a cornerstone of Vehicular Edge Computing (VEC), enabling resource-constrained vehicles to execute computationally intensive applications by migrating them to more powerful edge servers. The primary goal of task offloading is to optimize one or more performance metrics, such as latency, energy consumption, or computational cost. The research landscape for task offloading is diverse, addressing various scenarios from simple single-user, single-server setups to complex multi-user, multi-server environments with cloud integration.

Early research often focused on single-edge-server scenarios. In these models, a group of vehicles communicates with a single Roadside Unit (RSU). Mao et al. [12] utilized Lyapunov optimization to minimize a weighted sum of processing delay and task failure costs under energy constraints. Wang et al. [13] proposed an online task scheduling method for vehicular edge computing based on imitation learning to achieve near-optimal performance. To address the trade-off between latency and energy consumption, researchers have widely applied Deep Reinforcement Learning (DRL). Zhan et al. [14] utilized DRL to optimize offloading scheduling in vehicular edge computing, while Zhou and Hu [15] focused on maximizing computation efficiency in wireless-powered mobile edge computing networks. To better capture the dynamic characteristics of the vehicular environment, Liu et al. [16] developed a dependency-aware task scheduling model to handle complex tasks with interdependencies.

Furthermore, some researchers have investigated three-tier architectures that integrate a remote cloud layer with the edge-vehicle layers. This cloud-edge-end collaborative paradigm allows tasks to be offloaded to either the edge for low latency or the cloud for massive computational power. For example, Wu et al. [17] proposed an efficient task scheduling algorithm for servers with dynamic states in vehicular edge computing. Li et al. [18] introduced a multi-objective-oriented task scheduling method to balance multiple performance metrics in heterogeneous networks. Distributed approaches have also been explored to enhance scalability, with Tang and Wong [19] introducing a distributed DRL algorithm that allows each mobile device to make decentralized decisions based on local observations.

Furthermore, some researchers have investigated three-tier architectures that integrate a remote cloud layer with the edge and vehicle layers. Geng et al. [20] developed a DRL-based distributed computation offloading framework to leverage the advantages of collaborative edge-cloud computing. A summary of representative works is presented in Table 1.

While these works have laid a strong foundation for VEC task offloading, many treat applications as atomic, indivisible tasks. Our work differs by specifically addressing applications like real-time object detection, which can be decomposed into a pipeline of dependent sub-tasks. By modeling this pipeline as a Directed Acyclic Graph (DAG), we can achieve a more fine-grained and efficient scheduling of computational resources to minimize end-to-end latency.

Table 1: Summary of optimization objectives, constraints, and proposed solutions

Literature	Optimization Objective	Constraints	Proposed Solution
[1]	Weighted sum of processing delay and task failure cost	Battery output power	Method based on Lyapunov optimization
[2]	Energy consumption	Cache task size	Method combining block coordinate descent and convex optimization
[3]	Weighted sum of processing	Maximum task	Method based on Q-learning and DQN

	delay and energy consumption	processing deadline	
[4]	Energy consumption	Maximum task processing deadline	Method based on Q-learning and DDQN
[5]	Task processing latency	Maximum task processing deadline	Method combining RNN with deep reinforcement learning
[6]	Weighted sum of processing delay and energy consumption	Maximum task processing deadline	End-to-end deep reinforcement learning model
[7]	Weighted sum of processing delay and energy consumption	Maximum task processing deadline	Algorithm based on Q-learning
[8]	Weighted sum of processing delay and task discard penalty	Queue state of device and edge node	Algorithm combining LSTM with Dueling DDQN
[9]	Weighted sum of processing delay and energy consumption	Maximum task completion deadline, device power limit	Algorithm combining LSTM with deep reinforcement learning

2.2 Dag-based task scheduling in heterogeneous computing environments (review of heft, peft, etc.)

Scheduling tasks with dependencies, modeled as a Directed Acyclic Graph (DAG), onto a set of heterogeneous processors is a well-known NP-hard problem [11]. Consequently, a plethora of heuristic and meta-heuristic algorithms have been developed to find near-optimal solutions in polynomial time. These algorithms are crucial for applications like the YoloLite pipeline, where sub-tasks have precedence constraints.

List scheduling is one of the most prominent categories of heuristics due to its simplicity and effectiveness. These algorithms typically operate in two phases: a task prioritization phase and a processor selection phase. The Heterogeneous Earliest Finish Time (HEFT) algorithm, is a benchmark in this category [21]. HEFT prioritizes tasks based on their upward rank, which represents the length of the critical path from a task to the exit node, including both computation and communication costs. It then assigns each task to the processor that minimizes its earliest finish time, using an insertion-based policy that effectively utilizes idle time slots.

Building upon HEFT, several variants have been proposed to improve scheduling quality. The Predict Earliest Finish Time (PEFT) algorithm [22] enhances the processor selection phase by introducing an optimistic cost table (OCT). The OCT provides a lookahead mechanism by estimating the time from a given task to the end of the entire DAG, allowing for more informed scheduling decisions without increasing the algorithm's time complexity. Other approaches have focused on task duplication to eliminate communication overhead by executing a task's predecessors on the same processor, though this comes at the cost of redundant computation. Clustering-based methods, which group tasks to be executed on the same processor, have also been explored, but list scheduling often provides a better trade-off between performance and complexity. Our work adapts the core principles of list scheduling, specifically HEFT, to the unique constraints of the VEC environment, such as contended communication links and multi-core edge servers.

2.3 Reinforcement learning for resource management in edge computing

Reinforcement Learning (RL) provides a powerful framework for solving sequential decision-making problems under uncertainty, making it highly suitable for resource management in dynamic edge computing environments. Unlike traditional optimization methods that may require a precise system model, RL agents learn optimal policies through direct interaction with the environment. In the context of DAG scheduling, RL can be used to navigate the vast search space of possible task-to-processor mappings. Compared to RL-based methods [23, 24], ILS-YOLO leverages deterministic heuristics for faster, reproducible scheduling, avoiding RL's training overhead and hyperparameter sensitivity. While RL excels in dynamic environments, our approach prioritizes predictable performance for real-time VEC, as validated by comparisons against HEFT and PEFT Chen et al. [23] modeled the dependent task offloading problem as a Markov Decision Process (MDP) and proposed a DRL-based approach that jointly considers task topology and wireless channel interference. Similarly, Song et al. [24] formulated DAG offloading as a multi-objective optimization problem and developed an RL method to balance latency, energy, and cost.

A key challenge in applying RL to DAG scheduling is the "curse of dimensionality." The state-action space grows exponentially with the number of sub-tasks in the DAG and the number of available processors. For a DAG with $|V|$ tasks and $|P|$ processors, the number of possible scheduling policies can be immense, making it intractable for tabular RL methods like basic Q-learning to explore the environment effectively. This often leads to excessively long training times and convergence to suboptimal policies, as the agent fails to visit a representative sample of state-action pairs.

To mitigate this, our research adopts a hybrid approach that combines RL with a strong heuristic. By using an

established algorithm like HEFT to provide a baseline policy, we can effectively prune the search space. The RL agent's role shifts from learning a policy from scratch to learning a residual correction—identifying specific situations where deviating from the heuristic's greedy choice leads to a better long-term outcome. This synergy allows the RL agent to focus its exploration on the most promising regions of the solution space, which dramatically accelerates convergence and retains the ability to discover schedules superior to the baseline heuristic.

2.4 Lightweight object detection models for edge devices

The proliferation of real-time perception tasks in autonomous systems has driven the development of lightweight deep neural networks (DNNs) optimized for resource-constrained edge devices. While highly accurate models like the original YOLO [25] provide excellent performance, their computational and memory footprints are often too large for deployment on vehicular onboard units. This has led to a family of efficient architectures, such as MobileNets, SqueezeNets, and various YOLO variants (e.g., YOLO-tiny, Nano-YOLO). YoloLite [26, 27, 28] represents a specific effort in this direction, aiming to drastically reduce model complexity while maintaining acceptable accuracy for tasks like vehicle detection.

Furthermore, the core challenge of dynamic resource allocation in VEC shares conceptual parallels with problems in robust control theory, where the goal is to maintain system stability and performance under uncertainty. While the technical domains are distinct, the underlying principle of adaptive optimization provides a valuable high-level inspiration. For instance, sophisticated methodologies such as adaptive fuzzy control for chaotic and interconnected systems [29], output-feedback controllers for uncertain chaotic systems [30], robust neural adaptive control for complex multivariable systems [31], and adaptive backstepping control for uncertain nonlinear systems [32, 33] all demonstrate powerful frameworks for achieving optimal control in dynamic environments. These advanced control strategies, along with nonlinear optimal control approaches [34], although not directly applicable to DAG scheduling, inform the broader objective of developing intelligent and adaptive solutions for real-time resource management in unpredictable settings like vehicular networks.

These lightweight models achieve their efficiency through a variety of architectural innovations. Techniques such as depthwise separable convolutions, which factorize a standard convolution into two smaller layers, significantly reduce the number of parameters and floating-point operations (FLOPs). Other methods include using bottleneck layers with 1x1 convolutions to reduce channel dimensions, aggressive network pruning to remove redundant weights, and post-training quantization to represent weights with lower-bit integers. These optimizations create a critical trade-off between model accuracy, inference latency, and energy consumption, which must be carefully managed by system designers.

Even with these optimizations, the inference process of a lightweight model is not a monolithic block of computation. It remains a complex pipeline of distinct operational stages, such as a feature extraction backbone, a feature fusion neck (e.g., FPN, PAN), and multiple detection heads for predictions at different scales. Each stage has heterogeneous computational requirements. For example, the backbone may be computationally intensive but highly parallelizable, while post-processing steps like Non-Maximum Suppression (NMS) are typically sequential and latency-sensitive. By decomposing this pipeline into a DAG, we can exploit this heterogeneity through fine-grained offloading. This enables a strategic mapping of sub-tasks to the most suitable computational resource—vehicle or edge—thereby meeting strict real-time latency constraints that would be impossible to achieve with naive, all-or-nothing offloading.

3 System model and problem formulation

This chapter provides a comprehensive and formal definition of the system architecture, the application task model, and the underlying computation and communication models tailored to the VEC environment. We delve into the intricacies of resource contention and scheduling on both communication links and multi-core processors. Finally, we formulate the DAG scheduling challenge as a makespan minimization problem, establishing the theoretical foundation for the algorithms presented in the subsequent chapter. A list of key notations used throughout this chapter is summarized in Table 2. The system model comprises three key layers: (1) the vehicle's Onboard Unit (OBU) as the local server s_0, responsible for initial data capture and lightweight processing; (2) Roadside Units (RSUs) as edge servers $s_1, \ldots, s_{m-1}$, providing low-latency computation near the vehicle; and (3) the cloud server s_m, offering high computational power for intensive tasks. Interactions occur via wireless V2I links between OBU and RSUs, and wired/fiber links between RSUs and the cloud.

Table 2: Notations and their meanings

w_i	Subtask v_i
$\mathrm{pred}(v_i)$	Set of predecessor subtasks of subtask v_i
f_j	Processing frequency of the computational unit of server s_j
$n_{\max}^{j}$	Number of computational units of server s_j
$t_{\mathrm{calc}}^{i,j}$	Computational latency of subtask v_i on server s_j

$t_{\text{trans}}^{i,j,k,l}$	Transmission latency for data from v_i to v_k transmitted from server s_j to s_l
$t_{\text{prop}}^{k,l}$	Propagation latency from server s_k to s_l
$t_{\text{comm}}^{i,j,k,l}$	Total communication latency for transmitting data from v_i to v_k from server s_j to s_l
$r_{i,j}$	Data volume of the computational results that subtask v_i needs to transmit to v_j
x_i	Server allocated to subtask v_i
s_t^i	Start time of subtask v_i
f_t^i	End time of subtask v_i
$r_{t,i}$	Time when subtask v_i can begin processing

3.1 Vec architecture and resource

We consider a VEC architecture composed of a set of heterogeneous computational entities. Let $S = s_0, s_1, \ldots, s_P$ denote the set of all available servers (processors), where s_0 is the vehicle's onboard unit (OBU) and the remaining servers $s_1, \ldots, s_P$ are external resources, including RSUs at the edge and a remote cloud data center. Each server $s_j \in S$ is characterized by its computational capability and the number of its processing cores.

Computational resources: The processing capability of a server s_j is defined by its CPU frequency, f_j (measured in cycles per second). We explicitly model multi-core processors, which are common in edge and cloud servers. Let $n_j^{\max}$ be the number of identical CPU cores available on server s_j. This allows for the parallel execution of up to $n_j^{\max}$ independent sub-tasks on a single server. The heterogeneity of the system arises from the fact that both f_j and $n_j^{\max}$ can vary significantly across different servers (i.e., $f_j \neq f_k$ or $n_j^{\max} \neq n_k^{\max}$ for $j \neq k$).

Communication resources: Communication occurs over wireless links between the vehicle and RSUs (Vehicle-to-Infrastructure, V2I), and potentially wired links between edge servers and the cloud. We model the communication network as a set of links, where each link $l_{k,j}$ connecting server s_k to server s_j is characterized by its bandwidth $B_{k,j}$ and transmission power parameters. Crucially, these links are shared resources. If multiple data transfers are scheduled concurrently on the same link, they will experience queuing delays, a factor that is often overlooked in simpler models but is critical for accurately predicting latency in a VEC system.

Offloading resources: Offloading is a critical process for achieving efficient utilization of computational resources. As depicted in Fig.1, the offloading process consists of two key steps. First, the subtask to be offloaded is transmitted to the target server via a communication link. Once the transfer is complete, the subtask undergoes further processing on the server. Both steps may involve queuing due to resource contention. Specifically, when transmitting a subtask to another server, if the communication link has available resources, the transmission proceeds immediately; otherwise, the subtask is queued and waits for its turn. Similarly, upon arrival at the server, if computational resources are available, the subtask is processed immediately; otherwise, it is placed in a computation queue to await processing after prior subtasks are completed.

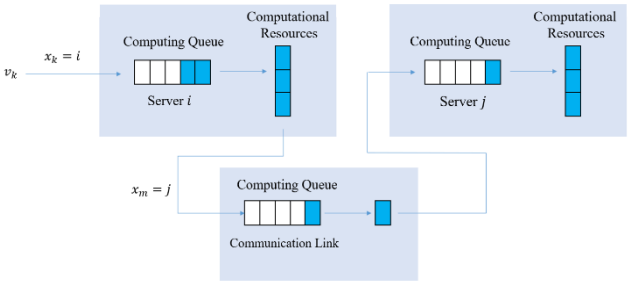


Figure 1: Offloading process

The problem is subject to the following assumptions:
1.A subtask can only be processed after all its predecessor subtasks have been completed.
2.Each computing unit can handle one subtask at a time, and processing cannot be interrupted once started.
3.Transmission of task results within the same server incurs no time cost.
4.A communication link can handle one transmission request at a time.
5.The start and end subtasks of a task are executed on the vehicle and do not consume computational resources.

3.2 Application model and latency formulation

We model the YoloLite application as a Directed Acyclic Graph（DAG）, $T = (V, E, W, R)$, which captures the intricate dependencies and workload characteristics of the inference pipeline, shown as Fig. 2. The set $V = v_{\text{start}}, v_1, \ldots, v_n, v_{\text{end}}$ contains nodes representing each computational sub－task. The set of directed edges, E, defines the dependencies, where an edge $e_{i,j}$ indicates that v_j depends on the output of v_i. The vector W contains the computational workloads w_i（in CPU cycles）for each sub－task, and the matrix R contains the data sizes $r_{i,j}$（in bits）to be transferred between dependent tasks. This fine－grained model allows us to move beyond coarse, all-or-nothing offloading and make intelligent decisions about where to place each individual stage of the YoloLite pipeline.

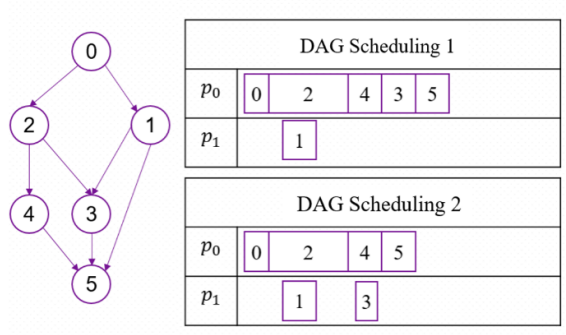


Figure 2: DAG scheduling examples

This study considers a computing environment consisting of N Roadside Units (RSUs) and one cloud server. Each RSU is associated with an edge server, and the set of all servers is denoted as $S = \{1,2,\cdots,N+1\}$. The processing frequency of server s_i is represented by f, and the number of computing units is denoted as $n_i^{\max}$. Let x_i represent the server assigned to subtask x_i, where $x_i = j$ indicates that the subtask is allocated to server s_j for processing. The communication link between servers s_i and s_j is denoted as $l_{i,j}$.

The start time of subtask v_i is denoted as st_i, and the finish time as ft_i, which can be calculated using the following equation:

$$ft_i = st_i + t_{i,x_i}^{\text{calc}} \tag{1}$$

Here, $t_{\text{calc}}(i,x_i)$ denotes the computational latency for subtask v_i on server s_{x_i}, calculated based on the subtask's workload and the server's processing frequency f_{x_i}. The finish time ft_i is linearly related to the start time st_i plus the overhead.

The start time of a subtask v_i depends on two factors: the time when the assigned server becomes available to process the subtask, avail$[x_i]$, and the time when all predecessor subtasks have transmitted their results to the server (ready time), denoted as rt_i. The ready time can be determined by the start time of the predecessor subtask's result transmission, $st_{i,j}$, as follows:

$$rt_i = \max_{v_m \in \text{pred}(v_i)} \{st_{m,j} + t_{m,i}^{\text{comm}}\} \tag{2}$$

Here, $t_{m,i}^{comm}$ represents the communication latency for transmitting the computational results from server S_m (where subtask v_m is located) to server s_i. Thus, the start time of the subtask is given by:

$$st_i = \max\{\text{avail}[x_i], rt_i\} \tag{3}$$

The availability avail (x_i) depends on the finish times of all previously allocated subtasks on server s_{x_i}, as tracked in the Unified Scheduling Table. This ensures that the server is free only after completing prior executions, modeling dependencies and preventing overlapping on the same core.

The objective of the problem is to minimize the task completion time, defined as $ft_n - st_0$. Consequently, the problem can be formulated as follows:

$$\begin{aligned} \min_x \quad & ft_n - st_0 \\ \text{s.t.} \quad & x_i \in S, \forall i \in \{1,2,\cdots,n\} \\ & st_i = \max\{\text{avail}[x_i], rt_i\}, \forall i \\ & rt_i = \max_{v_m \in \text{pred}(v_i)} \{st_{m,j} + t_{m,i}^{\text{comm}}\}, \forall i \\ & ft_i = st_i + t_{i,x_i}^{\text{calc}}, \forall i \end{aligned} \tag{4}$$

The objective function in Equation (4) minimizes the makespan by considering start times $st_i = \max(rt_i, \text{avail}(x_i))$. Here, rt_i captures delays from data dependencies, while avail(x_i) accounts for resource contention delays, ensuring all relevant overheads are modeled distinctly for clarity and accuracy. The makespan, measured in seconds (s), is defined as Makespan $= ft_{\text{end}} - st_{\text{start}}$, where st_{start} is the start time of the initial virtual task v_{start}, and ft_{end} is the finish time of the final virtual task v_{end}.

To accurately solve the scheduling problem, we must construct detailed models for both computation and communication latency that account for resource contention. The nominal execution time of a sub—task v_i on a server s_j with processing frequency f_j is defined as:

$$t_{i,j}^{calc} = w_i / f_j \tag{5}$$

However, the actual start time depends on resource availability. A naive approach of managing schedules for each core of a multi—core processor independently can lead to significant resource fragmentation, where a task may wait for a specific core despite other cores being idle. To address this, we introduce a Unified Scheduling Table for each server s_j. This mechanism tracks the total number of busy cores, $n_j(t)$, over time. A new task can be scheduled in the earliest time interval $[t_{\text{start}}, t_{\text{finish}}]$ where its data dependencies are met and a core is available $(n_j(t) < n_j^{\max}$ for the duration of the interval). This ensures efficient utilization of all available cores.

Similarly, communication latency must account for link contention. The theoretical data rate $d_{k,j}$ for a link $l_{k,j}$ between server s_k and s_j is given by the Shannon—Hartley theorem as

$$d_{k,j} = B_{k,j} \log_2 \left(1 + p_k^{\text{trans}} g_{k,j} / N_0\right) \tag{6}$$

where $B_{k,j}$ is bandwidth, p_k^{trans} is transmission power, $g_{k,j}$ is channel gain, and N_0 is noise power. In a realistic scenario, multiple data transfers contend for these links. We model each link as a single—server queue with a FIFO policy, managed by a Transfer Scheduling Table. The earliest a data transfer of size $r_{m,i}$ can start, $ST_{\text{comm}}(r_{m,i})$, is the maximum of when the data is produced by its predecessor, $FT(v_m, x_m)$, and when the link becomes available, avail $_{\text{link}}[k,j]$. The communication finish time is

$$FT_{\text{comm}}(r_{m,i}) = ST_{\text{comm}}(r_{m,i}) + (r_{m,i}/d_{k,j}) + t_{k,j}^{prop} \tag{7}$$

where $t_{k,j}^{prop}$ is the propagation delay, $ST_{\text{comm}}(m,i)$ is the link availability time from the Transfer Scheduling Table, $r_{m,i}/d_{k,j}$ is the transmission time based on data

volume and link bandwidth, and $t_{\text{prop}}(k,j)$ is the propagation delay.

This queuing model provides a realistic estimation of communication delays.

3.3. Adaptive and robust UKF variants

With these detailed models in place, the scheduling problem can be precisely formulated. The primary objective is to determine the optimal server assignment vector $X = [x_1, \ldots, x_n]$ and the corresponding start times that minimize the overall application completion time, or makespan. The scheduling process is regulated by two key factors for each subtask: the Earliest Ready Time (ERT) and the Earliest Start Time (EST).

The ERT for a subtask v_i on a potential server s_j is defined as the time when all necessary input data from its predecessor subtasks have arrived at s_j. This is determined by the finish time of the last arriving data packet, which can be formally expressed as:

$$\text{ERT}(v_i, s_j) = \max_{v_m \in \text{pred}(v_i)} \text{FT}_{\text{comm}}(r_{m,i}) \text{ on link } l_{x_m,j} \quad (8)$$

A task v_i cannot commence execution on server s_j before its $\text{ERT}(v_i, s_j)$ has been reached. The $\text{EST}(v_i, s_j)$ must satisfy both data readiness and processor availability. It is the earliest time t, which is greater than or equal to $\text{ERT}(v_i, s_j)$, at which a processing core on server s_j is available for the required duration $t_{i,j}^{\text{calc}}$. The Earliest Finish Time (EFT) is then simply calculated as:

$$\text{EFT}(v_i, s_j) = \text{EST}(v_i, s_j) + t_{i,j}^{\text{calc}} \quad (9)$$

The makespan of the entire schedule is determined by the finish time of the final virtual task, v_{end}, which is the maximum of the finish times of all its predecessor tasks. The optimization problem is therefore to minimize this makespan, $\text{FT}(v_{\text{end}}, x_{\text{end}})$, subject to all precedence and resource constraints. Formally, the problem can be stated as follows:

Minimize: Makespan(X)

Subject to:

1. Precedence Constraints: For every edge $e_{m,i} \in E$, the start time of v_i on its assigned server x_i must be greater than or equal to the communication finish time of the data from v_m : $\text{ST}(v_i, x_i) \geq \text{FT}_{\text{comm}}(r_{m,i})$ on the link l_{x_m,x_i}

2. Computational Resource Constraints: At any time t on any server s_j, the number of concurrently executing tasks, $n_j(t)$, must not exceed the number of available cores, $n_j^{\max}$.

3. Communication Resource Constraints: At any time t on any link $l_{k,j}$, at most one data transmission can be active.

This formulation accurately captures the complexities of scheduling a DAG-based application within a real-world Vehicular Edge Computing (VEC) environment, providing a foundation for the heuristic and learning-based algorithms proposed to address this challenge

.

4 Improved list scheduling for YoloLite DAGs (ILS-YOLO)

To address the makespan minimization problem formulated in chapter 3, we propose a novel scheduling algorithm named Improved List Scheduling for YoloLite DAGs (ILS-YOLO). This algorithm builds upon the well-known HEFT heuristic, enhancing its suitability for the specific resource constraints inherent to the VEC environment [28].

The DAG scheduling problem is formulated as follows:

Objective: Minimize Makespan (X) $= ft_{\text{end}}$, where X is the task-to-server mapping and ft_{end} is the finish time of the final virtual task.

Subject to:

(1) Precedence Constraints: For each subtask $v_i, st_i \geq \max_{v_m \in \text{pred}(v_i)}(ft_m + t_{comm}(m,i))$, ensuring data dependencies.

(2) Computational Constraints: Each subtask v_i is assigned to exactly one server s_{x_i}, with $ft_i = st_i + t_{\text{calc}}(i, x_i)$.

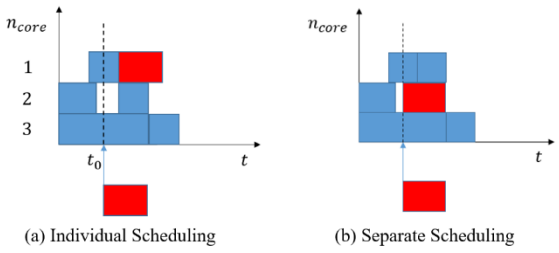
(3) Communication Constraints: Data transfers respect link bandwidth and contention via the Transfer Scheduling Table."

List scheduling algorithms typically manage individual computing units using dedicated scheduling tables. However, treating multi-core CPUs in edge servers as independent computing units with separate scheduling tables can lead to resource fragmentation, where contiguous computational resources remain underutilized over time. As illustrated in Fig. 3(a), a subtask arriving at time would be scheduled on computing unit 1 via insertion scheduling, even though the server has sufficient idle cores overall. This fragmentation arises from the disjoint management of cores, preventing holistic utilization of available resources. The ILS-YOLO algorithm operates in the following main stages: (1) Task Prioritization: Subtasks are sorted based on upward rank values to identify critical paths. (2) Processor Selection: For each subtask, calculate Earliest Ready Time (ERT) from data dependencies, Earliest Start Time (EST) from server availability, and Earliest Finish Time (EFT) to select the optimal server. (3) Resource Allocation and Offloading: Assign the subtask to the selected server and update the Unified Scheduling Table for computation and Transfer Scheduling Table for communication. (4) Iteration: Repeat until all subtasks are scheduled, minimizing overall makespan.

Additionally, static scheduling often assumes fixed communication costs and contention-free links, which contradicts real-world scenarios with limited channel capacity. In practice, data transfers on the same communication link compete for resources, necessitating a queuing model. Here, we model each link as a single-server queue (first-in-first-out, FIFO), where only one subtask can be transmitted at a time, and others must wait in the queue. During processor selection, ignoring queuing delays and allocation impacts on link schedules can severely degrade task completion times and system performance.

To address these issues, we enhance list scheduling with two key improvements:

(1) Unified Scheduling Table: Centralizes the management of all cores on a server to coordinate resource allocation. By tracking the total number of busy cores at each time slot, this table enables holistic scheduling, avoiding fragmentation and ensuring contiguous resource utilization.
(2) Transfer Scheduling Table: Explicitly records transmission schedules on communication links to account for queuing delays. This allows accurate calculation of communication latencies by incorporating both data transmission time and link availability.
As depicted in Fig. 3(b), the unified scheduling approach not only tracks assigned subtasks but also monitors real-time core availability, ensuring efficient resource utilization and reducing makespan.



(a) Individual Scheduling (b) Separate Scheduling

Figure 3: Example diagram of resource fragmentation

The first phase of the ILS-YOLO algorithm is task prioritization, which remains consistent with the HEFT algorithm. We compute an upward rank, rank_up (v_i), for each subtask v_i to estimate its priority. The upward rank represents the length of the critical path from v_i to the exit node v_{end} and is calculated recursively starting from the exit node:

$$\text{rank\_up}(v_i) = \overline{t_i^{\text{calc}}} + \max_{v_j \in \text{succ}(v_i)} \left(\overline{c_{i,j}} + \text{rank\_up}(v_j)\right) \quad (10)$$

Here, $t_{\text{calc}}(i)$ represents the average computation time of subtask v_i across all servers, computed as the mean of $t_{\text{calc}}(i,s)$ for all $s \in S$. Similarly, $c_{i,j}$ is the average communication time for data from v_i to v_j, ensuring hardware-agnostic task prioritization before processor-specific scheduling. The recursion base case is rank_up $(v_{\text{end}}) = t_{\text{end}}^{\text{calc}} = 0$. After computing the ranks for all tasks, they are sorted into a prioritized list in descending order of their rank values.
In the processor selection phase, the algorithm iterates through the prioritized list. For each subtask v_i, it evaluates the Earliest Finish Time (EFT) on every possible server $s_j \in S$. The subtask is assigned to the server yielding the minimum EFT. To calculate a realistic EFT, our algorithm leverages detailed latency models that incorporate both the Unified Scheduling Table for computation and the Transfer Scheduling Table for communication.
For each pair (v_i, s_j), the Earliest Ready Time $\left(\text{ERT}(v_i, s_j)\right)$ is determined by consulting the Transfer Scheduling Tables for all incoming data links to server s_j, providing a realistic estimate of when all prerequisite data will arrive. Next, using the Unified Scheduling Table for server s_j, the algorithm identifies the earliest available time slot that accommodates v_i 's computation requirements. This time marks the Earliest Start Time ($\text{EST}(v_i, s_j)$). The EFT is then calculated as:

$$\text{EFT}(v_i, s_j) = \text{EST}(v_i, s_j) + t_{i,j}^{\text{calc}} \quad (11)$$

After evaluating all servers, v_i is scheduled on the server s_{j^*} that minimizes its EFT. The scheduling tables are updated to reflect this allocation.
An example of the ERT calculation is shown in Fig.4. Consider subtasks v_3, v_4, v_5 as predecessors of v_6, with servers s_1, s_2, s_3. Subtask v_3 completes on s_1 and transmits results to s_3 via link l_{13}. Subtask v_4 completes on s_2 and transmits results to s_3 via link l_{23}. Subtask v_5 also completes on s_2 and transmits results to s_3 via link l_{23}. Since the transmission of r_{46} is not yet complete, r_{56} must wait for r_{46} to finish before it can start. The earliest ready time for v_6 is determined by the completion time of r_{56}.

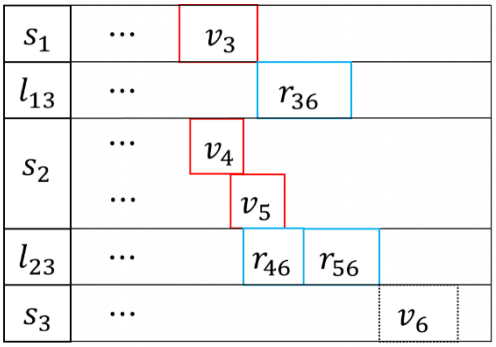


Figure 4: Example of the ERT calculation

The pseudocode for the ILS-YOLO algorithm is provided below:

Algorithm 1: ILS-YOLO Algorithm

Input: Task DAG *G=(V,E)*, set of computing resources *S*, communication overhead matrix *C*, computation overhead matrix *W*, list scheduling algorithm *alg*

Output: Server allocation for each subtask and corresponding start times to minimize makespan

1. Sort subtasks: Prioritize subtasks based on the ranking strategy of algorithm alg.
2. Initialize decision queue: Create an empty decision queue and insert the start subtask into the queue.
3. While the queue is not empty:

   Select subtask: Remove the highest-priority subtask v_i from the queue.

   For each server $s_j \in S$:

   Calculate ERT: Using the transfer scheduling table, determine the earliest time when all predecessor data for v_i will arrive at s_j.

   Determine EST: Based on ERT and server s_j 's unified scheduling table, find the earliest available time slot.

   Calculate EFT: Compute the earliest finish time for v_i on s_j.

4. Allocate subtask: Assign v_i to the server s_j that yields the minimum EFT according to alg's assignment strategy.
5. Update decision queue: Reflect the new resource allocation in the scheduling tables and update the decision queue.
6. End While

5 Performance evaluation

This section evaluates the performance of the proposed Improved List Scheduling for YoloLite DAGs (ILS-YOLO) algorithm through simulation experiments, focusing on its efficiency in task offloading and scheduling within vehicular edge computing (VEC) environments. This evaluation is designed not only to benchmark ILS-YOLO against established scheduling heuristics but also to rigorously test its robustness under adverse and varying network conditions, a critical aspect for real-world VEC deployments. Our comparative analysis also considers recent advancements in task offloading, as contextualized in our literature review, to validate the novelty and effectiveness of our approach. Both offline and online scenarios are considered to validate the algorithm's robustness under different conditions.

5.1 Experimental setup

In offline scenarios, we analyze the algorithm's performance on individual tasks without dynamic task arrivals, using Python-based simulations to isolate the effects of resource allocation and scheduling strategies. The simulation framework includes four modules: parameter configuration, DAG task generation, scenario construction, and scheduling execution, with key parameters listed in Table 3. The Bandwidth factor (Bw factor) scales the baseline link bandwidths (Table 3) to simulate varying network conditions. For instance, a Bw factor of 10^{-1} reduces the bandwidth to 10% of its default value, testing the algorithm's robustness under constrained communication.

Table 3: Simulation parameters

Parameter	Value
Edge server CPU frequency (f_e)	2 GHz
Cloud server CPU frequency (f_c)	4 GHz
Edge server CPU cores ($n_e^{\max}$)	1
Cloud server CPU cores ($n_c^{\max}$)	3
Edge-edge link bandwidth ($B_{e,e}$)	83 MHz
Edge-cloud link bandwidth ($B_{e,c}$)	17 MHz
Edge server transmit power (p_e^{trans})	0.5 W
Cloud server transmit power (p_c^{trans})	1.2 W
Channel gain (g)	−40 dB
White noise power (N_0)	10^{-6} W
Maximum computation ($w_{\max}$)	2×10^8 cycles

DAG tasks are generated with random structures to cover diverse workloads, as outlined in Table 4. The key parameters include node count (n), maximum out-degree, depth coefficient (α), regularity (β), and communication-to-computation ratio (CCR). The CCR (defined as $r_{\max}/w_{\max}$) characterizes task types, where lower CCR indicates computation-intensive tasks.

Table 4: DAG generation parameters

Parameter	Value
Node count (n)	[20, 30, 40, 50, 100, 200]
Maximum out-degree	[1, 2, 3, 4, 5]
Depth coefficient (α)	[0.5, 1.0, 2.0]
Regularity (β)	[0.0, 0.5, 1.0]
CCR	[0.005, 0.025, 0.05, 0.25]

The performance metric speedup is defined as the ratio of sequential execution time (seq) to makespan, where $seq = \min \sum_{i=1}^{n} t_{i,j}^{\text{calc}}$ represents the optimal single-processor execution time.

5.2. Impact of transfer scheduling table

To assess the role of communication resource management, we vary the initial link bandwidth and compare task completion times with and without the transfer scheduling table. As shown in Fig.5, reducing bandwidth increases makespan, highlighting the criticality of realistic communication modeling. Fig.6 demonstrates that the ILS-YOLO algorithm with a transfer scheduling table achieves an average 40% reduction in makespan compared to baseline methods, confirming that explicit modeling of link contention significantly improves communication efficiency and reduces latency.

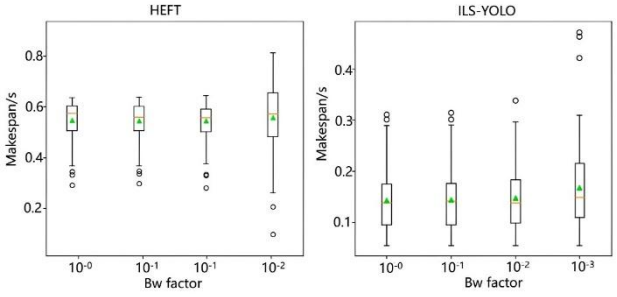
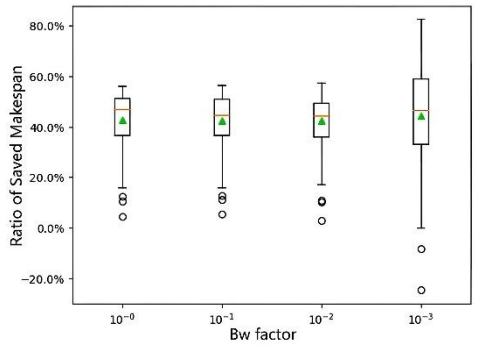


Figure 5: Comparison of task completion times under different bandwidths



Firure 6: Comparison of task completion time saving rates under different bandwidths

5.3 Field real-time vehicle detection system experiment

The experiment begins with preprocessing and analyzing a massive dataset of driving trajectories. First, the overall data is sorted and partitioned by vehicle ID and the time-series information of trajectory points to derive the trajectory path of each vehicle. Second, road intersections are divided by geographical regions, and vehicle trajectory paths passing through each intersection are extracted to reconstruct the overall vehicle flow status at each intersection. As shown in Fig. 7, the horizontal axis represents longitude and the vertical axis represents latitude. By mapping vehicle trajectory points to the map using latitude and longitude values, the real-time driving positions and trajectory status of vehicles can be obtained. Analyzing the road network information and all vehicle trajectory points further reveals vehicle trajectories passing through intersection regions at different time points, enabling the identification of traffic flow dynamics at road intersections. Traffic flow variation characteristics at different intersections can be extracted from massive vehicle trajectory data.

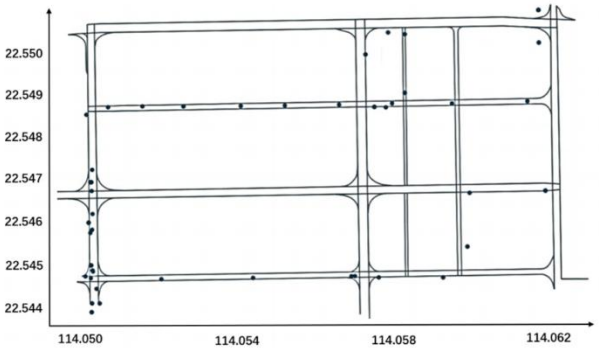


Figure 7: Shenzhen urban road data

The DAG model used in this experiment is based on the principles of PointFusion. PointFusion is a vision-point cloud based 3D object detection algorithm, whose architecture is shown in Figure 8. First, 3D point cloud features are extracted by PointNet, and RGB image features are extracted by Resnet. Then, the two types of features are further fused, and 3D bounding box corner offsets and position information are extracted by MLP. According to the structure of PointFusion, the computational amount and data volume of each stage are analyzed, and it is abstracted into a field DAG model.

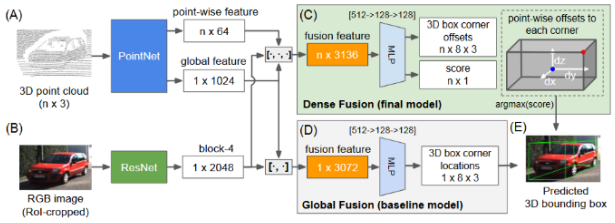


Figure 8: Field real-time DAG model

Our evaluation combines extensive synthetic DAG simulations (Table 4) with diverse structures (node counts, out-degrees, CCRs) to ensure generalizability across application scenarios. The Shenzhen urban road dataset (Figure 7) serves as a real-world anchor, abstracting a complex 3D object detection task (PointFusion, Figure 8) to validate ILS-YOLO's effectiveness in realistic vehicular settings.

(1) We evaluate the following algorithms:.Cloud-only: All tasks are offloaded to the cloud server.

(2) HEFT-singlecore: Standard HEFT without unified core scheduling.

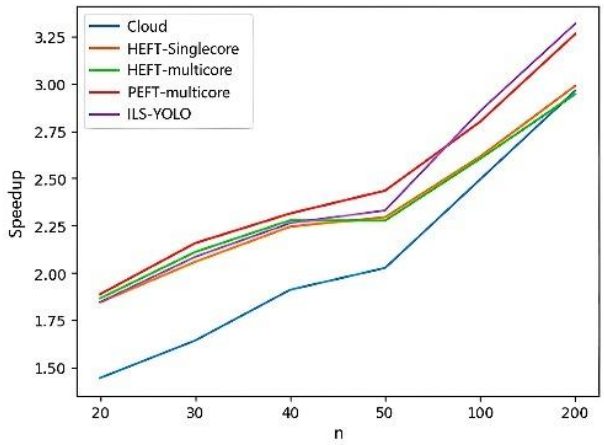
(3) HEFT-multicore: HEFT with unified scheduling of multi-core servers.

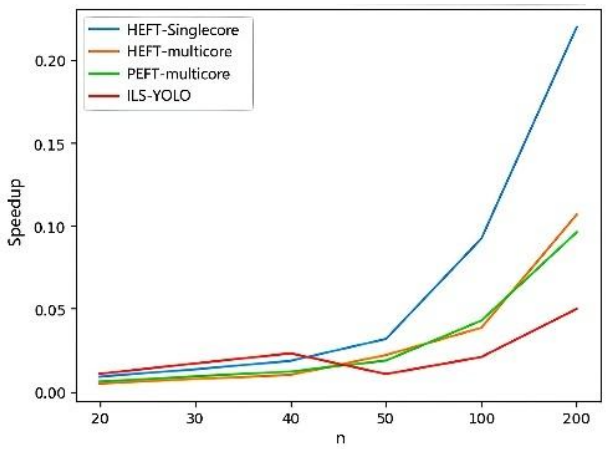
(4) PEFT-multicore: Predictive Earliest Finish Time algorithm with multi-core unification.

(5) ILS-YOLO: The proposed Improved List Scheduling algorithm integrating unified core scheduling and transfer scheduling tables.

Fig.9 compares speedup across algorithms for varying node counts. ILS-YOLO consistently outperforms HEFT-singlecore and Cloud-only, achieving up to 2.5× speedup for large DAGs (n=200), while HEFT-multicore and PEFT-multicore show moderate improvements. The superior performance of ILS-YOLO is attributed to its unified core scheduling, which mitigates resource fragmentation, and transfer scheduling, which optimizes communication utilization. The ILS-YOLO algorithm incurs overheads for calculating ERT, EST, EFT, and updating scheduling tables. These operations, while computationally intensive, are polynomial in complexity, as shown in Figure 9(b), where the algorithm's running time scales efficiently with DAG size, ensuring practicality for real-time vehicle detection.

(a)

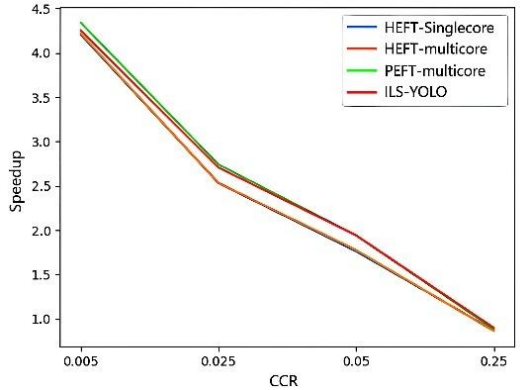




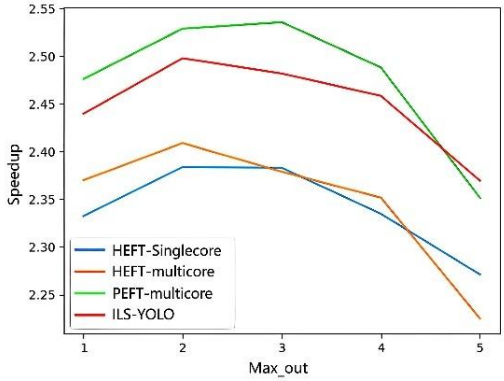
(b)

Figure 9: Comparison of algorithm performance under different numbers of nodes, (a) trend of speedup with the number of nodes, (b) trend of algorithm running time with the number of nodes

Fig. 10 illustrates makespan under different CCR values. For computation-intensive tasks (CCR=0.005), ILS-YOLO reduces makespan by 35–50% compared to HEFT-singlecore, as offloading to edge/cloud servers leverages parallel computation. For communication-intensive tasks (CCR=0.25), the gain narrows to 15–25%, reflecting the increased impact of link latency, but ILS-YOLO still outperforms baselines due to its queuing-aware communication model.



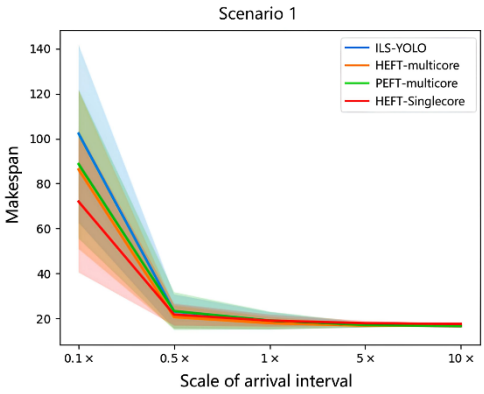
(a)



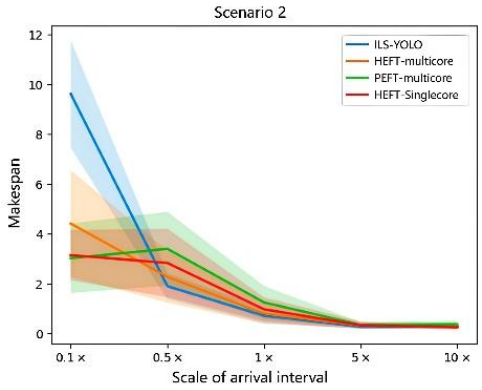
(b)

Figure 10: Comparison of algorithm performance under different CCR and maximum out-degree, (a) trend of speedup with CCR, (b) trend of speedup with maximum out-degree

In the online scenario, tasks are generated continuously, requiring the algorithm to make real-time offloading decisions for new tasks. This setup evaluates the algorithm's capability to handle continuous task streams. The DAG task data used in this experiment is derived from scientific workflow datasets commonly employed in related research, including Scenerios1-2. When the task inter-arrival time is small (high load), the average task completion time across all algorithms increases. This is because new tasks must be scheduled using remaining resources while old tasks are still being processed, impacting completion times. As the inter-arrival time increases (tasks become sparser), completion times decrease gradually, as new tasks can utilize more available resources, approaching the performance of offline scheduling. Figure 11 demonstrates ILS-YOLO's scalability in handling concurrent task arrivals under high load, maintaining performance advantages over baselines. In the online scenario, tasks arrive continuously, testing ILS-YOLO's real-time scheduling capability. Scenario 1 uses a scientific workflow dataset with a computation-intensive DAG (low CCR, high node count), while Scenario 2 employs a communication-intensive DAG (high CCR, dense edges). Figure 11 shows that under high load (small task inter-arrival times), makespan increases across all algorithms due to resource contention, as new tasks compete for limited server cores and bandwidth. ILS-YOLO outperforms baselines by leveraging the Unified and Transfer Scheduling Tables to mitigate contention, achieving lower makespan. As inter-arrival times increase, performance approaches offline scheduling, with ILS-YOLO maintaining a consistent advantage.



(a)



(b)

Figure 11: Makespan under different scenarios

A critical requirement for any scheduling algorithm intended for real-time systems is its ability to scale efficiently with the problem size. To address the reviewer's concern about scalability to more vehicles, tasks, and servers, we provide both a theoretical complexity analysis and an empirical validation of ILS-YOLO.

Theoretically, the computational complexity of the ILSYOLO algorithm is dominated by its two main phases. The first phase, task prioritization via upward rank calculation, involves a traversal of the DAG, which has a complexity of $O(n + e)$, where n is the number of subtasks and e is the number of dependencies (edges). The second phase, processor selection, iterates through each of the n tasks in the prioritized list. For each task, it evaluates the Earliest Finish Time (EFT) on all p available servers. This calculation requires checking the data arrival times from all predecessor tasks. In the worst case, this results in a complexity of O(n * p * dmax), where dmax is the maximum in-degree of any node in the DAG. Therefore, the overall time complexity of ILS-YOLO is polynomial, ensuring that its running time does not grow exponentially as the number of tasks or servers increases.

Theoretically, the computational complexity of the ILSYOLO algorithm is dominated by its two main phases. The first phase, task prioritization via upward rank calculation, involves a traversal of the DAG, which has a complexity of O(n+e), where n is the number of subtasks and e is the number of dependencies (edges). The second phase, processor selection, iterates through each of the n tasks in the prioritized list. For each task, it evaluates the Earliest Finish Time (EFT) on all p available servers. This calculation requires checking the data arrival times from all predecessor tasks. In the worst case, this results in a complexity of O(n * p * dmax), where dmax is the maximum in-degree of any node in the DAG. Therefore, the overall time complexity of ILS-YOLO is polynomial, ensuring that its running time does not grow exponentially as the number of tasks or servers increases.

However, we acknowledge that the current heuristic-based approach is not inherently adaptive to real-time, unpredictable changes in network state or server availability. While our online scenario experiments (Figure 11) demonstrate robustness to continuous task arrivals, developing a truly adaptive scheduling mechanism, perhaps by integrating reinforcement learning to dynamically adjust scheduling policies, remains a critical direction for future work to enhance generalization in highly stochastic VEC environments.

6 Conclusion

This paper has presented an innovative solution to enhance real-time vehicle detection in vehicular edge computing environments through the development of the Improved List Scheduling for YoloLite DAGs (ILS-YOLO) algorithm. The ILS-YOLO algorithm specifically addresses the challenge of minimizing end-to-end detection latency for the YoloLite pipeline, which is modeled as a Directed Acyclic Graph (DAG) to capture the dependencies and workload characteristics of the inference process. By introducing a Unified Scheduling Table and a Transfer Scheduling Table, the algorithm effectively mitigates resource fragmentation on multi-core servers and accurately models communication overhead, leading to obvious improvements in resource utilization and scheduling efficiency. The ILS-YOLO algorithm is scalable for large-scale networks, with polynomial complexity enabling efficient handling of many servers and requesters, as validated by simulations with DAGs up to 200 nodes (Section IV). Its focus on minimizing makespan through accurate resource and communication modeling makes it ideal for delay-sensitive applications like real-time vehicle detection, achieving up to 2.5× speedup over baselines, ensuring low-latency performance in intelligent transportation systems. These findings underscore the practical advantages of ILS-YOLO in supporting the demanding requirements of intelligent transportation systems and autonomous driving applications where low latency and high accuracy are paramount.

The proposed ILS-YOLO algorithm represents a substantial advancement in the field of vehicular edge computing for real-time AI applications. Its detailed consideration of both computational and communication constraints within the VEC environment ensures that it can dynamically adapt to the resource limitations and varying workloads typical of

vehicular networks. Future work will integrate dynamic vehicle mobility patterns, RSU handoffs, and fluctuating wireless conditions into the ILS-YOLO framework. This will enhance its applicability to real-world VEC deployments, building on the current model's focus on optimized DAG scheduling.

References

[1] Bai, T., Fu, S., & Yang, Q. (2022). Privacy-Preserving Object Detection with Secure Convolutional Neural Networks for Vehicular Edge Computing. Future Internet, 14(11), 316. DOI: 10.3390/fi14110316.

[2] Tang, S., Gu, Z., Fu, S., & Yang, Q. (2021). Vehicular Edge Computing for Multi-Vehicle Perception. 2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD), 2021, 1-6. DOI: 10.1109/MetroCAD51599.2021.00011.

[3] Ku, Y. J., Baidya, S., & Dey, S. (2021). Adaptive Computation Partitioning and Offloading in Real-Time Sustainable Vehicular Edge Computing. IEEE Transactions on Vehicular Technology, 70(10), 10393-10408. DOI: 10.1109/TVT.2021.3119585.

[4] Li, J., Zhang, S., Zhu, H., Chen, Y., & Liu, J. (2024). An Efficient Vehicular Intrusion Detection Method Based on Edge Intelligence. 2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD), 2024, 1-6. DOI: 10.1109/CSCWD61410.2024.10580497.

[5] Fu, H., Ma, H., Wang, G., Zhang, X., & Zhang, Y. (2020). Mcff-cnn: multiscale comprehensive feature fusion convolutional neural network for vehicle color recognition based on residual learning. Neurocomputing, 395, 178–187. DOI: 10.1016/j.neucom.2018.02.111

[7] Ma, Q., Xu, H., Wang, H., Xu, Y., Jia, Q., & Qiao, C. (2024). Fully Distributed Task Offloading in Vehicular Edge Computing. IEEE Transactions on Vehicular Technology, 73(4), 5630-5646. DOI: 10.1109/TVT.2023.3331344.

[8] Satzoda, R. K., & Trivedi, M. M. (2016). Looking at vehicles in the night: Detection and dynamics of rear lights. IEEE Transactions on Intelligent Transportation Systems, 20(11), 4297–4307. DOI: 10.1109/TITS.2016.2614545.

[9] Yan, G., Yu, M., Yu, Y., & Fan, L. (2016). Real-time vehicle detection using histograms of oriented gradients and AdaBoost classification. Optik - International Journal for Light and Electron Optics, 127(16), 7941–7951. DOI: 10.1016/j.ijleo.2016.05.092.

[10] Wu, L., Qu, J., Li, S., Zhang, C., Du, J., Sun, X., & Zhou, J. (2024). Attention-Augmented MADDPG in NOMA-Based Vehicular Mobile Edge Computational Offloading. IEEE Internet of Things Journal, 11(12), 21567-21580. DOI: 10.1109/JIOT.2024.3397648

[11] Zhao, P., Kuang, Z., Guo, Y., & Hou, F. (2024). Task Offloading and Resource Allocation in UAV-Assisted Vehicle Platoon System. IEEE Transactions on Vehicular Technology, 74(3), 2789-2804. DOI: 10.1109/TVT.2024.3458973.

[12] Mao, Y., Zhang, J., & Letaief, K. B. (2016). Dynamic computation offloading for mobile-edge computing with energy harvesting devices. IEEE Journal on Selected Areas in Communications, 34(12), 3590-3605. DOI: 10.1109/JSAC.2016.2611964

[13] X. Wang et al. (2022). Imitation Learning Enabled Task Scheduling for Online Vehicular Edge Computing. IEEE Transactions on Mobile Computing, vol. 21, no. 2, pp. 598-611. DOI: 10.1109/TMC.2020.3012509

[14] Zhan, W. et al. (2020). Deep-Reinforcement-Learning-Based Offloading Scheduling for Vehicular Edge Computing. IEEE Internet of Things Journal, 7(11), 11072-11085. DOI: 10.1109/JIOT.2020.2978830

[15] Zhou, F., & Hu, R. Q. (2019). Computation efficiency maximization in wireless-powered mobile edge computing networks. IEEE Transactions on Wireless Communications, 18(9), 4405-4419. DOI: 10.1109/TWC.2020.2970920

[16] Liu, Y et al. (2020). Dependency-Aware Task Scheduling in Vehicular Edge Computing. IEEE Internet of Things Journal, 7(8), 7810-7822. DOI: 10.1109/JIOT.2020.2972041

[17] Yalan W., Jigang W., Long C., Jiaquan Y., Yuchong L. (2020). Efficient task scheduling for servers with dynamic states in vehicular edge computing. Computer Communications, 166, 12-21. DOI: 10.1016/j.comcom.2019.11.019

[18] J. Li et al. (2022). Multiobjective Oriented Task Scheduling in Heterogeneous Mobile Edge Computing Networks. IEEE Transactions on Vehicular Technology, vol. 71, no. 8, pp. 8955-8966. DOI: 10.1109/TVT.2022.3174906

[19] Tang, M., & Wong, V. W. S. (2022). Deep reinforcement learning for task offloading in mobile edge computing systems. IEEE Transactions on Mobile Computing, 21(6), 1985-1997. DOI: 10.1109/TMC.2020.3036871

[20] L. Geng, H. Zhao, J. Wang, A. Kaushik, S. Yuan and W. Feng. (2023). Deep-Reinforcement-Learning-Based Distributed Computation Offloading in Vehicular Edge Computing Networks. IEEE Internet of Things Journal, vol. 10, no. 14, pp. 12416-12433. DOI: 10.1109/JIOT.2023.3247013

[21] Z. Liu, L. Huang, Z. Gao, M. Luo, S. Hosseinalipour and H. Dai. (2024). GA-DRL: Graph Neural Network-Augmented Deep Reinforcement Learning for DAG Task Scheduling Over Dynamic Vehicular Clouds. IEEE Transactions on Network and Service Management, vol. 21, no. 4, pp. 4226-4242. DOI: 10.1109/TNSM.2024.3387707

[22] Ullman, J. D. (1975). NP-complete scheduling problems. Journal of Computer and System Sciences, 10(3), 384-393. DOI: 10.1016/S0022-0000(75)80008-0

[23] Topcuoglu, H., Hariri, S., & Wu, M. Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems, 13(3), 260-274. DOI: 10.1109/71.993206

[24] Arabnejad, H., & Barbosa, J. G. (2013). List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Transactions on Parallel and Distributed Systems, 25(3), 682-694. DOI: 10.1109/TPDS.2013.57

[25] Gong, B., & Jiang, X. (2022). Deep reinforcement learning for dependent task offloading in mobile edge computing systems. In 2022 IEEE Smartworld, Ubiquitous Intelligence & Computing, Scalable Computing & Communications, Digital Twin, Privacy Computing, Metaverse, Autonomous & Trusted Vehicles (SmartWorld/UIC/ScalCom/DigitalTwin/PriComp/Meta) (pp. 1576-1581). IEEE. DOI: 10.1109/SmartWorld-UIC-ATC-ScalCom-DigitalTwin-PriComp-Metaverse56740.2022.00219

[26] Zhou, H., Gu, Q., Sun, P., Zhou, X., Leung, V. C., & Fan, X. (2025). Incentive-Driven and Energy Efficient Federated Learning in Mobile Edge Networks. IEEE Transactions on Cognitive Communications and Networking. DOI: 10.1109/TCCN.2025.3531464

[27] Zhou, H., Wang, J., Zhao, L., Meng, D., Feng, G., & Li, R. (2025). Joint Optimization of Charging Time and Resource Allocation in Wireless Power Transfer Aided Federated Learning. IEEE Internet of Things Journal. DOI: 10.1109/INFOCOMWKSHPS61880.2024.10620792

[28] Meng, D., Guo, J., Zhou, H., Zhang, Y., Zhao, L., Shu, Y., & Fan, X. (2024). Incentive-driven partial offloading and resource allocation in vehicular edge computing networks. IEEE Internet of Things Journal. DOI: 10.1109/JIOT.2024.3515075

[29] Liang, B., Zheng, S., Ahn, C. K., & Liu, F. (2022). Adaptive Fuzzy Control for Fractional-Order Interconnected Systems With Unknown Control Directions. IEEE Transactions on Fuzzy Systems, 30(1), 136-147. DOI: 10.1109/TFUZZ.2020.3031694

[30] Boulkroune, A., Hamel, S., Zouari, F., Boukabou, A., & Ibeas, A. (2017). Output-Feedback Controller Based Projective Lag-Synchronization of Uncertain Chaotic Systems in the Presence of Input Nonlinearities. Mathematical Problems in Engineering, 2017, 8045803.DOI: 10.1155/2017/8045803

[31] Zouari, F., Saad, K. B., & Benrejeb, M. (2012). Robust neural adaptive control for a class of uncertain nonlinear complex dynamical multivariable systems. International Review on Modelling and Simulations, 5(5), 2075-2103. DOI: 10.1002/rnc.1347

[32] Zouari, F., Saad, K. B., & Benrejeb, M. (2013, March). Adaptive backstepping control for a class of uncertain single input single output nonlinear systems. In 10th International Multi-Conferences on Systems, Signals & Devices 2013 (SSD13) (pp. 1-6). IEEE. DOI: 10.1109/SSD.2013.6564134

[33] Rigatos, G., Abbaszadeh, M., Sari, B., Siano, P., Cuccurullo, G., & Zouari, F. (2023). Nonlinear optimal control for a gas compressor driven by an induction motor. Results in Control and Optimization, 11, 100226. DOI: 10.1016/j.rico.2023.100226

[34] Zouari, F., Saad, K. B., & Benrejeb, M. (2013, May). Adaptive backstepping control for a single-link flexible robot manipulator driven DC motor. In 2013 International Conference on Control, Decision and Information Technologies (CoDIT) (pp. 864-871). IEEE. DOI: 10.1109/CoDIT.2013.6689656