Continuous Lifelong Conflict-Aware AGV Routing with Kinematic Constraints

Ruizhong Wu
DSA Thurst, HKUST(GZ)
rwu601@connect.hkust-gz.edu.cn

Mengxuan Zhang
The Australian National University
mengxuan.zhang@anu.edu.au

Shuxin Wang
Hong Kong Industrial Artificial Intelligence and Robotics Centre
saxonwang@hkflair.org

Frodo Kin Sun Chan
Hong Kong Industrial Artificial Intelligence and Robotics Centre
frodochan@hkflair.org

Yan Nei Law
Hong Kong Industrial Artificial Intelligence and Robotics Centre
ivylaw@hkflair.org

Lei Li
DSA & INTR Thrust, HKUST(HZ)
Department of CSE, HKUST
thorli@ust.hk

ABSTRACT

Automated Guided Vehicles (AGV) are becoming increasingly important in modern warehouses to cope with the enormous logistic demands of developing e-commerce and the growing operational costs. The key component of implementing such a system is planning the paths of a large horde of AGVs to deliver orders from shelves to packing locations. The existing solutions regard it as a *Multi Agent PathFinding (MAPF)* problem, but they can hardly be applied in practice because none of them could satisfy the continuous (temporal), lifelong (future task unknown and keeps appearing), kinematic (acceleration/deceleration/rotation), online (fast response), and scalability (large network, large AGV number, large task number) at the same time. Therefore, we first propose an AGV routing framework that can satisfy all these properties with its corresponding routing algorithm. Then, to improve the efficiency, we propose the Multi-Hop Conflict-Aware Search method (MHCAS) with action combination, MHSC to reduce the search space, and OHSMD to decompose motions such that routing time is reduced by three orders of magnitude. Extensive experimental studies verify the superiority of our methods compared with the state-of-the-art.

PVLDB Reference Format:
Ruizhong Wu, Mengxuan Zhang, Shuxin Wang, Frodo Kin Sun Chan, Yan Nei Law, and Lei Li. Continuous Lifelong Conflict-Aware AGV Routing with Kinematic Constraints. PVLDB, 18(7): 2254 - 2267, 2025.
doi:10.14778/3734839.3734859

1 INTRODUCTION

With the fast advancement of e-commerce and AI, *Automated Guided Vehicles (AGV)* are playing increasingly important roles in warehouses, where items are transferred from the storage racks to the packing areas [4, 51]. Such a robotized warehouse can save human labor significantly, especially when billions of orders could appear in one single day during the shopping festivals. Although AGVs have already been widely used in practice, their power is still not fully unleashed. This is because to achieve high throughput, AGVs must avoid conflicts to reduce traveling time. However, due to the lack of effective conflict-aware routing algorithms, the AGVs still heavily rely on themselves to avoid collisions during operation, which leads to the following operation modes: *non-cooperative* mode [51] where AGVs decide their routes by themselves, and *cooperative* mode [3, 5, 22, 38, 39, 43, 44, 47] that pre-plan the routes for the AGVs and let them resolve the conflicts on the fly. Specifically, the first mode does not have global information about the other AGVs' movements, so the conflicts can hardly be avoided beforehand. The second mode can reduce the forthcoming conflicts, but it cannot run as planned because the existing algorithms simplify restrictions of real-life operations to reduce algorithm running time.

The main difference between reality and the simplified environments is threefold: 1) They assume time is discrete while real-life time is *continuous*; 2) Most of them find the optimal schedules when each AGV has only one task to deliver while in real life the tasks keep appearing continuously (*lifelong*); 3) They assume AGVs can appear in the neighboring grid instantaneously while real-life AGVs have to follow *kinematic constraints* like acceleration, deceleration, and turning. If any of these requirements are simplified, the actual routes would take a longer time as more unforeseen conflicts would appear. Therefore, in this work, we propose a conflict-aware AGV routing algorithm that can satisfy these three requirements.

However, it is non-trivial to achieve them effectively and efficiently. The first challenge comes from the practical continuous mode with kinematic constraints. The existing discrete time-step modes assume the time-space is partitioned into uniform time steps, and the AGV could move to its neighboring grids instantly between time steps. However, these simplifications ignore the complexity coming along with the continuous mode in real-life scenarios, where the searching status is larger and expands more quickly in routing compared with the discrete mode. Moreover, with kinematic constraints, we need to consider the acceleration/deceleration and turning, *i.e.,* traveling time from one grid to another varies depending on the AGV's current status. For instance, it takes different times to turn to neighboring grids in different directions. Even when an AGV travels in the same direction, it takes different times to reach its next grid, depending on whether it stops on it or passes through it. To deal with those challenging scenarios, we propose a routing model that considers kinematic constraints with a time

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734859

* Lei Li is the corresponding author.

interval intersection-based conflict detection method to meet the practical requirements for AGV routing.

Subsequently, the second challenge is efficiency concerns brought about by the huge search statuses. To ensure that the fastest path can be accurately found, we propose the General Conflict-Aware Search (*GCAS*) by considering all possible traveling states of movement, but it is computationally expensive because there are more options available for each action, which leads to a rapid increase in the search space. In real-world dynamic and time-sensitive environments like warehouses, the ability to perform timely path planning is critical. However, *GCAS* not only fails to respond promptly in path planning, but the detailed per-step action planning is also inconsistent with the actual AGV control mechanism. To accelerate the path search process, we reduce the state number by combining multiple actions (or steps) along one direction into one action and propose the corresponding Multi-Hop Conflict-Aware Search method (*MHCAS*). After that, a heuristic strategy named *MHSC* is designed to further reduce the search space, and the *Motion Decomposition*-based method named *OHSMD* is put forward to improve the efficiency of *GCAS* by thousands of times.

The last challenge is how to efficiently organize the continuous execution of tasks by AGVs. The *Conflict-Based Search (CBS)* solutions [5, 23, 30, 32, 38, 39] simply assume the AGVs stay or disappear at the target after delivery so they only need to consider one batch of tasks, while in real life the orders keep emerging continuously and the destinations could not be blocked. The warehouse scenario solutions [40, 41] assume the AGVs return to their initial position after delivery, but this rigid strategy would increase the overall operation time due to the triangle inequality. What is more, the real-life AGV network has the following characteristics: 1) More AGVs are used to improve throughput; 2) To reduce the conflict possibility, the edges in the network are normally one-way roads; 3) Pickup locations are fixed. We take advantage of these features and propose a *flexible buffer* to reduce the triangle inequality as much as possible to improve the overall operational efficiency. Our contributions can be summarized as follows:

- We propose a realistic AGV routing model and its corresponding algorithm GCAS that is continuous in time, conflict-aware during routing, and considers kinematic constraints;
- We propose MHCAS to match the actual patterns of AGVs, and MHSC and OHSMD with progressive searching strategies to improve computational efficiency;
- We propose a parking location buffer for efficient lifelong task scheduling to improve the makespan of the system;
- We deploy and evaluate our methods with extensive experiments on various AGV networks. Results show that our approach outperforms the state-of-the-art.

2 RELATED WORKS

2.1 Single Agent Pathfinding

Depending on the availability of environmental information [29, 54], this problem can be categorized into *local pathfinding* where the environment is partially obtained through LiDAR or visual sensors, and *global pathfinding* where the complete environment is known. The environment could be modeled as *grid* [16] or *topological graphs* [7] as used in this work for flexible network, or Voronoi diagrams [3, 13, 46, 49], probabilistic roadmap methods [12], and *geometric methods* [27] that replace the obstacles with indexed geometry. In terms of the algorithm, *Dijkstra*'s [10] and A^* [14] are two base algorithms when a grid or network is available. D^* [45], *Lifelong* A^* [20], D^*-lite [19] are the dynamic version of A^* that re-route during the trip when new obstacles appear. When there is an open area with no specific network, the sampling-based methods like *RRT* [21] and RRT^* [18] generate points randomly to form a search space incrementally towards the destination. However, they can hardly deal with multiple agents for the optimal path without conflict. Path indexes like *CH* [11, 36] and *HL* [2, 8, 35, 55–57] are extremely fast in finding the shortest path in a network with both topological and weight changes, but they are not collision-aware and weights are not AGV deterministic. Considering the conflict in the future can be viewed as an extreme case of the time-dependent routing [25, 26], but they focus on finding the fastest path through the fixed time-dependent function computation, while the speed profiles and occupation of AGV network are highly dynamic and unpredictable. Finally, adapting the existing capacity-aware routing algorithm [9, 52] directly would result in an explosion of search space.

2.2 Multi-Agent Pathfinding MAPF

The simplest solution uses some path algorithms to plan routes for the AGVs and replan the routes on the fly when conflicts happen during operation [15, 48, 53]. When it comes to considering other agents, the problem becomes *Multi-Agent Pathfinding (MAPF)*. Classic optimization algorithms like *Ant Colony* [28], *Particle Swam* [33], and *Genetic Algorithm* [42] could be used but are too slow to use in a real-life large warehouse. *Conflict Based Search (CBS)* [38] and its variations [3, 5, 22, 23] are more efficient than the previous ones and could achieve near-optimal solutions, but they have to plan the routes for all AGVs at a time so cannot support real-life online routing requirement. Besides, they are "one-shot" that do not support lifelong working scenarios where tasks keep appearing. *MAPD* [31, 32] consider lifelong by assigning new tasks for the agents at their destinations, but it cannot handle our problem where delivery tasks from the same order share the same destination. *RHCR* [24] re-routes every *h* timesteps and include new tasks. Such a behavior is to support the *CBS*-like solutions but requires lots of unnecessary re-computations. [48] further considers cases where new agents could appear on the fly. *Cooperative* A^* [43] shares a similar procedure to our approach by planning the routes one by one, but it only works in the discrete environment without considering kinematic and lifelong. [40, 41] are similar and are only effective on the strip-based network. *Reinforcement Learning* [6, 17, 34, 37] are also utilized recently, but their efficiency is still not acceptable to use in practice. In summary, there is no conflict-aware solution for MAPF that supports online, lifelong, and full kinematics at the same time.

3 PRELIMINARY

3.1 General Settings and Definitions

The whole AGV system runs on a grid network as defined below:

*Definition 3.1 (**Grid Network**).* A grid network is made up of $m \times n$ same-size square grids (physical mats) put tightly together, where m and n are the numbers of rows and columns of the grids.

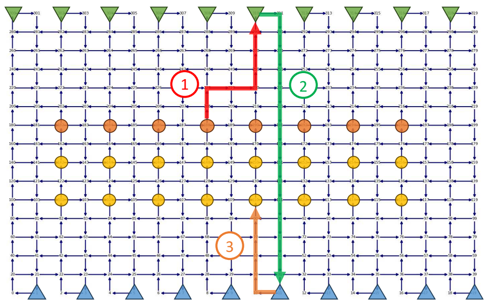

Figure 1: Grid Network Example. Green Triangle: Pickup Locations; Blue Triangle: Drop-Off Locations; Orange Circles: AGV Initial Locations; Yellow Circles: Parking Buffers.

Our AGV has a circular shape with a diameter the same as the grid's edge length. When it stops or turns, it has to reside exactly inside one grid; when it moves, it can only go along one row horizontally or one column vertically. Restricting AGVs' movement inside grids can reduce the conflicts introduced by arbitrary movements. Therefore, conceptually, a grid network is equivalent to a graph $G(V, E)$, where V is the set of vertices denoting the grid centers and E is the set of edges connecting these centers. Figure 1 shows a directed grid network example with an intersection denoting the center of a grid and the directed arrows restricting the AGV's moving direction. We will test different settings in the experiments, but use the directed grid as default for easy description in the remainder of this manuscript. Next, we define the AGV tasks.

*Definition 3.2 (**Task**).* A task is denoted as a triplet $\tau = (s, d, t)$, with s as the origin for item pickup, d as the destination for item dropoff, and t as the pickup time. $\mathrm{T} = \{\tau_i\}$ is the task set at hand.

There are three types of grids: 1) *Pickup S* are the task origins; 2) *Dropoff D* are the task destinations; 3) *Initial I* park the vacant AGVs. In a fully automated warehouse, both S and D are fixed and work in conjunction with specific machines for item pickup and dropoff. In our scenario, these two types of grids are located in the top row and the bottom row of the warehouse. The initial locations can be on any other grid. As shown in Figure 1, there are 10 pickups (green), 10 drop-offs (blue), and 8 initial parkings (orange).

To fulfill one task τ_i, the system selects one AGV to pick up the item at s_i from its current location no earlier than t_i and deliver it to d_i as fast as possible. When the AGV reaches s_i or d_i, it will stay to wait for the item to be placed on the AGV or unloaded to the packing machines. To support the lifelong routing, the AGV cannot stay at the drop-off location as it will block the latter tasks, so we assume it goes back to its initial location at this stage. Accordingly, we identify AGV as having the following basic ***actions***: 1) *Staying*. If we don't send commands to the AGV, it will stay where it is, *i.e., staying* action; 2) *Turning*. When we send *turning* commands to the AGV, it can change the direction by facing up, down, left, and right; 3) *Moving*. When we send *moving* commands to the AGV, it will move along the current direction. The AGV manufacturers specify the actual values of these constraints and actions, and we can compute AGV's different moving times accordingly. Many research works assume that the agent moves one step within a fixed time. However, in real-world conditions, AGV moves with **kinematic constraint**, and we model it with the following physical laws:

(1) A constant acceleration and deceleration speed;
(2) A maximum speed limit to move constantly;
(3) A constant rotation speed to change its heading direction and move only towards the current orientation;
(4) AGV needs to stop before changing the heading direction.

Topology Generalization. It should be noted that our algorithm can also work in other topologies. Because the underlying data structure is a graph and all the locations are virtual, we can extend along the following dimensions: 1) *Pickup, Drop-off, and Parking Locations*: they can be placed in arbitrary places. When the pickups are put on the center shelves, and drop-offs are placed on the boundary, we can get another classic warehouse design as used in [41]; 2) *Obstacles*: when a location is inaccessible (like a shelf or pillar), we can remove the edge from it to others. It can also be used to simulate ad-hoc emergencies; 3) *Diagonal Movement*: diagonal edges could be added with each vertex having eight neighbors.

3.2 AGV Movement

Now we define an AGV movement trajectory, *i.e.,* AGV path below:

*Definition 3.3 (**AGV Path**).* An AGV path is a sequence of actions along a set of consecutive vertices $p = \langle (v_0, ac_0, t_0), \ldots, (v_k, ac_k, t_k) \rangle$ where either 1) $(v_i, v_{i+1}) \in E$, or 2) $v_i = v_{i+1}, \forall i \in [0, k-1]$. A triple (v_i, ac_i, t_i) means AGV takes action ac_i at vertex v_i at time t_i.

The first condition restricts the movement towards the neighboring vertex, while the second condition allows one vertex to appear several times, which occurs when a turning or staying action appears consecutively. The time t_k associated with each vertex is the time when it takes action at the vertex center, and the following action's time should accord with the kinematic constraints.

Considering the shape and size of the AGV and grid, if there is spatial overlap between the AGV and grid, we say that this grid is **Occupied** by this AGV. For example in Figure 2-(a), the AGV a_1 occupies grid A and grid B while the AGV a_2 occupies grid B and grid C. Therefore, we say grid B is occupied by two AGVs at time t. The time interval during which an AGV a_j occupies a grid/vertex v_i is called **Occupation Interval (OI)**, denoted as $o_i^j = [t_1, t_2]$. For example in Figure 2-(b), AGV a_1 starts to enter B (leaves A's center) at t_1 and leaves B (arrives at C's center) at t_2, we say a_1 occupies grid B within time interval $[t_1, t_2]$. If a grid does not overlap spatially with any AGV, we regard this grid/vertex as **Vacant**.

From the perspective of the grid/vertex's occupation, an AGV path consists of a set of grids being traversed during its occupation time. Therefore, an AGV path of a_i can also be represented as $p_i = \langle (v_0, o_0^i), \ldots, (v_k, o_k^i) \rangle$. This format is crucial for conflict detection. Next, we discuss the cases when multiple AGVs are moving simultaneously in the system by starting with two paths:

*Definition 3.4 (**AGV Path Conflict**).* Given any two AGV path p_i and p_j, they have conflict if $\exists (v_k, o_k^i) \in p_i$ and $(v_k, o_k^j) \in p_j$ such that $o_k^i \cap o_k^j \neq \phi$. Otherwise, we say p_i and p_j are conflict-free.

At first glance, it seems to be conservative because, in some cases, even if two AGVs occupy the same grid at the same time, they may not have a collision. For instance, in Figure 2-(a), both a_1 and a_2 occupy grid B at time t, but they do not collide with each other. However, as shown in Figure 2-(c), suppose a_1 moves to the

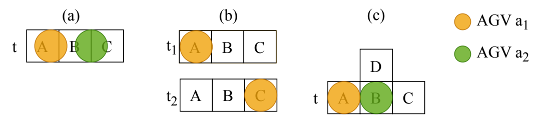

Figure 2: Gird Occupation and Vacancy Example

right and arrives at A, and a_2 moves upward and arrives at B. Then at the next discrete time $t+1$, a_1 arrives at B while a_2 arrives at D simultaneously. The classical definition of conflict assumes that no conflict occurs in this case, because at moments t and $t+1$, the two AGVs are in different grids. In real scenarios, however, it will cause collisions. Since AGVs have shape and volume, whenever a_1 moves before a_2 fully reaches D, it is bound to hit a_2 as their movement directions are orthogonal. Even if the movement directions are the same, collisions could still happen. Back to Figure 2-(a), assuming that a_1 and a_2 are moving to the right at the same time, they can both reach grid C at the next discrete time. As AGVs have different speeds during actual movement, if a_1 has a higher speed, it will hit a_2. Therefore, our definition of conflict is based on the idea that a grid can only be occupied by one AGV at any time. As for multiple paths, the following property is desired:

*Definition 3.5 (**Valid Path Set**).* Given a set of AGV paths P, it is a valid path set if all its paths are conflict-free.

For a path set P, we define its *makespan* $M(P)$, as the time difference between the last path's finish time and the earliest path's starting time, to evaluate the performance of the AGV routing algorithms. Now we can define our problem formally:

Problem Definition. Given a grid network $G(V,E)$, a set of AGVs $A = \{a_i\}$ with kinematic constraints, and a set of N tasks $T = \{\tau_j\}$, we aim to find a valid path set P such that its makespan $M(P)$ is minimal.

We call this problem *Kinematic Constraints Conflict-Aware AGV Routing Problem.* It should be noted that the AGV system operates in an online environment where tasks emerge dynamically and must collaborate with other machines. For instance, new tasks are generated only when goods are transported to the pickup location. Therefore, planning the paths in the task-appearing order is natural in real life. In addition, system errors caused by mechanical failures, unexpected task errors, and network delays could incur collisions from time to time, so online re-routing is necessary for error handling.

4 CONFLICT-AWARE AGV ROUTING

4.1 Conflict Detection and Collision Avoidance

In this section, we present how to detect conflicts by time interval and avoid them by taking conflict-free actions when planning a new path in the context of scheduled paths.

As illustrated in Figure 3, we identify four time points during AGV a_i's movement process from grid v_j's perspective:

(1) *Entering Time* t^e: a_i starts enter the next grid v_j;
(2) *Arrival Time* t^a: a_i arrives at the center of v_j ;
(3) *Leaving Time* t^l: a_i start to leave v_j and enter next grid;
(4) *Entire Left Time* t^{el}: a_i leaves v_j completely and arrives at the center of the next grid.

Accordingly, we can know v_j's occupation interval by a_i is $[t^e, t^{el}]$. Since an AGV's diameter is the same as the grid's length, the leaving

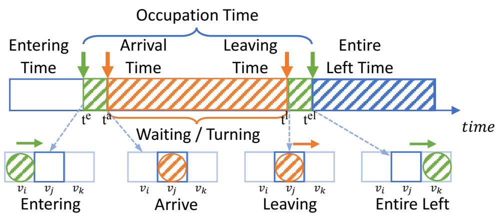

Figure 3: Grid Occupation Details Example

time of the previous grid is the same as the entering time of the following grid. We define $t_p = t^{el} - t^l$ as the ***Travel Time*** from v_j to v_j's neighbor, representing the time moving from the center of v_j to the center of v_j's neighbor.

Specifically, when we have a set of paths, then each grid is associated with a set of consecutive intervals, whereas each of them is either *Vacant* or *Occupied*. We denote the **Occupied Interval Sequence (OIS)** of grid v_j as $OIS_{v_j} = \{OI^0_{v_j}, \dots, OI^k_{v_j}\}$, where $OI^k_{v_j} = [t_k, t'_k]$ is v_j's k^{th} occupied interval. If $OI^k_{v_j}$ is occupied by a_i, then we say $A(OI^k_{v_j}) = a_i$. It should be noted that the grids are occupied First-In-First-Out. Meanwhile, we denote the **Vacant Interval Sequence (VIS)** of grid v_j as $VIS_{v_j} = \{VI^0_{v_j}, \dots, VI^i_{v_j}\}$, where $VI^i_{v_j} = [t_i, t'_i]$ is v_j's i^{th} vacant interval.

As OIS and VIS are complementary to each other, when we compute one of them, the other one will be generated easily. Therefore, we only describe how to convert an AGV path into an OIS. Initially, each grid has one big vacant interval for the whole time domain, which will be broken into smaller pieces as multiple OIs are inserted when paths are scheduled. For example in Figure 3, assuming that the initial VI of grid v_j is $[0, +\infty]$, when a scheduled AGV occupies v_j in time interval $[t^e, t^{el}]$, the VI will be split into two intervals: $[0, t^e]$ and $[t^{el}, +\infty]$.

Next, we use VIS (or OIS) to detect whether the planning path conflicts with the planned paths. Suppose, with an initial conflict-free state, an AGV arrives at v_j at time t^a, and $t \le t^a \le t'$, $[t,t'] \in VIS_{v_j}$, we will analyze how to keep it conflict-free by taking Actions:

1) Staying Action: As the vacant interval $[t,t']$ is upper bounded by t', the latest time the AGV can stay at v_j is t', beyond which it will conflict with other scheduled paths. Therefore, the longest time period the AGV can stay at v_j is $t' - t^a$;

2) Turning Action: The time for the AGV to turn depends on the turning angle, and we assume that the time it takes to turn is w. We set $t^a + w \le t'$ for the AGV to turn without conflict;

3) Moving Action: This action is needed to ensure that it passes through all contacting grids without conflict. Assume that after the moving action, the AGV moves from v_j to its neighbor v_k, with the travel time t_p depending on the initial state and the actions taken. From the perspective of v_j, if $t^a + t_p \le t'$, it can leave v_j without conflict; for v_k, if there exists a VI $= [t_i, t'_i] \in VIS_{v_k}$ that satisfies $t^a \ge t_i$ and $t^a + t_p \le t'_i$, then it can arrive v_k without conflict.

For example in Figure 4, suppose v_j and v_k are v_i's out-neighbors. The blue-shaded areas are the occupied intervals by the scheduled AGVs. Suppose an AGV a enters v_i at t^i_3 (the left end of the orange-shaded area) and starts to leave v_i at time t^j_3, which is also the entering time at v_j. The exact value of t^j_3 depends on a's actual movements, such as whether if stops or turns at v_i, how long it

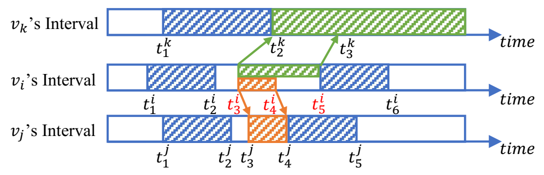

Figure 4: Conflict Detection Interval Example ($v_i \rightarrow v_j$ and $v_i \rightarrow v_k$). Shaded Area: Occupied Intervals; Empty Area: Vacant Intervals.

stops or passes through v_i. t_4^i is the latest staying time at v_i because if a entirely leaves v_i later than t_4^i, it cannot entirely leave v_j before t_4^j, after which v_j is scheduled to be occupied by another AGV. As for the other neighbor v_k, because its occupation time is longer, then waiting at v_i is the only option before reaching it. t_2^k (the left end of the green-shaded area) is the earliest possible entering time to v_k, since it is occupied before that; while t_5^i is the latest staying time at v_i because another AGV is scheduled to enter v_i at t_5^i.

4.2 Search State

In this section, we adopt a *Dijkstra's*-like search strategy to find the fastest route for the current AGV task on top of the previously scheduled routes. However, unlike the ordinary *Dijkstra*'s search where each edge has one fixed weight (denotes distance, travel time, cost, etc.) and the neighbor's connectivity is fixed, the edge weight and connectivity of the AGV routing are both dynamic and determined by the kinematic constraints. This is because the AGV routing is stricter than the ordinary routing to avoid conflicts.

In order to find a conflict-free path, an AGV needs to take conflict-free actions in each grid. We use a triad (t, v, $\mathcal{HD}$) to store the searching state, where t is the timestamp arriving at grid v and $\mathcal{HD}$ is the heading direction with four orthogonal values. Each searching state can take a conflict-free action to generate the next state. To search for the fastest path, an AGV needs to take all conflict-free actions to generate all possible states where two problems need to be addressed. Firstly, given a state, both turning action and moving action generate a finite number of states. However, the staying action can generate an unlimited number of states in a continuous environment because the staying time is a real number. So the first problem is how to represent all possible staying times? In a discrete environment, an AGV stays for an integer number of time units, so it is possible to increase the staying time by 1 unit each time to generate the next state. Since we can't enumerate all the real numbers, we use an interval to represent all possible values of the staying action in a continuous environment. Even so, different endpoint values of intervals also produce infinite states.

Then the second problem is how to determine the endpoint value. We propose to address it from the perspective of VIS. When the current grid is reached, the AGV must take a moving action and generate the state of arriving at the neighboring grid, no matter how long it stays at the current grid. The AGV's moving action must fall into the VIS of the neighboring grid, and there is a limited number of VI in the VIS. From the VI's perspective, the AGV has the earliest timestamp et and the latest timestamp lt to arrive at the neighboring grid, where et and lt belong to a VI. Therefore, we can iterate all VIs of the neighboring grid to compute the earliest and latest timestamp of each VI under the condition that the AGV can stay for the corresponding time without conflict.

Based on the above analysis, we transform the infinite search state (t, v, $\mathcal{HD}$) into a finite search state (et, lt, v, $\mathcal{HD}$). Then, we claim the correctness of our AGV path search process with the Theorem below:

Theorem 4.1 (**AGV Searching Correctness**). *Given the current OIS/VIS, the fastest path for a task can be found if and only if all the possible search states with smaller entering times at the current searched grid are considered, and none can result in an earlier arrival time at its neighboring grid.*

Proof. We first prove the necessity by contradiction. If any possible action with a smaller entering time was not considered when visiting any grid, then this action's consequential search spaces would never be fulfilled, and whether their paths' arrival times are earlier or later would never be known. Therefore, the optimality of the current fastest path is not guaranteed. Secondly, we prove the sufficiency. When all the possible states have been tested, and none could provide an earlier arrival time, then the current fastest one is the optimal one. □

4.3 General Conflict-Aware Search with Kinematic Constraints

In this section, we first model the AGV movement, then we present the conflict-aware search with kinematic constraints.

4.3.1 ***Movement Modeling.*** We analyze the movements of AGV towards its neighboring grid and use the following notations to describe the time of AGV actions:

Moving Time t_p: The time spent to move from the current grid to the neighbor grid with action A_{mov}, which makes the AGV move from a grid to its neighbor grid. This time period consists of four types of sub-action from a kinematic perspective:

(1) t_{ma}: The time to accelerate uniformly from the current grid to the neighbor grid with action $\underline{A_{ma}}$;
(2) t_{md}: The time to decelerate uniformly from the current grid to the neighbor grid with action $\underline{A_{md}}$;
(3) t_{mu}: The time to move with uniform speed from the current grid to the neighbor grid with action $\underline{A_{mu}}$;
(4) t_{mad}: The time to accelerate uniformly from the current grid by 1/2 grid length and then decelerate uniformly by 1/2 grid length to the neighbor grid such that it maintains its original speed, with action $\underline{A_{mad}}$;

Turning Time. w_{0°, w_{90°, w_{180°, w_{270° denote the time to turn 0°, 90°, 180°, 270°, respectively.

The AGV has two kinematic states: *static* (with zero speed) and *moving* (with non-zero speed). Specifically, when the AGV is static, it can take staying, turning, or moving actions. It can turn to any neighbor, but note that different neighbors have different directions compared with the AGV's current heading; when it is in the moving state, it can only take moving action towards its neighbor in the same direction. Because if the speed of the AGV is not zero, given the kinematic constraints, the AGV will continue to move forward due to inertance. Also, when the AGV has reached its maximum speed, it cannot take an acceleration action A_{ma} next, which indicates

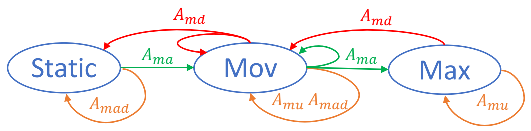

Figure 5: States Transition

the speed is also crucial for the next actions. Thus we modify the search state from (et, lt, v, $\mathcal{HD}$) to (et, lt, v, $speed$, $\mathcal{HD}$), where $speed$ is the speed when arriving at v. Therefore, we summarize the AGV states transition as shown in Figure 5.

4.3.2 ***Search Algorithm.*** Given the pickup and dropoff grids of an AGV, we search paths by expanding the search space incrementally, which looks like the same as *Dijkstra*'s search at first glance. Nevertheless, it is actually different because the movement is constrained by state transition as analysed in section 4.3.1.

Accordingly, we design the **General Conflict-Aware Search (GCAS)** algorithm as shown in Algorithm 1. Suppose an idle AGV will execute the current task and has arrived at the pickup location s at time t with its heading direction $\mathcal{HD}$ and zero speed. We initialize the earliest time et to enter the neighbor grid as t and the latest time lt (set to ∞). Then the state is inserted into a priority queue Q sorted by et (line 1). We iterate the top states of Q when it is not empty (line 3). When the destination d with the static state is reached, the search terminates (line 4). Then we retrieve the actual path according to the actions taken and update the VIS with the resulting Path (line 7).

Specifically, if the current state is static, neighbors in all directions can be traversed (line 10), which needs to compare the direction of AGV and the neighbor grid to determine the turning time (line 11). Then we compute the earliest and latest time for conflict-free leave from v based on t_p for each VI of the neighbors (line 14) with the Collision Avoidance (**CA**) procedure described in section 4.1. Here, two moving sub-actions A_{ma} and A_{mad} are allowed (line 13), with A_{mad} stopping at the neighbor and A_{ma} passing through it. If the state is moving, only the neighbor along the current direction can be visited. The actual possible actions depend on the current speed. If the AGV has achieved its maximum speed, then it can either keep the current speed with A_{mu} or decelerate with A_{md} (line 19). Otherwise, it can go on accelerating with A_{ma}, decelerate with A_{md}, stick with the current speed A_{mu}, or take A_{mad} action (line 20). Now, we prove the correctness of Algorithm 1 as below.

Theorem 4.2 (**GCAS Correctness**). *Algorithm 1 can find the conflict-aware fastest path correctly.*

Proof. According to Theorem 4.1, we only need to prove Algorithm 1 covers all the possible states. Firstly, depending on whether the speed is 0 or not, an AGV has only two categories of states, and they are both covered. Then for the static states, only four actions could be taken: staying, rotating to one neighbor, moving through the neighbor, and stopping at it. However, the waiting and the rotation have already been incorporated into the conflict detection, so only the two moving actions remain, and their corresponding states are covered. As for the moving state, all the possible actions are also covered. Therefore, the search space of Algorithm 1 is complete. As the states are visited in the increasing order of the arrival time (and the states' et is always larger than the top's), the remaining states all would have larger arrival time to destination, so the first top value is the earliest arrival one, and Algorithm 1 is correct. □

Algorithm 1: General Conflict-Aware Search

```
   Input: Graph $G(V,E)$, $OIS_v$ $\forall v \in V$, Task $\tau = (s,d,t)$
   Output: A valid Path set $Path$
1  $Q$ ←Insert Initialize $state(t,\infty,s,0,\mathcal{HD})$;
2  while $Q$ is not empty do
3      $state \leftarrow Q.pop()$;
4      if $state.v = d$ and $state.speed = 0$ then $break$;
5      if $state.speed = 0$ then $Static(state)$;
6      else $Motion(state)$;
7  Path Retrieval and $Update(VIS, Path)$;
8  return Path;
9  Function Static( State $S$):
10     foreach $u \in S.v$'s neighbor grid do
11         $\mathcal{HD} \leftarrow$ compare $S.\mathcal{HD}$ and $u$;
12         foreach $VI \in VIS_u$ do
13             foreach $Action \in \{A_{ma}, A_{mad}\}$ do
14                 $speed, et, lt \leftarrow$ CA(Action, VI);
15                 $Q \leftarrow$ Insert $nextState(et,lt,u,speed,\mathcal{HD})$;
16 Function Motion( State $S$):
17     $\mathcal{HD} \leftarrow State.\mathcal{HD}$ ;
18     $u \leftarrow S.v's$ neighbor with same $\mathcal{HD}$;
19     if $speed = max\_speed$ then $ActionList \leftarrow \{A_{mu}, A_{md}\}$;
20     else $ActionList \leftarrow \{A_{ma}, A_{mu}, A_{md}, A_{mad}\}$;
21     foreach $VI \in VIS_u$ do
22         foreach $Action \in ActionList$ do
23             $speed, et, lt \leftarrow$ CA(Action, VI);
24             $Q \leftarrow$ Insert $nextState(et,lt,u,speed,\mathcal{HD})$;
```

Complexity of Algorithm 1. We start by analyzing the possible largest number of states. Suppose there are $|A|$ AGVs in the system, then there are at most $|A| - 1$ OIs for each grid. When planning the routes, there are at most $|A| - 1$ other AGVs running with their current task scheduled, and in the vast majority of cases, these AGVs would pass through each grid at most once (because if a grid is to be passed through more than once by the same AGV, there is a loop in the AGV's valid path, which rarely happens). For a static state, we only need to identify the unique static state with the direction $\mathcal{HD}$, the grid ID, and VI in the current grid. The maximum number of directions is 4, the number of grids is $|V|$, and the maximum number of VI is $|A|$. Therefore, the maximum number of static states in the queue is $4|A| \cdot |V|$. Then we analyze how many moving states at most. A moving state can extend up to $4|A|$ (line 20 and line 21) states. Since moving states have a direction, they can only extend in one direction. The maximum number of grids in this direction is $max\{m,n\}$ steps. Therefore, a moving action can extend at most $(4|A|)^{max\{m,n\}}$ moving states. Since any moving state must be obtained by $k (k > 0)$ moving actions from the static state, any moving state must be traced back to the static state. We call the moving state obtained from the static state by 1 moving action the ancestor moving state. Since a static state extend at most $4|A|$ (line 10 and line 12) ancestor moving states and an ancestor moving state extend at most $(4|A|)^{max\{m,n\}}$ moving states, the maximum number of moving state in the queue is $(4|A| \cdot |V|) \times (4|A|) \times (4|A|)^{max\{m,n\}}$. We denote $N = (4|A| \cdot |V|) + (4|A| \cdot |V|) \times (4|A|) \times (4|A|)^{max\{m,n\}} = (4|A| \cdot |V|) \times (1 + (4|A|)^{max\{m,n\}+1})$. The network needs to store its vertices and their four edges, requiring $O(|V| + 4|V|)$ space. Additionally, we need to record the VIs, with each grid having up to $O(|A|)$ intervals, requiring $O(|A| \cdot |V|)$ space. Therefore, the total space complexity

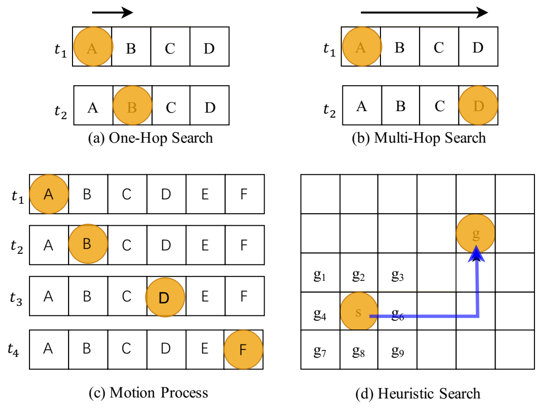

Figure 6: Search Strategies

is $O(|V| + 4|V| + |A| \cdot |V| + N) = O(N)$. As we use a binary heap for the priority, the overall time complexity for one task's general searching is $O(N \log N)$.

5 OPTIMIZED AGV ROUTING

We first propose Multi-Hop Conflict-Aware Search based on Algorithm 1. Then, we introduce two progressive strategies to further enhance the search efficiency.

5.1 Multi-Hop Conflict-Aware Search

The general search Algorithm1 has a large state number because of the detailed and less constrained per-step action selection every time a state is extended to a neighbor. However, it is unnatural in practice as we do not have control of AGV's detailed movement commands for each step. We can only tell the AGV to move from one grid to another vertically or horizontally. For example, in Figure 6-(b), when the AGV receives a command for moving to D, the AGV will move from A to D. There is no separate command for the intermediate process (passing through B and C). Such an observation inspires us with a new state generation method: could we break the constraint of only looking at the direct neighbors and change the search to a (command-like) destination-oriented fashion? Therefore, we propose the **Multi-Hop Conflict-Aware Search (MHCAS)** algorithm without intermediate moving states and only consider when a destination can be reached. In the following, we first model the multi-hop search with the movement analysis, then we present the corresponding search algorithm.

5.1.1 ***Multi-Hop Movement Modeling.*** First of all, we generalize the neighbors' concept to the *multi-hop neighbors* as those grids that can be reached horizontally or vertically, *i.e.,* all the vertices from v to the grid network's boundary along the vertical or horizontal directions are v's multi-hop neighbors. In multi-hop search, the AGV has only one kinematic state pushed into the queue that is static with zero speed, where each static state corresponds to an actual AGV moving command. As for the actions, it can take *staying*, *turning*, or *moving*, with staying and turning incorporated into the *conflict-aware* procedure. Therefore, A_{mov} is the only conflict-aware action. Then the VIS/OIS should also be further extended for collision avoidance, with **Multi-Vacant Interval Sequence (M-VIS** (u, v)**)** defined as the sequence of all vacant time intervals along the path from u to its multi-hop neighbor v. A path is valid only if it can pass through all the VIs in the M-VIS.

5.1.2 ***Multi-hop Neighbor Collision Avoidance.*** Although we focus only on the multi-hop neighbors, we need to avoid conflicts with the passing grids by analyzing the motion of the AGV as follows:

In the general search model, the moving action contains four kinds of sub-actions. But in the actual control of the AGV, there is no separate command for sub-actions. In our scenario, the working mode of the AGV is to receive the *k-step* command that drives the AGV to move k grids in the minimum time and finally stop at the command target grid. We analyse how the AGV motion process is accelerated and decelerated according to the laws of motion in physics. The fastest moving motion to reach the specified destination grid after receiving the command is that the AGV first accelerates uniformly (or at the maximum speed), and then decelerates uniformly to speed zero to reach the destination grid. For example, in Figure 6-(c), the AGV is at grid A at time t_1, and we assume that the AGV can reach the maximum speed after accelerating through two grids. Let's see how the movement situation would be if we sent different commands to the AGV. Suppose we send a command to reach B, and the AGV arrives at t_2. Then the action from A to B corresponds to A_{mad}, denoted as $Action(A, B) = A_{mad}$. If the command sent is to D (arriving at t_3), then $Action(A, B) = A_{ma}$, $Action(B, C) = A_{mad}$ and $Action(C, D) = A_{md}$. If the command is sent to F (arriving at t_4), then $Action(A, B) = Action(B, C) = A_{ma}$, $Action(C, D) = A_{mu}$ and $Action(D, E) = Action(E, F) = A_{md}$.

From the above analysis, each *k-hop* command (multi-hop neighbor) has a deterministic combination of sub-actions. Therefore, we can utilize collision avoidance in the general algorithm to implement **Multi-hop neighbor Collision Avoidance (M-CA)** procedure that ensures conflict-free with all passing grids, but now the sub-actions selection is unique according to hop/step number.

5.1.3 ***Multi-Hop Search Algorithm.*** Now we introduce our *Multi-Hop Search Algorithm* as shown in Algorithm 2. Because the speed information is no longer used, we adjust the search state as $(et, lt, v, \mathcal{HD})$, with et and lt contained in the same VI. Furthermore, each state in the queue is static. The state that reaches v at the moment of et can reach lt by staying on, which means that the case of lt is covered, so we can continue to simplify the state representation as $(et, v, \mathcal{HD})$. The main searching organization (line 1-8) is the same as Algorithm 1, with only one state handling remaining (line 6). Then inside the Static function, neighbors in all directions can be traversed (line 10), which needs to compare the direction of AGV and the neighbor grid (line 11). Then we test these multi-hop neighbors one by one, each of them calls the M-CA described in the previous Section 5.1.2 once, generates a new state, and puts it into Q. Now we prove the correctness of Algorithm 2.

THEOREM 5.1 (**MHCAS CORRECTNESS**). *Algorithm 2 can find the conflict-aware fastest path correctly.*

PROOF. Similar to Theorem 4.1, we only need to prove Algorithm 2 covers all the possible states. Firstly, the states in the queue are static, and our algorithm considers this type of state. During the search, we extended the state for all possible neighbors, and

Algorithm 2: Multi-Hop Conflict-Aware Search

```
Input: Graph $G(V, E)$ and $OIS_v$ $\forall v \in V$, Task $\tau = (s, d, t)$
Output: $Path$
Initialize $state \leftarrow (t, s, \mathcal{HD})$;
$Q.insert(state)$;
while $Q$ is not empty do
    $state \leftarrow Q.pop()$;
    if $state.v = d$ then $break$;
    $Static(state)$;
Path Retrieval and $Update(VIS, Path)$;
return Path;
Function Static( State $s$):
    foreach $u \in State.v$'s Multi-Hop neighbors do
        $\mathcal{HD} \leftarrow$ compare $State.\mathcal{HD}$ and $u$;
        foreach $VI \in VIS_u$ do
            $et \leftarrow$ M-CA($A_{mov}$, M-VI($v, u$));
            $Q \leftarrow$ Insert $nextState(et, u, \mathcal{HD})$;
```

we considered all the VI of the neighbors. Since all the possible actions/states are covered, the space of Algorithm 2 is complete. □

Complexity of Algorithm 2. For each M-CA, because it has at most ($max\{m, n\}$) multi-hop neighbors and $|A|$ VIs, the complexity for one multi-hop neighbor is $O(|A| \cdot max\{m, n\})$. If we compute the multi-neighbors straightly, then complexity becomes $O(|A| \cdot max\{m^2, n^2\})$. In terms of the state number, since there are only static states in the queue, by analysis of the general algorithm, we know that the the state number is $O(4|A| \cdot |V|)$. So the overall time complexity with Q is $O(4|A| \cdot |V| \log(4|A| \cdot |V|)$. Therefore, similar to Algorithm 1, the space complexity for storing the queue states is $O(4|A| \cdot |V|)$, and the overall time complexity of Algorithm 2 is $O((4|A| \cdot |V|) \times (|A| \cdot max\{m^2, n^2\} + \log(4|A| \cdot |V|)))$

Moreover, we propose to improve the efficiency of Algorithm 2 from two aspects: 1) *Shared M-CA*. Since the AGVs have the same mode of movement, there may be common sub-action combination prefixes for neighbors with different hops, especially for long-distance multi-neighbors. For example, in Figure 6-(c), for the move to D and move to F commands, the corresponding sub-actions from A to B are A_{ma}. Consequently, we have the opportunity to reuse the collision-avoidance results of these intermediate grids. It can reduce the quadratic calling number of the collision-avoid process to linear, and it is especially effective in long-range tasks as the grid number in a row could be hundreds in real-life warehouses; 2) *Early Termination.* During the M-CA procedure, when the forward testing phase could not find any vacant interval for some VI, then all the following multi-hop neighbors do not need testing. This could help reduce the length of each M-CA procedure.

5.2 Progressive Strategies

From the perspective of AGV execution commands, the general model (GCAS) sends one sub-action at a time that moves only one step, while the multi-hop model (MHCAS) sends moving actions that can move one or more steps, which is how AGVs are controlled in practice. However, the multi-hop model generates a large number of search states, resulting in poor time efficiency due to attempting to search all possible multi-hop neighbors. Meanwhile, the general model also generates a lot of states because there are a lot of sub-actions that can be selected. Therefore, we propose two progressive algorithms that can match the actual mode of controlling AGVs and improve the computational efficiency.

5.2.1 *Multi-Hop Search with Constrictions (MHSC) Algorithm*.

The multi-hop search generates the state of all possible multi-hop neighbors, which is certainly huge. Therefore, we first limit the number of hops to reduce the number of multi-hop neighbors. We restrict the horizontal (vertical) number of search hops to not exceed the absolute value of the difference between the current column (row) and the destination column (row). For example, in Figure 6-(d), s is the current grid, d is the destination grid since the distance between d and s in the horizontal direction is 3 units, we limit the hop number to 3 when searching from s, so the multi-hop neighbor does not include any neighbors right of g's column. Similarly, we limit the multi-hop neighbors to 2 in the vertical direction. Limiting the number of hops is not enough because s may search to the west, but obviously, d is located to the northeast of s. We continue to restrict the search direction towards the destination grid. In Figure 6-(d), we limit the search only to the east horizontally and the search only to the north vertically. Although these constrictions would affect the result's optimality, they could reduce the search space a lot.

5.2.2 *One-Hop Search with Motion Decomposition (OHSMD) Algorithm*.

The proposed multi-hop search considers all potential command scenarios, allowing for optimal utilization of AGV performance. However, this approach often leads to the expansion of a vast number of states, particularly in larger maps.

Inspired by previous AGV movement analysis, we can decompose each *k-hop* command into a virtual *one-hop* command (sub-action) from a motion perspective. Similarly, the virtual *one-hop* commands can be merged into the actual *k-hop* commands. Therefore, we propose the **One-Hop Search** to make the general search model work by introducing sub-actions selection restrictions. Specifically, in general search, each sub-action is independent and has no dependency. However, according to the decomposed motion process, the following constraints should be satisfied: 1) only when AGV reaches the maximum velocity can A_{mu} be allowed, 2) if the action corresponding to the extension of the previous state to the current state has the behavior of deceleration, then when the current state is extended to the next state, only A_{md} is allowed. Although there are no actual sub-actions, we follow the laws and constraints of the actual motion process (Section 5.1.2), and after the search is completed, we can merge multiple sub-actions into a single action command to control the AGV.

The difference between multi-hop and one-hop search is that multi-hop generates states based on specific hop numbers, and the search space is extended in four orthogonal directions. One-hop does not determine the corresponding hop number during the search. Until the search is complete and the path is retraced, the search space expands outward from the current position.

5.3 Heuristic Search and Action Space Pruning

In this section, we introduce the heuristic function and state pruning to further improve search efficiency.

We get the heuristic value h through heuristic functions, and then, we use $v.et + h$ for the sorting key of the state of grid v in Q. In the A* algorithm, a simple calculation of the Manhattan distance yields the h value, but in our search model, we have to take into account the speed and direction into the state. h needs to be less

Algorithm 3: Heuristic Value Computation

```
   Input: $v$, $speed$, $d$
   Output: Heuristic Value
1  $h \leftarrow 0$;
2  if $speed = 0$ then $Static(v, d)$;
3  else $Motion(v, speed, d)$;
4  $h \leftarrow t_f(rowDiff) + t_f(colDiff)$;
5  return $h$;
6  Function Static($v, d$):
7      $rowDiff \leftarrow |v.row - d.row|$, $colDiff \leftarrow |v.col - d.col|$;
8      if $d.row < v.row$ then $h \leftarrow h + 2 * \omega_{90^\circ}$;
9      else
10         if $v.col \neq d.col$ then $h \leftarrow h + \omega_{90^\circ}$;
11 Function Motion($v, speed, d$):
12     if $d.row \leq v.row$ then
13         $v\prime \leftarrow$ forward $speed$ step from $v$,$h \leftarrow h + t_p(speed)$;
14     else
15         if $|v.row - d.row| \geq speed$ then
16             $v' \leftarrow$ backward $speed$ step from $v$, $h \leftarrow h - t_p(speed)$
17         else $v\prime \leftarrow$ forward $speed$ step from $v$, $h \leftarrow h + t_p(speed)$;
18     $Static(v\prime, d)$ ;
```

than or equal to the time it actually takes to get from the current state to the destination, and the closer it is, the better. We calculate the fastest time to reach the destination grid based on the current state as h under the condition that there is only one AGV in the grid network with no orientation restrictions.

Suppose A_{md} makes the speed decrease by 1 and A_{ma} makes the speed increase by 1. We use $t_f(k)$ to denote the fastest time for AGV to move k grids and $t_p(k)$ to denote the time spent to reduce the speed to 0 after k A_{md} from the initial speed of k. Assuming the input direction $\mathcal{HD}$ is upward, the details are shown in Algorithm 3, and the computation of heuristic value for the other input directions $\mathcal{HD}$ is similar to Algorithm 3.

We discuss it in two cases depending on whether the speed is 0 or not: 1) When the speed is 0, if the row of the destination grid ($d.row$) is below the current grid ($v.row$), and the AGV needs to turn 90° twice to reach the target (line 8). For example in Figure 6-(d), this case corresponds to the situation that the input v is the grid s, input d is the grid g_7, g_8, or g_9. If $d.row \geq v.row$ and if the column of the destination grid ($d.col$) is not the same as the current grid ($v.col$), the AGV needs to turn 90° once to reach the target (line 10)). This case corresponds to the situation that input d is the grid g_1, g_3, g_4, or g_6. 2) When the speed is not 0, the situation is complex. If $d.row \leq v.row$ (line 12), the AGV needs to spend time $t_p(speed)$ to execute the $speed$ step A_{md} (line 13) to reach $v\prime$, because the current speed is not 0 and the destination position is in the opposite direction of the speed. The AGV arrives at $v\prime$ with a speed of 0, so the Function Static is called next (line 18). If the $d.row > v.row$ and there is enough distance to stop before reaching d (line 15), we backtrack $speed$ steps in the opposite direction to find the position $v\prime$, where the speed is zero (line 16). The new position $v\prime$ has a larger h than v, so h needs to minus $t_p(speed)$ (line 16). If the distance is not enough, the AGV will take time (line 17) to decelerate to reach v' to stop. For the two cases above, the turning time has been taken into account, and the time to move to the destination (line 4) is added to the final value of h. The h value is only related to the current direction, speed, and destination. Since the direction is only up, down, left, and right, the value of the speed is limited, and the destination grid is limited, h can be preprocessed and does not need to be recalculated every time.

Next, we introduce how to prune the state to further improve efficiency. Static states have the potential to generate a large number of states due to their ability to extend in all directions and to stay in the current grid. We improve the computation time by pruning the static states. In the analysis of the multi-hop model, we know that for the static state, only et needs to be recorded, since lt is already covered by et. Furthermore, we can prune different static states of the same vertex. For two static state $state_1(et_1, v, \mathcal{HD}_1)$ and $state_2(et_2, v, \mathcal{HD}_2)$, suppose that $et_2 > et_1$ and et_1 and et_2 both belong to the same VI of the vertex v. If one of the following two conditions is satisfied: 1) $\mathcal{HD}_1 = \mathcal{HD}_2$ 2)the turning time from $\mathcal{HD}_1$ to $\mathcal{HD}_2$ is less than $|et_2 - et_1|$, $state_2$ can be pruned directly since et_2 has been covered by et_1.

6 LIFELONG TASK SCHEDULING

This section presents the lifelong task scheduling to finish a set of tasks continuously. In real life, a complete task is made up of three trips as illustrated in Figure 1: 1) An AGV travels from its current location to the pickup location (red), 2) from the pickup location to the drop-off location (green), and 3) from the drop-off location to somewhere else to make room for the future tasks (orange). In the following, we discuss where the AGVs should go after delivery and which AGV should go for a task.

One straightforward solution is to assign each AGV a *Fixed Parking Location* such that it has to return to its parking location after delivery and before being able to be assigned to the next task. The parking locations are selected from those grids that the AGVs have to pass through towards the pickup locations to reduce travel time. However, this fixed strategy is not flexible enough because it may cause the pickup grid of the next task to be far away from the parking position, which leads to an increase in makespan. An improvement over the fixed parking is the *Flexible Parking Location*, where the AGVs do not have a fixed parking location but can go wherever is vacant and nearer to its drop-off location when it finishes delivery. However, when the tasks are not evenly distributed spatially and temporally, the AGVs might still go to a faraway location when the nearby ones are all occupied.

To further increase the availability of the parking locations, we propose the *Parking Location Buffer* strategy. Specifically, because a parking location has to take an upward location, the column that contains a parking location is blocked by it to some extent. Besides, the AGVs have to go upward towards the pickup locations eventually, so these "channels" are the ways that must be passed. Therefore, we could take advantage of this property and add more *buffer locations* (yellow circles in Figure 1) along the column of the current parking locations such that the parking locations are multiplied. In this way, when an AGV finishes delivery, it has a higher chance to pick a parking location that is nearest to it, such that detours could be avoided as much as possible. Specifically, when the AGV reaches the drop-off, it finds the column with the smallest horizontal distance from the drop-off and with unassigned parking spaces. In this column, it finds the first unassigned parking space from top to bottom.

In terms of the AGV assignment, we choose the AGV with the smallest estimated time to reach the pickup of the next task. Specifically, our estimated time is calculated from two parts. The first

Table 1: Performance of Different Search Modes

Graph	Performance	GCAS	MHCAS	MHSC	OHSMD
$G_{1,B}$	Runtime	114.43	1.94	0.67	**0.08**
	Flowtime	**2,173.11**	2,176.82	2,176.82	2,174.01
	Makespan	**40.08**	40.23	40.23	40.08
$G_{1,O}$	Runtime	89.36	1.04	0.33	**0.05**
	Flowtime	**2,257.17**	2,258.60	2,258.60	2,258.93
	Makespan	40.80	40.80	40.80	40.80

part is the time when the AGV arrives at the parking location, and this time represents the earliest time when the AGV can go to the next task. The second part is the Manhattan distance of the current parking space from the pickup location divided by the maximum speed. Although replacing the current three-part routes could be reduced to a two-part one where the third coming-back route could be removed and AGV assignment could start from the drop-off location, it does work in real life. This is because firstly, going upward is inevitable, and these channels are taken by the parking locations, so the two-part one has nearly the same time as the three-part one. Moreover, it cannot handle cases where the task number becomes sparse, or the dilemma of where an AGV should go to make room for the coming delivery tasks. After all, at the end of the day, the AGVs still need places to park.

7 EXPERIMENT

We implement a series of experiments to evaluate the performance of our proposed algorithms compared with the state-of-the-art.

7.1 Experimental Settings

All the algorithms are implemented in C++ with -O3 optimization, and tested on a Ubuntu 20.04 server with two Intel Xeon Platinum 8375C 2.9GHz (each has 32 cores and 64 threads) and 2TB memory.

Networks. ***1) Grids***: We obtain the grid network G_1 from the real warehouse with size 214 rows × 16 columns. Each grid is $0.25m$ long. The top (resp. bottom) row is 100 pickup (resp. drop-off) locations, with one grid placed between every two pickups (drop-offs). The grid network has no obstacles, and each inner grid has four connected neighbors. We denote the bidirectional network as $G_{1,B}$, and the uni-directed network $G_{1,O}$, with the direction of each row and column being set the same and that of neighboring rows/columns being reversed. We also expand the row and column number by 2×, 3× and 4× to get larger maps G_2 (428 × 32), G_3 (642 × 48), and G_4 (856 × 64). ***2) Diagonal Grids*** $G_{1,B,Diag}$: This is a bi-directional network where each grid connects to all its 8 neighbors, including the diagonal ones. ***3) Central Shelf*** G_{bm}: This is a benchmark warehouse [1] of size 170 × 32, where non-traversable shelves as pickups are placed in the center and drop-offs are on the boundary.

AGV. The maximum speed is 1.5 m/s, acceleration/deceleration is a constant value of 1.5 m/s^2, and angular velocity is 1 π rad/s.

Task Sets. We manually generated a task dataset T_1 based on G_1, which contains 100 tasks. For each $\tau_i = (s_i, d_i, t_i) \in T_1$, we randomly generate t_i and sort the tasks by t_i. For any other $\tau_j = (s_j, d_j, t_j) \in T_1 (i \neq j)$, we set $s_i \neq s_j, d_i \neq d_j$. We also collected three real-life task datasets T_2, T_3, T_4 used in G_1, which all contain 15201 tasks. To avoid the task start time's impact on the makespan, the start times are only used for task ordering, then all the tasks are processed sequentially and continuously. For the central shelf network, we generate 1000 tasks randomly.

Parking Buffer. We test different parking buffer configurations: $G_{1,B,fix}$ has only one parking position per column, so each AGV is fixed with one parking position; For a more flexible configuration, we allocate five parking positions per column. $G_{1,B,100}$ represents the densest setting with parking positions distributed across 100 columns. $G_{1,B,60}$ and $G_{1,B,20}$ are derived from $G_{1,B,100}$ by removing the central 40 and 80 columns.

Evaluation Metrics. 1) *Runtime*: algorithm's running time; 2) *Flowtime* [24]: the sum of the time each agent takes to complete its journey, which reflects the throughput of the system; 3) *Makespan*: the time required for the entire system to complete all tasks.

Compared algorithms. We will compare the performance of our four proposed methods General Conflict-Aware Search (*GCAS*), Multi-Hop Conflict-Aware Search (*MHCAS*), Multi-Hop Search with Constrictions (*MHSC*) and One-Hop Search with Motion Decomposition (*OHSMD*) with the existing state-of-the-art baselines SRP [41], CBS [38], and ECBS [3]. Due to the lengthy computation time without the absence of heuristics, the experimental results of all the search algorithms we present incorporate heuristic guidance (Section 5.3). Specifically, CBS/ECBS generates a sequence of actions for each agent/AGV within each discrete time step, denoted by $(Action_1, Action_2, ..., Action_i)$. $Action_i$ either keeps the AGV stopped on the current grid or moves to a neighboring grid. We tried two adaptation methods, the first one, denoted as $\text{CBS}_\text{o}/\text{ECBS}_\text{o}$, sequentially makes AGVs execute moving actions (searching out conflict-free paths to a one-hop neighbour) according to the timestamp ordering of the moving actions. This adaptation is rather natural, because the moving action obtained by CBS just moves one grid (corresponding to A_{mad}) at a time. This approach, however, cannot exploit the performance of the AGV. For example, suppose that $Action_{i-1}$, $Action_i$ and $Action_{i+1}$ are moving actions and their corresponding start positions are in the same column or row, which means that $Action_{i-1}$ and $Action_i$ can be combined into a single command. In this way, we will take advantage of the speed of AGV by sending a 2-step command directly rather than sending two 1-step commands. Therefore, we try the second adaptation method, denoted as $\text{CBS}_\text{m}/\text{ECBS}_\text{m}$, which is still based on the moving action ordering, but lets the AGV execute a longer command each time.

7.2 Experiment Results

7.2.1 ***Search Comparison.*** We first compared our GCAS, MHCAS, MHSC, and OHSMD algorithms with 100 AGVs running T_1 on G_1. The pickup location is in the same column as each AGV's initial location, and they stop at the drop-offs. As shown in Table 1, there is not much difference in terms of flowtime and makespan, with GCAS having a slight advantage. However, GCAS, which follows the conventional routing strategies, has the longest running time, 1000× of OHSMD. The decrease in runtime demonstrates the effectiveness of our optimized strategies. When we compare the performance on $G_{1,B}$ and $G_{1,O}$, the runtime decreases on $G_{1,O}$ but with increased flowtime and makespan. This is because the search space in the $G_{1,B}$ is larger to find the fastest paths, whereas $G_{1,O}$ narrows down the search space. Finally, as OHSMD has the best performance, we set OHSMD as our representative method.

Table 2: Performance of Different Path Search Algorithms (1 Batch of 100 Tasks)

Graph	Performance	GCAS	MHCAS	MHSC	OHSMD	SRP	CBS_o	CBS_m	$ECBS_o$	$ECBS_m$
$G_{1,O}$	Runtime	92.84	1.25	0.44	**0.07**	0.33	5.18	4.96	0.84	0.66
	Flowtime	**2,039.91**	2,041.93	2,041.93	2,042.27	3,057.68	7,473.56	3,110.45	7,480.06	3,169.67
	Makespan	38.63	38.63	38.63	38.63	69.52	165.80	66.48	165.80	67.18

Table 3: Lifelong Performance on Synthetic Task Dataset (1000 Tasks)

Graph	Performance	GCAS	MHCAS	MHSC	OHSMD	SRP	TP	CBS_0	CBS_m	$ECBS_0$	$ECBS_m$
$G_{1,O,fix}$	Runtime	1,867.28	25.01	8.86	1.13	6.21	**0.81**	52.44	49.91	9.37	7.27
	Flowtime	47,219.54	47,222.16	48,585.57	**47,100.12**	93,445.90	66,504.58	169,595.45	67,866.06	169,601.95	68,560.21
	Makespan	527.19	521.62	537.64	**524.82**	1,006.72	739.69	1,787.02	714.03	1,787.02	721.09
	Memory	31 GB	451.2 MB	382.4 MB	135.6 MB	133.4 MB	189.3MB	255.5 MB	262.7 MB	256.9 MB	262.7 MB

Table 4: Lifelong on Industrial Task Dataset (15201 Tasks)

Graph	Performance	T_2		
		OHSMD	SRP	TP
$G_{1,O,fix}$ (Fixed parking)	Runtime	**22.71**	92.37	308.61
	Flowtime	696,687.64	1,405,636.59	1,069,485.37
	Makespan	7,023.05	14,131.63	10,741.50
$G_{1,O,100}$ (100-column parking)	Runtime	36.57	66.28	308.91
	Flowtime	539,582.77	1,137,913.24	1,069,485.37
	Makespan	5,427.26	11,437.00	10,741.50
$G_{1,O,60}$ (60-column parking)	Runtime	55.21	77.29	314.81
	Flowtime	591,206.62	1,078,271.98	1,069,485.37
	Makespan	5,943.79	10,844.03	10,741.50
$G_{1,O,20}$ (20-column parking)	Runtime	104.71	122.82	314.84
	Flowtime	866,606.39	1,211,533.46	1,069,485.37
	Makespan	8,702.13	12,180.54	10,741.50

Next, we keep the same experimental setup as above, and shift the initial positions of AGVs to the pickup positions. Since the AGV is already in the pickup position, each AGV only needs to search out a valid path to reach its dropoffs. Therefore, it is a one-shot MAPF problem, and we compare it with the existing MAPF solutions SRP [41], CBS [38], and ECBS [3]. As shown in Table 2, SRP is second only to the OHSMD in terms of runtime, but produces paths of lower quality. Since SRP first determines the sequence of vertices to pass through, it makes the search directional and faster. However, SRP causes many tasks to move through the same vertices sequence, leading to congestion on certain paths, which increases the makespan and flowtime. The CBS/ECBS is able to solve conflict-free actions at each time, but it does not take kinematic constraints into account, so CBS/ECBS is less effective at flowtime and makespan. ECBS is faster than CBS, but its results are not guaranteed to be optimal. Compared with the CBS_0/$ECBS_0$, since CBS_m/$ECBS_m$ considers merging each individual moving action, both their flowtime and makespan are greatly improved. This also proves the importance of accounting for kinematics, as it is more likely to improve the average speed of AGv and the makespan.

Finally, we report OHSMD's distribution of each task's completion time and runtime in Figure 7 using the industrial 15021 tasks. The tasks are categorized by their pickup and dropoff's Manhattan distances. Generally, as the task becomes longer, it takes a longer time to complete and a longer time to compute. The worst-case complete time of each category is roughly the same as the next category's medium time, which demonstrates the stability of the result's quality. In terms of runtime, the worst case increases as the task becomes longer because it has a larger search space and conflicts. Nevertheless, the medium runtime increases more slowly, which explains the higher efficiency.

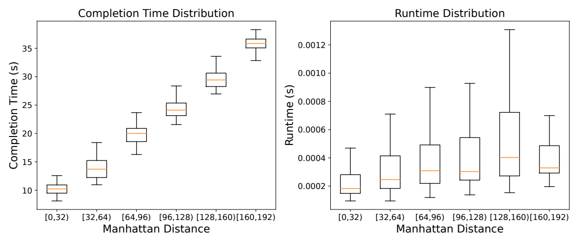

Figure 7: Distribution of Completion Time and Runtime

7.2.2 ***Lifelong Strategy.*** We test the lifelong strategies when the task set is large. We treat T_1 as a batch and copy it 10 times to get T_1'. We compared our algorithm with SRP, *Token Pass (TP)* [32] and CBS/ECBS. CBS/ECBS regards T_1' as 10 batch. For each batch, each AGV moves to the nearest pickup position and then to the corresponding drop-off position. So each batch needs to use the MAPF solver twice. Only after the tasks in each batch are completed will the next batch be continued. As shown in Table 3, OHSMD has the best makespan and flowtime, even though we only used the simplest fixed parking position strategy. The runtime of TP is slightly better than that of OHSMD, but the quality of the solution is similar to that of CBS/ECBS. The performance of CBS/ECBS is positively related to the number of batches and is easy to achieve lifelong on our synthetic data because every batch is consistent.

The last row shows the memory consumption. As analyzed in the space complexity, the memory size corresponds to the search space and action space. Specifically, the memory drops from GCAS's 31GB to OHSMD's 135MB, which demonstrates the effectiveness of the reduction in search space and action states and corresponds to the improved runtime.

As shown in Table 4, compared to $G_{1,O,100}$, the fixed parking strategy $G_{1,O,fix}$ is faster but has longer makespan. As the central buffer number decreases, performance degrades accordingly, with $G_{1,O,20}$ deteriorating to a level similar to the fixed parking. Unlike $G_{1,O,fix}$, the parking spaces in $G_{1,O,20}$ are located at the edges of the map. While reducing the central buffer can reduce congestion for certain tasks, it also increases the distance for AGVs to reach their next task. This validates the effectiveness of our flexible buffer strategy and the importance of setting buffers in each column.

7.2.3 ***Network Structure.*** We evaluate the impact of network topologies in Table 5. We first compare $G_{1,B}$ with $G_{1,B,diag}$, which further supports diagonal movements. Enabling diagonal movements reduces the makespan of both OHSMD and TP algorithms. However, as AGVs occupy two adjacent grids simultaneously when moving diagonally, this negatively affects SRP algorithms, which

Table 5: Network Topology

Graph	Performance	OHSMD	SRP	TP
$G_{1,B}$ (cardinal directions)	Runtime	**62.92**	123.29	898.43
	Flowtime	476,036.55	2,234,218.03	1,039,183.99
	Makespan	4,792.51	22,475.29	10,449.83
$G_{1,B,Diag}$ (cardinal and diagonal directions)	Runtime	376.99	164.31	3990.58
	Flowtime	**452,908.15**	3,988,646.84	997,480.41
	Makespan	**4,556.71**	40,021.21	10,038.38
G_{bm} (Central Shelf)	Runtime	172.92	**10.24**	63.69
	Flowtime	**48,744.84**	145,735.54	113,178.33
	Makespan	**534.52**	1,563.97	1,190.52

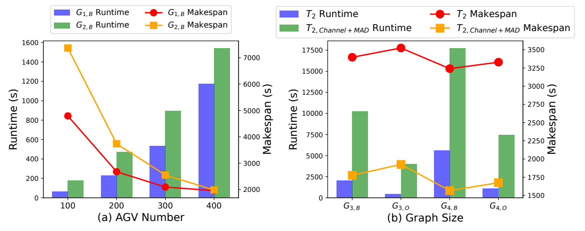

Figure 8: Scalability AGV Number and Network Size

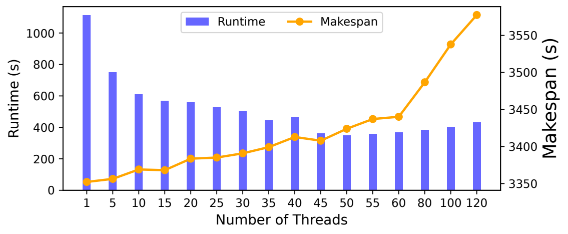

Figure 9: Scalability Thread Number.

first determine the grid sequence before resolving conflicts. The expansion of move directions leads to a larger search space and additional conflict resolution for two adjacent grids, so it has a higher computational cost and longer runtime.

In the central shelf warehouse, OHSMD runs slower than SRP and TP, and SRP is the fastest as it is optimized for this topology. Nevertheless, OHSMD achieves the smallest makespan, which is nearly 1/3 of SRP and 1/2 of TP. This is primarily due to the highly congested aisles between shelves, and OHSMD still figures out higher-quality paths while SRP simply regards the aisle as one contracted vertex and leaves the conflicts to post-processing.

7.2.4 ***Scalability Evaluation***. We first test the scalability with more AGVs with T_2 on G_1 and G_2. As shown in Figure 8-(a), as the AGV number grows, the network becomes congested with more conflicts, thus generating more VIs during path planning, which results in a runtime. On the other hand, more AGVs can run more tasks at the same time, so the makespan drops.

In real life, an order O_i consists of multiple tasks $\{\tau_j^i\}$ that share a common drop-off location d_i for packing together, so we also assess the impact of order pickup and drop-off distributions. 1) We evaluate the Minimum Average Distance (**MAD**) strategy [50]. For an order O_i with pickup locations $\{s_i^0, \ldots, s_i^k\}$, where each s_i^j is its column number, the drop-off's column number is computed as $d_i = \arg\min_{d \in D} \sum_{j=0}^{k} MD(d, s_i^j)$, with D being the set of possible drop-off locations and MD the Manhattan distance. 2) We organize the grids into vertical **channels** by grouping nearby columns, and assigning tasks from the same order to pickup locations within the same channel. Specifically, every 10 pickup positions form a channel, assigned based on task sorting, prioritizing those with fewer tasks already allocated.

Then, we test the scalability with network size and also the effectiveness of the Channel+MAD strategy. We set 300 AGVs on G_3 and 400 AGVs on G_4. As shown in Figure 8-(b), a larger network takes longer runtime and makespan as the search space is larger, even with more AGVs. It can also be demonstrated that the Channel+MAD method provides better flowtime and makespan. In addition, even though G_O has a longer makespan and flowtime than G_B, its runtime decreased significantly. The reason why runtime is much larger than makespan is that we set the system/AGV to work all the time, and there is no waiting for the task to be sent down to the system. This makes it fairer to compute and compare the makespan values. The runtime gap is even larger on larger networks, so a one-way network grid can be used to boost computational efficiency for some large warehouses.

7.2.5 ***Parallel Computation***. In $G_{4,O}$, although the makespan is around 3000s, its runtime exceeds 1000s. To reduce runtime, we parallelize path computing with k threads. After all threads finish, conflict-free paths are finalized, while conflicting ones are resolved through post-processing. Unlike SRP, which first computes paths without conflict consideration and adjusts them later, our method ensures that each thread produces conflict-free paths with the remaining $|A| - k$ AGVs. Figure 9 shows that as k increases, the makespan slightly increases (200s), while runtime initially decreases dramatically and then rises slightly. The best configuration reduces runtime by up to 760s, with only a 72s increase in makespan. The initial runtime reduction stems from concurrent task planning, whereas the increase is attributed to (1) potential inter-thread conflicts requiring additional replanning and (2) workload imbalance, where the slowest thread determines overall execution time.

8 CONCLUSION

In this paper, we introduced a conflict-free AGV routing system that is continuous in time, kinematic considered, and lifelong for tasks. Specifically, we propose two base search algorithms, General Conflict-Aware Search (GCAS) and Multi-Hop Conflict-Aware Search (MHCAS), together with two progressive search strategies (MHSC and OHSMD) to efficiently find the sequential optimal conflict-free paths. Additionally, we proposed the parking location buffer and task assignment strategy to support effective lifelong task scheduling for continuous tasks. Finally, we conduct comprehensive experimental studies to demonstrate our algorithm's high efficiency and effectiveness.

ACKNOWLEDGMENTS

Our research is funded by AIR@InnoHK research cluster of the Innovation and Technology Commission (ITC) of the HKSAR Government. The results presented in this paper within the scope of the research project 'Everything Factory - Ultra-flexible Factories based on Line-less Mobile Production' has come to a successful completion because of the support from ITC. We would like to express our sincere gratitude to them. This work is also supported by Natural Science Foundation of China #62202116 and Guangzhou municipality big data intelligence key lab #2023A03J0012.

REFERENCES

[1] [n.d.]. *Moving AI Pathfinding Benchmarks*. https://movingai.com/benchmarks/mapf/index.html

[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 349–360.

[3] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 5. 19–27.

[4] Robert Bogue. 2016. Growth in e-commerce boosts innovation in the warehouse robot market. *Industrial Robot: An International Journal* (2016).

[5] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, Oded Betzalel, David Tolpin, and Eyal Shimony. 2015. Icbs: The improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 6. 223–225.

[6] Frodo Kin Sun Chan, Yan Nei Law, Bonny Lu, Tom Chick, Edmond Shiao Bun Lai, and Ming Ge. 2022. Multi-Agent Pathfinding for Deadlock Avoidance on Rotational Movements. In *2022 17th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. IEEE, 765–770.

[7] Robert T Chien, Ling Zhang, and Bo Zhang. 1984. Planning collision-free paths for robotic arm among obstacles. *IEEE transactions on pattern analysis and machine intelligence* 1 (1984), 91–96.

[8] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[9] Chris Conlan, Teddy Cunningham, Gunduz Vehbi Demirci, and Hakan Ferhatosmanoglu. 2021. Collective shortest paths for minimizing congestion on temporal load-aware road networks. In *Proceedings of the 14th ACM SIGSPATIAL International Workshop on Computational Transportation Science*. 1–10.

[10] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[11] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International workshop on experimental and efficient algorithms*. Springer, 319–333.

[12] Roland Geraerts and Mark H Overmars. 2004. A comparative study of probabilistic roadmap planners. In *Algorithmic foundations of robotics V*. Springer, 43–57.

[13] Clara Gomez, Marius Fehr, Alex Millane, Alejandra C Hernandez, Juan Nieto, Ramon Barber, and Roland Siegwart. 2020. Hybrid topological and 3d dense mapping through autonomous exploration for large indoor environments. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 9673–9679.

[14] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[15] Wolfgang Hönig, TK Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. 2017. Summary: multi-agent path finding with kinematic constraints. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 4869–4873.

[16] William E Howden. 1968. The sofa problem. *The computer journal* 11, 3 (1968), 299–301.

[17] Ryota Kamoshida and Yoriko Kazama. 2017. Acquisition of automated guided vehicle route planning policy using deep reinforcement learning. In *2017 6th IEEE International Conference on Advanced Logistics and Transport (ICALT)*. IEEE, 1–6.

[18] Sertac Karaman and Emilio Frazzoli. 2011. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research* 30, 7 (2011), 846–894.

[19] Sven Koenig and Maxim Likhachev. 2005. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics* 21, 3 (2005), 354–363.

[20] Sven Koenig, Maxim Likhachev, and David Furcy. 2004. Lifelong planning A*. *Artificial Intelligence* 155, 1-2 (2004), 93–146.

[21] Steven LaValle. 1998. Rapidly-exploring random trees: A new tool for path planning. *Research Report 9811* (1998).

[22] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search.. In *IJCAI*, Vol. 2019. 442–449.

[23] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. 2021. Eecbs: A bounded-suboptimal search for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 12353–12362.

[24] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig. 2021. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 11272–11281.

[25] Lei Li, Wen Hua, Xingzhong Du, and Xiaofang Zhou. 2017. Minimal on-road time route scheduling on time-dependent graphs. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1274–1285.

[26] Lei Li, Sibo Wang, and Xiaofang Zhou. 2019. Time-dependent hop labeling on road network. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 902–913.

[27] Xiao Liang, Guanglei Meng, Yimin Xu, and Haitao Luo. 2018. A geometrical path planning method for unmanned aerial vehicle in 2D/3D complex environment. *Intelligent Service Robotics* 11 (2018), 301–312.

[28] Andrei Lissovoi and Carsten Witt. 2013. Runtime analysis of ant colony optimization on dynamic shortest path problems. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 1605–1612.

[29] Lixing Liu, Xu Wang, Xin Yang, Hongjie Liu, Jianping Li, and Pengfei Wang. 2023. Path planning techniques for mobile robots: Review and prospect. *Expert Systems with Applications* (2023), 120254.

[30] Hang Ma, Daniel Harabor, Peter J Stuckey, Jiaoyang Li, and Sven Koenig. 2019. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7643–7650.

[31] Hang Ma, Wolfgang Hönig, TK Satish Kumar, Nora Ayanian, and Sven Koenig. 2019. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7651–7658.

[32] Hang Ma, Jiaoyang Li, TK Satish Kumar, and Sven Koenig. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. 837–845.

[33] Mohammad Aijaz Mohiuddin, Salman A Khan, and Andries P Engelbrecht. 2016. Fuzzy particle swarm optimization algorithms for the open shortest path first weight setting problem. *Applied Intelligence* 45 (2016), 598–621.

[34] Afshin Oroojlooy and Davood Hajinezhad. 2023. A review of cooperative multi-agent deep reinforcement learning. *Applied Intelligence* 53, 11 (2023), 13677–13722.

[35] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*. 709–724.

[36] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proceedings of the VLDB Endowment* 13, 5 (2020), 602–615.

[37] Guillaume Sartoretti, Justin Kerr, Yunfei Shi, Glenn Wagner, TK Satish Kumar, Sven Koenig, and Howie Choset. 2019. Primal: Pathfinding via reinforcement and imitation multi-agent learning. *IEEE Robotics and Automation Letters* 4, 3 (2019), 2378–2385.

[38] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219 (2015), 40–66.

[39] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial intelligence* 195 (2013), 470–495.

[40] Dingyuan Shi, Yongxin Tong, Zimu Zhou, Ke Xu, Wenzhe Tan, and Hongbo Li. 2022. Adaptive Task Planning for Large-Scale Robotized Warehouses. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE.

[41] Dingyuan Shi, Nan Zhou, Yongxin Tong, Zimu Zhou, Yi Xu, and Ke Xu. 2023. Collision-Aware Route Planning in Warehouses Made Efficient: A Strip-based Framework. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE.

[42] Hamed Shorakaei, Mojtaba Vahdani, Babak Imani, and Ali Gholami. 2016. Optimal cooperative path planning of unmanned aerial vehicles by a parallel genetic algorithm. *Robotica* 34, 4 (2016), 823–836.

[43] David Silver. 2005. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, Vol. 1. 117–122.

[44] Trevor Standley. 2010. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 24. 173–178.

[45] Anthony Stentz. 1994. Optimal and efficient path planning for partially-known environments. In *Proceedings of the 1994 IEEE international conference on robotics and automation*. IEEE, 3310–3317.

[46] Jonas Stenzel, Dennis Lünsch, and Lea Schmitz. 2021. Automated topology creation for global path planning of large AGV fleets. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 3373–3380.

[47] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Kumar, et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 10. 151–158.

[48] Jiří Švancara, Marek Vlk, Roni Stern, Dor Atzmon, and Roman Barták. 2019. Online multi-agent pathfinding. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7732–7739.

[49] Qi Wang, Markus Wulfmeier, and Bernardo Wagner. 2016. Voronoi-based heuristic for nonholonomic search-based path planning. In *Intelligent Autonomous*

Systems 13: Proceedings of the 13th International Conference IAS-13. Springer, 445–458.

[50] Ruizhong Wu, Mengxuan Zhang, Shuxin Wang, Frodo Kin Sun Chan, Yan Nei Law, and Lei Li. 2025. A Lifelong Conflict-Aware AGV Routing System. In *Australasian Database Conference*. Springer, 447–462.

[51] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine* 29, 1 (2008), 9–9.

[52] Yehong Xu, Lei Li, Mengxuan Zhang, Zizhuo Xu, and Xiaofang Zhou. 2023. Global routing optimization in road networks. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2524–2537.

[53] Alexander Zelinsky. 1992. A mobile robot navigation exploration algorithm. *IEEE Transactions of Robotics and Automation* 8, 6 (1992), 707–717.

[54] Han-ye Zhang, Wei-ming Lin, and Ai-xia Chen. 2018. Path planning for the mobile robot: A review. *Symmetry* 10, 10 (2018), 450.

[55] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic hub labeling for road networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 336–347.

[56] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-hop labeling maintenance in dynamic small-world networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 133–144.

[57] Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2021. An experimental evaluation and guideline for path finding in weighted dynamic network. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2127–2140.