Walk, Stop, Count, and Swap: Decentralized Multi-Agent Path Finding With Theoretical Guarantees

Hanlin Wang and Michael Rubenstein

***Abstract*—For multi-agent path finding (MAPF) problems, finding the optimal solution has been shown to be NP-Complete. Here we present WSCaS (Walk, Stop, Count, and Swap), a decentralized multi-agent path-finding algorithm that can provide theoretical completeness and optimality guarantees. That is, WSCaS is able to deliver a worst case $\mathcal{O}(1)$-approximate distance-optimal solution to MAPF instances on square grids without narrow passages. Moreover, the algorithm's cost is independent of the swarm's size with respect to computation complexity, memory complexity, as well as communication complexity, therefore the algorithm can scale well with the number of agents in practice. The algorithm is executed on 1024 simulated agents as well as 100 physical robots, the results show that the WSCaS is robust to real-world non-idealitys.**

***Index Terms*—Path planning for multiple mobile robots or agents, distributed robot systems, swarms.**

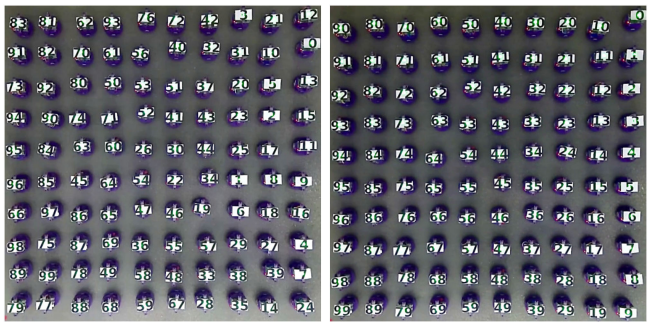

Fig. 1. Still images from a 100 robot experiment where 100 robots moved from a random configuration (*left*) to a column-major ordered configuration (*right*).

I. Introduction

RECENTLY, the multi-agent path finding (MAPF) problem has received a lot of attention for its extensive applications, including shape formation [1], automated warehouses [2], and more [3]. A general formulation of this problem is to compute a set of collision-free paths for a set of agents along which they can move from an initial configuration to a desired configuration.

The MAPF problem has been shown to be PSPACE-hard [4], and this conclusion holds even for the unlabeled case [5]. The hardness of the problem can be reduced to NP-complete by enforcing agents move along a graph [6]. There are multiple ways to evaluate the solution's quality [6], among them two metrics are frequently used in the literature: *makespan* and *total distance*. It has been shown in [6] that computing a solution that minimizes either one of these two metrics is NP-complete, which suggests that the computation of exact optimal solution is intractable in practice.

In the past, many algorithms that can obtain the optimal solution for MAPF problem have been presented. Some methods compute the solution by searching the joint configuration space of the team [7], [8]. In [9], authors established the equivalence relationship between MAPF and multiflow, and encoded the problem using the ILP model. These methods can provide safety, correctness and optimality guarantees. However, unsurprisingly, it is computationally prohibitive for them to work on large-scale swarms, as finding optimal solution to MAPF is NP-complete.

Past efforts try to reduce problem's computational complexity by shrinking the search space. One strategy is computing the paths sequentially, the agents with high priority are treated as dynamic obstacles by the agents with low priority [10], [11]. Another strategy is computing each agent's path independently, and resolve the conflicts along the way [12]. These strategies can significantly reduce problem's computational complexity, but often yield unbounded suboptimal solution and lack completeness guarantee. It is shown in [13], [14] that some pre-defined primitives can help to avoid unnecessary exploration of search space while maintain the completeness of the solution, however, the solutions often come with unbounded path length. PAF [15] can produce a worst case $\mathcal{O}(1)$-approximate time-optimal solution on a 2D or 3D holeless grid in quadratic runtime, but it is designed specifically for holeless grid therefore can not solve MAPF instances in complex environments directly.

As expected, the other strategy that helps to break problem's computational bottleneck is distributing the computational cost among the agents. The computational complexity of a decentralized system can be made independent of its scale in principle [16]. Moreover, compared to centralized methods, decentralized methods are inherently more robust to failures and uncertainties, and can be deployed in the environments where centralized method is hard to come by. Some algorithms formulate decentralized multi-agent path finding (DMAPF) problem as

Manuscript received September 10, 2019; accepted December 30, 2019. Date of publication January 17, 2020; date of current version January 30, 2020. This letter was recommended for publication by Associate Editor F. Arrichiello and Editor Nak Young Chong upon evaluation of the reviewers' comments. *(Corresponding author: Hanlin Wang.)*

H. Wang is with the Department of Computer Science, Northwestern University, Evanston, IL 60601 USA (e-mail: h.w@u.northwestern.edu).

M. Rubenstein is with the Department of Computer Science and the Department of Mechanical Engineering, Northwestern University, Evanston, IL 60208 USA (e-mail: rubenstein@northwestern.edu).

This letter has supplementary downloadable material available at https://ieeexplore.ieee.org, provided by the authors.

Digital Object Identifier 10.1109/LRA.2020.2967317

reactive control or local coordination problems [17]–[19], these methods scale well with the size of team but lack optimality and completeness guarantee. Some algorithms make use of *Token Passing* mechanism to guarantee solution's correctness [20], but the guarantee of completeness replies on the assumption that the agents' communication network is fully connected and lossless [21], or the communication channel can transfer the data packet with unbounded size during every communication round [20]. In general, these ideal assumptions can be difficult to guarantee when the algorithms are implemented in the real world.

In this paper, we present a decentralized multi-agent path-finding algorithm: WSCaS (Walk, Stop, Count, and Swap). In WSCaS, each agent moves along a independently-planned path to its goal. Along the way, agents moves in a "listen-think-walk" manner so as to avoid the collisions, and use the *figure-8 swap* [15] to resolve the deadlocks. Theoretical analysis shows that WSCaS can produce worst case $\mathcal{O}(1)$-approximate distance-optimal solutions to MAPF instances on square grids without narrow passages, and the algorithm's cost is independent of swarm size with respect to computational complexity, memory complexity, and communication complexity. Moreover, both the physical experiments and simulations showed that our algorithm is robust to real-world non-idealitys, such as communication errors, sensing error, and imperfect robot motion. All these desirable features make the WSCaS algorithm a solution to many real-world applications such as automated warehouse [2] and airport ground-traffic control [3].

II. Preliminaries

A. MAPF on Search Graphs: Problem Statement

A MAPF instance can be defined by a tuple $(\mathcal{G}, \mathcal{A})$, where $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ is an undirected graph and $\mathcal{A}$ is a set of mobile agents. Each agent $a_i \in \mathcal{A}$ can be characterized by a tuple (b_i, g_i) where b_i and g_i are agent a_i's initial position and goal position, respectively. Let t be the time, and s_i^t be agent a_i's position at time t, a *schedule* for agent a_i is defined to be a sequence of vertices $S_i = (s_i^0, \ldots, s_i^{T-1})$ such that: *(i)* $\forall s_i^t \in S_i, s_i^t \in \mathcal{V}$; *(ii)* $\forall s_i^t, s_i^{t+1} \in S_i, s_i^t = s_i^{t+1}$ or $(s_i^t, s_i^{t+1}) \in \mathcal{E}$.

The task of MAPF is finding a *schedule* S_i for each agent a_i along which it can move from b_i to g_i without colliding with the others. Specifically, given a MAPF instance $(\mathcal{G}, \mathcal{A})$, a solution is a set of *schedule* $\mathcal{S} = \{S_0, \ldots, S_{|\mathcal{A}|-1}\}$ such that:

- The solution is deadlock free, namely, there exists $T \in \mathbb{Z}^+$ s.t. $\forall S_i \in \mathcal{S}$, $|S_i| = T$, moreover, $s_i^0 = b_i, s_i^{T-1} = g_i$;
- There is no collision on vertices, namely, for any pair of *schedule* $S_i, S_j \in \mathcal{S}, \forall 0 \leq t < T$, $s_i^t \neq s_j^t$;
- There is no collision on edges, namely, for any pair of *schedule* $S_i, S_j \in \mathcal{S}, \forall 0 \leq t < T$, $(s_i^t, s_i^{t+1}) \neq (s_j^{t+1}, s_j^t)$.

B. Swap *on Figure-8 Graph*

As defined in last section, in a feasible solution, two agents located at two adjacent vertices cannot swap positions directly. In [15], the authors proposed a sequence of intermediate moves that allow two adjacent agents to swap positions within $\mathcal{O}(1)$ steps (Fig. 2). For the sake of description, in the rest of the paper, we slightly abuse the terminology by using *figure-8* graph to represent the *figure-8* graph that contains only six vertices.

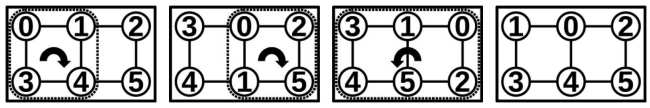

Fig. 2. Graphical illustration of a *figure-8 swap*. In this example, agent 1 and 0 swap their positions in 3 steps. In each step, the set of agents in dotted bounding box simultaneously move in the direction indicated by arrows.

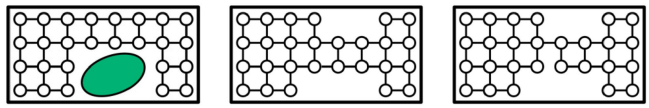

Fig. 3. *(left)* A workspace with obstacle (the object in green) and the corresponding grid. (*middle* and *right*) A graphical illustration of a *swappable* grid *(middle)* and a grid that is not *swappable (right)*. The difference between them is that the grid on the right has a bridge in the middle, hence it is not *swappable*.

C. Environment Discretization

In this paper, we use the square grid map to model the environment. Given a square grid map, each unblocked cell c_i on the map will be a vertex v_i on the search graph $\mathcal{G}$, and $(v_i, v_j) \in \mathcal{E}$ if the unblocked cells c_i and c_j on the map are horizontally or vertically adjacent to each other. Here, we are interested in a class of grid called the *swappable* grid.

Definition 1: Let $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ be a grid with holes, $\mathcal{G}$ is **swappable** if $\forall (v_0, v_1) \in \mathcal{E}$, $\exists v_2, v_3, v_4, v_5 \in \mathcal{V}$ such that $v_0, v_1, v_2, v_3, v_4, v_5$ together can form a figure-8 graph [15].

A less formal but more intuitive description of *swappable* grid is: a grid where the narrowest passage can allow at least two agents to go through in parallel. A graphical illustration of a **swappable** grid is shown in Fig. 3.

D. Agent Model

We treat each agent as a disk with a finite radius r. Let l be length of edge on $\mathcal{G}$, we assume $r \leq \frac{\sqrt{2}}{4} l$ so no collision will occur when two agents move along two adjacent orthogonal edges concurrently. Each agent constantly transmits messages at a frequency f_{comm} such that it can transmit at least one message to the neighbors during each time step, moreover, the transmitted data can be received by any other agent lying within the range R. We assume $R \geq 4\sqrt{2}l$ so as to allow each agent to sense whether two *figure-8 swaps*' footprints are overlapped with each other. Initially, each agent has no knowledge of any global information about the swarm, the only information known *a priori* is: the grid map $\mathcal{G}$, its own initial position b_i and its own goal position g_i. To simplify the analysis, we assume that all the agent agree on the same clock, including phase and frequency. Note that this assumption can be relaxed when the algorithm is implemented in the physical world, which is shown by our physical experiments.

III. Approach

In the WSCaS algorithm, each agent a_i first uses A* to generate a nominal path $\mathcal{P}_i = \{p_{i_0}, \ldots, p_{i_n}\}$ from b_i to g_i on $\mathcal{G}$, where $p_{i_0} = b_i$, $p_{i_n} = g_i$, and $\forall p_{i_j}, p_{i_{j+1}} \in \mathcal{P}_i, (p_{i_j}, p_{i_{j+1}}) \in \mathcal{E}$, then walks along the path $\mathcal{P}_i$ towards the goal g_i. Along the way, the agent uses the local communication to detect the traffic condition ahead of it and acts accordingly so as to avoid the collisions and resolve deadlocks. It's tempting to confuse S_i with $\mathcal{P}_i$, the difference between them is: path $\mathcal{P}_i$ is a reference path for agent

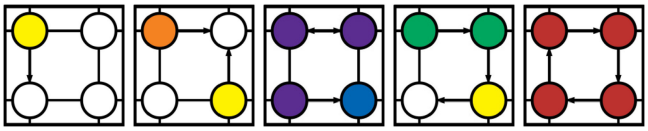

Fig. 4. Graphical illustration of 5 different types of traffic conditions. The empty circles are the vertices that are not occupied by any agent, and filled circles are the agents. We use the colors to indicate the type of traffic condition in each agent's view. Specifically, yellow-0, orange-1, purple-2, green-3, red-4, and the color blue indicates that the agent arrived at its goal already. The arrow points to agent's next waypoint on its nominal path. (*type-0*) Agent's next waypoint on nominal path is open, and there does not exist any other competitor that intends to go to the same vertex such that the agent's position $\succ$ competitor's position, where $\succ$ denotes the lexical order on $\mathbb{R}^2$ space. (*type-1*) Agent's next waypoint on nominal path is open however there is a competitor who intends to go to the same vertex and the agent's position $\succ$ the competitor's position. (*type-2*) Two adjacent agents intend to swap position (top), or agent waits for a agent who arrived at it goal already (bottom). (*type-3*) Agent is in a queue that is not circular. (*type-4*) Agent is in a circular queue.

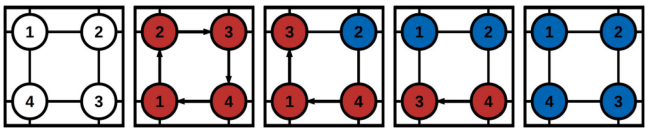

Fig. 5. Illustration of the resolution of a deadlock case. The color shows the same information as in Fig. 4, and the number indicates each agent's goal. From left to right: *(1)* The agents' goals, each goal is labeled with a number; *(2–5)* The agents get stuck in a deadlock case in *(2)*, and one agent (agent 3) traverses the circular queue backwards via *figure-8 swap* so as to resolve the deadlock in *(3–5)*.

a_i whereas the schedule S_i shows the a_i's actual position at each time step.

When an agent a_i executes its nominal path $\mathcal{P}_i$, for each step, in a_i's view, there are five possible traffic conditions, see Fig. 4 for a graphical illustration. If the traffic condition is *type-0*, then a_i can *walk* to the next waypoint on $\mathcal{P}_i$; if the traffic condition is *type-1 or type-3*, a_i needs to *stop* at the current waypoint so as to avoid colliding with the others; if the traffic condition is *type-2*, which is a deadlock case that involves two agents, these two agents will resolve this deadlock via *swap*; lastly, if the traffic condition is *type-4*, i.e., a_i gets stuck in a deadlock case that involves more than two agents, then one agent involved in this deadlock will traverse this circular queue backwards via *swap* so as to resolve the deadlock. One can observe that when this agent finish the backward traverse, all the agents who got stuck in this deadlock have moved at least one step forward on the their nominal paths, see Fig. 5 for an example of the resolution of a deadlock.

It is worth noting that each agent can sense only the information within a fixed range R, whereas the deadlock chain (the circular queue in *type-4* traffic condition) could be arbitrarily long, thus a_i cannot use the local communication to sense the difference between *type-3* and *type-4* traffic conditions directly. In our algorithm, the detection of deadlock (t*ype-4* traffic condition) is achieved by the **count** protocol. In the **count** protocol, each agent holds a number *alpha* and constantly transmits its own *alpha* to the neighbors. If a_i is not blocking any other's way, then it will set $alpha_i$ to 0; otherwise, if a_i is blocking the other agent a_j's way, then a_i will set $alpha_i$ to $alpha_j + 1$. One can observe that: if a_i is located in a circular queue, $alpha_i$ will increase infinitely, and if a_i is not located in a circular queue, then $alpha_i$ will be no greater than $|\mathcal{V}|$, i.e., the number of vertices on search graph $\mathcal{G}$. By checking whether its *alpha* is greater than $|\mathcal{V}|$, a_i can sense whether it is located in a circular queue.

Algorithm 1: WSCaS Planner Component.

```
Input: G, b_i, g_i
Use A* to generate P_i
tr ← 0, action ← stop, s_curr ← b_i, p_next ← s_curr
alpha ← 0, role ← Pedestrian, bid_helper ← s_curr, t ← 0
while True do
    t ← t + 1
    if action is completed then
        if role is Samaritan then
            if s_curr is p_next then
                role ← Pedestrian
        action ← get_next_action (role, tr, action)
        if action is walk then
            s_curr ← p_next
        if action is lead_swap then
            s_curr ← position of swap peer
        if role is Pedestrian then
            if action is walk or lead_swap then
                alpha ← 0
                if p_next is not g_i then
                    p_next ← s_curr's successor on P_i
    s_i^t ← get_next_step (action, s_curr)
    move to s_i^t and meanwhile executes the following:
        if tr is not 4 then
            bid_helper ← s_curr
        tr ← 0
        repeat
            if role is Pedestrian and receive a msg_in then
                count(msg_in)
        until next time step
```

Algorithm 2: WSCaS Broadcasting Component.

```
while True do
    msg_out←{s_curr, p_next, alpha, tr, action}
    /* Note that if the action is *_swap, the message will also contain
    the swap's footprint and labeling number                          */
    transmit msg_out
    sleep 1/f_comm
```

The implementation of WSCaS algorithm generally consists of two components: planner component and broadcasting component. The broadcasting component constantly transmits messages to neighbors at a constant frequency f_{comm}. The planner component handles the path planning task. In practice, these two components can be implemented using two separate threads that communicate through the shared memory. The sketches of these two components are shown in Algorithms 1 and 2. Note that all the variables are thread-public.

The rest of the section is organized as follows. To help the readers to have a clear grasp of WSCaS, we first explain the key variables used in the algorithm in Section III-A, and then explain how each subroutine works in detail in Section III-B. At the end, in Section III-C, we put everything together and walk through the complete algorithm.

A. Variables

- *tr:* Agent uses variable *tr* to encode the traffic condition. The possible values for variable *tr* are: 0, 1, 2, 3, 4, corresponding to *type-0, 1, 2, 3, 4* traffic conditions in Fig. 4.
- $s_{curr}, p_{next}, alpha$*:* These three variables are used in **count** protocol to identify the traffic condition. s_{curr} is agent's position after finishing current action, p_{next} is agent's next

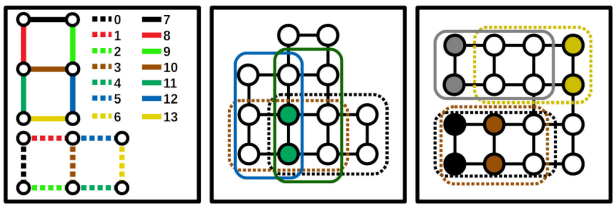

Fig. 6. (*left*) graphical illustration of the rule how each swap pair is labelled. The swap pair is encoded according to its position on the *figure-8* graph as well as *figure-8* graph's orientation. (*middle*) A pair of agents intend to swap their position. For this swap pair, there are multiple choices of footprints to perform the swap on. The *figure-8* graphs bounded by boxes are the agent's possible choices, in this case, the agent has four different choices, and according to the labeling rule introduced in the *(left)*, the agent picks the footprint bounded by black dotted box, as this choice gives the swap a labeling number of 0. (*right*) A time step where three swaps are being proposed and one swap is being executed. The gray circles are the agents that are in executing phase, and the green circles are the agents that are in proposing phase. The gray box shows the ongoing swap's footprint and the colored boxes show the swaps which are being proposed. In this case, only the swap pair in left lower corner (black) succeeds in proposing.

Algorithm 3: Function **count**.

```
   Input: msg_in
1  if msg_in.role is Pedestrian then
2      if msg_in.p_next is s_curr and p_next is not msg_in.s_curr then
3          if alpha > |V| then
4              tr ← 4
5          if tr is 4 and msg_in.tr is 4 then
6              if msg_in.bid_helper ≻ bid_helper then
7                  bid_helper ← msg_in.bid_helper
8              if msg_in.bid_helper is s_curr then
9                  role ← Samaritan
10         if msg_in.alpha + 1 > alpha then
11             alpha ← msg_in.alpha + 1
12 if tr is 0 then
13     if msg_in.p_next is p_next and s_curr ≻ msg_in.s_curr then
14         tr ← 1
15     if msg_in.s_curr is p_next then
16         if msg_in.p_next is s_curr or msg_in.s_curr is msg_in.g_i then
17             tr ← 2
18         else
19             tr ← 3
20     if msg_in.action is *_swap and p_next is located within that swap's
         footprint then
21         tr ← 3
```

waypoint on the nominal path $\mathcal{P}$, and *alpha* is for detecting the deadlocks.

- *action:* This variable has *walk*, *stop*, *propose_swap*, *lead_swap*, *follow_swap* five possible values, representing agent's five different actions. The description of each action is shown as follows:
  -*walk:* Agent moves from current position to the next waypoint on the nominal path.
  -*stop:* Agent stays at current position for one time step.
  -**_swap:* These three actions allow agents to execute *figure-8 swap* in a decentralized fashion. Due to the lack of global coordination, to avoid the cases where footprints of two set of swap's footprints overlap with each other, we here divide a single swap round into two phases: proposing phase and executing phase. When two agents intend to initiate a swap round, they first need to decide which *figure-8* graph they are going to perform swap on, as there may be multiple *figure-8* graphs in the neighborhood overlapped with them. We enforce all the agents to follow the same rule to label each swap with an integer, a graphical illustration of our labeling rule is shown in Fig. 6 *(left)*. Note that the labeling rule presented in Fig. 6 is not the only option, it is possible to improve the algorithm's average-case performance by selecting a different labeling rule according to the topology of the search graph $\mathcal{G}$. When agents initiate a swap, they always choose the feasible *figure-8* graph that gives the swap the lowest labeling number. An example of footprint selection is shown in Fig. 6 *(middle)*. After this, these two agents enter the proposing phase and start executing the action *propose_swap*. In proposing phase, at each time step, the agent transmits the proposed swap to neighbors, meanwhile keeps checking if the proposed swap's footprint overlap with footprint of any other ongoing swap or any other proposing swap with a lower labeling number. At the end of the time step, if none of these two events has happened, then the proposed swap is allowed and these two agents enter executing phase to execute the action *lead_swap*, otherwise, the agents stay in proposing phase and continue executing the action *propose_swap* at next time step. An example of the proposing phase is shown in Fig. 6 *(right)*. In the executing phase, the agents who initiated the swap executes the action *lead_swap*. When these agents execute the action *lead_swap*, at the first time step, they stay at original position and ask others located on the swap's footprint to execute the action *follow_swap* to cooperate with their motion. After this, all agents located on the swap's footprint cooperatively perform the *figure-8 swap* until the swap is completed.
- *role*, *bid_helper:* As shown in Fig. 5, our deadlock resolution strategy is to make one and only one agent traverses the circular waiting chain backwards via swap. The variable *role* is used to separate the agents who travel backwards from the regular ones. This variable has two possible values: *Pedestrian* and *Samaritan*, which represent the regular agents, and the agents who traverses backwards, respectively. In each deadlock case, a decentralized bidding protocol is used to guarantee that there is one and only one agent becoming *Samaritan* agent, and the variable *bid_helper*, which is a x-y pair representing a vertex's position on graph, assists this process. This bidding protocol is part of **count** protocol and the detailed description is shown in Section III-B.

B. Subroutines

- **count**: Agent uses **count** protocol to determine the traffic condition and its *role*. The protocol is shown in Algorithm 3. *Type-1, type-2,* and *type-3* traffic can be identified directly using the local observation (Algorithm 3 Line 12–21). We call agent *a* is agent *b*'s *next-in-line* if *a* gets blocked by *b*. To distinguish *type-4* traffic condition from *type-3* traffic, when one agent a_i gets blocked by another agent, say a_j, it keeps updating the $alpha_i$ using the *alpha* from its *next-in-line*s, if there is any, and then transmits its own $alpha_i$ so the agent a_j can have $alpha_j$ updated as well. The updating rule is shown in Algorithm 3 (Line 10-11). If an agent's *alpha* is greater than $|\mathcal{V}|$, then it sets its *tr* to 4, i.e., agent senses the deadlock. After agent's *alpha* becomes 4, the agent starts to bid to be the *Samaritan* (Algorithm 3 Line

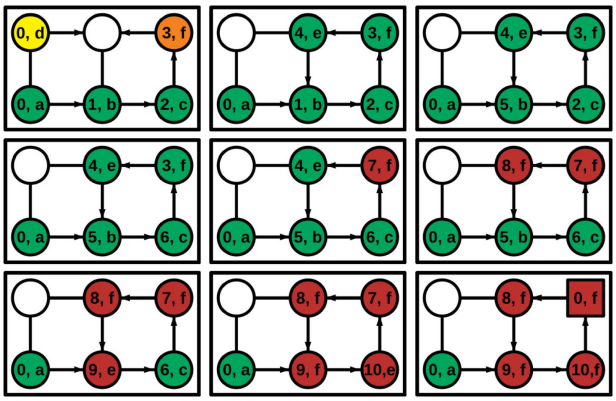

Fig. 7. Illustration of the **count** protocol. The figures are ordered left-to-right, top-to-bottom. The shape shows each agent's role, where square is *Samaritan* and circle is *Pedestrian*. The color shows each agent's *tr*, specifically, yellow-0, orange-1, green-3, red-4. The number and the letter on each agent indicate agent's *alpha* and *bid_helper*, where: a = (0, 0), b = (0, 1), c = (0, 2), d = (1, 0), e = (1, 1), f = (1, 2). In this example, $|\mathcal{V}|$ is 6.

5-9). When an agent bids to be *Samaritan*, it repeatedly executes the following: it first compares its own *bid_helper* with its *next-in-line*'s *bid_helper*, sets its *bid_helper* to be the one that is lexically greater, and then pushes its updated *bid_helper* forwards along the queue. An agent outbids the others, i.e., becomes a *Samaritan* agent, if it detects that one of its *next-in-line*'s *bid_helper* equals its s_{curr}. A minimal working example of the **count** protocol is shown in Fig. 7.

- **get_next_action**: Agent uses **get_next_action** function to select action to perform according its states. Note that the action *lead_swap* and the action *follow_swap* take multiple time steps to complete, therefore before executing the **get_next_action** function, the agent needs to check whether the last action is completed yet (Algorithm 1 Line 6). Each agent first checks if any other agent needs its cooperation to perform swap, and if so, it executes the action *follow_swap* (Algorithm 4 Line 1–2). For a *Pedestrian* agent, it executes the action *walk* when *tr* is 0 (Algorithm 4 Line 8-9), and executes the action *stop* when *tr* is 1, 3, or 4, so as to avoid collisions. There are two cases where the agent executes the action *propose_swap* and *lead_swap* (Algorithm 4 Line 3–7, 10–14), one is when agent is a *Samaritan* agent, and the other one is when agent is in *type-2* traffic condition.
- **get_next_step**: Agent uses this function to obtain the position where it will move to at the current time step.

C. Overview

In the planner component (Algorithm 1), initially, agent initializes the variables to certain values (Algorithm 1 Line 2–3) so as to act conservatively at the first time step, as it has not received any message from neighbors yet. After the initialization phase, agent enters the main loop (Algorithm 1 Line 5-28). Each iteration in main loop is a time step. At each iteration, agent first checks if the last action is completed (Algorithm 1 Line 6), if so, it use **get_next_action** to select the new *action* according to its states, including *tr* and *role* (Algorithm 1 Line 10). If agent is currently in *Samaritan* role, it also needs to check if the backward traverse is completed, and if so, it then changes its *role* back to *Pedestrian* (Algorithm 1 Line 7-9). Once agent gets the new action, it updates its states accordingly

Algorithm 4: Function **get_next_action**.

```
Input: role, tr, action
Output: action
1  if agent was recruited by lead_swap at last time step then
2      return follow_swap
3  if role is Samaritan then
4      if last action is propose_swap and proposing succeeded then
5          return lead_swap with its next-in-line whose tr is 4
6      else
7          return propose_swap with its next-in-line whose tr is 4
8  if tr is 0 then
9      return walk
10 if tr is 2 then
11     if last action is propose_swap and proposing succeeded then
12         return lead_swap with the agent blocking the way
13     else
14         return propose_swap with the agent blocking the way
15 return stop
```

right after (Algorithm 1 Line 11–19). Note that only the action *walk* and *lead_swap* can bring the agent to a new position after the action is completed, therefore s_{curr}, p_{next} and *alpha* are updated only when the new action is one of these two. Moreover, since the *Samaritan* agent needs to use the p_{next} to detect whether the backward traverse is completed (Algorithm 1 Line 7-9), only the *Pedestrian* agent updates its p_{next} (Algorithm 1 Line 18-19). After these, agent uses **get_next_step** function to determine the position where it will be at current time step, moves to this position, meanwhile uses the incoming messages to update its states (Algorithm 1 Line 22–28). The agent will use these updates next time when it runs the **get_next_action** function.

IV. Theoretical Results

A. Completeness

It is straight forward to examine that WSCaS is safe, i.e., the algorithm is collision-free, as long as each agent is able to successfully deliver at least one message to its neighbors within each time step. Next, in this section we show that it is also deadlock free.

Definition 2: Let d_i^t be the distance from agent a_i's current position to the goal, specifically:

1) if $role_i^t = Pedestrian$ and $s_{curr_i}^t$ is not adjacent to g_i, then d_i^t equals the distance between $s_{curr_i}^t$ and g_i on $\mathcal{P}_i$;
2) if $role_i^t = Pedestrian$ and $s_{curr_i}^t$ is adjacent to g_i, then $d_i^t = 1$;
3) if $role_i^t = Samaritan$, then $d_i^t = d_i^{t^*}$, where $t^* = \sup\{t' | t' < t, role_i^{t'} = Pedestrian\}$.

We define the swarm's **distance to the goal** at time t D^t as:

$$D^t = \sum_{i=0}^{|\mathcal{A}|-1} d_i^t$$

In WSCaS, agent may deviate from its nominal $\mathcal{P}_i$, for example, the deviation occurs when a *Samaritan* agent resolves the deadlock using *figure-8* swap. The case 2 and 3 in definition 2 help to make the objective to be well-defined in all the possible cases.

Lemma 1: For any agent a_i, d_i^t is well-defined, moreover, $d_i^t = 0$ only if $s_{curr_i}^t = g_i$ and $role_i^t = Pedestrian$.

Proof: See Section VII-A in [22]. ■

Lemma 2: If $D^t \neq 0$, then $D^t \geq D^{t+1}$.

Proof: Lemma 2 suggests that all the operations defined in the WSCaS will make D non-increasing. We prove lemma 2 via investigating the changes of agent's s_{curr} and *role*. We here outline all the events that involve the changes of these two variables:

a) agent executes action *walk:* When an agent a_i executes action *walk*, d_i decreases.

b) agent executes action *lead_swap:* Here we classify the *figure-8 swap* into three types: *type-1* swap is the swap happens between two agent blocking each other, as shown in Fig. 4 *(middle-top)*; *type-2* swap is the swap happens between an ongoing agent and an agent that is located at its goal, as shown in Fig. 4 *(middle-bottom)*; *type-3* swap is the swap happens between a *Samaritan* agent and its next-in-line. One can see that *type-1* swap and *type-3* swap will make D strictly decrease, whereas *type-2* swap does not change D.

c) agent's *role* changes from *Pedestrian* to *Samaritan:* This event does not change D by definition.

d) agent's *role* changes from *Samaritan* to *Pedestrian:* An agent can change its *role* from *Samaritan* to *Pedestrian* only after making one *net* step forward on its nominal path, as stated in Algorithm 1 (Line 8-9), thus D decrease by 1 in this case. ■

Theorem 1: Given a MAPF instance $(\mathcal{G}, \mathcal{A})$, let $len(\mathcal{P}_i)$ be the length of a_i's nominal path $\mathcal{P}_i$, figure-8 swap will occur $\mathcal{O}(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))$ times.

Proof: First, *type-1* and *type-3* swap can occur $\mathcal{O}(D^0)$ times. This is because each time when *type-1* or *type-3* swap occurs, D strictly decrease, additionally, D will never increase by lemma 2, thus these two types of swap can happen no more than D^0 times.

In order to trigger *type-2* swap, one agent a_j needs to be located at its goal g_j and the other agent a_i need to be **at least two steps** from g_i on $\mathcal{P}_i$, moreover, these two agent both need to be *Pedestrian* agents. One helpful observation here is that: for each agent a_i, on each waypoint that is **at least two steps** away from its goal, it can trigger *type-2* swap no more than one time, which suggests that the *type-2* swap can happen $\mathcal{O}(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i) - 2|\mathcal{A}|)$ times. As a result, the *figure-8 swap* will happen $\mathcal{O}(D^0) + \mathcal{O}(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i) - 2|\mathcal{A}|) = \mathcal{O}(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))$ times, completing the proof. ■

Theorem 2: Given a MAPF instance $(\mathcal{G}, \mathcal{A})$, if $\mathcal{G}$ is **swap-pable**, then:

$$\forall t, \text{if } D^t \neq 0, \text{then } D^t > D^{t+\mathcal{O}(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i)+|\mathcal{V}|)}.$$

Proof: Theorem 2 suggests that *D* will strictly decrease within finite amount of time. See Section VII-B in [22]. ■

Combining theorem 2 and lemma 1, we have that the WSCaS enables the agents to arrive at their goals within finite amount of time, in the other words, WSCaS is complete.

B. Performance

In this section, we study algorithm's worst case performance. Specifically, we are interested in two metrics: *total distance* and *makespan* [6], which evaluate solution's quality with respect to distance and time, respectively.

Theorem 3: Given a MAPF instance $(\mathcal{G}, \mathcal{A})$, if $\mathcal{G}$ is **swap-pable**, then the WSCaS can produce a worst case $\mathcal{O}(1)$-approximate distance-optimal solution.

Proof: Given a MAPF instance $(\mathcal{G}, \mathcal{A})$, since each $\mathcal{P}_i$ is planned independently via A*, the minimal total travel distance is $\Omega(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))$. On other other hand, *figure-8 swap* can occur $\mathcal{O}(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))$ times by theorem 1, moreover, since action *walk* makes *D* strictly decrease, and *D* is non-increasing by lemma 2, action *walk* can occur $\mathcal{O}(D^0)$ times. Therefore, the total travel distance incurred by WSCaS is $\mathcal{O}(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))$. From this it follows that:

$$\text{distance approximation ratio} = \frac{\mathcal{O}(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))}{\Omega(\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))} = \mathcal{O}(1)$$

■

Theorem 4: Given a MAPF instance $(\mathcal{G}, \mathcal{A})$, if $\mathcal{G}$ is **swap-pable**, then WSCaS can produce a solution with worst case $\mathcal{O}(|\mathcal{V}| \sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))$ makespan.

Proof: The only one case where no agent can move is that: each agent gets either stuck in or blocked by a circular queue. On the other hand, in this case, a agent will become *Samaritan* agent in $\mathcal{O}(|\mathcal{V}|)$ amount of time, then starts to resolve the deadlock via *type-3* swap. This conclusion, in combination with the result we obtained in theorem 3, suffices to show the worst case *makespan* is $\mathcal{O}(|\mathcal{V}| \sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i))$. ■

C. Complexity

Theorem 5: The cost of WSCaS is independent of the swarm's size with respect to memory complexity, communication complexity, and computation complexity.

Proof: The algorithm's memory complexity is dominated by the memory to store the input search graph $\mathcal{G}$ and the memory to generate the nominal path (Algorithm 1 Line 1), thus the WSCaS's memory complexity is independent of $|\mathcal{A}|$. On the other hand, during each time step, each agent will transmit f_{comm} amount of messages with a length of $\mathcal{O}(1)$ (Algorithm 2 Line 2), hence the algorithm's communication complexity, i.e., the amount of data transmitted in a unit of time, is $\mathcal{O}(f_{comm})$, which is independent of swarm size $|\mathcal{A}|$ as well.

To investigate WSCaS's computation complexity, we decompose the overall computation complexity into two parts: initialization complexity (Algorithm 1 Line 1-3) and motion planning complexity (Algorithm 1 Line 5-28). The initialization complexity is dominated by the computational cost to generate the nominal path, $\mathcal{O}((|\mathcal{E}| + |\mathcal{V}|) \log |\mathcal{V}|)$, which is independent of $|\mathcal{A}|$. On the other hand, each subroutine stated in Section III-B has a computation complexity of $\mathcal{O}(1)$, additionally, during each time step, each agent can receive $\mathcal{O}(f_{comm} \lfloor \frac{R}{l} \rfloor^2)$ messages, as each agent can have no more than $\lfloor \frac{2R}{l} \rfloor^2$ neighbors in communication range, where R is communication range and l is the grid length. As a result, the computation complexity for each agent planning its motion at each time step is $\mathcal{O}(f_{comm} \lfloor \frac{R}{l} \rfloor^2)$, completing the proof. ■

Remark: The result we obtained in theorem 5 suggests that the algorithm's motion planning complexity, i.e., the computation cost for each agent to plan each step, is linearly proportional to the number of neighbors in communication range R, which is the same as most reactive-control-based DMAPF algorithms such as the algorithm proposed in [17] and the ORCA algorithm [18].

D. The Effect of Real-World Uncertainties

First, we study the effect of the communication uncertainty (packet loss to be specific) on the algorithm's performance. In

reality, to accommodate the packet loss, each message needs to be transmitted multiple times so as to guarantee the delivery of the message. Let the packet loss rate ϵ be such that: if a message is transmitted m_ϵ times in a row, it can be successfully delivered to all the neighbors, we have that the effect of packet loss on communication is equivalent to reducing the communication frequency f_{comm} by m_ϵ times. In addition, recall that the minimal communication requirement for the collision avoidance is each agent being able to successfully deliver one message to the neighbors during each step. When the **effective** communication frequency decreases by m_ϵ times due to the packet loss (as each agent needs to transmit the same message multiple times), slowing down the agent's physical motion by m_ϵ times is sufficient to allow each agent to avoid the collisions. As a result, given a packet loss rate ϵ, the communication uncertainty (packet loss) will slow down the algorithm's *makespan* by at most m_ϵ times, additionally, it will **not** affect the *total distance*.

On the other hand, in practice, without any additional adjustment, agents may collide with each other due to motion and sensing error. To accommodate the imperfect agent motion and sensing, when determining the grid length l, we need to increase the grid spacing to give agents some "buffer" space. As a result, the motion and sensing error will affect the maximal swarm density that WSCaS can handle.

V. Empirical Evaluation

In this section, we study the performance of WSCaS empirically in a 100-robot swarm and in simulation.

A. Simulations Results

1) Performance on Benchmark Maps: We first experimented on benchmark maps from game Dragon Age: Origins [23]. In these experiments, each agent transmits the messages at a fixed frequency such that it can transmit 10 messages during each time step, and the transmitted messages can be received by any agent that is no further than $4\sqrt{2}$ times grid length away. Two key metrics to conservatively evaluate the solution's quality are defined as follows.

Definition 3: Let $len(\mathcal{P}_i)$ be the length of a_i's nominal path $\mathcal{P}_i$, we define **PDAR** (pessimistic distance approximation ratio) and **PMAR** (pessimistic makespan approximation ratio) as:

$$\text{PDAR} = \frac{WSCaS's\ \ total\ \ distance}{\sum_{i=0}^{|\mathcal{A}|-1} len(\mathcal{P}_i)},$$

$$\text{PMAR} = \frac{WSCaS's\ \ makespan}{\max_{0\le i\le|\mathcal{A}|-1} len(\mathcal{P}_i)}$$

First, we investigate the effect of agents' density on WSCaS's PDAR and PMAR. On each map, swarms of size 10% to 90% map occupancy rates moved to a random goal from a random configuration. For each swarm size, 50 trials are given. See Fig. 8 for results. One interesting observation here is that, despite that the structure of these two maps are completely different, the trends of PDAR over swarm density are approximately the same when the occupancy rate is no higher than 50%, which suggests that when the agent density is low, the average-case PDAR is independent of the structure of the map.

In second test, we study the algorithm's performance on large maps. Each map used in this experiments has around 11000 vertices. On each map, swarms of 1024 agents moved to a random goal from a random configuration. For each map, 100 trials are given. The results are shown in Fig. 9. First, we can see that the PDARs on all the maps are approximately the same, which is consistent with our observation in Fig. 8. On the other hand, we can observe that the PMARs on some maps, map *lak404* for example, are much lower than the others, this is because on these maps, the orthogonal passages intersect with each other less often, as a result, *type-4* traffic conditions are less likely to occur. Since the detection of each deadlock (*type-4* traffic condition) will take $\mathcal{O}(|\mathcal{V}|)$ amount of time, the more often *type-4* traffic condition occurs, the larger PMAR will be.

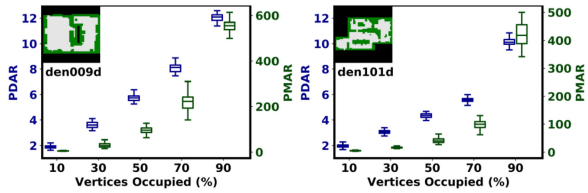

Fig. 8. Performance of the WSCaS on different maps with varying percents of vertices occupied.

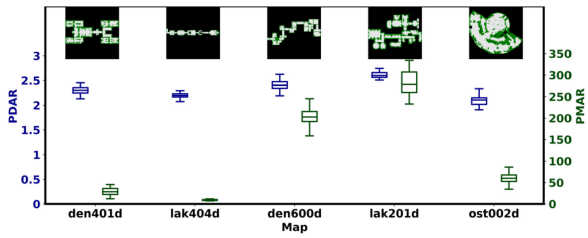

Fig. 9. Simulation results for experiments where 1024 agents move on the different maps.

2) Comparison With the Existing DMAPF Algorithms: Compared to the existing DMAPF algorithms, one merit of our algorithm is that it can provide theoretical completeness and approximate distance-optimal guarantees, requiring only the use of local communication. In fact, to our best acknowledge, and supported by [17], there is no existing DMAPF algorithm being able to provide absolute collision-free guarantee and completeness guarantee at the same time without any global coordination.

We also compared the performance of our method with the ORCA algorithm[1][18], which has been frequently used as a baseline for decentralized motion planning. We tasked a swarm of 1024 agents to use both algorithms to perform *Mirror* and *Reverse* tests [17], which are two particularly challenging DMAPF test cases, see Fig. 10 for a graphical illustration.

Let d be the distance between two adjacent goals (also the distance between two adjacent initial positions) in the test, and r be agent's radius, for each test, we ran multiple trials with different d varying between $2\sqrt{2}r$ to $4\sqrt{2}r$ so as to study the effect of swarm density on both algorithms' performance. In each trial, for both methods, agent's radius is 1 m, maximal speed is 1.0 m/s, sensing (communication) range is $2\sqrt{2}d$, and agent's states are updated at a frequency of 30 Hz. See Fig. 11 for the results.

For each trial, we study three metrics: *makespan, total distance*, and *minimal inter-agent distance (MID)*, which is the minimal distance between any pair of agents during each trial. The *MID* is for evaluating how "safe" the solution is. In each trial, a *makespan ratio* of 0 indicates that the ORCA algorithm failed to drive the swarm the goal configuration within 10^6 seconds of simulated time. One observation here is that in all trials, the

[1]We used the code available at: http://gamma.cs.unc.edu/RVO2/

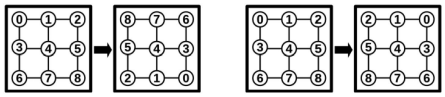

Fig. 10. *Reverse* test *(left)* and *Mirror* test *(right)* for 9 agents.

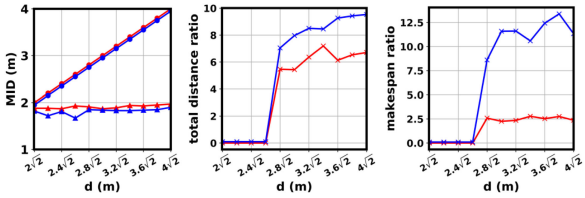

Fig. 11. Red plots are from the *Mirror* tests and blue plots are from the *Reverse* tests. *(left)* The plots with the circle markers are WSCaS's results and the plots with the triangle markers are ORCA's results. *(middle)* The *total distance ratio* is WSCaS's total distance over ORCA's total distance. *(right)* The *makespan ratio* is WSCaS's makespan over ORCA's makespan.

ORCA's *MID*s are below 2, this is because the ORCA algorithm tries to reduce deadlocks by allowing the agent to "slightly penetrate" the collision avoidance constraints, as a result, collision occurs. The result we obtained in these experiments suggests that despite the ORCA algorithm producing a more efficient solution, it often fails to provide the absolute collision-free guarantee in the area where the conflicts frequently occur, additionally, when the swarm density is extremely high, it fails to provide the completeness guarantee as well. In contrary, the WSCaS algorithm's solution is less efficient but will always complete on *swappable* maps, and avoids agent collisions. Therefore for handling the MAPF instances under high swarm density, where collisions must be avoided, such as in automated warehouses, our method may be desired over ORCA.

B. Physical Experiments Results

To validate the correctness and efficiency of our algorithm beyond simulation, we implemented our algorithm on a swarm of 100 *Coachbot V2.0* robots. The inter-agent communication is achieved via custom layer-2 broadcast packets using IEEE 802.11.ac wireless LAN. See Section VII-C in [22] for the detailed description for the *Coachbot V2.0* robot.

In these experiments, 100 physical robots performed three tests on a 12×11 grid: the *Mirror* test, the *Reverse* test, and a third test where robots moved from a random configuration to a column-major ordered pattern. The still images from the third test is shown in Fig. 1. In all experiments, all robots successfully reach the goal configuration.

VI. Conclusion and Future Work

In this paper, we presents a DMAPF algorithm called WSCaS. Theoretical analysis shows that WSCaS can produce a worst case $\mathcal{O}(1)$-approximate distance-optimal solution to MAPF instances on square grids without narrow passages. We implemented our algorithm on a swarm of 100 real robots, and 1024 simulated agents, the result from both the simulation and physical experiment showed that the WSCaS can drive robots to their goals reliably and efficiently.

One limitation of this work is that the algorithm cannot resolve the deadlock that occurs on the narrow passage, we plan on addressing this limitation in future. We also plan on extending our algorithm to the hexagonal grids [24] so as to make the agent's motion more efficient.

References

[1] B. Smith, M. Egerstedt, and A. Howard, "Automatic generation of persistent formations for multi-agent networks under range constraints," *Mobile Netw. Appl.*, vol. 14, no. 3, pp. 322–335, 2009.
[2] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI mag.*, vol. 29, no. 1, pp. 9–9, 2008.
[3] S. Trüg, J. Hoffmann, and B. Nebel, "Applying automatic planning systems to airport ground-traffic control–a feasibility study," in *Proc. Annu. Conf. Artif. Intell.*, 2004, pp. 183–197.
[4] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, "On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman's problem"," *Int. J. Robot. Res.*, vol. 3, no. 4, pp. 76–88, 1984.
[5] K. Solovey and D. Halperin, "On the hardness of unlabeled multi-robot motion planning," *Int. J. Robot. Res.*, vol. 35, no. 14, pp. 1750–1759, 2016.
[6] J. Yu, "Intractability of optimal multirobot path planning on planar graphs," *IEEE Robot. Autom. Lett.*, vol. 1, no. 1, pp. 33–40, Jan. 2016.
[7] G. Wagner and H. Choset, "M*: A complete multirobot path planning algorithm with performance bounds," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2011, pp. 3260–3267.
[8] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artif. Intell.*, vol. 219, pp. 40–66, 2015.
[9] J. Yu and S. M. LaValle, "Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics," *IEEE Trans. Robot.*, vol. 32, no. 5, pp. 1163–1177, Oct. 2016.
[10] M. Erdmann and T. Lozano-Perez, "On multiple moving objects," *Algorithmica*, vol. 2, no. 1–4, 1987, Art. no. 477.
[11] H. Ma, D. Harabor, P. J. Stuckey, J. Li, and S. Koenig, "Searching with consistent prioritization for multi-agent path finding," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, pp. 7643–7650, 2019.
[12] J. Peng and S. Akella, "Coordinating multiple robots with kinodynamic constraints along specified paths," *Int. J. Robot. Res.*, vol. 24, no. 4, pp. 295–310, 2005.
[13] S. Tang, J. Thomas, and V. Kumar, "Hold or take optimal plan (hoop): A quadratic programming approach to multi-robot trajectory generation," *Int. J. Robot. Res.*, vol. 37, no. 9, pp. 1062–1084, 2018.
[14] B. De Wilde, A. W. Ter Mors, and C. Witteveen, "Push and rotate: A complete multi-agent pathfinding algorithm," *J. Artif. Intell. Res.*, vol. 51, pp. 443–492, 2014.
[15] J. Yu, "Constant factor time optimal multi-robot routing on high-dimensional grids," in *Robot.: Sci. Syst.*, Jun. 2018, doi: 10.15607/RSS.2018.XIV.013.
[16] V. J. Lumelsky and K. Harinarayan, "Decentralized motion planning for multiple mobile robots: The cocktail party model," *Auton. Robots*, vol. 4, no. 1, pp. 121–135, 1997.
[17] D. Zhou, Z. Wang, S. Bandyopadhyay, and M. Schwager, "Fast, on-line collision avoidance for dynamic vehicles using buffered voronoi cells," *IEEE Robot. Autom. Lett.*, vol. 2, no. 2, pp. 1047–1054, 2017.
[18] J. Van Den Berg, S. J. Guy, M. Lin, and D. Manocha, "Reciprocal n-body collision avoidance," *Robot. Res,*, pp. 3–19, 2011.
[19] L. Wang, A. D. Ames, and M. Egerstedt, "Safety barrier certificates for collisions-free multirobot systems," *IEEE Trans. Robot.*, vol. 33, no. 3, pp. 661–674, 2017.
[20] H. Ma, J. Li, T. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *Proc. 16th Conf. Auton. Agents MultiAgent Syst.*, 2017.
[21] V. R. Desaraju and J. P. How, "Decentralized path planning for multi-agent teams in complex environments using rapidly-exploring random trees," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2011, pp. 4956–4961.
[22] H. Wang and M. Rubenstein. Walk, stop, count, and swap: Decentralized multi-agent path finding with theoretical guarantees (supplemental material). (2019). [Online]. Available: https://northwestern.box.com/s/m0b7bt1ofizmkwxmn7h5mnjpafbs8stt
[23] N. R. Sturtevant, "Benchmarks for grid-based pathfinding," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 2, pp. 144–148, 2012.
[24] S. D. Han, E. J. Rodriguez, and J. Yu, "SEAR: A polynomial-time multi-robot path planning algorithm with expected constant-factor optimality guarantee," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2018, pp. 1–9.