Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery*

Hang Ma, Wolfgang Hönig, T. K. Satish Kumar, Nora Ayanian, Sven Koenig
Department of Computer Science
University of Southern California
{hangma, whoenig}@usc.edu, tkskwork@gmail.com, {ayanian, skoenig}@usc.edu

Abstract

The Multi-Agent Pickup and Delivery (MAPD) problem models applications where a large number of agents attend to a stream of incoming pickup-and-delivery tasks. Token Passing (TP) is a recent MAPD algorithm that is efficient and effective. We make TP even more efficient and effective by using a novel combinatorial search algorithm, called Safe Interval Path Planning with Reservation Table (SIPPwRT), for single-agent path planning. SIPPwRT uses an advanced data structure that allows for fast updates and lookups of the current paths of all agents in an online setting. The resulting MAPD algorithm TP-SIPPwRT takes kinematic constraints of real robots into account directly during planning, computes continuous agent movements with given velocities that work on non-holonomic robots rather than discrete agent movements with uniform velocity, and is complete for well-formed MAPD instances. We demonstrate its benefits for automated warehouses using both an agent simulator and a standard robot simulator. For example, we demonstrate that it can compute paths for hundreds of agents and thousands of tasks in seconds and is more efficient and effective than existing MAPD algorithms that use a post-processing step to adapt their paths to continuous agent movements with given velocities.

Introduction

In the Multi-Agent Pickup and Delivery (MAPD) problem (Ma et al. 2017), m agents $a_1 \dots a_m$ attend to a stream of incoming pickup-and-delivery tasks in a given 2-dimensional 4-neighbor grid with blocked and unblocked cells of size $L \times L$ each. Agents have to avoid collisions with each other. A task τ_j is characterized by a pickup cell s_j and a delivery cell g_j. The task is inserted into the system at an unknown time. The task set $\mathcal{T}$ contains the unassigned tasks in the system. An agent not executing a task is called a free agent. It can be assigned any one task $\tau_j \in \mathcal{T}$ at a time and then has to move from its current cell via cell s_j to cell g_j, implying that it has to move an object from cell s_j to cell g_j and can carry at most one object at a time. Once it arrives at cell s_j, it starts to execute task τ_j and is called a task agent. Later, once it arrives at cell g_j, it has executed task τ_j and becomes a free agent again. A MAPD instance is solved iff all tasks are executed in a bounded amount of time after they have been inserted into the system. The MAPD problem models applications such as warehouse robots that move shelves (Wurman, D'Andrea, and Mountz 2008), aircraft towing robots that move planes (Morris et al. 2016), and office delivery robots that move packages (Veloso et al. 2015).

Most MAPD algorithms solve the multi-agent pathfinding problem (Ma and Koenig 2017) in an inner loop. The multi-agent pathfinding problem is to compute collision-free paths for multiple agents and is NP-hard to solve optimally (Yu and LaValle 2013b; Ma et al. 2016b). Ways of solving it (and its variants) include reductions to other well-studied combinatorial problems (Yu and LaValle 2013a; Erdem et al. 2013; Surynek 2015) and dedicated algorithms based on search and other techniques (Silver 2005; Standley 2010; Wang and Botea 2011; Luna and Bekris 2011; Sharon et al. 2013; Goldenberg et al. 2014; Wagner and Choset 2015; Sharon et al. 2015; Cohen et al. 2016; Ma and Koenig 2016; Ma, Kumar, and Koenig 2017; Nguyen et al. 2017). See (Ma et al. 2016a; Felner et al. 2017) for complete surveys.

Token Passing (TP) (Ma et al. 2017) is a recent MAPD algorithm that is efficient and effective. It assumes, like many multi-agent pathfinding algorithms, discrete agent movements in the main compass directions with uniform velocity but can use MAPF-POST (Hönig et al. 2016a; 2016b) in a post-processing step to adapt its paths to continuous forward movements with given translational velocities and point turns with given rotational velocities. However, the resulting paths might then not be effective since planning is oblivious to this transformation. TP needs to repeatedly plan time-minimal paths for agents that avoid collisions with the paths of the other agents. We show how TP can be made even more efficient by using Safe Interval Path Planning with Reservation Table (SIPPwRT), our contribution to improve SIPP (Phillips and Likhachev 2011) for this and many other applications. We also show how TP can be made more general by letting SIPPwRT directly compute continuous forward movements and point turns with given velocities. The resulting MAPD algorithm TP-SIPPwRT guarantees a safety distance between agents and solves all well-formed MAPD instances.

*Our research was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189 and 1837779 as well as a gift from Amazon.

TP-SIPPwRT

TP (Ma et al. 2017) is a recent MAPD algorithm that assumes discrete agent movements in the main compass directions with a uniform velocity of typically one cell per time unit on a grid. It is similar to Cooperative A* (Silver 2005) and can be generalized to a fully distributed MAPD algorithm. We describe TP very briefly but its important implication for this paper is that agents repeatedly plan paths for themselves (in Steps TP1 and TP3 below), considering the other agents as dynamic obstacles that follow their paths and with which collisions need to be avoided. The agents use space-time A* for this single-agent path planning.

A set of endpoints is any subset of cells that contains at least all start cells of agents and all pickup and delivery cells of tasks. The pickup and delivery cells are called task endpoints. The other endpoints are called non-task endpoints. A MAPD instance is well-formed iff the number of tasks is finite, there are no fewer non-task endpoints than agents, and there exists a path between any two endpoints that does not pass through other endpoints (Cáp, Vokrínek, and Kleiner 2015; Ma et al. 2017). TP solves all well-formed MAPD instances (Ma et al. 2017).

TP operates as follows for a given set of endpoints: It uses a token (a synchronized block of shared memory) that stores the task set and the current paths, one for each agent. The system repeatedly updates the task set in the token to contain all unassigned tasks in the system and then sends the token to some agent that is currently not following a path. The agent with the token considers all tasks in the task set whose pickup and delivery cells are different from the end cells of all paths in the token. **TP1:** If such tasks exist, then the agent assigns itself that task among these tasks whose pickup cell it can arrive at the earliest, removes the task from the task set, computes two time-minimal paths in the token, one that moves the agent from its current cell to the pickup cell of the task and then one that moves the agent from the pickup cell to the delivery cell of the task, concatenates the two paths into one path, and stores the resulting path. **TP2:** If no such tasks exist and the agent is not in the delivery cell of any task in the task set, then it stores the empty path in the token (to wait at its current cell). **TP3:** Otherwise, the agent computes and stores a time-minimal path in the token that moves the agent from its current cell to some endpoint that is different from both the delivery cells of all tasks in the task set and from the end cells of all paths in the token. (This rule is necessary to avoid deadlocks.) Each path the agent computes has two properties: (1) It avoids collisions with all other paths in the token; (2) No other paths in the token use its end cell after its end time. Finally, the agent releases the token, follows its path, and waits at the end cell of the path.

We now show how TP can be made more general by replacing space-time A* with SIPPwRT, a version of SIPP that computes continuous forward movements and point turns with given velocities rather than discrete agent movements in the main compass directions with uniform velocity. We make some simplifying assumptions throughout this paper even though TP-SIPPwRT and SIPPwRT could easily be generalized beyond them, mostly because these assumptions are necessary to compare TP-SIPPwRT against state-of-the-art MAPD algorithms and, as a bonus, make it easier to explain TP-SIPPwRT: We assume that each agent a_i is a disk with radius $R_i \leq L/2$ and use its center as its reference point. The *configuration* of an agent is a pair of its location (cell) and orientation (main compass direction). Agents always move from the center of their current unblocked cell to the center of an adjacent unblocked cell via the following available actions, besides waiting: a point turn $\pi/2$ rads (ninety degrees) in either clockwise or counterclockwise direction with a given rotational velocity and a forward movement to the center of the adjacent cell with a given translational velocity. The agents can accelerate and decelerate infinitely fast. The paths of two agents are free of collisions iff the interiors of the agent disks never intersect when they follow their paths.

SIPPwRT

Space-time A* and SIPP are two versions of A* that both plan time-minimal paths for agents from their current cells to given goal cells, considering the other agents as dynamic obstacles that follow their paths and with which collisions need to be avoided. They both assume discrete agent movements in the main compass directions with a uniform velocity of typically one cell per time unit on a grid. Space-time A* operates on pairs of cells and time steps, while SIPP groups contiguous time steps during which a cell is not occupied into safe (time) intervals for that cell and thus operates on pairs of cells and safe intervals. This affords the A* search of SIPP pruning opportunities because it is always preferable for an agent to arrive at a cell earlier during the same safe interval since it can then simply wait at the cell. Thus, if the A* search of SIPP has already found a path that arrives at some cell at some time during some safe interval and then discovers a path that arrives at the same cell at a later time in the same safe interval, then it can prune the latter path without losing optimality. SIPP has already been used for robotics applications (Narayanan, Phillips, and Likhachev 2012; Yakovlev and Andreychuk 2017). We generalize it to continuous forward movements and point turns with given velocities in the following, where a safe interval for a cell is now a maximal contiguous interval during which the cell is not occupied by dynamic obstacles. Since SIPPwRT, the resulting version of SIPP, is guaranteed to discover collision-free paths (like space-time A*) when used as part of TP, TP-SIPPwRT, the resulting version of TP, continues to solve all well-formed MAPD instances.

Reservation Table and Safe Intervals

SIPP represents the path of each dynamic obstacle as a chronologically ordered list of cells occupied by the dynamic obstacle, which is not efficient since SIPP has to iterate through all these lists to calculate all safe intervals of a given cell. On the other hand, space-time A* maintains a reservation table that is indexed by a cell and a time step, which allows for the efficient calculation of all safe intervals of a given cell.

SIPPwRT improves upon SIPP using a version of a reservation table that handles continuous agent movements

with given velocities and is indexed by a cell. A reservation table entry of a given cell is a priority queue that contains all reserved intervals for that cell in increasing order of their lower bounds. A reserved interval for a cell is a maximal contiguous interval during which the cell is occupied by some dynamic obstacle. The reservation table allows SIPPwRT to implement all operations efficiently that are needed by TP-SIPPwRT, namely to (1) calculate all safe intervals of a given cell; (2) add reservation table entries after a new path has been calculated; and (3) delete reservation table entries that refer to irrelevant times in the past in order to keep the reservation table small.

Function GetSafeIntervals. *GetSafeIntervals*(*cell*) returns all safe intervals for cell *cell* in increasing order of their lower bounds. The safe intervals for the cell are obtained as the complements of the reserved intervals for the cell with respect to interval $[0, \infty]$. For safe interval $i = [i.lb, i.ub]$ and a dynamic obstacle departing from cell *cell* at time $i.lb$, *dep_cfg*$[cell, i]$ is the configuration of the dynamic obstacle at time $i.lb$. It is *NULL* iff $i.lb \leq$ *current_t*. Similarly, for safe interval $i = [i.lb, i.ub]$ and the dynamic obstacle arriving at cell *cell* at time $i.ub$, *arr_cfg*$[cell, i]$ is the configuration of the dynamic obstacle at time $i.ub$. It is *NULL* iff $i.ub = \infty$.

Time Offsets

The safe intervals of a cell represent the times during which the cell is not occupied. However, this does not mean that an agent can arrive at any of those times at the cell since the agent might still collide with a dynamic obstacle that has just departed from the cell or is about to arrive at the cell. Thus, the lower and upper bounds of a safe interval have to be tightened using the following time offsets.

Function *Offset*(*cfg1*, *cfg2*) returns the time offset ΔT that expresses the minimum amount of time the center of some unknown agent a_1 with safety radius R_1 and translational velocity $v_{trans,1}$ needs to depart from the center of a cell *cfg1.cell* $= l$ with configuration *cfg1* before the center of some unknown agent a_2 with safety radius R_2 and translational velocity $v_{trans,2}$, coming from some cell l', arrives at the center of the same cell *cfg2.cell* $= l$ with configuration *cfg2* to avoid a collision. The time offset is zero iff either *cfg1* $=$ *NULL* or *cfg2* $=$ *NULL*, meaning that either agent a_1 or agent a_2 does not exist. The calculation of the time offset requires only knowledge of the configurations, safety radii, and velocities of both agents.[1]

Assume that agent a_2 departs from cell l' at time 0 (and thus arrives at cell l at time $t' = \frac{L}{v_{trans,2}}$) and agent a_1 departs from cell l at time $t_d \leq \frac{L}{v_{trans,2}}$. $D(t)$ is the distance between the agents, where t is the amount of time elapsed after agent a_2 departs from cell l'. It must hold that $D(t) \geq R_1 + R_2$ to avoid that the two agents collide. We distinguish three cases to calculate the time offset ΔT:

[1] In the pseudocode of SIPPwRT, we show how to keep track of the configurations but do not include the safety radii and velocities in the configurations for ease of readability (although this needs to be done in case they are not the same for all agents and times).

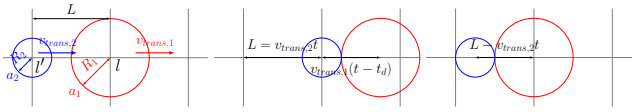

Figure 1: Left: Two agents move in the same direction. Middle: D is at its minimum for the $v_{trans,1} < v_{trans,2}$ case. Right: D is at its minimum for the $v_{trans,1} \geq v_{trans,2}$ case.

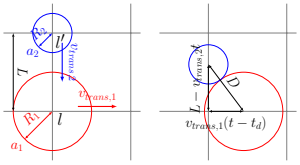

Figure 2: Left: Two agents move in orthogonal directions. Right: D is at its minimum.

(a) Same Direction. Both agents move in the same direction (meaning that the orientations of configurations *cfg1* and *cfg2* are equal), see Figure 1 (left), where gray lines connect the centers of cells. In this case, $D(t) = L - v_{trans,2}t + v_{trans,1}(t - t_d)$. We now distinguish two sub-cases to show that the time offset is $\Delta T = \frac{R_1+R_2}{\min(v_{trans,1}, v_{trans,2})}$.

(a1) Case $v_{trans,1} < v_{trans,2}$. This case is shown in Figure 1 (middle). $D(t)$ decreases as the time t increases. $D(t)$ thus reaches its minimum at the time $t = t'' = \frac{L}{v_{trans,2}}$ when agent a_2 arrives at cell l. Substituting $t = \frac{L}{v_{trans,2}}$ back into $D(t) \geq R_1 + R_2$, we have

$$D(t) = L - v_{trans,2}t + v_{trans,1}(t - t_d) = v_{trans,1}(\frac{L}{v_{trans,2}} - t_d) \quad \geq R_1 + R_2.$$

Therefore, $t_d \leq \frac{L}{v_{trans,2}} - \frac{R_1+R_2}{v_{trans,1}}$. The time offset ΔT is thus $\Delta T = t' - \max t_d = \frac{L}{v_{trans,2}} - (\frac{L}{v_{trans,2}} - \frac{R_1+R_2}{v_{trans,1}}) = \frac{R_1+R_2}{v_{trans,1}}$.

(a2) Case $v_{trans,1} \geq v_{trans,2}$. This case is shown in Figure 1 (right). $D(t)$ decreases before agent a_1 starts to move and then increases as the time t increases. $D(t)$ thus reaches its minimum at the time $t = t_d$ when agent a_1 starts to move. Substituting $t = t_d$ back into $D(t) \geq R_1 + R_2$, we have

$$D(t) = L - v_{trans,2}t + v_{trans,1}(t - t_d) = L - v_{trans,2}t_d \quad \geq R_1 + R_2.$$

Therefore, $t_d \leq \frac{L-(R_1+R_2)}{v_{trans,2}}$. The time offset is thus $\Delta T = t' - \max t_d = \frac{L}{v_{trans,2}} - \frac{L-(R_1+R_2)}{v_{trans,2}} = \frac{R_1+R_2}{v_{trans,2}}$.

(b) Orthogonal Directions. Both agents move in orthogonal directions, see Figure 2. In this case, $D(t) = \sqrt{(v_{trans,1}(t - t_d))^2 + (L - v_{trans,2}t)^2}$. We determine the time t at which $D(t) \geq 0$ reaches its minimum by solving $\frac{\partial D^2}{\partial t} = 0$. Substituting the result $t = \frac{v_{trans,1}^2 t_d + v_{trans,2} L}{v_{trans,1}^2 + v_{trans,2}^2}$ into $D^2 \geq (R_1 + R_2)^2$, we have

$$D^2(t) = (L - v_{trans,2}t)^2 + (v_{trans,1}(t - t_d))^2$$
$$= \frac{(v_{trans,1}(v_{trans,2}t_d - L))^2}{(v_{trans,1}^2 + v_{trans,2}^2)} \geq (R_1 + R_2)^2.$$

Since $L \geq v_{trans,2}t_d$, we have $v_{trans,1}(L - v_{trans,2}t_d) \geq \sqrt{v_{trans,1}^2 + v_{trans,2}^2}(R_1 + R_2)$. Therefore, $t_d \leq$

$\frac{v_{trans,1}L-\sqrt{v_{trans,1}^2+v_{trans,2}^2}(R_1+R_2)}{v_{trans,1}v_{trans,2}}$. The time offset is thus $\Delta T = t' - \max t_d = \frac{L}{v_{trans,2}} - \frac{v_{trans,1}L-\sqrt{v_{trans,1}^2+v_{trans,2}^2}(R_1+R_2)}{v_{trans,1}v_{trans,2}} = \frac{\sqrt{v_{trans,1}^2+v_{trans,2}^2}(R_1+R_2)}{v_{trans,1}v_{trans,2}}$.

(c) Opposite Directions. Both agents move in opposite directions, that is, agent a_1 moves from cell l to cell l' and agent a_2 moves from cell l' to cell l. The time offset is set to allow agent a_1 to arrive at cell l' even before agent a_2 departs from cell l'. In this case, the time offset is $\Delta T = \frac{L}{v_{trans,1}} + \frac{L}{v_{trans,2}}$, which is the sum of the times that agent a_1 needs to move from cell l to cell l' and that agent a_2 needs to move from cell l' to cell l. We later show that SIPPwRT avoids collisions when it uses all bounds simultaneously.

Increased/Decreased Bounds

The algorithm calls the following functions to tighten the lower and upper bounds of safe interval i during which an agent can stay at cell $l =$ *cfg.cell* safely.

The algorithm calls Function *GetLB1*(cfg, i) for an agent a_2 to return $\max_j(j.lb +$ *Offset*$($*dep_cfg*$[$*cfg.cell*$, j], cfg))$. Here, $j.lb +$ *Offset*$($*dep_cfg*$[$*cfg.cell*$, j], cfg)$ is the increased lower bound for each safe interval j in *GetSafeIntervals*$($*cfg.cell*$)$ with $j.lb \leq i.lb$. For agent a_2 that arrives at cell $l =$ *cfg.cell* from another cell l' with configuration *cfg*, the idea is to prevent it from colliding with any dynamic obstacle a_1 that departs from cell l with configuration *dep_cfg*$[$*cfg.cell*$, j]$ before a_2 arrives at cell l.

The algorithm calls Function *GetUB1*(cfg, i) for an agent a_1 to return $\min_j(j.ub -$ *Offset*$($*cfg*$,$ *arr_cfg*$[$*cfg.cell*$, j]))$. Here, $j.ub -$ *Offset*$($*cfg*$,$ *arr_cfg*$[$*cfg.cell*$, j])$ is the decreased upper bound for each safe interval j in *GetSafeIntervals*$($*cfg.cell*$)$ with $j.ub \geq i.ub$. For agent a_1 that departs from cell $l =$ *cfg.cell* with configuration *cfg*, the idea is to prevent it from colliding with any dynamic obstacle a_2 that arrives at cell l from another cell l' with configuration *arr_cfg*$[$*cfg.cell*$, j]$ after a_1 departs from cell l.

The algorithm calls Function *GetLB2*(cfg, i) for an agent a_1 to return $\max_j(j.lb + \frac{L}{v'_{trans}} - \frac{L}{v_{trans}})$. Here, $j.lb + \frac{L}{v'_{trans}} - \frac{L}{v_{trans}}$ is the increased lower bound for each safe interval j in *GetSafeIntervals*$($*cfg.cell*$)$ where the orientation of *dep_cfg*$[$*cfg.cell*$, j]$ is the same as that of *cfg* and $j.lb \leq i.lb$. For agent a_1 that departs from cell $l =$ *cfg.cell* with configuration *cfg* and translational velocity v_{trans} and moves also toward cell l', the idea is to prevent it from arriving at cell l' earlier than (and thus "passing through") any dynamic obstacle a_2 that departs from cell l before agent a_1 with configuration *dep_cfg*$[$*cfg.cell*$, j]$ and translational velocity v'_{trans} and moves also toward cell l'.

The algorithm calls Function *GetUB2*(cfg, i) for an agent a_1 to return $\min_j(j.lb + \frac{L}{v'_{trans}} - \frac{L}{v_{trans}})$. Here, $j.lb + \frac{L}{v'_{trans}} - \frac{L}{v_{trans}}$ is the decreased upper bound for each safe interval j in *GetSafeIntervals*$($*cfg.cell*$)$ where the orientation of *dep_cfg*$[$*cfg.cell*$, j]$ is the same as that of *cfg* and $j.lb \geq i.ub$. For agent a_1 that departs from cell $l =$ *cfg.cell* with configuration *cfg* and translational velocity v_{trans} and moves also toward cell l', the idea is to prevent it from arriving at cell l' later than (and thus "being passed through" by) any dynamic obstacle a_2 that departs from cell l after agent a_1 with configuration *dep_cfg*$[$*cfg.cell*$, j]$ and translational velocity v'_{trans} and moves toward cell l'.

Admissible H-Values

Step TP1 of TP-SIPPwRT requires an agent to use SIPPwRT twice, namely (1) to plan a time-minimal path from its current configuration to a candidate set of endpoints (pickup cells) and (2) to plan a time-minimal path from the resulting configuration to a particular endpoint (a delivery cell). Step TP3 requires the agent to use SIPPwRT once to plan a time-minimal path from its current configuration to a candidate set of endpoints (to avoid deadlocks). The agent always moves along each path with given (fixed) translational and rotational velocities (unless it waits). Thus, SIPPwRT has to plan only paths to a given set G of one or more endpoints. By ignoring the dynamic obstacles, we determine the admissible h-values needed for the A* search of SIPPwRT to plan time-minimal paths as follows: We calculate a time-minimal path that excludes waiting for the agent from each configuration *cfg* to each configuration *cfg*$'$ whose cell is an endpoint (by searching backward once from each configuration *cfg*$'$). We then use the minimum heuristic (Stern, Goldenberg, and Felner 2017) $h(cfg, G) = \min_{cfg'.cell \in G} h(cfg, cfg')$ as admissible h-value of configuration *cfg*, where $h(cfg, cfg')$ is the calculated time of the time-minimal path from *cfg* to *cfg*$'$. In practice, if set G is large and endpoints are densely distributed across the grid, it is more efficient to use $h(cfg, G) = 0$ (as we do for Step TP3 of TP-SIPPwRT) since it can be calculated faster even though SIPPwRT might expand more nodes.

Pseudocode

Algorithm 1 shows the pseudocode of SIPPwRT, which plans a time-minimal path for an agent with translational velocity v_{trans} and rotational velocity v_{rot} from its configuration *start_cfg* at time *current_t* to a cell in set G. SIPPwRT performs a regular A* search with nodes that are pairs of configurations of the agent and safe intervals. The g-value $g[n]$ of a node $n = \langle n.cfg, n.int \rangle$ with configuration $n.cfg$ and safe interval $n.int = [n.int.lb, n.int.ub]$ is the earliest discovered time in $n.int$ when the agent can be in configuration $n.cfg$. The start node is $n = \langle$*start_cfg*$, [$*current_t*$, \infty] \rangle$ with $g[n] =$ *current_t*. The safe interval $n.int$ of the start node expresses that the agent can wait forever in its current configuration. A node n is a goal node iff the cell of its configuration is in set G and the agent can wait forever in its configuration ($n.int.ub = \infty$).

In our implementation of SIPPwRT, each action is a *turn-and-move* action, i.e., a point turn into one of the four compass directions followed by a wait (when necessary) and then a forward movement to a neighboring unblocked cell. Since only forward movements define the temporal constraints between safe intervals of neighboring cells, the state space of our search remains unaffected by the use of turn-and-move actions instead of separate point turn, move, and wait actions independently.

Algorithm 1: SIPPwRT.

```
1  Function SIPPwRT(start_cfg, G, current_t, v_trans, v_rot)
2    n_start ← NewNode(⟨start_cfg, [current_t, ∞]⟩);
3    g[n_start] ← current_t;
4    OPEN ← {n_start};
5    while OPEN ≠ ∅ do
6      n ← arg min_{n'∈OPEN}(g[n'] + h(n'.cfg, G));
7      OPEN ← OPEN \ {n};
8      if n.cell ∈ G and n.int.ub = ∞ then
9        return path from start_cfg to n.cfg;
10     successors ← GetSuccessors(n);
11     foreach n' ∈ successors do
12       if g[n'] is undefined then
13         g[n'] ← ∞;
14       if g[n'] > g[n] + cost[n, n'] then
15         parent[n'] ← n;
16         g[n'] ← g[n] + cost[n, n'];
17         if n' ∉ OPEN then
18           OPEN ← OPEN ∪ {n'};
19   return no path exists (does not happen for well-formed MAPD instances);
20 Function GetSuccessors(n)
21   successors ← ∅;
22   foreach legal turn-and-move action in n do
23     cfg_t ← configuration resulting from executing the point turn of action in
         n.cfg (cfg_t.cell = n.cfg.cell);
24     ub1 ← GetUB1(cfg_t, n.int);
25     lb2 ← GetLB2(cfg_t, n.int);
26     ub2 ← GetUB2(cfg_t, n.int);
27     lb ← max((g[n] + Δt_turn(action, v_rot)), lb2);
28     ub ← min(ub1, ub2);
29     if lb ≤ ub then
30       cfg' ← configuration resulting from executing the forward movement in
           action in n.cfg_t;
31       i'.lb ← lb + Δt_move(action, v_trans);
32       i'.ub ← ub + Δt_move(action, v_trans);
33       safeIntervals ← GetSafeIntervals(cfg'.cell);
34       foreach i'' ∈ safeIntervals do
35         lb1 ← GetLB1(cfg', i'');
36         if [lb1, i''.ub] ∩ i' ≠ ∅ then
37           t' ← max(i'.lb, lb1);
38           n' ← NewNode(⟨cfg', i''⟩);
39           cost[n, n'] ← t' − g[n];
40           successors ← successors ∪ {n'};
41   return successors;
```

Function GetSuccessors. *GetSuccessors*(n) calculates the successors of node n by considering all legal turn-and-move actions *action* available to the agent in configuration $n.cfg$ [Line 22]. Assume that executing the point turn of action *action* takes $\Delta t_{turn}(action, v_{rot})$ time units and results in configuration *cfg_t* with which the agent departs from its current cell [Line 23]. The agent must depart from its current cell no later than *lb* and no earlier than *ub* to avoid colliding with dynamic obstacles that also visit its current cell [Lines 24-28]. If the agent can depart from its current cell [Line 29], then assume that executing the forward movement of action *action* in configuration *cfg_t* takes $\Delta t_{move}(action, v_{trans})$ time units and results in successor configuration cfg' [Line 30]. The agent waits an appropriate amount of time in configuration *cfg_t* after the point turn, then executes the forward movement, and arrives in configuration cfg' in interval $i' = [lb + \Delta t_{move}(action, v_{trans}), ub + \Delta t_{move}(action, v_{trans})]$ [Lines 31-32]. The successors of node n are generated by processing all safe intervals $i'' = [i''.lb, i''.ub]$ for the new cell $cfg'.cell$ of the agent [Lines 33-34]. The lower bound of safe interval i'' is increased from $i''.lb$ to *lb1* to ensure that the agent can arrive at its new cell without colliding with dynamic obstacles that also visit its new cell [Line 35]. The updated safe interval $[lb1, i''.ub]$ is intersected with interval i' [Line 36]. If their intersection is non-empty, then the agent can arrive at its successor configuration during safe interval i''. Only the earliest time t' in the intersection needs to be considered (since the agent can simply wait in its successor configuration and the later times in the intersection can thus be pruned, as argued earlier) [Line 37]. The resulting successor of node n is $n' = \langle cfg', i'' \rangle$ [Line 38], and the cost (here: time) of the transition from node n to node n' is $cost[n, n'] = t' - g[n]$ [Line 39] (consisting of executing the point turn of action *action* for $\Delta t_{turn}(action, v_{rot})$ time units, waiting for $t' - g[n] - \Delta t_{turn}(action, v_{rot}) - \Delta t_{move}(action, v_{trans})$ time units, and then executing the forward movement of action *action* for $\Delta t_{move}(action, v_{trans})$ time units), so that $g[n'] = g[n] + cost[n, n'] = t'$ is the earliest discovered time in $n'.int = i''$ when the agent can be in configuration $n'.cfg = cfg'$.

Main Routine. The main routine of SIPPwRT performs a regular A* search. It initializes the g-value of the start node and inserts the node into the OPEN list [Lines 2-4]. It then repeatedly removes a node n with the smallest sum of g-value and h-value $g[n] + h(n.cfg, G)$ from the OPEN list [Lines 6-7] and processes it: If the node is a goal node, then it returns the path found by following the parent pointers from the node to the start node [Lines 8-9]. Otherwise, it generates the successors of the node [Line 10]. For each successor, it initializes its g-value to infinity if the g-value is still undefined [Lines 12-13]. It then checks whether the g-value of the successor can be reduced by changing its parent pointer to node n [Line 14]. If so, it changes the parent pointer of the successor, reduces its g-value, and inserts it into the OPEN list (if necessary) [Lines 15-18].

Theorem 1. The path returned by SIPPwRT from the start configuration to a goal is free of collisions.

We prove Theorem 1 in the technical report.

Since all heuristics used by SIPPwRT are admissible as argued earlier, using the argument in (Phillips and Likhachev 2011) together with Theorem 1, it is straightforward to show that SIPPwRT returns a time-minimal path to a given set G of one or more endpoints that does not collide with the paths of other agents in the token and is complete for the single-agent path-planning problems for function calls TP1 and TP3. We can thus rely on the proof of Theorem 3 in (Ma et al. 2017) to show that TP-SIPPwRT is complete for well-formed MAPD instances.

Theorem 2. TP-SIPPwRT solves all well-formed MAPD instances.

Simulated Automated Warehouses

We demonstrate the benefits of TP-SIPPwRT for automated warehouses using both an agent simulator with perfect path execution and a standard robot simulator with imperfect path execution resulting from unmodeled kinodynamic constraints and motion noise by the MAPD algorithms. Figure 3 (left) shows an example on the agent simulator with 50

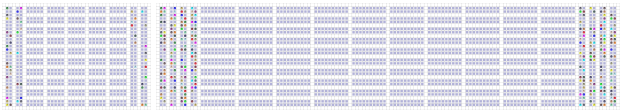

Figure 3: Left: Small simulated warehouse environment. Right: Large simulated warehouse environment.

Table 1: Experiment 1. (Inapplicable entries are dashed.)

algorithm	v_{task}	discrete srvc time	discrete makespan	srvc time	make span	plan time	post-proc time	thpt	stdy thpt
TP-SIPPwRT	0.50	–	–	944.03	2,475.58	0.90	–	0.397	0.433
	0.75			601.69	1,755.22	0.92		0.552	0.632
	1.00			435.26	1,392.00	0.83		0.689	0.782
CENTRAL	0.50	325.28	1,163	1,049.51	2,617.00	1,161.44	264.66	0.370	0.406
	0.75			691.90	1,895.68		254.36	0.504	0.552
	1.00			520.36	1,553.00		269.91	0.609	0.670
TP-A*	0.50	329.83	1,204	1,026.23	2,628.22	1.00	267.38	0.373	0.408
	0.75			675.65	1,909.45		295.54	0.508	0.558
	1.00			505.81	1,570.77		278.74	0.609	0.683

agents and cells of size $1\,\mathrm{m} \times 1\,\mathrm{m}$. Grey cells in columns of grey cells are potential start cells for the agents. Colored disks are the actual start cells, which are drawn randomly from all potential start cells and are the non-task endpoints. All agents face north in their start cells. Grey cells other than the start cells are task endpoints (that would house shelves in a warehouse even though we do not model shelves here). The pickup and delivery cells of all tasks are drawn randomly from all task endpoints. White cells are non-endpoints.

The agents model circular warehouse robots. All agents use the same rotational velocity v_{rot}. The following rules impose restrictions on their legal movements and translational velocities: All free agents can move with high translational free velocity $v_{trans} = v_{free}$ through all cells because warehouse robots that do not carry shelves can move through all cells, including those that house shelves. All task agents can move with slow translational task velocity $v_{trans} = v_{task}$ through only the pickup and delivery endpoints of their tasks and all other non-endpoints since warehouse robots that carry shelves cannot move through cells that house shelves.

Experimental Results

We now report our experimental results on a 2.50 GHz Intel Core i5-2450M laptop with 6 GB RAM. Videos of sample experiments can be found at
http://idm-lab.org/project-p.html

Experiment 1: MAPD Algorithms and Task Velocity. We compare TP-SIPPwRT for $v_{task} = 0.50$, 0.75, and $1.00\,\mathrm{m/s}$ on the agent simulator in the small simulated warehouse environment of Figure 3 (left) to two MAPD algorithms that both assume discrete agent movements with uniform velocity to the four neighboring cells, namely the original TP (Ma et al. 2017) (labeled TP-A*) and CENTRAL (Ma et al. 2017), which repeatedly uses the Hungarian Method (Kuhn 1955) to assign tasks to agents and then Conflict-Based Search (Sharon et al. 2015) to plan paths for the agents. We first convert the paths produced by these two MAPD algorithms from containing movements in the four compass directions to containing forward movements and point turns. We then use MAPF-POST (Hönig et al. 2016a) to adapt the paths in polynomial time to continuous agent movements with given velocities. Since MAPF-POST guarantees safety distances between agents of $L/\sqrt{2} = 0.71\,\mathrm{m}$, we use the same radius of $R = 0.5L/\sqrt{2} = 0.35\,\mathrm{m}$ for all agents. We use a runtime limit of 5 minutes per instance. We use 30 agents (*agts*) since CENTRAL, the most runtime-intensive of our MAPD algorithms, can handle only slightly more than 30 agents without any timeouts. We use $v_{free} = 1.00\,\mathrm{m/s}$ and $v_{rot} = \pi/2 = 1.57\,\mathrm{rad/s}$. We generate one sequence of 1,000 tasks and insert them in the generated order into the system with a task frequency (*task freq*) of 2 tasks in the beginning of every second.

Figure 4 visualizes the throughput at time t [number of tasks that finish execution per second in the 100-second window $(t - 100, t]$], measured in tasks per second, as a function of t, measured in seconds. The steady state is the time interval when the throughput remains mostly unchanged, determined by visual inspection of the graphs. We use as steady state $t \in [501, 2100]$ for $v_{task} = 0.50\,\mathrm{m/s}$, $t \in [501, 1500]$ for $v_{task} = 0.75\,\mathrm{m/s}$, and $t \in [501, 1100]$ for $v_{task} = 1.00\,\mathrm{m/s}$. The throughput at time t of TP-SIPPwRT decreases earlier than the ones of TP-A* and CENTRAL because fewer still unexecuted tasks are available toward the end for TP-SIPPwRT than for them. Thus, TP-SIPPwRT is more effective than them.

Table 1 reports the discrete service (*srvc*) time [time until a task has finished execution after insertion into the system according to the original plan with discrete agent movements, averaged over all tasks], discrete makespan [time when the last task has finished execution according to the original plan with discrete agent movements], service (*srvc*) time [time until a task has actually finished execution after insertion into the system, averaged over all tasks], makespan [time when the last task has actually finished execution], planning (*plan*) time [execution time of the MAPD algorithm], and post-processing (*post-proc*) time [execution time of MAPF-POST], all measured in seconds, as well as the throughput (*thpt*) [throughput at time t averaged over all times t whose throughputs are positive] and the throughput in the steady state (*stdy thpt*) [throughput at time t averaged over all times in the steady state], both measured in number of tasks per second. Service time, makespan, and throughput measure effectiveness, while the planning and post-processing times measure efficiency. The planning time of TP-SIPPwRT is less than one second for 30 agents and 1,000 tasks. It is on par with the one of TP-A* and smaller than the one of CENTRAL. Furthermore, TP-SIPPwRT does not have any post-processing time while both TP-A* and CENTRAL have post-processing times of more than 250 seconds. Thus, TP-SIPPwRT is more efficient than them. The service time and makespan of TP-SIPPwRT are smaller than the ones of TP-A* and CENTRAL, while its throughput is larger. Thus, TP-SIPPwRT is more effective than them.

Experiment 2: Number of Agents, Task Frequency, and Task Velocity. We run TP-SIPPwRT with the same setup as in Experiment 1 (including the same sequence of 1,000 tasks) for $v_{task} = 0.50$, 0.75, and $1.00\,\mathrm{m/s}$, 10, 20, 30, 40, and 50 agents, and task frequencies of 1, 2, 5, and 10 tasks per second. Table 2 shows that the planning time of TP-

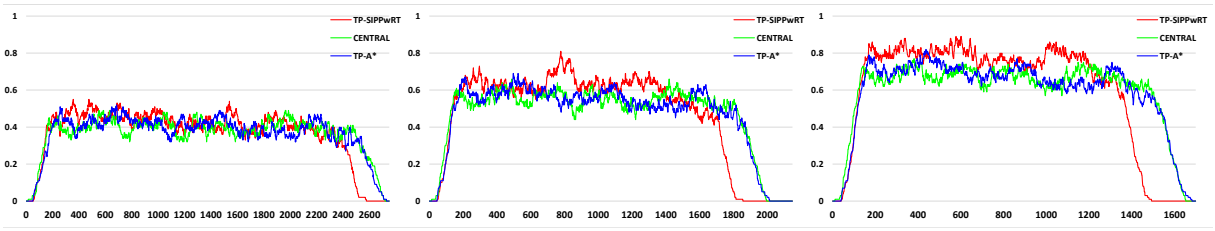

Figure 4: Number of tasks executed per second in a moving 100-second window $(t-100, t]$ (that is, throughput at time t) as a function of time t for different MAPD algorithms. Left: $v_{task} = 0.50\,\mathrm{m/s}$. Middle: $v_{task} = 0.75\,\mathrm{m/s}$. Right: $v_{task} = 1.00\,\mathrm{m/s}$.

Table 2: Experiment 2.

v_{task}		0.50				0.75				1.00			
agts	task freq	srvc time	make span	plan time	thpt	srvc time	make span	plan time	thpt	srvc time	make span	plan time	thpt
10	1	2,809.72	6,771.00	0.84	0.146	1,834.97	4,764.28	0.86	0.213	1,357.21	3,818.00	0.72	0.270
	2	3,029.59	6,759.41	0.85	0.157	2,077.68	4,768.89	0.84	0.215	1,584.62	3,784.00	0.73	0.274
	5	3,181.97	6,789.41	0.86	0.155	2,185.29	4,748.33	0.86	0.225	1,710.76	3763.71	0.75	0.274
	10	3,215.43	6,775.00	0.84	0.159	2,252.70	4,762.45	0.88	0.219	1,750.19	3,749.00	0.75	0.280
20	1	1,228.35	3,557.58	0.90	0.295	745.48	2,540.33	0.89	0.411	502.50	2,000.71	0.76	0.511
	2	1,450.40	3,503.00	0.89	0.298	966.27	2,493.67	0.91	0.392	714.20	1,980.00	0.79	0.489
	5	1,591.79	3,519.83	0.89	0.292	1,088.86	2,481.85	0.88	0.416	844.32	1,966.00	0.81	0.507
	10	1,661.62	3,502.83	0.88	0.290	1,136.08	2,479.45	0.90	0.417	892.22	1,964.00	0.81	0.507
30	1	723.03	2,482.41	0.94	0.396	389.15	1,763.50	0.91	0.551	222.21	1,431.71	0.82	0.672
	2	944.03	2,475.58	0.90	0.397	601.69	1,755.22	0.92	0.552	435.26	1,392.00	0.83	0.689
	5	1,079.62	2,435.83	0.90	0.398	728.33	1,724.18	0.92	0.555	563.14	1,372.71	0.83	0.688
	10	1,126.47	2,468.00	0.93	0.393	779.67	1,737.00	0.92	0.550	612.06	1,380.00	0.84	0.065
40	1	484.93	2,023.58	0.90	0.484	225.18	1,471.12	0.95	0.657	101.16	1,252.00	0.85	0.765
	2	701.23	1,945.00	0.94	0.503	432.11	1,430.33	0.95	0.674	298.04	1,122.71	0.89	0.847
	5	830.73	2,054.00	0.89	0.470	563.25	1,368.67	0.94	0.693	427.23	1,073.00	0.87	0.870
	10	880.46	1,905.00	0.88	0.506	605.10	1,382.67	0.94	0.686	469.92	1,095.71	0.89	0.853
50	1	331.66	1,680.41	0.98	0.641	122.98	1,262.00	0.98	0.771	63.45	1,140.41	0.96	0.845
	2	557.10	1,676.58	0.97	0.581	335.58	1,192.51	0.96	0.804	219.99	968.00	0.92	0.976
	5	683.56	1,674.41	0.97	0.573	454.42	1,153.51	0.93	0.814	344.44	931.00	0.91	0.992
	10	729.07	1,644.41	0.97	0.582	502.03	1,200.94	0.98	0.784	389.78	926.00	0.93	0.996

Table 3: Experiment 3.

v_{task}	0.50					0.75					1.00				
agts	srvc time	make span	plan time	thpt	stdy thpt	srvc time	make span	plan time	thpt	stdy thpt	srvc time	make span	plan time	thpt	stdy thpt
100	877.94	2,891.58	5.72	0.70	0.81	489.46	2,130.67	5.83	0.91	1.15	289.80	1,671.00	5.15	1.16	1.49
150	525.07	2,269.58	6.49	0.88	1.15	253.49	1,602.00	6.67	1.20	1.64	122.69	1,396.71	5.57	1.37	1.98
200	353.46	1,905.58	7.08	1.03	1.47	154.76	1,504.63	7.35	1.31	1.97	117.21	1,276.12	9.50	1.50	2.04
250	267.07	1,762.24	9.35	1.13	1.71	147.90	1,271.67	12.83	1.52	1.99	132.45	1,297.00	15.76	1.48	2.02

SIPPwRT is less than one second for up to 50 agents and 1,000 tasks. As expected, the service time decreases as the task frequency decreases; the service time and makespan decrease and the throughput increases as the number of agents increases; and the service time and makespan decrease and the throughput increases as the task velocity increases.

Experiment 3: Environment Size, Number of Agents, and Task Velocity. We run TP-SIPPwRT with the same setup as in Experiment 1 but in the large simulated warehouse environment of Figure 3 (right) for 100, 150, 200, and 250 agents and $v_{task} = 0.50$, 0.75, and $1.00\,\mathrm{m/s}$. We use one sequence of 2,000 tasks and a task frequency of 2 tasks per second. Figure 5 visualizes the throughput at time t. We use as steady state $t \in [501, 1000]$. Table 3 shows that the planning time of TP-SIPPwRT is less than 16 seconds for up to 250 agents and 2,000 tasks, justifying our claim that it can compute paths for hundreds of agents and thousands of tasks in seconds. Similarly to before, the service time and makespan decrease and the throughput and planning time increase as the number of agents increases; and the service time and makespan decrease and the throughput increases as the task velocity increases. There is an exception due to the congestion resulting from many agents for 250 agents and $v_{task} = 1.00\,\mathrm{m/s}$.

Experiment 4: Robot Simulator. We created a custom model of the kinodynamic constraints of a differential-drive Create2 robot from iRobot for the robot simulator V-REP (Rohmer, Singh, and Freese 2013). Create2 robots have a cylindrical shape with radius $0.175\,\mathrm{m}$ and can reach a translational speed of $0.5\,\mathrm{m/s}$ and a rotational speed of $4.2\,\mathrm{rad/s}$. We use $v_{free} = 0.40\,\mathrm{m/s}$, $v_{task} = 0.20\,\mathrm{m/s}$, $v_{rot} = \pi = 3.14\,\mathrm{rad/s}$, and $R = 0.40\,\mathrm{m}$ as conservative values to allow the robots to follow their paths safely despite unmodeled kinodynamic constraints and motion noise by TP-SIPPwRT. We implemented a PID controller that uses $[x, y, \theta]^T$ (given by V-REP) as the current state and the desired next cell with the associated desired arrival time (given by TP-SIPPwRT) as the goal state. The PID controller corrects for heading errors by orienting the robot to face the desired next cell while simultaneously adjusting the translational speed to let the robot arrive at the desired next cell at the desired arrival time. We limit our experiment to the small warehouse environment of Figure 6 for 10 robots due to the slow runtime of V-REP. We use one sequence of 20 tasks and a task frequency of 2 tasks per second. The planning time of TP-SIPPwRT is $2\,\mathrm{ms}$. All robots follow their paths safely, resulting in a service time of $90.57\,\mathrm{s}$ and a makespan of $171.16\,\mathrm{s}$.

Conclusions and Future Work

We presented the efficient and effective algorithm TP-SIPPwRT for the Multi-Agent Pickup and Delivery problem. We suggest the following future research directions: (1) Make existing (even optimal) multi-agent pathfinding algorithms more general by combining them with our SIPPwRT to compute continuous agent movements with given velocities. The resulting algorithms could, for example, be used to make CENTRAL more general. (2) Include additional kinodynamic constraints into SIPPwRT and TP-SIPPwRT, such as acceleration and deceleration constraints, to allow robots to follow their paths even more safely. (3) Make TP-SIPPwRT decentralized.

References

Cáp, M.; Vokrínek, J.; and Kleiner, A. 2015. Complete decentralized method for on-line multi-robot trajectory planning in well-formed infrastructures. In *ICAPS*, 324–332.

Cohen, L.; Uras, T.; Kumar, T. K. S.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Improved solvers for bounded-suboptimal multi-agent path finding. In *IJCAI*, 3067–3074.

Erdem, E.; Kisa, D. G.; Oztok, U.; and Schueller, P. 2013. A

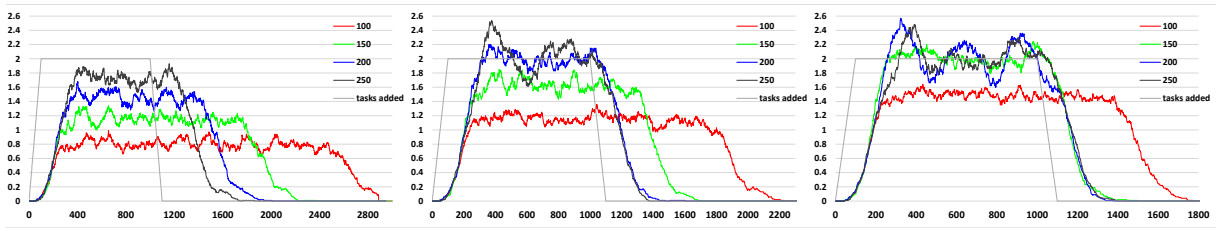

Figure 5: Number of tasks inserted (in light grey) and executed per second in a moving 100-second window $(t - 100, t]$ as a function of time t for different numbers of agents. Left: $v_{task} = 0.50\,\mathrm{m/s}$. Middle: $v_{task} = 0.75\,\mathrm{m/s}$. Right: $v_{task} = 1.00\,\mathrm{m/s}$.

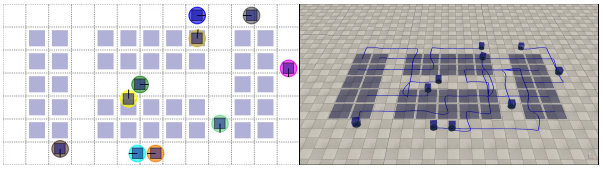

Figure 6: Screenshots for Experiment 3 at $t = 35\,\mathrm{s}$. Left: Agent simulator. Right: Robot simulator.

general formal framework for pathfinding problems with multiple agents. In *AAAI*, 290–296.

Felner, A.; Stern, R.; Shimony, S. E.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N.; Wagner, G.; and Surynek, P. 2017. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *SoCS*, 29–37.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced Partial Expansion A*. *Journal of Artificial Intelligence Research* 50:141–187.

Hönig, W.; Kumar, T. K. S.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2016a. Multi-agent path finding with kinematic constraints. In *ICAPS*, 477–485.

Hönig, W.; Kumar, T. K. S.; Ma, H.; Ayanian, N.; and Koenig, S. 2016b. Formation change for robot groups in occluded environments. In *IROS*, 4836–4842.

Kuhn, H. W. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2:83–97.

Luna, R., and Bekris, K. E. 2011. Push and Swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, 294–300.

Ma, H., and Koenig, S. 2016. Optimal target assignment and path finding for teams of agents. In *AAMAS*, 1144–1152.

Ma, H., and Koenig, S. 2017. AI buzzwords explained: Multi-agent path finding (MAPF). *AI Matters* 3(3):15–19.

Ma, H.; Koenig, S.; Ayanian, N.; Cohen, L.; Hönig, W.; Kumar, T. K. S.; Uras, T.; Xu, H.; Tovey, C.; and Sharon, G. 2016a. Overview: Generalizations of multi-agent path finding to real-world scenarios. In *IJCAI-16 Workshop on Multi-Agent Path Finding*.

Ma, H.; Tovey, C.; Sharon, G.; Kumar, T. K. S.; and Koenig, S. 2016b. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *AAAI*, 3166–3173.

Ma, H.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2017. Lifelong multi-agent path finding for online pickup and delivery tasks. In *AAMAS*, 837–845.

Ma, H.; Kumar, T. K. S.; and Koenig, S. 2017. Multi-agent path finding with delay probabilities. In *AAAI*, 3605–3612.

Morris, R.; Pasareanu, C.; Luckow, K.; Malik, W.; Ma, H.; Kumar, S.; and Koenig, S. 2016. Planning, scheduling and monitoring for airport surface operations. In *AAAI-16 Workshop on Planning for Hybrid Systems*.

Narayanan, V.; Phillips, M.; and Likhachev, M. 2012. Anytime safe interval path planning for dynamic environments. In *IROS*, 4708–4715.

Nguyen, V.; Obermeier, P.; Son, T. C.; Schaub, T.; and Yeoh, W. 2017. Generalized target assignment and path finding using answer set programming. In *IJCAI*, 1216–1223.

Phillips, M., and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In *ICRA*, 5628–5635.

Rohmer, E.; Singh, S. P. N.; and Freese, M. 2013. V-REP: A versatile and scalable robot simulation framework. In *IROS*, 1321–1326.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence* 195:470–495.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219:40–66.

Silver, D. 2005. Cooperative pathfinding. In *AIIDE*, 117–122.

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, 173–178.

Stern, R.; Goldenberg, M.; and Felner, A. 2017. Shortest path for K goals. In *SoCS*, 167–168.

Surynek, P. 2015. Reduced time-expansion graphs and goal decomposition for solving cooperative path finding sub-optimally. In *IJCAI*, 1916–1922.

Veloso, M.; Biswas, J.; Coltin, B.; and Rosenthal, S. 2015. CoBots: Robust symbiotic autonomous mobile service robots. In *IJCAI*, 4423–4429.

Wagner, G., and Choset, H. 2015. Subdimensional expansion for multirobot path planning. *Artificial Intelligence* 219:1–24.

Wang, K., and Botea, A. 2011. MAPP: A scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research* 42:55–90.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine* 29(1):9–20.

Yakovlev, K., and Andreychuk, A. 2017. Any-angle pathfinding for multiple agents based on SIPP algorithm. In *ICAPS*, 586–593.

Yu, J., and LaValle, S. M. 2013a. Planning optimal paths for multiple robots on graphs. In *ICRA*, 3612–3617.

Yu, J., and LaValle, S. M. 2013b. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*, 1444–1449.