Traffic Flow Optimisation for Lifelong Multi-Agent Path Finding

Zhe Chen[1], **Daniel Harabor**[1], **Jiaoyang Li**[2], **Peter J. Stuckey**[1,3]
[1]Department of Data Science and Artificial Intelligence, Monash University, Melbourne, Australia
[2]Robotics Institute, Carnegie Mellon University, Pittsburgh, USA
[3]OPTIMA Australian Research Council ITTC, Melbourne, Australia
{zhe.chen,daniel.harabor,peter.stuckey}@monash.edu, jiaoyangli@cmu.edu
a The paper was accepted for publication at AAAI 24

Abstract

Multi-Agent Path Finding (MAPF) is a fundamental problem in robotics that asks us to compute collision-free paths for a team of agents, all moving across a shared map. Although many works appear on this topic, all current algorithms struggle as the number of agents grows. The principal reason is that existing approaches typically plan free-flow optimal paths, which creates congestion. To tackle this issue, we propose a new approach for MAPF where agents are guided to their destination by following congestion-avoiding paths. We evaluate the idea in two large-scale settings: one-shot MAPF, where each agent has a single destination, and lifelong MAPF, where agents are continuously assigned new destinations. Empirically, we report large improvements in solution quality for one-short MAPF and in overall throughput for lifelong MAPF.

Introduction

Multi-Agent Path Finding (MAPF) (Stern et al. 2019) is a fundamental coordination problem in robotics and also at the heart of many industrial applications. For example, in automated fulfilment (Wurman, D'Andrea, and Mountz 2008) and sortation centres (Kou et al. 2020), a team of robots must work together to deliver packages. To complete such tasks, robots must navigate across a shared map to reach their goal locations. Robots must arrive at their goals as quickly as possible, and they must do so while avoiding collisions with static obstacles in the environment as well as with other moving robots. In the classical, sometimes called *one-shot*, MAPF problem (Stern et al. 2019), robots are modelled as simplified agents, with each being assigned a single goal location. A related setup, known as *lifelong* MAPF (Ma et al. 2017; Li et al. 2021d), continuously assigns new goal locations to agents as they arrive at their current goal locations.

Both one-shot and lifelong MAPF problems have been intensely studied, with a variety of substantial advancements reported in the literature. For example, leading optimal (Sharon et al. 2015; Li et al. 2021c, 2019; Lam et al. 2022; Gange, Harabor, and Stuckey 2019) and bounded-suboptimal (Li, Ruml, and Koenig 2021) MAPF algorithms now scale to hundreds of agents, while providing solution quality guarantees. Yet some real applications require up to *thousands of simultaneous agents*, and at this scale, only unbounded suboptimal approaches are currently applicable.

Two leading frameworks for unbounded suboptimal MAPF are Large Neighbourhood Search (LNS) (Li et al. 2021a, 2022, 2021b) and Priority Inheritance with Back Tracking (PIBT) (Okumura et al. 2022; Okumura 2023). Approaches based on LNS modify an existing (potentially infeasible) MAPF plan by iteratively changing the paths for a few agents to reduce the number of collisions or their travel times. A main drawback of LNS-based methods is that the performance of the single-agent path planner significantly degrades as map size and number of agents increase. This restricts the applicability of LNS as, when paired with an aggressive timeout (often required for real applications), timeout failure can occur before any iterations are completed. Meanwhile, PIBT-based approaches use rule-based collision avoidance to plan paths. They compute paths timestep by timestep, which is extremely efficient. For this reason, PIBT-based methods usually scale to substantially more agents than LNS-based methods. However, this type of planners guides agents toward their goals using individually optimal *free-flow* heuristics (i.e., considering only travel distance while ignoring other agents), a strategy known to create high levels of congestion. Moreover, its computed solutions tend to have higher costs than those computed by LNS-based methods (Shen et al. 2023).

A related body of work on the Traffic Assignment Problem (TAP) (Patriksson 2015) from the transportation community considers similar issues. Like MAPF, TAP approaches compute optimised paths for agents moving across a shared map. In TAP, multiple agents can use the same edge at the same time, but travel time increases with the number of agents. TAP algorithms thus try to compute a User Equilibrium (UE) solution, where no agent can improve their arrival time by switching to a different path with lower cost. Inspired by such ideas, we propose to compute time-independent routes for MAPF agents, which take into account *expected congestion* due to other agents following similar trajectories. We use the resulting *guide paths* as improved heuristics for PIBT, which allows fast planning for large numbers of agents. Yet we also continuously improve the guide paths, like LNS, which serves to further increase the quality of computed plans. Experimental results show convincing improvements for one-shot and lifelong MAPF against a range of leading algorithms, including PIBT, LNS and LaCAM.

Problem Definition

The Multi-Agent Path Finding (MAPF) problem takes as **input** is an undirected gridmap $G = (V, E)$ and a set of k agents $\{a_1 ... a_k\}$, where each agend begins at a start location $s_i \in V$ and must reach a goal location $g_i \in V$. Agents can move from one grid cell (equiv. vertex) to another adjacent grid cell using one of four possible **grid moves**: *North, South, East* and *West*. Time is discretised into unit-sized steps. At each timestep t, every agent must move to an adjacent grid cell or else *wait* in place. Each action (wait or move) has a cost of 1. We use the notation (v_i, t_j) to say that vertex v is occupied by agent i at timestep t_j.

A **vertex conflict**, $\langle a_i, a_j, v, t\rangle$, occurs when two agents a_i and a_j occupy the same vertex $v \in V$ at the same timestep t. An **edge conflict**, $\langle a_i, a_j, e, t\rangle$, occurs when two agents a_i and a_j pass through the same edge $e \in E$ in opposite directions at the same timestep t. A **path** is a sequence of actions that transitions an agent from a vertex v_i to another vertex v_j. We say that the path is valid (equiv. feasible or collision-free) if no action produces an edge or vertex conflict. The **cost** of a path is the sum of its action costs.

A **solution** to a MAPF problem is a collision-free path assignment that allows every agent i to move from s_i to g_i and then remain at g_i without any conflict. The **objective** in MAPF is to find a feasible solution that minimises the *Sum of Individual (path) Costs* (SIC).

Lifelong Multi-Agent Path Finding: in this variant, agents are assigned sequences of tasks that must be completed. A task for agent i is a request to visit a goal location g_i. Unlike MAPF, when g_i is reached, the agent is not required to wait indefinitely. Instead, agent i is assigned a new goal g'_i. The problem continues indefinitely with the objective being to maximise total task completions by an operational time limit T (i.e., max *throughput*). Note that we assume the *task assigner* is external to our path-planning system and that agents only know their current goal location.

Background

We briefly summarise previous works which are necessary for understanding our main contributions.

FOCAL Search

This is a best-first bounded-suboptimal algorithm (Pearl and Kim 1982) similar to A^*. It uses an *OPEN* list to prioritise node expansions in the usual $f = g + h$ ordering. A second priority queue, *FOCAL*, contains only those nodes from *OPEN* with $f \leq w \cdot f_{min}$, where f_{min} is the minimal f-value of any node in *OPEN* and $w \geq 1$ is a user-specified acceptance criterion that indicates the maximum suboptimality of an admissible solution. Since every node in *FOCAL* is admissible, they can be sorted with any alternative criteria, denoted $\hat{h}$. This approach guarantees solutions of cost not more than $w \cdot C^*$, where C^* is the optimal cost. Notice that A^* is a special case of FOCAL Search where $w = 1$.

Algorithm 1: PIBT. In each iteration *PlanStep* computes a next move θ for each agent $a \in A$, currently at position ϕ, using a priority ordering p.

```
1  $p' \leftarrow p$; // Create a copy of initial priority $p$;
2  PlanStep($A, \phi, p, p'$)
3      for $a_i \in A$ do
4          $\theta[a_i] \leftarrow \perp$; //$\perp$ means no action decided;
5          $g_i = \phi[a_i]$ ? $p_i \leftarrow p'_i : p_i \leftarrow p_i + 1$;
6      sort $A$ in decreasing order of $p_i \in p$ ;
7      for $a_i \in A$ do
8          if $\theta[a_i] = \perp$ then
9              PIBT($a_i, \perp, \phi, \theta, A$);
10     return $\theta$ ;
11
12 PIBT($a_i, a_j, \phi, \theta, A$)
13     $C \leftarrow \{v \mid (\phi[a_i], v) \in E\}$;
14     sort $C$ based on increasing $dist(v, g_i)$;
15     for $v \in C$ in order do
16         if $\exists a \in A, \theta[a] = v$ then continue;
17         if $a_j \neq \perp \wedge \phi[a_j] = v$ then continue;
18         $\theta[a_i] \leftarrow v$;
19         if $\exists a \in A, \phi[a] = v \wedge \theta[a] = \perp$ then
20             if $\neg$ PIBT($a, a_i, \phi, \theta, A$) then continue;
21         return $true$;
22     $\theta[a_i] \leftarrow \phi[a_i]$;
23     return $false$;
```

PIBT and LaCAM

PIBT is an iterative rule-based approach for solving MAPF problems (Okumura et al. 2022). The basic schema, which we adopt in this work, is sketched in Algorithm 1. In each iteration or timestep (lines 7-10) all agents plan a single step θ toward their goal locations by calling the PIBT function. Moves are selected according to the individual shortest distance to agents' goal locations, where the move toward a location with shorter distances to the goal location is preferred, with the next agent to plan being selected according to an initial priority order p. When two agents compete for the same next location, a simple 1-step reservation scheme is applied. It allows higher-priority agents to reserve their next moves while lower-priority agents are bumped to other less desirable locations. This strategy is applied recursively, which means that bumped agents are selected next, thus inheriting the priority of the higher priority agent.(line 20). Once every agent is planned the moves are executed. In other words, agents advance toward their goal locations and a new iteration begins.

PIBT guarantees that the highest priority agent will eventually reach its goal location, at which point it becomes the lowest priority agent (line 5). Thus PIBT never produces deadlock situations. However, in one-shot MAPF, the PIBT strategy may still produce livelocks, which can occur among pairs of agents where the goal position of one appears on the planned path of the other. This makes PIBT incomplete in general. In lifelong MAPF the situation is different. Here arriving agents are immediately assigned new tasks, which

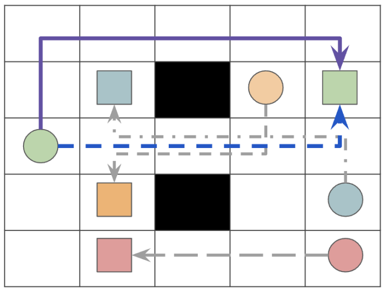

Figure 1: We show a small MAPF problem with 4 agents. Dashed (blue) lines indicate individually optimal paths from each s_i to each g_i. Solid (purple) lines indicate congestion-aware individually-optimal paths.

resolves any contention. Thus for lifelong MAPF the PIBT algorithm is indeed complete.

LaCAM*: is an anytime search strategy that combines a systematic joint-space search with PIBT to compute suboptimal MAPF plans more efficiently (Okumura 2023). Each node of LaCAM* is a configuration of the agents on the map. Successor nodes are generated by invoking PIBT, with the move selected for each agent being restricted by a set of associated constraints. LaCAM systematically explores the set of joint-space plans but uses a type of partial expansion known as Operator Decomposition (Standley 2010) to avoid the usual branching factor explosion. The addition of a systematic search allows LaCAM* to succeed more often than PIBT, and to compute higher-quality plans, while retaining its performance advantages.

Traffic Congestion Reasoning for MAPF

A simple approach to planning each agent in a MAPF problem is to simply generate the shortest path for each agent independently of all the rest. This is the policy employed by PIBT and it has two main advantages: (i) the paths are *time independent*, which makes them fast to compute; (ii) the path costs provide strong *lower-bounds* on the true time-dependent optimal cost, which makes these paths useful as guiding heuristics, for undertaking further state-space search, such as LaCAM*. The main drawback is that following these paths almost always leads agents into conflicts, as the vertices that appear on time-independent paths tend to have a greater *betweenness centrality* (Freeman 1977) than other vertices in the graph. In other words, a vertex appearing on one time-independent optimal path is more likely to also appear on a great many other optimal paths.

To tackle this issue we propose to compute time-independent optimal paths which are *congestion-aware*. In particular, after computing one time-independent shortest path, we update the edge costs of the graph, so that the edges which appear on that path become more expensive to traverse. A similar update strategy drives solution methods of Traffic Assignment Problems, where edge-cost increases reflect additional *traffic cost* from multiple vehicles using the same road link at the same time. We adapt these general ideas for MAPF by first defining a model for computing traffic costs and then applying these costs to compute new congestion-aware move policies and *guide paths* for PIBT. Figure 1 shows a representative example; we compare the set of individually optimal paths for a small group of agents and their corresponding congestion-aware counterparts.

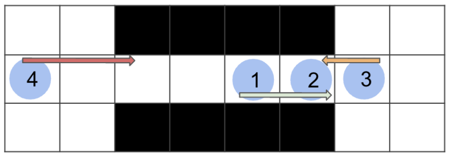

Figure 2: In this example high-priority agents enter a corridor just before a set of lower-priority agents exit, the number on each agent indicating its priority. Each time this occurs there is a large increase in the objective function value.

Traffic Costs

Define f_{v_1,v_2} as the *flow* from vertex v_1 to v_2 given a set of agents whose time-independent shortest paths each have the state v_1 followed by v_2. Thus f_{v_1,v_2} is the number of agents traverse the edge (v_1, v_2) from v_1 to v_2 direction in the current path assignment. Note that f_{v_2,v_1} indicates traversing through the same edge but from v_2 to v_1 direction.

Define the *vertex congestion* c_v of vertex $v \in V$ as $c_v = \frac{n\times(n-1)}{2}$ where $n = \sum_{v'\in V:(v',v)\in E} f_{v',v}$ is the total number of agents entering vertex v. c_v represents the least total delay that will occur assuming all agents enter the vertex at the same time, since in the best case each agent will have to wait for all the agents preceding it. Apportioning this congestion to each agent equally leads to a cost per agent of $p_v = \lceil \frac{c_v}{n} \rceil = \lceil \frac{n-1}{2} \rceil$.

Define the *contraflow congestion* c_e of undirected edge $e \equiv (v_1, v_2) \in E$ as $c_e = f_{v_1,v_2} \times f_{v_2,v_1}$. This formulation reflects our observation that PIBT can push agents of higher priority into a corridor which then forces lower-priority agents, already inside the corridor, to reverse direction. The result is an exponential increase in path cost for all agents attempting to cross. Figure 2 shows an example.

Traffic Assignment Problem uses congestion charges to directly modify affected edge weights and then replan shortest paths. This is also beneficial for MAPF but we found the *actual cost* of contraflow congestion is massively underestimated by our model above. To overcome this issue we generate two-part edge weights to more accurately model congestion. Each edge $e = (v_1, v_2)$ is given a two-part weighted cost $(c_e, 1 + p_{v_2})$, where 1 indicates the free-flow (i.e., zero congestion) cost of using edge e. We then search for (lexicographically) shortest paths for each agent using this two-part cost, i.e. first minimising contraflow costs, and then weighted edge costs. In addition, we explore other variants, such as $1 + c_e + p_{v_2}$ and $1 + p_{v_2}$, and comparable methods from existing literature (Han and Yu 2022) to compute edge weights in the experimental section.

Algorithm 2: We compute (approximate) user equilibrium paths π for a set of MAPF agents A on map (V, E). The set of start locations is denoted s, and goal locations g. We use FOCAL Search to compute paths, with parameter $w \geq 1$ indicating the admissibility criteria for suboptimal solutions.

```
FindPaths($A, V, E, s, g$)
    for $(v_1, v_2) \in E$ do
        $f[v_1, v_2] \leftarrow 0; f[v_2, v_1] \leftarrow 0$;
    for $a \in A$ do
        $\pi[a] \leftarrow$ SP($s_a, g_a, f, V, E, w$);
        for $i \in 2..|\pi[a]|$ do
            $f[\pi[a][i-1], \pi[a][i]] + +$;
    return $\pi$

PathRefinement($\pi$)
    while not meet termination condition do
        Choose subset $S \subset A$ ;
        $\pi \leftarrow$ Replan($\pi, S, A, f, V, E$) ;

Replan($\pi, S, A, f, V, E$)
    for $a \in S$ do
        for $i \in 2..|\pi[a]|$ do
            $f[\pi[a][i-1], \pi[a][i]] - -$ ;
    for $a \in S$ do
        $\pi[a] \leftarrow$ SP($s_a, g_a, f, V, E, w$);
        for $i \in 2..|\pi[a]|$ do
            $f[\pi[a][i-1], \pi[a][i]] + +$;
    return $\pi$
```

Path Planning and Improvement

Analogous to the process for solving the TAP (Chen, Jayakrishnan, and Tsai 2002), we start with *flow* of 0 for each edge on each traversing direction, and compute shortest paths iteratively. After computing one shortest path we immediately update affected edge costs. This means the next agent uses all available traffic information when computing its shortest path. The algorithm is formalised in Algorithm 2.

We begin by initialising flow counts to zero (line 3), and then plan agents one by one. We use the current flow counts $f[\cdot, \cdot]$ to compute the two-part edge weight $(c_e, 1 + p_v)$ for edge $e \equiv (u, v)$ during the shortest path computation SP. The subsequent path for agent a, denoted $\pi[a]$, is a sequence of vertices starting from $\pi[a][1] = s_a$ where $|\pi[a]|$ is its length (number of actions) and $\pi[a][|\pi[a]|] = g_a$. We then update flow counts (line 7) and plan the next agent.

Once each agent has a path we call the procedure PathRefinement (lines 10-13). This iterated method selects (at random) a subset of agents $|S|$ whose paths will be replanned, so as to improve their goal arrival time. The function Replan (lines 15-23) removes the paths of selected agents from the stored flows and computes a new path for each agent $a \in S$ followed by updating flows, continuing until a termination condition is met (= max iterations).

We end with a *time independent* path for each agent, that tries to take into account likely delays caused by congestion.

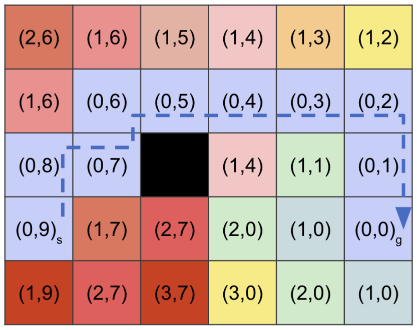

Figure 3: A small figure with a guide path (the blue dashed line) and marked heuristic values.

Next, we discuss how these paths can be used to improve PIBT performance for one-shot and lifelong MAPF.

Guide Paths and Guide Heuristics

In PIBT the free-flow distance (i.e., not considering traffic costs) from each vertex to the goal is used to decide the preferred movement direction for each agent. The agent will always choose this move unless prevented by a higher priority agent, in which case it selects the next-best move. Since this policy does not consider the paths of any other agent, this approach invariably leads to high congestion.

In this work, we propose to modify PIBT so that each agent tries to follow a congestion-aware *guide path*. For each agent a_i we thus compute a *guide heuristic* $h_i(v)$. Given a vertex $v \in V$ we compute a two-part value $h_i(v) = (dp, dg)$, where dp is the shortest free-flow distance, from v to the guide path. The value dg meanwhile is the shortest remaining distance to the goal g_i, as reached subsequently by strictly following the guide path. To efficiently compute guide heuristics we use a lazy backward Breadth First Search (BFS). Every location on the input guide path is pushed to the BFS queue as a root-level node. We then expand nodes lazily, layer by layer, up to the current requested location v. We cache the h_i-value of each expanded node along the way. When we receive a request for a non-cached h_i-value the search can be resumed. The heuristic is updated (re-computed) only when the agent receives a new goal location. Notice that the computation of the guide heuristic does not consider traffic costs. This is because they are already reflected by the guide path, which is given as input. Meanwhile, the computed h_i values are optimal because every action has cost=1. Applying the guide heuristic in PIBT is trivial: we simply sort the adjacent vertices C by $h_i(v)$ rather than $dist(v, g_i)$.

Example 1. *Figure 3 shows guided planning with PIBT. Agent a_i is currently at vertex v. Its move preference is toward an adjacent vertex v' where $h_i(v')$ is lexicographically least. When the agent is not on the guide path its preferred move is the one that minimises the number of steps to reach a vertex on the guide path. In the case of ties, the agent prefers to move toward a vertex on the guide path with the minimum remaining distance to the goal.*

Algorithm 3: Lifelong Procedure to determine the next move θ for each agent $a \in A$ from current position ϕ, with current goal vitices g, a guide heuristic table h, and a set of guide path π.

```
1  GuidedPlanStep(A,$\pi$, t, g, $\phi$)
2      //Initialising;
3      if $\exists a \in A, \pi[a] = \emptyset$ then
4          A'←A;
5          if Relax then  A'← SelectNotInitialized(A);
6          $\pi \leftarrow$ FindPaths(A',V,E,$\phi$,g);
7      //Updating;
8      for $\forall a \in A, g[a]$ changed, $\pi[a] \neq \emptyset$ do
9          $\pi \leftarrow$ Replan($\pi$, a, A,f,V,E);
10     //Refining;
11     if $\forall a \in A, \pi[a] \neq \emptyset \wedge$ Refine then
        PathRefinement($\pi$);
12     for $\forall a \in A$ where $\pi[a]$ has changed do  $h_a \leftarrow$
        Get($\pi[a]$) ;
13     //PIBT with Guide Heuristic;
14     $\theta \leftarrow$ PlanStep(A);
15     return $\theta$;
```

In one recent and related work (Han and Yu 2022) authors also compute a per-vertex traffic cost metric, for use as a distance/heuristic tiebreaker in prioritised planning with A^* search. In other words, when planning the path of each agent the search tie-breaks on vertices with lower traffic cost, provided those vertices also minimise the shortest distance to the goal. The main drawback of this approach is that the objective function does not explicitly consider traffic costs, which diminishes its effectiveness.

Lifelong Procedure

In lifelong MAPF the arrival of an agent at its goal makes redundant its guiding heuristic. The arrival of other agents meanwhile causes guidance data to become outdated, as traffic costs change. To handle these situations we propose to compute guidance data continuously and entirely online. Algorithm 3 shows the corresponding procedure.

At the start of a lifelong problem, we need to compute guide paths and guide heuristics for all the agents. Undertaken sequentially, these operations can require up to dozens of seconds, during which time agents are typically assumed not to move. To smooth the response time over the operation period we propose a lazy initialisation scheme (lines 2-6), where guidance is only computed for a certain number of agents per timestep. Agents without guide paths simply follow their individual shortest distance (i.e., unmodified PIBT) until they can be processed at a later timestep.

Whenever any agent is assigned a new task guidance data for all agents becomes outdated to various degrees. For this reason, we call the PathRefine procedure (Algorithm 2) every timestep. If guidance paths for any agent have changed as a result, we compute new heuristics for those agents (lines 10-12). The refining of the guide paths can be undertaken even if no new tasks are assigned to any agent. One advantage of this strategy is that guide paths can be recomputed based on agents' current locations, which allows us to remove flow costs due to past trajectories. After refinement we plan agents using PIBT (line 14, with the distance function replaced by guide heuristics if available.

Experimental Results

In our main experiment, we compare guided PIBT against the original baseline for Lifelong MAPF. We also integrate guidance into (one-shot) LaCAM* and compare against the original baseline for one-shot MAPF. Implementations[1] are written in C++ and evaluated on a Nectar Cloud VM instance with 32 AMD EPYC-Rome CPUs and 64 GB RAM.

We run large-scale experiments with up to 12 thousand agents across 4 distinct map types. For each map and each number of agents, we evaluate 24 randomly sampled instances, resulting in 600 problem instances in total. For Lifelong MAPF, the maximum simulation time is based on the size of the map: we compute a maximum number of timesteps as $(with + height) \times 5$ with the intention that each agent has the opportunity to complete approximately 5 tasks. Our maps are:

- *Warehouse*: a 500×140 synthetic fulfillment center map with 38589 traversable cells. There are 144 instances, with between 2000 to 12000 agents (which occupy 31% traversable cells). Simulation time: 3200 timesteps.
- *Sortation*: a 33×57 synthetic sortation centre map with 1564 traversable cells. There are 168 instances, with between 200 to 1400 agents (which occupy 92% traversable cells). Simulation time: 450 timesteps.
- *Game*: ost003d, a 194×194 map with 13214 traversable cells, from video games. There are 144 instances with between 2000 to 12000 agents (which occupy 90% traversable cells). Simulation time: 1940 timesteps.
- *Room*: room-64-64-8, a 64×64 synthetic map with 3232 traversable cells. There are 144 instances, with between 500 to 3000 agents (which occupy 93% traversable cells). Simulation time: 640 timesteps.

In one-shot MAPF experiments, planners have 60 seconds timelimit. In lifelong MAPF experiments, planners have 10 seconds to return actions for all agents at every timestep. Failing to return in any timestep leads to timeout failure.

Results for Lifelong MAPF

The basic implementation of our algorithm is `GP-R100`, where GP indicates guide path heuristic guidance and `R`x indicates the initialization of guide paths x is limited to x paths per timestep. Algorithm names with `F`w indicate the algorithm uses *Focal Search* to bound the path length up to $w \cdot C^*$. Names with `Re`i means i iterations of online refinement are applied at each timestep. For each online refinement iteration, 10 agents are selected by two selection methods, one randomly selects agents from all agents, and the other one selects the agent with the highest traffic cost on the guide path and other agents with guide paths intersecting with it. We use adaptive LNS (Li et al. 2021a) to record

[1]https://github.com/nobodyczcz/Guided-PIBT

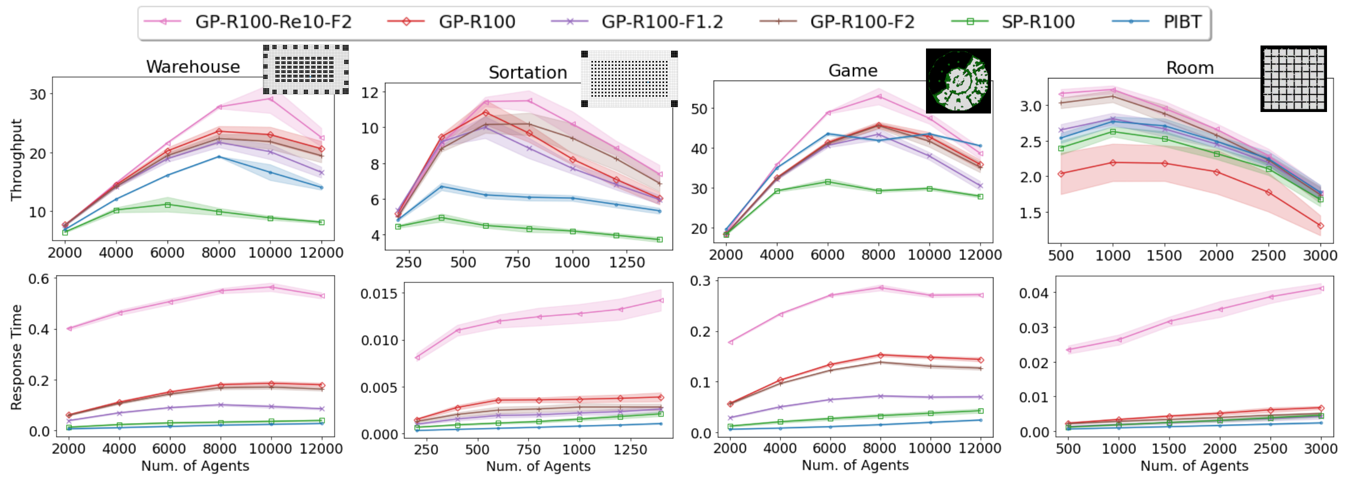

Figure 4: Lifelong MAPF. Average throughput (top) and response time in second (bottom). Shaded regions show standard deviation.

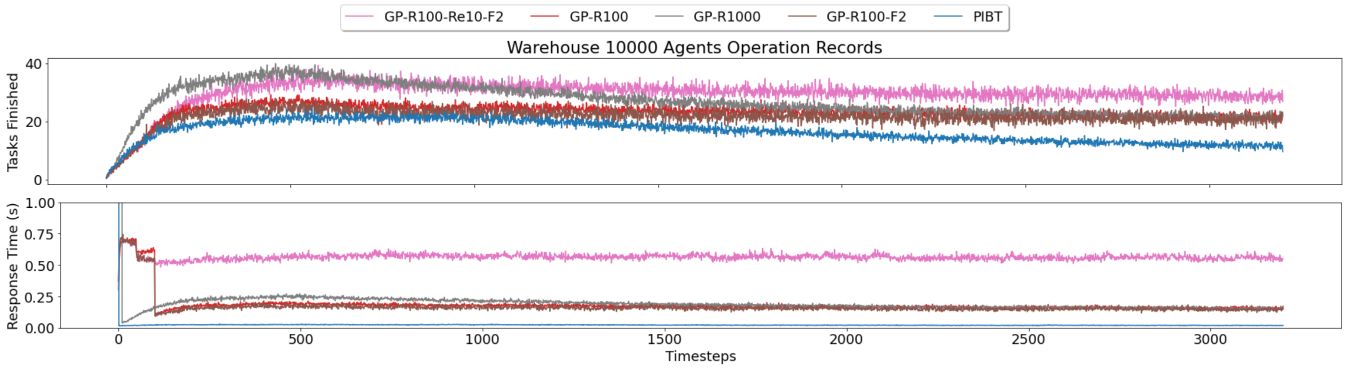

Figure 5: Lifelong MAPF. Average tasks finished (top) and average response time (in the range 0-1 second; bottom). 24 instances, with 10,000 agents each. PIBT and GP-R1000 incur setup costs of 6.26 and 6.98 seconds to compute heuristic data.

the two methods' relative success in improving the solutions and choosing the most promising method to select agents for replan. `SP` is a reference algorithm using individual shortest paths to compute guide heuristics, ignoring traffic cost.

PIBT with Guide Path: Figure 4 measures the average throughput and average response time, which is the average time the planner returns actions for all agents at each timestep. The base implementation `GP-R100` shows a great advantage over PIBT on throughput with Warehouse and Sortation maps on all numbers of agents. Performance is lower for Room as this map has many single-length corridors. It is challenging to form efficient two-way traffic here as fewer alternative paths with similar costs exist. With *Focal Search* enabled, `GP-R100-F2` exploits the additional slack to improve throughput on Room, whereas `GP-R100-F1.2` does not help as the restriction is too tight.

By enabling the online refinement, `GP-R100-Re10-F2` refines 10 iterations on 100 paths every timestep. This bumps the throughput on every map on every number of agents. We noticed that in lifelong MAPF there is a peak agent density, beyond which adding more agents decreases throughput due to increasingly severe congestion. Compared with PIBT, our methods shifted this peak to the right on each map, with `GP-R100-Re10-F2` improving Warehouse and Game by 2000 agents and Sortation by 200 agents.

Steady Operation: Figure 5 shows timestep breakdown records for 10,000 agents on Warehouse over 24 instances. Initialisation for PIBT requires 6 seconds at the start of the operation while `GP-R1000` requires 7 seconds each for the first 10 timesteps, with 1000 guide paths being initialised per timestep. Clearly, guide paths are more expensive to compute than PIBT unit-cost distance tables (7ms per agent versus 0.6 ms per agent on the Warehouse map). This highlights the importance of lazy initialisation.

`GP-R100` and `GP-R1100-Re10-F2` have steady response times, within 1 second, for the entire planning episode. Interestingly, PIBT and `GP-R1000` start with an advantage for initialisation their tasks finished and soon hit a peak and then drop for the remainder of the episode, as more agents are pushed to congested areas. `GP-R100-Re10-F2` remains steady throughout.

Comparisons vs. RHCR

Metric	ALG	200	400	600	800	1000
TP	RHCR-ECBS	5.7±0.1	9.1±0.1	9.1±0.2	5.2±0.3	3.5±0.2
	RHCR-PBS	**6.0±0.1**	**11.1±0.1**	6.5±0.6	5.2±0.5	3.5±0.3
	GP-R100-Re10-F2	5.0±0.1	9.1±0.1	**11.4±0.3**	**11.5±0.6**	**10.2±0.7**
R-Time per Timestep	RHCR-ECBS	**0.005±0.0**	0.038±0.002	0.78±0.121	1.987±0.009	2.0±0.0
	RHCR-PBS	0.023±0.0	0.29±0.016	1.997±0.015	2.01±0.006	2.006±0.003
	GP-R100-Re10-F2	0.008±0.0	**0.011±0.001**	**0.012±0.001**	**0.012±0.001**	**0.013±0.001**
R-Time per Planner Run	RHCR-ECBS	0.026±0.0	0.191±0.012	3.905±0.619	9.936±0.048	10.001±0.0
	RHCR-PBS	0.114±0.002	1.449±0.078	9.985±0.074	10.051±0.028	10.03±0.013
	GP-R100-Re10-F2	**0.008±0.0**	**0.011±0.001**	**0.012±0.001**	**0.012±0.001**	**0.013±0.001**

Table 1: RHCR vs. Guided PIBT on Sortation centre map, with number of agents varies from 200 to 1000. It compares the Throughput (TP), Response Time (R-Time) per timestep, and Response Time (R-Time) per planner run.

The *Rolling-Horizon Collision Resolution* (RHCR) (Li et al. 2021d) solves lifelong MAPF by decomposing the problem into a sequence of Windowed MAPF instances, where a Windowed MAPF solver resolves collisions among the paths of the agents only within a bounded time horizon and ignores collisions beyond it. In this experiment, we use implementation source codes from the original authors.[2]

Although RHCR is a leading planner for lifelong MAPF, it has difficulty scaling to problems much larger than 1000 agents. Most of the benchmark instances in this work are larger than this size. We thus evaluated RHCR only on our small sortation centre benchmark set, which covers the range from 200 to 1000 agents. Table 1 show the throughput, response time per timestep and response time per planner run comparisons. Note that RHCR-PBS uses PBS (Ma et al. 2019) as MAPF planner, and RHCR-ECBS uses ECBS (Barer et al. 2014) with a suboptimality weight of 1.5. The planning window is set to 10 and the execution window is 5, which indicates the simulator calls the MAPF planner every 5 timesteps with the bounded time horizon set to 10.

RHCR does not call the MAPF planner every timestep. Thus we compute results for response time per timestep as total planning time divided by timesteps. Similarly, response time per planner run is the total planning time divided by the amount of MAPF planner runs. The time limit for the planner is set to 10 seconds. The results show that our Guided PIBT easily scales to a larger team size with higher throughput, and uses dramatically less computing resources.

Variants for Lifelong MAPF

In this section, we explore alternative approaches to compute the guide path using different congestion cost formulations:

- $\mathtt{GP}_v\mathtt{-R100}$ and $\mathtt{GP}_v\mathtt{-R100-F2}$ ignores *contraflow congestion* and compute guide paths on *vertex congestion* only. This variant examines if *vertex congestion* itself is enough for avoiding congestion.
- $\mathtt{GP}_{svc}$ using the sum of congestion costs $1 + c_e + p_v$ as objective for guide path planning. It examines if this formulation better balances congestion avoidance detour and path length.
- $\mathtt{GP}_{sui}$. Existing study (Han and Yu 2022) suggests adding SU-I cost-to-come congestion cost on the action cost, where the sum of added congestion cost along a path is guaranteed smaller than 1, to compute optimal length paths that tie-breaks towards less congestions. We examine this method for guiding PIBT.
- $\mathtt{GP}_{sui}\mathtt{-F2}$ the original SU-I cost-to-come and cost-to-go heuristic only acts as a tie-breaker, and is limited to the optimal path. Here we search for the SU-I cost-to-come minimised path within the bound of $C * \times 2$.

Table 2 compares `GP-R100` and `GP-R100-F2` with the above variants. Compared with $\mathtt{GP}_v\mathtt{-R100}$ and $\mathtt{GP}_v\mathtt{-R100-F2}$, the two-part approaches (`GP-R100`) have higher throughput on most maps, showing the necessity of considering *contraflow* costs.

The results of $\mathtt{GP}_{svc}$ shows the sum-of-congestion-costs model balances path length and traffic congestion avoidance better, as it gives outstanding performance on most maps, except for the warehouse map, indicating compared with the two-part approach it underestimated the congestion in a warehouse scenario.

$\mathtt{GP}_{sui}$ shows following an optimal length path that tie-breaking towards less congestion is not enough. By allowing suboptimal paths with costs up to $C * \times 2$, $\mathtt{GP}_{sui}\mathtt{-F2}$ computes more effective guide paths.

Additionally, we explore alternative approaches, **Traffic Cost Heuristics**, to guide PIBT agents after guide paths are computed and traffic flow are recorded on edges. $\mathtt{TH}_v$ and $\mathtt{TH}_v\mathtt{-R100}$ use the recorded flow on each edge and compute vertex congestion cost minimised heuristics, where the cost on each vertex is $1 + p_v$, using Reverse Resumable A* (Silver 2005). $\mathtt{TH}_v$ recompute all heuristics for all agents at each timestep if any agent has new tasks, as new tasks led to the change of cost-minimised single agent path and thus the cost to reach each vertex changed. $\mathtt{TH}_v\mathtt{-R100}$ relaxed the recomputation for up to 100 heuristics per timestep, except for those with new tasks.

Table 2 shows $\mathtt{TH}_v$ suffers from timeout failure on large maps, $\mathtt{TH}_v\mathtt{-R100}$ suffers less but still fails on Warehouse and with lower throughput on Game. On small maps, $\mathtt{TH}_v$ has a slightly higher throughput than $\mathtt{GP}_v\mathtt{-R100}$, but is 5 to 10 times slower to return actions.

[2]https://github.com/Jiaoyang-Li/RHCR

ALG	Warehouse (8000)		Sortation (600)		Game (8000)		Room (1000)	
	TP	R-Time (s)	TP	R-Time (s)	TP	R-Time (s)	TP	R-Time (s)
GP-R100	**23.6±5.8**	0.18±0.083	10.9±3.5	0.004±0.002	45.7±8.6	0.153±0.041	2.2±1.5	0.003±0.006
GP-R100-F2	22.3±5.6	0.168±0.08	10.2±3.4	0.002±0.002	45.5±8.6	0.138±0.036	3.1±1.8	0.003±0.004
GP$_v$-R100	14.9±4.2	0.084±0.059	9.1±3.2	0.002±0.001	41.0±7.7	0.092±0.029	2.9±1.7	0.002±0.004
GP$_v$-R100-F2	14.8±4.2	0.114±0.076	9.0±3.2	0.002±0.002	41.1±7.7	0.086±0.026	2.9±1.7	0.002±0.004
GP$_{svc}$-R100	22.9±5.7	0.17±0.083	**11.8±3.8**	0.003±0.002	**46.5±8.7**	0.138±0.037	3.1±1.8	0.003±0.005
GP$_{sui}$-R100	20.0±5.2	0.076±0.029	8.8±3.1	0.002±0.001	44.3±8.4	0.06±0.015	2.7±1.6	0.002±0.002
GP$_{sui}$-R100-F2	21.9±5.5	0.149±0.069	11.2±3.6	0.003±0.002	45.6±8.5	0.124±0.032	3.1±1.8	0.003±0.005
TH$_v$	timeout	timeout	10.9±3.5	0.05±0.008	timeout	timeout	**3.5±1.9**	0.187±0.026
TH$_v$-R100	timeout	timeout	10.7±3.5	0.011±0.004	31.1±11.6	0.27±0.221	3.4±1.9	0.023±0.015
PIBT	19.3±5.0	**0.021±0.092**	6.2±2.6	**0.001±0.0**	41.9±7.7	**0.015±0.065**	2.8±1.7	**0.001±0.002**

Table 2: Throughput (TP) and response time (R-Time) of vertex traffic cost heuristics (TH$_v$), vertex guide path heuristics (GPV), $1 + c_e + p_{v2}$ guide path heuristics (GP$_{svc}$), SU-I cost-to-come (SUI) guide path heuristics (GP$_{sui}$), bounded SUI minimised guide path heuristics (GP$_{sui}$-F2), and two-part guide path heuristics (GP) per map, with the peak agents derived from figure 4. Warehouse (8000) indicates 8000 agents on the Warehouse map, the same for other maps.

Map	Agents	RC		Solved		
		F1.2	F2	F1.2	F2	LC*
Warehouse	2000	0.982	1.467	24	24	24
	4000	0.971	1.166	24	24	24
	6000	0.942	1.106	24	24	24
	8000	0.852	1.009	24	24	24
	10000	0.770	0.954	24	24	24
	12000	-	-	0	0	24
Sortation	200	0.965	0.992	22	24	24
	400	0.860	0.888	22	24	24
	600	0.754	0.751	22	24	24
	800	0.728	0.695	21	24	24
	1000	0.759	0.735	22	24	24
	1200	0.815	0.817	22	24	24
	1400	0.903	0.906	22	24	24
Game	2000	1.058	1.084	19	24	24
	4000	1.070	1.121	19	24	24
	6000	1.072	1.099	19	24	24
	8000	1.029	1.053	6	3	22
	10000	-	-	0	0	14
Room	500	0.904	0.888	22	24	24
	1000	0.890	0.804	22	24	24
	1500	0.921	0.801	22	24	24
	2000	0.948	0.828	20	15	19
	2500	-	-	0	0	1

Table 3: The Relative SIC (RC) of Guided LaCAM* (F1.2 and F2) to LaCAM*(LC*) on co-sovable instances. F1.2 and F2 are with guide path computed by focal search on sub-optimality bound 1.2 and 2. The solved column shows the problems are solved within the 60s runtime limit. Guided LaCAM* uses 30s to compute and optimise guide paths and 30s to find a feasible solution.

Results for One-Shot MAPF

For one-shot MAPF, 60 seconds runtime limit is given to compute and improve solution qualities. LaCAM* (LC*) is modified to optimise SIC. Guided LaCAM* requires a 30 seconds setup, for computing and refining guide paths, and uses the rest of the time to search for a MAPF solution. If 30 seconds is not enough to compute the guide paths for all agents, we take what we have and let agents without guide paths follow their individual shortest distances. We evaluated two versions of Guided LaCAM*, one uses *Focal Search* with $w = 1.2$ to compute guide paths, labelled as `F1.2`, and the other one uses $w = 2$, labelled as `F2`.

Table 3 shows the average of the relative cost (RC), which is the SIC_{guided}/SIC_{lacam}. Our method finds solutions with higher quality in the Warehouse, Sortation and Room. On Warehouse, with the environment getting dense, Guided LaCAM* finds better solutions. While Sortation shows this trend ends with 90% traversable cells occupied. The drawback is that Guided LaCAM* only has 30 seconds left to find feasible solutions, which sacrifices the scalability if having limited runtime. But it concludes that for finding higher-quality solutions, it's reasonable to spend time on the computing guide path rather than relying on the anytime improvement from LaCAM*.

Conclusions

In this work, we investigate how congestion-avoiding guide paths can improve performance for one-shot and lifelong MAPF. Drawing inspiration from the literature on Traffic Assignment Problems, we develop guided variants of PIBT and LaCAM*, two recent and highly scalable MAPF planners, which nevertheless rely on "myopic" planning strategies. By considering congestion costs we achieve substantial improvements in throughput for lifelong MAPF vs PIBT, successfully planning operations for 10,000+ agents and always returning actions for all agents in under 1 second. For one-shot MAPF we substantially improve solution quality for LaCAM*, at the cost of small reductions in scalability. Future work will focus on the design of more informed and accurate objective functions for computing guide paths, as the current method relies on *Focal Search* to balance between detour distance and congestion avoidance.

Acknowledgments

The research at Monash University was partially supported by the Australian Research Council under grants DP190100013 and DP200100025 as well as a gift from Amazon. The research at Carnegie Mellon University was supported by the National Science Foundation (NSF) under grant number 2328671.

We extend our special thanks to Federico Pecora from Amazon Robotics for his valuable insights and discussions that greatly contributed to this work.

References

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search*, 19–27.

Chen, A.; Jayakrishnan, R.; and Tsai, W. K. 2002. Faster Frank-Wolfe traffic assignment with new flow update scheme. *Journal of Transportation Engineering*, 128(1): 31–39.

Freeman, L. C. 1977. A set of measures of centrality based on betweenness. *Sociometry*, 35–41.

Gange, G.; Harabor, D.; and Stuckey, P. J. 2019. Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 155–162.

Han, S. D.; and Yu, J. 2022. Optimizing space utilization for more effective multi-robot path planning. In *2022 International Conference on Robotics and Automation (ICRA)*, 10709–10715. IEEE.

Kou, N. M.; Peng, C.; Ma, H.; Kumar, T. S.; and Koenig, S. 2020. Idle time optimization for target assignment and path finding in sortation centers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9925–9932.

Lam, E.; Le Bodic, P.; Harabor, D.; and Stuckey, P. J. 2022. Branch-and-cut-and-price for multi-agent path finding. *Computers and Operations Research*, 144: 105809.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021a. Anytime Multi-Agent Path Finding via Large Neighborhood Search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 4127–4135.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2022. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 10256–10265.

Li, J.; Chen, Z.; Zheng, Y.; Chan, S.-H.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2021b. Scalable rail planning and replanning: Winning the 2020 flatland challenge. In *Proceedings of the international conference on automated planning and scheduling*, volume 31, 477–485.

Li, J.; Felner, A.; Boyarski, E.; Ma, H.; and Koenig, S. 2019. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 442–449.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; Gange, G.; and Koenig, S. 2021c. Pairwise Symmetry Reasoning for Multi-Agent Path Finding Search. *Artificial Intelligence*, 301: 103574.

Li, J.; Ruml, W.; and Koenig, S. 2021. Eecbs: A bounded-suboptimal search for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 12353–12362.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. S.; and Koenig, S. 2021d. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 11272–11281.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, 7643–7650.

Ma, H.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 837–845.

Okumura, K. 2023. Improving LaCAM for Scalable Eventually Optimal Multi-Agent Pathfinding. *arXiv preprint arXiv:2305.03632*.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2022. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence*, 310: 103752.

Patriksson, M. 2015. *The Traffic Assignment Problem: Models and Methods*. ISBN 978-0486787909.

Pearl, J.; and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE transactions on pattern analysis and machine intelligence*, (4): 392–399.

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

Shen, B.; Chen, Z.; Cheema, M. A.; Harabor, D. D.; and Stuckey, P. J. 2023. Tracking Progress in Multi-Agent Path Finding. *arXiv preprint arXiv:2305.08446*.

Silver, D. 2005. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, 117–122.

Standley, T. 2010. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 173–178.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. S.; et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the Twelfth Annual Symposium on Combinatorial Search*, 151–158.

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